常用设计模式总结

目录

[常用设计模式总结 1](#_Toc29106624)

[1 创建型 3](#_Toc29106625)

[1.1 单例模式 3](#_Toc29106626)

[1.1.1 定义 3](#_Toc29106627)

[1.1.2 一般模式 3](#_Toc29106628)

[1.1.3 举例 5](#_Toc29106629)

[1.1.4 应用场景 7](#_Toc29106630)

[1.1.5 优缺点 7](#_Toc29106631)

[1.2 工厂模式-简单工厂模式 8](#_Toc29106632)

[1.2.1 定义 8](#_Toc29106633)

[1.2.2 一般模式 8](#_Toc29106634)

[1.2.3 举例 8](#_Toc29106635)

[1.2.4 应用场景 11](#_Toc29106636)

[1.2.5 优缺点 11](#_Toc29106637)

[1.3 工厂模式-工厂方法模式 12](#_Toc29106638)

[1.3.1 定义 12](#_Toc29106639)

[1.3.2 一般模式 12](#_Toc29106640)

[1.3.3 举例 13](#_Toc29106641)

[1.3.4 应用场景 15](#_Toc29106642)

[1.3.5 优缺点 15](#_Toc29106643)

[1.4 工厂模式-抽象工厂模式 16](#_Toc29106644)

[1.4.1 定义 16](#_Toc29106645)

[1.4.2 一般模式 16](#_Toc29106646)

[1.4.3 举例 17](#_Toc29106647)

[1.4.4 应用场景 22](#_Toc29106648)

[1.4.5 优缺点 23](#_Toc29106649)

[2 结构型 24](#_Toc29106650)

[2.1 代理模式 24](#_Toc29106651)

[2.1.1 定义 24](#_Toc29106652)

[2.1.2 一般模式 25](#_Toc29106653)

[2.1.3 举例 25](#_Toc29106654)

[2.1.4 应用场景 36](#_Toc29106655)

[2.1.5 优缺点 37](#_Toc29106656)

[2.2 适配器模式 37](#_Toc29106657)

[2.2.1 定义 37](#_Toc29106658)

[2.2.2 一般模式 37](#_Toc29106659)

[2.2.3 举例 38](#_Toc29106660)

[2.2.4 应用场景 44](#_Toc29106661)

[2.2.5 优缺点 44](#_Toc29106662)

[2.3 装饰器模式 45](#_Toc29106663)

[2.3.1 定义 45](#_Toc29106664)

[2.3.2 一般模式 45](#_Toc29106665)

[2.3.3 举例 46](#_Toc29106666)

[2.3.4 应用场景 50](#_Toc29106667)

[2.3.5 优缺点 50](#_Toc29106668)

[3 行为型 51](#_Toc29106669)

[3.1 策略模式 51](#_Toc29106670)

[3.1.1 定义 51](#_Toc29106671)

[3.1.2 一般模式 52](#_Toc29106672)

[3.1.3 举例 52](#_Toc29106673)

[3.1.4 应用场景 57](#_Toc29106674)

[3.1.5 优缺点 58](#_Toc29106675)

[3.2 模板方法模式 58](#_Toc29106676)

[3.2.1 定义 58](#_Toc29106677)

[3.2.2 一般模式 59](#_Toc29106678)

[3.2.3 举例 59](#_Toc29106679)

[3.2.4 应用场景 62](#_Toc29106680)

[3.2.5 优缺点 63](#_Toc29106681)

[3.3 观察者模式 63](#_Toc29106682)

[3.3.1 定义 63](#_Toc29106683)

[3.3.2 一般模式 64](#_Toc29106684)

[3.3.3 举例 64](#_Toc29106685)

[3.3.4 应用场景 68](#_Toc29106686)

[3.3.5 优缺点 68](#_Toc29106687)

[4 参考文献 69](#_Toc29106688)

