Spring系列

## Spring系列第1篇：spring是什么？我们为什么要学习它？

### 环境

1. jdk1.8
2. idea
3. maven-3.6.1
4. spring-5.2.3.RELEASE

整个spring系列使用官方最新版本5.2.3.RELEASE，会有大量案例，都是通过maven来管理的，所以maven是必备技能，对这块不熟悉的可以去看一下：[Maven高手系列](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933753&idx=1&sn=0b41083939980be87a61c4f573792459&chksm=88621d47bf1594516092b662c545abfac299d296e232bf25e9f50be97e002e2698ea78218828&token=799820369&lang=zh_CN" \l "rd)

### 为什么需要学习spring？

1. 最大程度的简化了开发

* spring是一个非常优秀的java框架，其目标是为了简化java企业级开发，spring出来已经十几年了，这期间也一直围绕着这个目标在进行，像后面需要学习的springmvc、springboot、springcloud，这些技术也都是围绕着简化开发的目标在努力，到目前为止已经做的足够好了，可以说spring除了不能帮助我们实现业务逻辑代码之外，其他的事情spring都尽量去帮我们简化了，使用spring可以帮助我们节约大量开发时间。
* 不使用spring的情况下，开发一个项目可能需要2个月，用了spring可能1个月都不需要，你说这样的技术你想学么？

1. 大量公司使用

* 目前99%的公司使用了spring，可以去各大招聘网站看一下，spring算是必备技能，所以一定要掌握。

1. 顶级的源码设计

* spring框架源码设计非常优秀，在java开源项目中可以说是顶级的，个人到目前为止还没有发现比spring代码更优秀的开源项目，所以想提升代码能力的，强烈建议多看看spring的源码；关于提升代码能力的，还可以去看一下jdk的源码，也是非常棒的，里面有很多大师的杰作。

### 什么是spring？

spring是一个简化java企业级开发的一个框架，内部包含了很多技术，比如：控制反转&依赖注入、面向切面编程、spring事务管理、通过spring集成其他框架、springmvc、springboot、springcloud等等，这些都是围绕简化开发展开的技术，后面会对每个知识点详细介绍。

### 感受一下spring的强大

先让大家感受一下spring的强大，来激发大家学习的兴趣。

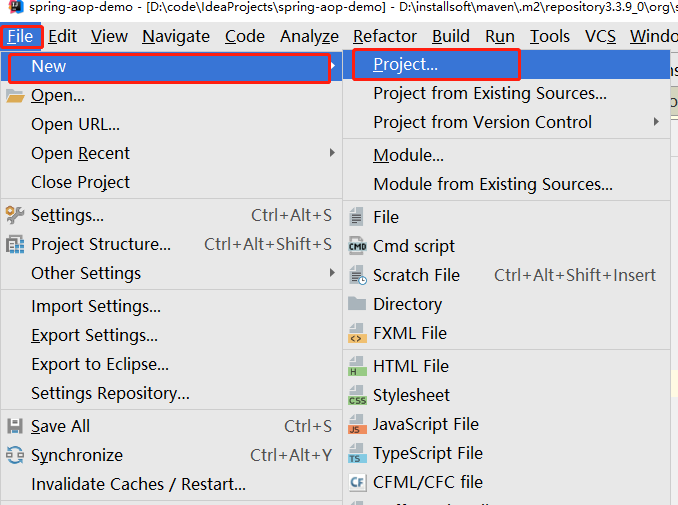
我们需要为移动端提供api接口，接口数据以json格式输出，使用java技术，实现过程大家应该比较熟悉，如下：

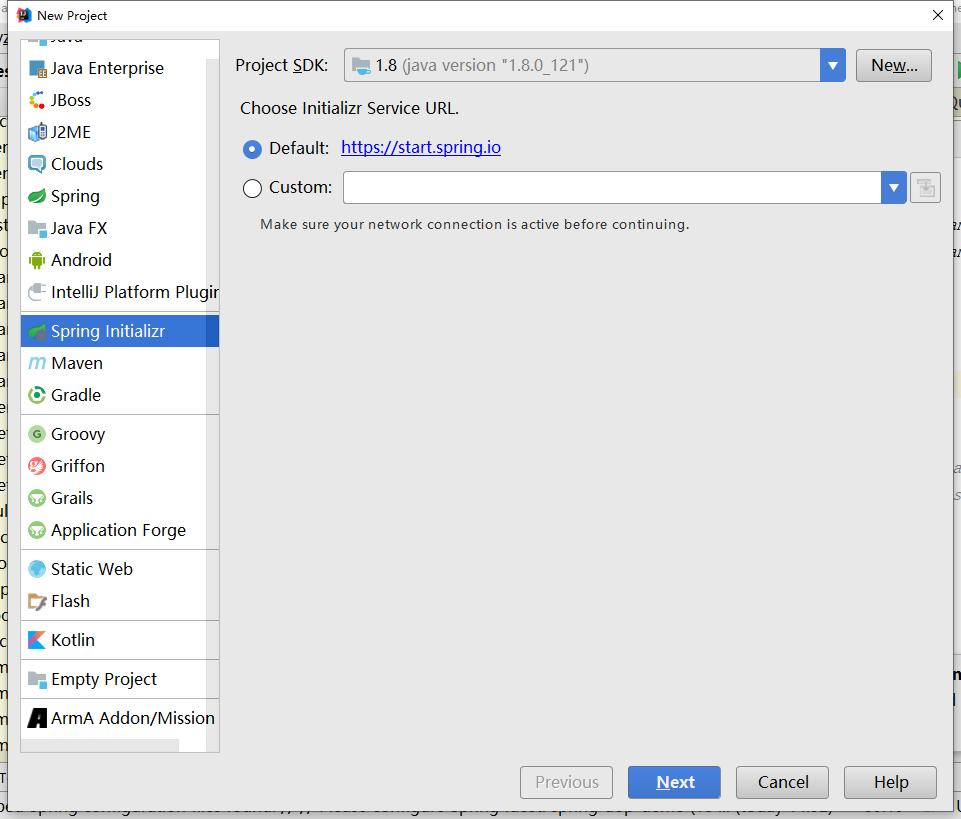
1. 使用idea创建一个web项目
2. 创建servlet，servlet中写业务代码，组装需要数据，将数据以json格式输出
3. 在web.xml中配置servlet信息
4. 将项目部署到tomcat中

原生的web项目开发过程都是上面这样，和业务相关的其实只有第2步，而其他几个步骤是每个web项目都需要的，像比较大的项目，会有很多web模块，每个模块都需要重复这些1/3/4操作步骤，还是相当繁琐的，也会耗费大量时间。

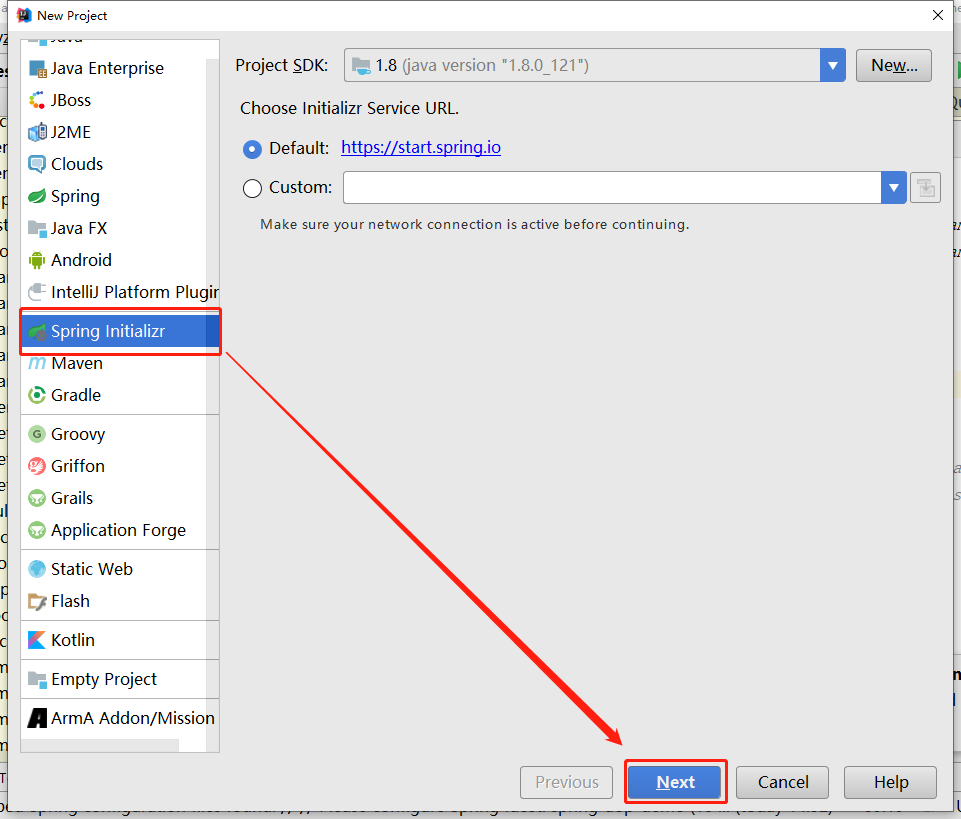
如果使用springboot来实现，过程会简化很多，来感受一下：

打开idea，点击File->New->Project，如下：

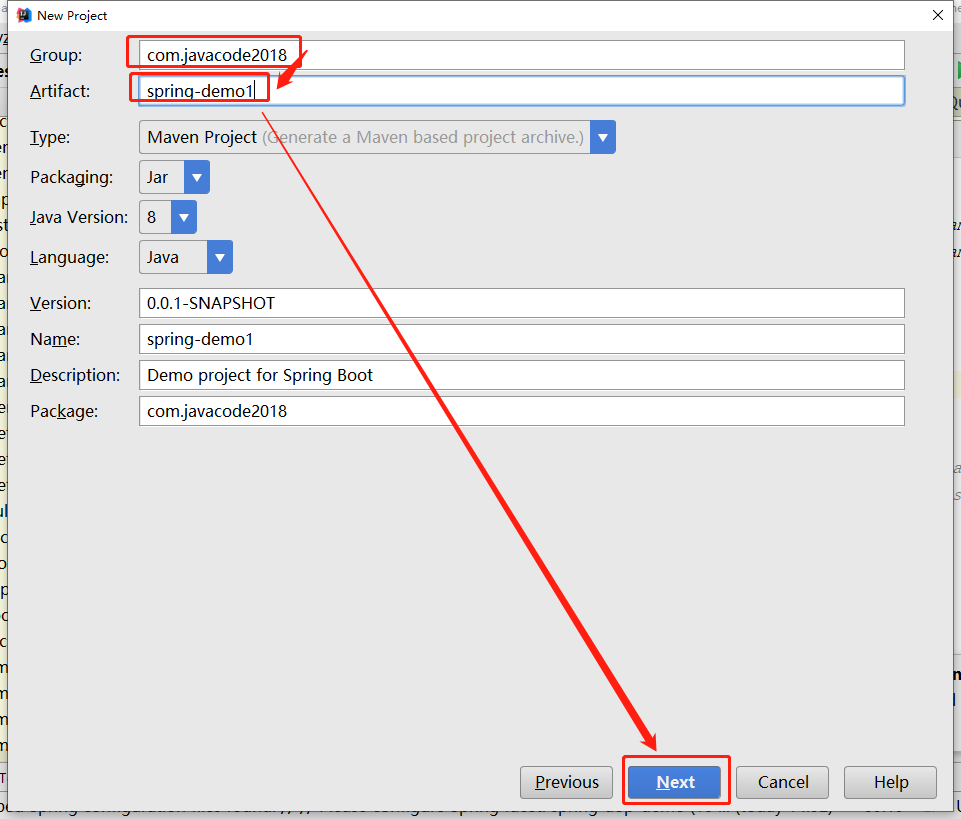




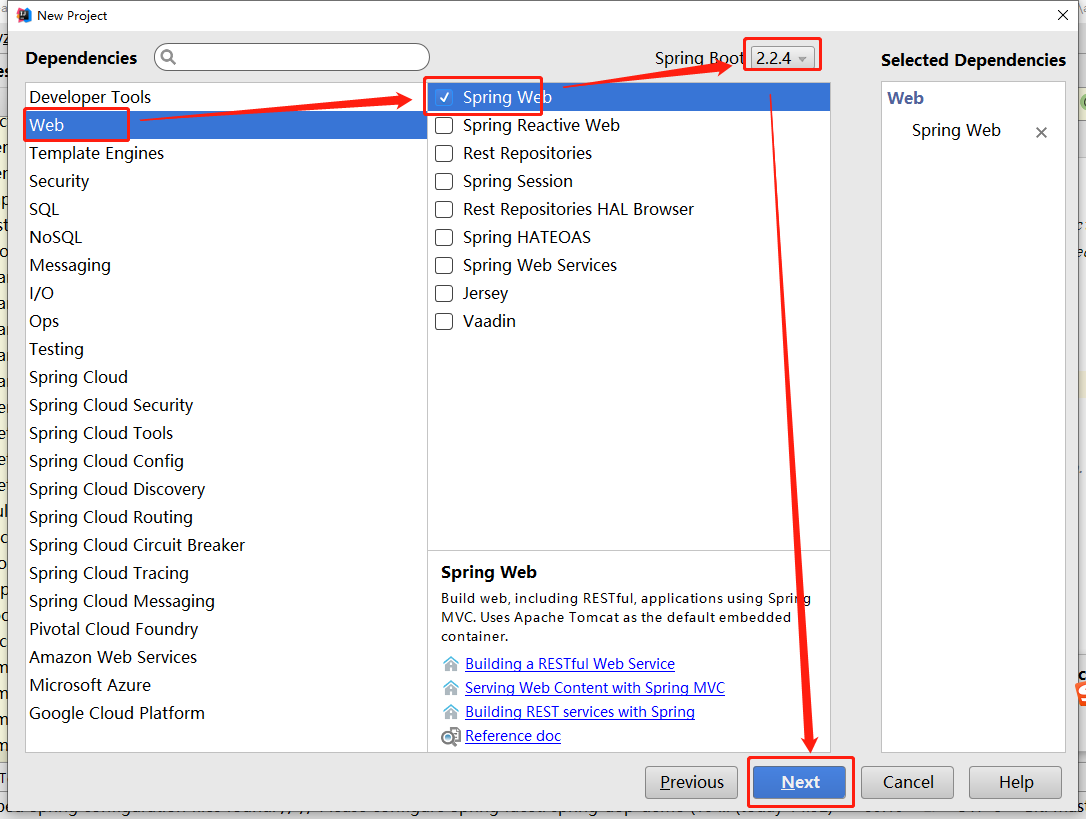
点击Spring Initializr->Next，如下图：



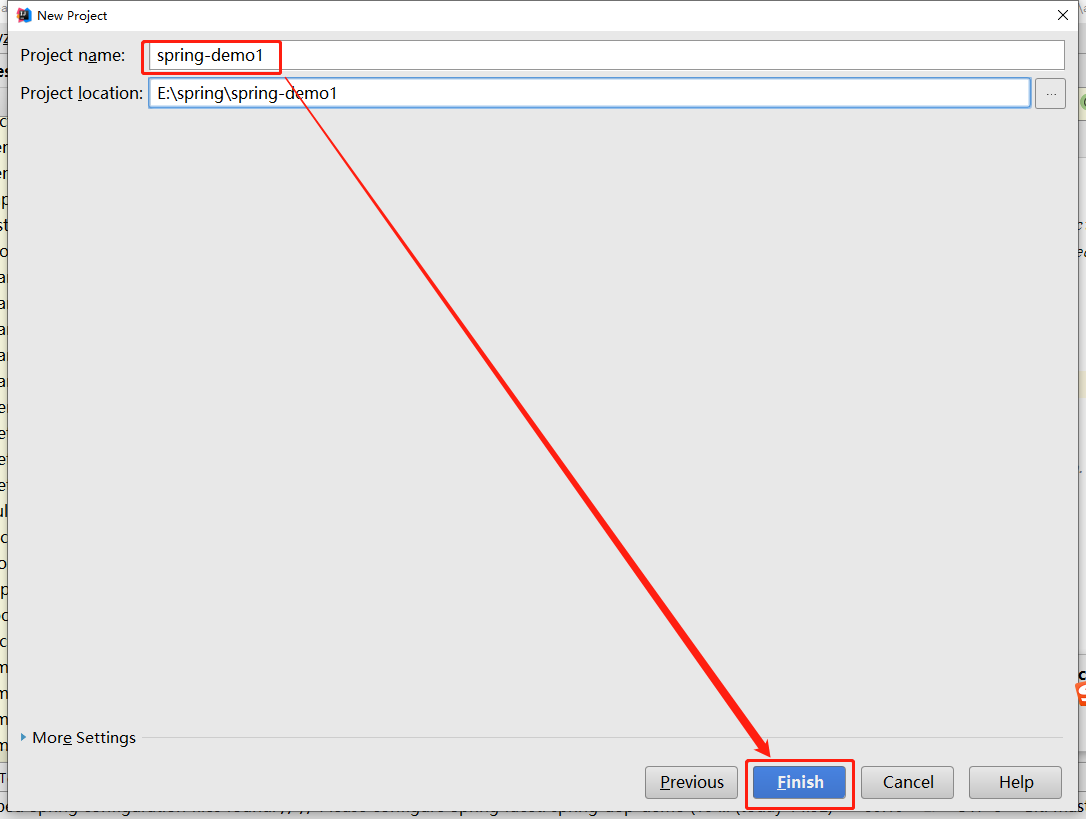
输入项目坐标信息，如下图：



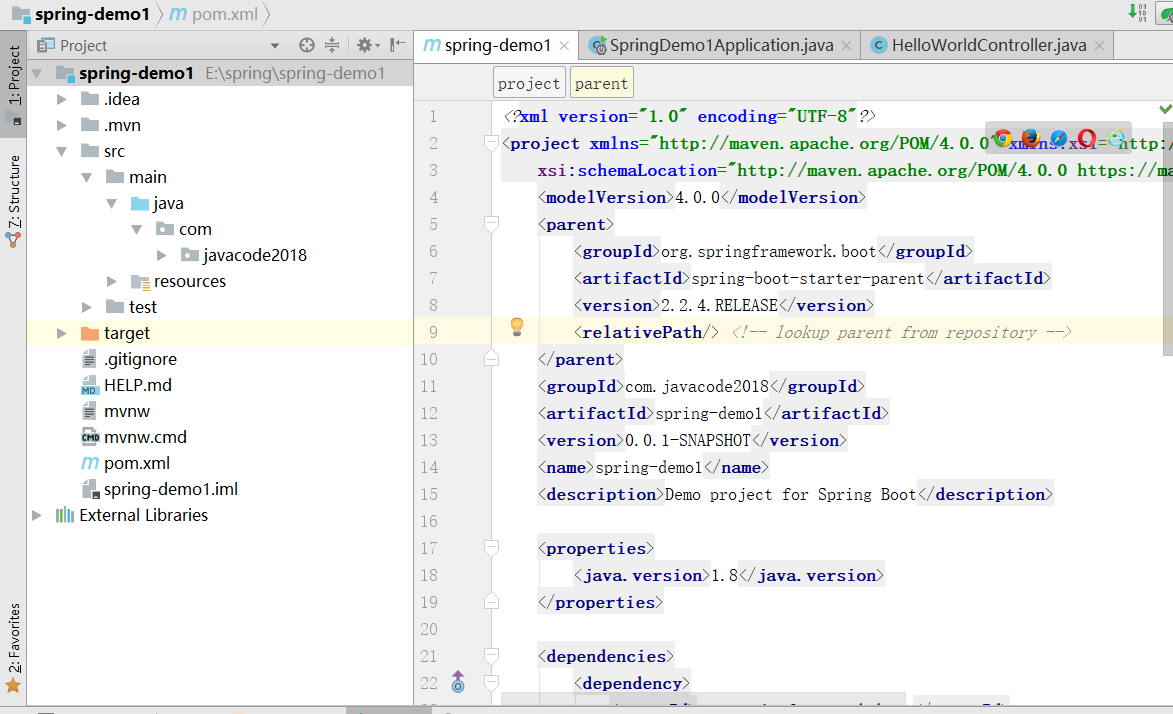
选择Web->Spring Web->Spring Boot(2.24)->Next，如下图：



点击Finish，完成项目的创建，如下图：



项目结构如下图：



目前为止我们还未写一行代码，下面我们创建一个类HelloWorldController，代码如下：

package com.javacode2018;  
  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@RestController  
@RequestMapping("/helloWorld")  
public class HelloWorldController {  
 @RequestMapping("/index")  
 public String index() {  
 return "hello world!";  
 }  
}

大家先不用关注项目中代码什么意思，主要关注一下效果。

下面我们来启动项目。

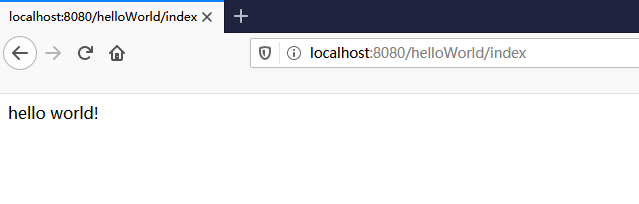
注意项目中有一个自动生成的类：com.javacode2018.SpringDemo1Application，这个类包含了main方法，说明是可以运行的，我们运行一下，控制台输出：

. \_\_\_\_ \_ \_\_ \_ \_  
 /\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \  
( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \  
 \\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )  
 ' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /  
 =========|\_|==============|\_\_\_/=/\_/\_/\_/  
 :: Spring Boot :: (v2.2.4.RELEASE)  
  
2020-02-03 17:42:52.965 INFO 985848 --- [ main] com.javacode2018.SpringDemo1Application : Starting SpringDemo1Application on DESKTOP-3OB6NA3 with PID 985848 (E:\spring\spring-demo1\target\classes started by Think in E:\spring\spring-demo1)  
2020-02-03 17:42:52.968 INFO 985848 --- [ main] com.javacode2018.SpringDemo1Application : No active profile set, falling back to default profiles: default  
2020-02-03 17:42:55.440 INFO 985848 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)  
2020-02-03 17:42:55.450 INFO 985848 --- [ main] o.apache.catalina.core.StandardService : Starting service [Tomcat]  
2020-02-03 17:42:55.451 INFO 985848 --- [ main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]  
2020-02-03 17:42:55.554 INFO 985848 --- [ main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext  
2020-02-03 17:42:55.554 INFO 985848 --- [ main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2546 ms  
2020-02-03 17:42:55.691 INFO 985848 --- [ main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'  
2020-02-03 17:42:55.800 INFO 985848 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''  
2020-02-03 17:42:55.802 INFO 985848 --- [ main] com.javacode2018.SpringDemo1Application : Started SpringDemo1Application in 3.219 seconds (JVM running for 4.096)  
2020-02-03 17:43:17.691 INFO 985848 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'  
2020-02-03 17:43:17.691 INFO 985848 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
2020-02-03 17:43:17.695 INFO 985848 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms

打开浏览器，访问下面地址：

http://localhost:8080/helloWorld/index

效果如下图：



此时我们需要的功能已经实现了，然而到目前为止，只有HelloWorldController代码是我们写的，而我们没有去创建servlet、将返回的结果以json格式输出、配置web.xml，没有将项目发布到tomcat，没有去启动tomcat，而只启动了一个main方法，是不是非常神奇。创建servlet、通过response将数据以json格式输出、配置web.xml、发布项目到tomcat、启动tomcat等这些操作并不是说不需要，而是springboot内部已经帮我们实现了，对于开发者来说这些都被隐藏了，springboot让我们更关注业务代码的实现，从而最大限度的帮我们提升开发效率，简化项目的开发过程。

### 总结

这篇文章主要是让大家了解一下spring是什么，总的来说spring是一个非常优秀的java框架，这个框架的目标是为了简化企业级开发中需要用到的各种技术而存在的，后面的文章我们将一一介绍spring是如何做到简化项目开发的，如何让我们的项目更容易维护和扩展的，敬请期待！

## Spring系列第2篇：控制反转（IoC）与依赖注入（DI）

Spring中有3个核心的概念：**控制反转(Ioc)、依赖注入(DI)、面向切面编程(AOP)**，spring中其他的技术都是依靠3个核心的技术建立起来的，所以玩spring需要先对这3个概念有个深入的理解。

本文我们先介绍其中的两个：控制反转和依赖注入，而aop我们后面有专门的文章详解。

### 引出spring

有2个类，A和B，如下：

public class A{  
 public void m1(){}  
}  
  
public class B{  
 public void m1();  
}

上面2个类都有同样的m1方法。

现在我们调用B的m1方法完成一些事情，而B中的m1方法需要调用A中的m1方法才可以完成这个事情，所以B的代码变成了下面这样：

public class B{  
   
 private A a; // @1  
   
 public B(){  
 this.a = new A(); //@2  
 }  
   
 public void m1(){  
 this.a.m1(); //@3  
 }  
  
}

分析一下上面代码：

@1：B类中声明了一个A类型的属性a

@2：new了一个A对象，赋给了a属性

@3：B类中的m1方法中去调用a.m1()完成业务操作

**先说一下什么是依赖关系？**

当a对象完成某些操作需要调用b对象中的方法来实现时，说明a依赖于对象b，a和b是依赖关系。

上面代码中B的m1需要调用A的m1方法，说明了B依赖于A

**上面代码存在一些问题问题**

**B类中a对象的创建被写死在B的构造方法中了，如果我们想在创建不同的B对象的时候，使用不同的a对象，此时是无能为力的；代码也不利于测试，由于B中a的创建被写死在构造方法中了，我们想测试一下B中不同a对象的效果，此时只能去修改B中的构造方法。**

上面代码需要优化，B中a对象的创建不能写死，可以让外部传入进去，调整一下变成了下面这样：

public class B{  
   
 private A a;  
   
 public B(A a){  
 this.a = a;  
 }  
   
 public void m1(){  
 this.a.m1();   
 }  
}

上面代码可以在创建B对象的时候，将外部创建好的a对象传入进去，此时a的控制权交给了使用者，创建B对象如下：

A a = new A();  
B b = new B(a);  
b.m1();

上面代码我们再扩展一下，如果B类中还需要依赖很多类似于A的对象，比如需要依赖于C、D、E、F或者更多对象，首先是需要调整B的构造方法，修改老的构造方法不是很好，可以在B中新增一些构造方法。

但是使用B的时候就变成了下面这样：

A a = new A();  
C c = new C();  
D d = new D();  
E e = new E();  
F f = new F();  
...  
B b = new B(a,c,d,e,f,...);  
b.m1();

使用者创建B对象之前，需要先将B依赖的对象都给创建好，然后B依赖的这些对象传递给B对象，如果有很多地方都需要用到B类型的对象，都采用这种new的写法，代码量比较大，也不方便维护，如果B中新增了依赖，又需采用new的方式先创建好被依赖的对象，然后将被依赖的对象填充给B对象。

上面创建对象之前，需要先将被依赖对象通过new的方式创建好，然后将其传递给B，这些工作都是B的使用者自己去做的，所有对象的创建都是由使用者自己去控制的，弊端上面也说了，代码量也比较大，代码耦合度比较高（依赖有调整，改动也比较大），也不利于扩展。

那么有没有更好的方式来解决这些问题呢？

上面B对象以及B依赖的对象都是使用者自己主动去控制其创建的，能不能找一个第三方来把这个事情给做了，比如给第三方一个清单，清单中告诉第三方我需要用到B对象以及B需要依赖的对象，然后由这个第三方去负责创建和组装B对象，使用者需要使用B对象的时候，只需要向第三方发起一个查找，如果第三方那边有B对象，直接将其内部组装好的B对象返回就可以了，整个系统中所有需要用到的对象都可以列个清单，让第三方帮忙创造，用的时候只需要向第三方索取就可以了，当B中依赖的对象有新增或者删除的时候，只需要去调整一下清单就可以了，这个事情spring已经帮我们实现了。

### spring容器

spring容器的概念，容器这个名字起的相当好，容器可以放很多东西，我们的程序启动的时候会创建spring容器，会给spring容器一个清单，清单中列出了需要创建的对象以及对象依赖关系，spring容器会创建和组装好清单中的对象，然后将这些对象存放在spring容器中，当程序中需要使用的时候，可以到容器中查找获取，然后直接使用。

### IOC：控制反转

使用者之前使用B对象的时候都需要自己去创建和组装，而现在这些创建和组装都交给spring容器去给完成了，使用者只需要去spring容器中查找需要使用的对象就可以了；这个过程中B对象的创建和组装过程被反转了，之前是使用者自己主动去控制的，现在交给spring容器去创建和组装了，对象的构建过程被反转了，所以叫做控制反转；IOC是是面相对象编程中的一种设计原则，主要是为了降低系统代码的耦合度，让系统利于维护和扩展。

### DI：依赖注入

依赖注入是spring容器中创建对象时给其设置依赖对象的方式，比如给spring一个清单，清单中列出了需要创建B对象以及其他的一些对象（可能包含了B类型中需要依赖对象），此时spring在创建B对象的时候，会看B对象需要依赖于哪些对象，然后去查找一下清单中有没有包含这些被依赖的对象，如果有就去将其创建好，然后将其传递给B对象；可能B需要依赖于很多对象，B创建之前完全不需要知道其他对象是否存在或者其他对象在哪里以及被他们是如何创建，而spring容器会将B依赖对象主动创建好并将其注入到B中去，比如spring容器创建B的时候，发现B需要依赖于A，那么spring容器在清单中找到A的定义并将其创建好之后，注入到B对象中。

### 总结

1. IOC控制反转，是一种设计理念，将对象创建和组装的主动控制权利交给了spring容器去做，控制的动作被反转了，降低了系统的耦合度，利于系统维护和扩展，**主要就是指需要使用的对象的组装控制权被反转了，之前是自己要做的，现在交给spring容器做了**。
2. DI依赖注入，表示spring容器中创建对象时给其设置依赖对象的方式，通过某些注入方式可以让系统更灵活，比如自动注入等可以让系统变的很灵活，这个后面的文章会细说。
3. spring容器：主要负责容器中对象的创建、组装、对象查找、对象生命周期的管理等等操作。
4. **下一篇开始详细讲解spring的使用了**

### MyBatis系列

1. Spring系列第1篇：为何要学spring？

## Spring系列第3篇：Spring容器基本使用及原理

### 环境

1. jdk1.8
2. idea
3. maven-3.6.1
4. spring-5.2.3.RELEASE

### IOC容器

IOC容器是具有依赖注入功能的容器，负责**对象的实例化、对象的初始化，对象和对象之间依赖关系配置、对象的销毁、对外提供对象的查找**等操作，对象的整个生命周期都是由容器来控制。我们需要使用的对象都由ioc容器进行管理，不需要我们再去手动通过new的方式去创建对象，由ioc容器直接帮我们组装好，当我们需要使用的时候直接从ioc容器中直接获取就可以了。

**那么spring ioc容器是如何知道需要管理哪些对象呢？**

需要我们给ioc容器提供一个配置清单，这个配置**支持xml格式**和**java注解的方式**，在配置文件中列出需要让ioc容器管理的对象，以及可以指定让ioc容器如何构建这些对象，当spring容器启动的时候，就会去加载这个配置文件，然后将这些对象给组装好以供外部访问者使用。

这里所说的IOC容器也叫spring容器。

### Bean概念

由spring容器管理的对象统称为Bean对象。Bean就是普通的java对象，和我们自己new的对象其实是一样的，只是这些对象是由spring去创建和管理的，我们需要在配置文件中告诉spring容器需要创建哪些bean对象，所以需要先在配置文件中定义好需要创建的bean对象，这些配置统称为bean定义配置元数据信息，spring容器通过读取这些bean配置元数据信息来构建和组装我们需要的对象。

### Spring容器使用步骤

1. 引入spring相关的maven配置
2. 创建bean配置文件，比如bean xml配置文件
3. 在bean xml文件中定义好需要spring容器管理的bean对象
4. 创建spring容器，并给容器指定需要装载的bean配置文件，当spring容器启动之后，会加载这些配置文件，然后创建好配置文件中定义好的bean对象，将这些对象放在容器中以供使用
5. 通过容器提供的方法获取容器中的对象，然后使用

### Spring容器对象

spring内部提供了很多表示spring容器的接口和对象，我们来看看比较常见的几个容器接口和具体的实现类。

#### BeanFactory接口

org.springframework.beans.factory.BeanFactory

spring容器中具有代表性的容器就是BeanFactory接口，这个是spring容器的顶层接口，提供了容器最基本的功能。

##### 常用的几个方法

//按bean的id或者别名查找容器中的bean  
Object getBean(String name) throws BeansException  
  
//这个是一个泛型方法，按照bean的id或者别名查找指定类型的bean，返回指定类型的bean对象  
<T> T getBean(String name, Class<T> requiredType) throws BeansException;  
  
//返回容器中指定类型的bean对象  
<T> T getBean(Class<T> requiredType) throws BeansException;  
  
//获取指定类型bean对象的获取器，这个方法比较特别，以后会专门来讲  
<T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);

#### ApplicationContext接口

org.springframework.context.ApplicationContext

这个接口继承了BeanFactory接口，所以内部包含了BeanFactory所有的功能，并且在其上进行了扩展，增加了很多企业级功能，比如AOP、国际化、事件支持等等。。

#### ClassPathXmlApplicationContext类

org.springframework.context.support.ClassPathXmlApplicationContext

这个类实现了ApplicationContext接口，注意一下这个类名称包含了ClassPath Xml，说明这个容器类可以从classpath中加载bean xml配置文件，然后创建xml中配置的bean对象，一会后面的案例就会用到这个类。

#### AnnotationConfigApplicationContext类

org.springframework.context.annotation.AnnotationConfigApplicationContext

这个类也实现了ApplicationContext接口，注意其类名包含了Annotation和config两个单词，上面我们有说过，bean的定义支持xml的方式和注解的方式，当我们使用注解的方式定义bean的时候，就需要用到这个容器来装载了，这个容器内部会解析注解来构建构建和管理需要的bean。

注解的方式相对于xml方式更方便一些，也是我们比较推荐的方式，后面我们会大量使用这种方式，具体会详解。

### 案例

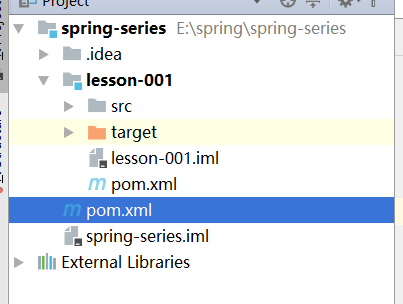
来个helloworld来详细看一下spring如何使用。

#### 创建项目spring-series

使用idea创建maven项目spring-series，项目坐标：

<groupId>com.javacode2018</groupId>  
<artifactId>spring-series</artifactId>  
<packaging>pom</packaging>  
<version>1.0-SNAPSHOT</version>

spring-series项目中创建一个子模块lesson-001，项目maven父子结构，如下图：



#### spring-series/pom.xml

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.javacode2018</groupId>  
 <artifactId>spring-series</artifactId>  
 <packaging>pom</packaging>  
 <version>1.0-SNAPSHOT</version>  
 <modules>  
 <module>lesson-001</module>  
 </modules>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <!-- 配置maven编译的时候采用的编译器版本 -->  
 <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>  
 <!-- 指定源代码是什么版本的，如果源码和这个版本不符将报错，maven中执行编译的时候会用到这个配置，默认是1.5，这个相当于javac命令后面的-source参数 -->  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <!-- 该命令用于指定生成的class文件将保证和哪个版本的虚拟机进行兼容，maven中执行编译的时候会用到这个配置，默认是1.5，这个相当于javac命令后面的-target参数 -->  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <spring.version>5.2.3.RELEASE</spring.version>  
 </properties>  
  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-beans</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
</project>

目前我们使用spring最新的版本5.2.3.RELEASE，需要引入spring提供的3个构件

spring-core、spring-context、spring-beans

#### lesson-001\pom.xml

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <parent>  
 <artifactId>spring-series</artifactId>  
 <groupId>com.javacode2018</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
  
 <artifactId>lesson-001</artifactId>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-beans</artifactId>  
 </dependency>  
 </dependencies>  
  
</project>

#### lesson-001中创建HelloWord类

package com.javacode2018.lesson001.demo1;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class HelloWorld {  
 public void say() {  
 System.out.println("hello,欢迎和【马士兵老师】一起学spring!");  
 }  
}

HelloWord中我们创建了一个say方法，里面会输一段文字。

#### 使用spring容器

下面我们通过spring容器来创建HelloWord对象，并从容器中获取这个对象，然后调用其say方法输出文字。

#### 创建bean xml配置文件

新建一个文件，文件路径如下：

spring-series\lesson-001\src\main\resources\com\javacode2018\lesson001\demo1\bean.xml

bean.xml内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <!--  
 定义一个bean  
 id：bean的唯一标识，可以通过这个标识从容器中获取这个bean对象  
 clss：这个bean的类型，完整类名称  
 -->  
 <bean id="helloWorld" class="com.javacode2018.lesson001.demo1.HelloWorld"/>  
  
</beans>

上面就是bean的定义文件，每个xml中可以定义多个bean元素，通过bean元素定义需要spring容器管理的对象，bean元素需指定id和class属性

* id表示这个bean的标识，在容器中需要唯一，可以通过这个id从容器中获取这个对象；
* class用来指定这个bean的完整类名

上面的配置文件中我们定义了一个helloWorld标识的HellWorld类型的bean对象。

#### 创建测试类

创建一个Client类，如下：

package com.javacode2018.lesson001.demo1;  
  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class Client {  
  
 public static void main(String[] args) {  
 //1.bean配置文件位置  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo1/beans.xml";  
  
 //2.创建ClassPathXmlApplicationContext容器，给容器指定需要加载的bean配置文件  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 //3.从容器中获取需要的bean  
 HelloWorld helloWorld = context.getBean("helloWorld", HelloWorld.class);  
  
 //4.使用对象  
 helloWorld.say();  
  
 }  
}

上面main方法中有容器的详细使用步骤，需要先创建容器对象，创建容器的对象的时候需要指定bean xml文件的位置，容器启动之后会加载这些配文件，然后将这些对象构建好。

代码中通过容器提供的getBean方法从容器中获取了HellWorld对象，第一个参数就是xml中bean的id，第二个参数为bean对应的Class对象。

#### 运行输出

hello,欢迎和【马士兵老师】一起学spring!

### 总结

本文主要介绍了spring容器的概念、bean的概念、常见的spring容器，以及spring容器的使用步骤；下一篇我们将详解bean的定义。

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）

## Spring系列第4篇：xml中bean定义详解

### 本文内容

1. bean xml配置文件
2. bean元素详解
3. 名称和别名详解
4. alias元素详解
5. 通过import元素引入外部配置



### 环境

1. jdk1.8
2. idea
3. maven-3.6.1
4. spring-5.2.3.RELEASE

### bean概念回顾

我们再来回顾一下，被spring管理的对象统称为bean，我们程序中需要用到很多对象，我们将这些对象让spring去帮我们创建和管理，我们可以通过bean xml配置文件告诉spring容器需要管理哪些bean，spring帮我们创建和组装好这些bean对象；那么我们如何从spring中获取想要的bean对象呢，我们需要给bean定义一个名称，spring内部将这些名称和具体的bean对象进行绑定，然后spring容器可以通过这个的名称找对我们需要的对象，这个名称叫做bean的名称，在一个spring容器中需要是唯一的。

### bean xml配置文件格式

bean xml文件用于定义spring容器需要管理的bean，常见的格式如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
   
 <import resource="引入其他bean xml配置文件" />  
 <bean id="bean标识" class="玩转类型名称"/>  
 <alias name="bean标识" alias="别名" />  
  
</beans>

beans是根元素，下面可以包含任意数量的import、bean、alias元素，下面我们对每个元素进行详解。

### bean元素

用来定义一个bean对象。

#### 格式

<bean id="bean唯一标识" name="bean名称" class="完整类型名称" factory-bean="工厂bean名称" factory-method="工厂方法"/>

#### bean名称

每个bean都有一个名称，叫做bean名称，bean名称在一个spring容器中必须唯一，否则会报错，通过bean名称可以从spring容器获取对应的bean对象。

#### bean别名

先来说一下什么是别名？

相当于人的外号一样，一个人可能有很多外号，当别人喊这个人的名称或者外号的时候，都可以找到这个人。那么bean也一样，也可以给bean起几个外号，这个外号在spring中叫做bean的别名，spring容器允许使用者通过名称或者别名获取对应的bean对象。

#### bean名称别名定义规则

名称和别名可以通过bean元素中的id和name来定义，具体定义规则如下：：

1. 当id存在的时候，不管name有没有，取id为bean的名称
2. 当id不存在，此时需要看name，name的值可以通过,;或者空格分割，最后会按照分隔符得到一个String数组，数组的第一个元素作为bean的名称，其他的作为bean的别名
3. 当id和name都存在的时候，id为bean名称，name用来定义多个别名
4. 当id和name都不指定的时候，bean名称自动生成，生成规则下面详细说明

#### 案例

下面演示一下bean名称和别名的各种写法。

<!-- 通过id定义bean名称：user1 -->  
<bean id="user1" class="com.javacode2018.lesson001.demo2.UserModel"/>  
  
<!-- 通过name定义bean名称：user2 -->  
<bean name="user2" class="com.javacode2018.lesson001.demo2.UserModel"/>  
  
<!-- id为名称，name为别名；bean名称：user3，1个别名：[user3\_1] -->  
<bean id="user3" name="user3\_1" class="com.javacode2018.lesson001.demo2.UserModel"/>  
  
<!-- bean名称：user4，多个别名：[user4\_1,user4\_2,user4\_3,user4\_4] -->  
<bean id="user4" name="user4\_1,user4\_2;user4\_3 user4\_4" class="com.javacode2018.lesson001.demo2.UserModel"/>  
  
<!-- bean名称：user5，别名：[user5\_1,user5\_2,user5\_3,user5\_4] -->  
<bean name="user5,user5\_1,user5\_2;user5\_3 user5\_4" class="com.javacode2018.lesson001.demo2.UserModel"/>

我们来写个java来输出上面所有bean的名称和其别名，如下：

package com.javacode2018.lesson001.demo2;  
  
import org.springframework.beans.factory.BeanFactory;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.util.Arrays;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class Client {  
  
 public static void main(String[] args) {  
 //1.bean配置文件位置  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo2/beans.xml";  
  
 //2.创建ClassPathXmlApplicationContext容器，给容器指定需要加载的bean配置文件  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 for (String beanName : Arrays.asList("user1", "user2", "user3", "user4", "user5")) {  
 //获取bean的别名  
 String[] aliases = context.getAliases(beanName);  
 System.out.println(String.format("beanName:%s,别名:[%s]", beanName, String.join(",", aliases)));  
 }  
  
 System.out.println("spring容器中所有bean如下：");  
  
 //getBeanDefinitionNames用于获取容器中所有bean的名称  
 for (String beanName : context.getBeanDefinitionNames()) {  
 //获取bean的别名  
 String[] aliases = context.getAliases(beanName);  
 System.out.println(String.format("beanName:%s,别名:[%s]", beanName, String.join(",", aliases)));  
 }  
  
 }  
}

上面有2个新的方法：

getAliases：通过bean名称获取这个bean的所有别名

getBeanDefinitionNames：返回spring容器中定义的所有bean的名称

运行输出：

beanName:user1,别名:[]  
beanName:user2,别名:[]  
beanName:user3,别名:[user3\_1]  
beanName:user4,别名:[user4\_1,user4\_4,user4\_2,user4\_3]  
beanName:user5,别名:[user5\_3,user5\_4,user5\_1,user5\_2]  
spring容器中所有bean如下：  
beanName:user1,别名:[]  
beanName:user2,别名:[]  
beanName:user3,别名:[user3\_1]  
beanName:user4,别名:[user4\_1,user4\_4,user4\_2,user4\_3]  
beanName:user5,别名:[user5\_3,user5\_4,user5\_1,user5\_2]

#### id和name都未指定

当id和name都未指定的时候，bean的名称和别名又是什么呢？此时由spring自动生成，bean名称为：

bean的class的完整类名#编号

上面的编号是从0开始的，同种类型的没有指定名称的依次递增。

如：

下面定义了2个UserModel和2个String类型的bean，这些bean都没有指定id和name。

<bean class="com.javacode2018.lesson001.demo2.UserModel" />  
<bean class="com.javacode2018.lesson001.demo2.UserModel" />  
  
<bean class="java.lang.String" />  
<bean class="java.lang.String" />

我们再次运行一下上面Client类的main方法，输出：

beanName:user1,别名:[]  
beanName:user2,别名:[]  
beanName:user3,别名:[user3\_1]  
beanName:user4,别名:[user4\_1,user4\_4,user4\_2,user4\_3]  
beanName:user5,别名:[user5\_3,user5\_4,user5\_1,user5\_2]  
spring容器中所有bean如下：  
beanName:user1,别名:[]  
beanName:user2,别名:[]  
beanName:user3,别名:[user3\_1]  
beanName:user4,别名:[user4\_1,user4\_4,user4\_2,user4\_3]  
beanName:user5,别名:[user5\_3,user5\_4,user5\_1,user5\_2]  
beanName:com.javacode2018.lesson001.demo2.UserModel#0,别名:[com.javacode2018.lesson001.demo2.UserModel]  
beanName:com.javacode2018.lesson001.demo2.UserModel#1,别名:[]  
beanName:java.lang.String#0,别名:[java.lang.String]  
beanName:java.lang.String#1,别名:[]

注意看一下上面最后4行的输出，bean名称和别名都是自动生成的，未指定id和name的bean对象，第一个会有别名，别名为完整的类名。bean名称为完整类名#编号。

### alias元素

alias元素也可以用来给某个bean定义别名，语法：

<alias name="需要定义别名的bean" alias="别名" />

如：

<bean id="user6" class="com.javacode2018.lesson001.demo2.UserModel" />  
<alias name="user6" alias="user6\_1" />  
<alias name="user6" alias="user6\_2" />

上面第一行通过bean元素定义了一个名称为user6的UserModel对象，后面2行给user6这个bean定义了2个别名，分别是user6*1和user6*2。

运行Client，会输出一行：

beanName:user6,别名:[user6\_2,user6\_1]

### import元素

当我们的系统比较大的时候，会分成很多模块，每个模块会对应一个bean xml文件，我们可以在一个总的bean xml中对其他bean xml进行汇总，相当于把多个bean xml的内容合并到一个里面了，可以通过import元素引入其他bean配置文件。

语法：

<import resource="其他配置文件的位置" />

如：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
   
 <import resource="user.xml" />  
 <import resource="order.xml" />  
   
</beans>

### 总结

本文主要介绍了xml bean的定义，名称和别名的使用，通过import元素引入外部配置文件。

### MyBatis系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理

## Spring系列第5篇：容器创建bean实例有多少种？

### 本文内容

1. 通过反射调用构造方法创建bean对象
2. 通过静态工厂方法创建bean对象
3. 通过实例工厂方法创建bean对象
4. 通过FactoryBean创建bean对象

Spring容器内部创建bean实例对象常见的有4种方式。

### 通过反射调用构造方法创建bean对象

调用类的构造方法获取对应的bean实例，是使用最多的方式，这种方式只需要在xml bean元素中指定class属性，spring容器内部会自动调用该类型的构造方法来创建bean对象，将其放在容器中以供使用。

#### 语法

<bean id="bean名称" name="bean名称或者别名" class="bean的完整类型名称">  
 <constructor-arg index="0" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="1" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="2" value="bean的值" ref="引用的bean名称" />  
 ....  
 <constructor-arg index="n" value="bean的值" ref="引用的bean名称" />  
</bean>

constructor-arg用于指定构造方法参数的值

index：构造方法中参数的位置，从0开始，依次递增

value：指定参数的值

ref：当插入的值为容器内其他bean的时候，这个值为容器中对应bean的名称

#### 案例

##### UserModel类

@Getter  
@Setter  
@ToString  
public class UserModel {  
  
 private String name;  
 private int age;  
  
 public UserModel() {  
 this.name = "我是通过UserModel的无参构造方法创建的!";  
 }  
  
 public UserModel(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}

##### beans.xml配置

<!-- 通过UserModel的默认构造方法创建UserModel对象 -->  
<bean id="createBeanByConstructor1" class="com.javacode2018.lesson001.demo3.UserModel"/>  
  
<!-- 通过UserModel有参构造方法创建UserModel对象 -->  
<bean id="createBeanByConstructor2" class="com.javacode2018.lesson001.demo3.UserModel">  
 <constructor-arg index="0" value="我是通过UserModel的有参方法构造的对象!"/>  
 <constructor-arg index="1" value="30"/>  
</bean>

上面这2种写法，spring容器创建这两个UserModel的时候，都会通过反射的方式去调用UserModel类中对应的构造函数来创建UserModel对象。

##### 测试用例

package com.javacode2018.lesson001.demo3;  
  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.net.URL;  
import java.net.URLClassLoader;  
import java.util.Arrays;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class Client {  
  
 public static void main(String[] args) {  
 //1.bean配置文件位置  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo3/beans.xml";  
  
 //2.创建ClassPathXmlApplicationContext容器，给容器指定需要加载的bean配置文件  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println("spring容器中所有bean如下：");  
  
 //getBeanDefinitionNames用于获取容器中所有bean的名称  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + ":" + context.getBean(beanName));  
 }  
  
 }  
}

代码中会输出spring容器中所有bean的名称和其对应的bean对象。

##### 运行输出

spring容器中所有bean如下：  
createBeanByConstructor1:UserModel(name=我是通过UserModel的无参构造方法创建的!, age=0)  
createBeanByConstructor2:UserModel(name=我是通过UserModel的有参方法构造的对象!, age=30)

### 通过静态工厂方法创建bean对象

我们可以创建静态工厂，内部提供一些静态方法来生成所需要的对象，将这些静态方法创建的对象交给spring以供使用。

#### 语法

<bean id="bean名称" name="" class="静态工厂完整类名" factory-method="静态工厂的方法">  
 <constructor-arg index="0" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="1" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="2" value="bean的值" ref="引用的bean名称" />  
 ....  
 <constructor-arg index="n" value="bean的值" ref="引用的bean名称" />  
</bean>

class：指定静态工厂完整的类名

factory-method：静态工厂中的静态方法，返回需要的对象。

constructor-arg用于指定静态方法参数的值，用法和上面介绍的构造方法一样。

spring容器会自动调用静态工厂的静态方法获取指定的对象，将其放在容器中以供使用。

#### 案例

##### 定义静态工厂

创建一个静态工厂类，用于生成UserModel对象。

package com.javacode2018.lesson001.demo3;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class UserStaticFactory {  
  
 /\*\*  
 \* 静态无参方法创建UserModel  
 \*  
 \* @return  
 \*/  
 public static UserModel buildUser1() {  
  
 System.out.println(UserStaticFactory.class + ".buildUser1()");  
  
 UserModel userModel = new UserModel();  
 userModel.setName("我是无参静态构造方法创建的!");  
 return userModel;  
 }  
  
 /\*\*  
 \* 静态有参方法创建UserModel  
 \*  
 \* @param name 名称  
 \* @param age 年龄  
 \* @return  
 \*/  
 public static UserModel buildUser2(String name, int age) {  
  
 System.out.println(UserStaticFactory.class + ".buildUser2()");  
  
 UserModel userModel = new UserModel();  
 userModel.setName(name);  
 userModel.setAge(age);  
 return userModel;  
 }  
}

##### beans.xml配置

<!-- 通过工厂静态无参方法创建bean对象 -->  
<bean id="createBeanByStaticFactoryMethod1" class="com.javacode2018.lesson001.demo3.UserStaticFactory"  
 factory-method="buildUser1"/>  
  
<!-- 通过工厂静态有参方法创建bean对象 -->  
<bean id="createBeanByStaticFactoryMethod2" class="com.javacode2018.lesson001.demo3.UserStaticFactory"  
 factory-method="buildUser2">  
 <constructor-arg index="0" value="通过工厂静态有参方法创建UerModel实例对象"/>  
 <constructor-arg index="1" value="30"/>  
</bean>

上面配置中，spring容器启动的时候会自动调用UserStaticFactory中的buildUser1静态方法获取UserModel对象，将其作为createBeanByStaticFactoryMethod1名称对应的bean对象放在spring容器中。

会调用UserStaticFactory的buildUser2方法，并且会传入2个指定的参数，得到返回的UserModel对象，将其作为createBeanByStaticFactoryMethod2名称对应的bean对象放在spring容器中。

##### 运行Client

class com.javacode2018.lesson001.demo3.UserStaticFactory.buildUser1()  
class com.javacode2018.lesson001.demo3.UserStaticFactory.buildUser2()  
spring容器中所有bean如下：  
createBeanByStaticFactoryMethod1:UserModel(name=我是无参静态构造方法创建的!, age=0)  
createBeanByStaticFactoryMethod2:UserModel(name=通过工厂静态有参方法创建UerModel实例对象, age=30)

从输出中可以看出，两个静态方法都被调用了，createBeanByStaticFactoryMethod1对应的bean对象是通过buildUser1方法创建的；createBeanByStaticFactoryMethod2对应的bean对象是通过buildUser2方法创建的。

### 通过实例工厂方法创建bean对象

让spring容器去调用某些对象的某些实例方法来生成bean对象放在容器中以供使用。

#### 语法

<bean id="bean名称" factory-bean="需要调用的实例对象bean名称" factory-method="bean对象中的方法">  
 <constructor-arg index="0" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="1" value="bean的值" ref="引用的bean名称" />  
 <constructor-arg index="2" value="bean的值" ref="引用的bean名称" />  
 ....  
 <constructor-arg index="n" value="bean的值" ref="引用的bean名称" />  
</bean>

spring容器以factory-bean的值为bean名称查找对应的bean对象，然后调用该对象中factory-method属性值指定的方法，将这个方法返回的对象作为当前bean对象放在容器中供使用。

#### 案例

##### 定义一个实例工厂

内部写2个方法用来创建UserModel对象。

package com.javacode2018.lesson001.demo3;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class UserFactory {  
 public UserModel buildUser1() {  
 System.out.println("----------------------1");  
 UserModel userModel = new UserModel();  
 userModel.setName("bean实例方法创建的对象!");  
 return userModel;  
 }  
  
 public UserModel buildUser2(String name, int age) {  
 System.out.println("----------------------2");  
 UserModel userModel = new UserModel();  
 userModel.setName(name);  
 userModel.setAge(age);  
 return userModel;  
 }  
}

##### beans.xml

<!-- 定义一个工厂实例 -->  
<bean id="userFactory" class="com.javacode2018.lesson001.demo3.UserFactory"/>  
<!-- 通过userFactory实例的无参user方法创建UserModel对象 -->  
<bean id="createBeanByBeanMethod1" factory-bean="userFactory" factory-method="buildUser1"/>  
  
<!-- 通过userFactory实例的有参user方法创建UserModel对象 -->  
<bean id="createBeanByBeanMethod2" factory-bean="userFactory" factory-method="buildUser2">  
 <constructor-arg index="0" value="通过bean实例有参方法创建UserModel实例对象"/>  
 <constructor-arg index="1" value="30"/>  
</bean>

createBeanByBeanMethod1对应的bean是通过userFactory的buildUser1方法生成的。

createBeanByBeanMethod2对应的bean是通过userFactory的buildUser2方法生成的。

##### 运行Client

spring容器中所有bean如下：  
createBeanByBeanMethod1:UserModel(name=bean实例方法创建的对象!, age=0)  
createBeanByBeanMethod2:UserModel(name=通过bean实例有参方法创建UserModel实例对象, age=30)

### 通过FactoryBean来创建bean对象

前面我们学过了BeanFactory接口，BeanFactory是spring容器的顶层接口，而这里要说的是FactoryBean，也是一个接口，这两个接口很容易搞混淆，FactoryBean可以让spring容器通过这个接口的实现来创建我们需要的bean对象。

FactoryBean接口源码：

public interface FactoryBean<T> {  
  
 /\*\*  
 \* 返回创建好的对象  
 \*/  
 @Nullable  
 T getObject() throws Exception;  
  
 /\*\*  
 \* 返回需要创建的对象的类型  
 \*/  
 @Nullable  
 Class<?> getObjectType();  
  
 /\*\*  
 \* bean是否是单例的  
 \*\*/  
 default boolean isSingleton() {  
 return true;  
 }  
  
}

接口中有3个方法，前面2个方法需要我们去实现，getObject方法内部由开发者自己去实现对象的创建，然后将创建好的对象返回给Spring容器，getObjectType需要指定我们创建的bean的类型；最后一个方法isSingleton表示通过这个接口创建的对象是否是单例的，如果返回false，那么每次从容器中获取对象的时候都会调用这个接口的getObject() 去生成bean对象。

#### 语法

<bean id="bean名称" class="FactoryBean接口实现类" />

#### 案例

##### 创建一个FactoryBean实现类

package com.javacode2018.lesson001.demo3;  
  
import org.springframework.beans.factory.FactoryBean;  
import org.springframework.lang.Nullable;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class UserFactoryBean implements FactoryBean<UserModel> {  
 int count = 1;  
 @Nullable  
 @Override  
 public UserModel getObject() throws Exception { //@1  
 UserModel userModel = new UserModel();  
 userModel.setName("我是通过FactoryBean创建的第"+count+++ "对象");//@4  
 return userModel;  
 }  
  
 @Nullable  
 @Override  
 public Class<?> getObjectType() {  
 return UserModel.class; //@2  
 }  
  
 @Override  
 public boolean isSingleton() {   
 return true; //@3  
 }  
}

@1：返回了一个创建好的UserModel对象

@2：返回对象的Class对象

@3：返回true，表示创建的对象是单例的，那么我们每次从容器中获取这个对象的时候都是同一个对象

@4：此处用到了一个count，通过这个一会可以看出isSingleton不同返回值的时候从容器获取的bean是否是同一个

##### bean xml配置

<!-- 通过FactoryBean 创建UserModel对象 -->  
<bean id="createByFactoryBean" class="com.javacode2018.lesson001.demo3.UserFactoryBean"/>

##### Client代码

package com.javacode2018.lesson001.demo3;  
  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.net.URL;  
import java.net.URLClassLoader;  
import java.util.Arrays;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class Client {  
  
 public static void main(String[] args) {  
 //1.bean配置文件位置  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo3/beans.xml";  
  
 //2.创建ClassPathXmlApplicationContext容器，给容器指定需要加载的bean配置文件  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println("spring容器中所有bean如下：");  
  
 //getBeanDefinitionNames用于获取容器中所有bean的名称  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + ":" + context.getBean(beanName));  
 }  
  
 System.out.println("--------------------------");  
 //多次获取createByFactoryBean看看是否是同一个对象  
 System.out.println("createByFactoryBean:" + context.getBean("createByFactoryBean"));  
 System.out.println("createByFactoryBean:" + context.getBean("createByFactoryBean"));  
 }  
}

运行输出

class com.javacode2018.lesson001.demo3.UserStaticFactory.buildUser1()  
class com.javacode2018.lesson001.demo3.UserStaticFactory.buildUser2()  
----------------------1  
----------------------2  
spring容器中所有bean如下：  
createBeanByConstructor1:UserModel(name=我是通过UserModel的无参构造方法创建的!, age=0)  
createBeanByConstructor2:UserModel(name=我是通过UserModel的有参方法构造的对象!, age=30)  
createBeanByStaticFactoryMethod1:UserModel(name=我是无参静态构造方法创建的!, age=0)  
createBeanByStaticFactoryMethod2:UserModel(name=通过工厂静态有参方法创建UerModel实例对象, age=30)  
userFactory:com.javacode2018.lesson001.demo3.UserFactory@610694f1  
createBeanByBeanMethod1:UserModel(name=bean实例方法创建的对象!, age=0)  
createBeanByBeanMethod2:UserModel(name=通过bean实例有参方法创建UserModel实例对象, age=30)  
createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第1对象, age=0)  
--------------------------  
createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第1对象, age=0)  
createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第1对象, age=0)

注意最后4行输出，有3行输出的都是同一个createByFactoryBean，程序中通过getBean从spring容器中查找createByFactoryBean了3次，3次结果都是一样的，说明返回的都是同一个UserModel对象。

下面我们将UserFactoryBean中的isSingleton调整一下，返回false

@Override  
public boolean isSingleton() {  
 return false;  
}

当这个方法返回false的时候，表示由这个FactoryBean创建的对象是多例的，那么我们每次从容器中getBean的时候都会去重新调用FactoryBean中的getObject方法获取一个新的对象。

再运行一下Client，最后4行输出：

createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第1对象, age=0)  
--------------------------  
createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第2对象, age=0)  
createByFactoryBean:UserModel(name=我是通过FactoryBean创建的第3对象, age=0)

这3次获取的对象不一样了。

### 总结

spring容器提供了4种创建bean实例的方式，除了构造函数的方式，其他几种方式可以让我们手动去控制对象的创建，这几种方式大家都掌握一下，能够灵活使用。

### 案例代码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)

## Spring系列第6篇：bean作用域scope详解

### 本文内容

1. 详细介绍5中bean的sope及使用注意点
2. 自定义作用域的实现

应用中，有时候我们需要一个对象在整个应用中只有一个，有些对象希望每次使用的时候都重新创建一个，spring对我们这种需求也提供了支持，在spring中这个叫做bean的作用域，xml中定义bean的时候，可以通过scope属性指定bean的作用域，如：

<bean id="" class="" scope="作用域" />

spring容器中scope常见的有5种，下面我们分别来介绍一下。

### singleton

当scope的值设置为singleton的时候，整个spring容器中只会存在一个bean实例，通过容器多次查找bean的时候（调用BeanFactory的getBean方法或者bean之间注入依赖的bean对象的时候），返回的都是同一个bean对象，singleton是scope的默认值，所以spring容器中默认创建的bean对象是单例的，通常spring容器在启动的时候，会将scope为singleton的bean创建好放在容器中（有个特殊的情况，当bean的lazy被设置为true的时候，表示懒加载，那么使用的时候才会创建），用的时候直接返回。

#### 案例

##### bean xml配置

<!-- 单例bean，scope设置为singleton -->  
<bean id="singletonBean" class="com.javacode2018.lesson001.demo4.BeanScopeModel" scope="singleton">  
 <constructor-arg index="0" value="singleton"/>  
</bean>

##### BeanScopeModel代码

package com.javacode2018.lesson001.demo4;  
  
public class BeanScopeModel {  
 public BeanScopeModel(String beanScope) {  
 System.out.println(String.format("create BeanScopeModel,{sope=%s},{this=%s}", beanScope, this));  
 }  
}

上面构造方法中输出了一段文字，一会我们可以根据输出来看一下这个bean什么时候创建的，是从容器中获取bean的时候创建的还是容器启动的时候创建的。

##### 测试用例

package com.javacode2018.lesson001.demo4;  
  
import org.junit.Before;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* <p>  
 \* bean作用域  
 \*/  
public class ScopeTest {  
  
 ClassPathXmlApplicationContext context;  
  
 @Before  
 public void before() {  
 System.out.println("spring容器准备启动.....");  
 //1.bean配置文件位置  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo4/beans.xml";  
 //2.创建ClassPathXmlApplicationContext容器，给容器指定需要加载的bean配置文件  
 this.context = new ClassPathXmlApplicationContext(beanXml);  
 System.out.println("spring容器启动完毕！");  
 }  
  
 /\*\*  
 \* 单例bean  
 \*/  
 @Test  
 public void singletonBean() {  
 System.out.println("---------单例bean，每次获取的bean实例都一样---------");  
 System.out.println(context.getBean("singletonBean"));  
 System.out.println(context.getBean("singletonBean"));  
 System.out.println(context.getBean("singletonBean"));  
 }  
  
}

上面代码中before方法上面有@Before注解，这个是junit提供的功能，这个方法会在所有@Test标注的方法之前之前运行，before方法中我们对容器进行初始化，并且在容器初始化前后输出了一段文字。

上面代码中，singletonBean方法中，3次获取singletonBean对应的bean。

##### 运行测试用例

spring容器准备启动.....  
create BeanScopeModel,{sope=singleton},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@e874448}  
spring容器启动完毕！  
---------单例bean，每次获取的bean实例都一样---------  
com.javacode2018.lesson001.demo4.BeanScopeModel@e874448  
com.javacode2018.lesson001.demo4.BeanScopeModel@e874448  
com.javacode2018.lesson001.demo4.BeanScopeModel@e874448

##### 结论

从输出中得到2个结论

* 前3行的输出可以看出，BeanScopeModel的构造方法是在容器启动过程中调用的，说明这个bean实例在容器启动过程中就创建好了，放在容器中缓存着
* 最后3行输出的是一样的，说明返回的是同一个bean对象

#### 单例bean使用注意

**单例bean是整个应用共享的，所以需要考虑到线程安全问题，之前在玩springmvc的时候，springmvc中controller默认是单例的，有些开发者在controller中创建了一些变量，那么这些变量实际上就变成共享的了，controller可能会被很多线程同时访问，这些线程并发去修改controller中的共享变量，可能会出现数据错乱的问题；所以使用的时候需要特别注意。**

### prototype

如果scope被设置为prototype类型的了，表示这个bean是多例的，通过容器每次获取的bean都是不同的实例，每次获取都会重新创建一个bean实例对象。

#### 案例

##### bean xml配置

<!-- 多例bean，scope设置为prototype-->  
<bean id="prototypeBean" class="com.javacode2018.lesson001.demo4.BeanScopeModel" scope="prototype">  
 <constructor-arg index="0" value="prototype"/>  
</bean>

##### 新增一个测试用例

ScopeTest中新增一个方法

/\*\*  
 \* 多例bean  
 \*/  
@Test  
public void prototypeBean() {  
 System.out.println("---------单例bean，每次获取的bean实例都一样---------");  
 System.out.println(context.getBean("prototypeBean"));  
 System.out.println(context.getBean("prototypeBean"));  
 System.out.println(context.getBean("prototypeBean"));  
}

##### 运行测试用例

spring容器准备启动.....  
spring容器启动完毕！  
---------单例bean，每次获取的bean实例都一样---------  
create BeanScopeModel,{sope=prototype},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@289d1c02}  
com.javacode2018.lesson001.demo4.BeanScopeModel@289d1c02  
create BeanScopeModel,{sope=prototype},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@22eeefeb}  
com.javacode2018.lesson001.demo4.BeanScopeModel@22eeefeb  
create BeanScopeModel,{sope=prototype},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@17d0685f}  
com.javacode2018.lesson001.demo4.BeanScopeModel@17d0685f

##### 结论

输出中可以看出，容器启动过程中并没有去创建BeanScopeModel对象，3次获取prototypeBean得到的都是不同的实例，每次获取的时候才会去调用构造方法创建bean实例。

#### 多例bean使用注意

**多例bean每次获取的时候都会重新创建，如果这个bean比较复杂，创建时间比较长，会影响系统的性能，这个地方需要注意。**

**下面要介绍的3个：request、session、application都是在spring web容器环境中才会有的。**

### request

当一个bean的作用域为request，表示在一次http请求中，一个bean对应一个实例；对每个http请求都会创建一个bean实例，request结束的时候，这个bean也就结束了，request作用域用在spring容器的web环境中，这个以后讲springmvc的时候会说，spring中有个web容器接口WebApplicationContext，这个里面对request作用域提供了支持，配置方式：

<bean id="" class="" scope="request" />

### session

这个和request类似，也是用在web环境中，session级别共享的bean，每个会话会对应一个bean实例，不同的session对应不同的bean实例，springmvc中我们再细说。

<bean id="" class="" scope="session" />

### application

全局web应用级别的作用于，也是在web环境中使用的，一个web应用程序对应一个bean实例，通常情况下和singleton效果类似的，不过也有不一样的地方，singleton是每个spring容器中只有一个bean实例，一般我们的程序只有一个spring容器，但是，一个应用程序中可以创建多个spring容器，不同的容器中可以存在同名的bean，但是sope=aplication的时候，不管应用中有多少个spring容器，这个应用中同名的bean只有一个。

<bean id="" class="" scope="application" />

### 自定义scope

有时候，spring内置的几种sope都无法满足我们的需求的时候，我们可以自定义bean的作用域。

#### 自定义Scope 3步骤

##### 第1步：实现Scope接口

我们来看一下这个接口定义

package org.springframework.beans.factory.config;  
  
import org.springframework.beans.factory.ObjectFactory;  
import org.springframework.lang.Nullable;  
  
public interface Scope {  
  
 /\*\*  
 \* 返回当前作用域中name对应的bean对象  
 \* name：需要检索的bean的名称  
 \* objectFactory：如果name对应的bean在当前作用域中没有找到，那么可以调用这个ObjectFactory来创建这个对象  
 \*\*/  
 Object get(String name, ObjectFactory<?> objectFactory);  
  
 /\*\*  
 \* 将name对应的bean从当前作用域中移除  
 \*\*/  
 @Nullable  
 Object remove(String name);  
  
 /\*\*  
 \* 用于注册销毁回调，如果想要销毁相应的对象,则由Spring容器注册相应的销毁回调，而由自定义作用域选择是不是要销毁相应的对象  
 \*/  
 void registerDestructionCallback(String name, Runnable callback);  
  
 /\*\*  
 \* 用于解析相应的上下文数据，比如request作用域将返回request中的属性。  
 \*/  
 @Nullable  
 Object resolveContextualObject(String key);  
  
 /\*\*  
 \* 作用域的会话标识，比如session作用域将是sessionId  
 \*/  
 @Nullable  
 String getConversationId();  
  
}

##### 第2步：将自定义的scope注册到容器

需要调用org.springframework.beans.factory.config.ConfigurableBeanFactory#registerScope的方法，看一下这个方法的声明

/\*\*  
\* 向容器中注册自定义的Scope  
\*scopeName：作用域名称  
\* scope：作用域对象  
\*\*/  
void registerScope(String scopeName, Scope scope);

##### 第3步：使用自定义的作用域

定义bean的时候，指定bean的scope属性为自定义的作用域名称。

#### 案例

##### 需求

下面我们来实现一个线程级别的bean作用域，同一个线程中同名的bean是同一个实例，不同的线程中的bean是不同的实例。

##### 实现分析

需求中要求bean在线程中是贡献的，所以我们可以通过ThreadLocal来实现，ThreadLocal可以实现线程中数据的共享。

下面我们来上代码。

##### ThreadScope

package com.javacode2018.lesson001.demo4;  
  
import org.springframework.beans.factory.ObjectFactory;  
import org.springframework.beans.factory.config.Scope;  
import org.springframework.lang.Nullable;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.Objects;  
  
/\*\*  
 \* 自定义本地线程级别的bean作用域，不同的线程中对应的bean实例是不同的，同一个线程中同名的bean是同一个实例  
 \*/  
public class ThreadScope implements Scope {  
  
 public static final String THREAD\_SCOPE = "thread";//@1  
  
 private ThreadLocal<Map<String, Object>> beanMap = new ThreadLocal() {  
 @Override  
 protected Object initialValue() {  
 return new HashMap<>();  
 }  
 };  
  
 @Override  
 public Object get(String name, ObjectFactory<?> objectFactory) {  
 Object bean = beanMap.get().get(name);  
 if (Objects.isNull(bean)) {  
 bean = objectFactory.getObject();  
 beanMap.get().put(name, bean);  
 }  
 return bean;  
 }  
  
 @Nullable  
 @Override  
 public Object remove(String name) {  
 return this.beanMap.get().remove(name);  
 }  
  
 @Override  
 public void registerDestructionCallback(String name, Runnable callback) {  
 //bean作用域范围结束的时候调用的方法，用于bean清理  
 System.out.println(name);  
 }  
  
 @Nullable  
 @Override  
 public Object resolveContextualObject(String key) {  
 return null;  
 }  
  
 @Nullable  
 @Override  
 public String getConversationId() {  
 return Thread.currentThread().getName();  
 }  
}

@1：定义了作用域的名称为一个常量thread，可以在定义bean的时候给scope使用

##### BeanScopeModel

package com.javacode2018.lesson001.demo4;  
  
public class BeanScopeModel {  
 public BeanScopeModel(String beanScope) {  
 System.out.println(String.format("线程:%s,create BeanScopeModel,{sope=%s},{this=%s}", Thread.currentThread(), beanScope, this));  
 }  
}

上面的构造方法中会输出当前线程的信息，到时候可以看到创建bean的线程。

##### bean配置文件

beans-thread.xml内容

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <!-- 自定义scope的bean -->  
 <bean id="threadBean" class="com.javacode2018.lesson001.demo4.BeanScopeModel" scope="thread">  
 <constructor-arg index="0" value="thread"/>  
 </bean>  
</beans>

注意上面的scope是我们自定义的，值为thread

##### 测试用例

package com.javacode2018.lesson001.demo4;  
  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.util.concurrent.TimeUnit;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* <p>  
 \* 自定义scope  
 \*/  
public class ThreadScopeTest {  
 public static void main(String[] args) throws InterruptedException {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo4/beans-thread.xml";  
 //手动创建容器  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(){  
 @Override  
 protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {  
 //向容器中注册自定义的scope  
 beanFactory.registerScope(ThreadScope.THREAD\_SCOPE, new ThreadScope());//@1  
 super.postProcessBeanFactory(beanFactory);  
 }  
 };  
 //设置配置文件位置  
 context.setConfigLocation(beanXml);  
 //启动容器  
 context.refresh();  
  
 //使用容器获取bean  
 for (int i = 0; i < 2; i++) {//@2  
 new Thread(() -> {  
 System.out.println(Thread.currentThread() + "," + context.getBean("threadBean"));  
 System.out.println(Thread.currentThread() + "," + context.getBean("threadBean"));  
 }).start();  
 TimeUnit.SECONDS.sleep(1);  
 }  
 }  
}

注意上面代码，重点在@1，这个地方向容器中注册了自定义的ThreadScope。

@2：创建了2个线程，然后在每个线程中去获取同样的bean 2次，然后输出，我们来看一下效果。

##### 运行输出

线程:Thread[Thread-1,5,main],create BeanScopeModel,{sope=thread},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@4049d530}  
Thread[Thread-1,5,main],com.javacode2018.lesson001.demo4.BeanScopeModel@4049d530  
Thread[Thread-1,5,main],com.javacode2018.lesson001.demo4.BeanScopeModel@4049d530  
线程:Thread[Thread-2,5,main],create BeanScopeModel,{sope=thread},{this=com.javacode2018.lesson001.demo4.BeanScopeModel@87a76da}  
Thread[Thread-2,5,main],com.javacode2018.lesson001.demo4.BeanScopeModel@87a76da  
Thread[Thread-2,5,main],com.javacode2018.lesson001.demo4.BeanScopeModel@87a76da

从输出中可以看到，bean在同样的线程中获取到的是同一个bean的实例，不同的线程中bean的实例是不同的。

### 总结

1. spring容器自带的有2种作用域，分别是singleton和prototype；还有3种分别是spring web容器环境中才支持的request、session、application
2. singleton是spring容器默认的作用域，一个spring容器中同名的bean实例只有一个，多次获取得到的是同一个bean；单例的bean需要考虑线程安全问题
3. prototype是多例的，每次从容器中获取同名的bean，都会重新创建一个；多例bean使用的时候需要考虑创建bean对性能的影响
4. 一个应用中可以有多个spring容器
5. 自定义scope 3个步骤，实现Scope接口，将实现类注册到spring容器，使用自定义的sope

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？

## Spring系列第7篇：依赖注入之手动注入

### 本文内容

1. **主要介绍xml中依赖注入的配置**
2. **构造器注入的3种方式详解**
3. **set方法注入详解**
4. **注入容器中的其他bean的2种方式**
5. **其他常见类型注入详解**

### 依赖回顾

通常情况下，系统中类和类之间是有依赖关系的，如果一个类对外提供的功能需要通过调用其他类的方法来实现的时候，说明这两个类之间存在依赖关系，如：

public class UserService{  
 public void insert(UserModel model){  
 //插入用户信息  
 }  
}  
  
public class UserController{  
   
 private UserService userService;  
   
 public void insert(UserModel model){  
 this.userService.insert(model);  
 }  
  
}

UserController中的insert方法中需要调用userService的insert方法，说明UserController依赖于UserService，如果userService不存在，此时UserControler无法对外提供insert操作。

那么我们创建UserController对象的时候如何将给userService设置值呢？通常有2中方法。

#### 依赖对象的初始化方式

##### 通过构造器设置依赖对象

UserController中添加一个有参构造方法，如下：

public class UserController{  
   
 private UserService userService;  
   
 public UserController(UserService userService){  
 this.userService = userService;  
 }  
   
 public void insert(UserModel model){  
 this.userService.insert(model);  
 }  
  
}  
  
//UserController使用  
UserSerivce userService = new UserService();  
UserController userController = new UserController(userService);  
//然后就可以使用userController对象了

##### 通过set方法设置依赖对象

可以在UserController中给userService添加一个set方法，如：

public class UserController{  
   
 private UserService userService;  
   
 public setUserService(UserService userService){  
 this.userService = userService;  
 }  
   
 public void insert(UserModel model){  
 this.userService.insert(model);  
 }  
  
}  
  
//UserController使用  
UserSerivce userService = new UserService();  
UserController userController = new UserController();  
userController.setService(userService);  
//然后就可以使用userController对象了

上面这些操作，将被依赖的对象设置到依赖的对象中，spring容器内部都提供了支持，这个在spirng中叫做依赖注入。

### spring依赖注入

spring中依赖注入主要分为手动注入和自动注入，本文我们主要说一下手动注入，手动注入需要我们明确配置需要注入的对象。

刚才上面我们回顾了，将被依赖方注入到依赖方，通常有2种方式：构造函数的方式和set属性的方式，spring中也是通过这两种方式实现注入的，下面详解2种方式。

### 通过构造器注入

构造器的参数就是被依赖的对象，构造器注入又分为3种注入方式：

* 根据构造器参数索引注入
* 根据构造器参数类型注入
* 根据构造器参数名称注入

### 根据构造器参数索引注入

#### 用法

<bean id="diByConstructorParamIndex" class="com.javacode2018.lesson001.demo5.UserModel">  
 <constructor-arg index="0" value="马士兵老师"/>  
 <constructor-arg index="1" value="上海市"/>  
</bean>

constructor-arg用户指定构造器的参数

index：构造器参数的位置，从0开始

value：构造器参数的值，value只能用来给简单的类型设置值，value对应的属性类型只能为byte,int,long,float,double,boolean,Byte,Long,Float,Double,枚举，spring容器内部注入的时候会将value的值转换为对应的类型。

#### 案例

##### UserModel.java

package com.javacode2018.lesson001.demo5;  
  
public class UserModel {  
 private String name;  
 private int age;  
 //描述信息  
 private String desc;  
  
 public UserModel() {  
 }  
  
 public UserModel(String name, String desc) {  
 this.name = name;  
 this.desc = desc;  
 }  
  
 public UserModel(String name, int age, String desc) {  
 this.name = name;  
 this.age = age;  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "UserModel{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", desc='" + desc + '\'' +  
 '}';  
 }  
}

注意上面的3个构造函数。

##### diByConstructorParamIndex.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <!-- 通过构造器参数的索引注入 -->  
 <bean id="diByConstructorParamIndex" class="com.javacode2018.lesson001.demo5.UserModel">  
 <constructor-arg index="0" value="马士兵老师"/>  
 <constructor-arg index="1" value="我是通过构造器参数位置注入的"/>  
 </bean>  
  
</beans>

上面创建UserModel实例代码相当于下面代码：

UserModel userModel = new UserModel("马士兵老师","我是通过构造器参数类型注入的");

##### 工具类IoUtils.java

package com.javacode2018.lesson001.demo5;  
  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class IocUtil {  
 public static ClassPathXmlApplicationContext context(String beanXml) {  
 return new ClassPathXmlApplicationContext(beanXml);  
 }  
}

##### 测试用例DiTest.java

package com.javacode2018.lesson001.demo5;  
  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public class DiTest {  
  
 /\*\*  
 \* 通过构造器的参数位置注入  
 \*/  
 @Test  
 public void diByConstructorParamIndex() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diByConstructorParamIndex.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diByConstructorParamIndex"));  
 }  
  
}

##### 效果

运行diByConstructorParamIndex输出

UserModel{name='马士兵老师', age=0, desc='我是通过构造器参数位置注入的'}

#### 优缺点

**参数位置的注入对参数顺序有很强的依赖性，若构造函数参数位置被人调整过，会导致注入出错。**

**不过通常情况下，不建议去在代码中修改构造函数，如果需要新增参数的，可以新增一个构造函数来实现，这算是一种扩展，不会影响目前已有的功能。**

### 根据构造器参数类型注入

#### 用法

<bean id="diByConstructorParamType" class="com.javacode2018.lesson001.demo5.UserModel">  
 <constructor-arg type="参数类型" value="参数值"/>  
 <constructor-arg type="参数类型" value="参数值"/>  
</bean>

constructor-arg用户指定构造器的参数

type：构造函数参数的完整类型，如：java.lang.String,int,double

value：构造器参数的值，value只能用来给简单的类型设置值

#### 案例

##### diByConstructorParamType.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <!-- 通过构造器参数的类型注入 -->  
 <bean id="diByConstructorParamType" class="com.javacode2018.lesson001.demo5.UserModel">  
 <constructor-arg type="int" value="30"/>  
 <constructor-arg type="java.lang.String" value="马士兵老师"/>  
 <constructor-arg type="java.lang.String" value="我是通过构造器参数类型注入的"/>  
 </bean>  
  
</beans>

上面创建UserModel实例代码相当于下面代码：

UserModel userModel = new UserModel("马士兵老师",30,"我是通过构造器参数类型注入的");

##### 新增测试用例

DiTest类中新增一个测试用例

/\*\*  
 \* 通过构造器的参数类型注入  
 \*/  
@Test  
public void diByConstructorParamType() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diByConstructorParamType.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diByConstructorParamType"));  
}

##### 效果

运行diByConstructorParamType输出

UserModel{name='马士兵老师', age=30, desc='我是通过构造器参数类型注入的'}

#### 优缺点

**实际上按照参数位置或者按照参数的类型注入，都有一个问题，很难通过bean的配置文件，知道这个参数是对应UserModel中的那个属性的，代码的可读性不好，比如我想知道这每个参数对应UserModel中的那个属性，必须要去看UserModel的源码，下面要介绍按照参数名称注入的方式比上面这2种更优秀一些。**

### 根据构造器参数名称注入

#### 用法

<bean id="diByConstructorParamName" class="com.javacode2018.lesson001.demo5.UserModel">  
 <constructor-arg name="参数类型" value="参数值"/>  
 <constructor-arg name="参数类型" value="参数值"/>  
</bean>

constructor-arg用户指定构造器的参数

name：构造参数名称

value：构造器参数的值，value只能用来给简单的类型设置值

#### 关于方法参数名称的问题

java通过反射的方式可以获取到方法的参数名称，不过源码中的参数通过编译之后会变成class对象，通常情况下源码变成class文件之后，参数的真实名称会丢失，参数的名称会变成arg0,arg1,arg2这样的，和实际参数名称不一样了，**如果需要将源码中的参数名称保留在编译之后的class文件中，编译的时候需要用下面的命令**：

javac -parameters java源码

但是我们难以保证编译代码的时候，操作人员一定会带上-parameters参数，所以方法的参数可能在class文件中会丢失，导致反射获取到的参数名称和实际参数名称不符，这个我们需要先了解一下。

**参数名称可能不稳定的问题，spring提供了解决方案，通过ConstructorProperties注解来定义参数的名称，将这个注解加在构造方法上面**，如下：

@ConstructorProperties({"第一个参数名称", "第二个参数的名称",..."第n个参数的名称"})  
public 类名(String p1, String p2...,参数n) {  
}

#### 案例

##### CarModel.java

package com.javacode2018.lesson001.demo5;  
  
import java.beans.ConstructorProperties;  
  
public class CarModel {  
 private String name;  
 //描述信息  
 private String desc;  
  
 public CarModel() {  
 }  
  
 @ConstructorProperties({"name", "desc"})  
 public CarModel(String p1, String p2) {  
 this.name = p1;  
 this.desc = p2;  
 }  
  
 @Override  
 public String toString() {  
 return "CarModel{" +  
 "name='" + name + '\'' +  
 ", desc='" + desc + '\'' +  
 '}';  
 }  
}

##### diByConstructorParamName.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <!-- 通过构造器参数的名称注入 -->  
 <bean id="diByConstructorParamName" class="com.javacode2018.lesson001.demo5.CarModel">  
 <constructor-arg name="desc" value="我是通过构造器参数类型注入的"/>  
 <constructor-arg name="name" value="保时捷Macans"/>  
 </bean>  
  
</beans>

上面创建CarModel实例代码相当于下面代码：

CarModel carModel = new CarModel("保时捷Macans","我是通过构造器参数类型注入的");

##### 新增测试用例

DiTest类中新增一个测试用例

/\*\*  
 \* 通过构造器的参数名称注入  
 \*/  
@Test  
public void diByConstructorParamName() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diByConstructorParamName.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diByConstructorParamName"));  
}

##### 效果

运行diByConstructorParamName输出

CarModel{name='保时捷Macans', desc='我是通过构造器参数类型注入的'}

### setter注入

**通常情况下，我们的类都是标准的javabean，javabean类的特点：**

* **属性都是private访问级别的**
* **属性通常情况下通过一组setter（修改器）和getter（访问器）方法来访问**
* **setter方法，以set 开头，后跟首字母大写的属性名，如：setUserName，简单属性一般只有一个方法参数，方法返回值通常为void;**
* **getter方法，一般属性以get开头，对于boolean类型一般以is开头，后跟首字母大写的属性名，如：getUserName，isOk；**

spring对符合javabean特点类，提供了setter方式的注入，会调用对应属性的setter方法将被依赖的对象注入进去。

#### 用法

<bean id="" class="">  
 <property name="属性名称" value="属性值" />  
 ...  
 <property name="属性名称" value="属性值" />  
</bean>

property用于对属性的值进行配置，可以有多个

name：属性的名称

value：属性的值

#### 案例

##### MenuModel.java

package com.javacode2018.lesson001.demo5;  
  
/\*\*  
 \* 菜单类  
 \*/  
public class MenuModel {  
 //菜单名称  
 private String label;  
 //同级别排序  
 private Integer theSort;  
  
 public String getLabel() {  
 return label;  
 }  
  
 public void setLabel(String label) {  
 this.label = label;  
 }  
  
 public Integer getTheSort() {  
 return theSort;  
 }  
  
 public void setTheSort(Integer theSort) {  
 this.theSort = theSort;  
 }  
  
 @Override  
 public String toString() {  
 return "MenuModel{" +  
 "label='" + label + '\'' +  
 ", theSort=" + theSort +  
 '}';  
 }  
}

##### diBySetter.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="diBySetter" class="com.javacode2018.lesson001.demo5.MenuModel">  
 <property name="label" value="spring系列"/>  
 </bean>  
  
</beans>

##### 新增测试用例

DiTest类中添加一个测试方法

/\*\*  
 \* 通过setter方法注入  
 \*/  
@Test  
public void diBySetter() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diBySetter.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diBySetter"));  
}

##### 效果

运行diBySetter输出

MenuModel{label='spring系列', theSort=null}

#### 优缺点

**setter注入相对于构造函数注入要灵活一些，构造函数需要指定对应构造函数中所有参数的值，而setter注入的方式没有这种限制，不需要对所有属性都进行注入，可以按需进行注入。**

上面介绍的都是注入普通类型的对象，都是通过value属性来设置需要注入的对象的值的，value属性的值是String类型的，spring容器内部自动会将value的值转换为对象的实际类型。

**若我们依赖的对象是容器中的其他bean对象的时候，需要用下面的方式进行注入。**

### 注入容器中的bean

**注入容器中的bean有两种写法：**

* **ref属性方式**
* **内置bean的方式**

#### ref属性方式

将上面介绍的constructor-arg或者property元素的value属性名称替换为ref，ref属性的值为容器中其他bean的名称，如：

构造器方式，将value替换为ref：

<constructor-arg ref="需要注入的bean的名称"/>

setter方式，将value替换为ref：

<property name="属性名称" ref="需要注入的bean的名称" />

#### 内置bean的方式

构造器的方式：

<constructor-arg>  
 <bean class=""/>  
</constructor-arg>

setter方式：

<property name="属性名称">  
 <bean class=""/>  
</property>

#### 案例

##### PersonModel.java

package com.javacode2018.lesson001.demo5;  
  
public class PersonModel {  
 private UserModel userModel;  
 private CarModel carModel;  
  
 public PersonModel() {  
 }  
  
 public PersonModel(UserModel userModel, CarModel carModel) {  
 this.userModel = userModel;  
 this.carModel = carModel;  
 }  
  
 public UserModel getUserModel() {  
 return userModel;  
 }  
  
 public void setUserModel(UserModel userModel) {  
 this.userModel = userModel;  
 }  
  
 public CarModel getCarModel() {  
 return carModel;  
 }  
  
 public void setCarModel(CarModel carModel) {  
 this.carModel = carModel;  
 }  
  
 @Override  
 public String toString() {  
 return "PersonModel{" +  
 "userModel=" + userModel +  
 ", carModel=" + carModel +  
 '}';  
 }  
}

PersonModel中有依赖于2个对象UserModel、CarModel，下面我们通过spring将UserModel和CarModel创建好，然后注入到PersonModel中，下面创建bean配置文件

##### diBean.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="user" class="com.javacode2018.lesson001.demo5.UserModel"></bean>  
  
 <!-- 通过构造器方式注入容器中的bean -->  
 <bean id="diBeanByConstructor" class="com.javacode2018.lesson001.demo5.PersonModel">  
 <!-- 通过ref引用容器中定义的其他bean，user对应上面定义的id="user"的bean -->  
 <constructor-arg index="0" ref="user"/>  
 <constructor-arg index="1">  
 <bean class="com.javacode2018.lesson001.demo5.CarModel">  
 <constructor-arg index="0" value="宾利"/>  
 <constructor-arg index="1" value=""/>  
 </bean>  
 </constructor-arg>  
 </bean>  
  
 <!-- 通过setter方式注入容器中的bean -->  
 <bean id="diBeanBySetter" class="com.javacode2018.lesson001.demo5.PersonModel">  
 <!-- 通过ref引用容器中定义的其他bean，user对应上面定义的id="user"的bean -->  
 <property name="userModel" ref="user"/>  
 <property name="carModel">  
 <bean class="com.javacode2018.lesson001.demo5.CarModel">  
 <constructor-arg index="0" value="保时捷"/>  
 <constructor-arg index="1" value=""/>  
 </bean>  
 </property>  
 </bean>  
  
</beans>

##### 新增测试用例

DiTest中新增一个测试方法

@Test  
public void diBean(){  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diBeanByConstructor"));  
 System.out.println(context.getBean("diBeanBySetter"));  
}

##### 效果

运行diBean用例，输出：

PersonModel{userModel=UserModel{name='null', age=0, desc='null'}, carModel=CarModel{name='宾利', desc=''}}  
PersonModel{userModel=UserModel{name='null', age=0, desc='null'}, carModel=CarModel{name='保时捷', desc=''}}

### 其他类型注入

#### 注入java.util.List（list元素）

<list>  
 <value>Spring</value>  
 或  
 <ref bean="bean名称"/>  
 或  
 <list></list>  
 或  
 <bean></bean>  
 或  
 <array></array>  
 或  
 <map></map>  
</list>

#### 注入java.util.Set（set元素）

<set>  
 <value>Spring</value>  
 或  
 <ref bean="bean名称"/>  
 或  
 <list></list>  
 或  
 <bean></bean>  
 或  
 <array></array>  
 或  
 <map></map>  
</set>

#### 注入java.util.Map（map元素）

<map>  
 <entry key="马士兵老师" value="30" key-ref="key引用的bean名称" value-ref="value引用的bean名称"/>  
 或  
 <entry>  
 <key>  
 value对应的值，可以为任意类型  
 </key>  
 <value>  
 value对应的值，可以为任意类型  
 </value>  
 </entry>  
</map>

#### 注入数组（array元素）

<array>  
 数组中的元素  
</array>

#### 注入java.util.Properties（props元素）

Properties类相当于键值都是String类型的Map对象，使用props进行注入，如下：

<props>  
 <prop key="key1">java高并发系列</prop>  
 <prop key="key2">mybatis系列</prop>  
 <prop key="key3">mysql系列</prop>  
</props>

#### 案例

对于上面这些类型来个综合案例。

##### DiOtherTypeModel.java

package com.javacode2018.lesson001.demo5;  
  
import java.util.\*;  
  
public class DiOtherTypeModel {  
 private List<String> list1;  
 private Set<UserModel> set1;  
 private Map<String, Integer> map1;  
 private int[] array1;  
 private Properties properties1;  
  
 public List<String> getList1() {  
 return list1;  
 }  
  
 public void setList1(List<String> list1) {  
 this.list1 = list1;  
 }  
  
 public Set<UserModel> getSet1() {  
 return set1;  
 }  
  
 public void setSet1(Set<UserModel> set1) {  
 this.set1 = set1;  
 }  
  
 public Map<String, Integer> getMap1() {  
 return map1;  
 }  
  
 public void setMap1(Map<String, Integer> map1) {  
 this.map1 = map1;  
 }  
  
 public int[] getArray1() {  
 return array1;  
 }  
  
 public void setArray1(int[] array1) {  
 this.array1 = array1;  
 }  
  
 public Properties getProperties1() {  
 return properties1;  
 }  
  
 public void setProperties1(Properties properties1) {  
 this.properties1 = properties1;  
 }  
  
 @Override  
 public String toString() {  
 return "DiOtherTypeModel{" +  
 "list1=" + list1 +  
 ", set1=" + set1 +  
 ", map1=" + map1 +  
 ", array1=" + Arrays.toString(array1) +  
 ", properties1=" + properties1 +  
 '}';  
 }  
}

上面这个类中包含了刚才我们介绍的各种类型，下面来进行注入。

##### diOtherType.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="user1" class="com.javacode2018.lesson001.demo5.UserModel"/>  
 <bean id="user2" class="com.javacode2018.lesson001.demo5.UserModel"/>  
  
 <bean id="diOtherType" class="com.javacode2018.lesson001.demo5.DiOtherTypeModel">  
 <!-- 注入java.util.List对象 -->  
 <property name="list1">  
 <list>  
 <value>Spring</value>  
 <value>SpringBoot</value>  
 </list>  
 </property>  
  
 <!-- 注入java.util.Set对象 -->  
 <property name="set1">  
 <set>  
 <ref bean="user1"/>  
 <ref bean="user2"/>  
 <ref bean="user1"/>  
 </set>  
 </property>  
  
 <!-- 注入java.util.Map对象 -->  
 <property name="map1">  
 <map>  
 <entry key="马士兵老师" value="30"/>  
 <entry key="路人" value="28"/>  
 </map>  
 </property>  
  
 <!-- 注入数组对象 -->  
 <property name="array1">  
 <array>  
 <value>10</value>  
 <value>9</value>  
 <value>8</value>  
 </array>  
 </property>  
  
 <!-- 注入java.util.Properties对象 -->  
 <property name="properties1">  
 <props>  
 <prop key="key1">java高并发系列</prop>  
 <prop key="key2">mybatis系列</prop>  
 <prop key="key3">mysql系列</prop>  
 </props>  
 </property>  
  
 </bean>  
</beans>

##### 新增测试用例

DiTest类中新增一个方法

/\*\*  
 \* 其他各种类型注入  
 \*/  
@Test  
public void diOtherType() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo5/diOtherType.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diOtherType"));  
}

##### 效果

运行测试用例输出：

DiOtherTypeModel{list1=[Spring, SpringBoot], set1=[UserModel{name='null', age=0, desc='null'}, UserModel{name='null', age=0, desc='null'}], map1={马士兵老师=30, 路人=28}, array1=[10, 9, 8], properties1={key3=mysql系列, key2=mybatis系列, key1=java高并发系列}}

### 总结

1. **本文主要讲解了xml中bean的依赖注入，都是采用硬编码的方式进行注入的，这种算是手动的方式**
2. **注入普通类型通过value属性或者value元素设置注入的值；注入对象如果是容器的其他bean的时候，需要使用ref属性或者ref元素或者内置bean元素的方式**
3. **还介绍了其他几种类型List、Set、Map、数组、Properties类型的注入，多看几遍加深理解**
4. **后面我们将介绍spring为我们提供的更牛逼的自动注入**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！

## Spring系列第8篇：依赖注入之自动注入（autowire）详解

Spring系列第8篇，在此也感谢各位一路的支持，请大家帮忙点个再看并转发到朋友圈让更多的朋友一起学习，感激不尽！

### 本文内容

* **手动注入的不足**
* **Class.isAssignableFrom方法介绍**
* **3种自动注入方式详解及案例**
  + **按名称自动注入**
  + **按类型自动注入**
  + **按构造器进行自动注入**
* **按类型自动注入某种类型的所有bean给List和Map（重点）**
* **autowire=default介绍**
* **总结**
* **案例源码**

### 手动注入的不足

上篇文章中介绍了依赖注入中的手动注入，所谓手动注入是指在xml中采用硬编码的方式来配置注入的对象，比如通过构造器注入或者set方法注入，这些注入的方式都存在不足，比如下面代码：

public class A{  
 private B b;  
 private C c;  
 private D d;  
 private E e;  
 ....  
 private N n;  
   
 //上面每个private属性的get和set方法  
}

使用spring容器来管理，xml配置如下：

<bean class="b" class="B"/>  
<bean class="c" class="C"/>  
<bean class="d" class="D"/>  
<bean class="e" class="E"/>  
<bean class="a" class="A">  
 <property name="b" ref="b"/>  
 <property name="c" ref="c"/>  
 <property name="d" ref="d"/>  
 <property name="e" ref="e"/>  
 ...  
</bean>

上面的注入存在的问题：

* **如果需要注入的对象比较多，比如A类中有几十个属性，那么上面的property属性是不是需要写几十个，此时配置文件代码量暴增**
* **如果A类中新增或者删除了一些依赖，还需要手动去调整bean xml中的依赖配置信息，否则会报错**
* **总的来说就是不利于维护和扩展**

**为了解决上面这些问题，spring为我们提供了更强大的功能：自动注入**

在介绍自动注入之前需要先介绍一些基础知识。

### Class.isAssignableFrom方法

#### 用法

isAssignableFrom是Class类中的一个方法，看一下这个方法的定义：

public native boolean isAssignableFrom(Class<?> cls)

用法如下：

c1.isAssignableFrom(c2)

用来判断c2和c1是否相等，或者c2是否是c1的子类。

#### 案例

@Test  
public void isAssignableFrom(){  
 System.out.println(Object.class.isAssignableFrom(Integer.class)); //true  
 System.out.println(Object.class.isAssignableFrom(int.class)); //false  
 System.out.println(Object.class.isAssignableFrom(List.class)); //true  
 System.out.println(Collection.class.isAssignableFrom(List.class)); //true  
 System.out.println(List.class.isAssignableFrom(Collection.class)); //false  
}

### 自动注入

自动注入是采用约定大约配置的方式来实现的，程序和spring容器之间约定好，遵守某一种都认同的规则，来实现自动注入。

xml中可以在bean元素中通过autowire属性来设置自动注入的方式：

<bean id="" class="" autowire="byType|byName|constructor|default" />

* **byteName：按照名称进行注入**
* **byType：按类型进行注入**
* **constructor：按照构造方法进行注入**
* **default：默认注入方式**

下面我们详解每种注入方式的用法。

### 按照名称进行注入（byName）

#### 用法

autowire设置为byName

<bean id="" class="X类" autowire="byName"/>

spring容器会按照set属性的名称去容器中查找同名的bean对象，然后将查找到的对象通过set方法注入到对应的bean中，未找到对应名称的bean对象则set方法不进行注入

需要注入的set属性的名称和被注入的bean的名称必须一致。

来看看案例吧。

#### 案例

##### DiByName.java

package com.javacode2018.lesson001.demo6;  
  
/\*\*  
 \* 按照名称自动注入  
 \*/  
public class DiAutowireByName {  
 public static class Service1 { //@1  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "Service1{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 public static class Service2 { //@1  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 private Service1 service1;//@3  
 private Service2 service2;//@4  
  
 public Service1 getService1() {  
 return service1;  
 }  
  
 public void setService1(Service1 service1) {  
 System.out.println("setService1->" + service1);  
 this.service1 = service1;  
 }  
  
 public Service2 getService2() {  
 return service2;  
 }  
  
 public void setService2(Service2 service2) {  
 System.out.println("setService2->" + service2);  
 this.service2 = service2;  
 }  
  
 @Override  
 public String toString() {  
 return "DiAutowireByName{" +  
 "service1=" + service1 +  
 ", service2=" + service2 +  
 '}';  
 }  
}

这个类中有2个属性，名称为：

* service1
* service2

这两个属性都有对应的set方法。

下面我们在bean xml中定义2个和这2个属性同名的bean，然后使用按照名称进行自动注入。

##### diAutowireByName.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="service1" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service1">  
 <property name="desc" value="service1"/>  
 </bean>  
 <bean id="service2" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service2">  
 <property name="desc" value="service2"/>  
 </bean>  
 <bean id="service2-1" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service2">  
 <property name="desc" value="service2-1"/>  
 </bean>  
  
 <!-- autowire：byName 配置按照name进行自动注入 -->  
 <bean id="diAutowireByName1" class="com.javacode2018.lesson001.demo6.DiAutowireByName" autowire="byName"/>  
  
 <!-- 当配置了自动注入，还可以使用手动的方式自动注入进行覆盖，手动的优先级更高一些 -->  
 <bean id="diAutowireByName2" class="com.javacode2018.lesson001.demo6.DiAutowireByName" autowire="byName">  
 <property name="service2" ref="service2-1"/>  
 </bean>  
  
</beans>

上面注释认真看一下。

@1：定义了一个名称为service1的bean

@2：定义了一个名称为service2的bean

@3：定义diAutowireByName需要将autowire的值置为byName，表示按名称进行自动注入。

spring容器创建diAutowireByName对应的bean时，会遍历DiAutowireByName类中的所有set方法，然后得到set对应的属性名称列表：{"service1","service2"}，然后遍历这属性列表，在容器中查找和属性同名的bean对象，然后调用属性对应的set方法，将bean对象注入进去

##### 测试用例

package com.javacode2018.lesson001.demo6;  
  
import com.javacode2018.lesson001.demo5.IocUtils;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.util.ArrayList;  
import java.util.Collection;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* xml中自动注入配置案例  
 \*/  
public class DiAutowireTest {  
  
 /\*\*  
 \* 按照名称进行注入  
 \*/  
 @Test  
 public void diAutowireByName() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo6/diAutowireByName.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diAutowireByName"));  
 }  
}

##### 效果

运行diAutowireByName输出：

setService1->Service1{desc='service1'}  
setService2->Service2{desc='service2'}  
setService2->Service2{desc='service2-1'}  
setService1->Service1{desc='service1'}  
DiAutowireByName{service1=Service1{desc='service1'}, service2=Service2{desc='service2'}}  
DiAutowireByName{service1=Service1{desc='service1'}, service2=Service2{desc='service2-1'}}

#### 优缺点

按名称进行注入的时候，要求名称和set属性的名称必须同名，相对于硬编码的方式注入，确实节省了不少代码。

### 按照类型进行自动注入

#### 用法

autowire设置为byType

<bean id="" class="X类" autowire="byType"/>

spring容器会遍历x类中所有的set方法，会在容器中查找和set参数类型相同的bean对象，将其通过set方法进行注入，未找到对应类型的bean对象则set方法不进行注入。

**需要注入的set属性的类型和被注入的bean的类型需要满足isAssignableFrom关系。**

按照类型自动装配的时候，如果按照类型找到了多个符合条件的bean，系统会报错。

**set方法的参数如果是下面的类型或者下面类型的数组的时候，这个set方法会被跳过注入：**

**Object,Boolean,boolean,Byte,byte,Character,char,Double,double,Float,float,Integer,int,Long,Short,shot,Enum,CharSequence,Number,Date,java.time.temporal.Temporal,java.net.URI,java.net.URI,java.util.Locale,java.lang.Class**

来看看案例吧。

#### 案例

##### DiByType.java

package com.javacode2018.lesson001.demo6;  
  
/\*\*  
 \* 按照类型自动注入  
 \*/  
public class DiAutowireByType {  
 public static class Service1 {  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "Service1{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 public static class Service2 {  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 private Service1 service1;  
 private Service2 service2;  
  
 public Service1 getService1() {  
 return service1;  
 }  
  
 public void setService1(Service1 service1) {  
 System.out.println("setService1->" + service1); //@1  
 this.service1 = service1;  
 }  
  
 public Service2 getService2() {  
 return service2;  
 }  
  
 public void setService2(Service2 service2) {  
 System.out.println("setService2->" + service2); //@2  
 this.service2 = service2;  
 }  
  
 @Override  
 public String toString() {  
 return "DiByType{" +  
 "service1=" + service1 +  
 ", service2=" + service2 +  
 '}';  
 }  
}

DiAutowireByType类中有2个set方法分别来注入Service1和Service2，两个set方法中都输出了一行文字，一会执行的时候可以通过这个输出可以看出set方法是否被调用了。

##### diAutowireByName.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean class="com.javacode2018.lesson001.demo6.DiAutowireByType$Service1">  
 <property name="desc" value="service1"/>  
 </bean>  
 <bean class="com.javacode2018.lesson001.demo6.DiAutowireByType$Service2">  
 <property name="desc" value="service2"/>  
 </bean>  
  
 <!-- autowire：byType 配置按照set参数类型进行自动注入 -->  
 <bean id="diAutowireByType1" class="com.javacode2018.lesson001.demo6.DiAutowireByType" autowire="byType"/>  
  
</beans>

上面定义了3个bean，前2个bean并没有指定名称

第3个bean的autowire为byType表示按类型进行注入，会按照set方法的参数类型进行注入。

##### 测试用例

DiAutowireTest类中添加一个方法

/\*\*  
 \* 按照set方法参数类型进行注入  
 \*/  
@Test  
public void diAutowireByType() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo6/diAutowireByType.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diAutowireByType1"));  
}

##### 效果

运行diAutowireByType输出：

setService1->Service1{desc='service1'}  
setService2->Service2{desc='service2'}  
DiByType{service1=Service1{desc='service1'}, service2=Service2{desc='service2'}}

#### 优缺点

相对于手动注入，节省了不少代码，新增或者删除属性，只需要增减对应的set方法就可以了，更容易扩展了。

### 注入类型匹配的所有bean(重点)

#### 按照类型注入还有2中比较牛逼的用法

* **一个容器中满足某种类型的bean可以有很多个，将容器中某种类型中的所有bean，通过set方法注入给一个java.util.List<需要注入的Bean的类型或者其父类型或者其接口>对象**
* **将容器中某种类型中的所有bean，通过set方法注入给一个java.util.Map<String,需要注入的Bean的类型或者其父类型或者其接口>对象**

来看个案例就懂了。

#### 案例

##### DiAutowireByTypeExtend.java

package com.javacode2018.lesson001.demo6;  
  
import java.util.List;  
import java.util.Map;  
  
/\*\*  
 \* 满足条件的bean注入到集合中(重点)  
 \*/  
public class DiAutowireByTypeExtend {  
  
 //定义了一个接口  
 public interface IService1 {  
 }  
  
 public static class BaseServie {  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "BaseServie{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 //Service1实现了IService1接口  
 public static class Service1 extends BaseServie implements IService1 {  
  
 }  
  
 //Service1实现了IService1接口  
 public static class Service2 extends BaseServie implements IService1 {  
 }  
  
 private List<IService1> serviceList;//@1  
 private List<BaseServie> baseServieList;//@2  
 private Map<String, IService1> service1Map;//@3  
 private Map<String, BaseServie> baseServieMap;//@4  
  
 public List<IService1> getServiceList() {  
 return serviceList;  
 }  
  
 public void setServiceList(List<IService1> serviceList) {//@5  
 this.serviceList = serviceList;  
 }  
  
 public List<BaseServie> getBaseServieList() {  
 return baseServieList;  
 }  
  
 public void setBaseServieList(List<BaseServie> baseServieList) {//@6  
 this.baseServieList = baseServieList;  
 }  
  
 public Map<String, IService1> getService1Map() {  
 return service1Map;  
 }  
  
 public void setService1Map(Map<String, IService1> service1Map) {//@7  
 this.service1Map = service1Map;  
 }  
  
 public Map<String, BaseServie> getBaseServieMap() {  
 return baseServieMap;  
 }  
  
 public void setBaseServieMap(Map<String, BaseServie> baseServieMap) {//@8  
 this.baseServieMap = baseServieMap;  
 }  
  
 @Override  
 public String toString() { //9  
 return "DiAutowireByTypeExtend{" +  
 "serviceList=" + serviceList +  
 ", baseServieList=" + baseServieList +  
 ", service1Map=" + service1Map +  
 ", baseServieMap=" + baseServieMap +  
 '}';  
 }  
}

@1,@2,@3,@4：定义了4个属性，都是泛型类型的，都有对应的set方法。

@5：参数类型是List<BaseServie>，这个集合集合中元素的类型是BaseServie，spring会找到容器中所有满足BaseServie.isAssignableFrom(bean的类型)的bean列表，将其通过@5的set方法进行注入。

@6：同@5的代码

@7：这个参数类型是一个map了，map的key是string类型，value是IService1类型，spring容器会将所有满足IService1类型的bean找到，按照name->bean对象这种方式丢到一个map中，然后调用@7的set方法进行注入，最后注入的这个map就是bean的名称和bean对象进行映射的一个map对象。

@8：同@7的代码

@9：重写了toString方法，输出的时候好看一些

##### 测试用例

DiAutowireTest新增一个方法：

/\*\*  
 \* 按照类型注入集合  
 \*/  
@Test  
public void diAutowireByTypeExtend() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo6/diAutowireByTypeExtend.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 //从容器中获取DiAutowireByTypeExtend  
 DiAutowireByTypeExtend diAutowireByTypeExtend = context.getBean(DiAutowireByTypeExtend.class);  
 //输出diAutowireByTypeExtend中的属性看一下  
 System.out.println("serviceList：" + diAutowireByTypeExtend.getServiceList());  
 System.out.println("baseServieList：" + diAutowireByTypeExtend.getBaseServieList());  
 System.out.println("service1Map：" + diAutowireByTypeExtend.getService1Map());  
 System.out.println("baseServieMap：" + diAutowireByTypeExtend.getBaseServieMap());  
}

##### 效果

运行diAutowireByTypeExtend输出：

serviceList：[BaseServie{desc='service1-1'}, BaseServie{desc='service1-2'}, BaseServie{desc='service2-1'}]  
baseServieList：[BaseServie{desc='service1-1'}, BaseServie{desc='service1-2'}, BaseServie{desc='service2-1'}]  
service1Map：{service1-1=BaseServie{desc='service1-1'}, service1-2=BaseServie{desc='service1-2'}, service2-1=BaseServie{desc='service2-1'}}  
baseServieMap：{service1-1=BaseServie{desc='service1-1'}, service1-2=BaseServie{desc='service1-2'}, service2-1=BaseServie{desc='service2-1'}}

下面我们来介绍另外一种自动注入方式。

### 按照构造函数进行自动注入

#### 用法

autowire设置为constructor

<bean id="" class="X类" autowire="constructor"/>

**spring会找到x类中所有的构造方法（一个类可能有多个构造方法），然后将这些构造方法进行排序（先按修饰符进行排序，public的在前面，其他的在后面，如果修饰符一样的，会按照构造函数参数数量倒叙，也就是采用贪婪的模式进行匹配，spring容器会尽量多注入一些需要的对象）得到一个构造函数列表，会轮询这个构造器列表，判断当前构造器所有参数是否在容器中都可以找到匹配的bean对象，如果可以找到就使用这个构造器进行注入，如果不能找到，那么就会跳过这个构造器，继续采用同样的方式匹配下一个构造器，直到找到一个合适的为止。**

来看看案例吧。

#### 案例

##### DiAutowireByConstructor.java

package com.javacode2018.lesson001.demo6;  
  
/\*\*  
 \* 构造函数的方式进行自动注入  
 \*/  
public class DiAutowireByConstructor {  
  
 public static class BaseServie {  
 private String desc;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "BaseServie{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
 }  
  
 //Service1实现了IService1接口  
 public static class Service1 extends BaseServie {  
 }  
  
 //Service1实现了IService1接口  
 public static class Service2 extends BaseServie {  
 }  
  
 private Service1 service1;  
 private Service2 service2;  
  
 public DiAutowireByConstructor() { //@0  
 }  
  
 public DiAutowireByConstructor(Service1 service1) { //@1  
 System.out.println("DiAutowireByConstructor(Service1 service1)");  
 this.service1 = service1;  
 }  
  
 public DiAutowireByConstructor(Service1 service1, Service2 service2) { //@2  
 System.out.println("DiAutowireByConstructor(Service1 service1, Service2 service2)");  
 this.service1 = service1;  
 this.service2 = service2;  
 }  
  
 public Service1 getService1() {  
 return service1;  
 }  
  
 public void setService1(Service1 service1) {  
 this.service1 = service1;  
 }  
  
 public Service2 getService2() {  
 return service2;  
 }  
  
 public void setService2(Service2 service2) {  
 this.service2 = service2;  
 }  
  
 @Override  
 public String toString() {  
 return "DiAutowireByConstructor{" +  
 "service1=" + service1 +  
 ", service2=" + service2 +  
 '}';  
 }  
}

@1：1个参数的构造函数

@2：2个参数的构造函数

2个有参构造函数第一行都打印了一段文字，一会在输出中可以看到代码是调用了那个构造函数创建对象。

##### diAutowireByConstructor.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean class="com.javacode2018.lesson001.demo6.DiAutowireByConstructor$Service1">  
 <property name="desc" value="service1"/>  
 </bean>  
  
 <bean id="diAutowireByConstructor" class="com.javacode2018.lesson001.demo6.DiAutowireByConstructor"  
 autowire="constructor"/>  
  
</beans>

##### 测试用例

DiAutowireTest新增一个方法

/\*\*  
 \* 构造函数的方式进行自动注入  
 \*/  
@Test  
public void diAutowireByConstructor() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo6/diAutowireByConstructor.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diAutowireByConstructor"));  
}

##### 效果

运行diAutowireByConstructor输出：

DiAutowireByConstructor(Service1 service1)  
DiAutowireByConstructor{service1=BaseServie{desc='service1'}, service2=null}

从输出中可以看到调用的是DiAutowireByConstructor类中的第一个构造函数注入了service1 bean。

构造函数匹配采用贪婪匹配，多个构造函数结合容器找到一个合适的构造函数，最匹配的就是第一个有参构造函数，而第二个有参构造函数的第二个参数在spring容器中找不到匹配的bean对象，所以被跳过了。

我们在diAutowireByConstructor.xml加入Service2的配置：

<bean class="com.javacode2018.lesson001.demo6.DiAutowireByConstructor$Service2">  
 <property name="desc" value="service2"/>  
</bean>

再来运行一下diAutowireByConstructor输出：

DiAutowireByConstructor(Service1 service1, Service2 service2)  
DiAutowireByConstructor{service1=BaseServie{desc='service1'}, service2=BaseServie{desc='service2'}}

此时可以看到第二个有参构造函数被调用了，满足了贪婪方式的注入原则，最大限度的注入所有依赖的对象。

### autowire=default

#### 用法

bean xml的根元素为beans，注意根元素有个default-autowire属性，这个属性可选值有(no|byName|byType|constructor|default)，这个属性可以批量设置当前文件中所有bean的自动注入的方式，bean元素中如果省略了autowire属性，那么会取default-autowire的值作为其autowire的值，而每个bean元素还可以单独设置自己的autowire覆盖default-autowire的配置，如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd"  
 default-autowire="byName">  
  
</beans>

#### 案例

##### diAutowireByDefault.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd"  
 default-autowire="byName"> //@1  
  
 <bean id="service1" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service1">  
 <property name="desc" value="service1"/>  
 </bean>  
 <bean id="service2" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service2">  
 <property name="desc" value="service2"/>  
 </bean>  
 <bean id="service2-1" class="com.javacode2018.lesson001.demo6.DiAutowireByName$Service2">  
 <property name="desc" value="service2-1"/>  
 </bean>  
  
 <!-- autowire：default，会采用beans中的default-autowire指定的配置 -->  
 <bean id="diAutowireByDefault1" class="com.javacode2018.lesson001.demo6.DiAutowireByName" autowire="default"/> //@2  
  
 <!-- autowire：default，会采用beans中的default-autowire指定的配置，还可以使用手动的方式自动注入进行覆盖，手动的优先级更高一些 -->  
 <bean id="diAutowireByDefault2" class="com.javacode2018.lesson001.demo6.DiAutowireByName" autowire="default"> //@3  
 <property name="service2" ref="service2-1"/>  
 </bean>  
  
</beans>

注意上面的@1配置的default-autowire="byName"，表示全局默认的自动注入方式是：按名称注入

@2和@3的autowire=default，那么注入方式会取default-autowire的值。

##### 测试用例

DiAutowireTest中新增一个方法

/\*\*  
 \* autowire=default  
 \*/  
@Test  
public void diAutowireByDefault() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo6/diAutowireByDefault.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean("diAutowireByDefault1"));  
 System.out.println(context.getBean("diAutowireByDefault2"));  
}

##### 效果

运行diAutowireByDefault输出

setService1->Service1{desc='service1'}  
setService2->Service2{desc='service2'}  
setService2->Service2{desc='service2-1'}  
setService1->Service1{desc='service1'}  
DiAutowireByName{service1=Service1{desc='service1'}, service2=Service2{desc='service2'}}  
DiAutowireByName{service1=Service1{desc='service1'}, service2=Service2{desc='service2-1'}}

### 总结

* **xml中手动注入存在的不足之处，可以通过自动注入的方式来解决，本文介绍了3中自动注入：通过名称自动注入、通过类型自动注入、通过构造器自动注入**
* **按类型注入中有个比较重要的是注入匹配类型所有的bean，可以将某种类型所有的bean注入给一个List对象，可以将某种类型的所有bean按照bean名称->bean对象的映射方式注入给一个Map对象，这种用法比较重要，用途比较大，要掌握**
* **spring中还有其他自动注入的方式，用起来会更爽，后面的文章中我们会详细介绍。**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入

## Spring系列第9篇：depend-on干什么的？

**本文主要讨论一下bean的创建和销毁的顺序，如何来干预bean的创建和销毁的顺序。**

### 无依赖bean创建和销毁的顺序

我们先来看一下没有任何依赖的bean的创建和销毁的顺序。

下面的xml中定义了3个bean：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="bean3" class="com.javacode2018.lesson001.demo7.NormalBean$Bean3"/>  
 <bean id="bean2" class="com.javacode2018.lesson001.demo7.NormalBean$Bean2"/>  
 <bean id="bean1" class="com.javacode2018.lesson001.demo7.NormalBean$Bean1"/>  
</beans>

注意上面xml中bean定义顺序是：bean3、bean2、bean1。

对应java代码如下：

package com.javacode2018.lesson001.demo7;  
  
import org.springframework.beans.factory.DisposableBean;  
  
/\*\*  
 \* 无任何依赖的bean创建的顺序  
 \*/  
public class NormalBean {  
 public static class Bean1 implements DisposableBean {  
  
 public Bean1() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean2 implements DisposableBean {  
  
 public Bean2() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean3 implements DisposableBean {  
  
 public Bean3() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
}

上面代码中使用到了DisposableBean接口，这个是spring容器提供的一个接口，这个接口中有个destroy方法，我们的bean类可以实现这个接口，当我们调用容器的close方法关闭容器的时候，spring会调用容器中所有bean的destory方法，用来做一些清理的工作，这个以后还会细讲的。

上面几个类中构造方法和destory方法中都有输出。

下面我们来搞个测试用例看一下spring容器启动和关闭的过程中，定义的3个bean的创建和销毁的顺序。

package com.javacode2018.lesson001.demo7;  
  
import com.javacode2018.lesson001.demo5.IocUtils;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* dependon详解  
 \*/  
public class DependOnTest {  
  
 /\*\*  
 \* 无依赖的bean创建和销毁的顺序  
 \*/  
 @Test  
 public void normalBean() {  
 System.out.println("容器启动中!");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo7/normalBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println("容器启动完毕，准备关闭spring容器!");  
 //关闭容器  
 context.close();  
 System.out.println("spring容器已关闭!");  
 }  
  
}

运行上面的normalBean方法，输出：

容器启动中!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean3 constructor!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean2 constructor!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean1 constructor!  
容器启动完毕，准备关闭spring容器!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean1 destroy()  
class com.javacode2018.lesson001.demo7.NormalBean$Bean2 destroy()  
class com.javacode2018.lesson001.demo7.NormalBean$Bean3 destroy()  
spring容器已关闭!

bean的定义结合上面输出我们来对比一下：

|  |  |  |
| --- | --- | --- |
| bean定义顺序 | 创建顺序 | 销毁顺序 |
| bean3 | bean3 | bean1 |
| bean2 | bean2 | bean2 |
| bean1 | bean1 | bean3 |

从输出中可以得到2点结论：

1. **bean对象的创建顺序和bean xml中定义的顺序一致**
2. **bean销毁的顺序和bean xml中定义的顺序相反**

### 通过构造器强依赖bean创建和销毁顺序

我们将上面案例改造一下，通过构造函数注入的方式使bean之间产生强依赖。

package com.javacode2018.lesson001.demo7;  
  
import org.springframework.beans.factory.DisposableBean;  
  
/\*\*  
 \* 强依赖的bean创建和销毁顺序  
 \*/  
public class StrongDependenceBean {  
 public static class Bean1 implements DisposableBean {  
  
 public Bean1() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean2 implements DisposableBean {  
  
 private Bean1 bean1;  
  
 public Bean2(Bean1 bean1) { //@1  
 this.bean1 = bean1;  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean3 implements DisposableBean {  
  
 private Bean2 bean2;  
  
 public Bean3(Bean2 bean2) { //@2  
 this.bean2 = bean2;  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
}

代码解释：

@1：创建Bean2的时候需要传入一个bean1对象，对bean1产生了强依赖

@2：创建Bean3的时候需要传入一个bean2对象，对bean2产生了强依赖

依赖关系是：

bean3->bean2->bean1

对应的配置(strongDependenceBean.xml)：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="bean3" class="com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean3">  
 <constructor-arg index="0" ref="bean2"/> //@1  
 </bean>  
 <bean id="bean2" class="com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean2">  
 <constructor-arg index="0" ref="bean1"/> //@2  
 </bean>  
 <bean id="bean1" class="com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean1">  
 </bean>  
</beans>

注意上面xml中bean定义顺序是：bean3、bean2、bean1。

@1：bean3中通过构造器注入bean2

@2：bean2中通过构造器注入bean1

DependOnTest中创建一个测试用例，如下：

/\*\*  
 \* 强依赖的bean的创建和销毁顺序测试  
 \*/  
@Test  
public void strongDependenceBean() {  
 System.out.println("容器启动中!");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo7/strongDependenceBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println("容器启动完毕，准备关闭spring容器!");  
 context.close();  
 System.out.println("spring容器已关闭!");  
}

运行strongDependenceBean方法输出：

容器启动中!  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean1 constructor!  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean2 constructor!  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean3 constructor!  
容器启动完毕，准备关闭spring容器!  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean3 destroy()  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean2 destroy()  
class com.javacode2018.lesson001.demo7.StrongDependenceBean$Bean1 destroy()  
spring容器已关闭!

bean的定义结合上面输出我们来对比一下：

|  |  |  |  |
| --- | --- | --- | --- |
| bean定义顺序 | 依赖顺序（下面依赖上面的） | 创建顺序 | 销毁顺序 |
| bean3 | bean1 | bean1 | bean3 |
| bean2 | bean2 | bean2 | bean2 |
| bean1 | bean3 | bean3 | bean1 |

从输出中可以得到2点结论：

1. **bean对象的创建顺序和bean依赖的顺序一致**
2. **bean销毁的顺序和bean创建的顺序相反**

### 通过depend-on干预bean创建和销毁顺序

上面看到了对于无依赖的bean，通过定义的顺序确实可以干预bean的创建顺序，通过强依赖也可以干预bean的创建顺序。

那么如果xml中定义的bean特别多，而有些bean之间也没有强依赖关系，此时如果想去调整bean的创建和销毁的顺序，得去调整xml中bean的定义顺序，或者去加强依赖，这样是非常不好的，spring中可以通过depend-on来解决这些问题，在不调整bean的定义顺序和强加依赖的情况下，可以通过通过depend-on属性来设置当前bean的依赖于哪些bean，那么可以保证depend-on指定的bean在当前bean之前先创建好，销毁的时候在当前bean之后进行销毁。

**depend-on使用方式：**

<bean id="bean1" class="" depend-on="bean2,bean3; bean4" />

**depend-on：设置当前bean依赖的bean名称，可以指定多个，多个之间可以用”,;空格“进行分割**

**上面不管bean2,bean2,bean4在任何地方定义，都可以确保在bean1创建之前，会先将bean2,bean3,bean4创建好，表示bean1依赖于这3个bean，可能bean1需要用到bean2、bean3、bean4中生成的一些资源或者其他的功能等，但是又没有强制去在bean1类中通过属性定义强依赖的方式去依赖于bean2、bean3、bean4；当然销毁的时候也会先销毁当前bean，再去销毁被依赖的bean，即先销毁bean1，再去销毁depend-on指定的bean。**

下面我们来个案例看一下：

dependOnBean.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="bean3" class="com.javacode2018.lesson001.demo7.NormalBean$Bean3" depends-on="bean2,bean1"/>  
 <bean id="bean2" class="com.javacode2018.lesson001.demo7.NormalBean$Bean2"/>  
 <bean id="bean1" class="com.javacode2018.lesson001.demo7.NormalBean$Bean1"/>  
</beans>

上面xml中先定义的bean3，然后定义了bean2和bean1，并且指定了bean3的depend-on=“bean2,bean1”，根据depend-on的规则，所以会先创建bean2和bean1,然后再创建bean3，销毁的时候，会按照和创建相反的顺序来，即：bean1、bean2、bean3，下面我们来看看效果是不是这样：

上面xml对应的java代码如下：

package com.javacode2018.lesson001.demo7;  
  
import org.springframework.beans.factory.DisposableBean;  
  
/\*\*  
 \* 通过depend-on来干预bean创建和销毁顺序  
 \*/  
public class DependOnBean {  
 public static class Bean1 implements DisposableBean {  
  
 public Bean1() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean2 implements DisposableBean {  
  
 public Bean2() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
  
 public static class Bean3 implements DisposableBean {  
  
 public Bean3() {  
 System.out.println(this.getClass() + " constructor!");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println(this.getClass() + " destroy()");  
 }  
 }  
}

DependOnTest中创建测试用例：

/\*\*  
 \* 通过depend-on来干预bean创建和销毁顺序  
 \*/  
@Test  
public void dependOnBean() {  
 System.out.println("容器启动中!");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo7/dependOnBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println("容器启动完毕，准备关闭spring容器!");  
 context.close();  
 System.out.println("spring容器已关闭!");  
}

运行dependOnBean方法输出：

容器启动中!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean2 constructor!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean1 constructor!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean3 constructor!  
容器启动完毕，准备关闭spring容器!  
class com.javacode2018.lesson001.demo7.NormalBean$Bean3 destroy()  
class com.javacode2018.lesson001.demo7.NormalBean$Bean1 destroy()  
class com.javacode2018.lesson001.demo7.NormalBean$Bean2 destroy()  
spring容器已关闭!

