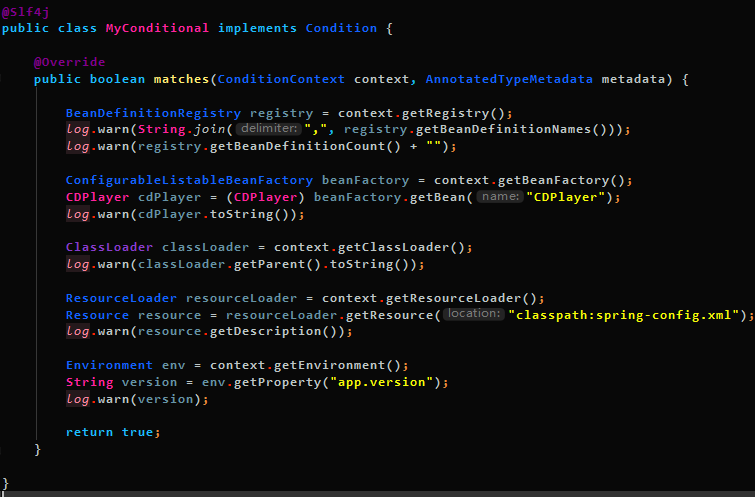
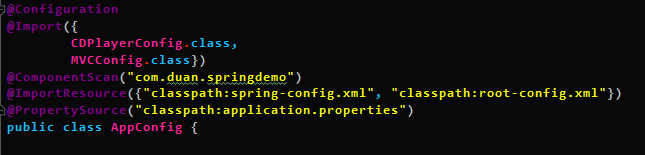
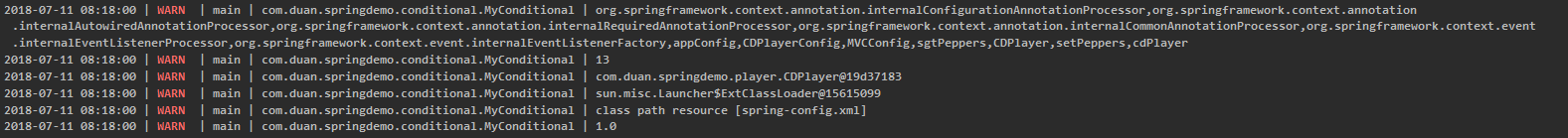
## **3.2条件化 Bean**

@Conditional 注解，可以应用到带有 @Bean 注解的方法上，如果给定的条件为 true，Bean 才会被创建，否则不创建。









## **3.3自动装配的歧义性**

自动装配仅在只有一个 bean 匹配时有效，如果不只有一个 bean 的话，就会阻碍 spring 自 动装配属性、构造器参数或方法参数。

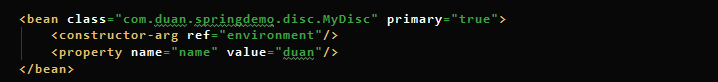
解决：

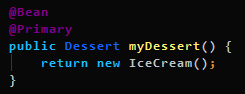
1. 将可选 bean 中的每一个设为首选（ primary ）

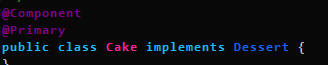
2. 使用限定符来帮助 spring 缩小范围（ qualifier ）

**标识首选的 bean**

@Primary 可与 @Component 和 @Bean 注解配合使用。xml 中对于 bean 标签有对应的 boolean 属性 primary。



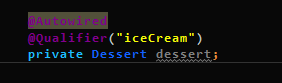




当有两个可选的 Dessert 时，spring 无法从多个首选 bean 中做出选择，因此 @Primary 只在唯一时有效。

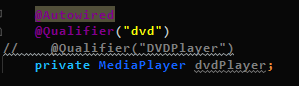
**明确限定 bean**

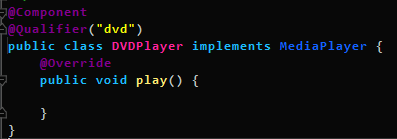
@Qualifier 注解是使用限定符的主要方式，可以与 @Autoware 和 @Inject 注解配合使用。



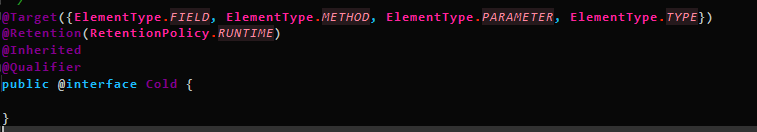
缺点：@Qualifier 指定的 bean id 与类名（方法名）紧耦合，类名修改时 @Qualifer 注解将失效。