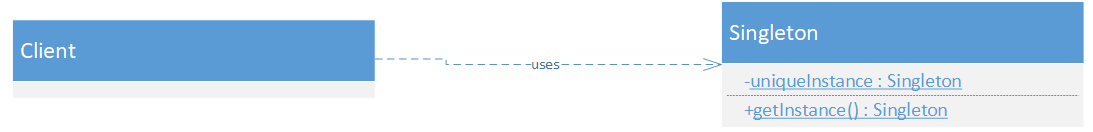
# 创建型

## 单例模式

### 定义

一个类只能有一个实例，这种模式称为单例模式。

### 一般模式



对于单例模式，一般要考虑的问题是如何实现线程安全，即多线程环境下如何保证实例只有一个，有以下三种思路：

1. 饿汉式：利用JVM来保证线程安全。

采取直接实例化uniqueInstance的方式，这样就不会产生线程不安全问题。

这种方式比较常用，但容易产生垃圾对象(丢失了延迟实例化(lazy loading)带来的节约资源的好处)。

它基于classloader机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法，但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

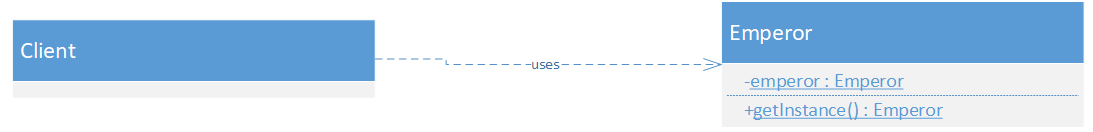
1. **public** **class** Singleton
2. {
3. // 急切地创建了uniqueInstance，不管是不是要调用getInstance，所以叫饿汉式
4. **private** **static** Singleton uniqueInstance = **new** Singleton();
6. **private** Singleton()
7. {
8. }
10. **public** **static** Singleton getInstance()
11. {
12. **return** uniqueInstance;
13. }
14. }
15. 懒汉式：利用synchronized来保证线程安全

我们可以直接在getInstance()方法上面直接加上一把synchronized同步锁,那么在一个时间点只能有一个线程能够进入该方法，从而避免了实例化多次uniqueInstance。但是当一个线程进入该方法之后，其它试图进入该方法的线程都必须等待，即使uniqueInstance已经被实例化了。这会让线程阻塞时间过长，因此该方法有性能问题。

1. // 有延迟加载: 不是在类加载的时候就创建了，而是在调用getInstance()的时候才会创建，所以叫懒汉式
2. **public** **class** Singleton
3. {
5. **private** **static** Singleton uniqueInstance;
7. **private** Singleton()
8. {
10. }
12. **public** **static** **synchronized** Singleton getInstance()
13. {
14. **if** (uniqueInstance == **null**)
15. uniqueInstance = **new** Singleton();
16. **return** uniqueInstance;
17. }
18. }
19. 双检锁式：利用锁来保证线程安全
20. **public** **class** Singleton
21. {
23. // 和饿汉模式相比，这边不需要先实例化出来
24. // 注意这里的 volatile，使用 volatile 可以禁止 JVM 的指令重排，让线程每次都是从主存中取数据而不是自己的缓存中，保证在多线程环境下也能正常运行
25. **private** **static** **volatile** Singleton uniqueInstance;
27. **private** Singleton()
28. {
29. }
31. **public** **static** Singleton getInstance()
32. {
33. // 第一次检测是避免已经有实例的情况下再去试图获得锁，提高了性能
34. **if** (uniqueInstance == **null**)
35. {
36. **synchronized** (Singleton.**class**)
37. {
38. // 第二次检测是保证线程安全
39. // 因为有可能在获得实例之前，有的线程在通过第一次检测之后被调度下CPU，等这个线程再上CPU后会获得锁，那么需要这一次检测使得它不能再获得新实例
40. **if** (uniqueInstance == **null**)
41. {
42. uniqueInstance = **new** Singleton();
43. }
44. }
45. }
46. **return** uniqueInstance;
47. }
48. }

### 举例

#### 例1 皇帝只能有一个



皇帝类的实例只能有一个，不管什么时候见皇帝。皇帝类定义如下：

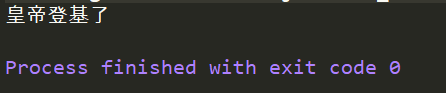
1. **package** CZ.singleton;
3. **public** **class** Emperor
4. {
6. **private** **volatile** **static** Emperor emperor;
8. **private** Emperor()
9. {
10. System.out.println("皇帝登基了");
11. }
13. **public** **static** Emperor getInstance()
14. {
15. **if** (emperor == **null**)
16. {
17. **synchronized** (Emperor.**class**)
18. {
19. **if** (emperor == **null**)
20. {
21. emperor = **new** Emperor();
22. }
23. }
24. }
25. **return** emperor;
26. }
27. }