### 总结

1. **无依赖的bean创建顺序和定义的顺序一致，销毁顺序刚好相反**
2. **通过构造器强依赖的bean，会先创建构造器参数中对应的bean，然后才会创建当前bean，销毁顺序刚好相反**
3. **depend-on可以指定档期bean依赖的bean，通过这个可以确保depend-on指定的bean在当前bean创建之前先创建好，销毁顺序刚好相反**

### 问题？

通过setter方式注入依赖的bean的时候，bean的创建顺序和销毁的顺序是什么样的？大家可以研究一下，欢迎留言。

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持

## Spring系列第10篇：primary可以解决什么问题？

### 存在的问题以及解决方案

直接上案例，通过案例来看技术是如何使用的：

package com.javacode2018.lesson001.demo8;  
  
public class NormalBean {  
 public interface IService{} //@1  
 public static class ServiceA implements IService{} //@2  
 public static class ServiceB implements IService{} //@3  
}

上面代码很简单，@1：定义了一个接口IService，@2和@3创建了两个类都实现了IService接口。

下面我们通过spring来定义ServiceA和ServiceB两个bean，配置文件（normalBean.xml）如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo8.NormalBean$ServiceA"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo8.NormalBean$ServiceB"/>  
  
</beans>

来个测试用例来从spring容器中获取上面定义的bean对象，如下：

package com.javacode2018.lesson001.demo8;  
  
import com.javacode2018.lesson001.demo5.IocUtils;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* bean元素的primary属性可以解决什么问题？  
 \*/  
public class PrimaryTest {  
 @Test  
 public void normalBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo8/normalBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 //下面我们通过spring容器的T getBean(Class<T> requiredType)方法获取容器中对应的bean  
 NormalBean.IService service = context.getBean(NormalBean.IService.class); //@1  
 System.out.println(service);  
 }  
}

注意@1的代码，从spring容器中在容器中查找NormalBean.IService.class类型的bean对象，我们来运行一下看看效果，部分输出如下：

org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.javacode2018.lesson001.demo8.NormalBean$IService' available: expected single matching bean but found 2: serviceA,serviceB  
  
 at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBean(DefaultListableBeanFactory.java:1180)  
 at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:416)  
 at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:349)  
 at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)  
 at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1126)

发生异常了，错误中有一段提示比较重要，如下：

org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.javacode2018.lesson001.demo8.NormalBean$IService' available: expected single matching bean but found 2: serviceA,serviceB

这个详细说出了错误原因：spring容器中定义了2个bean，分别是serviceA和serviceB，这两个bean对象都实现了IService接口，而用例中我们想从容器中获取IService接口对应的bean，此时容器中有2个候选者（serviceA和serviceB）满足我们的需求，此时spring容器不知道如何选择，到底是返回serviceA呢还是返回serviceB呢？spring容器也懵逼了，所以报错了。

再来看一个通过setter方法注入的案例：

package com.javacode2018.lesson001.demo8;  
  
public class SetterBean {  
 public interface IService{} //@1  
 public static class ServiceA implements IService{} //@2  
 public static class ServiceB implements IService{} //@3  
  
 private IService service;  
  
 public void setService(IService service) {  
 this.service = service;  
 }  
}

下面我们通过xml来定义SetterBean，并且使用setter方式将IService注入到SetterBean中，配置如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo8.SetterBean$ServiceA"/>  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo8.SetterBean$ServiceA"/>  
  
 <bean id="setterBean" class="com.javacode2018.lesson001.demo8.SetterBean" autowire="byType" />  
</beans>

注意上面setterBean的定义，autowire="byType"采用了按照类型自动注入的方式，容器启动的时候，会自动取调用SetterBean的setService方法，在容器中查找和这个方法参数类型匹配的bean，将查找的bean通过setService方法注入进去。

来个测试用例，PrimaryTest中加个方法：

@Test  
public void setterBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo8/setterBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
}

运行输出：

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'setterBean' defined in class path resource [com/javacode2018/lesson001/demo8/setterBean.xml]: Unsatisfied dependency expressed through bean property 'service'; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.javacode2018.lesson001.demo8.SetterBean$IService' available: expected single matching bean but found 2: serviceA,serviceB  
  
 at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireByType(AbstractAutowireCapableBeanFactory.java:1526)  
 at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanFactory.java:1406)

错误中重点信息：

org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.javacode2018.lesson001.demo8.SetterBean$IService' available: expected single matching bean but found 2: serviceA,serviceB

容器中去找IService接口对应的bean，期望有一个匹配的，实际上却找到了2个匹配的，不知道如何选择，报错了。

上面2个案例报的异常都是下面这个异常：

org.springframework.beans.factory.NoUniqueBeanDefinitionException

**当希望从容器中获取到一个bean对象的时候，容器中却找到了多个匹配的bean，此时spring不知道如何选择了，处于懵逼状态，就会报这个异常。**

**spring中可以通过bean元素的primary属性来解决这个问题，可以通过这个属性来指定当前bean为主要候选者，当容器查询一个bean的时候，如果容器中有多个候选者匹配的时候，此时spring会返回主要的候选者。**

下面我们使用primary来解决上面案例的问题：

package com.javacode2018.lesson001.demo8;  
  
public class PrimaryBean {  
 public interface IService{} //@1  
 public static class ServiceA implements IService{} //@2  
 public static class ServiceB implements IService{} //@3  
  
 private IService service;  
  
 public void setService(IService service) {  
 this.service = service;  
 }  
  
 @Override  
 public String toString() {  
 return "PrimaryBean{" +  
 "service=" + service +  
 '}';  
 }  
}

spring配置文件如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo8.PrimaryBean$ServiceA" primary="true"/>  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo8.PrimaryBean$ServiceA"/>  
  
 <bean id="setterBean" class="com.javacode2018.lesson001.demo8.PrimaryBean" autowire="byType" />  
</beans>

上面配置中我们将serviceA的primary属性置为true了，将其置为主要候选者，容器中如果查找bean的时候，如果有多个匹配的，就以他为主。

我们来个测试用例：

@Test  
public void primaryBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo8/primaryBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 PrimaryBean.IService service = context.getBean(PrimaryBean.IService.class); //@1  
 System.out.println(service);  
 PrimaryBean primaryBean = context.getBean(PrimaryBean.class); //@2  
 System.out.println(primaryBean);  
}

@1：从容器中查找IService类型匹配的bean，这个接口有2个实现类（ServiceA和Service2），这类在容器中都定义了，但是serviceA为主要的bean，所以这行代码会返回serviceA

@2：从容器中查找PrimaryBean类型的bean，容器中有一个，这个bean按照byType默认注入IService接口匹配的bean，注入的时候如果候选者有多个，以primary="true"的bean为主来注入，所以此处会注入service2

我们来运行一下，看看和分析的是否一致，运行输出：

com.javacode2018.lesson001.demo8.PrimaryBean$ServiceA@7b227d8d  
PrimaryBean{service=com.javacode2018.lesson001.demo8.PrimaryBean$ServiceA@7b227d8d}

上面输出确实都是ServiceA，和我们分析的一致。

还有当候选者中如果有多个bean都将primary置为true，此时spring还是会懵逼的，也会报错，不知道如何选择了。

### 总结

**当从容器中查找一个bean的时候，如果容器中出现多个Bean候选者时，可以通过primary="true"将当前bean置为首选者，那么查找的时候就会返回主要的候选者，否则将抛出异常。**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### 

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？

## Spring系列第11篇：bean中的autowire-candidate属性又是干什么的？

### autowire-candidate到底是干什么的？

上一篇文章[Spring系列第10篇：primary可以解决什么问题？](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934003&idx=1&sn=961c1371af45ae702e2cbba615de60a3&chksm=88621e4dbf15975b03eb67556e9408f9c2ff16c3920a0a9ad2e1dfc50f68b498a61ca4f6d0b5&token=1156408467&lang=zh_CN" \l "rd)中遇到的问题我们再来回顾一下，当容器中某种类型的bean存在多个的时候，此时如果我们从容器中查找这种类型的bean的时候，会报下面这个异常：

org.springframework.beans.factory.NoUniqueBeanDefinitionException

**原因：当从容器中按照类型查找一个bean对象的时候，容器中却找到了多个匹配的bean，此时spring不知道如何选择了，处于懵逼状态，就会报这个异常。**

这种异常主要出现在2种场景中：

#### 场景1：

从容器容器中查找符合指定类型的bean，对应BeanFactory下面的方法：

<T> T getBean(Class<T> requiredType) throws BeansException;

#### 场景2：

自动注入方式设置为byType的时候，如下：

package com.javacode2018.lesson001.demo8;  
  
public class SetterBean {  
 public interface IService{} //@1  
 public static class ServiceA implements IService{} //@2  
 public static class ServiceB implements IService{} //@3  
  
 private IService service;  
  
 public void setService(IService service) {  
 this.service = service;  
 }  
}

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo8.SetterBean$ServiceA"/>  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo8.SetterBean$ServiceB"/>  
  
 <bean id="setterBean" class="com.javacode2018.lesson001.demo8.SetterBean" autowire="byType" />  
</beans>

setterBean的autowire设置的是byType，即按setter方法的参数类型自动注入，SetterBean的setService的类型是IService，而IService类有2个实现类：ServiceA和ServiceB，而容器容器中刚好有这2个实现类的bean：serviceA和serviceB，所以上面代码会报错，不知道注入的时候选择那个对象注入。

**我们可以通过primary属性来指定一个主要的bean，当从容器中查找的时候，如果有多个候选的bean符合查找的类型，此时容器将返回primary="true"的bean对象。**

spring还有一种方法也可以解决这个问题，可以设置某个bean是否在自动注入的时候是否为作为候选bean，通过bean元素的autowire-candidate属性类配置，如下：

<bean id="serviceA" class="com.javacode2018.lesson001.demo8.SetterBean$ServiceA" autowire-candidate="false"/>

autowire-candidate：设置当前bean在被其他对象作为自动注入对象的时候，是否作为候选bean，默认值是true。

来举例说明一下，以上面的setter注入的案例先来说一下注入的过程：

**容器在创建setterBean的时候，发现其autowire为byType，即按类型自动注入，此时会在SetterBean类中查找所有setter方法列表，其中就包含了setService方法，setService方法参数类型是IService，然后就会去容器中按照IService类型查找所有符合条件的bean列表，此时容器中会返回满足IService这种类型并且autowire-candidate="true"的bean，刚才有说过bean元素的autowire-candidate的默认值是true，所以容器中符合条件的候选bean有2个：serviceA和serviceB，setService方法只需要一个满足条件的bean，此时会再去看这个列表中是否只有一个主要的bean（即bean元素的primary=“ture”的bean），而bean元素的primary默认值都是false，所以没有primary为true的bean，此时spring容器懵了，不知道选哪个了，此时就报错了，抛出NoUniqueBeanDefinitionException异常**

从上面过程中可以看出将某个候选bean的primary置为true就可以解决问题了。

或者只保留一个bean的autowire-candidate为true，将其余的满足条件的bean的autowire-candidate置为false，此时也可以解决这个问题，下面我们使用autowire-candidate来解决上面问题看一下效果：

**SetterBean.java**

package com.javacode2018.lesson001.demo9;  
  
public class SetterBean {  
 public interface IService {} //@1  
  
 public static class ServiceA implements IService {} //@2  
  
 public static class ServiceB implements IService {} //@3  
  
 private IService service;  
  
 public void setService(IService service) {  
 this.service = service;  
 }  
  
 @Override  
 public String toString() {  
 return "SetterBean{" +  
 "service=" + service +  
 '}';  
 }  
}

**autowireCandidateBean.xml**

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo9.SetterBean$ServiceA" autowire-candidate="false"/>  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo9.SetterBean$ServiceB"/>  
  
 <bean id="setterBean" class="com.javacode2018.lesson001.demo9.SetterBean" autowire="byType" />  
</beans>

上面我们将serviceA的autowire-candidate置为false了，serviceA在被其他bean自动按照类型注入的时候，将不再放入候选名单中

**测试用例**

package com.javacode2018.lesson001.demo9;  
  
import com.javacode2018.lesson001.demo5.IocUtils;  
import com.javacode2018.lesson001.demo8.NormalBean;  
import com.javacode2018.lesson001.demo8.PrimaryBean;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.util.Arrays;  
import java.util.Map;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* bean元素的autowire-candidate可以设置当前bean是否作为其他bean自动注入的候选bean  
 \*/  
public class AutowireCandidateTest {  
  
 @Test  
 public void setterBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo9/autowireCandidateBean.xml";  
 ClassPathXmlApplicationContext context = IocUtils.context(beanXml);  
 System.out.println(context.getBean(SetterBean.class)); //@1  
  
 SetterBean.IService service = context.getBean(SetterBean.IService.class); //@2  
 System.out.println(service);  
 }  
  
}

@1：查找容器中SetterBean类型的bean对象

@2：查找容器中SetterBean.IService接口类型的bean，实际上面容器中serviceA和serviceB都是这种类型的

下面我们运行一下，看看输出：

SetterBean{service=com.javacode2018.lesson001.demo9.SetterBean$ServiceB@29176cc1}  
com.javacode2018.lesson001.demo9.SetterBean$ServiceB@29176cc1

注意一下输出，2行输出中都是ServiceB，因为serviceB的autowire-candidate是默认值true，自动注入的时候作为候选bean，而serviceA的autowire-candidate是false，自动注入的时候不作为候选bean，所以上面输出的都是serviceB。

### autowire-candidates属性解析源码

beans元素是xml中定义bean的根元素，beans元素有个default-autowire-candidates属性，用于定义哪些bean可以作为候选者，default-autowire-candidates的值是个通配符如：

default-autowire-candidates="\*Service"

再来说一下bean元素的autowire-candidate属性，这个属性有3个可选值：

* default：这个是默认值，autowire-candidate如果不设置，其值就是default
* true：作为候选者
* false：不作为候选者

spring中由beans元素的default-autowire-candidates和bean元素的autowire-candidate来决定最终bean元素autowire-candidate的值，我们来看一下bean元素autowire-candidates的解析源码：

源码位置：org.springframework.beans.factory.xml.BeanDefinitionParserDelegate#parseBeanDefinitionAttributes

主要代码如下：

//获取bean元素的autowire-candidate元素，autowire-candidate如果不设置，其值就是default  
String autowireCandidate = ele.getAttribute(AUTOWIRE\_CANDIDATE\_ATTRIBUTE);  
//判断bean元素的autowire-candidate元素是否等于"default"或者是否等于""  
if (isDefaultValue(autowireCandidate)) {   
 //获取beans元素default-autowire-candidates属性值  
 String candidatePattern = this.defaults.getAutowireCandidates();  
 //判断获取beans元素default-autowire-candidates属性值是否为空，default-autowire-candidates默认值就是null  
 if (candidatePattern != null) {  
 //判断bean的名称是否和default-autowire-candidates的值匹配，如果匹配就将bean的autowireCandidate置为true，否则置为false  
 String[] patterns = StringUtils.commaDelimitedListToStringArray(candidatePattern);  
 bd.setAutowireCandidate(PatternMatchUtils.simpleMatch(patterns, beanName));  
 }  
}else {  
 //判断bean的autowire-candidate的值是否等于"true"  
 bd.setAutowireCandidate(TRUE\_VALUE.equals(autowireCandidate));  
}

如果上面判断都没有进去，autowireCandidate属性默认值就是true，这个在下面定义的：

org.springframework.beans.factory.support.AbstractBeanDefinition#autowireCandidate  
  
private boolean autowireCandidate = true;

**所有的bean元素最后都会被解析为spring中的org.springframework.beans.factory.config.BeanDefinition对象，关于BeanDefinition以后我们会细说**。

### 问题

**对上面的案例做个扩展，SetterBean类中加个方法：**

public void setService1(List<IService> service) {//@0  
 System.out.println(service); //@1  
}

**@0：需要注入一个IService的集合，判断一下@1输出的是容器中的serviceA，还是serviceB，还是2个都有呢？为什么？**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？

## Spring系列第12篇：lazy-init：bean延迟初始化

### bean初始化的2种方式

1. 实时初始化
2. 延迟初始化

### bean实时初始化

在容器启动过程中被创建组装好的bean，称为实时初始化的bean，spring中默认定义的bean都是实时初始化的bean，这些bean默认都是单例的，在容器启动过程中会被创建好，然后放在spring容器中以供使用。

#### 实时初始化bean的有一些优点

1. 更早发下bean定义的错误：实时初始化的bean如果定义有问题，会在容器启动过程中会抛出异常，让开发者快速发现问题
2. 查找bean更快：容器启动完毕之后，实时初始化的bean已经完全创建好了，此时被缓存在spring容器中，当我们需要使用的时候，容器直接返回就可以了，速度是非常快的

#### 案例

##### ActualTimeBean类

package com.javacode2018.lesson001.demo11;  
  
/\*\*  
 \* 实时初始化的bean  
 \*/  
public class ActualTimeBean {  
 public ActualTimeBean() {  
 System.out.println("我是实时初始化的bean!");  
 }  
}

一会我们在spring中创建上面这个对象，构造函数中会输出一段话，这段话会在spring容器创建过程中输出。

##### actualTimeBean.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="actualTimeBean" class="com.javacode2018.lesson001.demo11.ActualTimeBean"/>  
  
</beans>

##### 测试用例

package com.javacode2018.lesson001.demo11;  
  
  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* bean默认是实时初始化的，可以通过bean元素的lazy-init="true"将bean定义为延迟初始化  
 \*/  
public class LazyBeanTest {  
  
 @Test  
 public void actualTimeBean() {  
 System.out.println("spring容器启动中...");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo11/actualTimeBean.xml";  
 new ClassPathXmlApplicationContext(beanXml); //启动spring容器  
 System.out.println("spring容器启动完毕...");  
 }  
}

注意上面代码，容器启动前后有输出，运行actualTimeBean输出：

spring容器启动中...  
我是实时初始化的bean!  
spring容器启动完毕...

可以看出actualTimeBean这个bean是在容器启动过程中被创建好的。

### 延迟初始化的bean

从上面我们可以看出，实时初始化的bean都会在容器启动过程中创建好，如果程序中定义的bean非常多，并且有些bean创建的过程中比较耗时的时候，会导致系统消耗的资源比较多，并且会让整个启动时间比较长，这个我估计大家都是有感受的，使用spring开发的系统比较大的时候，整个系统启动耗时是比较长的，基本上多数时间都是在创建和组装bean。

**spring对这些问题也提供了解决方案：bean延迟初始化。**

所谓延迟初始化，就是和实时初始化刚好相反，延迟初始化的bean在容器启动过程中不会创建，而是需要使用的时候才会去创建，先说一下bean什么时候会被使用：

1. 被其他bean作为依赖进行注入的时候，比如通过property元素的ref属性进行引用，通过构造器注入、通过set注入、通过自动注入，这些都会导致被依赖bean的创建
2. 开发者自己写代码向容器中查找bean的时候，如调用容器的getBean方法获取bean。

上面这2中情况会导致延迟初始化的bean被创建。

#### 延迟bean的配置

在bean定义的时候通过lazy-init属性来配置bean是否是延迟加载，true：延迟初始化，false：实时初始化

<bean lazy-init="是否是延迟初始化" />

我们来2个案例看一下效果。

#### 案例1

##### LazyInitBean类

package com.javacode2018.lesson001.demo11;  
  
public class LazyInitBean {  
 public LazyInitBean() {  
 System.out.println("我是延迟初始化的bean!");  
 }  
}

##### lazyInitBean.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="lazyInitBean" class="com.javacode2018.lesson001.demo11.LazyInitBean" lazy-init="true"/>  
  
</beans>

注意上面的lazy-init="true"表示定义的这个bean是延迟初始化的bean。

##### 测试用例

LazyBeanTest中加个方法

@Test  
public void lazyInitBean() {  
 System.out.println("spring容器启动中...");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo11/lazyInitBean.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml); //启动spring容器  
 System.out.println("spring容器启动完毕...");  
 System.out.println("从容器中开始查找LazyInitBean");  
 LazyInitBean lazyInitBean = context.getBean(LazyInitBean.class);  
 System.out.println("LazyInitBean:" + lazyInitBean);  
}

注意上面的输出，容器启动前后有输出，然后又从容器中查找LazyInitBean。

##### 运行输出

执行lazyInitBean方法，输出：

spring容器启动中...  
spring容器启动完毕...  
从容器中开始查找LazyInitBean  
我是延迟初始化的bean!  
LazyInitBean:com.javacode2018.lesson001.demo11.LazyInitBean@11dc3715

代码结合输出可以看出来，LazyInitBean在容器启动过程中并没有创建，当我们调用context.getBean方法的时候，LazyInitBean才被创建的。

#### 案例2

上面这种方式是我们主动从容器中获取bean的时候，延迟初始化的bean才被容器创建的，下面我们再来看一下当延迟初始化的bean被其他实时初始化的bean依赖的时候，是什么时候创建的。

##### ActualTimeDependencyLazyBean类

package com.javacode2018.lesson001.demo11;  
  
public class ActualTimeDependencyLazyBean {  
  
 public ActualTimeDependencyLazyBean() {  
 System.out.println("ActualTimeDependencyLazyBean实例化!");  
 }  
  
 private LazyInitBean lazyInitBean;  
  
 public LazyInitBean getLazyInitBean() {  
 return lazyInitBean;  
 }  
  
 public void setLazyInitBean(LazyInitBean lazyInitBean) {  
 this.lazyInitBean = lazyInitBean;  
 System.out.println("ActualTimeDependencyLazyBean.setLazyInitBean方法!");  
 }  
}

ActualTimeDependencyLazyBean类中有个lazyInitBean属性，对应的有get和set方法，我们将通过set方法将lazyInitBean对象注入。

##### actualTimeDependencyLazyBean.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="lazyInitBean" class="com.javacode2018.lesson001.demo11.LazyInitBean" lazy-init="true"/>  
  
 <bean id="actualTimeDependencyLazyBean" class="com.javacode2018.lesson001.demo11.ActualTimeDependencyLazyBean">  
 <property name="lazyInitBean" ref="lazyInitBean"/>  
 </bean>  
</beans>

注意上面定义了2个bean：

lazyInitBean：lazy-init为true，说明这个bean是延迟创建的

actualTimeDependencyLazyBean：通过property元素来注入lazyInitBean，actualTimeDependencyLazyBean中没有指定lazy-init，默认为false，表示是实时创建的bean，会在容器创建过程中被初始化

##### 测试用例

LazyBeanTest中加个方法，如下：

@Test  
public void actualTimeDependencyLazyBean() {  
 System.out.println("spring容器启动中...");  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo11/actualTimeDependencyLazyBean.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml); //启动spring容器  
 System.out.println("spring容器启动完毕...");  
}

##### 运行输出

spring容器启动中...  
ActualTimeDependencyLazyBean实例化!  
我是延迟初始化的bean!  
ActualTimeDependencyLazyBean.setLazyInitBean方法!  
spring容器启动完毕...

从容器中可以看到，xml中定义的2个bean都在容器启动过程中被创建好了。

有些朋友比较迷茫，lazyInitBean的lazy-init为true，怎么也在容器启动过程中被创建呢？

由于actualTimeDependencyLazyBean为实时初始化的bean，而这个bean在创建过程中需要用到lazyInitBean，此时容器会去查找lazyInitBean这个bean，然后会进行初始化，所以这2个bean都在容器启动过程中被创建的。

### 总结

**延迟初始化的bean无法在程序启动过程中迅速发现bean定义的问题，第一次获取的时候可能耗时会比较长。在实际工作中用的比较少，作为了解，以后碰到的时候会有个印象。**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？

## Spring系列第13篇：使用继承简化bean配置(abstract & parent)

### 先来看一个案例

#### ServiceA.java

package com.javacode2018.lesson001.demo12;  
  
public class ServiceA {  
}

#### ServiceB.java

package com.javacode2018.lesson001.demo12;  
  
public class ServiceB {  
 private String name;  
 private ServiceA serviceA;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public ServiceA getServiceA() {  
 return serviceA;  
 }  
  
 public void setServiceA(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
  
 @Override  
 public String toString() {  
 return "ServiceB{" +  
 "name='" + name + '\'' +  
 ", serviceA=" + serviceA +  
 '}';  
 }  
}

上面类中有2个属性，下面我们再创建一个ServiceC类，和ServiceB中的内容一样。

#### ServiceC.java

package com.javacode2018.lesson001.demo12;  
  
public class ServiceC {  
 private String name;  
 private ServiceA serviceA;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public ServiceA getServiceA() {  
 return serviceA;  
 }  
  
 public void setServiceA(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
  
 @Override  
 public String toString() {  
 return "ServiceC{" +  
 "name='" + name + '\'' +  
 ", serviceA=" + serviceA +  
 '}';  
 }  
}

下面我们使用spring来创建上面3个类对应的bean。

#### beanExtend.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo12.ServiceA"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo12.ServiceB">  
 <property name="name" value="马士兵老师"/>  
 <property name="serviceA" ref="serviceA"/>  
 </bean>  
   
 <bean id="serviceC" class="com.javacode2018.lesson001.demo12.ServiceB">  
 <property name="name" value="马士兵老师"/>  
 <property name="serviceA" ref="serviceA"/>  
 </bean>  
  
</beans>

创建测试用例。

#### BeanExtendTest.java

package com.javacode2018.lesson001.demo12;  
  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* bean定义继承案例  
 \*/  
public class BeanExtendTest {  
 @Test  
 public void normalBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo12/normalBean.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
 System.out.println("serviceB:" + context.getBean(ServiceB.class));  
 System.out.println("serviceC:" + context.getBean(ServiceC.class));  
 }  
}

#### 运行输出

serviceB:ServiceB{name='马士兵老师', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}  
serviceC:ServiceC{name='马士兵老师', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}

### 通过继承优化代码

我们再回头去看一下上面xml中，serviceB和serviceC两个bean的定义如下：

<bean id="serviceB" class="com.javacode2018.lesson001.demo12.ServiceB">  
 <property name="name" value="马士兵老师"/>  
 <property name="serviceA" ref="serviceA"/>  
</bean>  
  
<bean id="serviceC" class="com.javacode2018.lesson001.demo12.ServiceC">  
 <property name="name" value="马士兵老师"/>  
 <property name="serviceA" ref="serviceA"/>  
</bean>

这2个bean需要注入的属性的值是一样的，都需要注入name和serviceA两个属性，并且2个属性的值也是一样的，我们可以将上面的公共的代码抽取出来，通过spring中继承的方式来做到代码重用。

可以将上面xml调整一下，我们来新建一个extendBean.xml，内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo12.ServiceA"/>  
  
 <bean id="baseService" abstract="true">  
 <property name="name" value="马士兵老师"/>  
 <property name="serviceA" ref="serviceA"/>  
 </bean>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo12.ServiceB" parent="baseService"/>  
  
 <bean id="serviceC" class="com.javacode2018.lesson001.demo12.ServiceC" parent="baseService"/>  
  
</beans>

**上面多了一个baseService的bean，这个bean没有指定class对象，但是多了一个abstract="true"的属性，表示这个bean是抽象的，abstract为true的bean在spring容器中不会被创建，只是会将其当做bean定义的模板，而serviceB和serviceC的定义中多了一个属性parent，用来指定当前bean的父bean名称，此处是baseService，此时serviceB和serviceC会继承baseService中定义的配置信息。**

来个测试用例看一下效果:

@Test  
public void extendBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo12/extendBean.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
 System.out.println("serviceB:" + context.getBean(ServiceB.class));  
 System.out.println("serviceC:" + context.getBean(ServiceC.class));  
}

运行输出：

serviceB:ServiceB{name='马士兵老师', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}  
serviceC:ServiceC{name='马士兵老师', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}

输出和上面案例的输出基本一致。

但是这次bean xml中bean的定义简化了很多，将公共的bean配置提取出来了，通过parent属性来配置需要继承的bean。

**子bean中也可以重新定义父bean中已经定义好的配置，这样子配置会覆盖父bean中的配置信息**，我们将extendBean.xml中serviceC的定义改一下：

<bean id="serviceC" class="com.javacode2018.lesson001.demo12.ServiceC" parent="baseService">  
 <property name="name" value="欢迎和【马士兵老师】一起学些spring!"/>  
</bean>

运行extendBean输出：

serviceB:ServiceB{name='马士兵老师', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}  
serviceC:ServiceC{name='欢迎和【马士兵老师】一起学些spring!', serviceA=com.javacode2018.lesson001.demo12.ServiceA@222114ba}

从输出中可以看出serviceC中的name对父bean中name的值进行了覆盖。

我们再来从容器中获取一下baseService，如下：

System.out.println(context.getBean("baseService"));

运行会输出：

org.springframework.beans.factory.BeanIsAbstractException: Error creating bean with name 'baseService': Bean definition is abstract  
  
 at org.springframework.beans.factory.support.AbstractBeanFactory.checkMergedBeanDefinition(AbstractBeanFactory.java:1412)

会报BeanIsAbstractException异常，因为baseService是抽象的，不能够创建这个bean实例。

### 总结

1. **bean元素的abstract属性为true的时候可以定义某个bean为一个抽象的bean，相当于定义了一个bean模板，spring容器并不会创建这个bean，从容器中查找abstract为true的bean的时候，会报错BeanIsAbstractException异常**
2. **bean元素的parent属性可以指定当前bean的父bean，子bean可以继承父bean中配置信息，也可以自定义配置信息，这样可以覆盖父bean中的配置**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化

## Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？

### lookup-method：方法查找

**通常情况下，我们使用的bean都是单例的，如果一个bean需要依赖于另一个bean的时候，可以在当前bean中声明另外一个bean引用，然后注入依赖的bean，此时被依赖的bean在当前bean中自始至终都是同一个实例。**

#### 先来个案例回顾一下

package com.javacode2018.lesson001.demo13.normal;  
  
public class ServiceA {  
}

package com.javacode2018.lesson001.demo13.normal;  
  
public class ServiceB {  
  
 private ServiceA serviceA;  
  
 public ServiceA getServiceA() {  
 return serviceA;  
 }  
  
 public void setServiceA(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
}

上面2个类，ServiceA和ServiceB，而ServiceB中需要用到ServiceA，可以通过setServiceA将serviceA注入到ServiceB中，spring配置如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo13.normal.ServiceA" scope="prototype"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo13.normal.ServiceB">  
 <property name="serviceA" ref="serviceA"/>  
 </bean>  
  
</beans>

上面serviceA的scope是prototype，表示serviceA是多例的，每次从容器中获取serviceA都会返回一个新的对象。

而serviceB的scope没有配置，默认是单例的，通过property元素将serviceA注入。

来个测试案例，如下：

package com.javacode2018.lesson001.demo13;  
  
  
import com.javacode2018.lesson001.demo13.normal.ServiceA;  
import com.javacode2018.lesson001.demo13.normal.ServiceB;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* lookupMethod的使用  
 \*/  
public class LookupMethodTest {  
  
 @Test  
 public void normalBean() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo13/normalBean.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println(context.getBean(ServiceA.class)); //@1  
 System.out.println(context.getBean(ServiceA.class)); //@2  
  
 System.out.println("serviceB中的serviceA");  
 ServiceB serviceB = context.getBean(ServiceB.class); //@3  
 System.out.println(serviceB.getServiceA()); //@4  
 System.out.println(serviceB.getServiceA()); //@5  
 }  
}

@1和@2从容器中按照类型查找ServiceA对应的bean。

@3：从容器中获取ServiceB

@4和@5：获取serviceB中的serviceA对象

运行normalBean()看一下效果：

com.javacode2018.lesson001.demo13.normal.ServiceA@5bfa9431  
com.javacode2018.lesson001.demo13.normal.ServiceA@5db250b4  
serviceB中的serviceA  
com.javacode2018.lesson001.demo13.normal.ServiceA@223f3642  
com.javacode2018.lesson001.demo13.normal.ServiceA@223f3642

从输出中可以看出，@1和@2输出了不同的ServiceA，而@4和@5输出的是同一个serviceA，这是因为serviceB是单例的，serviceB中的serviceA会在容器创建serviceB的时候，从容器中获取一个serviceA将其注入到serviceB中，所以自始至终serviceB中的serviceA都是同一个对象。

如果我们希望beanB中每次使用beanA的时候beanA都是一个新的实例，我们怎么实现呢？

我们可以在serviceB中加个方法去获取serviceA，这个方法中我们主动去容器中获取serviceA，那么每次获取到的都是不同的serviceA实例。

那么问题来了，我们如何在serviceB中获取到spring容器呢？

spring中有个接口ApplicationContextAware：

org.springframework.context.ApplicationContextAware  
  
public interface ApplicationContextAware extends Aware {  
 void setApplicationContext(ApplicationContext applicationContext) throws BeansException;  
}

上面这个接口有一个方法setApplicationContext，这个接口给了自定义的bean中获取applicationContext的能力，当我们的类实现这个接口之后，spring容器创建bean对象的时候，如果bean实现了这个接口，那么容器会自动调用setApplicationContext方法，将容器对象applicationContext传入，此时在我们的bean对象中就可以使用容器的任何方法了。

下面我们就通过ApplicationContextAware接口来实现单例bean中使用多例bean的案例。

#### 单例bean中使用多例bean：ApplicationContext接口的方式

##### ServiceA.java

package com.javacode2018.lesson001.demo13.applicationcontextaware;  
  
public class ServiceA {  
}

##### ServiceB.java

package com.javacode2018.lesson001.demo13.applicationcontextaware;  
  
import org.springframework.beans.BeansException;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
  
public class ServiceB implements ApplicationContextAware { //@1  
  
 public void say(){  
 ServiceA serviceA = this.getServiceA();//@2  
 System.out.println("this:"+this+",serviceA:"+ serviceA);  
 }  
  
 public ServiceA getServiceA() {  
 return this.context.getBean(ServiceA.class);//@3  
 }  
  
 private ApplicationContext context;  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 this.context = applicationContext;  
 }  
}

注意上面代码，ServiceB实现了ApplicationContextAware接口，然后实现了这个接口中的setApplicationContext方法，spring容器在创建ServiceB的时候会自动调用setApplicationContext方法。

@3：从容器中主动去获取ServiceA，这样每次获取到的ServiceA都是一个新的实例。

@2：say方法中调用getServiceA方法获取ServiceA对象，然后将其输出。

##### alicationcontextaware.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA" scope="prototype"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceB"/>  
  
</beans>

上面定义了2个bean，第一个是多例的。

##### 测试用例

@Test  
public void alicationcontextaware() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo13/alicationcontextaware.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA.class)); //@1  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA.class)); //@2  
  
 System.out.println("serviceB中的serviceA");  
 com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceB serviceB = context.getBean(com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceB.class); //@3  
 serviceB.say();  
 serviceB.say();  
}

运行输出：

com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA@78047b92  
com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA@8909f18  
serviceB中的serviceA  
this:com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceB@79ca92b9,serviceA:com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA@1460a8c0  
this:com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceB@79ca92b9,serviceA:com.javacode2018.lesson001.demo13.applicationcontextaware.ServiceA@4f638935

最后2行是是serviceB中的say方法输出的，可以看出serviceB是一个对象，而serviceA是不同的对象。

#### 单例bean中使用多例bean：lookup-method方式实现

上面这种方式实现了单例bean中使用多例bean的需求，但是用到spring中的接口ApplicationContextAware，此时对spring的api有耦合的作用，我们一直推行高内聚低耦合，所以我们得寻求更好的办法。

能不能有这样的功能，当serviceB中调用getServiceA的时候，系统自动将这个方法拦截，然后去spring容器中查找对应的serviceA对象然后返回，spring中的lookup-method就可以实现这样的功能。

下面我们使用lookup-method来实现一下。

##### ServiceA.java

package com.javacode2018.lesson001.demo13.lookupmethod;  
  
public class ServiceA {  
}

##### ServiceB.java

package com.javacode2018.lesson001.demo13.lookupmethod;  
  
public class ServiceB {  
  
 public void say() {  
 ServiceA serviceA = this.getServiceA();  
 System.out.println("this:" + this + ",serviceA:" + serviceA);  
 }  
  
 public ServiceA getServiceA() { //@1  
 return null;  
 }  
  
}

**注意上面的@1，这个方法中返回了一个null对象，下面我们通过spring来创建上面2个bean对象，然后让spring对上面的getServiceA方法进行拦截，返回指定的bean，如下：**

##### lookupmethod.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo13.lookupmethod.ServiceA" scope="prototype"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo13.lookupmethod.ServiceB">  
 <lookup-method name="getServiceA" bean="serviceA"/>  
 </bean>  
  
</beans>

注意上面的配置，重点在于这行配置：

<lookup-method name="getServiceA" bean="serviceA"/>

当我们调用serviceB中的getServiceA方法的时候，这个方法会拦截，然后会按照lookup-method元素中bean属性的值作为bean的名称去容器中查找对应bean，然后作为getServiceA的返回值返回，即调用getServiceA方法的时候，会从spring容器中查找id为serviceA的bean然后返回。

##### 测试用例

LookupMethodTest中加个方法，如下：

@Test  
public void lookupmethod() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo13/lookupmethod.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo13.lookupmethod.ServiceA.class)); //@1  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo13.lookupmethod.ServiceA.class)); //@2  
  
 System.out.println("serviceB中的serviceA");  
 com.javacode2018.lesson001.demo13.lookupmethod.ServiceB serviceB = context.getBean(com.javacode2018.lesson001.demo13.lookupmethod.ServiceB.class); //@3  
 serviceB.say();  
 serviceB.say();  
}

运行看看效果：

com.javacode2018.lesson001.demo13.lookupmethod.ServiceA@619713e5  
com.javacode2018.lesson001.demo13.lookupmethod.ServiceA@708f5957  
serviceB中的serviceA  
this:com.javacode2018.lesson001.demo13.lookupmethod.ServiceB$$EnhancerBySpringCGLIB$$aca8be5a@68999068,serviceA:com.javacode2018.lesson001.demo13.lookupmethod.ServiceA@7722c3c3  
this:com.javacode2018.lesson001.demo13.lookupmethod.ServiceB$$EnhancerBySpringCGLIB$$aca8be5a@68999068,serviceA:com.javacode2018.lesson001.demo13.lookupmethod.ServiceA@2ef3eef9

注意最后2行的输出，serviceA是调用this.getServiceA()方法获取 ，源码中这个方法返回的是null，但是spring内部对这个方法进行了拦截，每次调用这个方法的时候，都会去容器中查找serviceA，然后返回，所以上面最后2行的输出中serviceA是有值的，并且是不同的serviceA实例。

**lookup-method：看其名字，就知道意思：方法查找，调用name属性指定的方法的时候，spring会对这个方法进行拦截，然后去容器中查找lookup-method元素中bean属性指定的bean，然后将找到的bean作为方法的返回值返回。**

**这个地方底层是使用cglib代理实现的，后面有篇文章会详细介绍代理的2种实现，到时候大家注意下，spring中很多牛逼的功能都是靠代理实现的。**

spring提供的还有一个功能，同样可以可以解决上面单例bean中用到多例bean的问题，也就是下面我们要说的replaced-method。

### replaced-method：方法替换

replaced-method：方法替换，比如我们要调用serviceB中的getServiceA的时候，我们可以对serviceB这个bean中的getServiceA方法进行拦截，把这个调用请求转发到一个替换者处理。这就是replaced-method可以实现的功能，比lookup-method更强大更灵活。

##### replaced-method的使用3个步骤

###### 步骤一：定义替换者

自定义一个替换者，替换者需要实现spring中的MethodReplacer接口，看一下这个接口的定义：

package org.springframework.beans.factory.support;  
  
import java.lang.reflect.Method;  
  
public interface MethodReplacer {  
  
 /\*\*  
 \* @param obj 被替换方法的目标对象  
 \* @param method 目标对象的方法  
 \* @param args 方法的参数  
 \* @return return value for the method  
 \*/  
 Object reimplement(Object obj, Method method, Object[] args) throws Throwable;  
  
}

当调用目标对象需要被替换的方法的时候，这个调用请求会被转发到上面的替换者的reimplement方法进行处理。

如：

package com.javacode2018.lesson001.demo14;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.support.MethodReplacer;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
  
import java.lang.reflect.Method;  
import java.util.Map;  
  
/\*\*  
 \* servieB的方法替换者  
 \*/  
public class ServiceBMethodReplacer implements MethodReplacer, ApplicationContextAware {  
  
 @Override  
 public Object reimplement(Object obj, Method method, Object[] args) throws Throwable {  
 return this.context.getBean(ServiceA.class);  
 }  
  
 private ApplicationContext context;  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 this.context = applicationContext;  
 }  
}

###### 步骤二：定义替换者bean

<!-- 定义替换者bean -->  
<bean id="serviceBMethodReplacer" class="com.javacode2018.lesson001.demo14.ServiceBMethodReplacer" />

###### 步骤二：通过replaced-method元素配置目标bean需要被替换的方法

<bean id="serviceB" class="com.javacode2018.lesson001.demo14.ServiceB">  
 <replaced-method name="getServiceA" replacer="serviceAMethodReplacer"/>  
</bean>

注意上面的replaced-method元素的2个属性：

name：用于指定当前bean需要被替换的方法

replacer：替换者，即实现了MethodReplacer接口的类对应的bean

上面配置中当调用serviceB的getServiceA的时候，会自动调用serviceAMethodReplacer这个bean中的reimplement方法进行处理。

##### 案例

###### ServiceA.java

package com.javacode2018.lesson001.demo14;  
  
public class ServiceA {  
}

###### ServiceB.java

package com.javacode2018.lesson001.demo14;  
  
public class ServiceB {  
  
 public void say() {  
 ServiceA serviceA = this.getServiceA();  
 System.out.println("this:" + this + ",serviceA:" + serviceA);  
 }  
  
 public ServiceA getServiceA() { //@1  
 return null;  
 }  
}

上面getServiceA需要返回一个ServiceA对象，此处返回的是null，下面我们通过spring对这个方法进行替换，然后从容器中获取ServiceA然后返回，下面我们来看看替换者的代码。

###### 替换者ServiceBMethodReplacer.java

这个替换者会替换ServiceB中的getServiceA方法

package com.javacode2018.lesson001.demo14;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.support.MethodReplacer;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
  
import java.lang.reflect.Method;  
import java.util.Map;  
  
/\*\*  
 \* servieB的方法替换者  
 \*/  
public class ServiceBMethodReplacer implements MethodReplacer, ApplicationContextAware {  
  
 @Override  
 public Object reimplement(Object obj, Method method, Object[] args) throws Throwable {  
 return this.context.getBean(ServiceA.class);  
 }  
  
 private ApplicationContext context;  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 this.context = applicationContext;  
 }  
}

###### spring中bean配置文件：replacedmethod.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
 <!-- 定义替换者bean -->  
 <bean id="serviceBMethodReplacer" class="com.javacode2018.lesson001.demo14.ServiceBMethodReplacer" />  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo14.ServiceA" scope="prototype"/>  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo14.ServiceB">  
 <replaced-method name="getServiceA" replacer="serviceBMethodReplacer"/>  
 </bean>  
  
</beans>

###### 测试用例ReplacedMethodTest

package com.javacode2018.lesson001.demo14;  
  
  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \* replaced-method：方法替换  
 \*/  
public class ReplacedMethodTest {  
  
 @Test  
 public void replacedmethod() {  
 String beanXml = "classpath:/com/javacode2018/lesson001/demo14/replacedmethod.xml";  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(beanXml);  
  
 System.out.println(context.getBean(ServiceA.class)); //@1  
 System.out.println(context.getBean(ServiceA.class)); //@2  
  
 System.out.println("serviceB中的serviceA");  
 ServiceB serviceB = context.getBean(ServiceB.class); //@3  
 serviceB.say();  
 serviceB.say();  
 }  
  
}

###### 运行输出

com.javacode2018.lesson001.demo14.ServiceA@7722c3c3  
com.javacode2018.lesson001.demo14.ServiceA@2ef3eef9  
serviceB中的serviceA  
this:com.javacode2018.lesson001.demo14.ServiceB$$EnhancerBySpringCGLIB$$21bb8912@243c4f91,serviceA:com.javacode2018.lesson001.demo14.ServiceA@291ae  
this:com.javacode2018.lesson001.demo14.ServiceB$$EnhancerBySpringCGLIB$$21bb8912@243c4f91,serviceA:com.javacode2018.lesson001.demo14.ServiceA@61df66b6

从输出中可以看出结果和lookup-method案例效果差不多，实现了单例bean中使用多例bean的案例。

输出中都有CGLIB这样的字样，说明这玩意也是通过cglib实现的。

### 总结

1. **lookup-method：方法查找，可以对指定的bean的方法进行拦截，然后从容器中查找指定的bean作为被拦截方法的返回值**
2. **replaced-method：方法替换，可以实现bean方法替换的效果，整体来说比lookup-method更灵活一些**
3. **单例bean中使用多例bean，本文中列出了3种方式，大家消化一下。**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)

## Spring系列第15篇：代理详解（java动态代理&CGLIB代理)

若有收获，点个在看、转发朋友圈，让更多朋友一起学习！

代理spring中用到的挺多的，比如上篇文章中的lookup-method和replaced-method，以及后面我们要学的aop、spring中的事务、spring中解析@configuration注解等等，这些都是依靠代理来实现的，所以我们先把代理拿出来讲讲，有利于我们理解其他内容。

### 本文内容

1. **为什么需要用代理**
2. **jdk动态代理玩法详解**
3. **cglib代理常见的各种玩法详解**

### 为什么要用代理

我们先来看一个案例。

有一个接口IService，如下：

package com.javacode2018.lesson001.demo15;  
  
public interface IService {  
 void m1();  
 void m2();  
 void m3();  
}

接口有2个实现类ServiceA和ServiceB，如下：

package com.javacode2018.lesson001.demo15;  
  
public class ServiceA implements IService {  
 @Override  
 public void m1() {  
 System.out.println("我是ServiceA中的m1方法!");  
 }  
  
 @Override  
 public void m2() {  
 System.out.println("我是ServiceA中的m2方法!");  
 }  
  
 @Override  
 public void m3() {  
 System.out.println("我是ServiceA中的m3方法!");  
 }  
}

package com.javacode2018.lesson001.demo15;  
  
public class ServiceB implements IService {  
 @Override  
 public void m1() {  
 System.out.println("我是ServiceB中的m1方法!");  
 }  
  
 @Override  
 public void m2() {  
 System.out.println("我是ServiceB中的m2方法!");  
 }  
  
 @Override  
 public void m3() {  
 System.out.println("我是ServiceB中的m3方法!");  
 }  
}

来个测试用例来调用上面类的方法，如下：

package com.javacode2018.lesson001.demo15;  
  
import org.junit.Test;  
  
public class ProxyTest {  
 @Test  
 public void m1() {  
 IService serviceA = new ServiceA();  
 IService serviceB = new ServiceB();  
 serviceA.m1();  
 serviceA.m2();  
 serviceA.m3();  
  
 serviceB.m1();  
 serviceB.m2();  
 serviceB.m3();  
 }  
}

上面的代码很简单，就不解释了，我们运行一下m1()方法，输出：

我是ServiceA中的m1方法!  
我是ServiceA中的m2方法!  
我是ServiceA中的m3方法!  
我是ServiceA中的m1方法!  
我是ServiceA中的m2方法!  
我是ServiceA中的m3方法!

**上面是我们原本的程序，突然领导有个需求：调用IService接口中的任何方法的时候，需要记录方法的耗时。**

**此时你会怎么做呢？**

**IService接口有2个实现类ServiceA和ServiceB，我们可以在这两个类的所有方法中加上统计耗时的代码，如果IService接口有几十个实现，是不是要修改很多代码，所有被修改的方法需重新测试？是不是非常痛苦，不过上面这种修改代码的方式倒是可以解决问题，只是增加了很多工作量（编码 & 测试）。**

**突然有一天，领导又说，要将这些耗时统计发送到监控系统用来做监控报警使用。**

**此时是不是又要去一个修改上面的代码？又要去测试？此时的系统是难以维护。**

**还有假如上面这些类都是第三方以jar包的方式提供给我们的，此时这些类都是class文件，此时我们无法去修改源码。**

**比较好的方式**：可以为IService接口创建一个代理类，通过这个代理类来间接访问IService接口的实现类，在这个代理类中去做耗时及发送至监控的代码，代码如下：

package com.javacode2018.lesson001.demo15;  
  
// IService的代理类  
public class ServiceProxy implements IService {  
 //目标对象，被代理的对象  
 private IService target;  
  
 public ServiceProxy(IService target) {  
 this.target = target;  
 }  
  
 @Override  
 public void m1() {  
 long starTime = System.nanoTime();  
 this.target.m1();  
 long endTime = System.nanoTime();  
 System.out.println(this.target.getClass() + ".m1()方法耗时(纳秒):" + (endTime - starTime));  
 }  
  
 @Override  
 public void m2() {  
 long starTime = System.nanoTime();  
 this.target.m1();  
 long endTime = System.nanoTime();  
 System.out.println(this.target.getClass() + ".m1()方法耗时(纳秒):" + (endTime - starTime));  
 }  
  
 @Override  
 public void m3() {  
 long starTime = System.nanoTime();  
 this.target.m1();  
 long endTime = System.nanoTime();  
 System.out.println(this.target.getClass() + ".m1()方法耗时(纳秒):" + (endTime - starTime));  
 }  
}

ServiceProxy是IService接口的代理类，target为被代理的对象，即实际需要访问的对象，也实现了IService接口，上面的3个方法中加了统计耗时的代码，当我们需要访问IService的其他实现类的时候，可以通过ServiceProxy来间接的进行访问，用法如下：

@Test  
public void serviceProxy() {  
 IService serviceA = new ServiceProxy(new ServiceA());//@1  
 IService serviceB = new ServiceProxy(new ServiceB()); //@2  
 serviceA.m1();  
 serviceA.m2();  
 serviceA.m3();  
  
 serviceB.m1();  
 serviceB.m2();  
 serviceB.m3();  
}

上面代码重点在于@1和@2，创建的是代理对象ServiceProxy，ServiceProxy构造方法中传入了被代理访问的对象，现在我们访问ServiceA或者ServiceB，都需要经过ServiceProxy，运行输出：

我是ServiceA中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceA.m1()方法耗时(纳秒):90100  
我是ServiceA中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceA.m1()方法耗时(纳秒):31600  
我是ServiceA中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceA.m1()方法耗时(纳秒):25800  
我是ServiceB中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceB.m1()方法耗时(纳秒):142100  
我是ServiceB中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceB.m1()方法耗时(纳秒):35000  
我是ServiceB中的m1方法!  
class com.javacode2018.lesson001.demo15.ServiceB.m1()方法耗时(纳秒):32900

上面实现中我们没有去修改ServiceA和ServiceB中的方法，只是给IService接口创建了一个代理类，通过代理类去访问目标对象，需要添加的一些共有的功能都放在代理中，当领导有其他需求的时候，我们只需修改ServiceProxy的代码，方便系统的扩展和测试。

假如现在我们需要给系统中所有接口都加上统计耗时的功能，若按照上面的方式，我们需要给每个接口创建一个代理类，此时代码量和测试的工作量也是巨大的，那么我们能不能写一个通用的代理类，来满足上面的功能呢？

**通用代理的2种实现：**

1. **jdk动态代理**
2. **cglib代理**

### jdk动态代理详解

jdk中为实现代理提供了支持，主要用到2个类：

java.lang.reflect.Proxy  
java.lang.reflect.InvocationHandler

**jdk自带的代理使用上面有个限制，只能为接口创建代理类，如果需要给具体的类创建代理类，需要用后面要说的cglib**

#### java.lang.reflect.Proxy

这是jdk动态代理中主要的一个类，里面有一些静态方法会经常用到，我们来熟悉一下：

##### getProxyClass方法

为指定的接口创建代理类，返回代理类的Class对象

public static Class<?> getProxyClass(ClassLoader loader,  
 Class<?>... interfaces)

参数说明：

loader：定义代理类的类加载器

interfaces：指定需要实现的接口列表，创建的代理默认会按顺序实现interfaces指定的接口

##### newProxyInstance方法

创建代理类的实例对象

public static Object newProxyInstance(ClassLoader loader,  
 Class<?>[] interfaces,  
 InvocationHandler h)

这个方法先为指定的接口创建代理类，然后会生成代理类的一个实例，最后一个参数比较特殊，是InvocationHandler类型的，这个是个借口如下：

public Object invoke(Object proxy, Method method, Object[] args)  
 throws Throwable;

上面方法会返回一个代理对象，当调用代理对象的任何方法的时候，会就被InvocationHandler接口的invoke方法处理，所以主要代码需要卸载invoke方法中，稍后会有案例细说。

##### isProxy方法

判断指定的类是否是一个代理类

public static boolean isProxyClass(Class<?> cl)

##### getInvocationHandler方法

获取代理对象的InvocationHandler对象

public static InvocationHandler getInvocationHandler(Object proxy)  
 throws IllegalArgumentException

**上面几个方法大家熟悉一下，下面我们来看创建代理具体的2种方式。**

#### 创建代理：方式一

##### 步骤

1.调用Proxy.getProxyClass方法获取代理类的Class对象  
2.使用InvocationHandler接口创建代理类的处理器  
3.通过代理类和InvocationHandler创建代理对象  
4.上面已经创建好代理对象了，接着我们就可以使用代理对象了

##### 案例

###### 先来个接口IService

package com.javacode2018.lesson001.demo16;  
  
public interface IService {  
 void m1();  
 void m2();  
 void m3();  
}

###### 创建IService接口的代理对象

@Test  
public void m1() throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {  
 // 1. 获取接口对应的代理类  
 Class<IService> proxyClass = (Class<IService>) Proxy.getProxyClass(IService.class.getClassLoader(), IService.class);  
 // 2. 创建代理类的处理器  
 InvocationHandler invocationHandler = new InvocationHandler() {  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.out.println("我是InvocationHandler，被调用的方法是：" + method.getName());  
 return null;  
 }  
 };  
 // 3. 创建代理实例  
 IService proxyService = proxyClass.getConstructor(InvocationHandler.class).newInstance(invocationHandler);  
 // 4. 调用代理的方法  
 proxyService.m1();  
 proxyService.m2();  
 proxyService.m3();  
}

###### 运行输出

我是InvocationHandler，被调用的方法是：m1  
我是InvocationHandler，被调用的方法是：m2  
我是InvocationHandler，被调用的方法是：m3

#### 创建代理：方式二

创建代理对象有更简单的方式。

##### 步骤

1.使用InvocationHandler接口创建代理类的处理器  
2.使用Proxy类的静态方法newProxyInstance直接创建代理对象  
3.使用代理对象

##### 案例

@Test  
public void m2() throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {  
 // 1. 创建代理类的处理器  
 InvocationHandler invocationHandler = new InvocationHandler() {  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.out.println("我是InvocationHandler，被调用的方法是：" + method.getName());  
 return null;  
 }  
 };  
 // 2. 创建代理实例  
 IService proxyService = (IService) Proxy.newProxyInstance(IService.class.getClassLoader(), new Class[]{IService.class}, invocationHandler);  
 // 3. 调用代理的方法  
 proxyService.m1();  
 proxyService.m2();  
 proxyService.m3();  
}

运行输出：

我是InvocationHandler，被调用的方法是：m1  
我是InvocationHandler，被调用的方法是：m2  
我是InvocationHandler，被调用的方法是：m3

#### 案例：任意接口中的方法耗时统计

**下面我们通过jdk动态代理实现一个通用的代理，解决统计所有接口方法耗时的问题。**

主要的代码在代理处理器InvocationHandler实现上面，如下：

package com.javacode2018.lesson001.demo16;  
  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
  
public class CostTimeInvocationHandler implements InvocationHandler {  
  
 private Object target;  
  
 public CostTimeInvocationHandler(Object target) {  
 this.target = target;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 long starTime = System.nanoTime();  
 Object result = method.invoke(this.target, args);//@1  
 long endTime = System.nanoTime();  
 System.out.println(this.target.getClass() + ".m1()方法耗时(纳秒):" + (endTime - starTime));  
 return result;  
 }  
  
 /\*\*  
 \* 用来创建targetInterface接口的代理对象  
 \*  
 \* @param target 需要被代理的对象  
 \* @param targetInterface 被代理的接口  
 \* @param <T>  
 \* @return  
 \*/  
 public static <T> T createProxy(Object target, Class<T> targetInterface) {  
 if (!targetInterface.isInterface()) {  
 throw new IllegalStateException("targetInterface必须是接口类型!");  
 } else if (!targetInterface.isAssignableFrom(target.getClass())) {  
 throw new IllegalStateException("target必须是targetInterface接口的实现类!");  
 }  
 return (T) Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), new CostTimeInvocationHandler(target));  
 }  
}

上面主要是createProxy方法用来创建代理对象，2个参数：

target：目标对象，需要实现targetInterface接口

targetInterface：需要创建代理的接口

invoke方法中通过method.invoke(this.target, args)调用目标方法，然后统计方法的耗时。

**测试用例**

@Test  
public void costTimeProxy() {  
 IService serviceA = CostTimeInvocationHandler.createProxy(new ServiceA(), IService.class);  
 IService serviceB = CostTimeInvocationHandler.createProxy(new ServiceB(), IService.class);  
 serviceA.m1();  
 serviceA.m2();  
 serviceA.m3();  
  
 serviceB.m1();  
 serviceB.m2();  
 serviceB.m3();  
}

**运行输出**

我是ServiceA中的m1方法!  
class com.javacode2018.lesson001.demo16.ServiceA.m1()方法耗时(纳秒):61300  
我是ServiceA中的m2方法!  
class com.javacode2018.lesson001.demo16.ServiceA.m1()方法耗时(纳秒):22300  
我是ServiceA中的m3方法!  
class com.javacode2018.lesson001.demo16.ServiceA.m1()方法耗时(纳秒):18700  
我是ServiceB中的m1方法!  
class com.javacode2018.lesson001.demo16.ServiceB.m1()方法耗时(纳秒):54700  
我是ServiceB中的m2方法!  
class com.javacode2018.lesson001.demo16.ServiceB.m1()方法耗时(纳秒):27200  
我是ServiceB中的m3方法!  
class com.javacode2018.lesson001.demo16.ServiceB.m1()方法耗时(纳秒):19800

**我们再来的接口，也需要统计耗时的功能，此时我们无需去创建新的代理类即可实现同样的功能，如下：**

**IUserService接口**

package com.javacode2018.lesson001.demo16;  
  
public interface IUserService {  
 /\*\*  
 \* 插入用户信息  
 \* @param name  
 \*/  
 void insert(String name);  
}

**IUserService接口实现类：**

package com.javacode2018.lesson001.demo16;  
  
public class UserService implements IUserService {  
 @Override  
 public void insert(String name) {  
 System.out.println(String.format("用户[name:%s]插入成功!", name));  
 }  
}

**测试用例**

@Test  
public void userService() {  
 IUserService userService = CostTimeInvocationHandler.createProxy(new UserService(), IUserService.class);  
 userService.insert("马士兵老师");  
}

**运行输出：**

用户[name:马士兵老师]插入成功!  
class com.javacode2018.lesson001.demo16.UserService.m1()方法耗时(纳秒):193000

**上面当我们创建一个新的接口的时候，不需要再去新建一个代理类了，只需要使用CostTimeInvocationHandler.createProxy创建一个新的代理对象就可以了，方便了很多。**

#### Proxy使用注意

1. **jdk中的Proxy只能为接口生成代理类，如果你想给某个类创建代理类，那么Proxy是无能为力的，此时需要我们用到下面要说的cglib了。**
2. **Proxy类中提供的几个常用的静态方法大家需要掌握**
3. **通过Proxy创建代理对象，当调用代理对象任意方法时候，会被InvocationHandler接口中的invoke方法进行处理，这个接口内容是关键**

### cglib代理详解

#### 什么是cglib

jdk动态代理只能为接口创建代理，使用上有局限性。实际的场景中我们的类不一定有接口，此时如果我们想为普通的类也实现代理功能，我们就需要用到cglib来实现了。

cglib是一个强大、高性能的字节码生成库，它用于在运行时扩展Java类和实现接口；本质上它是通过动态的生成一个子类去覆盖所要代理的类（非final修饰的类和方法）。Enhancer可能是CGLIB中最常用的一个类，和jdk中的Proxy不同的是，Enhancer既能够代理普通的class，也能够代理接口。Enhancer创建一个被代理对象的子类并且拦截所有的方法调用（包括从Object中继承的toString和hashCode方法）。Enhancer不能够拦截final方法，例如Object.getClass()方法，这是由于Java final方法语义决定的。基于同样的道理，Enhancer也不能对final类进行代理操作。

CGLIB作为一个开源项目，其代码托管在github，地址为：

https://github.com/cglib/cglib

#### cglib组成结构

/img/1.png)

CGLIB底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。除了CGLIB库外，脚本语言（如Groovy和BeanShell）也使用ASM生成字节码。ASM使用类似SAX的解析器来实现高性能。我们不鼓励直接使用ASM，因为它需要对Java字节码的格式足够的了解。

spring已将第三方cglib jar包中所有的类集成到spring自己的jar包中，本系列内容都是和spring相关的，为了方便，我们直接使用spring内部已集成的来讲解

#### 5个案例来演示cglib常见的用法

##### 案例1：拦截所有方法（MethodInterceptor）

创建一个具体的类，如下：

package com.javacode2018.lesson001.demo17;  
  
public class Service1 {  
 public void m1() {  
 System.out.println("我是m1方法");  
 }  
  
 public void m2() {  
 System.out.println("我是m2方法");  
 }  
}

下面我们为这个类创建一个代理，代理中实现打印每个方法的调用日志。

package com.javacode2018.lesson001.demo17;  
  
import org.junit.Test;  
import org.springframework.cglib.proxy.Enhancer;  
import org.springframework.cglib.proxy.MethodInterceptor;  
import org.springframework.cglib.proxy.MethodProxy;  
  
import java.lang.reflect.Method;  
  
public class CglibTest {  
  
 @Test  
 public void test1() {  
 //使用Enhancer来给某个类创建代理类，步骤  
 //1.创建Enhancer对象  
 Enhancer enhancer = new Enhancer();  
 //2.通过setSuperclass来设置父类型，即需要给哪个类创建代理类  
 enhancer.setSuperclass(Service1.class);  
 /\*3.设置回调，需实现org.springframework.cglib.proxy.Callback接口，  
 此处我们使用的是org.springframework.cglib.proxy.MethodInterceptor，也是一个接口，实现了Callback接口，  
 当调用代理对象的任何方法的时候，都会被MethodInterceptor接口的invoke方法处理\*/  
 enhancer.setCallback(new MethodInterceptor() {  
 /\*\*  
 \* 代理对象方法拦截器  
 \* @param o 代理对象  
 \* @param method 被代理的类的方法，即Service1中的方法  
 \* @param objects 调用方法传递的参数  
 \* @param methodProxy 方法代理对象  
 \* @return  
 \* @throws Throwable  
 \*/  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 System.out.println("调用方法:" + method);  
 //可以调用MethodProxy的invokeSuper调用被代理类的方法  
 Object result = methodProxy.invokeSuper(o, objects);  
 return result;  
 }  
 });  
 //4.获取代理对象,调用enhancer.create方法获取代理对象，这个方法返回的是Object类型的，所以需要强转一下  
 Service1 proxy = (Service1) enhancer.create();  
 //5.调用代理对象的方法  
 proxy.m1();  
 proxy.m2();  
 }  
}

上面代码中的注释很详细，列出了给指定的类创建代理的具体步骤，整个过程中主要用到了Enhancer类和MethodInterceptor接口。

enhancer.setSuperclass用来设置代理类的父类，即需要给哪个类创建代理类，此处是Service1

enhancer.setCallback传递的是MethodInterceptor接口类型的参数，MethodInterceptor接口有个intercept方法，这个方法会拦截代理对象所有的方法调用。

还有一个重点是Object result = methodProxy.invokeSuper(o, objects);可以调用被代理类，也就是Service1类中的具体的方法，从方法名称的意思可以看出是调用父类，实际对某个类创建代理，cglib底层通过修改字节码的方式为Service1类创建了一个子类。

运行输出：

调用方法:public void com.javacode2018.lesson001.demo17.Service1.m1()  
我是m1方法  
调用方法:public void com.javacode2018.lesson001.demo17.Service1.m2()  
我是m2方法

从输出中可以看出Service1中的2个方法都被MethodInterceptor中的invoke拦截处理了。

##### 案例2：拦截所有方法（MethodInterceptor）

在创建一个类，如下：

package com.javacode2018.lesson001.demo17;  
  
  
public class Service2 {  
 public void m1() {  
 System.out.println("我是m1方法");  
 this.m2(); //@1  
 }  
  
 public void m2() {  
 System.out.println("我是m2方法");  
 }  
}

这个类和上面的Service1类似，有点不同是@1，在m1方法中调用了m2方法。

下面来采用案例1中同样的方式来给Service2创建代理，如下：

@Test  
public void test2() {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(Service2.class);  
 enhancer.setCallback(new MethodInterceptor() {  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 System.out.println("调用方法:" + method);  
 Object result = methodProxy.invokeSuper(o, objects);  
 return result;  
 }  
 });  
 Service2 proxy = (Service2) enhancer.create();  
 proxy.m1(); //@1  
}

注意上面@1的代码，只调用了m1方法，看一下输出效果：

调用方法:public void com.javacode2018.lesson001.demo17.Service2.m1()  
我是m1方法  
调用方法:public void com.javacode2018.lesson001.demo17.Service2.m2()  
我是m2方法

从输出中可以看出m1和m2方法都被拦截器处理了，而m2方法是在Service1的m1方法中调用的，也被拦截处理了。

spring中的@configuration注解就是采用这种方式实现的，给大家上个@configuration案例眼熟一下：

package com.javacode2018.lesson001.demo17;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Config {  
  
 @Bean  
 public C1 c1(){  
 return new C1();  
 }  
 @Bean  
 public C2 c2(){  
 C1 c1 = this.c1(); //@1  
 return new C2(c1);  
 }  
 @Bean  
 public C3 c3(){  
 C1 c1 = this.c1(); //@2  
 return new C3(c1);  
 }  
  
 public static class C1{}  
  
 public static class C2{  
 private C1 c1;  
  
 public C2(C1 c1) {  
 this.c1 = c1;  
 }  
 }  
 public static class C3{  
 private C1 c1;  
  
 public C3(C1 c1) {  
 this.c1 = c1;  
 }  
 }  
  
}

上面代码中C2和C3依赖于C1，都是通过构造器注入C1，注意代码中的@1和@2都是调用c1方法获取容器中的C1，如何确保多次获取的C1都是一个的？这个地方就是使用cglib代理拦截@Bean注解的方法来实现的。

##### 案例3：拦截所有方法并返回固定值（FixedValue）

当调用某个类的任何方法的时候，都希望返回一个固定的值，此时可以使用FixedValue接口，如下：

enhancer.setCallback(new FixedValue() {  
 @Override  
 public Object loadObject() throws Exception {  
 return "路人甲";  
 }  
 });

上面创建的代理对象，调用其任意方法返回的都是"路人甲"。

案例代码如下：

创建一个类Service3，如下：

package com.javacode2018.lesson001.demo17;  
  
public class Service3 {  
 public String m1() {  
 System.out.println("我是m1方法");  
 return "hello:m1";  
 }  
  
 public String m2() {  
 System.out.println("我是m2方法");  
 return "hello:m2";  
 }  
}

对用的测试用例：

@Test  
public void test3() {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(Service3.class);  
 enhancer.setCallback(new FixedValue() {  
 @Override  
 public Object loadObject() throws Exception {  
 return "路人甲";  
 }  
 });  
 Service3 proxy = (Service3) enhancer.create();  
 System.out.println(proxy.m1());//@1  
 System.out.println(proxy.m2()); //@2  
 System.out.println(proxy.toString());//@3  
}

@1、@2、@3调用了代理对象的3个方法，运行输出：

运行输出：

路人甲  
路人甲  
路人甲

可以看出输出的都是一个拱顶的值。

##### 案例4：直接放行，不做任何操作（NoOp.INSTANCE）

Callback接口下面有个子接口org.springframework.cglib.proxy.NoOp，将这个作为Callback的时候，被调用的方法会直接放行，像没有任何代理一样，感受一下效果：

@Test  
public void test6() {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(Service3.class);  
 enhancer.setCallback(NoOp.INSTANCE);  
 Service3 proxy = (Service3) enhancer.create();  
 System.out.println(proxy.m1());  
 System.out.println(proxy.m2());  
}

运行输出：

我是m1方法  
hello:m1  
我是m2方法  
hello:m2

从输出中可以看出，被调用的方法没有被代理做任何处理，直接进到目标类Service3的方法中了。

##### 案例5：不同的方法使用不同的拦截器（CallbackFilter）

有个类如下：

package com.javacode2018.lesson001.demo17;  
  
public class Service4 {  
 public void insert1() {  
 System.out.println("我是insert1");  
 }  
  
 public void insert2() {  
 System.out.println("我是insert2");  
 }  
  
 public String get1() {  
 System.out.println("我是get1");  
 return "get1";  
 }  
  
 public String get2() {  
 System.out.println("我是get2");  
 return "get2";  
 }  
}

**需求，给这个类创建一个代理需要实现下面的功能：**

1. **以insert开头的方法需要统计方法耗时**
2. **以get开头的的方法直接返回固定字符串欢迎和【马士兵老师】一起学spring！**

下来看代码，然后再解释：

@Test  
public void test4() {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(Service4.class);  
 //创建2个Callback  
 Callback[] callbacks = {  
 //这个用来拦截所有insert开头的方法  
 new MethodInterceptor() {  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 long starTime = System.nanoTime();  
 Object result = methodProxy.invokeSuper(o, objects);  
 long endTime = System.nanoTime();  
 System.out.println(method + "，耗时(纳秒):" + (endTime - starTime));  
 return result;  
 }  
 },  
 //下面这个用来拦截所有get开头的方法，返回固定值的  
 new FixedValue() {  
 @Override  
 public Object loadObject() throws Exception {  
 return "马士兵老师";  
 }  
 }  
 };  
 enhancer.setCallbackFilter(new CallbackFilter() {  
 @Override  
 public int accept(Method method) {  
 return 0;  
 }  
 });  
 //调用enhancer的setCallbacks传递Callback数组  
 enhancer.setCallbacks(callbacks);  
 /\*\*  
 \* 设置过滤器CallbackFilter  
 \* CallbackFilter用来判断调用方法的时候使用callbacks数组中的哪个Callback来处理当前方法  
 \* 返回的是callbacks数组的下标  
 \*/  
 enhancer.setCallbackFilter(new CallbackFilter() {  
 @Override  
 public int accept(Method method) {  
 //获取当前调用的方法的名称  
 String methodName = method.getName();  
 /\*\*  
 \* 方法名称以insert开头，  
 \* 返回callbacks中的第1个Callback对象来处理当前方法，  
 \* 否则使用第二个Callback处理被调用的方法  
 \*/  
 return methodName.startsWith("insert") ? 0 : 1;  
 }  
 });  
 Service4 proxy = (Service4) enhancer.create();  
 System.out.println("---------------");  
 proxy.insert1();  
 System.out.println("---------------");  
 proxy.insert2();  
 System.out.println("---------------");  
 System.out.println(proxy.get1());  
 System.out.println("---------------");  
 System.out.println(proxy.get2());  
  
}

运行输出：

---------------  
我是insert1  
public void com.javacode2018.lesson001.demo17.Service4.insert1()，耗时(纳秒):15396100  
---------------  
我是insert2  
public void com.javacode2018.lesson001.demo17.Service4.insert2()，耗时(纳秒):66200  
---------------  
马士兵老师  
---------------  
马士兵老师

代码说明：

由于需求中要对不同的方法做不同的处理，所以需要有2个Callback对象，当调用代理对象的方法的时候，具体会走哪个Callback呢，此时会通过CallbackFilter中的accept来判断，这个方法返回callbacks数组的索引。

上面这个案例还有一种简单的实现，见案例6

##### 案例6：对案例5的优化（CallbackHelper）

cglib中有个CallbackHelper类，可以对案例5的代码进行有环，CallbackHelper类相当于对一些代码进行了封装，方便实现案例5的需求，实现如下：

@Test  
public void test5() {  
 Enhancer enhancer = new Enhancer();  
 //创建2个Callback  
 Callback costTimeCallback = (MethodInterceptor) (Object o, Method method, Object[] objects, MethodProxy methodProxy) -> {  
 long starTime = System.nanoTime();  
 Object result = methodProxy.invokeSuper(o, objects);  
 long endTime = System.nanoTime();  
 System.out.println(method + "，耗时(纳秒):" + (endTime - starTime));  
 return result;  
 };  
 //下面这个用来拦截所有get开头的方法，返回固定值的  
 Callback fixdValueCallback = (FixedValue) () -> "马士兵老师";  
 CallbackHelper callbackHelper = new CallbackHelper(Service4.class, null) {  
 @Override  
 protected Object getCallback(Method method) {  
 return method.getName().startsWith("insert") ? costTimeCallback : fixdValueCallback;  
 }  
 };  
 enhancer.setSuperclass(Service4.class);  
 //调用enhancer的setCallbacks传递Callback数组  
 enhancer.setCallbacks(callbackHelper.getCallbacks());  
 /\*\*  
 \* 设置CallbackFilter,用来判断某个方法具体走哪个Callback  
 \*/  
 enhancer.setCallbackFilter(callbackHelper);  
 Service4 proxy = (Service4) enhancer.create();  
 System.out.println("---------------");  
 proxy.insert1();  
 System.out.println("---------------");  
 proxy.insert2();  
 System.out.println("---------------");  
 System.out.println(proxy.get1());  
 System.out.println("---------------");  
 System.out.println(proxy.get2());  
  
}

运行输出：

---------------  
我是insert1  
public void com.javacode2018.lesson001.demo17.Service4.insert1()，耗时(纳秒):9777500  
---------------  
我是insert2  
public void com.javacode2018.lesson001.demo17.Service4.insert2()，耗时(纳秒):50600  
---------------  
马士兵老师  
---------------  
马士兵老师

输出效果和案例4一模一样的，上面重点在于CallbackHelper，里面做了一些封装，有兴趣的可以去看一下源码，比较简单。

##### 案例6：实现通用的统计任意类方法耗时代理类

直接上代码，比较简单，如下：

package com.javacode2018.lesson001.demo17;  
  
  
import org.springframework.cglib.proxy.Enhancer;  
import org.springframework.cglib.proxy.MethodInterceptor;  
import org.springframework.cglib.proxy.MethodProxy;  
  
import java.lang.reflect.Method;  
  
public class CostTimeProxy implements MethodInterceptor {  
 //目标对象  
 private Object target;  
  
 public CostTimeProxy(Object target) {  
 this.target = target;  
 }  
  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 long starTime = System.nanoTime();  
 //调用被代理对象（即target）的方法，获取结果  
 Object result = method.invoke(target, objects); //@1  
 long endTime = System.nanoTime();  
 System.out.println(method + "，耗时(纳秒)：" + (endTime - starTime));  
 return result;  
 }  
  
 /\*\*  
 \* 创建任意类的代理对象  
 \*  
 \* @param target  
 \* @param <T>  
 \* @return  
 \*/  
 public static <T> T createProxy(T target) {  
 CostTimeProxy costTimeProxy = new CostTimeProxy(target);  
 Enhancer enhancer = new Enhancer();  
 enhancer.setCallback(costTimeProxy);  
 enhancer.setSuperclass(target.getClass());  
 return (T) enhancer.create();  
 }  
}

我们可以直接使用上面的静态方法createProxy来为目标对象target创建一个代理对象，被代理的对象自动实现方法调用耗时统计。

@1：调用被代理对象的方法获取真正的结果。

使用非常简单，来个测试用例，如下：

@Test  
public void test7() {  
 //创建Service1代理  
 Service1 service1 = CostTimeProxy.createProxy(new Service1());  
 service1.m1();  
  
 //创建Service3代理  
 Service3 service3 = CostTimeProxy.createProxy(new Service3());  
 System.out.println(service3.m1());  
}

运行输出：

我是m1方法  
public void com.javacode2018.lesson001.demo17.Service1.m1()，耗时(纳秒)：53200  
我是m1方法  
public java.lang.String com.javacode2018.lesson001.demo17.Service3.m1()，耗时(纳秒)：49200  
hello:m1

### CGLIB和Java动态代理的区别

1. **Java动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；CGLIB能够代理普通类；**
2. **Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效**
3. **代理的就介绍到这里，spring中很多地方会用到这块，所以大家一定要熟悉，欢迎大家留言交流**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？

## Spring系列第16篇：深入理解java注解（预备知识）

**最近有个朋友去阿里面试，被面试官来了个灵魂拷问：**

1. **注解是干什么的？**
2. **一个注解可以使用多次么？如何使用？**
3. **@Inherited是做什么的？**
4. **@Target中的TYPE\_PARAMETER和TYPE\_USER用在什么地方？**
5. **泛型中如何使用注解？**
6. **注解定义可以实现继承么？**
7. **spring中对注解有哪些增强？@Aliasfor注解是干什么的？**

**第1个他回答上来了，后面的几个直接懵了，然后就没有然后了。**

**之后跑来问我，然后我让他看本文，准备下次去吊打面试官。**

### 本文内容

**带你玩转java注解，解决上面的所有问题。**

### 什么是注解？

代码中注释大家都熟悉吧，注释是给开发者看的，可以提升代码的可读性和可维护性，但是对于java编译器和虚拟机来说是没有意义的，编译之后的字节码文件中是没有注释信息的；而注解和注释有点类似，唯一的区别就是注释是给人看的，而注释是给编译器和虚拟机看的，编译器和虚拟机在运行的过程中可以获取注解信息，然后可以根据这些注解的信息做各种想做的事情。比如：大家对@Override应该比较熟悉，就是一个注解，加在方法上，标注当前方法重写了父类的方法，当编译器编译代码的时候，会对@Override标注的方法进行验证，验证其父类中是否也有同样签名的方法，否则报错，通过这个注解是不是增强了代码的安全性。

**总的来说：注解是对代码的一种增强，可以在代码编译或者程序运行期间获取注解的信息，然后根据这些信息做各种牛逼的事情。**

### 注解如何使用？

**3个步骤：**

1. **定义注解**
2. **使用注解**
3. **获取注解信息做各种牛逼的事情**

### 定义注解

关于注解的定义，先来几个问题：

1. 如何为注解定义参数？
2. 注解可以用在哪里？
3. 注解会被保留到什么时候？

#### 定义注解语法

jdk中注解相关的类和接口都定义在java.lang.annotation包中。

注解的定义和我们常见的类、接口类似，只是注解使用@interface来定义，如下定义一个名称为MyAnnotation的注解：

public @interface MyAnnotation {  
}

#### 注解中定义参数

注解有没有参数都可以，定义参数如下：

public @interface 注解名称{  
 [public] 参数类型 参数名称1() [default 参数默认值];  
 [public] 参数类型 参数名称2() [default 参数默认值];  
 [public] 参数类型 参数名称n() [default 参数默认值];  
}

注解中可以定义多个参数，参数的定义有以下特点：

1. 访问修饰符必须为public，不写默认为public
2. 该元素的类型只能是基本数据类型、String、Class、枚举类型、注解类型（体现了注解的嵌套效果）以及上述类型的一位数组
3. 该元素的名称一般定义为名词，如果注解中只有一个元素，请把名字起为value（后面使用会带来便利操作）
4. 参数名称后面的()不是定义方法参数的地方，也不能在括号中定义任何参数，仅仅只是一个特殊的语法
5. default代表默认值，值必须和第2点定义的类型一致
6. 如果没有默认值，代表后续使用注解时必须给该类型元素赋值

#### 指定注解的使用范围：@Target

使用@Target注解定义注解的使用范围，如下：

@Target(value = {ElementType.TYPE,ElementType.METHOD})  
public @interface MyAnnotation {  
}

上面指定了MyAnnotation注解可以用在类、接口、注解类型、枚举类型以及方法上面，**自定义注解上也可以不使用@Target注解，如果不使用，表示自定义注解可以用在任何地方**。

看一下@Target源码：

@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION\_TYPE)  
public @interface Target {  
 ElementType[] value();  
}

有一个参数value，是ElementType类型的一个数组，再来看一下ElementType，是个枚举，源码如下：

package java.lang.annotation;  
/\*注解的使用范围\*/  
public enum ElementType {  
 /\*类、接口、枚举、注解上面\*/  
 TYPE,  
 /\*字段上\*/  
 FIELD,  
 /\*方法上\*/  
 METHOD,  
 /\*方法的参数上\*/  
 PARAMETER,  
 /\*构造函数上\*/  
 CONSTRUCTOR,  
 /\*本地变量上\*/  
 LOCAL\_VARIABLE,  
 /\*注解上\*/  
 ANNOTATION\_TYPE,  
 /\*包上\*/  
 PACKAGE,  
 /\*类型参数上\*/  
 TYPE\_PARAMETER,  
 /\*类型名称上\*/  
 TYPE\_USE  
}

#### 指定注解的保留策略：@Retention

我们先来看一下java程序的3个过程

1. 源码阶段
2. 源码被编译为字节码之后变成class文件
3. 字节码被虚拟机加载然后运行

那么自定义注解会保留在上面哪个阶段呢？可以通过@Retention注解来指定，如：

@Retention(RetentionPolicy.SOURCE)  
public @interface MyAnnotation {  
}

上面指定了MyAnnotation只存在于源码阶段，后面的2个阶段都会丢失。

来看一下@Retention

@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION\_TYPE)  
public @interface Retention {  
 RetentionPolicy value();  
}

有一个value参数，类型为RetentionPolicy枚举，如下：

public enum RetentionPolicy {  
 /\*注解只保留在源码中，编译为字节码之后就丢失了，也就是class文件中就不存在了\*/  
 SOURCE,  
 /\*注解只保留在源码和字节码中，运行阶段会丢失\*/  
 CLASS,  
 /\*源码、字节码、运行期间都存在\*/  
 RUNTIME  
}

### 使用注解

#### 语法

将注解加载使用的目标上面，如下：

@注解名称(参数1=值1,参数2=值2,参数n=值n)  
目标对象

直接来案例说明。

#### 无参注解

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann1 { //@1  
}  
  
@Ann1 //@2  
public class UseAnnotation1 {  
}

@1：Ann1为无参注解

@2：类上使用@Ann1注解，没有参数

#### 一个参数的注解

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann2 { //@1  
 String name();  
}  
  
@Ann2(name = "我是马士兵老师") //@2  
public class UseAnnotation2 {  
  
}

#### 一个参数为value的注解，可以省略参数名称

只有一个参数，名称为value的时候，使用时参数名称可以省略

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann3 {  
 String value();//@1  
}  
  
@Ann3("我是马士兵老师") //@2  
public class UseAnnotation3 {  
  
}

@1：注解之后一个参数，名称为value

@2：使用注解，参数名称value省略了

#### 数组类型参数

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann4 {  
 String[] name();//@1  
}  
  
@Ann4(name = {"我是马士兵老师", "欢迎和我一起学spring"}) //@2  
public class UseAnnotation4 {  
 @Ann4(name = "如果只有一个值，{}可以省略") //@3  
 public class T1 {  
 }  
}

@1：name的类型是一个String类型的数组

@2：name有多个值的时候，需要使用{}包含起来

@3：如果name只有一个值，{}可以省略

#### 为参数指定默认值

通过default为参数指定默认值，用的时候如果没有设置值，则取默认值，没有指定默认值的参数，使用的时候必须为参数设置值，如下：

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann5 {  
 String[] name() default {"马士兵老师", "spring系列"};//@1  
 int[] score() default 1; //@2  
 int age() default 30; //@3  
 String address(); //@4  
}  
  
@Ann5(age = 32,address = "上海") //@5  
public class UseAnnotation5 {  
  
}

@1：数组类型通过{}指定默认值

@2：数组类型参数，默认值只有一个省略了{}符号

@3：默认值为30

@4：未指定默认值

@5：age=32对默认值进行了覆盖，并且为address指定了值

#### 综合案例

@Target(value = {  
 ElementType.TYPE,  
 ElementType.METHOD,  
 ElementType.FIELD,  
 ElementType.PARAMETER,  
 ElementType.CONSTRUCTOR,  
 ElementType.LOCAL\_VARIABLE  
})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann6 {  
 String value();  
  
 ElementType elementType();  
}  
  
@Ann6(value = "我用在类上", elementType = ElementType.TYPE)  
public class UseAnnotation6 {  
 @Ann6(value = "我用在字段上", elementType = ElementType.FIELD)  
 private String a;  
  
 @Ann6(value = "我用在构造方法上", elementType = ElementType.CONSTRUCTOR)  
 public UseAnnotation6(@Ann6(value = "我用在方法参数上", elementType = ElementType.PARAMETER) String a) {  
 this.a = a;  
 }  
  
 @Ann6(value = "我用在了普通方法上面", elementType = ElementType.METHOD)  
 public void m1() {  
 @Ann6(value = "我用在了本地变量上", elementType = ElementType.LOCAL\_VARIABLE) String a;  
 }  
}

上面演示了自定义注解在在类、字段、构造器、方法参数、方法、本地变量上的使用，@Ann6注解有个elementType参数，我想通过这个参数的值来告诉大家对应@Target中的那个值来限制使用目标的，大家注意一下上面每个elementType的值。

#### @Target(ElementType.TYPE\_PARAMETER)

这个是1.8加上的，用来标注类型参数，类型参数一般在类后面声明或者方法上声明，这块需要先了解一下泛型[泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933878&idx=1&sn=bebd543c39d02455456680ff12e3934b&chksm=88621dc8bf1594de6b50a760e4141b80da76442ba38fb93a91a3d18ecf85e7eee368f2c159d3&token=799820369&lang=zh_CN&scene=21" \l "wechat_redirect)不然理解起来比较吃力，来个案例感受一下：

@Target(value = {  
 ElementType.TYPE\_PARAMETER  
})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann7 {  
 String value();  
}  
  
public class UseAnnotation7<@Ann7("T0是在类上声明的一个泛型类型变量") T0, @Ann7("T1是在类上声明的一个泛型类型变量") T1> {  
  
 public <@Ann7("T2是在方法上声明的泛型类型变量") T2> void m1() {  
 }  
  
 public static void main(String[] args) throws NoSuchMethodException {  
 for (TypeVariable typeVariable : UseAnnotation7.class.getTypeParameters()) {  
 print(typeVariable);  
 }  
  
 for (TypeVariable typeVariable : UseAnnotation7.class.getDeclaredMethod("m1").getTypeParameters()) {  
 print(typeVariable);  
 }  
 }  
  
 private static void print(TypeVariable typeVariable) {  
 System.out.println("类型变量名称:" + typeVariable.getName());  
 Arrays.stream(typeVariable.getAnnotations()).forEach(System.out::println);  
 }  
}

类和方法上面可以声明泛型类型的变量，上面有3个泛型类型变量，我们运行一下看看效果：

类型变量名称:T0  
@com.javacode2018.lesson001.demo18.Ann7(value=T0是在类上声明的一个泛型类型变量)  
类型变量名称:T1  
@com.javacode2018.lesson001.demo18.Ann7(value=T1是在类上声明的一个泛型类型变量)  
类型变量名称:T2  
@com.javacode2018.lesson001.demo18.Ann7(value=T2是在方法上声明的泛型类型变量)

#### @Target(ElementType.TYPE\_USE)

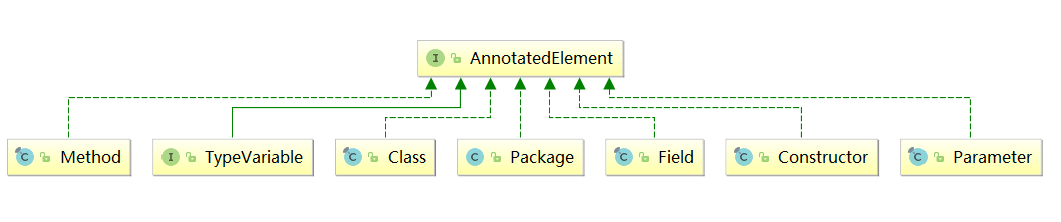
这个是1.8加上的，能用在任何类型名称上，来个案例感受一下：

@Target({ElementType.TYPE\_USE})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann10 {  
 String value();  
}  
  
@Ann10("用在了类上")  
public class UserAnnotation10<@Ann10("用在了类变量类型V1上") V1, @Ann10("用在了类变量类型V2上") V2> {  
  
 private Map<@Ann10("用在了泛型类型上") String, Integer> map;  
  
 public <@Ann10("用在了参数上") T> String m1(String name) {  
 return null;  
 }  
  
}

类后面的V1、V2都是类型名称，Map后面的尖括号也是类型名称，m1方法前面也定义了一个类型变量，名称为T

### 注解信息的获取

为了运行时能准确获取到注解的相关信息，Java在java.lang.reflect 反射包下新增了AnnotatedElement接口，它主要用于表示目前正在虚拟机中运行的程序中已使用注解的元素，通过该接口提供的方法可以利用反射技术地读取注解的信息，看一下UML图:



* Package：用来表示包的信息
* Class：用来表示类的信息
* Constructor：用来表示构造方法信息
* Field：用来表示类中属性信息
* Method：用来表示方法信息
* Parameter：用来表示方法参数信息
* TypeVariable：用来表示类型变量信息，如：类上定义的泛型类型变量，方法上面定义的泛型类型变量

#### AnnotatedElement常用方法

fdsfd

|  |  |  |
| --- | --- | --- |
| 返回值 | 方法名称 | 说明 |
| <A extends Annotation> | getAnnotation(Class<A> annotationClass) | 该元素如果存在指定类型的注解，则返回这些注解，否则返回 null。 |
| Annotation[] | getAnnotations() | 返回此元素上存在的所有注解，包括从父类继承的 |
| boolean | isAnnotationPresent(Class<? extends Annotation> annotationClass) | 如果指定类型的注解存在于此元素上，则返回 true，否则返回 false。 |
| Annotation[] | getDeclaredAnnotations() | 返回直接存在于此元素上的所有注解，注意，不包括父类的注解，调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响，没有则返回长度为0的数组 |

#### 案例

##### 要解析的列如下

package com.javacode2018.lesson001.demo18;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
import java.util.Map;  
  
@Target({ElementType.PACKAGE,  
 ElementType.TYPE,  
 ElementType.FIELD,  
 ElementType.CONSTRUCTOR,  
 ElementType.METHOD,  
 ElementType.PARAMETER,  
 ElementType.TYPE\_PARAMETER,  
 ElementType.TYPE\_USE})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann11 {  
 String value();  
}  
  
@Target({ElementType.PACKAGE,  
 ElementType.TYPE,  
 ElementType.FIELD,  
 ElementType.CONSTRUCTOR,  
 ElementType.METHOD,  
 ElementType.PARAMETER,  
 ElementType.TYPE\_PARAMETER,  
 ElementType.TYPE\_USE})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Ann11\_0 {  
 int value();  
}  
  
@Ann11("用在了类上")  
@Ann11\_0(0)  
public class UseAnnotation11<@Ann11("用在了类变量类型V1上") @Ann11\_0(1) V1, @Ann11("用在了类变量类型V2上") @Ann11\_0(2) V2> {  
 @Ann11("用在了字段上")  
 @Ann11\_0(3)  
 private String name;  
  
 private Map<@Ann11("用在了泛型类型上,String") @Ann11\_0(4) String, @Ann11("用在了泛型类型上,Integer") @Ann11\_0(5) Integer> map;  
  
 @Ann11("用在了构造方法上")  
 @Ann11\_0(6)  
 public UseAnnotation11() {  
 this.name = name;  
 }  
  
 @Ann11("用在了返回值上")  
 @Ann11\_0(7)  
 public String m1(@Ann11("用在了参数上") @Ann11\_0(8) String name) {  
 return null;  
 }  
  
}

##### 解析类上的注解

###### 解析这部分

@Ann11("用在了类上")

###### 代码

@Test  
public void m1() {  
 for (Annotation annotation : UserAnnotation10.class.getAnnotations()) {  
 System.out.println(annotation);  
 }  
}

###### 运行输出

@com.javacode2018.lesson001.demo18.Ann11(value=用在了类上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=0)

##### 解析类上的类型变量

解析类名后面的尖括号的部分，即下面的部分：

UseAnnotation11<@Ann11("用在了类变量类型V1上") @Ann11\_0(1) V1, @Ann11("用在了类变量类型V2上") @Ann11\_0(2) V2>

###### 用例代码

@Test  
public void m2() {  
 TypeVariable<Class<UserAnnotation10>>[] typeParameters = UserAnnotation10.class.getTypeParameters();  
 for (TypeVariable<Class<UserAnnotation10>> typeParameter : typeParameters) {  
 System.out.println(typeParameter.getName() + "变量类型注解信息：");  
 Annotation[] annotations = typeParameter.getAnnotations();  
 for (Annotation annotation : annotations) {  
 System.out.println(annotation);  
 }  
 }  
}

###### 运行输出

V1变量类型注解信息：  
@com.javacode2018.lesson001.demo18.Ann11(value=用在了类变量类型V1上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=1)  
V2变量类型注解信息：  
@com.javacode2018.lesson001.demo18.Ann11(value=用在了类变量类型V2上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=2)

##### 解析字段name上的注解

###### 用例代码

@Test  
public void m3() throws NoSuchFieldException {  
 Field nameField = UserAnnotation10.class.getDeclaredField("name");  
 for (Annotation annotation : nameField.getAnnotations()) {  
 System.out.println(annotation);  
 }  
}

###### 运行输出

@com.javacode2018.lesson001.demo18.Ann11(value=用在了字段上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=3)

##### 解析泛型字段map上的注解

###### 用例代码

@Test  
public void m4() throws NoSuchFieldException, ClassNotFoundException {  
 Field field = UseAnnotation11.class.getDeclaredField("map");  
 Type genericType = field.getGenericType();  
 Type[] actualTypeArguments = ((ParameterizedType) genericType).getActualTypeArguments();  
  
 AnnotatedType annotatedType = field.getAnnotatedType();  
 AnnotatedType[] annotatedActualTypeArguments = ((AnnotatedParameterizedType) annotatedType).getAnnotatedActualTypeArguments();  
 int i = 0;  
 for (AnnotatedType actualTypeArgument : annotatedActualTypeArguments) {  
 Type actualTypeArgument1 = actualTypeArguments[i++];  
 System.out.println(actualTypeArgument1.getTypeName() + "类型上的注解如下：");  
 for (Annotation annotation : actualTypeArgument.getAnnotations()) {  
 System.out.println(annotation);  
 }  
 }  
}

###### 运行输出

java.lang.String类型上的注解如下：  
@com.javacode2018.lesson001.demo18.Ann11(value=用在了泛型类型上,String)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=4)  
java.lang.Integer类型上的注解如下：  
@com.javacode2018.lesson001.demo18.Ann11(value=用在了泛型类型上,Integer)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=5)

##### 解析构造函数上的注解

###### 用例代码

@Test  
public void m5() {  
 Constructor<?> constructor = UseAnnotation11.class.getConstructors()[0];  
 for (Annotation annotation : constructor.getAnnotations()) {  
 System.out.println(annotation);  
 }  
}

###### 运行输出

@com.javacode2018.lesson001.demo18.Ann11(value=用在了构造方法上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=6)

##### 解析m1方法上的注解

###### 用例代码

@Test  
public void m6() throws NoSuchMethodException {  
 Method method = UseAnnotation11.class.getMethod("m1", String.class);  
 for (Annotation annotation : method.getAnnotations()) {  
 System.out.println(annotation);  
 }  
}

###### 运行输出

@com.javacode2018.lesson001.demo18.Ann11(value=用在了返回值上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=7)

##### 解析m1方法参数注解

###### 用例代码

@Test  
public void m7() throws NoSuchMethodException {  
 Method method = UseAnnotation11.class.getMethod("m1", String.class);  
 for (Parameter parameter : method.getParameters()) {  
 System.out.println(String.format("参数%s上的注解如下:", parameter.getName()));  
 for (Annotation annotation : parameter.getAnnotations()) {  
 System.out.println(annotation);  
 }  
 }  
}

###### 运行输出

参数arg0上的注解如下:  
@com.javacode2018.lesson001.demo18.Ann11(value=用在了参数上)  
@com.javacode2018.lesson001.demo18.Ann11\_0(value=8)

###### 上面参数名称为arg0，如果想让参数名称和源码中真实名称一致，操作如下：

如果你编译这个class的时候没有添加参数–parameters，运行的时候你会得到这个结果：  
  
Parameter: arg0  
  
编译的时候添加了–parameters参数的话，运行结果会不一样：  
  
Parameter: args  
  
对于有经验的Maven使用者，–parameters参数可以添加到maven-compiler-plugin的配置部分：  
  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.1</version>  
 <configuration>  
 <compilerArgument>-parameters</compilerArgument>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
</plugin>

### @Inherit：实现类之间的注解继承

#### 用法

来看一下这个注解的源码

@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION\_TYPE)  
public @interface Inherited {  
}

我们通过@Target元注解的属性值可以看出，这个@Inherited 是专门修饰注解的。

**作用：让子类可以继承父类中被@Inherited修饰的注解，注意是继承父类中的，如果接口中的注解也使用@Inherited修饰了，那么接口的实现类是无法继承这个注解的**

#### 案例

package com.javacode2018.lesson001.demo18;  
  
import java.lang.annotation.\*;  
  
public class InheritAnnotationTest {  
 @Target(ElementType.TYPE)  
 @Retention(RetentionPolicy.RUNTIME)  
 @Inherited  
 @interface A1{ //@1  
 }  
 @Target(ElementType.TYPE)  
 @Retention(RetentionPolicy.RUNTIME)  
 @Inherited  
 @interface A2{ //@2  
 }  
  
 @A1 //@3  
 interface I1{}  
 @A2 //@4  
 static class C1{}  
  
 static class C2 extends C1 implements I1{} //@5  
  
 public static void main(String[] args) {  
 for (Annotation annotation : C2.class.getAnnotations()) { //@6  
 System.out.println(annotation);  
 }  
 }  
}

@1：定义了一个注解A1，上面使用了@Inherited，表示这个具有继承功能

@2：定义了一个注解A2，上面使用了@Inherited，表示这个具有继承功能

@3：定义接口I1，上面使用了@A1注解

@4：定义了一个C1类，使用了A2注解

@5：C2继承了C1并且实现了I1接口

@6：获取C2上以及从父类继承过来的所有注解，然后输出

运行输出：

@com.javacode2018.lesson001.demo18.InheritAnnotationTest$A2()

**从输出中可以看出类可以继承父类上被@Inherited修饰的注解，而不能继承接口上被@Inherited修饰的注解，这个一定要注意**

### @Repeatable重复使用注解

来看一段代码：

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@interface Ann12{}  
  
@Ann12  
@Ann12  
public class UseAnnotation12 {  
}

上面代码会报错，原因是：UseAnnotation12上面重复使用了@Ann12注解，默认情况下@Ann12注解是不允许重复使用的。

像上面这样，如果我们想重复使用注解的时候，需要用到@Repeatable注解

#### 使用步骤

##### 先定义容器注解

@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.FIELD})  
@interface Ann12s {  
 Ann12[] value(); //@1  
}

容器注解中必须有个value类型的参数，参数类型为子注解类型的数组。

##### 为注解指定容器

要让一个注解可以重复使用，需要在注解上加上@Repeatable注解，@Repeatable中value的值为容器注解，如下代码中的@2

@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.FIELD})  
@Repeatable(Ann12s.class)//@2  
@interface Ann12 {  
 String name();  
}

##### 使用注解

重复使用相同的注解有2种方式，如下面代码

1. 重复使用注解，如下面的类上重复使用@Ann12注解
2. 通过容器注解来使用更多个注解，如下面的字段v1上使用@Ann12s容器注解

@Ann12(name = "马士兵老师")  
@Ann12(name = "Spring系列")  
public class UseAnnotation12 {  
 @Ann12s(  
 {@Ann12(name = "Java高并发系列，见公众号"),  
 @Ann12(name = "mysql高手系列，见公众号")}  
 )  
 private String v1;  
}

##### 获取注解信息

com.javacode2018.lesson001.demo18.UseAnnotation12  
  
@Test  
public void test1() throws NoSuchFieldException {  
 Annotation[] annotations = UseAnnotation12.class.getAnnotations();  
 for (Annotation annotation : annotations) {  
 System.out.println(annotation);  
 }  
 System.out.println("-------------");  
 Field v1 = UseAnnotation12.class.getDeclaredField("v1");  
 Annotation[] declaredAnnotations = v1.getDeclaredAnnotations();  
 for (Annotation declaredAnnotation : declaredAnnotations) {  
 System.out.println(declaredAnnotation);  
 }  
}

运行输出：

@com.javacode2018.lesson001.demo18.Ann12s(value=[@com.javacode2018.lesson001.demo18.Ann12(name=马士兵老师), @com.javacode2018.lesson001.demo18.Ann12(name=Spring系列)])  
-------------  
@com.javacode2018.lesson001.demo18.Ann12s(value=[@com.javacode2018.lesson001.demo18.Ann12(name=Java高并发系列，见公众号), @com.javacode2018.lesson001.demo18.Ann12(name=mysql高手系列，见公众号)])

上面就是java中注解的功能，下面我们来介绍spring对于注解方面的支持。

### 先来看一个问题

代码如下：

package com.javacode2018.lesson001.demo18;  
  
import org.junit.Test;  
import org.springframework.core.annotation.AnnotatedElementUtils;  
import org.springframework.core.annotation.AnnotationUtils;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface A1 {  
 String value() default "a";//@0  
}  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@A1  
@interface B1 { //@1  
 String value() default "b";//@2  
}  
  
@B1("马士兵老师") //@3  
public class UseAnnotation13 {  
 @Test  
 public void test1() {  
 //AnnotatedElementUtils是spring提供的一个查找注解的工具类  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation13.class, B1.class));  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation13.class, A1.class));  
 }  
}

@0：A1注解value参数值默认为a

@1：B1注解上使用到了@A1注解

@2：B1注解value参数值默认为b

@2：UseAnnotation13上面使用了@B1注解，value参数的值为：马士兵老师

test1方法中使用到了spring中的一个类AnnotatedElementUtils，通过这个工具类可以很方便的获取注解的各种信息，方法中的2行代码用于获取UseAnnotation13类上B1注解和A1注解的信息。

运行test1方法输出：

@com.javacode2018.lesson001.demo18.B1(value=马士兵老师)  
@com.javacode2018.lesson001.demo18.A1(value=a)

上面用法很简单，没什么问题。

**此时有个问题：此时如果想在UseAnnotation13上给B1上的A1注解设置值是没有办法的，注解定义无法继承导致的，如果注解定义上面能够继承，那用起来会爽很多，spring通过@Aliasfor方法解决了这个问题。**

### Spring @AliasFor：对注解进行增强

直接上案例，然后解释代码。

#### 案例1：通过@AliasFor解决刚才难题

package com.javacode2018.lesson001.demo18;  
  
import org.junit.Test;  
import org.springframework.core.annotation.AliasFor;  
import org.springframework.core.annotation.AnnotatedElementUtils;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface A14 {  
 String value() default "a";//@0  
}  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@A14 //@6  
@interface B14 { //@1  
  
 String value() default "b";//@2  
  
 @AliasFor(annotation = A14.class, value = "value") //@5  
 String a14Value();  
}  
  
@B14(value = "马士兵老师",a14Value = "通过B14给A14的value参数赋值") //@3  
public class UseAnnotation14 {  
 @Test  
 public void test1() {  
 //AnnotatedElementUtils是spring提供的一个查找注解的工具类  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation14.class, B14.class));  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation14.class, A14.class));  
 }  
}

运行输出：

@com.javacode2018.lesson001.demo18.B14(a14Value=通过B14给A14的value参数赋值, value=马士兵老师)  
@com.javacode2018.lesson001.demo18.A14(value=通过B14给A14的value参数赋值)

注意上面diam的@3只使用了B14注解，大家认真看一下，上面输出汇总可以看出A14的value值和B14的a14Value参数值一样，说明通过B14给A14设置值成功了。

重点在于代码@5，这个地方使用到了@AliasFor注解：

@AliasFor(annotation = A14.class, value = "value")

**这个相当于给某个注解指定别名，即将B1注解中a14Value参数作为A14中value参数的别名，当给B1的a14Value设置值的时候，就相当于给A14的value设置值，有个前提是@AliasFor注解的annotation参数指定的注解需要加载当前注解上面，如：@6**

#### 案例2：同一个注解中使用@AliasFor

package com.javacode2018.lesson001.demo18;  
  
import org.junit.Test;  
import org.springframework.core.annotation.AliasFor;  
import org.springframework.core.annotation.AnnotatedElementUtils;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Target({ElementType.TYPE, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@interface A15 {  
 @AliasFor("v2")//@1  
 String v1() default "";  
  
 @AliasFor("v1")//@2  
 String v2() default "";  
}  
  
@A15(v1 = "我是v1") //@3  
public class UseAnnotation15 {  
  
 @A15(v2 = "我是v2") //@4  
 private String name;  
  
 @Test  
 public void test1() throws NoSuchFieldException {  
 //AnnotatedElementUtils是spring提供的一个查找注解的工具类  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation15.class, A15.class));  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation15.class.getDeclaredField("name"), A15.class));  
 }  
}

注意上面代码，A15注解中（@1和@2）的2个参数都设置了@AliasFor，@AliasFor如果不指定annotation参数的值，那么annotation默认值就是当前注解，所以上面2个属性互为别名，当给v1设置值的时候也相当于给v2设置值，当给v2设置值的时候也相当于给v1设置值。

运行输出

@com.javacode2018.lesson001.demo18.A15(v1=我是v1, v2=我是v1)  
@com.javacode2018.lesson001.demo18.A15(v1=我是v2, v2=我是v2)

从输出中可以看出v1和v2的值始终是相等的，上面如果同时给v1和v2设置值的时候运行代码会报错。

我们回头来看看@AliasFor的源码：

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
@Documented  
public @interface AliasFor {  
  
 @AliasFor("attribute")  
 String value() default "";  
  
 @AliasFor("value")  
 String attribute() default "";  
  
 Class<? extends Annotation> annotation() default Annotation.class;  
  
}

AliasFor注解中value和attribute互为别名，随便设置一个，同时会给另外一个设置相同的值。

#### 案例2：@AliasFor中不指定value和attribute

当@AliasFor中不指定value或者attribute的时候，自动将@AliasFor修饰的参数作为value和attribute的值，如下@AliasFor注解的value参数值为name

package com.javacode2018.lesson001.demo18;  
  
import org.junit.Test;  
import org.springframework.core.annotation.AliasFor;  
import org.springframework.core.annotation.AnnotatedElementUtils;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface A16 {  
 String name() default "a";//@0  
}  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@A16  
@interface B16 { //@1  
  
 @AliasFor(annotation = A16.class) //@5  
 String name() default "b";//@2  
}  
  
@B16(name="我是v1") //@3  
public class UseAnnotation16 {  
  
  
 @Test  
 public void test1() throws NoSuchFieldException {  
 //AnnotatedElementUtils是spring提供的一个查找注解的工具类  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation16.class, A16.class));  
 System.out.println(AnnotatedElementUtils.getMergedAnnotation(UseAnnotation16.class, B16.class));  
 }  
}

运行输出：

@com.javacode2018.lesson001.demo18.A16(name=我是v1)  
@com.javacode2018.lesson001.demo18.B16(name=我是v1)

### 总结

**到目前为止文章开头的问题，想必各位都可以回答上来了，文章内容比较多，大家最好去敲一遍，加深理解。**

**欢迎留言！**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？

### Spring系列第17篇：@Configration和@Bean注解

**上次去头条面试，如下：**

**面试官**：spring中，类上加不加@Configuration注解，有什么区别？

**我**：当某个类上有@Configuration注解的时候，可以在这个类中使用@Bean注解向spring容器中注册bean；如果不加@Configuration注解，不能通过@Bean注解注册bean。

**面试官**：你确定不用@Configuration注解，不能通过@Bean注解来注册bean？

**我**：看着面试官，犹豫了10秒钟，说：不加@Configuration通过@Bean注解也可以注册bean

**面试官**：你确定可以注册？

**我**：嗯。。。。嗯。。。。嗯，我确定可以注册

**面试官**：那加不加到底有什么区别呢？

**我**：好像没有什么区别啊

**面试官**：好像没区别。。。。先回去等通知吧！

结果可想而知，没戏了！

回去之后立即看spring的源码，终于搞清了这个问题，原来加不加@Configuration还是有相当大的区别的。

下面我们就来看看@Configuration注解到底是干什么的，加不加到底有什么区别，下次你们去面试的时候就可以给面试官吹吹了。

之前我们都是通过xml的方式定义bean，里面会写很多bean元素，然后spring启动的时候，就会读取bean xml配置文件，然后解析这些配置，然后会将这些bean注册到spring容器中，供使用者使用。

jdk1.5里面有了注解的功能，spring也没闲着，觉得注解挺好用的，就将注解加了进来，让我们通过注解的方式来定义bean，用起来能达到xml中定义bean一样的效果，并且更简洁一些，这里面需要用到的注解就有@Configuration注解和@Bean注解。

### @Configuration注解

#### 用法

@Configuration这个注解可以加在类上，让这个类的功能等同于一个bean xml配置文件，如下：

@Configuration  
public class ConfigBean {  
   
}

上面代码类似于下面的xml：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
</beans>

通过AnnotationConfigApplicationContext来加载@Configuration修饰的类，如下：

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ConfigBean.class);

此时ConfigBean类中没有任何内容，相当于一个空的xml配置文件，此时我们要在ConfigBean类中注册bean，那么我们就要用到@Bean注解了。

#### 总结一下

@Configuration使用步骤：

1. 在类上使用@Configuration注解
2. 通过AnnotationConfigApplicationContext容器来加@Configuration注解修饰的类

### @Bean注解

#### 用法

这个注解类似于bean xml配置文件中的bean元素，用来在spring容器中注册一个bean。

@Bean注解用在方法上，表示通过方法来定义一个bean，默认将方法名称作为bean名称，将方法返回值作为bean对象，注册到spring容器中。

如：

@Bean  
public User user1() {  
 return new User();  
}

@Bean注解还有很多属性，我们来看一下其源码：

@Target({ElementType.METHOD, ElementType.ANNOTATION\_TYPE}) //@1  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Bean {  
  
 @AliasFor("name")  
 String[] value() default {};  
  
 @AliasFor("value")  
 String[] name() default {};  
  
 @Deprecated  
 Autowire autowire() default Autowire.NO;  
  
 boolean autowireCandidate() default true;  
   
 String initMethod() default "";  
  
 String destroyMethod() default AbstractBeanDefinition.INFER\_METHOD;  
}

@1：说明这个注解可以用在方法和注解类型上面。

每个参数含义：

1. value和name是一样的，设置的时候，这2个参数只能选一个，原因是@AliasFor导致的

* @AliasFor这个注解不清楚的可以看这个文章：[详解注解](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934095&idx=1&sn=26d539ef61389bae5d293f1b2f5210b2&chksm=88621ef1bf1597e756ccaeb6c6c6f4b74c6e3ba22ca6adba496b05e81558cd3801c62b21b8d9&token=1042984313&lang=zh_CN" \l "rd)

1. value：字符串数组，第一个值作为bean的名称，其他值作为bean的别名
2. autowire：这个参数上面标注了@Deprecated，表示已经过期了，不建议使用了
3. autowireCandidate：是否作为其他对象注入时候的候选bean，之前的文章中专门介绍过这个属性，不清楚的可以去看看：[autowire-candidate详解](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934008&idx=1&sn=ac156fe2788c49e0014bb5056139206e&chksm=88621e46bf1597505eba3e716148efcd9acec72ee6c0d95cf3936be70241fd41b180f0de02b5&token=1248115129&lang=zh_CN&scene=21" \l "wechat_redirect)
4. initMethod：bean初始化的方法，这个和生命周期有关，以后详解
5. destroyMethod：bean销毁的方法，也是和生命周期相关的，以后详解

#### 案例

##### User类

package com.javacode2018.lesson001.demo20;  
  
public class User {  
}

##### Bean配置类：ConfigBean

package com.javacode2018.lesson001.demo20;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ConfigBean {  
  
 //bean名称为方法默认值：user1  
 @Bean  
 public User user1() {  
 return new User();  
 }  
  
 //bean名称通过value指定了：user2Bean  
 @Bean("user2Bean")  
 public User user2() {  
 return new User();  
 }  
  
 //bean名称为：user3Bean，2个别名：[user3BeanAlias1,user3BeanAlias2]  
 @Bean({"user3Bean", "user3BeanAlias1", "user3BeanAlias2"})  
 public User user3() {  
 return new User();  
 }  
  
}

上面通过@Bean注解定义了3个bean，比较简单

##### 来个测试类

package com.javacode2018.lesson001.demo20;  
  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.Arrays;  
  
public class ConfigurationTest {  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ConfigBean.class);//@1  
 for (String beanName : context.getBeanDefinitionNames()) {  
 //别名  
 String[] aliases = context.getAliases(beanName);  
 System.out.println(String.format("bean名称:%s,别名:%s,bean对象:%s",  
 beanName,  
 Arrays.asList(aliases),  
 context.getBean(beanName)));  
 }  
 }  
}

@1：通过AnnotationConfigApplicationContext来加载配置类ConfigBean，会将配置类中所有的bean注册到spring容器中

for循环中输出了bean名称、别名、bean对象

##### 运行test1方法输出

bean名称:org.springframework.context.annotation.internalConfigurationAnnotationProcessor,别名:[],bean对象:org.springframework.context.annotation.ConfigurationClassPostProcessor@3bd82cf5  
bean名称:org.springframework.context.annotation.internalAutowiredAnnotationProcessor,别名:[],bean对象:org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor@544fa968  
bean名称:org.springframework.context.annotation.internalCommonAnnotationProcessor,别名:[],bean对象:org.springframework.context.annotation.CommonAnnotationBeanPostProcessor@247bddad  
bean名称:org.springframework.context.event.internalEventListenerProcessor,别名:[],bean对象:org.springframework.context.event.EventListenerMethodProcessor@d35dea7  
bean名称:org.springframework.context.event.internalEventListenerFactory,别名:[],bean对象:org.springframework.context.event.DefaultEventListenerFactory@7770f470  
bean名称:configBean,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean$$EnhancerBySpringCGLIB$$dde45976@5e5d171f  
bean名称:user1,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@24313fcc  
bean名称:user2Bean,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@7d20d0b  
bean名称:user3Bean,别名:[user3BeanAlias2, user3BeanAlias1],bean对象:com.javacode2018.lesson001.demo20.User@77f1baf5

上面的输出，我们主要关注与最后4行，前面的可以先忽略。

从输出中可以看出，有个名称为configBean的bean，正是ConfigBean这个类型，可以得出，被@Configuration修饰的类，也被注册到spring容器中了

最后3行输出就是几个User的bean对象了。

**上面的用法应该很多人都比较熟悉，下面的属于重点了。**

### 去掉@Configuration会怎么样？

我们来看一下没有@Configuration的时候，什么效果。

##### 新建一个ConfigBean1类

内容和ConfigBean类一样，只是将@Configuration注解去掉了，如下：

public class ConfigBean1 {  
  
 //bean名称为方法默认值：user1  
 @Bean  
 public User user1() {  
 return new User();  
 }  
  
 //bean名称通过value指定了：user2Bean  
 @Bean("user2Bean")  
 public User user2() {  
 return new User();  
 }  
  
 //bean名称为：user3Bean，2个别名：[user3BeanAlias1,user3BeanAlias2]  
 @Bean({"user3Bean", "user3BeanAlias1", "user3BeanAlias2"})  
 public User user3() {  
 return new User();  
 }  
  
}

##### 来个测试用例test2

代码类似于test1，给spring容器传递ConfigBean1

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ConfigBean1.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 //别名  
 String[] aliases = context.getAliases(beanName);  
 System.out.println(String.format("bean名称:%s,别名:%s,bean对象:%s",  
 beanName,  
 Arrays.asList(aliases),  
 context.getBean(beanName)));  
 }  
}

##### 运行输出

bean名称:org.springframework.context.annotation.internalConfigurationAnnotationProcessor,别名:[],bean对象:org.springframework.context.annotation.ConfigurationClassPostProcessor@333291e3  
bean名称:org.springframework.context.annotation.internalAutowiredAnnotationProcessor,别名:[],bean对象:org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor@479d31f3  
bean名称:org.springframework.context.annotation.internalCommonAnnotationProcessor,别名:[],bean对象:org.springframework.context.annotation.CommonAnnotationBeanPostProcessor@40ef3420  
bean名称:org.springframework.context.event.internalEventListenerProcessor,别名:[],bean对象:org.springframework.context.event.EventListenerMethodProcessor@498d318c  
bean名称:org.springframework.context.event.internalEventListenerFactory,别名:[],bean对象:org.springframework.context.event.DefaultEventListenerFactory@6e171cd7  
bean名称:configBean1,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean1@402bba4f  
bean名称:user1,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@795cd85e  
bean名称:user2Bean,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@59fd97a8  
bean名称:user3Bean,别名:[user3BeanAlias2, user3BeanAlias1],bean对象:com.javacode2018.lesson001.demo20.User@f5ac9e4

##### 分析结果

我们将2个输出的最后4行拿来对比一下：

###### 有@Configuration注解的

bean名称:configBean,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean$$EnhancerBySpringCGLIB$$dde45976@5e5d171f  
bean名称:user1,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@24313fcc  
bean名称:user2Bean,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@7d20d0b  
bean名称:user3Bean,别名:[user3BeanAlias2, user3BeanAlias1],bean对象:com.javacode2018.lesson001.demo20.User@77f1baf5

###### 没有@Configuration注解的

bean名称:configBean1,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean1@402bba4f  
bean名称:user1,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@795cd85e  
bean名称:user2Bean,别名:[],bean对象:com.javacode2018.lesson001.demo20.User@59fd97a8  
bean名称:user3Bean,别名:[user3BeanAlias2, user3BeanAlias1],bean对象:com.javacode2018.lesson001.demo20.User@f5ac9e4

###### 对比得出

1. 对比最后3行，可以看出：**有没有@Configuration注解，@Bean都会起效，都会将@Bean修饰的方法作为bean注册到容器中**
2. 两个内容的第一行有点不一样，被@Configuration修饰的bean最后输出的时候带有EnhancerBySpringCGLIB的字样，而没有@Configuration注解的bean没有Cglib的字样；有EnhancerBySpringCGLIB字样的说明这个bean被cglib处理过的，变成了一个代理对象。

**目前为止我们还是看不出二者本质上的区别，继续向下看。**

### @Configuration加不加到底区别在哪？

通常情况下，bean之间是有依赖关系的，我们来创建个有依赖关系的bean，通过这个案例你就可以看出根本的区别了。

#### 再来一个加@Configuration的案例

定义2个类。

##### ServiceA

package com.javacode2018.lesson001.demo20;  
  
public class ServiceA {  
}

##### ServiceB

package com.javacode2018.lesson001.demo20;  
  
public class ServiceB {  
 private ServiceA serviceA;  
  
 public ServiceB(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
  
 @Override  
 public String toString() {  
 return "ServiceB{" +  
 "serviceA=" + serviceA +  
 '}';  
 }  
}

上面定义了2个类，ServiceB依赖于ServiceA，ServiceB通过构造器注入ServiceA。

来个@Configuration类管理上面对象。

##### ConfigBean2

package com.javacode2018.lesson001.demo20;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ConfigBean2 {  
  
 @Bean  
 public ServiceA serviceA() {  
 System.out.println("调用serviceA()方法"); //@0  
 return new ServiceA();  
 }  
  
 @Bean  
 ServiceB serviceB1() {  
 System.out.println("调用serviceB1()方法");  
 ServiceA serviceA = this.serviceA(); //@1  
 return new ServiceB(serviceA);  
 }  
  
 @Bean  
 ServiceB serviceB2() {  
 System.out.println("调用serviceB2()方法");  
 ServiceA serviceA = this.serviceA(); //@2  
 return new ServiceB(serviceA);  
 }  
  
}

上面通过@Bean注解，向容器中注册了3个bean

注意@1和@2，通过this.serviceA()获取需要注入的ServiceA对象。

上面每个方法第一行都输出了一行日志。

**重点关注一下@0这行日志会输出几次，大家先思考一下1次还是3次？**

##### 测试用例

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ConfigBean2.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 //别名  
 String[] aliases = context.getAliases(beanName);  
 System.out.println(String.format("bean名称:%s,别名:%s,bean对象:%s",  
 beanName,  
 Arrays.asList(aliases),  
 context.getBean(beanName)));  
 }  
}

##### 运行输出

截取了几行输出如下：

调用serviceA()方法  
调用serviceB1()方法  
调用serviceB2()方法  
bean名称:configBean2,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean2$$EnhancerBySpringCGLIB$$ffa0178@77f1baf5  
bean名称:serviceA,别名:[],bean对象:com.javacode2018.lesson001.demo20.ServiceA@41a2befb  
bean名称:serviceB1,别名:[],bean对象:ServiceB{serviceA=com.javacode2018.lesson001.demo20.ServiceA@41a2befb}  
bean名称:serviceB2,别名:[],bean对象:ServiceB{serviceA=com.javacode2018.lesson001.demo20.ServiceA@41a2befb}

##### 分析结果

从输出中可以看出

1. **前三行可以看出，被@Bean修饰的方法都只被调用了一次，这个很关键**
2. **最后三行中可以看出都是同一个ServiceA对象，都是ServiceA@41a2befb这个实例**

##### 这是为什么？

**被@Configuration修饰的类，spring容器中会通过cglib给这个类创建一个代理，代理会拦截所有被@Bean修饰的方法，默认情况（bean为单例）下确保这些方法只被调用一次，从而确保这些bean是同一个bean，即单例的。**

至于底层是如何实现的，可以去看一下公众号里面这篇文章：[详解java中的动态代理和cglib代理](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934082&idx=1&sn=c919886400135a0152da23eaa1f276c7&chksm=88621efcbf1597eab943b064147b8fb8fd3dfbac0dc03f41d15d477ef94b60d4e8f78c66b262&token=1042984313&lang=zh_CN" \l "rd)。

现在各位应该感受到了，我写的文章前后一般都是有依赖的，所以也建议大家按顺序看，这样知识就是贯通的。

#### 不加的案例

我们再来看看将ConfigBean2上的的@Configuration去掉，效果如何，代码就不写了，直接上输出结果：

调用serviceA()方法  
调用serviceB1()方法  
调用serviceA()方法  
调用serviceB2()方法  
调用serviceA()方法  
bean名称:configBean2,别名:[],bean对象:com.javacode2018.lesson001.demo20.ConfigBean2@6e171cd7  
bean名称:serviceA,别名:[],bean对象:com.javacode2018.lesson001.demo20.ServiceA@402bba4f  
bean名称:serviceB1,别名:[],bean对象:ServiceB{serviceA=com.javacode2018.lesson001.demo20.ServiceA@795cd85e}  
bean名称:serviceB2,别名:[],bean对象:ServiceB{serviceA=com.javacode2018.lesson001.demo20.ServiceA@59fd97a8}

##### 结果分析

1. **serviceA()方法被调用了3次**
2. **configBean2这个bean没有代理效果了**
3. **最后3行可以看出，几个ServiceA对象都是不一样的**

### spring这块的源码

spring中用下面这个类处理@Configuration这个注解：

org.springframework.context.annotation.ConfigurationClassPostProcessor

这里面重点关注这几个方法：

public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)   
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)  
public void enhanceConfigurationClasses(ConfigurableListableBeanFactory beanFactory)

最后一个方法会创建cglib代理，大家可以设置断点进去看看，有问题欢迎交流。

### 总结

1. **到目前为止加不加@Configuration注解，有什么区别，大家估计比我都清楚了**
2. **@Configuration注解修饰的类，会被spring通过cglib做增强处理，通过cglib会生成一个代理对象，代理会拦截所有被@Bean注解修饰的方法，可以确保一些bean是单例的**
3. **不管@Bean所在的类上是否有@Configuration注解，都可以将@Bean修饰的方法作为一个bean注册到spring容器中**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）

## Spring系列第18篇：@ComponentScan、@ComponentScans详解

### 灵魂拷问

1. @ComponentScan注解是做什么的？
2. basePackages的方式和basePackageClasses的方式有什么区别？你建议用哪个？为什么？
3. useDefaultFilters有什么用？
4. 常见的过滤器有哪些类型？说说你知道的几个
5. @ComponentScan是在哪个类中处理的？说一下大概的解析过程？

这些问题如果都ok，恭喜你，太优秀了，不知道没关系，一起来看看。

### 背景介绍

到目前为止，介绍了2种注册bean的方式：

1. xml中bean元素的方式
2. @Bean注解标注方法的方式

通常情况下，项目中大部分类都需要交给spring去管理，按照上面这2种方式，代码量还是挺大的。

为了更方便bean的注册，Spring提供了批量的方式注册bean，方便大量bean批量注册，spring中的@ComponentScan就是干这个事情的。

### @ComponentScan

@ComponentScan用于批量注册bean。

这个注解会让spring去扫描某些包及其子包中所有的类，然后将满足一定条件的类作为bean注册到spring容器容器中。

具体需要扫描哪些包？以及这些包中的类满足什么条件时被注册到容器中，这些都可以通过这个注解中的参数动态配置。

先来看一下这个注解的定义：

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Documented  
@Repeatable(ComponentScans.class) //@1  
public @interface ComponentScan {  
  
 @AliasFor("basePackages")  
 String[] value() default {};  
  
 @AliasFor("value")  
 String[] basePackages() default {};  
  
 Class<?>[] basePackageClasses() default {};  
  
 Class<? extends BeanNameGenerator> nameGenerator() default BeanNameGenerator.class;  
  
 Class<? extends ScopeMetadataResolver> scopeResolver() default AnnotationScopeMetadataResolver.class;  
  
 ScopedProxyMode scopedProxy() default ScopedProxyMode.DEFAULT;  
  
 String resourcePattern() default "\*\*/\*.class";  
  
 boolean useDefaultFilters() default true;  
  
 Filter[] includeFilters() default {};  
  
