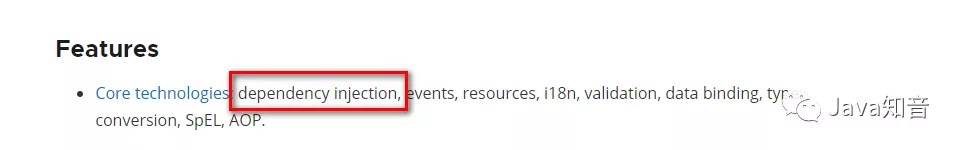
https://mp.weixin.qq.com/s/x59XP-FUPm3omW9nNpe9vw

# Spring的依赖注入

## 前言

提起Spring,大家肯定不陌生,它是每一个Java开发者绕不过去的坎。Spring 框架为基于 java 的企业应用程序提供了一整套解决方案,方便开发人员在框架基础快速进行业务开发。

在官网中，我们发现它的核心技术之一：Dependency Injection，简称：DI ,翻译过来就是**依赖注入**。今天我们就来盘一盘它。



在本文中，我们将深入研究 Spring 框架 DI背后的故事，包括 Spring Inversion of Control(控制反转)、 DI 和 ApplicationContext 接口。基于这些基本概念，我们将研究如何使用基于 java 和基于 XML 的配置来 创建Spring 应用程序。最后，我们将探讨在创建 Spring 应用程序时遇到的一些常见问题，包括 **bean冲突**和**循环依赖性**。

## 一 控制反转(Inversion of Control)

在学习DI之前,我们先学习一下 IoC（**控制反转**），接下来的一段可能读起来会让你感觉比较啰嗦,但是要细细体会每一次改变的意图,和我们的解决方案，对于理解**控制反转**非常重要。

首先来了解下我们通常实例化一个对象的方式。在 平时，我们使用 new 关键字实例化一个对象。例如，如果有一个 Car 类，我们可以使用以下方法实例化一个对象 Car



因为汽车有很多零部件组成,我们定义Engine接口来模拟汽车引擎,然后将engine对象作为成员变量放在Car类现在,我们可以调用start()方法吗?显然是不行的,一眼可以看出会报NullPointerException (NPE)，因为我们没有在Car的构造函数中初始化engine。通常我们采用的方案就是在Car的构造函数中觉得使用Engine接口的哪个实现,并直接将该实现分配给engine字段;

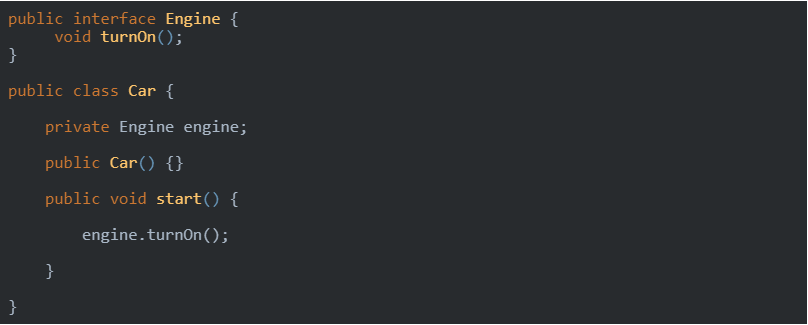
现在,我们来首先创建Engine接口的实现类

现在,我们可以调用start()方法吗?显然是不行的,一眼可以看出会报NullPointerException (NPE)，因为我们没有在Car的构造函数中初始化engine。通常我们采用的方案就是在Car的构造函数中觉得使用Engine接口的哪个实现,并直接将该实现分配给engine字段;

现在,我们来首先创建Engine接口的实现类

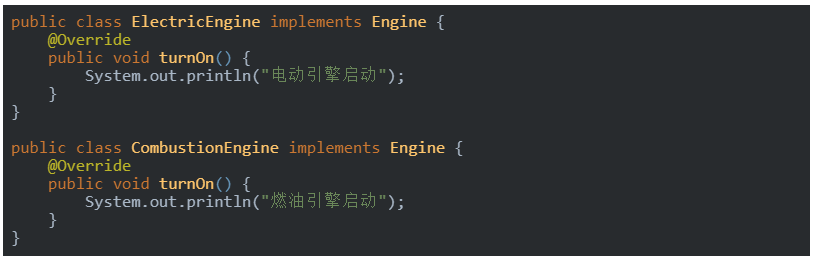
现在,我们可以调用start()方法吗?显然是不行的,一眼可以看出会报NullPointerException (NPE)，因为我们没有在Car的构造函数中初始化engine。通常我们采用的方案就是在Car的构造函数中觉得使用Engine接口的哪个实现,并直接将该实现分配给engine字段;

现在,我们来首先创建Engine接口的实现类

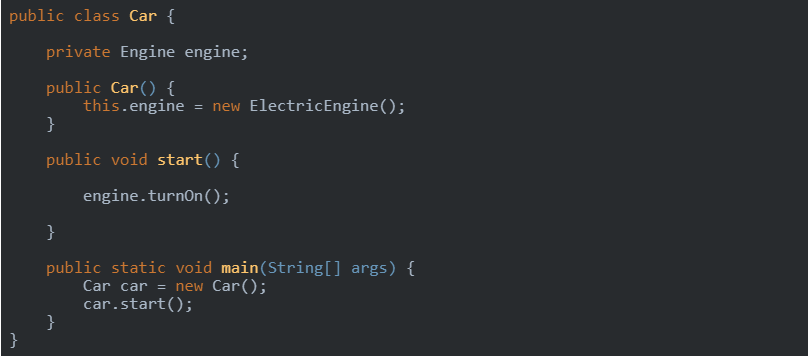


现在,我们可以调用start()方法吗?显然是不行的,一眼可以看出会报NullPointerException (NPE)，因为我们没有在Car的构造函数中初始化engine。通常我们采用的方案就是在Car的构造函数中觉得使用Engine接口的哪个实现,并直接将该实现分配给engine字段;

