# Spring @Configuration配置类-Full和Lite模式

Full模式和Lite模式均是针对于Spring配置类而言的，和xml配置文件无关。值得注意的是：判断是Full模式 or Lite模式的前提是，首先你得是个容器组件。

## Lite模式

当@Bean方法在没有使用@Configuration注释的类中声明时，它们被称为在Lite模式下处理。它包括：在@Component中声明的@Bean方法，甚至只是在一个非常普通的类中声明的Bean方法，都被认为是Lite版的配置类。@Bean方法是一种通用的工厂方法（factory-method）机制。

和Full模式的@Configuration不同，Lite模式的@Bean方法不能声明Bean之间的依赖关系。因此，这样的@Bean方法不应该调用其他@Bean方法。每个这样的方法实际上只是一个特定Bean引用的工厂方法(factory-method)，没有任何特殊的运行时语义。

如下case均认为是Lite模式的配置类：

* 类上标注有@Component注解
* 类上标注有@ComponentScan注解
* 类上标注有@Import注解
* 类上标注有@ImportResource注解
* 若类上没有任何注解，但类内存在@Bean方法
* 在Spring 5.2之后新增，标注有@Configuration(proxyBeanMethods = false)。

优缺点：

* 运行时不再需要给对应类生成CGLIB子类，提高了运行性能，降低了启动时间
* 可以该配置类当作一个普通类使用喽：也就是说@Bean方法 可以是private、可以是final
* 不能声明@Bean之间的依赖，也就是说不能通过方法调用来依赖其它Bean

代码示例：

@ComponentScan("com.yourbatman.fullliteconfig.liteconfig")

@Configuration

public class AppConfig {

}

@Component

// @Configuration(proxyBeanMethods = false) // 这样也是Lite模式

public class LiteConfig {

@Bean

public User user() {

User user = new User();

user.setName("A哥-lite");

user.setAge(18);

return user;

}

@Bean

private final User user2() {

User user = new User();

user.setName("A哥-lite2");

user.setAge(18);

// 模拟依赖于user实例 看看是否是同一实例

System.out.println(System.identityHashCode(user()));

System.out.println(System.identityHashCode(user()));

return user;

}

public static class InnerConfig {

@Bean

// private final User userInner() { // 只在lite模式下才好使

public User userInner() {

User user = new User();

user.setName("A哥-lite-inner");

user.setAge(18);

return user;

}

}

}

public class Application {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// 配置类情况

System.out.println(context.getBean(LiteConfig.class).getClass());

System.out.println(context.getBean(LiteConfig.InnerConfig.class).getClass());

String[] beanNames = context.getBeanNamesForType(User.class);

for (String beanName : beanNames) {

User user = context.getBean(beanName, User.class);

System.out.println("beanName:" + beanName);

System.out.println(user.getClass());

System.out.println(user);

System.out.println("------------------------");

}

}

}

结果输出：

2073640037

932257672

class org.example.config.LiteConfig

class org.example.config.LiteConfig$InnerConfig

beanName:userInner

class org.example.bean.User

User(name=A哥-lite-inner, age=18)

------------------------

beanName:user

class org.example.bean.User

User(name=A哥-lite, age=18)

------------------------

beanName:user2

class org.example.bean.User

User(name=A哥-lite2, age=18)

------------------------

总结：

* 该模式下，配置类本身不会被CGLIB增强，放进IoC容器内的就是本尊
* 该模式下，对于内部类是没有限制的：可以是Full模式或者Lite模式
* 该模式下，配置类内部不能通过方法调用来处理依赖，否则每次生成的都是一个新实例而并非IoC容器内的单例
* 该模式下，配置类就是一普通类，所以@Bean方法可以使用private/final等进行修饰（static自然也是可以）

## Full模式

在常见的场景中，@Bean方法都会在标注有@Configuration的类中声明，以确保总是使用“Full模式”，这么一来，交叉方法引用会被重定向到容器的生命周期管理，所以就可以更方便的管理Bean依赖。