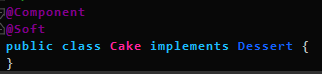
通过在 Bean 定义处和使用处同时使用 @Qualifier 注解，可以实现基于 String 描述的对多个限定特性进行限定（java 8 才支持重复注解，但 @Qualifier 不支持重复注解），因此只能明确限定一个特性。

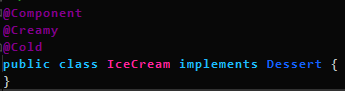
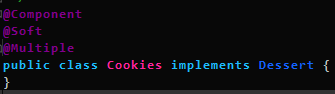




可以通过创建自定义的注解来实现限定符的功能，这些注解需要被 @Qualifier 注解注解，那么这个注解本身可以作为一个限定特性。









原始的 @Qualifier 注解使用 String 进行限定，而基于 @Qualifier 注解的注解使用注解进行限定，类型更为安全，同时能够解耦。

## **3.4 bean 的作用域**

在默认的情况下，Spring 上下文的所有 bean 都是作为单例存在的。

单例会保持一些状态，对象会被污染，有时重用时不安全的。

作用域：

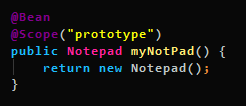
1. 单例（Singleton）：整个应用中，只创建 bean 的一个实例

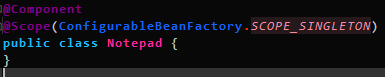
2. 原型（Prototype）：每次注入或通过 Spring 上下文获取时都创建新的实例

3. 会话（Session）：在 Web 应用中，为每个会话创建一个 bean 实例

4. 请求（Request）：在 Web 应用中，为每个请求创建一个 bean 实例

使用 @Scop 组合 @Bean 和 @Component（包括自注解，被 @Component 注解的注解）。

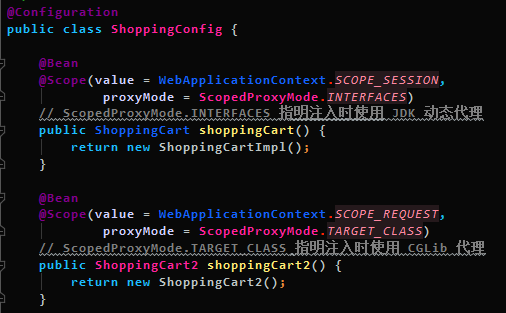




在 xml 中，bean 标签有对应的 socp 属性。

## **会话和请求作用域**

就购物车而言，会话作用域最为合适。



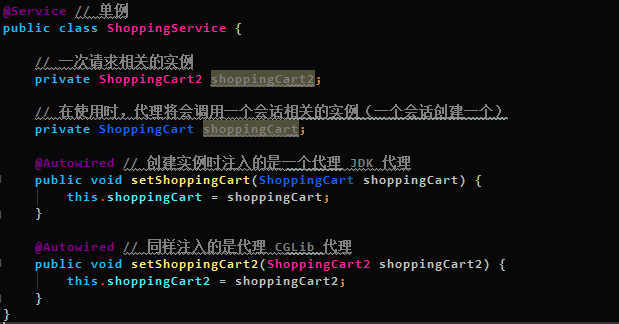
SCOPE\_SESSION 对于一次会话会创建新的实例。

proxyMode 解决了将会话或请求作用域 bean 注入到单例 bean 中的问题。

实际注入时会注入目标 bean 的一个代理，使用时代理会对其进行懒解析并将调用委托给会话作用域内正真的 bean。

ScopedProxyMode.INTERFACE 指明目标类是一个接口，注入时需使用 JDK 动态代理

ScopedProxyMode.TARGET\_CLASS 指明目标类是类，注入时使用 CGLib 代理



xml 中也提供了相应属性：bean 元素的 scop 属性，以及 bean 元素下的子元素 aop:scoped-proxy 。

<aop:scoped-proxy proxy-target-class="false"/>

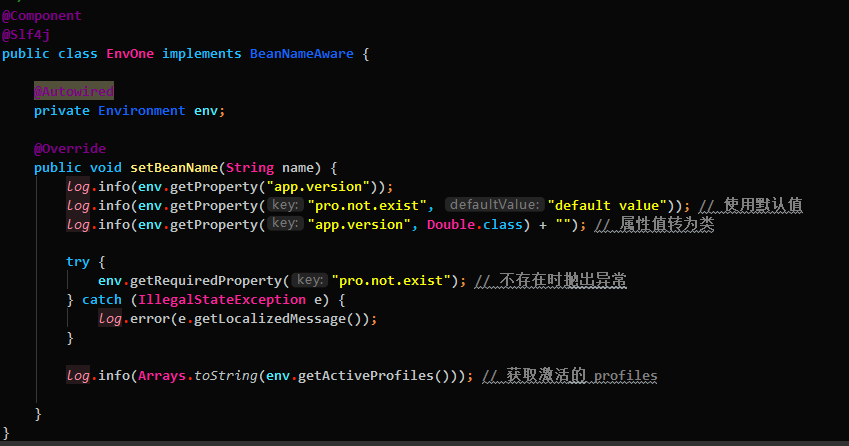
通过 proxy-target-class 属性指明目标 bean 是接口还是类。

