**1.双向适配器**

将一个类的接口适配成用户所期待的。一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

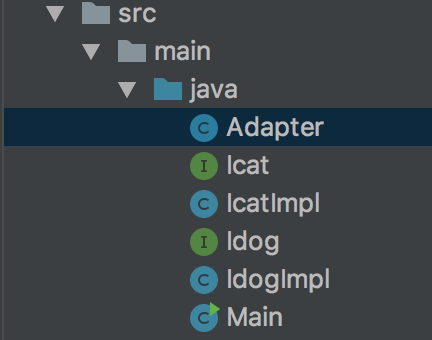
**例子：**

设计和实现一个双向适配器实例，使得猫Cat可以学狗Dog叫Cry() ，狗可以学猫抓老鼠CatchMouse() 。

**类图：**

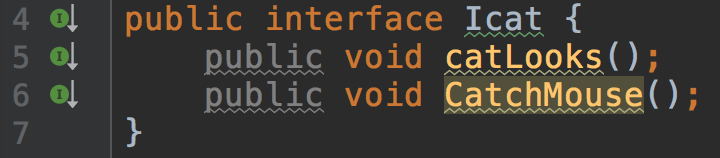


**项目结构：**

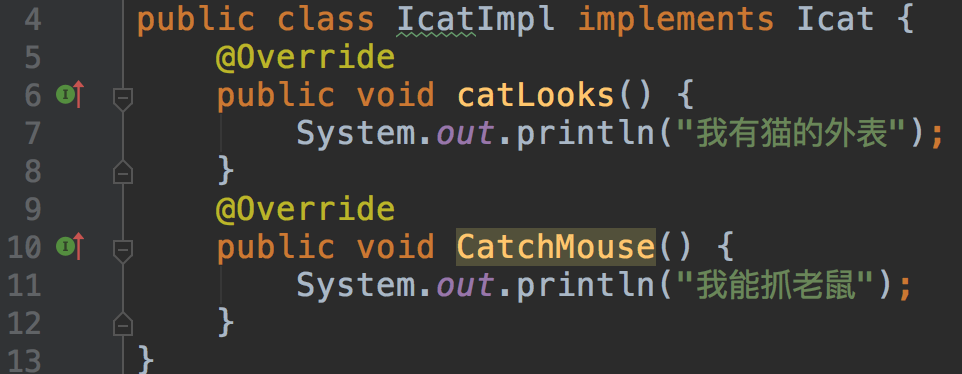


**代码：**

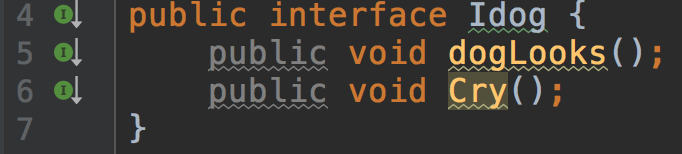
**1.Icat.java**



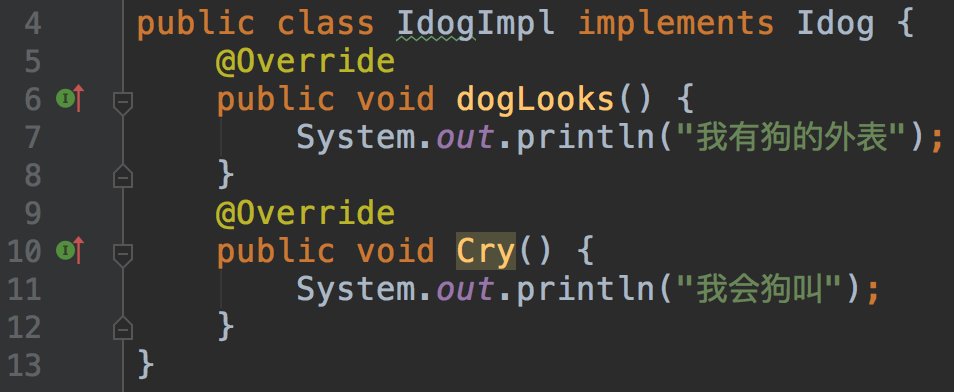
**2.IcatImpl.java**



**3.Idog.java**



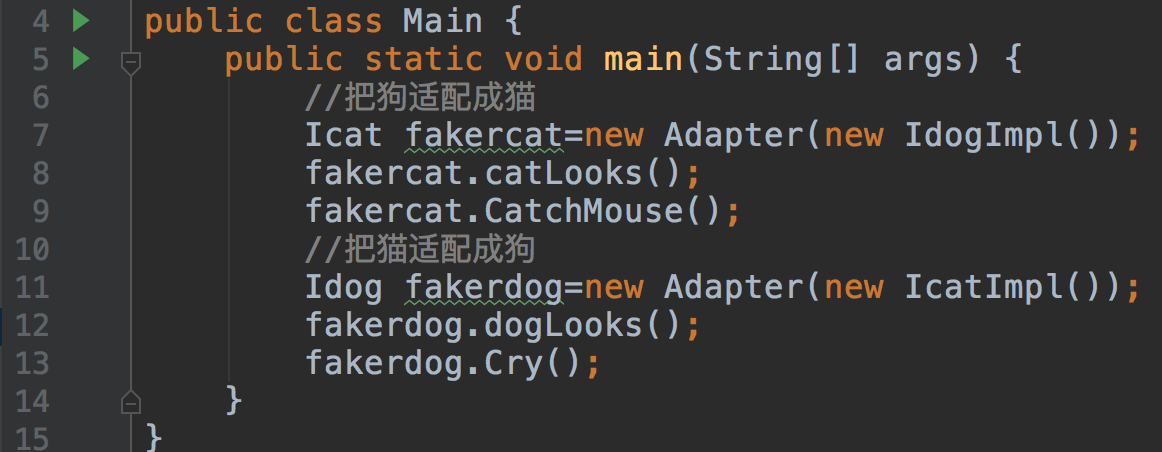
**4.IdogImpl.java**



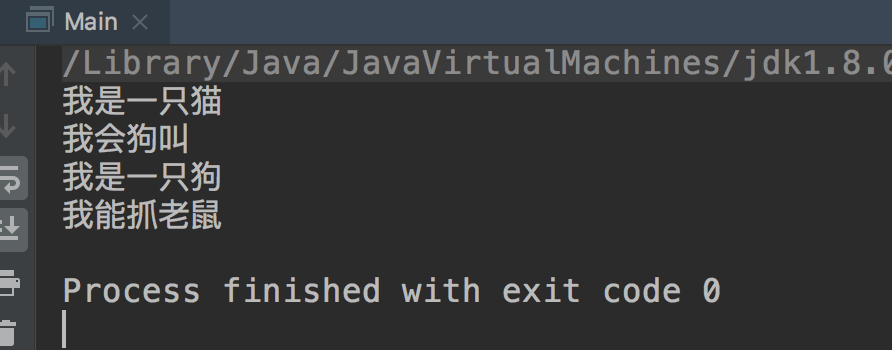
**5.Adapter.java**



**6.Main.java**



**运行结果：**



**2.DIP(Dependence Inversion Principle)、IoC(Inversion of Control)、DI(Dependency Injection)**

**概念：**

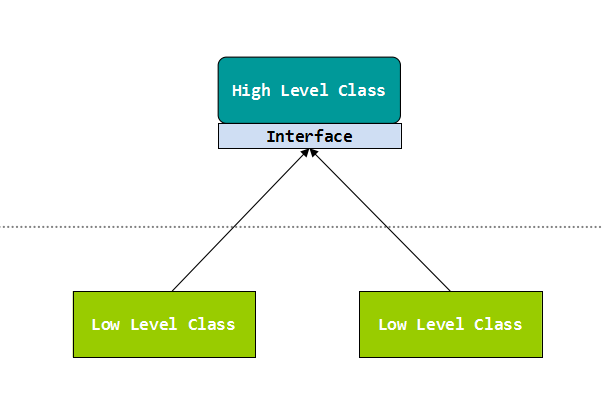