 Filter[] excludeFilters() default {};  
  
 boolean lazyInit() default false;  
}

定义上可以看出此注解可以用在任何类型上面，不过我们通常将其用在类上面。

常用参数：

value：指定需要扫描的包，如：com.javacode2018

basePackages：作用同value；value和basePackages不能同时存在设置，可二选一

basePackageClasses：指定一些类，spring容器会扫描这些类所在的包及其子包中的类

nameGenerator：自定义bean名称生成器

resourcePattern：需要扫描包中的那些资源，默认是：*\*/\*.class，即会扫描指定包中所有的class文件

useDefaultFilters：对扫描的类是否启用默认过滤器，默认为true

includeFilters：过滤器：用来配置被扫描出来的那些类会被作为组件注册到容器中

excludeFilters：过滤器，和includeFilters作用刚好相反，用来对扫描的类进行排除的，被排除的类不会被注册到容器中

lazyInit：是否延迟初始化被注册的bean

@1：@Repeatable(ComponentScans.class)，这个注解可以同时使用多个。

**@ComponentScan工作的过程：**

1. **Spring会扫描指定的包，且会递归下面子包，得到一批类的数组**
2. **然后这些类会经过上面的各种过滤器，最后剩下的类会被注册到容器中**

**所以玩这个注解，主要关注2个问题：**

**第一个：需要扫描哪些包？通过value、backPackages、basePackageClasses这3个参数来控制**

**第二：过滤器有哪些？通过useDefaultFilters、includeFilters、excludeFilters这3个参数来控制过滤器**

**这两个问题搞清楚了，就可以确定哪些类会被注册到容器中。**

**默认情况下，任何参数都不设置的情况下，此时，会将@ComponentScan修饰的类所在的包作为扫描包；默认情况下useDefaultFilters为true，这个为true的时候，spring容器内部会使用默认过滤器，规则是：凡是类上有@Repository、@Service、@Controller、@Component这几个注解中的任何一个的，那么这个类就会被作为bean注册到spring容器中，所以默认情况下，只需在类上加上这几个注解中的任何一个，这些类就会自动交给spring容器来管理了。**

### @Component、@Repository、@Service、@Controller

这几个注解都是spring提供的。

先说一下@Component这个注解，看一下其定义：

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Indexed  
public @interface Component {  
 String value() default "";  
}

从定义中可以看出，这个注解可以用在任何类型上面。

通常情况下将这个注解用在类上面，标注这个类为一个组件，默认情况下，被扫描的时候会被作为bean注册到容器中。

value参数：被注册为bean的时候，用来指定bean的名称，如果不指定，默认为类名首字母小写。如：类UserService对应的beanname为userService

再来看看@Repository源码如下：

@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Component  
public @interface Repository {  
  
 @AliasFor(annotation = Component.class)  
 String value() default "";  
  
}

Repository上面有@Component注解。

value参数上面有@AliasFor(annotation = Component.class)，设置value参数的时候，也相当于给@Component注解中的value设置值。

其他两个注解@Service、@Controller源码和@Repository源码类似。

这4个注解本质上是没有任何差别，都可以用在类上面，表示这个类被spring容器扫描的时候，可以作为一个bean组件注册到spring容器中。

spring容器中对这4个注解的解析并没有进行区分，统一采用@Component注解的方式进行解析，所以这几个注解之间可以相互替换。

spring提供这4个注解，是为了让系统更清晰，通常情况下，系统是分层结构的，多数系统一般分为controller层、service层、dao层。

@controller通常用来标注controller层组件，@service注解标注service层的组件，@Repository标注dao层的组件，这样可以让整个系统的结构更清晰，当看到这些注解的时候，会和清晰的知道属于哪个层，对于spring来说，将这3个注解替换成@Component注解，对系统没有任何影响，产生的效果是一样的。

下面通过案例来感受@ComponentScan各种用法。

### 案例1：任何参数未设置

##### UserController

package com.javacode2018.lesson001.demo22.test1.controller;  
  
import org.springframework.stereotype.Controller;  
  
@Controller  
public class UserController {  
}

##### UserService

package com.javacode2018.lesson001.demo22.test1.service;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserService {  
}

##### UserDao

package com.javacode2018.lesson001.demo22.test1.dao;  
  
import org.springframework.stereotype.Repository;  
  
@Repository  
public class UserDao {  
}

##### UserModel

package com.javacode2018.lesson001.demo22.test1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserModel {  
}

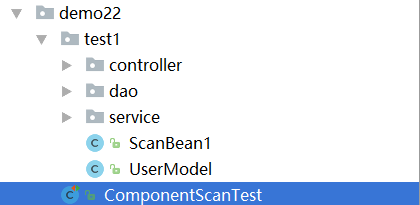
上面几个类中，分别使用了4种注解。

##### @CompontentScan修饰的类

package com.javacode2018.lesson001.demo22.test1;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class ScanBean1 {  
}

##### 上面几个类的结构图

fsdfds



##### 测试用例

package com.javacode2018.lesson001.demo22;  
  
import com.javacode2018.lesson001.demo22.test1.ScanBean1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class ComponentScanTest {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean1.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
 }  
}

@1：使用AnnotationConfigApplicationContext作为ioc容器，将ScanBean作为参数传入。

默认会扫描ScanBean类所在的包中的所有类，类上有@Component、@Repository、@Service、@Controller任何一个注解的都会被注册到容器中

##### 运行输出

部分输出如下：

userModel->com.javacode2018.lesson001.demo22.test1.UserModel@595b007d  
userController->com.javacode2018.lesson001.demo22.test1.controller.UserController@72d1ad2e  
userDao->com.javacode2018.lesson001.demo22.test1.dao.UserDao@2d7275fc  
userService->com.javacode2018.lesson001.demo22.test1.service.UserService@399f45b1

注意最后4行这几个bean，都被注册成功了。

### 案例2：指定需要扫描的包

指定需要扫毛哪些包，可以通过value或者basePackage来配置，二者选其一，都配置运行会报错，下面我们通过value来配置。

##### ScanBean2

package com.javacode2018.lesson001.demo22.test2;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan({  
 "com.javacode2018.lesson001.demo22.test1.controller",  
 "com.javacode2018.lesson001.demo22.test1.service"  
})  
public class ScanBean2 {  
}

上面指定了2需要扫描的包，这两个包中有2个类。

##### 测试用例

ComponentScanTest中新增个方法

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean2.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
}

##### 运行输出

截取了关键几行如下：

userController->com.javacode2018.lesson001.demo22.test1.controller.UserController@dd8ba08  
userService->com.javacode2018.lesson001.demo22.test1.service.UserService@245b4bdc

可以看出只有controller包和service包中的2个类被注册为bean了。

##### 注意

**指定包名的方式扫描存在的一个隐患，若包被重名了，会导致扫描会失效，一般情况下面我们使用basePackageClasses的方式来指定需要扫描的包，这个参数可以指定一些类型，默认会扫描这些类所在的包及其子包中所有的类，这种方式可以有效避免这种问题。**

下面来看一下basePackageClasses的方式。

### 案例：basePackageClasses指定扫描范围

我们可以在需要扫描的包中定义一个标记的接口或者类，他们的唯一的作用是作为basePackageClasses的值，其他没有任何用途。

##### 下面我们定义这样一个接口

package com.javacode2018.lesson001.demo22.test6.beans;  
  
public interface ScanClass {  
}

##### 再来定义2个类，用@Component注解标记

package com.javacode2018.lesson001.demo22.test6.beans;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

package com.javacode2018.lesson001.demo22.test6.beans;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
}

##### 来一个@CompontentScan标记的类

package com.javacode2018.lesson001.demo22.test6;  
  
import com.javacode2018.lesson001.demo22.test6.beans.ScanClass;  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan(basePackageClasses = ScanClass.class)  
public class ScanBean6 {  
}

##### 测试用例

ComponentScanTest中新增个方法

@Test  
public void test6() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean6.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo22.test6.beans.Service1@79924b  
service2->com.javacode2018.lesson001.demo22.test6.beans.Service2@7b9a4292

### includeFilters的使用

#### 用法

再来看一下includeFilters这个参数的定义：

Filter[] includeFilters() default {};

是一个Filter类型的数组，**多个Filter之间为或者关系，即满足任意一个就可以了**，看一下Filter的代码：

@Retention(RetentionPolicy.RUNTIME)  
@Target({})  
@interface Filter {  
  
 FilterType type() default FilterType.ANNOTATION;  
  
 @AliasFor("classes")  
 Class<?>[] value() default {};  
  
 @AliasFor("value")  
 Class<?>[] classes() default {};  
  
 String[] pattern() default {};  
  
}

可以看出Filter也是一个注解，参数：

**type：过滤器的类型，是个枚举类型，5种类型**

ANNOTATION：通过注解的方式来筛选候选者，即判断候选者是否有指定的注解

ASSIGNABLE\_TYPE：通过指定的类型来筛选候选者，即判断候选者是否是指定的类型

ASPECTJ：ASPECTJ表达式方式，即判断候选者是否匹配ASPECTJ表达式

REGEX：正则表达式方式，即判断候选者的完整名称是否和正则表达式匹配

CUSTOM：用户自定义过滤器来筛选候选者，对候选者的筛选交给用户自己来判断

**value：和参数classes效果一样，二选一**

**classes：3种情况如下**

当type=FilterType.ANNOTATION时，通过classes参数可以指定一些注解，用来判断被扫描的类上是否有classes参数指定的注解

当type=FilterType.ASSIGNABLE\_TYPE时，通过classes参数可以指定一些类型，用来判断被扫描的类是否是classes参数指定的类型

当type=FilterType.CUSTOM时，表示这个过滤器是用户自定义的，classes参数就是用来指定用户自定义的过滤器，自定义的过滤器需要实现org.springframework.core.type.filter.TypeFilter接口

**pattern：2种情况如下**

当type=FilterType.ASPECTJ时，通过pattern来指定需要匹配的ASPECTJ表达式的值

当type=FilterType.REGEX时，通过pattern来自正则表达式的值

#### 案例：扫描包含注解的类

##### 需求

我们自定义一个注解，让标注有这些注解的类自动注册到容器中

##### 代码实现

下面的代码都在com.javacode2018.lesson001.demo22.test3包中。

###### 定义一个注解

package com.javacode2018.lesson001.demo22.test3;  
  
import java.lang.annotation.\*;  
  
@Documented  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface MyBean {  
}

###### 创建一个类，使用这个注解标注

package com.javacode2018.lesson001.demo22.test3;  
  
@MyBean  
public class Service1 {  
}

###### 再来一个类，使用spring中的@Compontent标注

package com.javacode2018.lesson001.demo22.test3;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
}

###### 再来一个类，使用@CompontentScan标注

package com.javacode2018.lesson001.demo22.test3;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.FilterType;  
  
@ComponentScan(includeFilters = {  
 @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = MyBean.class)  
})  
public class ScanBean3 {  
}

上面指定了Filter的type为注解的类型，只要类上面有@MyBean注解的，都会被作为bean注册到容器中。

###### 测试用例

ComponentScanTest中新增个测试用例

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean3.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
}

###### 运行输出，截取了主要的几行

service1->com.javacode2018.lesson001.demo22.test3.Service1@6b81ce95  
service2->com.javacode2018.lesson001.demo22.test3.Service2@2a798d51

**Service1上标注了@MyBean注解，被注册到容器了，但是Service2上没有标注@MyBean啊，怎么也被注册到容器了？**

**原因：Service2上标注了@Compontent注解，而@CompontentScan注解中的useDefaultFilters默认是true，表示也会启用默认的过滤器，而默认的过滤器会将标注有@Component、@Repository、@Service、@Controller这几个注解的类也注册到容器中**

如果我们只想将标注有@MyBean注解的bean注册到容器，需要将默认过滤器关闭，即：useDefaultFilters=false，我们修改一下ScanBean3的代码如下：

@ComponentScan(  
 useDefaultFilters = false, //不启用默认过滤器  
 includeFilters = {  
 @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = MyBean.class)  
 })  
public class ScanBean3 {  
}

再次运行test3输出：

service1->com.javacode2018.lesson001.demo22.test3.Service1@294425a7

##### 扩展：自定义注解支持定义bean名称

上面的自定义的@MyBean注解，是无法指定bean的名称的，可以对这个注解做一下改造，加个value参数来指定bean的名称，如下：

@Documented  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Component //@1  
public @interface MyBean {  
 @AliasFor(annotation = Component.class) //@2  
 String value() default ""; //@3  
}

重点在于@1和@2这2个地方的代码，通过上面的参数可以间接给@Component注解中的value设置值。

这块用到了@AliasFor注解，对这块不了解的，可以去看一下：[java注解详解及spring对注解的增强](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934137&idx=1&sn=3775d5d7a23c43616d1274b0b52a9f99&chksm=88621ec7bf1597d1b16d91cfb28e63bef485f10883c7ca30d09838667f65e3d214b9e1cebd47&token=1372043037&lang=zh_CN" \l "rd)

修改一下Service1的代码：

@MyBean("service1Bean")  
public class Service1 {  
}

运行test3用例输出：

service1Bean->com.javacode2018.lesson001.demo22.test3.Service1@222545dc

此时bean名称就变成了service1Bean。

#### 案例：包含指定类型的类

下面的代码都位于com.javacode2018.lesson001.demo22.test4包中。

##### 来个接口

package com.javacode2018.lesson001.demo22.test4;  
  
public interface IService {  
}

让spring来进行扫描，类型满足IService的都将其注册到容器中。

##### 来2个实现类

package com.javacode2018.lesson001.demo22.test4;  
  
public class Service1 implements IService {  
}

package com.javacode2018.lesson001.demo22.test4;  
  
public class Service2 implements IService {  
}

##### 来一个@CompontentScan标注的类

package com.javacode2018.lesson001.demo22.test4;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.FilterType;  
  
@ComponentScan(  
 useDefaultFilters = false, //不启用默认过滤器  
 includeFilters = {  
 @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, classes = IService.class) //@1  
 })  
public class ScanBean4 {  
}

@1：被扫描的类满足IService.class.isAssignableFrom(被扫描的类)条件的都会被注册到spring容器中

##### 来个测试用例

ComponentScanTest中新增个测试用例

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean4.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo22.test4.Service1@6379eb  
service2->com.javacode2018.lesson001.demo22.test4.Service2@294425a7

### 自定义Filter

#### 用法

有时候我们需要用到自定义的过滤器，使用自定义过滤器的步骤：

1.设置@Filter中type的类型为：FilterType.CUSTOM  
2.自定义过滤器类，需要实现接口：org.springframework.core.type.filter.TypeFilter  
3.设置@Filter中的classses为自定义的过滤器类型

来看一下TypeFilter这个接口的定义：

@FunctionalInterface  
public interface TypeFilter {  
 boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory)  
 throws IOException;  
}

是一个函数式接口，包含一个match方法，方法返回boolean类型，有2个参数，都是接口类型的，下面介绍一下这2个接口。

##### MetadataReader接口

**类元数据读取器，可以读取一个类上的任意信息，如类上面的注解信息、类的磁盘路径信息、类的class对象的各种信息，spring进行了封装，提供了各种方便使用的方法。**

看一下这个接口的定义：

public interface MetadataReader {  
  
 /\*\*  
 \* 返回类文件的资源引用  
 \*/  
 Resource getResource();  
  
 /\*\*  
 \* 返回一个ClassMetadata对象，可以通过这个读想获取类的一些元数据信息，如类的class对象、是否是接口、是否有注解、是否是抽象类、父类名称、接口名称、内部包含的之类列表等等，可以去看一下源码  
 \*/  
 ClassMetadata getClassMetadata();  
  
 /\*\*  
 \* 获取类上所有的注解信息  
 \*/  
 AnnotationMetadata getAnnotationMetadata();  
  
}

##### MetadataReaderFactory接口

**类元数据读取器工厂，可以通过这个类获取任意一个类的MetadataReader对象。**

源码：

public interface MetadataReaderFactory {  
  
 /\*\*  
 \* 返回给定类名的MetadataReader对象  
 \*/  
 MetadataReader getMetadataReader(String className) throws IOException;  
  
 /\*\*  
 \* 返回指定资源的MetadataReader对象  
 \*/  
 MetadataReader getMetadataReader(Resource resource) throws IOException;  
  
}

#### 自定义Filter案例

##### 需求

我们来个自定义的Filter，判断被扫描的类如果是IService接口类型的，就让其注册到容器中。

##### 代码实现

来个自定义的TypeFilter类：

package com.javacode2018.lesson001.demo22.test5;  
  
import com.javacode2018.lesson001.demo22.test4.IService;  
import org.springframework.core.type.ClassMetadata;  
import org.springframework.core.type.classreading.MetadataReader;  
import org.springframework.core.type.classreading.MetadataReaderFactory;  
import org.springframework.core.type.filter.TypeFilter;  
  
import java.io.IOException;  
  
public class MyFilter implements TypeFilter {  
 /\*\*  
 \* @param metadataReader  
 \* @param metadataReaderFactory  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {  
 Class curClass = null;  
 try {  
 //当前被扫描的类  
 curClass = Class.forName(metadataReader.getClassMetadata().getClassName());  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 //判断curClass是否是IService类型  
 boolean result = IService.class.isAssignableFrom(curClass);  
 return result;  
 }  
}

##### 来一个@CompontentScan标注的类

package com.javacode2018.lesson001.demo22.test5;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.FilterType;  
  
@ComponentScan(  
 basePackages = {"com.javacode2018.lesson001.demo22.test4"},  
 useDefaultFilters = false, //不启用默认过滤器  
 includeFilters = {  
 @ComponentScan.Filter(type = FilterType.CUSTOM, classes = MyFilter.class) //@1  
 })  
public class ScanBean5 {  
}

@1：type为FilterType.CUSTOM，表示Filter是用户自定义的，classes为自定义的过滤器

##### 再来个测试用例

ComponentScanTest中新增个测试用例

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ScanBean5.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(beanName + "->" + context.getBean(beanName));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo22.test4.Service1@4cc451f2  
service2->com.javacode2018.lesson001.demo22.test4.Service2@6379eb

### excludeFilters

**配置排除的过滤器，满足这些过滤器的类不会被注册到容器中，用法上面和includeFilters用一样，这个我就不演示了，可以自己玩玩**

### @ComponentScan重复使用

从这个注解的定义上可以看出这个注解可以同时使用多个，如：

@ComponentScan(basePackageClasses = ScanClass.class)  
@ComponentScan(  
 useDefaultFilters = false, //不启用默认过滤器  
 includeFilters = {  
 @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, classes = IService.class)  
 })  
public class ScanBean7 {  
}

还有一种写法，使用@ComponentScans的方式：

@ComponentScans({  
 @ComponentScan(basePackageClasses = ScanClass.class),  
 @ComponentScan(  
 useDefaultFilters = false, //不启用默认过滤器  
 includeFilters = {  
 @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, classes = IService.class)  
 })})  
public class ScanBean7 {  
}

### Spring中这块的源码

**@CompontentScan注解是被下面这个类处理的**

org.springframework.context.annotation.ConfigurationClassPostProcessor

**这个类非常非常关键，主要用户bean的注册，前面我们介绍的@Configuration,@Bean注解也是被这个类处理的。**

还有下面这些注解：

@PropertySource  
@Import  
@ImportResource  
@Compontent

**以上这些注解都是被ConfigurationClassPostProcessor这个类处理的，内部会递归处理这些注解，完成bean的注册。**

**以@CompontentScan来说一下过程，第一次扫描之后会得到一批需要注册的类，然后会对这些需要注册的类进行遍历，判断是否有上面任意一个注解，如果有，会将这个类交给ConfigurationClassPostProcessor继续处理，直到递归完成所有bean的注册。**

**想成为高手，这个类是必看的。**

### 总结

1. **@ComponentScan用于批量注册bean，spring会按照这个注解的配置，递归扫描指定包中的所有类，将满足条件的类批量注册到spring容器中**
2. **可以通过value、basePackages、basePackageClasses 这几个参数来配置包的扫描范围**
3. **可以通过useDefaultFilters、includeFilters、excludeFilters这几个参数来配置类的过滤器，被过滤器处理之后剩下的类会被注册到容器中**
4. **指定包名的方式配置扫描范围存在隐患，包名被重命名之后，会导致扫描实现，所以一般我们在需要扫描的包中可以创建一个标记的接口或者类，作为basePackageClasses的值，通过这个来控制包的扫描范围**
5. **@CompontScan注解会被ConfigurationClassPostProcessor类递归处理，最终得到所有需要注册的类。**

### 案例源码

链接：https://pan.baidu.com/s/1p6rcfKOeWQIVkuhVybzZmQ   
提取码：zr99

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(Bean注册)

## Spring系列第19篇：@Import批量注册bean

### 先看几个问题

1. @Import你用过么？是做什么的？
2. @Import使用有几种方式？有何区别？
3. DeferredImportSelector是做什么的？他和ImportSelector有什么区别？
4. 可以介绍介绍一下spring中哪些功能是通过@Import来实现的？
5. 可以介绍一下spring中是如何解析@Import注解的么？

### @Import出现的背景

目前为止，注解的方式批量注册bean，前面2篇文章中，我们介绍了2种方式：

到目前，我们知道的批量定义bean的方式有2种：

1. @Configuration结合@Bean注解的方式
2. @CompontentScan扫描包的方式

下面我们来看几个问题。

#### 问题1

如果需要注册的类是在第三方的jar中，那么我们如果想注册这些bean有2种方式：

1. 通过@Bean标注方法的方式，一个个来注册
2. @CompontentScan的方式：默认的@CompontentScan是无能为力的，默认情况下只会注册@Compontent标注的类，此时只能自定义@CompontentScan中的过滤器来实现了

这2种方式都不是太好，每次有变化，调整的代码都比较多。

#### 问题2

通常我们的项目中有很多子模块，可能每个模块都是独立开发的，最后通过jar的方式引进来，每个模块中都有各自的@Configuration、@Bean标注的类，或者使用@CompontentScan标注的类，**被@Configuration、@Bean、@CompontentScan标注的类，我们统称为bean配置类，配置类可以用来注册bean**，此时如果我们只想使用其中几个模块的配置类，怎么办？

@Import可以很好的解决这2个问题，下面我们来看@Import怎么玩的。

### @Import使用

先看Spring对它的注释，总结下来作用就是和xml配置的 <import />标签作用一样，允许通过它引入@Configuration标注的类 ， 引入ImportSelector接口和ImportBeanDefinitionRegistrar接口的实现，也包括 @Component注解的普通类。

**总的来说：@Import可以用来批量导入需要注册的各种类，如普通的类、配置类，完后完成普通类和配置类中所有bean的注册。**

@Import的源码：

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Import {  
  
 /\*\*  
 \* {@link Configuration @Configuration}, {@link ImportSelector},  
 \* {@link ImportBeanDefinitionRegistrar}, or regular component classes to import.  
 \*/  
 Class<?>[] value();  
  
}

@Import可以使用在任何类型上，通常情况下，类和注解上用的比较多。

value：一个Class数组，设置需要导入的类，可以是@Configuration标注的列，可以是ImportSelector接口或者ImportBeanDefinitionRegistrar接口类型的，或者需要导入的普通组件类。

#### 使用步骤

1. 将@Import标注在类上，设置value参数
2. 将@Import标注的类作为AnnotationConfigApplicationContext构造参数创建AnnotationConfigApplicationContext对象
3. 使用AnnotationConfigApplicationContext对象

#### @Import的value常见的有5种用法

1. **value为普通的类**
2. **value为@Configuration标注的类**
3. **value为@CompontentScan标注的类**
4. **value为ImportBeanDefinitionRegistrar接口类型**
5. **value为ImportSelector接口类型**
6. **value为DeferredImportSelector接口类型**

下面我们通过案例来一个个详细介绍。

### value为普通的类

来2个类

#### Service1

package com.javacode2018.lesson001.demo24.test1;  
  
public class Service1 {  
}

#### Service2

package com.javacode2018.lesson001.demo24.test1;  
  
public class Service2 {  
}

#### 总配置类：使用@Import标注

package com.javacode2018.lesson001.demo24.test1;  
  
import org.springframework.context.annotation.Import;  
  
@Import({Service1.class, Service2.class})  
public class MainConfig1 {  
}

@Import中导入了2个普通的类：Service1、Service2，这两个类会被自动注册到容器中

#### 测试用例

package com.javacode2018.lesson001.demo24;  
  
import com.javacode2018.lesson001.demo24.test1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class ImportTest {  
 @Test  
 public void test1() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig1.class);  
 //2.输出容器中定义的所有bean信息  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
 }  
}

#### 运行输出

部分输出如下：

com.javacode2018.lesson001.demo24.test1.Service1->com.javacode2018.lesson001.demo24.test1.Service1@7e0b85f9  
com.javacode2018.lesson001.demo24.test1.Service2->com.javacode2018.lesson001.demo24.test1.Service2@63355449

#### 结果分析

从输出中可以看出：

1. Service1和Service2成功注册到容器了。
2. 通过@Import导入的2个类，bean名称为完整的类名

我们也可以指定被导入类的bean名称，使用@Compontent注解就可以了，如下：

@Component("service1")  
public class Service1 {  
}

再次运行test1输出：

service1->com.javacode2018.lesson001.demo24.test1.Service1@45efd90f

#### 总结一下

**按模块的方式进行导入，需要哪个导入哪个，不需要的时候，直接修改一下总的配置类，调整一下@Import就可以了，非常方便。**

### value为@Configuration标注的配置类

项目比较大的情况下，会按照模块独立开发，每个模块在maven中就表现为一个个的构建，然后通过坐标的方式进行引入需要的模块。

假如项目中有2个模块，2个模块都有各自的配置类，如下

#### 模块1配置类

package com.javacode2018.lesson001.demo24.test2;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
/\*\*  
 \* 模块1配置类  
 \*/  
@Configuration  
public class ConfigModule1 {  
 @Bean  
 public String module1() {  
 return "我是模块1配置类！";  
 }  
}

#### 模块2配置类

package com.javacode2018.lesson001.demo24.test2;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
/\*\*  
 \* 模块2配置类  
 \*/  
@Configuration  
public class ConfigModule2 {  
 @Bean  
 public String module2() {  
 return "我是模块2配置类！";  
 }  
}

#### 总配置类：通过@Import导入2个模块的配置类

package com.javacode2018.lesson001.demo24.test2;  
  
import org.springframework.context.annotation.Import;  
  
/\*\*  
 \* 通过Import来汇总多个@Configuration标注的配置类  
 \*/  
@Import({ConfigModule1.class, ConfigModule2.class}) //@1  
public class MainConfig2 {  
}

@1导入了2个模块中的模块配置类，可以按需导入。

#### 测试用例

ImportTest中新增个方法

@Test  
public void test2() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig2.class);  
 //2.输出容器中定义的所有bean信息  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

#### 运行输出

mainConfig2->com.javacode2018.lesson001.demo24.test2.MainConfig2@ba2f4ec  
com.javacode2018.lesson001.demo24.test2.ConfigModule1->com.javacode2018.lesson001.demo24.test2.ConfigModule1$$EnhancerBySpringCGLIB$$700e65cd@1c1bbc4e  
module1->我是模块1配置类！  
com.javacode2018.lesson001.demo24.test2.ConfigModule2->com.javacode2018.lesson001.demo24.test2.ConfigModule2$$EnhancerBySpringCGLIB$$a87108ee@55fe41ea  
module2->我是模块2配置类！

### value为@CompontentScan标注的类

项目中分多个模块，每个模块有各自独立的包，我们在每个模块所在的包中配置一个@CompontentScan类，然后通过@Import来导入需要启用的模块。

#### 定义模块1

2个组件和一个组件扫描类，模块1所有类所在的包为：

com.javacode2018.lesson001.demo24.test3.module1

##### 组件1：Module1Service1

package com.javacode2018.lesson001.demo24.test3.module1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Module1Service1 {  
}

##### 组件2：Module1Service2

package com.javacode2018.lesson001.demo24.test3.module1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Module1Service2 {  
}

##### 组件扫描类：CompontentScanModule1

负责扫描当前模块中的组件

package com.javacode2018.lesson001.demo24.test3.module1;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 模块1的主键扫描  
 \*/  
@ComponentScan  
public class CompontentScanModule1 {  
}

#### 同样的方式定义模块2

2个组件和一个组件扫描类，模块1所有类所在的包为：

com.javacode2018.lesson001.demo24.test3.module2

##### 组件1：Module2Service1

package com.javacode2018.lesson001.demo24.test3.module2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Module2Service1 {  
}

##### 组件2：Module2Service2

package com.javacode2018.lesson001.demo24.test3.module2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Module2Service2 {  
}

##### 组件扫描类：CompontentScanModule1

负责扫描当前模块中的组件

package com.javacode2018.lesson001.demo24.test3.module2;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 模块2的组件扫描  
 \*/  
@ComponentScan  
public class CompontentScanModule2 {  
}

#### 总配置类：通过@Import导入每个模块中的组件扫描类

package com.javacode2018.lesson001.demo24.test3;  
  
import com.javacode2018.lesson001.demo24.test3.module1.CompontentScanModule1;  
import com.javacode2018.lesson001.demo24.test3.module2.CompontentScanModule2;  
import org.springframework.context.annotation.Import;  
  
/\*\*  
 \* 通过@Import导入多个@CompontentScan标注的配置类  
 \*/  
@Import({CompontentScanModule1.class, CompontentScanModule2.class}) //@1  
public class MainConfig3 {  
}

@1导入了2个模块中的组件扫描类，可以按需导入。

#### 测试用例

ImportTest中新增个方法

@Test  
public void test3() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig3.class);  
 //2.输出容器中定义的所有bean信息  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

#### 运行输出

部分输出如下：

module1Service1->com.javacode2018.lesson001.demo24.test3.module1.Module1Service1@5b239d7d  
module1Service2->com.javacode2018.lesson001.demo24.test3.module1.Module1Service2@6572421  
module2Service1->com.javacode2018.lesson001.demo24.test3.module2.Module2Service1@6b81ce95  
module2Service2->com.javacode2018.lesson001.demo24.test3.module2.Module2Service2@2a798d51

两个模块中通过@Compontent定义的4个bean都输出了。

如果只想注册模块1中的bean，只需要修改一下@Import，去掉CompontentScanModule2，如下：

@Import({CompontentScanModule1.class})

再次运行输出：

module1Service1->com.javacode2018.lesson001.demo24.test3.module1.Module1Service1@6379eb  
module1Service2->com.javacode2018.lesson001.demo24.test3.module1.Module1Service2@294425a7

此时模块2的bean就没有了。

### 先来了解一下相关的几个接口

#### ImportBeanDefinitionRegistrar接口

**这个接口提供了通过spring容器api的方式直接向容器中注册bean**。

接口的完整名称：

org.springframework.context.annotation.ImportBeanDefinitionRegistrar

源码：

public interface ImportBeanDefinitionRegistrar {  
  
 default void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry,  
 BeanNameGenerator importBeanNameGenerator) {  
  
 registerBeanDefinitions(importingClassMetadata, registry);  
 }  
  
 default void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
 }  
  
}

2个默认方法，都可以用来调用spring容器api来注册bean。

##### 2个方法中主要有3个参数

###### importingClassMetadata

AnnotationMetadata类型的，通过这个可以获取被@Import注解标注的类所有注解的信息。

###### registry

BeanDefinitionRegistry类型，是一个接口，内部提供了注册bean的各种方法。

###### importBeanNameGenerator

BeanNameGenerator类型，是一个接口，内部有一个方法，用来生成bean的名称。

关于BeanDefinitionRegistry和BeanNameGenerator接口在来细说一下。

#### BeanDefinitionRegistry接口：bean定义注册器

bean定义注册器，提供了bean注册的各种方法，来看一下源码：

public interface BeanDefinitionRegistry extends AliasRegistry {  
  
 /\*\*  
 \* 注册一个新的bean定义  
 \* beanName：bean的名称  
 \* beanDefinition：bean定义信息  
 \*/  
 void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  
 throws BeanDefinitionStoreException;  
  
 /\*\*  
 \* 通过bean名称移除已注册的bean  
 \* beanName：bean名称  
 \*/  
 void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;  
  
 /\*\*  
 \* 通过名称获取bean的定义信息  
 \* beanName：bean名称  
 \*/  
 BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;  
  
 /\*\*  
 \* 查看beanName是否注册过  
 \*/  
 boolean containsBeanDefinition(String beanName);  
  
 /\*\*  
 \* 获取已经定义（注册）的bean名称列表  
 \*/  
 String[] getBeanDefinitionNames();  
  
 /\*\*  
 \* 返回注册器中已注册的bean数量  
 \*/  
 int getBeanDefinitionCount();  
  
 /\*\*  
 \* 确定给定的bean名称或者别名是否已在此注册表中使用  
 \* beanName：可以是bean名称或者bean的别名  
 \*/  
 boolean isBeanNameInUse(String beanName);  
  
}

基本上所有bean工厂都实现了这个接口，让bean工厂拥有bean注册的各种能力。

上面我们用到的AnnotationConfigApplicationContext类也实现了这个接口。

#### BeanNameGenerator接口：bean名称生成器

bean名称生成器，这个接口只有一个方法，用来生成bean的名称：

public interface BeanNameGenerator {  
 String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry registry);  
}

spring内置了3个实现

##### DefaultBeanNameGenerator

默认bean名称生成器，xml中bean未指定名称的时候，默认就会使用这个生成器，默认为：完整的类名#bean编号

##### AnnotationBeanNameGenerator

注解方式的bean名称生成器，比如通过@Component(bean名称)的方式指定bean名称，如果没有通过注解方式指定名称，默认会将完整的类名作为bean名称。

##### FullyQualifiedAnnotationBeanNameGenerator

将完整的类名作为bean的名称

#### BeanDefinition接口：bean定义信息

用来表示bean定义信息的接口，我们向容器中注册bean之前，会通过xml或者其他方式定义bean的各种配置信息，bean的所有配置信息都会被转换为一个BeanDefinition对象，然后通过容器中BeanDefinitionRegistry接口中的方法，将BeanDefinition注册到spring容器中，完成bean的注册操作。

这个接口有很多实现类，有兴趣的可以去看看源码，BeanDefinition的各种用法，以后会通过专题细说。

### value为ImportBeanDefinitionRegistrar接口类型

#### 用法（4个步骤）

1. 定义ImportBeanDefinitionRegistrar接口实现类，在registerBeanDefinitions方法中使用registry来注册bean  
2. 使用@Import来导入步骤1中定义的类  
3. 使用步骤2中@Import标注的类作为AnnotationConfigApplicationContext构造参数创建spring容器  
4. 使用AnnotationConfigApplicationContext操作bean

#### 案例

来2个普通的类。

##### Service1

package com.javacode2018.lesson001.demo24.test4;  
  
public class Service1 {  
}

##### Service2

这个类中需要注入Service1

package com.javacode2018.lesson001.demo24.test4;  
  
public class Service2 {  
 private Service1 service1;  
  
 public Service1 getService1() {  
 return service1;  
 }  
  
 public void setService1(Service1 service1) {  
 this.service1 = service1;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

来个类实现ImportBeanDefinitionRegistrar接口，然后在里面实现上面2个类的注册，如下：

##### MyImportBeanDefinitionRegistrar

package com.javacode2018.lesson001.demo24.test4;  
  
import org.springframework.beans.factory.config.BeanDefinition;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.BeanDefinitionRegistry;  
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;  
import org.springframework.core.type.AnnotationMetadata;  
  
  
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {  
 @Override  
 public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
 //定义一个bean：Service1  
 BeanDefinition service1BeanDinition = BeanDefinitionBuilder.genericBeanDefinition(Service1.class).getBeanDefinition();  
 //注册bean  
 registry.registerBeanDefinition("service1", service1BeanDinition);  
  
 //定义一个bean：Service2，通过addPropertyReference注入service1  
 BeanDefinition service2BeanDinition = BeanDefinitionBuilder.genericBeanDefinition(Service2.class).  
 addPropertyReference("service1", "service1").  
 getBeanDefinition();  
 //注册bean  
 registry.registerBeanDefinition("service2", service2BeanDinition);  
 }  
}

注意上面的registerBeanDefinitions方法，内部注册了2个bean，Service1和Service2。

上面使用了BeanDefinitionBuilder这个类，这个是BeanDefinition的构造器，内部提供了很多静态方法方便构建BeanDefinition对象。

上面定义的2个bean，和下面xml方式效果一样：

<bean id="service1" class="com.javacode2018.lesson001.demo24.test4.Service1" />  
<bean id="service2" class="com.javacode2018.lesson001.demo24.test4.Service2">  
 <property name="service1" ref="service1"/>  
</bean>

##### 来个测试用例

ImportTest中新增个方法

@Test  
public void test4() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig4.class);  
 //2.输出容器中定义的所有bean信息  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo24.test4.Service1@62150f9e  
service2->Service2{service1=com.javacode2018.lesson001.demo24.test4.Service1@62150f9e}

### value为ImportSelector接口类型

先来看一下ImportSelector接口

#### ImportSelector接口

导入选择器，看一下源码：

public interface ImportSelector {  
  
 /\*\*  
 \* 返回需要导入的类名的数组，可以是任何普通类，配置类（@Configuration、@Bean、@CompontentScan等标注的类）  
 \* @importingClassMetadata：用来获取被@Import标注的类上面所有的注解信息  
 \*/  
 String[] selectImports(AnnotationMetadata importingClassMetadata);  
  
}

#### 用法（4个步骤）

1. 定义ImportSelector接口实现类，在selectImports返回需要导入的类的名称数组  
2. 使用@Import来导入步骤1中定义的类  
3. 使用步骤2中@Import标注的类作为AnnotationConfigApplicationContext构造参数创建spring容器  
4. 使用AnnotationConfigApplicationContext操作bean

#### 案例

##### 来个普通类：Service1

package com.javacode2018.lesson001.demo24.test5;  
  
public class Service1 {  
}

##### 来个@Configuration标注的配置类：Module1Config

package com.javacode2018.lesson001.demo24.test5;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Module1Config {  
 @Bean  
 public String name() {  
 return "公众号：马士兵老师";  
 }  
  
 @Bean  
 public String address() {  
 return "上海市";  
 }  
}

上面定义了两个string类型的bean：name和address

##### 下面自定义一个ImportSelector，然后返回上面2个类的名称

package com.javacode2018.lesson001.demo24.test5;  
  
import org.springframework.context.annotation.ImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class MyImportSelector implements ImportSelector {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{  
 Service1.class.getName(),  
 Module1Config.class.getName()  
 };  
 }  
}

##### 来个@Import标注的类，导入MyImportSelector

package com.javacode2018.lesson001.demo24.test5;  
  
import com.javacode2018.lesson001.demo24.test4.MyImportBeanDefinitionRegistrar;  
import org.springframework.context.annotation.Import;  
  
/\*\*  
 \* 通过@Import导入MyImportSelector接口实现类  
 \*/  
@Import({MyImportSelector.class})  
public class MainConfig5 {  
}

##### 新增测试用例

ImportTest中新增个方法

@Test  
public void test5() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig5.class);  
 //2.输出容器中定义的所有bean信息  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

部分输出如下：

com.javacode2018.lesson001.demo24.test5.Service1->com.javacode2018.lesson001.demo24.test5.Service1@45b4c3a9  
name->公众号：马士兵老师  
address->上海市

输出中可以看到Service1以及Module1Config中定义的2个bean都有了。

### 来一个牛逼的案例

#### 需求

凡是类名中包含service的，调用他们内部任何方法，我们希望调用之后能够输出这些方法的耗时。

#### 实现分析

之前我们讲过代理， 此处我们就可以通过代理来实现，bean实例创建的过程中，我们可以给这些bean生成一个代理，在代理中统计方法的耗时，这里面有2点：

1. 创建一个代理类，通过代理来间接访问需要统计耗时的bean对象
2. 拦截bean的创建，给bean实例生成代理生成代理

#### 具体实现

##### 先来两个Service类

###### Service1

package com.javacode2018.lesson001.demo24.test6;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
 public void m1() {  
 System.out.println(this.getClass() + ".m1()");  
 }  
}

###### Service2

package com.javacode2018.lesson001.demo24.test6;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 public void m1() {  
 System.out.println(this.getClass() + ".m1()");  
 }  
}

##### 创建统计耗时的代理类

下面我们使用cglib来实现一个代理类，如下：

package com.javacode2018.lesson001.demo24.test6;  
  
  
import org.springframework.cglib.proxy.Enhancer;  
import org.springframework.cglib.proxy.MethodInterceptor;  
import org.springframework.cglib.proxy.MethodProxy;  
  
import java.lang.reflect.Method;  
  
public class CostTimeProxy implements MethodInterceptor {  
 //目标对象  
 private Object target;  
  
 public CostTimeProxy(Object target) {  
 this.target = target;  
 }  
  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 long starTime = System.nanoTime();  
 //调用被代理对象（即target）的方法，获取结果  
 Object result = method.invoke(target, objects); //@1  
 long endTime = System.nanoTime();  
 System.out.println(method + "，耗时(纳秒)：" + (endTime - starTime));  
 return result;  
 }  
  
 /\*\*  
 \* 创建任意类的代理对象  
 \*  
 \* @param target  
 \* @param <T>  
 \* @return  
 \*/  
 public static <T> T createProxy(T target) {  
 CostTimeProxy costTimeProxy = new CostTimeProxy(target);  
 Enhancer enhancer = new Enhancer();  
 enhancer.setCallback(costTimeProxy);  
 enhancer.setSuperclass(target.getClass());  
 return (T) enhancer.create();  
 }  
}

createProxy方法可以用来给某个对象生成代理对象

需要了解cglib的可以看：代理详解（Java动态代理&cglib代理）

##### 拦截bean实例的创建，返回代理对象

这里我们需要用到spring中的一个接口：

org.springframework.beans.factory.config.BeanPostProcessor  
  
public interface BeanPostProcessor {  
  
 /\*\*  
 \* bean初始化之后会调用的方法  
 \*/  
 @Nullable  
 default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 return bean;  
 }  
  
 /\*\*  
 \* bean初始化之后会调用的方法  
 \*/  
 @Nullable  
 default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 return bean;  
 }  
  
}

这个接口是bean处理器，内部有2个方法，分别在bean初始化前后会进行调用，以后讲声明周期的时候还会细说的，这里你只需要知道bean初始化之后会调用postProcessAfterInitialization方法就行，这个方法中我们会给bean创建一个代理对象。

下面我们创建一个BeanPostProcessor实现类：

package com.javacode2018.lesson001.demo24.test6;  
  
import com.javacode2018.lesson001.demo23.test4.CostTimeProxy;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.lang.Nullable;  
  
public class MethodCostTimeProxyBeanPostProcessor implements BeanPostProcessor {  
 @Nullable  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 if (bean.getClass().getName().toLowerCase().contains("service")) {  
 return CostTimeProxy.createProxy(bean); //@1  
 } else {  
 return bean;  
 }  
 }  
}

@1：使用上面创建代理类来给当前bean对象创建一个代理

需要将MethodCostTimeProxyBeanPostProcessor注册到容器中才会起作用，下面我们通过@Import结合ImportSelector的方式来导入这个类，将其注册到容器中。

##### MethodCostTimeImportSelector

package com.javacode2018.lesson001.demo24.test6;  
  
  
import com.javacode2018.lesson001.demo23.test4.MethodCostTimeProxyBeanPostProcessor;  
import org.springframework.context.annotation.ImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class MethodCostTimeImportSelector implements ImportSelector {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{MethodCostTimeProxyBeanPostProcessor.class.getName()};  
 }  
}

##### 来一个@Import来导入MethodCostTimeImportSelector

下面我们使用注解的方式，在注解上使用@Import，如下：

package com.javacode2018.lesson001.demo24.test6;  
  
  
import com.javacode2018.lesson001.demo23.test4.MethodCostTimeImportSelector;  
import org.springframework.context.annotation.Import;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Import(MethodCostTimeImportSelector.class)  
public @interface EnableMethodCostTime {  
}

##### 来一个总的配置类

package com.javacode2018.lesson001.demo24.test6;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
@EnableMethodCostTime //@1  
public class MainConfig6 {  
}

上面使用了@CompontentScan注解，此时会将Servce1和Service2这两个类注册到容器中。

@1：此处使用了@EnableMethodCostTime注解，而@EnableMethodCostTime注解上使用了@Import(MethodCostTimeImportSelector.class)，此时MethodCostTimeImportSelector类中的MethodCostTimeProxyBeanPostProcessor会被注册到容器，会拦截bean的创建，创建耗时代理对象。

##### 来个测试用例

ImportTest中新增个方法

@Test  
public void test6() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig6.class);  
 Service1 service1 = context.getBean(Service1.class);  
 Service2 service2 = context.getBean(Service2.class);  
 service1.m1();  
 service2.m1();  
}

上面会调用service1和service2的方法

##### 运行输出

class com.javacode2018.lesson001.demo24.test6.Service1.m1()  
public void com.javacode2018.lesson001.demo24.test6.Service1.m1()，耗时(纳秒)：74200  
class com.javacode2018.lesson001.demo24.test6.Service2.m1()  
public void com.javacode2018.lesson001.demo24.test6.Service2.m1()，耗时(纳秒)：33800

**太牛逼了，需求实现了。**

**如果我们不想开启方法耗时统计，只需要将MainConfig6上的@EnableMethodCostTime去掉就可以了，用起来是不是特别爽。**

**spring中有很多类似的注解，以@EnableXXX开头的注解，基本上都是通过上面这种方式实现的，如：**

@EnableAspectJAutoProxy  
@EnableCaching  
@EnableAsync

继续向下看，还有一个更牛逼的接口DeferredImportSelector。

### DeferredImportSelector接口

**先给你透露一下，springboot中的核心功能@EnableAutoConfiguration就是靠DeferredImportSelector来实现的。**

DeferredImportSelector是ImportSelector的子接口，既然是ImportSelector的子接口，所以也可以通过@Import进行导入，这个接口和ImportSelector不同地方有两点：

1. 延迟导入
2. 指定导入的类的处理顺序

#### 延迟导入

比如@Import的value包含了多个普通类、多个@Configuration标注的配置类、多个ImportSelector接口的实现类，多个ImportBeanDefinitionRegistrar接口的实现类，还有DeferredImportSelector接口实现类，此时spring处理这些被导入的类的时候，**会将DeferredImportSelector类型的放在最后处理，会先处理其他被导入的类，其他类会按照value所在的前后顺序进行处理**。

那么我们是可以做很多事情的，比如我们可以在DeferredImportSelector导入的类中判断一下容器中是否已经注册了某个bean，如果没有注册过，那么再来注册。

以后我们会讲到另外一个注解@Conditional，这个注解可以按条件来注册bean，比如可以判断某个bean不存在的时候才进行注册，某个类存在的时候才进行注册等等各种条件判断，通过@Conditional来结合DeferredImportSelector可以做很多事情。

#### 来个延迟导入的案例

来3个配置类，每个配置类中都通过@Bean定一个string类型的bean，内部输出一句文字。

##### Configuration1

package com.javacode2018.lesson001.demo24.test7;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Configuration1 {  
 @Bean  
 public String name1() {  
 System.out.println("name1");  
 return "name1";  
 }  
}

##### Configuration2

package com.javacode2018.lesson001.demo24.test7;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Configuration2 {  
 @Bean  
 public String name2() {  
 System.out.println("name2");  
 return "name2";  
 }  
}

##### Configuration3

package com.javacode2018.lesson001.demo24.test7;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Configuration3 {  
 @Bean  
 public String name3() {  
 System.out.println("name3");  
 return "name3";  
 }  
}

##### 来一个ImportSelector实现类，导入Configuration1

package com.javacode2018.lesson001.demo24.test7;  
  
import org.springframework.context.annotation.ImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class ImportSelector1 implements ImportSelector {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{  
 Configuration1.class.getName()  
 };  
 }  
}

##### 来一个DeferredImportSelector实现类，导入Configuration2

package com.javacode2018.lesson001.demo24.test7;  
  
import org.springframework.context.annotation.DeferredImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class DeferredImportSelector1 implements DeferredImportSelector {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{Configuration2.class.getName()};  
 }  
}

##### 来一个总的配置类

package com.javacode2018.lesson001.demo24.test7;  
  
import org.springframework.context.annotation.Import;  
  
@Import({  
 DeferredImportSelector1.class,  
 Configuration3.class,  
 ImportSelector1.class,  
})  
public class MainConfig7 {  
}

注意上面的@Import中被导入类的顺序：

DeferredImportSelector1->Configuration3->ImportSelector1

下面来个测试用例，看一下3个配置文件中@Bean标注的方法被执行的先后顺序。

##### 测试用例

ImportTest中新增个方法

@Test  
public void test7() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig7.class);  
}

##### 运行输出

name3  
name1  
name2

**输出的结果结合一下@Import中被导入的3个类的顺序，可以看出DeferredImportSelector1是被最后处理的，其他2个是按照value中所在的先后顺序处理的。**

#### 指定导入的类的处理顺序

当@Import中有多个DeferredImportSelector接口的实现类时候，可以指定他们的顺序，指定顺序常见2种方式

##### 实现Ordered接口的方式

org.springframework.core.Ordered  
  
public interface Ordered {  
  
 int HIGHEST\_PRECEDENCE = Integer.MIN\_VALUE;  
  
 int LOWEST\_PRECEDENCE = Integer.MAX\_VALUE;  
  
 int getOrder();  
  
}

value的值越小，优先级越高。

##### 实现Order注解的方式

org.springframework.core.annotation.Order  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})  
@Documented  
public @interface Order {  
  
 int value() default Ordered.LOWEST\_PRECEDENCE;  
  
}

value的值越小，优先级越高。

下面我们来个案例感受一下。

#### 来个指定导入类处理顺序的案例

来2个配置类，内部都有一个@Bean标注的方法，用来注册一个bean，方法内部输出一行文字

##### Configuration1

package com.javacode2018.lesson001.demo24.test8;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Configuration1 {  
 @Bean  
 public String name1() {  
 System.out.println("name1");  
 return "name1";  
 }  
}

##### Configuration2

package com.javacode2018.lesson001.demo24.test8;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class Configuration2 {  
 @Bean  
 public String name2() {  
 System.out.println("name2");  
 return "name2";  
 }  
}

来2个DeferredImportSelector实现类，分别来导入上面2个配置文件，顺便通过Ordered接口指定一下顺序

##### DeferredImportSelector1

package com.javacode2018.lesson001.demo24.test8;  
  
import org.springframework.context.annotation.DeferredImportSelector;  
import org.springframework.core.Ordered;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class DeferredImportSelector1 implements DeferredImportSelector, Ordered {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{Configuration1.class.getName()};  
 }  
  
 @Override  
 public int getOrder() {  
 return 2;  
 }  
}

##### DeferredImportSelector2

package com.javacode2018.lesson001.demo24.test8;  
  
import com.javacode2018.lesson001.demo24.test7.Configuration2;  
import org.springframework.context.annotation.DeferredImportSelector;  
import org.springframework.core.Ordered;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class DeferredImportSelector2 implements DeferredImportSelector, Ordered {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 return new String[]{Configuration2.class.getName()};  
 }  
  
 @Override  
 public int getOrder() {  
 return 1;  
 }  
}

**DeferredImportSelector1的order为2，DeferredImportSelector2的order为1，order值越小优先级越高。**

来个总的配置类，引入上面两个ImportSelector

##### MainConfig8

package com.javacode2018.lesson001.demo24.test8;  
  
  
import org.springframework.context.annotation.Import;  
  
@Import({  
 DeferredImportSelector1.class,  
 DeferredImportSelector2.class,  
})  
public class MainConfig8 {  
}

##### 测试用例

ImportTest中新增个方法

@Test  
public void test8() {  
 //1.通过AnnotationConfigApplicationContext创建spring容器，参数为@Import标注的类  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig8.class);  
}

##### 运行输出

name2  
name1

**结果配合order的值，按照order从小到大来处理，可以看出DeferredImportSelector2先被处理的。**

### Spring中这块的源码

**@Import注解是被下面这个类处理的**

org.springframework.context.annotation.ConfigurationClassPostProcessor

前面介绍的@Configuration、@Bean、@CompontentScan、@CompontentScans都是被这个类处理的，这个类是高手必经之路，建议花点时间研究研究。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 总结

1. @Import可以用来批量导入任何普通的组件、配置类，将这些类中定义的所有bean注册到容器中
2. @Import常见的5种用法需要掌握
3. 掌握ImportSelector、ImportBeanDefinitionRegistrar、DeferredImportSelector的用法
4. DeferredImportSelector接口可以实现延迟导入、按序导入的功能
5. spring中很多以@Enable开头的都是使用@Import集合ImportSelector方式实现的
6. BeanDefinitionRegistry接口：bean定义注册器，这个需要掌握常见的方法

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)

## Spring系列第20篇：@Conditional通过条件来控制bean的注册

### 先看几个问题

1. @Conditional是做什么的?
2. @Conditional多个条件是什么逻辑关系？
3. 条件判断在什么时候执行？
4. ConfigurationCondition和Condition有什么区别？什么时候使用ConfigurationCondition？
5. 多个Condition执行的顺序是什么样的？可以配置优先级么？
6. 可以介绍一下@Conditional常见的一些用法么？

### @Conditional注解

**@Conditional注解是从spring4.0才有的，可以用在任何类型或者方法上面，通过@Conditional注解可以配置一些条件判断，当所有条件都满足的时候，被@Conditional标注的目标才会被spring容器处理。**

比如可以通过@Conditional来控制bean是否需要注册，控制被@Configuration标注的配置类是需要需要被解析等。

效果就像这段代码，相当于在spring容器解析目标前面加了一个条件判断：

if(@Conditional中配置的多个条件是否都匹配){  
//spring继续处理被@Conditional注解标注的对象  
}

@Conditional源码：

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Conditional {  
 Class<? extends Condition>[] value();  
}

这个注解只有一个value参数，Condition类型的数组，Condition是一个接口，表示一个条件判断，内部有个方法返回true或false，当所有Condition都成立的时候，@Conditional的结果才成立。

下面我们来看一下Condition接口。

### Condition接口

用来表示条件判断的接口，源码如下：

@FunctionalInterface  
public interface Condition {  
  
 /\*\*  
 \* 判断条件是否匹配  
 \* context:条件判断上下文  
 \*/  
 boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);  
  
}

是一个函数式接口，内部只有一个matches方法，用来判断条件是否成立的，2个参数：

* context：条件上下文，ConditionContext接口类型的，可以用来获取容器中的个人信息
* metadata：用来获取被@Conditional标注的对象上的所有注解信息

### ConditionContext接口

这个接口中提供了一些常用的方法，可以用来获取spring容器中的各种信息，看一下源码：

public interface ConditionContext {  
  
 /\*\*  
 \* 返回bean定义注册器，可以通过注册器获取bean定义的各种配置信息  
 \*/  
 BeanDefinitionRegistry getRegistry();  
  
 /\*\*  
 \* 返回ConfigurableListableBeanFactory类型的bean工厂，相当于一个ioc容器对象  
 \*/  
 @Nullable  
 ConfigurableListableBeanFactory getBeanFactory();  
  
 /\*\*  
 \* 返回当前spring容器的环境配置信息对象  
 \*/  
 Environment getEnvironment();  
  
 /\*\*  
 \* 返回资源加载器  
 \*/  
 ResourceLoader getResourceLoader();  
  
 /\*\*  
 \* 返回类加载器  
 \*/  
 @Nullable  
 ClassLoader getClassLoader();  
  
}

### 比较关键性的问题：条件判断在什么时候执行？

Spring对配置类的处理主要分为2个阶段：

#### 配置类解析阶段

会得到一批配置类的信息，和一些需要注册的bean

#### bean注册阶段

将配置类解析阶段得到的配置类和需要注册的bean注册到spring容器中

#### 看一下什么是配置类

类中有下面任意注解之一的就属于配置类：

1. 类上有@Compontent注解
2. 类上有@Configuration注解
3. 类上有@CompontentScan注解
4. 类上有@Import注解
5. 类上有@ImportResource注解
6. 类中有@Bean标注的方法

判断一个类是不是一个配置类，是否的是下面这个方法，有兴趣的可以看一下：

org.springframework.context.annotation.ConfigurationClassUtils#isConfigurationCandidate

spring中处理这2个过程会循环进行，直到完成所有配置类的解析及所有bean的注册。

#### Spring对配置类处理过程

源码位置：

org.springframework.context.annotation.ConfigurationClassPostProcessor#processConfigBeanDefinitions

整个过程大致的过程如下：

1. 通常我们会通过new AnnotationConfigApplicationContext()传入多个配置类来启动spring容器
2. spring对传入的多个配置类进行解析
3. 配置类解析阶段：这个过程就是处理配置类上面6中注解的过程，此过程中又会发现很多新的配置类，比如@Import导入的一批新的类刚好也符合配置类，而被@CompontentScan扫描到的一些类刚好也是配置类；此时会对这些新产生的配置类进行同样的过程解析
4. bean注册阶段：配置类解析后，会得到一批配置类和一批需要注册的bean，此时spring容器会将这批配置类作为bean注册到spring容器，同样也会将这批需要注册的bean注册到spring容器
5. 经过上面第3个阶段之后，spring容器中会注册很多新的bean，这些新的bean中可能又有很多新的配置类
6. Spring从容器中将所有bean拿出来，遍历一下，会过滤得到一批未处理的新的配置类，继续交给第3步进行处理
7. step3到step6，这个过程会经历很多次，直到完成所有配置类的解析和bean的注册

从上面过程中可以了解到：

1. 可以在配置类上面加上@Conditional注解，来控制是否需要解析这个配置类，配置类如果不被解析，那么这个配置上面6种注解的解析都会被跳过
2. 可以在被注册的bean上面加上@Conditional注解，来控制这个bean是否需要注册到spring容器中
3. 如果配置类不会被注册到容器，那么这个配置类解析所产生的所有新的配置类及所产生的所有新的bean都不会被注册到容器

一个配置类被spring处理有2个阶段：配置类解析阶段、bean注册阶段（将配置类作为bean被注册到spring容器)。

如果将Condition接口的实现类作为配置类上@Conditional中，那么这个条件会对两个阶段都有效，此时通过Condition是无法精细的控制某个阶段的，如果想控制某个阶段，比如可以让他解析，但是不能让他注册，此时就就需要用到另外一个接口了：ConfigurationCondition

### ConfigurationCondition接口

看一下这个接口的源码：

public interface ConfigurationCondition extends Condition {  
  
 /\*\*  
 \* 条件判断的阶段，是在解析配置类的时候过滤还是在创建bean的时候过滤  
 \*/  
 ConfigurationPhase getConfigurationPhase();  
  
  
 /\*\*  
 \* 表示阶段的枚举：2个值  
 \*/  
 enum ConfigurationPhase {  
  
 /\*\*  
 \* 配置类解析阶段，如果条件为false，配置类将不会被解析  
 \*/  
 PARSE\_CONFIGURATION,  
  
 /\*\*  
 \* bean注册阶段，如果为false，bean将不会被注册  
 \*/  
 REGISTER\_BEAN  
 }  
  
}

ConfigurationCondition接口相对于Condition接口多了一个getConfigurationPhase方法，用来指定条件判断的阶段，是在解析配置类的时候过滤还是在创建bean的时候过滤。

### @Conditional使用的3步骤

1. 自定义一个类，实现Condition或ConfigurationCondition接口，实现matches方法
2. 在目标对象上使用@Conditional注解，并指定value的指为自定义的Condition类型
3. 启动spring容器加载资源，此时@Conditional就会起作用了

### 案例1：阻止配置类的处理

在配置类上面使用@Conditional，这个注解的value指定的Condition当有一个为false的时候，spring就会跳过处理这个配置类。

自定义一个Condition类：

package com.javacode2018.lesson001.demo25.test3;  
  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
public class MyCondition1 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 return false;  
 }  
}

matches方法内部我们可以随意发挥，此处为了演示效果就直接返回false。

来个配置类，在配置类上面使用上面这个条件，此时会让配置类失效，如下：

package com.javacode2018.lesson001.demo25.test3;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
  
@Conditional(MyCondition1.class) //@1  
@Configuration  
public class MainConfig3 {  
 @Bean  
 public String name() { //@1  
 return "马士兵老师";  
 }  
}

@1：使用了自定义的条件类

@2：通过@Bean标注这name这个方法，如果这个配置类成功解析，会将name方法的返回值作为bean注册到spring容器

来个测试类，启动spring容器加载MainConfig3配置类，如下：

package com.javacode2018.lesson001.demo25;  
  
import com.javacode2018.lesson001.demo25.test3.MainConfig3;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.Map;  
  
public class ConditionTest {  
  
 @Test  
 public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig3.class);  
 Map<String, String> serviceMap = context.getBeansOfType(String.class);  
 serviceMap.forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
 }  
}

test3中，从容器中获取String类型的bean，运行test3没有任何输出。

我们可以将MainConfig3上面的@Conditional去掉，再次运行输出：

name->马士兵老师

### 案例2：阻止bean的注册

来个配置类，如下：

package com.javacode2018.lesson001.demo25.test4;  
  
import com.javacode2018.lesson001.demo25.test3.MyCondition1;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class MainConfig4 {  
 @Conditional(MyCondition1.class) //@1  
 @Bean  
 public String name() {  
 return "马士兵老师";  
 }  
  
 @Bean  
 public String address() {  
 return "上海市";  
 }  
}

上面2个方法上面使用了@Bean注解来定义了2个bean，name方法上面使用了@Conditional注解，这个条件会在name这个bean注册到容器之前会进行判断，当条件为true的时候，name这个bean才会被注册到容器。

ConditionTest中新增个测试用例来加载上面这个配置类，从容器中获取String类型所有bean输出，代码如下：

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig4.class);  
 Map<String, String> serviceMap = context.getBeansOfType(String.class);  
 serviceMap.forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
}

运行输出：

address->上海市

可以看到容器中只有一个address被注册了，而name这个bean没有被注册。

### 案例3：bean不存在的时候才注册

#### 需求

IService接口有两个实现类Service1和Service1，这两个类会放在2个配置类中通过@Bean的方式来注册到容器，此时我们想加个限制，只允许有一个IService类型的bean被注册到容器。

可以在@Bean标注的2个方法上面加上条件限制，当容器中不存在IService类型的bean时，才将这个方法定义的bean注册到容器，下面来看代码实现。

#### 代码实现

##### 条件判断类：OnMissingBeanCondition

package com.javacode2018.lesson001.demo25.test1;  
  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.context.annotation.ConfigurationCondition;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
import java.util.Map;  
  
public class OnMissingBeanCondition implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 //获取bean工厂  
 ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  
 //从容器中获取IService类型bean  
 Map<String, IService> serviceMap = beanFactory.getBeansOfType(IService.class);  
 //判断serviceMap是否为空  
 return serviceMap.isEmpty();  
 }  
  
}

上面matches方法中会看容器中是否存在IService类型的bean，不存在的时候返回true

##### IService接口

package com.javacode2018.lesson001.demo25.test1;  
  
public interface IService {  
}

##### 接口有2个实现类

###### Service1

package com.javacode2018.lesson001.demo25.test1;  
  
public class Service1 implements IService {  
}

###### Service2

package com.javacode2018.lesson001.demo25.test1;  
  
public class Service2 implements IService {  
}

##### 来一个配置类负责注册Service1到容器

package com.javacode2018.lesson001.demo25.test1;  
  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class BeanConfig1 {  
 @Conditional(OnMissingBeanCondition.class) //@1  
 @Bean  
 public IService service1() {  
 return new Service1();  
 }  
}

@1：方法之前使用了条件判断

##### 再来一个配置类负责注册Service2到容器

package com.javacode2018.lesson001.demo25.test1;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class BeanConfig2 {  
 @Conditional(OnMissingBeanCondition.class)//@1  
 @Bean  
 public IService service2() {  
 return new Service2();  
 }  
}

@1：方法之前使用了条件判断

##### 来一个总的配置类，导入另外2个配置类

package com.javacode2018.lesson001.demo25.test1;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;  
  
@Configuration  
@Import({BeanConfig1.class,BeanConfig2.class}) //@1  
public class MainConfig1 {  
}

@1：通过@Import将其他2个配置类导入

##### 来个测试用例

ConditionTest新增一个方法，方法中从容器中获取IService类型的bean，然后输出：

@Test  
public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig1.class);  
 Map<String, IService> serviceMap = context.getBeansOfType(IService.class);  
 serviceMap.forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
}

运行输出：

service1->com.javacode2018.lesson001.demo25.test1.Service1@2cd76f31

可以看出容器中只有一个IService类型的bean。

可以将@Bean标注的2个方法上面的@Conditional去掉，再运行会输出：

service1->com.javacode2018.lesson001.demo25.test1.Service1@49438269  
service2->com.javacode2018.lesson001.demo25.test1.Service2@ba2f4ec

此时没有条件限制，2个Service都会注册到容器。

### 案例4：根据环境选择配置类

平常我们做项目的时候，有开发环境、测试环境、线上环境，每个环境中有些信息是不一样的，比如数据库的配置信息，下面我们来模拟不同环境中使用不同的配置类来注册不同的bean。

#### 自定义一个条件的注解

package com.javacode2018.lesson001.demo25.test2;  
  
import org.springframework.context.annotation.Conditional;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Conditional(EnvCondition.class) //@1  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface EnvConditional {  
 //环境(测试环境、开发环境、生产环境)  
 enum Env { //@2  
 TEST, DEV, PROD  
 }  
  
 //环境  
 Env value() default Env.DEV; //@3  
}

@1：注意这个注解比较特别，这个注解上面使用到了@Conditional注解，这个地方使用到了一个自定义Conditione类：EnvCondition

@2：枚举，表示环境，定义了3个环境

@3：这个参数用指定环境

上面这个注解一会我们会用在不同环境的配置类上面

#### 下面来3个配置类

让3个配置类分别在不同环境中生效，会在这些配置类上面使用上面自定义的@EnvConditional注解来做条件限定。

每个配置类中通过@Bean来定义一个名称为name的bean，一会通过输出这个bean来判断哪个配置类生效了。

下面来看3个配置类的代码

##### 测试环境配置类

package com.javacode2018.lesson001.demo25.test2;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@EnvConditional(EnvConditional.Env.TEST)//@1  
public class TestBeanConfig {  
 @Bean  
 public String name() {  
 return "我是测试环境!";  
 }  
}

@1指定的测试环境

##### 开发环境配置类

package com.javacode2018.lesson001.demo25.test2;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@EnvConditional(EnvConditional.Env.DEV) //@1  
public class DevBeanConfig {  
 @Bean  
 public String name() {  
 return "我是开发环境!";  
 }  
}

@1：指定的开发环境

##### 生产环境配置类

package com.javacode2018.lesson001.demo25.test2;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@EnvConditional(EnvConditional.Env.PROD) //@1  
public class ProdBeanConfig {  
 @Bean  
 public String name() {  
 return "我是生产环境!";  
 }  
}

@1：指定的生产环境

#### 下面来看一下条件类：EnvCondition

条件类会解析配置类上面@EnvConditional注解，得到环境信息。

然后和目前的环境对比，决定返回true还是false，如下：

package com.javacode2018.lesson001.demo25.test2;  
  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
public class EnvCondition implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 //当前需要使用的环境  
 EnvConditional.Env curEnv = EnvConditional.Env.DEV; //@1  
 //获取使用条件的类上的EnvCondition注解中对应的环境  
 EnvConditional.Env env = (EnvConditional.Env) metadata.getAllAnnotationAttributes(EnvConditional.class.getName()).get("value").get(0);  
 return env.equals(curEnv);  
 }  
  
}

@1：这个用来指定当前使用的环境，此处假定当前使用的是开发环境，这个我们以后可以任意发挥，比如将这些放到配置文件中，此处方便演示效果。

#### 来个测试用例

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig2.class);  
 System.out.println(context.getBean("name"));  
}

#### 运行输出

我是开发环境!

可以看到开发环境生效了。

修改一下EnvCondition的代码，切换到生产环境：

EnvConditional.Env curEnv = EnvConditional.Env.PROD;

再次运行test2方法输出：

我是生产环境!

生产环境配置类生效了。

### 案例5：Condition指定优先级

#### 多个Condition按顺序执行

@Condtional中value指定多个Condtion的时候，默认情况下会按顺序执行，还是通过代码来看一下效果。

下面代码中定义了3个Condition，每个Condition的matches方法中会输出当前类名，然后在配置类上面同时使用这3个Condition：

package com.javacode2018.lesson001.demo25.test5;  
  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
class Condition1 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
}  
  
class Condition2 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
}  
  
class Condition3 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
}  
  
@Configuration  
@Conditional({Condition1.class, Condition2.class, Condition3.class})  
public class MainConfig5 {  
}

来个测试用例

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig5.class);  
}

运行输出：

com.javacode2018.lesson001.demo25.test5.Condition1  
com.javacode2018.lesson001.demo25.test5.Condition2  
com.javacode2018.lesson001.demo25.test5.Condition3  
com.javacode2018.lesson001.demo25.test5.Condition1  
com.javacode2018.lesson001.demo25.test5.Condition2  
com.javacode2018.lesson001.demo25.test5.Condition3  
com.javacode2018.lesson001.demo25.test5.Condition1  
com.javacode2018.lesson001.demo25.test5.Condition2  
com.javacode2018.lesson001.demo25.test5.Condition3

上面有多行输出，是因为spring解析整个配置类的过程中，有好几个地方都会执行条件判断。

咱们只用关注前3行，可以看出输出的属性和@Conditional中value值的顺序是一样的。

#### 指定Condition的顺序

自定义的Condition可以实现PriorityOrdered接口或者继承Ordered接口，或者使用@Order注解，通过这些来指定这些Condition的优先级。

**排序规则：先按PriorityOrdered排序，然后按照order的值进行排序；也就是：PriorityOrdered asc,order值 asc**

下面这几个都可以指定order的值  
接口：org.springframework.core.Ordered，有个getOrder方法用来返回int类型的值  
接口：org.springframework.core.PriorityOrdered，继承了Ordered接口，所以也有getOrder方法  
注解：org.springframework.core.annotation.Order，有个int类型的value参数指定Order的大小

看案例代码：

package com.javacode2018.lesson001.demo25.test6;  
  
  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.context.annotation.Conditional;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.core.Ordered;  
import org.springframework.core.PriorityOrdered;  
import org.springframework.core.annotation.Order;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
@Order(1) //@1  
class Condition1 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
}  
  
class Condition2 implements Condition, Ordered { //@2  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
  
 @Override  
 public int getOrder() { //@3  
 return 0;  
 }  
}  
  
class Condition3 implements Condition, PriorityOrdered { //@4  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 System.out.println(this.getClass().getName());  
 return true;  
 }  
  
 @Override  
 public int getOrder() {  
 return 1000;  
 }  
}  
  
@Configuration  
@Conditional({Condition1.class, Condition2.class, Condition3.class})//@5  
public class MainConfig6 {  
}

@1：Condition1通过@Order指定顺序，值为1

@2：Condition2通过实现了Ordered接口来指定顺序，@3：getOrder方法返回1

@4：Condition3实现了PriorityOrdered接口，实现这个接口需要重写getOrder方法，返回1000

@5：Condtion顺序为1、2、3

根据排序的规则，PriorityOrdered的会排在前面，然后会再按照order升序，最后可以顺序是：

Condtion3->Condtion2->Condtion1

来个测试用例看看效果是不是我们分析的这样：

@Test  
public void test6() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig6.class);  
}

运行test6，部分输出如下：

com.javacode2018.lesson001.demo25.test6.Condition3  
com.javacode2018.lesson001.demo25.test6.Condition2  
com.javacode2018.lesson001.demo25.test6.Condition1

结果和我们分析的一致。

### 案例6：ConfigurationCondition使用

ConfigurationCondition使用的比较少，很多地方对这个基本上也不会去介绍，Condition接口基本上可以满足99%的需求了，但是springboot中却大量用到了ConfigurationCondition这个接口。

ConfigurationCondition通过解释比较难理解，来个案例感受一下：

#### 来一个普通的类：Service

package com.javacode2018.lesson001.demo25.test7;  
  
public class Service {  
}

#### 来一个配置类，通过配置类注册上面这个Service

package com.javacode2018.lesson001.demo25.test7;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class BeanConfig1 {  
 @Bean  
 public Service service() {  
 return new Service();  
 }  
}

#### 再来一个配置类：BeanConfig2

package com.javacode2018.lesson001.demo25.test7;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class BeanConfig2 {  
 @Bean  
 public String name() {  
 return "马士兵老师";  
 }  
}

#### 来一个总的配置类

package com.javacode2018.lesson001.demo25.test7;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;  
  
@Configuration  
@Import({BeanConfig1.class, BeanConfig2.class})  
public class MainConfig7 {  
}

上面通过@Import引入了另外2个配置类

#### 来个测试用例加载MainConfig7配置类

@Test  
public void test7() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig7.class);  
 context.getBeansOfType(String.class).forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
}

上面从容器中获取String类型的bean，然后输出。

#### 运行输出

name->马士兵老师

#### 现在我们有个需求

当容器中有Service这种类型的bean的时候，BeanConfig2才生效。

很简单吧，加个Condition就行了，内部判断容器中是否有Service类型的bean，继续

#### 来个自定义的Condition

package com.javacode2018.lesson001.demo25.test7;  
  
  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
public class MyCondition1 implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 //获取spring容器  
 ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  
 //判断容器中是否存在Service类型的bean  
 boolean existsService = !beanFactory.getBeansOfType(Service.class).isEmpty();  
 return existsService;  
 }  
}

上面代码很简单，判断容器中是否有IService类型的bean。

#### BeanConfig2上使用Condition条件判断

@Configuration  
@Conditional(MyCondition1.class)  
public class BeanConfig2 {  
 @Bean  
 public String name() {  
 return "马士兵老师";  
 }  
}

#### 再次运行test7输出

无任何输出

#### 为什么？

在文章前面我们说过，配置类的处理会依次经过2个阶段：配置类解析阶段和bean注册阶段，Condition接口类型的条件会对这两个阶段都有效，解析阶段的时候，容器中是还没有Service这个bean的，配置类中通过@Bean注解定义的bean在bean注册阶段才会被注册到spring容器，所以BeanConfig2在解析阶段去容器中是看不到Service这个bean的，所以就被拒绝了。

**此时我们需要用到ConfigurationCondition了，让条件判断在bean注册阶段才起效。**

#### 自定义一个ConfigurationCondition类

package com.javacode2018.lesson001.demo25.test7;  
  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.context.annotation.ConfigurationCondition;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
public class MyConfigurationCondition1 implements ConfigurationCondition {  
 @Override  
 public ConfigurationPhase getConfigurationPhase() {  
 return ConfigurationPhase.REGISTER\_BEAN; //@1  
 }  
  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 //获取spring容器  
 ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  
 //判断容器中是否存在Service类型的bean  
 boolean existsService = !beanFactory.getBeansOfType(Service.class).isEmpty();  
 return existsService;  
 }  
}

@1：指定条件在bean注册阶段，这个条件才有效

matches方法中的内容直接复制过来，判断规则不变。

#### 修改BeanConfig2的类容

将  
@Conditional(MyCondition1.class)  
替换为  
@Conditional(MyConfigurationCondition1.class)

#### 再次运行test7输出

name->马士兵老师

此时name这个bean被输出了。

可以再试试将BeanConfig1中service方法上面的@Bean去掉，此时Service就不会被注册到容器，再运行一下test7，会发现没有输出了，此时BeanConfig2会失效。

**判断bean存不存在的问题，通常会使用ConfigurationCondition这个接口，阶段为：REGISTER\_BEAN，这样可以确保条件判断是在bean注册阶段执行的。**

对springboot比较熟悉的，它里面有很多@Conditionxxx这样的注解，可以去看一下这些注解，很多都实现了ConfigurationCondition接口。

### Spring中这块的源码

**@Conditional注解是被下面这个类处理的**

org.springframework.context.annotation.ConfigurationClassPostProcessor

**又是这个类，说了很多次了，非常重要的一个类，大家下去了多撸一下这个类的源码，这样理解起来更顺畅一些。**

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 总结

1. @Conditional注解可以标注在spring需要处理的对象上（配置类、@Bean方法），相当于加了个条件判断，通过判断的结果，让spring觉得是否要继续处理被这个注解标注的对象
2. spring处理配置类大致有2个过程：解析配置类、注册bean，这两个过程中都可以使用@Conditional来进行控制spring是否需要处理这个过程
3. Condition默认会对2个过程都有效
4. ConfigurationCondition控制得更细一些，可以控制到具体那个阶段使用条件判断

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第19篇：@import详解(bean批量注册)

## Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）

### 先来看几个问题

1. 通过注解的方式注入依赖对象，介绍一下你知道的几种方式
2. @Autowired和@Resource有何区别
3. 说一下@Autowired查找候选者的过程
4. 说一下@Resource查找候选者的过程
5. @Qulifier有哪些用法？
6. @Qulifier加在类上面是干什么用的？
7. @Primary是做什么的？
8. 泛型注入用过么？

**这些问题如果你都ok，那么恭喜你，很厉害。**

### 本文内容

1. 介绍spring中通过注解实现依赖注入的所有方式
   * @Autowired注解
   * @Qualifier注解
   * @Resource注解
   * @Primary注解
   * @Bean中注入的几种方式
2. 将指定类型的所有bean，注入到集合中
3. 将指定类型的所有bean，注入到map中
4. 注入泛型
5. 依赖注入源码方面的一些介绍

**本文内容比较多，所有知识点均有详细案例，大家一定要敲一遍，加深理解。**

### @Autowired：注入依赖对象

#### 作用

**实现依赖注入，spring容器会对bean中所有字段、方法进行遍历，标注有@Autowired注解的，都会进行注入。**

看一下其定义：

@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD, ElementType.ANNOTATION\_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Autowired {  
  
 /\*\*  
 \* Declares whether the annotated dependency is required.  
 \* <p>Defaults to {@code true}.  
 \*/  
 boolean required() default true;  
  
}

可以用在构造器、方法、方法参数、字段、注解上。

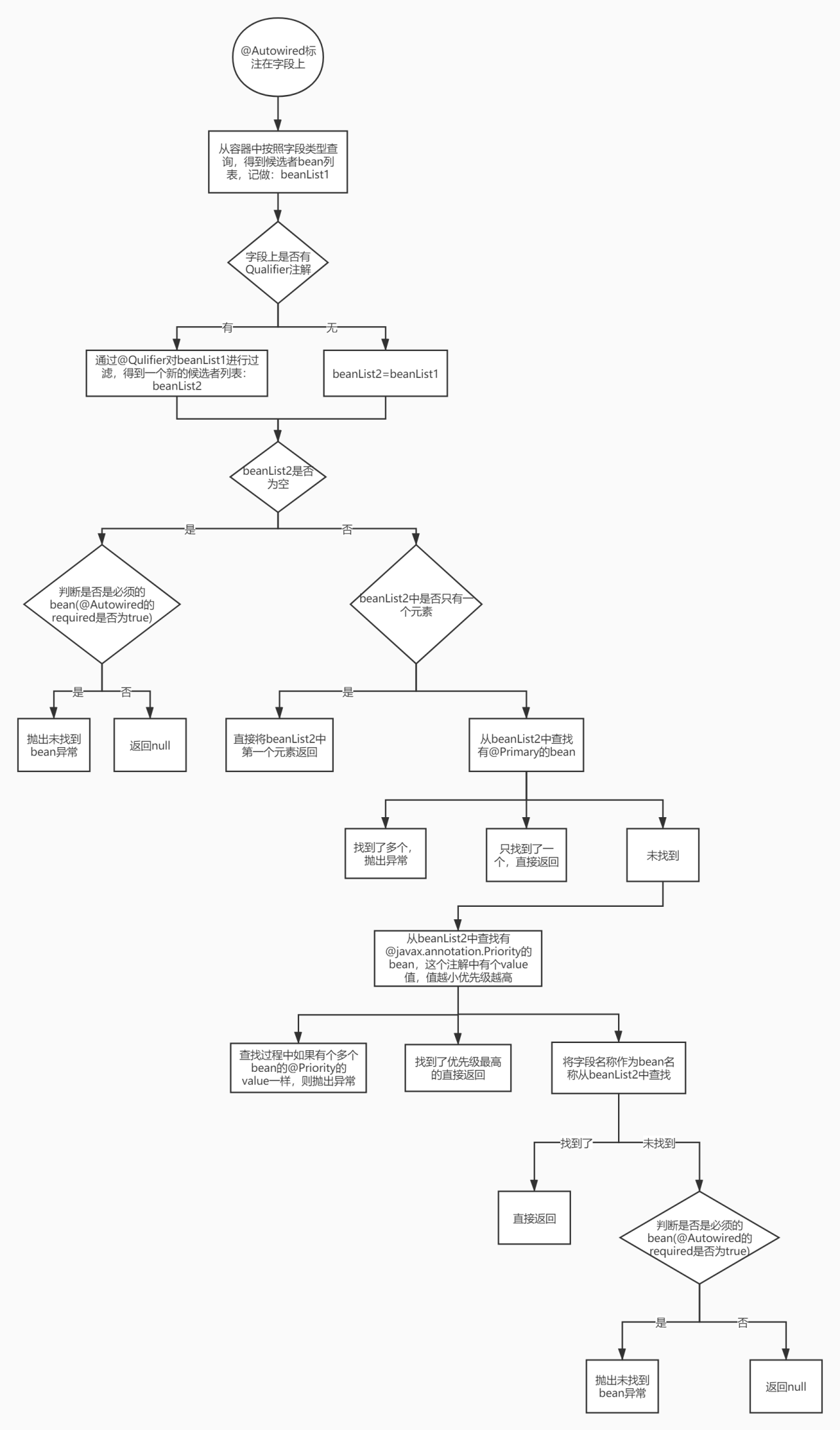
参数：

required：标注的对象是否必须注入，可能这个对象在容器中不存在，如果为true的时候，找不到匹配的候选者就会报错，为false的时候，找不到也没关系 。

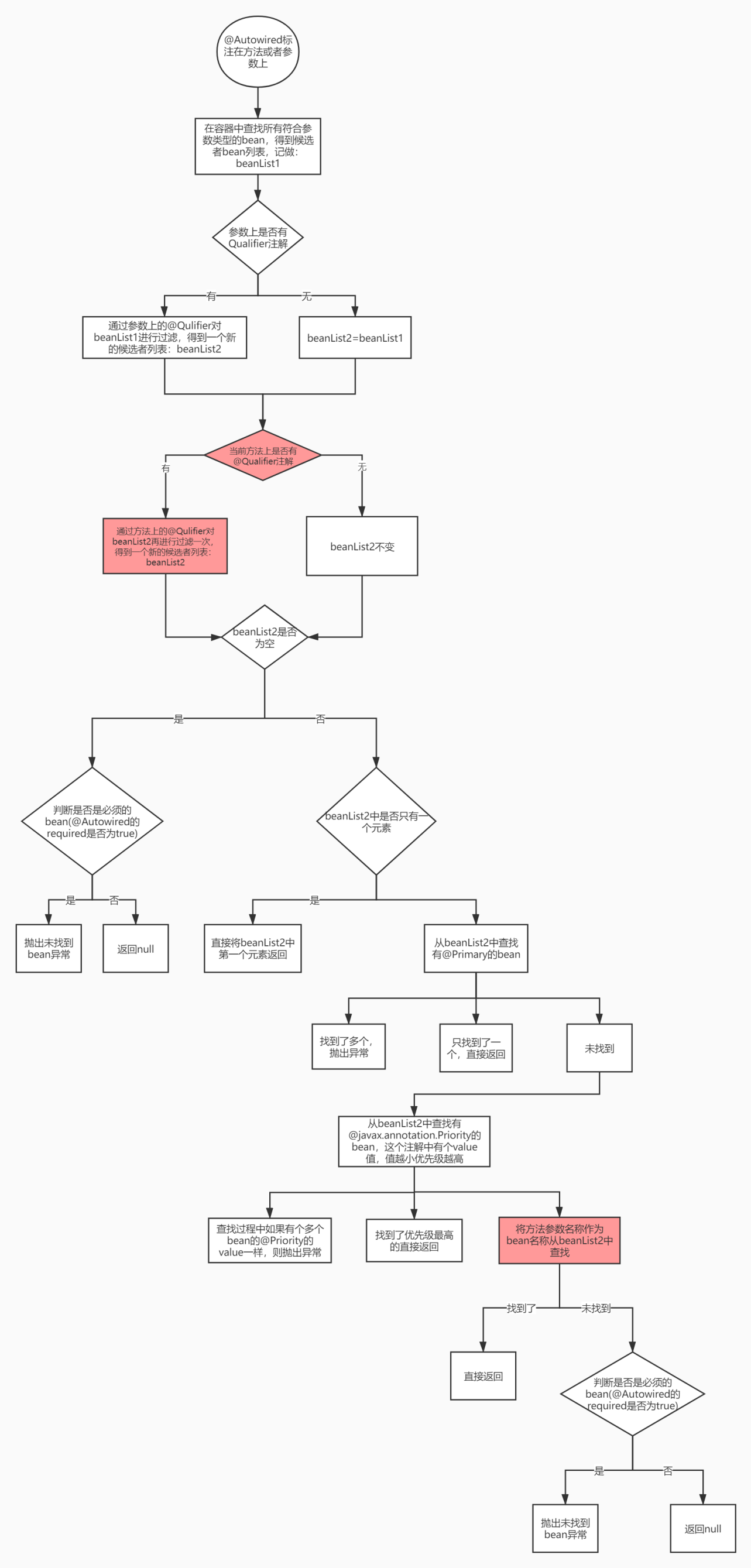
#### @Autowire查找候选者的过程

**查找过程有点复杂，看不懂的可以先跳过，先看后面案例，本文看完之后，可以回头再来看这个过程。**

##### @Autowired标注在字段上面：假定字段类型为一个自定义的普通的类型，候选者查找过程如下



##### @Autowired标注在方法上或者方法参数上面：假定参数类型为为一个自定义的普通的类型，候选者查找过程如下：



上图中深色的表示方法注入和字段注入查找过程的不同点。

上图中展示的是方法中只有一个参数的情况，如果有多个参数，就重复上面的过程，直到找到所有需要注入的参数。

##### 将指定类型的所有bean注入到Collection中

如果被注入的对象是Collection类型的，可以指定泛型的类型，然后会按照上面的方式查找所有满足泛型类型所有的bean

##### 将指定类型的所有bean注入到Map中

如果被注入的对象是Map类型的，可以指定泛型的类型，key通常为String类型，value为需要查找的bean的类型，然后会按照上面方式查找所有注入value类型的bean，将bean的name作为key，bean对象作为value，放在HashMap中，然后注入。

##### @Autowired查找候选者可以简化为下面这样

按类型找->通过限定符@Qualifier过滤->@Primary->@Priority->根据名称找（字段名称或者参数名称）

**概括为：先按类型找，然后按名称找**

#### 案例1：@Autowired标注在构造器上，通过构造器注入依赖对象

##### Service1

package com.javacode2018.lesson001.demo26.test0;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test0;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 private Service1 service1;  
  
 public Service2() { //@1  
 System.out.println(this.getClass() + "无参构造器");  
 }  
  
 public Service2(Service1 service1) { //@2  
 System.out.println(this.getClass() + "有参构造器");  
 this.service1 = service1;  
 }  
  
 @Override  
 public String toString() { //@2  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

Service2中依赖于Service1，有2个构造方法

@1：无参构造器

@2：有参构造器，可以通过这个传入依赖的Service1

@3：重写了toString方法，一会打印测试的时候方便查看

##### 来个总的配置文件

package com.javacode2018.lesson001.demo26.test0;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan //@1  
public class MainConfig0 {  
}

@1：会自动扫描当前类所在的包，会将Service1和Service2注册到容器。

##### 来个测试用例

package com.javacode2018.lesson001.demo26;  
  
import com.javacode2018.lesson001.demo26.test0.MainConfig0;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class InjectTest {  
  
 @Test  
 public void test0() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig0.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
 }  
   
}

main方法中启动容器，加载MainConfig0配置类，然后输出容器中所有的bean

##### 运行部分输出

class com.javacode2018.lesson001.demo26.test0.Service2无参构造器  
service1->com.javacode2018.lesson001.demo26.test0.Service1@4a94ee4  
service2->Service2{service1=null}

输出中可以看出调用了Service2的无参构造器，service2中的service1为null

##### 通过@Autowired指定注入的构造器

在Service2有参有参构造器上面加上@Autowired注解，如下：

@Autowired  
public Service2(Service1 service1) {  
 System.out.println(this.getClass() + "有参构造器");  
 this.service1 = service1;  
}

##### 再次运行test0()

class com.javacode2018.lesson001.demo26.test0.Service2有参构造器  
service1->com.javacode2018.lesson001.demo26.test0.Service1@4ec4f3a0  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test0.Service1@4ec4f3a0}

Service2有参构造器被调用了，service2中的service1有值了。

#### 案例2：@Autowired标注在方法上，通过方法注入依赖的对象

##### Service1

package com.javacode2018.lesson001.demo26.test1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test1;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 private Service1 service1;  
  
 @Autowired  
 public void injectService1(Service1 service1) { //@1  
 System.out.println(this.getClass().getName() + ".injectService1()");  
 this.service1 = service1;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：方法上标注了@Autowired，spring容器会调用这个方法，从容器中查找Service1类型的bean，然后注入。

##### 来个总的配置文件

package com.javacode2018.lesson001.demo26.test1;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig1 {  
}

##### 来个测试用例

InjectTest中加个方法

@Test  
public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig1.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

com.javacode2018.lesson001.demo26.test1.Service2.injectService1()  
service1->com.javacode2018.lesson001.demo26.test1.Service1@9597028  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test1.Service1@9597028}

通过injectService1方法成功注入service1

#### 案例3：@Autowired标注在setter方法上，通过setter方法注入

上面2种通过构造器，和通过普通的一个方法注入，不是很常见，可以将@Autowired标注在set方法上面，来注入指定的对象

##### Service1

package com.javacode2018.lesson001.demo26.test2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test2;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
  
 private Service1 service1;  
  
 @Autowired  
 public void setService1(Service1 service1) { //@1  
 System.out.println(this.getClass().getName() + ".setService1方法");  
 this.service1 = service1;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：标准的set方法，方法上使用了 @Autowired，会通过这个方法注入Service1类型的bean对象。

##### 来个总的配置文件

package com.javacode2018.lesson001.demo26.test2;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig2 {  
}

##### 来个测试用例

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig2.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

com.javacode2018.lesson001.demo26.test2.Service2.setService1方法  
service1->com.javacode2018.lesson001.demo26.test2.Service1@6069db50  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test2.Service1@6069db50}

#### 案例4：@Autowired标注在方法参数上

##### Service1

package com.javacode2018.lesson001.demo26.test3;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test3;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
  
 private Service1 service1;  
  
 @Autowired  
 public void injectService1(Service1 service1, String name) { //@1  
 System.out.println(String.format("%s.injectService1(),{service1=%s,name=%s}", this.getClass().getName(), service1, name));  
 this.service1 = service1;  
 }  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：方法上标注了@Autowired，表示会将这个方法作为注入方法，这个方法有2个参数，spring查找这2个参数对应的bean，然后注入。

第一个参数对应的bean是存在的，第二个是一个String类型的，我们并没有定义String类型bean，一会看看效果

##### 来个总的配置文件

package com.javacode2018.lesson001.demo26.test3;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig3 {  
}

##### 来个测试用例

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig3.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'service2': Unsatisfied dependency expressed through method 'injectService1' parameter 1; nested exception is org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'java.lang.String' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {}

报错了，从错误信息中可以看出，通过injectService1方法注入的时候，第二个参数为String类型，spring从容器中没有找到String类型的候选bean，所以报错了。

##### 我们可以这么做

**多个参数的时候，方法上面的@Autowire默认对方法中所有参数起效，如果我们想对某个参数进行特定的配置，可以在参数上加上@Autowired，这个配置会覆盖方法上面的@Autowired配置。**

在第二个参数上面加上@Autowired，设置required为false：表示这个bean不是强制注入的，能找到就注入，找不到就注入一个null对象，调整一下代码，如下：

@Autowired  
public void injectService1(Service1 service1, @Autowired(required = false) String name) { //@1  
 System.out.println(String.format("%s.injectService1(),{service1=%s,name=%s}", this.getClass().getName(), service1, name));  
 this.service1 = service1;  
}

此时方法的第一个参数被方法上面的@Autowired约束

第二个参数受@Autowired(required = false)约束

##### 再次运行输出

com.javacode2018.lesson001.demo26.test3.Service2.injectService1(),{service1=com.javacode2018.lesson001.demo26.test3.Service1@59309333,name=null}  
service1->com.javacode2018.lesson001.demo26.test3.Service1@59309333  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test3.Service1@59309333}

注入成功了，service1有值，name为null

#### 案例5：@Autowired用在字段上

##### Service1

package com.javacode2018.lesson001.demo26.test4;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test4;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
}

##### Service3

package com.javacode2018.lesson001.demo26.test4;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service3 {  
  
 @Autowired  
 private Service1 service1;//@1  
  
 @Autowired  
 private Service2 service2;//@2  
  
 @Override  
 public String toString() {  
 return "Service3{" +  
 "service1=" + service1 +  
 ", service2=" + service2 +  
 '}';  
 }  
}

@1和@2：定义了2个字段，上面都标注了@Autowired，spring会去容器中按照类型查找这2种类型的bean，然后设置给这2个属性。

##### 来个总的配置文件

package com.javacode2018.lesson001.demo26.test4;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig4 {  
}

##### 来个测试用例

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig4.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo26.test4.Service1@7e07db1f  
service2->com.javacode2018.lesson001.demo26.test4.Service2@1189dd52  
service3->Service3{service1=com.javacode2018.lesson001.demo26.test4.Service1@7e07db1f, service2=com.javacode2018.lesson001.demo26.test4.Service2@1189dd52}

service3中标注@Autowired的2个属性都有值了，都被注入成功了。

#### 案例6：@Autowire标注字段，多个候选者的时候，按字段名称注入

##### IService接口

package com.javacode2018.lesson001.demo26.test5;  
  
public interface IService {  
}

##### 接口来2个实现

2个实现类上都标注了@Component注解，都会被注册到容器中。

###### Service0

package com.javacode2018.lesson001.demo26.test5;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service0 implements IService {  
}

###### Service1

package com.javacode2018.lesson001.demo26.test5;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 implements IService {  
}

##### 来个Service2

package com.javacode2018.lesson001.demo26.test5;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 @Autowired  
 private IService service1; //@1  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：标注了@Autowired注解，需要注入类型为IService类型的bean，满足这种类型的有2个：service0和service1

按照上面介绍的候选者查找过程，最后会注入和字段名称一样的bean，即：service1

##### 来个总的配置类，负责扫描当前包中的组件

package com.javacode2018.lesson001.demo26.test5;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig5 {  
}

##### 来个测试用例

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig5.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service0->com.javacode2018.lesson001.demo26.test5.Service0@36902638  
service1->com.javacode2018.lesson001.demo26.test5.Service1@223d2c72  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test5.Service1@223d2c72}

注意最后一行，service2中的service1被注入了bean：service1

#### 案例7：将指定类型的所有bean，注入到Collection、Map中

##### 注入到Collection中

**被注入的类型为Collection类型或者Collection子接口类型，注意必须是接口类型**，如：

Collection<IService>  
List<IService>  
Set<IService>

**会在容器中找到所有IService类型的bean，放到这个集合中**。

##### 注入到Map中

**被注入的类型为Map类型或者Map子接口类型，注意必须是接口类型**，如：

Map<String,IService>

**会在容器中找到所有IService类型的bean，放到这个Map中，key为bean的名称，value为bean对象**。

来看案例代码。

##### 来个接口

package com.javacode2018.lesson001.demo26.test6;  
  
public interface IService {  
}

##### 来2个实现类，标注@Component注解

###### Service0

package com.javacode2018.lesson001.demo26.test6;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service0 implements IService {  
}

###### Service1

package com.javacode2018.lesson001.demo26.test6;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 implements IService {  
}

##### 再来个类Service2

package com.javacode2018.lesson001.demo26.test6;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
import java.util.List;  
import java.util.Map;  
  
@Component  
public class Service2 {  
  
 @Autowired  
 private List<IService> services;  
  
 @Autowired  
 private Map<String, IService> serviceMap;  
  
 @Override  
 public String toString() {  
 return "Service2{\n" +  
 "services=" + services +  
 ", \n serviceMap=" + serviceMap +  
 '}';  
 }  
}

@1：注入IService类型的所有bean

@2：注入一个map

##### 来个总的配置类

package com.javacode2018.lesson001.demo26.test6;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig6 {  
}

##### 来个测试用例

@Test  
public void test6() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig6.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service0->com.javacode2018.lesson001.demo26.test6.Service0@1189dd52  
service1->com.javacode2018.lesson001.demo26.test6.Service1@36bc55de  
service2->Service2{  
services=[com.javacode2018.lesson001.demo26.test6.Service0@1189dd52, com.javacode2018.lesson001.demo26.test6.Service1@36bc55de],   
 serviceMap={service0=com.javacode2018.lesson001.demo26.test6.Service0@1189dd52, service1=com.javacode2018.lesson001.demo26.test6.Service1@36bc55de}}

注意看一下上面services和serviceMap的值。

#### @Autowired源码

spring使用下面这个类处理@Autowired注解

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor

### @Resource：注意依赖对象

#### 作用

**和@Autowired注解类似，也是用来注入依赖的对象的，spring容器会对bean中所有字段、方法进行遍历，标注有@Resource注解的，都会进行注入。**

看一下这个注解定义：

javax.annotation.Resource  
  
@Target({TYPE, FIELD, METHOD})  
@Retention(RUNTIME)  
public @interface Resource {  
 String name() default "";  
 ..其他不常用的参数省略  
}

这个注解是javax中定义的，并不是spring中定义的注解。

从定义上可以见，这个注解可以用在任何类型上面、字段、方法上面。

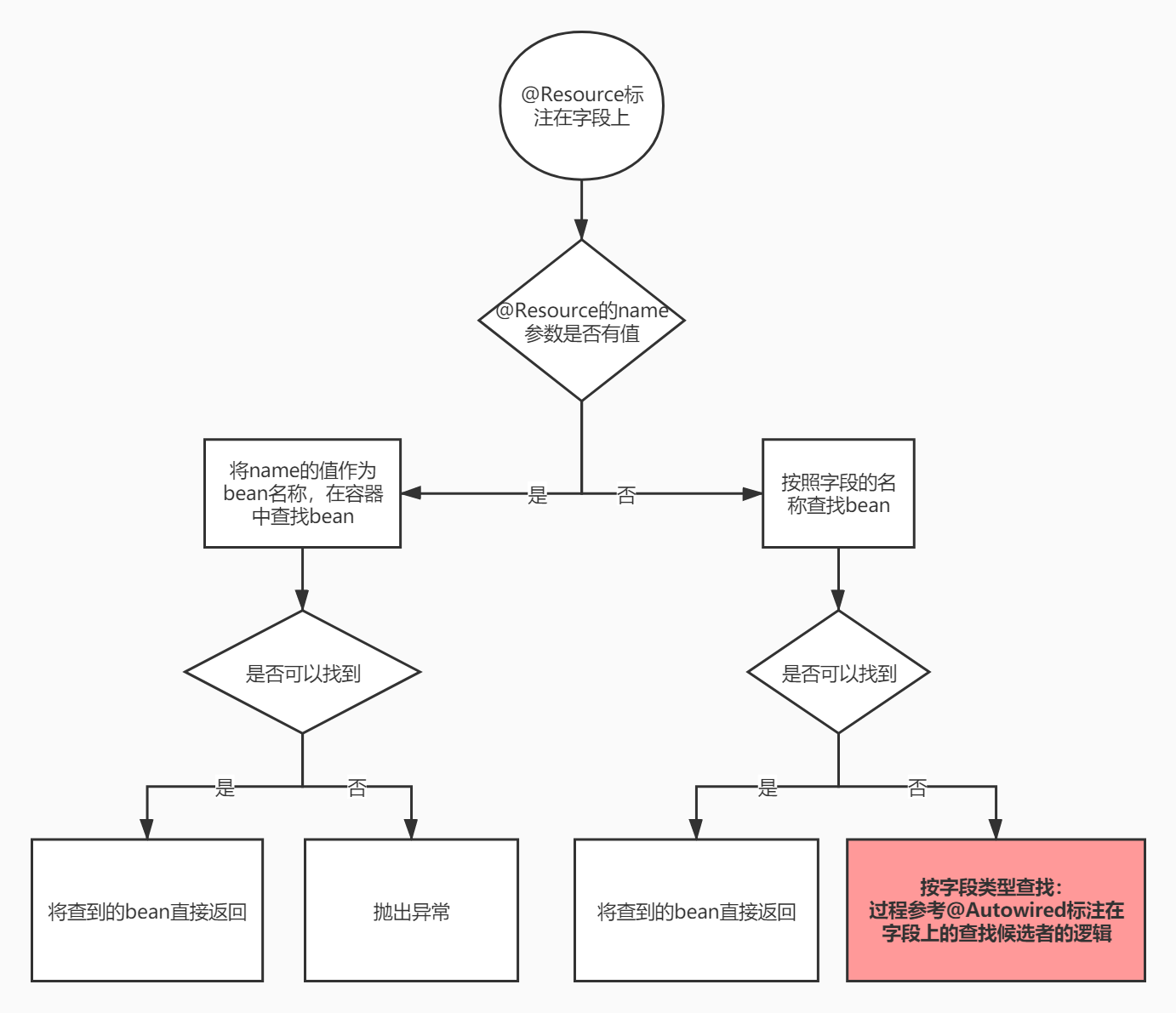
注意点：

**用在方法上的时候，方法参数只能有一个。**

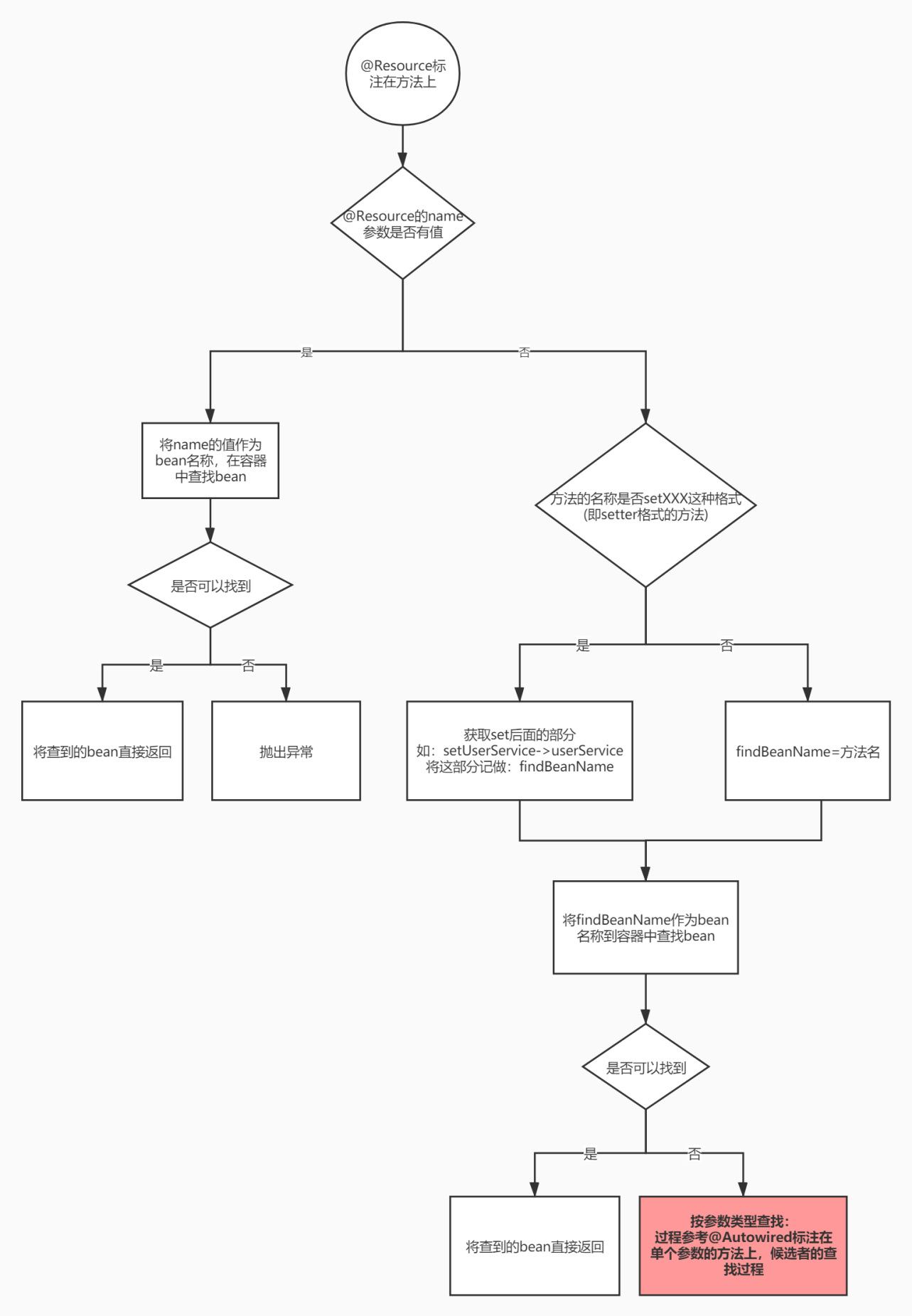
#### @Resource查找候选者的过程

**查找过程有点复杂，看不懂的可以先跳过，先看后面案例，本文看完之后，可以回头再来看这个过程。**

##### @Resource标注在字段上面：假定字段类型为一个自定义的普通的类型，候选者查找过程如下



##### @Autowired标注在方法上或者方法参数上面：假定参数类型为为一个自定义的普通的类型，候选者查找过程如下：



##### 将指定类型的所有bean注入到Collection中

如果被注入的对象是Collection类型的，可以指定泛型的类型，然后会按照上面的方式查找所有满足泛型类型所有的bean

##### 将指定类型的所有bean注入到Map中

如果被注入的对象是Map类型的，可以指定泛型的类型，key通常为String类型，value为需要查找的bean的类型，然后会按照上面方式查找所有注入value类型的bean，将bean的name作为key，bean对象作为value，放在HashMap中，然后注入。

##### @Resource查找候选者可以简化为

先按Resource的name值作为bean名称找->按名称（字段名称、方法名称、set属性名称）找->按类型找->通过限定符@Qualifier过滤->@Primary->@Priority->根据名称找（字段名称或者方法参数名称）

**概括为：先按名称找，然后按类型找**

#### 案例1：将@Resource标注在字段上

##### IService接口

package com.javacode2018.lesson001.demo26.test7;  
  
public interface IService {  
}

##### 2个实现类

###### Service0

package com.javacode2018.lesson001.demo26.test7;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service0 implements IService {  
}

@Component标注的bean名称默认为service0

###### Service1

package com.javacode2018.lesson001.demo26.test7;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 implements IService {  
}

@Component标注的bean名称默认为service1

再来一个类

package com.javacode2018.lesson001.demo26.test7;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
import javax.annotation.Resource;  
import java.util.List;  
import java.util.Map;  
  
@Component  
public class Service2 {  
  
 @Resource  
 private IService service1;//@1  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：字段名称为service1，按照字段名称查找bean，会找到Service1

##### 来个配置类

package com.javacode2018.lesson001.demo26.test7;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig7 {  
}

##### 测试用例

@Test  
public void test7() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig7.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service0->com.javacode2018.lesson001.demo26.test7.Service0@222545dc  
service1->com.javacode2018.lesson001.demo26.test7.Service1@5c5eefef  
service2->Service2{service1=com.javacode2018.lesson001.demo26.test7.Service1@5c5eefef}

最后一行可以看出注入了service1

##### 如果将Service2中的代码调整一下

@Resource  
private IService service0;

此时会注入service0这个bean

同样@Resource可以用在方法上，也可以将所有类型的bean注入到Collection、Map中，这里就不演示了，重点了解一下候选者查找的过程，使用上就比较简单了，@Resource的其他案例，大家可以自己写写练练。

下面来说另外几个注解，也是比较重要的。

#### @Resource源码

spring使用下面这个类处理@Resource注解

org.springframework.context.annotation.CommonAnnotationBeanPostProcessor

### @Qualifier：限定符

#### 作用

这个单词的意思是：限定符。

**可以在依赖注入查找候选者的过程中对候选者进行过滤。**

看一下其定义：

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE, ElementType.ANNOTATION\_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface Qualifier {  
  
 String value() default "";  
  
}

可以用在字段、方法、参数、任意类型、注解上面

有一个参数value

还是来看案例，通过案例理解更容易。

#### 案例1：用在类上

用在类上，你可以理解为给通过@Qulifier给这个bean打了一个标签。

##### 先来一个接口

package com.javacode2018.lesson001.demo26.test8;  
  
public interface IService {  
}

##### 来3个实现类

**前2个@Qulifier的value为tag1，第3个实现类为tag2**

##### Service1

package com.javacode2018.lesson001.demo26.test8;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
@Qualifier("tag1") //@1  
public class Service1 implements IService {  
}

@1：tag1

Service2

package com.javacode2018.lesson001.demo26.test8;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
@Qualifier("tag1")  
public class Service2 implements IService {  
}

@1：tag1

##### Service3

package com.javacode2018.lesson001.demo26.test8;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
@Qualifier("tag2")//@1  
public class Service3 implements IService {  
}

@1：tag2

##### 来一个类，来注入上面几个bean

package com.javacode2018.lesson001.demo26.test8;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
import java.util.Map;  
  
@Component  
public class InjectService {  
 @Autowired  
 @Qualifier("tag1") //@1  
 private Map<String, IService> serviceMap1;  
  
 @Autowired  
 @Qualifier("tag2") //@2  
 private Map<String, IService> serviceMap2;  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "serviceMap1=" + serviceMap1 +  
 ", serviceMap2=" + serviceMap2 +  
 '}';  
 }  
}

@1：限定符的值为tag1，此时会将类上限定符为tag1的所有bean注入进来

@2：限定符的值为tag2，此时会将类上限定符为tag2的所有bean注入进来

##### 来个配置类

package com.javacode2018.lesson001.demo26.test8;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig8 {  
}

##### 测试用例

@Test  
public void test8() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig8.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

运行输出

injectService->InjectService{serviceMap1={service1=com.javacode2018.lesson001.demo26.test8.Service1@9597028, service2=com.javacode2018.lesson001.demo26.test8.Service2@6069db50}, serviceMap2={service3=com.javacode2018.lesson001.demo26.test8.Service3@4efbca5a}}  
service1->com.javacode2018.lesson001.demo26.test8.Service1@9597028  
service2->com.javacode2018.lesson001.demo26.test8.Service2@6069db50  
service3->com.javacode2018.lesson001.demo26.test8.Service3@4efbca5a

注意第一行的输出，看一下serviceMap1和serviceMap2的值。

serviceMap1注入了@Qulifier的value为tag1的所有IService类型的bean

serviceMap1注入了@Qulifier的value为tag2的所有IService类型的bean

实现了bean分组的效果。

#### 案例2：@Autowired结合@Qulifier指定注入的bean

被注入的类型有多个的时候，可以使用@Qulifier来指定需要注入那个bean，将@Qulifier的value设置为需要注入bean的名称

看案例代码

##### 来个接口

package com.javacode2018.lesson001.demo26.test9;  
  
public interface IService {  
}

##### 有2个实现类

2个实现类上面没有使用@Qulifier注解了

###### Service1

package com.javacode2018.lesson001.demo26.test9;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 implements IService {  
}

###### Service2

package com.javacode2018.lesson001.demo26.test9;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 implements IService {  
}

我们可以知道上面2个bean的名称分别为：service1、service2

##### 来个类，注入IService类型的bean

package com.javacode2018.lesson001.demo26.test9;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
public class InjectService {  
 @Autowired  
 @Qualifier("service2") //@1  
 private IService service;  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "service=" + service +  
 '}';  
 }  
}

**@1：这里限定符的值为service2，容器中IService类型的bean有2个[service1和service2]，当类上没有标注@Qualifier注解的时候，可以理解为：bean的名称就是限定符的值，所以@1这里会匹配到service2**

##### 来个配置类

package com.javacode2018.lesson001.demo26.test9;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.stereotype.Component;  
  
@ComponentScan  
public class MainConfig9 {  
}

##### 来个测试用例

@Test  
public void test9() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig9.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

injectService->InjectService{service=com.javacode2018.lesson001.demo26.test9.Service2@223d2c72}  
service1->com.javacode2018.lesson001.demo26.test9.Service1@8f4ea7c  
service2->com.javacode2018.lesson001.demo26.test9.Service2@223d2c72

从第一行可以看出注入了service1

#### 案例3：用在方法参数上

##### 代码

package com.javacode2018.lesson001.demo26.test10;  
  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
public class InjectService {  
  
 private IService s1;  
 private IService s2;  
  
 @Autowired  
 public void injectBean(@Qualifier("service2") IService s1, @Qualifier("service1") IService s2) { //@1  
 this.s1 = s1;  
 this.s2 = s2;  
 }  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "s1=" + s1 +  
 ", s2=" + s2 +  
 '}';  
 }  
}

@1：方法上标注了@Autowired注解，说明会被注入依赖，2个参数上分别使用了限定符来指定具体需要注入哪个bean

##### 测试用例

@Test  
public void test10() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig10.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

injectService->InjectService{s1=com.javacode2018.lesson001.demo26.test10.Service2@55183b20, s2=com.javacode2018.lesson001.demo26.test10.Service1@4f83df68}  
service1->com.javacode2018.lesson001.demo26.test10.Service1@4f83df68  
service2->com.javacode2018.lesson001.demo26.test10.Service2@55183b20

第一行中的

s1：service2

s2：service1

#### 案例4：用在setter方法上

不管是用在setter方法还是普通方法上面，都是一样的效果

##### 代码

package com.javacode2018.lesson001.demo26.test11;  
  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
public class InjectService {  
  
 private IService s1;  
 private IService s2;  
  
 @Autowired  
 @Qualifier("service2")  
 public void setS1(IService s1) {  
 this.s1 = s1;  
 }  
  
 @Autowired  
 @Qualifier("service2")  
 public void setS2(IService s2) {  
 this.s2 = s2;  
 }  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "s1=" + s1 +  
 ", s2=" + s2 +  
 '}';  
 }  
}

上面2个setter方法上都有@Autowired注解，并且结合了@Qulifier注解来限定需要注入哪个bean

##### 测试用例

@Test  
public void test11() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig11.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

运行输出

injectService->InjectService{s1=com.javacode2018.lesson001.demo26.test11.Service2@35e2d654, s2=com.javacode2018.lesson001.demo26.test11.Service2@35e2d654}  
service1->com.javacode2018.lesson001.demo26.test11.Service1@1bd4fdd  
service2->com.javacode2018.lesson001.demo26.test11.Service2@35e2d654

输出中可以看出：s1为service2，s2为service1

### @Primary：设置为主要候选者

注入依赖的过程中，当有多个候选者的时候，可以指定哪个候选者为主要的候选者。

看一下其定义

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Primary {  
  
}

可以用在类上或者方法上面。

通常定义bean常见的有2种方式：

方式1：在类上标注@Component注解，此时可以配合@Primary，标注这个bean为主要候选者

方式2：在配置文件中使用@Bean注解标注方法，来注册bean，可以在@Bean标注的方法上加上@Primary，标注这个bean为主要候选bean。

看案例。

#### 案例1：用在类上

##### 来个接口

package com.javacode2018.lesson001.demo26.test12;  
  
public interface IService {  
}

##### 2个实现类

###### Service1

package com.javacode2018.lesson001.demo26.test12;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 implements IService {  
}

###### Service2

package com.javacode2018.lesson001.demo26.test12;  
  
import org.springframework.context.annotation.Primary;  
import org.springframework.stereotype.Component;  
  
@Component  
@Primary  
public class Service2 implements IService {  
}

Service2上面使用了@Primary，表示这是个主要的候选者

##### 再来个类，注入IService类型的bean

package com.javacode2018.lesson001.demo26.test12;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class InjectService {  
  
 @Autowired  
 private IService service1; //@1  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：容器中IService类型的bean有2个，但是service2为主要的候选者，所以此处会注入service2

##### 总的配置类

package com.javacode2018.lesson001.demo26.test12;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig12 {  
}

##### 测试用例

@Test  
public void test12() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig12.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

injectService->InjectService{service1=com.javacode2018.lesson001.demo26.test12.Service2@49ec71f8}  
service1->com.javacode2018.lesson001.demo26.test12.Service1@1d2adfbe  
service2->com.javacode2018.lesson001.demo26.test12.Service2@49ec71f8

#### 案例2：用在方法上，结合@Bean使用

##### 来个接口

package com.javacode2018.lesson001.demo26.test13;  
  
public interface IService {  
}

##### 2个实现类

###### Service1

package com.javacode2018.lesson001.demo26.test13;  
  
public class Service1 implements IService {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test13;  
  
public class Service2 implements IService {  
}

##### InjectService

package com.javacode2018.lesson001.demo26.test13;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class InjectService {  
  
 @Autowired  
 private IService service1;//@1  
  
 @Override  
 public String toString() {  
 return "InjectService{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

使用了@Autowired，需要注入

##### 来个配置类，通过@Bean定义上面3个类型的bean

package com.javacode2018.lesson001.demo26.test13;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
  
@Configuration  
public class MainConfig13 {  
  
 @Bean  
 public IService service1() {  
 return new Service1();  
 }  
  
 @Bean  
 @Primary //@1  
 public IService service2() {  
 return new Service2();  
 }  
  
 @Bean  
 public InjectService injectService() {  
 return new InjectService();  
 }  
}

上面是一个配置类，定义了3个bean

@1：这个bean被标注为主要的候选者

##### 来个测试用例

@Test  
public void test13() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig13.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo26.test13.Service1@6913c1fb  
service2->com.javacode2018.lesson001.demo26.test13.Service2@66d18979  
injectService->InjectService{service1=com.javacode2018.lesson001.demo26.test13.Service2@66d18979}

注意最后一行，service1注入的是service2这个bean

### @Bean定义bean时注入依赖的几种方式

#### 常见3种方式

1. 硬编码方式
2. @Autowired、@Resource的方式
3. @Bean标注的方法参数的方式

#### 方式1：硬编码方式

来3个类

##### Service1

package com.javacode2018.lesson001.demo26.test14;  
  
public class Service1 {  
}

##### Service2

package com.javacode2018.lesson001.demo26.test14;  
  
public class Service2 {  
}

##### Service3

package com.javacode2018.lesson001.demo26.test14;  
  
public class Service3 {  
 private Service1 service1;  
 private Service2 service2;  
  
 public Service1 getService1() {  
 return service1;  
 }  
  
 public void setService1(Service1 service1) {  
 this.service1 = service1;  
 }  
  
 public Service2 getService2() {  
 return service2;  
 }  
  
 public void setService2(Service2 service2) {  
 this.service2 = service2;  
 }  
  
 @Override  
 public String toString() {  
 return "Service3{" +  
 "service1=" + service1 +  
 ", service2=" + service2 +  
 '}';  
 }  
}

上面类中会用到service1和service2，提供了对应的setter方法，一会我们通过setter方法注入依赖对象

##### 来个配置类，通过@Bean的方式创建上面对象

package com.javacode2018.lesson001.demo26.test14;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class MainConfig14 {  
 @Bean  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 @Bean  
 public Service2 service2() {  
 return new Service2();  
 }  
  
 @Bean  
 public Service3 service3() {  
 Service3 service3 = new Service3(); //@0  
 service3.setService1(this.service1()); //@1  
 service3.setService2(this.service2()); //@2  
 return service3;  
 }  
}

上面代码中通过@Bean定义了3个bean

Service3中需要用到Service1和Service2，注意@1和@2直接调用当前方法获取另外2个bean，注入到service3中

##### 测试用例

@Test  
public void test14() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig14.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

##### 运行输出

service1->com.javacode2018.lesson001.demo26.test14.Service1@41a2befb  
service2->com.javacode2018.lesson001.demo26.test14.Service2@6c40365c  
service3->Service3{service1=com.javacode2018.lesson001.demo26.test14.Service1@41a2befb, service2=com.javacode2018.lesson001.demo26.test14.Service2@6c40365c}

#### 方式2：@Autowired、@Resource的方式

这种方式就不讲了直接在需要注入的对象上面加上这2个注解的任意一个就行了，可以参考文章前面的部分。

#### 方式3：@Bean标注的方法使用参数来进行注入

package com.javacode2018.lesson001.demo26.test15;  
  
import com.javacode2018.lesson001.demo26.test14.Service1;  
import com.javacode2018.lesson001.demo26.test14.Service2;  
import com.javacode2018.lesson001.demo26.test14.Service3;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class MainConfig15 {  
 @Bean  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 @Bean  
 public Service2 service2() {  
 return new Service2();  
 }  
  
 @Bean  
 public Service3 service3(Service1 s1, Service2 s2) { //@0  
 Service3 service3 = new Service3();  
 service3.setService1(s1); //@1  
 service3.setService2(s2); //@2  
 return service3;  
 }  
}

@0：这个地方是关键，方法上标注了@Bean，并且方法中是有参数的，spring调用这个方法创建bean的时候，会将参数中的两个参数注入进来。

注入对象的查找逻辑可以参考上面@Autowired标注方法时查找候选者的逻辑。

来个测试用例

@Test  
public void test15() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig15.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

运行输出

service1->com.javacode2018.lesson001.demo26.test14.Service1@4009e306  
service2->com.javacode2018.lesson001.demo26.test14.Service2@43c1b556  
service3->Service3{service1=com.javacode2018.lesson001.demo26.test14.Service1@4009e306, service2=com.javacode2018.lesson001.demo26.test14.Service2@43c1b556}

同样注入成功了。

#### 其他

##### @Bean标注的方法参数上使用@Autowired注解

@Bean  
public Service3 service3\_0(Service1 s1, @Autowired(required = false) Service2 s2) { //@0  
 Service3 service3 = new Service3();  
 service3.setService1(s1); //@1  
 service3.setService2(s2); //@2  
 return service3;  
}

@0：方法由2个参数，第二个参数上标注了@Autowired(required = false)，说明第二个参数候选者不是必须的，找不到会注入一个null对象；第一个参数候选者是必须的，找不到会抛出异常

##### @Bean结合@Qualifier

package com.javacode2018.lesson001.demo26.test17;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
import java.util.Map;  
  
@Configuration  
public class MainConfig17 {  
  
 @Bean  
 @Qualifier("tag1") //@1  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 @Bean  
 @Qualifier("tag1") //@2  
 public Service2 service2() {  
 return new Service2();  
 }  
  
 @Bean  
 @Qualifier("tag2") //@3  
 public Service3 service3() {  
 return new Service3();  
 }  
  
 @Bean  
 public InjectService injectService(@Qualifier("tag1") Map<String, IService> map1) { //@4  
 InjectService injectService = new InjectService();  
 injectService.setServiceMap1(map1);  
 return injectService;  
 }  
}

Service1,Service2,Service3都实现了IService接口

@1,@2,@3这3个方法上面使用了@Bean注解，用来定义3个bean，这3个方法上还是用了@Qualifier注解，用来给这些bean定义标签，service1()方法类似于下面的写法：

@Compontent  
@Qualifier("tag1")  
public class Service1 implements IService{  
}

再回到MainConfig17中的@4：参数中需要注入Map<String, IService>，会查找IService类型的bean，容器中有3个，但是这个参数前面加上了@Qualifier限定符，值为tag1，所以会通过这个过滤，最后满足的候选者为：[service1,service]

对应测试用例

@Test  
public void test17() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig17.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

运行输出

service1->com.javacode2018.lesson001.demo26.test17.Service1@1190200a  
service2->com.javacode2018.lesson001.demo26.test17.Service2@6a2f6f80  
service3->com.javacode2018.lesson001.demo26.test17.Service3@45b4c3a9  
injectService->InjectService{serviceMap1={service1=com.javacode2018.lesson001.demo26.test17.Service1@1190200a, service2=com.javacode2018.lesson001.demo26.test17.Service2@6a2f6f80}, serviceMap2=null}

注意最后一行serviceMap1，注入了service1和service2

### 泛型注入

#### 先来2个普通的类

##### UserModel

package com.javacode2018.lesson001.demo26.test18;  
  
public class UserModel {  
}

##### OrderModel

package com.javacode2018.lesson001.demo26.test18;  
  
public class OrderModel {  
}

记住上面2个普通的类UserModel和OrderModel，一会下面会用到。

#### 来个泛型接口

package com.javacode2018.lesson001.demo26.test18;  
  
public interface IDao<T> {  
}

上面是个泛型类，类名后面后尖括号

#### 来2个实现类

两个实现类都会标注@Compontent，交给spring容器管理

##### UserDao

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserDao implements IDao<UserModel> { //@1  
}

@1：指定了IDao后面泛型的类型为UserModel

##### OrderDao

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class OrderDao implements IDao<OrderModel> {//@1  
}

@1：指定了IDao后面泛型的类型为OrderModel

#### 在来个泛型类型

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class BaseService<T> {  
 @Autowired  
 private IDao<T> dao; //@1  
  
 public IDao<T> getDao() {  
 return dao;  
 }  
  
 public void setDao(IDao<T> dao) {  
 this.dao = dao;  
 }  
}

BaseService同样是个泛型类

@1：这个地方要注意了，上面使用了@Autowired，来注入IDao对象

#### BaseService来2个子类

两个子类都会标注@Compontent，交给spring容器管理

##### UserService

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserService extends BaseService<UserModel> {//@1  
}

@1：指定了BaseService后面泛型的类型为UserModel

##### OrderService

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class OrderService extends BaseService<OrderModel> {//@1  
}

@1：指定了BaseService后面泛型的类型为OrderModel

**UserService和OrderService继承了BaseService，所以一会BaseService中的dao属性会被注入，一会我们关注一下dao这个属性的值，会是什么样的**

#### 来个总的配置类

package com.javacode2018.lesson001.demo26.test18;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig18 {  
}

上面有@CompontentScan注解，会自动扫描当前包中的所有类，并进行自动注入

#### 来个测试用例

@Test  
public void test18() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig18.class);  
 System.out.println(context.getBean(UserService.class).getDao());  
 System.out.println(context.getBean(OrderService.class).getDao());  
}

上面代码中会将两个service中的dao输出，我们来看一下效果

#### 运行输出

com.javacode2018.lesson001.demo26.test18.UserDao@6adbc9d  
com.javacode2018.lesson001.demo26.test18.OrderDao@4550bb58