标注有@Configuration注解的类被称为full模式的配置类。自Spring5.2后改为：标注有@Configuration或者@Configuration(proxyBeanMethods = true)的类被称为Full模式的配置类。

优缺点：

* 可以支持通过常规Java调用相同类的@Bean方法而保证是容器内的Bean，这有效规避了在“Lite模式”下操作时难以跟踪的细微错误。
* 运行时会给该类生成一个CGLIB子类放进容器，有一定的性能、时间开销（这个开销在Spring Boot这种拥有大量配置类的情况下是不容忽视的，这也是为何Spring 5.2新增了proxyBeanMethods属性的最直接原因）
* 正因为被代理了，所以@Bean方法 不可以是private、不可以是final

代码实例：

@Configuration

public class FullConfig {

@Bean

public User user() {

User user = new User();

user.setName("A哥-lite");

user.setAge(18);

return user;

}

@Bean

protected User user2() {

User user = new User();

user.setName("A哥-lite2");

user.setAge(18);

// 模拟依赖于user实例 看看是否是同一实例

System.out.println(System.identityHashCode(user()));

System.out.println(System.identityHashCode(user()));

return user;

}

public static class InnerConfig {

@Bean

// private final User userInner() { // 只在lite模式下才好使

public User userInner() {

User user = new User();

user.setName("A哥-lite-inner");

user.setAge(18);

return user;

}

}

public class Application {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// 配置类情况

System.out.println(context.getBean(FullConfig.class).getClass());

System.out.println(context.getBean(FullConfig.InnerConfig.class).getClass());

String[] beanNames = context.getBeanNamesForType(User.class);

for (String beanName : beanNames) {

User user = context.getBean(beanName, User.class);

System.out.println("beanName:" + beanName);

System.out.println(user.getClass());

System.out.println(user);

System.out.println("------------------------");

}

}

}

结果输出：

550668305

550668305

class com.yourbatman.fullliteconfig.fullconfig.FullConfig$$EnhancerBySpringCGLIB$$70a94a63

class com.yourbatman.fullliteconfig.fullconfig.FullConfig$InnerConfig

beanName:userInner

class com.yourbatman.fullliteconfig.User

User{name='A哥-lite-inner', age=18}

------------------------

beanName:user

class com.yourbatman.fullliteconfig.User

User{name='A哥-lite', age=18}

------------------------

beanName:user2

class com.yourbatman.fullliteconfig.User

User{name='A哥-lite2', age=18}

------------------------

总结：

* 该模式下，配置类会被CGLIB增强(生成代理对象)，放进IoC容器内的是代理
* 该模式下，对于内部类是没有限制的：可以是Full模式或者Lite模式
* 该模式下，配置类内部可以通过方法调用来处理依赖，并且能够保证是同一个实例，都指向IoC内的那个单例
* 该模式下，@Bean方法不能被private/final等进行修饰（因为方法需要被复写嘛，所以不能私有和final。defualt/protected/public都可以哦），否则启动报错：

# @ConditionalOnProperty

ConditionalOnProperty注解类源码如下：

@Retention(RetentionPolicy.RUNTIME)

@Target({ ElementType.TYPE, ElementType.METHOD })

@Documented

@Conditional(OnPropertyCondition.class)

public @interface ConditionalOnProperty {

// 数组，获取对应property名称的值，与name不可同时使用

String[] value() default {};

// 配置属性名称的前缀，比如spring.http.encoding

String prefix() default "";

// 数组，配置属性完整名称或部分名称

// 可与prefix组合使用，组成完整的配置属性名称，与value不可同时使用

String[] name() default {};

// 可与name组合使用，比较获取到的属性值与havingValue给定的值是否相同，相同才加载配置

String havingValue() default "";