**依赖倒置原则（DIP）：一种软件架构设计的原则（抽象概念）。**

**控制反转（IoC）：一种反转流、依赖和接口的方式（DIP的具体实现方式）。**

**依赖注入（DI）：IoC的一种实现方式，用来反转依赖（IoC的具体实现方式）。**

**(1)DIP：依赖倒置原则（Dependency Inversion Principle）**

依赖倒置原则，它转换了依赖，高层模块不依赖于低层模块的实现，而低层模块依赖于高层模块定义的接口。通俗的讲，就是**高层模块定义接口，低层模块负责实现**。



在这个图中，我们发现高层模块定义了接口，将不再直接依赖于低层模块，低层模块负责实现高层模块定义的接口。这样，当有新的低层模块实现时，不需要修改高层模块的代码。

由此，我们可以总结出使用DIP的**优点：**

**系统更柔韧：**可以修改一部分代码而不影响其他模块。

**系统更健壮：**可以修改一部分代码而不会让系统崩溃。

**系统更高效：**组件松耦合，且可复用，提高开发效率。

**(2)IOC：控制反转(Inversion of Control)**

**DIP是一种软件设计原则**，它仅仅告诉你两个模块之间应该如何依赖，但是它并没有告诉如何做。

**IoC则是一种软件设计模式**，它告诉你应该如何做，来解除相互依赖模块的耦合。**控制反转（IoC），它为相互依赖的组件提供抽象，将依赖（低层模块）对象的获得交给第三方（系统）来控制，即依赖对象不在被依赖模块的类中直接通过new来获取。**