测试类：

1. **package** CZ;
3. **import** CZ.singleton.Emperor;
5. **public** **class** Main
6. {
8. **public** **static** **void** main(String[] args)
9. {
10. **for** (**int** day = 0; day < 10; day++)
11. {
12. Runnable r = () ->
13. {
14. **final** Emperor emperorToday = Emperor.getInstance();
15. };
16. **new** Thread(r).start();
17. }
18. }
19. }

运行结果：

在不同的线程中调用了10次getInstance方法，但是只显示一条皇帝登基了，说明只有一个皇帝实例，即10天见的都是同一个皇帝。



### 应用场景

在一个系统中，要求一个类有且仅有一个对象，如果出现多个对象就会出现“不良反应”时，则可以采用单例模式，具体的场景如下：

（1）要求生成唯一序列号的环境。

（2）在整个项目中需要有访问一个共享访问点或共享数据，例如一个Web页面上的计数器，可以不用每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的。

（3）创建一个对象需要消耗的资源过多，如要访问IO、访问数据库等资源。

（4）需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为static的方式）。

### 优缺点

#### 优点

（1）由于单例模式在内存中只有一个实例，减少了内存开支，特别是一个对象需要频繁的被创建、销毁，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显。

（2）由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

（3）单例模式可以避免对资源的多重占用，例如一个写文件动作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。

（4）单例模式可以在系统设置全局的访问点，优化环共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

#### 缺点

（1）单例模式没有接口，扩展很困难，若要扩展，除了修改代码没有第二种途径可以实现。单例模式为什么不能增加接口呢？因为接口对单例模式是没有任何的意义，它要求“自行实例化”，并且提供单一实例、接口或抽象类是不可能被实例化的。

（2）单例模式对测试是不利的。在并行开发环境中，如果单例模式没有完成，是不能进行测试的，没有接口也不能使用mock的方式虚拟一个对象。

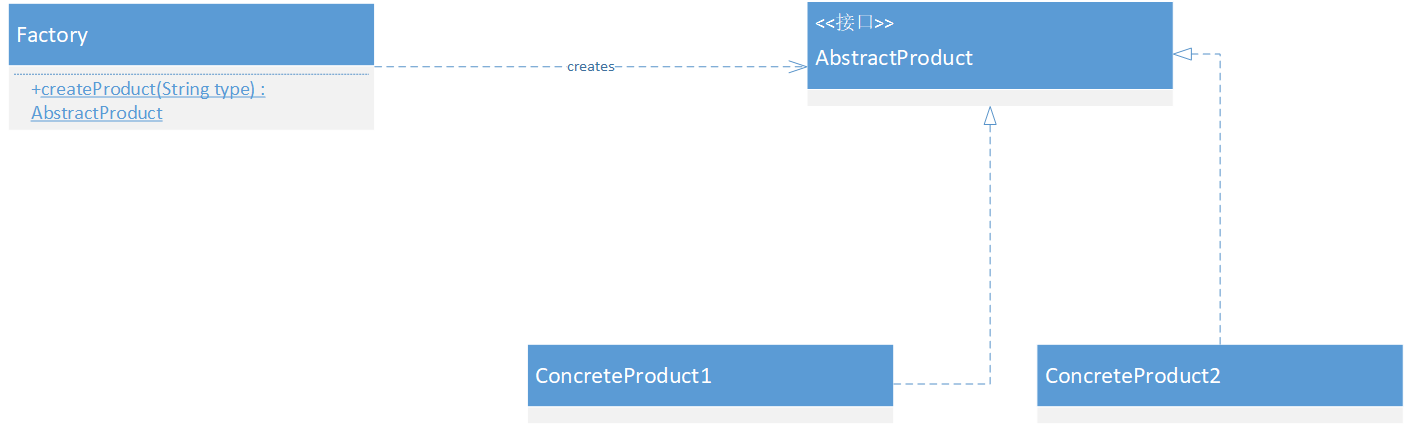
（3）单例模式与单一职责原则有冲突。一个类应该只实现一个的逻辑，而不关心它是否是单例的，决定它是不是要单例是环境决定的，单例模式把“要单例”和业务逻辑融合也在一个类中。