结果就是重点了，dao属性并没有指定具体需要注入那个bean，此时是根据尖括号中的泛型类型来匹配的，这个功能也是相当厉害的。

### 总结

**这篇文中内容比较多，每个案例大家都要去敲一遍，不清楚的，可以留言，或者直接微信中@我**

1. 需要掌握@Autowired注解和@Resource注解中候选者查找的过程
2. @Autowired：先通过类型找，然后通过名称找
3. @Resource：先通过名称找，然后通过类型找
4. @Autowired和@Resource，建议开发中使用@Autowired来实现依赖注入，spring的注解用起来更名正言顺一些
5. @Qulifier：限定符，可以用在类上；也可以用在依赖注入的地方，可以对候选者的查找进行过滤
6. @Primary：多个候选者的时候，可以标注某个候选者为主要的候选者
7. @Bean中注入依赖的3种方式需要掌握
8. 掌握泛型注入的使用
9. **主要还是掌握候选者的查找过程，过程熟悉了，其他的都是小意思，回头再去看看上面的几个查找的流程图。**

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册

## Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy

### 面试问题

1. @Scope是做什么的？常见的用法有几种？
2. @DependsOn是做什么的？常见的用法有几种？
3. @ImportResource干什么的？通常用在什么地方？
4. @Lazy做什么的，通常用在哪些地方？常见的用法有几种？

上面几个问题中涉及到了4个注解，都是比较常用的，下面我们来一一介绍。

### @Scope：指定bean的作用域

#### 用法

关于什么是bean的作用域，可以去看一下之前的一篇文章：[Spring系列第6篇：玩转bean scope，避免跳坑里！](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933960&idx=1&sn=f4308f8955f87d75963c379c2a0241f4&chksm=88621e76bf159760d404c253fa6716d3ffce4de8df0fc1d0d5dd0cf00a81bc170a30829ee58f&token=1314297026&lang=zh_CN&scene=21" \l "wechat_redirect)

@Scope用来配置bean的作用域，等效于bean xml中的bean元素中的scope属性。

看一下其源码：

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Scope {  
  
 @AliasFor("scopeName")  
 String value() default "";  
  
 @AliasFor("value")  
 String scopeName() default "";  
  
 ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;  
  
}

@Scope可以用在类上和方法上

参数：value和scopeName效果一样，用来指定bean作用域名称，如：singleton、prototype

#### 常见2种用法

1. 和@Compontent一起使用在类上
2. 和@Bean一起标注在方法上

#### 案例1：和@Compontent一起使用在类上

@Component  
@Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON)//@1  
public class ServiceA {  
}

上面定义了一个bean，作用域为单例的。

@1：ConfigurableBeanFactory接口中定义了几个作用域相关的常量，可以直接拿来使用，如：

String SCOPE\_SINGLETON = "singleton";

String SCOPE\_PROTOTYPE = "prototype";

#### 案例2：和@Bean一起标注在方法上

@Bean标注在方法上，可以通过这个方法来向spring容器中注册一个bean，在此方法上加上@Scope可以指定这个bean的作用域，如：

@Configurable  
public class MainConfig2 {  
 @Bean  
 @Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)  
 public ServiceA serviceA() {  
 return new ServiceA();  
 }  
}

### @DependsOn：指定当前bean依赖的bean

#### 用法

前面有篇文章中介绍了bean xml中depend-on的使用，建议先看一下：[Spring系列第9篇：depend-on到底是干什么的？](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933982&idx=1&sn=69a2906f5db1953030ff40225b3ac788&chksm=88621e60bf159776093398f89652fecc99fb78ddf6f7434afbe65f8511d3e41c65d729303507&token=880944996&lang=zh_CN&scene=21" \l "wechat_redirect)

@DependsOn等效于bean xml中的bean元素中的depend-on属性。

spring在创建bean的时候，如果bean之间没有依赖关系，那么spring容器很难保证bean实例创建的顺序，如果想确保容器在创建某些bean之前，需要先创建好一些其他的bean，可以通过@DependsOn来实现，**@DependsOn可以指定当前bean依赖的bean，通过这个可以确保@DependsOn指定的bean在当前bean创建之前先创建好**

看一下其源码：

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface DependsOn {  
  
 String[] value() default {};  
  
}

可以用在任意类型和方法上。

value：string类型的数组，用来指定当前bean需要依赖的bean名称，可以确保当前容器在创建被@DependsOn标注的bean之前，先将value指定的多个bean先创建好。

#### 常见2种用法

1. 和@Compontent一起使用在类上
2. 和@Bean一起标注在方法上

#### 案例1：和@Compontent一起使用在类上

下面定义3个bean：service1、service2、service3；service1需要依赖于其他2个service，需要确保容器在创建service1之前需要先将其他2个bean先创建好。

看代码：

##### Service2

package com.javacode2018.lesson001.demo27.test3;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 public Service2() {  
 System.out.println("create Service2");  
 }  
}

##### Service3

package com.javacode2018.lesson001.demo27.test3;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service3 {  
 public Service3() {  
 System.out.println("create Service3");  
 }  
}

##### Service1

package com.javacode2018.lesson001.demo27.test3;  
  
import org.springframework.context.annotation.DependsOn;  
import org.springframework.stereotype.Component;  
  
@DependsOn({"service2", "service3"}) //@1  
@Component  
public class Service1 {  
 public Service1() {  
 System.out.println("create Service1");  
 }  
}

@1：使用了@DependsOn，指定了2个bean：service2和service3，那么spring容器在创建上面这个service1的时候会先将@DependsOn中指定的2个bean先创建好

##### 来个配置类

package com.javacode2018.lesson001.demo27.test3;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig3 {  
}

##### 测试用例

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig3.class);  
 System.out.println(context.getBean(Service1.class));  
}

##### 运行输出

create Service2  
create Service3  
create Service1  
com.javacode2018.lesson001.demo27.test3.Service1@9f116cc

从输出中可以看到，spring容器在创建service1之前，先将service2和service3创建好了。

#### 案例2：和@Bean一起标注在方法上

下面通过配置文件的方式来创建bean，如下：

package com.javacode2018.lesson001.demo27.test4;  
  
import org.springframework.beans.factory.annotation.Configurable;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.DependsOn;  
  
@Configurable  
public class MainConfig4 {  
  
 @Bean  
 @DependsOn({"service2", "service3"})//@1  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 @Bean  
 public Service2 service2() {  
 return new Service2();  
 }  
  
 @Bean  
 public Service3 service3() {  
 return new Service3();  
 }  
  
}

上面是一个spring的配置类，类中3个方法定义了3个bean

@1：这个地方使用了@DependsOn，表示service1这个bean创建之前，会先创建好service2和service3

来个测试用例

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig4.class);  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo27.test4.Service1.class));  
}

运行输出

create Service2  
create Service3  
create Service1  
com.javacode2018.lesson001.demo27.test4.Service1@6e20b53a

### @ImportResource：配置类中导入bean定义的配置文件

#### 用法

有些项目，前期可能采用xml的方式配置bean，后期可能想采用spring注解的方式来重构项目，但是有些老的模块可能还是xml的方式，spring为了方便在注解方式中兼容老的xml的方式，提供了@ImportResource注解来引入bean定义的配置文件。

bean定义配置文件：目前我们主要介绍了xml的方式，还有一种properties文件的方式，以后我们会介绍，此时我们还是以引入bean xml来做说明。

看一下这个注解的定义：

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Documented  
public @interface ImportResource {  
  
 @AliasFor("locations")  
 String[] value() default {};  
  
 @AliasFor("value")  
 String[] locations() default {};  
  
 Class<? extends BeanDefinitionReader> reader() default BeanDefinitionReader.class;  
  
}

**通常将其用在配置类上。**

有3个参数：

* value和locations效果一样，只能配置其中一个，是一个string类型的数组，用来指定需要导入的配置文件的路径。
* reader：用来指定bean定义的读取器，目前我们知道的配置bean的方式有xml文件的方式，注解的方式，其实还有其他的方式，比如properties文件的方式，如果用其他的方式，你得告诉spring具体要用那种解析器去解析这个bean配置文件，这个解析器就是BeanDefinitionReader，以后我们讲BeanDefinition的时候再细说。

#### 资源文件路径的写法

通常我们的项是采用maven来组织的，配置文件一般会放在resources目录，这个目录中的文件被编译之后会在target/classes目录中。

spring中资源文件路径最常用的有2种写法：

1. **以classpath:开头**：检索目标为当前项目的classes目录
2. **以classpath\*:开头**：检索目标为当前项目的classes目录，以及项目中所有jar包中的目录，如果你确定jar不是检索目标，就不要用这种方式，由于需要扫描所有jar包，所以速度相对于第一种会慢一些

那我们再来说classpath:和classpath\*:后面的部分，后面的部分是确定资源文件的位置地方，几种常见的如下：

##### 相对路径的方式

classpath:com/javacode2018/lesson001/demo27/test5/beans.xml  
或者  
classpath\*:com/javacode2018/lesson001/demo27/test5/beans.xml

##### /：绝对路径的方式

classpath:/com/javacode2018/lesson001/demo27/test5/beans.xml

##### \*：文件通配符的方式

classpath:/com/javacode2018/lesson001/demo27/test5/beans-\*.xml

会匹配test5目录中所有以beans-开头的xml结尾的文件

##### \*：目录通配符的方式

classpath:/com/javacode2018/lesson001/demo27/\*/beans-\*.xml

会匹配demo27中所有子目录中所有以beans-开头的xml结尾的文件，注意这个地方只包含demo27的子目录，不包含子目录的子目录，不会进行递归

##### \*\*：递归任意子目录的方式

classpath:/com/javacode2018/\*\*/beans-\*.xml

\*\*会递归当前目录以及下面任意级的子目录

ok，继续回到@ImportResource上来，来看案例

#### 案例代码

来2个类，这两个类我们分别用2个xml来定义bean

##### ServiceA

package com.javacode2018.lesson001.demo27.test5;  
  
public class ServiceA {  
}

##### ServiceB

package com.javacode2018.lesson001.demo27.test5;  
  
public class ServiceB {  
}

##### beans1.xml来定义serviceA这个bean，如下

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceA" class="com.javacode2018.lesson001.demo27.test5.ServiceA"/>  
  
</beans>

##### beans2.xml来定义serviceB这个bean，如下

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="serviceB" class="com.javacode2018.lesson001.demo27.test5.ServiceB"/>  
  
</beans>

##### 下面来个配置类，来引入上面2个配置文件

package com.javacode2018.lesson001.demo27.test5;  
  
import org.springframework.beans.factory.annotation.Configurable;  
import org.springframework.context.annotation.ImportResource;  
  
@Configurable  
@ImportResource("classpath:/com/javacode2018/lesson001/demo27/test5/beans\*.xml")  
public class MainConfig5 {  
}

这个类上使用了@Configurable表示这是个配置类

并且使用了@ImportResource注解来导入上面2个配置文件

##### 来个测试用例加载上面这个配置类

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig5.class);  
 for (String beanName : context.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 }  
}

上面会输出MainConfig5配置类中所有定义的bean

##### 运行输出

mainConfig5->com.javacode2018.lesson001.demo27.test5.MainConfig5@4ec4f3a0  
serviceA->com.javacode2018.lesson001.demo27.test5.ServiceA@223191a6  
serviceB->com.javacode2018.lesson001.demo27.test5.ServiceB@49139829

从输出中可以看出2个xml中定义的bean也被注册了

### @Lazy：延迟初始化

#### 用法

@Lazy等效于bean xml中bean元素的lazy-init属性，可以实现bean的延迟初始化。

**所谓延迟初始化：就是使用到的时候才会去进行初始化。**

来看一下其定义：

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.PARAMETER, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Lazy {  
  
 boolean value() default true;  
  
}

可以用在任意类型、方法、构造器、参数、字段上面

参数：

value：boolean类型，用来配置是否应发生延迟初始化，默认为true。

#### 常用3种方式

1. 和@Compontent一起标注在类上，可以是这个类延迟初始化
2. 和@Configuration一起标注在配置类中，可以让当前配置类中通过@Bean注册的bean延迟初始化
3. 和@Bean一起使用，可以使当前bean延迟初始化

来看一下这3种方式案例代码。

#### 案例1：和@Compontent一起使用

##### Service1

package com.javacode2018.lesson001.demo27.test6;  
  
import org.springframework.context.annotation.Lazy;  
import org.springframework.stereotype.Component;  
  
@Component  
@Lazy //@1  
public class Service1 {  
 public Service1() {  
 System.out.println("创建Service1");  
 }  
}

@1：使用到了@Lazy，默认值为true，表示会被延迟初始化，在容器启动过程中不会被初始化，当从容器中查找这个bean的时候才会被初始化。

##### 配置类

package com.javacode2018.lesson001.demo27.test6;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig6 {  
}

##### 测试用例

@Test  
public void test6() {  
 System.out.println("准备启动spring容器");  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig6.class);  
 System.out.println("spring容器启动完毕");  
 System.out.println(context.getBean(com.javacode2018.lesson001.demo27.test6.Service1.class));  
}

运行输出

准备启动spring容器  
spring容器启动完毕  
创建Service1  
com.javacode2018.lesson001.demo27.test6.Service1@4fb61f4a

可以看出service1这个bean在spring容器启动过程中并没有被创建，而是在我们调用getBean进行查找的时候才进行创建的，此时起到了延迟创建的效果。

#### 案例2：和@Configuration一起使用加在配置类上

@Lazy和@Configuration一起使用，此时配置类中所有通过@Bean方式注册的bean都会被延迟初始化，不过也可以在@Bean标注的方法上使用@Lazy来覆盖配置类上的@Lazy配置，看下面代码：

##### 配置类MainConfig7

package com.javacode2018.lesson001.demo27.test7;  
  
import org.springframework.beans.factory.annotation.Configurable;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Lazy;  
  
@Lazy //@1  
@Configurable  
public class MainConfig7 {  
  
 @Bean  
 public String name() {  
 System.out.println("create bean:name");  
 return "马士兵老师";  
 }  
  
 @Bean  
 public String address() {  
 System.out.println("create bean:address");  
 return "上海市";  
 }  
  
 @Bean  
 @Lazy(false) //@2  
 public Integer age() {  
 System.out.println("create bean:age");  
 return 30;  
 }  
}

@1：配置类上使用了@Lazy，此时会对当前类中所有@Bean标注的方法生效

@2：这个方法上面使用到了@Lazy(false)，此时age这个bean不会被延迟初始化。其他2个bean会被延迟初始化。

##### 测试用例

@Test  
public void test7() {  
 System.out.println("准备启动spring容器");  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig7.class);  
 System.out.println("spring容器启动完毕");  
  
 for (String beanName : Arrays.asList("name", "age", "address")) {  
 System.out.println("----------");  
 System.out.println("getBean:" + beanName + ",start");  
 System.out.println(String.format("%s->%s", beanName, context.getBean(beanName)));  
 System.out.println("getBean:" + beanName + ",end");  
 }  
}

上面会输出配置类中定义的3个bean的信息。

##### 运行输出

准备启动spring容器  
create bean:age  
spring容器启动完毕  
----------  
getBean:name,start  
create bean:name  
name->马士兵老师  
getBean:name,end  
----------  
getBean:age,start  
age->30  
getBean:age,end  
----------  
getBean:address,start  
create bean:address  
address->上海市  
getBean:address,end

输出中可以看到age是在容器启动过程中创建的，其他2个是在通过getBean查找的时候才创建的。

### 总结

1. 本文介绍的几个注解也算是比较常用的，大家一定要熟悉他们的用法
2. @Scope：用来定义bean 的作用域；2种用法：第1种：标注在类上；第2种：和@Bean一起标注在方法上
3. @DependsOn：用来指定当前bean依赖的bean，可以确保在创建当前bean之前，先将依赖的bean创建好；2种用法：第1种：标注在类上；第2种：和@Bean一起标注在方法上
4. @ImportResource：标注在配置类上，用来引入bean定义的配置文件
5. @Lazy：让bean延迟初始化；常见3种用法：第1种：标注在类上；第2种：标注在配置类上，会对配置类中所有的@Bean标注的方法有效；第3种：和@Bean一起标注在方法上

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）

## Spring系列第23篇：Bean生命周期详解

### 来看几个问题

1. 想不想面试的时候每月多1万？
2. 想不想进入公司架构组？
3. 想不想成为项目组的负责人？
4. 想不想成为spring的高手，超越99%的对手？

那么本文内容是你必须要掌握的。

本文主要详解bean的生命周期，分为12个环节，每个环节中spring都提供了一些扩展点，我们都将进行详细说明，让大家全面掌握这块的知识。

### Spring bean生命周期12个环节

1. 阶段1：Bean元信息配置阶段
2. 阶段2：Bean元信息解析阶段
3. 阶段3：将Bean注册到容器中
4. 阶段4：BeanDefinition合并阶段
5. 阶段5：Bean Class加载阶段
6. 阶段6：Bean实例化阶段（2个小阶段）
   * Bean实例化前阶段
   * Bean实例化阶段
7. 阶段7：合并后的BeanDefinition处理
8. 阶段8：属性赋值阶段（3个小阶段）
   * Bean实例化后阶段
   * Bean属性赋值前阶段
   * Bean属性赋值阶段
9. 阶段9：Bean初始化阶段（5个小阶段）
   * Bean Aware接口回调阶段
   * Bean初始化前阶段
   * Bean初始化阶段
   * Bean初始化后阶段
10. 阶段10：所有单例bean初始化完成后阶段
11. 阶段11：Bean的使用阶段
12. 阶段12：Bean销毁前阶段
13. 阶段13：Bean销毁阶段

### 阶段1：Bean元信息配置阶段

这个阶段主要是bean信息的定义阶段。

#### Bean信息定义4种方式

* API的方式
* Xml文件方式
* properties文件的方式
* 注解的方式

#### API的方式

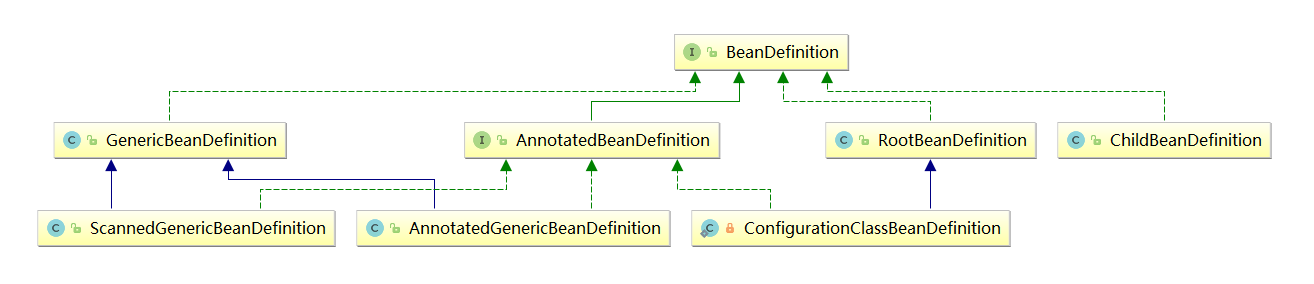
先来说这种方式，因为其他几种方式最终都会采用这种方式来定义bean配置信息。

**Spring容器启动的过程中，会将Bean解析成Spring内部的BeanDefinition结构**。 不管是是通过xml配置文件的<Bean>标签，还是通过注解配置的@Bean，还是@Compontent标注的类，还是扫描得到的类，它最终都会被解析成一个BeanDefinition对象，最后我们的Bean工厂就会根据这份Bean的定义信息，对bean进行实例化、初始化等等操作。

你可以把BeanDefinition丢给Bean工厂，然后Bean工厂就会根据这个信息帮你生产一个Bean实例，拿去使用。

BeanDefinition里面里面包含了bean定义的各种信息，如：bean对应的class、scope、lazy信息、dependOn信息、autowireCandidate（是否是候选对象）、primary（是否是主要的候选者）等信息。

BeanDefinition是个接口，有几个实现类，看一下类图：



##### BeanDefinition接口：bean定义信息接口

表示bean定义信息的接口，里面定义了一些获取bean定义配置信息的各种方法，来看一下源码：

public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {  
  
 /\*\*  
 \* 设置此bean的父bean名称（对应xml中bean元素的parent属性）  
 \*/  
 void setParentName(@Nullable String parentName);  
  
 /\*\*  
 \* 返回此bean定义时指定的父bean的名称  
 \*/  
 @Nullable  
 String getParentName();  
  
 /\*\*  
 \* 指定此bean定义的bean类名(对应xml中bean元素的class属性)  
 \*/  
 void setBeanClassName(@Nullable String beanClassName);  
  
 /\*\*  
 \* 返回此bean定义的当前bean类名  
 \* 注意，如果子定义重写/继承其父类的类名，则这不一定是运行时使用的实际类名。此外，这可能只是调用工厂方法的类，或者在调用方法的工厂bean引用的情况下，它甚至可能是空的。因此，不要认为这是运行时的最终bean类型，而只将其用于单个bean定义级别的解析目的。  
 \*/  
 @Nullable  
 String getBeanClassName();  
  
 /\*\*  
 \* 设置此bean的生命周期，如：singleton、prototype（对应xml中bean元素的scope属性）  
 \*/  
 void setScope(@Nullable String scope);  
  
 /\*\*  
 \* 返回此bean的生命周期，如：singleton、prototype  
 \*/  
 @Nullable  
 String getScope();  
  
 /\*\*  
 \* 设置是否应延迟初始化此bean（对应xml中bean元素的lazy属性）  
 \*/  
 void setLazyInit(boolean lazyInit);  
  
 /\*\*  
 \* 返回是否应延迟初始化此bean，只对单例bean有效  
 \*/  
 boolean isLazyInit();  
  
 /\*\*  
 \* 设置此bean依赖于初始化的bean的名称,bean工厂将保证dependsOn指定的bean会在当前bean初始化之前先初始化好  
 \*/  
 void setDependsOn(@Nullable String... dependsOn);  
  
 /\*\*  
 \* 返回此bean所依赖的bean名称  
 \*/  
 @Nullable  
 String[] getDependsOn();  
  
 /\*\*  
 \* 设置此bean是否作为其他bean自动注入时的候选者  
 \* autowireCandidate  
 \*/  
 void setAutowireCandidate(boolean autowireCandidate);  
  
 /\*\*  
 \* 返回此bean是否作为其他bean自动注入时的候选者  
 \*/  
 boolean isAutowireCandidate();  
  
 /\*\*  
 \* 设置此bean是否为自动注入的主要候选者  
 \* primary：是否为主要候选者  
 \*/  
 void setPrimary(boolean primary);  
  
 /\*\*  
 \* 返回此bean是否作为自动注入的主要候选者  
 \*/  
 boolean isPrimary();  
  
 /\*\*  
 \* 指定要使用的工厂bean（如果有）。这是要对其调用指定工厂方法的bean的名称。  
 \* factoryBeanName：工厂bean名称  
 \*/  
 void setFactoryBeanName(@Nullable String factoryBeanName);  
  
 /\*\*  
 \* 返回工厂bean名称（如果有）（对应xml中bean元素的factory-bean属性）  
 \*/  
 @Nullable  
 String getFactoryBeanName();  
  
 /\*\*  
 \* 指定工厂方法（如果有）。此方法将使用构造函数参数调用，如果未指定任何参数，则不使用任何参数调用。该方法将在指定的工厂bean（如果有的话）上调用，或者作为本地bean类上的静态方法调用。  
 \* factoryMethodName：工厂方法名称  
 \*/  
 void setFactoryMethodName(@Nullable String factoryMethodName);  
  
 /\*\*  
 \* 返回工厂方法名称（对应xml中bean的factory-method属性）  
 \*/  
 @Nullable  
 String getFactoryMethodName();  
  
 /\*\*  
 \* 返回此bean的构造函数参数值  
 \*/  
 ConstructorArgumentValues getConstructorArgumentValues();  
  
 /\*\*  
 \* 是否有构造器参数值设置信息（对应xml中bean元素的<constructor-arg />子元素）  
 \*/  
 default boolean hasConstructorArgumentValues() {  
 return !getConstructorArgumentValues().isEmpty();  
 }  
  
 /\*\*  
 \* 获取bean定义是配置的属性值设置信息  
 \*/  
 MutablePropertyValues getPropertyValues();  
  
 /\*\*  
 \* 这个bean定义中是否有属性设置信息（对应xml中bean元素的<property />子元素）  
 \*/  
 default boolean hasPropertyValues() {  
 return !getPropertyValues().isEmpty();  
 }  
  
 /\*\*  
 \* 设置bean初始化方法名称  
 \*/  
 void setInitMethodName(@Nullable String initMethodName);  
  
 /\*\*  
 \* bean初始化方法名称  
 \*/  
 @Nullable  
 String getInitMethodName();  
  
 /\*\*  
 \* 设置bean销毁方法的名称  
 \*/  
 void setDestroyMethodName(@Nullable String destroyMethodName);  
  
 /\*\*  
 \* bean销毁的方法名称  
 \*/  
 @Nullable  
 String getDestroyMethodName();  
  
 /\*\*  
 \* 设置bean的role信息  
 \*/  
 void setRole(int role);  
  
 /\*\*  
 \* bean定义的role信息  
 \*/  
 int getRole();  
  
 /\*\*  
 \* 设置bean描述信息  
 \*/  
 void setDescription(@Nullable String description);  
  
 /\*\*  
 \* bean描述信息  
 \*/  
 @Nullable  
 String getDescription();  
  
 /\*\*  
 \* bean类型解析器  
 \*/  
 ResolvableType getResolvableType();  
  
 /\*\*  
 \* 是否是单例的bean  
 \*/  
 boolean isSingleton();  
  
 /\*\*  
 \* 是否是多列的bean  
 \*/  
 boolean isPrototype();  
  
 /\*\*  
 \* 对应xml中bean元素的abstract属性，用来指定是否是抽象的  
 \*/  
 boolean isAbstract();  
  
 /\*\*  
 \* 返回此bean定义来自的资源的描述（以便在出现错误时显示上下文）  
 \*/  
 @Nullable  
 String getResourceDescription();  
  
 @Nullable  
 BeanDefinition getOriginatingBeanDefinition();  
  
}

BeanDefinition接口上面还继承了2个接口：

* AttributeAccessor
* BeanMetadataElement

###### AttributeAccessor接口：属性访问接口

public interface AttributeAccessor {  
  
 /\*\*  
 \* 设置属性->值  
 \*/  
 void setAttribute(String name, @Nullable Object value);  
  
 /\*\*  
 \* 获取某个属性对应的值  
 \*/  
 @Nullable  
 Object getAttribute(String name);  
  
 /\*\*  
 \* 移除某个属性  
 \*/  
 @Nullable  
 Object removeAttribute(String name);  
  
 /\*\*  
 \* 是否包含某个属性  
 \*/  
 boolean hasAttribute(String name);  
  
 /\*\*  
 \* 返回所有的属性名称  
 \*/  
 String[] attributeNames();  
  
}

这个接口相当于key->value数据结构的一种操作，BeanDefinition继承这个，内部实际上是使用了LinkedHashMap来实现这个接口中的所有方法，通常我们通过这些方法来保存BeanDefinition定义过程中产生的一些附加信息。

###### BeanMetadataElement接口

看一下其源码：

public interface BeanMetadataElement {  
  
 @Nullable  
 default Object getSource() {  
 return null;  
 }  
  
}

BeanDefinition继承这个接口，getSource返回BeanDefinition定义的来源，比如我们通过xml定义BeanDefinition的，此时getSource就表示定义bean的xml资源；若我们通过api的方式定义BeanDefinition，我们可以将source设置为定义BeanDefinition时所在的类，出错时，可以根据这个来源方便排错。

##### RootBeanDefinition类：表示根bean定义信息

通常bean中没有父bean的就使用这种表示

##### ChildBeanDefinition类：表示子bean定义信息

如果需要指定父bean的，可以使用ChildBeanDefinition来定义子bean的配置信息，里面有个parentName属性，用来指定父bean的名称。

##### GenericBeanDefinition类：通用的bean定义信息

既可以表示没有父bean的bean配置信息，也可以表示有父bean的子bean配置信息，这个类里面也有parentName属性，用来指定父bean的名称。

##### ConfigurationClassBeanDefinition类：表示通过配置类中@Bean方法定义bean信息

可以通过配置类中使用@Bean来标注一些方法，通过这些方法来定义bean，这些方法配置的bean信息最后会转换为ConfigurationClassBeanDefinition类型的对象

##### AnnotatedBeanDefinition接口：表示通过注解的方式定义的bean信息

里面有个方法

AnnotationMetadata getMetadata();

用来获取定义这个bean的类上的所有注解信息。

##### BeanDefinitionBuilder：构建BeanDefinition的工具类

spring中为了方便操作BeanDefinition，提供了一个类：BeanDefinitionBuilder，内部提供了很多静态方法，通过这些方法可以非常方便的组装BeanDefinition对象，下面我们通过案例来感受一下。

##### 案例1：组装一个简单的bean

###### 来个简单的类

package com.javacode2018.lesson002.demo1;  
  
public class Car {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Car{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

###### 测试用例

@Test  
public void test1() {  
 //指定class  
 BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.rootBeanDefinition(Car.class.getName());  
 //获取BeanDefinition  
 BeanDefinition beanDefinition = beanDefinitionBuilder.getBeanDefinition();  
 System.out.println(beanDefinition);  
}

等效于

<bean class="com.javacode2018.lesson002.demo1.Car" />

###### 运行输出

Root bean: class [com.javacode2018.lesson002.demo1.Car]; scope=; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null

##### 案例2：组装一个有属性的bean

###### 代码

@Test  
public void test2() {  
 //指定class  
 BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.rootBeanDefinition(Car.class.getName());  
 //设置普通类型属性  
 beanDefinitionBuilder.addPropertyValue("name", "奥迪"); //@1  
 //获取BeanDefinition  
 BeanDefinition carBeanDefinition = beanDefinitionBuilder.getBeanDefinition();  
 System.out.println(carBeanDefinition);  
  
 //创建spring容器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory(); //@2  
 //调用registerBeanDefinition向容器中注册bean  
 factory.registerBeanDefinition("car", carBeanDefinition); //@3  
 Car bean = factory.getBean("car", Car.class); //@4  
 System.out.println(bean);  
}

@1：调用addPropertyValue给Car中的name设置值

@2：创建了一个spring容器

@3：将carBeanDefinition这个bean配置信息注册到spring容器中，bean的名称为car

@4：从容器中获取car这个bean，最后进行输出

###### 运行输出

Root bean: class [com.javacode2018.lesson002.demo1.Car]; scope=; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
Car{name='奥迪'}

第二行输出了从容器中获取的car这个bean实例对象。

##### 案例3：组装一个有依赖关系的bean

###### 再来个类

下面这个类中有个car属性，我们通过spring将这个属性注入进来。

package com.javacode2018.lesson002.demo1;  
  
public class User {  
 private String name;  
  
 private Car car;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Car getCar() {  
 return car;  
 }  
  
 public void setCar(Car car) {  
 this.car = car;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "name='" + name + '\'' +  
 ", car=" + car +  
 '}';  
 }  
}

###### 重点代码

@Test  
public void test3() {  
 //先创建car这个BeanDefinition  
 BeanDefinition carBeanDefinition = BeanDefinitionBuilder.rootBeanDefinition(Car.class.getName()).addPropertyValue("name", "奥迪").getBeanDefinition();  
 //创建User这个BeanDefinition  
 BeanDefinition userBeanDefinition = BeanDefinitionBuilder.rootBeanDefinition(User.class.getName()).  
 addPropertyValue("name", "马士兵老师").  
 addPropertyReference("car", "car"). //@1  
 getBeanDefinition();  
   
 //创建spring容器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 //调用registerBeanDefinition向容器中注册bean  
 factory.registerBeanDefinition("car", carBeanDefinition);   
 factory.registerBeanDefinition("user", userBeanDefinition);  
 System.out.println(factory.getBean("car"));  
 System.out.println(factory.getBean("user"));  
}

@1：注入依赖的bean，需要使用addPropertyReference方法，2个参数，第一个为属性的名称，第二个为需要注入的bean的名称

上面代码等效于

<bean id="car" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="奥迪"/>  
</bean>  
  
<bean id="user" class="com.javacode2018.lesson002.demo1.User">  
 <property name="name" value="马士兵老师"/>  
 <property name="car" ref="car"/>  
</bean>

###### 运行输出

Car{name='奥迪'}  
User{name='马士兵老师', car=Car{name='奥迪'}}

##### 案例4：来2个有父子关系的bean

@Test  
public void test4() {  
 //先创建car这个BeanDefinition  
 BeanDefinition carBeanDefinition1 = BeanDefinitionBuilder.  
 genericBeanDefinition(Car.class).  
 addPropertyValue("name", "保时捷").  
 getBeanDefinition();  
  
 BeanDefinition carBeanDefinition2 = BeanDefinitionBuilder.  
 genericBeanDefinition(). //内部生成一个GenericBeanDefinition对象  
 setParentName("car1"). //@1：设置父bean的名称为car1  
 getBeanDefinition();  
  
 //创建spring容器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 //调用registerBeanDefinition向容器中注册bean  
 //注册car1->carBeanDefinition1  
 factory.registerBeanDefinition("car1", carBeanDefinition1);  
 //注册car2->carBeanDefinition2  
 factory.registerBeanDefinition("car2", carBeanDefinition2);  
 //从容器中获取car1  
 System.out.println(String.format("car1->%s", factory.getBean("car1")));  
 //从容器中获取car2  
 System.out.println(String.format("car2->%s", factory.getBean("car2")));  
}

等效于

<bean id="car1" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="保时捷"/>  
</bean>  
<bean id="car2" parent="car1" />

###### 运行输出

car1->Car{name='保时捷'}  
car2->Car{name='保时捷'}

##### 案例5：通过api设置（Map、Set、List）属性

下面我们来演示注入List、Map、Set，内部元素为普通类型及其他bean元素。

###### 来个类

package com.javacode2018.lesson002.demo1;  
  
import java.util.List;  
import java.util.Map;  
import java.util.Set;  
  
public class CompositeObj {  
  
 private String name;  
 private Integer salary;  
  
 private Car car1;  
 private List<String> stringList;  
 private List<Car> carList;  
  
 private Set<String> stringSet;  
 private Set<Car> carSet;  
  
 private Map<String, String> stringMap;  
 private Map<String, Car> stringCarMap;  
  
 //此处省略了get和set方法，大家写的时候记得补上  
  
 @Override  
 public String toString() {  
 return "CompositeObj{" +  
 "name='" + name + '\'' +  
 "\n\t\t\t, salary=" + salary +  
 "\n\t\t\t, car1=" + car1 +  
 "\n\t\t\t, stringList=" + stringList +  
 "\n\t\t\t, carList=" + carList +  
 "\n\t\t\t, stringSet=" + stringSet +  
 "\n\t\t\t, carSet=" + carSet +  
 "\n\t\t\t, stringMap=" + stringMap +  
 "\n\t\t\t, stringCarMap=" + stringCarMap +  
 '}';  
 }  
}

**注意：上面省略了get和set方法，大家写的时候记得补上**

###### 先用xml来定义一个CompositeObj的bean，如下

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="car1" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="奥迪"/>  
 </bean>  
  
 <bean id="car2" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="保时捷"/>  
 </bean>  
  
 <bean id="compositeObj" class="com.javacode2018.lesson002.demo1.CompositeObj">  
 <property name="name" value="马士兵老师"/>  
 <property name="salary" value="50000"/>  
 <property name="car1" ref="car1"/>  
 <property name="stringList">  
 <list>  
 <value>java高并发系列</value>  
 <value>mysql系列</value>  
 <value>maven高手系列</value>  
 </list>  
 </property>  
 <property name="carList">  
 <list>  
 <ref bean="car1"/>  
 <ref bean="car2"/>  
 </list>  
 </property>  
 <property name="stringSet">  
 <set>  
 <value>java高并发系列</value>  
 <value>mysql系列</value>  
 <value>maven高手系列</value>  
 </set>  
 </property>  
 <property name="carSet">  
 <set>  
 <ref bean="car1"/>  
 <ref bean="car2"/>  
 </set>  
 </property>  
 <property name="stringMap">  
 <map>  
 <entry key="系列1" value="java高并发系列"/>  
 <entry key="系列2" value="Maven高手系列"/>  
 <entry key="系列3" value="mysql系列"/>  
 </map>  
 </property>  
 <property name="stringCarMap">  
 <map>  
 <entry key="car1" value-ref="car1"/>  
 <entry key="car2" value-ref="car2"/>  
 </map>  
 </property>  
 </bean>  
</beans>

###### 下面我们采用纯api的方式实现，如下

@Test  
public void test5() {  
 //定义car1  
 BeanDefinition car1 = BeanDefinitionBuilder.  
 genericBeanDefinition(Car.class).  
 addPropertyValue("name", "奥迪").  
 getBeanDefinition();  
 //定义car2  
 BeanDefinition car2 = BeanDefinitionBuilder.  
 genericBeanDefinition(Car.class).  
 addPropertyValue("name", "保时捷").  
 getBeanDefinition();  
  
 //定义CompositeObj这个bean  
 //创建stringList这个属性对应的值  
 ManagedList<String> stringList = new ManagedList<>();  
 stringList.addAll(Arrays.asList("java高并发系列", "mysql系列", "maven高手系列"));  
  
 //创建carList这个属性对应的值,内部引用其他两个bean的名称[car1,car2]  
 ManagedList<RuntimeBeanReference> carList = new ManagedList<>();  
 carList.add(new RuntimeBeanReference("car1"));  
 carList.add(new RuntimeBeanReference("car2"));  
  
 //创建stringList这个属性对应的值  
 ManagedSet<String> stringSet = new ManagedSet<>();  
 stringSet.addAll(Arrays.asList("java高并发系列", "mysql系列", "maven高手系列"));  
  
 //创建carSet这个属性对应的值,内部引用其他两个bean的名称[car1,car2]  
 ManagedList<RuntimeBeanReference> carSet = new ManagedList<>();  
 carSet.add(new RuntimeBeanReference("car1"));  
 carSet.add(new RuntimeBeanReference("car2"));  
  
 //创建stringMap这个属性对应的值  
 ManagedMap<String, String> stringMap = new ManagedMap<>();  
 stringMap.put("系列1", "java高并发系列");  
 stringMap.put("系列2", "Maven高手系列");  
 stringMap.put("系列3", "mysql系列");  
  
 ManagedMap<String, RuntimeBeanReference> stringCarMap = new ManagedMap<>();  
 stringCarMap.put("car1", new RuntimeBeanReference("car1"));  
 stringCarMap.put("car2", new RuntimeBeanReference("car2"));  
  
  
 //下面我们使用原生的api来创建BeanDefinition  
 GenericBeanDefinition compositeObj = new GenericBeanDefinition();  
 compositeObj.setBeanClassName(CompositeObj.class.getName());  
 compositeObj.getPropertyValues().add("name", "马士兵老师").  
 add("salary", 50000).  
 add("car1", new RuntimeBeanReference("car1")).  
 add("stringList", stringList).  
 add("carList", carList).  
 add("stringSet", stringSet).  
 add("carSet", carSet).  
 add("stringMap", stringMap).  
 add("stringCarMap", stringCarMap);  
  
 //将上面bean 注册到容器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 factory.registerBeanDefinition("car1", car1);  
 factory.registerBeanDefinition("car2", car2);  
 factory.registerBeanDefinition("compositeObj", compositeObj);  
  
 //下面我们将容器中所有的bean输出  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, factory.getBean(beanName)));  
 }  
}

有几点需要说一下：

RuntimeBeanReference：用来表示bean引用类型，类似于xml中的ref

ManagedList：属性如果是List类型的，t需要用到这个类进行操作，这个类继承了ArrayList

ManagedSet：属性如果是Set类型的，t需要用到这个类进行操作，这个类继承了LinkedHashSet

ManagedMap：属性如果是Map类型的，t需要用到这个类进行操作，这个类继承了LinkedHashMap

上面也就是这几个类结合的结果。

###### 看一下效果，运行输出

car1->Car{name='奥迪'}  
car2->Car{name='保时捷'}  
compositeObj->CompositeObj{name='马士兵老师'  
 , salary=50000  
 , car1=Car{name='奥迪'}  
 , stringList=[java高并发系列, mysql系列, maven高手系列]  
 , carList=[Car{name='奥迪'}, Car{name='保时捷'}]  
 , stringSet=[java高并发系列, mysql系列, maven高手系列]  
 , carSet=[Car{name='奥迪'}, Car{name='保时捷'}]  
 , stringMap={系列1=java高并发系列, 系列2=Maven高手系列, 系列3=mysql系列}  
 , stringCarMap={car1=Car{name='奥迪'}, car2=Car{name='保时捷'}}}

#### Xml文件方式

这种方式已经讲过很多次了，大家也比较熟悉，即通过xml的方式来定义bean，如下

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="bean名称" class="bean完整类名"/>  
  
</beans>

xml中的bean配置信息会被解析器解析为BeanDefinition对象，一会在第二阶段详解。

#### properties文件的方式

这种方式估计大家比较陌生，将bean定义信息放在properties文件中，然后通过解析器将配置信息解析为BeanDefinition对象。

properties内容格式如下：

employee.(class)=MyClass // 等同于：<bean class="MyClass" />  
employee.(abstract)=true // 等同于：<bean abstract="true" />  
employee.group=Insurance // 为属性设置值，等同于：<property name="group" value="Insurance" />  
employee.usesDialUp=false // 为employee这个bean中的usesDialUp属性设置值,等同于：等同于：<property name="usesDialUp" value="false" />  
  
salesrep.(parent)=employee // 定义了一个id为salesrep的bean，指定父bean为employee，等同于：<bean id="salesrep" parent="employee" />  
salesrep.(lazy-init)=true // 设置延迟初始化，等同于：<bean lazy-init="true" />  
salesrep.manager(ref)=tony // 设置这个bean的manager属性值，是另外一个bean，名称为tony，等同于：<property name="manager" ref="tony" />  
salesrep.department=Sales // 等同于：<property name="department" value="Sales" />  
  
techie.(parent)=employee // 定义了一个id为techie的bean，指定父bean为employee，等同于：<bean id="techie" parent="employee" />  
techie.(scope)=prototype // 设置bean的作用域，等同于<bean scope="prototype" />  
techie.manager(ref)=jeff // 等同于：<property name="manager" ref="jeff" />  
techie.department=Engineering // <property name="department" value="Engineering" />  
techie.usesDialUp=true // <property name="usesDialUp" value="true" />  
  
ceo.$0(ref)=secretary // 设置构造函数第1个参数值，等同于：<constructor-arg index="0" ref="secretary" />  
ceo.$1=1000000 // 设置构造函数第2个参数值，等同于：<constructor-arg index="1" value="1000000" />

#### 注解的方式

常见的2种：

1. 类上标注@Compontent注解来定义一个bean
2. 配置类中使用@Bean注解来定义bean

#### 小结

**bean注册者只识别BeanDefinition对象，不管什么方式最后都会将这些bean定义的信息转换为BeanDefinition对象，然后注册到spring容器中。**

### 阶段2：Bean元信息解析阶段

Bean元信息的解析就是将各种方式定义的bean配置信息解析为BeanDefinition对象。

#### Bean元信息的解析主要有3种方式

1. xml文件定义bean的解析
2. properties文件定义bean的解析
3. 注解方式定义bean的解析

#### XML方式解析：XmlBeanDefinitionReader

spring中提供了一个类XmlBeanDefinitionReader，将xml中定义的bean解析为BeanDefinition对象。

直接来看案例代码

##### 来一个bean xml配置文件

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="car" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="奥迪"/>  
 </bean>  
  
 <bean id="car1" class="com.javacode2018.lesson002.demo1.Car">  
 <property name="name" value="保时捷"/>  
 </bean>  
  
 <bean id="car2" parent="car1"/>  
  
 <bean id="user" class="com.javacode2018.lesson002.demo1.User">  
 <property name="name" value="马士兵老师"/>  
 <property name="car" ref="car1"/>  
 </bean>  
</beans>

上面注册了4个bean，不多解释了。

##### 将bean xml解析为BeanDefinition对象

/\*\*  
 \* xml方式bean配置信息解析  
 \*/  
@Test  
public void test1() {  
 //定义一个spring容器，这个容器默认实现了BeanDefinitionRegistry，所以本身就是一个bean注册器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //定义一个xml的BeanDefinition读取器，需要传递一个BeanDefinitionRegistry（bean注册器）对象  
 XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(factory);  
  
 //指定bean xml配置文件的位置  
 String location = "classpath:/com/javacode2018/lesson002/demo2/beans.xml";  
 //通过XmlBeanDefinitionReader加载bean xml文件，然后将解析产生的BeanDefinition注册到容器容器中  
 int countBean = xmlBeanDefinitionReader.loadBeanDefinitions(location);  
 System.out.println(String.format("共注册了 %s 个bean", countBean));  
  
 //打印出注册的bean的配置信息  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 //通过名称从容器中获取对应的BeanDefinition信息  
 BeanDefinition beanDefinition = factory.getBeanDefinition(beanName);  
 //获取BeanDefinition具体使用的是哪个类  
 String beanDefinitionClassName = beanDefinition.getClass().getName();  
 //通过名称获取bean对象  
 Object bean = factory.getBean(beanName);  
 //打印输出  
 System.out.println(beanName + ":");  
 System.out.println(" beanDefinitionClassName：" + beanDefinitionClassName);  
 System.out.println(" beanDefinition：" + beanDefinition);  
 System.out.println(" bean：" + bean);  
 }  
}

上面注释比较详细，这里就不解释了。

注意一点：创建XmlBeanDefinitionReader的时候需要传递一个bean注册器(BeanDefinitionRegistry)，解析过程中生成的BeanDefinition会丢到bean注册器中。

##### 运行输出

共注册了 4 个bean  
car:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.Car]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo2/beans.xml]  
 bean：Car{name='奥迪'}  
car1:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.Car]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo2/beans.xml]  
 bean：Car{name='保时捷'}  
car2:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean with parent 'car1': class [null]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo2/beans.xml]  
 bean：Car{name='保时捷'}  
user:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.User]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo2/beans.xml]  
 bean：User{name='马士兵老师', car=Car{name='奥迪'}}

上面的输出认真看一下，这几个BeanDefinition都是GenericBeanDefinition这种类型的，也就是说xml中定义的bean被解析之后都是通过GenericBeanDefinition这种类型表示的。

#### properties文件定义bean的解析：PropertiesBeanDefinitionReader

spring中提供了一个类XmlBeanDefinitionReader，将xml中定义的bean解析为BeanDefinition对象，过程和xml的方式类似。

来看案例代码。

下面通过properties文件的方式实现上面xml方式定义的bean。

##### 来个properties文件：beans.properties

car.(class)=com.javacode2018.lesson002.demo1.Car  
car.name=奥迪  
  
car1.(class)=com.javacode2018.lesson002.demo1.Car  
car1.name=保时捷  
  
car2.(parent)=car1  
  
user.(class)=com.javacode2018.lesson002.demo1.User  
user.name=马士兵老师  
user.car(ref)=car

##### 将bean properties文件解析为BeanDefinition对象

/\*\*  
 \* properties文件方式bean配置信息解析  
 \*/  
@Test  
public void test2() {  
 //定义一个spring容器，这个容器默认实现了BeanDefinitionRegistry，所以本身就是一个bean注册器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //定义一个properties的BeanDefinition读取器，需要传递一个BeanDefinitionRegistry（bean注册器）对象  
 PropertiesBeanDefinitionReader propertiesBeanDefinitionReader = new PropertiesBeanDefinitionReader(factory);  
  
 //指定bean xml配置文件的位置  
 String location = "classpath:/com/javacode2018/lesson002/demo2/beans.properties";  
 //通过PropertiesBeanDefinitionReader加载bean properties文件，然后将解析产生的BeanDefinition注册到容器容器中  
 int countBean = propertiesBeanDefinitionReader.loadBeanDefinitions(location);  
 System.out.println(String.format("共注册了 %s 个bean", countBean));  
  
 //打印出注册的bean的配置信息  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 //通过名称从容器中获取对应的BeanDefinition信息  
 BeanDefinition beanDefinition = factory.getBeanDefinition(beanName);  
 //获取BeanDefinition具体使用的是哪个类  
 String beanDefinitionClassName = beanDefinition.getClass().getName();  
 //通过名称获取bean对象  
 Object bean = factory.getBean(beanName);  
 //打印输出  
 System.out.println(beanName + ":");  
 System.out.println(" beanDefinitionClassName：" + beanDefinitionClassName);  
 System.out.println(" beanDefinition：" + beanDefinition);  
 System.out.println(" bean：" + bean);  
 }  
}

##### 运行输出

user:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.User]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：User{name='马士兵老师', car=Car{name='奥迪'}}  
car1:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.Car]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：Car{name='保时捷'}  
car:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo1.Car]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：Car{name='奥迪'}  
car2:  
 beanDefinitionClassName：org.springframework.beans.factory.support.GenericBeanDefinition  
 beanDefinition：Generic bean with parent 'car1': class [null]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：Car{name='保时捷'}

输出和xml方式输出基本上一致。

properties方式使用起来并不是太方便，所以平时我们很少看到有人使用。

#### 注解方式：PropertiesBeanDefinitionReader

注解的方式定义的bean，需要使用PropertiesBeanDefinitionReader这个类来进行解析，方式也和上面2种方式类似，直接来看案例。

##### 通过注解来标注2个类

###### Service1

package com.javacode2018.lesson002.demo2;  
  
  
import org.springframework.beans.factory.config.ConfigurableBeanFactory;  
import org.springframework.context.annotation.Lazy;  
import org.springframework.context.annotation.Primary;  
import org.springframework.context.annotation.Scope;  
  
@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)  
@Primary  
@Lazy  
public class Service1 {  
}

这个类上面使用了3个注解，这些注解前面都介绍过，可以用来配置bean的信息

上面这个bean是个多例的。

###### Service2

package com.javacode2018.lesson002.demo2;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class Service2 {  
  
 @Autowired  
 private Service1 service1; //@1  
  
 @Override  
 public String toString() {  
 return "Service2{" +  
 "service1=" + service1 +  
 '}';  
 }  
}

@1：标注了@Autowired，说明需要注入这个对象

##### 注解定义的bean解析为BeanDefinition，如下：

@Test  
public void test3() {  
 //定义一个spring容器，这个容器默认实现了BeanDefinitionRegistry，所以本身就是一个bean注册器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //定义一个注解方式的BeanDefinition读取器，需要传递一个BeanDefinitionRegistry（bean注册器）对象  
 AnnotatedBeanDefinitionReader annotatedBeanDefinitionReader = new AnnotatedBeanDefinitionReader(factory);  
  
 //通过PropertiesBeanDefinitionReader加载bean properties文件，然后将解析产生的BeanDefinition注册到容器容器中  
 annotatedBeanDefinitionReader.register(Service1.class, Service2.class);  
  
 //打印出注册的bean的配置信息  
 for (String beanName : new String[]{"service1", "service2"}) {  
 //通过名称从容器中获取对应的BeanDefinition信息  
 BeanDefinition beanDefinition = factory.getBeanDefinition(beanName);  
 //获取BeanDefinition具体使用的是哪个类  
 String beanDefinitionClassName = beanDefinition.getClass().getName();  
 //通过名称获取bean对象  
 Object bean = factory.getBean(beanName);  
 //打印输出  
 System.out.println(beanName + ":");  
 System.out.println(" beanDefinitionClassName：" + beanDefinitionClassName);  
 System.out.println(" beanDefinition：" + beanDefinition);  
 System.out.println(" bean：" + bean);  
 }  
}

##### 运行输出

service1:  
 beanDefinitionClassName：org.springframework.beans.factory.annotation.AnnotatedGenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo2.Service1]; scope=prototype; abstract=false; lazyInit=true; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=true; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：com.javacode2018.lesson002.demo2.Service1@21a947fe  
service2:  
 beanDefinitionClassName：org.springframework.beans.factory.annotation.AnnotatedGenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo2.Service2]; scope=singleton; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：Service2{service1=null}

输出中可以看出service1这个bean的beanDefinition中lazyInit确实为true，primary也为true，scope为prototype，说明类Service1注解上标注3个注解信息被解析之后放在了beanDefinition中。

**注意下：最后一行中的service1为什么为null，不是标注了@Autowired么？**

这个地方提前剧透一下，看不懂的没关系，这篇文章都结束之后，就明白了。

调整一下上面的代码，加上下面@1这行代码，如下：

@Test  
public void test3() {  
 //定义一个spring容器，这个容器默认实现了BeanDefinitionRegistry，所以本身就是一个bean注册器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //定义一个注解方式的BeanDefinition读取器，需要传递一个BeanDefinitionRegistry（bean注册器）对象  
 AnnotatedBeanDefinitionReader annotatedBeanDefinitionReader = new AnnotatedBeanDefinitionReader(factory);  
  
 //通过PropertiesBeanDefinitionReader加载bean properties文件，然后将解析产生的BeanDefinition注册到容器容器中  
 annotatedBeanDefinitionReader.register(Service1.class, Service2.class);  
  
 factory.getBeansOfType(BeanPostProcessor.class).values().forEach(factory::addBeanPostProcessor); // @1  
 //打印出注册的bean的配置信息  
 for (String beanName : new String[]{"service1", "service2"}) {  
 //通过名称从容器中获取对应的BeanDefinition信息  
 BeanDefinition beanDefinition = factory.getBeanDefinition(beanName);  
 //获取BeanDefinition具体使用的是哪个类  
 String beanDefinitionClassName = beanDefinition.getClass().getName();  
 //通过名称获取bean对象  
 Object bean = factory.getBean(beanName);  
 //打印输出  
 System.out.println(beanName + ":");  
 System.out.println(" beanDefinitionClassName：" + beanDefinitionClassName);  
 System.out.println(" beanDefinition：" + beanDefinition);  
 System.out.println(" bean：" + bean);  
 }  
}

再次运行一下，最后一行有值了：

service1:  
 beanDefinitionClassName：org.springframework.beans.factory.annotation.AnnotatedGenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo2.Service1]; scope=prototype; abstract=false; lazyInit=true; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=true; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：com.javacode2018.lesson002.demo2.Service1@564718df  
service2:  
 beanDefinitionClassName：org.springframework.beans.factory.annotation.AnnotatedGenericBeanDefinition  
 beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo2.Service2]; scope=singleton; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
 bean：Service2{service1=com.javacode2018.lesson002.demo2.Service1@52aa2946}

**目前进行到第二个阶段了，还有14个阶段，本文内容比较长，建议先收藏起来，慢慢看，咱们继续。**

### 阶段3：Spring Bean注册阶段

bean注册阶段需要用到一个非常重要的接口：BeanDefinitionRegistry

#### Bean注册接口：BeanDefinitionRegistry

这个接口中定义了注册bean常用到的一些方法，源码如下：

public interface BeanDefinitionRegistry extends AliasRegistry {  
  
 /\*\*  
 \* 注册一个新的bean定义  
 \* beanName：bean的名称  
 \* beanDefinition：bean定义信息  
 \*/  
 void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  
 throws BeanDefinitionStoreException;  
  
 /\*\*  
 \* 通过bean名称移除已注册的bean  
 \* beanName：bean名称  
 \*/  
 void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;  
  
 /\*\*  
 \* 通过名称获取bean的定义信息  
 \* beanName：bean名称  
 \*/  
 BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;  
  
 /\*\*  
 \* 查看beanName是否注册过  
 \*/  
 boolean containsBeanDefinition(String beanName);  
  
 /\*\*  
 \* 获取已经定义（注册）的bean名称列表  
 \*/  
 String[] getBeanDefinitionNames();  
  
 /\*\*  
 \* 返回注册器中已注册的bean数量  
 \*/  
 int getBeanDefinitionCount();  
  
 /\*\*  
 \* 确定给定的bean名称或者别名是否已在此注册表中使用  
 \* beanName：可以是bean名称或者bean的别名  
 \*/  
 boolean isBeanNameInUse(String beanName);  
  
}

#### 别名注册接口：AliasRegistry

BeanDefinitionRegistry接口继承了AliasRegistry接口，这个接口中定义了操作bean别名的一些方法，看一下其源码：

public interface AliasRegistry {  
  
 /\*\*  
 \* 给name指定别名alias  
 \*/  
 void registerAlias(String name, String alias);  
  
 /\*\*  
 \* 从此注册表中删除指定的别名  
 \*/  
 void removeAlias(String alias);  
  
 /\*\*  
 \* 判断name是否作为别名已经被使用了  
 \*/  
 boolean isAlias(String name);  
  
 /\*\*  
 \* 返回name对应的所有别名  
 \*/  
 String[] getAliases(String name);  
  
}

#### BeanDefinitionRegistry唯一实现：DefaultListableBeanFactory

spring中BeanDefinitionRegistry接口有一个唯一的实现类：

org.springframework.beans.factory.support.DefaultListableBeanFactory

大家可能看到有很多类也实现了BeanDefinitionRegistry接口，比如我们经常用到的AnnotationConfigApplicationContext，但实际上其内部是转发给了DefaultListableBeanFactory进行处理的，所以真正实现这个接口的类是DefaultListableBeanFactory。

大家再回头看一下开头的几个案例，都使用的是DefaultListableBeanFactory作为bean注册器，此时你们应该可以理解为什么了。

下面我们来个案例演示一下上面常用的一些方法。

#### 案例

##### 代码

package com.javacode2018.lesson002.demo3;  
  
import org.junit.Test;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
import org.springframework.beans.factory.support.GenericBeanDefinition;  
  
import java.util.Arrays;  
  
/\*\*  
 \* BeanDefinitionRegistry 案例  
 \*/  
public class BeanDefinitionRegistryTest {  
  
 @Test  
 public void test1() {  
 //创建一个bean工厂，这个默认实现了BeanDefinitionRegistry接口，所以也是一个bean注册器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //定义一个bean  
 GenericBeanDefinition nameBdf = new GenericBeanDefinition();  
 nameBdf.setBeanClass(String.class);  
 nameBdf.getConstructorArgumentValues().addIndexedArgumentValue(0, "马士兵老师");  
  
 //将bean注册到容器中  
 factory.registerBeanDefinition("name", nameBdf);  
  
 //通过名称获取BeanDefinition  
 System.out.println(factory.getBeanDefinition("name"));  
 //通过名称判断是否注册过BeanDefinition  
 System.out.println(factory.containsBeanDefinition("name"));  
 //获取所有注册的名称  
 System.out.println(Arrays.asList(factory.getBeanDefinitionNames()));  
 //获取已注册的BeanDefinition的数量  
 System.out.println(factory.getBeanDefinitionCount());  
 //判断指定的name是否使用过  
 System.out.println(factory.isBeanNameInUse("name"));  
  
 //别名相关方法  
 //为name注册2个别名  
 factory.registerAlias("name", "alias-name-1");  
 factory.registerAlias("name", "alias-name-2");  
  
 //判断alias-name-1是否已被作为别名使用  
 System.out.println(factory.isAlias("alias-name-1"));  
  
 //通过名称获取对应的所有别名  
 System.out.println(Arrays.asList(factory.getAliases("name")));  
  
 //最后我们再来获取一下这个bean  
 System.out.println(factory.getBean("name"));  
  
  
 }  
}

##### 运行输出

Generic bean: class [java.lang.String]; scope=; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null  
true  
[name]  
1  
true  
true  
[alias-name-2, alias-name-1]  
马士兵老师

**下面要介绍的从阶段4到阶段14，也就是从：BeanDefinition合并阶段到Bean初始化完成阶段，都是在调用getBean从容器中获取bean对象的过程中发送的操作，要注意细看了，大家下去了建议去看getBean这个方法的源码，以下过程均来自于这个方法：**

org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean

### 阶段4：BeanDefinition合并阶段

#### 合并阶段是做什么的？

可能我们定义bean的时候有父子bean关系，此时子BeanDefinition中的信息是不完整的，比如设置属性的时候配置在父BeanDefinition中，此时子BeanDefinition中是没有这些信息的，需要将子bean的BeanDefinition和父bean的BeanDefinition进行合并，得到最终的一个RootBeanDefinition，合并之后得到的RootBeanDefinition包含bean定义的所有信息，包含了从父bean中继继承过来的所有信息，后续bean的所有创建工作就是依靠合并之后BeanDefinition来进行的。

合并BeanDefinition会使用下面这个方法：

org.springframework.beans.factory.support.AbstractBeanFactory#getMergedBeanDefinition

**bean定义可能存在多级父子关系，合并的时候进进行递归合并，最终得到一个包含完整信息的RootBeanDefinition**

#### 案例

##### 来一个普通的类

package com.javacode2018.lesson002.demo4;  
  
public class LessonModel {  
 //课程名称  
 private String name;  
 //课时  
 private int lessonCount;  
 //描述信息  
 private String description;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getLessonCount() {  
 return lessonCount;  
 }  
  
 public void setLessonCount(int lessonCount) {  
 this.lessonCount = lessonCount;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public void setDescription(String description) {  
 this.description = description;  
 }  
  
 @Override  
 public String toString() {  
 return "LessonModel{" +  
 "name='" + name + '\'' +  
 ", lessonCount=" + lessonCount +  
 ", description='" + description + '\'' +  
 '}';  
 }  
}

##### 通过xml定义3个具有父子关系的bean

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">  
  
 <bean id="lesson1" class="com.javacode2018.lesson002.demo4.LessonModel"/>  
  
 <bean id="lesson2" parent="lesson1">  
 <property name="name" value="spring高手系列"/>  
 <property name="lessonCount" value="100"/>  
 </bean>  
  
 <bean id="lesson3" parent="lesson2">  
 <property name="description" value="马士兵老师带你学spring，超越90%开发者!"/>  
 </bean>  
  
</beans>

lesson2相当于lesson1的儿子，lesson3相当于lesson1的孙子。

##### 解析xml注册bean

下面将解析xml，进行bean注册，然后遍历输出bean的名称，解析过程中注册的原始的BeanDefinition，合并之后的BeanDefinition，以及合并前后BeanDefinition中的属性信息

package com.javacode2018.lesson002.demo4;  
  
import org.junit.Test;  
import org.springframework.beans.factory.config.BeanDefinition;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;  
  
/\*\*  
 \* BeanDefinition 合并  
 \*/  
public class MergedBeanDefinitionTest {  
 @Test  
 public void test1() {  
 //创建bean容器  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 //创建一个bean xml解析器  
 XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(factory);  
 //解析bean xml，将解析过程中产生的BeanDefinition注册到DefaultListableBeanFactory中  
 beanDefinitionReader.loadBeanDefinitions("com/javacode2018/lesson002/demo4/beans.xml");  
 //遍历容器中注册的所有bean信息  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 //通过bean名称获取原始的注册的BeanDefinition信息  
 BeanDefinition beanDefinition = factory.getBeanDefinition(beanName);  
 //获取合并之后的BeanDefinition信息  
 BeanDefinition mergedBeanDefinition = factory.getMergedBeanDefinition(beanName);  
  
 System.out.println(beanName);  
 System.out.println("解析xml过程中注册的beanDefinition：" + beanDefinition);  
 System.out.println("beanDefinition中的属性信息" + beanDefinition.getPropertyValues());  
 System.out.println("合并之后得到的mergedBeanDefinition：" + mergedBeanDefinition);  
 System.out.println("mergedBeanDefinition中的属性信息" + mergedBeanDefinition.getPropertyValues());  
 System.out.println("---------------------------");  
 }  
 }  
}

##### 运行输出

lesson1  
解析xml过程中注册的beanDefinition：Generic bean: class [com.javacode2018.lesson002.demo4.LessonModel]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
beanDefinition中的属性信息PropertyValues: length=0  
合并之后得到的mergedBeanDefinition：Root bean: class [com.javacode2018.lesson002.demo4.LessonModel]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
mergedBeanDefinition中的属性信息PropertyValues: length=0  
---------------------------  
lesson2  
解析xml过程中注册的beanDefinition：Generic bean with parent 'lesson1': class [null]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
beanDefinition中的属性信息PropertyValues: length=2; bean property 'name'; bean property 'lessonCount'  
合并之后得到的mergedBeanDefinition：Root bean: class [com.javacode2018.lesson002.demo4.LessonModel]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
mergedBeanDefinition中的属性信息PropertyValues: length=2; bean property 'name'; bean property 'lessonCount'  
---------------------------  
lesson3  
解析xml过程中注册的beanDefinition：Generic bean with parent 'lesson2': class [null]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
beanDefinition中的属性信息PropertyValues: length=1; bean property 'description'  
合并之后得到的mergedBeanDefinition：Root bean: class [com.javacode2018.lesson002.demo4.LessonModel]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [com/javacode2018/lesson002/demo4/beans.xml]  
mergedBeanDefinition中的属性信息PropertyValues: length=3; bean property 'name'; bean property 'lessonCount'; bean property 'description'  
---------------------------

从输出的结果中可以看到，合并之前，BeanDefinition是不完整的，比lesson2和lesson3中的class是null，属性信息也不完整，但是合并之后这些信息都完整了。

合并之前是GenericBeanDefinition类型的，合并之后得到的是RootBeanDefinition类型的。

获取lesson3合并的BeanDefinition时，内部会递归进行合并，先将lesson1和lesson2合并，然后将lesson2再和lesson3合并，最后得到合并之后的BeanDefinition。

**后面的阶段将使用合并产生的RootBeanDefinition。**

### 阶段5：Bean Class加载阶段

**这个阶段就是将bean的class名称转换为Class类型的对象。**

BeanDefinition中有个Object类型的字段：beanClass

private volatile Object beanClass;

用来表示bean的class对象，通常这个字段的值有2种类型，一种是bean对应的Class类型的对象，另一种是bean对应的Class的完整类名，第一种情况不需要解析，第二种情况：即这个字段是bean的类名的时候，就需要通过类加载器将其转换为一个Class对象。

此时会对阶段4中合并产生的RootBeanDefinition中的beanClass进行解析，将bean的类名转换为Class对象，然后赋值给beanClass字段。

源码位置：

org.springframework.beans.factory.support.AbstractBeanFactory#resolveBeanClass

上面得到了Bean Class对象以及合并之后的BeanDefinition，下面就开始进入实例化这个对象的阶段了。

**Bean实例化分为3个阶段：前阶段、实例化阶段、后阶段；下面详解介绍。**

### 阶段6：Bean实例化阶段

#### 分2个小的阶段

1. Bean实例化前操作
2. Bean实例化操作

#### Bean实例化前操作

先来看一下DefaultListableBeanFactory，这个类中有个非常非常重要的字段：

private final List<BeanPostProcessor> beanPostProcessors = new CopyOnWriteArrayList<>();

是一个BeanPostProcessor类型的集合

**BeanPostProcessor是一个接口，还有很多子接口，这些接口中提供了很多方法，spring在bean生命周期的不同阶段，会调用上面这个列表中的BeanPostProcessor中的一些方法，来对生命周期进行扩展，bean生命周期中的所有扩展点都是依靠这个集合中的BeanPostProcessor来实现的，所以如果大家想对bean的生命周期进行干预，这块一定要掌握好。**

**注意：本文中很多以BeanPostProcessor结尾的，都实现了BeanPostProcessor接口，有些是直接实现的，有些是实现了它的子接口。**

Bean实例化之前会调用一段代码：

@Nullable  
 protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
 Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);  
 if (result != null) {  
 return result;  
 }  
 }  
 }  
 return null;  
 }

这段代码在bean实例化之前给开发者留了个口子，开发者自己可以在这个地方直接去创建一个对象作为bean实例，而跳过spring内部实例化bean的过程。

上面代码中轮询beanPostProcessors列表，如果类型是InstantiationAwareBeanPostProcessor， 尝试调用InstantiationAwareBeanPostProcessor#postProcessBeforeInstantiation获取bean的实例对象，如果能够获取到，那么将返回值作为当前bean的实例，那么spring自带的实例化bean的过程就被跳过了。

postProcessBeforeInstantiation方法如下：

default Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {  
 return null;  
}

这个地方给开发者提供了一个扩展点，允许开发者在这个方法中直接返回bean的一个实例。

下面我们来个案例看一下。

##### 案例

package com.javacode2018.lesson002.demo5;  
  
import com.javacode2018.lesson002.demo1.Car;  
import org.junit.Test;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor;  
import org.springframework.beans.factory.support.AbstractBeanDefinition;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
import org.springframework.lang.Nullable;  
  
/\*\*  
 \* bean初始化前阶段，会调用：{@link org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor#postProcessBeforeInitialization(Object, String)}  
 \*/  
public class InstantiationAwareBeanPostProcessorTest {  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //添加一个BeanPostProcessor：InstantiationAwareBeanPostProcessor  
 factory.addBeanPostProcessor(new InstantiationAwareBeanPostProcessor() { //@1  
 @Nullable  
 @Override  
 public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {  
 System.out.println("调用postProcessBeforeInstantiation()");  
 //发现类型是Car类型的时候，硬编码创建一个Car对象返回  
 if (beanClass == Car.class) {  
 Car car = new Car();  
 car.setName("保时捷");  
 return car;  
 }  
 return null;  
 }  
 });  
  
 //定义一个car bean,车名为：奥迪  
 AbstractBeanDefinition carBeanDefinition = BeanDefinitionBuilder.  
 genericBeanDefinition(Car.class).  
 addPropertyValue("name", "奥迪"). //@2  
 getBeanDefinition();  
 factory.registerBeanDefinition("car", carBeanDefinition);  
 //从容器中获取car这个bean的实例，输出  
 System.out.println(factory.getBean("car"));  
  
 }  
}

@1：创建了一个InstantiationAwareBeanPostProcessor，丢到了容器中的BeanPostProcessor列表中

@2：创建了一个car bean，name为奥迪

##### 运行输出

调用postProcessBeforeInstantiation()  
Car{name='保时捷'}

bean定义的时候，名称为：奥迪，最后输出的为：保时捷

定义和输出不一致的原因是因为我们在InstantiationAwareBeanPostProcessor#postProcessBeforeInstantiation方法中手动创建了一个实例直接返回了，而不是依靠spring内部去创建这个实例。

##### 小结

实际上，在实例化前阶段对bean的创建进行干预的情况，用的非常少，所以大部分bean的创建还会继续走下面的阶段。

#### Bean实例化操作

##### 这个过程可以干什么？

这个过程会通过反射来调用bean的构造器来创建bean的实例。

具体需要使用哪个构造器，spring为开发者提供了一个接口，允许开发者自己来判断用哪个构造器。

看一下这块的代码逻辑：

for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {  
 SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;  
 Constructor<?>[] ctors = ibp.determineCandidateConstructors(beanClass, beanName);  
 if (ctors != null) {  
 return ctors;  
 }  
 }  
}

会调用SmartInstantiationAwareBeanPostProcessor接口的determineCandidateConstructors方法，这个方法会返回候选的构造器列表，也可以返回空，看一下这个方法的源码：

@Nullable  
default Constructor<?>[] determineCandidateConstructors(Class<?> beanClass, String beanName)  
throws BeansException {  
  
 return null;  
}

这个方法有个比较重要的实现类

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor

可以将@Autowired标注的方法作为候选构造器返回，有兴趣的可以去看一下代码。

##### 案例

**下面我们来个案例，自定义一个注解，当构造器被这个注解标注的时候，让spring自动选择使用这个构造器创建对象。**

###### 自定义一个注解

下面这个注解可以标注在构造器上面，使用这个标注之后，创建bean的时候将使用这个构造器。

package com.javacode2018.lesson002.demo6;  
  
import java.lang.annotation.\*;  
  
@Target(ElementType.CONSTRUCTOR)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface MyAutowried {  
}

###### 来个普通的类

下面这个类3个构造器，其中一个使用@MyAutowried，让其作为bean实例化的方法。

package com.javacode2018.lesson002.demo6;  
  
public class Person {  
 private String name;  
 private Integer age;  
  
 public Person() {  
 System.out.println("调用 Person()");  
 }  
  
 @MyAutowried  
 public Person(String name) {  
 System.out.println("调用 Person(String name)");  
 this.name = name;  
 }  
  
 public Person(String name, Integer age) {  
 System.out.println("调用 Person(String name, int age)");  
 this.name = name;  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

###### 自定义一个SmartInstantiationAwareBeanPostProcessor

代码的逻辑：将@MyAutowried标注的构造器列表返回

package com.javacode2018.lesson002.demo6;  
  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.SmartInstantiationAwareBeanPostProcessor;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Constructor;  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class MySmartInstantiationAwareBeanPostProcessor implements SmartInstantiationAwareBeanPostProcessor {  
 @Nullable  
 @Override  
 public Constructor<?>[] determineCandidateConstructors(Class<?> beanClass, String beanName) throws BeansException {  
 System.out.println(beanClass);  
 System.out.println("调用 MySmartInstantiationAwareBeanPostProcessor.determineCandidateConstructors 方法");  
 Constructor<?>[] declaredConstructors = beanClass.getDeclaredConstructors();  
 if (declaredConstructors != null) {  
 //获取有@MyAutowried注解的构造器列表  
 List<Constructor<?>> collect = Arrays.stream(declaredConstructors).  
 filter(constructor -> constructor.isAnnotationPresent(MyAutowried.class)).  
 collect(Collectors.toList());  
 Constructor[] constructors = collect.toArray(new Constructor[collect.size()]);  
 return constructors.length != 0 ? constructors : null;  
 } else {  
 return null;  
 }  
 }  
}

###### 来个测试用例

package com.javacode2018.lesson002.demo6;  
  
import org.junit.Test;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
  
/\*\*  
 \* 通过{@link org.springframework.beans.factory.config.SmartInstantiationAwareBeanPostProcessor#determineCandidateConstructors(Class, String)}来确定使用哪个构造器来创建bean实例  
 \*/  
public class SmartInstantiationAwareBeanPostProcessorTest {  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //创建一个SmartInstantiationAwareBeanPostProcessor,将其添加到容器中  
 factory.addBeanPostProcessor(new MySmartInstantiationAwareBeanPostProcessor());  
  
 factory.registerBeanDefinition("name",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("马士兵老师").  
 getBeanDefinition());  
  
 factory.registerBeanDefinition("age",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(Integer.class).  
 addConstructorArgValue(30).  
 getBeanDefinition());  
  
 factory.registerBeanDefinition("person",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(Person.class).  
 getBeanDefinition());  
  
 Person person = factory.getBean("person", Person.class);  
 System.out.println(person);  
  
 }  
}

###### 运行输出

class com.javacode2018.lesson002.demo6.Person  
调用 MySmartInstantiationAwareBeanPostProcessor.determineCandidateConstructors 方法  
class java.lang.String  
调用 MySmartInstantiationAwareBeanPostProcessor.determineCandidateConstructors 方法  
调用 Person(String name)  
Person{name='马士兵老师', age=null}

从输出中可以看出调用了Person中标注@MyAutowired标注的构造器。

到目前为止bean实例化阶段结束了，继续进入后面的阶段。

### 阶段7：合并后的BeanDefinition处理

这块的源码如下

protected void applyMergedBeanDefinitionPostProcessors(RootBeanDefinition mbd, Class<?> beanType, String beanName) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof MergedBeanDefinitionPostProcessor) {  
 MergedBeanDefinitionPostProcessor bdp = (MergedBeanDefinitionPostProcessor) bp;  
 bdp.postProcessMergedBeanDefinition(mbd, beanType, beanName);  
 }  
 }  
 }

会调用MergedBeanDefinitionPostProcessor接口的postProcessMergedBeanDefinition方法，看一下这个方法的源码：

void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName);

spring会轮询BeanPostProcessor，依次调用MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition

第一个参数为beanDefinition，表示合并之后的RootBeanDefinition，我们可以在这个方法内部对合并之后的BeanDefinition进行再次处理

**postProcessMergedBeanDefinition有2个实现类，前面我们介绍过，用的也比较多，面试的时候也会经常问的：**

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor  
在 postProcessMergedBeanDefinition 方法中对 @Autowired、@Value 标注的方法、字段进行缓存  
  
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor  
在 postProcessMergedBeanDefinition 方法中对 @Resource 标注的字段、@Resource 标注的方法、 @PostConstruct 标注的字段、 @PreDestroy标注的方法进行缓存

### 阶段8：Bean属性设置阶段

#### 属性设置阶段分为3个小的阶段

* 实例化后阶段
* Bean属性赋值前处理
* Bean属性赋值

#### 实例化后阶段

会调用InstantiationAwareBeanPostProcessor接口的postProcessAfterInstantiation这个方法，调用逻辑如下：

看一下具体的调用逻辑如下：

for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  
 return;  
 }  
 }  
}

postProcessAfterInstantiation方法返回false的时候，后续的**Bean属性赋值前处理、Bean属性赋值**都会被跳过了。

来看一下postProcessAfterInstantiation这个方法的定义

default boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException {  
 return true;  
}

**来看个案例，案例中返回false，跳过属性的赋值操作。**

##### 案例

###### 来个类

package com.javacode2018.lesson002.demo7;  
  
  
public class UserModel {  
 private String name;  
 private Integer age;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "UserModel{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

###### 测试用例

下面很简单，来注册一个UserModel的bean

package com.javacode2018.lesson002.demo7;  
  
  
import org.junit.Test;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
  
/\*\*  
 \* {@link InstantiationAwareBeanPostProcessor#postProcessAfterInstantiation(java.lang.Object, java.lang.String)}  
 \* 返回false，可以阻止bean属性的赋值  
 \*/  
public class InstantiationAwareBeanPostProcessoryTest1 {  
  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 factory.registerBeanDefinition("user1", BeanDefinitionBuilder.  
 genericBeanDefinition(UserModel.class).  
 addPropertyValue("name", "马士兵老师").  
 addPropertyValue("age", 30).  
 getBeanDefinition());  
  
 factory.registerBeanDefinition("user2", BeanDefinitionBuilder.  
 genericBeanDefinition(UserModel.class).  
 addPropertyValue("name", "刘德华").  
 addPropertyValue("age", 50).  
 getBeanDefinition());  
  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, factory.getBean(beanName)));  
 }  
 }  
  
}

上面定义了2个bean：[user1,user2]，获取之后输出

###### 运行输出

user1->UserModel{name='马士兵老师', age=30}  
user2->UserModel{name='刘德华', age=50}

此时UserModel中2个属性都是有值的。

下面来阻止user1的赋值，对代码进行改造，加入下面代码：

factory.addBeanPostProcessor(new InstantiationAwareBeanPostProcessor() {  
 @Override  
 public boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException {  
 if ("user1".equals(beanName)) {  
 return false;  
 } else {  
 return true;  
 }  
 }  
});

再次运行测试输出：

user1->UserModel{name='null', age=null}  
user2->UserModel{name='刘德华', age=50}

user1的属性赋值被跳过了。

#### Bean属性赋值前阶段

这个阶段会调用InstantiationAwareBeanPostProcessor接口的postProcessProperties方法，调用逻辑：

for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
 PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);  
 if (pvsToUse == null) {  
 if (filteredPds == null) {  
 filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  
 }  
 pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  
 if (pvsToUse == null) {  
 return;  
 }  
 }  
 pvs = pvsToUse;  
 }  
}

从上面可以看出，如果InstantiationAwareBeanPostProcessor中的postProcessProperties和postProcessPropertyValues都返回空的时候，表示这个bean不需要设置属性，直接返回了，直接进入下一个阶段。

来看一下postProcessProperties这个方法的定义：

@Nullable  
default PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String beanName)  
 throws BeansException {  
  
 return null;  
}

PropertyValues中保存了bean实例对象中所有属性值的设置，所以我们可以在这个这个方法中对PropertyValues值进行修改。

##### 这个方法有2个比较重要的实现类

###### AutowiredAnnotationBeanPostProcessor在这个方法中对@Autowired、@Value标注的字段、方法注入值。

###### CommonAnnotationBeanPostProcessor在这个方法中对@Resource标注的字段和方法注入值。

**来个案例，我们在案例中对pvs进行修改。**

##### 案例

###### 案例代码

@Test  
public void test3() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 factory.addBeanPostProcessor(new InstantiationAwareBeanPostProcessor() { // @0  
 @Nullable  
 @Override  
 public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String beanName) throws BeansException {  
 if ("user1".equals(beanName)) {  
 if (pvs == null) {  
 pvs = new MutablePropertyValues();  
 }  
 if (pvs instanceof MutablePropertyValues) {  
 MutablePropertyValues mpvs = (MutablePropertyValues) pvs;  
 //将姓名设置为：路人  
 mpvs.add("name", "路人");  
 //将年龄属性的值修改为18  
 mpvs.add("age", 18);  
 }  
 }  
 return null;  
 }  
 });  
  
 //注意 user1 这个没有给属性设置值  
 factory.registerBeanDefinition("user1", BeanDefinitionBuilder.  
 genericBeanDefinition(UserModel.class).  
 getBeanDefinition()); //@1  
  
 factory.registerBeanDefinition("user2", BeanDefinitionBuilder.  
 genericBeanDefinition(UserModel.class).  
 addPropertyValue("name", "刘德华").  
 addPropertyValue("age", 50).  
 getBeanDefinition());  
  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, factory.getBean(beanName)));  
 }  
}

@1：user1这个bean没有设置属性的值

@0：这个实现 org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor#postProcessProperties 方法，在其内部对 user1 这个bean进行属性值信息进行修改。

###### 运行输出

user1->UserModel{name='路人', age=18}  
user2->UserModel{name='刘德华', age=50}

上面过程都ok，进入bean赋值操作

#### Bean属性赋值阶段

这个过程比较简单了，循环处理PropertyValues中的属性值信息，通过反射调用set方法将属性的值设置到bean实例中。

PropertyValues中的值是通过bean xml中property元素配置的，或者调用MutablePropertyValues中add方法设置的值。

### 阶段9：Bean初始化阶段

#### 这个阶段分为5个小的阶段

* Bean Aware接口回调
* Bean初始化前操作
* Bean初始化操作
* Bean初始化后操作
* Bean初始化完成操作

#### Bean Aware接口回调

这块的源码：

private void invokeAwareMethods(final String beanName, final Object bean) {  
 if (bean instanceof Aware) {  
 if (bean instanceof BeanNameAware) {  
 ((BeanNameAware) bean).setBeanName(beanName);  
 }  
 if (bean instanceof BeanClassLoaderAware) {  
 ClassLoader bcl = getBeanClassLoader();  
 if (bcl != null) {  
 ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);  
 }  
 }  
 if (bean instanceof BeanFactoryAware) {  
 ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);  
 }  
 }  
 }

如果我们的bean实例实现了上面的接口，会按照下面的顺序依次进行调用：

BeanNameAware：将bean的名称注入进去  
BeanClassLoaderAware：将BeanClassLoader注入进去  
BeanFactoryAware：将BeanFactory注入进去

来个案例感受一下

来个类，实现上面3个接口。

package com.javacode2018.lesson002.demo8;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.BeanClassLoaderAware;  
import org.springframework.beans.factory.BeanFactory;  
import org.springframework.beans.factory.BeanFactoryAware;  
import org.springframework.beans.factory.BeanNameAware;  
  
public class AwareBean implements BeanNameAware, BeanClassLoaderAware, BeanFactoryAware {  
  
 @Override  
 public void setBeanName(String name) {  
 System.out.println("setBeanName：" + name);  
 }  
  
 @Override  
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  
 System.out.println("setBeanFactory：" + beanFactory);  
 }  
  
 @Override  
 public void setBeanClassLoader(ClassLoader classLoader) {  
 System.out.println("setBeanClassLoader：" + classLoader);  
 }  
  
}

来个测试类，创建上面这个对象的的bean

package com.javacode2018.lesson002.demo8;  
  
import org.junit.Test;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
  
public class InvokeAwareTest {  
  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 factory.registerBeanDefinition("awareBean", BeanDefinitionBuilder.genericBeanDefinition(AwareBean.class).getBeanDefinition());  
 //调用getBean方法获取bean，将触发bean的初始化  
 factory.getBean("awareBean");  
 }  
}

运行输出

setBeanName：awareBean  
setBeanClassLoader：sun.misc.Launcher$AppClassLoader@18b4aac2  
setBeanFactory：org.springframework.beans.factory.support.DefaultListableBeanFactory@5bb21b69: defining beans [awareBean]; root of factory hierarchy

#### Bean初始化前操作

这个阶段的源码：

@Override  
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor processor : getBeanPostProcessors()) {  
 Object current = processor.postProcessBeforeInitialization(result, beanName);  
 if (current == null) {  
 return result;  
 }  
 result = current;  
 }  
 return result;  
}

会调用BeanPostProcessor的postProcessBeforeInitialization方法，若返回null，当前方法将结束。

**通常称postProcessBeforeInitialization这个方法为：bean初始化前操作。**

这个接口有2个实现类，比较重要：

org.springframework.context.support.ApplicationContextAwareProcessor  
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor

##### ApplicationContextAwareProcessor注入6个Aware接口对象

如果bean实现了下面的接口，在ApplicationContextAwareProcessor#postProcessBeforeInitialization中会依次调用下面接口中的方法，将Aware前缀对应的对象注入到bean实例中。

EnvironmentAware：注入Environment对象  
EmbeddedValueResolverAware：注入EmbeddedValueResolver对象  
ResourceLoaderAware：注入ResourceLoader对象  
ApplicationEventPublisherAware：注入ApplicationEventPublisher对象  
MessageSourceAware：注入MessageSource对象  
ApplicationContextAware：注入ApplicationContext对象

从名称上可以看出这个类以ApplicationContext开头的，说明这个类只能在ApplicationContext环境中使用。

##### CommonAnnotationBeanPostProcessor调用@PostConstruct标注的方法

CommonAnnotationBeanPostProcessor#postProcessBeforeInitialization中会调用bean中所有标注@PostConstruct注解的方法

来个案例，感受一下。

##### 案例

###### 来个类

下面的类有2个方法标注了@PostConstruct，并且实现了上面说的那6个Aware接口。

package com.javacode2018.lesson002.demo9;  
  
import org.springframework.beans.BeansException;  
import org.springframework.context.\*;  
import org.springframework.core.env.Environment;  
import org.springframework.core.io.ResourceLoader;  
import org.springframework.util.StringValueResolver;  
  
import javax.annotation.PostConstruct;  
  
public class Bean1 implements EnvironmentAware, EmbeddedValueResolverAware, ResourceLoaderAware, ApplicationEventPublisherAware, MessageSourceAware, ApplicationContextAware {  
  
 @PostConstruct  
 public void postConstruct1() { //@1  
 System.out.println("postConstruct1()");  
 }  
  
 @PostConstruct  
 public void postConstruct2() { //@2  
 System.out.println("postConstruct2()");  
 }  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 System.out.println("setApplicationContext:" + applicationContext);  
 }  
  
 @Override  
 public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {  
 System.out.println("setApplicationEventPublisher:" + applicationEventPublisher);  
 }  
  
 @Override  
 public void setEmbeddedValueResolver(StringValueResolver resolver) {  
 System.out.println("setEmbeddedValueResolver:" + resolver);  
 }  
  
 @Override  
 public void setEnvironment(Environment environment) {  
 System.out.println("setEnvironment:" + environment.getClass());  
 }  
  
 @Override  
 public void setMessageSource(MessageSource messageSource) {  
 System.out.println("setMessageSource:" + messageSource);  
 }  
  
 @Override  
 public void setResourceLoader(ResourceLoader resourceLoader) {  
 System.out.println("setResourceLoader:" + resourceLoader);  
 }  
}

###### 来个测试案例

package com.javacode2018.lesson002.demo9;  
  
  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class PostProcessBeforeInitializationTest {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(Bean1.class);  
 context.refresh();  
 }  
}

###### 运行输出

setEmbeddedValueResolver:org.springframework.beans.factory.config.EmbeddedValueResolver@15b204a1  
setResourceLoader:org.springframework.context.annotation.AnnotationConfigApplicationContext@64bf3bbf, started on Sun Apr 05 21:16:00 CST 2020  
setApplicationEventPublisher:org.springframework.context.annotation.AnnotationConfigApplicationContext@64bf3bbf, started on Sun Apr 05 21:16:00 CST 2020  
setMessageSource:org.springframework.context.annotation.AnnotationConfigApplicationContext@64bf3bbf, started on Sun Apr 05 21:16:00 CST 2020  
setApplicationContext:org.springframework.context.annotation.AnnotationConfigApplicationContext@64bf3bbf, started on Sun Apr 05 21:16:00 CST 2020  
postConstruct1()  
postConstruct2()

大家可以去看一下AnnotationConfigApplicationContext的源码，其内部会添加很多BeanPostProcessor到DefaultListableBeanFactory中。

#### Bean初始化阶段

##### 2个步骤

1. 调用InitializingBean接口的afterPropertiesSet方法
2. 调用定义bean的时候指定的初始化方法。

##### 调用InitializingBean接口的afterPropertiesSet方法

来看一下InitializingBean这个接口

public interface InitializingBean {  
  
 void afterPropertiesSet() throws Exception;  
  
}

当我们的bean实现了这个接口的时候，会在这个阶段被调用

##### 调用bean定义的时候指定的初始化方法

**先来看一下如何指定bean的初始化方法，3种方式**

###### 方式1：xml方式指定初始化方法

<bean init-method="bean中方法名称"/>

###### 方式2：@Bean的方式指定初始化方法

@Bean(initMethod = "初始化的方法")

###### 方式3：api的方式指定初始化方法

this.beanDefinition.setInitMethodName(methodName);

初始化方法最终会赋值给下面这个字段

org.springframework.beans.factory.support.AbstractBeanDefinition#initMethodName

##### 案例

###### 来个类

package com.javacode2018.lesson002.demo10;  
  
import org.springframework.beans.factory.InitializingBean;  
  
public class Service implements InitializingBean{  
 public void init() {  
 System.out.println("调用init()方法");  
 }  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 System.out.println("调用afterPropertiesSet()");  
 }  
}

###### 下面我们定义Service这个bean，指定init方法为初始化方法

package com.javacode2018.lesson002.demo10;  
  
import org.junit.Test;  
import org.springframework.beans.factory.config.BeanDefinition;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
  
/\*\*  
 \* 初始化方法测试  
 \*/  
public class InitMethodTest {  
  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 BeanDefinition service = BeanDefinitionBuilder.genericBeanDefinition(Service.class).  
 setInitMethodName("init"). //@1：指定初始化方法  
 getBeanDefinition();  
  
 factory.registerBeanDefinition("service", service);  
  
 System.out.println(factory.getBean("service"));  
 }  
}

###### 运行输出

调用afterPropertiesSet()  
调用init()方法  
com.javacode2018.lesson002.demo10.Service@12f41634

调用顺序：InitializingBean中的afterPropertiesSet、然后在调用自定义的初始化方法

#### Bean初始化后阶段

这块的源码：

@Override  
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor processor : getBeanPostProcessors()) {  
 Object current = processor.postProcessAfterInitialization(result, beanName);  
 if (current == null) {  
 return result;  
 }  
 result = current;  
 }  
 return result;  
}

调用BeanPostProcessor接口的postProcessAfterInitialization方法，返回null的时候，会中断上面的操作。

**通常称postProcessAfterInitialization这个方法为：bean初始化后置操作。**

来个案例：

package com.javacode2018.lesson002.demo11;  
  
import org.junit.Test;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
import org.springframework.lang.Nullable;  
  
/\*\*  
 \* {@link BeanPostProcessor#postProcessAfterInitialization(java.lang.Object, java.lang.String)}  
 \* bean初始化后置处理  
 \*/  
public class PostProcessAfterInitializationTest {  
  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //加入bean初始化后置处理器方法实现  
 factory.addBeanPostProcessor(new BeanPostProcessor() {  
 @Nullable  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 System.out.println("postProcessAfterInitialization：" + beanName);  
 return bean;  
 }  
 });  
  
 //下面注册2个String类型的bean  
 factory.registerBeanDefinition("name",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("公众号：【马士兵老师】").  
 getBeanDefinition());  
 factory.registerBeanDefinition("personInformation",  
 BeanDefinitionBuilder.genericBeanDefinition(String.class).  
 addConstructorArgValue("带领大家成为java高手！").  
 getBeanDefinition());  
  
 System.out.println("-------输出bean信息---------");  
  
 for (String beanName : factory.getBeanDefinitionNames()) {  
 System.out.println(String.format("%s->%s", beanName, factory.getBean(beanName)));  
 }  
 }  
}

运行输出

-------输出bean信息---------  
postProcessAfterInitialization：name  
name->公众号：【马士兵老师】  
postProcessAfterInitialization：personInformation  
personInformation->带领大家成为java高手！

### 阶段10：所有单例bean初始化完成后阶段

所有单例bean实例化完成之后，spring会回调下面这个接口：

public interface SmartInitializingSingleton {  
 void afterSingletonsInstantiated();  
}

调用逻辑在下面这个方法中

/\*\*  
 \* 确保所有非lazy的单例都被实例化，同时考虑到FactoryBeans。如果需要，通常在工厂设置结束时调用。  
 \*/  
org.springframework.beans.factory.support.DefaultListableBeanFactory#preInstantiateSingletons

这个方法内部会先触发所有非延迟加载的单例bean初始化，然后从容器中找到类型是SmartInitializingSingleton的bean，调用他们的afterSingletonsInstantiated方法。

有兴趣的可以去看一下带有ApplicationContext的容器，内部最终都会调用上面这个方法触发所有单例bean的初始化。

来个2个案例演示一下SmartInitializingSingleton的使用。

#### 案例1：ApplicationContext自动回调SmartInitializingSingleton接口

Service1：

package com.javacode2018.lesson002.demo12;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
  
 public Service1() {  
 System.out.println("create " + this.getClass());  
 }  
}

Service2：

package com.javacode2018.lesson002.demo12;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 public Service2() {  
 System.out.println("create " + this.getClass());  
 }  
}

自定义一个SmartInitializingSingleton

package com.javacode2018.lesson002.demo12;  
  
import org.springframework.beans.factory.SmartInitializingSingleton;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MySmartInitializingSingleton implements SmartInitializingSingleton {  
 @Override  
 public void afterSingletonsInstantiated() {  
 System.out.println("所有bean初始化完毕！");  
 }  
}

来个测试类，通过包扫描的方式注册上面3个bean

package com.javacode2018.lesson002.demo12;  
  
import org.junit.Test;  
import org.springframework.beans.factory.SmartInitializingSingleton;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
import org.springframework.context.annotation.ComponentScan;  
  
/\*\*  
 \* 所有bean初始化完毕，容器会回调{@link SmartInitializingSingleton#afterSingletonsInstantiated()}  
 \*/  
@ComponentScan  
public class SmartInitializingSingletonTest {  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(SmartInitializingSingletonTest.class);  
 System.out.println("开始启动容器!");  
 context.refresh();  
 System.out.println("容器启动完毕!");  
 }  
}

运行输出

开始启动容器!  
create class com.javacode2018.lesson002.demo12.Service1  
create class com.javacode2018.lesson002.demo12.Service2  
所有bean初始化完毕！  
容器启动完毕!

#### 案例2：通过api的方式让DefaultListableBeanFactory去回调SmartInitializingSingleton

@Test  
public void test2() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 factory.registerBeanDefinition("service1", BeanDefinitionBuilder.genericBeanDefinition(Service1.class).getBeanDefinition());  
 factory.registerBeanDefinition("service2", BeanDefinitionBuilder.genericBeanDefinition(Service2.class).getBeanDefinition());  
 factory.registerBeanDefinition("mySmartInitializingSingleton", BeanDefinitionBuilder.genericBeanDefinition(MySmartInitializingSingleton.class).getBeanDefinition());  
 System.out.println("准备触发所有单例bean初始化");  
 //触发所有bean初始化，并且回调 SmartInitializingSingleton#afterSingletonsInstantiated 方法  
 factory.preInstantiateSingletons();  
}

上面通过api的方式注册bean

最后调用factory.preInstantiateSingletons触发所有非lazy单例bean初始化，所有bean装配完毕之后，会回调SmartInitializingSingleton接口。

### 阶段11：Bean使用阶段

这个阶段就不说了，调用getBean方法得到了bean之后，大家可以随意使用，任意发挥。

### 阶段12：Bean销毁阶段

#### 触发bean销毁的几种方式

1. 调用org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#destroyBean
2. 调用org.springframework.beans.factory.config.ConfigurableBeanFactory#destroySingletons
3. 调用ApplicationContext中的close方法

#### Bean销毁阶段会依次执行

1. 轮询beanPostProcessors列表，如果是DestructionAwareBeanPostProcessor这种类型的，会调用其内部的postProcessBeforeDestruction方法
2. 如果bean实现了org.springframework.beans.factory.DisposableBean接口，会调用这个接口中的destroy方法
3. 调用bean自定义的销毁方法

#### DestructionAwareBeanPostProcessor接口

看一下源码：

public interface DestructionAwareBeanPostProcessor extends BeanPostProcessor {  
  
 /\*\*  
 \* bean销毁前调用的方法  
 \*/  
 void postProcessBeforeDestruction(Object bean, String beanName) throws BeansException;  
  
 /\*\*  
 \* 用来判断bean是否需要触发postProcessBeforeDestruction方法  
 \*/  
 default boolean requiresDestruction(Object bean) {  
 return true;  
 }  
  
}

这个接口有个关键的实现类：

org.springframework.context.annotation.CommonAnnotationBeanPostProcessor

**CommonAnnotationBeanPostProcessor#postProcessBeforeDestruction方法中会调用bean中所有标注了@PreDestroy的方法。**

#### 再来说一下自定义销毁方法有3种方式

##### 方式1：xml中指定销毁方法

<bean destroy-method="bean中方法名称"/>

##### 方式2：@Bean中指定销毁方法

@Bean(destroyMethod = "初始化的方法")

##### 方式3：api的方式指定销毁方法

this.beanDefinition.setDestroyMethodName(methodName);

初始化方法最终会赋值给下面这个字段

org.springframework.beans.factory.support.AbstractBeanDefinition#destroyMethodName

下面来看销毁的案例

#### 案例1：自定义DestructionAwareBeanPostProcessor

##### 来个类

package com.javacode2018.lesson002.demo13;  
  
public class ServiceA {  
 public ServiceA() {  
 System.out.println("create " + this.getClass());  
 }  
}

##### 自定义一个DestructionAwareBeanPostProcessor

package com.javacode2018.lesson002.demo13;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.DestructionAwareBeanPostProcessor;  
  
public class MyDestructionAwareBeanPostProcessor implements DestructionAwareBeanPostProcessor {  
 @Override  
 public void postProcessBeforeDestruction(Object bean, String beanName) throws BeansException {  
 System.out.println("准备销毁bean：" + beanName);  
 }  
}

##### 来个测试类

package com.javacode2018.lesson002.demo13;  
  
import org.junit.Test;  
import org.springframework.beans.factory.support.BeanDefinitionBuilder;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
  
/\*\*  
 \* 自定义 {@link org.springframework.beans.factory.config.DestructionAwareBeanPostProcessor}  
 \*/  
public class DestructionAwareBeanPostProcessorTest {  
  
 @Test  
 public void test1() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //添加自定义的DestructionAwareBeanPostProcessor  
 factory.addBeanPostProcessor(new MyDestructionAwareBeanPostProcessor());  
  
 //向容器中注入3个单例bean  
 factory.registerBeanDefinition("serviceA1", BeanDefinitionBuilder.genericBeanDefinition(ServiceA.class).getBeanDefinition());  
 factory.registerBeanDefinition("serviceA2", BeanDefinitionBuilder.genericBeanDefinition(ServiceA.class).getBeanDefinition());  
 factory.registerBeanDefinition("serviceA3", BeanDefinitionBuilder.genericBeanDefinition(ServiceA.class).getBeanDefinition());  
  
 //触发所有单例bean初始化  
 factory.preInstantiateSingletons(); //@1  
  
 System.out.println("销毁serviceA1");   
 //销毁指定的bean  
 factory.destroySingleton("serviceA1");//@2  
  
 System.out.println("触发所有单例bean的销毁");  
 factory.destroySingletons();  
 }  
}

上面使用了2种方式来触发bean的销毁[@1和@2]

##### 运行输出

create class com.javacode2018.lesson002.demo13.ServiceA  
create class com.javacode2018.lesson002.demo13.ServiceA  
create class com.javacode2018.lesson002.demo13.ServiceA  
销毁serviceA1  
准备要销毁bean：serviceA1  
触发所有单例bean的销毁  
准备要销毁bean：serviceA3  
准备要销毁bean：serviceA2

可以看到postProcessBeforeDestruction被调用了3次，依次销毁3个自定义的bean

#### 案例2：触发@PreDestroy标注的方法被调用

上面说了这个注解是在CommonAnnotationBeanPostProcessor#postProcessBeforeDestruction中被处理的，所以只需要将这个加入BeanPostProcessor列表就可以了。

##### 再来个类

package com.javacode2018.lesson002.demo13;  
  
import javax.annotation.PreDestroy;  
  
public class ServiceB {  
 public ServiceB() {  
 System.out.println("create " + this.getClass());  
 }  
  
 @PreDestroy  
 public void preDestroy() { //@1  
 System.out.println("preDestroy()");  
 }  
}

@1：标注了@PreDestroy注解

##### 测试用例

@Test  
public void test2() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
  
 //添加自定义的DestructionAwareBeanPostProcessor  
 factory.addBeanPostProcessor(new MyDestructionAwareBeanPostProcessor()); //@1  
 //将CommonAnnotationBeanPostProcessor加入  
 factory.addBeanPostProcessor(new CommonAnnotationBeanPostProcessor()); //@2  
  
 //向容器中注入bean  
 factory.registerBeanDefinition("serviceB", BeanDefinitionBuilder.genericBeanDefinition(ServiceB.class).getBeanDefinition());  
  
 //触发所有单例bean初始化  
 factory.preInstantiateSingletons();  
  
 System.out.println("销毁serviceB");  
 //销毁指定的bean  
 factory.destroySingleton("serviceB");  
}

@1：放入了一个自定义的DestructionAwareBeanPostProcessor

@2：放入了CommonAnnotationBeanPostProcessor，这个会处理bean中标注@PreDestroy注解的方法

##### 看效果运行输出

create class com.javacode2018.lesson002.demo13.ServiceB  
销毁serviceB  
准备销毁bean：serviceB  
preDestroy()

#### 案例3：看一下销毁阶段的执行顺序

实际上ApplicationContext内部已经将spring内部一些常见的必须的BeannPostProcessor自动装配到beanPostProcessors列表中，比如我们熟悉的下面的几个：

1.org.springframework.context.annotation.CommonAnnotationBeanPostProcessor  
 用来处理@Resource、@PostConstruct、@PreDestroy的  
2.org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor  
 用来处理@Autowired、@Value注解  
3.org.springframework.context.support.ApplicationContextAwareProcessor  
 用来回调Bean实现的各种Aware接口

所以通过ApplicationContext来销毁bean，会触发3中方式的执行。

下面我们就以AnnotationConfigApplicationContext来演示一下销毁操作。

来一个类

package com.javacode2018.lesson002.demo14;  
  
import org.springframework.beans.factory.DisposableBean;  
  
import javax.annotation.PreDestroy;  
  
public class ServiceA implements DisposableBean {  
  
 public ServiceA() {  
 System.out.println("创建ServiceA实例");  
 }  
  
 @PreDestroy  
 public void preDestroy1() {  
 System.out.println("preDestroy1()");  
 }  
  
 @PreDestroy  
 public void preDestroy2() {  
 System.out.println("preDestroy2()");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println("DisposableBean接口中的destroy()");  
 }  
  
 //自定义的销毁方法  
 public void customDestroyMethod() { //@1  
 System.out.println("我是自定义的销毁方法:customDestroyMethod()");  
 }  
}

上面的类中有2个方法标注了@PreDestroy

这个类实现了DisposableBean接口，重写了接口的中的destroy方法

@1：这个destroyMethod我们一会通过@Bean注解的方式，将其指定为自定义方法。

来看测试用例

package com.javacode2018.lesson002.demo14;  
  
import org.junit.Test;  
import org.springframework.beans.factory.annotation.Configurable;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
import org.springframework.context.annotation.Bean;  
  
@Configurable  
public class DestroyTest {  
  
 @Bean(destroyMethod = "customDestroyMethod") //@1  
 public ServiceA serviceA() {  
 return new ServiceA();  
 }  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(DestroyTest.class);  
 //启动容器  
 System.out.println("准备启动容器");  
 context.refresh();  
 System.out.println("容器启动完毕");  
 System.out.println("serviceA：" + context.getBean(ServiceA.class));  
 //关闭容器  
 System.out.println("准备关闭容器");  
 //调用容器的close方法，会触发bean的销毁操作  
 context.close(); //@2  
 System.out.println("容器关闭完毕");  
 }  
}

上面这个类标注了@Configuration，表示是一个配置类，内部有个@Bean标注的方法，表示使用这个方法来定义一个bean。

@1：通过destroyMethod属性将customDestroyMethod指定为自定义销毁方法

@2：关闭容器，触发bean销毁操作

来运行test1，输出

准备启动容器  
创建ServiceA实例  
容器启动完毕  
serviceA：com.javacode2018.lesson002.demo14.ServiceA@243c4f91  
准备关闭容器  
preDestroy1()  
preDestroy2()  
DisposableBean接口中的destroy()  
我是自定义的销毁方法:customDestroyMethod()  
容器关闭完毕

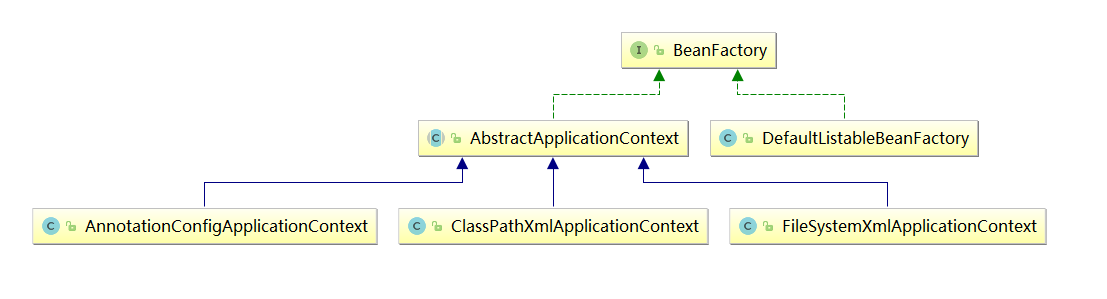
可以看出销毁方法调用的顺序：

1. @PreDestroy标注的所有方法
2. DisposableBean接口中的destroy()
3. 自定义的销毁方法

下面来说一个非常非常重要的类，打起精神，一定要注意看。

### AbstractApplicationContext类（非常重要的类）

来看一下UML图：



#### BeanFactory接口

这个我们已经很熟悉了，Bean工厂的顶层接口

#### DefaultListableBeanFactory类

实现了BeanFactory接口，可以说这个可以是BeanFactory接口真正的唯一实现，内部真正实现了bean生命周期中的所有代码。

其他的一些类都是依赖于DefaultListableBeanFactory类，将请求转发给DefaultListableBeanFactory进行bean的处理的。

#### 其他3个类

我们经常用到的就是这3个类：AnnotationConfigApplicationContext/ClassPathXmlApplicationContext/FileSystemXmlApplicationContext这3个类，他们的主要内部的功能是依赖他的父类AbstractApplicationContext来实现的，所以大家主要看AbstractApplicationContext这个类。

#### AbstractApplicationContext类

这个类中有2个比较重要的方法

public abstract ConfigurableListableBeanFactory getBeanFactory() throws IllegalStateException;  
protected void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory)

大家是否注意过我们使用AnnotationConfigApplicationContext的时候，经常调用reflush方法，这个方法内部就会调用上面这2个方法。

##### 第一个方法：getBeanFactory()

返回当前应用上下文中的ConfigurableListableBeanFactory，这也是个接口类型的，这个接口有一个唯一的实现类：DefaultListableBeanFactory。

有没有很熟悉，上面说过：DefaultListableBeanFactory是BeanFactory真正的唯一实现。

应用上线文中就会使用这个ConfigurableListableBeanFactory来操作spring容器。

##### 第二个方法：registerBeanPostProcessors

**说的通俗点：这个方法就是向ConfigurableListableBeanFactory中注册BeanPostProcessor，内容会从spring容器中获取所有类型的BeanPostProcessor，将其添加到DefaultListableBeanFactory#beanPostProcessors列表中**

看一下这个方法的源码：

protected void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory) {  
 PostProcessorRegistrationDelegate.registerBeanPostProcessors(beanFactory, this);  
}

会将请求转发给PostProcessorRegistrationDelegate#registerBeanPostProcessors。

内部比较长，大家可以去看一下源码，这个方法内部主要用到了4个BeanPostProcessor类型的List集合。

List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();  
List<BeanPostProcessor> orderedPostProcessors  
List<BeanPostProcessor> nonOrderedPostProcessors;  
List<BeanPostProcessor> internalPostProcessors = new ArrayList<>();

**先说一下：当到方法的时候，spring容器中已经完成了所有Bean的注册。**

spring会从容器中找出所有类型的BeanPostProcessor列表，然后按照下面的规则将其分别放到上面的4个集合中，上面4个集合中的BeanPostProcessor会被依次添加到DefaultListableBeanFactory#beanPostProcessors列表中，来看一下4个集合的分别放的是那些BeanPostProcessor：

###### priorityOrderedPostProcessors（指定优先级的BeanPostProcessor）

实现org.springframework.core.PriorityOrdered接口的BeanPostProcessor，但是不包含MergedBeanDefinitionPostProcessor类型的

###### orderedPostProcessors（指定了顺序的BeanPostProcessor）

实现了org.springframework.core.annotation.Order接口的BeanPostProcessor，但是不包含MergedBeanDefinitionPostProcessor类型的

###### nonOrderedPostProcessors（未指定顺序的BeanPostProcessor）

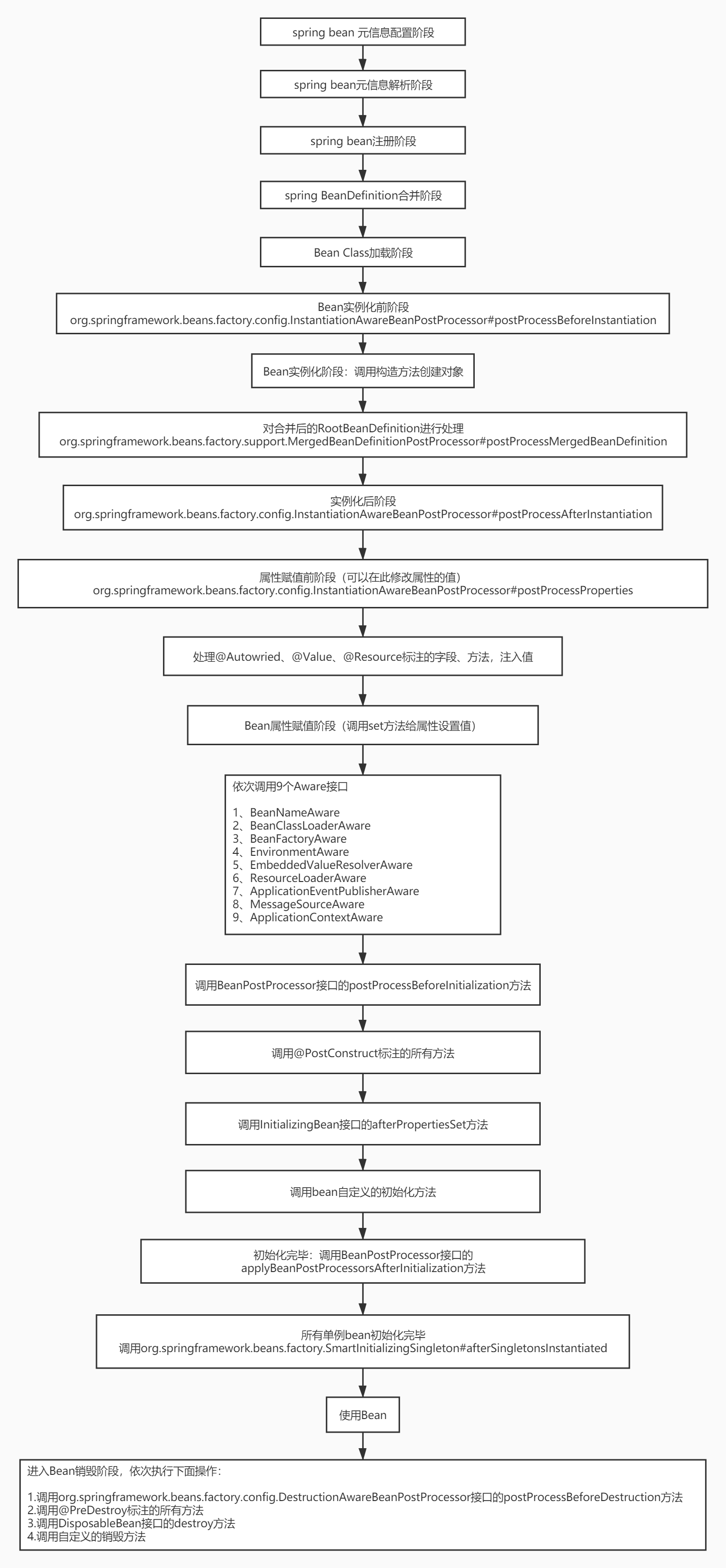
上面2中类型置为以及MergedBeanDefinitionPostProcessor之外的

###### internalPostProcessors

MergedBeanDefinitionPostProcessor类型的BeanPostProcessor列表。

大家可以去看一下CommonAnnotationBeanPostProcessor和AutowiredAnnotationBeanPostProcessor，这两个类都实现了PriorityOrdered接口，但是他们也实现了MergedBeanDefinitionPostProcessor接口，所以最终他们会被丢到internalPostProcessors这个集合中，会被放入BeanPostProcessor的最后面。

### Bean生命周期流程图



### Spring学习建议

这里我列一下自己学习spring的一些方法。

##### 看官方文档

spring中的所有知识点，官网上都有，质量方面是非常高的，可能对英文有些要求，不过可以使用翻译软件翻一下。

官网地址：

https://docs.spring.io/spring/docs/5.2.3.RELEASE/spring-framework-reference/

##### 看好的视频

视频算是学习技术最快的一种方式，视频中可以呈现更多细节的东西，同样的内容，可能视频只需要1小时，而用文档描述出来可能需要写一整天，视频的信息量更大，让观看者更容易理解一些。

网上关于spring的视频也比较多，质量不一，很难去选择，不过有一套我强烈建议大家去看，这套视频是有门槛的，前提是你对spring要有一定的了解。

就是极客时间上面的《小马哥讲Spring核心编程思想》，这个系列如果你都能看懂，spring方面不说多的，在大部分公司基本上是无敌状态，目前这套课程已经出了一半了，所以的内容我都看过，质量非常棒，有兴趣的可以扫描下面链接购买。

##### 看源码

想窥视spring中的原貌，成为顶尖高手，源码是必须要看的，spring整体的设计是非常优秀的，里面用到了很多设计模式，很多java中的api，看源码的过程中，这些好东西慢慢就变成你自己的了。

##### 写博客

技术有没有掌握，你写篇文章就知道了，如果你能通过文章的方式将技术描述出来，让别人能够看懂，那么说明你确实是掌握了这个技术。写博客的过程中会让你想各种办法去深入了解需要介绍的要点，这样收获是非常大的。

### 总结

本文内容比较多，强烈建议大家先收藏，要反复看，反复去联系，一定要掌握。

关于spring方面有任何问题的，欢迎大家留言。

顺便说下：帮忙点个再看，转发到朋友圈，让更多朋友一起学习。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解

## Spring系列第24篇：父子容器

又一次被面试官带到坑里面了。

面试官：springmvc用过么？

我：用过啊，经常用呢

面试官：springmvc中为什么需要用父子容器？

我：嗯。。。没听明白你说的什么。

面试官：就是controller层交给一个spring容器加载，其他的service和dao层交给另外一个spring容器加载，web.xml中有这块配置，这两个容器组成了父子容器的关系。

我：哦，原来是这块啊，我想起来了，我看大家都这么用，所以我也这么用

面试官：有没有考虑过为什么？

我：我在网上看大家都这么用，所以我也这么用了，具体也不知道为什么，不过用起来还挺顺手的

面试官：如果只用一个容器可以么，所有的配置都交给一个spring容器加载？

我：应该不行吧！

面试官：确定不行么？

我：让我想一会。。。。。我感觉是可以的，也可以正常运行。

面试官：那我们又回到了开头的问题，为什么要用父子容器呢？

我：我叫你哥好么，别这么玩我了，被你绕晕了？

面试官：好吧，你回去试试看吧，下次再来告诉我，出门右转，不送！

我：脸色变绿了，灰头土脸的走了。

回去之后，我好好研究了一番。

### 主要的问题

1. 什么是父子容器？
2. 为什么需要用父子容器？
3. 父子容器如何使用？

下面我们就来探讨探讨。

### 我们先来看一个案例

系统中有2个模块：module1和module2，两个模块是独立开发的，module2会使用到module1中的一些类，module1会将自己打包为jar提供给module2使用，我们来看一下这2个模块的代码。

#### 模块1

放在module1包中，有3个类

##### Service1

package com.javacode2018.lesson002.demo17.module1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
 public String m1() {  
 return "我是module1中的Servce1中的m1方法";  
 }  
}

##### Service2

package com.javacode2018.lesson002.demo17.module1;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
  
 @Autowired  
 private com.javacode2018.lesson002.demo17.module1.Service1 service1; //@1  
  
 public String m1() { //@2  
 return this.service1.m1();  
 }  
  
}

上面2个类，都标注了@Compontent注解，会被spring注册到容器中。

@1：Service2中需要用到Service1，标注了@Autowired注解，会通过spring容器注入进来

@2：Service2中有个m1方法，内部会调用service的m1方法。

##### 来个spring配置类：Module1Config

package com.javacode2018.lesson002.demo17.module1;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class Module1Config {  
}

上面使用了@CompontentScan注解，会自动扫描当前类所在的包中的所有类，将标注有@Compontent注解的类注册到spring容器，即Service1和Service2会被注册到spring容器。

#### 再来看模块2

放在module2包中，也是有3个类，和模块1中的有点类似。

##### Service1

模块2中也定义了一个Service1，内部提供了一个m2方法，如下：

package com.javacode2018.lesson002.demo17.module2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
 public String m2() {  
 return "我是module2中的Servce1中的m2方法";  
 }  
}

##### Service3

package com.javacode2018.lesson002.demo17.module2;  
  
import com.javacode2018.lesson002.demo17.module1.Service2;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service3 {  
 //使用模块2中的Service1  
 @Autowired  
 private com.javacode2018.lesson002.demo17.module2.Service1 service1; //@1  
 //使用模块1中的Service2  
 @Autowired  
 private com.javacode2018.lesson002.demo17.module1.Service2 service2; //@2  
  
 public String m1() {  
 return this.service2.m1();  
 }  
  
 public String m2() {  
 return this.service1.m2();  
 }  
  
}

@1：使用module2中的Service1

@2：使用module1中的Service2

##### 先来思考一个问题

**上面的这些类使用spring来操作会不会有问题？会有什么问题？**

这个问题还是比较简单的，大部分人都可以看出来，会报错，因为两个模块中都有Service1，被注册到spring容器的时候，bean名称会冲突，导致注册失败。

##### 来个测试类，看一下效果

package com.javacode2018.lesson002.demo17;  
  
import com.javacode2018.lesson001.demo21.Config;  
import com.javacode2018.lesson002.demo17.module1.Module1Config;  
import com.javacode2018.lesson002.demo17.module2.Module2Config;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class ParentFactoryTest {  
  
 @Test  
 public void test1() {  
 //定义容器  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 //注册bean  
 context.register(Module1Config.class, Module2Config.class); //@1  
 //启动容器  
 context.refresh();  
 }  
}

@1：将Module1Config、Module2Config注册到容器，spring内部会自动解析这两个类上面的注解，即：@CompontentScan注解，然后会进行包扫描，将标注了@Compontent的类注册到spring容器。

##### 运行test1输出

下面是部分输出：

Caused by: org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'service1' for bean class [com.javacode2018.lesson002.demo17.module2.Service1] conflicts with existing, non-compatible bean definition of same name and class [com.javacode2018.lesson002.demo17.module1.Service1]

service1这个bean的名称冲突了。

##### 那么我们如何解决？

对module1中的Service1进行修改？这个估计是行不通的，module1是别人以jar的方式提供给我们的，源码我们是无法修改的。

而module2是我们自己的开发的，里面的东西我们可以随意调整，那么我们可以去修改一下module2中的Service1，可以修改一下类名，或者修改一下这个bean的名称，此时是可以解决问题的。

不过大家有没有想过一个问题：如果我们的模块中有很多类都出现了这种问题，此时我们一个个去重构，还是比较痛苦的，并且代码重构之后，还涉及到重新测试的问题，工作量也是蛮大的，这些都是风险。

而spring中的父子容器就可以很好的解决上面这种问题。

### 什么是父子容器

创建spring容器的时候，可以给当前容器指定一个父容器。

#### BeanFactory的方式

//创建父容器parentFactory  
DefaultListableBeanFactory parentFactory = new DefaultListableBeanFactory();  
//创建一个子容器childFactory  
DefaultListableBeanFactory childFactory = new DefaultListableBeanFactory();  
//调用setParentBeanFactory指定父容器  
childFactory.setParentBeanFactory(parentFactory);

#### ApplicationContext的方式

//创建父容器  
AnnotationConfigApplicationContext parentContext = new AnnotationConfigApplicationContext();  
//启动父容器  
parentContext.refresh();  
  
//创建子容器  
AnnotationConfigApplicationContext childContext = new AnnotationConfigApplicationContext();  
//给子容器设置父容器  
childContext.setParent(parentContext);  
//启动子容器  
childContext.refresh();

上面代码还是比较简单的，大家都可以看懂。

我们需要了解父子容器的特点，这些是比较关键的，如下。

#### 父子容器特点

1. **父容器和子容器是相互隔离的，他们内部可以存在名称相同的bean**
2. **子容器可以访问父容器中的bean，而父容器不能访问子容器中的bean**
3. **调用子容器的getBean方法获取bean的时候，会沿着当前容器开始向上面的容器进行查找，直到找到对应的bean为止**
4. **子容器中可以通过任何注入方式注入父容器中的bean，而父容器中是无法注入子容器中的bean，原因是第2点**

### 使用父子容器解决开头的问题

#### 关键代码

@Test  
public void test2() {  
 //创建父容器  
 AnnotationConfigApplicationContext parentContext = new AnnotationConfigApplicationContext();  
 //向父容器中注册Module1Config配置类  
 parentContext.register(Module1Config.class);  
 //启动父容器  
 parentContext.refresh();  
  
 //创建子容器  
 AnnotationConfigApplicationContext childContext = new AnnotationConfigApplicationContext();  
 //向子容器中注册Module2Config配置类  
 childContext.register(Module2Config.class);  
 //给子容器设置父容器  
 childContext.setParent(parentContext);  
 //启动子容器  
 childContext.refresh();  
  
 //从子容器中获取Service3  
 Service3 service3 = childContext.getBean(Service3.class);  
 System.out.println(service3.m1());  
 System.out.println(service3.m2());  
}

#### 运行输出

我是module1中的Servce1中的m1方法  
我是module2中的Servce1中的m2方法

这次正常了。

### 父子容器使用注意点

我们使用容器的过程中，经常会使用到的一些方法，这些方法通常会在下面的两个接口中

org.springframework.beans.factory.BeanFactory  
org.springframework.beans.factory.ListableBeanFactory

这两个接口中有很多方法，这里就不列出来了，大家可以去看一下源码，这里要说的是使用父子容器的时候，有些需要注意的地方。

BeanFactory接口，是spring容器的顶层接口，这个接口中的方法是支持容器嵌套结构查找的，比如我们常用的getBean方法，就是这个接口中定义的，调用getBean方法的时候，会从沿着当前容器向上查找，直到找到满足条件的bean为止。

而ListableBeanFactory这个接口中的方法是不支持容器嵌套结构查找的，比如下面这个方法

String[] getBeanNamesForType(@Nullable Class<?> type)

获取指定类型的所有bean名称，调用这个方法的时候只会返回当前容器中符合条件的bean，而不会去递归查找其父容器中的bean。

来看一下案例代码，感受一下：

@Test  
public void test3() {  
 //创建父容器parentFactory  
 DefaultListableBeanFactory parentFactory = new DefaultListableBeanFactory();  
 //向父容器parentFactory注册一个bean[userName->"马士兵老师"]  
 parentFactory.registerBeanDefinition("userName",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("马士兵老师").  
 getBeanDefinition());  
  
 //创建一个子容器childFactory  
 DefaultListableBeanFactory childFactory = new DefaultListableBeanFactory();  
 //调用setParentBeanFactory指定父容器  
 childFactory.setParentBeanFactory(parentFactory);  
 //向子容器parentFactory注册一个bean[address->"上海"]  
 childFactory.registerBeanDefinition("address",  
 BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("上海").  
 getBeanDefinition());  
  
 System.out.println("获取bean【userName】：" + childFactory.getBean("userName"));//@1  
  
 System.out.println(Arrays.asList(childFactory.getBeanNamesForType(String.class))); //@2  
}

上面定义了2个容器

父容器：parentFactory，内部定义了一个String类型的bean：userName->马士兵老师

子容器：childFactory，内部也定义了一个String类型的bean：address->上海

@1：调用子容器的getBean方法，获取名称为userName的bean，userName这个bean是在父容器中定义的，而getBean方法是BeanFactory接口中定义的，支持容器层次查找，所以getBean是可以找到userName这个bean的

@2：调用子容器的getBeanNamesForType方法，获取所有String类型的bean名称，而getBeanNamesForType方法是ListableBeanFactory接口中定义的，这个接口中方法不支持层次查找，只会在当前容器中查找，所以这个方法只会返回子容器的address

我们来运行一下看看效果：

获取bean【userName】：马士兵老师  
[address]

结果和分析的一致。

**那么问题来了：有没有方式解决ListableBeanFactory接口不支持层次查找的问题？**

spring中有个工具类就是解决这个问题的，如下：

org.springframework.beans.factory.BeanFactoryUtils

这个类中提供了很多静态方法，有很多支持层次查找的方法，源码你们可以去细看一下，名称中包含有Ancestors的都是支持层次查找的。

在test2方法中加入下面的代码：

//层次查找所有符合类型的bean名称  
String[] beanNamesForTypeIncludingAncestors = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(childFactory, String.class);  
System.out.println(Arrays.asList(beanNamesForTypeIncludingAncestors));  
  
Map<String, String> beansOfTypeIncludingAncestors = BeanFactoryUtils.beansOfTypeIncludingAncestors(childFactory, String.class);  
System.out.println(Arrays.asList(beansOfTypeIncludingAncestors));

运行输出

[address, userName]  
[{address=上海, userName=马士兵老师}]

查找过程是按照层次查找所有满足条件的bean。

### 回头看一下springmvc父子容器的问题

**问题1：springmvc中只使用一个容器是否可以？**

只使用一个容器是可以正常运行的。

**问题2：那么springmvc中为什么需要用到父子容器？**

通常我们使用springmvc的时候，采用3层结构，controller层，service层，dao层；父容器中会包含dao层和service层，而子容器中包含的只有controller层；这2个容器组成了父子容器的关系，controller层通常会注入service层的bean。