IoC有2种常见的实现方式：依赖注入和服务定位。其中，依赖注入使用最为广泛。

**(3)DI：依赖注入(Dependency Injection)**

控制反转（IoC）一种重要的方式，**就是将依赖对象的创建和绑定转移到被依赖对象类的外部来实现**。

依赖注入（DI），**它提供一种机制，将需要依赖（低层模块）对象的引用传递给被依赖（高层模块）对象**。

**1）方法一 构造函数注入**

构造函数函数注入，毫无疑问通过构造函数传递依赖。因此，构造函数的参数必然用来接收一个依赖对象。那么参数的类型是什么呢？具体依赖对象的类型？还是一个抽象类型？根据DIP原则，我们知道高层模块不应该依赖于低层模块，两者应该依赖于抽象。那么构造函数的参数应该是一个抽象类型。

**2）方法二 属性注入**

属性注入是通过属性来传递依赖。

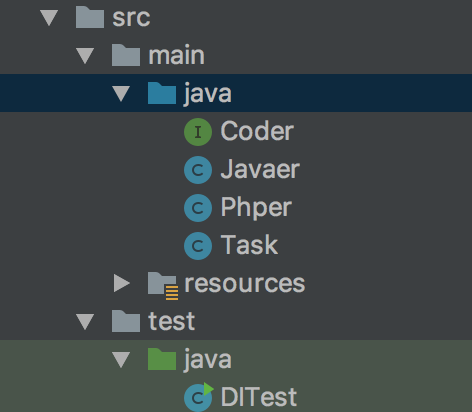