## 1.2 工厂模式-简单工厂模式

### 1.2.1 定义

类实例的创建由一个工厂完成。工厂调用创建方法，通过传入不同的参数来穿件不同的类实例这种模式称为简单工厂模式。简单工厂模式又称为静态工厂方法模式。

### 1.2.2 一般模式

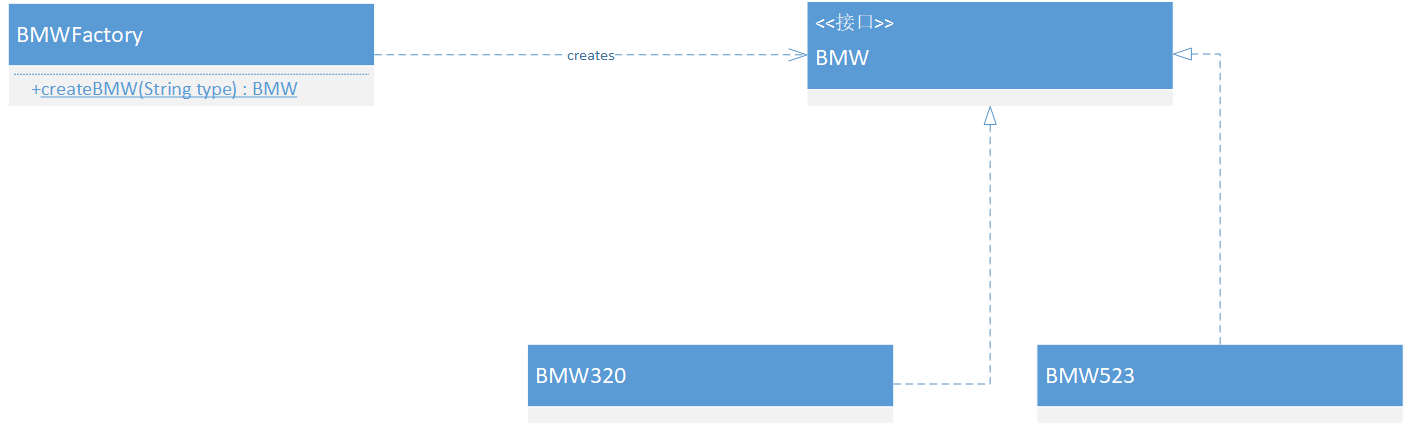


有一个工厂类，其中有个创建产品的方法。这个方法根据不同的输入参数创建不同的具体产品。

### 1.2.3 举例

#### 例1 汽车厂

宝马汽车厂生产两种不同型号的汽车，一种是BMW523，另一种是BMW320。



类的定义如下：

抽象产品类-BMW：

1. **package** com.cz.product;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 9:13
8. \*/
9. **public** **interface** BMW
10. {
11. }

具体产品类-BMW320：

1. **package** com.cz.product.concrete;
3. **import** com.cz.product.BMW;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 9:16
10. \*/
11. **public** **class** BMW320 **implements** BMW
12. {
13. **public** BMW320()
14. {
15. System.out.println("BMW320 produced");
16. }
17. }

具体产品类-BMW523：

1. **package** com.cz.product.concrete;
3. **import** com.cz.product.BMW;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 9:17
10. \*/
11. **public** **class** BMW523 **implements** BMW
12. {
13. **public** BMW523()
14. {
15. System.out.println("BMW523 produced");
16. }
17. }

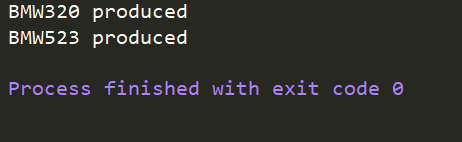
工厂类-汽车厂：

1. **package** com.cz.factory;
3. **import** com.cz.product.BMW;
4. **import** com.cz.product.concrete.BMW320;
5. **import** com.cz.product.concrete.BMW523;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 9:17
12. \*/
13. **public** **class** BMWFactory
14. {
15. **public** **static** BMW createBMW(String type)
16. {
17. **if** ("BMW320".equals(type))
18. {
19. **return** **new** BMW320();
20. }
21. **else** **if** ("BMW523".equals(type))
22. {
23. **return** **new** BMW523();
24. }
25. **return** **null**;
26. }
27. }