采用父子容器可以避免有些人在service层去注入controller层的bean，导致整个依赖层次是比较混乱的。

父容器和子容器的需求也是不一样的，比如父容器中需要有事务的支持，会注入一些支持事务的扩展组件，而子容器中controller完全用不到这些，对这些并不关心，子容器中需要注入一下springmvc相关的bean，而这些bean父容器中同样是不会用到的，也是不关心一些东西，将这些相互不关心的东西隔开，可以有效的避免一些不必要的错误，而父子容器加载的速度也会快一些。

### 总结

1. 本文需掌握父子容器的用法，了解父子容器的特点：子容器可以访问父容器中bean，父容器无法访问子容器中的bean
2. BeanFactory接口支持层次查找
3. ListableBeanFactory接口不支持层次查找
4. BeanFactoryUtils工具类中提供了一些非常实用的方法，比如支持bean层次查找的方法等等

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解

## Spring系列第25篇：@PropertySource、@Value注解及动态刷新实现

疫情期间找工作确实有点难度，想拿到满意的薪资，确实要点实力啊！

面试官：Spring中的@Value用过么，介绍一下

我：@Value可以标注在字段上面，可以将外部配置文件中的数据，比如可以将数据库的一些配置信息放在配置文件中，然后通过@Value的方式将其注入到bean的一些字段中

面试官：那就是说@Value的数据来源于配置文件了？

我：嗯，我们项目最常用更多就是通过@Value来引用Properties文件中的配置

面试官：@Value数据来源还有其他方式么？

我：此时我异常开心，刚好问的我都研究过，我说：当然有，可以将配置信息放在db或者其他存储介质中，容器启动的时候，可以将这些信息加载到Environment中，@Value中应用的值最终是通过Environment来解析的，所以只需要扩展一下Environment就可以实现了。

面试官：不错嘛，看来你对spring研究的还是可以，是不是喜欢研究spring源码？

我：笑着说，嗯，平时有空的时候确实喜欢捣鼓捣鼓源码，感觉自己对spring了解的还可以，不能算精通，也算是半精通吧

面试官：看着我笑了笑，那@Value的注入的值可以动态刷新么？

我：应该可以吧，我记得springboot中有个@RefreshScope注解就可以实现你说的这个功能

面试官：那你可以说一下@RefreshScope是如何实现的么，可以大概介绍一下？

我：嗯。。。这个之前看过一点，不过没有看懂

面试官：没关系，你可以回去了再研究一下；你期望工资多少？

我：3万吧

面试官：今天的面试还算是可以的，不过如果@RefreshScope能回答上来就更好了，这块是个加分项，不过也确实有点难度，2.5万如何？

我：（心中默默想了想：2.5万，就是一个问题没有回答好，砍了5000，有点狠啊，我要回去再研究研究，3万肯定是没问题的），我说：最低2.9万

面试官：那谢谢你，今天面试就到这里，出门右拐，不送！

我有个好习惯，每次面试回去之后，都会进行复盘，把没有搞定的问题一定要想办法搞定，这样才不虚。

### 这次面试问题如下

1. @Value的用法
2. @Value数据来源
3. @Value动态刷新的问题

下面我们一个个来整理一下，将这几个问题搞定，助大家在疫情期间面试能够过关斩将，拿高薪。

### @Value的用法

系统中需要连接db，连接db有很多配置信息。

系统中需要发送邮件，发送邮件需要配置邮件服务器的信息。

还有其他的一些配置信息。

我们可以将这些配置信息统一放在一个配置文件中，上线的时候由运维统一修改。

那么系统中如何使用这些配置信息呢，spring中提供了@Value注解来解决这个问题。

通常我们会将配置信息以key=value的形式存储在properties配置文件中。

通过@Value("${配置文件中的key}")来引用指定的key对应的value。

#### @Value使用步骤

##### 步骤一：使用@PropertySource注解引入配置文件

将@PropertySource放在类上面，如下

@PropertySource({"配置文件路径1","配置文件路径2"...})

@PropertySource注解有个value属性，字符串数组类型，可以用来指定多个配置文件的路径。

如：

@Component  
@PropertySource({"classpath:com/javacode2018/lesson002/demo18/db.properties"})  
public class DbConfig {  
}

##### 步骤二：使用@Value注解引用配置文件的值

通过@Value引用上面配置文件中的值：

语法

@Value("${配置文件中的key:默认值}")  
@Value("${配置文件中的key}")

如：

@Value("${password:123}")

上面如果password不存在，将123作为值

@Value("${password}")

上面如果password不存在，值为${password}

假如配置文件如下

jdbc.url=jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8  
jdbc.username=javacode  
jdbc.password=javacode

使用方式如下：

@Value("${jdbc.url}")  
private String url;  
  
@Value("${jdbc.username}")  
private String username;  
  
@Value("${jdbc.password}")  
private String password;

下面来看案例

##### 案例

###### 来个配置文件db.properties

jdbc.url=jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8  
jdbc.username=javacode  
jdbc.password=javacode

###### 来个配置类，使用@PropertySource引入上面的配置文件

package com.javacode2018.lesson002.demo18.test1;  
  
import org.springframework.beans.factory.annotation.Configurable;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.PropertySource;  
  
@Configurable  
@ComponentScan  
@PropertySource({"classpath:com/javacode2018/lesson002/demo18/db.properties"})  
public class MainConfig1 {  
}

###### 来个类，使用@Value来使用配置文件中的信息

package com.javacode2018.lesson002.demo18.test1;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
@Component  
public class DbConfig {  
  
 @Value("${jdbc.url}")  
 private String url;  
  
 @Value("${jdbc.username}")  
 private String username;  
  
 @Value("${jdbc.password}")  
 private String password;  
  
 public String getUrl() {  
 return url;  
 }  
  
 public void setUrl(String url) {  
 this.url = url;  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
 @Override  
 public String toString() {  
 return "DbConfig{" +  
 "url='" + url + '\'' +  
 ", username='" + username + '\'' +  
 ", password='" + password + '\'' +  
 '}';  
 }  
}

上面重点在于注解@Value注解，注意@Value注解中的

###### 来个测试用例

package com.javacode2018.lesson002.demo18;  
  
import com.javacode2018.lesson002.demo18.test1.DbConfig;  
import com.javacode2018.lesson002.demo18.test1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class ValueTest {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
  
 DbConfig dbConfig = context.getBean(DbConfig.class);  
 System.out.println(dbConfig);  
 }  
}

###### 运行输出

DbConfig{url='jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8', username='javacode', password='javacode'}

上面用起来比较简单，很多用过的人看一眼就懂了，这也是第一个问题，多数人都是ok的，下面来看@Value中数据来源除了配置文件的方式，是否还有其他方式。

### @Value数据来源

通常情况下我们@Value的数据来源于配置文件，不过，还可以用其他方式，比如我们可以将配置文件的内容放在数据库，这样修改起来更容易一些。

我们需要先了解一下@Value中数据来源于spring的什么地方。

spring中有个类

org.springframework.core.env.PropertySource

可以将其理解为一个配置源，里面包含了key->value的配置信息，可以通过这个类中提供的方法获取key对应的value信息

内部有个方法：

public abstract Object getProperty(String name);

通过name获取对应的配置信息。

系统有个比较重要的接口

org.springframework.core.env.Environment

用来表示环境配置信息，这个接口有几个方法比较重要

String resolvePlaceholders(String text);  
MutablePropertySources getPropertySources();

resolvePlaceholders用来解析${text}的，@Value注解最后就是调用这个方法来解析的。

getPropertySources返回MutablePropertySources对象，来看一下这个类

public class MutablePropertySources implements PropertySources {  
  
 private final List<PropertySource<?>> propertySourceList = new CopyOnWriteArrayList<>();  
   
}

内部包含一个propertySourceList列表。

spring容器中会有一个Environment对象，最后会调用这个对象的resolvePlaceholders方法解析@Value。

大家可以捋一下，最终解析@Value的过程：

1. 将@Value注解的value参数值作为Environment.resolvePlaceholders方法参数进行解析  
2. Environment内部会访问MutablePropertySources来解析  
3. MutablePropertySources内部有多个PropertySource，此时会遍历PropertySource列表，调用PropertySource.getProperty方法来解析key对应的值

通过上面过程，如果我们想改变@Value数据的来源，只需要将配置信息包装为PropertySource对象，丢到Environment中的MutablePropertySources内部就可以了。

下面我们就按照这个思路来一个。

来个邮件配置信息类，内部使用@Value注入邮件配置信息

package com.javacode2018.lesson002.demo18.test2;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 邮件配置信息  
 \*/  
@Component  
public class MailConfig {  
  
 @Value("${mail.host}")  
 private String host;  
  
 @Value("${mail.username}")  
 private String username;  
  
 @Value("${mail.password}")  
 private String password;  
  
 public String getHost() {  
 return host;  
 }  
  
 public void setHost(String host) {  
 this.host = host;  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
 @Override  
 public String toString() {  
 return "MailConfig{" +  
 "host='" + host + '\'' +  
 ", username='" + username + '\'' +  
 ", password='" + password + '\'' +  
 '}';  
 }  
}

再来个类DbUtil，getMailInfoFromDb方法模拟从db中获取邮件配置信息，存放在map中

package com.javacode2018.lesson002.demo18.test2;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class DbUtil {  
 /\*\*  
 \* 模拟从db中获取邮件配置信息  
 \*  
 \* @return  
 \*/  
 public static Map<String, Object> getMailInfoFromDb() {  
 Map<String, Object> result = new HashMap<>();  
 result.put("mail.host", "smtp.qq.com");  
 result.put("mail.username", "路人");  
 result.put("mail.password", "123");  
 return result;  
 }  
}

来个spring配置类

package com.javacode2018.lesson002.demo18.test2;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ComponentScan  
public class MainConfig2 {  
}

下面是重点代码

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
   
 /\*下面这段是关键 start\*/  
 //模拟从db中获取配置信息  
 Map<String, Object> mailInfoFromDb = DbUtil.getMailInfoFromDb();  
 //将其丢在MapPropertySource中（MapPropertySource类是spring提供的一个类，是PropertySource的子类）  
 MapPropertySource mailPropertySource = new MapPropertySource("mail", mailInfoFromDb);  
 //将mailPropertySource丢在Environment中的PropertySource列表的第一个中，让优先级最高  
 context.getEnvironment().getPropertySources().addFirst(mailPropertySource);  
 /\*上面这段是关键 end\*/  
   
 context.register(MainConfig2.class);  
 context.refresh();  
 MailConfig mailConfig = context.getBean(MailConfig.class);  
 System.out.println(mailConfig);  
}

注释比较详细，就不详细解释了。

直接运行，看效果

MailConfig{host='smtp.qq.com', username='路人', password='123'}

有没有感觉很爽，此时你们可以随意修改DbUtil.getMailInfoFromDb，具体数据是从db中来，来时从redis或者其他介质中来，任由大家发挥。

上面重点是下面这段代码，大家需要理解

/\*下面这段是关键 start\*/  
//模拟从db中获取配置信息  
Map<String, Object> mailInfoFromDb = DbUtil.getMailInfoFromDb();  
//将其丢在MapPropertySource中（MapPropertySource类是spring提供的一个类，是PropertySource的子类）  
MapPropertySource mailPropertySource = new MapPropertySource("mail", mailInfoFromDb);  
//将mailPropertySource丢在Environment中的PropertySource列表的第一个中，让优先级最高  
context.getEnvironment().getPropertySources().addFirst(mailPropertySource);  
/\*上面这段是关键 end\*/

咱们继续看下一个问题

**如果我们将配置信息放在db中，可能我们会通过一个界面来修改这些配置信息，然后保存之后，希望系统在不重启的情况下，让这些值在spring容器中立即生效。**

@Value动态刷新的问题的问题，springboot中使用@RefreshScope实现了。

### 实现@Value动态刷新

#### 先了解一个知识点

这块需要先讲一个知识点，用到的不是太多，所以很多人估计不太了解，但是非常重要的一个点，我们来看一下。

这个知识点是自定义bean作用域，对这块不了解的先看一下这篇文章：[bean作用域详解](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933960&idx=1&sn=f4308f8955f87d75963c379c2a0241f4&chksm=88621e76bf159760d404c253fa6716d3ffce4de8df0fc1d0d5dd0cf00a81bc170a30829ee58f&token=1314297026&lang=zh_CN&scene=21" \l "wechat_redirect)

bean作用域中有个地方没有讲，来看一下@Scope这个注解的源码，有个参数是：

ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;

这个参数的值是个ScopedProxyMode类型的枚举，值有下面4中

public enum ScopedProxyMode {  
 DEFAULT,  
 NO,  
 INTERFACES,  
 TARGET\_CLASS;  
}

前面3个，不讲了，直接讲最后一个值是干什么的。

当@Scope中proxyMode为TARGET\_CLASS的时候，会给当前创建的bean通过cglib生成一个代理对象，通过这个代理对象来访问目标bean对象。

理解起来比较晦涩，还是来看代码吧，容易理解一些，来个自定义的Scope案例。

**自定义一个bean作用域的注解**

package com.javacode2018.lesson002.demo18.test3;  
  
import org.springframework.context.annotation.Scope;  
import org.springframework.context.annotation.ScopedProxyMode;  
  
import java.lang.annotation.\*;  
  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Scope(BeanMyScope.SCOPE\_MY) //@1  
public @interface MyScope {  
 /\*\*  
 \* @see Scope#proxyMode()  
 \*/  
 ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET\_CLASS;//@2  
}

@1：使用了@Scope注解，value为引用了一个常量，值为my，一会下面可以看到。

@2：注意这个地方，参数名称也是proxyMode，类型也是ScopedProxyMode，而@Scope注解中有个和这个同样类型的参数，spring容器解析的时候，会将这个参数的值赋给@MyScope注解上面的@Scope注解的proxyMode参数，所以此处我们设置proxyMode值，最后的效果就是直接改变了@Scope中proxyMode参数的值。此处默认值取的是ScopedProxyMode.TARGET\_CLASS

**@MyScope注解对应的Scope实现如下**

package com.javacode2018.lesson002.demo18.test3;  
  
import org.springframework.beans.factory.ObjectFactory;  
import org.springframework.beans.factory.config.Scope;  
import org.springframework.lang.Nullable;  
  
/\*\*  
 \* @see MyScope 作用域的实现  
 \*/  
public class BeanMyScope implements Scope {  
  
 public static final String SCOPE\_MY = "my"; //@1  
  
 @Override  
 public Object get(String name, ObjectFactory<?> objectFactory) {   
 System.out.println("BeanMyScope >>>>>>>>> get:" + name); //@2  
 return objectFactory.getObject(); //@3  
 }  
  
 @Nullable  
 @Override  
 public Object remove(String name) {  
 return null;  
 }  
  
 @Override  
 public void registerDestructionCallback(String name, Runnable callback) {  
  
 }  
  
 @Nullable  
 @Override  
 public Object resolveContextualObject(String key) {  
 return null;  
 }  
  
 @Nullable  
 @Override  
 public String getConversationId() {  
 return null;  
 }  
}

@1：定义了一个常量，作为作用域的值

@2：这个get方法是关键，自定义作用域会自动调用这个get方法来创建bean对象，这个地方输出了一行日志，为了一会方便看效果

@3：通过objectFactory.getObject()获取bean实例返回。

**下面来创建个类，作用域为上面自定义的作用域**

package com.javacode2018.lesson002.demo18.test3;  
  
import org.springframework.stereotype.Component;  
  
import java.util.UUID;  
  
@Component  
@MyScope //@1   
public class User {  
  
 private String username;  
  
 public User() {   
 System.out.println("---------创建User对象" + this); //@2  
 this.username = UUID.randomUUID().toString(); //@3  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
}

@1：使用了自定义的作用域@MyScope

@2：构造函数中输出一行日志

@3：给username赋值，通过uuid随机生成了一个

**来个spring配置类，加载上面@Compontent标注的组件**

package com.javacode2018.lesson002.demo18.test3;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@ComponentScan  
@Configuration  
public class MainConfig3 {  
}

**下面重点来了，测试用例**

@Test  
public void test3() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 //将自定义作用域注册到spring容器中  
 context.getBeanFactory().registerScope(BeanMyScope.SCOPE\_MY, new BeanMyScope());//@1  
 context.register(MainConfig3.class);  
 context.refresh();  
  
 System.out.println("从容器中获取User对象");  
 User user = context.getBean(User.class); //@2  
 System.out.println("user对象的class为：" + user.getClass()); //@3  
   
 System.out.println("多次调用user的getUsername感受一下效果\n");  
 for (int i = 1; i <= 3; i++) {  
 System.out.println(String.format("\*\*\*\*\*\*\*\*\n第%d次开始调用getUsername", i));  
 System.out.println(user.getUsername());  
 System.out.println(String.format("第%d次调用getUsername结束\n\*\*\*\*\*\*\*\*\n", i));  
 }  
}

@1：将自定义作用域注册到spring容器中

@2：从容器中获取User对应的bean

@3：输出这个bean对应的class，一会认真看一下，这个类型是不是User类型的

代码后面又搞了3次循环，调用user的getUsername方法，并且方法前后分别输出了一行日志。

**见证奇迹的时候到了，运行输出**

从容器中获取User对象  
user对象的class为：class com.javacode2018.lesson002.demo18.test3.User$$EnhancerBySpringCGLIB$$80233127  
多次调用user的getUsername感受一下效果  
  
\*\*\*\*\*\*\*\*  
第1次开始调用getUsername  
BeanMyScope >>>>>>>>> get:scopedTarget.user  
---------创建User对象com.javacode2018.lesson002.demo18.test3.User@6a370f4  
7b41aa80-7569-4072-9d40-ec9bfb92f438  
第1次调用getUsername结束  
\*\*\*\*\*\*\*\*  
  
\*\*\*\*\*\*\*\*  
第2次开始调用getUsername  
BeanMyScope >>>>>>>>> get:scopedTarget.user  
---------创建User对象com.javacode2018.lesson002.demo18.test3.User@1613674b  
01d67154-95f6-44bb-93ab-05a34abdf51f  
第2次调用getUsername结束  
\*\*\*\*\*\*\*\*  
  
\*\*\*\*\*\*\*\*  
第3次开始调用getUsername  
BeanMyScope >>>>>>>>> get:scopedTarget.user  
---------创建User对象com.javacode2018.lesson002.demo18.test3.User@27ff5d15  
76d0e86f-8331-4303-aac7-4acce0b258b8  
第3次调用getUsername结束  
\*\*\*\*\*\*\*\*

从输出的前2行可以看出：

1. 调用context.getBean(User.class)从容器中获取bean的时候，此时并没有调用User的构造函数去创建User对象
2. 第二行输出的类型可以看出，getBean返回的user对象是一个cglib代理对象。

**后面的日志输出可以看出，每次调用user.getUsername方法的时候，内部自动调用了BeanMyScope#get 方法和 User的构造函数。**

**通过上面的案例可以看出，当自定义的Scope中proxyMode=ScopedProxyMode.TARGET\_CLASS的时候，会给这个bean创建一个代理对象，调用代理对象的任何方法，都会调用这个自定义的作用域实现类（上面的BeanMyScope）中get方法来重新来获取这个bean对象。**

#### 动态刷新@Value具体实现

那么我们可以利用上面讲解的这种特性来实现@Value的动态刷新，可以实现一个自定义的Scope，这个自定义的Scope支持@Value注解自动刷新，需要使用@Value注解自动刷新的类上面可以标注这个自定义的注解，当配置修改的时候，调用这些bean的任意方法的时候，就让spring重启初始化一下这个bean，这个思路就可以实现了，下面我们来写代码。

##### 先来自定义一个Scope：RefreshScope

package com.javacode2018.lesson002.demo18.test4;  
  
import org.springframework.context.annotation.Scope;  
import org.springframework.context.annotation.ScopedProxyMode;  
  
import java.lang.annotation.\*;  
  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Scope(BeanRefreshScope.SCOPE\_REFRESH)  
@Documented  
public @interface RefreshScope {  
 ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET\_CLASS; //@1  
}

要求标注@RefreshScope注解的类支持动态刷新@Value的配置

@1：这个地方是个关键，使用的是ScopedProxyMode.TARGET\_CLASS

##### 这个自定义Scope对应的解析类

下面类中有几个无关的方法去掉了，可以忽略

package com.javacode2018.lesson002.demo18.test4;  
  
  
import org.springframework.beans.factory.ObjectFactory;  
import org.springframework.beans.factory.config.Scope;  
import org.springframework.lang.Nullable;  
  
import java.util.concurrent.ConcurrentHashMap;  
  
public class BeanRefreshScope implements Scope {  
  
 public static final String SCOPE\_REFRESH = "refresh";  
  
 private static final BeanRefreshScope INSTANCE = new BeanRefreshScope();  
  
 //来个map用来缓存bean  
 private ConcurrentHashMap<String, Object> beanMap = new ConcurrentHashMap<>(); //@1  
  
 private BeanRefreshScope() {  
 }  
  
 public static BeanRefreshScope getInstance() {  
 return INSTANCE;  
 }  
  
 /\*\*  
 \* 清理当前  
 \*/  
 public static void clean() {  
 INSTANCE.beanMap.clear();  
 }  
  
 @Override  
 public Object get(String name, ObjectFactory<?> objectFactory) {  
 Object bean = beanMap.get(name);  
 if (bean == null) {  
 bean = objectFactory.getObject();  
 beanMap.put(name, bean);  
 }  
 return bean;  
 }  
  
}

上面的get方法会先从beanMap中获取，获取不到会调用objectFactory的getObject让spring创建bean的实例，然后丢到beanMap中

上面的clean方法用来清理beanMap中当前已缓存的所有bean

##### 来个邮件配置类，使用@Value注解注入配置，这个bean作用域为自定义的@RefreshScope

package com.javacode2018.lesson002.demo18.test4;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 邮件配置信息  
 \*/  
@Component  
@RefreshScope //@1  
public class MailConfig {  
  
 @Value("${mail.username}") //@2  
 private String username;  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 @Override  
 public String toString() {  
 return "MailConfig{" +  
 "username='" + username + '\'' +  
 '}';  
 }  
}

@1：使用了自定义的作用域@RefreshScope

@2：通过@Value注入mail.username对一个的值

重写了toString方法，一会测试时候可以看效果。

##### 再来个普通的bean，内部会注入MailConfig

package com.javacode2018.lesson002.demo18.test4;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MailService {  
 @Autowired  
 private MailConfig mailConfig;  
  
 @Override  
 public String toString() {  
 return "MailService{" +  
 "mailConfig=" + mailConfig +  
 '}';  
 }  
}

代码比较简单，重写了toString方法，一会测试时候可以看效果。

##### 来个类，用来从db中获取邮件配置信息

package com.javacode2018.lesson002.demo18.test4;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.UUID;  
  
public class DbUtil {  
 /\*\*  
 \* 模拟从db中获取邮件配置信息  
 \*  
 \* @return  
 \*/  
 public static Map<String, Object> getMailInfoFromDb() {  
 Map<String, Object> result = new HashMap<>();  
 result.put("mail.username", UUID.randomUUID().toString());  
 return result;  
 }  
}

##### 来个spring配置类，扫描加载上面的组件

package com.javacode2018.lesson002.demo18.test4;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ComponentScan  
public class MainConfig4 {  
}

##### 来个工具类

内部有2个方法，如下：

package com.javacode2018.lesson002.demo18.test4;  
  
import org.springframework.context.support.AbstractApplicationContext;  
import org.springframework.core.env.MapPropertySource;  
  
import java.util.Map;  
  
public class RefreshConfigUtil {  
 /\*\*  
 \* 模拟改变数据库中都配置信息  
 \*/  
 public static void updateDbConfig(AbstractApplicationContext context) {  
 //更新context中的mailPropertySource配置信息  
 refreshMailPropertySource(context);  
  
 //清空BeanRefreshScope中所有bean的缓存  
 BeanRefreshScope.getInstance().clean();  
 }  
  
 public static void refreshMailPropertySource(AbstractApplicationContext context) {  
 Map<String, Object> mailInfoFromDb = DbUtil.getMailInfoFromDb();  
 //将其丢在MapPropertySource中（MapPropertySource类是spring提供的一个类，是PropertySource的子类）  
 MapPropertySource mailPropertySource = new MapPropertySource("mail", mailInfoFromDb);  
 context.getEnvironment().getPropertySources().addFirst(mailPropertySource);  
 }  
  
}

updateDbConfig方法模拟修改db中配置的时候需要调用的方法，方法中2行代码，第一行代码调用refreshMailPropertySource方法修改容器中邮件的配置信息

BeanRefreshScope.getInstance().clean()用来清除BeanRefreshScope中所有已经缓存的bean，那么调用bean的任意方法的时候，会重新出发spring容器来创建bean，spring容器重新创建bean的时候，会重新解析@Value的信息，此时容器中的邮件配置信息是新的，所以@Value注入的信息也是新的。

来个测试用例

@Test  
public void test4() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.getBeanFactory().registerScope(BeanRefreshScope.SCOPE\_REFRESH, BeanRefreshScope.getInstance());  
 context.register(MainConfig4.class);  
 //刷新mail的配置到Environment  
 RefreshConfigUtil.refreshMailPropertySource(context);  
 context.refresh();  
  
 MailService mailService = context.getBean(MailService.class);  
 System.out.println("配置未更新的情况下,输出3次");  
 for (int i = 0; i < 3; i++) { //@1  
 System.out.println(mailService);  
 TimeUnit.MILLISECONDS.sleep(200);  
 }  
  
 System.out.println("模拟3次更新配置效果");  
 for (int i = 0; i < 3; i++) { //@2  
 RefreshConfigUtil.updateDbConfig(context); //@3  
 System.out.println(mailService);  
 TimeUnit.MILLISECONDS.sleep(200);  
 }  
}

@1：循环3次，输出mailService的信息

@2：循环3次，内部先通过@3来模拟更新db中配置信息，然后在输出mailService信息

##### 见证奇迹的时刻，来看效果

配置未更新的情况下,输出3次  
MailService{mailConfig=MailConfig{username='df321543-8ca7-4563-993a-bd64cbf50d53'}}  
MailService{mailConfig=MailConfig{username='df321543-8ca7-4563-993a-bd64cbf50d53'}}  
MailService{mailConfig=MailConfig{username='df321543-8ca7-4563-993a-bd64cbf50d53'}}  
模拟3次更新配置效果  
MailService{mailConfig=MailConfig{username='6bab8cea-9f4f-497d-a23a-92f15d0d6e34'}}  
MailService{mailConfig=MailConfig{username='581bf395-f6b8-4b87-84e6-83d3c7342ca2'}}  
MailService{mailConfig=MailConfig{username='db337f54-20b0-4726-9e55-328530af6999'}}

上面MailService输出了6次，前3次username的值都是一样的，后面3次username的值不一样了，说明修改配置起效了。

#### 小结

动态@Value实现的关键是@Scope中proxyMode参数，值为ScopedProxyMode.DEFAULT，会生成一个代理，通过这个代理来实现@Value动态刷新的效果，这个地方是关键。

有兴趣的可以去看一下springboot中的@RefreshScope注解源码，和我们上面自定义的@RefreshScope类似，实现原理类似的。

##### 

### 总结

本次面试过程中3个问题，我们都搞定了，希望你也已经掌握了，有问题的欢迎给我留言，交流！

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解

## Spring系列第26篇：国际化详解

上次去一个电商公司面试：

面试官：Spring中国际化这块的东西用过么？可以介绍一下么？

我：spring中对国际化支持挺好的，比较简单，只需要按照语言配置几个properties文件，然后主要注册一个国际化的相关的bean，同时需指定一下配置文件的位置，基本上就可以了

面试官：那如果配置文件内容有变化？你们怎么解决的？

我：这块啊，spring国际化这块有个实现类，可以检测到配置文件的变化，就可以解决你这个问题

面试官：那我们是否可以将这些国际化的配置丢到db中去管理呢？

我：这个地方我没有搞过，基本上我们这边都是将国际化的配置文件放在项目中的properties文件中；不过以我对spring的理解，spring扩展方面是非常优秀的，应该是可以这么做的，自己去实现一下spring国际化相关接口就可以了。

面试官：工资期望多少？

我：2万

面试官：恭喜你，下周来上班！

为了方便大家，准备把这块知识细化一下，方便大家面试及使用。

### 本次问题

1. Spring中国际化怎么用？
2. 国际化如何处理资源文件变化的问题？
3. 国际化资源配置放在db中如何实现？

### 先说一下什么是国际化

**简单理解，就是对于不同的语言，做出不同的响应。**

比如页面中有个填写用户信息的表单，有个姓名的输入框

浏览器中可以选择语言

选中文的时候会显示：

姓名：一个输入框

选英文的时候会显示：

Full name：一个输入框

国际化就是做这个事情的，根据不同的语言显示不同的信息。

所以需要支持国际化，得先知道选择的是哪种地区的哪种语言，java中使用java.util.Locale来表示地区语言这个对象，内部包含了国家和语言的信息。

Locale中有个比较常用的构造方法

public Locale(String language, String country) {  
 this(language, country, "");  
}

2个参数：

language：语言

country：国家

语言和国家这两个参数的值不是乱写的，国际上有统一的标准：

比如language的值：zh表示中文，en表示英语，而中文可能很多地区在用，比如大陆地区可以用：CN，新加坡用：SG；英语也是有很多国家用的，GB表示英国，CA表示加拿大

国家语言简写格式：language-country，如：zh-CN（中文【中国】），zh-SG（中文【新加坡】），en-GB（英语【英国】），

en-CA（英语【加拿大】）。

还有很多，这里就不细说了，国家语言编码给大家提供一个表格：http://www.itsoku.com/article/282

**Locale类中已经创建好了很多常用的Locale对象，直接可以拿过来用**，随便列几个看一下：

static public final Locale SIMPLIFIED\_CHINESE = createConstant("zh", "CN"); //zh\_CN  
static public final Locale UK = createConstant("en", "GB"); //en\_GB  
static public final Locale US = createConstant("en", "US"); //en\_US  
static public final Locale CANADA = createConstant("en", "CA"); //en\_CA

再回头看前面的问题：页面中显示姓名对应的标签，需要我们根据一个key及Locale信息来获取对应的国际化信息，spring中提供了这部分的实现，下面我们来看详情。

### Spring中国际化怎么用？

#### MessageSource接口

spring中国际化是通过MessageSource这个接口来支持的

org.springframework.context.MessageSource

内部有3个常用的方法用来获取国际化信息，来看一下

public interface MessageSource {  
  
 /\*\*  
 \* 获取国际化信息  
 \* @param code 表示国际化资源中的属性名；  
 \* @param args用于传递格式化串占位符所用的运行期参数；  
 \* @param defaultMessage 当在资源找不到对应属性名时，返回defaultMessage参数所指定的默认信息；  
 \* @param locale 表示本地化对象  
 \*/  
 @Nullable  
 String getMessage(String code, @Nullable Object[] args, @Nullable String defaultMessage, Locale locale);  
  
 /\*\*  
 \* 与上面的方法类似，只不过在找不到资源中对应的属性名时，直接抛出NoSuchMessageException异常  
 \*/  
 String getMessage(String code, @Nullable Object[] args, Locale locale) throws NoSuchMessageException;  
  
 /\*\*  
 \* @param MessageSourceResolvable 将属性名、参数数组以及默认信息封装起来，它的功能和第一个方法相同  
 \*/  
 String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException;  
  
}

#### 常见3个实现类

##### ResourceBundleMessageSource

这个是基于Java的ResourceBundle基础类实现，允许仅通过资源名加载国际化资源

##### ReloadableResourceBundleMessageSource

这个功能和第一个类的功能类似，多了定时刷新功能，允许在不重启系统的情况下，更新资源的信息

##### StaticMessageSource

它允许通过编程的方式提供国际化信息，一会我们可以通过这个来实现db中存储国际化信息的功能。

### Spring中使用国际化的3个步骤

通常我们使用spring的时候，都会使用带有ApplicationContext字样的spring容器，这些容器一般是继承了AbstractApplicationContext接口，而这个接口实现了上面说的国际化接口MessageSource，所以通常我们用到的ApplicationContext类型的容器都自带了国际化的功能。

通常我们在ApplicationContext类型的容器中使用国际化3个步骤

**步骤一：创建国际化文件**

**步骤二：向容器中注册一个MessageSource类型的bean，bean名称必须为：messageSource**

**步骤三：调用AbstractApplicationContext中的getMessage来获取国际化信息，其内部将交给第二步中注册的messageSource名称的bean进行处理**

#### 来个案例感受一下

##### 创建国际化文件

**国际化文件命名格式：名称\_语言\_地区.properties**

我们来3个文件，文件都放在下面这个目录中

com/javacode2018/lesson002/demo19/

###### message.properties

name=您的姓名  
personal\_introduction=默认个人介绍:{0},{1}

这个文件名称没有指定Local信息，当系统找不到的时候会使用这个默认的

###### message*cn*ZH.properties：中文【中国】

name=姓名  
personal\_introduction=个人介绍:{0},{1},{0}

###### message*en*GB.properties：英文【英国】

name=Full name  
personal\_introduction=personal\_introduction:{0},{1},{0}

##### spring中注册国际化的bean

注意必须是MessageSource类型的，bean名称必须为messageSource，此处我们就使用ResourceBundleMessageSource这个类

package com.javacode2018.lesson002.test19.demo1;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.support.ResourceBundleMessageSource;  
  
@Configuration  
public class MainConfig1 {  
 @Bean  
 public ResourceBundleMessageSource messageSource() {  
 ResourceBundleMessageSource result = new ResourceBundleMessageSource();  
 //可以指定国际化化配置文件的位置，格式：路径/文件名称,注意不包含【语言\_国家.properties】含这部分  
 result.setBasenames("com/javacode2018/lesson002/demo19/message"); //@1  
 return result;  
 }  
}

@1：这个地方的写法需要注意，可以指定国际化化配置文件的位置，格式：路径/文件名称,注意不包含**【语言\_国家.properties】**含这部分

##### 来个测试用例

package com.javacode2018.lesson002.test19;  
  
import com.javacode2018.lesson002.test19.demo1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.Locale;  
  
public class MessageSourceTest {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 //未指定Locale，此时系统会取默认的locale对象，本地默认的值中文【中国】，即：zh\_CN  
 System.out.println(context.getMessage("name", null, null));  
 System.out.println(context.getMessage("name", null, Locale.CHINA)); //CHINA对应：zh\_CN  
 System.out.println(context.getMessage("name", null, Locale.UK)); //UK对应en\_GB  
 }  
}

##### 运行输出

您的姓名  
您的姓名  
Full name

第一行未指定Locale，此时系统会取默认的locale对象，本地默认的值中文【中国】，即：zh*CN，所以会获取到`message*zh\_CN.properties`中的内容。

后面2行，都指定了Locale对象，找到对应的国际化文件，取值。

##### 动态参数使用

注意配置文件中的personal\_introduction，个人介绍，比较特别，包含了{0},{1},{0}这样一部分内容，这个就是动态参数，调用getMessage的时候，通过第二个参数传递过去，来看一下用法：

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 //未指定Locale，此时系统会取默认的，本地电脑默认的值中文【中国】，即：zh\_CN  
 System.out.println(context.getMessage("personal\_introduction", new String[]{"spring高手", "java高手"}, Locale.CHINA)); //CHINA对应：zh\_CN  
 System.out.println(context.getMessage("personal\_introduction", new String[]{"spring", "java"}, Locale.UK)); //UK对应en\_GB  
}

##### 运行输出

默认个人介绍:spring高手,java高手  
personal\_introduction:spring,java,spring

### 监控国际化文件的变化

用ReloadableResourceBundleMessageSource这个类，功能和上面案例中的ResourceBundleMessageSource类似，不过多了个可以监控国际化资源文件变化的功能，有个方法用来设置缓存时间：

public void setCacheMillis(long cacheMillis)

-1：表示永远缓存

0：每次获取国际化信息的时候，都会重新读取国际化文件

大于0：上次读取配置文件的时间距离当前时间超过了这个时间，重新读取国际化文件

还有个按秒设置缓存时间的方法setCacheSeconds，和setCacheMillis类似

下面我们来案例

package com.javacode2018.lesson002.test19.demo2;  
  
import org.springframework.context.MessageSource;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.support.ReloadableResourceBundleMessageSource;  
  
@Configuration  
public class MainConfig2 {  
 @Bean  
 public MessageSource messageSource() {  
 ReloadableResourceBundleMessageSource result = new ReloadableResourceBundleMessageSource();  
 result.setBasenames("com/javacode2018/lesson002/demo19/message");  
 //设置缓存时间1000毫秒  
 result.setCacheMillis(1000);  
 return result;  
 }  
}

message*zh*CN.properties中新增一行内容

address=上海

对应的测试用例

@Test  
public void test3() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig2.class);  
 context.refresh();  
 //输出2次  
 for (int i = 0; i < 2; i++) {  
 System.out.println(context.getMessage("address", null, Locale.CHINA));  
 TimeUnit.SECONDS.sleep(5);  
 }  
}

上面有个循环，当第一次输出之后，修改一下message\_zh\_CN.properties中的address为上海松江，最后运行结果如下：

上海  
上海松江

**使用注意：线上环境，缓存时间最好设置大一点，性能会好一些。**

### 国际化信息存在db中

上面我们介绍了一个类：StaticMessageSource，这个类它允许通过编程的方式提供国际化信息，我们通过这个类来实现从db中获取国际化信息的功能。

这个类中有2个方法比较重要：

public void addMessage(String code, Locale locale, String msg);  
public void addMessages(Map<String, String> messages, Locale locale);

通过这两个方法来添加国际化配置信息。

下面来看案例

自定义一个StaticMessageSource类

package com.javacode2018.lesson002.test19.demo3;  
  
import org.springframework.beans.factory.InitializingBean;  
import org.springframework.context.support.StaticMessageSource;  
  
import java.util.Locale;  
  
public class MessageSourceFromDb extends StaticMessageSource implements InitializingBean {  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 //此处我们在当前bean初始化之后，模拟从db中获取国际化信息，然后调用addMessage来配置国际化信息  
 this.addMessage("desc", Locale.CHINA, "我是从db来的信息");  
 this.addMessage("desc", Locale.UK, "MessageSource From Db");  
 }  
}

上面的类实现了spring的InitializingBean接口，重写了接口中干掉afterPropertiesSet方法，这个方法会在当前bean初始化之后调用，在这个方法中模拟从db中获取国际化信息，然后调用addMessage来配置国际化信息

来个spring配置类，将MessageSourceFromDb注册到spring容器

package com.javacode2018.lesson002.test19.demo3;  
  
import org.springframework.context.MessageSource;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class MainConfig3 {  
 @Bean  
 public MessageSource messageSource(){  
 return new MessageSourceFromDb();  
 }  
}

上测试用例

@Test  
public void test4() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig3.class);  
 context.refresh();  
 System.out.println(context.getMessage("desc", null, Locale.CHINA));  
 System.out.println(context.getMessage("desc", null, Locale.UK));  
}

运行输出

我是从db来的信息  
MessageSource From Db

### bean名称为什么必须是messageSource

上面我容器启动的时候会调用refresh方法，过程如下：

org.springframework.context.support.AbstractApplicationContext#refresh  
内部会调用  
org.springframework.context.support.AbstractApplicationContext#initMessageSource  
这个方法用来初始化MessageSource,方法内部会查找当前容器中是否有messageSource名称的bean，如果有就将其作为处理国际化的对象  
如果没有找到，此时会注册一个名称为messageSource的MessageSource

### 自定义bean中使用国际化

自定义的bean如果想使用国际化，比较简单，只需实现下面这个接口，spring容器会自动调用这个方法，将MessageSource注入，然后我们就可以使用MessageSource获取国际化信息了。

public interface MessageSourceAware extends Aware {  
 void setMessageSource(MessageSource messageSource);  
}

### 总结

**本文介绍了国际化的使用，涉及到了java中的Locale类，这个类用来表示语言国家信息，获取国际化信息的时候需要携带这个参数，spring中通过MessageSource接口来支持国际化的功能，有3个常用的实现类需要了解，StaticMessageSource支持硬编码的方式配置国际化信息。**

**如果需要spring支撑国际化，需要注册一个bean名称为messageSource的MessageSource，这个一定要注意。**

到此，上面面试的3个问题，大家都能轻松应对了。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】

## Spring系列第27篇：事件详解

又被面试官刁难了

面试官：Spring中的事件用过么？

我：用过

面试官：可以介绍一下为什么需要使用事件么？

我：使用事件的模式可以对系统进行解耦，事件源发布一个事件，事件监听器可以消费这个事件，而事件源不用关注发布的事件有哪些监听器，这可以可以对系统进行解耦

面试官：Spring事件的实现有几种方式？

我：整体来说2种方式，第一种是通过接口的方式，第二种是在方法上使用注解的方式

面试官：Spring中事件监听器的处理是同步方式还是异步方式？

我：不好意思，没听懂问题

面试官：事件的发布和事件监听器的执行是否在同一个线程中运行？

我：在一个线程中执行，是同步的方式

面试官：是否支持异步方式？

我：支持

面试官：你确定么？

我：嗯。。。，这块没有用过，不过我感觉是可以的，事件监听器中的逻辑一般不是主要业务，可以不再当前线程中执行。

面试官：那spring中事件监听器支持自定义顺序么？

我：这个不知道

面试官：行吧，今天的面试到此为止吧，回去之后巩固下自己的技术，多看看源码

我：好的。。。。此时脑子里已经是浆糊了。

回去之后赶紧将spring事件这块源码翻出来又好好研究了几遍。

### 面试过程中的主要问题

1. 为什么需要使用事件这种模式？
2. spring中实现事件有几种方式？
3. spring中事件监听器消费事件是否支持异步模式？
4. spring中事件监听器消费事件是否支持自定义顺序？

下面我们就一个个来介绍。

### 为什么需要使用时间这种模式？

先来看一个业务场景：

产品经理：路人，这两天你帮我实现一个注册的功能

我：注册功能比较简单，将用户信息入库就可以了，伪代码如下：

public void registerUser(UserModel user){  
 //插入用户信息到db，完成注册  
 this.insertUser(user);  
}

过了几天，产品经理：路人，注册成功之后，给用户发送一封注册成功的邮件

我：修改了一下上面注册代码，如下：

public void registerUser(UserModel user){  
 //插入用户信息到db，完成注册  
 this.insertUser(user);  
 //发送邮件  
 this.sendEmailToUser(user);  
}

由于修改了注册接口，所以所有调用这个方法的地方都需要重新测试一遍，让测试的兄弟们帮忙跑了一遍。

又过了几天，产品经理：路人，注册成功之后，给用户发一下优惠券

我：好的，又调整了一下代码

public void registerUser(UserModel user){  
 //插入用户信息到db，完成注册  
 this.insertUser(user);  
 //发送邮件  
 this.sendEmailToUser(user);  
 //发送优惠券  
 this.sendCouponToUser(user);  
}

我：测试的兄弟们，辛苦一下大家，注册接口又修改了，帮忙再过一遍。

过了一段时间，公司效益太好，产品经理：路人，注册的时候，取消给用户发送优惠券的功能。

我：又跑去调整了一下上面代码，将发送优惠券的功能干掉了，如下

public void registerUser(UserModel user){  
 //插入用户信息到db，完成注册  
 this.insertUser(user);  
 //发送邮件  
 this.sendEmailToUser(user);  
}

由于调整了代码，而注册功能又属于核心业务，所以需要让测试再次帮忙过一遍，又要麻烦测试来一遍了。

突然有一天，产品经理：路人，注册接口怎么这么慢啊，并且还经常失败？你这让公司要损失多少用户啊

我：赶紧跑去查看了一下运行日志，发现注册的时候给用户发送邮件不稳定，依赖于第三方邮件服务器，耗时比较长，并且容易失败。

跑去给产品经理说：由于邮件服务器不稳定的原因，导致注册不稳定。

产品经理：邮件你可以不发，但是你得确保注册功能必须可以用啊。

我想了想，将上面代码改成了下面这样，发送邮件放在了子线程中执行：

public void registerUser(UserModel user){  
 //插入用户信息到db，完成注册  
 this.insertUser(user);  
 //发送邮件，放在子线程中执行，邮件的发送结果对注册逻辑不会有干扰作用  
 new Thread(()->{  
 this.sendEmailToUser(user);  
 }).start();  
}

又过了几天，产品经理又跑来了说：路人，最近效益不好，需要刺激用户消费，注册的时候继续发送优惠券。

我：倒，这是玩我么，反反复复让我调整注册的代码，让我改还好，让测试也反反复复来回搞，这是要玩死我们啊。

花了点时间，好好复盘整理了一下：发现问题不在于产品经理，从业务上来看，产品提的这些需求都是需求合理的，而结果代码反复调整、测试反复测试，以及一些次要的功能导致注册接口不稳定，这些问题归根到底，主要还是我的设计不合理导致的，将注册功能中的一些次要的功能耦合到注册的方法中了，并且这些功能可能会经常调整，导致了注册接口的不稳定性。

其实上面代码可以这么做：

找3个人：注册器、路人A、路人B。

注册器：负责将用户信息落库，落库成功之后，喊一声：用户XXX注册成功了。

路人A和路人B，竖起耳朵，当听到有人喊：XXX注册成功 的声音之后，立即行动做出下面反应：

路人A：负责给XXX发送一封注册邮件

路人B：负责给XXX发送优惠券

我们来看一下：

注册器只负责将用户信息落库，及广播一条用户注册成功的消息。

A和B相当于一个监听者，只负责监听用户注册成功的消息，当听到有这个消息产生的时候，A和B就去做自己的事情。

这里面注册器是感知不到A/B存在的，A和B也不用感知注册器的存在，A/B只用关注是否有人广播：XXX注册成功了的消息，当AB听到有人广播注册成功的消息，他们才做出反应，其他时间闲着休息。

这种方式就非常好：

当不想给用户发送优惠券的时候，只需要将B去掉就行了，此时基本上也不用测试，注册一下B的代码就行了。

若注册成功之后需要更多业务，比如还需要给用户增加积分，只需新增一个监听者C，监听到注册成功消息后，负责给用户添加积分，此时根本不用去调整注册的代码，开发者和测试人员只需要确保监听者C中的正确性就可以了。

上面这种模式就是事件模式。

### 事件模式中的几个概念

**事件源**：事件的触发者，比如上面的注册器就是事件源。

**事件**：描述发生了什么事情的对象，比如上面的：xxx注册成功的事件

**事件监听器**：监听到事件发生的时候，做一些处理，比如上面的：路人A、路人B

### 下面我们使用事件模式实现用户注册的业务

我们先来定义和事件相关的几个类。

#### 事件对象

表示所有事件的父类，内部有个source字段，表示事件源；我们自定义的事件需要继承这个类。

package com.javacode2018.lesson003.demo1.test0.event;  
  
/\*\*  
 \* 事件对象  
 \*/  
public abstract class AbstractEvent {  
  
 //事件源  
 protected Object source;  
  
 public AbstractEvent(Object source) {  
 this.source = source;  
 }  
  
 public Object getSource() {  
 return source;  
 }  
  
 public void setSource(Object source) {  
 this.source = source;  
 }  
}

#### 事件监听器

我们使用一个接口来表示事件监听器，是个泛型接口，后面的类型E表示当前监听器需要监听的事件类型，此接口中只有一个方法，用来实现处理事件的业务；其定义的监听器需要实现这个接口。

package com.javacode2018.lesson003.demo1.test0.event;  
  
/\*\*  
 \* 事件监听器  
 \*  
 \* @param <E> 当前监听器感兴趣的事件类型  
 \*/  
public interface EventListener<E extends AbstractEvent> {  
 /\*\*  
 \* 此方法负责处理事件  
 \*  
 \* @param event 要响应的事件对象  
 \*/  
 void onEvent(E event);  
}

#### 事件广播器

* 负责事件监听器的管理（注册监听器&移除监听器，将事件和监听器关联起来）
* 负责事件的广播（将事件广播给所有的监听器，对该事件感兴趣的监听器会处理该事件）

package com.javacode2018.lesson003.demo1.test0.event;  
  
/\*\*  
 \* 事件广播器：  
 \* 1.负责事件监听器的管理（注册监听器&移除监听器，将事件和监听器关联起来）  
 \* 2.负责事件的广播（将事件广播给所有的监听器，对该事件感兴趣的监听器会处理该事件）  
 \*/  
public interface EventMulticaster {  
  
 /\*\*  
 \* 广播事件给所有的监听器，对该事件感兴趣的监听器会处理该事件  
 \*  
 \* @param event  
 \*/  
 void multicastEvent(AbstractEvent event);  
  
 /\*\*  
 \* 添加一个事件监听器（监听器中包含了监听器中能够处理的事件）  
 \*  
 \* @param listener 需要添加监听器  
 \*/  
 void addEventListener(EventListener<?> listener);  
  
  
 /\*\*  
 \* 将事件监听器移除  
 \*  
 \* @param listener 需要移除的监听器  
 \*/  
 void removeEventListener(EventListener<?> listener);  
}

#### 事件广播默认实现

package com.javacode2018.lesson003.demo1.test0.event;  
  
import java.lang.reflect.ParameterizedType;  
import java.lang.reflect.Type;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
  
/\*\*  
 \* 事件广播器简单实现  
 \*/  
public class SimpleEventMulticaster implements EventMulticaster {  
  
 private Map<Class<?>, List<EventListener>> eventObjectEventListenerMap = new ConcurrentHashMap<>();  
  
 @Override  
 public void multicastEvent(AbstractEvent event) {  
 List<EventListener> eventListeners = this.eventObjectEventListenerMap.get(event.getClass());  
 if (eventListeners != null) {  
 for (EventListener eventListener : eventListeners) {  
 eventListener.onEvent(event);  
 }  
 }  
 }  
  
 @Override  
 public void addEventListener(EventListener<?> listener) {  
 Class<?> eventType = this.getEventType(listener);  
 List<EventListener> eventListeners = this.eventObjectEventListenerMap.get(eventType);  
 if (eventListeners == null) {  
 eventListeners = new ArrayList<>();  
 this.eventObjectEventListenerMap.put(eventType, eventListeners);  
 }  
 eventListeners.add(listener);  
 }  
  
 @Override  
 public void removeEventListener(EventListener<?> listener) {  
 Class<?> eventType = this.getEventType(listener);  
 List<EventListener> eventListeners = this.eventObjectEventListenerMap.get(eventType);  
 if (eventListeners != null) {  
 eventListeners.remove(listener);  
 }  
 }  
  
 /\*\*  
 \* 获取事件监听器需要监听的事件类型  
 \*  
 \* @param listener  
 \* @return  
 \*/  
 protected Class<?> getEventType(EventListener listener) {  
 ParameterizedType parameterizedType = (ParameterizedType) listener.getClass().getGenericInterfaces()[0];  
 Type eventType = parameterizedType.getActualTypeArguments()[0];  
 return (Class<?>) eventType;  
 }  
  
}

**上面3个类支撑了整个时间模型，下面我们使用上面三个类来实现注册的功能，目标是：高内聚低耦合，让注册逻辑方便扩展。**

#### 自定义用户注册成功事件类

继承了AbstractEvent类

package com.javacode2018.lesson003.demo1.test0.userregister;  
  
import com.javacode2018.lesson003.demo1.test0.event.AbstractEvent;  
  
/\*\*  
 \* 用户注册成功事件  
 \*/  
public class UserRegisterSuccessEvent extends AbstractEvent {  
 //用户名  
 private String userName;  
  
 /\*\*  
 \* 创建用户注册成功事件对象  
 \*  
 \* @param source 事件源  
 \* @param userName 当前注册的用户名  
 \*/  
 public UserRegisterSuccessEvent(Object source, String userName) {  
 super(source);  
 this.userName = userName;  
 }  
  
 public String getUserName() {  
 return userName;  
 }  
  
 public void setUserName(String userName) {  
 this.userName = userName;  
 }  
}

#### 用户注册服务

负责实现用户注册逻辑

package com.javacode2018.lesson003.demo1.test0.userregister;  
  
import com.javacode2018.lesson003.demo1.test0.event.EventMulticaster;  
  
/\*\*  
 \* 用户注册服务  
 \*/  
public class UserRegisterService {  
 //事件发布者  
 private EventMulticaster eventMulticaster; //@0  
  
 /\*\*  
 \* 注册用户  
 \*  
 \* @param userName 用户名  
 \*/  
 public void registerUser(String userName) { //@1  
 //用户注册(将用户信息入库等操作)  
 System.out.println(String.format("用户【%s】注册成功", userName)); //@2  
 //广播事件  
 this.eventMulticaster.multicastEvent(new UserRegisterSuccessEvent(this, userName)); //@3  
 }  
  
 public EventMulticaster getEventMulticaster() {  
 return eventMulticaster;  
 }  
  
 public void setEventMulticaster(EventMulticaster eventMulticaster) {  
 this.eventMulticaster = eventMulticaster;  
 }  
}

@0：事件发布者

@1：registerUser这个方法负责用户注册，内部主要做了2个事情

@2：模拟将用户信息落库

@3：使用事件发布者eventPublisher发布用户注册成功的消息:

#### 下面我们使用spring来将上面的对象组装起来

package com.javacode2018.lesson003.demo1.test0.userregister;  
  
import com.javacode2018.lesson003.demo1.test0.event.EventListener;  
import com.javacode2018.lesson003.demo1.test0.event.EventMulticaster;  
import com.javacode2018.lesson003.demo1.test0.event.SimpleEventMulticaster;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
import java.util.List;  
  
@Configuration  
@ComponentScan  
public class MainConfig0 {  
  
 /\*\*  
 \* 注册一个bean：事件发布者  
 \*  
 \* @param eventListeners  
 \* @return  
 \*/  
 @Bean  
 @Autowired(required = false)  
 public EventMulticaster eventMulticaster(List<EventListener> eventListeners) { //@1  
 EventMulticaster eventPublisher = new SimpleEventMulticaster();  
 if (eventListeners != null) {  
 eventListeners.forEach(eventPublisher::addEventListener);  
 }  
 return eventPublisher;  
 }  
  
 /\*\*  
 \* 注册一个bean：用户注册服务  
 \*  
 \* @param eventMulticaster  
 \* @return  
 \*/  
 @Bean  
 public UserRegisterService userRegisterService(EventMulticaster eventMulticaster) { //@2  
 UserRegisterService userRegisterService = new UserRegisterService();  
 userRegisterService.setEventMulticaster(eventMulticaster);  
 return userRegisterService;  
 }  
}

上面有2个方法，负责向spring容器中注册2个bean。

@1：向spring容器中注册了一个bean：事件发布者，方法传入了EventListener类型的List，这个地方会将容器中所有的事件监听器注入进来，丢到EventMulticaster中。

@2：向spring容器中注册了一个bean：用户注册服务

#### 来个测试用例模拟用户注册

package com.javacode2018.lesson003.demo1;  
  
import com.javacode2018.lesson003.demo1.test0.userregister.MainConfig0;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class EventTest {  
  
 @Test  
 public void test0() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig0.class);  
 //获取用户注册服务  
 com.javacode2018.lesson003.demo1.test0.userregister.UserRegisterService userRegisterService =  
 context.getBean(com.javacode2018.lesson003.demo1.test0.userregister.UserRegisterService.class);  
 //模拟用户注册  
 userRegisterService.registerUser("马士兵老师");  
 }  
  
}

#### 运行输出

用户【马士兵老师】注册成功

#### 添加注册成功发送邮件功能

下面添加一个注册成功发送邮件的功能，只需要自定义一个监听用户注册成功事件的监听器就可以了，其他代码不需要任何改动，如下

package com.javacode2018.lesson003.demo1.test0.userregister;  
  
  
import com.javacode2018.lesson003.demo1.test0.event.EventListener;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 用户注册成功事件监听器->负责给用户发送邮件  
 \*/  
@Component  
public class SendEmailOnUserRegisterSuccessListener implements EventListener<UserRegisterSuccessEvent> {  
 @Override  
 public void onEvent(UserRegisterSuccessEvent event) {  
 System.out.println(  
 String.format("给用户【%s】发送注册成功邮件!", event.getUserName()));  
 }  
}

上面这个类使用了@Component，会被自动扫描注册到spring容器。

#### 再次运行测试用例输出

用户【马士兵老师】注册成功  
给用户【马士兵老师】发送注册成功邮件!

#### 小结

上面将注册的主要逻辑（用户信息落库）和次要的业务逻辑（发送邮件）通过事件的方式解耦了。次要的业务做成了可插拔的方式，比如不想发送邮件了，只需要将邮件监听器上面的@Component注释就可以了，非常方便扩展。

上面用到的和事件相关的几个类，都是我们自己实现的，其实这些功能在spring中已经帮我们实现好了，用起来更容易一些，下面带大家来体验一下。

### Spring中实现事件模式

#### 事件相关的几个类

Spring中事件相关的几个类需要先了解一下，下面来个表格，将spring中事件相关的类和我们上面自定义的类做个对比，方便大家理解

|  |  |  |
| --- | --- | --- |
| spring中事件类 | 我们自定义的事件类 | 作用 |
| org.springframework.context.ApplicationEvent | AbstractEvent | 表示事件对象的父类 |
| org.springframework.context.ApplicationListener | EventListener | 事件监听器接口 |
| org.springframework.context.event.ApplicationEventMulticaster | EventPublisher | 事件广播器 |
| org.springframework.context.event.SimpleApplicationEventMulticaster | SimpleEventMulticaster | 事件广播器的简单实现 |

这些类和我们自定义的类中代码有点类似，有兴趣的可以去看一下源码，这里就不列出来了。

#### 硬编码的方式使用spring事件3步骤

##### 步骤1：定义事件

自定义事件，需要继承ApplicationEvent类，

##### 步骤2：定义监听器

自定义事件监听器，需要实现ApplicationListener接口，这个接口有个方法onApplicationEvent需要实现，用来处理感兴趣的事件。

@FunctionalInterface  
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {  
  
 /\*\*  
 \* Handle an application event.  
 \* @param event the event to respond to  
 \*/  
 void onApplicationEvent(E event);  
  
}

##### 步骤3：创建事件广播器

创建事件广播器ApplicationEventMulticaster，这是个接口，你可以自己实现这个接口，也可以直接使用系统给我们提供的SimpleApplicationEventMulticaster，如下：

ApplicationEventMulticaster applicationEventMulticaster = new SimpleApplicationEventMulticaster();

##### 步骤4：向广播器中注册事件监听器

将事件监听器注册到广播器ApplicationEventMulticaster中，如：

ApplicationEventMulticaster applicationEventMulticaster = new SimpleApplicationEventMulticaster();  
applicationEventMulticaster.addApplicationListener(new SendEmailOnOrderCreateListener());

##### 步骤5：通过广播器发布事件

广播事件，调用ApplicationEventMulticaster#multicastEvent方法广播事件，此时广播器中对这个事件感兴趣的监听器会处理这个事件。

applicationEventMulticaster.multicastEvent(new OrderCreateEvent(applicationEventMulticaster, 1L));

下面我们来个案例将这5个步骤串起来感受一下。

#### 案例

实现功能：电商中订单创建成功之后，给下单人发送一封邮件，发送邮件的功能放在监听器中实现。

下面上代码

##### 来个事件类：订单创建成功事件

package com.javacode2018.lesson003.demo1.test1;  
  
import org.springframework.context.ApplicationEvent;  
  
/\*\*  
 \* 订单创建事件  
 \*/  
public class OrderCreateEvent extends ApplicationEvent {  
 //订单id  
 private Long orderId;  
  
 /\*\*  
 \* @param source 事件源  
 \* @param orderId 订单id  
 \*/  
 public OrderCreateEvent(Object source, Long orderId) {  
 super(source);  
 this.orderId = orderId;  
 }  
  
 public Long getOrderId() {  
 return orderId;  
 }  
  
 public void setOrderId(Long orderId) {  
 this.orderId = orderId;  
 }  
}

##### 来个监听器：负责监听订单成功事件，发送邮件

package com.javacode2018.lesson003.demo1.test1;  
  
import org.springframework.context.ApplicationListener;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 订单创建成功给用户发送邮件  
 \*/  
@Component  
public class SendEmailOnOrderCreateListener implements ApplicationListener<OrderCreateEvent> {  
 @Override  
 public void onApplicationEvent(OrderCreateEvent event) {  
 System.out.println(String.format("订单【%d】创建成功，给下单人发送邮件通知!", event.getOrderId()));  
 }  
}

##### 测试用例

@Test  
public void test2() throws InterruptedException {  
 //创建事件广播器  
 ApplicationEventMulticaster applicationEventMulticaster = new SimpleApplicationEventMulticaster();  
 //注册事件监听器  
 applicationEventMulticaster.addApplicationListener(new SendEmailOnOrderCreateListener());  
 //广播事件订单创建事件  
 applicationEventMulticaster.multicastEvent(new OrderCreateEvent(applicationEventMulticaster, 1L));  
}

##### 运行输出

订单【1】创建成功，给下单人发送邮件通知!

#### ApplicationContext容器中事件的支持

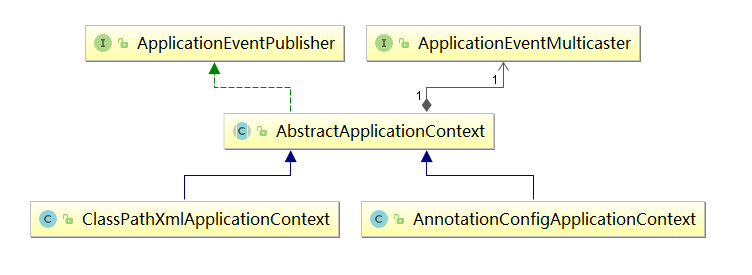
上面演示了spring中事件的使用，那么平时我们使用spring的时候就这么使用？

非也非也，上面只是我给大家演示了一下原理。

通常情况下，我们会使用以ApplicationContext结尾的类作为spring的容器来启动应用，下面2个是比较常见的

AnnotationConfigApplicationContext  
ClassPathXmlApplicationContext

来看一个类图



对这个图我们来解释一下：

1.AnnotationConfigApplicationContext和ClassPathXmlApplicationContext都继承了AbstractApplicationContext  
2.AbstractApplicationContext实现了ApplicationEventPublisher接口  
3.AbstractApplicationContext内部有个ApplicationEventMulticaster类型的字段

上面第三条，说明了AbstractApplicationContext内部已经集成了事件广播器ApplicationEventMulticaster，说明AbstractApplicationContext内部是具体事件相关功能的，这些功能是通过其内部的ApplicationEventMulticaster来实现的，也就是说将事件的功能委托给了内部的ApplicationEventMulticaster来实现。

#### ApplicationEventPublisher接口

上面类图中多了一个新的接口ApplicationEventPublisher，来看一下源码

@FunctionalInterface  
public interface ApplicationEventPublisher {  
  
 default void publishEvent(ApplicationEvent event) {  
 publishEvent((Object) event);  
 }  
  
 void publishEvent(Object event);  
  
}

这个接口用来发布事件的，内部定义2个方法都是用来发布事件的。

spring中不是有个ApplicationEventMulticaster接口么，此处怎么又来了一个发布事件的接口？

这个接口的实现类中，比如AnnotationConfigApplicationContext内部将这2个方法委托给ApplicationEventMulticaster#multicastEvent进行处理了。

所以调用AbstractApplicationContext中的publishEvent方法，也实现广播事件的效果，不过使用AbstractApplicationContext也只能通过调用publishEvent方法来广播事件。

#### 获取ApplicationEventPublisher对象

如果我们想在普通的bean中获取ApplicationEventPublisher对象，需要实现ApplicationEventPublisherAware接口

public interface ApplicationEventPublisherAware extends Aware {  
 void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher);  
}

spring容器会自动通过上面的setApplicationEventPublisher方法将ApplicationEventPublisher注入进来，此时我们就可以使用这个来发布事件了。

#### Spring为了简化事件的使用，提供了2种使用方式

1. **面相接口的方式**
2. **面相@EventListener注解的方式**

### 面相接口的方式

#### 案例

实现用户注册成功后发布事件，然后在监听器中发送邮件的功能。

##### 用户注册事件

需要继承ApplicationEvent

package com.javacode2018.lesson003.demo1.test2;  
  
import org.springframework.context.ApplicationEvent;  
  
/\*\*  
 \* 用户注册事件  
 \*/  
public class UserRegisterEvent extends ApplicationEvent {  
 //用户名  
 private String userName;  
  
 public UserRegisterEvent(Object source, String userName) {  
 super(source);  
 this.userName = userName;  
 }  
  
 public String getUserName() {  
 return userName;  
 }  
}

##### 发送邮件监听器

需实现ApplicationListener接口

package com.javacode2018.lesson003.demo1.test2;  
  
import org.springframework.context.ApplicationListener;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 用户注册成功发送邮件  
 \*/  
@Component  
public class SendEmailListener implements ApplicationListener<UserRegisterEvent> {  
  
 @Override  
 public void onApplicationEvent(UserRegisterEvent event) {  
 System.out.println(String.format("给用户【%s】发送注册成功邮件!", event.getUserName()));  
  
 }  
}

##### 用户注册服务

内部提供用户注册的功能，并发布用户注册事件

package com.javacode2018.lesson003.demo1.test2;  
  
  
import org.springframework.context.ApplicationEventPublisher;  
import org.springframework.context.ApplicationEventPublisherAware;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 用户注册服务  
 \*/  
@Component  
public class UserRegisterService implements ApplicationEventPublisherAware {  
  
 private ApplicationEventPublisher applicationEventPublisher;  
  
 /\*\*  
 \* 负责用户注册及发布事件的功能  
 \*  
 \* @param userName 用户名  
 \*/  
 public void registerUser(String userName) {  
 //用户注册(将用户信息入库等操作)  
 System.out.println(String.format("用户【%s】注册成功", userName));  
 //发布注册成功事件  
 this.applicationEventPublisher.publishEvent(new UserRegisterEvent(this, userName));  
 }  
  
 @Override  
 public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) { //@1  
 this.applicationEventPublisher = applicationEventPublisher;  
 }  
}

注意上面实现了ApplicationEventPublisherAware接口，spring容器会通过@1将ApplicationEventPublisher注入进来，然后我们就可以使用这个来发布事件了。

##### 来个spring配置类

package com.javacode2018.lesson003.demo1.test2;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig2 {  
}

##### 上测试用例

@Test  
public void test2() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig2.class);  
 context.refresh();  
 //获取用户注册服务  
 com.javacode2018.lesson003.demo1.test2.UserRegisterService userRegisterService =  
 context.getBean(com.javacode2018.lesson003.demo1.test2.UserRegisterService.class);  
 //模拟用户注册  
 userRegisterService.registerUser("马士兵老师");  
}

##### 运行输出

用户【马士兵老师】注册成功  
给用户【马士兵老师】发送注册成功邮件!

#### 原理

spring容器在创建bean的过程中，会判断bean是否为ApplicationListener类型，进而会将其作为监听器注册到AbstractApplicationContext#applicationEventMulticaster中，这块的源码在下面这个方法中，有兴趣的可以看一下

org.springframework.context.support.ApplicationListenerDetector#postProcessAfterInitialization

#### 小结

从上面这个案例中可以看出，事件类、监听器类都是通过基于spring中的事件相关的一些接口来实现事件的功能，这种方式我们就称作面相接口的方式。

### 面相@EventListener注解方式

#### 用法

上面是通过接口的方式创建一个监听器，spring还提供了通过@EventListener注解的方式来创建一个监听器，直接将这个注解标注在一个bean的方法上，那么这个方法就可以用来处理感兴趣的事件，使用更简单，如下，方法参数类型为事件的类型：

@Component  
public class UserRegisterListener {  
 @EventListener  
 public void sendMail(UserRegisterEvent event) {  
 System.out.println(String.format("给用户【%s】发送注册成功邮件!", event.getUserName()));  
 }  
}

#### 案例

注册成功之后：来2个监听器：一个负责发送邮件、一个负责发送优惠券。

其他代码都不上了，和上面案例中的一样，主要看监听器的代码，如下：

package com.javacode2018.lesson003.demo1.test3;  
  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 用户注册监听器  
 \*/  
@Component  
public class UserRegisterListener {  
 @EventListener  
 public void sendMail(UserRegisterEvent event) {  
 System.out.println(String.format("给用户【%s】发送注册成功邮件!", event.getUserName()));  
 }  
  
 @EventListener  
 public void sendCompon(UserRegisterEvent event) {  
 System.out.println(String.format("给用户【%s】发送优惠券!", event.getUserName()));  
 }  
}

这块案例代码

com.javacode2018.lesson003.demo1.EventTest#test3

运行结果

用户【马士兵老师】注册成功  
给用户【马士兵老师】发送优惠券!  
给用户【马士兵老师】发送注册成功邮件!

#### 原理

spring中处理@EventListener注解源码位于下面的方法中

org.springframework.context.event.EventListenerMethodProcessor#afterSingletonsInstantiated

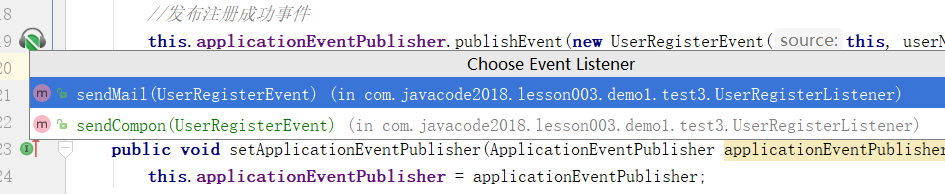
EventListenerMethodProcessor实现了SmartInitializingSingleton接口，SmartInitializingSingleton接口中的afterSingletonsInstantiated方法会在所有单例的bean创建完成之后被spring容器调用，这块的内容可以去看一下：[Bean生命周期详解](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934322&idx=1&sn=647edffeedeb8978c18ad403b1f3d8d7&chksm=88621f8cbf15969af1c5396903dcce312c1f316add1af325327d287e90be49bbeda52bc1e736&token=718443976&lang=zh_CN" \l "rd)

#### idea对注解的方式支持比较好

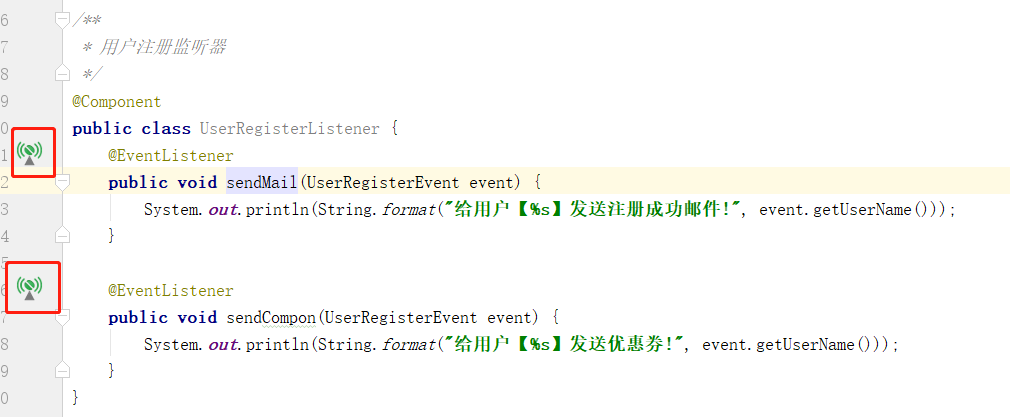
注解的方式实现监听器，idea对这块支持比较好，时间发布的地方会显示一个耳机，点击这个耳机的时候，spring会帮我们列出这个事件有哪些监听器



点击耳机列出了2个监听器，可以快速定位到监听器，如下



同样监听器的地方也有一个广播的图标，如下图



点击上面这个广播的图标，可以快速导航到事件发布的地方，相当方便。

### 监听器支持排序功能

如果某个事件有多个监听器，默认情况下，监听器执行顺序是无序的，不过我们可以为监听器指定顺序。

#### 通过接口实现监听器的情况

如果自定义的监听器是通过ApplicationListener接口实现的，那么指定监听器的顺序有三种方式

##### 方式1：实现org.springframework.core.Ordered接口

需要实现一个getOrder方法，返回顺序值，值越小，顺序越高

int getOrder();

##### 方式2：实现org.springframework.core.PriorityOrdered接口

PriorityOrdered接口继承了方式一中的Ordered接口，所以如果你实现PriorityOrdered接口，也需要实现getOrder方法。

##### 方式3：类上使用@org.springframework.core.annotation.Order注解

看一下这个注解的源码

@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})  
@Documented  
public @interface Order {  
  
 int value() default Ordered.LOWEST\_PRECEDENCE;  
  
}

value属性用来指定顺序

##### 这几种方式排序规则

PriorityOrdered#getOrder ASC,Ordered或@Order ASC

#### 通过@EventListener实现事件监听器的情况

可以在标注@EventListener的方法上面使用@Order(顺序值)注解来标注顺序，如：

@EventListener  
@Order(1)  
public void sendMail(com.javacode2018.lesson003.demo1.test3.UserRegisterEvent event) {  
 System.out.println(String.format("给用户【%s】发送注册成功邮件!", event.getUserName()));  
}

#### 案例

package com.javacode2018.lesson003.demo1.test4;  
  
import org.springframework.context.event.EventListener;  
import org.springframework.core.annotation.Order;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* 用户注册监听器  
 \*/  
@Component  
public class UserRegisterListener {  
 @EventListener  
 @Order(1)  
 public void sendMail(UserRegisterEvent event) {  
 System.out.println(String.format("【%s】，给用户【%s】发送注册成功邮件!", Thread.currentThread(), event.getUserName()));  
 }  
  
 @EventListener  
 @Order(0)  
 public void sendCompon(UserRegisterEvent event) {  
 System.out.println(String.format("【%s】，给用户【%s】发送优惠券!", Thread.currentThread(), event.getUserName()));  
 }  
}

上面会先发送优惠券、然后再发送邮件。

上面输出中顺便将线程信息也输出了。

对应测试用例

com.javacode2018.lesson003.demo1.EventTest#test4

运行输出

【Thread[main,5,main]】,用户【马士兵老师】注册成功  
【Thread[main,5,main]】，给用户【马士兵老师】发送优惠券!  
【Thread[main,5,main]】，给用户【马士兵老师】发送注册成功邮件!

从输出中可以看出上面程序的执行都在主线程中执行的，说明监听器中的逻辑和注册逻辑在一个线程中执行的，此时如果监听器中的逻辑比较耗时或者失败，直接会导致注册失败，通常我们将一些非主要逻辑可以放在监听器中执行，至于这些非主要逻辑成功或者失败，最好不要对主要的逻辑产生影响，所以我们最好能将监听器的运行和主业务隔离开，放在不同的线程中执行，主业务不用关注监听器的结果，spring中支持这种功能，下面继续看。

### 监听器异步模式

#### 先来看看到底如何实现？

监听器最终是通过ApplicationEventMulticaster内部的实现来调用的，所以我们关注的重点就是这个类，这个类默认有个实现类SimpleApplicationEventMulticaster，这个类是支持监听器异步调用的，里面有个字段：

private Executor taskExecutor;

高并发比较熟悉的朋友对Executor这个接口是比较熟悉的，可以用来异步执行一些任务。

我们常用的线程池类java.util.concurrent.ThreadPoolExecutor就实现了Executor接口。

再来看一下SimpleApplicationEventMulticaster中事件监听器的调用，最终会执行下面这个方法

@Override  
public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) {  
 ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));  
 Executor executor = getTaskExecutor();  
 for (ApplicationListener<?> listener : getApplicationListeners(event, type)) {  
 if (executor != null) { //@1  
 executor.execute(() -> invokeListener(listener, event));  
 }  
 else {  
 invokeListener(listener, event);  
 }  
 }  
}

上面的invokeListener方法内部就是调用监听器，从代码@1可以看出，如果当前executor不为空，监听器就会被异步调用，所以如果需要异步只需要让executor不为空就可以了，但是默认情况下executor是空的，此时需要我们来给其设置一个值，下面我们需要看容器中是如何创建广播器的，我们在那个地方去干预。

通常我们使用的容器是AbstractApplicationContext类型的，需要看一下AbstractApplicationContext中广播器是怎么初始化的，就是下面这个方法，容器启动的时候会被调用，用来初始化AbstractApplicationContext中的事件广播器applicationEventMulticaster

public static final String APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME = "applicationEventMulticaster";  
  
protected void initApplicationEventMulticaster() {  
 ConfigurableListableBeanFactory beanFactory = getBeanFactory();  
 if (beanFactory.containsLocalBean(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME)) {  
 this.applicationEventMulticaster =  
 beanFactory.getBean(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME, ApplicationEventMulticaster.class);  
 }  
 else {  
 this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);  
 beanFactory.registerSingleton(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME, this.applicationEventMulticaster);  
 }  
}

上面逻辑解释一下：判断spring容器中是否有名称为applicationEventMulticaster的bean，如果有就将其作为事件广播器，否则创建一个SimpleApplicationEventMulticaster作为广播器，并将其注册到spring容器中。

从上面可以得出结论：我们只需要自定义一个类型为SimpleApplicationEventMulticaster名称为applicationEventMulticaster的bean就可以了，顺便给executor设置一个值，就可以实现监听器异步执行了。

#### 具体实现如下

package com.javacode2018.lesson003.demo1.test5;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.event.ApplicationEventMulticaster;  
import org.springframework.context.event.SimpleApplicationEventMulticaster;  
import org.springframework.scheduling.concurrent.ThreadPoolExecutorFactoryBean;  
  
import java.util.concurrent.Executor;  
  
@ComponentScan  
@Configuration  
public class MainConfig5 {  
 @Bean  
 public ApplicationEventMulticaster applicationEventMulticaster() { //@1  
 //创建一个事件广播器  
 SimpleApplicationEventMulticaster result = new SimpleApplicationEventMulticaster();  
 //给广播器提供一个线程池，通过这个线程池来调用事件监听器  
 Executor executor = this.applicationEventMulticasterThreadPool().getObject();  
 //设置异步执行器  
 result.setTaskExecutor(executor);//@1  
 return result;  
 }  
  
 @Bean  
 public ThreadPoolExecutorFactoryBean applicationEventMulticasterThreadPool() {  
 ThreadPoolExecutorFactoryBean result = new ThreadPoolExecutorFactoryBean();  
 result.setThreadNamePrefix("applicationEventMulticasterThreadPool-");  
 result.setCorePoolSize(5);  
 return result;  
 }  
}

@1：定义了一个名称为applicationEventMulticaster的事件广播器，内部设置了一个线程池用来异步调用监听器

这段代码对应的测试用例

com.javacode2018.lesson003.demo1.EventTest#test5

运行输出

当前线程【Thread[main,5,main]】,用户【马士兵老师】注册成功  
当前线程【Thread[applicationEventMulticasterThreadPool-2,5,main]】,给用户【马士兵老师】发送注册成功邮件!  
当前线程【Thread[applicationEventMulticasterThreadPool-1,5,main]】,给用户【马士兵老师】发放一些优惠券!

此时实现了监听器异步执行的效果。

### 关于事件使用建议

1. spring中事件是使用接口的方式还是使用注解的方式？具体使用哪种方式都可以，不过在公司内部最好大家都统一使用一种方式
2. 异步事件的模式，通常将一些非主要的业务放在监听器中执行，因为监听器中存在失败的风险，所以使用的时候需要注意。如果只是为了解耦，但是被解耦的次要业务也是必须要成功的，可以使用消息中间件的方式来解决这些问题。
3. 事件的使用就到这里，有问题的欢迎留言讨论。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解

## Spring系列第28篇：循环bean详解

今天要说的是spring中循环依赖的问题，最近有大量粉丝问这个问题，也是高薪面试中经常会被问到的一个问题。

关于循环依赖的问题，来感受一下连环炮，试试自己否可以过关斩将，轻松应对。

1. 什么是循环依赖？
2. 如何检测是否存在循环依赖？
3. 如何解决循环依赖？
4. 多例的情况下，循环依赖问题为什么无法解决？
5. 单例的情况下，虽然可以解决循环依赖，是否存在其他问题？
6. 为什么采用三级缓存解决循环依赖？如果直接将早期bean丢到二级缓存可以么？

前面4个ok的，超越了80%的人，后面2个难度指数递增，能回答出来的算是千分之一，如果能回答上来，会让面试官相当佩服你的。

下面我们来一个个突破。

### 什么是循环依赖？

这个很好理解，多个bean之间相互依赖，形成了一个闭环。

比如：A依赖于B、B依赖于C、C依赖于A。

代码中表示：

public class A{  
 B b;  
}  
public class B{  
 C c;  
}  
public class C{  
 A a;  
}

### 如何检测是否存在循环依赖？

检测循环依赖比较简单，使用一个列表来记录正在创建中的bean，bean创建之前，先去记录中看一下自己是否已经在列表中了，如果在，说明存在循环依赖，如果不在，则将其加入到这个列表，bean创建完毕之后，将其再从这个列表中移除。

源码方面来看一下，spring创建单例bean时候，会调用下面方法

protected void beforeSingletonCreation(String beanName) {  
 if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.add(beanName)) {  
 throw new BeanCurrentlyInCreationException(beanName);  
 }  
 }

singletonsCurrentlyInCreation就是用来记录目前正在创建中的bean名称列表，this.singletonsCurrentlyInCreation.add(beanName)返回false，说明beanName已经在当前列表中了，此时会抛循环依赖的异常BeanCurrentlyInCreationException，这个异常对应的源码：

public BeanCurrentlyInCreationException(String beanName) {  
 super(beanName,  
 "Requested bean is currently in creation: Is there an unresolvable circular reference?");  
 }

上面是单例bean检测循环依赖的源码，再来看看非单例bean的情况。

以prototype情况为例，源码位于org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean方法中，将主要代码列出来看一下：

//检查正在创建的bean列表中是否存在beanName，如果存在，说明存在循环依赖，抛出循环依赖的异常  
if (isPrototypeCurrentlyInCreation(beanName)) {  
 throw new BeanCurrentlyInCreationException(beanName);  
}  
  
//判断scope是否是prototype  
if (mbd.isPrototype()) {  
 Object prototypeInstance = null;  
 try {  
 //将beanName放入正在创建的列表中  
 beforePrototypeCreation(beanName);  
 prototypeInstance = createBean(beanName, mbd, args);  
 }  
 finally {  
 //将beanName从正在创建的列表中移除  
 afterPrototypeCreation(beanName);  
 }  
}

### Spring如何解决循环依赖的问题

这块建议大家先看一下：[详解spring bean的生命周期](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648934322&idx=1&sn=647edffeedeb8978c18ad403b1f3d8d7&chksm=88621f8cbf15969af1c5396903dcce312c1f316add1af325327d287e90be49bbeda52bc1e736&token=718443976&lang=zh_CN" \l "rd)。

spring创建bean主要的几个步骤：

1. **步骤1：实例化bean，即调用构造器创建bean实例**
2. **步骤2：填充属性，注入依赖的bean，比如通过set方式、@Autowired注解的方式注入依赖的bean**
3. **步骤3：bean的初始化，比如调用init方法等。**

从上面3个步骤中可以看出，注入依赖的对象，有2种情况：

1. 通过步骤1中构造器的方式注入依赖
2. 通过步骤2注入依赖

先来看构造器的方式注入依赖的bean，下面两个bean循环依赖

@Component  
public class ServiceA {  
 private ServiceB serviceB;  
  
 public ServiceA(ServiceB serviceB) {  
 this.serviceB = serviceB;  
 }  
}  
  
@Component  
public class ServiceB {  
 private ServiceA serviceA;  
  
 public ServiceB(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
}

构造器的情况比较容易理解，实例化ServiceA的时候，需要有serviceB，而实例化ServiceB的时候需要有serviceA，构造器循环依赖是无法解决的，大家可以尝试一下使用编码的方式创建上面2个对象，是无法创建成功的！

再来看看非构造器的方式注入相互依赖的bean，以set方式注入为例，下面是2个单例的bean：serviceA和serviceB：

@Component  
public class ServiceA {  
 private ServiceB serviceB;  
  
 @Autowired  
 public void setServiceB(ServiceB serviceB) {  
 this.serviceB = serviceB;  
 }  
}  
  
  
@Component  
public class ServiceB {  
 private ServiceA serviceA;  
  
 @Autowired  
 public void setServiceA(ServiceA serviceA) {  
 this.serviceA = serviceA;  
 }  
}

如果我们采用硬编码的方式创建上面2个对象，过程如下：

//创建serviceA  
ServiceA serviceA = new ServiceA();  
//创建serviceB  
ServiceB serviceB = new ServiceB();  
//将serviceA注入到serviceB中  
serviceB.setServiceA(serviceA);  
//将serviceB注入到serviceA中  
serviceA.setServiceB(serviceB);

由于单例bean在spring容器中只存在一个，所以spring容器中肯定是有一个缓存来存放所有已创建好的单例bean；获取单例bean之前，可以先去缓存中找，找到了直接返回，找不到的情况下再去创建，创建完毕之后再将其丢到缓存中，可以使用一个map来存储单例bean，比如下面这个

Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

下面来看一下spring中set方法创建上面2个bean的过程

1.spring轮询准备创建2个bean：serviceA和serviceB  
2.spring容器发现singletonObjects中没有serviceA  
3.调用serviceA的构造器创建serviceA实例  
4.serviceA准备注入依赖的对象，发现需要通过setServiceB注入serviceB  
5.serviceA向spring容器查找serviceB  
6.spring容器发现singletonObjects中没有serviceB  
7.调用serviceB的构造器创建serviceB实例  
8.serviceB准备注入依赖的对象，发现需要通过setServiceA注入serviceA  
9.serviceB向spring容器查找serviceA  
10.此时又进入步骤2了

卧槽，上面过程死循环了，怎么才能终结？

可以在第3步后加一个操作：将实例化好的serviceA丢到singletonObjects中，此时问题就解决了。

spring中也采用类似的方式，稍微有点区别，上面使用了一个缓存，而spring内部采用了3级缓存来解决这个问题，我们一起来细看一下。

3级缓存对应的代码：

/\*\* 第一级缓存：单例bean的缓存 \*/  
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);  
  
/\*\* 第二级缓存：早期暴露的bean的缓存 \*/  
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);  
  
/\*\* 第三级缓存：单例bean工厂的缓存 \*/  
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

下面来看spring中具体的过程，我们一起来分析源码

开始的时候，获取serviceA，会调用下面代码

org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean  
  
protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,  
 @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {  
 //1.查看缓存中是否已经有这个bean了  
 Object sharedInstance = getSingleton(beanName); //@1  
 if (sharedInstance != null && args == null) {  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  
 }else {  
 //若缓存中不存在，准备创建这个bean  
 if (mbd.isSingleton()) {  
 //2.下面进入单例bean的创建过程  
 sharedInstance = getSingleton(beanName, () -> {  
 try {  
 return createBean(beanName, mbd, args);  
 }  
 catch (BeansException ex) {  
 throw ex;  
 }  
 });  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
 }  
 }  
 return (T) bean;  
}

@1：查看缓存中是否已经有这个bean了，如下：

public Object getSingleton(String beanName) {  
 return getSingleton(beanName, true);  
}

然后进入下面方法，会依次尝试从3级缓存中查找bean，注意下面的第2个参数，为ture的时候，才会从第3级中查找，否则只会查找1、2级缓存

//allowEarlyReference:是否允许从三级缓存singletonFactories中通过getObject拿到bean  
protected Object getSingleton(String beanName, boolean allowEarlyReference) {  
 //1.先从一级缓存中找  
 Object singletonObject = this.singletonObjects.get(beanName);  
 if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
 synchronized (this.singletonObjects) {  
 //2.从二级缓存中找  
 singletonObject = this.earlySingletonObjects.get(beanName);  
 if (singletonObject == null && allowEarlyReference) {  
 //3.二级缓存中没找到 && allowEarlyReference为true的情况下,从三级缓存中找  
 ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);  
 if (singletonFactory != null) {  
 //三级缓存返回的是一个工厂，通过工厂来获取创建bean  
 singletonObject = singletonFactory.getObject();  
 //将创建好的bean丢到二级缓存中  
 this.earlySingletonObjects.put(beanName, singletonObject);  
 //从三级缓存移除  
 this.singletonFactories.remove(beanName);  
 }  
 }  
 }  
 }  
 return singletonObject;  
}

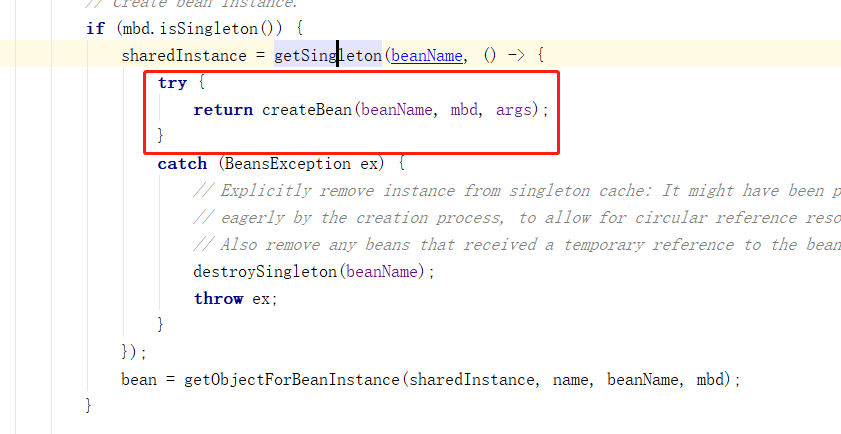
刚开始，3个缓存中肯定是找不到的，会返回null，接着会执行下面代码准备创建serviceA

if (mbd.isSingleton()) {  
 sharedInstance = getSingleton(beanName, () -> { //@1  
 try {  
 return createBean(beanName, mbd, args);  
 }  
 catch (BeansException ex) {  
 destroySingleton(beanName);  
 throw ex;  
 }  
 });  
}

@1：进入getSingleton方法，而getSingleton方法代码比较多，为了方便大家理解，无关的代码我给剔除了，如下：

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  
 synchronized (this.singletonObjects) {  
 Object singletonObject = this.singletonObjects.get(beanName);  
 if (singletonObject == null) {  
 //单例bean创建之前调用，将其加入正在创建的列表中，上面有提到过，主要用来检测循环依赖用的  
 beforeSingletonCreation(beanName);  
 boolean newSingleton = false;  
   
 try {  
 //调用工厂创建bean  
 singletonObject = singletonFactory.getObject();//@1  
 newSingleton = true;  
 }  
 finally {  
 //单例bean创建之前调用,主要是将其从正在创建的列表中移除  
 afterSingletonCreation(beanName);  
 }  
 if (newSingleton) {  
 //将创建好的单例bean放入缓存中  
 addSingleton(beanName, singletonObject);//@2  
 }  
 }  
 return singletonObject;  
 }  
}

上面@1和@2是关键代码，先来看一下@1，这个是一个ObjectFactory类型的，从外面传入的，如下



红框中的createBean最终会调用下面这个方法

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean

其内部主要代码如下：

BeanWrapper instanceWrapper = null;  
  
if (instanceWrapper == null) {  
 //通过反射调用构造器实例化serviceA  
 instanceWrapper = createBeanInstance(beanName, mbd, args);  
}  
//变量bean：表示刚刚同构造器创建好的bean示例  
final Object bean = instanceWrapper.getWrappedInstance();  
  
//判断是否需要暴露早期的bean，条件为（是否是单例bean && 当前容器允许循环依赖 && bean名称存在于正在创建的bean名称清单中）  
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  
 isSingletonCurrentlyInCreation(beanName));  
if (earlySingletonExposure) {  
 //若earlySingletonExposure为true，通过下面代码将早期的bean暴露出去  
 addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));//@1  
}

**这里需要理解一下什么是早期bean？**

**刚刚实例化好的bean就是早期的bean，此时bean还未进行属性填充，初始化等操作**

@1：通过addSingletonFactory用于将早期的bean暴露出去，主要是将其丢到第3级缓存中，代码如下：

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {  
 Assert.notNull(singletonFactory, "Singleton factory must not be null");  
 synchronized (this.singletonObjects) {  
 //第1级缓存中不存在bean  
 if (!this.singletonObjects.containsKey(beanName)) {  
 //将其丢到第3级缓存中  
 this.singletonFactories.put(beanName, singletonFactory);  
 //后面的2行代码不用关注  
 this.earlySingletonObjects.remove(beanName);  
 this.registeredSingletons.add(beanName);  
 }  
 }  
}

上面的方法执行之后，serviceA就被丢到第3级的缓存中了。

后续的过程serviceA开始注入依赖的对象，发现需要注入serviceB，会从容器中获取serviceB，而serviceB的获取又会走上面同样的过程实例化serviceB，然后将serviceB提前暴露出去，然后serviceB开始注入依赖的对象，serviceB发现自己需要注入serviceA，此时去容器中找serviceA，找serviceA会先去缓存中找，会执行getSingleton("serviceA",true)，此时会走下面代码：

protected Object getSingleton(String beanName, boolean allowEarlyReference) {  
 //1.先从一级缓存中找  
 Object singletonObject = this.singletonObjects.get(beanName);  
 if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
 synchronized (this.singletonObjects) {  
 //2.从二级缓存中找  
 singletonObject = this.earlySingletonObjects.get(beanName);  
 if (singletonObject == null && allowEarlyReference) {  
 //3.二级缓存中没找到 && allowEarlyReference为true的情况下,从三级缓存中找  
 ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);  
 if (singletonFactory != null) {  
 //三级缓存返回的是一个工厂，通过工厂来获取创建bean  
 singletonObject = singletonFactory.getObject();  
 //将创建好的bean丢到二级缓存中  
 this.earlySingletonObjects.put(beanName, singletonObject);  
 //从三级缓存移除  
 this.singletonFactories.remove(beanName);  
 }  
 }  
 }  
 }  
 return singletonObject;  
}

上面的方法走完之后，serviceA会被放入二级缓存earlySingletonObjects中，会将serviceA返回，此时serviceB中的serviceA注入成功，serviceB继续完成创建，然后将自己返回给serviceA，此时serviceA通过set方法将serviceB注入。

serviceA创建完毕之后，会调用addSingleton方法将其加入到缓存中，这块代码如下：

protected void addSingleton(String beanName, Object singletonObject) {  
 synchronized (this.singletonObjects) {  
 //将bean放入第1级缓存中  
 this.singletonObjects.put(beanName, singletonObject);  
 //将其从第3级缓存中移除  
 this.singletonFactories.remove(beanName);  
 //将其从第2级缓存中移除  
 this.earlySingletonObjects.remove(beanName);  
 }  
}

到此，serviceA和serviceB之间的循环依赖注入就完成了。

下面捋一捋整个过程：

1.从容器中获取serviceA  
2.容器尝试从3个缓存中找serviceA，找不到  
3.准备创建serviceA  
4.调用serviceA的构造器创建serviceA，得到serviceA实例，此时serviceA还未填充属性，未进行其他任何初始化的操作  
5.将早期的serviceA暴露出去：即将其丢到第3级缓存singletonFactories中  
6.serviceA准备填充属性，发现需要注入serviceB，然后向容器获取serviceB  
7.容器尝试从3个缓存中找serviceB，找不到  
8.准备创建serviceB  
9.调用serviceB的构造器创建serviceB，得到serviceB实例，此时serviceB还未填充属性，未进行其他任何初始化的操作  
10.将早期的serviceB暴露出去：即将其丢到第3级缓存singletonFactories中  
11.serviceB准备填充属性，发现需要注入serviceA，然后向容器获取serviceA  
12.容器尝试从3个缓存中找serviceA，发现此时serviceA位于第3级缓存中，经过处理之后，serviceA会从第3级缓存中移除，然后会存到第2级缓存中，然后将其返回给serviceB，此时serviceA通过serviceB中的setServiceA方法被注入到serviceB中  
13.serviceB继续执行后续的一些操作，最后完成创建工作，然后会调用addSingleton方法，将自己丢到第1级缓存中，并将自己从第2和第3级缓存中移除  
14.serviceB将自己返回给serviceA  
15.serviceA通过setServiceB方法将serviceB注入进去  
16.serviceB继续执行后续的一些操作，最后完成创建工作,然后会调用addSingleton方法，将自己丢到第1级缓存中，并将自己从第2和第3级缓存中移除

### 循环依赖无法解决的情况

**只有单例的bean会通过三级缓存提前暴露来解决循环依赖的问题，而非单例的bean，每次从容器中获取都是一个新的对象，都会重新创建，所以非单例的bean是没有缓存的，不会将其放到三级缓存中。**

那就会有下面几种情况需要注意。

还是以2个bean相互依赖为例：serviceA和serviceB

#### 情况1

##### 条件

serviceA：多例

serviceB：多例

##### 结果

此时不管是任何方式都是无法解决循环依赖的问题，最终都会报错，因为每次去获取依赖的bean都会重新创建。

#### 情况2

##### 条件

serviceA：单例

serviceB：多例

##### 结果

若使用构造器的方式相互注入，是无法完成注入操作的，会报错。

若采用set方式注入，所有bean都还未创建的情况下，若去容器中获取serviceB，会报错，为什么？我们来看一下过程：

1.从容器中获取serviceB  
2.serviceB由于是多例的，所以缓存中肯定是没有的  
3.检查serviceB是在正在创建的bean名称列表中，没有  
4.准备创建serviceB  
5.将serviceB放入正在创建的bean名称列表中  
6.实例化serviceB（由于serviceB是多例的，所以不会提前暴露，必须是单例的才会暴露）  
7.准备填充serviceB属性，发现需要注入serviceA  
8.从容器中查找serviceA  
9.尝试从3级缓存中找serviceA，找不到  
10.准备创建serviceA  
11.将serviceA放入正在创建的bean名称列表中  
12.实例化serviceA  
13.由于serviceA是单例的，将早期serviceA暴露出去，丢到第3级缓存中  
14.准备填充serviceA的属性，发现需要注入serviceB  
15.从容器中获取serviceB  
16.先从缓存中找serviceB，找不到  
17.检查serviceB是在正在创建的bean名称列表中,发现已经存在了，抛出循环依赖的异常

这个有演示的源码，位置：

com.javacode2018.lesson003.demo2.CircleDependentTest#test2

**在这里给大家留个问题，如果此处不是去获取serviceB，而是先去获取serviceA呢，会不会报错？欢迎各位留言。**

### 探讨：为什么需要用3级缓存

#### 问题

**如果只使用2级缓存，直接将刚实例化好的bean暴露给二级缓存出是否可以否？**

先下个结论吧：不行。

#### 原因

**这样做是可以解决：早期暴露给其他依赖者的bean和最终暴露的bean不一致的问题。**

若将刚刚实例化好的bean直接丢到二级缓存中暴露出去，如果后期这个bean对象被更改了，比如可能在上面加了一些拦截器，将其包装为一个代理了，那么暴露出去的bean和最终的这个bean就不一样的，将自己暴露出去的时候是一个原始对象，而自己最终却是一个代理对象，最终会导致被暴露出去的和最终的bean不是同一个bean的，将产生意向不到的效果，而三级缓存就可以发现这个问题，会报错。

下面我们通过代码来演示一下效果。

#### 案例

下面来2个bean，相互依赖，通过set方法相互注入，并且其内部都有一个m1方法，用来输出一行日志。

##### Service1

package com.javacode2018.lesson003.demo2.test3;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service1 {  
 public void m1() {  
 System.out.println("Service1 m1");  
 }  
  
 private Service2 service2;  
  
 @Autowired  
 public void setService2(Service2 service2) {  
 this.service2 = service2;  
 }  
  
}

##### Service2

package com.javacode2018.lesson003.demo2.test3;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
  
 public void m1() {  
 System.out.println("Service2 m1");  
 this.service1.m1();//@1  
 }  
  
 private Service1 service1;  
  
 @Autowired  
 public void setService1(Service1 service1) {  
 this.service1 = service1;  
 }  
  
 public Service1 getService1() {  
 return service1;  
 }  
}

注意上面的@1，service2的m1方法中会调用service1的m1方法。

##### 需求

在service1上面加个拦截器，要求在调用service1的任何方法之前需要先输出一行日志

你好,service1

##### 实现

新增一个Bean后置处理器来对service1对应的bean进行处理，将其封装为一个代理暴露出去。

package com.javacode2018.lesson003.demo2.test3;  
  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.lang.Nullable;  
import org.springframework.stereotype.Component;  
  
import java.lang.reflect.Method;  
  
@Component  
public class MethodBeforeInterceptor implements BeanPostProcessor {  
 @Nullable  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 if ("service1".equals(beanName)) {  
 //代理创建工厂，需传入被代理的目标对象  
 ProxyFactory proxyFactory = new ProxyFactory(bean);  
 //添加一个方法前置通知，会在方法执行之前调用通知中的before方法  
 proxyFactory.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("你好,service1");  
 }  
 });  
 //返回代理对象  
 return proxyFactory.getProxy();  
 }  
 return bean;  
 }  
}

上面的postProcessAfterInitialization方法内部会在service1初始化之后调用，内部会对service1这个bean进行处理，返回一个代理对象，通过代理来访问service1的方法，访问service1中的任何方法之前，会先输出：你好，service1。

代码中使用了ProxyFactory，这块不熟悉的没关系，后面介绍aop的时候会细说。

##### 来个配置类

@ComponentScan  
public class MainConfig3 {  
  
}

##### 来个测试用例

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig3.class);  
 context.refresh();  
}

##### 运行：报错了

org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'service1': Bean with name 'service1' has been injected into other beans [service2] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.  
  
 at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:624)

可以看出是AbstractAutowireCapableBeanFactory.java:624这个地方整出来的异常，将这块代码贴出来给大家看一下：

if (earlySingletonExposure) {  
 //@1  
 Object earlySingletonReference = getSingleton(beanName, false);  
 if (earlySingletonReference != null) {  
 //@2  
 if (exposedObject == bean) {  
 exposedObject = earlySingletonReference;  
 }  
 //@3  
 else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  
 String[] dependentBeans = getDependentBeans(beanName);  
 Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);  
 for (String dependentBean : dependentBeans) {  
 if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  
 actualDependentBeans.add(dependentBean);  
 }  
 }  
 if (!actualDependentBeans.isEmpty()) {  
 throw new BeanCurrentlyInCreationException(beanName,  
 "Bean with name '" + beanName + "' has been injected into other beans [" +  
 StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +  
 "] in its raw version as part of a circular reference, but has eventually been " +  
 "wrapped. This means that said other beans do not use the final version of the " +  
 "bean. This is often the result of over-eager type matching - consider using " +  
 "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  
 }  
 }  
 }  
}

上面代码主要用来判断当有循环依赖的情况下，早期暴露给别人使用的bean是否和最终的bean不一样的情况下，会抛出一个异常。

我们再来通过代码级别的来解释上面代码：

@1：调用getSingleton(beanName, false)方法，这个方法用来从3个级别的缓存中获取bean，但是注意了，这个地方第二个参数是false，此时只会尝试从第1级和第2级缓存中获取bean，如果能够获取到，说明了什么？说明了第2级缓存中已经有这个bean了，而什么情况下第2级缓存中会有bean？说明这个bean从第3级缓存中已经被别人获取过，然后从第3级缓存移到了第2级缓存中，说明这个早期的bean被别人通过getSingleton(beanName, true)获取过

@2：这个地方用来判断早期暴露的bean和最终spring容器对这个bean走完创建过程之后是否还是同一个bean，上面我们的service1被代理了，所以这个地方会返回false，此时会走到@3

@3：allowRawInjectionDespiteWrapping这个参数用来控制是否允许循环依赖的情况下，早期暴露给被人使用的bean在后期是否可以被包装，通俗点理解就是：是否允许早期给别人使用的bean和最终bean不一致的情况，这个值默认是false，表示不允许，也就是说你暴露给别人的bean和你最终的bean需要是一直的，你给别人的是1，你后面不能将其修改成2了啊，不一样了，你给我用个鸟。

而上面代码注入到service2中的service1是早期的service1，而最终spring容器中的service1变成一个代理对象了，早期的和最终的不一致了，而allowRawInjectionDespiteWrapping又是false，所以报异常了。

那么如何解决这个问题：

很简单，将allowRawInjectionDespiteWrapping设置为true就可以了，下面改一下代码如下：

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 //创建一个BeanFactoryPostProcessor：BeanFactory后置处理器  
 context.addBeanFactoryPostProcessor(beanFactory -> {  
 if (beanFactory instanceof DefaultListableBeanFactory) {  
 //将allowRawInjectionDespiteWrapping设置为true  
 ((DefaultListableBeanFactory) beanFactory).setAllowRawInjectionDespiteWrapping(true);  
 }  
 });  
 context.register(MainConfig3.class);  
 context.refresh();  
   
 System.out.println("容器初始化完毕");  
}

上面代码中将allowRawInjectionDespiteWrapping设置为true了，是通过一个BeanFactoryPostProcessor来实现的，后面会有一篇文章来详解BeanFactoryPostProcessor，目前你只需要知道BeanFactoryPostProcessor可以在bean创建之前用来干预BeanFactory的创建过程，可以用来修改BeanFactory中的一些配置。

##### 再次输出

容器初始化完毕

此时正常了，我们继续，看看我们加在service1上的拦截器起效了没有，上面代码中加入下面代码：

//获取service1  
Service1 service1 = context.getBean(Service1.class);  
//获取service2  
Service2 service2 = context.getBean(Service2.class);  
  
System.out.println("----A-----");  
service2.m1(); //@1  
System.out.println("----B-----");  
service1.m1(); //@2  
System.out.println("----C-----");  
System.out.println(service2.getService1() == service1);

上面为了区分结果，使用了----格式的几行日志将输出结果分开了，来运行一下，输出：

容器初始化完毕  
----A-----  
Service2 m1  
Service1 m1  
----B-----  
你好,service1  
Service1 m1  
----C-----  
false

从输出中可以看出。

service2.m1()对应输出：

Service2 m1  
Service1 m1

service1.m1()对应输出：

你好,service1  
Service1 m1

而service2.m1方法中调用了service1.m1,这个里面拦截器没有起效啊，但是单独调用service1.m1方法，却起效了，说明service2中注入的service1不是代理对象，所以没有加上拦截器的功能，那是因为service2中注入的是早期的service1，注入的时候service1还不是一个代理对象，所以没有拦截器中的功能。

再看看最后一行输出为false，说明service2中的service1确实和spring容器中的service1不是一个对象了。

ok，那么这种情况是不是很诧异，如何解决这个问题？

既然最终service1是一个代理对象，那么你提前暴露出去的时候，注入到service2的时候，你也必须得是个代理对象啊，需要确保给别人和最终是同一个对象。

这个怎么整？继续看暴露早期bean的源码，注意了下面是重点：

addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

注意有个getEarlyBeanReference方法，来看一下这个方法是干什么的，源码如下：

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {  
 Object exposedObject = bean;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {  
 SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;  
 exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);  
 }  
 }  
 }  
 return exposedObject;  
}

从3级缓存中获取bean的时候，会调用上面这个方法来获取bean，这个方法内部会看一下容器中是否有SmartInstantiationAwareBeanPostProcessor这种处理器，然后会依次调用这种处理器中的getEarlyBeanReference方法，那么思路来了，我们可以自定义一个SmartInstantiationAwareBeanPostProcessor，然后在其getEarlyBeanReference中来创建代理不就可以了，聪明，我们来试试，将MethodBeforeInterceptor代码改成下面这样：

@Component  
public class MethodBeforeInterceptor implements SmartInstantiationAwareBeanPostProcessor {  
 @Override  
 public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {  
 if ("service1".equals(beanName)) {  
 //代理创建工厂，需传入被代理的目标对象  
 ProxyFactory proxyFactory = new ProxyFactory(bean);  
 //添加一个方法前置通知，会在方法执行之前调用通知中的before方法  
 proxyFactory.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("你好,service1");  
 }  
 });  
 //返回代理对象  
 return proxyFactory.getProxy();  
 }  
 return bean;  
 }  
}

对应测试用例

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig4.class);  
 System.out.println("容器初始化完毕");  
  
 //获取service1  
 com.javacode2018.lesson003.demo2.test4.Service1 service1 = context.getBean(com.javacode2018.lesson003.demo2.test4.Service1.class);  
 //获取service2  
 com.javacode2018.lesson003.demo2.test4.Service2 service2 = context.getBean(com.javacode2018.lesson003.demo2.test4.Service2.class);  
  
 System.out.println("----A-----");  
 service2.m1(); //@1  
 System.out.println("----B-----");  
 service1.m1(); //@2  
 System.out.println("----C-----");  
 System.out.println(service2.getService1() == service1);  
}

运行输出

容器初始化完毕  
----A-----  
Service2 m1  
你好,service1  
Service1 m1  
----B-----  
你好,service1  
Service1 m1  
----C-----  
true

### 单例bean解决了循环依赖，还存在什么问题？

循环依赖的情况下，由于注入的是早期的bean，此时早期的bean中还未被填充属性，初始化等各种操作，也就是说此时bean并没有被完全初始化完毕，此时若直接拿去使用，可能存在有问题的风险。

### 总结

到目前为止，开头连环炮的问题，大家肯定都知道答案了，如果有不清楚的，欢迎留言。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 课后问题？

有两个bean：serviceA(单例)，serviceB(多例)，他们之间相互依赖，通过set方法相互注入依赖。

若两个bean都还未创建的时候，从容器中获取serviceA会不会报错？

欢迎留言讨论，可以将查找的过程写出来。

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解

## Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）

**Spring中有2个非常重要的接口：BeanFactoryPostProcessor和BeanDefinitionRegistryPostProcessor，这2个接口面试中也会经常问到，本文我们一起来拿下他们俩。**

### 先来看几个问题

1. BeanFactoryPostProcessor是做什么的？
2. BeanDefinitionRegistryPostProcessor是干什么的？
3. BeanFactoryPostProcessor和BeanDefinitionRegistryPostProcessor有什么区别？
4. 这几个接口的执行顺序是什么样的？

### Spring容器中主要的4个阶段

* 阶段1：Bean注册阶段，此阶段会完成所有bean的注册
* 阶段2：BeanFactory后置处理阶段
* 阶段3：注册BeanPostProcessor
* 阶段4：bean创建阶段，此阶段完成所有单例bean的注册和装载操作，这个阶段不是我们本文关注的重点，有兴趣的，可以去看之前的文章中有详细介绍：Bean生命周期详解

本文介绍的2个接口主要和前2个阶段有关系，下面我们主要来看前2个阶段。

### 阶段1：Bean注册阶段

#### 概述

spring中所有bean的注册都会在此阶段完成，按照规范，所有bean的注册必须在此阶段进行，其他阶段不要再进行bean的注册。

这个阶段spring为我们提供1个接口：BeanDefinitionRegistryPostProcessor，spring容器在这个阶段中会获取容器中所有类型为BeanDefinitionRegistryPostProcessor的bean，然后会调用他们的postProcessBeanDefinitionRegistry方法，源码如下，方法参数类型是BeanDefinitionRegistry，这个类型大家都比较熟悉，即bean定义注册器，内部提供了一些方法可以用来向容器中注册bean。

public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {  
 void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException;  
}

这个接口还继承了BeanFactoryPostProcessor接口，这个大家先不用关心，一会阶段2中会介绍。

当容器中有多个BeanDefinitionRegistryPostProcessor的时候，可以通过下面任意一种方式来指定顺序

1. 实现org.springframework.core.PriorityOrdered接口
2. 实现org.springframework.core.Ordered接口

执行顺序：

PriorityOrdered.getOrder() asc,Ordered.getOrder() asc

下面通过案例来感受一下效果。

#### 案例1：简单实用

此案例演示BeanDefinitionRegistryPostProcessor的简单使用

##### 自定义一个BeanDefinitionRegistryPostProcessor

下面我们定义了一个类，需要实现BeanDefinitionRegistryPostProcessor接口，然后会让我们实现2个方法，大家重点关注postProcessBeanDefinitionRegistry这个方法，另外一个方法来自于BeanFactoryPostProcessor，一会我们后面在介绍这个方法，在postProcessBeanDefinitionRegistry方法中，我们定义了一个bean，然后通过registry将其注册到容器了，代码很简单

package com.javacode2018.lesson003.demo3.test0;  
  
@Component  
public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor {  
 @Override  
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {  
 //定义一个字符串类型的bean  
 AbstractBeanDefinition userNameBdf = BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("路人").  
 getBeanDefinition();  
 //将userNameBdf注册到spring容器中  
 registry.registerBeanDefinition("userName", userNameBdf);  
 }  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
  
 }  
}

##### 同包中来个配置类

package com.javacode2018.lesson003.demo3.test0;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig0 {  
}

##### 测试用例

package com.javacode2018.lesson003.demo3;  
  
import com.javacode2018.lesson003.demo3.test0.MainConfig0;  
import com.javacode2018.lesson003.demo3.test1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class BeanDefinitionRegistryPostProcessorTest {  
 @Test  
 public void test0() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig0.class);  
 context.refresh();  
 System.out.println(context.getBean("userName"));  
 }  
}

##### 运行输出

路人

#### 案例2：多个指定顺序

下面我们定义2个BeanDefinitionRegistryPostProcessor，都实现Ordered接口，第一个order的值为2，第二个order的值为1，我们来看一下具体执行的顺序。

##### 第一个

package com.javacode2018.lesson003.demo3.test1;  
  
@Component  
public class BeanDefinitionRegistryPostProcessor1 implements BeanDefinitionRegistryPostProcessor, Ordered {  
 @Override  
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {  
 System.out.println(String.format("BeanDefinitionRegistryPostProcessor1{order=%d},注册name bean,", this.getOrder()));  
 //定义一个bean  
 AbstractBeanDefinition nameBdf = BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("马士兵老师").  
 getBeanDefinition();  
 //将定义的bean注册到容器  
 registry.registerBeanDefinition("name", nameBdf);  
 }  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
  
 }  
  
 @Override  
 public int getOrder() {  
 return 2;  
 }  
}

##### 第二个

package com.javacode2018.lesson003.demo3.test1;  
  
@Component  
public class BeanDefinitionRegistryPostProcessor2 implements BeanDefinitionRegistryPostProcessor, Ordered {  
 @Override  
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {  
 System.out.println(String.format("BeanDefinitionRegistryPostProcessor2{order=%d},注册car bean,", this.getOrder()));  
 //定义一个bean  
 AbstractBeanDefinition nameBdf = BeanDefinitionBuilder.  
 genericBeanDefinition(String.class).  
 addConstructorArgValue("保时捷").  
 getBeanDefinition();  
 //将定义的bean注册到容器  
 registry.registerBeanDefinition("car", nameBdf);  
 }  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
  
 }  
  
 @Override  
 public int getOrder() {  
 return 1;  
 }  
}

上面2个类中的postProcessBeanDefinitionRegistry方法第一行都有输出，一个可以通过运行结果看到执行的顺序。

##### 同包中添加配置类

package com.javacode2018.lesson003.demo3.test1;  
  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class MainConfig1 {  
}

##### 测试案例

@Test  
public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 context.getBeansOfType(String.class).forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
}

##### 运行输出

BeanDefinitionRegistryPostProcessor2{order=1},注册car bean,  
BeanDefinitionRegistryPostProcessor1{order=2},注册name bean,  
car->保时捷  
name->马士兵老师

#### 小结

BeanDefinitionRegistryPostProcessor有个非常重要的实现类：

org.springframework.context.annotation.ConfigurationClassPostProcessor

这个类可能有些人不熟悉，下面这些注解大家应该比较熟悉吧，这些注解都是在上面这个类中实现的，通过这些注解来实现bean的批量注册

@Configuration  
@ComponentScan  
@Import  
@ImportResource  
@PropertySource

有兴趣的朋友可以去看一下ConfigurationClassPostProcessor#postProcessBeanDefinitionRegistry研究一下上面这些注解的解析过程，可以学到很多东西。

### 阶段2：BeanFactory后置处理阶段

#### 概述

到这个阶段的时候，spring容器已经完成了所有bean的注册，这个阶段中你可以对BeanFactory中的一些信息进行修改，比如修改阶段1中一些bean的定义信息，修改BeanFactory的一些配置等等，此阶段spring也提供了一个接口来进行扩展：BeanFactoryPostProcessor，简称bfpp，接口中有个方法postProcessBeanFactory，spring会获取容器中所有BeanFactoryPostProcessor类型的bean，然后调用他们的postProcessBeanFactory，来看一下这个接口的源码：

@FunctionalInterface  
public interface BeanFactoryPostProcessor {  
  
 void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;  
  
}

当容器中有多个BeanFactoryPostProcessor的时候，可以通过下面任意一种方式来指定顺序

1. 实现org.springframework.core.PriorityOrdered接口
2. 实现org.springframework.core.Ordered接口

执行顺序：

PriorityOrdered.getOrder() asc,Ordered.getOrder() asc

下面通过案例来感受一下效果。

#### 案例

这个案例中演示，在BeanFactoryPostProcessor来修改bean中已经注册的bean定义的信息，给一个bean属性设置一个值。

##### 先来定义一个bean

package com.javacode2018.lesson003.demo3.test2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class LessonModel {  
 //课程名称  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "LessonModel{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

上面这个bean有个name字段，并没有设置值，下面我们在BeanFactoryPostProcessor来对其设置值。

##### 自定义的BeanFactoryPostProcessor

下面代码中，我们先获取lessonModel这个bean的定义信息，然后给其name属性设置了一个值。

package com.javacode2018.lesson003.demo3.test2;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanDefinition;  
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
 System.out.println("准备修改lessonModel bean定义信息!");  
 BeanDefinition beanDefinition = beanFactory.getBeanDefinition("lessonModel");  
 beanDefinition.getPropertyValues().add("name", "spring高手系列!");  
 }  
  
}

##### 测试用例

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig2.class);  
 context.refresh();  
  
 System.out.println(context.getBean(LessonModel.class));  
}

##### 运行输出

准备修改lessonModel bean定义信息!  
LessonModel{name='spring高手系列!'}

结果中可以看出，通过BeanFactoryPostProcessor修改了容器中已经注册的bean定义信息。

#### 这个接口的几个重要实现类

##### PropertySourcesPlaceholderConfigurer

这个接口做什么的，大家知道么？来看一段代码

<bean class="xxxxx">  
 <property name="userName" value="${userName}"/>  
 <property name="address" value="${address}"/>  
</bean>

这个大家比较熟悉吧，spring就是在PropertySourcesPlaceholderConfigurer#postProcessBeanFactory中来处理xml中属性中的${xxx}，会对这种格式的进行解析处理为真正的值。

##### CustomScopeConfigurer

向容器中注册自定义的Scope对象，即注册自定义的作用域实现类，关于自定义的作用域，不了解的朋友，建议先看一下：Spring系列第6篇：玩转bean scope，避免跳坑里！

这个用法比较简单，定义一个CustomScopeConfigurer的bean就可以了，然后通过这个类来注册自定义的bean。

##### EventListenerMethodProcessor

处理@EventListener注解的，即spring中事件机制，需要了解spring事件的：spring事件机制详解

还有一些实现类，这里就不介绍了。

#### 使用注意

BeanFactoryPostProcessor接口的使用有一个需要注意的地方，在其postProcessBeanFactory方法中，强烈禁止去通过容器获取其他bean，此时会导致bean的提前初始化，会出现一些意想不到的问题，因为这个阶段中BeanPostProcessor还未准备好，本文开头4个阶段中有介绍，BeanPostProcessor是在第3个阶段中注册到spring容器的，而BeanPostProcessor可以对bean的创建过程进行干预，比如spring中的aop就是在BeanPostProcessor的一些子类中实现的，@Autowired也是在BeanPostProcessor的子类中处理的，此时如果去获取bean，此时bean不会被BeanPostProcessor处理，所以创建的bean可能是有问题的，还是通过一个案例给大家演示一下把，通透一些。

##### 来一个简单的类

package com.javacode2018.lesson003.demo3.test3;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class UserModel {  
 @Autowired  
 private String name; //@1  
  
 @Override  
 public String toString() {  
 return "UserModel{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

@1：使用了@Autowired，会指定注入

#### 来个配置类

配置类中定义了2个UserModel类型的bean：user1、user2

并且定义了一个String类型的bean：name，这个会注入到UserModel中的name属性中去。

package com.javacode2018.lesson003.demo3.test3;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ComponentScan  
public class MainConfig3 {  
 @Bean  
 public UserModel user1() {  
 return new UserModel();  
 }  
  
 @Bean  
 public UserModel user2() {  
 return new UserModel();  
 }  
  
 @Bean  
 public String name() {  
 return "马士兵老师,带大家成功java高手!";  
 }  
}

##### 测试用例

输出容器中所有UserModel类型的bean

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig3.class);  
 context.refresh();  
  
 context.getBeansOfType(UserModel.class).forEach((beanName, bean) -> {  
 System.out.println(String.format("%s->%s", beanName, bean));  
 });  
}

##### 运行输出

user1->UserModel{name='马士兵老师,带大家成功java高手!'}  
user2->UserModel{name='马士兵老师,带大家成功java高手!'}

效果不用多解释，大家一看就懂，下面来重点。

#### 添加一个BeanFactoryPostProcessor

在postProcessBeanFactory方法中去获取一下user1这个bean

package com.javacode2018.lesson003.demo3.test3;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
 beanFactory.getBean("user1");  
 }  
  
}

##### 再次运行输出

user1->UserModel{name='null'}  
user2->UserModel{name='马士兵老师,带大家成功java高手!'}

注意，user1中的name变成null了，什么情况？

是因为@Autowired注解是在AutowiredAnnotationBeanPostProcessor中解析的，spring容器调用BeanFactoryPostProcessor#postProcessBeanFactory的使用，此时spring容器中还没有AutowiredAnnotationBeanPostProcessor，所以此时去获取user1这个bean的时候，@Autowired并不会被处理，所以name是null。

### 源码

#### 4个阶段的源码

4个阶段的源码为位于下面这个方法中

org.springframework.context.support.AbstractApplicationContext#refresh

这个方法中截取部分代码如下：

// 对应阶段1和阶段2：调用上下文中注册为bean的工厂处理器，即调用本文介绍的2个接口中的方法  
invokeBeanFactoryPostProcessors(beanFactory);  
  
// 对应阶段3：注册拦截bean创建的bean处理器，即注册BeanPostProcessor  
registerBeanPostProcessors(beanFactory);  
  
// 对应阶段3：实例化所有剩余的（非延迟初始化）单例。  
finishBeanFactoryInitialization(beanFactory);

阶段1和阶段2的源码位于下面方法中，代码比较简单，强烈建议大家去看一下，几分钟就可以看懂了。

org.springframework.context.support.PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors(org.springframework.beans.factory.config.ConfigurableListableBeanFactory, java.util.List<org.springframework.beans.factory.config.BeanFactoryPostProcessor>)

### 总结

1. 注意spring的4个阶段：bean定义阶段、BeanFactory后置处理阶段、BeanPostProcessor注册阶段、单例bean创建组装阶段
2. BeanDefinitionRegistryPostProcessor会在第一个阶段被调用，用来实现bean的注册操作，这个阶段会完成所有bean的注册
3. BeanFactoryPostProcessor会在第2个阶段被调用，到这个阶段时候，bean此时已经完成了所有bean的注册操作，这个阶段中你可以对BeanFactory中的一些信息进行修改，比如修改阶段1中一些bean的定义信息，修改BeanFactory的一些配置等等
4. 阶段2的时候，2个禁止操作：禁止注册bean、禁止从容器中获取bean
5. 本文介绍的2个接口的实现类可以通过PriorityOrdered接口或者Ordered接口来指定顺序

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 课后作业

2个问题，欢迎留言讨论，会有红包惊喜哦！

#### 问题1

大家去读一下org.springframework.context.support.AbstractApplicationContext#invokeBeanFactoryPostProcessors的源码，将内部主要的执行过程通过文字描述出来，然后留言

#### 问题2

BeanPostProcessor和BeanFactoryPostProcessors有何区别？留言

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解

## Spring系列第30篇：jdk动态代理和cglib代理

Spring中有个非常重要的知识点，AOP，即面相切面编程，spring中提供的一些非常牛逼的功能都是通过aop实现的，比如下面这些大家比较熟悉的功能

1. spring事务管理：@Transactional
2. spring异步处理：@EnableAsync
3. spring缓存技术的使用：@EnableCaching
4. spring中各种拦截器：@EnableAspectJAutoProxy

大家想玩转spring，成为一名spring高手，aop是必须要掌握的，aop这块东西比较多，我们将通过三四篇文章来详解介绍这块的内容，由浅入深，让大家全面掌握这块知识。

说的简单点，spring中的aop就是依靠代理实现的各种功能，通过代理来对bean进行增强。

spring中的aop功能主要是通过2种代理来实现的

1. jdk动态代理
2. cglib代理

继续向下之前，必须先看一下这篇文章：Spring系列第15篇：代理详解（Java动态代理&cglib代理）？

spring aop中用到了更多的一些特性，上面这边文章中没有介绍到，所以通过本文来做一个补充，这2篇文章看过之后，再去看spring aop的源码，理解起来会容易一些，这2篇算是最基础的知识，所以一定要消化理解，不然aop那块的原理你很难了解，会晕车，

### jdk动态代理

#### 特征

1. 只能为接口创建代理对象
2. 创建出来的代理都是java.lang.reflect.Proxy的子类

#### 案例

案例源码位置：

com.javacode2018.aop.demo1.JdkAopTest1

有2个接口

interface IService1 {  
 void m1();  
}  
  
interface IService2 {  
 void m2();  
}

下面的类实现了上面2个接口

public static class Service implements IService1, IService2 {  
 @Override  
 public void m1() {  
 System.out.println("我是m1");  
 }  
  
 @Override  
 public void m2() {  
 System.out.println("我是m2");  
 }  
}

下面通过jdk动态代理创建一个代理对象，实现上面定义的2个接口，将代理对象所有的请求转发给Service去处理，需要在代理中统计2个接口中所有方法的耗时。

比较简单，自定义一个InvocationHandler

public static class CostTimeInvocationHandler implements InvocationHandler {  
  
 private Object target;  
  
 public CostTimeInvocationHandler(Object target) {  
 this.target = target;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 long startime = System.nanoTime();  
 Object result = method.invoke(this.target, args); //将请求转发给target去处理  
 System.out.println(method + "，耗时(纳秒):" + (System.nanoTime() - startime));  
 return result;  
 }  
}

测试方法

{  
 Service target = new Service();  
 CostTimeInvocationHandler costTimeInvocationHandler = new CostTimeInvocationHandler(target);  
 //创建代理对象  
 Object proxyObject = Proxy.newProxyInstance(  
 target.getClass().getClassLoader(),  
 new Class[]{IService1.class, IService2.class}, //创建的代理对象实现了2个接口  
 costTimeInvocationHandler);  
 //判断代理对象是否是Service类型的，肯定是false咯  
 System.out.println(String.format("proxyObject instanceof Service = %s", proxyObject instanceof Service));  
 //判断代理对象是否是IService1类型的，肯定是true  
 System.out.println(String.format("proxyObject instanceof IService1 = %s", proxyObject instanceof IService1));  
 //判断代理对象是否是IService2类型的，肯定是true  
 System.out.println(String.format("proxyObject instanceof IService2 = %s", proxyObject instanceof IService2));  
  