**3）方法三 接口注入**

相比构造函数注入和属性注入，接口注入显得有些复杂，使用也不常见。具体思路是先定义一个接口，包含一个设置依赖的方法。然后依赖类，继承并实现这个接口。

**例子：**

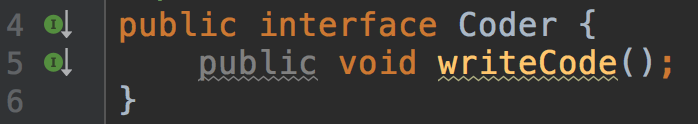
设计和实现一个DI实例，使得某项任务可以指派Java或Php的某个程序员。

**项目结构：**

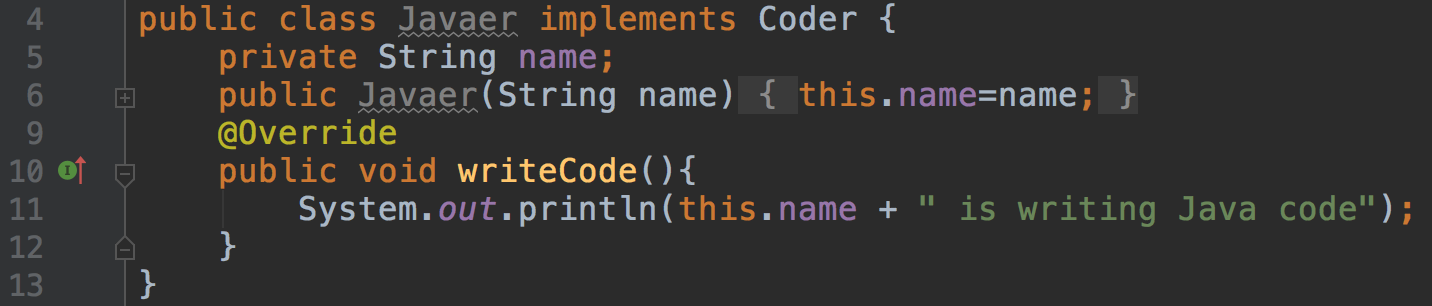


**代码：**

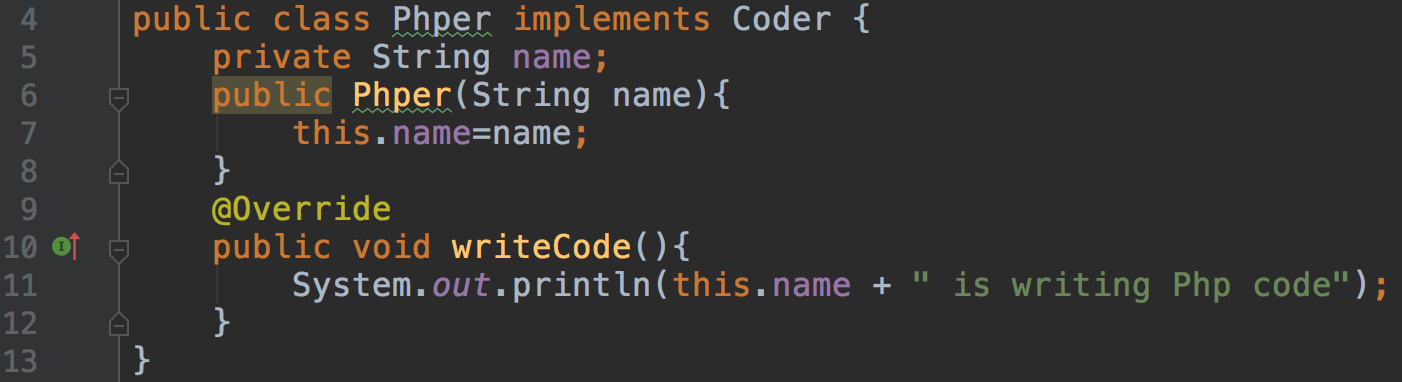
**1.Coder.java**



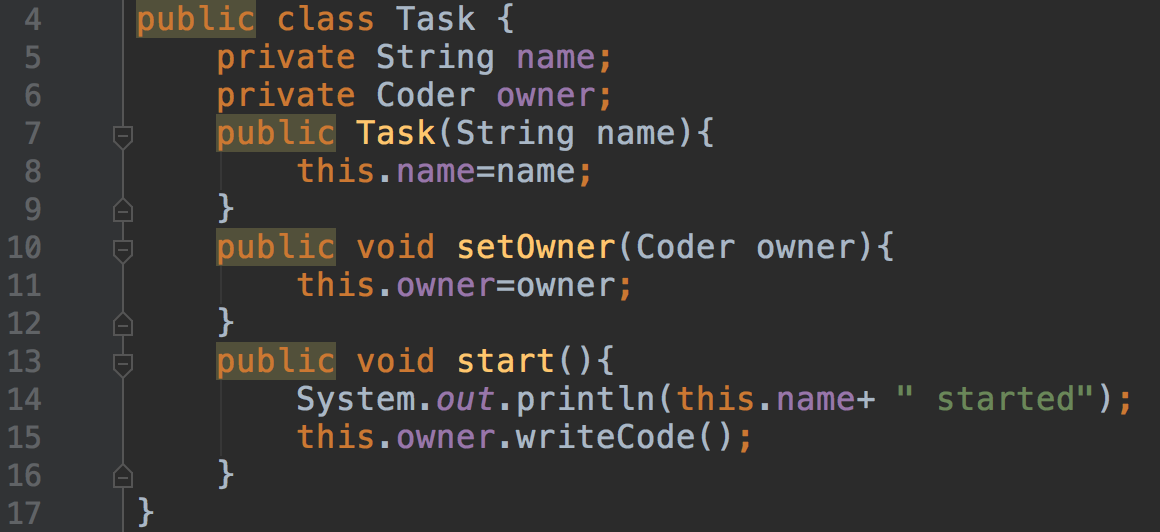
**2.Javaer.java**



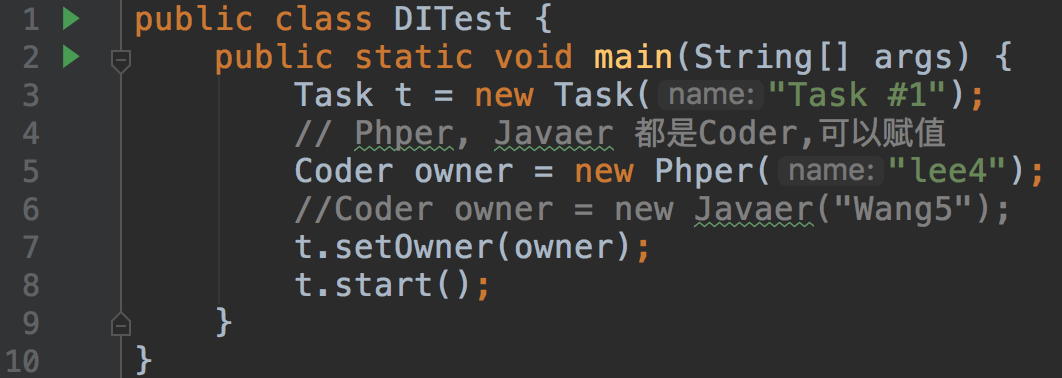
**3.Phper.java**



**4.Task.java**



**5.DITest.java**



**运行结果：**