测试类：

1. **package** com.cz;
3. **import** com.cz.factory.BMWFactory;
4. **import** com.cz.product.BMW;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 9:12
11. \*/
12. **public** **class** Main
13. {
14. **public** **static** **void** main(String[] args)
15. {
16. **final** BMW bmw320 = BMWFactory.createBMW("BMW320");
17. **final** BMW bmw523 = BMWFactory.createBMW ("BMW523");
18. }
19. }

运行结果：



### 1.2.4 应用场景

在以下情况下可以使用简单工厂模式：

（1）工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

（2）客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

### 1.2.5 优缺点

#### 优点

（1）工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象。

（2）客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以减少使用者的记忆量。

（3）通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

#### 缺点

（1）由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。

（2）使用简单工厂模式将会增加系统中类的个数，在一定程序上增加了系统的复杂度和理解难度。

（3）系统扩展困难，一旦添加新产品就不得不修改工厂逻辑。比如例1中如果要增加一款车型就必须修改工厂类，破坏了“开闭原则”；在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。比如例1中如果要生产奔驰车，那么显然和宝马车是不同的产品类型。这样的树状产品线会让汽车厂类变得很复杂。

（4）简单工厂模式使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

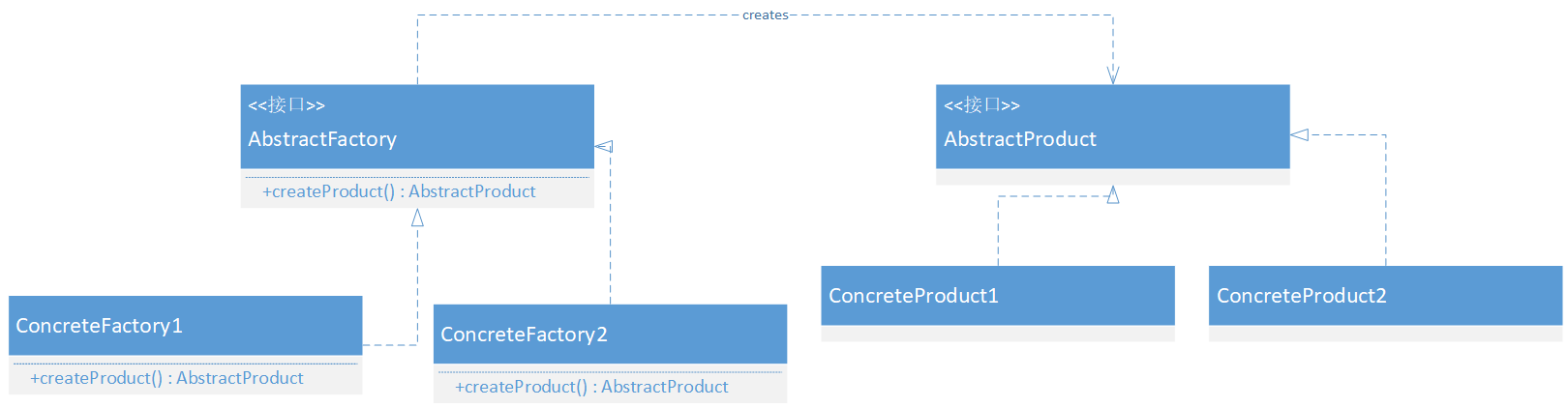
## 1.3 工厂模式-工厂方法模式

### 1.3.1 定义

定义一个用于创建对象的抽象工厂类，让具体的子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其具体的子类。