 //将代理转换为IService1类型  
 IService1 service1 = (IService1) proxyObject;  
 //调用IService2的m1方法  
 service1.m1();  
 //将代理转换为IService2类型  
 IService2 service2 = (IService2) proxyObject;  
 //调用IService2的m2方法  
 service2.m2();  
 //输出代理的类型  
 System.out.println("代理对象的类型:" + proxyObject.getClass());  
}

运行输出

proxyObject instanceof Service = false  
proxyObject instanceof IService1 = true  
proxyObject instanceof IService2 = true  
我是m1  
public abstract void com.javacode2018.aop.demo1.JdkAopTest1$IService1.m1()，耗时(纳秒):225600  
我是m2  
public abstract void com.javacode2018.aop.demo1.JdkAopTest1$IService2.m2()，耗时(纳秒):36000  
代理对象的类型:class com.javacode2018.aop.demo1.$Proxy0

m1方法和m2方法被CostTimeInvocationHandler#invoke给增强了，调用目标方法的过程中统计了耗时。

最后一行输出可以看出代理对象的类型，类名中包含了$Proxy的字样，所以以后注意，看到这种字样的，基本上都是通过jdk动态代理创建的代理对象。

下面来说cglib代理的一些特殊案例。

### cglib代理

#### cglib的特点

1. cglib弥补了jdk动态代理的不足，jdk动态代理只能为接口创建代理，而cglib非常强大，不管是接口还是类，都可以使用cglib来创建代理
2. cglib创建代理的过程，相当于创建了一个新的类，可以通过cglib来配置这个新的类需要实现的接口，以及需要继承的父类
3. cglib可以为类创建代理，但是这个类不能是final类型的，cglib为类创建代理的过程，实际上为通过继承来实现的，相当于给需要被代理的类创建了一个子类，然后会重写父类中的方法，来进行增强，继承的特性大家应该都知道，final修饰的类是不能被继承的，final修饰的方法不能被重写，static修饰的方法也不能被重写，private修饰的方法也不能被子类重写，而其他类型的方法都可以被子类重写，被重写的这些方法可以通过cglib进行拦截增强

#### cglib整个过程如下

1. Cglib根据父类,Callback, Filter 及一些相关信息生成key
2. 然后根据key 生成对应的子类的二进制表现形式
3. 使用ClassLoader装载对应的二进制,生成Class对象,并缓存
4. 最后实例化Class对象,并缓存

#### 案例1：为多个接口创建代理

代码比较简单，定义了2个接口，然后通过cglib来创建一个代理类，代理类会实现这2个接口，通过setCallback来对2个接口的方法进行增强。

public class CglibTest1 {  
 interface IService1 {  
 void m1();  
 }  
  
 interface IService2 {  
 void m2();  
 }  
  
 public static void main(String[] args) {  
 Enhancer enhancer = new Enhancer();  
 //设置代理对象需要实现的接口  
 enhancer.setInterfaces(new Class[]{IService1.class, IService2.class});  
 //通过Callback来对被代理方法进行增强  
 enhancer.setCallback(new MethodInterceptor() {  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 System.out.println("方法：" + method.getName());  
 return null;  
 }  
 });  
 Object proxy = enhancer.create();  
 if (proxy instanceof IService1) {  
 ((IService1) proxy).m1();  
 }  
 if (proxy instanceof IService2) {  
 ((IService2) proxy).m2();  
 }  
 //看一下代理对象的类型  
 System.out.println(proxy.getClass());  
 //看一下代理类实现的接口  
 System.out.println("创建代理类实现的接口如下：");  
 for (Class<?> cs : proxy.getClass().getInterfaces()) {  
 System.out.println(cs);  
 }  
 }  
}

运行输出

方法：m1  
方法：m2  
class com.javacode2018.aop.demo2.CglibTest1$IService1$$EnhancerByCGLIB$$1d32a82  
创建代理类实现的接口如下：  
interface com.javacode2018.aop.demo2.CglibTest1$IService1  
interface com.javacode2018.aop.demo2.CglibTest1$IService2  
interface org.springframework.cglib.proxy.Factory

上面创建的代理类相当于下面代码

public class CglibTest1$IService1$$EnhancerByCGLIB$$1d32a82 implements IService1, IService2 {  
 @Override  
 public void m1() {  
 System.out.println("方法：m1");  
 }  
  
 @Override  
 public void m2() {  
 System.out.println("方法：m2");  
  
 }  
}

#### 案例2：为类和接口同时创建代理

下面定义了2个接口：IService1和IService2，2个接口有个实现类：Service，然后通过cglib创建了个代理类，实现了这2个接口，并且将Service类作为代理类的父类。

public class CglibTest2 {  
 interface IService1 {  
 void m1();  
 }  
  
 interface IService2 {  
 void m2();  
 }  
  
 public static class Service implements IService1, IService2 {  
 @Override  
 public void m1() {  
 System.out.println("m1");  
 }  
  
 @Override  
 public void m2() {  
 System.out.println("m2");  
 }  
 }  
  
 public static void main(String[] args) {  
 Enhancer enhancer = new Enhancer();  
 //设置代理类的父类  
 enhancer.setSuperclass(Service.class);  
 //设置代理对象需要实现的接口  
 enhancer.setInterfaces(new Class[]{IService1.class, IService2.class});  
 //通过Callback来对被代理方法进行增强  
 enhancer.setCallback(new MethodInterceptor() {  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 long startime = System.nanoTime();  
 Object result = methodProxy.invokeSuper(o, objects); //调用父类中的方法  
 System.out.println(method + "，耗时(纳秒):" + (System.nanoTime() - startime));  
 return result;  
 }  
 });  
 //创建代理对象  
 Object proxy = enhancer.create();  
 //判断代理对象是否是Service类型的  
 System.out.println("proxy instanceof Service" + (proxy instanceof Service));  
 if (proxy instanceof Service) {  
 Service service = (Service) proxy;  
 service.m1();  
 service.m2();  
 }  
 //看一下代理对象的类型  
 System.out.println(proxy.getClass());  
 //输出代理对象的父类  
 System.out.println("代理类的父类：" + proxy.getClass().getSuperclass());  
 //看一下代理类实现的接口  
 System.out.println("创建代理类实现的接口如下：");  
 for (Class<?> cs : proxy.getClass().getInterfaces()) {  
 System.out.println(cs);  
 }  
 }  
}

运行输出

proxy instanceof Servicetrue  
m1  
public void com.javacode2018.aop.demo2.CglibTest2$Service.m1()，耗时(纳秒):14219700  
m2  
public void com.javacode2018.aop.demo2.CglibTest2$Service.m2()，耗时(纳秒):62800  
class com.javacode2018.aop.demo2.CglibTest2$Service$$EnhancerByCGLIB$$80494536  
代理类的父类：class com.javacode2018.aop.demo2.CglibTest2$Service  
创建代理类实现的接口如下：  
interface com.javacode2018.aop.demo2.CglibTest2$IService1  
interface com.javacode2018.aop.demo2.CglibTest2$IService2  
interface org.springframework.cglib.proxy.Factory

输出中可以代理对象的类型是：

class com.javacode2018.aop.demo2.CglibTest2$Service$$EnhancerByCGLIB$$80494536

带有$$EnhancerByCGLIB$$字样的，在调试spring的过程中，发现有这样字样的，基本上都是cglib创建的代理对象。

上面创建的代理类相当于下面代码

public class CglibTest2$Service$$EnhancerByCGLIB$$80494536 extends Service implements IService1, IService2 {  
 @Override  
 public void m1() {  
 long starttime = System.nanoTime();  
 super.m1();  
 System.out.println("方法m1，耗时(纳秒):" + (System.nanoTime() - starttime));  
 }  
  
 @Override  
 public void m2() {  
 long starttime = System.nanoTime();  
 super.m1();  
 System.out.println("方法m1，耗时(纳秒):" + (System.nanoTime() - starttime));  
 }  
}

#### 案例3：LazyLoader的使用

LazyLoader是cglib用于实现懒加载的callback。当被增强bean的方法初次被调用时，会触发回调，之后每次再进行方法调用都是对LazyLoader第一次返回的bean调用，hibernate延迟加载有用到过这个。

看案例吧，通过案例理解容易一些。

public class LazyLoaderTest1 {  
  
 public static class UserModel {  
 private String name;  
  
 public UserModel() {  
 }  
  
 public UserModel(String name) {  
 this.name = name;  
 }  
  
 public void say() {  
 System.out.println("你好：" + name);  
 }  
 }  
  
 public static void main(String[] args) {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(UserModel.class);  
 //创建一个LazyLoader对象  
 LazyLoader lazyLoader = new LazyLoader() {  
 @Override  
 public Object loadObject() throws Exception {  
 System.out.println("调用LazyLoader.loadObject()方法");  
 return new UserModel("马士兵老师");  
 }  
 };  
 enhancer.setCallback(lazyLoader);  
 Object proxy = enhancer.create();  
 UserModel userModel = (UserModel) proxy;  
 System.out.println("第1次调用say方法");  
 userModel.say();  
 System.out.println("第1次调用say方法");  
 userModel.say();  
 }  
}

运行输出

第1次调用say方法  
调用LazyLoader.loadObject()方法  
你好：马士兵老师  
第1次调用say方法  
你好：马士兵老师

当第1次调用say方法的时候，会被cglib拦截，进入lazyLoader的loadObject内部，将这个方法的返回值作为say方法的调用者，loadObject中返回了一个马士兵老师的UserModel，cglib内部会将loadObject方法的返回值和say方法关联起来，然后缓存起来，而第2次调用say方法的时候，通过方法名去缓存中找，会直接拿到第1次返回的UserModel，所以第2次不会进入到loadObject方法中了。

将下代码拆分开来

System.out.println("第1次调用say方法");  
userModel.say();  
System.out.println("第1次调用say方法");  
userModel.say();

相当于下面的代码

System.out.println("第1次调用say方法");  
System.out.println("调用LazyLoader.loadObject()方法");  
userModel = new UserModel("马士兵老师");  
userModel.say();  
System.out.println("第1次调用say方法");  
userModel.say();

下面通过LazyLoader实现延迟加载的效果。

#### 案例4：LazyLoader实现延迟加载

博客的内容一般比较多，需要用到内容的时候，我们再去加载，下面来模拟博客内容延迟加载的效果。

public class LazyLoaderTest2 {  
 //博客信息  
 public static class BlogModel {  
 private String title;  
 //博客内容信息比较多，需要的时候再去获取  
 private BlogContentModel blogContentModel;  
  
 public BlogModel() {  
 this.title = "spring aop详解!";  
 this.blogContentModel = this.getBlogContentModel();  
 }  
  
 private BlogContentModel getBlogContentModel() {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(BlogContentModel.class);  
 enhancer.setCallback(new LazyLoader() {  
 @Override  
 public Object loadObject() throws Exception {  
 //此处模拟从数据库中获取博客内容  
 System.out.println("开始从数据库中获取博客内容.....");  
 BlogContentModel result = new BlogContentModel();  
 result.setContent("欢迎大家和我一起学些spring，我们一起成为spring高手！");  
 return result;  
 }  
 });  
 return (BlogContentModel) enhancer.create();  
 }  
 }  
  
 //表示博客内容信息  
 public static class BlogContentModel {  
 //博客内容  
 private String content;  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
 }  
  
 public static void main(String[] args) {  
 //创建博客对象  
 BlogModel blogModel = new BlogModel();  
 System.out.println(blogModel.title);  
 System.out.println("博客内容");  
 System.out.println(blogModel.blogContentModel.getContent()); //@1  
 }  
}

@1：调用blogContentModel.getContent()方法的时候，才会通过LazyLoader#loadObject方法从db中获取到博客内容信息

运行输出

spring aop详解!  
博客内容  
开始从数据库中获取博客内容.....  
欢迎大家和我一起学些spring，我们一起成为spring高手！

#### 案例5：Dispatcher

Dispatcher和LazyLoader作用很相似，区别是用Dispatcher的话每次对增强bean进行方法调用都会触发回调。

看案例代码

public class DispatcherTest1 {  
 public static class UserModel {  
 private String name;  
  
 public UserModel() {  
 }  
  
 public UserModel(String name) {  
 this.name = name;  
 }  
  
 public void say() {  
 System.out.println("你好：" + name);  
 }  
 }  
  
 public static void main(String[] args) {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(LazyLoaderTest1.UserModel.class);  
 //创建一个Dispatcher对象  
 Dispatcher dispatcher = new Dispatcher() {  
 @Override  
 public Object loadObject() throws Exception {  
 System.out.println("调用Dispatcher.loadObject()方法");  
 return new LazyLoaderTest1.UserModel("马士兵老师," + UUID.randomUUID().toString());  
 }  
 };  
 enhancer.setCallback(dispatcher);  
 Object proxy = enhancer.create();  
 LazyLoaderTest1.UserModel userModel = (LazyLoaderTest1.UserModel) proxy;  
 System.out.println("第1次调用say方法");  
 userModel.say();  
 System.out.println("第1次调用say方法");  
 userModel.say();  
 }  
}

运行输出

第1次调用say方法  
调用Dispatcher.loadObject()方法  
你好：马士兵老师,514f911e-06ac-4e3b-aee4-595f82c16a5f  
第1次调用say方法  
调用Dispatcher.loadObject()方法  
你好：马士兵老师,bc062990-bc16-4226-97e3-b1b321a03468

#### 案例6：通过Dispathcer对类扩展一些接口

下面有个UserService类，我们需要对这个类创建一个代理。

代码中还定义了一个接口：IMethodInfo，用来统计被代理类的一些方法信息，有个实现类：DefaultMethodInfo。

通过cglib创建一个代理类，父类为UserService，并且实现IMethodInfo接口，将接口IMethodInfo所有方法的转发给DefaultMethodInfo处理，代理类中的其他方法，转发给其父类UserService处理。

这个代码相当于对UserService这个类进行了增强，使其具有了IMethodInfo接口中的功能。

public class DispatcherTest2 {  
  
 public static class UserService {  
 public void add() {  
 System.out.println("新增用户");  
 }  
  
 public void update() {  
 System.out.println("更新用户信息");  
 }  
 }  
  
 //用来获取方法信息的接口  
 public interface IMethodInfo {  
 //获取方法数量  
 int methodCount();  
  
 //获取被代理的对象中方法名称列表  
 List<String> methodNames();  
 }  
  
 //IMethodInfo的默认实现  
 public static class DefaultMethodInfo implements IMethodInfo {  
  
 private Class<?> targetClass;  
  
 public DefaultMethodInfo(Class<?> targetClass) {  
 this.targetClass = targetClass;  
 }  
  
 @Override  
 public int methodCount() {  
 return targetClass.getDeclaredMethods().length;  
 }  
  
 @Override  
 public List<String> methodNames() {  
 return Arrays.stream(targetClass.getDeclaredMethods()).  
 map(Method::getName).  
 collect(Collectors.toList());  
 }  
 }  
  
 public static void main(String[] args) {  
 Class<?> targetClass = UserService.class;  
 Enhancer enhancer = new Enhancer();  
 //设置代理的父类  
 enhancer.setSuperclass(targetClass);  
 //设置代理需要实现的接口列表  
 enhancer.setInterfaces(new Class[]{IMethodInfo.class});  
 //创建一个方法统计器  
 IMethodInfo methodInfo = new DefaultMethodInfo(targetClass);  
 //创建会调用器列表，此处定义了2个，第1个用于处理UserService中所有的方法，第2个用来处理IMethodInfo接口中的方法  
 Callback[] callbacks = {  
 new MethodInterceptor() {  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 return methodProxy.invokeSuper(o, objects);  
 }  
 },  
 new Dispatcher() {  
 @Override  
 public Object loadObject() throws Exception {  
 /\*\*  
 \* 用来处理代理对象中IMethodInfo接口中的所有方法  
 \* 所以此处返回的为IMethodInfo类型的对象，  
 \* 将由这个对象来处理代理对象中IMethodInfo接口中的所有方法  
 \*/  
 return methodInfo;  
 }  
 }  
 };  
 enhancer.setCallbacks(callbacks);  
 enhancer.setCallbackFilter(new CallbackFilter() {  
 @Override  
 public int accept(Method method) {  
 //当方法在IMethodInfo中定义的时候，返回callbacks中的第二个元素  
 return method.getDeclaringClass() == IMethodInfo.class ? 1 : 0;  
 }  
 });  
  
 Object proxy = enhancer.create();  
 //代理的父类是UserService  
 UserService userService = (UserService) proxy;  
 userService.add();  
 //代理实现了IMethodInfo接口  
 IMethodInfo mf = (IMethodInfo) proxy;  
 System.out.println(mf.methodCount());  
 System.out.println(mf.methodNames());  
 }  
}

运行输出

新增用户  
2  
[add, update]

#### 案例7：cglib中的NamingPolicy接口

接口NamingPolicy表示生成代理类的名字的策略，通过Enhancer.setNamingPolicy方法设置命名策略。

默认的实现类：DefaultNamingPolicy， 具体cglib动态生成类的命名控制。

DefaultNamingPolicy中有个getTag方法。

DefaultNamingPolicy生成的代理类的类名命名规则：

被代理class name + "$$" + 使用cglib处理的class name + "ByCGLIB" + "$$" + key的hashcode

如：

com.javacode2018.aop.demo2.DispatcherTest2$UserService$$EnhancerByCGLIB$$e7ec0be5@17d10166

自定义NamingPolicy，通常会继承DefaultNamingPolicy来实现，spring中默认就提供了一个，如下

public class SpringNamingPolicy extends DefaultNamingPolicy {  
  
 public static final SpringNamingPolicy INSTANCE = new SpringNamingPolicy();  
  
 @Override  
 protected String getTag() {  
 return "BySpringCGLIB";  
 }  
  
}

案例代码

public class NamingPolicyTest {  
 public static void main(String[] args) {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(NamingPolicyTest.class);  
 enhancer.setCallback(NoOp.INSTANCE);  
 //通过Enhancer.setNamingPolicy来设置代理类的命名策略  
 enhancer.setNamingPolicy(new DefaultNamingPolicy() {  
 @Override  
 protected String getTag() {  
 return "-test-";  
 }  
 });  
 Object proxy = enhancer.create();  
 System.out.println(proxy.getClass());  
 }  
}

输出

class com.javacode2018.aop.demo2.NamingPolicyTest$$Enhancer-test-$$5946713

### Objenesis：实例化对象的一种方式

先来看一段代码，有一个有参构造函数：

public static class User {  
 private String name;  
  
 public User(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

**大家来思考一个问题：如果不使用这个有参构造函数的情况下，如何创建这个对象？**

通过反射？大家可以试试，如果不使用有参构造函数，是无法创建对象的。

cglib中提供了一个接口：Objenesis，通过这个接口可以解决上面这种问题，它专门用来创建对象，即使你没有空的构造函数，都木有问题，它不使用构造方法创建Java对象，所以即使你有空的构造方法，也是不会执行的。

用法比较简单：

@Test  
public void test1() {  
 Objenesis objenesis = new ObjenesisStd();  
 User user = objenesis.newInstance(User.class);  
 System.out.println(user);  
}

输出

User{name='null'}

大家可以在User类中加一个默认构造函数，来验证一下上面的代码会不会调用默认构造函数？

public User() {  
 System.out.println("默认构造函数");  
}

再次运行会发现，并不会调用默认构造函数。

如果需要多次创建User对象，可以写成下面方式重复利用

@Test  
public void test2() {  
 Objenesis objenesis = new ObjenesisStd();  
 ObjectInstantiator<User> userObjectInstantiator = objenesis.getInstantiatorOf(User.class);  
 User user1 = userObjectInstantiator.newInstance();  
 System.out.println(user1);  
 User user2 = userObjectInstantiator.newInstance();  
 System.out.println(user2);  
 System.out.println(user1 == user2);  
}

运行输出

User{name='null'}  
User{name='null'}  
false

代码位置

com.javacode2018.aop.demo2.CreateObjectTest

### 总结

1. 代理这2篇文章是spring aop的基础，基础牢靠了，才能走的更远，大家一定要将这2篇文章中的内容吃透，全面掌握jdk动态代理和cglib代理的使用
2. 这些知识点spring aop中全部都用到了，大家消化一下，下一篇讲解spring aop具体是如何玩的

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）

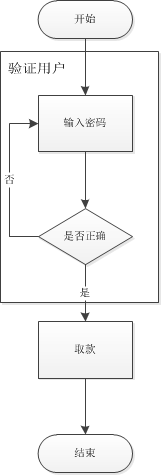
## Spring系列第31篇：Aop概念详解

### 本文主要内容

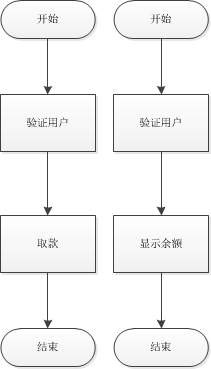
1. 什么是Aop？
2. Spring AOP中重要的一些概念详解
3. Spring AOP 硬编码实现

### 什么是AOP？

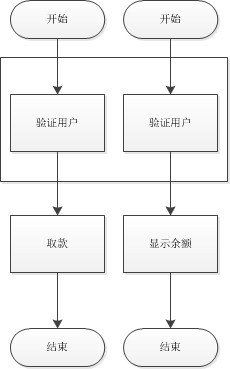
先看一下传统程序的流程，比如银行系统会有一个取款流程



我们可以把方框里的流程合为一个，另外系统还会有一个查询余额流程，我们先把这两个流程放到一起：



有没有发现，这个两者有一个相同的验证流程，我们先把它们圈起来再说下一步：

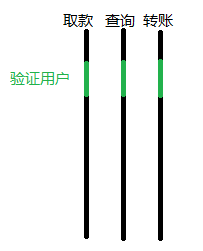


上面只是2个操作，如果有更多的操作，验证用户的功能是不是需要写很多次？

有没有想过可以把这个验证用户的代码是提取出来，不放到主流程里去呢，这就是AOP的作用了，有了AOP，你写代码时不要把这个验证用户步骤写进去，即完全不考虑验证用户，你写完之后，在另我一个地方，写好验证用户的代码，然后告诉Spring你要把这段代码加到哪几个地方，Spring就会帮你加过去，而不要你自己Copy过去，如果你有多个控制流呢，这个写代码的方法可以大大减少你的时间。

再举一个通用的例子，经常在debug的时候要打log吧，你也可以写好主要代码之后，把打log的代码写到另一个单独的地方，然后命令AOP把你的代码加过去，注意AOP不会把代码加到源文件里，但是它会正确的影响最终的机器代码。

现在大概明白了AOP了吗，我们来理一下头绪，上面那个方框像不像个平面，你可以把它当块板子，这块板子插入一些控制流程，这块板子就可以当成是AOP中的一个切面。所以AOP的本质是在一系列纵向的控制流程中，把那些相同的子流程提取成一个横向的面，这句话应该好理解吧，我们把纵向流程画成一条直线，然把相同的部分以绿色突出，如下图左，而AOP相当于把相同的地方连一条横线，如下图右，这个图没画好，大家明白意思就行。



这个验证用户这个子流程就成了一个条线，也可以理解成一个切面，这里的切面只插了两三个流程，如果其它流程也需要这个子流程，也可以插到其它地方去。

**通熟易懂的理解为：在程序中具有公共特性的某些类/某些方法上进行拦截, 在方法执行的前面/后面/执行结果返回后增加执行一些方法。**

### 先来考虑几个问题

1. aop中用什么来表示这些公共的功能？
2. aop中如何知道这些公共的功能用到哪些类的那些方法中去？
3. aop中需要将这些公共的功能用在目标方法的什么地方，前面？后面？还是其他什么地方？
4. aop底层是用什么实现的？

spring中有些概念，不是太好理解，带着这些问题，理解起来会容易很多，概念理解了，后面的路会容易很多，下面我们先来理解概念。

### Spring中AOP一些概念

#### 目标对象(target)

目标对象指将要被增强的对象，即包含主业务逻辑的类对象。

#### 连接点(JoinPoint)

连接点，程序运行的某一个点，比如执行某个方法，在Spring AOP中Join Point总是表示一个方法的执行

#### 代理对象(Proxy)

AOP中会通过代理的方式，对目标对象生成一个代理对象，代理对象中会加入需要增强功能，通过代理对象来间接的方式目标对象，起到增强目标对象的效果。

#### 通知(Advice)

需要在目标对象中增强的功能，如上面说的：业务方法前验证用户的功能、方法执行之后打印方法的执行日志。

通知中有2个重要的信息：**方法的什么地方**，**执行什么操作**，这2个信息通过通知来指定。

方法的什么地方？之前、之后、包裹目标方法、方法抛出异常后等。

如：

在方法执行之前验证用户是否有效。

在方法执行之后，打印方法的执行耗时。

在方法抛出异常后，记录异常信息发送到mq。

#### 切入点(Pointcut )

用来指定需要将通知使用到哪些地方，比如需要用在哪些类的哪些方法上，切入点就是做这个配置的。

#### 切面（Aspect）

通知（Advice）和切入点（Pointcut）的组合。切面来定义在哪些地方（Pointcut）执行什么操作（Advice）。

#### 顾问（Advisor)

Advisor 其实它就是 Pointcut 与 Advice 的组合，Advice 是要增强的逻辑，而增强的逻辑要在什么地方执行是通过Pointcut来指定的，所以 Advice 必需与 Pointcut 组合在一起，这就诞生了 Advisor 这个类，spring Aop中提供了一个Advisor接口将Pointcut 与 Advice 的组合起来。

Advisor有好几个称呼：顾问、通知器。

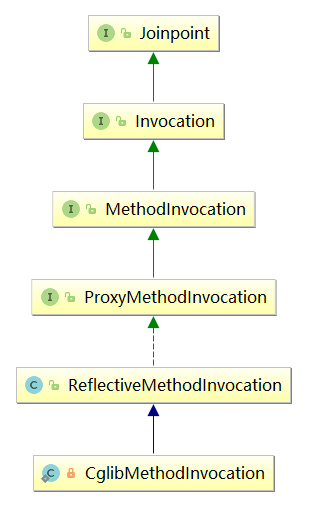
其中这4个：连接点(JoinPoint)、通知(advise)、切入点(pointcut)、顾问（advisor)，在spring中都定义了接口和类来表示这些对象，下面我们一个个来看一下。

### 连接点（JoinPoint）

#### JoinPoint接口

package org.aopalliance.intercept;  
  
public interface Joinpoint {  
  
 /\*\*  
 \* 转到拦截器链中的下一个拦截器  
 \*/  
 Object proceed() throws Throwable;  
  
 /\*\*  
 \* 返回保存当前连接点静态部分【的对象】，这里一般指被代理的目标对象  
 \*/  
 Object getThis();  
  
 /\*\*  
 \* 返回此静态连接点 一般就为当前的Method(至少目前的唯一实现是MethodInvocation,所以连接点得静态部分肯定就是本方法)  
 \*/  
 AccessibleObject getStaticPart();  
  
}

几个重要的子接口和实现类，如下：



#### Invocation接口

package org.aopalliance.intercept;  
  
/\*\*  
 \* 此接口表示程序中的调用  
 \* 调用是一个连接点，可以被拦截器拦截。  
 \*/  
public interface Invocation extends Joinpoint {  
  
 /\*\*  
 \* 将参数作为数组对象获取。可以更改此数组中的元素值以更改参数。  
 \* 通常用来获取调用目标方法的参数  
 \*/  
 Object[] getArguments();  
  
}

#### MethodInvocation接口

package org.aopalliance.intercept;  
  
import java.lang.reflect.Method;  
  
/\*\*  
 \* 方法调用的描述，在方法调用时提供给拦截器。  
 \* 方法调用是一个连接点，可以被方法拦截器拦截。  
 \*/  
public interface MethodInvocation extends Invocation {  
  
 /\*\*  
 \* 返回正在被调用得方法~~~ 返回的是当前Method对象。  
 \* 此时，效果同父类的AccessibleObject getStaticPart() 这个方法  
 \*/  
 Method getMethod();  
  
}

### 通知(Advice)

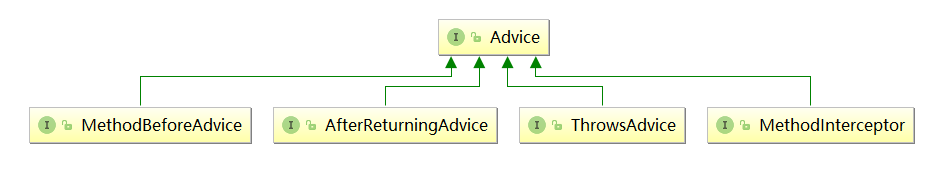
通知中用来实现被增强的逻辑，通知中有2个关注点，再强调一下：方法的什么地方，执行什么操作。

#### Advice接口

通知的顶层接口，这个接口内部没有定义任何方法。

package org.aopalliance.aop;  
  
public interface Advice {  
}

#### Advice 4个子接口



#### MethodBeforeAdvice接口

方法执行前通知，需要在目标方法执行前执行一些逻辑的，可以通过这个实现。

通俗点说：需要在目标方法执行之前增强一些逻辑，可以通过这个接口来实现。before方法：在调用给定方法之前回调。

package org.springframework.aop;  
  
public interface MethodBeforeAdvice extends BeforeAdvice {  
  
 /\*\*  
 \* 调用目标方法之前会先调用这个before方法  
 \* method：需要执行的目标方法  
 \* args：目标方法的参数  
 \* target：目标对象  
 \*/  
 void before(Method method, Object[] args, @Nullable Object target) throws Throwable;  
}

如同

public Object invoke(){  
 调用MethodBeforeAdvice#before方法  
 return 调用目标方法;  
}

#### AfterReturningAdvice接口

方法执行后通知，需要在目标方法执行之后执行增强一些逻辑的，可以通过这个实现。

**不过需要注意一点：目标方法正常执行后，才会回调这个接口，当目标方法有异常，那么这通知会被跳过。**

package org.springframework.aop;  
  
public interface AfterReturningAdvice extends AfterAdvice {  
  
 /\*\*  
 \* 目标方法执行之后会回调这个方法  
 \* method：需要执行的目标方法  
 \* args：目标方法的参数  
 \* target：目标对象  
 \*/  
 void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable;  
  
}

如同

public Object invoke(){  
 Object retVal = 调用目标方法;  
 调用AfterReturningAdvice#afterReturning方法  
 return retVal;  
}

#### ThrowsAdvice接口

package org.springframework.aop;  
  
public interface ThrowsAdvice extends AfterAdvice {  
  
}

此接口上没有任何方法，因为方法由反射调用，实现类必须实现以下形式的方法，前3个参数是可选的，最后一个参数为需要匹配的异常的类型。

void afterThrowing([Method, args, target], ThrowableSubclass);

有效方法的一些例子如下：

public void afterThrowing(Exception ex)  
public void afterThrowing(RemoteException)  
public void afterThrowing(Method method, Object[] args, Object target, Exception ex)  
public void afterThrowing(Method method, Object[] args, Object target, ServletException ex)

#### MethodInterceptor接口

方法拦截器，这个接口最强大，可以实现上面3种类型的通知，上面3种通知最终都通过适配模式将其转换为MethodInterceptor方式去执行。

package org.aopalliance.intercept;  
  
@FunctionalInterface  
public interface MethodInterceptor extends Interceptor {  
  
 /\*\*  
 \* 拦截目标方法的执行，可以在这个方法内部实现需要增强的逻辑，以及主动调用目标方法  
 \*/  
 Object invoke(MethodInvocation invocation) throws Throwable;  
  
}

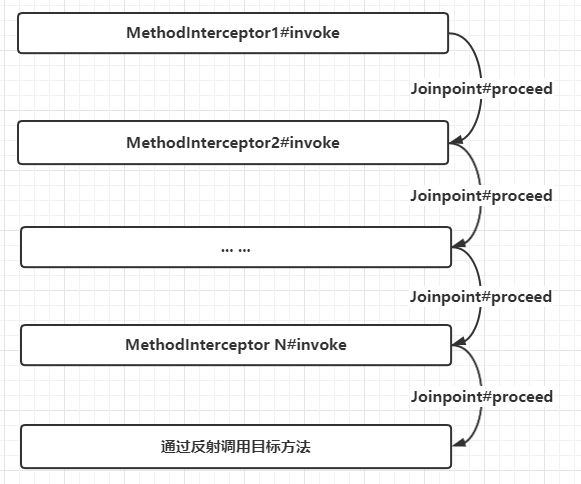
使用方式如：

public class TracingInterceptor implements MethodInterceptor {  
 Object invoke(MethodInvocation i) throws Throwable {  
 System.out.println("method "+i.getMethod()+" is called on "+ i.getThis()+" with args "+i.getArguments());  
 Object ret=i.proceed();//转到拦截器链中的下一个拦截器  
 System.out.println("method "+i.getMethod()+" returns "+ret);  
 return ret;  
 }  
}

#### 拦截器链

一个目标方法中可以添加很多Advice，这些Advice最终都会被转换为MethodInterceptor类型的方法拦截器，最终会有多个MethodInterceptor，这些MethodInterceptor会组成一个方法调用链。

Aop内部会给目标对象创建一个代理，代理对象中会放入这些MethodInterceptor会组成一个方法调用链，当调用代理对象的方法的时候，会按顺序执行这些方法调用链，一个个执行，最后会通过反射再去调用目标方法，进而对目标方法进行增强。



### 切入点(PointCut)

通知（Advice）用来指定需要增强的逻辑，但是哪些类的哪些方法中需要使用这些通知呢？这个就是通过切入点来配置的，切入点在spring中对应了一个接口

#### PointCut接口

package org.springframework.aop;  
  
public interface Pointcut {  
  
 /\*\*  
 \* 类过滤器, 可以知道哪些类需要拦截  
 \*/  
 ClassFilter getClassFilter();  
   
 /\*\*  
 \* 方法匹配器, 可以知道哪些方法需要拦截  
 \*/  
 MethodMatcher getMethodMatcher();  
   
 /\*\*  
 \* 匹配所有对象的 Pointcut，内部的2个过滤器默认都会返回true  
 \*/  
 Pointcut TRUE = TruePointcut.INSTANCE;  
  
}

#### ClassFilter接口

比较简单，用来过滤类的

@FunctionalInterface  
public interface ClassFilter {  
  
 /\*\*  
 \* 用来判断目标类型是否匹配  
 \*/  
 boolean matches(Class<?> clazz);  
  
}

#### MethodMatcher接口

用来过滤方法的。

public interface MethodMatcher {  
  
 /\*\*  
 \* 执行静态检查给定方法是否匹配  
 \* @param method 目标方法  
 \* @param targetClass 目标对象类型  
 \*/  
 boolean matches(Method method, Class<?> targetClass);  
  
 /\*\*  
 \* 是否是动态匹配，即是否每次执行目标方法的时候都去验证一下  
 \*/  
 boolean isRuntime();  
  
 /\*\*  
 \* 动态匹配验证的方法，比第一个matches方法多了一个参数args，这个参数是调用目标方法传入的参数  
 \*/  
 boolean matches(Method method, Class<?> targetClass, Object... args);  
  
  
 /\*\*  
 \* 匹配所有方法，这个内部的2个matches方法任何时候都返回true  
 \*/  
 MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;  
  
}

我估计大家看MethodMatcher还是有点晕的，为什么需要2个maches方法？什么是动态匹配？

比如下面一个类

public class UserService{  
 public void work(String userName){  
 System.out.print(userName+",开始工作了!");  
 }  
}

work方法表示当前用户的工作方法，内部可以实现一些工作的逻辑。

我们希望通过aop对这个类进行增强，调用这个方法的时候，当传入的用户名是路人的粉丝的的时候，需要先进行问候，其他用户的时候，无需问候，将这个问题的代码可以放在MethodBeforeAdvice中实现，这种情况就是当参数满足一定的条件了，才会使用这个通知，不满足的时候，通知无效，此时就可以使用上面的动态匹配来实现，MethodMatcher类中3个参数的matches方法可以用来对目标方法的参数做校验。

来看一下MethodMatcher过滤的整个过程

1.调用matches(Method method, Class<?> targetClass)方法，验证方法是否匹配  
2.isRuntime方法是否为true，如果为false，则以第一步的结果为准，否则继续向下  
3.调用matches(Method method, Class<?> targetClass, Object... args)方法继续验证，这个方法多了一个参数，可以对目标方法传入的参数进行校验。

通过上面的过程，大家可以看出来，如果isRuntime为false的时候，只需要对方法名称进行校验，当目标方法调用多次的时候，实际上第一步的验证结果是一样的，所以如果isRuntime为false的情况，可以将验证结果放在缓存中，提升效率，而spring内部就是这么做的，isRuntime为true的时候，需要每次都进行校验，效率会低一些，不过对性能的影响基本上可以忽略。

### 顾问(Advisor)

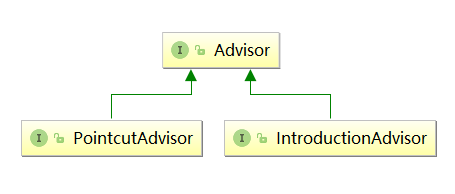
通知定义了需要做什么，切入点定义了在哪些类的哪些方法中执行通知，那么需要将他们2个组合起来才有效啊。

顾问（Advisor）就是做这个事情的

#### Advisor接口

package org.springframework.aop;  
  
import org.aopalliance.aop.Advice;  
  
/\*\*  
 \* 包含AOP通知（在joinpoint处执行的操作）和确定通知适用性的过滤器（如切入点[PointCut]）的基本接口。  
 \* 这个接口不是供Spring用户使用的，而是为了支持不同类型的建议的通用性。  
 \*/  
public interface Advisor {  
 /\*\*  
 \* 返回引用的通知  
 \*/  
 Advice getAdvice();  
  
}

上面这个接口通常不会直接使用，这个接口有2个子接口，通常我们会和这2个子接口来打交道，下面看一下这2个子接口。



#### PointcutAdvisor接口

通过名字就能看出来，这个和Pointcut有关，内部有个方法用来获取Pointcut，AOP使用到的大部分Advisor都属于这种类型的。

在目标方法中实现各种增强功能基本上都是通过PointcutAdvisor来实现的。

package org.springframework.aop;  
  
/\*\*  
 \* 切入点类型的Advisor  
 \*/  
public interface PointcutAdvisor extends Advisor {  
  
 /\*\*  
 \* 获取顾问中使用的切入点  
 \*/  
 Pointcut getPointcut();  
  
}

#### IntroductionAdvisor接口

这个接口，估计大家比较陌生，干什么的呢？

**一个Java类，没有实现A接口，在不修改Java类的情况下，使其具备A接口的功能。可以通过IntroductionAdvisor给目标类引入更多接口的功能，这个功能是不是非常牛逼。**

### 案例

上面都是一些概念，看起来比较枯燥乏味，下面来个使用硬编码的方式来用一下上面提到的一些类或者接口，加深理解。

#### 来个类

package com.javacode2018.aop.demo3;  
  
public class UserService {  
 public void work(String userName) {  
 System.out.println(userName + ",正在和马士兵老师一起学Spring Aop，欢迎大家一起来！");  
 }  
}

下面通过aop来实现一些需求，对work方法进行增强。

#### 案例1

需求：在work方法执行之前，打印一句：你好：userName

下面直接上代码，注释比较详细，就不细说了。

@Test  
public void test1() {  
 //定义目标对象  
 UserService target = new UserService();  
 //创建pointcut，用来拦截UserService中的work方法  
 Pointcut pointcut = new Pointcut() {  
 @Override  
 public ClassFilter getClassFilter() {  
 //判断是否是UserService类型的  
 return clazz -> UserService.class.isAssignableFrom(clazz);  
 }  
  
 @Override  
 public MethodMatcher getMethodMatcher() {  
 return new MethodMatcher() {  
 @Override  
 public boolean matches(Method method, Class<?> targetClass) {  
 //判断方法名称是否是work  
 return "work".equals(method.getName());  
 }  
  
 @Override  
 public boolean isRuntime() {  
 return false;  
 }  
  
 @Override  
 public boolean matches(Method method, Class<?> targetClass, Object... args) {  
 return false;  
 }  
 };  
 }  
 };  
 //创建通知，此处需要在方法之前执行操作，所以需要用到MethodBeforeAdvice类型的通知  
 MethodBeforeAdvice advice = (method, args, target1) -> System.out.println("你好:" + args[0]);  
  
 //创建Advisor，将pointcut和advice组装起来  
 DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut, advice);  
  
 //通过spring提供的代理创建工厂来创建代理  
 ProxyFactory proxyFactory = new ProxyFactory();  
 //为工厂指定目标对象  
 proxyFactory.setTarget(target);  
 //调用addAdvisor方法，为目标添加增强的功能，即添加Advisor，可以为目标添加很多个Advisor  
 proxyFactory.addAdvisor(advisor);  
 //通过工厂提供的方法来生成代理对象  
 UserService userServiceProxy = (UserService) proxyFactory.getProxy();  
  
 //调用代理的work方法  
 userServiceProxy.work("路人");  
}

运行输出

你好:路人  
路人,正在和马士兵老师一起学Spring Aop，欢迎大家一起来！

上面是采用硬编码的方式来感受一下aop的用法，大家看了上面代码之后，估计会有疑问：我晕，这么复杂？？？

如果大家有使用过spring中的aop经验，可能只需要几行代码就实现了上面的功能，的确，spring中把整个功能简化了很多，不过我们得去了解他的内部是如何实现的，然后才能走的更远。

#### 案例2

需求：统计一下work方法的耗时，将耗时输出

@Test  
public void test2() {  
 //定义目标对象  
 UserService target = new UserService();  
 //创建pointcut，用来拦截UserService中的work方法  
 Pointcut pointcut = new Pointcut() {  
 @Override  
 public ClassFilter getClassFilter() {  
 //判断是否是UserService类型的  
 return clazz -> UserService.class.isAssignableFrom(clazz);  
 }  
  
 @Override  
 public MethodMatcher getMethodMatcher() {  
 return new MethodMatcher() {  
 @Override  
 public boolean matches(Method method, Class<?> targetClass) {  
 //判断方法名称是否是work  
 return "work".equals(method.getName());  
 }  
  
 @Override  
 public boolean isRuntime() {  
 return false;  
 }  
  
 @Override  
 public boolean matches(Method method, Class<?> targetClass, Object... args) {  
 return false;  
 }  
 };  
 }  
 };  
 //创建通知，需要拦截方法的执行，所以需要用到MethodInterceptor类型的通知  
 MethodInterceptor advice = new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 System.out.println("准备调用:" + invocation.getMethod());  
 long starTime = System.nanoTime();  
 Object result = invocation.proceed();  
 long endTime = System.nanoTime();  
 System.out.println(invocation.getMethod() + "，调用结束！");  
 System.out.println("耗时(纳秒):" + (endTime - starTime));  
 return result;  
 }  
 };  
  
 //创建Advisor，将pointcut和advice组装起来  
 DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut, advice);  
  
 //通过spring提供的代理创建工厂来创建代理  
 ProxyFactory proxyFactory = new ProxyFactory();  
 //为工厂指定目标对象  
 proxyFactory.setTarget(target);  
 //调用addAdvisor方法，为目标添加增强的功能，即添加Advisor，可以为目标添加很多个Advisor  
 proxyFactory.addAdvisor(advisor);  
 //通过工厂提供的方法来生成代理对象  
 UserService userServiceProxy = (UserService) proxyFactory.getProxy();  
  
 //调用代理的work方法  
 userServiceProxy.work("路人");  
}

运行输出

准备调用:public void com.javacode2018.aop.demo3.UserService.work(java.lang.String)  
路人,正在和马士兵老师一起学Spring Aop，欢迎大家一起来！  
public void com.javacode2018.aop.demo3.UserService.work(java.lang.String)，调用结束！  
耗时(纳秒):9526200

#### 案例3

需求：userName中包含“粉丝”关键字，输出一句：感谢您一路的支持

此处需要用到 MethodMatcher 中的动态匹配了，通过参数来进行判断。

重点在于Pointcut中的getMethodMatcher方法，返回的MethodMatcher，@1必须返回true，此时才会进入到@2中对参数进行校验。

代码如下：

@Test  
public void test2() {  
 //定义目标对象  
 UserService target = new UserService();  
 //创建pointcut，用来拦截UserService中的work方法  
 Pointcut pointcut = new Pointcut() {  
 @Override  
 public ClassFilter getClassFilter() {  
 //判断是否是UserService类型的  
 return clazz -> UserService.class.isAssignableFrom(clazz);  
 }  
  
 @Override  
 public MethodMatcher getMethodMatcher() {  
 return new MethodMatcher() {  
 @Override  
 public boolean matches(Method method, Class<?> targetClass) {  
 //判断方法名称是否是work  
 return "work".equals(method.getName());  
 }  
  
 @Override  
 public boolean isRuntime() {  
 return true; // @1：注意这个地方要返回true  
 }  
  
 @Override  
 public boolean matches(Method method, Class<?> targetClass, Object... args) {  
 // @2：isRuntime为true的时候，会执行这个方法  
 if (Objects.nonNull(args) && args.length == 1) {  
 String userName = (String) args[0];  
 return userName.contains("粉丝");  
 }  
 return false;  
 }  
 };  
 }  
 };  
 //创建通知，此处需要在方法之前执行操作，所以需要用到MethodBeforeAdvice类型的通知  
 MethodBeforeAdvice advice = (method, args, target1) -> System.out.println("感谢您一路的支持!");  
  
 //创建Advisor，将pointcut和advice组装起来  
 DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut, advice);  
  
 //通过spring提供的代理创建工厂来创建代理  
 ProxyFactory proxyFactory = new ProxyFactory();  
 //为工厂指定目标对象  
 proxyFactory.setTarget(target);  
 //调用addAdvisor方法，为目标添加增强的功能，即添加Advisor，可以为目标添加很多个Advisor  
 proxyFactory.addAdvisor(advisor);  
 //通过工厂提供的方法来生成代理对象  
 UserService userServiceProxy = (UserService) proxyFactory.getProxy();  
  
 //调用代理的work方法  
 userServiceProxy.work("粉丝：A");  
}

运行输出

感谢您一路的支持!  
粉丝：A,正在和马士兵老师一起学Spring Aop，欢迎大家一起来！

#### 本文案例代码入口

com.javacode2018.aop.demo3.Test3

**上面的一些案例中都用到了ProxyFactory这个类，内部将各种对象进行组装，然后创建代理对象，ProxyFactory这块关联的的东西挺多的，下一篇文章将详说这块的东西，是非常重要的内容。**

### 课后问题

对上面案例进行改造，实现下面需求：

1. work方法执行之后，打印一句：再见：userName
2. 在work方法中抛出一个异常，然后通过aop中的ThrowsAdvice类型的通知来拦截这个异常信息，然后将异常错误信息打印出来

欢迎大家留言。

### 总结

这篇文章中主要介绍了一些概念性的东西，不是太好理解，建议大家多看几遍，有关的类和接口大家也去看看源码，理解起来也会容易一些，也欢迎大家留言一起交流这块的东西。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理

## Spring系列第32篇：AOP核心源码、原理详解

**本文目标：带领大家阅读aop的源码，深入理解aop的原理，内容有点长，消化需要大概一周时间，做好准备**

阅读本文之前，需要先掌握下面3篇文章内容，不然会比较吃力。

1. Spring系列第15篇：代理详解（java动态代理&CGLIB代理)
2. Spring系列第30篇：jdk动态代理和cglib代理
3. Spring系列第31篇：Aop概念详解

### 本文主要分4部分

1. Aop原理介绍
2. 介绍aop相关的一些类
3. 通过源码详解aop代理的创建过程
4. 通过源码详解aop代理的调用过程
5. Aop代理一些特性的使用案例

### Spring AOP原理

原理比较简单，主要就是使用jdk动态代理和cglib代理来创建代理对象，通过代理对象来访问目标对象，而代理对象中融入了增强的代码，最终起到对目标对象增强的效果。

### aop相关的一些类

1. 连接点（JoinPoint）相关类
2. 通知（Advice）相关的类
3. 切入点（Pointcut）相关的类
4. 切面（Advisor）相关的类

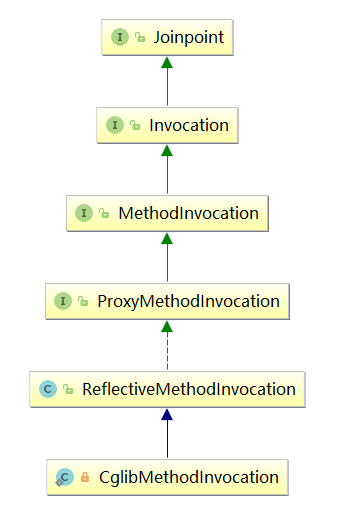
### 连接点（JoinPoint）相关类

#### JoinPoint接口

这个接口表示一个通用的运行时连接点（在AOP术语中）

package org.aopalliance.intercept;  
  
public interface Joinpoint {  
  
 /\*\*  
 \* 转到拦截器链中的下一个拦截器  
 \*/  
 Object proceed() throws Throwable;  
  
 /\*\*  
 \* 返回保存当前连接点静态部分【的对象】，这里一般指被代理的目标对象  
 \*/  
 Object getThis();  
  
 /\*\*  
 \* 返回此静态连接点 一般就为当前的Method(至少目前的唯一实现是MethodInvocation,所以连接点得静态部分肯定就是本方法)  
 \*/  
 AccessibleObject getStaticPart();  
  
}

几个重要的子接口和实现类，如下：



#### Invocation接口

此接口表示程序中的调用，调用是一个连接点，可以被拦截器拦截。

package org.aopalliance.intercept;  
  
/\*\*  
 \* 此接口表示程序中的调用  
 \* 调用是一个连接点，可以被拦截器拦截。  
 \*/  
public interface Invocation extends Joinpoint {  
  
 /\*\*  
 \* 将参数作为数组对象获取，可以更改此数组中的元素值以更改参数。  
 \* 通常用来获取调用目标方法的参数  
 \*/  
 Object[] getArguments();  
}

#### MethodInvocation接口

用来表示连接点中方法的调用，可以获取调用过程中的目标方法。

package org.aopalliance.intercept;  
  
import java.lang.reflect.Method;  
  
/\*\*  
 \* 方法调用的描述，在方法调用时提供给拦截器。  
 \* 方法调用是一个连接点，可以被方法拦截器拦截。  
 \*/  
public interface MethodInvocation extends Invocation {  
  
 /\*\*  
 \* 返回正在被调用得方法~~~ 返回的是当前Method对象。  
 \* 此时，效果同父类的AccessibleObject getStaticPart() 这个方法  
 \*/  
 Method getMethod();  
  
}

#### ProxyMethodInvocation接口

表示代理方法的调用

public interface ProxyMethodInvocation extends MethodInvocation {  
  
 /\*\*  
 \* 获取被调用的代理对象  
 \*/  
 Object getProxy();  
  
 /\*\*  
 \* 克隆一个方法调用器MethodInvocation  
 \*/  
 MethodInvocation invocableClone();  
  
 /\*\*  
 \* 克隆一个方法调用器MethodInvocation，并为方法调用器指定参数  
 \*/  
 MethodInvocation invocableClone(Object... arguments);  
  
 /\*\*  
 \* 设置要用于此链中任何通知的后续调用的参数。  
 \*/  
 void setArguments(Object... arguments);  
  
 /\*\*  
 \* 添加一些扩展用户属性，这些属性不在AOP框架内使用。它们只是作为调用对象的一部分保留，用于特殊的拦截器。  
 \*/  
 void setUserAttribute(String key, @Nullable Object value);  
  
 /\*\*  
 \* 根据key获取对应的用户属性  
 \*/  
 @Nullable  
 Object getUserAttribute(String key);  
  
}

通俗点理解：连接点表示方法的调用过程，内部包含了方法调用过程中的所有信息，比如被调用的方法、目标、代理对象、执行拦截器链等信息。

上面定义都是一些接口，最终有2个实现。

#### ReflectiveMethodInvocation

当代理对象是采用jdk动态代理创建的，通过代理对象来访问目标对象的方法的时，最终过程是由ReflectiveMethodInvocation来处理的，内部会通过递归调用方法拦截器，最终会调用到目标方法。

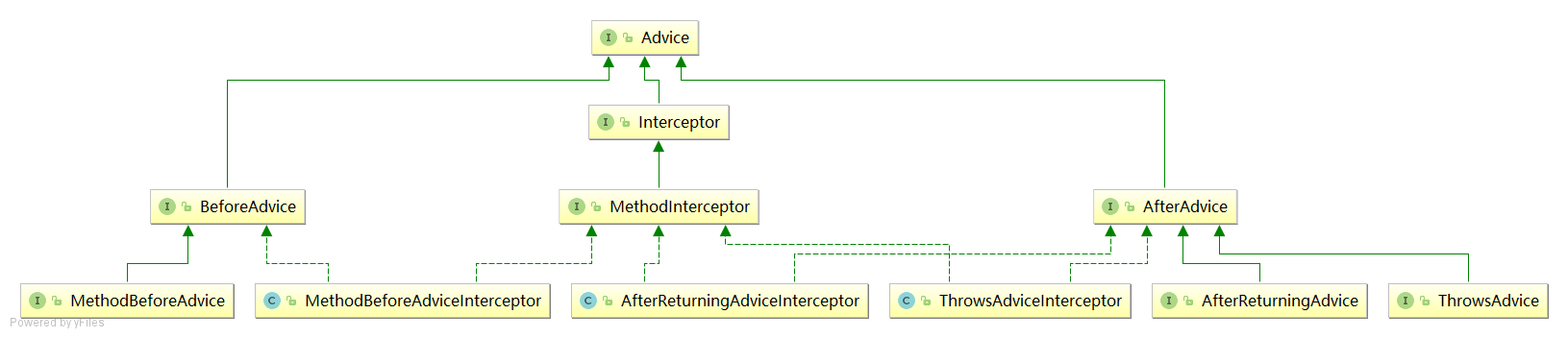
#### CglibMethodInvocation

功能和上面的类似，当代理对象是采用cglib创建的，通过代理对象来访问目标对象的方法的时，最终过程是由CglibMethodInvocation来处理的，内部会通过递归调用方法拦截器，最终会调用到目标方法。

这2个类源码稍后详解。

### 通知相关的类

通知用来定义需要增强的逻辑。



#### Advice接口

通知的底层接口

package org.aopalliance.aop;  
public interface Advice {  
}

#### BeforeAdvice接口

方法前置通知，内部空的

package org.springframework.aop;  
  
public interface BeforeAdvice extends Advice {  
}

#### Interceptor接口

此接口表示通用拦截器

package org.aopalliance.intercept;  
  
public interface Interceptor extends Advice {  
}

#### MethodInterceptor接口

方法拦截器，所有的通知均需要转换为MethodInterceptor类型的，最终多个MethodInterceptor组成一个方法拦截器连。

package org.aopalliance.intercept;  
  
@FunctionalInterface  
public interface MethodInterceptor extends Interceptor {  
 /\*\*  
 \* 拦截目标方法的执行，可以在这个方法内部实现需要增强的逻辑，以及主动调用目标方法  
 \*/  
 Object invoke(MethodInvocation invocation) throws Throwable;  
}

#### AfterAdvice接口

后置通知的公共标记接口

package org.springframework.aop;  
  
public interface AfterAdvice extends Advice {  
}

#### MethodBeforeAdvice接口

方法执行前通知，需要在目标方法执行前执行一些逻辑的，可以通过这个实现。

通俗点说：需要在目标方法执行之前增强一些逻辑，可以通过这个接口来实现。before方法：在调用给定方法之前回调。

package org.springframework.aop;  
  
public interface MethodBeforeAdvice extends BeforeAdvice {  
  
 /\*\*  
 \* 调用目标方法之前会先调用这个before方法  
 \* method：需要执行的目标方法  
 \* args：目标方法的参数  
 \* target：目标对象  
 \*/  
 void before(Method method, Object[] args, @Nullable Object target) throws Throwable;  
}

如同

public Object invoke(){  
 调用MethodBeforeAdvice#before方法  
 return 调用目标方法;  
}

#### AfterReturningAdvice接口

方法执行后通知，需要在目标方法执行之后执行增强一些逻辑的，可以通过这个实现。

**不过需要注意一点：目标方法正常执行后，才会回调这个接口，当目标方法有异常，那么这通知会被跳过。**

package org.springframework.aop;  
  
public interface AfterReturningAdvice extends AfterAdvice {  
  
 /\*\*  
 \* 目标方法执行之后会回调这个方法  
 \* method：需要执行的目标方法  
 \* args：目标方法的参数  
 \* target：目标对象  
 \*/  
 void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable;  
  
}

如同

public Object invoke(){  
 Object retVal = 调用目标方法;  
 调用AfterReturningAdvice#afterReturning方法  
 return retVal;  
}

#### ThrowsAdvice接口

package org.springframework.aop;  
  
public interface ThrowsAdvice extends AfterAdvice {  
  
}

此接口上没有任何方法，因为方法由反射调用，实现类必须实现以下形式的方法，前3个参数是可选的，最后一个参数为需要匹配的异常的类型。

void afterThrowing([Method, args, target], ThrowableSubclass);

有效方法的一些例子如下：

public void afterThrowing(Exception ex)  
public void afterThrowing(RemoteException)  
public void afterThrowing(Method method, Object[] args, Object target, Exception ex)  
public void afterThrowing(Method method, Object[] args, Object target, ServletException ex)

#### 通知包装器

**负责将各种非MethodInterceptor类型的通知(Advice)包装为MethodInterceptor类型。**

**刚才有说过：Aop中所有的Advice最终都会转换为MethodInterceptor类型的，组成一个方法调用链，然后执行**

3个包装器类

* MethodBeforeAdviceInterceptor
* AfterReturningAdviceInterceptor
* ThrowsAdviceInterceptor

#### MethodBeforeAdviceInterceptor类

这个类实现了MethodInterceptor接口，负责将MethodBeforeAdvice方法前置通知包装为MethodInterceptor类型，创建这个类型的对象的时候需要传递一个MethodBeforeAdvice类型的参数，重点是invoke方法

package org.springframework.aop.framework.adapter;  
  
@SuppressWarnings("serial")  
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, BeforeAdvice, Serializable {  
  
 private final MethodBeforeAdvice advice;  
  
 public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //负责调用前置通知的方法  
 this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());  
 //继续执行方法调用链  
 return mi.proceed();  
 }  
  
}

#### AfterReturningAdviceInterceptor类

这个类实现了MethodInterceptor接口，负责将AfterReturningAdvice方法后置通知包装为MethodInterceptor类型，创建这个类型的对象的时候需要传递一个AfterReturningAdvice类型的参数，重点是invoke方法

public class AfterReturningAdviceInterceptor implements MethodInterceptor, AfterAdvice, Serializable {  
  
 private final AfterReturningAdvice advice;  
  
 public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //先执行方法调用链,可以获取目标方法的执行结果  
 Object retVal = mi.proceed();  
 //执行后置通知  
 this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.getThis());  
 //返回结果  
 return retVal;  
 }  
  
}

#### ThrowsAdviceInterceptor类

这个类实现了MethodInterceptor接口，负责将ThrowsAdvice异常通知包装为MethodInterceptor类型，创建这个类型的对象的时候需要传递一个Object类型的参数，通常这个参数是ThrowsAdvice类型的，重点是invoke方法

package org.springframework.aop.framework.adapter;  
  
public class ThrowsAdviceInterceptor implements MethodInterceptor, AfterAdvice {  
  
 private static final String AFTER\_THROWING = "afterThrowing";  
   
 private final Object throwsAdvice;  
  
 //创建ThrowsAdviceInterceptor  
 public ThrowsAdviceInterceptor(Object throwsAdvice) {  
 Assert.notNull(throwsAdvice, "Advice must not be null");  
 this.throwsAdvice = throwsAdvice;  
 //获取异常通知中定义的所有方法（public、默认的、protected、private）  
 Method[] methods = throwsAdvice.getClass().getMethods();  
 //轮询methods  
 for (Method method : methods) {  
 //方法名称为afterThrowing && 方法参数为1或者4  
 if (method.getName().equals(AFTER\_THROWING) &&  
 (method.getParameterCount() == 1 || method.getParameterCount() == 4)) {  
 //获取方法的最后一个参数类型  
 Class<?> throwableParam = method.getParameterTypes()[method.getParameterCount() - 1];  
 //判断方法参数类型是不是Throwable类型的  
 if (Throwable.class.isAssignableFrom(throwableParam)) {  
 // 缓存异常处理方法到map中（异常类型->异常处理方法）  
 this.exceptionHandlerMap.put(throwableParam, method);  
 }  
 }  
 }  
 //如果exceptionHandlerMap，抛出异常，所以最少要有一个异常处理方法  
 if (this.exceptionHandlerMap.isEmpty()) {  
 throw new IllegalArgumentException(  
 "At least one handler method must be found in class [" + throwsAdvice.getClass() + "]");  
 }  
 }  
  
  
 /\*\*  
 \* 获取异常通知中自定义的处理异常方法的数量  
 \*/  
 public int getHandlerMethodCount() {  
 return this.exceptionHandlerMap.size();  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //调用通知链  
 return mi.proceed();  
 }  
 catch (Throwable ex) {  
 //获取异常通知中自定义的处理异常的方法  
 Method handlerMethod = getExceptionHandler(ex);  
 //当处理的方法不为空  
 if (handlerMethod != null) {  
 //调用异常处理方法  
 invokeHandlerMethod(mi, ex, handlerMethod);  
 }  
 //继续向外抛出异常  
 throw ex; //@1  
 }  
 }  
  
 /\*\*  
 \* 获取throwsAdvice中处理exception参数指定的异常的方法  
 \*/  
 @Nullable  
 private Method getExceptionHandler(Throwable exception) {  
 //获取异常类型  
 Class<?> exceptionClass = exception.getClass();  
 //从缓存中获取异常类型对应的方法  
 Method handler = this.exceptionHandlerMap.get(exceptionClass);  
 //来一个循环，查询处理方法，循环条件：方法为空 && 异常类型!=Throwable  
 while (handler == null && exceptionClass != Throwable.class) {  
 //获取异常的父类型  
 exceptionClass = exceptionClass.getSuperclass();  
 //从缓存中查找异常对应的处理方法  
 handler = this.exceptionHandlerMap.get(exceptionClass);  
 }  
 //将查找结果返回  
 return handler;  
 }  
  
 //通过反射调用异常通知中的异常方法  
 private void invokeHandlerMethod(MethodInvocation mi, Throwable ex, Method method) throws Throwable {  
 //构建方法请求参数  
 Object[] handlerArgs;  
 //若只有1个参数，参数为：异常对象  
 if (method.getParameterCount() == 1) {  
 handlerArgs = new Object[] {ex};  
 }  
 else {  
 //4个参数（方法、方法请求参数、目标对象、异常对象）  
 handlerArgs = new Object[] {mi.getMethod(), mi.getArguments(), mi.getThis(), ex};  
 }  
 try {  
 //通过反射调用异常通知中的方法  
 method.invoke(this.throwsAdvice, handlerArgs);  
 }  
 catch (InvocationTargetException targetEx) {  
 throw targetEx.getTargetException();  
 }  
 }  
  
}

从上面可以看出，异常通知，自定义处理异常的方法有几个特点

1. 方法名称必须为afterThrowing
2. 方法参数必须1个或4个，最后一个参数是Throwable类型或其子类型
3. 可以在异常处理中记录一些异常信息，这个还是比较有用的，但是注意一点目标方法抛出的异常最后还是会向外继续抛出@1

**光讲源码，大家看着枯燥乏味，来点案例。**

先来一个类，用来模拟用户资金操作：充值、提现、查询资金余额；提现的时候余额不足的时候，会抛出异常。

package com.javacode2018.aop.demo4;  
  
//模拟资金操作  
public class FundsService {  
 //账户余额  
 private double balance = 1000;  
  
 //模拟提现  
 double recharge(String userName, double price) {  
 System.out.println(String.format("%s提现%s", userName, price));  
 balance += price;  
 return balance;  
 }  
  
 //模拟提现  
 double cashOut(String userName, double price) {  
 if (balance < price) {  
 throw new RuntimeException("余额不足!");  
 }  
 System.out.println(String.format("%s提现%s", userName, price));  
 balance -= price;  
 return balance;  
 }  
  
 //获取余额  
 double getBalance(String userName) {  
 return balance;  
 }  
}

#### 案例1：前置通知拦截非法访问

资金操作的所有方法都需要验证用户名，当用户名不是“路人”的时候，直接抛出非法访问异常。

package com.javacode2018.aop.demo4;  
  
import org.junit.Test;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class AopTest4 {  
  
 @Test  
 public void test1() {  
 //代理工厂  
 ProxyFactory proxyFactory = new ProxyFactory(new FundsService());  
 //添加一个方法前置通知，判断用户名不是“路人”的时候，抛出非法访问异常  
 proxyFactory.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 String userName = (String) args[0];  
 //如果不是路人的时候，抛出非法访问异常  
 if (!"路人".equals(userName)) {  
 throw new RuntimeException(String.format("[%s]非法访问!", userName));  
 }  
 }  
 });  
 //通过代理工厂创建代理  
 FundsService proxy = (FundsService) proxyFactory.getProxy();  
 //调用代理的方法  
 proxy.recharge("路人", 100);  
 proxy.recharge("张学友", 100);  
 }  
}

运行输出

路人提现100.0  
  
java.lang.RuntimeException: [张学友]非法访问!  
  
 at com.javacode2018.aop.demo4.AopTest4$1.before(AopTest4.java:25)  
 at org.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor.invoke(MethodBeforeAdviceInterceptor.java:55)

#### 案例2：通过异常通知记录异常

通过异常通知来捕获所有方法的运行，发现异常之后，通知开发修复bug。

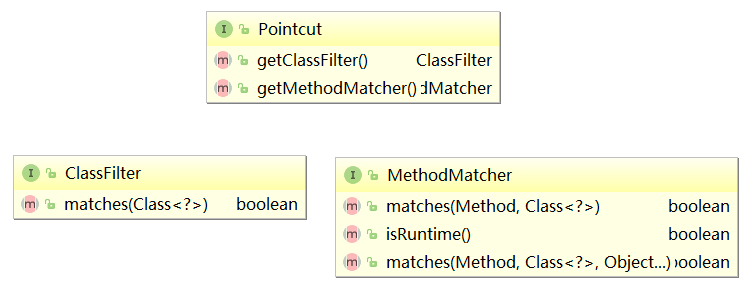
public static class SendMsgThrowsAdvice implements ThrowsAdvice {  
 //注意方法名称必须为afterThrowing  
 public void afterThrowing(Method method, Object[] args, Object target, RuntimeException e) {  
 //监控到异常后发送消息通知开发者  
 System.out.println("异常警报：");  
 System.out.println(String.format("method:[%s]，args:[%s]", method.toGenericString(), Arrays.stream(args).collect(Collectors.toList())));  
 System.out.println(e.getMessage());  
 System.out.println("请尽快修复bug！");  
 }  
}  
  
@Test  
public void test2() {  
 //代理工厂  
 ProxyFactory proxyFactory = new ProxyFactory(new FundsService());  
 //添加一个异常通知，发现异常之后发送消息给开发者尽快修复bug  
 proxyFactory.addAdvice(new SendMsgThrowsAdvice());  
 //通过代理工厂创建代理  
 FundsService proxy = (FundsService) proxyFactory.getProxy();  
 //调用代理的方法  
 proxy.cashOut("路人", 2000);  
}

运行输出

异常警报：  
method:[double com.javacode2018.aop.demo4.FundsService.cashOut(java.lang.String,double)]，args:[[路人, 2000.0]]  
余额不足!  
请尽快修复bug！  
  
java.lang.RuntimeException: 余额不足!  
  
 at com.javacode2018.aop.demo4.FundsService.cashOut(FundsService.java:18)

### 切入点(PointCut)相关类

通知（Advice）用来指定需要增强的逻辑，但是哪些类的哪些方法中需要使用这些通知呢？这个就是通过切入点来配置的。



#### PointCut接口

package org.springframework.aop;  
  
public interface Pointcut {  
  
 /\*\*  
 \* 类过滤器, 可以知道哪些类需要拦截  
 \*/  
 ClassFilter getClassFilter();  
   
 /\*\*  
 \* 方法匹配器, 可以知道哪些方法需要拦截  
 \*/  
 MethodMatcher getMethodMatcher();  
   
 /\*\*  
 \* 匹配所有对象的 Pointcut，内部的2个过滤器默认都会返回true  
 \*/  
 Pointcut TRUE = TruePointcut.INSTANCE;  
  
}

#### ClassFilter接口

类过滤器。

@FunctionalInterface  
public interface ClassFilter {  
  
 /\*\*  
 \* 用来判断目标类型是否匹配  
 \*/  
 boolean matches(Class<?> clazz);  
  
}

#### MethodMatcher接口

方法过滤器。

public interface MethodMatcher {  
  
 /\*\*  
 \* 执行静态检查给定方法是否匹配  
 \* @param method 目标方法  
 \* @param targetClass 目标对象类型  
 \*/  
 boolean matches(Method method, Class<?> targetClass);  
  
 /\*\*  
 \* 是否是动态匹配，即是否每次执行目标方法的时候都去验证一下  
 \*/  
 boolean isRuntime();  
  
 /\*\*  
 \* 动态匹配验证的方法，比第一个matches方法多了一个参数args，这个参数是调用目标方法传入的参数  
 \*/  
 boolean matches(Method method, Class<?> targetClass, Object... args);  
  
  
 /\*\*  
 \* 匹配所有方法，这个内部的2个matches方法任何时候都返回true  
 \*/  
 MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;  
  
}

我估计大家看MethodMatcher还是有点晕的，为什么需要2个maches方法？什么是动态匹配？

比如下面一个类

public class UserService{  
 public void work(String userName){  
 System.out.print(userName+",开始工作了!");  
 }  
}

work方法表示当前用户的工作方法，内部可以实现一些工作的逻辑。

我们希望通过aop对这个类进行增强，调用这个方法的时候，当传入的用户名是路人的粉丝的的时候，需要先进行问候，其他用户的时候，无需问候，将这个问题的代码可以放在MethodBeforeAdvice中实现，这种情况就是当参数满足一定的条件了，才会使用这个通知，不满足的时候，通知无效，此时就可以使用上面的动态匹配来实现，MethodMatcher类中3个参数的matches方法可以用来对目标方法的参数做校验。

来看一下MethodMatcher过滤的整个过程

1.调用matches(Method method, Class<?> targetClass)方法，验证方法是否匹配  
2.isRuntime方法是否为true，如果为false，则以第一步的结果为准，否则继续向下  
3.调用matches(Method method, Class<?> targetClass, Object... args)方法继续验证，这个方法多了一个参数，可以对目标方法传入的参数进行校验。

通过上面的过程，大家可以看出来，如果isRuntime为false的时候，只需要对方法名称进行校验，当目标方法调用多次的时候，实际上第一步的验证结果是一样的，所以如果isRuntime为false的情况，可以将验证结果放在缓存中，提升效率，而spring内部就是这么做的，isRuntime为false的时候，需要每次都进行校验，效率会低一些，不过对性能的影响基本上可以忽略。

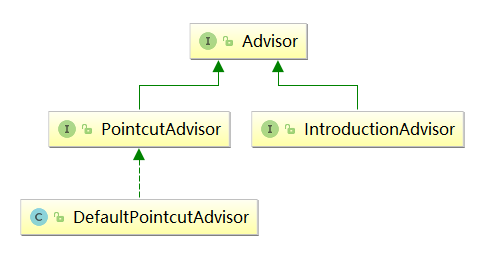
### 顾问(Advisor)

通知定义了需要做什么，切入点定义了在哪些类的哪些方法中执行通知，那么需要将他们2个组合起来才有效啊。

顾问（Advisor）就是做这个事情的。

在spring aop中，你可以将advisor理解为切面，切面中通常有2个关键信息：

1. 需要增强的目标方法列表，这个通过切入点(Pointcut)来指定
2. 需要在目标方法中增强的逻辑，这个通过(Advice)通知来指定



#### Advisor接口

package org.springframework.aop;  
  
import org.aopalliance.aop.Advice;  
  
/\*\*  
 \* 包含AOP通知（在joinpoint处执行的操作）和确定通知适用性的过滤器（如切入点[PointCut]）的基本接口。  
 \* 这个接口不是供Spring用户使用的，而是为了支持不同类型的建议的通用性。  
 \*/  
public interface Advisor {  
 /\*\*  
 \* 返回引用的通知  
 \*/  
 Advice getAdvice();  
  
}

上面这个接口通常不会直接使用，这个接口有2个子接口，通常我们会和这2个子接口来打交道，下面看一下这2个子接口。

#### PointcutAdvisor接口

通过名字就能看出来，这个和Pointcut有关，内部有个方法用来获取Pointcut，AOP使用到的大部分Advisor都属于这种类型的。

在目标方法中实现各种增强功能基本上都是通过PointcutAdvisor来实现的。

package org.springframework.aop;  
  
/\*\*  
 \* 切入点类型的Advisor  
 \*/  
public interface PointcutAdvisor extends Advisor {  
  
 /\*\*  
 \* 获取顾问中使用的切入点  
 \*/  
 Pointcut getPointcut();  
  
}

#### DefaultPointcutAdvisor类

PointcutAdvisor的默认实现，这是最常用的Advisor实现，它可以用于任何Pointcut和Advice类型，代码相当简单，里面定义了2个属性：pointcut和advisor，由使用者指定。

#### IntroductionAdvisor接口

这个接口，估计大家比较陌生，干什么的呢？

**一个Java类，没有实现A接口，在不修改Java类的情况下，使其具备A接口的功能。可以通过IntroductionAdvisor给目标类引入更多接口的功能，这个功能是不是非常牛逼。**

下面开始2个重点工作。

* 通过源码介绍aop中代理创建过程
* 通过源码介绍代理方法的调用执行过程

### 代理创建过程源码解析

#### 先看一段代码

//代理工厂  
ProxyFactory proxyFactory = new ProxyFactory(new FundsService());  
//添加一个方法前置通知，判断用户名不是“路人”的时候，抛出非法访问异常  
proxyFactory.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 String userName = (String) args[0];  
 //如果不是路人的时候，抛出非法访问异常  
 if (!"路人".equals(userName)) {  
 throw new RuntimeException(String.format("[%s]非法访问!", userName));  
 }  
 }  
});  
//通过代理工厂创建代理  
FundsService proxy = (FundsService) proxyFactory.getProxy();

我们将上面代码拆分一下，变成下面这样

//1.创建代理所需参数配置（如：采用什么方式的代理、通知列表等）  
AdvisedSupport advisedSupport = new AdvisedSupport();  
//如：添加一个前置通知  
advisedSupport.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 String userName = (String) args[0];  
 //如果不是路人的时候，抛出非法访问异常  
 if (!"路人".equals(userName)) {  
 throw new RuntimeException(String.format("[%s]非法访问!", userName));  
 }  
 }  
});  
//设置被代理的目标对象  
FundsService target = new FundsService();  
advisedSupport.setTarget(target);  
  
//2.根据配置信息获取AopProxy对象，AopProxy用来负责创建最终的代理对象  
// AopProxy接口有2个实现类（JDK动态代理、cglib代理）  
// 具体最终会使用哪种方式，需要根据AdvisedSupport中指定的参数来判断  
// 创建AopProxy使用了简单工厂模式  
AopProxyFactory aopProxyFactory = new DefaultAopProxyFactory();  
//通过AopProxy工厂获取AopProxy对象  
AopProxy aopProxy = aopProxyFactory.createAopProxy(advisedSupport);  
  
//3.通过AopProxy创建代理对象  
Object proxy = aopProxy.getProxy();

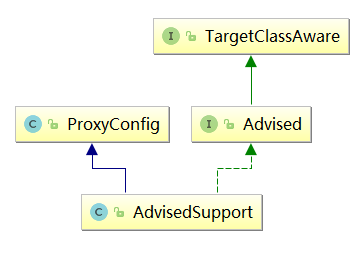
从上面可以看出创建代理有3个步骤。

#### 创建代理3大步骤

1. 创建代理所需参数配置
2. 根据代理参数获取AopProxy对象
3. 通过AopProxy获取代理对象

#### 创建代理所需参数配置

创建代理所需参数配置主要是通过AdvisedSupport这个类来做的，看一下类图，下面一个个来介绍。



#### 根据代理参数获取AopProxy对象

##### TargetClassAware接口

比较简单的一个接口，定义了一个方法，用来获取目标对象类型。

所谓目标对象：就是被代理对象，比如上面的fundsService对象。

package org.springframework.aop;  
  
public interface TargetClassAware {  
 @Nullable  
 Class<?> getTargetClass();  
}

##### ProxyConfig类

这个类比较关键了，代理配置类，内部包含了创建代理时需要配置的各种参数。

package org.springframework.aop.framework;  
  
/\*\*  
 \* 对外提供统一的代理参数配置类，以确保所有代理创建程序具有一致的属性  
 \*/  
public class ProxyConfig implements Serializable {  
  
 // 标记是否直接对目标类进行代理，而不是通过接口产生代理  
 private boolean proxyTargetClass = false;  
  
 // 标记是否对代理进行优化。启动优化通常意味着在代理对象被创建后，增强的修改将不会生效，因此默认值为false。  
 // 如果exposeProxy设置为true，即使optimize为true也会被忽略。  
 private boolean optimize = false;  
   
 // 标记是否需要阻止通过该配置创建的代理对象转换为Advised类型，默认值为false，表示代理对象可以被转换为Advised类型  
 boolean opaque = false;  
  
 // 标记代理对象是否应该被aop框架通过AopContext以ThreadLocal的形式暴露出去。  
 // 当一个代理对象需要调用它自己的另外一个代理方法时，这个属性将非常有用。默认是是false，以避免不必要的拦截。  
 boolean exposeProxy = false;  
  
 // 标记该配置是否需要被冻结，如果被冻结，将不可以修改增强的配置。  
 // 当我们不希望调用方修改转换成Advised对象之后的代理对象时，这个配置将非常有用。  
 private boolean frozen = false;  
  
  
 //省略了属性的get set方法  
}

##### Advised接口

这个接口中定义了操作Aop代理配置的各种方法（比如指定被代理的目标对象、添加通知、添加顾问等等）。

所有由spring aop创建的代理对象默认都会实现这个接口。

public interface Advised extends TargetClassAware {  
  
 /\*\*  
 \* 返回配置是否已冻结，被冻结之后，无法修改已创建好的代理对象中的通知  
 \*/  
 boolean isFrozen();  
  
 /\*\*  
 \* 是否对目标类直接创建代理，而不是对接口创建代理，通俗点讲：如果是通过cglib创建代理，此方法返回true，否则返回false  
 \*/  
 boolean isProxyTargetClass();  
  
 /\*\*  
 \* 获取配置中需要代理的接口列表  
 \*/  
 Class<?>[] getProxiedInterfaces();  
  
 /\*\*  
 \* 判断某个接口是否被代理  
 \*/  
 boolean isInterfaceProxied(Class<?> intf);  
  
 /\*\*  
 \* 设置被代理的目标源，创建代理的时候，通常需要传入被代理的对象，最终被代理的对象会被包装为TargetSource类型的  
 \*/  
 void setTargetSource(TargetSource targetSource);  
  
 /\*\*  
 \* 返回被代理的目标源  
 \*/  
 TargetSource getTargetSource();  
  
 /\*\*  
 \* 设置是否需要将代理暴露在ThreadLocal中，这样可以在线程中获取到被代理对象，这个配置挺有用的，稍后会举例说明使用场景  
 \*/  
 void setExposeProxy(boolean exposeProxy);  
  
 /\*\*  
 \* 返回exposeProxy  
 \*/  
 boolean isExposeProxy();  
  
 /\*\*  
 \* 设置此代理配置是否经过预筛选，以便它只包含适用的顾问(匹配此代理的目标类)。  
 \* 默认设置是“假”。如果已经对advisor进行了预先筛选，则将其设置为“true”  
 \* 这意味着在为代理调用构建实际的advisor链时可以跳过ClassFilter检查。  
 \*/  
 void setPreFiltered(boolean preFiltered);  
  
 /\*\*  
 \* 返回preFiltered  
 \*/  
 boolean isPreFiltered();  
  
 /\*\*  
 \* 返回代理配置中干掉所有Advisor列表  
 \*/  
 Advisor[] getAdvisors();  
  
 /\*\*  
 \* 添加一个Advisor  
 \*/  
 void addAdvisor(Advisor advisor) throws AopConfigException;  
  
 /\*\*  
 \* 指定的位置添加一个Advisor  
 \*/  
 void addAdvisor(int pos, Advisor advisor) throws AopConfigException;  
  
 /\*\*  
 \* 移除一个Advisor  
 \*/  
 boolean removeAdvisor(Advisor advisor);  
  
 /\*\*  
 \* 移除指定位置的Advisor  
 \*/  
 void removeAdvisor(int index) throws AopConfigException;  
  
 /\*\*  
 \* 查找某个Advisor的位置  
 \*/  
 int indexOf(Advisor advisor);  
  
 /\*\*  
 \* 对advisor列表中的a替换为b  
 \*/  
 boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;  
  
 /\*\*  
 \* 添加一个通知  
 \*/  
 void addAdvice(Advice advice) throws AopConfigException;  
  
 /\*\*  
 \* 向指定的位置添加一个通知  
 \*/  
 void addAdvice(int pos, Advice advice) throws AopConfigException;  
  
 /\*\*  
 \* 移除一个通知  
 \*/  
 boolean removeAdvice(Advice advice);  
  
 /\*\*  
 \* 获取通知的位置  
 \*/  
 int indexOf(Advice advice);  
  
 /\*\*  
 \* 将代理配置转换为字符串，这个方便排错和调试使用的  
 \*/  
 String toProxyConfigString();  
  
}

##### AdvisedSupport类

这个类是个重点，AOP代理配置管理器的基类，继承了ProxyConfig并且实现了Advised接口，创建aop代理之前，所有需要配置的信息都是通过这个类来操作的。

比如：设置是否为目标类创建代理、设置目标对象、配置通知列表等等。

package org.springframework.aop.framework;  
  
public class AdvisedSupport extends ProxyConfig implements Advised {  
  
 public static final TargetSource EMPTY\_TARGET\_SOURCE = EmptyTargetSource.INSTANCE;  
  
 TargetSource targetSource = EMPTY\_TARGET\_SOURCE;  
  
 /\*\* 建议器是否已经针对特定的目标类进行筛选 \*/  
 private boolean preFiltered = false;  
  
 /\*\* 调用链工厂，用来获取目标方法的调用链 \*/  
 AdvisorChainFactory advisorChainFactory = new DefaultAdvisorChainFactory();  
  
 /\*\* 方法调用链缓存：以方法为键，以顾问链表为值的缓存。 \*/  
 private transient Map<MethodCacheKey, List<Object>> methodCache;  
  
 //代理对象需要实现的接口列表。保存在列表中以保持注册的顺序，以创建具有指定接口顺序的JDK代理。  
 private List<Class<?>> interfaces = new ArrayList<>();  
  
 //配置的顾问列表。所有添加的Advise对象都会被包装为Advisor对象  
 private List<Advisor> advisors = new ArrayList<>();  
  
 //数组更新了对advisor列表的更改，这更容易在内部操作。  
 private Advisor[] advisorArray = new Advisor[0];  
  
  
 //无参构造方法  
 public AdvisedSupport() {  
 this.methodCache = new ConcurrentHashMap<>(32);  
 }  
  
 //有参构造方法，参数为：代理需要实现的接口列表  
 public AdvisedSupport(Class<?>... interfaces) {  
 this();  
 setInterfaces(interfaces);  
 }  
  
 //设置需要被代理的目标对象，目标对象会被包装为TargetSource格式的对象  
 public void setTarget(Object target) {  
 setTargetSource(new SingletonTargetSource(target));  
 }  
  
 //设置被代理的目标源  
 @Override  
 public void setTargetSource(@Nullable TargetSource targetSource) {  
 this.targetSource = (targetSource != null ? targetSource : EMPTY\_TARGET\_SOURCE);  
 }  
  
 //获取被代理的目标源  
 @Override  
 public TargetSource getTargetSource() {  
 return this.targetSource;  
 }  
  
 //设置被代理的目标类  
 public void setTargetClass(@Nullable Class<?> targetClass) {  
 this.targetSource = EmptyTargetSource.forClass(targetClass);  
 }  
  
 //获取被代理的目标类型  
 @Override  
 @Nullable  
 public Class<?> getTargetClass() {  
 return this.targetSource.getTargetClass();  
 }  
  
 /\*\*   
 \* 设置此代理配置是否经过预筛选，这个什么意思呢：通过目标方法调用代理的时候，  
 \* 需要通过匹配的方式获取这个方法上的调用链列表，查找过程需要2个步骤：  
 \* 第一步：类是否匹配，第二步：方法是否匹配，当这个属性为true的时候，会直接跳过第一步，这个懂了不  
 \*/  
 @Override  
 public void setPreFiltered(boolean preFiltered) {  
 this.preFiltered = preFiltered;  
 }  
  
 // 返回preFiltered  
 @Override  
 public boolean isPreFiltered() {  
 return this.preFiltered;  
 }  
  
 /\*\*  
 \* 设置顾问链工厂，当调用目标方法的时候，需要获取这个方法上匹配的Advisor列表，  
 \* 获取目标方法上匹配的Advisor列表的功能就是AdvisorChainFactory来负责的  
 \*/  
 public void setAdvisorChainFactory(AdvisorChainFactory advisorChainFactory) {  
 Assert.notNull(advisorChainFactory, "AdvisorChainFactory must not be null");  
 this.advisorChainFactory = advisorChainFactory;  
 }  
  
 // 返回顾问链工厂对象  
 public AdvisorChainFactory getAdvisorChainFactory() {  
 return this.advisorChainFactory;  
 }  
  
  
 //设置代理对象需要实现的接口  
 public void setInterfaces(Class<?>... interfaces) {  
 Assert.notNull(interfaces, "Interfaces must not be null");  
 this.interfaces.clear();  
 for (Class<?> ifc : interfaces) {  
 addInterface(ifc);  
 }  
 }  
  
 //为代理对象添加需要实现的接口  
 public void addInterface(Class<?> intf) {  
 Assert.notNull(intf, "Interface must not be null");  
 if (!intf.isInterface()) {  
 throw new IllegalArgumentException("[" + intf.getName() + "] is not an interface");  
 }  
 if (!this.interfaces.contains(intf)) {  
 this.interfaces.add(intf);  
 adviceChanged();  
 }  
 }  
  
 //移除代理对象需要实现的接口  
 public boolean removeInterface(Class<?> intf) {  
 return this.interfaces.remove(intf);  
 }  
  
 //获取代理对象需要实现的接口列表  
 @Override  
 public Class<?>[] getProxiedInterfaces() {  
 return ClassUtils.toClassArray(this.interfaces);  
 }  
  
 //判断代理对象是否需要实现某个接口  
 @Override  
 public boolean isInterfaceProxied(Class<?> intf) {  
 for (Class<?> proxyIntf : this.interfaces) {  
 if (intf.isAssignableFrom(proxyIntf)) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 //获取配置的所有顾问列表  
 @Override  
 public final Advisor[] getAdvisors() {  
 return this.advisorArray;  
 }  
  
 //添加顾问  
 @Override  
 public void addAdvisor(Advisor advisor) {  
 int pos = this.advisors.size();  
 addAdvisor(pos, advisor);  
 }  
  
 //指定的位置添加顾问  
 @Override  
 public void addAdvisor(int pos, Advisor advisor) throws AopConfigException {  
 //这块先忽略，以后讲解  
 if (advisor instanceof IntroductionAdvisor) {  
 validateIntroductionAdvisor((IntroductionAdvisor) advisor);  
 }  
 addAdvisorInternal(pos, advisor);  
 }  
  
 //移除指定的顾问  
 @Override  
 public boolean removeAdvisor(Advisor advisor) {  
 int index = indexOf(advisor);  
 if (index == -1) {  
 return false;  
 }  
 else {  
 removeAdvisor(index);  
 return true;  
 }  
 }  
   
 //移除指定位置的顾问  
 @Override  
 public void removeAdvisor(int index) throws AopConfigException {  
 //当配置如果是冻结状态，是不允许对顾问进行修改的，否则会抛出异常  
 if (isFrozen()) {  
 throw new AopConfigException("Cannot remove Advisor: Configuration is frozen.");  
 }  
 if (index < 0 || index > this.advisors.size() - 1) {  
 throw new AopConfigException("Advisor index " + index + " is out of bounds: " +  
 "This configuration only has " + this.advisors.size() + " advisors.");  
 }  
 //移除advisors中的顾问  
 Advisor advisor = this.advisors.remove(index);  
 if (advisor instanceof IntroductionAdvisor) {  
 IntroductionAdvisor ia = (IntroductionAdvisor) advisor;  
 // We need to remove introduction interfaces.  
 for (Class<?> ifc : ia.getInterfaces()) {  
 removeInterface(ifc);  
 }  
 }  
 //更新advisorArray  
 updateAdvisorArray();  
 //通知已改变，内部会清除方法调用链缓存信息。  
 adviceChanged();  
 }  
  
 @Override  
 public int indexOf(Advisor advisor) {  
 Assert.notNull(advisor, "Advisor must not be null");  
 return this.advisors.indexOf(advisor);  
 }  
  
 @Override  
 public boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException {  
 Assert.notNull(a, "Advisor a must not be null");  
 Assert.notNull(b, "Advisor b must not be null");  
 int index = indexOf(a);  
 if (index == -1) {  
 return false;  
 }  
 removeAdvisor(index);  
 addAdvisor(index, b);  
 return true;  
 }  
  
 //批量添加顾问  
 public void addAdvisors(Advisor... advisors) {  
 addAdvisors(Arrays.asList(advisors));  
 }  
  
 //批量添加顾问  
 public void addAdvisors(Collection<Advisor> advisors) {  
 //配置如果是冻结状态，会抛出异常  
 if (isFrozen()) {  
 throw new AopConfigException("Cannot add advisor: Configuration is frozen.");  
 }  
 if (!CollectionUtils.isEmpty(advisors)) {  
 for (Advisor advisor : advisors) {  
 if (advisor instanceof IntroductionAdvisor) {  
 validateIntroductionAdvisor((IntroductionAdvisor) advisor);  
 }  
 Assert.notNull(advisor, "Advisor must not be null");  
 this.advisors.add(advisor);  
 }  
 updateAdvisorArray();  
 adviceChanged();  
 }  
 }  
   
 //此方法先忽略，用来为目标类引入接口的  
 private void validateIntroductionAdvisor(IntroductionAdvisor advisor) {  
 advisor.validateInterfaces();  
 // If the advisor passed validation, we can make the change.  
 Class<?>[] ifcs = advisor.getInterfaces();  
 for (Class<?> ifc : ifcs) {  
 addInterface(ifc);  
 }  
 }  
   
 //指定的位置添加顾问  
 private void addAdvisorInternal(int pos, Advisor advisor) throws AopConfigException {  
 Assert.notNull(advisor, "Advisor must not be null");  
 if (isFrozen()) {  
 throw new AopConfigException("Cannot add advisor: Configuration is frozen.");  
 }  
 if (pos > this.advisors.size()) {  
 throw new IllegalArgumentException(  
 "Illegal position " + pos + " in advisor list with size " + this.advisors.size());  
 }  
 this.advisors.add(pos, advisor);  
 updateAdvisorArray();  
 adviceChanged();  
 }  
  
 //将advisorArray和advisors保持一致  
 protected final void updateAdvisorArray() {  
 this.advisorArray = this.advisors.toArray(new Advisor[0]);  
 }  
  
 //获取顾问列表  
 protected final List<Advisor> getAdvisorsInternal() {  
 return this.advisors;  
 }  
  
 //添加通知  
 @Override  
 public void addAdvice(Advice advice) throws AopConfigException {  
 int pos = this.advisors.size();  
 addAdvice(pos, advice);  
 }  
  
 //指定的位置添加通知  
 @Override  
 public void addAdvice(int pos, Advice advice) throws AopConfigException {  
 //此处会将advice通知包装为DefaultPointcutAdvisor类型的Advisor  
 addAdvisor(pos, new DefaultPointcutAdvisor(advice));  
 }  
  
 //移除通知  
 @Override  
 public boolean removeAdvice(Advice advice) throws AopConfigException {  
 int index = indexOf(advice);  
 if (index == -1) {  
 return false;  
 }  
 else {  
 removeAdvisor(index);  
 return true;  
 }  
 }  
   
 //获取通知的位置  
 @Override  
 public int indexOf(Advice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 for (int i = 0; i < this.advisors.size(); i++) {  
 Advisor advisor = this.advisors.get(i);  
 if (advisor.getAdvice() == advice) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 //是否包含某个通知  
 public boolean adviceIncluded(@Nullable Advice advice) {  
 if (advice != null) {  
 for (Advisor advisor : this.advisors) {  
 if (advisor.getAdvice() == advice) {  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 //获取当前配置中某种类型通知的数量  
 public int countAdvicesOfType(@Nullable Class<?> adviceClass) {  
 int count = 0;  
 if (adviceClass != null) {  
 for (Advisor advisor : this.advisors) {  
 if (adviceClass.isInstance(advisor.getAdvice())) {  
 count++;  
 }  
 }  
 }  
 return count;  
 }  
  
  
 //基于当前配置，获取给定方法的方法调用链列表（即org.aopalliance.intercept.MethodInterceptor对象列表）  
 public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, @Nullable Class<?> targetClass) {  
 MethodCacheKey cacheKey = new MethodCacheKey(method);  
 //先从缓存中获取  
 List<Object> cached = this.methodCache.get(cacheKey);  
 //缓存中没有时，从advisorChainFactory中获取  
 if (cached == null) {  
 cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(  
 this, method, targetClass);  
 this.methodCache.put(cacheKey, cached);  
 }  
 return cached;  
 }  
  
 //通知更改时调用，会清空当前方法调用链缓存  
 protected void adviceChanged() {  
 this.methodCache.clear();  
 }  
  
 //将other中的配置信息复制到当前对象中  
 protected void copyConfigurationFrom(AdvisedSupport other) {  
 copyConfigurationFrom(other, other.targetSource, new ArrayList<>(other.advisors));  
 }  
  
 //将other中的配置信息复制到当前对象中  
 protected void copyConfigurationFrom(AdvisedSupport other, TargetSource targetSource, List<Advisor> advisors) {  
 copyFrom(other);  
 this.targetSource = targetSource;  
 this.advisorChainFactory = other.advisorChainFactory;  
 this.interfaces = new ArrayList<>(other.interfaces);  
 for (Advisor advisor : advisors) {  
 if (advisor instanceof IntroductionAdvisor) {  
 validateIntroductionAdvisor((IntroductionAdvisor) advisor);  
 }  
 Assert.notNull(advisor, "Advisor must not be null");  
 this.advisors.add(advisor);  
 }  
 updateAdvisorArray();  
 adviceChanged();  
 }  
  
 //构建此AdvisedSupport的仅配置副本，替换TargetSource。  
 AdvisedSupport getConfigurationOnlyCopy() {  
 AdvisedSupport copy = new AdvisedSupport();  
 copy.copyFrom(this);  
 copy.targetSource = EmptyTargetSource.forClass(getTargetClass(), getTargetSource().isStatic());  
 copy.advisorChainFactory = this.advisorChainFactory;  
 copy.interfaces = this.interfaces;  
 copy.advisors = this.advisors;  
 copy.updateAdvisorArray();  
 return copy;  
 }  
}

上面几个类有几个结论，这里说一下。

1. 配置中添加的Advice对象最终都会被转换为DefaultPointcutAdvisor对象，此时DefaultPointcutAdvisor未指定pointcut，大家可以去看一下DefaultPointcutAdvisor中pointcut有个默认值，默认会匹配任意类的任意方法。
2. 当配置被冻结的时候，即frozen为true的时，此时配置中的Advisor列表是不允许修改的。
3. 上面的getInterceptorsAndDynamicInterceptionAdvice方法，通过代理调用目标方法的时候，最后需要通过方法和目标类的类型，从当前配置中会获取匹配的方法拦截器列表，获取方法拦截器列表是由AdvisorChainFactory负责的。getInterceptorsAndDynamicInterceptionAdvice会在调用代理的方法时会执行，稍后在执行阶段会详解。
4. 目标方法和其关联的方法拦截器列表会被缓存在methodCache中，当顾问列表有变化的时候，methodCache缓存会被清除。

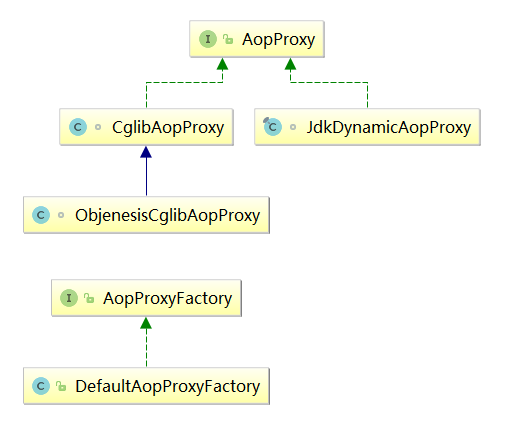
**配置阶段完成之后，下面进入AopProxy获取阶段。**

#### 根据配置获取AopProxy

这个阶段对应的代码：

// 创建AopProxy使用了简单工厂模式  
AopProxyFactory aopProxyFactory = new DefaultAopProxyFactory();  
//通过AopProxy工厂获取AopProxy对象  
AopProxy aopProxy = aopProxyFactory.createAopProxy(advisedSupport);

此阶段会根据AdvisedSupport中配置信息，判断具体是采用cglib的方式还是采用jdk动态代理的方式获取代理对象，先看一下涉及到的一些类。



##### AopProxy接口

这个接口定义了一个方法，用来创建最终的代理对象，这个接口有2个实现类

* CglibAopProxy：采用cglib的方式创建代理对象
* JkdDynamicAopProxy：采用jdk动态代理的方式创建代理对象

package org.springframework.aop.framework;  
  
public interface AopProxy {  
  
 /\*\*  
 \* 创建一个新的代理对象  
 \*/  
 Object getProxy();  
  
 /\*\*  
 \* 创建一个新的代理对象  
 \*/  
 Object getProxy(@Nullable ClassLoader classLoader);  
  
}

AopProxy的2个实现类，实现了上面定义的2个方法，稍后在代理的创建阶段详细介绍。

##### AopProxyFactory接口

通过名称就可以看出来，是一个工厂，负责创建AopProxy，使用的是简单工厂模式。

接口中定义了一个方法，会根据Aop的配置信息AdvisedSupport来获取AopProxy对象，主要是判断采用cglib的方式还是采用jdk动态代理的方式。

package org.springframework.aop.framework;  
  
public interface AopProxyFactory {  
 /\*\*  
 \* 根据aop配置信息获取AopProxy对象  
 \*/  
 AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException;  
}

##### DefaultAopProxyFactory类

AopProxyFactory接口的默认实现，代码比较简单，我们来细看一下

package org.springframework.aop.framework;  
  
/\*\*  
 \* 默认AopProxyFactory实现，创建CGLIB代理或JDK动态代理。  
 \* 对于给定的AdvisedSupport实例，以下条件为真，则创建一个CGLIB代理:  
 \* optimize = true  
 \* proxyTargetClass = true  
 \* 未指定代理接口  
 \* 通常，指定proxyTargetClass来强制执行CGLIB代理，或者指定一个或多个接口来使用JDK动态代理。  
 \*/  
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {  
  
 @Override  
 public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  
 // optimize==true || proxyTargetClass 为true || 配置中没有需要代理的接口  
 if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {  
 //获取需要被代理的类  
 Class<?> targetClass = config.getTargetClass();  
 if (targetClass == null) {  
 throw new AopConfigException("TargetSource cannot determine target class: " +  
 "Either an interface or a target is required for proxy creation.");  
 }  
 //如果被代理的类为接口 或者 被代理的类是jdk动态代理创建代理类，则采用JdkDynamicAopProxy的方式，否则采用cglib代理的方式  
 if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {  
 //采用jdk动态代理的方式  
 return new JdkDynamicAopProxy(config);  
 }  
 //采用cglib代理的方式  
 return new ObjenesisCglibAopProxy(config);  
 }  
 else {  
 //采用jdk动态代理的方式  
 return new JdkDynamicAopProxy(config);  
 }  
 }  
  
 /\*\*  
 \* 确定所提供的AdvisedSupport是否只指定了SpringProxy接口(或者根本没有指定代理接口)  
 \*/  
 private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {  
 Class<?>[] ifcs = config.getProxiedInterfaces();  
 return (ifcs.length == 0 || (ifcs.length == 1 && SpringProxy.class.isAssignableFrom(ifcs[0])));  
 }  
  
}

#### 代理创建阶段

到目前为止我们已经根据aop配置信息得到了AopProxy对象了，下面就可以调用AopProxy.getProxy方法获取代理对象了。

AopProxy.createAopProxy方法返回的结果有2种情况

* JdkDynamicAopProxy：以jdk动态代理的方式创建代理
* ObjenesisCglibAopProxy：以cglib的方式创建动态代理

项目详解这2个类的源码 。

##### JdkDynamicAopProxy类

**作用：采用jdk动态代理的方式创建代理对象，并处理代理对象的所有方法调用。**

final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {  
  
 //代理的配置信息  
 private final AdvisedSupport advised;  
  
 //需要被代理的接口中是否定义了equals方法  
 private boolean equalsDefined;  
  
 //需要被代理的接口中是否定义了hashCode方法  
 private boolean hashCodeDefined;  
  
 //通过AdvisedSupport创建实例  
 public JdkDynamicAopProxy(AdvisedSupport config) throws AopConfigException {  
 Assert.notNull(config, "AdvisedSupport must not be null");  
 if (config.getAdvisors().length == 0 && config.getTargetSource() == AdvisedSupport.EMPTY\_TARGET\_SOURCE) {  
 throw new AopConfigException("No advisors and no TargetSource specified");  
 }  
 this.advised = config;  
 }  
  
 //生成一个代理对象  
 @Override  
 public Object getProxy() {  
 return getProxy(ClassUtils.getDefaultClassLoader());  
 }  
  
 //生成一个代理对象  
 @Override  
 public Object getProxy(@Nullable ClassLoader classLoader) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Creating JDK dynamic proxy: " + this.advised.getTargetSource());  
 }  
 //@0：根据advised的信息获取代理需要被代理的所有接口列表  
 Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);   
 //查找被代理的接口中是否定义了equals、hashCode方法  
 findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);  
 /\*\*  
 \* 这个大家应该很熟悉吧，通过jdk动态代理创建代理对象，注意最后一个参数是this  
 \* 表示当前类，当前类是InvocationHandler类型的，当调用代理对象的任何方法的时候  
 \* 都会被被当前类的 invoke 方法处理  
 \*/  
 return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);  
 }  
  
 //判断需要代理的接口中是否定义了这几个方法（equals、hashCode）  
 private void findDefinedEqualsAndHashCodeMethods(Class<?>[] proxiedInterfaces) {  
 for (Class<?> proxiedInterface : proxiedInterfaces) {  
 //获取接口中定义的方法  
 Method[] methods = proxiedInterface.getDeclaredMethods();  
 for (Method method : methods) {  
 //是否是equals方法  
 if (AopUtils.isEqualsMethod(method)) {  
 this.equalsDefined = true;  
 }  
 //是否是hashCode方法  
 if (AopUtils.isHashCodeMethod(method)) {  
 this.hashCodeDefined = true;  
 }  
 //如果发现这2个方法都定义了，结束循环查找  
 if (this.equalsDefined && this.hashCodeDefined) {  
 return;  
 }  
 }  
 }  
 }  
  
  
 // 这个方法比较关键了，当在程序中调用代理对象的任何方法，最终都会被下面这个invoke方法处理  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 //旧的代理对象  
 Object oldProxy = null;  
 //用来标记是否需要将代理对象暴露在ThreadLocal中  
 boolean setProxyContext = false;  
 //获取目标源  
 TargetSource targetSource = this.advised.targetSource;  
 //目标对象  
 Object target = null;  
   
 //下面进入代理方法的处理阶段  
 try {  
 // 处理equals方法：被代理的接口中没有定义equals方法 && 当前调用是equals方法  
 if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {  
 // 直接调用当前类中的equals方法  
 return equals(args[0]);  
 }  
 // 处理hashCode方法：被代理的接口中没有定义hashCode方法 && 当前调用是hashCode方法  
 else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {  
 // 直接调用当前类中的hashCode方法  
 return hashCode();  
 }  
 /\*\*  
 \* 方法来源于 DecoratingProxy 接口，这个接口中定义了一个方法  
 \* 用来获取原始的被代理的目标类，主要是用在嵌套代理的情况下（所谓嵌套代理：代理对象又被作为目标对象进行了代理）  
 \*/  
 else if (method.getDeclaringClass() == DecoratingProxy.class) {  
 // 调用AopProxyUtils工具类的方法，内部通过循环遍历的方式，找到最原始的被代理的目标类  
 return AopProxyUtils.ultimateTargetClass(this.advised);  
 }  
 // 方法来源于 Advised 接口，代理对象默认情况下会实现 Advised 接口，可以通过代理对象来动态向代理对象中添加通知等  
 else if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&  
 method.getDeclaringClass().isAssignableFrom(Advised.class)) {  
 // this.advised是AdvisedSupport类型的，AdvisedSupport实现了Advised接口中的所有方法  
 // 所以最终通过通过反射方式交给this.advised来响应当前调用  
 return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);  
 }  
 // 用来记录方法返回值  
 Object retVal;  
   
 //是否需要在threadLocal中暴露代理对象  
 if (this.advised.exposeProxy) {  
 // 将代理对象暴露在上线文中，即暴露在threadLocal中，那么在当前线程中可以通过静态方法  
 // AopContext#currentProxy获取当前被暴露的代理对象，这个是非常有用的，稍后用案例来讲解，瞬间就会明白  
 oldProxy = AopContext.setCurrentProxy(proxy);  
 // 将setProxyContext标记为true  
 setProxyContext = true;  
 }  
  
 // 通过目标源获取目标对象  
 target = targetSource.getTarget();  
 // 获取目标对象类型  
 Class<?> targetClass = (target != null ? target.getClass() : null);  
  
 // @1：获取当前方法的拦截器链  
 List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);   
  
 // 拦截器链为空的情况下，表示这个方法上面没有找到任何增强的通知，那么会直接通过反射直接调用目标对象  
 if (chain.isEmpty()) {  
 // 获取方法请求的参数（有时候方法中有可变参数，所谓可变参数就是带有省略号(...)这种格式的参数，传入的参数类型和这种类型不一样的时候，会通过下面的adaptArgumentsIfNecessary方法进行转换）  
 Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);  
 //通过反射直接调用目标方法  
 retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);  
 }  
 else {  
 // 创建一个方法调用器（包含了代理对象、目标对象、调用的方法、参数、目标类型、方法拦截器链）  
 MethodInvocation invocation =  
 new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);  
 // @3：通过拦截器链一个个调用最终到目标方法的调用  
 retVal = invocation.proceed();  
 }  
  
 // 下面会根据方法返回值的类型，做一些处理，比如方法返回的类型为自己，则最后需要将返回值置为代理对象  
 Class<?> returnType = method.getReturnType();  
 if (retVal != null && retVal == target &&  
 returnType != Object.class && returnType.isInstance(proxy) &&  
 !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {  
 // 将返回值设置为代理对象  
 retVal = proxy;  
 }  
 // 方法的返回值类型returnType为原始类型（即int、byte、double等这种类型的） && retVal为null，  
 // 此时如果将null转换为原始类型会报错，所以此处直接抛出异常  
 else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {  
 throw new AopInvocationException(  
 "Null return value from advice does not match primitive return type for: " + method);  
 }  
 // 返回方法调用结果  
 return retVal;  
 }  
 finally {  
 // 目标对象不为null && 目标源不是静态的  
 //所谓静态的，你可以理解为是否是单例的  
 // isStatic为true，表示目标对象是单例的，同一个代理对象中所有方法共享一个目标对象  
 // isStatic为false的时候，通常每次调用代理的方法，target对象是不一样的，所以方法调用万之后需要进行释放，可能有些资源清理，连接的关闭等操作  
 if (target != null && !targetSource.isStatic()) {  
 // 必须释放来自TargetSource中的目标对象  
 targetSource.releaseTarget(target);  
 }  
 // setProxyContext为ture  
 if (setProxyContext) {  
 // 需要将旧的代理再放回到上线文中  
 AopContext.setCurrentProxy(oldProxy);  
 }  
 }  
 }  
  
}

关于上面代码，有几点细说一下

###### @0：completeProxiedInterfaces方法

@0处的代码如下，根据代理配置信息，获取需要被代理的所有接口

Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);

AopProxyUtils.completeProxiedInterfaces方法源码如下

static Class<?>[] completeProxiedInterfaces(AdvisedSupport advised, boolean decoratingProxy) {  
 //获取代理配置中需要被代理的接口  
 Class<?>[] specifiedInterfaces = advised.getProxiedInterfaces();  
 // 需要被代理的接口数量为0  
 if (specifiedInterfaces.length == 0) {  
 // 获取需要被代理的目标类型  
 Class<?> targetClass = advised.getTargetClass();  
 //目标类型不为空  
 if (targetClass != null) {  
 //目标类型为接口  
 if (targetClass.isInterface()) {  
 //将其添加到需要代理的接口中  
 advised.setInterfaces(targetClass);  
 }  
 // 目标类型为jdk动态代理创建的代理对象  
 else if (Proxy.isProxyClass(targetClass)) {  
 // 获取目标类型上的所有接口，将其添加到需要被代理的接口中  
 advised.setInterfaces(targetClass.getInterfaces());  
 }  
 //再次获取代理配置中需要被代理的接口  
 specifiedInterfaces = advised.getProxiedInterfaces();  
 }  
 }  
 //判断SpringProxy接口是否已经在被代理的接口中  
 boolean addSpringProxy = !advised.isInterfaceProxied(SpringProxy.class);  
 //判断Advised接口是否已经在被代理的接口中  
 boolean addAdvised = !advised.isOpaque() && !advised.isInterfaceProxied(Advised.class);  
 //判断DecoratingProxy接口是否已经在被代理的接口中  
 boolean addDecoratingProxy = (decoratingProxy && !advised.isInterfaceProxied(DecoratingProxy.class));  
 //一个计数器，会根据上面三个boolean值做递增  
 int nonUserIfcCount = 0;  
 if (addSpringProxy) {  
 nonUserIfcCount++;  
 }  
 if (addAdvised) {  
 nonUserIfcCount++;  
 }  
 if (addDecoratingProxy) {  
 nonUserIfcCount++;  
 }  
 // 下面就是构建所有需要被代理的接口  
 Class<?>[] proxiedInterfaces = new Class<?>[specifiedInterfaces.length + nonUserIfcCount];  
 System.arraycopy(specifiedInterfaces, 0, proxiedInterfaces, 0, specifiedInterfaces.length);  
 int index = specifiedInterfaces.length;  
 if (addSpringProxy) {  
 proxiedInterfaces[index] = SpringProxy.class;  
 index++;  
 }  
 if (addAdvised) {  
 proxiedInterfaces[index] = Advised.class;  
 index++;  
 }  
 if (addDecoratingProxy) {  
 proxiedInterfaces[index] = DecoratingProxy.class;  
 }  
 return proxiedInterfaces;  
}

上面的方法执行完毕之后，会得到一个被代理的接口列表，默认情况下会得到下面的一个列表

[开发者硬编码指定的需要被代理的接口列表,SpringProxy,Advised,DecoratingProxy]

最终创建出来的代理对象，默认会实现上面列的所有接口，后面3个接口是aop中自动给我们加上的。

###### @1：getInterceptorsAndDynamicInterceptionAdvice

这个方法位于AdvisedSupport中，根据方法和目标类型获取方法上面匹配的拦截器链

public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, @Nullable Class<?> targetClass) {  
 //会先尝试从还中获取，如果获取不到，会从advisorChainFactory中获取，然后将其丢到缓存中  
 MethodCacheKey cacheKey = new MethodCacheKey(method);  
 List<Object> cached = this.methodCache.get(cacheKey);  
 if (cached == null) {  
 cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(  
 this, method, targetClass);  
 this.methodCache.put(cacheKey, cached);  
 }  
 return cached;  
}

**从advisorChainFactory中获取拦截器链稍后细说，我们把这个阶段叫做连接器链的获取阶段。**

###### @3：ReflectiveMethodInvocation.proceed()

这个是一次会调用拦截器链，最终会调用到目标方法，获得目标方法的返回值，里面的细节见后面的代理方法调用处理阶段

##### JdkDynamicAopProxy小结

1. 被创建的代理对象默认会实现SpringProxy,Advised,DecoratingProxy 3个接口
2. SpringProxy这个接口中没有任何方法，只是起一个标记作用，用来标记代理对象是使用spring aop创建的
3. 代理对象默认都会实现Advised接口，所以可以通过这个接口动态变更代理对象中的通知
4. DecoratingProxy接口中定义了一个方法getDecoratedClass，用来获取被代理的原始目标对象的类型

下面来看另外一个类：ObjenesisCglibAopProxy，这个继承了CglibAopProxy，大部分逻辑都在CglibAopProxy中，所以我们主要看CglibAopProxy中代码。

##### CglibAopProxy类

**作用：采用cglib代理的方式创建代理对象，并处理代理对象的所有方法调用。**

以getProxy方法为入口，通过方法一个个来解说。

###### getProxy方法

public Object getProxy(@Nullable ClassLoader classLoader) {  
 // 获取被代理的类  
 Class<?> rootClass = this.advised.getTargetClass();  
  
 // 代理对象的父类（cglib是采用继承的方式是创建代理对象的，所以将被代理的类作为代理对象的父类）  
 Class<?> proxySuperClass = rootClass;  
 // 判断被代理的类是不是cglib创建的类，如果是cblib创建的类，会将其父类作为被代理的类  
 if (rootClass.getName().contains(ClassUtils.CGLIB\_CLASS\_SEPARATOR)) {  
 proxySuperClass = rootClass.getSuperclass();  
 //添加需要被代理的接口  
 Class<?>[] additionalInterfaces = rootClass.getInterfaces();  
 for (Class<?> additionalInterface : additionalInterfaces) {  
 this.advised.addInterface(additionalInterface);  
 }  
 }  
  
 // 开始cglib创建代理，这个大家对cglib比较熟悉的一看就懂  
 Enhancer enhancer = createEnhancer();  
 // 设置被代理的父类  
 enhancer.setSuperclass(proxySuperClass);  
 // 设置被代理的接口[开发者硬编码指定的需要被代理的接口列表,SpringProxy,Advised]，这个比jdk动态代理的方式少了一个DecoratingProxy接口  
 enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advised));  
 // 设置代理类类名生成策略  
 enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);  
 // 设置字节码的生成策略  
 enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(classLoader));  
  
 // @1：获取Callback列表，这个稍后详解  
 Callback[] callbacks = getCallbacks(rootClass);  
 Class<?>[] types = new Class<?>[callbacks.length];  
 for (int x = 0; x < types.length; x++) {  
 types[x] = callbacks[x].getClass();  
 }  
 // @2：设置CallbackFilter，CallbackFilter内部会判断被代理对象中的方法最终会被callbacks列表中的那个Callback来处理  
 enhancer.setCallbackFilter(new ProxyCallbackFilter(  
 this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));  
 enhancer.setCallbackTypes(types);  
  
 // 获取代理对象（内部会先创建代理类，然后会根据代理类生成一个代理对象）  
 return createProxyClassAndInstance(enhancer, callbacks);  
}

上面方法中有2个点比较难，需要说明，分别是@1：getCallbacks方法和@2:创建ProxyCallbackFilter对象

###### @1：getCallbacks方法

通过被代理的类来获取Callback列表，Callback是用来处理代理对象的方法调用的，代理对象中可能有很多方法，每个方法可能采用不同的处理方式，所以会有多个Callback

private Callback[] getCallbacks(Class<?> rootClass) throws Exception {  
 // 是否需要将代理暴露在threadLocal中  
 boolean exposeProxy = this.advised.isExposeProxy();  
 // 配置是否是冻结的  
 boolean isFrozen = this.advised.isFrozen();  
 // 被代理的目标对象是否是动态的（是否是单例的）  
 boolean isStatic = this.advised.getTargetSource().isStatic();  
   
 // 当方法上有需要执行的拦截器的时候，会用这个来处理  
 Callback aopInterceptor = new DynamicAdvisedInterceptor(this.advised);  
  
 // 当方法上没有需要执行的拦截器的时候，会使用targetInterceptor来处理，内部会通过反射直接调用目标对象的方法  
 Callback targetInterceptor;  
 /\*\*  
 \* 这块根据是否需要暴露代理到threadLocal中以及目标对象是否是动态的，会创建不同的Callback  
 \* isStatic为true的时候，同一个代理的不同方法可能都是新的目标对象，所以当代理方法执行完毕之后，需要对目标对象进行释放  
 \*/  
 if (exposeProxy) {  
 targetInterceptor = (isStatic ?  
 new StaticUnadvisedExposedInterceptor(this.advised.getTargetSource().getTarget()) :  
 new DynamicUnadvisedExposedInterceptor(this.advised.getTargetSource()));  
 }  
 else {  
 targetInterceptor = (isStatic ?  
 new StaticUnadvisedInterceptor(this.advised.getTargetSource().getTarget()) :  
 new DynamicUnadvisedInterceptor(this.advised.getTargetSource()));  
 }  
  
 // targetDispatcher会直接调用目标方法  
 Callback targetDispatcher = (isStatic ?  
 new StaticDispatcher(this.advised.getTargetSource().getTarget()) : new SerializableNoOp());  
  
 Callback[] mainCallbacks = new Callback[] {  
 aopInterceptor, // 处理匹配到拦截器的方法  
 targetInterceptor, // 处理未匹配到拦截器的方法  
 new SerializableNoOp(),   
 targetDispatcher, // 处理未匹配到拦截器的方法，和targetInterceptor有何不同呢？目标方法如果返回值的结果是目标对象类型的，会使用 targetInterceptor 处理，内部会返回代理对象  
 this.advisedDispatcher, // 处理Advised接口中定义的方法  
 new EqualsInterceptor(this.advised), // 处理equals方法  
 new HashCodeInterceptor(this.advised) // 处理hashCode方法  
 };  
  
 Callback[] callbacks;  
  
 // 如果被代理的对象是单例的 && 配置是冻结的，此时会进行优化，怎么优化呢？  
 // 配置冻结的情况下，生成好的代理中通知是无法修改的，所以可以提前将每个方法对应的拦截器链找到给缓存起来  
 // 调用方法的时候，就直接从缓存中可以拿到方法对应的缓存信息，效率会高一些  
 if (isStatic && isFrozen) {  
 Method[] methods = rootClass.getMethods();  
 Callback[] fixedCallbacks = new Callback[methods.length];  
 this.fixedInterceptorMap = new HashMap<>(methods.length);  
  
 // 获取每个方法的调用链，然后给缓存在fixedInterceptorMap中  
 for (int x = 0; x < methods.length; x++) {  
 Method method = methods[x];  
 List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, rootClass);  
 fixedCallbacks[x] = new FixedChainStaticTargetInterceptor(  
 chain, this.advised.getTargetSource().getTarget(), this.advised.getTargetClass());  
 this.fixedInterceptorMap.put(method, x);  
 }  
 callbacks = new Callback[mainCallbacks.length + fixedCallbacks.length];  
 System.arraycopy(mainCallbacks, 0, callbacks, 0, mainCallbacks.length);  
 System.arraycopy(fixedCallbacks, 0, callbacks, mainCallbacks.length, fixedCallbacks.length);  
 this.fixedInterceptorOffset = mainCallbacks.length;  
 }  
 else {  
 callbacks = mainCallbacks;  
 }  
 return callbacks;  
}

###### @2:创建ProxyCallbackFilter对象

enhancer.setCallbackFilter(new ProxyCallbackFilter(  
 this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));

这块重点在于ProxyCallbackFilter中的accept方法，这个方法会根据目标放，获取目标对方最后会让callbacks列表中的哪个Callback处理，大家可以看一下源码，比较简单。

上面getCallbacks方法中涉及到了5个类如下

* DynamicAdvisedInterceptor
* StaticUnadvisedExposedInterceptor
* StaticUnadvisedInterceptor
* DynamicUnadvisedInterceptor
* StaticDispatcher

后面4个比较简单，大家可以去看一下源码，主要来看第一个类，基本上代理对象中的大部分自定义的方法都会进入到这个类的intercept方法中进行处理，代码如下

###### DynamicAdvisedInterceptor类

private static class DynamicAdvisedInterceptor implements MethodInterceptor, Serializable {  
 //代理配置信息  
 private final AdvisedSupport advised;  
 //构造器，需要一个AdvisedSupport  
 public DynamicAdvisedInterceptor(AdvisedSupport advised) {  
 this.advised = advised;  
 }  
  
 //这个方法是关键，用来处理代理对象中方法的调用  
 public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  
 //被暴露在threadLocal中旧的代理对象  
 Object oldProxy = null;  
 //用来标记代理对象是否被暴露在threadLocal中  
 boolean setProxyContext = false;  
 //目标对象  
 Object target = null;  
 //目标源  
 TargetSource targetSource = this.advised.getTargetSource();  
 try {  
 //代理配置中是否需要将代理暴露在threadLocal中  
 if (this.advised.exposeProxy) {  
 //将代理对象暴露出去  
 oldProxy = AopContext.setCurrentProxy(proxy);  
 //将setProxyContext置为true  
 setProxyContext = true;  
 }  
 //获取目标对象（即被代理的对象）  
 target = targetSource.getTarget();  
 Class<?> targetClass = (target != null ? target.getClass() : null);  
 //@1：获取当前方法的拦截器链  
 List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
 //记录方法返回值  
 Object retVal;  
 //拦截器链不为空 && 方法是public类型的  
 if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {  
 //获取方法调用参数  
 Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);  
 // 直接调用目标对象的方法  
 retVal = methodProxy.invoke(target, argsToUse);  
 }  
 else {  
 // 创建一个方法调用器（包含了代理对象、目标对象、调用的方法、参数、目标类型、方法拦截器链）  
 // @2：并执行方法调用器的processd()方法，此方法会一次执行方法调用链，最终会调用目标方法，获取返回结果  
 retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed();  
 }  
 // 处理方法返回结果：会根据方法返回值的类型，做一些处理，比如方法返回的类型为自己，则最后需要将返回值置为代理对象  
 retVal = processReturnType(proxy, target, method, retVal);  
 return retVal;  
 }  
 finally {  
 // 目标对象不为null && 目标源不是静态的  
 //所谓静态的，你可以理解为是否是单例的  
 // isStatic为true，表示目标对象是单例的，同一个代理对象中所有方法共享一个目标对象  
 // isStatic为false的时候，通常每次调用代理的方法，target对象是不一样的，所以方法调用万之后需要进行释放，可能有些资源清理，连接的关闭等操作  
 if (target != null && !targetSource.isStatic()) {  
 targetSource.releaseTarget(target);  
 }  
 // setProxyContext为ture  
 if (setProxyContext) {  
 // 需要将旧的代理再放回到上线文中  
 AopContext.setCurrentProxy(oldProxy);  
 }  
 }  
 }  
}

上面代码中2个重点：@1和@2

@1：获取当前方法的拦截器链，这个在JdkDynamicAopProxy的也有，稍后说。

@2：调用CglibMethodInvocation.proceed()，内部会一次调用方法拦截器链，最终会调用目标方法，获取目标方法返回值，这个稍后放在代理方法处理阶段详解。

下面来看一下方法拦截器链的获取。

##### 方法拦截器链的获取

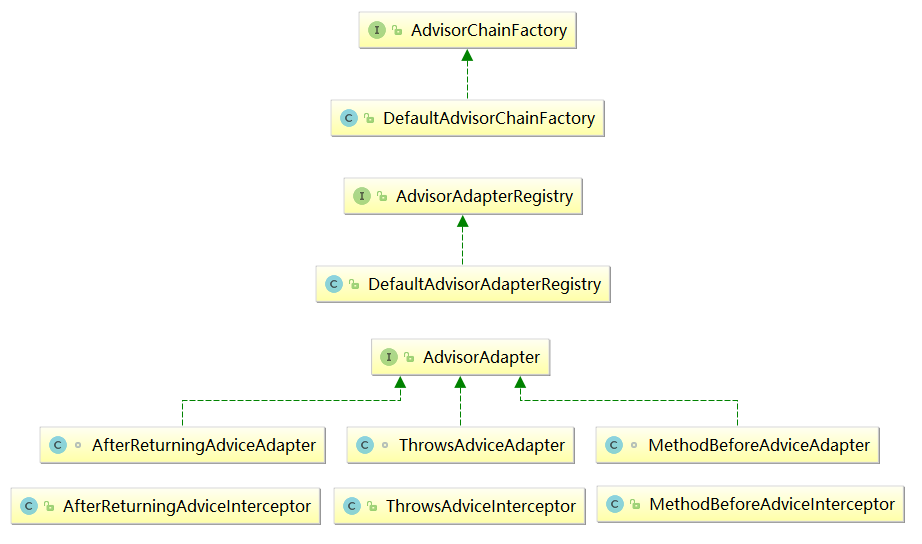
**我们在创建代理的时候，增强的代码通常都放在Advise通知中，但是最终调用方法的时候，这些通知都会被转换为MethodInterceptor来执行，调用方法的过程中，需要先获取方法上匹配的所有方法连接器连，然后依次执行，最终会调用到目标方法。**

获取方法对应的拦截器链，对应下面这段代码

org.springframework.aop.framework.AdvisedSupport#getInterceptorsAndDynamicInterceptionAdvice  
  
AdvisorChainFactory advisorChainFactory = new DefaultAdvisorChainFactory();  
   
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, @Nullable Class<?> targetClass) {  
 MethodCacheKey cacheKey = new MethodCacheKey(method);  
 List<Object> cached = this.methodCache.get(cacheKey);  
 if (cached == null) {  
 cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(  
 this, method, targetClass);  
 this.methodCache.put(cacheKey, cached);  
 }  
 return cached;  
}

会调用DefaultAdvisorChainFactory#getInterceptorsAndDynamicInterceptionAdvice方法获取方法上匹配的拦截器链。

涉及到的类



###### AdvisorChainFactory接口

拦截器链工厂接口，定义了一个方法，用来获取方法匹配的拦截器链列表

package org.springframework.aop.framework;  
  
public interface AdvisorChainFactory {  
  
 /\*\*  
 \* 获取方法匹配的拦截器链列表  
 \* @param config：代理配置信息，里面包含了创建代理的所有信息，如：Advisor列表，此方法会从Advisor列表中找到和mehod匹配的  
 \* @param targetClass：目标类  
 \*/  
 List<Object> getInterceptorsAndDynamicInterceptionAdvice(Advised config, Method method, @Nullable Class<?> targetClass);  
}