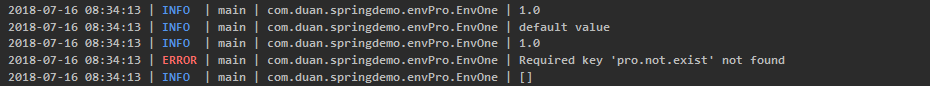
## **3.5 运行时注入**

spring 提供了两种在运行时求值的方式：

1. 属性占位符（Property placeholder）

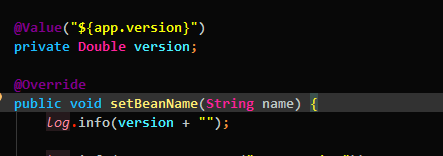
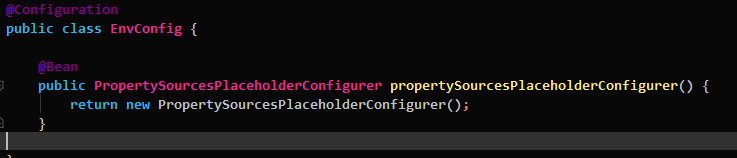
2. spring 表达式语言（SpEL）





**属性占位符**

形式为 "${...}" ，使用 @Value 注解，java 配置需要配置 PropertySourcesPlaceholderConfigurer bean，xml 可通过 <context:property-placeholder /> 开启。