// 缺少该配置属性时是否可以加载。如果为true，没有该配置属性时也会正常加载；反之则不会生效

boolean matchIfMissing() default false;

}

# spring部分条件注解

// 执行顺序

@AutoConfigureBefore：在指定的配置类初始化前加载

@AutoConfigureAfter：在指定的配置类初始化后再加载

@AutoConfigureOrder：数越小越先初始化

// 条件配置

@ConditionalOnClass ：classpath中存在该类时起效

@ConditionalOnMissingClass ：classpath中不存在该类时起效

@ConditionalOnBean ：DI容器中存在该类型Bean时起效

@ConditionalOnMissingBean ：DI容器中不存在该类型Bean时起效

@ConditionalOnSingleCandidate ：DI容器中该类型Bean只有一个或@Primary的只有一个时起效

@ConditionalOnExpression ：SpEL表达式结果为true时

@ConditionalOnProperty ：参数设置或者值一致时起效

@ConditionalOnResource ：指定的文件存在时起效

@ConditionalOnJndi ：指定的JNDI存在时起效

@ConditionalOnJava ：指定的Java版本存在时起效

@ConditionalOnWebApplication ：Web应用环境下起效

@ConditionalOnNotWebApplication ：非Web应用环境下起效

@ConfigurationProperties：获取到配置文件数据

@ConditionalOnAvailableEndpoint：当指定的管理端点（译者注：接口地址）可用时加载 bean。如果一个端点被单独启用或使用 management.endpoints.web.exposure.include 暴露出来，都视为可用。

@ConditionalOnEnabledHealthIndicator：仅当健康指示器配置 management.health. .enabled 启用时才加载此健康检测指示器类， 需要指定为具体的值

# JavaScript 真值与虚值

在JavaScript 中，truthy（真值）指的是在布尔值上下文中，转换后的值为真的值。所有值都是真值，除非它们被定义为 假值（即除 false、0、-0、0n、“”、null、undefined 和 NaN 以外皆为真值）。

falsy 值 (虚值) 是在 Boolean 上下文中认定为 false 的值。在 JavaScript 中只有 8 个 falsy 值。

if (false)

if (null)

if (undefined)

if (0)

if (0n)

if (NaN)

if ('')

if ("")

if (``)

if (document.all)

# Spring Factories

@ComponentScan 注解的作用是扫描 @SpringBootApplication 所在的 Application 类所在的包（basepackage）下所有的 @component 注解（或拓展了 @component 的注解）标记的 bean，并注册到 spring 容器中。

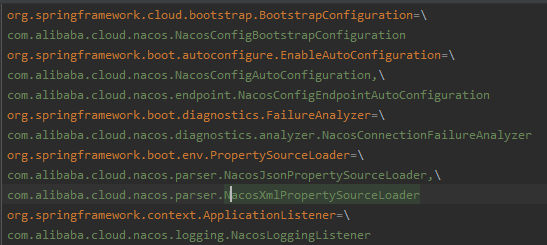
Spring Boot 中不能被默认路径扫描的配置类的方式，有 2 种：

* 在Spring Boot主类上使用@Import注解
* 使用 spring.factories 文件

spring-core 包里定义了 SpringFactoriesLoader 类，这个类实现了检索 META-INF/spring.factories 文件，并获取指定接口的配置的功能。在这个类中定义了两个对外的方法：

* loadFactories 根据接口类获取其实现类的实例，这个方法返回的是对象列表。
* loadFactoryNames 根据接口获取其接口类的名称，这个方法返回的是类名的列表。