现在,我们来首先创建Engine接口的实现类



我们修改Car的构造函数,使用ElectricEngine实现,将我们的engine字段分配给一个实例化的ElectricEngine对象



现在我们执行start()方法,我们会看到如下输出:

电动引擎启动

燃油引擎启动

大功告成,我们成功解决了 NPE(空指针)问题,但是我们胜利了吗?哈哈哈,显然没有!

在解决问题的同时,我们又引入了另一个问题。尽管我们通过抽象Engine接口，然后通过不同的Engine实现类来负责不同类型引擎的业务逻辑，的确是很好的设计策略。但是**细心**的伙伴可能已经发现了，我们Car类的构造函数中将engine声明为CombustionEngine,这将导致**所有车**都有一个燃油引擎。假如我们现在要创建不同的汽车对象，它有一个电动引擎，我们将不得不改变我们的设计。比较常见的方法是创建两个独立里的类，各司其职，在他们的构造函数中将engine分配给Engine接口的不同实现；

例如:



通过上面的一顿**骚**操作,我们成功的解决了我们引擎的问题。如果是一个日常需求,我们已经可以成功交工了。但是这显然不是我写这篇文章的目的。

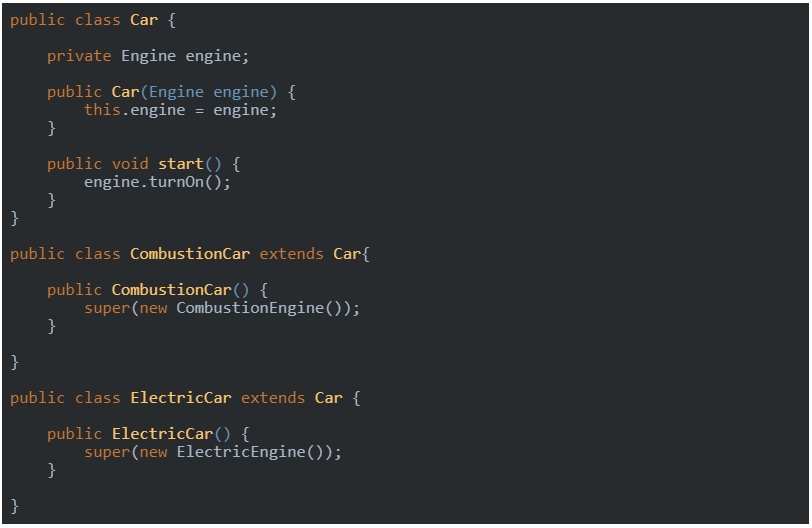
从设计的角度来说，目前的代码是糟糕的，有以下两点原因：

1. 在两个不同的类中，存在重复的start()方法;
2. 我们需要为每个新的Engine 实现类创建一个新的类;

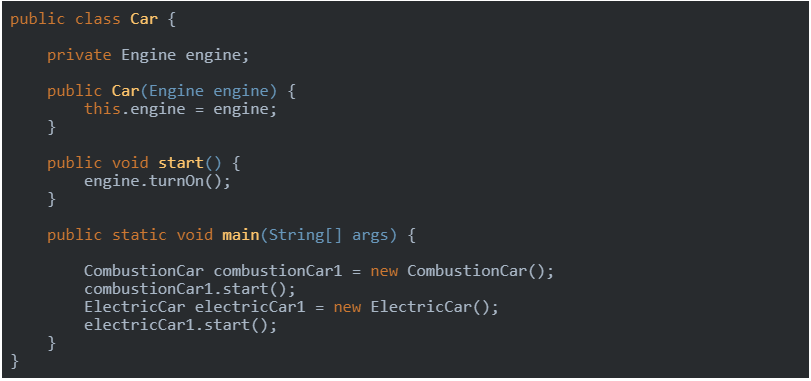
尤其后一个问题更加难以解决,因为我们不控制Engine的实现,随着开发人员不断的创建自己的实现类,这个问题会更加恶化;

**带着上面的问题,我们继续思考…………………**

我们可以创建一个父类Car,将**公共代码**抽取到父类中,可以轻松解决第一个问题。由于Engine字段是私有的,我们在父类Car的构造函数中接收Engine对象,并且进行赋值。



通过这种方法,我们成功的解决了**代码重复**的问题,我们来测试一下:



电动引擎启动

燃油引擎启动

**那么我们该如何解决我们提出的第二个问题那?**

其实这个问题我们可以换个角度看:为什么我们要去关注CombustionCar和ElectricCar,我们现在将关注点回到我们的Car,我们现在已经允许客户端实例化Car对象时候将Engine对象作为构造函数的参数传入,其实已经消除了为每个Engine对象创建新Car的问题。因为现在Car类依赖于Engine接口,并不知道任何Engine的实现;

通过带有Engine参数的构造函数,我们已将要使用哪个Engine实现的决定从Car类本身（最初由CombustionEngine决定）**更改为**实例化Car类的客户端。决策过程的这种逆转称为IoC原则。现在，由客户端控制使用哪种实现，而不是由Car类本身控制使用哪种Engine实现。