xml 中也可通过 “${...}” 的形式使用。

**使用 spring 表达式语言装配**

1. 使用 bean id 引用其他 bean

2. 调用方法和访问对象属性

3. 对值进行算术、关系和逻辑运算

4. 正则表达式匹配

5. 集合操作

SpEL 要放到 “#{...}” 表达式中，在 xml 中也可以使用。

**#{T(System).currentTimeMillis()}**

T()表达式用于将括号内表达式视为 java 类，能够访问目标类型的静态方法和常量。也直接装配目标类型。

**#{envOne.version}**

envOne 为 bean 的 id，可获得其可访问成员的值

**#{systemProperties['jnidispatch.path']}**

systemProperties获取系统属性

**#{envOne.getClass().toString().toUpperCase()}**

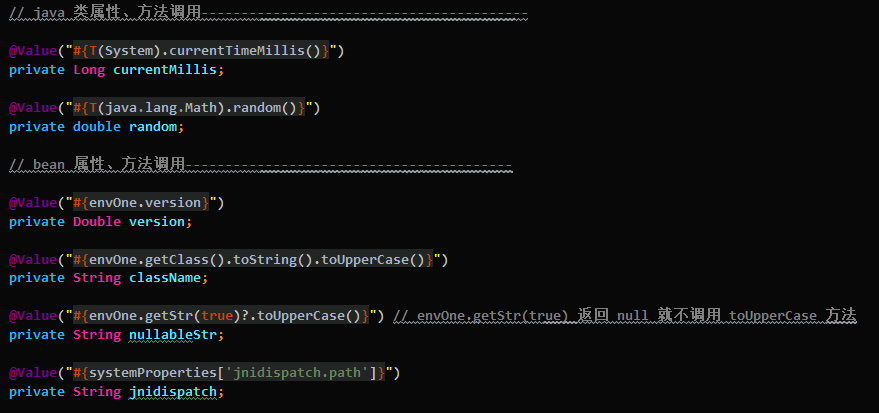
调用 bean 的方法

**#{envOne.getStr(true)?.toUpperCase()}**

envOne.getStr(true) 返回 null 就不调用 toUpperCase 方法

**#{T(java.lang.Math).random()}**

调用 java 类的静态方法



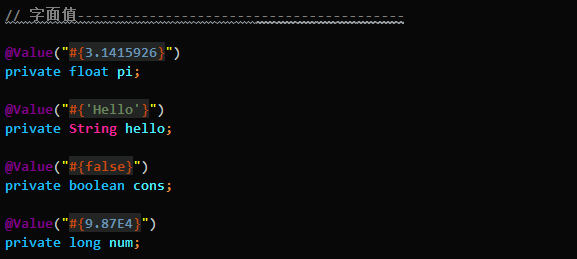
**字面值**

#{3.1415926}

#{'Hello'}

#{false}

#{9.87E4}



**运算符**

算术：+ - \* / % ^

比较：< > == <= >= lt gt eq le ge

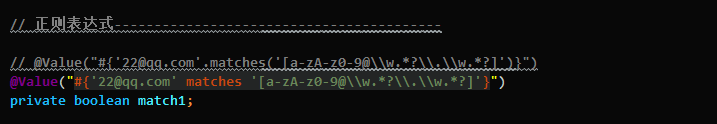
逻辑：and or not |

条件：?: (ternary) ?: (Elvis) 三元运算符，Elvis 表达式

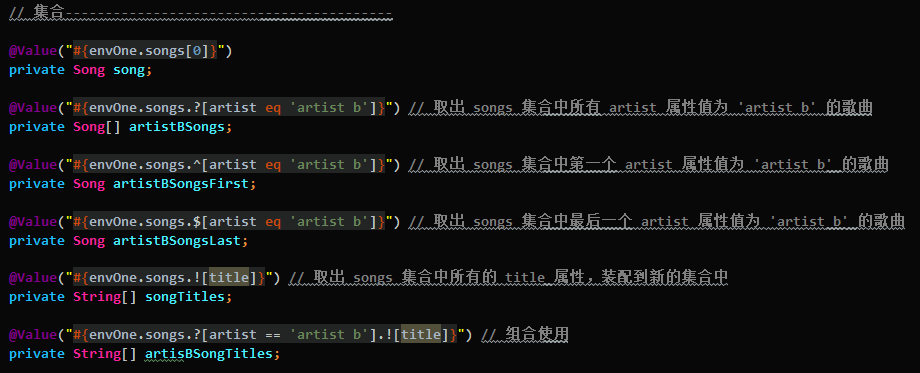
正则：matches

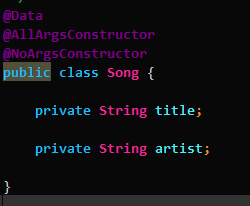


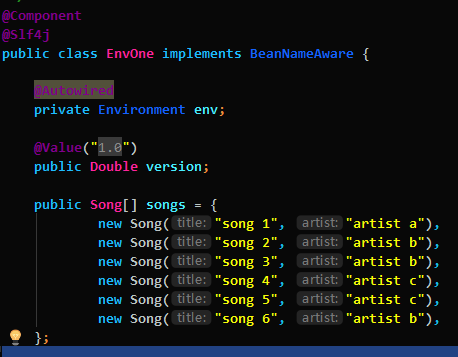
**正则表达式**



**集合**

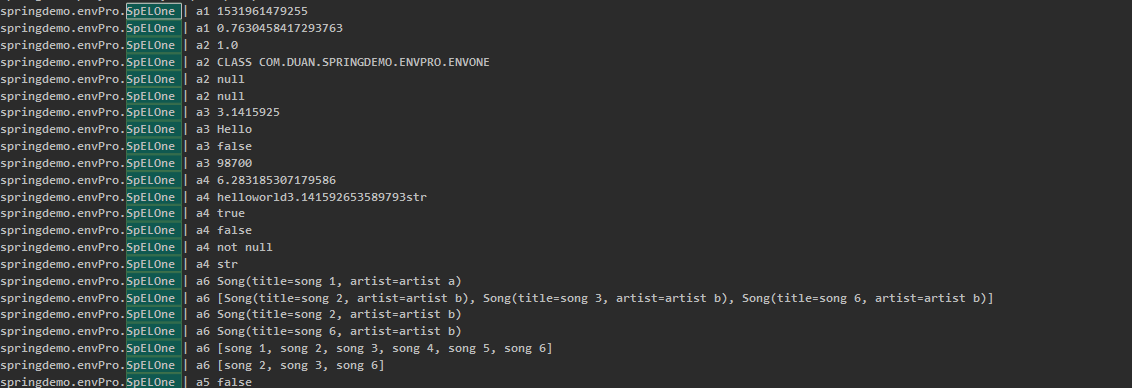






**log输出**





## **4.1 什么是面向切面编程**

软件系统中的一些功能需要应用到应用程序的多个地方，如日志、安全、缓存和声明式事务。

散布于应用中多处的功能被称为横切关注点，这些横切关注点从逻辑上与应用的业务逻辑相分离，但往往会直接嵌入到业务逻辑之中，把这些横切关注点从业务逻辑分离正是面向切面编程要解决的问题。