在这个方法中会遍历整个 spring-boot 项目的 classpath 下 ClassLoader 中所有 jar 包下的 spring.factories文件。也就是说我们可以在自己的 jar 中配置 spring.factories 文件，不会影响到其它地方的配置，也不会被别人的配置覆盖。



spring.factories用键值对的方式记录了所有需要加入容器的类。

# Java Type

Type是Java 编程语言中所有类型的公共高级接口（官方解释），也就是Java中所有类型的“爹”；其中，“所有类型”的描述尤为值得关注。它并不是我们平常工作中经常使用的 int、String、List、Map等数据类型，而是从Java语言角度来说，对基本类型、引用类型向上的抽象；Type体系中类型的包括：原始类型(Class)、参数化类型(ParameterizedType)、数组类型(GenericArrayType)、类型变量(TypeVariable)、基本类型(Class)：

* 原始类型，不仅仅包含我们平常所指的类，还包括枚举、数组、注解等；
* 参数化类型，就是我们平常所用到的泛型List、Map；
* 数组类型，并不是我们工作中所使用的数组String[] 、byte[]，而是带有泛型的数组，即T[] ；
* 基本类型，也就是我们所说的java的基本类型，即int,float,double等

Type的直接子接口：

* ParameterizedType：表示一种参数化的类型，比如Collection，即普通的泛型。

ParameterizedType 是参数化类型，即泛型，类似List、Map<Integer, String>、List<? extends Number>带有类型参数的类型，也可以是自定义的，再调用getRawType()与getActualTypeArguments()两个方法，就可以得到声明此参数化类型的类(java.lang.Comparable)和实际的类型参数数组([? super T])，而这个? super T又是一个WildcardType类型。

public interface ParameterizedType extends Type {

// 获取泛型类型Map<Integer, String>，获得的是Integer和String

Type[] getActualTypeArguments();

// 获取<>前面的类型

Type getRawType();

// 返回的是这个 ParameterizedType 所在的类的 Type (注意当前的 ParameterizedType 必须属于所在类的 member)。

Type getOwnerType();

}

* TypeVariable：是各种类型变量的公共父接口，就是泛型里面的类似T、E。 \*\* 在这需要强调的是，TypeVariable代表着泛型中的变量，而ParameterizedType则代表整个泛型；

public interface TypeVariable<D extends GenericDeclaration> extends Type, AnnotatedElement {

// 获取变量的上边界

Type[] getBounds();

// 在哪个类上进行泛型声明

D getGenericDeclaration();

// 声明时写的名字

String getName();

// 返回AnnotatedType对象的数组，这些对象表示使用类型来表示此TypeVariable表示的类型参数的上限。数组中对象的顺序与类型参数声明中边界的顺序相对应。如果类型参数声明没有边界，则返回长度为0的数组。

AnnotatedType[] getAnnotatedBounds();

}

GenericDecalaration，GenericDeclaration 有三个直接子类 Class,Construtor,Method,也就是说只能在这几种对象上进行范型变量的声明（定义）。GenericDeclaration的接口方法getTypeParameters用来逐个获取该GenericDeclaration的范型变量声明。

public interface GenericDeclaration extends AnnotatedElement {

// 返回的泛型变量数组代表了泛型声明的内容。

public TypeVariable<?>[] getTypeParameters();

}

类型变量声明（定义）的时候不能有下限（既不能有super），否则编译报错。为什么？T extends classA表示泛型有上限classA，当然可以，因为这样，每一个传进来的类型必定是classA（具有classA的一切属性和方法），但若是T super classA，传进来的类型不一定具有classA的属性和方法，当然就不适用于泛型

* GenericArrayType：表示一种元素类型是参数化类型或者类型变量的数组类型，比如List<>[]，T[]这种。

public interface GenericArrayType extends Type {

// 获取泛型数组中的元素类型

Type getGenericComponentType();

}

* WildcardType：代表一种通配符类型表达式，类似? super T这样的通配符表达式。WildcardType是通配符表达式，或泛型表达式，它虽然是Type的一个子接口，但并不是Java类型中的一种，表示的仅仅是类似 ? extends T、? super K这样的通配符表达式。

public interface WildcardType extends Type {

// 获取表达式上届

Type[] getUpperBounds();

// 获取表达式下届

Type[] getLowerBounds();

}