**有点绕，大家结合下面的示例代码，细细琢磨**



执行上面的代码,我们发现都可以获得我们想要的结果:

电动引擎启动

燃油引擎启动

从上面的例子我们可以看到，实例化Car类的客户端可以控制所使用的Engine实现，并且取决于将哪个Engine实现传递给Car构造函数，Car对象的行为发生巨大变化。为什么这么说，接着看下面

## 二 依赖注入(Dependency Injection)

在上面控制反转的知识点,我们已经解决了由谁决定使用哪种Engine实现的问题,但是不可避免,我们也更改了实例化一个Car对象的步骤;

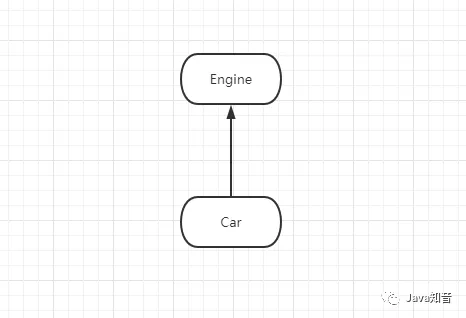
最开始,我们实例化Car不需要参数,因为在它的构造函数里面已经为我们new了Engine对象。使用IoC方法之后，我们要求在实例化一个Car之前,我们需要先创建一个Engine对象,并作为参数传递给Car构造对象。换句话说，最初，我们首先实例化Car对象,然后实例化Engine对象。但是，使用IoC之后,我们首先实例化Engine对象,然后实例化Car对象;

因此,我们在上面的过程中创建了一个依赖关系。不过这种依赖关系不是指编译时候Car类对Engine接口的依赖关系,相反,我们引入了一个运行时依赖关系。在运行时，实例化Car对象之前,必须首先实例化Engine对象。

### 2.1 依赖关系树

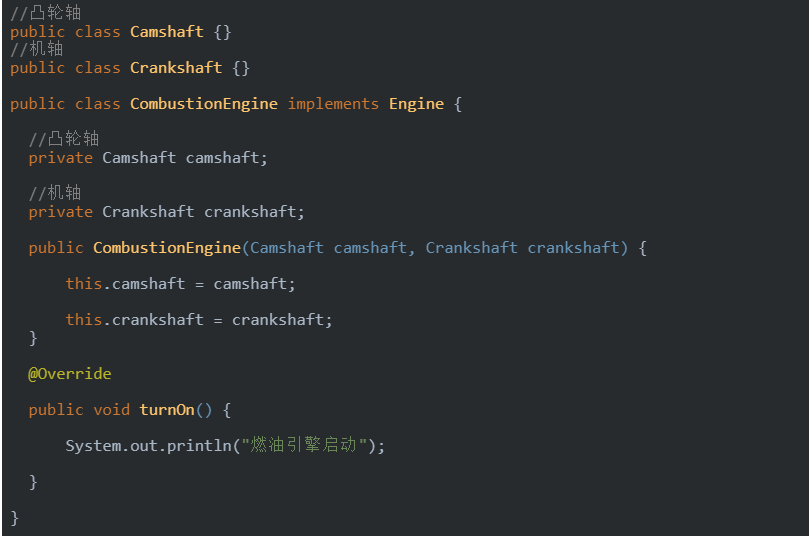
**某一个具体的依赖对象大家可以理解为Spring中的bean,对于两个有依赖关系的bean,其中被依赖的那个bean,我们把它称为依赖对象**

我们用图形化的方式来看看它们之间的依赖关系，其中图形的节点代表对象，箭头代表依赖关系（箭头指向依赖对象）。对于我们我的Car类,依赖关系树非常简单:

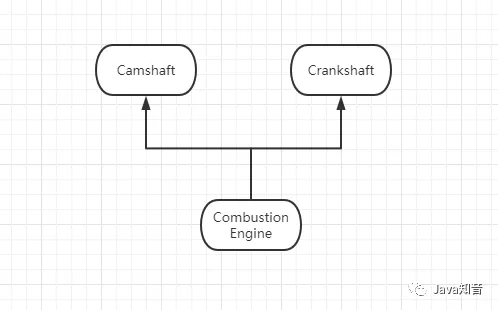


如果**依赖关系树**的终端结点还有自己的附加依赖关系，那么这个**依赖关系树**将变得更加复杂。现在再看我们上面的例子，如果CombustionEngine 还有其他依赖对象，我们首先需要创建CombustionEngine的依赖对象,然后才能实例化一个CombustionEngine对象。这样在创建Car对象时候,才能将Combu

stionEngine传递给Car的构造函数;



经过我们改造,我们现在的依赖关系树变为下面的样子



### 2.2 依赖注入框架

随着我们不断引入更多的依赖关系,这种复杂性将继续增长。为了解决这个复杂问题，我们需要基于依赖关系树抽取对象的创建过程。这就是**依赖注入框架**。

一般来说，我们可以把这个过程分为三个部分:

1. 声明需要创建的对象需要哪些依赖对象
2. 注册创建这些依赖对象所需要的类
3. 提供一种使用1和2两点思想创建对象的机制

通过反射，我们可以查看 Car 类的构造函数，并且知道它需要一个 Engine 参数。因此为了创建Car对象,我们必须创建至少一个Engine接口的实现类用作依赖项来使用。在这里，我们创建一个CombustionEngine 对象(**为了方便,暂时当做只有一个实现类,bean冲突问题待会再说**)来声明它作为依赖项来使用,就满足Car对象创建时的需求.