1. 继承委托使对象体系变得复杂，切面提供了另一种更加清晰简洁的方案。

2. 每个关注点都集中在一个地方，不用分散在代码中。

3. 服务模块更简洁，因为他们只关注核心功能。

**通知（Advice）**

切面要完成的工作（**什么**），何时完成这个工作（**何时**）。

1. before：前置通知

2. After：后置通知

3. After-returning：目标方法执行之后执行通知

4. After-throwing：在目标方法抛出异常后执行通知

5. Around：包裹了目标方法，在方法执行前和调用后执行通知

**连接点（Join point）**

应用执行过程中能够插入切面的点，切面可以利用这些点插入到正常业务逻辑之中，并添加新的行为。

**切点（Poincut）**

一个切面不需要通知应用的所有连接点，切点有助于缩小切面所通知的连接点的范围（**何处**）。

**切面（Aspect）**

切面是通知和切点的结合，通知和切点共同定义了切面的全部内容——**切面要完成什么功能，在何时何处完成。**

**引入（Introduction）**

向现有类添加新的方法或属性。

**织入（Weaving）**

织入把切面应用到目标对象并创建新的代理对象，切面在指定的连接点织入到目标对象，可在多个可选的对象声明周期里织入：

1. 编译期：需要特殊的编译器，如 AspectJ 的织入编译器

2. 类加载期：目标类加载到 JVM 时织入，需要特殊的类加载器（ClassLoader），增强目标类的字节码

3. 运行期：运行时，AOP 容器动态的为目标对象创建代理对象，如 Spring AOP

**Spring 对 aop 的支持**

spring 提供了 4 种 aop 支持

1. 基于代理的经典 spring aop

2. 纯 POJO 切面（POJO：Plain ordinary java object - java beans）

3. @AspectJ 注解驱动的切面

4. 注入式 AspectJ 切面

spring aop 构建在动态代理之上，所有spring 对 aop 的支持局限于方法拦截。

**spring 在运行时通知对象**

1. spring 代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的 bean，当代理拦截到方法调用时，在调用目标 bean 方法之前，会执行切面逻辑。

2. spring运行时才创建代理对象，所以不需要特殊的编译器织入 spring aop 切面。

AspectJ 最初是以 java 语言扩展的方式实现的，有自己的编译器，除方法切点外，还提供了字段（拦截字段的修改）和构造器（构造器调用时）接入点。（JBoss 也提供了同样的粒度）

**spring aop 与 AspectJ 项目**

1. spring aop 与 AspectJ 项目间有大量的协作，Spring 对 aop 的支持在很多方面借鉴了 AspectJ 项目（注解驱动、切点表达式语言的子集 ...）

2. spring 的 AspectJ 自动代理仅仅使用 @AspectJ 作为创建切面的指定，切面依然是基于代理的。这意味着尽管使用的是 @AspectJ 注解，我们还是限定于代理方法的调用，如果想利用 AspectJ 的所有能力，我们必须在运行时使用 AspectJ 并且不依赖 Spring 来创建基于代理的切面。

## **4.2 通过切点选择连接器**

在 spring aop 中，要使用 AspectJ 的切点表达式语言来定义切点，spring 仅支持 AspectJ 切点指示器（pointcut designator）的一个子集。

AspectJ 指示器

1. arg() 限制参数

2. @args() 限制参数注解

3. execution() 匹配连接点

4. this() 限制代理的 bean 引用的类型

5. target 限制目标对象类型

6. @target() 限制目标对象类型，同时类型需要有特定注解

7. within() 限制类型，包

8. @within() ...?

9. annotation 限定注解

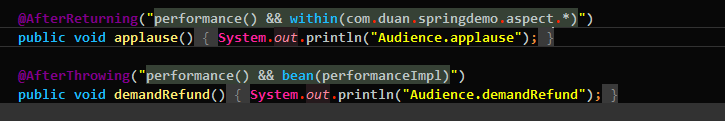
10. bean() spring 提供的指示器，用于限定指定 bean

连接指示器

1. "(..)" 表示不限定参数，"\*" 任意返回值类型

2. &&(and)、||(or)、!(not)





## **4.3 使用注解创建切面**

1. @After

2. @AfterReturning

3. @AfterThrowing

4. @Around

5. @Before

使用 javaConfig 时需要在配置类上加上 @EnableAspectJAutoProxy 注解，xml 时使用：<aop:aspectj-autoproxy>

- java 8 允许在一个条目上重复使用一个注解，只要这个注解本身有 @Repeatable 注解。