###### DefaultAdvisorChainFactory类

AdvisorChainFactory接口的默认实现。

public class DefaultAdvisorChainFactory implements AdvisorChainFactory, Serializable {  
  
 @Override  
 public List<Object> getInterceptorsAndDynamicInterceptionAdvice(  
 Advised config, Method method, @Nullable Class<?> targetClass) {  
  
 // 获取Advisor适配器注册器，前面我们有提到过一个知识点：所有的Advisor最终都会转换为MethodInterceptor类型的，  
 // 然后注册方法调用链去执行，AdvisorAdapterRegistry就是搞这个事情的,  
 // 其内部会将非MethodInterceptor类型通知通过适配器转换为MethodInterceptor类型  
 AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();  
 //获取配置中的Advisor列表  
 Advisor[] advisors = config.getAdvisors();  
 List<Object> interceptorList = new ArrayList<>(advisors.length);  
 //获取被调用方法所在类实际的类型  
 Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaringClass());  
 Boolean hasIntroductions = null;  
 //遍历Advisor列表，找到和actualClass和方法匹配的所有方法拦截器（MethodInterceptor）链列表  
 for (Advisor advisor : advisors) {  
 //判断是否是PointcutAdvisor类型的，这种类型的匹配分为2个阶段，先看类是否匹配，然后再看方法是否匹配  
 if (advisor instanceof PointcutAdvisor) {  
 PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;  
 // 如果isPreFiltered为ture，表示类以及匹配过，不需要看类是否匹配了  
 if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {  
 MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();  
 boolean match;  
 if (mm instanceof IntroductionAwareMethodMatcher) {  
 if (hasIntroductions == null) {  
 hasIntroductions = hasMatchingIntroductions(advisors, actualClass);  
 }  
 match = ((IntroductionAwareMethodMatcher) mm).matches(method, actualClass, hasIntroductions);  
 }  
 else {  
 //方法是否匹配  
 match = mm.matches(method, actualClass);  
 }  
 //方法匹配  
 if (match) {  
 // 通过AdvisorAdapterRegistry的getInterceptors将advisor转换为MethodInterceptor列表  
 MethodInterceptor[] interceptors = registry.getInterceptors(advisor);  
 //方法是否动态匹配  
 if (mm.isRuntime()) {  
 //轮询连接器，将其包装为InterceptorAndDynamicMethodMatcher对象，后续方法调用的时候可以做动态匹配  
 for (MethodInterceptor interceptor : interceptors) {  
 interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm));  
 }  
 }  
 else {  
 interceptorList.addAll(Arrays.asList(interceptors));  
 }  
 }  
 }  
 }  
 else if (advisor instanceof IntroductionAdvisor) {  
 IntroductionAdvisor ia = (IntroductionAdvisor) advisor;  
 if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass)) {  
 Interceptor[] interceptors = registry.getInterceptors(advisor);  
 interceptorList.addAll(Arrays.asList(interceptors));  
 }  
 }  
 else {  
 Interceptor[] interceptors = registry.getInterceptors(advisor);  
 interceptorList.addAll(Arrays.asList(interceptors));  
 }  
 }  
  
 return interceptorList;  
 }  
}

下面来看AdvisorAdapterRegistry这个接口。

###### AdvisorAdapterRegistry接口

AdvisorAdapter注册器，AdvisorAdapter可以将Advisor中的Advice适配为MethodInterceptor

package org.springframework.aop.framework.adapter;  
  
public interface AdvisorAdapterRegistry {  
  
 //将一个通知（Advice）包装为Advisor对象  
 Advisor wrap(Object advice) throws UnknownAdviceTypeException;  
  
 //根据Advisor获取方法MethodInterceptor列表  
 MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException;  
  
 //注册AdvisorAdapter，AdvisorAdapter可以将Advisor中的Advice适配为MethodInterceptor  
 void registerAdvisorAdapter(AdvisorAdapter adapter);  
  
}

###### DefaultAdvisorAdapterRegistry类

AdvisorAdapterRegistry的默认实现，目前里面做的事情主要是将负责将前置通知，异常通知，后置通知转换为MethodInterceptor类型的，源码比较简单，大家看一下就懂了。

public class DefaultAdvisorAdapterRegistry implements AdvisorAdapterRegistry, Serializable {  
   
 //AdvisorAdapter转换器列表，AdvisorAdapter负责将Advisor中的Advice转换为MethodInterceptor类型的  
 private final List<AdvisorAdapter> adapters = new ArrayList<>(3);  
   
 //默认会注册3个AdvisorAdapter，这3个负责将前置通知，异常通知，后置通知转换为MethodInterceptor类型的  
 public DefaultAdvisorAdapterRegistry() {  
 registerAdvisorAdapter(new MethodBeforeAdviceAdapter());  
 registerAdvisorAdapter(new AfterReturningAdviceAdapter());  
 registerAdvisorAdapter(new ThrowsAdviceAdapter());  
 }  
  
  
 @Override  
 public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {  
 if (adviceObject instanceof Advisor) {  
 return (Advisor) adviceObject;  
 }  
 if (!(adviceObject instanceof Advice)) {  
 throw new UnknownAdviceTypeException(adviceObject);  
 }  
 Advice advice = (Advice) adviceObject;  
 if (advice instanceof MethodInterceptor) {  
 // So well-known it doesn't even need an adapter.  
 return new DefaultPointcutAdvisor(advice);  
 }  
 //轮询adapters  
 for (AdvisorAdapter adapter : this.adapters) {  
 //adapter是否支持适配advice这个通知  
 if (adapter.supportsAdvice(advice)) {  
 return new DefaultPointcutAdvisor(advice);  
 }  
 }  
 throw new UnknownAdviceTypeException(advice);  
 }  
  
 //将Advisor对象转换为MethodInterceptor列表，不过通常情况下一个advisor会返回一个MethodInterceptor  
 @Override  
 public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException {  
 List<MethodInterceptor> interceptors = new ArrayList<>(3);  
 Advice advice = advisor.getAdvice();  
 if (advice instanceof MethodInterceptor) {  
 interceptors.add((MethodInterceptor) advice);  
 }  
 //轮询adapters  
 for (AdvisorAdapter adapter : this.adapters) {  
 //先看一下adapter是否支持适配advice这个通知  
 if (adapter.supportsAdvice(advice)) {  
 //如果匹配，这调用适配器的getInterceptor方法将advisor转换为MethodInterceptor  
 interceptors.add(adapter.getInterceptor(advisor));  
 }  
 }  
 if (interceptors.isEmpty()) {  
 throw new UnknownAdviceTypeException(advisor.getAdvice());  
 }  
 return interceptors.toArray(new MethodInterceptor[0]);  
 }  
  
 @Override  
 public void registerAdvisorAdapter(AdvisorAdapter adapter) {  
 this.adapters.add(adapter);  
 }  
  
}

###### AdvisorAdapter接口

package org.springframework.aop.framework.adapter;  
  
public interface AdvisorAdapter {  
 //判断这个适配器支持advice这个通知么  
 boolean supportsAdvice(Advice advice);  
  
 //获取advisor对应的MethodInterceptor  
 MethodInterceptor getInterceptor(Advisor advisor);  
}

###### MethodBeforeAdviceAdapter类

适配MethodBeforeAdvice前置通知，负责将MethodBeforeAdvice类型的通知转换为MethodBeforeAdviceInterceptor类型的

class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {  
  
 @Override  
 public boolean supportsAdvice(Advice advice) {  
 return (advice instanceof MethodBeforeAdvice);  
 }  
  
 @Override  
 public MethodInterceptor getInterceptor(Advisor advisor) {  
 MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();  
 return new MethodBeforeAdviceInterceptor(advice);  
 }  
  
}

###### MethodBeforeAdviceInterceptor类

将MethodBeforeAdvice通知适配为MethodInterceptor类型的，代码很简单，大家一看就懂。

package org.springframework.aop.framework.adapter;  
  
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, BeforeAdvice, Serializable {  
  
 private final MethodBeforeAdvice advice;  
  
 public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //先调用前置通知  
 this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());  
 //然后继续处理连接器连，内部会调用目标方法  
 return mi.proceed();  
 }  
  
}

###### AfterReturningAdviceAdapter类

适配AfterReturningAdvice后置通知，负责将AfterReturningAdvice类型的通知转换为AfterReturningAdviceInterceptor类型的

class AfterReturningAdviceAdapter implements AdvisorAdapter, Serializable {  
  
 @Override  
 public boolean supportsAdvice(Advice advice) {  
 return (advice instanceof AfterReturningAdvice);  
 }  
  
 @Override  
 public MethodInterceptor getInterceptor(Advisor advisor) {  
 AfterReturningAdvice advice = (AfterReturningAdvice) advisor.getAdvice();  
 return new AfterReturningAdviceInterceptor(advice);  
 }  
  
}

###### AfterReturningAdviceInterceptor类

将AfterReturningAdvice通知适配为MethodInterceptor类型的，代码很简单，大家一看就懂。

public class AfterReturningAdviceInterceptor implements MethodInterceptor, AfterAdvice, Serializable {  
  
 private final AfterReturningAdvice advice;  
  
 public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //先调用拦截器链，内部会调用目标方法  
 Object retVal = mi.proceed();  
 //然后执行后置通知  
 this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.getThis());  
 //返回结果  
 return retVal;  
 }  
  
}

###### ThrowsAdviceAdapter类

适配ThrowsAdvice前置通知，负责将MethodBeforeAdvice类型的通知转换为MethodBeforeAdviceInterceptor类型的

class ThrowsAdviceAdapter implements AdvisorAdapter, Serializable {  
  
 @Override  
 public boolean supportsAdvice(Advice advice) {  
 return (advice instanceof ThrowsAdvice);  
 }  
  
 @Override  
 public MethodInterceptor getInterceptor(Advisor advisor) {  
 return new ThrowsAdviceInterceptor(advisor.getAdvice());  
 }  
  
}

###### MethodBeforeAdviceInterceptor类

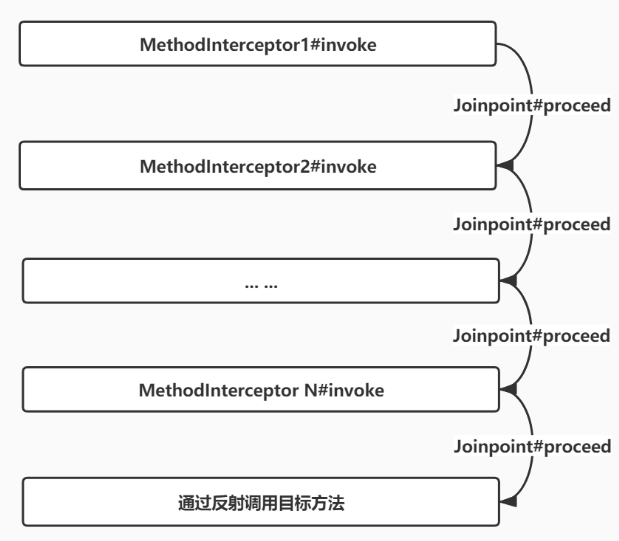
将ThrowsAdvice通知适配为MethodInterceptor类型的，代码很简单，大家一看就懂。

package org.springframework.aop.framework.adapter;  
  
public class ThrowsAdviceInterceptor implements MethodInterceptor, AfterAdvice {  
  
 private static final String AFTER\_THROWING = "afterThrowing";  
   
 private final Object throwsAdvice;  
  
 //创建ThrowsAdviceInterceptor  
 public ThrowsAdviceInterceptor(Object throwsAdvice) {  
 Assert.notNull(throwsAdvice, "Advice must not be null");  
 this.throwsAdvice = throwsAdvice;  
 //获取异常通知中定义的所有方法（public、默认的、protected、private）  
 Method[] methods = throwsAdvice.getClass().getMethods();  
 //轮询methods  
 for (Method method : methods) {  
 //方法名称为afterThrowing && 方法参数为1或者4  
 if (method.getName().equals(AFTER\_THROWING) &&  
 (method.getParameterCount() == 1 || method.getParameterCount() == 4)) {  
 //获取方法的最后一个参数类型  
 Class<?> throwableParam = method.getParameterTypes()[method.getParameterCount() - 1];  
 //判断方法参数类型是不是Throwable类型的  
 if (Throwable.class.isAssignableFrom(throwableParam)) {  
 // 缓存异常处理方法到map中（异常类型->异常处理方法）  
 this.exceptionHandlerMap.put(throwableParam, method);  
 }  
 }  
 }  
 //如果exceptionHandlerMap，抛出异常，所以最少要有一个异常处理方法  
 if (this.exceptionHandlerMap.isEmpty()) {  
 throw new IllegalArgumentException(  
 "At least one handler method must be found in class [" + throwsAdvice.getClass() + "]");  
 }  
 }  
  
  
 /\*\*  
 \* 获取异常通知中自定义的处理异常方法的数量  
 \*/  
 public int getHandlerMethodCount() {  
 return this.exceptionHandlerMap.size();  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //调用通知链  
 return mi.proceed();  
 }  
 catch (Throwable ex) {  
 //获取异常通知中自定义的处理异常的方法  
 Method handlerMethod = getExceptionHandler(ex);  
 //当处理的方法不为空  
 if (handlerMethod != null) {  
 //调用异常处理方法  
 invokeHandlerMethod(mi, ex, handlerMethod);  
 }  
 //继续向外抛出异常  
 throw ex; //@1  
 }  
 }  
  
 /\*\*  
 \* 获取throwsAdvice中处理exception参数指定的异常的方法  
 \*/  
 @Nullable  
 private Method getExceptionHandler(Throwable exception) {  
 //获取异常类型  
 Class<?> exceptionClass = exception.getClass();  
 //从缓存中获取异常类型对应的方法  
 Method handler = this.exceptionHandlerMap.get(exceptionClass);  
 //来一个循环，查询处理方法，循环条件：方法为空 && 异常类型!=Throwable  
 while (handler == null && exceptionClass != Throwable.class) {  
 //获取异常的父类型  
 exceptionClass = exceptionClass.getSuperclass();  
 //从缓存中查找异常对应的处理方法  
 handler = this.exceptionHandlerMap.get(exceptionClass);  
 }  
 //将查找结果返回  
 return handler;  
 }  
  
 //通过反射调用异常通知中的异常方法  
 private void invokeHandlerMethod(MethodInvocation mi, Throwable ex, Method method) throws Throwable {  
 //构建方法请求参数  
 Object[] handlerArgs;  
 //若只有1个参数，参数为：异常对象  
 if (method.getParameterCount() == 1) {  
 handlerArgs = new Object[] {ex};  
 }  
 else {  
 //4个参数（方法、方法请求参数、目标对象、异常对象）  
 handlerArgs = new Object[] {mi.getMethod(), mi.getArguments(), mi.getThis(), ex};  
 }  
 try {  
 //通过反射调用异常通知中的方法  
 method.invoke(this.throwsAdvice, handlerArgs);  
 }  
 catch (InvocationTargetException targetEx) {  
 throw targetEx.getTargetException();  
 }  
 }  
}

### 代理方法的调用过程（拦截器链的执行）

#### 拦截器链执行过程

到目前，已经获取到代理对象，接着会开始使用这个代理对象，在代理对象上执行一些方法调用，此时会依次调用此方法上的所有MethodInterceptor，最终会调用到目标上对应的方法，执行过程如下图



jdk动态代理方式创建代理最终会调用ReflectiveMethodInvocation#proceed方法。

cglib方式创建的代理最终会调用CglibAopProxy.CglibMethodInvocation#proceed方法。

下面来看一下这个两个类的代码。

#### ReflectiveMethodInvocation类

package org.springframework.aop.framework;  
  
public class ReflectiveMethodInvocation implements ProxyMethodInvocation, Cloneable {  
  
 //生成的代理对象  
 protected final Object proxy;  
 //被代理的目标对象  
 protected final Object target;  
 //被调用的方法  
 protected final Method method;  
 //调用方法传入参数  
 protected Object[] arguments;  
 //目标对象类型  
 private final Class<?> targetClass;  
  
 /\*\*  
 \* 当期被调用的方法上匹配的 MethodInterceptor and InterceptorAndDynamicMethodMatcher 列表  
 \* 即方法调用链列表  
 \*/  
 protected final List<?> interceptorsAndDynamicMethodMatchers;  
  
 //当前正在调用的连接器索引  
 private int currentInterceptorIndex = -1;  
  
 //构造器  
 protected ReflectiveMethodInvocation(  
 Object proxy, @Nullable Object target, Method method, @Nullable Object[] arguments,  
 @Nullable Class<?> targetClass, List<Object> interceptorsAndDynamicMethodMatchers) {  
 this.proxy = proxy;  
 this.target = target;  
 this.targetClass = targetClass;  
 //获取桥接方法，关于什么是桥接方法，比较简单，百度一下，这里不做说明  
 this.method = BridgeMethodResolver.findBridgedMethod(method);  
 this.arguments = AopProxyUtils.adaptArgumentsIfNecessary(method, arguments);  
 this.interceptorsAndDynamicMethodMatchers = interceptorsAndDynamicMethodMatchers;  
 }  
  
 //这里是重点，用来处理被调用的方法，会递归进行调用，所有的拦截器都执行完毕之后，会通过反射调用目标方法  
 public Object proceed() throws Throwable {  
 // 拦截器都执行完毕之后，通过反射调用目标对象中的方法  
 if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {  
 return invokeJoinpoint();  
 }  
   
 //获取++this.currentInterceptorIndex指定的拦截器  
 Object interceptorOrInterceptionAdvice =  
 this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);  
 //判断拦截器是否是InterceptorAndDynamicMethodMatcher，这种表示是动态拦截器，  
 // 所谓动态拦截器就是要根据方法的参数的值来判断拦截器是否需要执行  
 if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {  
 InterceptorAndDynamicMethodMatcher dm =  
 (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;  
 Class<?> targetClass = (this.targetClass != null ? this.targetClass : this.method.getDeclaringClass());  
 //判断动态拦截器是否需要执行  
 if (dm.methodMatcher.matches(this.method, targetClass, this.arguments)) {  
 //执行当前拦截器的调用  
 return dm.interceptor.invoke(this);  
 }  
 else {  
 //如果不匹配，直接递归进入下一个拦截器的调用  
 return proceed();  
 }  
 }  
 else {  
 //执行拦截器的调用  
 return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);  
 }  
 }  
  
 //通过反射调用目标方法  
 @Nullable  
 protected Object invokeJoinpoint() throws Throwable {  
 return AopUtils.invokeJoinpointUsingReflection(this.target, this.method, this.arguments);  
 }  
}

### ProxyFactory简化代理的创建

上面代理的整个创建过程和使用过程还是挺复杂的，spring在AdvisedSupport类的基础上又添加2个子类

* ProxyCreatorSupport
* ProxyFactory

通过这2个子类，将步骤稍微简化了一些，这2个类的代码比较简单，上面的如果理解了，看这2个类的代码会非常的轻松，源码这里就不细说了。

ProxyCreatorSupport用来对代理的创建提供支持，内部添加了AopProxyFactory对象的引用，将代理的创建过程给简化了。

AopProxyFactory aopProxyFactory;

ProxyFactory类继承了ProxyCreatorSupport，让创建代理的过程更简单了，如果采用硬编码的方式，通常我们会使用ProxyFactory来创建代理对象，代码只需要下面几行了

//通过spring提供的代理创建工厂来创建代理  
ProxyFactory proxyFactory = new ProxyFactory();  
//ProxyFactory继承了AdvisedSupport类，所以可以直接可以通过ProxyFactory来设置创建代理需要的参数  
//为工厂指定目标对象  
proxyFactory.setTarget(target);  
//添加顾问  
proxyFactory.addAdvisor(advisor);  
//调用proxyFactory.getProxy();创建代理  
Object proxy = proxyFactory.getProxy();

### 案例

下面来一些案例，通过案例理解会更容易一些。

#### 案例1

这个案例主要看一下生成的代理对象的一些信息。

package com.javacode2018.aop.demo5;  
  
import com.javacode2018.aop.demo4.FundsService;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.aop.support.DefaultPointcutAdvisor;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class AopTest5 {  
 public static void main(String[] args) {  
 ProxyFactory proxyFactory = new ProxyFactory();  
 proxyFactory.setTarget(new FundsService());  
 proxyFactory.addAdvisor(new DefaultPointcutAdvisor(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println(method);  
 }  
 }));  
 //创建代理对象  
 Object proxy = proxyFactory.getProxy();  
 System.out.println("代理对象的类型：" + proxy.getClass());  
 System.out.println("代理对象的父类：" + proxy.getClass().getSuperclass());  
 System.out.println("代理对象实现的接口列表");  
 for (Class<?> cf : proxy.getClass().getInterfaces()) {  
 System.out.println(cf);  
 }  
 }  
}

运行输出

代理对象的类型：class com.javacode2018.aop.demo4.FundsService$$EnhancerBySpringCGLIB$$ad14161a  
代理对象的父类：class com.javacode2018.aop.demo4.FundsService  
代理对象实现的接口列表  
interface org.springframework.aop.SpringProxy  
interface org.springframework.aop.framework.Advised  
interface org.springframework.cglib.proxy.Factory

输出中可以看出默认帮我们实现了3个接口[SpringProxy,Advised,Factory]

#### 案例2：代理接口

有接口的情况默认会通过jdk动态代理的方式生成代理，下面来看一下。

来个接口

package com.javacode2018.aop.demo6;  
  
public interface IService {  
 void say(String name);  
}

实现类

package com.javacode2018.aop.demo6;  
  
public class Service implements IService {  
 @Override  
 public void say(String name) {  
 System.out.println("hello：" + name);  
 }  
}

测试案例

package com.javacode2018.aop.demo6;  
  
import org.junit.Test;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class AopTest6 {  
 @Test  
 public void test1() {  
 Service target = new Service();  
  
 ProxyFactory proxyFactory = new ProxyFactory();  
 //设置需要被代理的对象  
 proxyFactory.setTarget(target);  
 //设置需要代理的接口  
 proxyFactory.addInterface(IService.class);  
  
 proxyFactory.addAdvice(new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println(method);  
 }  
 });  
  
 IService proxy = (IService) proxyFactory.getProxy();  
 System.out.println("代理对象的类型：" + proxy.getClass());  
 System.out.println("代理对象的父类：" + proxy.getClass().getSuperclass());  
 System.out.println("代理对象实现的接口列表");  
 for (Class<?> cf : proxy.getClass().getInterfaces()) {  
 System.out.println(cf);  
 }  
 //调用代理的方法  
 System.out.println("\n调用代理的方法");  
 proxy.say("spring aop");  
 }  
}

运行输出

代理对象的类型：class com.sun.proxy.$Proxy4  
代理对象的父类：class java.lang.reflect.Proxy  
代理对象实现的接口列表  
interface com.javacode2018.aop.demo6.IService  
interface org.springframework.aop.SpringProxy  
interface org.springframework.aop.framework.Advised  
interface org.springframework.core.DecoratingProxy  
  
调用代理的方法  
public abstract void com.javacode2018.aop.demo6.IService.say(java.lang.String)  
hello：spring aop

从第一行输出中可以看出是采用jdk动态代理方式创建的代理

第二行验证了，所有通过jdk动态代理方式创建的代理对象都是Proxy的子类

输出的接口列表中可以看出，默认帮我们实现了3个接口[SpringAop,Advised,DecoratingProxy]

#### 案例3：强制使用cglib代理

在案例2中加入下面代码，设置proxyTargetClass为true，会强制使用cglib代理。

//强制使用cglib代理  
proxyFactory.setProxyTargetClass(true);

#### 案例4：将代理暴露在threadLocal中

先来看一段代码，Service类中有2个方法，m1方法中会调用m2，通过aop代理对这个类创建了一个代理，通过代理来统计所有调用方法的耗时

package com.javacode2018.aop.demo7;  
  
class Service {  
 public void m1() {  
 System.out.println("m1");  
 this.m2();  
 }  
  
 public void m2() {  
 System.out.println("m2");  
 }  
}  
  
public class AopTest7 {  
 @Test  
 public void test1() {  
 Service target = new Service();  
  
 ProxyFactory proxyFactory = new ProxyFactory();  
 proxyFactory.setTarget(target);  
  
 proxyFactory.addAdvice(new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 long startTime = System.nanoTime();  
 Object result = invocation.proceed();  
 long endTime = System.nanoTime();  
 System.out.println(String.format("%s方法耗时(纳秒):%s", invocation.getMethod().getName(), endTime - startTime));  
 return result;  
 }  
 });  
  
 Service proxy = (Service) proxyFactory.getProxy();  
 proxy.m1();  
 }  
}

运行输出

m1  
m2  
m1方法耗时(纳秒):11056000

**为什么没有输出m2方法的耗时?**

原因：m2方法是在m1方法中通过this的方式来调用的，this实际上指向的是上面代码中的target对象。

那么我们如何能让此处的m2也能被增强，你需要通过代理来调用m2方法才可以，可以将代理对象暴露在threadLocal中，然后在m1方法中获取到threadLoca中的代理对象，通过代理对象来调用m2就可以了。

需要调整改动2处。

第1处：配置代理创建时，将其暴露出去

proxyFactory.setExposeProxy(true);

第2处：m1中调用m2的方法需要修改为下面这样

((Service)AopContext.currentProxy()).m2();

再次执行，结果如下

m1  
m2  
m2方法耗时(纳秒):51200  
m1方法耗时(纳秒):12794500

这个功能还是挺有用的，以后我估计大家是可以用到的。

### 总结

本文内容比较多，大家好好吸收一下，有问题的欢迎留言。

下一篇文章将深入详解介绍spring中如何将aop搞成自动化的，东西也是比较多的，敬请期待。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解

## Spring系列第33篇：ProxyFactoryBean创建AOP代理

### Aop相关阅读

阅读本文之前，需要先掌握下面3篇文章内容，不然会比较吃力。

1. Spring系列第15篇：代理详解（java动态代理&CGLIB代理)
2. Spring系列第30篇：jdk动态代理和cglib代理
3. Spring系列第31篇：Aop概念详解
4. Spring系列第32篇：AOP核心源码、原理详解

本文继续Aop。

### AOP创建代理的方式主要分为2大类

#### 手动方式

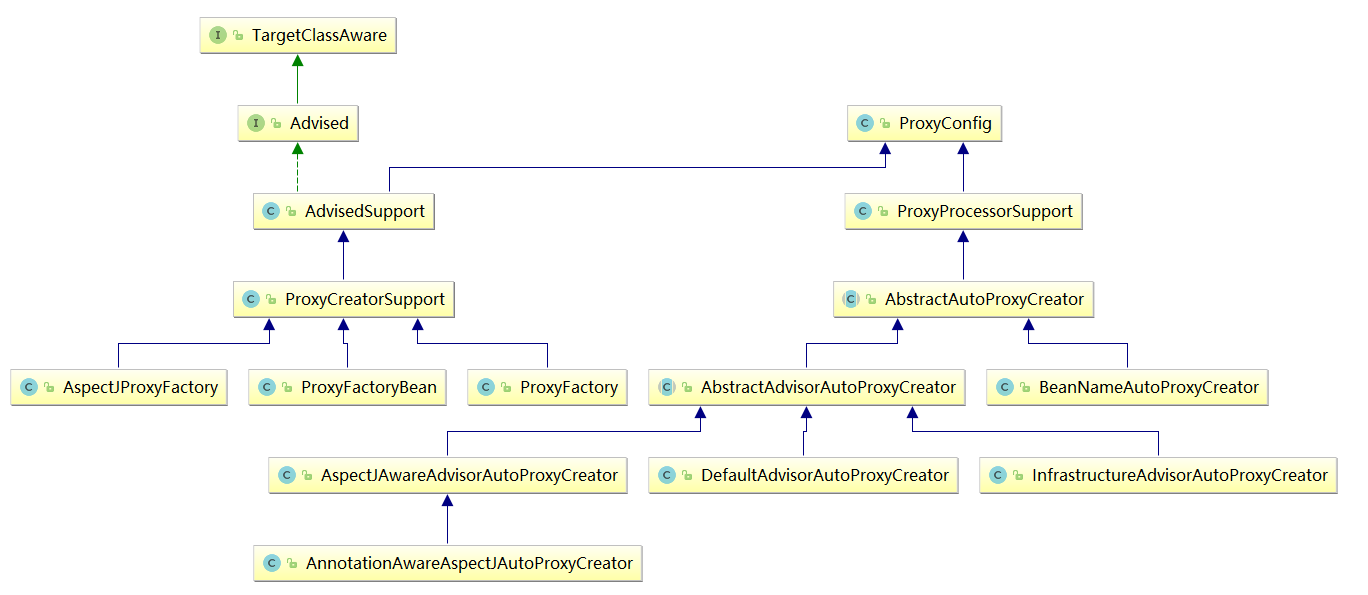
也称为手动的方式，需要通过硬编码一个个创建代理。

#### 自动化的方式

也称为批量的方式，批量的方式用在spring环境中，通过bean后置处理器来对符合条件的bean创建代理

手动的方式基本上是采用硬编码的方式，相对来说更灵活一些，可以脱离spring环境使用，而自动化的方式主要用在spring环境中，和spring集成起来更容易一些，更强大一些。

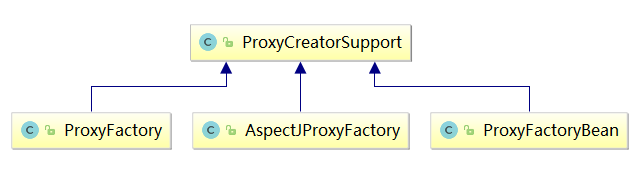
AOP创建代理相关的类



左边的ProxyCreatorSupport下面的都是手动的方式，有3个类。

右边的AbstractAutoProxyCreator下面挂的都是自动创建代理的方式，主要有5个实现类。

### 手动3种方式



#### ProxyFactory方式

这种是硬编码的方式，可以脱离spring直接使用，用到的比较多，自动化方式创建代理中都是依靠ProxyFactory来实现的，所以这种方式的原理大家一定要了解，上篇文章中已经有介绍过了，不清楚的可以去看一下：[Spring系列第32篇：AOP核心源码、原理详解](https://mp.weixin.qq.com/s/8B0cgwIREzhFWo7QNBLGug)

#### AspectJProxyFactory方式

AspectJ是一个面向切面的框架，是目前最好用，最方便的AOP框架，Spring将其集成进来了，通过Aspectj提供的一些功能实现aop代理非常方便，下篇文章将详解。

#### ProxyFactoryBean方式

Spring环境中给指定的bean创建代理的一种方式，本文主要介绍这个。

### ProxyFactoryBean

这个类实现了一个接口FactoryBean，FactoryBean不清楚的可以看一下：Spring系列第5篇：创建bean实例这些方式你们都知道？

ProxyFactoryBean就是通过FactoryBean的方式来给指定的bean创建一个代理对象。

创建代理，有3个信息比较关键：

1. 需要增强的功能，这个放在通知（Advice）中实现
2. 目标对象（target）：表示你需要给哪个对象进行增强
3. 代理对象（proxy）：将增强的功能和目标对象组合在一起，然后形成的一个代理对象，通过代理对象来访问目标对象，起到对目标对象增强的效果。

使用ProxyFactoryBean也是围绕着3部分来的，ProxyFactoryBean使用的步骤：

1.创建ProxyFactoryBean对象  
2.通过ProxyFactoryBean.setTargetName设置目标对象的bean名称，目标对象是spring容器中的一个bean  
3.通过ProxyFactoryBean。setInterceptorNames添加需要增强的通知  
4.将ProxyFactoryBean注册到Spring容器，假设名称为proxyBean  
5.从Spring查找名称为proxyBean的bean，这个bean就是生成好的代理对象

上案例。

来个类Service1

package com.javacode2018.aop.demo8.test1;  
  
public class Service1 {  
  
 public void m1() {  
 System.out.println("我是 m1 方法");  
 }  
  
 public void m2() {  
 System.out.println("我是 m2 方法");  
 }  
}

需求

在spring容器中注册上面这个类的bean，名称为service1，通过代理的方式来对这个bean进行增强，来2个通知

一个前置通知：在调用service1中的任意方法之前，输出一条信息：准备调用xxxx方法

一个环绕通知：复制统计所有方法的耗时。

下面是代码的实现

package com.javacode2018.aop.demo8.test1;  
  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactoryBean;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
@Configuration  
public class MainConfig1 {  
 //注册目标对象  
 @Bean  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 //注册一个前置通知  
 @Bean  
 public MethodBeforeAdvice beforeAdvice() {  
 MethodBeforeAdvice advice = new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("准备调用：" + method);  
 }  
 };  
 return advice;  
 }  
  
 //注册一个后置通知  
 @Bean  
 public MethodInterceptor costTimeInterceptor() {  
 MethodInterceptor methodInterceptor = new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 long starTime = System.nanoTime();  
 Object result = invocation.proceed();  
 long endTime = System.nanoTime();  
 System.out.println(invocation.getMethod() + ",耗时(纳秒)：" + (endTime - starTime));  
 return result;  
 }  
 };  
 return methodInterceptor;  
 }  
  
 //注册ProxyFactoryBean  
 @Bean  
 public ProxyFactoryBean service1Proxy() {  
 //1.创建ProxyFactoryBean  
 ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();  
 //2.设置目标对象的bean名称  
 proxyFactoryBean.setTargetName("service1");  
 //3.设置拦截器的bean名称列表，此处2个（advice1和advice2)  
 proxyFactoryBean.setInterceptorNames("beforeAdvice", "costTimeInterceptor");  
 return proxyFactoryBean;  
 }  
}

下面启动spring容器，并获取代理对象

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig1.class);  
//获取代理对象，代理对象bean的名称为注册ProxyFactoryBean的名称，即：service1Proxy  
Service1 bean = context.getBean("service1Proxy", Service1.class);  
System.out.println("----------------------");  
//调用代理的方法  
bean.m1();  
System.out.println("----------------------");  
//调用代理的方法  
bean.m2();

运行输出

----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m1()  
我是 m1 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m1(),耗时(纳秒)：8680400  
----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m2()  
我是 m2 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m2(),耗时(纳秒)：82400

从输出中可以看到，目标对象service1已经被增强了。

### ProxyFactoryBean中的interceptorNames

interceptorNames用来指定拦截器的bean名称列表，常用的2种方式。

* 批量的方式
* 非批量的方式

#### 批量的方式

##### 使用方法

proxyFactoryBean.setInterceptorNames("需要匹配的bean名称\*");

需要匹配的bean名称后面跟上一个\*，可以用来批量的匹配，如：interceptor\*，此时spring会从容器中找到下面2中类型的所有bean，bean名称以interceptor开头的将作为增强器

org.springframework.aop.Advisor  
org.aopalliance.intercept.Interceptor

这个地方使用的时候需要注意，批量的方式注册的时候，如果增强器的类型不是上面2种类型的，比如下面3种类型通知，我们需要将其包装为Advisor才可以，而MethodInterceptor是Interceptor类型的，可以不用包装为Advisor类型的。

MethodBeforeAdvice（方法前置通知）  
AfterReturningAdvice（方法后置通知）  
ThrowsAdvice（异常通知）

下面来个案例感受一下。

##### 案例

下面批量注册2个增强器。

package com.javacode2018.aop.demo8.test2;  
  
import com.javacode2018.aop.demo8.test1.Service1;  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.aop.Advisor;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactoryBean;  
import org.springframework.aop.support.DefaultPointcutAdvisor;  
import org.springframework.context.annotation.Bean;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class MainConfig2 {  
 //注册目标对象  
 @Bean  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 //定义一个增强器：interceptor1，内部是一个前置通知，需要将其包装为Advisor类型的  
 @Bean  
 public Advisor interceptor1() {  
 MethodBeforeAdvice advice = new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("准备调用：" + method);  
 }  
 };  
 DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();  
 advisor.setAdvice(advice);  
 return advisor;  
 }  
  
 //定义一个增强器：interceptor2  
 @Bean  
 public MethodInterceptor interceptor2() {  
 MethodInterceptor methodInterceptor = new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 long starTime = System.nanoTime();  
 Object result = invocation.proceed();  
 long endTime = System.nanoTime();  
 System.out.println(invocation.getMethod() + ",耗时(纳秒)：" + (endTime - starTime));  
 return result;  
 }  
 };  
 return methodInterceptor;  
 }  
  
 //注册ProxyFactoryBean  
 @Bean  
 public ProxyFactoryBean service1Proxy() {  
 //1.创建ProxyFactoryBean  
 ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();  
 //2.设置目标对象的bean名称  
 proxyFactoryBean.setTargetName("service1");  
 //3.设置拦截器的bean名称列表，此处批量注册  
 proxyFactoryBean.setInterceptorNames("interceptor\*"); //@1  
 return proxyFactoryBean;  
 }  
}

上面定义了2个增强器：

interceptor1：前置通知，包装为Advisor类型了

interceptor2：环绕通知，MethodInterceptor类型的

测试代码

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig2.class);  
 //获取代理对象，代理对象bean的名称为注册ProxyFactoryBean的名称，即：service1Proxy  
 Service1 bean = context.getBean("service1Proxy", Service1.class);  
 System.out.println("----------------------");  
 //调用代理的方法  
 bean.m1();  
 System.out.println("----------------------");  
 //调用代理的方法  
 bean.m2();  
}

运行输出

----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m1()  
我是 m1 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m1(),耗时(纳秒)：10326200  
----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m2()  
我是 m2 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m2(),耗时(纳秒)：52000

#### 非批量的方式

##### 用法

非批量的方式，需要注册多个增强器，需明确的指定多个增强器的bean名称，多个增强器按照参数中指定的顺序执行，如

proxyFactoryBean.setInterceptorNames("advice1","advice2");

advice1、advice2对应的bean类型必须为下面列表中指定的类型，类型这块比匹配的方式范围广一些

MethodBeforeAdvice（方法前置通知）  
AfterReturningAdvice（方法后置通知）  
ThrowsAdvice（异常通知）  
org.aopalliance.intercept.MethodInterceptor（环绕通知）  
org.springframework.aop.Advisor（顾问）

下面来个案例。

##### 案例

这次给service1来3个通知：前置、环绕、后置

package com.javacode2018.aop.demo8.test3;  
  
import com.javacode2018.aop.demo8.test1.Service1;  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.aop.AfterReturningAdvice;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.framework.ProxyFactoryBean;  
import org.springframework.context.annotation.Bean;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class MainConfig3 {  
 //注册目标对象  
 @Bean  
 public Service1 service1() {  
 return new Service1();  
 }  
  
 //定义一个前置通知  
 @Bean  
 public MethodBeforeAdvice methodBeforeAdvice() {  
 MethodBeforeAdvice methodBeforeAdvice = new MethodBeforeAdvice() {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("准备调用：" + method);  
 }  
 };  
 return methodBeforeAdvice;  
 }  
  
 //定义一个环绕通知  
 @Bean  
 public MethodInterceptor methodInterceptor() {  
 MethodInterceptor methodInterceptor = new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 long starTime = System.nanoTime();  
 Object result = invocation.proceed();  
 long endTime = System.nanoTime();  
 System.out.println(invocation.getMethod() + ",耗时(纳秒)：" + (endTime - starTime));  
 return result;  
 }  
 };  
 return methodInterceptor;  
 }  
  
 //定义一个后置通知  
 @Bean  
 public AfterReturningAdvice afterReturningAdvice() {  
 AfterReturningAdvice afterReturningAdvice = new AfterReturningAdvice() {  
 @Override  
 public void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println(method + "，执行完毕!");  
 }  
 };  
 return afterReturningAdvice;  
 }  
  
 //注册ProxyFactoryBean  
 @Bean  
 public ProxyFactoryBean service1Proxy() {  
 //1.创建ProxyFactoryBean  
 ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();  
 //2.设置目标对象的bean名称  
 proxyFactoryBean.setTargetName("service1");  
 //3.设置拦截器的bean名称列表，此处批量注册  
 proxyFactoryBean.setInterceptorNames("methodBeforeAdvice", "methodInterceptor", "afterReturningAdvice");  
 return proxyFactoryBean;  
 }  
}

测试代码

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig3.class);  
 //获取代理对象，代理对象bean的名称为注册ProxyFactoryBean的名称，即：service1Proxy  
 Service1 bean = context.getBean("service1Proxy", Service1.class);  
 System.out.println("----------------------");  
 //调用代理的方法  
 bean.m1();  
 System.out.println("----------------------");  
 //调用代理的方法  
 bean.m2();  
}

运行输出

----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m1()  
我是 m1 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m1()，执行完毕!  
public void com.javacode2018.aop.demo8.test1.Service1.m1(),耗时(纳秒)：12724100  
----------------------  
准备调用：public void com.javacode2018.aop.demo8.test1.Service1.m2()  
我是 m2 方法  
public void com.javacode2018.aop.demo8.test1.Service1.m2()，执行完毕!  
public void com.javacode2018.aop.demo8.test1.Service1.m2(),耗时(纳秒)：76700

### 源码解析

重点在于下面这个方法

org.springframework.aop.framework.ProxyFactoryBean#getObject

源码：

public Object getObject() throws BeansException {  
 //初始化advisor(拦截器)链  
 initializeAdvisorChain();  
 //是否是单例  
 if (isSingleton()) {  
 //创建单例代理对象  
 return getSingletonInstance();  
 }  
 else {  
 //创建多例代理对象  
 return newPrototypeInstance();  
 }  
}

initializeAdvisorChain方法，用来初始化advisor(拦截器)链，是根据interceptorNames配置，找到spring容器中符合条件的拦截器，将其放入创建aop代理的配置中

private synchronized void initializeAdvisorChain() throws AopConfigException, BeansException {  
 if (!ObjectUtils.isEmpty(this.interceptorNames)) {  
 // 轮询 interceptorNames  
 for (String name : this.interceptorNames) {  
 //批量注册的方式：判断name是否以\*结尾  
 if (name.endsWith(GLOBAL\_SUFFIX)) {  
 //@1：从容器中匹配查找匹配的增强器，将其添加到aop配置中  
 addGlobalAdvisor((ListableBeanFactory) this.beanFactory,  
 name.substring(0, name.length() - GLOBAL\_SUFFIX.length()));  
 }  
 else {  
 //非匹配的方式：按照name查找bean，将其包装为Advisor丢到aop配置中  
 Object advice;  
 //从容器中查找bean  
 advice = this.beanFactory.getBean(name);  
 //@2：将advice添加到拦截器列表中  
 addAdvisorOnChainCreation(advice, name);  
 }  
 }  
 }  
}

@1：addGlobalAdvisor批量的方式Advisor，看一下源码，比较简单

/\*\*  
 \* 添加所有全局拦截器和切入点，  
 \* 容器中所有类型为Advisor/Interceptor的bean，bean名称prefix开头的都会将其添加到拦截器链中  
 \*/  
private void addGlobalAdvisor(ListableBeanFactory beanFactory, String prefix) {  
 //获取容器中所有类型为Advisor的bean  
 String[] globalAdvisorNames =  
 BeanFactoryUtils.beanNamesForTypeIncludingAncestors(beanFactory, Advisor.class);  
 //获取容器中所有类型为Interceptor的bean  
 String[] globalInterceptorNames =  
 BeanFactoryUtils.beanNamesForTypeIncludingAncestors(beanFactory, Interceptor.class);  
 List<Object> beans = new ArrayList<>(globalAdvisorNames.length + globalInterceptorNames.length);  
 Map<Object, String> names = new HashMap<>(beans.size());  
 for (String name : globalAdvisorNames) {  
 Object bean = beanFactory.getBean(name);  
 beans.add(bean);  
 names.put(bean, name);  
 }  
 for (String name : globalInterceptorNames) {  
 Object bean = beanFactory.getBean(name);  
 beans.add(bean);  
 names.put(bean, name);  
 }  
 //对beans进行排序，可以实现Ordered接口，排序规则：order asc  
 AnnotationAwareOrderComparator.sort(beans);  
 for (Object bean : beans) {  
 String name = names.get(bean);  
 //判断bean是否已prefix开头  
 if (name.startsWith(prefix)) {  
 //将其添加到拦截器链中  
 addAdvisorOnChainCreation(bean, name);  
 }  
 }  
}

@2：addAdvisorOnChainCreation

private void addAdvisorOnChainCreation(Object next, String name) {  
 //namedBeanToAdvisor用来将bean转换为advisor  
 Advisor advisor = namedBeanToAdvisor(next);  
 //将advisor添加到拦截器链中  
 addAdvisor(advisor);  
}

namedBeanToAdvisor方法

private AdvisorAdapterRegistry advisorAdapterRegistry = new DefaultAdvisorAdapterRegistry();  
  
private Advisor namedBeanToAdvisor(Object next) {  
 //将对象包装为Advisor对象  
 return this.advisorAdapterRegistry.wrap(next);  
}

对AdvisorAdapterRegistry不清楚的，看一下上一篇文章：[Spring系列第32篇：AOP核心源码、原理详解](https://mp.weixin.qq.com/s/8B0cgwIREzhFWo7QNBLGug)

advisorAdapterRegistry#wrap方法将adviceObject包装为Advisor对象，代码如下，比较简单

public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {  
 if (adviceObject instanceof Advisor) {  
 return (Advisor) adviceObject;  
 }  
 if (!(adviceObject instanceof Advice)) {  
 throw new UnknownAdviceTypeException(adviceObject);  
 }  
 Advice advice = (Advice) adviceObject;  
 if (advice instanceof MethodInterceptor) {  
 return new DefaultPointcutAdvisor(advice);  
 }  
 for (AdvisorAdapter adapter : this.adapters) {  
 if (adapter.supportsAdvice(advice)) {  
 return new DefaultPointcutAdvisor(advice);  
 }  
 }  
 throw new UnknownAdviceTypeException(advice);  
}

### 总结

1. spring中创建代理主要分为2种：手动方式和自动化的方式
2. 手动方式采用硬编码的方式，一次只能给一个目标对象创建代理对象，相对来说灵活一下，对开发者来说更灵活一些，通常可以独立spring环境使用；自动化的方式主要在spring环境中使用，通常是匹配的方式来为符合条件的目标bean创建代理，使用起来更简单一些
3. 本文介绍的ProxyFactoryBean用来在spring环境中给指定的bean创建代理对象，用到的不是太多，大家可以作为了解即可

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解

## Spring系列第34篇：@Aspect中@Pointcut 12种用法

**本文主要内容：掌握@Pointcut的12种用法。**

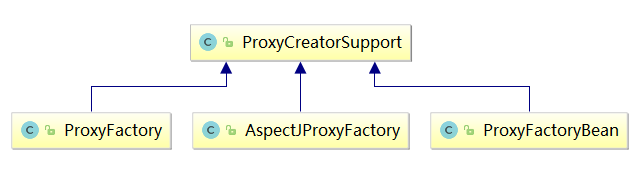
### Aop相关阅读

阅读本文之前，需要先掌握下面几篇篇文章内容，不然会比较吃力。

1. Spring系列第15篇：代理详解（java动态代理&CGLIB代理)
2. Spring系列第30篇：jdk动态代理和cglib代理
3. Spring系列第31篇：Aop概念详解
4. Spring系列第32篇：AOP核心源码、原理详解
5. Spring系列第33篇：ProxyFactoryBean创建AOP代理

本文继续AOP，目前手动Aop中三种方式已经介绍2种了，本文将介绍另外一种：AspectJProxyFactory，可能大家对这个比较陌生，但是@Aspect这个注解大家应该很熟悉吧，通过这个注解在spring环境中实现aop特别的方便。

而AspectJProxyFactory这个类可以通过解析@Aspect标注的类来生成代理aop代理对象，对开发者来说，使创建代理变的更简洁了。



### 先了解几个概念

文中会涉及几个概念，先了解一下。

#### target

用来表示目标对象，即需要通过aop来增强的对象。

#### proxy

代理对象，target通过aop增强之后生成的代理对象。

### AspectJ

#### AspectJ是什么?

AspectJ是一个面向切面的框架，是目前最好用，最方便的AOP框架，和spring中的aop可以集成在一起使用，通过Aspectj提供的一些功能实现aop代理变得非常方便。

#### AspectJ使用步骤

1.创建一个类，使用@Aspect标注  
2.@Aspect标注的类中，通过@Pointcut定义切入点  
3.@Aspect标注的类中，通过AspectJ提供的一些通知相关的注解定义通知  
4.使用AspectJProxyFactory结合@Ascpect标注的类，来生成代理对象

先来个案例，感受一下AspectJ是多么的方便。

来个类

package com.javacode2018.aop.demo9.test1;  
  
public class Service1 {  
  
 public void m1() {  
 System.out.println("我是 m1 方法");  
 }  
  
 public void m2() {  
 System.out.println(10 / 0);  
 System.out.println("我是 m2 方法");  
 }  
}

通过AspectJ来对Service1进行增强，来2个通知，一个前置通知，一个异常通知，这2个通知需要对Service1中的所有方法生效，实现如下：

package com.javacode2018.aop.demo9.test1;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.AfterThrowing;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
//@1：这个类需要使用@Aspect进行标注  
@Aspect  
public class Aspect1 {  
  
 //@2：定义了一个切入点，可以匹配Service1中所有方法  
 @Pointcut("execution(\* com.javacode2018.aop.demo9.test1.Service1.\*(..))")  
 public void pointcut1() {  
 }  
  
 //@3：定义了一个前置通知，这个通知对刚刚上面我们定义的切入点中的所有方法有效  
 @Before(value = "pointcut1()")  
 public void before(JoinPoint joinPoint) {  
 //输出连接点的信息  
 System.out.println("前置通知，" + joinPoint);  
 }  
  
 //@4：定义了一个异常通知，这个通知对刚刚上面我们定义的切入点中的所有方法有效  
 @AfterThrowing(value = "pointcut1()", throwing = "e")  
 public void afterThrowing(JoinPoint joinPoint, Exception e) {  
 //发生异常之后输出异常信息  
 System.out.println(joinPoint + ",发生异常：" + e.getMessage());  
 }  
  
}

@1：类上使用@Aspect标注

@2：通过@Pointcut注解标注在方法上面，用来定义切入点

@3：使用@Before标注在方法上面，定义了一个前置通知，通过value引用了上面已经定义的切入点，表示这个通知会对Service1中的所有方法生效，在通知中可以通过这个类名.方法名()引用@Pointcut定义的切入点，表示这个通知对这些切入点有效，若@Before和@Pointcut在一个类的时候，直接通过方法名()引用当前类中定义的切入点

@4：这个使用@AfterThrowing定义了一个异常通知，也是对通过value引用了上面已经定义的切入点，表示这个通知会对Service1中的所有方法生效，若Service1中的方法抛出了Exception类型的异常，都会回调afterThrowing方法。

来个测试类

package com.javacode2018.aop.demo9;  
  
import com.javacode2018.aop.demo9.test1.Aspect1;  
import com.javacode2018.aop.demo9.test1.Service1;  
import org.junit.Test;  
import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;  
  
public class AopTest9 {  
 @Test  
 public void test1() {  
 try {  
 //对应目标对象  
 Service1 target = new Service1();  
 //创建AspectJProxyFactory对象  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 //设置被代理的目标对象  
 proxyFactory.setTarget(target);  
 //设置标注了@Aspect注解的类  
 proxyFactory.addAspect(Aspect1.class);  
 //生成代理对象  
 Service1 proxy = proxyFactory.getProxy();  
 //使用代理对象  
 proxy.m1();  
 proxy.m2();  
 } catch (Exception e) {  
 }  
 }  
}

运行输出

前置通知，execution(void com.javacode2018.aop.demo9.test1.Service1.m1())  
我是 m1 方法  
前置通知，execution(void com.javacode2018.aop.demo9.test1.Service1.m2())  
execution(void com.javacode2018.aop.demo9.test1.Service1.m2()),发生异常：/ by zero

使用是不是特方便。

#### AspectJProxyFactory原理

@Aspect标注的类上，这个类中，可以通过通过@Pointcut来定义切入点，可以通过@Before、@Around、@After、@AfterRunning、@AfterThrowing标注在方法上来定义通知，定义好了之后，将@Aspect标注的这个类交给AspectJProxyFactory来解析生成Advisor链，进而结合目标对象一起来生成代理对象，大家可以去看一下源码，比较简单，这里就不多解释了。

本文的重点在@Aspect标注的类上，@Aspect中有2个关键点比较重要

* @Pointcut：标注在方法上，用来定义切入点，有11种用法，本文主要讲解这11种用法。
* @Aspect类中定义通知：可以通过@Before、@Around、@After、@AfterRunning、@AfterThrowing标注在方法上来定义通知，这个下一篇介绍。

### @Pointcut的12种用法

#### 作用

用来标注在方法上来定义切入点。

#### 定义

格式：@ 注解(value=“表达标签 (表达式格式)”)

如：

@Pointcut("execution(\* com.javacode2018.aop.demo9.test1.Service1.\*(..))")

#### 表达式标签（10种）

* execution：用于匹配方法执行的连接点
* within：用于匹配指定类型内的方法执行
* this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也类型匹配
* target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配
* args：用于匹配当前执行的方法传入的参数为指定类型的执行方法
* @within：用于匹配所以持有指定注解类型内的方法
* @target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解
* @args：用于匹配当前执行的方法传入的参数持有指定注解的执行
* @annotation：用于匹配当前执行方法持有指定注解的方法
* bean：Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法

**10种标签组成了12种用法**

#### 1、execution

使用execution(方法表达式)匹配方法执行。

##### execution格式

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

* 其中带 ?号的 modifiers-pattern?，declaring-type-pattern?，hrows-pattern?是可选项
* ret-type-pattern,name-pattern, parameters-pattern是必选项
* modifier-pattern? 修饰符匹配，如public 表示匹配公有方法
* ret-type-pattern 返回值匹配，\* 表示任何返回值，全路径的类名等
* declaring-type-pattern? 类路径匹配
* name-pattern 方法名匹配，\* 代表所有，set\*，代表以set开头的所有方法
* (param-pattern) 参数匹配，指定方法参数(声明的类型)，(..)代表所有参数，(\*,String)代表第一个参数为任何值,第二个为String类型，(..,String)代表最后一个参数是String类型
* throws-pattern? 异常类型匹配

##### 举例说明

|  |  |
| --- | --- |
| **表达式** | **描述** |
| public \*.\*(..) | 任何公共方法的执行 |
| \* com.javacode2018..IPointcutService.\*() | com.javacode2018包及所有子包下IPointcutService接口中的任何无参方法 |
| \* com.javacode2018..\*.\*(..) | com.javacode2018包及所有子包下任何类的任何方法 |
| \* com.javacode2018..IPointcutService.() | com.javacode2018包及所有子包下IPointcutService接口的任何只有一个参数方法 |
| \* com.javacode2018..IPointcutService+.\*() | com.javacode2018包及所有子包下IPointcutService接口及子类型的的任何无参方法 |
| \* Service1.\*(String) | 匹配Service1中只有1个参数的且参数类型是String的方法 |
| \* Service1.\*(\*,String) | 匹配Service1中只有2个参数的且第二个参数类型是String的方法 |
| \* Service1.\*(..,String) | 匹配Service1中最后1个参数类型是String的方法 |

##### 类型匹配语法

很多地方会按照类型的匹配，先来说一下类型匹配的语法。

首先让我们来了解下AspectJ类型匹配的通配符：

* **\***：匹配任何数量字符
* **..**：匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数（0个或者多个参数）
* **+：**匹配指定类型及其子类型；仅能作为后缀放在类型模式后边

|  |  |
| --- | --- |
| 表达式 | 说明 |
| java.lang.String | 匹配String类型 |
| java.\*.String | 匹配java包下的任何一级子包下的String类型，如匹配java.lang.String，但不匹配java.lang.ss.String |
| java..\* | 匹配java包及任何子包下的任何类型，如匹配java.lang.String、java.lang.annotation.Annotation |
| java.lang.\*ing | 匹配任何java.lang包下的以ing结尾的类型 |
| java.lang.Number+ | 匹配java.lang包下的任何Number类型及其子类型，如匹配java.lang.Number，也匹配java.lang.Integer、java.math.BigInteger |

#### 2、within

##### 用法

within(类型表达式)：目标对象target的类型是否和within中指定的类型匹配

|  |  |
| --- | --- |
| **表达式** | **描述** |
| within(com.javacode2018..\*) | com.javacode2018包及子包下的任何方法执行 |
| within(com.javacode2018..IPointcutService+) | com.javacode2018包或所有子包下IPointcutService类型及子类型的任何方法 |
| within(com.javacode2018.Service1) | 匹配类com.javacode2018.Service1中定义的所有方法，不包含其子类中的方法 |

##### 匹配原则

target.getClass().equals(within表达式中指定的类型)

##### 案例

有2个类，父子关系

父类C1

package com.javacode2018.aop.demo9.test2;  
  
public class C1 {  
 public void m1() {  
 System.out.println("我是m1");  
 }  
  
 public void m2() {  
 System.out.println("我是m2");  
 }  
}

子类C2

package com.javacode2018.aop.demo9.test2;  
  
public class C2 extends C1 {  
 @Override  
 public void m2() {  
 super.m2();  
 }  
  
 public void m3() {  
 System.out.println("我是m3");  
 }  
}

来个Aspect类

package com.javacode2018.aop.demo9.test2;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest2 {  
  
 @Pointcut("within(C1)") //@1  
 public void pc() {  
 }  
  
 @Before("pc()") //@2  
 public void beforeAdvice(JoinPoint joinpoint) {  
 System.out.println(joinpoint);  
 }  
  
}

注意@1匹配的类型是C1，也就是说被代理的对象的类型必须是C1类型的才行，需要和C1完全匹配

下面我们对C2创建代理

@Test  
public void test2(){  
 C2 target = new C2();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest2.class);  
  
 C2 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 proxy.m2();  
 proxy.m3();  
}

运行输出

我是m1  
我是m2  
我是m3

原因是目标对象是C2类型的，C2虽然是C1的子类，但是within中表达式指定的是要求类型必须是C1类型的才匹配。

如果将within表达式修改为下面任意一种就可以匹配了

@Pointcut("within(C1+)")   
@Pointcut("within(C2)")

再次运行输出

execution(void com.javacode2018.aop.demo9.test2.C1.m1())  
我是m1  
execution(void com.javacode2018.aop.demo9.test2.C2.m2())  
我是m2  
execution(void com.javacode2018.aop.demo9.test2.C2.m3())  
我是m3

#### 3、this

##### 用法

this(类型全限定名)：通过aop创建的代理对象的类型是否和this中指定的类型匹配；注意判断的目标是代理对象；this中使用的表达式必须是类型全限定名，不支持通配符。

##### 匹配原则

如:this(x)，则代理对象proxy满足下面条件时会匹配  
x.getClass().isAssignableFrom(proxy.getClass());

##### 案例

来个接口

package com.javacode2018.aop.demo9.test3;  
  
public interface I1 {  
 void m1();  
}

来个实现类

package com.javacode2018.aop.demo9.test3;  
  
public class Service3 implements I1 {  
  
 @Override  
 public void m1() {  
 System.out.println("我是m1");  
 }  
  
}

来个@Aspect类

package com.javacode2018.aop.demo9.test3;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest3 {  
  
 //@1：匹配proxy是Service3类型的所有方法  
 @Pointcut("this(Service3)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinpoint) {  
 System.out.println(joinpoint);  
 }  
  
}

测试代码

@Test  
public void test3() {  
 Service3 target = new Service3();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 //获取目标对象上的接口列表  
 Class<?>[] allInterfaces = ClassUtils.getAllInterfaces(target);  
 //设置需要代理的接口  
 proxyFactory.setInterfaces(allInterfaces);  
 proxyFactory.addAspect(AspectTest3.class);  
 //获取代理对象  
 Object proxy = proxyFactory.getProxy();  
 //调用代理对象的方法  
 ((I1) proxy).m1();  
  
 System.out.println("proxy是否是jdk动态代理对象：" + AopUtils.isJdkDynamicProxy(proxy));  
 System.out.println("proxy是否是cglib代理对象：" + AopUtils.isCglibProxy(proxy));  
 //判断代理对象是否是Service3类型的  
 System.out.println(Service3.class.isAssignableFrom(proxy.getClass()));  
}

运行输出

我是m1  
proxy是否是jdk动态代理对象：true  
proxy是否是cglib代理对象：false  
false

从输出中可以看出m1方法没有被增强，原因：this表达式要求代理对象必须是Service3类型的，输出中可以看出代理对象并不是Service3类型的，此处代理对象proxy是使用jdk动态代理生成的。

我们可以将代码调整一下，使用cglib来创建代理

proxyFactory.setProxyTargetClass(true);

再次运行，会发现m2被拦截了，结果如下

execution(void com.javacode2018.aop.demo9.test3.Service3.m1())  
我是m1  
proxy是否是jdk动态代理对象：false  
proxy是否是cglib代理对象：true  
true

#### 4、target

##### 用法

target(类型全限定名)：判断目标对象的类型是否和指定的类型匹配；注意判断的是目标对象的类型；表达式必须是类型全限定名，不支持通配符。

##### 匹配原则

如:target(x)，则目标对象target满足下面条件时会匹配  
x.getClass().isAssignableFrom(target.getClass());

##### 案例

package com.javacode2018.aop.demo9.test4;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest4 {  
  
 //@1：目标类型必须是Service3类型的  
 @Pointcut("target(com.javacode2018.aop.demo9.test3.Service3)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinpoint) {  
 System.out.println(joinpoint);  
 }  
  
}

测试代码

@Test  
public void test4() {  
 Service3 target = new Service3();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setProxyTargetClass(true);  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest4.class);  
 //获取代理对象  
 Object proxy = proxyFactory.getProxy();  
 //调用代理对象的方法  
 ((I1) proxy).m1();  
 //判断target对象是否是Service3类型的  
 System.out.println(Service3.class.isAssignableFrom(target.getClass()));  
}

运行输出

execution(void com.javacode2018.aop.demo9.test3.Service3.m1())  
我是m1  
true

##### within、this、target对比

|  |  |  |
| --- | --- | --- |
| 表达式标签 | 判断的对象 | 判断规则(x：指表达式中指定的类型) |
| **within** | target对象 | target.getClass().equals(表达式中指定的类型) |
| **this** | proxy对象 | x.getClass().isAssignableFrom(proxy.getClass()); |
| **target** | target对象 | x.getClass().isAssignableFrom(target.getClass()); |

#### 5、args

##### 用法

args(参数类型列表)匹配当前执行的方法传入的参数是否为args中指定的类型；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，不支持通配符；args属于动态切入点，也就是执行方法的时候进行判断的，这种切入点开销非常大，非特殊情况最好不要使用。

##### 举例说明

|  |  |
| --- | --- |
| **表达式** | **描述** |
| args(String) | 匹配只有一个参数且传入的参数类型是String类型的方法 |
| args(\*,String) | 匹配只有2个参数的且第2个参数类型是String的方法 |
| args(..,String) | 匹配最后1个参数类型是String的方法 |

##### 案例

下面的m1方法参数是Object类型的。

package com.javacode2018.aop.demo9.test5;  
  
public class Service5 {  
 public void m1(Object object) {  
 System.out.println("我是m1方法,参数：" + object);  
 }  
}

Aspect类

package com.javacode2018.aop.demo9.test5;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
import java.util.Arrays;  
import java.util.stream.Collectors;  
  
@Aspect  
public class AspectTest5 {  
 //@1：匹配只有1个参数其类型是String类型的  
 @Pointcut("args(String)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinpoint) {  
 System.out.println("请求参数：" + Arrays.stream(joinpoint.getArgs()).collect(Collectors.toList()));  
 }  
}

测试代码，调用2次m1方法，第一次传入一个String类型的，第二次传入一个int类型的，看看效果

@Test  
public void test5() {  
 Service5 target = new Service5();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest5.class);  
 Service5 proxy = proxyFactory.getProxy();  
 proxy.m1("路人");  
 proxy.m1(100);  
}

运行输出

请求参数：[路人]  
我是m1方法,参数：路人  
我是m1方法,参数：100

输出中可以看出，m1第一次调用被增强了，第二次没有被增强。

**args会在调用的过程中对参数实际的类型进行匹配，比较耗时，慎用。**

#### 6、@within

##### 用法

@within(注解类型)：匹配指定的注解内定义的方法。

##### 匹配规则

调用目标方法的时候，通过java中Method.getDeclaringClass()获取当前的方法是哪个类中定义的，然后会看这个类上是否有指定的注解。

被调用的目标方法Method对象.getDeclaringClass().getAnnotation(within中指定的注解类型) != null

来看3个案例。

##### 案例1

**目标对象上有@within中指定的注解，这种情况时，目标对象的所有方法都会被拦截。**

###### 来个注解

package com.javacode2018.aop.demo9.test9;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface Ann9 {  
}

###### 来个目标类，用@Ann9标注

package com.javacode2018.aop.demo9.test9;  
  
@Ann9  
public class S9 {  
 public void m1() {  
 System.out.println("我是m1方法");  
 }  
}

###### 来个Aspect类

package com.javacode2018.aop.demo9.test9;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest9 {  
 /\*\*  
 \* 定义目标方法的类上有Ann9注解  
 \*/  
 @Pointcut("@within(Ann9)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试代码

@Test  
public void test9() {  
 S9 target = new S9();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest9.class);  
 S9 proxy = proxyFactory.getProxy();  
 proxy.m1();  
}

m1方法在类S9中定义的，S9上面有Ann9注解，所以匹配成功

###### 运行输出

execution(void com.javacode2018.aop.demo9.test9.S9.m1())  
我是m1方法

##### 案例2

**定义注解时未使用@Inherited，说明子类无法继承父类上的注解**，这个案例中我们将定义一个这样的注解，将注解放在目标类的父类上，来看一下效果。

###### 定义注解Ann10

package com.javacode2018.aop.demo9.test10;  
  
import java.lang.annotation.\*;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Inherited  
public @interface Ann10 {  
}

###### 来2个父子类

注意：

S10Parent为父类，并且使用了Anno10注解，内部定义了2个方法大家注意一下

而S10位代理的目标类，继承了S10Parent，内部重写了父类的m2方法，并且又新增了一个m3方法

package com.javacode2018.aop.demo9.test10;  
  
@Ann10  
class S10Parent {  
  
 public void m1() {  
 System.out.println("我是S10Parent.m1()方法");  
 }  
  
 public void m2() {  
 System.out.println("我是S10Parent.m2()方法");  
 }  
}  
  
public class S10 extends S10Parent {  
  
 @Override  
 public void m2() {  
 System.out.println("我是S10.m2()方法");  
 }  
  
 public void m3() {  
 System.out.println("我是S10.m3()方法");  
 }  
}

###### 来个Aspect类

package com.javacode2018.aop.demo9.test10;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest10 {  
 //匹配目标方法声明的类上有@Anno10注解  
 @Pointcut("@within(com.javacode2018.aop.demo9.test10.Ann10)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试用例

S10为目标类，依次执行代理对象的m1、m2、m3方法，最终会调用目标类target中对应的方法。

@Test  
public void test10() {  
 S10 target = new S10();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest10.class);  
 S10 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 proxy.m2();  
 proxy.m3();  
}

###### 运行输出

execution(void com.javacode2018.aop.demo9.test10.S10Parent.m1())  
我是S10Parent.m1()方法  
我是S10.m2()方法  
我是S10.m3()方法

###### 分析结果

从输出中可以看出，只有m1方法被拦截了，其他2个方法没有被拦截。

确实是这样的，m1方法的是由S10Parent定义的，这个类上面有Ann10注解。

而m2方法虽然也在S10Parent中定义了，但是这个方法被子类S10重写了，所以调用目标对象中的m2方法的时候，此时发现m2方法是由S10定义的，而S10.class.getAnnotation(Ann10.class)为空，所以这个方法不会被拦截。

同样m3方法也是S10中定义的，也不会被拦截。

##### 案例3

对案例2进行改造，在注解的定义上面加上@Inherited，此时子类可以继承父类的注解，此时3个方法都会被拦截了。

下面上代码，下面代码为案例2代码的一个拷贝，不同地方只是注解的定义上多了@Inherited

###### 定义注解Ann11

import java.lang.annotation.\*;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Inherited  
public @interface Ann11 {  
}

###### 2个父子类

package com.javacode2018.aop.demo9.test11;  
  
@Ann11  
class S11Parent {  
  
 public void m1() {  
 System.out.println("我是S11Parent.m1()方法");  
 }  
  
 public void m2() {  
 System.out.println("我是S11Parent.m2()方法");  
 }  
}  
  
public class S11 extends S11Parent {  
  
 @Override  
 public void m2() {  
 System.out.println("我是S11.m2()方法");  
 }  
  
 public void m3() {  
 System.out.println("我是S11.m3()方法");  
 }  
}

###### Aspect类

package com.javacode2018.aop.demo9.test11;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest11 {  
  
 @Pointcut("@within(com.javacode2018.aop.demo9.test11.Ann11)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试用例

@Test  
public void test11() {  
 S11 target = new S11();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest11.class);  
 S11 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 proxy.m2();  
 proxy.m3();  
}

###### 运行输出

execution(void com.javacode2018.aop.demo9.test11.S11Parent.m1())  
我是S11Parent.m1()方法  
execution(void com.javacode2018.aop.demo9.test11.S11.m2())  
我是S11.m2()方法  
execution(void com.javacode2018.aop.demo9.test11.S11.m3())  
我是S11.m3()方法

这次3个方法都被拦截了。

###### 

#### 7、@target

##### 用法

@target(注解类型)：判断目标对象target类型上是否有指定的注解；@target中注解类型也必须是全限定类型名。

##### 匹配规则

target.class.getAnnotation(指定的注解类型) != null

2种情况可以匹配

* 注解直接标注在目标类上
* 注解标注在父类上，但是注解必须是可以继承的，即定义注解的时候，需要使用@Inherited标注

##### 案例1

**注解直接标注在目标类上，这种情况目标类会被匹配到。**

###### 自定义一个注解Ann6

package com.javacode2018.aop.demo9.test6;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface Ann6 {  
}

###### 目标类S6上直接使用@Ann1

package com.javacode2018.aop.demo9.test6;  
  
@Ann6  
public class S6 {  
 public void m1() {  
 System.out.println("我是m1");  
 }  
}

###### 来个Aspect类

package com.javacode2018.aop.demo9.test6;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest6 {  
 //@1：目标类上有@Ann1注解  
 @Pointcut("@target(Ann1)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试代码

@Test  
public void test6() {  
 S6 target = new S6();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest6.class);  
 S6 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 System.out.println("目标类上是否有 @Ann6 注解：" + (target.getClass().getAnnotation(Ann6.class) != null));  
}

###### 运行输出

execution(void com.javacode2018.aop.demo9.test6.S6.m1())  
我是m1  
目标类上是否有 @Ann6 注解：true

##### 案例2

**注解标注在父类上，注解上没有@Inherited，这种情况下，目标类无法匹配到，下面看代码**

###### 注解Ann7

package com.javacode2018.aop.demo9.test7;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface Ann7 {  
}

###### 来2个父子类，父类上有@Ann7，之类S7为目标类

package com.javacode2018.aop.demo9.test7;  
  
import java.lang.annotation.Target;  
  
@Ann7  
class S7Parent {  
}  
  
public class S7 extends S7Parent {  
 public void m1() {  
 System.out.println("我是m1");  
 }  
  
 public static void main(String[] args) {  
 System.out.println(S7.class.getAnnotation(Target.class));  
 }  
}

###### 来个Aspect类

package com.javacode2018.aop.demo9.test7;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest7 {  
 /\*\*  
 \* 匹配目标类上有Ann7注解  
 \*/  
 @Pointcut("@target(com.javacode2018.aop.demo9.test7.Ann7)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试代码

@Test  
public void test7() {  
 S7 target = new S7();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest7.class);  
 S7 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 System.out.println("目标类上是否有 @Ann7 注解：" + (target.getClass().getAnnotation(Ann7.class) != null));  
}

###### 运行输出

我是m1  
目标类上是否有 @Ann7 注解：false

###### 分析结果

@Ann7标注在了父类上，但是@Ann7定义的时候没有使用@Inherited，说明之类无法继承父类上面的注解，所以上面的目标类没有被拦截，下面我们将@Ann7的定义改一下，加上@Inherited

package com.javacode2018.aop.demo9.test7;  
  
import java.lang.annotation.\*;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Inherited  
public @interface Ann7 {  
}

###### 再次运行输出

execution(void com.javacode2018.aop.demo9.test7.S7.m1())  
我是m1  
目标类上是否有 @Ann7 注解：true

此时目标对象被拦截了。

#### 8、@args

##### 用法

@args(注解类型)：方法参数所属的类上有指定的注解；注意不是参数上有指定的注解，而是参数类型的类上有指定的注解。

##### 案例1

@Pointcut("@args(Ann8)")：匹配方法只有一个参数，并且参数所属的类上有Ann8注解

可以匹配下面的代码，m1方法的第一个参数类型是Car类型，Car类型上有注解Ann8

@Ann8  
class Car {  
}  
  
public void m1(Car car) {  
 System.out.println("我是m1");  
}

##### 案例2

@Pointcut("@args(\*,Ann8)")：匹配方法只有2个参数，且第2个参数所属的类型上有Ann8注解

可以匹配下面代码

@Ann8  
class Car {  
}  
  
public void m1(String name,Car car) {  
 System.out.println("我是m1");  
}

##### 案例3

@Pointcut("@args(..,com.javacode2018.aop.demo9.test8.Ann8)")：匹配参数数量大于等于1，且最后一个参数所属的类型上有Ann8注解  
@Pointcut("@args(\*,com.javacode2018.aop.demo9.test8.Ann8,..)")：匹配参数数量大于等于2，且第2个参数所属的类型上有Ann8注解  
@Pointcut("@args(..,com.javacode2018.aop.demo9.test8.Ann8,\*)")：匹配参数数量大于等于2，且倒数第2个参数所属的类型上有Ann8注解

这个案例代码，大家自己写一下，体验一下。

#### 9、@annotation

##### 用法

@annotation(注解类型)：匹配被调用的方法上有指定的注解。

##### 案例

###### 定义一个注解，可以用在方法上

package com.javacode2018.aop.demo9.test12;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Ann12 {  
}

###### 定义2个类

S12Parent为父类，内部定义了2个方法，2个方法上都有@Ann12注解

S12是代理的目标类，也是S12Parent的子类，内部重写了m2方法，重写之后m2方法上并没有@Ann12注解，S12内部还定义2个方法m3和m4，而m3上面有注解@Ann12

package com.javacode2018.aop.demo9.test12;  
  
class S12Parent {  
  
 @Ann12  
 public void m1() {  
 System.out.println("我是S12Parent.m1()方法");  
 }  
  
 @Ann12  
 public void m2() {  
 System.out.println("我是S12Parent.m2()方法");  
 }  
}  
  
public class S12 extends S12Parent {  
  
 @Override  
 public void m2() {  
 System.out.println("我是S12.m2()方法");  
 }  
  
 @Ann12  
 public void m3() {  
 System.out.println("我是S12.m3()方法");  
 }  
  
 public void m4() {  
 System.out.println("我是S12.m4()方法");  
 }  
}

###### 来个Aspect类

当被调用的目标方法上有@Ann12注解的时，会被beforeAdvice处理。

package com.javacode2018.aop.demo9.test12;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class AspectTest12 {  
  
 @Pointcut("@annotation(com.javacode2018.aop.demo9.test12.Ann12)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 测试用例

S12作为目标对象，创建代理，然后分别调用4个方法

@Test  
public void test12() {  
 S12 target = new S12();  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(AspectTest12.class);  
 S12 proxy = proxyFactory.getProxy();  
 proxy.m1();  
 proxy.m2();  
 proxy.m3();  
 proxy.m4();  
}

###### 运行输出

execution(void com.javacode2018.aop.demo9.test12.S12Parent.m1())  
我是S12Parent.m1()方法  
我是S12.m2()方法  
execution(void com.javacode2018.aop.demo9.test12.S12.m3())  
我是S12.m3()方法  
我是S12.m4()方法

###### 分析结果

m1方法位于S12Parent中，上面有@Ann12注解，被拦截了，m3方法上有@Ann12注解，被拦截了，而m4上没有@Ann12注解，没有被拦截，这3个方法的执行结果都很容易理解。

重点在于m2方法的执行结果，没有被拦截，m2方法虽然在S12Parent中定义的时候也有@Ann12注解标注，但是这个方法被S1给重写了，在S1中定义的时候并没有@Ann12注解，代码中实际上调用的是S1中的m2方法，发现这个方法上并没有@Ann12注解，所以没有被拦截。

#### 10、bean

##### 用法

bean(bean名称)：这个用在spring环境中，匹配容器中指定名称的bean。

##### 案例

###### 来个类BeanService

package com.javacode2018.aop.demo9.test13;  
  
public class BeanService {  
 private String beanName;  
  
 public BeanService(String beanName) {  
 this.beanName = beanName;  
 }  
  
 public void m1() {  
 System.out.println(this.beanName);  
 }  
}

###### 来个Aspect类

package com.javacode2018.aop.demo9.test13;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
import org.springframework.stereotype.Component;  
  
@Aspect  
public class Aspect13 {  
 //拦截spring容器中名称为beanService2的bean  
 @Pointcut("bean(beanService2)")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void beforeAdvice(JoinPoint joinPoint) {  
 System.out.println(joinPoint);  
 }  
}

###### 来个spring配置类

package com.javacode2018.aop.demo9.test13;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.EnableAspectJAutoProxy;  
  
@Configuration  
@EnableAspectJAutoProxy // 这个可以启用通过AspectJ方式自动为符合条件的bean创建代理  
public class MainConfig13 {  
  
 //将Aspect13注册到spring容器  
 @Bean  
 public Aspect13 aspect13() {  
 return new Aspect13();  
 }  
  
 @Bean  
 public BeanService beanService1() {  
 return new BeanService("beanService1");  
 }  
  
 @Bean  
 public BeanService beanService2() {  
 return new BeanService("beanService2");  
 }  
}

这个配置类中有个@EnableAspectJAutoProxy，这个注解大家可能比较陌生，这个属于aop中自动代理的范围，后面会有文章详细介绍这块，这里大家暂时先不用关注。

###### 测试用例

下面启动spring容器，加载配置类MainConfig13，然后分别获取beanService1和beanService2，调用他们的m1方法，看看效果

@Test  
public void test13() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig13.class);  
 //从容器中获取beanService1  
 BeanService beanService1 = context.getBean("beanService1", BeanService.class);  
 beanService1.m1();  
 //从容器中获取beanService2  
 BeanService beanService2 = context.getBean("beanService2", BeanService.class);  
 beanService2.m1();  
}

###### 运行输出

beanService1  
execution(void com.javacode2018.aop.demo9.test13.BeanService.m1())  
beanService2

beanService2的m1方法被拦截了。

#### 11、reference pointcut

表示引用其他命名切入点。

有时，我们可以将切入专门放在一个类中集中定义。

其他地方可以通过引用的方式引入其他类中定义的切入点。

语法如下：

@Pointcut("完整包名类名.方法名称()")

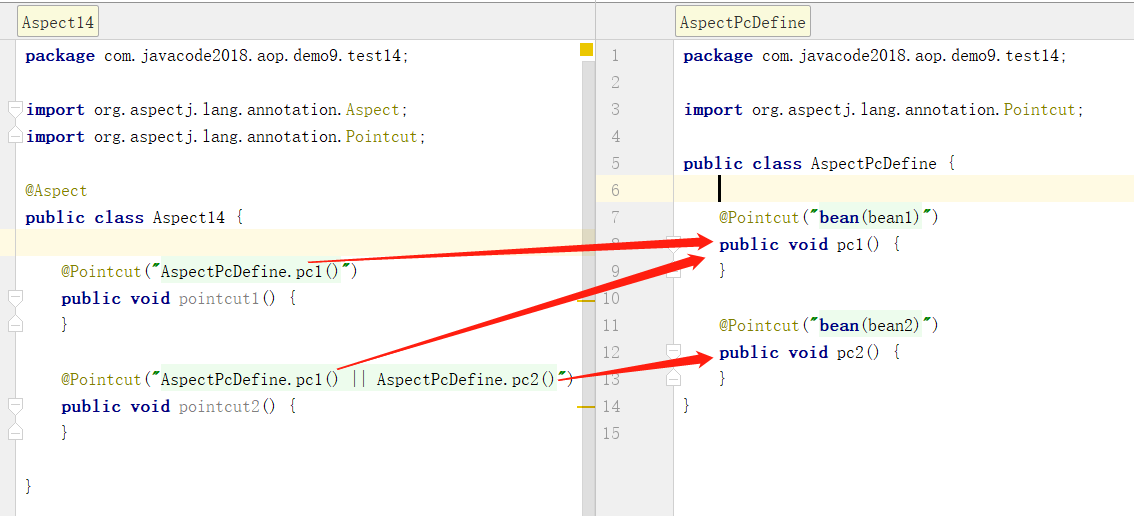
若引用同一个类中定义切入点，包名和类名可以省略，直接通过方法就可以引用。

比如下面，我们可以将所有切入点定义在一个类中

package com.javacode2018.aop.demo9.test14;  
  
import org.aspectj.lang.annotation.Pointcut;  
  
public class AspectPcDefine {  
 @Pointcut("bean(bean1)")  
 public void pc1() {  
 }  
  
 @Pointcut("bean(bean2)")  
 public void pc2() {  
 }  
}

下面顶一个一个Aspect类，来引用上面的切入点

package com.javacode2018.aop.demo9.test14;  
  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class Aspect14 {  
  
 @Pointcut("com.javacode2018.aop.demo9.test14.AspectPcDefine.pc1()")  
 public void pointcut1() {  
 }  
  
 @Pointcut("com.javacode2018.aop.demo9.test14.AspectPcDefine.pc1() || com.javacode2018.aop.demo9.test14.AspectPcDefine.pc2()")  
 public void pointcut2() {  
 }  
  
}



#### 12、组合型的pointcut

Pointcut定义时，还可以使用&&、||、!运算符。

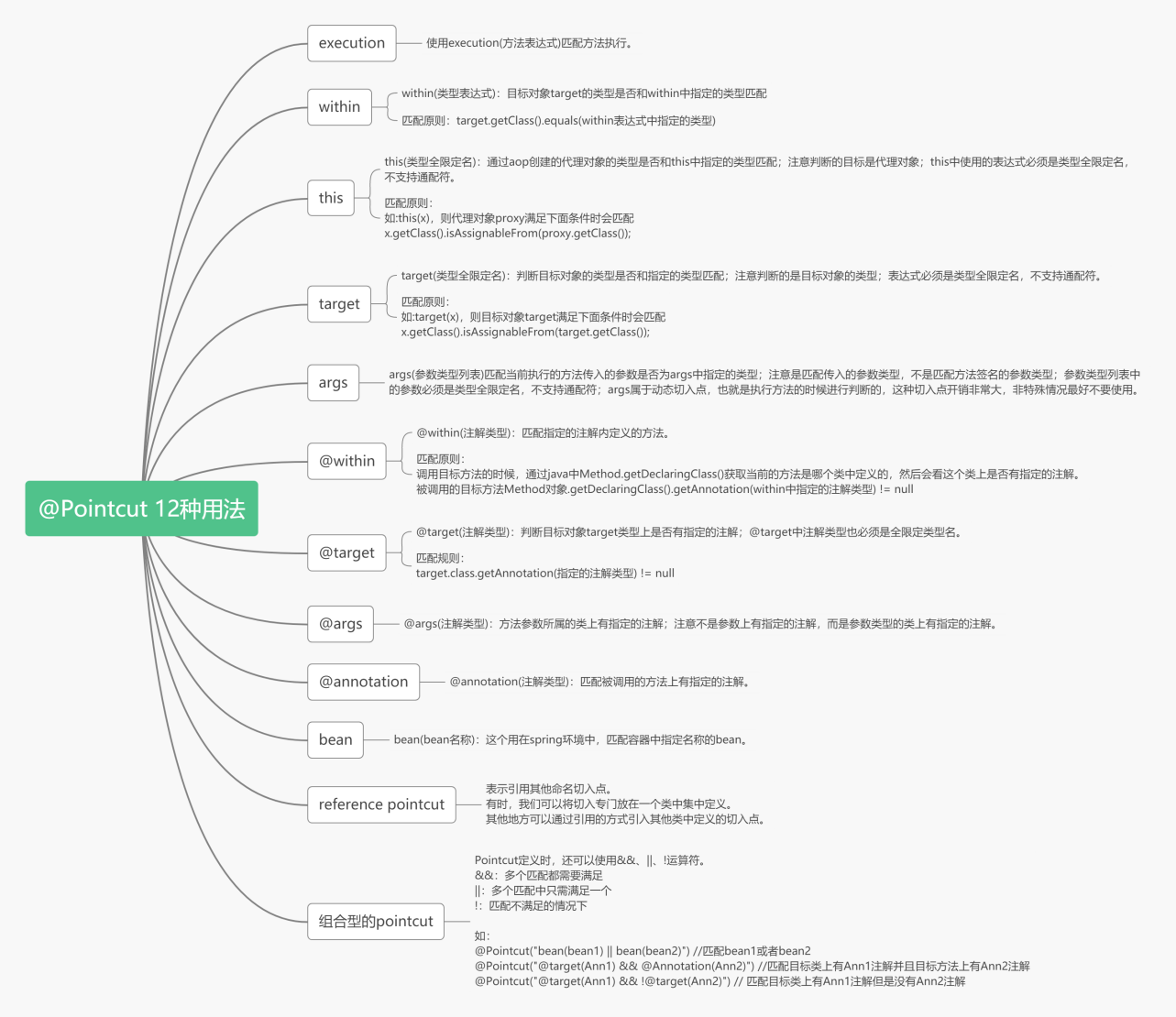
* &&：多个匹配都需要满足
* ||：多个匹配中只需满足一个
* !：匹配不满足的情况下

@Pointcut("bean(bean1) || bean(bean2)") //匹配bean1或者bean2  
@Pointcut("@target(Ann1) && @Annotation(Ann2)") //匹配目标类上有Ann1注解并且目标方法上有Ann2注解  
@Pointcut("@target(Ann1) && !@target(Ann2)") // 匹配目标类上有Ann1注解但是没有Ann2注解

### 总结

本文详解了@Pointcut的12种用法，案例大家一定要敲一遍，敲的过程中，会遇到问题，然后解决问题，才能够加深理解。

有问题的也欢迎大家留言交流，谢谢！



### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理

## Spring系列第35篇：@Aspect中5中通知详解

本文内容：详解@Aspect中5中通知的使用。

### Aop相关阅读

阅读本文之前，需要先掌握下面几篇文章内容，不然会比较吃力。

1. Spring系列第15篇：代理详解（java动态代理&CGLIB代理)
2. Spring系列第30篇：jdk动态代理和cglib代理
3. Spring系列第31篇：Aop概念详解
4. Spring系列第32篇：AOP核心源码、原理详解
5. Spring系列第33篇：ProxyFactoryBean创建AOP代理
6. Spring系列第34篇：@Aspect中@Pointcut 12种用法

### @Aspect中有5种通知

1. @Before：前置通知, 在方法执行之前执行
2. @Aroud：环绕通知, 围绕着方法执行
3. @After：后置通知, 在方法执行之后执行
4. @AfterReturning：返回通知, 在方法返回结果之后执行
5. @AfterThrowing：异常通知, 在方法抛出异常之后

这几种通知用起来都比较简单，都是通过注解的方式，将这些注解标注在@Aspect类的方法上，这些方法就会对目标方法进行拦截，下面我们一个个来看一下。

### @Before：前置通知

#### 介绍

定义一个前置通知

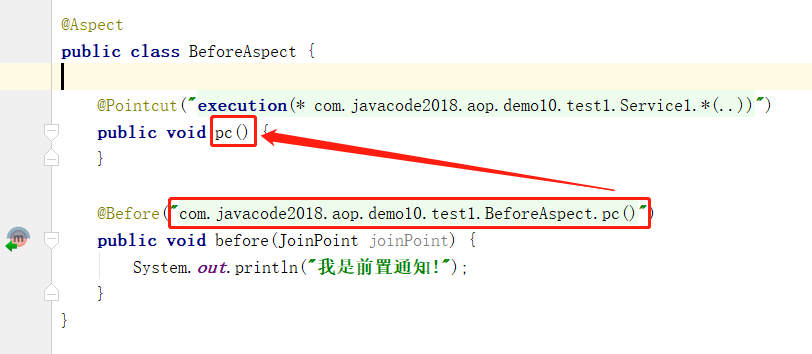
@Aspect  
public class BeforeAspect {  
  
 @Before("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void before(JoinPoint joinPoint) {  
 System.out.println("我是前置通知!");  
 }  
}

1. 类上需要使用@Aspect标注
2. 任意方法上使用@Before标注，将这个方法作为前置通知，目标方法被调用之前，会自动回调这个方法
3. 被@Before标注的方法参数可以为空，或者为JoinPoint类型，当为JoinPoint类型时，必须为第一个参数
4. 被@Before标注的方法名称可以随意命名，符合java规范就可以，其他通知也类似

@Before中value的值为切入点表达式，也可以采用引用的方式指定切入点，如：

package com.javacode2018.aop.demo10.test1;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class BeforeAspect {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @Before("com.javacode2018.aop.demo10.test1.BeforeAspect.pc()")  
 public void before(JoinPoint joinPoint) {  
 System.out.println("我是前置通知!");  
 }  
}

此时，before方法上面的切入引用了pc方法上面的@Pointcut的值



#### 案例

来个普通的service

package com.javacode2018.aop.demo10.test1;  
  
public class Service1 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
  
 public String work(String name) {  
 return "开始工作了：" + name;  
 }  
}

给上面的类定义一个前置通知，Service1中的所有方法执行执行，输出一段文字我是前置通知!

package com.javacode2018.aop.demo10.test1;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class BeforeAspect1 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @Before("com.javacode2018.aop.demo10.test1.BeforeAspect1.pc()")  
 public void before(JoinPoint joinPoint) {  
 System.out.println("我是前置通知!");  
 }  
}

测试代码

package com.javacode2018.aop.demo10;  
  
import com.javacode2018.aop.demo10.test1.BeforeAspect1;  
import com.javacode2018.aop.demo10.test1.Service1;  
import org.junit.Test;  
import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;  
  
public class AopTest10 {  
  
 @Test  
 public void test1() {  
 Service1 target = new Service1();  
 Class<BeforeAspect1> aspectClass = BeforeAspect1.class;  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(aspectClass);  
 Service1 proxy = proxyFactory.getProxy();  
 System.out.println(proxy.say("路人"));  
 System.out.println(proxy.work("路人"));  
 }  
}

运行输出

我是前置通知!  
你好：路人  
我是前置通知!  
开始工作了：路人

#### 对应的通知类

@Before通知最后会被解析为下面这个通知类

org.springframework.aop.aspectj.AspectJMethodBeforeAdvice

### 通知中获取被调方法信息

通知中如果想获取被调用方法的信息，分2种情况

1. 非环绕通知，可以将org.aspectj.lang.JoinPoint作为通知方法的第1个参数，通过这个参数获取被调用方法的信息
2. 如果是环绕通知，可以将org.aspectj.lang.ProceedingJoinPoint作为方法的第1个参数，通过这个参数获取被调用方法的信息

#### JoinPoint：连接点信息

org.aspectj.lang.JoinPoint

提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

package org.aspectj.lang;    
import org.aspectj.lang.reflect.SourceLocation;  
  
public interface JoinPoint {    
    String toString();         //连接点所在位置的相关信息    
    String toShortString();     //连接点所在位置的简短相关信息    
    String toLongString();     //连接点所在位置的全部相关信息    
    Object getThis();         //返回AOP代理对象  
    Object getTarget();       //返回目标对象    
    Object[] getArgs();       //返回被通知方法参数列表，也就是目前调用目标方法传入的参数    
    Signature getSignature();  //返回当前连接点签名，这个可以用来获取目标方法的详细信息，如方法Method对象等  
    SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置    
    String getKind();        //连接点类型    
    StaticPart getStaticPart(); //返回连接点静态部分    
}

#### ProceedingJoinPoint：环绕通知连接点信息

用于环绕通知，内部主要关注2个方法，一个有参的，一个无参的，用来继续执行拦截器链上的下一个通知。

package org.aspectj.lang;  
import org.aspectj.runtime.internal.AroundClosure;  
  
public interface ProceedingJoinPoint extends JoinPoint {  
  
 /\*\*  
 \* 继续执行下一个通知或者目标方法的调用  
 \*/  
 public Object proceed() throws Throwable;  
  
 /\*\*  
 \* 继续执行下一个通知或者目标方法的调用  
 \*/  
 public Object proceed(Object[] args) throws Throwable;  
  
}

#### Signature：连接点签名信息

注意JoinPoint#getSignature()这个方法，用来获取连接点的签名信息，这个比较重要

Signature getSignature();

通常情况，spring中的aop都是用来对方法进行拦截，所以通常情况下连接点都是一个具体的方法，Signature有个子接口

org.aspectj.lang.reflect.MethodSignature

JoinPoint#getSignature()都可以转换转换为MethodSignature类型，然后可以通过这个接口提供的一些方法来获取被调用的方法的详细信息。

下面对上面的前置通知的案例改造一下，获取被调用方法的详细信息，新建一个Aspect类：BeforeAspect2

package com.javacode2018.aop.demo10.test2;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.Signature;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
import org.aspectj.lang.reflect.MethodSignature;  
  
import java.lang.reflect.Method;  
  
@Aspect  
public class BeforeAspect2 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @Before("com.javacode2018.aop.demo10.test2.BeforeAspect2.pc()")  
 public void before(JoinPoint joinPoint) {  
 //获取连接点签名  
 Signature signature = joinPoint.getSignature();  
 //将其转换为方法签名  
 MethodSignature methodSignature = (MethodSignature) signature;  
 //通过方法签名获取被调用的目标方法  
 Method method = methodSignature.getMethod();  
 //输出方法信息  
 System.out.println(method);  
 }  
}

测试用例

@Test  
public void test2() {  
 Service1 target = new Service1();  
 Class<BeforeAspect2> aspectClass = BeforeAspect2.class;  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(aspectClass);  
 Service1 proxy = proxyFactory.getProxy();  
 System.out.println(proxy.say("路人"));  
 System.out.println(proxy.work("路人"));  
}

运行输出

public java.lang.String com.javacode2018.aop.demo10.test1.Service1.say(java.lang.String)  
你好：路人  
public java.lang.String com.javacode2018.aop.demo10.test1.Service1.work(java.lang.String)  
开始工作了：路人

### @Around：环绕通知

#### 介绍

环绕通知会包裹目标目标方法的执行，可以在通知内部调用ProceedingJoinPoint.process方法继续执行下一个拦截器。

用起来和@Before类似，但是有2点不一样

1. 若需要获取目标方法的信息，需要将ProceedingJoinPoint作为第一个参数
2. 通常使用Object类型作为方法的返回值，返回值也可以为void

#### 特点

环绕通知比较特殊，其他4种类型的通知都可以用环绕通知来实现。

#### 案例

通过环绕通知来统计方法的耗时。

package com.javacode2018.aop.demo10.test3;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.Signature;  
import org.aspectj.lang.annotation.Around;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
import org.aspectj.lang.reflect.MethodSignature;  
  
import java.lang.reflect.Method;  
  
@Aspect  
public class AroundAspect3 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @Around("com.javacode2018.aop.demo10.test3.AroundAspect3.pc()")  
 public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
 //获取连接点签名  
 Signature signature = joinPoint.getSignature();  
 //将其转换为方法签名  
 MethodSignature methodSignature = (MethodSignature) signature;  
 //通过方法签名获取被调用的目标方法  
 Method method = methodSignature.getMethod();  
  
 long startTime = System.nanoTime();  
 //调用proceed方法，继续调用下一个通知  
 Object returnVal = joinPoint.proceed();  
 long endTime = System.nanoTime();  
 long costTime = endTime - startTime;  
 //输出方法信息  
 System.out.println(String.format("%s，耗时(纳秒)：%s", method.toString(), costTime));  
 //返回方法的返回值  
 return returnVal;  
 }  
}

测试用例

@Test  
public void test3() {  
 Service1 target = new Service1();  
 Class<AroundAspect3> aspectClass = AroundAspect3.class;  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(aspectClass);  
 Service1 proxy = proxyFactory.getProxy();  
 System.out.println(proxy.say("路人"));  
 System.out.println(proxy.work("路人"));  
}

运行输出

public java.lang.String com.javacode2018.aop.demo10.test1.Service1.say(java.lang.String)，耗时(纳秒)：19000500  
你好：路人  
public java.lang.String com.javacode2018.aop.demo10.test1.Service1.work(java.lang.String)，耗时(纳秒)：59600  
开始工作了：路人

#### 对应的通知类

@Around通知最后会被解析为下面这个通知类

org.springframework.aop.aspectj.AspectJAroundAdvice

### @After：后置通知

#### 介绍

后置通知，在方法执行之后执行，用法和前置通知类似。

#### 特点

* **不管目标方法是否有异常，后置通知都会执行**
* 这种通知无法获取方法返回值
* 可以使用JoinPoint作为方法的第一个参数，用来获取连接点的信息

#### 案例

在Service1中任意方法执行完毕之后，输出一行日志。

package com.javacode2018.aop.demo10.test4;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.After;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
import org.aspectj.lang.reflect.MethodSignature;  
  
@Aspect  
public class AfterAspect4 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @After("com.javacode2018.aop.demo10.test4.AfterAspect4.pc()")  
 public void after(JoinPoint joinPoint) throws Throwable {  
 MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();  
 System.out.println(String.format("%s,执行完毕!", methodSignature.getMethod()));  
 }  
}

测试案例

@Test  
public void test4() {  
 Service1 target = new Service1();  
 Class<AfterAspect4> aspectClass = AfterAspect4.class;  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(aspectClass);  
 Service1 proxy = proxyFactory.getProxy();  
 System.out.println(proxy.say("路人"));  
 System.out.println(proxy.work("路人"));  
}

运行输出

public java.lang.String com.javacode2018.aop.demo10.test1.Service1.say(java.lang.String),执行完毕!  
你好：路人  
public java.lang.String com.javacode2018.aop.demo10.test1.Service1.work(java.lang.String),执行完毕!  
开始工作了：路人

#### 对应的通知类

@After通知最后会被解析为下面这个通知类

org.springframework.aop.aspectj.AspectJAfterAdvice

这个类中有invoke方法，这个方法内部会调用被通知的方法，其内部采用try..finally的方式实现的，所以不管目标方法是否有异常，通知一定会被执行。

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //继续执行下一个拦截器  
 return mi.proceed();  
 }  
 finally {  
 //内部通过反射调用被@After标注的方法  
 invokeAdviceMethod(getJoinPointMatch(), null, null);  
 }  
}

### @AfterReturning：返回通知

#### 用法

返回通知，在方法返回结果之后执行。

#### 特点

* 可以获取到方法的返回值
* 当目标方法返回异常的时候，这个通知不会被调用，这点和@After通知是有区别的

#### 案例

后置通知中打印出方法及返回值信息。

package com.javacode2018.aop.demo10.test5;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
import org.aspectj.lang.reflect.MethodSignature;  
  
@Aspect  
public class AfterReturningAspect5 {  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @AfterReturning(value = "com.javacode2018.aop.demo10.test5.AfterReturningAspect5.pc()", returning = "retVal")  
 public void afterReturning(JoinPoint joinPoint, Object retVal) throws Throwable {  
 MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();  
 System.out.println(String.format("%s返回值：%s", methodSignature.getMethod(), retVal));  
 }  
  
}

注意@AfterReturning注解，用到了2个参数

* value：用来指定切入点
* returning：用来指定返回值对应方法的参数名称，返回值对应方法的第二个参数，名称为retVal

#### 对应的通知类

@AfterReturning通知最后会被解析为下面这个通知类

org.springframework.aop.aspectj.AspectJAfterReturningAdvice

### @AfterThrowing：异常通知

#### 用法

在方法抛出异常之后会回调@AfterThrowing标注的方法。

@AfterThrowing标注的方法可以指定异常的类型，当被调用的方法触发该异常及其子类型的异常之后，会触发异常方法的回调。也可以不指定异常类型，此时会匹配所有异常。

##### 未指定异常类型

未指定异常类型，可以匹配所有异常类型，如下

@AfterThrowing(value = "切入点")  
public void afterThrowing()

##### 指定异常类型

通过@AfterThrowing的throwing指定参数异常参数名称，我们用方法的第二个参数用来接收异常，第二个参数名称为e，下面的代码，当目标方法发生IllegalArgumentException异常及其子类型异常时，下面的方法会被回调。

@AfterThrowing(value = "com.javacode2018.aop.demo10.test6.AfterThrowingAspect6.pc()", throwing = "e")  
public void afterThrowing(JoinPoint joinPoint, IllegalArgumentException e)

#### 特点

* 不论异常是否被异常通知捕获，异常还会继续向外抛出。

#### 案例

Service1中加了login方法，用户名不是马士兵老师时抛出异常。

package com.javacode2018.aop.demo10.test1;  
  
public class Service1 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
  
 public String work(String name) {  
 return "开始工作了：" + name;  
 }  
  
 public boolean login(String name) {  
 if (!"马士兵老师".equals(name)) {  
 throw new IllegalArgumentException("非法访问!");  
 }  
 return true;  
 }  
}

来个异常通知

package com.javacode2018.aop.demo10.test6;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.AfterThrowing;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
import org.aspectj.lang.reflect.MethodSignature;  
  
@Aspect  
public class AfterThrowingAspect6 {  
 @Pointcut("execution(\* com.javacode2018.aop.demo10.test1.Service1.\*(..))")  
 public void pc() {  
 }  
  
 @AfterThrowing(value = "com.javacode2018.aop.demo10.test6.AfterThrowingAspect6.pc()", throwing = "e")  
 public void afterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {  
 MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();  
 System.out.println(String.format("%s发生异常,异常信息：%s", methodSignature.getMethod(), e.getMessage()));  
 }  
  
}

测试用例

@Test  
public void test6() {  
 Service1 target = new Service1();  
 Class<AfterThrowingAspect6> aspectClass = AfterThrowingAspect6.class;  
 AspectJProxyFactory proxyFactory = new AspectJProxyFactory();  
 proxyFactory.setTarget(target);  
 proxyFactory.addAspect(aspectClass);  
 Service1 proxy = proxyFactory.getProxy();  
 proxy.login("路人");  
}

运行输出

public boolean com.javacode2018.aop.demo10.test1.Service1.login(java.lang.String)发生异常,异常信息：非法访问!  
  
java.lang.IllegalArgumentException: 非法访问!  
  
 at com.javacode2018.aop.demo10.test1.Service1.login(Service1.java:14)  
 at com.javacode2018.aop.demo10.test1.Service1$$FastClassBySpringCGLIB$$ea03ccbe.invoke(<generated>)  
 at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)  
 at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:769)

#### 对应的通知类

@AfterThrowing通知最后会被解析为下面这个通知类

org.springframework.aop.aspectj.AspectJAfterThrowingAdvice

来看一下这个类的invoke方法，这个方法是关键

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //继续调用下一个拦截器链  
 return mi.proceed();  
 }  
 catch (Throwable ex) {  
 //判断ex和需要不糊的异常是否匹配  
 if (shouldInvokeOnThrowing(ex)) {  
 //通过反射调用@AfterThrowing标注的方法  
 invokeAdviceMethod(getJoinPointMatch(), null, ex);  
 }  
 //继续向外抛出异常  
 throw ex;  
 }  
}

### 几种通知对比

|  |  |  |  |
| --- | --- | --- | --- |
| 通知类型 | 执行时间点 | 可获取返回值 | 目标方法异常时是否会执行 |
| @Before | 方法执行之前 | 否 | 是 |
| @Around | 环绕方法执行 | 是 | 自己控制 |
| @After | 方法执行后 | 否 | 是 |
| @AfterReturning | 方法执行后 | 是 | 否 |
| @AfterThrowing | 方法发生异常后 | 否 | 是 |



### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法

## Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解

**这是aop最后一篇文章了，本文带你深入理解@EnableAspectJAutoProxy，这篇文章可能会颠覆你以前所掌握的一些知识，让你醍醐灌顶，欣喜若狂！**

### 1、Aop相关阅读

阅读本文之前，需要先掌握下面几篇文章内容，不然会比较吃力。

1. Spring系列第15篇：代理详解（java动态代理&CGLIB代理)
2. Spring系列第30篇：jdk动态代理和cglib代理
3. Spring系列第31篇：Aop概念详解
4. Spring系列第32篇：AOP核心源码、原理详解
5. Spring系列第33篇：ProxyFactoryBean创建AOP代理
6. Spring系列第34篇：@Aspect中@Pointcut 12种用法
7. Spring系列第35篇：@Aspect中5中通知详解

目前为止，上面的文章基本上都是硬编码的方式一个个为目标对象创建代理的，但是，我们使用spring的过程中，可能需要对大量bean创建代理，比如我们需拦截所有的service的方法，打印耗时日志，对大量service bean做权限校验，做事务处理等等，这些功能都可以通过aop的方式来实现，若采用硬编码的方式一个个创建，那是相当难受的事情。

Spring中提供了批量的方式，为容器中符合条件的bean，自动创建代理对象，也就是我们本文要说的@EnableAspectJAutoProxy。

### 2、@EnableAspectJAutoProxy自动为bean创建代理对象

@EnableAspectJAutoProxy可以自动为spring容器中符合条件的bean创建代理对象，@EnableAspectJAutoProxy需要结合@Aspect注解一起使用。用法比较简单，下面我们通过案例来看一下。

先在com.javacode2018.aop.demo11.test1包中定义2个bean

UserService bean

package com.javacode2018.aop.demo11.test1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserService {  
 public void say(){  
 System.out.println("我是UserService");  
 }  
}

CarService bean

package com.javacode2018.aop.demo11.test1;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class CarService {  
 public void say() {  
 System.out.println("我是CarService");  
 }  
}

通过Aspect来定义一个前置通知，需要拦截上面2个bean的所有方法，在方法执行之前输出一行日志

package com.javacode2018.aop.demo11.test1;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
import org.springframework.stereotype.Component;  
  
@Component //@1  
@Aspect //@2  
public class Aspect1 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo11.test1..\*(..))") //@3  
 public void pc() {  
 }  
  
 @Before("com.javacode2018.aop.demo11.test1.Aspect1.pc()") //@4  
 public void before(JoinPoint joinPoint) {  
 System.out.println("我是前置通知,target:" + joinPoint.getTarget()); //5  
 }  
}

Aspect1中有4个关键信息

@1：使用 @Component 将这个类注册到spring容器；

@2：使用 @Aspect 标注着是一个 AspectJ 来定义通知的配置类；

@3：定义切入点，目前的配置，会拦截test1包及其子包中所有类的所有方法，而CarService和UserService刚好满足，所以会被拦截；

@4：定义一个前置通知，这个通知会对@3定义的切入点起效；

@5：目标方法执行执行，输出一行日志；

下面来一个spring配置类

package com.javacode2018.aop.demo11.test1;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.EnableAspectJAutoProxy;  
  
@ComponentScan //@1  
@EnableAspectJAutoProxy //@2  
public class MainConfig1 {  
}

@1：@ComponentScan 注解的作用会扫描当前包中的类，将标注有 @Component 的类注册到spring容器；

@2：@EnableAspectJAutoProxy 这个注解比较关键，用来启用自动代理的创建，简单点理解：会找到容器中所有标注有@Aspect注解的bean以及Advisor类型的bean，会将他们转换为Advisor集合，spring会通过Advisor集合对容器中满足切入点表达式的bean生成代理对象，整个都是spring容器启动的过程中自动完成的，原理稍后介绍。

下面来测试用例代码，启动spring容器，加载配置类，验证

package com.javacode2018.aop.demo11;  
  
import com.javacode2018.aop.demo11.test1.CarService;  
import com.javacode2018.aop.demo11.test1.MainConfig1;  
import com.javacode2018.aop.demo11.test1.UserService;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class AspectTest11 {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 UserService userService = context.getBean(UserService.class);  
 userService.say();  
 CarService carService = context.getBean(CarService.class);  
 carService.say();  
 }  
   
}

运行输出

我是前置通知,target:com.javacode2018.aop.demo11.test1.UserService@dc7df28  
我是UserService  
我是前置通知,target:com.javacode2018.aop.demo11.test1.CarService@821330f  
我是CarService

### 3、通知执行顺序

@EnableAspectJAutoProxy 允许spring容器中通过Advisor 、@Aspect 来定义通知，当spring容器中存在多个Advisor、@Aspect时，组成的拦截器调用链顺序是什么样的呢？在介绍这个之前，我们需要先回顾一下aop中4种通知相关知识。

#### spring aop中4种通知（Advice）

org.aopalliance.intercept.MethodInterceptor  
org.springframework.aop.MethodBeforeAdvice  
org.springframework.aop.AfterReturningAdvice  
org.springframework.aop.ThrowsAdvice

所有的通知最终都需要转换为MethodInterceptor类型的通知，然后组成一个MethodInterceptor列表，我们称之为方法调用链或者拦截器链，上面列表中后面3通过下面的转换器将其包装为MethodInterceptor类型的通知：

org.springframework.aop.MethodBeforeAdvice -> org.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor  
org.springframework.aop.AfterReturningAdvice -> org.springframework.aop.framework.adapter.AfterReturningAdviceInterceptor  
org.springframework.aop.ThrowsAdvice -> org.springframework.aop.framework.adapter.ThrowsAdviceInterceptor

下面我们再来看一下4种通知的用法和执行过程，以方便我们理解其执行顺序。

##### org.aopalliance.intercept.MethodInterceptor：方法拦截器

方法拦截器，这个比较强大，可以在方法执行前后执行一些增强操作，其他类型的通知最终都会被包装为 MethodInterceptor 来执行。

下面我们自定义一个MethodInterceptor

class MyMethodInterceptor implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 System.out.println("我是MethodInterceptor start");  
 //调用invocation.proceed()执行下一个拦截器  
 Object result = invocation.proceed();  
 System.out.println("我是MethodInterceptor end");  
 //返回结果  
 return result;  
 }  
}

##### org.springframework.aop.MethodBeforeAdvice：方法前置通知

方法前置通知，可以在方法之前定义增强操作。

下面我们自定义一个MethodBeforeAdvice

class MyMethodBeforeAdvice implements MethodBeforeAdvice {  
  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是MethodBeforeAdvice");  
 }  
}

**MethodBeforeAdvice最终会被包装为MethodBeforeAdviceInterceptor类型，然后放到拦截器链中去执行，通过MethodBeforeAdviceInterceptor代码可以理解MethodBeforeAdvice的执行过程**

public class MethodBeforeAdviceInterceptor implements MethodInterceptor, BeforeAdvice, Serializable {  
  
 private final MethodBeforeAdvice advice;  
  
 public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //调用MethodBeforeAdvice的before方法，执行前置通知  
 this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());  
 //执行下一个拦截器  
 return mi.proceed();  
 }  
  
}

##### org.springframework.aop.AfterReturningAdvice：方法返回通知

方法返回通知，用来在方法执行完毕之后执行一些增强操作。

下面我们自定义一个AfterReturningAdvice

class MyAfterReturningAdvice implements AfterReturningAdvice {  
  
 @Override  
 public void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是AfterReturningAdvice");  
 }  
}

**AfterReturningAdvice最终会被包装为AfterReturningAdviceInterceptor类型，然后放到拦截器链中去执行，通过AfterReturningAdviceInterceptor代码可以理解AfterReturningAdvice的执行过程**

public class AfterReturningAdviceInterceptor implements MethodInterceptor, AfterAdvice, Serializable {  
  
 private final AfterReturningAdvice advice;  
  
 public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {  
 this.advice = advice;  
 }  
  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //执行下一个拦截器，可以获取目标方法的返回结果  
 Object retVal = mi.proceed();  
 //调用方法返回通知的afterReturning方法，会传入目标方法的返回值等信息  
 this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.getThis());  
 return retVal;  
 }  
  
}

##### org.springframework.aop.ThrowsAdvice：异常通知

当目标方法发生异常时，可以通过 ThrowsAdvice 来指定需要回调的方法，我们在此可以记录一些异常信息，或者将异常信息发送到监控系统等。

下面我们自定义一个ThrowsAdvice

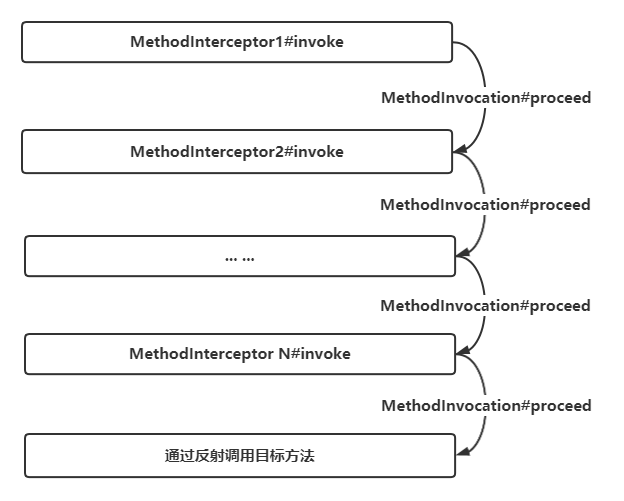
/\*\*  
 \* 用来定义异常通知  
 \* 方法名必须是afterThrowing，格式参考下面2种定义  
 \* 1. public void afterThrowing(Exception ex)  
 \* 2. public void afterThrowing(Method method, Object[] args, Object target, Exception ex)  
 \*/  
class MyThrowsAdvice implements ThrowsAdvice {  
 public void afterThrowing(Method method, Object[] args, Object target, Exception ex) {  
 System.out.println("我是ThrowsAdvice");  
 }  
}

**ThrowsAdvice最终会被包装为ThrowsAdviceInterceptor类型，然后放到拦截器链中去执行，通过ThrowsAdviceInterceptor代码可以理解ThrowsAdvice的执行过程，ThrowsAdviceInterceptor 构造参数传入一个自定义的 ThrowsAdvice 对象**

public class ThrowsAdviceInterceptor implements MethodInterceptor, AfterAdvice {  
  
 private final Object throwsAdvice;  
  
 public ThrowsAdviceInterceptor(Object throwsAdvice) {  
 this.throwsAdvice = throwsAdvice;  
 }  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 return mi.proceed();  
 } catch (Throwable ex) {  
 //调用 ThrowsAdvice 中的 afterThrowing 方法来处理异常  
 this.throwsAdvice.afterThrowing(。。。。);  
 //将异常继续往外抛  
 throw ex;  
 }  
 }  
}

#### 拦截器链执行过程

假如目标方法上面有好几个通知，调用目标方法执行，spring会将所有的通知转换得到一个MethodInterceptor列表，然后依次按照下面的方式执行，会先调用第一个拦截器的MethodInterceptor#invoke(MethodInvocation invocation)方法，会传递一个MethodInvocation类型的参数，在此方法中，我们可以调用MethodInvocation#processd方法去执行第二个拦截器，然后依次按照这样的过程执行，到了最后一个MethodInterceptor中，再次调用MethodInvocation#processd时，会调用目标方法。



#### 4种通知的执行顺序

结合上面的过程，假如目标方法上面依次添加了下面4种通知，我们来分析一下他们的执行过程

class MyMethodInterceptor implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 System.out.println("我是MethodInterceptor start");  
 //调用invocation.proceed()执行下一个拦截器  
 Object result = invocation.proceed();  
 System.out.println("我是MethodInterceptor end");  
 //返回结果  
 return result;  
 }  
}  
  
class MyMethodBeforeAdvice implements MethodBeforeAdvice {  
  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是MethodBeforeAdvice");  
 }  
}  
  
class MyAfterReturningAdvice implements AfterReturningAdvice {  
  
 @Override  
 public void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是AfterReturningAdvice");  
 }  
}  
  
class MyThrowsAdvice implements ThrowsAdvice {  
 public void afterThrowing(Method method, Object[] args, Object target, Exception ex) {  
 System.out.println("我是ThrowsAdvice");  
 }  
}

根据通知的规定，非MethodInterceptor类型的通知，都会被包装为MethodInterceptor类型的，上面除了第一个之外，其他3个都会被转换为MethodInterceptor，转换之后变成了下面这样：

class MyMethodInterceptor implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 System.out.println("我是MethodInterceptor start");  
 //调用mi.proceed()执行下一个拦截器  
 Object retVal = mi.proceed();  
 System.out.println("我是MethodInterceptor end");  
 //返回结果  
 return retVal;  
 }  
}  
  
class MyMethodBeforeAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 System.out.println("我是MethodBeforeAdvice");  
 //调用mi.proceed()执行下一个拦截器  
 Object retVal = mi.proceed();  
 return retVal;  
 }  
}  
  
class MyAfterReturningAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //调用mi.proceed()执行下一个拦截器  
 Object retVal = mi.proceed();  
 System.out.println("我是AfterReturningAdvice");  
 return retVal;  
 }  
}  
  
class MyThrowsAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //调用mi.proceed()执行下一个拦截器  
 return mi.proceed();  
 } catch (Throwable ex) {  
 System.out.println("我是ThrowsAdvice");  
 throw ex;  
 }  
 }  
}

根据通知链的执行过程，最终变成了下面这样：

System.out.println("我是MethodInterceptor start");  
System.out.println("我是MethodBeforeAdvice");  
Object retVal = null;  
try {  
 retVal = 通过反射调用目标方法获取返回值;  
} catch (Throwable ex) {  
 System.out.println("我是ThrowsAdvice");  
 throw ex;  
}  
System.out.println("我是AfterReturningAdvice");  
System.out.println("我是MethodInterceptor end");  
return retVal;

将上面4个通知用到下面目标对象中

public static class Service3 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
}

执行下面代码生成代理，然后通过代理调用say方法

Service3 target = new Service3();  
Service3 proxy = 对target通过aop生成代理对象;  
System.out.println(proxy.say("路人"));

被4个拦截器链包裹之后，System.out.println(proxy.say("路人"));执行过程变成了下面这样

System.out.println("我是MethodInterceptor start");  
System.out.println("我是MethodBeforeAdvice");  
Object retVal = null;  
try {  
 retVal = target.say("路人");  
} catch (Throwable ex) {  
 System.out.println("我是ThrowsAdvice");  
 throw ex;  
}  
System.out.println("我是AfterReturningAdvice");  
System.out.println("我是MethodInterceptor end");  
System.out.println(retVal);

再次简化

System.out.println("我是MethodInterceptor start");  
System.out.println("我是MethodBeforeAdvice");  
Object retVal = null;  
try {  
 retVal = "你好：" + name;  
} catch (Throwable ex) {  
 System.out.println("我是ThrowsAdvice");  
 throw ex;  
}  
System.out.println("我是AfterReturningAdvice");  
System.out.println("我是MethodInterceptor end");  
System.out.println(retVal);

最终会输出

我是MethodInterceptor start  
我是MethodBeforeAdvice  
我是AfterReturningAdvice  
我是MethodInterceptor end  
你好：路人

上案例代码，我们来看一下最终的执行结果是不是和我们分析的一样，下面为需要被代理的类Service3以及需要使用的4个通知。

package com.javacode2018.aop.demo11.test3;  
  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.aop.AfterReturningAdvice;  
import org.springframework.aop.MethodBeforeAdvice;  
import org.springframework.aop.ThrowsAdvice;  
import org.springframework.lang.Nullable;  
  
import java.lang.reflect.Method;  
  
public class MoreAdvice {  
  
 public static class Service3 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
 }  
  
 public static class MyMethodInterceptor implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 System.out.println("我是MethodInterceptor start");  
 //调用invocation.proceed()执行下一个拦截器  
 Object result = invocation.proceed();  
 System.out.println("我是MethodInterceptor end");  
 //返回结果  
 return result;  
 }  
 }  
  
 public static class MyMethodBeforeAdvice implements MethodBeforeAdvice {  
  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是MethodBeforeAdvice");  
 }  
 }  
  
 public static class MyAfterReturningAdvice implements AfterReturningAdvice {  
  
 @Override  
 public void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable {  
 System.out.println("我是AfterReturningAdvice");  
 }  
 }  
  
 public static class MyThrowsAdvice implements ThrowsAdvice {  
 public void afterThrowing(Method method, Object[] args, Object target, Exception ex) {  
 System.out.println("我是ThrowsAdvice");  
 }  
 }  
}

对应测试代码

@Test  
public void test3() {  
 //创建目标对象  
 MoreAdvice.Service3 target = new MoreAdvice.Service3();  
 //创建代理工厂，通过代理工厂来创建代理对象  
 ProxyFactory proxyFactory = new ProxyFactory();  
 proxyFactory.setTarget(target);  
 //依次为目标对象添加4种通知  
 proxyFactory.addAdvice(new MoreAdvice.MyMethodInterceptor());  
 proxyFactory.addAdvice(new MoreAdvice.MyMethodBeforeAdvice());  
 proxyFactory.addAdvice(new MoreAdvice.MyAfterReturningAdvice());  
 proxyFactory.addAdvice(new MoreAdvice.MyThrowsAdvice());  
 //获取到代理对象  
 MoreAdvice.Service3 proxy = (MoreAdvice.Service3) proxyFactory.getProxy();  
 //通过代理对象访问目标方法say  
 System.out.println(proxy.say("路人"));  
}

运行输出

我是MethodInterceptor start  
我是MethodBeforeAdvice  
我是AfterReturningAdvice  
我是MethodInterceptor end  
你好：路人

和我们上面分析的确实一模一样。

### 4、单个@Aspect中多个通知的执行顺序

@Aspect标注的类中可以使用下面5种注解来定义通知

@Before  
@Around  
@After  
@AfterReturning  
@AfterThrowing

**当单个@Aspect中定义了多种类型的通知时，@EnableAspectJAutoProxy内部会对其进行排序，排序顺序如下**

@AfterThrowing  
@AfterReturning  
@After  
@Around  
@Before

下面我们来个@Aspect类，同时定义5种通知，然后来一步步分析一下其执行的属性。

package com.javacode2018.aop.demo11.test4;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.\*;  
import org.springframework.stereotype.Component;  
  
@Component  
@Aspect  
public class Aspect4 {  
 @Pointcut("execution(\* com.javacode2018.aop.demo11.test4.Service4.\*(..))")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void before() {  
 System.out.println("@Before通知!");  
 }  
  
 @Around("pc()")  
 public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
 System.out.println("@Around通知start");  
 Object result = joinPoint.proceed();  
 System.out.println("@Around绕通知end");  
 return result;  
 }  
  
 @After("pc()")  
 public void after() throws Throwable {  
 System.out.println("@After通知!");  
 }  
  
 @AfterReturning("pc()")  
 public void afterReturning() throws Throwable {  
 System.out.println("@AfterReturning通知!");  
 }  
  
 @AfterThrowing("pc()")  
 public void afterThrowing() {  
 System.out.println("@AfterThrowing通知!");  
 }  
  
}

上面会拦截com.javacode2018.aop.demo11.test4.Service4这个类中的所有方法，下面是Service4的源码。

package com.javacode2018.aop.demo11.test4;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service4 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
}

来个spring的配置类，使用@EnableAspectJAutoProxy标注

package com.javacode2018.aop.demo11.test4;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.EnableAspectJAutoProxy;  
  
@EnableAspectJAutoProxy  
@ComponentScan  
public class MainConfig4 {  
}

测试代码

@Test  
public void test4(){  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig4.class);  
 context.refresh();  
 Service4 service4 = context.getBean(Service4.class);  
 System.out.println(service4.say("路人"));  
}

运行依次输出如下

@Around通知start  
@Before通知!  
@Around绕通知end  
@After通知!  
@AfterReturning通知!  
你好：路人

**卧槽，这输出好像和我们上面说的不一样的，上面说的会按照下面的顺序执行，这到底是什么情况？**

@AfterThrowing  
@AfterReturning  
@After  
@Around  
@Before

**别急，排序规则和输出结果都没有问题，听我慢慢分析，下面的分析非常重要，注意看了**

### 5、@Aspect中5种通知回顾

#### 5种通知对应的Advice类

@Aspect中通过5中注解来定义通知，这些注解最终都需要转换为Advice去执行，转换关系如下

|  |  |
| --- | --- |
| 通知 | 对应的Advice类 |
| @AfterThrowing | org.springframework.aop.aspectj.AspectJAfterThrowingAdvice |
| @AfterReturning | org.springframework.aop.aspectj.AspectJAfterReturningAdvice |
| @After | org.springframework.aop.aspectj.AspectJAfterAdvice |
| @Around | org.springframework.aop.aspectj.AspectJAroundAdvice |
| @Before | org.springframework.aop.aspectj.AspectJMethodBeforeAdvice |

重点就在于表格右边的Advice类，当了解这些Advice的源码之后，他们的执行顺序大家就可以理解了，我们来看一下这些类的源码，重点看invoke方法

#### @AfterThrowing：AspectJAfterThrowingAdvice

public class AspectJAfterThrowingAdvice implements MethodInterceptor {  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //执行下一个拦截器  
 return mi.proceed();  
 } catch (Throwable ex) {  
 //通过反射调用@AfterThrowing标注的方法  
 //继续抛出异常  
 throw ex;  
 }  
 }  
}

AspectJAfterThrowingAdvice 实现了 MethodInterceptor 接口，不需要进行包装。

#### @AfterReturning：AspectJAfterReturningAdvice

AspectJAfterReturningAdvice 源码：

public class AspectJAfterReturningAdvice implements AfterReturningAdvice {  
 @Override  
 public void afterReturning(@Nullable Object returnValue, Method method, Object[] args, @Nullable Object target) throws Throwable {  
 // 调用@AfterReturning标注的方法  
 }  
}

AspectJAfterReturningAdvice 实现了 AfterReturningAdvice 接口，是一个方法返回通知，不是MethodInterceptor类型的，所以最终需包装为MethodInterceptor类型，变成下面这样

public class AspectJAfterReturningAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //执行下一个拦截器  
 Object retVal = mi.proceed();  
 //调用@AfterReturning标注的方法  
 return retVal;  
 }  
}

#### @After：AspectJAfterAdvice

AspectJAfterAdvice 源码：

public class AspectJAfterAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 //执行下一个拦截器  
 return mi.proceed();  
 } finally {  
 //调用@After标注的方法  
 }  
 }  
}

AspectJAfterAdvice 实现了 MethodInterceptor接口，所以最终执行的时候不需要进行包装。

**注意 invoke 方法内部使用了 try...finally 的方式，@After方法的调用放在了finally中，所以不管是否有异常，@After类型的通知都会被执行。**

#### @Around：AspectJAroundAdvice

AspectJAroundAdvice 源码：

public class AspectJAroundAdvice extends AbstractAspectJAdvice implements MethodInterceptor, Serializable {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 return 调用 @Around标注的方法 ;  
 }  
}

AspectJAroundAdvice 实现了 MethodInterceptor接口，最终执行的时候也不需要进行包装。

#### @Before：AspectJMethodBeforeAdvice

AspectJMethodBeforeAdvice 源码：

public class AspectJMethodBeforeAdvice implements MethodBeforeAdvice, Serializable {  
 @Override  
 public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {  
 invokeAdviceMethod(getJoinPointMatch(), null, null);  
 }  
}