其实,这个过程是递归的,因为CombustionEngine 依赖于其他对象,我们需要不断重复第一个过程,直到把所有依赖对象声明完毕,然后注册创建这些依赖对象所需要的类。

第三点其实就是将前面两点思想付诸实施，从而形成一种创建对象的机制

举个例子:比如我们需要一个Car对象,我们必须遍历依赖关系树并检查是否存在至少一个符合条件的类来满足所有依赖关系。例如，声明CombustionEngine类可满足Engine节点要求。如果存在这种依赖关系，我们将实例化该依赖关系，然后移至下一个节点。

如果有一个以上的类满足所需的依赖关系，那么我们必须显式声明应该选择哪一种依赖关系。稍后我们将讨论 Spring 是如何做到这一点的。

一旦我们确定所有的依赖关系都准备好了，我们就可以从终端节点开始创建依赖对象。对于 Car 对象，我们首先实例化 Camshaft 和Crankshaftーー因为这些对象没有依赖关系ーー然后将这些对象传递给 CombustionEngine 构造函数，以实例化 CombunstionEngine 对象。最后，我们将 CombunstionEngine 对象传递给 Car 构造函数，以实例化所需的 Car 对象。

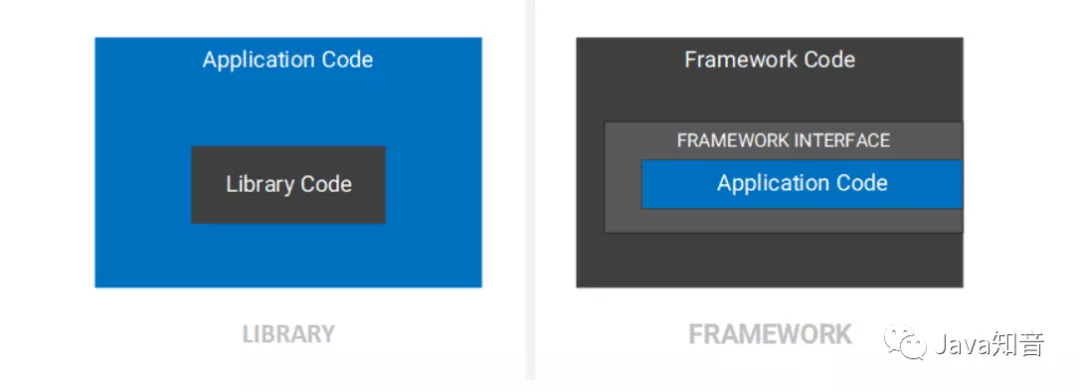
了解了 DI 的基本原理之后，我们现在可以继续讨论 Spring 如何执行 DI。

2.3 Spring的依赖注入

Spring的核心是一个DI框架，它可以将DI配置转换为Java应用程序。

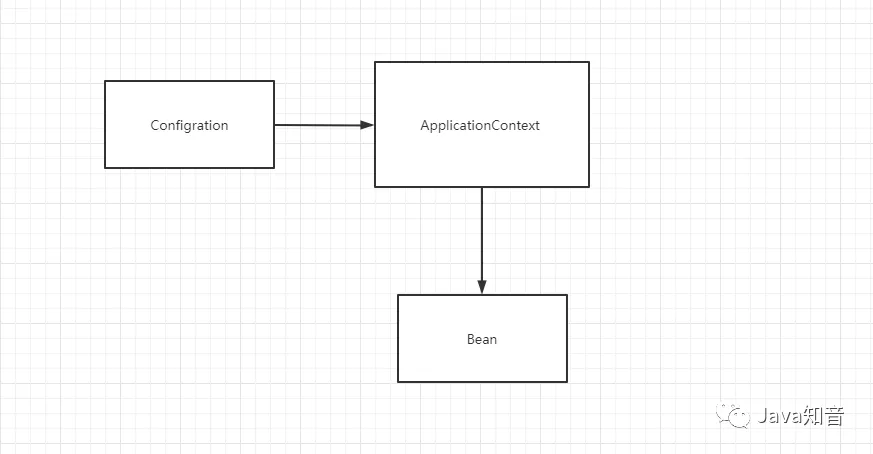
在这里我们要阐述一个问题:那就是**库和框架的区别**。库只是类定义的集合。背后的原因仅仅是代码重用，即获取其他开发人员已经编写的代码。这些类和方法通常在域特定区域中定义特定操作。例如，有一些数学库可让开发人员仅调用函数而无需重做算法工作原理的实现。

框架通常被认为是一个骨架，我们在其中插入代码以创建应用程序。许多框架保留了特定于应用程序的部分，并要求我们开发人员提供适合框架的代码。在实践中，这意味着编写接口的实现，然后在框架中注册实现。



### 2.4 ApplicationContext

在 Spring 中，框架围绕 ApplicationContext 接口实现上一节中概述的三个 DI 职责。通常这个接口代表了一个上下文。因此，我们通过基于 java 或基于 xml 的配置向 ApplicationContext 注册合适的类，并从 ApplicationContext 请求创建 bean 对象。然后 ApplicationContext 构建一个依赖关系树并遍历它以创建所需的 bean对象。



以有多个 ApplicationContext，每个 ApplicationContext 可以有单独的配置。例如，一个 ApplicationContext 可能被配置为使用 CombustionEngine 作为其引擎实现，而另一个容器可能被配置为使用 ElectricEngine 作为其实现。

在本文中，我们将重点讨论每个应用程序的单个 ApplicationContext，但是下面描述的概念即使在一个应用程序有多个 ApplicationContext 实例时也适用。

## 三 基于 java 的配置

Spring为我们提供了两种基于 java 的配置方式

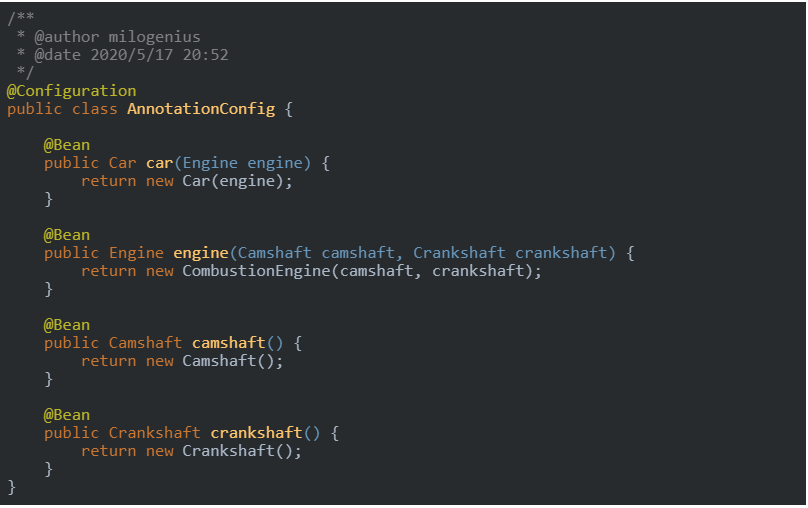
1. 基本配置
2. 自动配置

### 3.1 基于 java 的基本配置

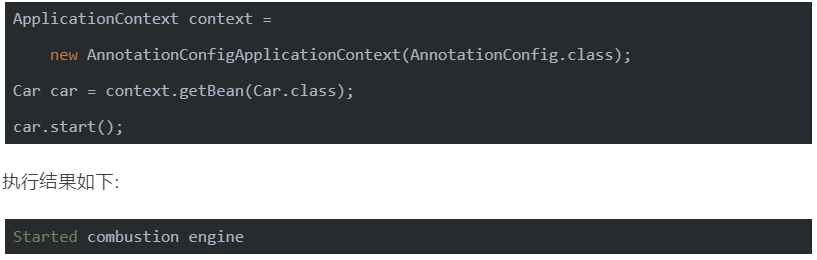
基于java的基本配置的核心，其实是下面两个注解:

1. @Configuration: 定义配置类
2. @Bean: 创建一个bean

例如，给出我们之前定义的 Car, CombustionEngine, Camshaft, 和Crankshaft 类，我们可以创建一个下面 的配置类:



接下来，我们创建一个 ApplicationContext 对象，从 ApplicationContext 对象获取一个 Car 对象，然后在创建的 Car 对象上调用 start 方法:



虽然@Configuration 和@Bean 注解的组合为 Spring 提供了足够的信息来执行依赖注入，但我们仍然需要手动手动定义每个将被注入的 bean，并显式地声明它们的依赖关系。为了减少配置 DI 框架所需的开销，Spring 提供了基于java的自动配置。

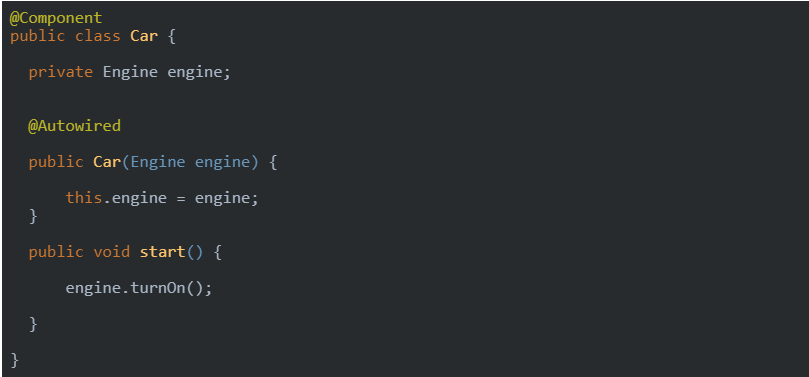
### 3.2 基于 java 的自动配置

为了支持基于 java 的自动配置，Spring 提供了额外的注解。虽然我们平时可能加过很多这种类型的注解，但是有三个最基本的注解:

1. @Component: 注册为由 Spring 管理的类
2. @Autowired: 指示 Spring 注入一个依赖对象
3. @ComponentScan: 指示Spring在何处查找带有@Component注解的类

#### 3.2.1 构造函数注入

@Autowired注解用来指导 Spring ,我们打算在使用注解的位置注入一个依赖对象。例如，在 Car 构造函数中，我们期望注入一个 Engine 对象，因此，我们给 Car 构造函数添加@Autowired注解。通过使用@Component 和@Autowired注解改造我们Car类，如下所示：

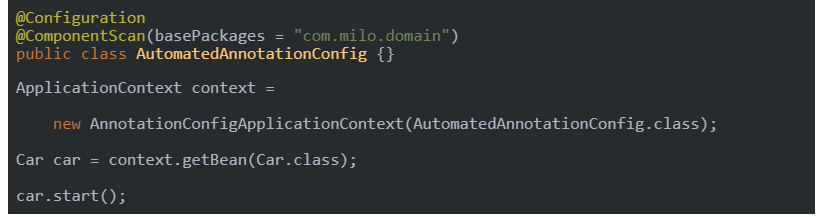


我们可以在其他类中重复这个过程:



改造完成相关类之后，我们需要创建一个@Configuration 类来指导 Spring 如何自动配置我们的应用程序。对于基于 java 的基本配置，我们明确指示 Spring 如何使用@Bean 注解创建每个 bean，但在自动配置中，我们已经通过@Component 和@Autowired 注解提供了足够的信息，说明如何创建所需的所有 bean。唯一缺少的信息是 Spring 应该在哪里寻找我们的带有@Component注解的 类,并把它注册为对应的bean。

@ Componentscan 注释包含一个参数 basePackages，它允许我们将包名称指定为一个 String，Spring 将通过递归搜索来查找@Component 类。在我们的示例中，包是 com.milo.domain，因此，我们得到的配置类是:



执行结果:



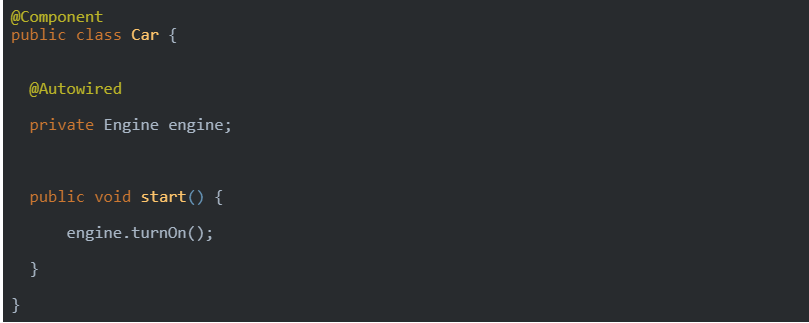
通过和基于java的基础配置比较,我们发现基于 java 的自动配置方法有两个主要优点:

1. 所需的配置要简洁得多
2. 注解直接应用于类，而不是在配置类

**所以无特殊情况,自动配置是首选**

#### 3.2.2 字段注入

除了构造函数注入，我们还可以通过字段直接注入。我们可以将@Autowired 注解应用到所需的字段来实现这一点:



这种方法极大地减少了我们的编码压力，但是它也有一个缺点,就是在使用字段之前，我们将无法检查自动注入的对象是否为空。

#### 3.2.3 Setter注入

构造函数注入的最后一种替代方法是 setter 注入，其中@Autowired 注解应用于与字段关联的 setter。例如，我们可以改变 Car 类，通过 setter 注入获得 Engine 对象，方法是用@Autowired注解 setEngine 方法:

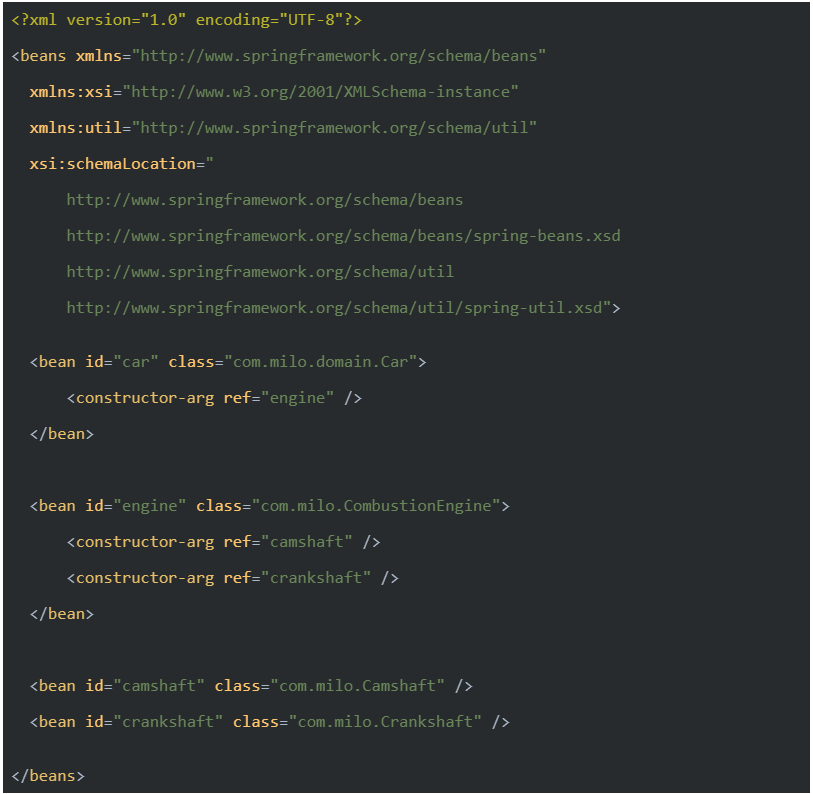


Setter 注入类似于字段注入，但它允许我们与 注入对象交互。在有些情况下，setter 注入可能特别有用，例如具有循环依赖关系，但 setter 注入可能是三种注入技术中最不常见的，**尽可能优先使用构造函数注入**。

## 四 基于 xml 的配置

另一种配置方法是基于 xml 的配置。我们在 XML 配置文件中定义 bean 以及它们之间的关系，然后指示 Spring 在哪里找到我们的配置文件。

第一步是定义 bean。我们基本遵循与基于 java 的基本配置相同的步骤，但使用 xmlbean 元素代替。在 XML 的情况下，我们还必须显式地声明我们打算使用 constructor-arg 元素注入到其他构造函数中的 bean。结合 bean 和 constructor-arg 元素，我们得到以下 XML 配置:



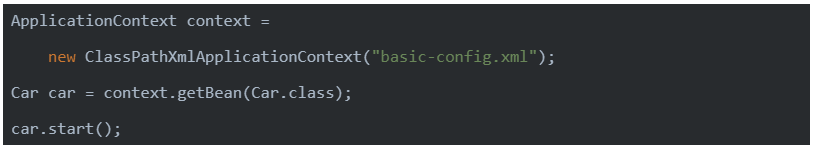
在 bean 元素中，我们必须指定两个属性:

1. id : bean 的唯一 ID (**相当于带有`@Bean` 注解方法名**)
2. class : 类的全路径(包括包名)

对于 constructor-arg 元素，我们只需要指定 ref 属性，它是对现有 bean ID 的引用。例如，元素构造函数 <constructor-arg ref="engine" /> 规定，具有 ID engine(直接定义在 car bean 之下)的 bean 应该被用作注入 car bean 构造函数的 bean。

构造函数参数的顺序由 constructor-arg 元素的顺序决定。例如，在定义 engine bean 时，传递给 CombustionEngine 构造函数的第一个构造函数参数是 camshaft bean，而第二个参数是 crankshaft bean。

获取ApplicationContext对象，我们只需修改 ApplicationContext 实现类型。因为我们将 XML 配置文件放在类路径上，所以我们使用 ClassPathXmlApplicationContext:



执行结果:



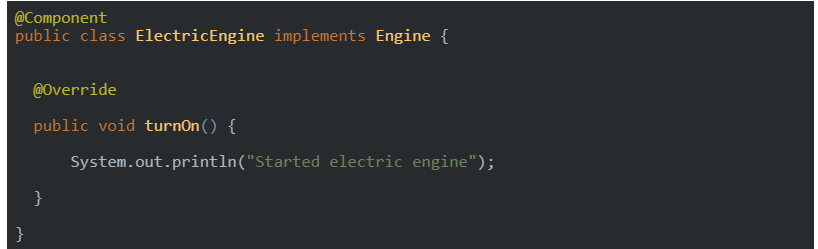
## 五 常见问题

现在，我们已经摸清了Spring框架如何进行DI，并正确地将所有依赖关系注入到我们的应用程序中，但是我们必须处理两个棘手的问题:

1. 依赖对象冲突
2. 依赖对象间存在循环依赖

### 5.1 具有多个符合条件的依赖对象

在基于 java 和基于 xml 的方法中，我们已经指示 Spring 只使用 CombustionEngine 作为我们的Engine实现。如果我们将ElectricEngine注册为符合 di 标准的部件会发生什么？为了测试结果，我们将修改基于 java 的自动配置示例，并用@Component 注解 ElectricEngine 类:



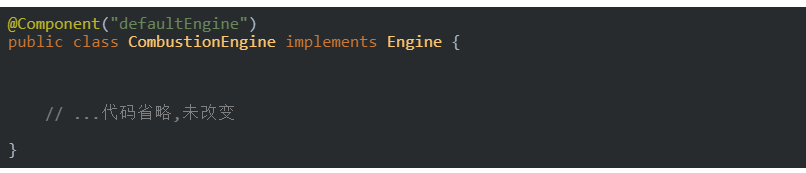
如果我们重新运行基于 java 的自动配置应用程序，我们会看到以下错误:

No qualifying bean of type 'com.dzone.albanoj2.spring.di.domain.Engine' available: expected single matching bean but found 2: combustionEngine,electricEngine

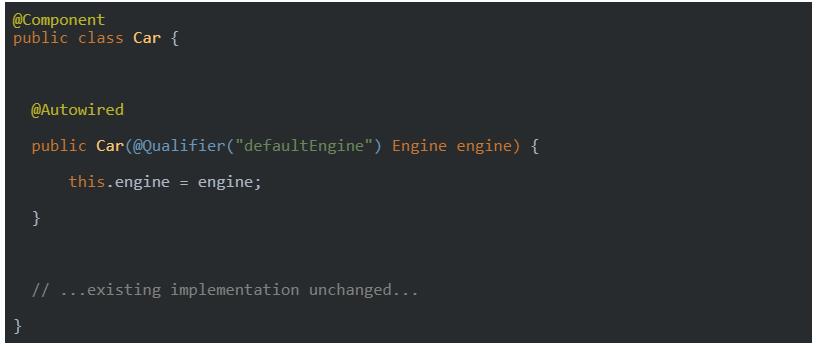
由于我们已经注释了用@Component 实现 Engine 接口的两个类ーー即 CombustionEngine 和ElectricEngine ーー spring 现在无法确定在实例化 Car 对象时应该使用这两个类中的哪一个来满足 Engine 依赖性。为了解决这个问题，我们必须明确地指示 Spring 使用这两个 bean 中的哪一个。

#### 5.1.1 @ Qualifier 注解

一种方法是给我们的依赖对象命名,并在应用@Autowired注解的地方使用@Qualifier注解来确定注入哪一个依赖对象。所以,@Qualifier 注解限定了自动注入的 bean，从而将满足需求的 bean 数量减少到一个。例如，我们可以命名我们的CombustionEngine依赖对象:



然后我们可以添加@Qualifier 注解,其名称和我们想要注入的依赖对象的名称保持一致,这样,我们Engine 对象在 Car 构造函数中被自动注入



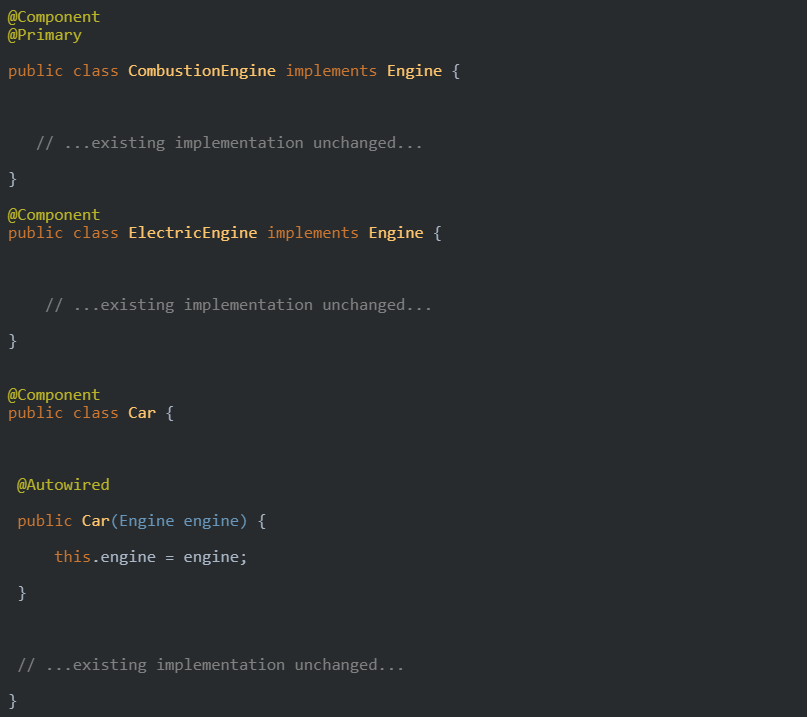
如果我们重新运行我们的应用程序，我们不再报以前的错误:



注意，如果没有显式申明bean名称的类都有一个默认名称，该默认名称就是类名首字母小写。例如，我们的 Combusttionengine 类的默认名称是 combusttionengine

#### 5.1.2 @ Primary 注解

如果我们知道默认情况下我们更喜欢一个实现，那么我们可以放弃@Qualifier 注释，直接将@Primary 注释添加到类中。例如，我们可以将我们的 Combusttionengine、 ElectricEngine 和 Car 类更改为:



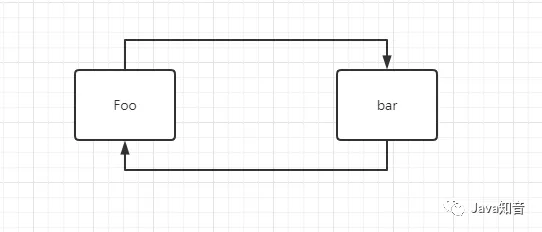
我们重新运行我们的应用程序，我们会得到以下输出:



这证明，虽然有两种可能性满足 Engine 依赖性，即 CombustionEngine 和 Electricengine，但 Spring 能够根据@Primary 注释决定两种实现中哪一种应该优先使用。

### 5.2 循环依赖

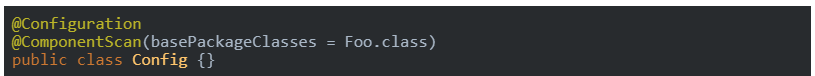
虽然我们已经深入讨论了 Spring DI 的基础知识，但是还有一个主要问题没有解决: 如果依赖关系树有一个循环引用会发生什么？例如，假设我们创建了一个 Foo 类，它的构造函数需要一个 Bar 对象，但是 Bar 构造函数需要一个 Foo 对象。



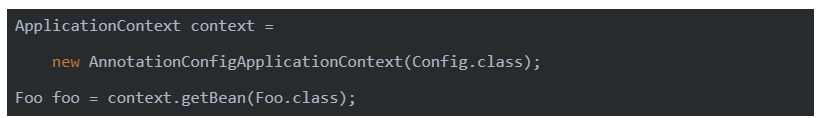
我们可以使用代码实现上面问题:



然后我们可以定义以下配置:



最后，我们可以创建我们的 ApplicationContext:



当我们执行这个代码片段时，我们看到以下错误:

Unsatisfied dependency expressed through constructor parameter 0; nested exception is org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'bar': Requested bean is currently in creation: Is there an unresolvable circular reference?

首先，Spring 尝试创建 Foo 对象。在这个过程中，Spring 认识到需要一个 Bar 对象。为了构造 Bar 对象，需要一个 Foo 对象。由于 Foo 对象目前正在构建中(这也是创建 Bar 对象的原因) ，spring 认识到可能发生了循环引用。

这个问题最简单的解决方案之一是在一个类和注入点上使用@Lazy注解。这指示 Spring 推迟带注解的 bean 和带注释的@Autowired 位置的初始化。这允许成功地初始化其中一个 bean，从而打破循环依赖链。理解了这一点，我们可以改变 Foo 和 Bar 类:



如果使用@Lazy 注解后重新运行应用程序，没有发现报告任何错误。

## 六 总结

在本文中，我们探讨了 Spring 的基础知识，包括 IoC、 DI 和 Spring ApplicationContext。然后，我们介绍了使用基于 java 的配置和基于 xml 的配置创建 Spring 应用程序的基本知识，同时研究了使用 Spring DI 时可能遇到的一些常见问题。