AspectJMethodBeforeAdvice 实现了 MethodBeforeAdvice接口，是一个前置通知，不是MethodInterceptor类型的，所以最终需包装为MethodInterceptor类型，变成下面这样

public class AspectJMethodBeforeAdvice implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 //调用@Before标注的方法  
 //执行下一个拦截器  
 return mi.proceed();  
 }  
}

### 6、分析单个@Aspect中多个通知执行顺序

大家对@Aspect中5种通知内容理解之后，我们再回头看一下代码Aspect4中定义的5个通知

public class Aspect4 {  
 @Pointcut("execution(\* com.javacode2018.aop.demo11.test4.Service4.\*(..))")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void before() {  
 System.out.println("@Before通知!");  
 }  
  
 @Around("pc()")  
 public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
 System.out.println("@Around通知start");  
 Object result = joinPoint.proceed();  
 System.out.println("@Around绕通知end");  
 return result;  
 }  
  
 @After("pc()")  
 public void after() throws Throwable {  
 System.out.println("@After通知!");  
 }  
  
 @AfterReturning("pc()")  
 public void afterReturning() throws Throwable {  
 System.out.println("@AfterReturning通知!");  
 }  
  
 @AfterThrowing("pc()")  
 public void afterThrowing() {  
 System.out.println("@AfterThrowing通知!");  
 }  
  
}

我们给出的结论是，会按照下面的顺序执行

@AfterThrowing  
@AfterReturning  
@After  
@Around  
@Before

按照上面的顺序，一步步来分析。

先执行第1个通知@AfterThrowing，变成下面这样

try {  
 //执行下一个拦截器  
 return mi.proceed();  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

mi.processed()会执行第2个通知@AfterReturning，变成了下面这样

try {  
 //执行下一个拦截器  
 Object retVal = mi.proceed();  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

继续mi.proceed()执行第3个通知@After，变成了下面这样

try {  
 Object result = null;  
 try {  
 //执行下一个拦截器  
 result = mi.proceed();  
 } finally {  
 System.out.println("@After通知!");  
 }  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

继续mi.proceed()执行第4个通知@Around，变成了下面这样

try {  
 Object result = null;  
 try {  
 System.out.println("@Around通知start");  
 result = joinPoint.proceed();  
 System.out.println("@Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("@After通知!");  
 }  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

继续joinPoint.proceed()执行第5个通知@Before，变成了下面这样

try {  
 Object result = null;  
 try {  
 System.out.println("@Around通知start");  
 System.out.println("@Before通知!");  
 result = mi.proceed();  
 System.out.println("@Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("@After通知!");  
 }  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

继续joinPoint.proceed()会调用目标方法，变成了下面这样

try {  
 Object result = null;  
 try {  
 System.out.println("@Around通知start");  
 System.out.println("@Before通知!");  
 result = // 通过反射调用目标方法; //@1  
 System.out.println("@Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("@After通知!");  
 }  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

将上面的@1替换为目标方法的调用，就变成下面这样了

try {  
 Object result = null;  
 try {  
 System.out.println("@Around通知start");  
 System.out.println("@Before通知!");  
 result = service4.say("路人");  
 System.out.println("@Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("@After通知!");  
 }  
 System.out.println("@AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("@AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

所以最终输出

@Around通知start  
@Before通知!  
@Around绕通知end  
@After通知!  
@AfterReturning通知!  
你好：路人

### 7、@EnableAspectJAutoProxy中为通知指定顺序

@EnableAspectJAutoProxy用在spring环境中，可以通过@Aspect以及Advisor来定义多个通知，当spring容器中有多个@Aspect、Advisor时，他们的顺序是什么样的呢？

我们先看一下如何为@Aspect、自定义Advisor指定顺序。

#### 为@Aspect指定顺序：用@Order注解

需要在@Aspect标注的类上使用@org.springframework.core.annotation.Order注解，值越小，通知的优先级越高。

@Aspect  
@Order(1)  
public class AspectOrder1{}

#### 为Advisor指定顺序：实现Ordered接口

自定义的Advisor通过org.springframework.core.Ordered接口来指定顺序，这个接口有个public int getOrder()方法，用来返回通知的顺序。

spring为我们提供了一个Advisor类型的抽象类org.springframework.aop.support.AbstractPointcutAdvisor，这个类实现了Ordered接口，spring中大部分Advisor会是继承AbstractPointcutAdvisor，若需要自定义Advisor，也可以继承这个类，这个类的getOrder方法比较关键，来看一下

public abstract class AbstractPointcutAdvisor implements PointcutAdvisor, Ordered, Serializable {  
  
 @Nullable  
 private Integer order;  
  
 public void setOrder(int order) {  
 this.order = order;  
 }  
  
 @Override  
 public int getOrder() {  
 //若当前Advisor指定了order，则直接返回  
 if (this.order != null) {  
 return this.order;  
 }  
 //获取当前类中配置的通知对象Advice  
 Advice advice = getAdvice();  
 //若advice实现了Ordered接口，这从advice中获取通知的顺序  
 if (advice instanceof Ordered) {  
 return ((Ordered) advice).getOrder();  
 }  
 //否则通知的优先级最低，Integer.MAX\_VALUE  
 return Ordered.LOWEST\_PRECEDENCE;  
 }  
}

Spring为我们提供了一个默认的Advisor类：DefaultPointcutAdvisor，这个类就继承了AbstractPointcutAdvisor，通常我们可以直接使用DefaultPointcutAdvisor来自定义通知。

### 8、多个@Aspect、Advisor排序规则

#### 排序规则

**1、在spring容器中获取@Aspect、Advisor类型的所有bean，得到一个列表 list1**

**2、对list1按照order的值升序排序，得到结果list2**

**3、然后再对list2中@Aspect类型的bean内部的通知进行排序，规则**

@AfterThrowing  
@AfterReturning  
@After  
@Around  
@Before

4、最后运行的时候会得到上面排序产生的方法调用链列表去执行。

#### 案例

下面我们定义2个@Aspect类，一个Advisor类，并且给这3个都指定，然后来验证一下通知执行的顺序。

##### 先定义目标类

package com.javacode2018.aop.demo11.test2;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class Service2 {  
 public String say(String name) {  
 return "你好：" + name;  
 }  
}

##### Aspect1：第2个@Aspect

package com.javacode2018.aop.demo11.test2;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.\*;  
import org.springframework.core.annotation.Order;  
import org.springframework.stereotype.Component;  
  
@Aspect  
@Order(1)  
@Component  
public class MyAspect1 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo11.test2.Service2.\*(..))")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void before() {  
 System.out.println("MyAspect1 @Before通知!");  
 }  
  
 @Around("pc()")  
 public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
 System.out.println("MyAspect1 @Around通知start");  
 Object result = joinPoint.proceed();  
 System.out.println("MyAspect1 @Around绕通知end");  
 return result;  
 }  
  
 @After("pc()")  
 public void after() throws Throwable {  
 System.out.println("MyAspect1 @After通知!");  
 }  
  
 @AfterReturning("pc()")  
 public void afterReturning() throws Throwable {  
 System.out.println("MyAspect1 @AfterReturning通知!");  
 }  
  
 @AfterThrowing("pc()")  
 public void afterThrowing() {  
 System.out.println("MyAspect1 @AfterThrowing通知!");  
 }  
  
}

##### Aspect1：第2个@Aspect

package com.javacode2018.aop.demo11.test2;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.\*;  
import org.springframework.core.annotation.Order;  
import org.springframework.stereotype.Component;  
  
@Aspect  
@Order(3)  
@Component  
public class MyAspect2 {  
  
 @Pointcut("execution(\* com.javacode2018.aop.demo11.test2.Service2.\*(..))")  
 public void pc() {  
 }  
  
 @Before("pc()")  
 public void before() {  
 System.out.println("MyAspect2 @Before通知!");  
 }  
  
 @Around("pc()")  
 public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
 System.out.println("MyAspect2 @Around通知start");  
 Object result = joinPoint.proceed();  
 System.out.println("MyAspect2 @Around绕通知end");  
 return result;  
 }  
  
 @After("pc()")  
 public void after() throws Throwable {  
 System.out.println("MyAspect2 @After通知!");  
 }  
  
 @AfterReturning("pc()")  
 public void afterReturning() throws Throwable {  
 System.out.println("MyAspect2 @AfterReturning通知!");  
 }  
  
 @AfterThrowing("pc()")  
 public void afterThrowing() {  
 System.out.println("MyAspect2 @AfterThrowing通知!");  
 }  
  
}

##### 自定义一个Advisor

package com.javacode2018.aop.demo11.test2;  
  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.aop.support.DefaultPointcutAdvisor;  
import org.springframework.stereotype.Component;  
  
@Component  
public class Advisor1 extends DefaultPointcutAdvisor {  
  
 public Advisor1() {  
 MethodInterceptor methodInterceptor = new MethodInterceptor() {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 System.out.println("Advisor1 start");  
 Object result = invocation.proceed();  
 System.out.println("Advisor1 end");  
 return result;  
 }  
 };  
 this.setAdvice(methodInterceptor);  
 }  
  
 @Override  
 public int getOrder() {  
 return 2;  
 }  
}

##### 来个spring配置类

标注@EnableAspectJAutoProxy来启用自动化的aop功能

package com.javacode2018.aop.demo11.test2;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.EnableAspectJAutoProxy;  
  
@ComponentScan  
@EnableAspectJAutoProxy  
public class MainConfig2 {  
}

##### 测试代码

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig2.class);  
 context.refresh();  
 Service2 service2 = context.getBean(Service2.class);  
 System.out.println(service2.say("路人"));  
}

##### 运行输出

MyAspect1 @Around通知start  
MyAspect1 @Before通知!  
Advisor1 start  
MyAspect2 @Around通知start  
MyAspect2 @Before通知!  
MyAspect2 @Around绕通知end  
MyAspect2 @After通知!  
MyAspect2 @AfterReturning通知!  
Advisor1 end  
MyAspect1 @Around绕通知end  
MyAspect1 @After通知!  
MyAspect1 @AfterReturning通知!  
你好：路人

##### 结果分析

下面我们一步步来推出结果为什么是上面这样。

先获取spring容器中@Aspect、Advisor类型的所有bean，根据其order升序排序，得到：

Aspect1：顺序是1  
Advisor1：顺序是2  
MyAspect2：顺序是3

然后对每个Aspect内部的通知进行排序，根据单个@Aspect内部通知排序规则，可以得到：

Aspect1：顺序是1  
 @AfterThrowing  
 @AfterReturning  
 @After  
 @Around  
 @Before  
  
Advisor1：顺序是2  
  
MyAspect2：顺序是3  
 @AfterThrowing  
 @AfterReturning  
 @After  
 @Around  
 @Before

下面将代码拿过来一步步填充。

先对Aspect1进行填充，得到：

try {  
 Object result = null;  
 try {  
 System.out.println("MyAspect1 @Around通知start");  
 System.out.println("MyAspect1 @Before通知!");  
 result = mi.proceed(); //@1  
 System.out.println("MyAspect1 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect1 @After通知!");  
 }  
 System.out.println("MyAspect1 @AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("MyAspect1 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

@1执行mi.proceed()会调用下一个拦截器，即Advisor1中定义的拦截器，然后会得到下面代码：

try {  
 Object result = null;  
 try {  
 System.out.println("MyAspect1 @Around通知start");  
 System.out.println("MyAspect1 @Before通知!");  
 System.out.println("Advisor1 start");  
 result = invocation.proceed(); //@2  
 System.out.println("Advisor1 end");  
 System.out.println("MyAspect1 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect1 @After通知!");  
 }  
 System.out.println("MyAspect1 @AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("MyAspect1 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

@2执行mi.proceed()会调用下一个拦截器，即Aspect2中定义的拦截器，而Aspect2和Aspect1类似，然后会得到下面代码：

try {  
 Object result = null;  
 try {  
 System.out.println("MyAspect1 @Around通知start");  
 System.out.println("MyAspect1 @Before通知!");  
 System.out.println("Advisor1 start");  
 try {  
 try {  
 System.out.println("MyAspect2 @Around通知start");  
 System.out.println("MyAspect2 @Before通知!");  
 result = mi.proceed(); //@3  
 System.out.println("MyAspect2 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect2 @After通知!");  
 }  
 System.out.println("MyAspect2 @AfterReturning通知!");  
 } catch (Throwable ex) {  
 System.out.println("MyAspect2 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
 }  
 System.out.println("Advisor1 end");  
 System.out.println("MyAspect1 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect1 @After通知!");  
 }  
 System.out.println("MyAspect1 @AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("MyAspect1 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

@3继续执行mi.proceed()，此时会调用目标方法say("路人")，然后就进化成下面这样了

try {  
 Object result = null;  
 try {  
 System.out.println("MyAspect1 @Around通知start");  
 System.out.println("MyAspect1 @Before通知!");  
 System.out.println("Advisor1 start");  
 try {  
 try {  
 System.out.println("MyAspect2 @Around通知start");  
 System.out.println("MyAspect2 @Before通知!");  
 result = "你好：路人";  
 System.out.println("MyAspect2 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect2 @After通知!");  
 }  
 System.out.println("MyAspect2 @AfterReturning通知!");  
 } catch (Throwable ex) {  
 System.out.println("MyAspect2 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
 }  
 System.out.println("Advisor1 end");  
 System.out.println("MyAspect1 @Around绕通知end");  
 return result;  
 } finally {  
 System.out.println("MyAspect1 @After通知!");  
 }  
 System.out.println("MyAspect1 @AfterReturning通知!");  
 return retVal;  
} catch (Throwable ex) {  
 System.out.println("MyAspect1 @AfterThrowing通知!");  
 //继续抛出异常  
 throw ex;  
}

再来和输出结果对比一下，是完全一致的。

MyAspect1 @Around通知start  
MyAspect1 @Before通知!  
Advisor1 start  
MyAspect2 @Around通知start  
MyAspect2 @Before通知!  
MyAspect2 @Around绕通知end  
MyAspect2 @After通知!  
MyAspect2 @AfterReturning通知!  
Advisor1 end  
MyAspect1 @Around绕通知end  
MyAspect1 @After通知!  
MyAspect1 @AfterReturning通知!  
你好：路人

### 9、@EnableAspectJAutoProxy另外2个功能

这个注解还有2个参数，大家看一下下面的注释，比较简单，就不用案例演示了。

public @interface EnableAspectJAutoProxy {  
  
 /\*\*  
 \* 是否基于类来创建代理，而不是基于接口来创建代理  
 \* 当为true的时候会使用cglib来直接对目标类创建代理对象  
 \* 默认为 false：即目标bean如果有接口的会采用jdk动态代理来创建代理对象，没有接口的目标bean，会采用cglib来创建代理对象  
 \*/  
 boolean proxyTargetClass() default false;  
  
 /\*\*  
 \* 是否需要将代理对象暴露在ThreadLocal中，当为true的时候  
 \* 可以通过org.springframework.aop.framework.AopContext#currentProxy获取当前代理对象  
 \*/  
 boolean exposeProxy() default false;  
  
}

### 10、@EnableAspectJAutoProxy原理

@EnableAspectJAutoProxy会在spring容器中注册一个bean

org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator

AnnotationAwareAspectJAutoProxyCreator是BeanPostProcessor类型的，BeanPostProcessor大家应该比较熟悉了，bean后置处理器，可以在bean声明周期中对bean进行操作，比如对bean生成代理等；而AnnotationAwareAspectJAutoProxyCreator就是对符合条件的bean，自动生成代理对象，源码就这里就不细说了，有兴趣的可以从postProcessAfterInitialization方法看，比较简单。

### 11、总结

今天内容还是挺多的，大家好好消化一下。

主要要掌握@EnableAspectJAutoProxy中多个@Aspect、Advisor时，通知的执行顺序，这个多看看，要理解其原理，记起来才会更容易，用起来也会更顺手。

### 12、案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 13、Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解

## Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用

### 1、本文内容

详解 @EnableAsync & @Async，主要分下面几个点进行介绍。

1. **作用**
2. **用法**
3. **获取异步执行结果**
4. **自定义异步执行的线程池**
5. **自定义异常处理**
6. **线程隔离**
7. **源码 & 原理**

### 2、作用

**spring容器中实现bean方法的异步调用。**

比如有个logService的bean，logservice中有个log方法用来记录日志，当调用logService.log(msg)的时候，希望异步执行，那么可以通过@EnableAsync & @Async来实现。

### 3、用法

#### 2步

1. 需要异步执行的方法上面使用@Async注解标注，若bean中所有的方法都需要异步执行，可以直接将@Async加载类上。
2. 将@EnableAsync添加在spring配置类上，此时@Async注解才会起效。

#### 常见2种用法

1. 无返回值的
2. 可以获取返回值的

### 4、无返回值的

#### 用法

方法返回值不是Future类型的，被执行时，会立即返回，并且无法获取方法返回值，如：

@Async  
public void log(String msg) throws InterruptedException {  
 System.out.println("开始记录日志," + System.currentTimeMillis());  
 //模拟耗时2秒  
 TimeUnit.SECONDS.sleep(2);  
 System.out.println("日志记录完毕," + System.currentTimeMillis());  
}

#### 案例

实现日志异步记录的功能。

LogService.log方法用来异步记录日志，需要使用@Async标注

package com.javacode2018.async.demo1;  
  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.stereotype.Component;  
  
import java.util.concurrent.TimeUnit;  
  
@Component  
public class LogService {  
 @Async  
 public void log(String msg) throws InterruptedException {  
 System.out.println(Thread.currentThread() + "开始记录日志," + System.currentTimeMillis());  
 //模拟耗时2秒  
 TimeUnit.SECONDS.sleep(2);  
 System.out.println(Thread.currentThread() + "日志记录完毕," + System.currentTimeMillis());  
 }  
}

来个spring配置类，需要加上@EnableAsync开启bean方法的异步调用.

package com.javacode2018.async.demo1;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.EnableAspectJAutoProxy;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@ComponentScan  
@EnableAsync  
public class MainConfig1 {  
}

测试代码

package com.javacode2018.async;  
  
import com.javacode2018.async.demo1.LogService;  
import com.javacode2018.async.demo1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.concurrent.TimeUnit;  
  
public class AsyncTest {  
  
 @Test  
 public void test1() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 LogService logService = context.getBean(LogService.class);  
 System.out.println(Thread.currentThread() + " logService.log start," + System.currentTimeMillis());  
 logService.log("异步执行方法!");  
 System.out.println(Thread.currentThread() + " logService.log end," + System.currentTimeMillis());  
  
 //休眠一下，防止@Test退出  
 TimeUnit.SECONDS.sleep(3);  
 }  
  
}

运行输出

Thread[main,5,main] logService.log start,1595223990417  
Thread[main,5,main] logService.log end,1595223990432  
Thread[SimpleAsyncTaskExecutor-1,5,main]开始记录日志,1595223990443  
Thread[SimpleAsyncTaskExecutor-1,5,main]日志记录完毕,1595223992443

前2行输出，可以看出logService.log立即就返回了，后面2行来自于log方法，相差2秒左右。

前面2行在主线程中执行，后面2行在异步线程中执行。

### 5、获取异步返回值

#### 用法

若需取异步执行结果，方法返回值必须为Future类型，使用spring提供的静态方法org.springframework.scheduling.annotation.AsyncResult#forValue创建返回值，如：

public Future<String> getGoodsInfo(long goodsId) throws InterruptedException {  
 return AsyncResult.forValue(String.format("商品%s基本信息!", goodsId));  
}

#### 案例

场景：电商中商品详情页通常会有很多信息：商品基本信息、商品描述信息、商品评论信息，通过3个方法来或者这几个信息。

这3个方法之间无关联，所以可以采用异步的方式并行获取，提升效率。

下面是商品服务，内部3个方法都需要异步，所以直接在类上使用@Async标注了，每个方法内部休眠500毫秒，模拟一下耗时操作。

package com.javacode2018.async.demo2;  
  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.scheduling.annotation.AsyncResult;  
import org.springframework.stereotype.Component;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.concurrent.Future;  
import java.util.concurrent.TimeUnit;  
  
@Async  
@Component  
public class GoodsService {  
 //模拟获取商品基本信息，内部耗时500毫秒  
 public Future<String> getGoodsInfo(long goodsId) throws InterruptedException {  
 TimeUnit.MILLISECONDS.sleep(500);  
 return AsyncResult.forValue(String.format("商品%s基本信息!", goodsId));  
 }  
  
 //模拟获取商品描述信息，内部耗时500毫秒  
 public Future<String> getGoodsDesc(long goodsId) throws InterruptedException {  
 TimeUnit.MILLISECONDS.sleep(500);  
 return AsyncResult.forValue(String.format("商品%s描述信息!", goodsId));  
 }  
  
 //模拟获取商品评论信息列表，内部耗时500毫秒  
 public Future<List<String>> getGoodsComments(long goodsId) throws InterruptedException {  
 TimeUnit.MILLISECONDS.sleep(500);  
 List<String> comments = Arrays.asList("评论1", "评论2");  
 return AsyncResult.forValue(comments);  
 }  
}

来个spring配置类，需要加上@EnableAsync开启bean方法的异步调用.

package com.javacode2018.async.demo2;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@ComponentScan  
@EnableAsync  
public class MainConfig2 {  
}

测试代码

@Test  
public void test2() throws InterruptedException, ExecutionException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig2.class);  
 context.refresh();  
 GoodsService goodsService = context.getBean(GoodsService.class);  
  
 long starTime = System.currentTimeMillis();  
 System.out.println("开始获取商品的各种信息");  
  
 long goodsId = 1L;  
 Future<String> goodsInfoFuture = goodsService.getGoodsInfo(goodsId);  
 Future<String> goodsDescFuture = goodsService.getGoodsDesc(goodsId);  
 Future<List<String>> goodsCommentsFuture = goodsService.getGoodsComments(goodsId);  
  
 System.out.println(goodsInfoFuture.get());  
 System.out.println(goodsDescFuture.get());  
 System.out.println(goodsCommentsFuture.get());  
  
 System.out.println("商品信息获取完毕,总耗时(ms)：" + (System.currentTimeMillis() - starTime));  
  
 //休眠一下，防止@Test退出  
 TimeUnit.SECONDS.sleep(3);  
}

运行输出

开始获取商品的各种信息  
商品1基本信息!  
商品1描述信息!  
[评论1, 评论2]  
商品信息获取完毕,总耗时(ms)：525

3个方法总计耗时500毫秒左右。

如果不采用异步的方式，3个方法会同步执行，耗时差不多1.5秒，来试试，将GoodsService上的@Async去掉，然后再次执行测试案例，输出

开始获取商品的各种信息  
商品1基本信息!  
商品1描述信息!  
[评论1, 评论2]  
商品信息获取完毕,总耗时(ms)：1503

这个案例大家可以借鉴一下，**按照这个思路可以去优化一下你们的代码，方法之间无关联的可以采用异步的方式，并行去获取，最终耗时为最长的那个方法，整体相对于同步的方式性能提升不少。**

### 6、自定义异步执行的线程池

默认情况下，@EnableAsync使用内置的线程池来异步调用方法，不过我们也可以自定义异步执行任务的线程池。

#### 有2种方式来自定义异步处理的线程池

##### 方式1

**在spring容器中定义一个线程池类型的bean，bean名称必须是taskExecutor**

@Bean  
public Executor taskExecutor() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(10);  
 executor.setMaxPoolSize(100);  
 executor.setThreadNamePrefix("my-thread-");  
 return executor;  
}

##### 方式2

定义一个bean，实现AsyncConfigurer接口中的getAsyncExecutor方法，这个方法需要返回自定义的线程池，案例代码：

package com.javacode2018.async.demo3;  
  
import com.javacode2018.async.demo1.LogService;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.lang.Nullable;  
import org.springframework.scheduling.annotation.AsyncConfigurer;  
import org.springframework.scheduling.annotation.EnableAsync;  
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
import java.util.concurrent.Executor;  
  
@EnableAsync  
public class MainConfig3 {  
  
 @Bean  
 public LogService logService() {  
 return new LogService();  
 }  
  
 /\*\*  
 \* 定义一个AsyncConfigurer类型的bean，实现getAsyncExecutor方法，返回自定义的线程池  
 \*  
 \* @param executor  
 \* @return  
 \*/  
 @Bean  
 public AsyncConfigurer asyncConfigurer(@Qualifier("logExecutors") Executor executor) {  
 return new AsyncConfigurer() {  
 @Nullable  
 @Override  
 public Executor getAsyncExecutor() {  
 return executor;  
 }  
 };  
 }  
  
 /\*\*  
 \* 定义一个线程池，用来异步处理日志方法调用  
 \*  
 \* @return  
 \*/  
 @Bean  
 public Executor logExecutors() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(10);  
 executor.setMaxPoolSize(100);  
 //线程名称前缀  
 executor.setThreadNamePrefix("log-thread-"); //@1  
 return executor;  
 }  
  
}

@1自定义的线程池中线程名称前缀为log-thread-，运行下面测试代码

@Test  
public void test3() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig3.class);  
 context.refresh();  
 LogService logService = context.getBean(LogService.class);  
 System.out.println(Thread.currentThread() + " logService.log start," + System.currentTimeMillis());  
 logService.log("异步执行方法!");  
 System.out.println(Thread.currentThread() + " logService.log end," + System.currentTimeMillis());  
  
 //休眠一下，防止@Test退出  
 TimeUnit.SECONDS.sleep(3);  
}

输出

Thread[main,5,main] logService.log start,1595228732914  
Thread[main,5,main] logService.log end,1595228732921  
Thread[log-thread-1,5,main]开始记录日志,1595228732930  
Thread[log-thread-1,5,main]日志记录完毕,1595228734931

最后2行日志中线程名称是log-thread-，正是我们自定义线程池中的线程。

### 7、自定义异常处理

异步方法若发生了异常，我们如何获取异常信息呢？此时可以通过自定义异常处理来解决。

#### 异常处理分2种情况

1. 当返回值是Future的时候，方法内部有异常的时候，异常会向外抛出，可以对Future.get采用try..catch来捕获异常
2. 当返回值不是Future的时候，可以自定义一个bean，实现AsyncConfigurer接口中的getAsyncUncaughtExceptionHandler方法，返回自定义的异常处理器

#### 情况1：返回值为Future类型

##### 用法

通过try..catch来捕获异常，如下

try {  
 Future<String> future = logService.mockException();  
 System.out.println(future.get());  
} catch (ExecutionException e) {  
 System.out.println("捕获 ExecutionException 异常");  
 //通过e.getCause获取实际的异常信息  
 e.getCause().printStackTrace();  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}

##### 案例

LogService中添加一个方法，返回值为Future，内部抛出一个异常，如下：

@Async  
public Future<String> mockException() {  
 //模拟抛出一个异常  
 throw new IllegalArgumentException("参数有误!");  
}

测试代码如下

@Test  
public void test5() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 LogService logService = context.getBean(LogService.class);  
 try {  
 Future<String> future = logService.mockException();  
 System.out.println(future.get());  
 } catch (ExecutionException e) {  
 System.out.println("捕获 ExecutionException 异常");  
 //通过e.getCause获取实际的异常信息  
 e.getCause().printStackTrace();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //休眠一下，防止@Test退出  
 TimeUnit.SECONDS.sleep(3);  
}

运行输出

java.lang.IllegalArgumentException: 参数有误!  
捕获 ExecutionException 异常  
 at com.javacode2018.async.demo1.LogService.mockException(LogService.java:23)  
 at com.javacode2018.async.demo1.LogService$$FastClassBySpringCGLIB$$32a28430.invoke(<generated>)  
 at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)

#### 情况2：无返回值异常处理

##### 用法

当返回值不是Future的时候，可以自定义一个bean，实现AsyncConfigurer接口中的getAsyncUncaughtExceptionHandler方法，返回自定义的异常处理器，当目标方法执行过程中抛出异常的时候，此时会自动回调AsyncUncaughtExceptionHandler#handleUncaughtException这个方法，可以在这个方法中处理异常，如下：

@Bean  
public AsyncConfigurer asyncConfigurer() {  
 return new AsyncConfigurer() {  
 @Nullable  
 @Override  
 public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  
 return new AsyncUncaughtExceptionHandler() {  
 @Override  
 public void handleUncaughtException(Throwable ex, Method method, Object... params) {  
 //当目标方法执行过程中抛出异常的时候，此时会自动回调这个方法，可以在这个方法中处理异常  
 }  
 };  
 }  
 };  
}

##### 案例

LogService中添加一个方法，内部抛出一个异常，如下：

@Async  
public void mockNoReturnException() {  
 //模拟抛出一个异常  
 throw new IllegalArgumentException("无返回值的异常!");  
}

来个spring配置类，通过AsyncConfigurer来自定义异常处理器AsyncUncaughtExceptionHandler

package com.javacode2018.async.demo4;  
  
import com.javacode2018.async.demo1.LogService;  
import org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler;  
import org.springframework.context.annotation.Bean;  
import org.springframework.lang.Nullable;  
import org.springframework.scheduling.annotation.AsyncConfigurer;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
import java.lang.reflect.Method;  
import java.util.Arrays;  
  
@EnableAsync  
public class MainConfig4 {  
  
 @Bean  
 public LogService logService() {  
 return new LogService();  
 }  
  
 @Bean  
 public AsyncConfigurer asyncConfigurer() {  
 return new AsyncConfigurer() {  
 @Nullable  
 @Override  
 public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  
 return new AsyncUncaughtExceptionHandler() {  
 @Override  
 public void handleUncaughtException(Throwable ex, Method method, Object... params) {  
 String msg = String.format("方法[%s],参数[%s],发送异常了，异常详细信息:", method, Arrays.asList(params));  
 System.out.println(msg);  
 ex.printStackTrace();  
 }  
 };  
 }  
 };  
 }  
  
}

运行输出

方法[public void com.javacode2018.async.demo1.LogService.mockNoReturnException()],参数[[]],发送异常了，异常详细信息:  
java.lang.IllegalArgumentException: 无返回值的异常!  
 at com.javacode2018.async.demo1.LogService.mockNoReturnException(LogService.java:29)  
 at com.javacode2018.async.demo1.LogService$$FastClassBySpringCGLIB$$32a28430.invoke(<generated>)  
 at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)

### 8、线程池隔离

#### 什么是线程池隔离？

一个系统中可能有很多业务，比如充值服务、提现服务或者其他服务，这些服务中都有一些方法需要异步执行，默认情况下他们会使用同一个线程池去执行，如果有一个业务量比较大，占用了线程池中的大量线程，此时会导致其他业务的方法无法执行，那么我们可以采用线程隔离的方式，对不同的业务使用不同的线程池，相互隔离，互不影响。

@Async注解有个value参数，用来指定线程池的bean名称，方法运行的时候，就会采用指定的线程池来执行目标方法。

#### 使用步骤

1. 在spring容器中，自定义线程池相关的bean
2. @Async("线程池bean名称")

#### 案例

模拟2个业务：异步充值、异步提现；2个业务都采用独立的线程池来异步执行，互不影响。

###### 异步充值服务

package com.javacode2018.async.demo5;  
  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.stereotype.Component;  
  
@Component  
public class RechargeService {  
 //模拟异步充值  
 @Async(MainConfig5.RECHARGE\_EXECUTORS\_BEAN\_NAME)  
 public void recharge() {  
 System.out.println(Thread.currentThread() + "模拟异步充值");  
 }  
}

###### 异步提现服务

package com.javacode2018.async.demo5;  
  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.stereotype.Component;  
  
@Component  
public class CashOutService {  
 //模拟异步提现  
 @Async(MainConfig5.CASHOUT\_EXECUTORS\_BEAN\_NAME)  
 public void cashOut() {  
 System.out.println(Thread.currentThread() + "模拟异步提现");  
 }  
}

###### spring配置类

注意@0、@1、@2、@3、@4这几个地方的代码，采用线程池隔离的方式，注册了2个线程池，分别用来处理上面的2个异步业务。

package com.javacode2018.async.demo5;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.scheduling.annotation.EnableAsync;  
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
import java.util.concurrent.Executor;  
  
@EnableAsync //@0：启用方法异步调用  
@ComponentScan  
public class MainConfig5 {  
  
 //@1：值业务线程池bean名称  
 public static final String RECHARGE\_EXECUTORS\_BEAN\_NAME = "rechargeExecutors";  
 //@2：提现业务线程池bean名称  
 public static final String CASHOUT\_EXECUTORS\_BEAN\_NAME = "cashOutExecutors";  
  
 /\*\*  
 \* @3：充值的线程池，线程名称以recharge-thread-开头  
 \* @return  
 \*/  
 @Bean(RECHARGE\_EXECUTORS\_BEAN\_NAME)  
 public Executor rechargeExecutors() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(10);  
 executor.setMaxPoolSize(100);  
 //线程名称前缀  
 executor.setThreadNamePrefix("recharge-thread-");  
 return executor;  
 }  
  
 /\*\*  
 \* @4: 充值的线程池，线程名称以cashOut-thread-开头  
 \*  
 \* @return  
 \*/  
 @Bean(CASHOUT\_EXECUTORS\_BEAN\_NAME)  
 public Executor cashOutExecutors() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(10);  
 executor.setMaxPoolSize(100);  
 //线程名称前缀  
 executor.setThreadNamePrefix("cashOut-thread-");  
 return executor;  
 }  
}

###### 测试代码

@Test  
public void test7() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig5.class);  
 context.refresh();  
  
 RechargeService rechargeService = context.getBean(RechargeService.class);  
 rechargeService.recharge();  
 CashOutService cashOutService = context.getBean(CashOutService.class);  
 cashOutService.cashOut();  
  
 //休眠一下，防止@Test退出  
 TimeUnit.SECONDS.sleep(3);  
}

###### 运行输出

Thread[recharge-thread-1,5,main]模拟异步充值  
Thread[cashOut-thread-1,5,main]模拟异步提现

输出中可以看出2个业务使用的是不同的线程池执行的。

### 9、源码 & 原理

内部使用aop实现的，@EnableAsync会引入一个bean后置处理器：AsyncAnnotationBeanPostProcessor，将其注册到spring容器，这个bean后置处理器在所有bean创建过程中，判断bean的类上是否有@Async注解或者类中是否有@Async标注的方法，如果有，会通过aop给这个bean生成代理对象，会在代理对象中添加一个切面：org.springframework.scheduling.annotation.AsyncAnnotationAdvisor，这个切面中会引入一个拦截器：AnnotationAsyncExecutionInterceptor，方法异步调用的关键代码就是在这个拦截器的invoke方法中实现的，可以去看一下。

### 10、总结



### 11、案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### 12、Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解

## Spring系列第38篇：@Scheduled & @EnableScheduling定时器详解

spring中 @Scheduled & @EnableScheduling 这2个注解，可以用来快速开发定时器，使用特别的简单。

### 如何使用？

#### 用法

1、需要定时执行的方法上加上@Scheduled注解，这个注解中可以指定定时执行的规则，稍后详细介绍。

2、Spring容器中使用@EnableScheduling开启定时任务的执行，此时spring容器才可以识别@Scheduled标注的方法，然后自动定时执行。

#### 案例

db中有很多需要推送的任务，然后将其检索出来，推送到手机端，来个定时器，每秒一次从库中检测需要推送的消息，然后推送到手机端。

package com.javacode2018.scheduled.demo1;  
  
import org.springframework.scheduling.annotation.Scheduled;  
import org.springframework.stereotype.Component;  
  
@Component  
public class PushJob {  
  
 //推送方法，每秒执行一次  
 @Scheduled(fixedRate = 1000)  
 public void push() throws InterruptedException {  
 System.out.println("模拟推送消息，" + System.currentTimeMillis());  
 }  
  
}

来个spring配置类，需要使用@EnableScheduling标注

package com.javacode2018.scheduled.demo1;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.scheduling.annotation.EnableScheduling;  
  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
  
@ComponentScan  
@EnableScheduling //在spring容器中启用定时任务的执行  
public class MainConfig1 {  
  
 @Bean  
 public ScheduledExecutorService scheduledExecutorService() {  
 return Executors.newScheduledThreadPool(20);  
 }  
}

测试类

package com.javacode2018.scheduled;  
  
import com.javacode2018.scheduled.demo1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.concurrent.TimeUnit;  
  
public class ScheduledTest {  
 @Test  
 public void test1() throws InterruptedException {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 //休眠一段时间，房子junit自动退出  
 TimeUnit.SECONDS.sleep(10000);  
 }  
  
}

运行输出，每秒会输出一次，如下：

模拟推送消息，1595840822998  
模拟推送消息，1595840823998  
模拟推送消息，1595840824998  
模拟推送消息，1595840825998  
模拟推送消息，1595840826998  
模拟推送消息，1595840827998  
模拟推送消息，1595840828998

### @Scheduled配置定时规则

@Scheduled可以用来配置定时器的执行规则，非常强大，@Scheduled中主要有8个参数，我们一一来了解一下。

@Target({ElementType.METHOD, ElementType.ANNOTATION\_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Repeatable(Schedules.class)  
public @interface Scheduled {  
  
 String cron() default "";  
  
 String zone() default "";  
  
 long fixedDelay() default -1;  
  
 String fixedDelayString() default "";  
  
 long fixedRate() default -1;  
  
 String fixedRateString() default "";  
  
 long initialDelay() default -1;  
  
 String initialDelayString() default "";  
  
}

#### 1. cron

该参数接收一个cron表达式，cron表达式是一个字符串，字符串以5或6个空格隔开，分开共6或7个域，每一个域代表一个含义。

##### cron表达式语法

[秒] [分] [小时] [日] [月] [周] [年]

注：[年]不是必须的域，可以省略[年]，则一共6个域

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 说明 | 必填 | 允许填写的值 | 允许的通配符 |
| 1 | 秒 | 是 | 0-59 | , - \* / |
| 2 | 分 | 是 | 0-59 | , - \* / |
| 3 | 时 | 是 | 0-23 | , - \* / |
| 4 | 日 | 是 | 1-31 | , - \* ? / L W |
| 5 | 月 | 是 | 1-12 / JAN-DEC | , - \* / |
| 6 | 周 | 是 | 1-7 or SUN-SAT | , - \* ? / L # |
| 7 | 年 | 否 | 1970-2099 | , - \* / |

###### 通配符说明:

* \* 表示所有值。 例如:在分的字段上设置 \*,表示每一分钟都会触发。
* ? 表示不指定值。使用的场景为不需要关心当前设置这个字段的值。例如:要在每月的10号触发一个操作，但不关心是周几，所以需要周位置的那个字段设置为”?” 具体设置为 0 0 0 10 \* ?
* - 表示区间。例如 在小时上设置 “10-12”,表示 10,11,12点都会触发。
* , 表示指定多个值，例如在周字段上设置 “MON,WED,FRI” 表示周一，周三和周五触发
* / 用于递增触发。如在秒上面设置”5/15” 表示从5秒开始，每增15秒触发(5,20,35,50)。 在日字段上设置’1/3’所示每月1号开始，每隔三天触发一次。
* L 表示最后的意思。在日字段设置上，表示当月的最后一天(依据当前月份，如果是二月还会依据是否是润年[leap]), 在周字段上表示星期六，相当于”7”或”SAT”。如果在”L”前加上数字，则表示该数据的最后一个。例如在周字段上设置”6L”这样的格式,则表示“本月最后一个星期五”
* W 表示离指定日期的最近那个工作日(周一至周五). 例如在日字段上置”15W”，表示离每月15号最近的那个工作日触发。如果15号正好是周六，则找最近的周五(14号)触发, 如果15号是周未，则找最近的下周一(16号)触发.如果15号正好在工作日(周一至周五)，则就在该天触发。如果指定格式为 “1W”,它则表示每月1号往后最近的工作日触发。如果1号正是周六，则将在3号下周一触发。(注，”W”前只能设置具体的数字,不允许区间”-“)。
* # 序号(表示每月的第几个周几)，例如在周字段上设置”6#3”表示在每月的第三个周六.注意如果指定”#5”,正好第五周没有周六，则不会触发该配置(用在母亲节和父亲节再合适不过了) ；小提示：’L’和 ‘W’可以一组合使用。如果在日字段上设置”LW”,则表示在本月的最后一个工作日触发；周字段的设置，若使用英文字母是不区分大小写的，即MON与mon相同。

###### 示例

每隔5秒执行一次：\*/5 \* \* \* \* ?

每隔1分钟执行一次：0 \*/1 \* \* \* ?

每天23点执行一次：0 0 23 \* \* ?

每天凌晨1点执行一次：0 0 1 \* \* ?

每月1号凌晨1点执行一次：0 0 1 1 \* ?

每月最后一天23点执行一次：0 0 23 L \* ?

每周星期六凌晨1点实行一次：0 0 1 ? \* L

在26分、29分、33分执行一次：0 26,29,33 \* \* \* ?

每天的0点、13点、18点、21点都执行一次：0 0 0,13,18,21 \* \* ?

###### cron表达式使用占位符

另外，cron属性接收的cron表达式支持占位符。

如：配置文件：

time:  
 cron: \*/5 \* \* \* \* \*  
 interval: 5

每5秒执行一次：

@Scheduled(cron="${time.cron}")  
void testPlaceholder1() {  
 System.out.println("Execute at " + System.currentTimeMillis());  
}  
  
@Scheduled(cron="\*/${time.interval} \* \* \* \* \*")  
void testPlaceholder2() {  
 System.out.println("Execute at " + System.currentTimeMillis());  
}

#### 2. zone

时区，接收一个java.util.TimeZone#ID。cron表达式会基于该时区解析。默认是一个空字符串，即取服务器所在地的时区。比如我们一般使用的时区Asia/Shanghai。该字段我们一般留空。

#### 3. fixedDelay

上一次执行完毕时间点之后多长时间再执行。

如：

@Scheduled(fixedDelay = 5000) //上一次执行完毕时间点之后5秒再执行

#### 4. fixedDelayString

与 3. fixedDelay 意思相同，只是使用字符串的形式。唯一不同的是支持占位符。

如：

@Scheduled(fixedDelayString = "5000") //上一次执行完毕时间点之后5秒再执行

占位符的使用（配置文件中有配置：time.fixedDelay=5000）

@Scheduled(fixedDelayString = "${time.fixedDelay}")  
void testFixedDelayString() {  
 System.out.println("Execute at " + System.currentTimeMillis());  
}

#### 5. fixedRate

上一次开始执行时间点之后多长时间再执行。

如：

@Scheduled(fixedRate = 5000) //上一次开始执行时间点之后5秒再执行

#### 6. fixedRateString

与 fixedRate 意思相同，只是使用字符串的形式，唯一不同的是支持占位符。

#### 7. initialDelay

第一次延迟多长时间后再执行。

如：

@Scheduled(initialDelay=1000, fixedRate=5000) //第一次延迟1秒后执行，之后按fixedRate的规则每5秒执行一次

#### 8. initialDelayString

与 initialDelay 意思相同，只是使用字符串的形式，唯一不同的是支持占位符。

### @Schedules注解

这个注解不用多解释，看一下源码就知道作用了，当一个方法上面需要同时指定多个定时规则的时候，可以通过这个来配置

@Target({ElementType.METHOD, ElementType.ANNOTATION\_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Schedules {  
 Scheduled[] value();  
}

如：

//2个定时器，500毫秒的，1000毫秒的  
@Schedules({@Scheduled(fixedRate = 500), @Scheduled(fixedRate = 1000)})  
public void push3() {  
}

### 为定时器定义线程池

定时器默认情况下使用下面的线程池来执行定时任务的

new ScheduledThreadPoolExecutor(1)

只有一个线程，相当于只有一个干活的人，如果需要定时执行的任务太多，这些任务只能排队执行，会出现什么问题？

如果有些任务耗时比较长，导致其他任务排队时间比较长，不能有效的正常执行，直接影响到业务。

看下面代码，2个方法，都使用了@Scheduled(fixedRate = 1000)，表示每秒执行一次，而push1方法中模拟耗时2秒，方法会中打印出线程名称、时间等信息，一会注意观察输出

package com.javacode2018.scheduled.demo2;  
  
import org.springframework.scheduling.annotation.Scheduled;  
import org.springframework.stereotype.Component;  
  
import java.util.concurrent.TimeUnit;  
  
@Component  
public class PushJob {  
  
 //推送方法，每秒执行一次  
 @Scheduled(fixedRate = 1000)  
 public void push1() throws InterruptedException {  
 //休眠2秒，模拟耗时操作  
 TimeUnit.SECONDS.sleep(2);  
 System.out.println(Thread.currentThread().getName() + " push1 模拟推送消息，" + System.currentTimeMillis());  
 }  
  
 //推送方法，每秒执行一次  
 @Scheduled(fixedRate = 1000)  
 public void push2() {  
 System.out.println(Thread.currentThread().getName() + " push2 模拟推送消息，" + System.currentTimeMillis());  
 }  
  
}

运行输出

pool-1-thread-1 push1 模拟推送消息，1595902615507  
pool-1-thread-1 push2 模拟推送消息，1595902615507  
pool-1-thread-1 push1 模拟推送消息，1595902617507  
pool-1-thread-1 push2 模拟推送消息，1595902617507  
pool-1-thread-1 push1 模拟推送消息，1595902619508  
pool-1-thread-1 push2 模拟推送消息，1595902619508

注意上面的输出，线程名称都是pool-1-thread-1，并且有个问题，push2中2次输出时间间隔是2秒，这就是由于线程池中只有一个线程导致了排队执行而产生的问题。

可以通过自定义定时器中的线程池来解决这个问题，定义一个ScheduledExecutorService类型的bean，名称为taskScheduler

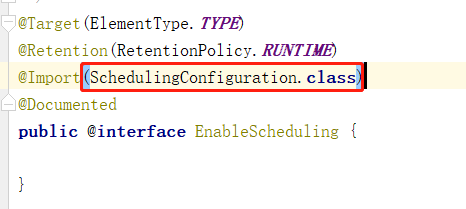
@Bean  
public ScheduledExecutorService taskScheduler() {  
 //设置需要并行执行的任务数量  
 int corePoolSize = 20;  
 return new ScheduledThreadPoolExecutor(corePoolSize);  
}

此时问题就解决了，再次运行一下上面案例代码，结果如下，此时线程名称不一样了，且push2运行正常了

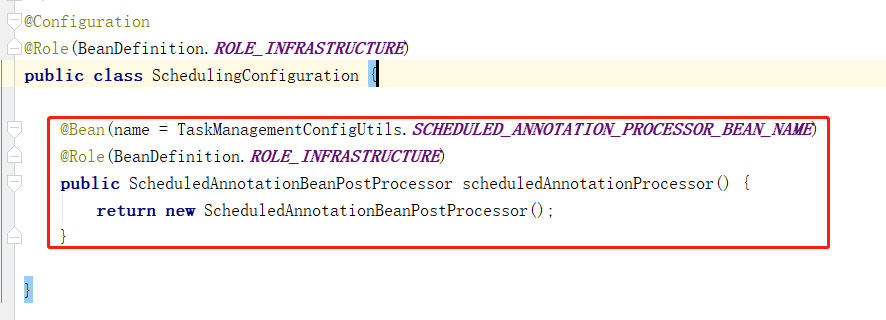
pool-1-thread-2 push2 模拟推送消息，1595903154636  
pool-1-thread-2 push2 模拟推送消息，1595903155636  
pool-1-thread-1 push1 模拟推送消息，1595903156636  
pool-1-thread-3 push2 模拟推送消息，1595903156636  
pool-1-thread-1 push2 模拟推送消息，1595903157636

### 源码 & 原理

从EnableScheduling注解开始看，这个注解会导入SchedulingConfiguration类



SchedulingConfiguration是一个配置类，内部定义了ScheduledAnnotationBeanPostProcessor类型的bean



ScheduledAnnotationBeanPostProcessor是一个bean后置处理器，内部有个postProcessAfterInitialization方法，spring中任何bean在初始化完毕之后，会自动调用postProcessAfterInitialization方法，而ScheduledAnnotationBeanPostProcessor在这个方法中会解析bean中标注有@Scheduled注解的方法，这些方法也就是需要定时执行的方法。

ScheduledAnnotationBeanPostProcessor还实现了一个接口：SmartInitializingSingleton，SmartInitializingSingleton中有个方法afterSingletonsInstantiated会在spring容器中所有单例bean初始化完毕之后调用，定期器的装配及启动都是在这个方法中进行的。

org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor#afterSingletonsInstantiated

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解
37. Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用

## Spring系列第39篇：强大的Spel表达式

**本文带你玩转spring中强大的spel表达式！**

### Spel概述

Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于Struts2x中使用的OGNL表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与Spring功能完美整合，如能用来配置Bean定义。

**表达式语言给静态Java语言增加了动态功能。**

SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可以单独使用。

### Spel能干什么?

表达式语言一般是用最简单的形式完成最主要的工作，减少我们的工作量。

SpEL支持如下表达式：

**一、基本表达式：** 字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及Elivis表达式、正则表达式、括号优先级表达式；

**二、类相关表达式：** 类类型表达式、类实例化、instanceof表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean引用；

**三、集合相关表达式：** 内联List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

**四、其他表达式**： 模板表达式。

**注：SpEL表达式中的关键字是不区分大小写的。**

### SpEL基础

#### HelloWorld

首先准备支持SpEL的Jar包：“org.springframework.expression-3.0.5.RELEASE.jar”将其添加到类路径中。

SpEL在求表达式值时一般分为四步，其中第三步可选：首先构造一个解析器，其次解析器解析字符串表达式，在此构造上下文，最后根据上下文得到表达式运算后的值。

让我们看下代码片段吧：

package com.javacode2018.spel;  
  
import org.junit.Test;  
import org.springframework.expression.EvaluationContext;  
import org.springframework.expression.Expression;  
import org.springframework.expression.ExpressionParser;  
import org.springframework.expression.spel.standard.SpelExpressionParser;  
import org.springframework.expression.spel.support.StandardEvaluationContext;  
  
public class SpelTest {  
 @Test  
 public void test1() {  
 ExpressionParser parser = new SpelExpressionParser();  
 Expression expression = parser.parseExpression("('Hello' + ' World').concat(#end)");  
 EvaluationContext context = new StandardEvaluationContext();  
 context.setVariable("end", "!");  
 System.out.println(expression.getValue(context));  
 }  
}

输出

Hello World!

接下来让我们分析下代码：

1）创建解析器：**SpEL使用ExpressionParser接口表示解析器，提供SpelExpressionParser默认实现；**

2）解析表达式：使用ExpressionParser的parseExpression来解析相应的表达式为Expression对象。

3）构造上下文：准备比如变量定义等等表达式需要的上下文数据。

4）求值：通过Expression接口的getValue方法根据上下文获得表达式值。

是不是很简单，接下来让我们看下其具体实现及原理吧。

#### SpEL原理及接口

SpEL提供简单的接口从而简化用户使用，在介绍原理前让我们学习下几个概念：

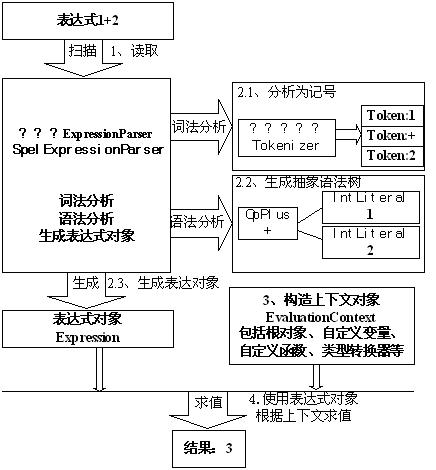
**一、表达式：** 表达式是表达式语言的核心，所以表达式语言都是围绕表达式进行的，从我们角度来看是“干什么”；

**二、解析器：** 用于将字符串表达式解析为表达式对象，从我们角度来看是“谁来干”；

**三、上下文：** 表达式对象执行的环境，该环境可能定义变量、定义自定义函数、提供类型转换等等，从我们角度看是“在哪干”；

**四、根对象及活动上下文对象：** 根对象是默认的活动上下文对象，活动上下文对象表示了当前表达式操作的对象，从我们角度看是“对谁干”。

理解了这些概念后，让我们看下SpEL如何工作的呢，如图5-1所示：



##### 工作原理

1.首先定义表达式：“1+2”；  
2.定义解析器ExpressionParser实现，SpEL提供默认实现SpelExpressionParser；  
 2.1.SpelExpressionParser解析器内部使用Tokenizer类进行词法分析，即把字符串流分析为记号流，记号在SpEL使用Token类来表示；  
 2.2.有了记号流后，解析器便可根据记号流生成内部抽象语法树；在SpEL中语法树节点由SpelNode接口实现代表：如OpPlus表示加操作节点、IntLiteral表示int型字面量节点；使用SpelNodel实现组成了抽象语法树；  
 2.3.对外提供Expression接口来简化表示抽象语法树，从而隐藏内部实现细节，并提供getValue简单方法用于获取表达式值；SpEL提供默认实现为SpelExpression；  
3.定义表达式上下文对象（可选），SpEL使用EvaluationContext接口表示上下文对象，用于设置根对象、自定义变量、自定义函数、类型转换器等，SpEL提供默认实现StandardEvaluationContext；  
4.使用表达式对象根据上下文对象（可选）求值（调用表达式对象的getValue方法）获得结果。

接下来让我们看下SpEL的主要接口吧：

##### ExpressionParser接口

表示解析器，默认实现是org.springframework.expression.spel.standard包中的SpelExpressionParser类，使用parseExpression方法将字符串表达式转换为Expression对象，对于ParserContext接口用于定义字符串表达式是不是模板，及模板开始与结束字符：

public interface ExpressionParser {  
 Expression parseExpression(String expressionString) throws ParseException;  
 Expression parseExpression(String expressionString, ParserContext context) throws ParseException;  
}

来看下示例：

@Test  
public void testParserContext() {  
 ExpressionParser parser = new SpelExpressionParser();  
 ParserContext parserContext = new ParserContext() {  
 @Override  
 public boolean isTemplate() {  
 return true;  
 }  
  
 @Override  
 public String getExpressionPrefix() {  
 return "#{";  
 }  
  
 @Override  
 public String getExpressionSuffix() {  
 return "}";  
 }  
 };  
 String template = "#{'Hello '}#{'World!'}";  
 Expression expression = parser.parseExpression(template, parserContext);  
 System.out.println(expression.getValue());  
}

在此我们演示的是使用ParserContext的情况，此处定义了ParserContext实现：定义表达式是模块，表达式前缀为“#{”，后缀为“}”；使用parseExpression解析时传入的模板必须以“#{”开头，以“}”结尾，如"#{'Hello '}#{'World!'}"。

默认传入的字符串表达式不是模板形式，如之前演示的Hello World。

##### EvaluationContext接口

表示上下文环境，默认实现是org.springframework.expression.spel.support包中的StandardEvaluationContext类，使用setRootObject方法来设置根对象，使用setVariable方法来注册自定义变量，使用registerFunction来注册自定义函数等等。

##### Expression接口

表示表达式对象，默认实现是org.springframework.expression.spel.standard包中的SpelExpression，提供getValue方法用于获取表达式值，提供setValue方法用于设置对象值。

了解了SpEL原理及接口，接下来的事情就是SpEL语法了。

### SpEL语法

#### 基本表达式

##### 字面量表达式

SpEL支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null类型。

|  |  |
| --- | --- |
| 类型 | 示例 |
| 字符串 | String str1 = parser.parseExpression("'Hello World!'").getValue(String.class); |
| 数字类型 | int int1 = parser.parseExpression("1").getValue(Integer.class);<br />long long1 = parser.parseExpression("-1L").getValue(long.class);<br />float float1 = parser.parseExpression("1.1").getValue(Float.class);<br />double double1 = parser.parseExpression("1.1E+2").getValue(double.class);<br />int hex1 = parser.parseExpression("0xa").getValue(Integer.class);<br />long hex2 = parser.parseExpression("0xaL").getValue(long.class); |
| 布尔类型 | boolean true1 = parser.parseExpression("true").getValue(boolean.class);<br />boolean false1 = parser.parseExpression("false").getValue(boolean.class); |
| null类型 | Object null1 = parser.parseExpression("null").getValue(Object.class); |

@Test  
public void test2() {  
 ExpressionParser parser = new SpelExpressionParser();  
  
 String str1 = parser.parseExpression("'Hello World!'").getValue(String.class);  
 int int1 = parser.parseExpression("1").getValue(Integer.class);  
 long long1 = parser.parseExpression("-1L").getValue(long.class);  
 float float1 = parser.parseExpression("1.1").getValue(Float.class);  
 double double1 = parser.parseExpression("1.1E+2").getValue(double.class);  
 int hex1 = parser.parseExpression("0xa").getValue(Integer.class);  
 long hex2 = parser.parseExpression("0xaL").getValue(long.class);  
 boolean true1 = parser.parseExpression("true").getValue(boolean.class);  
 boolean false1 = parser.parseExpression("false").getValue(boolean.class);  
 Object null1 = parser.parseExpression("null").getValue(Object.class);  
  
 System.out.println("str1=" + str1);  
 System.out.println("int1=" + int1);  
 System.out.println("long1=" + long1);  
 System.out.println("float1=" + float1);  
 System.out.println("double1=" + double1);  
 System.out.println("hex1=" + hex1);  
 System.out.println("hex2=" + hex2);  
 System.out.println("true1=" + true1);  
 System.out.println("false1=" + false1);  
 System.out.println("null1=" + null1);  
}

输出

str1=Hello World!  
int1=1  
long1=-1  
float1=1.1  
double1=110.0  
hex1=10  
hex2=10  
true1=true  
false1=false  
null1=null

##### 算数运算表达式

SpEL支持加(+)、减(-)、乘(\*)、除(/)、求余（%）、幂（^）运算。

|  |  |
| --- | --- |
| 类型 | 示例 |
| 加减乘除 | int result1 = parser.parseExpression("1+2-3\*4/2").getValue(Integer.class);//-3 |
| 求余 | int result2 = parser.parseExpression("4%3").getValue(Integer.class);//1 |
| 幂运算 | int result3 = parser.parseExpression("2^3").getValue(Integer.class);//8 |

SpEL还提供求余（MOD）和除（DIV）而外两个运算符，与“%”和“/”等价，不区分大小写。

##### 关系表达式

等于（==）、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算。

如parser.parseExpression("1>2").getValue(boolean.class);将返回false；

而parser.parseExpression("1 between {1, 2}").getValue(boolean.class);将返回true。

between运算符右边操作数必须是列表类型，且只能包含2个元素。第一个元素为开始，第二个元素为结束，区间运算是包含边界值的，即 xxx>=list.get(0) && xxx<=list.get(1)。

SpEL同样提供了等价的“EQ” 、“NE”、 “GT”、“GE”、 “LT” 、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写。

@Test  
public void test3() {  
 ExpressionParser parser = new SpelExpressionParser();  
 boolean v1 = parser.parseExpression("1>2").getValue(boolean.class);  
 boolean between1 = parser.parseExpression("1 between {1,2}").getValue(boolean.class);  
 System.out.println("v1=" + v1);  
 System.out.println("between1=" + between1);  
}

输出

v1=false  
between1=true

##### 逻辑表达式

且（and或者&&）、或(or或者||)、非(!或NOT)。

@Test  
public void test4() {  
 ExpressionParser parser = new SpelExpressionParser();  
  
 boolean result1 = parser.parseExpression("2>1 and (!true or !false)").getValue(boolean.class);  
 boolean result2 = parser.parseExpression("2>1 && (!true || !false)").getValue(boolean.class);  
  
 boolean result3 = parser.parseExpression("2>1 and (NOT true or NOT false)").getValue(boolean.class);  
 boolean result4 = parser.parseExpression("2>1 && (NOT true || NOT false)").getValue(boolean.class);  
  
 System.out.println("result1=" + result1);  
 System.out.println("result2=" + result2);  
 System.out.println("result3=" + result3);  
 System.out.println("result4=" + result4);  
}

输出

result1=true  
result2=true  
result3=true  
result4=false

##### 字符串连接及截取表达式

使用“+”进行字符串连接，使用“'String'[0] [index]”来截取一个字符，目前只支持截取一个，如“'Hello ' + 'World!'”得到“Hello World!”；而“'Hello World!'[0]”将返回“H”。

##### 三目运算

三目运算符 **“表达式1?表达式2:表达式3”**用于构造三目运算表达式，如“2>1?true:false”将返回true；

##### Elivis运算符

Elivis运算符**“表达式1?:表达式2”**从Groovy语言引入用于简化三目运算符的，当表达式1为非null时则返回表达式1，当表达式1为null时则返回表达式2，简化了三目运算符方式“表达式1? 表达式1:表达式2”，如“null?:false”将返回false，而“true?:false”将返回true；

##### 正则表达式

使用“str matches regex，如“'123' matches '\d{3}'”将返回true；

##### 括号优先级表达式

使用“(表达式)”构造，括号里的具有高优先级。

#### 类相关表达式

##### 类类型表达式

使用“T(Type)”来表示java.lang.Class实例，“Type”必须是类全限定名，“java.lang”包除外，即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。

具体使用方法如下：

@Test  
public void testClassTypeExpression() {  
 ExpressionParser parser = new SpelExpressionParser();  
 //java.lang包类访问  
 Class<String> result1 = parser.parseExpression("T(String)").getValue(Class.class);  
 System.out.println(result1);  
  
 //其他包类访问  
 String expression2 = "T(com.javacode2018.spel.SpelTest)";  
 Class<SpelTest> value = parser.parseExpression(expression2).getValue(Class.class);  
 System.out.println(value == SpelTest.class);  
  
 //类静态字段访问  
 int result3 = parser.parseExpression("T(Integer).MAX\_VALUE").getValue(int.class);  
 System.out.println(result3 == Integer.MAX\_VALUE);  
  
 //类静态方法调用  
 int result4 = parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);  
 System.out.println(result4);  
}

输出

class java.lang.String  
true  
true  
1

对于java.lang包里的可以直接使用“T(String)”访问；其他包必须是类全限定名；可以进行静态字段访问如“T(Integer).MAX\_VALUE”；也可以进行静态方法访问如“T(Integer).parseInt('1')”。

##### 类实例化

类实例化同样使用java关键字“new”，类名必须是全限定名，但java.lang包内的类型除外，如String、Integer。

@Test  
public void testConstructorExpression() {  
 ExpressionParser parser = new SpelExpressionParser();  
 String result1 = parser.parseExpression("new String('马士兵老师')").getValue(String.class);  
 System.out.println(result1);  
  
 Date result2 = parser.parseExpression("new java.util.Date()").getValue(Date.class);  
 System.out.println(result2);  
}

实例化完全跟Java内方式一样，运行输出

马士兵老师  
Tue Aug 03 20:22:43 CST 2020

##### instanceof表达式

SpEL支持instanceof运算符，跟Java内使用同义；如“'haha' instanceof T(String)”将返回true。

@Test  
public void testInstanceOfExpression() {  
 ExpressionParser parser = new SpelExpressionParser();  
 Boolean value = parser.parseExpression("'路人甲' instanceof T(String)").getValue(Boolean.class);  
 System.out.println(value);  
}

输出

true

##### 变量定义及引用

变量定义通过EvaluationContext接口的setVariable(variableName, value)方法定义；在表达式中使用"#variableName"引用；除了引用自定义变量，SpE还允许引用根对象及当前上下文对象，使用"#root"引用根对象，使用"#this"引用当前上下文对象；

@Test  
public void testVariableExpression() {  
 ExpressionParser parser = new SpelExpressionParser();  
 EvaluationContext context = new StandardEvaluationContext();  
 context.setVariable("name", "马士兵老师");  
 context.setVariable("lesson", "Spring系列");  
  
 //获取name变量，lesson变量  
 String name = parser.parseExpression("#name").getValue(context, String.class);  
 System.out.println(name);  
 String lesson = parser.parseExpression("#lesson").getValue(context, String.class);  
 System.out.println(lesson);  
  
 //StandardEvaluationContext构造器传入root对象，可以通过#root来访问root对象  
 context = new StandardEvaluationContext("我是root对象");  
 String rootObj = parser.parseExpression("#root").getValue(context, String.class);  
 System.out.println(rootObj);  
  
 //#this用来访问当前上线文中的对象  
 String thisObj = parser.parseExpression("#this").getValue(context, String.class);  
 System.out.println(thisObj);  
}

输出

马士兵老师  
Spring系列  
我是root对象  
我是root对象

使用“#variable”来引用在EvaluationContext定义的变量；除了可以引用自定义变量，还可以使用“#root”引用根对象，“#this”引用当前上下文对象，此处“#this”即根对象。

##### 自定义函数

目前只支持类静态方法注册为自定义函数；SpEL使用StandardEvaluationContext的registerFunction方法进行注册自定义函数，其实完全可以使用setVariable代替，两者其实本质是一样的；

@Test  
public void testFunctionExpression() throws SecurityException, NoSuchMethodException {  
 //定义2个函数,registerFunction和setVariable都可以，不过从语义上面来看用registerFunction更恰当  
 StandardEvaluationContext context = new StandardEvaluationContext();  
 Method parseInt = Integer.class.getDeclaredMethod("parseInt", String.class);  
 context.registerFunction("parseInt1", parseInt);  
 context.setVariable("parseInt2", parseInt);  
  
 ExpressionParser parser = new SpelExpressionParser();  
 System.out.println(parser.parseExpression("#parseInt1('3')").getValue(context, int.class));  
 System.out.println(parser.parseExpression("#parseInt2('3')").getValue(context, int.class));  
   
 String expression1 = "#parseInt1('3') == #parseInt2('3')";  
 boolean result1 = parser.parseExpression(expression1).getValue(context, boolean.class);  
 System.out.println(result1);  
}

此处可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

运行输出

3  
3  
true

##### 表达式赋值

使用Expression#setValue方法可以给表达式赋值

@Test  
public void testAssignExpression1() {  
 Object user = new Object() {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "$classname{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
 };  
 {  
 //user为root对象  
 ExpressionParser parser = new SpelExpressionParser();  
 EvaluationContext context = new StandardEvaluationContext(user);  
 parser.parseExpression("#root.name").setValue(context, "马士兵老师");  
 System.out.println(parser.parseExpression("#root").getValue(context, user.getClass()));  
 }  
 {  
 //user为变量  
 ExpressionParser parser = new SpelExpressionParser();  
 EvaluationContext context = new StandardEvaluationContext();  
 context.setVariable("user", user);  
 parser.parseExpression("#user.name").setValue(context, "马士兵老师");  
 System.out.println(parser.parseExpression("#user").getValue(context, user.getClass()));  
 }  
}

运行输出

$classname{name='马士兵老师'}  
$classname{name='马士兵老师'}

##### 对象属性存取及安全导航表达式

对象属性获取非常简单，即使用如“a.property.property”这种点缀式获取，SpEL对于属性名首字母是不区分大小写的；SpEL还引入了Groovy语言中的安全导航运算符“**(对象|属性)?.属性**”，用来避免“?.”前边的表达式为null时抛出空指针异常，而是返回null；修改对象属性值则可以通过赋值表达式或Expression接口的setValue方法修改。

public static class Car {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Car{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}  
  
public static class User {  
 private Car car;  
  
 public Car getCar() {  
 return car;  
 }  
  
 public void setCar(Car car) {  
 this.car = car;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "car=" + car +  
 '}';  
 }  
}  
  
@Test  
public void test5() {  
 User user = new User();  
 EvaluationContext context = new StandardEvaluationContext();  
 context.setVariable("user", user);  
  
 ExpressionParser parser = new SpelExpressionParser();  
 //使用.符号，访问user.car.name会报错，原因：user.car为空  
 try {  
 System.out.println(parser.parseExpression("#user.car.name").getValue(context, String.class));  
 } catch (EvaluationException | ParseException e) {  
 System.out.println("出错了：" + e.getMessage());  
 }  
 //使用安全访问符号?.，可以规避null错误  
 System.out.println(parser.parseExpression("#user?.car?.name").getValue(context, String.class));  
  
 Car car = new Car();  
 car.setName("保时捷");  
 user.setCar(car);  
  
 System.out.println(parser.parseExpression("#user?.car?.toString()").getValue(context, String.class));  
}

运行输出

出错了：EL1007E: Property or field 'name' cannot be found on null  
null  
Car{name='保时捷'}

##### 对象方法调用

对象方法调用更简单，跟Java语法一样；如“'haha'.substring(2,4)”将返回“ha”；而对于根对象可以直接调用方法；

##### Bean引用

SpEL支持使用“@”符号来引用Bean，在引用Bean时需要使用BeanResolver接口实现来查找Bean，Spring提供BeanFactoryResolver实现。

@Test  
public void test6() {  
 DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
 User user = new User();  
 Car car = new Car();  
 car.setName("保时捷");  
 user.setCar(car);  
 factory.registerSingleton("user", user);  
  
 StandardEvaluationContext context = new StandardEvaluationContext();  
 context.setBeanResolver(new BeanFactoryResolver(factory));  
  
 ExpressionParser parser = new SpelExpressionParser();  
 User userBean = parser.parseExpression("@user").getValue(context, User.class);  
 System.out.println(userBean);  
 System.out.println(userBean == factory.getBean("user"));  
}

运行输出

User{car=Car{name='保时捷'}}  
true

#### 集合相关表达式

##### 内联List

从Spring3.0.4开始支持内联List，使用{表达式，……}定义内联List，如“{1,2,3}”将返回一个整型的ArrayList，而“{}”将返回空的List，对于字面量表达式列表，SpEL会使用java.util.Collections.unmodifiableList方法将列表设置为不可修改。

@Test  
public void test7() {  
 ExpressionParser parser = new SpelExpressionParser();  
 //将返回不可修改的空List  
 List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);  
 //对于字面量列表也将返回不可修改的List  
 List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);  
 Assert.assertEquals(new Integer(1), result1.get(0));  
 try {  
 result1.set(0, 2);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 //对于列表中只要有一个不是字面量表达式，将只返回原始List，  
 //不会进行不可修改处理  
 String expression3 = "{{1+2,2+4},{3,4+4}}";  
 List<List<Integer>> result3 = parser.parseExpression(expression3).getValue(List.class);  
 result3.get(0).set(0, 1);  
 System.out.println(result3);  
 //声明二维数组并初始化  
 int[] result4 = parser.parseExpression("new int[2]{1,2}").getValue(int[].class);  
 System.out.println(result4[1]);  
 //定义一维数组并初始化  
 int[] result5 = parser.parseExpression("new int[1]").getValue(int[].class);  
 System.out.println(result5[0]);  
}

输出

java.lang.UnsupportedOperationException  
 at java.util.Collections$UnmodifiableList.set(Collections.java:1311)  
 at com.javacode2018.spel.SpelTest.test7(SpelTest.java:315)  
[[1, 6], [3, 8]]  
2  
0

##### 内联数组

和Java 数组定义类似，只是在定义时进行多维数组初始化。

int[][][] result4 = parser.parseExpression("new int[1][2][3]{{1}{2}{3}}").getValue(int[][][].class);

##### 集合，字典元素访问

SpEL目前支持所有集合类型和字典类型的元素访问，使用“集合[索引]”访问集合元素，使用“map[key]”访问字典元素；

//SpEL内联List访问   
int result1 = parser.parseExpression("{1,2,3}[0]").getValue(int.class);   
  
//SpEL目前支持所有集合类型的访问   
Collection<Integer> collection = new HashSet<Integer>();   
collection.add(1);   
collection.add(2);   
  
EvaluationContext context2 = new StandardEvaluationContext();   
context2.setVariable("collection", collection);   
int result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);   
  
  
//SpEL对Map字典元素访问的支持   
Map<String, Integer> map = new HashMap<String, Integer>();   
map.put("a", 1);   
  
EvaluationContext context3 = new StandardEvaluationContext();   
context3.setVariable("map", map);   
int result3 = parser.parseExpression("#map['a']").getValue(context3, int.class);

##### 列表，字典，数组元素修改

可以使用赋值表达式或Expression接口的setValue方法修改；

@Test  
public void test8() {  
 ExpressionParser parser = new SpelExpressionParser();  
  
 //修改list元素值  
 List<Integer> list = new ArrayList<Integer>();  
 list.add(1);  
 list.add(2);  
  
 EvaluationContext context1 = new StandardEvaluationContext();  
 context1.setVariable("collection", list);  
 parser.parseExpression("#collection[1]").setValue(context1, 4);  
 int result1 = parser.parseExpression("#collection[1]").getValue(context1, int.class);  
 System.out.println(result1);  
  
 //修改map元素值  
 Map<String, Integer> map = new HashMap<String, Integer>();  
 map.put("a", 1);  
 EvaluationContext context2 = new StandardEvaluationContext();  
 context2.setVariable("map", map);  
 parser.parseExpression("#map['a']").setValue(context2, 4);  
 Integer result2 = parser.parseExpression("#map['a']").getValue(context2, int.class);  
 System.out.println(result2);  
}

输出

4  
4

##### 集合投影

在SQL中投影指从表中选择出列，而在SpEL指根据集合中的元素中通过选择来构造另一个集合，该集合和原集合具有相同数量的元素；SpEL使用“（list|map）.![投影表达式]”来进行投影运算：

@Test  
public void test9() {  
 ExpressionParser parser = new SpelExpressionParser();  
  
 //1.测试集合或数组  
 List<Integer> list = new ArrayList<Integer>();  
 list.add(4);  
 list.add(5);  
 EvaluationContext context1 = new StandardEvaluationContext();  
 context1.setVariable("list", list);  
 Collection<Integer> result1 = parser.parseExpression("#list.![#this+1]").getValue(context1, Collection.class);  
 result1.forEach(System.out::println);  
  
 System.out.println("------------");  
 //2.测试字典  
 Map<String, Integer> map = new HashMap<String, Integer>();  
 map.put("a", 1);  
 map.put("b", 2);  
 EvaluationContext context2 = new StandardEvaluationContext();  
 context2.setVariable("map", map);  
 List<Integer> result2 = parser.parseExpression("#map.![value+1]").getValue(context2, List.class);  
 result2.forEach(System.out::println);  
}

对于集合或数组使用如上表达式进行投影运算，其中投影表达式中“#this”代表每个集合或数组元素，可以使用比如“#this.property”来获取集合元素的属性，其中“#this”可以省略。

Map投影最终只能得到List结果，如上所示，对于投影表达式中的“#this”将是Map.Entry，所以可以使用“value”来获取值，使用“key”来获取键。

##### 集合选择

在SQL中指使用select进行选择行数据，而在SpEL指根据原集合通过条件表达式选择出满足条件的元素并构造为新的集合，SpEL使用“(list|map).?[选择表达式]”，其中选择表达式结果必须是boolean类型，如果true则选择的元素将添加到新集合中，false将不添加到新集合中。

@Test  
public void test10() {  
 ExpressionParser parser = new SpelExpressionParser();  
  
 //1.测试集合或数组  
 List<Integer> list = new ArrayList<Integer>();  
 list.add(1);  
 list.add(4);  
 list.add(5);  
 list.add(7);  
 EvaluationContext context1 = new StandardEvaluationContext();  
 context1.setVariable("list", list);  
 Collection<Integer> result1 = parser.parseExpression("#list.?[#this>4]").getValue(context1, Collection.class);  
 result1.forEach(System.out::println);  
  
 System.out.println("------------");  
   
}

输出

5  
7

对于集合或数组选择，如“#collection.?[#this>4]”将选择出集合元素值大于4的所有元素。选择表达式必须返回布尔类型，使用“#this”表示当前元素。

//2.测试字典  
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("a", 1);  
map.put("b", 2);  
map.put("c", 3);  
EvaluationContext context2 = new StandardEvaluationContext();  
context2.setVariable("map", map);  
Map<String, Integer> result2 = parser.parseExpression("#map.?[key!='a']").getValue(context2, Map.class);  
result2.forEach((key, value) -> {  
 System.out.println(key + ":" + value);  
});  
System.out.println("------------");  
List<Integer> result3 = parser.parseExpression("#map.?[key!='a'].![value+1]").getValue(context2, List.class);  
result3.forEach(System.out::println);

输出

b:2  
c:3  
------------  
3  
4

对于字典选择，如“#map.?[#this.key != 'a']”将选择键值不等于”a”的，其中选择表达式中“#this”是Map.Entry类型，而最终结果还是Map，这点和投影不同；集合选择和投影可以一起使用，如“#map.?[key != 'a'].![value+1]”将首先选择键值不等于”a”的，然后在选出的Map中再进行“value+1”的投影。

#### 表达式模板

模板表达式就是由字面量与一个或多个表达式块组成。每个表达式块由“前缀+表达式+后缀”形式组成，如“${1+2}”即表达式块。在前边我们已经介绍了使用ParserContext接口实现来定义表达式是否是模板及前缀和后缀定义。在此就不多介绍了，如“Error ${#v0} ${#v1}”表达式表示由字面量“Error ”、模板表达式“#v0”、模板表达式“#v1”组成，其中v0和v1表示自定义变量，需要在上下文定义。

解析表达式的时候需要指定模板，模板通过ParserContext接口来定义

public interface ParserContext {  
 //是否是模板  
 boolean isTemplate();  
 //模板表达式前缀  
 String getExpressionPrefix();  
 //模板表达式后缀  
 String getExpressionSuffix();  
}

有个子类，我们直接可以拿来用：TemplateParserContext。

@Test  
public void test11() {  
 //创建解析器  
 SpelExpressionParser parser = new SpelExpressionParser();  
 //创建解析器上下文  
 ParserContext context = new TemplateParserContext("%{", "}");  
 Expression expression = parser.parseExpression("你好:%{#name},我们正在学习:%{#lesson}", context);  
  
 //创建表达式计算上下文  
 EvaluationContext evaluationContext = new StandardEvaluationContext();  
 evaluationContext.setVariable("name", "马士兵老师");  
 evaluationContext.setVariable("lesson", "spring高手系列!");  
 //获取值  
 String value = expression.getValue(evaluationContext, String.class);  
 System.out.println(value);  
}

运行输出

你好:马士兵老师,我们正在学习:spring高手系列!

### 在Bean定义中使用spel表达式

#### xml风格的配置

SpEL支持在Bean定义时注入，默认使用“#{SpEL表达式}”表示，其中“#root”根对象默认可以认为是ApplicationContext，只有ApplicationContext实现默认支持SpEL，获取根对象属性其实是获取容器中的Bean。

如：

<bean id="world" class="java.lang.String">   
 <constructor-arg value="#{' World!'}"/>   
</bean>   
  
<bean id="hello1" class="java.lang.String">   
 <constructor-arg value="#{'Hello'}#{world}"/>   
</bean>   
  
<bean id="hello2" class="java.lang.String">   
 <constructor-arg value="#{'Hello' + world}"/>  
</bean>   
  
<bean id="hello3" class="java.lang.String">   
 <constructor-arg value="#{'Hello' + @world}"/>   
</bean>

模板默认以前缀“#{”开头，以后缀“}”结尾，且不允许嵌套，如“#{'Hello'#{world}}”错误，如“#{'Hello' + world}”中“world”默认解析为Bean。当然可以使用“@bean”引用了。

是不是很简单，除了XML配置方式，Spring还提供一种注解方式@Value，接着往下看吧。

#### 注解风格的配置

基于注解风格的SpEL配置也非常简单，使用@Value注解来指定SpEL表达式，该注解可以放到字段、方法及方法参数上。

测试Bean类如下，使用@Value来指定SpEL表达式：

public class SpELBean {   
 @Value("#{'Hello' + world}")   
 private String value;   
}

#### 在Bean定义中SpEL的问题

如果有同学问“#{我不是SpEL表达式}”不是SpEL表达式，而是公司内部的模板，想换个前缀和后缀该如何实现呢？

我们使用BeanFactoryPostProcessor接口提供postProcessBeanFactory回调方法，它是在IoC容器创建好但还未进行任何Bean初始化时被ApplicationContext实现调用，因此在这个阶段把SpEL前缀及后缀修改掉是安全的，具体代码如下：

package com.javacode2018.spel.test1;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanExpressionResolver;  
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.context.expression.StandardBeanExpressionResolver;  
import org.springframework.stereotype.Component;  
  
@Component  
public class SpelBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
 BeanExpressionResolver beanExpressionResolver = beanFactory.getBeanExpressionResolver();  
 if (beanExpressionResolver instanceof StandardBeanExpressionResolver) {  
 StandardBeanExpressionResolver resolver = (StandardBeanExpressionResolver) beanExpressionResolver;  
 resolver.setExpressionPrefix("%{");  
 resolver.setExpressionSuffix("}");  
 }  
 }  
}

上测试代码

package com.javacode2018.spel.test1;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
@Component  
public class LessonModel {  
 @Value("你好,%{@name},%{@msg}")  
 private String desc;  
  
 @Override  
 public String toString() {  
 return "LessonModel{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
}

@name：容器中name的bean

@msg：容器中msg的bean

下面我们来个配置类，顺便定义name和msg这2个bean，顺便扫描上面2个配置类

package com.javacode2018.spel.test1;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@ComponentScan  
@Configuration  
public class MainConfig {  
 @Bean  
 public String name() {  
 return "路粉";  
 }  
  
 @Bean  
 public String msg() {  
 return "欢迎和我一起学习java各种技术！";  
 }  
}

测试用例

@Test  
public void test12() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig.class);  
 context.refresh();  
 LessonModel lessonModel = context.getBean(LessonModel.class);  
 System.out.println(lessonModel);  
}

运行输出

LessonModel{desc='你好,路粉,欢迎和我一起学习java各种技术！'}

### 总结

1. Spel功能还是比较强大的，可以脱离spring环境独立运行
2. spel可以用在一些动态规则的匹配方面，比如监控系统中监控规则的动态匹配；其他的一些条件动态判断等等
3. 本文内容比较长，建议大家把案例都敲一遍，可以设置一些断点去研究一下源码，有问题的，欢迎大家留言交流。

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解
37. Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用
38. Spring系列第38篇：@Scheduled & @EnableScheduling定时器详解

## Spring系列第40篇：缓存使用（@EnableCaching、@Cacheable、@CachePut、@CacheEvict、@Caching、@CacheConfig）

**本文主要详解spring中缓存的使用。**

### 背景

缓存大家都有了解过吧，主要用来提升系统查询速度。

比如电商中商品详情信息，这些信息通常不会经常变动但是会高频访问，我们可以将这些信息从db中拿出来放在缓存中（比如redis中、本地内存中），当获取的时候，先从缓存中获取，缓存中没有的时候，再从db中获取，然后将其再丢到缓存中，当商品信息被变更之后，可以将缓存中的信息剔除或者将最新的数据丢到缓存中。

Spring中提供了一整套的缓存解决方案，使用起来特别的容易，主要通过注解的方式使用缓存，常用的有5个注解，我们一个个来介绍。

本文中会大量用到spel表达式，对这块不熟悉的建议先看一下：[Spring中Spel详解](https://mp.weixin.qq.com/s/U2fT4Pxpg1oyDm0o3sUTmQ)

### @EnableCaching：启用缓存功能

开启缓存功能，配置类中需要加上这个注解，有了这个注解之后，spring才知道你需要使用缓存的功能，其他和缓存相关的注解才会有效，spring中主要是通过aop实现的，通过aop来拦截需要使用缓存的方法，实现缓存的功能。

### @Cacheable：赋予缓存功能

#### 作用

@Cacheable可以标记在一个方法上，也可以标记在一个类上。当标记在一个方法上时表示该方法是支持缓存的，当标记在一个类上时则表示该类所有的方法都是支持缓存的。对于一个支持缓存的方法，Spring会在其被调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果，而不需要再次执行该方法。Spring在缓存方法的返回值时是以键值对进行缓存的，值就是方法的返回结果，至于键的话，Spring又支持两种策略，默认策略和自定义策略，这个稍后会进行说明。需要注意的是当一个支持缓存的方法在对象内部被调用时是不会触发缓存功能的。@Cacheable可以指定三个属性，value、key和condition。

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface Cacheable {  
 String[] value() default {};  
 String[] cacheNames() default {};  
 String key() default "";  
 String condition() default "";  
 String unless() default "";  
}

#### value属性：指定Cache名称

value和cacheNames属性作用一样，必须指定其中一个，表示当前方法的返回值是会被缓存在哪个Cache上的，对应Cache的名称。其可以是一个Cache也可以是多个Cache，当需要指定多个Cache时其是一个数组。

可以将Cache想象为一个HashMap，系统中可以有很多个Cache，每个Cache有一个名字，你需要将方法的返回值放在哪个缓存中，需要通过缓存的名称来指定。

##### 案例1

下面list方法加上了缓存的功能，将其结果放在缓存cache1中。

package com.javacode2018.cache.demo1;  
  
import org.springframework.cache.annotation.Cacheable;  
import org.springframework.stereotype.Component;  
  
import java.util.Arrays;  
import java.util.List;  
  
@Component  
public class ArticleService {  
  
 @Cacheable(cacheNames = {"cache1"})  
 public List<String> list() {  
 System.out.println("获取文章列表!");  
 return Arrays.asList("spring", "mysql", "java高并发", "maven");  
 }  
}

下面来个配置类MainConfig1，必须加上@EnableCaching注解用来启用缓存功能。

然后在配置类中需要定义一个bean：缓存管理器，类型为CacheManager，CacheManager这个是个接口，有好几个实现（比如使用redis、ConcurrentMap来存储缓存信息），此处我们使用ConcurrentMapCacheManager，内部使用ConcurrentHashMap将缓存信息直接存储在本地jvm内存中，不过线上环境一般是集群的方式，可以通过redis实现，下一篇文章介绍spring缓存集成redis。

package com.javacode2018.cache.demo1;  
  
import org.springframework.cache.CacheManager;  
import org.springframework.cache.annotation.EnableCaching;  
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@EnableCaching //@0  
@ComponentScan  
@Configuration  
public class MainConfig1 {  
  
 //@1：缓存管理器  
 @Bean  
 public CacheManager cacheManager() {  
 //创建缓存管理器(ConcurrentMapCacheManager：其内部使用ConcurrentMap实现的)，构造器用来指定缓存的名称，可以指定多个  
 ConcurrentMapCacheManager cacheManager = new ConcurrentMapCacheManager("cache1");  
 return cacheManager;  
 }  
  
}

来个测试类，2次调用list方法看看效果

package com.javacode2018.cache;  
  
import com.javacode2018.cache.demo1.ArticleService;  
import com.javacode2018.cache.demo1.MainConfig1;  
import org.junit.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class CacheTest {  
  
 @Test  
 public void test1() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
 System.out.println(articleService.list());  
 System.out.println(articleService.list());  
 }  
  
}

输出

获取文章列表!  
[spring, mysql, java高并发, maven]  
[spring, mysql, java高并发, maven]

从第一行可以看出，第一次进入到list方法内部了，第二次没有进入list方法内部，而是从缓存中获取的。

#### key属性：自定义key

key属性用来指定Spring缓存方法的返回结果时对应的key的，上面说了你可以将Cache理解为一个hashMap，缓存以key->value的形式存储在hashmap中，value就是需要缓存值（即方法的返回值）

key属性支持SpEL表达式；当我们没有指定该属性时，Spring将使用默认策略生成key（org.springframework.cache.interceptor.SimpleKeyGenerator），默认会方法参数创建key。

自定义策略是指我们可以通过SpEL表达式来指定我们的key，这里的SpEL表达式可以使用方法参数及它们对应的属性，使用方法参数时我们可以直接使用“#参数名”或者“#p参数index”。

Spring还为我们提供了一个root对象可以用来生成key，通过该root对象我们可以获取到以下信息。

|  |  |  |
| --- | --- | --- |
| 属性名称 | 描述 | 示例 |
| methodName | 当前方法名 | #root.methodName |
| method | 当前方法 | #root.method.name |
| target | 当前被调用的对象 | #root.target |
| targetClass | 当前被调用的对象的class | #root.targetClass |
| args | 当前方法参数组成的数组 | #root.args[0] |
| caches | 当前被调用的方法使用的Cache | #root.caches[0].name |

这里我们主要看一下自定义策略。

##### 案例2

ArticleService加入下面代码

@Cacheable(cacheNames = {"cache1"}, key = "#root.target.class.name+'-'+#page+'-'+#pageSize")  
public String getPage(int page, int pageSize) {  
 String msg = String.format("page-%s-pageSize-%s", page, pageSize);  
 System.out.println("从db中获取数据：" + msg);  
 return msg;  
}

com.javacode2018.cache.CacheTest新增测试用例

@Test  
public void test2() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
   
 //page=1,pageSize=10调用2次  
 System.out.println(articleService.getPage(1, 10));  
 System.out.println(articleService.getPage(1, 10));  
   
 //page=2,pageSize=10调用2次  
 System.out.println(articleService.getPage(2, 10));  
 System.out.println(articleService.getPage(2, 10));  
  
 {  
 System.out.println("下面打印出cache1缓存中的key列表");  
 ConcurrentMapCacheManager cacheManager = context.getBean(ConcurrentMapCacheManager.class);  
 ConcurrentMapCache cache1 = (ConcurrentMapCache) cacheManager.getCache("cache1");  
 cache1.getNativeCache().keySet().stream().forEach(System.out::println);  
 }  
}

运行输出

从db中获取数据：page-1-pageSize-10  
page-1-pageSize-10  
page-1-pageSize-10  
从db中获取数据：page-2-pageSize-10  
page-2-pageSize-10  
page-2-pageSize-10  
下面打印出cache1缓存中的key列表  
com.javacode2018.cache.demo1.ArticleService-getPage-1-10  
com.javacode2018.cache.demo1.ArticleService-getPage-2-10

#### condition属性：控制缓存的使用条件

有时候，可能我们希望查询不走缓存，同时返回的结果也不要被缓存，那么就可以通过condition属性来实现，condition属性默认为空，表示将缓存所有的调用情形，其值是通过spel表达式来指定的，当为true时表示先尝试从缓存中获取；若缓存中不存在，则只需方法，并将方法返回值丢到缓存中；当为false的时候，不走缓存、直接执行方法、并且返回结果也不会丢到缓存中。

其值spel的写法和key属性类似。

##### 案例3

ArticleService添加下面代码，方法的第二个参数cache用来控制是否走缓存，将condition的值指定为#cache

/\*\*  
 \* 通过文章id获取文章  
 \*  
 \* @param id 文章id  
 \* @param cache 是否尝试从缓存中获取  
 \* @return  
 \*/  
@Cacheable(cacheNames = "cache1", key = "'getById'+#id", condition = "#cache")  
public String getById(Long id, boolean cache) {  
 System.out.println("获取数据!");  
 return "spring缓存:" + UUID.randomUUID().toString();  
}

来个测试用例，4次调用getById方法，前面2次和最后一次cache参数都是true，第3次为false

@Test  
public void test3() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
  
 System.out.println(articleService.getById(1L, true));  
 System.out.println(articleService.getById(1L, true));  
 System.out.println(articleService.getById(1L, false));  
 System.out.println(articleService.getById(1L, true));  
}

运行输出

获取数据!  
spring缓存:27e7c11a-26ed-4c8b-8444-78257daafed5  
spring缓存:27e7c11a-26ed-4c8b-8444-78257daafed5  
获取数据!  
spring缓存:05ff7612-29cb-4863-b8bf-d1b7c2c192b7  
spring缓存:27e7c11a-26ed-4c8b-8444-78257daafed5

从输出中可以看出，第1次和第3次，都进到方法里面去了，而2和4走了缓存，第一次执行完毕之后结果被丢到了缓存中，所以2和4这2次获取的结果和第1次是一样的。

#### unless属性：控制是否需要将结果丢到缓存中

用于否决方法缓存的SpEL表达式。 与condition不同，此表达式是在调用方法后计算的，因此可以引用结果。 默认值为“”，这意味着缓存永远不会被否决。

**前提是condition为空或者为true的情况下，unless才有效，condition为false的时候，unless无效，unless为true，方法返回结果不会丢到缓存中；unless为false，方法返回结果会丢到缓存中。**

其值spel的写法和key属性类似。

##### 案例4

下面来个案例，当返回结果为null的时候，不要将结果进行缓存，ArticleService添加下面代码

Map<Long, String> articleMap = new HashMap<>();  
/\*\*  
 \* 获取文章，先从缓存中获取，如果获取的结果为空，不要将结果放在缓存中  
 \*  
 \* @param id  
 \* @return  
 \*/  
@Cacheable(cacheNames = "cache1", key = "'findById'+#id", unless = "#result==null")  
public String findById(Long id) {  
 this.articleMap.put(1L, "spring系列");  
 System.out.println("----获取文章:" + id);  
 return articleMap.get(id);  
}

来个测试用例，4次调用findById，前面2次有数据，后面2次返回null，并将缓存中的key打印了出来

@Test  
public void test4() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
  
 System.out.println(articleService.findById(1L));  
 System.out.println(articleService.findById(1L));  
 System.out.println(articleService.findById(3L));  
 System.out.println(articleService.findById(3L));  
  
 {  
 System.out.println("下面打印出缓cache1缓存中的key列表");  
 ConcurrentMapCacheManager cacheManager = context.getBean(ConcurrentMapCacheManager.class);  
 ConcurrentMapCache cache1 = (ConcurrentMapCache) cacheManager.getCache("cache1");  
 cache1.getNativeCache().keySet().stream().forEach(System.out::println);  
 }  
}

运行输出

----获取文章:1  
spring系列  
spring系列  
----获取文章:3  
null  
----获取文章:3  
null  
下面打印出缓cache1缓存中的key列表  
findById1

可以看出文章id为1的结果被缓存了，文件id为3的没有被缓存。

#### condition和unless对比

缓存的使用过程中有2个点：

1. 查询缓存中是否有数据
2. 如果缓存中没有数据，则去执行目标方法，然后将方法结果丢到缓存中。

spring中通过condition和unless对这2点进行干预。

condition作用域上面2个过程，当为true的时候，会尝试从缓存中获取数据，如果没有，会执行方法，然后将方法返回值丢到缓存中；如果为false，则直接调用目标方法，并且结果不会放在缓存中。

而unless在condition为true的情况下才有效，用来判断上面第2点中，是否不要将结果丢到缓存中，如果为true，则结果不会丢到缓存中，如果为false，则结果会丢到缓存中，并且unless中可以使用spel表达式通过#result来获取方法返回值。

### @CachePut：将结果放入缓存

#### 作用

@CachePut也可以标注在类或者方法上，被标注的方法每次都会被调用，然后方法执行完毕之后，会将方法结果丢到缓存中；当标注在类上，相当于在类的所有方法上标注了@CachePut。

有3种情况，结果不会丢到缓存

1. 当方法向外抛出的时候
2. condition的计算结果为false的时候
3. unless的计算结果为true的时候

源码和Cacheable类似，包含的参数类似的。

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface CachePut {  
 String[] value() default {};  
 String[] cacheNames() default {};  
 String key() default "";  
 String condition() default "";  
 String unless() default "";  
}

* value和cacheNames：用来指定缓存名称，可以指定多个
* key：缓存的key，spel表达式，写法参考@Cacheable中的key
* condition：spel表达式，写法和@Cacheable中的condition一样，当为空或者计算结果为true的时候，方法的返回值才会丢到缓存中；否则结果不会丢到缓存中
* unless：当condition为空或者计算结果为true的时候，unless才会起效；true：结果不会被丢到缓存，false：结果会被丢到缓存。

#### 案例5

来个案例，实现新增文章的操作，然后将文章丢到缓存中，注意下面@CachePut中的cacheNames、key 2个参数和案例4中findById方法上@Cacheable中的一样，说明他们共用一个缓存，key也是一样的，那么当add方法执行完毕之后，再去调用findById方法，则可以从缓存中直接获取到数据。

@CachePut(cacheNames = "cache1", key = "'findById'+#id")  
public String add(Long id, String content) {  
 System.out.println("新增文章:" + id);  
 this.articleMap.put(id, content);  
 return content;  
}

测试用例

@Test  
public void test5() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
  
 //新增3个文章，由于add方法上面有@CachePut注解，所以新增之后会自动丢到缓存中  
 articleService.add(1L, "java高并发系列");  
 articleService.add(2L, "Maven高手系列");  
 articleService.add(3L, "MySQL高手系列");  
  
 //然后调用findById获取，看看是否会走缓存  
 System.out.println("调用findById方法，会尝试从缓存中获取");  
 System.out.println(articleService.findById(1L));  
 System.out.println(articleService.findById(2L));  
 System.out.println(articleService.findById(3L));  
  
 {  
 System.out.println("下面打印出cache1缓存中的key列表");  
 ConcurrentMapCacheManager cacheManager = context.getBean(ConcurrentMapCacheManager.class);  
 ConcurrentMapCache cache1 = (ConcurrentMapCache) cacheManager.getCache("cache1");  
 cache1.getNativeCache().keySet().stream().forEach(System.out::println);  
 }  
}

运行输出

新增文章:1  
新增文章:2  
新增文章:3  
调用findById方法，会尝试从缓存中获取  
java高并发系列  
Maven高手系列  
MySQL高手系列  
下面打印出缓cache1缓存中的key列表  
findById3  
findById2  
findById1

看几眼输出结果，然后再来看一下findById方法的代码

@Cacheable(cacheNames = "cache1", key = "'findById'+#id", unless = "#result==null")  
public String findById(Long id) {  
 this.articleMap.put(1L, "spring系列");  
 System.out.println("----获取文章:" + id);  
 return articleMap.get(id);  
}

输出中并没有----这样的内容，说明调用findById方法获取结果是从缓存中得到的。

### @CacheEvict：缓存清理

#### 作用

用来清除缓存的，@CacheEvict也可以标注在类或者方法上，被标注在方法上，则目标方法被调用的时候，会清除指定的缓存；当标注在类上，相当于在类的所有方法上标注了@CacheEvict。

来看一下源码，每个参数的注释大家详细看一下。

public @interface CacheEvict {  
  
 /\*\*  
 \* cache的名称，和cacheNames效果一样  
 \*/  
 String[] value() default {};  
  
 /\*\*  
 \* cache的名称，和cacheNames效果一样  
 \*/  
 String[] cacheNames() default {};  
  
 /\*\*  
 \* 缓存的key，写法参考上面@Cacheable注解的key  
 \*/  
 String key() default "";  
  
 /\*\*  
 \* @CacheEvict 注解生效的条件，值为spel表达式，写法参考上面 @Cacheable注解中的condition  
 \*/  
 String condition() default "";  
  
 /\*\*  
 \* 是否清理 cacheNames 指定的缓存中的所有缓存信息，默认是false  
 \* 可以将一个cache想象为一个HashMap，当 allEntries 为true的时候，相当于HashMap.clear()  
 \* 当 allEntries 为false的时候，只会干掉key对应的数据，相当于HashMap.remove(key)  
 \*/  
 boolean allEntries() default false;  
  
 /\*\*  
 \* 何事执行清除操作（方法执行前 or 方法执行成功之后）  
 \* true：@CacheEvict 标注的方法执行之前，执行清除操作  
 \* false：@CacheEvict 标注的方法执行成功之后，执行清除操作，当方法弹出异常的时候，不会执行清除操作  
 \*/  
 boolean beforeInvocation() default false;  
}

#### condition属性

@CacheEvict 注解生效的条件，值为spel表达式，写法参考上面 @Cacheable注解中的condition

#### 会清除哪些缓存？

默认情况下会清除cacheNames指定的缓存中key参数指定的缓存信息。

但是当 allEntries 为true的时候，会清除 cacheNames 指定的缓存中的所有缓存信息。

#### 具体什么时候清除缓存？

这个是通过 beforeInvocation 参数控制的，这个参数默认是false，默认会在目标方法成功执行之后执行清除操作，若方法向外抛出了异常，不会执行清理操作；

如果 beforeInvocation 为true，则方法被执行之前就会执行缓存清理操作，方法执行之后不会再执行了。

#### 案例6

ArticleService中新增个方法，使用@CacheEvict标注，这个方法执行完毕之后，会清理cache1中key=findById+参数id的缓存信息，注意cacheNames和key两个参数的值和findById中这2个参数的值一样。

@CacheEvict(cacheNames = "cache1", key = "'findById'+#id") //@1  
public void delete(Long id) {  
 System.out.println("删除文章：" + id);  
 this.articleMap.remove(id);  
}

新增测试用例，注释比较清晰，就不解释了

@Test  
public void test6() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
 context.register(MainConfig1.class);  
 context.refresh();  
 ArticleService articleService = context.getBean(ArticleService.class);  
  
 //第1次调用findById，缓存中没有，则调用方法，将结果丢到缓存中  
 System.out.println(articleService.findById(1L));  
 //第2次调用findById，缓存中存在，直接从缓存中获取  
 System.out.println(articleService.findById(1L));  
  
 //执行删除操作，delete方法上面有@CacheEvict方法，会清除缓存  
 articleService.delete(1L);  
  
 //再次调用findById方法，发现缓存中没有了，则会调用目标方法  
 System.out.println(articleService.findById(1L));  
}

运行输出

----获取文章:1  
spring系列  
spring系列  
删除文章：1  
----获取文章:1  
spring系列

调用了3次findById，第1次，缓存中没有，所以进到方法内部了，然后将结果丢到缓存了，第2次缓存中有，所以从缓存获取，然后执行了delete方法，这个方法执行完毕之后，会清除缓存中文章id为1L的文章信息，最后执行第三次findById方法，此时缓存中没有发现数据，然后进到目标方法内部了，目标方法内部输出了----内容。

### @Caching：缓存注解组

当我们在类上或者同一个方法上同时使用@Cacheable、@CachePut和@CacheEvic这几个注解中的多个的时候，此时可以使用@Caching这个注解来实现。

@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface Caching {  
  
 Cacheable[] cacheable() default {};  
  
 CachePut[] put() default {};  
  
 CacheEvict[] evict() default {};  
  
}

### @CacheConfig：提取公共配置

这个注解标注在类上，可以将其他几个缓存注解（@Cacheable、@CachePut和@CacheEvic）的公共参数给提取出来放在@CacheConfig中。

比如当一个类中有很多方法都需要使用（@Cacheable、@CachePut和@CacheEvic）这些缓存注解的时候，大家可以看一下这3个注解的源码，他们有很多公共的属性，比如：cacheNames、keyGenerator、cacheManager、cacheResolver，若这些属性值都是一样的，可以将其提取出来，放在@CacheConfig中，不过这些注解（@Cacheable、@CachePut和@CacheEvic）中也可以指定属性的值对@CacheConfig中的属性值进行覆盖。

@CacheConfig(cacheNames = "cache1")  
public class ArticleService {  
 @Cacheable(key = "'findById'+#id")  
 public String findById(Long id) {  
 this.articleMap.put(1L, "spring系列");  
 System.out.println("----获取文章:" + id);  
 return articleMap.get(id);  
 }  
}

### 原理

spring中的缓存主要是利用spring中aop实现的，通过Aop对需要使用缓存的bean创建代理对象，通过代理对象拦截目标方法的执行，实现缓存功能。

重点在于@EnableCaching这个注解，可以从@Import这个注解看起

@Import(CachingConfigurationSelector.class)  
public @interface EnableCaching {  
}

最终会给需要使用缓存的bean创建代理对象，并且会在代理中添加一个拦截器org.springframework.cache.interceptor.CacheInterceptor，这个类中的invoke方法是关键，会拦截所有缓存相关的目标方法的执行，大家可以去细看一下。

### 总结

**Spring系列到此已经40篇了，能和我一起坚持到现在的，真的不容易。**

**还没有看完的朋友，建议大家按序都看一遍，文章最好按顺序看，前后知识点是有依赖的。**

**如果前面的文章都看过了，那么本文的原理，不用我介绍了，大家自己很容易就搞懂了。**

### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解
37. Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用
38. Spring系列第38篇：@Scheduled & @EnableScheduling定时器详解
39. Spring系列第39篇：强大的Spel表达式

## Spring系列第41篇：@EnableCaching集成redis缓存

上篇文章主要介绍了[spring中缓存的使用](https://mp.weixin.qq.com/s/MHh3a1P89XP5iCYCOXi4oQ)，不过文中的案例都是以本地内存作为存储介质的，但是实际上我们的项目上线之后，基本上都会采用集群的方式进行部署，如果将数据存储在本地内存中，集群之间是无法共享的，我们可以将数据存储在redis中，从而实现缓存的共享，下面我们一起来看下**Spring中@EnableCaching如何对接redis**。

### 安装redis

下载地址：https://redis.io/download

### pom.xml中引入redis配置

<dependency>  
 <groupId>org.redisson</groupId>  
 <artifactId>redisson</artifactId>  
 <version>3.13.3</version>  
</dependency>

### 项目中创建redis配置文件

新建com/javacode2018/cache/demo2/redis.yml，内容如下：

singleServerConfig:  
 address: "redis://127.0.0.1:6379"  
 password: null  
 clientName: null  
 database: 7 #选择使用哪个数据库0~15  
 idleConnectionTimeout: 10000  
 connectTimeout: 10000  
 timeout: 3000  
codec:  
 class: "org.redisson.codec.JsonJacksonCodec"

### 创建redis相关的bean

package com.javacode2018.cache.demo2;  
  
import org.redisson.Redisson;  
import org.redisson.api.RedissonClient;  
import org.redisson.config.Config;  
import org.redisson.spring.cache.RedissonSpringCacheManager;  
import org.springframework.cache.CacheManager;  
import org.springframework.cache.annotation.EnableCaching;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.util.Arrays;  
  
@ComponentScan  
@EnableCaching //@1  
public class MainConfig2 {  
 @Bean //@2  
 public CacheManager cacheManager() throws IOException {  
 RedissonSpringCacheManager cacheManager = new RedissonSpringCacheManager(this.redissonClient());  
 cacheManager.setCacheNames(Arrays.asList("cache1"));  
 return cacheManager;  
 }  
  
 @Bean //@3  
 public RedissonClient redissonClient() throws IOException {  
 InputStream is = MainConfig2.class.getResourceAsStream("/com/javacode2018/cache/demo2/redis.yml");  
 Config config = Config.fromYAML(is);  
 return Redisson.create(config);  
 }  
}

@1：开启spring cache功能。

@2：自定义spring中cache管理器，这个地方我们定义了一个redis类型的管理器，底层使用redis来作为缓存的存储介质。

@3：通过redis.yml配置文件来创建一个RedissonClient，用于和redis进行交互。

### 来个测试类

package com.javacode2018.cache.demo2;  
  
import org.springframework.cache.annotation.Cacheable;  
import org.springframework.stereotype.Component;  
  
import java.util.Arrays;  
import java.util.List;  
  
@Component  
public class BookService {  
  
 @Cacheable(cacheNames = "cache1", key = "#root.targetClass.name+'-'+#root.method.name")  
 public List<String> list() {  
 System.out.println("---模拟从db中获取数据---");  
 return Arrays.asList("java高并发", "springboot", "springcloud");  
 }  
  
}

### 测试用例

@Test  
public void test7() {  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainConfig2.class);  
  
 BookService bookService = context.getBean(BookService.class);  
 System.out.println(bookService.list());  
 System.out.println(bookService.list());  
  
 {  
 System.out.println("下面打印出cache1缓存中的key列表");  
 RedissonSpringCacheManager cacheManager = context.getBean(RedissonSpringCacheManager.class);  
 RedissonCache cache1 = (RedissonCache) cacheManager.getCache("cache1");  
 cache1.getNativeCache().keySet().stream().forEach(System.out::println);  
 }  
}

### 运行输出

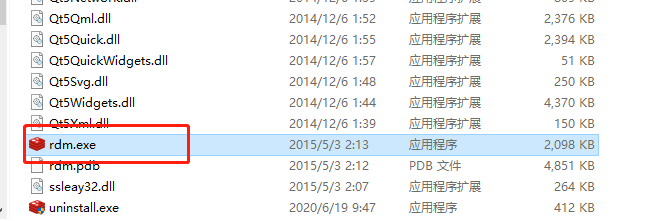
---模拟从db中获取数据---  
[java高并发, springboot, springcloud]  
[java高并发, springboot, springcloud]  
下面打印出cache1缓存中的key列表  
com.javacode2018.cache.demo2.BookService-list

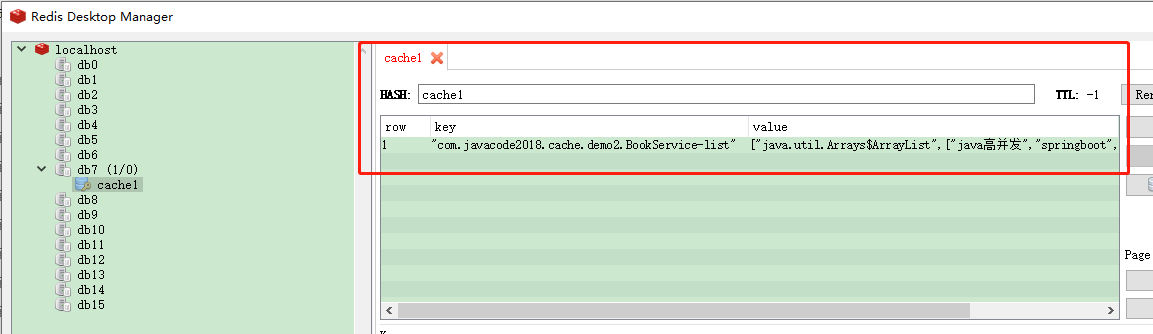
此时数据已经进入redis了，我们用redis客户端工具RedisDesktopManager来看一下。

RedisDesktopManager下载地址

链接：https://pan.baidu.com/s/1WCd-tk8dDDJnFIKciVIQsA   
提取码：x728

解压之后，点击下面的直接运行





### 案例源码

https://gitee.com/javacode2018/spring-series

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解
37. Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用
38. Spring系列第38篇：@Scheduled & @EnableScheduling定时器详解
39. Spring系列第39篇：强大的Spel表达式
40. Spring系列第40篇：缓存使用（@EnableCaching、@Cacheable、@CachePut、@CacheEvict、@Caching、@CacheConfig）

## Spring系列第42篇：JdbcTemplate实现增删改查如此简单？

本来这篇文章要写spring事务的，但是事务中大部分案例会用到JdbcTemplate相关的功能，所以先把JdbcTemplate拿出来说一下。

### 什么是JdbcTemplate？

大家来回顾一下，java中操作db最原始的方式就是纯jdbc了，是不是每次操作db都需要加载数据库驱动、获取连接、获取PreparedStatement、执行sql、关闭PreparedStatement、关闭连接等等，操作还是比较繁琐的，spring中提供了一个模块，对jdbc操作进行了封装，使其更简单，就是本文要讲的JdbcTemplate，JdbcTemplate是Spring对JDBC的封装，目的是使JDBC更加易于使用。

下面我们来看一下JdbcTemplate到底怎么玩的？

### JdbcTemplate使用步骤

1. 创建数据源DataSource
2. 创建JdbcTemplate，new JdbcTemplate(dataSource)
3. 调用JdbcTemplate的方法操作db，如增删改查

public class DataSourceUtils {  
 public static DataSource getDataSource() {  
 org.apache.tomcat.jdbc.pool.DataSource dataSource = new org.apache.tomcat.jdbc.pool.DataSource();  
 dataSource.setDriverClassName("com.mysql.jdbc.Driver");  
 dataSource.setUrl("jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8");  
 dataSource.setUsername("root");  
 dataSource.setPassword("root123");  
 dataSource.setInitialSize(5);  
 return dataSource;  
 }  
}  
  
@Test  
public void test0() {  
 //1.创建数据源DataSource  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 //2.创建JdbcTemplate，new JdbcTemplate(dataSource)  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 //3.调用JdbcTemplate的方法操作db，如增删改查  
 List<Map<String, Object>> maps = jdbcTemplate.queryForList("select \* from t\_user");  
 System.out.println(maps);  
}

输出

[{id=114, name=路人}, {id=115, name=java高并发}, {id=116, name=spring系列}]

t\_user表数据

mysql> select id,name from t\_user;  
+-----+---------------+  
| id | name |  
+-----+---------------+  
| 114 | 路人 |  
| 115 | java高并发 |  
| 116 | spring系列 |  
+-----+---------------+  
3 rows in set (0.00 sec)

上面查询返回了t\_user表所有的记录，返回了一个集合，集合中是一个Map，Map表示一行记录，key为列名，value为列对应的值。

有没有感觉到特别的方便，只需要jdbcTemplate.queryForList("select \* from t\_user")这么简单的一行代码，数据就被获取到了。

下面我们继续探索更强大更好用的功能。

### 增加、删除、修改操作

**JdbcTemplate中以update开头的方法，用来执行增、删、改操作**，下面来看几个常用的。

#### 无参情况

##### Api

int update(final String sql)

##### 案例

@Test  
public void test1() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 int updateRows = jdbcTemplate.update("INSERT INTO t\_user (name) VALUE ('maven系列')");  
 System.out.println("影响行数：" + updateRows);  
}

#### 有参情况1

##### Api

int update(String sql, Object... args)

##### 案例

**sql中使用?作为占位符。**

@Test  
public void test2() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 int updateRows = jdbcTemplate.update("INSERT INTO t\_user (name) VALUE (?)", "mybatis系列");  
 System.out.println("影响行数：" + updateRows);  
}

#### 有参情况2

##### Api

int update(String sql, PreparedStatementSetter pss)

通过PreparedStatementSetter来设置参数，是个函数式接口，内部有个setValues方法会传递一个PreparedStatement参数，我们可以通这个参数手动的设置参数的值。

##### 案例

@Test  
public void test3() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 int updateRows = jdbcTemplate.update("INSERT INTO t\_user (name) VALUE (?)", new PreparedStatementSetter() {  
 @Override  
 public void setValues(PreparedStatement ps) throws SQLException {  
 ps.setString(1, "mysql系列");  
 }  
 });  
 System.out.println("影响行数：" + updateRows);  
}

### 获取自增列的值

#### Api

public int update(final PreparedStatementCreator psc, final KeyHolder generatedKeyHolder)

#### 案例

@Test  
public void test4() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "INSERT INTO t\_user (name) VALUE (?)";  
 KeyHolder keyHolder = new GeneratedKeyHolder();  
 int rowCount = jdbcTemplate.update(new PreparedStatementCreator() {  
 @Override  
 public PreparedStatement createPreparedStatement(Connection con) throws SQLException {  
 //手动创建PreparedStatement，注意第二个参数：Statement.RETURN\_GENERATED\_KEYS  
 PreparedStatement ps = con.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);  
 ps.setString(1, "获取自增列的值");  
 return ps;  
 }  
 }, keyHolder);  
 System.out.println("新记录id：" + keyHolder.getKey().intValue());  
}

输出

新记录id：122

mysql> select id,name from t\_user;  
+-----+-----------------------+  
| id | name |  
+-----+-----------------------+  
| 114 | 路人 |  
| 115 | java高并发 |  
| 116 | spring系列 |  
| 117 | maven系列 |  
| 118 | mysql系列 |  
| 122 | 获取自增列的值 |  
+-----+-----------------------+  
6 rows in set (0.00 sec)

### 批量增删改操作

#### Api

int[] batchUpdate(final String[] sql);  
int[] batchUpdate(String sql, List<Object[]> batchArgs);  
int[] batchUpdate(String sql, List<Object[]> batchArgs, int[] argTypes);

#### 案例

@Test  
public void test5() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 List<Object[]> list = Arrays.asList(  
 new Object[]{"刘德华"},   
 new Object[]{"郭富城"},   
 new Object[]{"张学友"},   
 new Object[]{"黎明"});  
 int[] updateRows = jdbcTemplate.batchUpdate("INSERT INTO t\_user (name) VALUE (?)", list);  
 for (int updateRow : updateRows) {  
 System.out.println(updateRow);  
 }  
}

### 查询操作

#### 查询一列单行

##### Api

/\*\*  
 \* sql：执行的sql，如果有参数，参数占位符?  
 \* requiredType：返回的一列数据对应的java类型，如String  
 \* args：?占位符对应的参数列表  
 \*\*/  
<T> T queryForObject(String sql, Class<T> requiredType, @Nullable Object... args)

##### 案例

@Test  
public void test6() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String name = jdbcTemplate.queryForObject("select name from t\_user where id = ?", String.class, 114);  
 System.out.println(name);  
}

输出

路人

db中对应数据

mysql> select name from t\_user where id = 114;  
+--------+  
| name |  
+--------+  
| 路人 |  
+--------+  
1 row in set (0.00 sec)

##### 使用注意

**若queryForObject中sql查询无结果时，会报错**

如id为0的记录不存在

mysql> select name from t\_user where id = 0;  
Empty set (0.00 sec)

@Test  
public void test7() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String name = jdbcTemplate.queryForObject("select name from t\_user where id = ?", String.class, 0);  
 System.out.println(name);  
}

运行，会弹出一个异常EmptyResultDataAccessException，期望返回一条记录，但实际上却没有找到记录，和期望结果不符，所以报错了

org.springframework.dao.EmptyResultDataAccessException: Incorrect result size: expected 1, actual 0  
  
 at org.springframework.dao.support.DataAccessUtils.nullableSingleResult(DataAccessUtils.java:97)  
 at org.springframework.jdbc.core.JdbcTemplate.queryForObject(JdbcTemplate.java:784)

这种如何解决呢，需要用到查询多行的方式来解决了，即下面要说到的queryForList相关的方法，无结果的时候会返回一个空的List，我们可以在这个空的List上面做文章。

#### 查询一列多行

##### Api

以queryForList开头的方法。

<T> List<T> queryForList(String sql, Class<T> elementType);  
<T> List<T> queryForList(String sql, Class<T> elementType, @Nullable Object... args);  
<T> List<T> queryForList(String sql, Object[] args, Class<T> elementType);  
<T> List<T> queryForList(String sql, Object[] args, int[] argTypes, Class<T> elementType);

**注意：**

上面这个T虽然是泛型，但是只支持Integer.class String.class 这种单数据类型的，自己定义的Bean不支持。（所以用来查询单列数据）

elementType：查询结果需要转换为哪种类型？如String、Integer、Double。

##### 案例

@Test  
public void test8() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 //<T> List<T> queryForList(String sql, Class<T> elementType);  
 List<String> list1 = jdbcTemplate.queryForList("select name from t\_user where id>131", String.class);  
 System.out.println("list1:" + list1);  
  
 //<T> List<T> queryForList(String sql, Class<T> elementType, @Nullable Object... args);  
 List<String> list2 = jdbcTemplate.queryForList("select name from t\_user where id>?", String.class, 131);  
 System.out.println("list2:" + list2);  
  
 //<T> List<T> queryForList(String sql, Object[] args, Class<T> elementType);  
 List<String> list3 = jdbcTemplate.queryForList("select name from t\_user where id>?", new Object[]{131}, String.class);  
 System.out.println("list3:" + list3);  
  
 //<T> List<T> queryForList(String sql, Object[] args, int[] argTypes, Class<T> elementType);  
 List<String> list4 = jdbcTemplate.queryForList("select name from t\_user where id>?", new Object[]{131}, new int[]{java.sql.Types.INTEGER}, String.class);  
 System.out.println("list4:" + list4);  
}

输出

list1:[郭富城, 张学友, 黎明]  
list2:[郭富城, 张学友, 黎明]  
list3:[郭富城, 张学友, 黎明]  
list4:[郭富城, 张学友, 黎明]

sql结果：

mysql> select name from t\_user where id>131;  
+-----------+  
| name |  
+-----------+  
| 郭富城 |  
| 张学友 |  
| 黎明 |  
+-----------+  
3 rows in set (0.00 sec)

#### 查询单行记录，将记录转换成一个对象

##### Api

<T> T queryForObject(String sql, RowMapper<T> rowMapper);  
<T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper);  
<T> T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper);  
<T> T queryForObject(String sql, RowMapper<T> rowMapper, Object... args);

上面这些方法的参数中都有一个rowMapper参数，行映射器，可以将当前行的结果映射为一个自定义的对象。

@FunctionalInterface  
public interface RowMapper<T> {  
  
 /\*\*  
 \* @param ResultSet 结果集  
 \* @param 当前结果集的第几行  
 \* @return 当前行的结果对象，将当前行的结果映射为一个自定义的对象返回  
 \*/  
 @Nullable  
 T mapRow(ResultSet rs, int rowNum) throws SQLException;  
  
}

JdbcTemplate内部会遍历ResultSet，然后循环调用RowMapper#mapRow，得到当前行的结果，将其丢到List中返回，如下：

List<T> results = new ArrayList<>();  
int rowNum = 0;  
while (rs.next()) {  
 results.add(this.rowMapper.mapRow(rs, rowNum++));  
}  
return results;

##### 案例

@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@ToString  
public class User {  
 private Integer id;  
 private String name;  
}

@Test  
public void test9() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "select id,name from t\_user where id = ?";  
 //查询id为34的用户信息  
 User user = jdbcTemplate.queryForObject(sql, new RowMapper<User>() {  
 @Nullable  
 @Override  
 public User mapRow(ResultSet rs, int rowNum) throws SQLException {  
 User user = new User();  
 user.setId(rs.getInt(1));  
 user.setName(rs.getString(1));  
 return user;  
 }  
 }, 134);  
 System.out.println(user);  
}

输出

User(id=134, name=134)

##### 使用注意

**当queryForObject中sql查询无结果的时候，会报错，必须要返回一行记录**

#### 查询单行记录，返回指定的javabean

RowMapper 有个实现了类 BeanPropertyRowMapper，可以将结果映射为javabean。

@Test  
public void test10() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "select id,name from t\_user where id = ?";  
 //查询id为34的用户信息  
 RowMapper<User> rowMapper = new BeanPropertyRowMapper<>(User.class);  
 User user = jdbcTemplate.queryForObject(sql, rowMapper, 134);  
 System.out.println(user);  
}

#### 查询多列多行，每行结果为一个Map

##### Api

List<Map<String, Object>> queryForList(String sql);  
List<Map<String, Object>> queryForList(String sql, Object... args);

每行结果为一个Map，key为列名小写，value为列对应的值。

##### 案例

@Test  
public void test11() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "select id,name from t\_user where id>?";  
 List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql, 130);  
 System.out.println(maps);  
}

输出

[{id=131, name=刘德华}, {id=132, name=郭富城}, {id=133, name=张学友}, {id=134, name=黎明}]

#### 查询多列多行，将结果映射为javabean

##### Api

<T> List<T> query(String sql, RowMapper<T> rowMapper, @Nullable Object... args)

##### 案例

@Test  
public void test12() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "select id,name from t\_user where id>?";  
 List<User> maps = jdbcTemplate.query(sql, new RowMapper<User>() {  
 @Nullable  
 @Override  
 public User mapRow(ResultSet rs, int rowNum) throws SQLException {  
 User user = new User();  
 user.setId(rs.getInt(1));  
 user.setName(rs.getString(1));  
 return user;  
 }  
 }, 130);  
 System.out.println(maps);  
}

运行输出

[User(id=131, name=刘德华), User(id=132, name=郭富城), User(id=133, name=张学友), User(id=134, name=黎明)]

更简单的方式，使用BeanPropertyRowMapper

@Test  
public void test13() {  
 DataSource dataSource = DataSourceUtils.getDataSource();  
 JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
 String sql = "select id,name from t\_user where id>?";  
 List<User> maps = jdbcTemplate.query(sql, new BeanPropertyRowMapper<User>(User.class), 130);  
 System.out.println(maps);  
}

输出

[User(id=131, name=刘德华), User(id=132, name=郭富城), User(id=133, name=张学友), User(id=134, name=黎明)]

### 总结

1. 使用注意：JdbcTemplate中的getObject开头的方法，要求sql必须返回一条记录，否则会报错
2. BeanPropertyRowMapper可以将行记录映射为javabean
3. JdbcTemplate采用模板的方式操作jdbc变的特别的容易，代码特别的简洁，不过其内部没有动态sql的功能，即通过参数，动态生成指定的sql，mybatis在动态sql方面做的比较好，大家用的时候可以根据需求进行选择。

### 案例源码

git地址：  
https://gitee.com/javacode2018/spring-series  
  
本文案例对应源码：  
spring-series\lesson-003-jdbctemplate\src\main\java\com\javacode2018\jdbctemplate\demo1\Demo1Test.java

**马士兵老师所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。**

### Spring系列

1. Spring系列第1篇：为何要学spring？
2. Spring系列第2篇：控制反转（IoC）与依赖注入（DI）
3. Spring系列第3篇：Spring容器基本使用及原理
4. Spring系列第4篇：xml中bean定义详解(-)
5. Spring系列第5篇：创建bean实例这些方式你们都知道？
6. Spring系列第6篇：玩转bean scope，避免跳坑里！
7. Spring系列第7篇：依赖注入之手动注入
8. Spring系列第8篇：自动注入（autowire）详解，高手在于坚持
9. Spring系列第9篇：depend-on到底是干什么的？
10. Spring系列第10篇：primary可以解决什么问题？
11. Spring系列第11篇：bean中的autowire-candidate又是干什么的？
12. Spring系列第12篇：lazy-init：bean延迟初始化
13. Spring系列第13篇：使用继承简化bean配置(abstract & parent)
14. Spring系列第14篇：lookup-method和replaced-method比较陌生，怎么玩的？
15. Spring系列第15篇：代理详解（Java动态代理&cglib代理）？
16. Spring系列第16篇：深入理解java注解及spring对注解的增强（预备知识）
17. Spring系列第17篇：@Configration和@Bean注解详解(bean批量注册)
18. Spring系列第18篇：@ComponentScan、@ComponentScans详解(bean批量注册)
19. Spring系列第18篇：@import详解(bean批量注册)
20. Spring系列第20篇：@Conditional通过条件来控制bean的注册
21. Spring系列第21篇：注解实现依赖注入（@Autowired、@Resource、@Primary、@Qulifier）
22. Spring系列第22篇：@Scope、@DependsOn、@ImportResource、@Lazy 详解
23. Spring系列第23篇：Bean生命周期详解
24. Spring系列第24篇：父子容器详解
25. Spring系列第25篇：@Value【用法、数据来源、动态刷新】
26. Spring系列第26篇：国际化详解
27. Spring系列第27篇：spring事件机制详解
28. Spring系列第28篇：Bean循环依赖详解
29. Spring系列第29篇：BeanFactory扩展（BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor）
30. Spring系列第30篇：jdk动态代理和cglib代理
31. Spring系列第31篇：aop概念详解
32. Spring系列第32篇：AOP核心源码、原理详解
33. Spring系列第33篇：ProxyFactoryBean创建AOP代理
34. Spring系列第34篇：@Aspect中@Pointcut 12种用法
35. Spring系列第35篇：@Aspect中5中通知详解
36. Spring系列第36篇：@EnableAspectJAutoProxy、@Aspect中通知顺序详解
37. Spring系列第37篇：@EnableAsync & @Async 实现方法异步调用
38. Spring系列第38篇：@Scheduled & @EnableScheduling定时器详解
39. Spring系列第39篇：强大的Spel表达式
40. Spring系列第40篇：缓存使用（@EnableCaching、@Cacheable、@CachePut、@CacheEvict、@Caching、@CacheConfig）
41. Spring系列第41篇：@EnableCaching集成redis缓存