相比于简单工厂模式，工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。

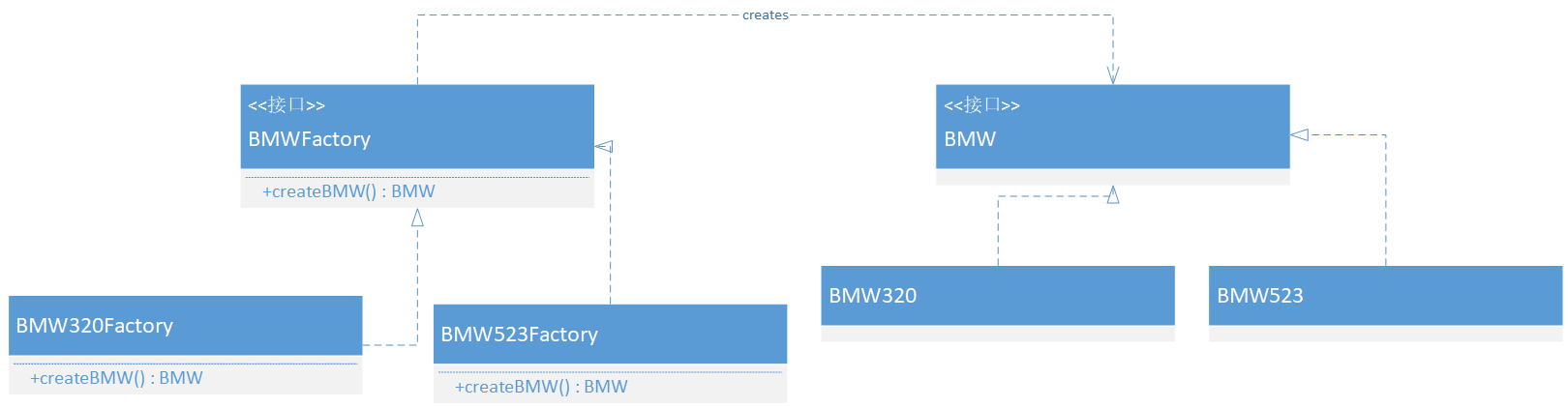
### 1.3.2 一般模式



相比较简单工厂模式，有一个抽象工厂类。然后具体的工厂子类去创建不同的产品。

### 1.3.3 举例

#### 例1 汽车厂



各个类的定义如下：

抽象产品类-BMW以及两个具体产品类和简单工厂模式相同。抽象工厂类：

1. **package** com.cz.factory;
3. **import** com.cz.product.BMW;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 10:27
10. \*/
11. **public** **interface** BMWFactory
12. {
13. BMW createBMW();
14. }

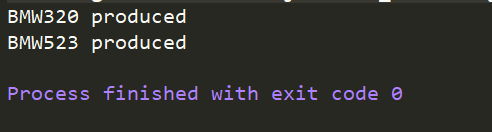
具体工厂类：

1. **package** com.cz.factory.concrete;
3. **import** com.cz.factory.BMWFactory;
4. **import** com.cz.product.BMW;
5. **import** com.cz.product.concrete.BMW320;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 10:27
12. \*/
13. **public** **class** BMW320Factory **implements** BMWFactory
14. {
15. **public** BMW createBMW()
16. {
17. **return** **new** BMW320();
18. }
19. }
20. **package** com.cz.factory.concrete;
22. **import** com.cz.factory.BMWFactory;
23. **import** com.cz.product.BMW;
24. **import** com.cz.product.concrete.BMW523;
26. /\*\*
27. \* @author CHANG Zheng
28. \* @version 1.0.0
29. \* @projectName design-patterns
30. \* @date 2020/1/4 10:28
31. \*/
32. **public** **class** BMW523Factory **implements** BMWFactory
33. {
34. **public** BMW createBMW()
35. {
36. **return** **new** BMW523();
37. }
38. }

测试类：

1. **package** com.cz;
3. **import** com.cz.factory.BMWFactory;
4. **import** com.cz.factory.concrete.BMW320Factory;
5. **import** com.cz.factory.concrete.BMW523Factory;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 10:28
12. \*/
13. **public** **class** Main
14. {
15. **public** **static** **void** main(String[] args)
16. {
17. **final** BMWFactory bmw320Factory = **new** BMW320Factory();
18. **final** BMWFactory bmw523Factory = **new** BMW523Factory();
19. bmw320Factory.createBMW();
20. bmw523Factory.createBMW();
21. }
22. }

运行结果：



### 1.3.4 应用场景

在以下使用环境下，考虑采用工厂方法模式：

（1）一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。

（2）一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

（3）将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

### 1.3.5 优缺点

#### 优点

（1）在工厂方法模式中，工厂方法用来创建客户端所需要的产品，同时还向客户端隐藏了哪种具体产品类将被实例化这一细节，客户端只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。

（2）基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。

（3）使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。

#### 缺点

（1）在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

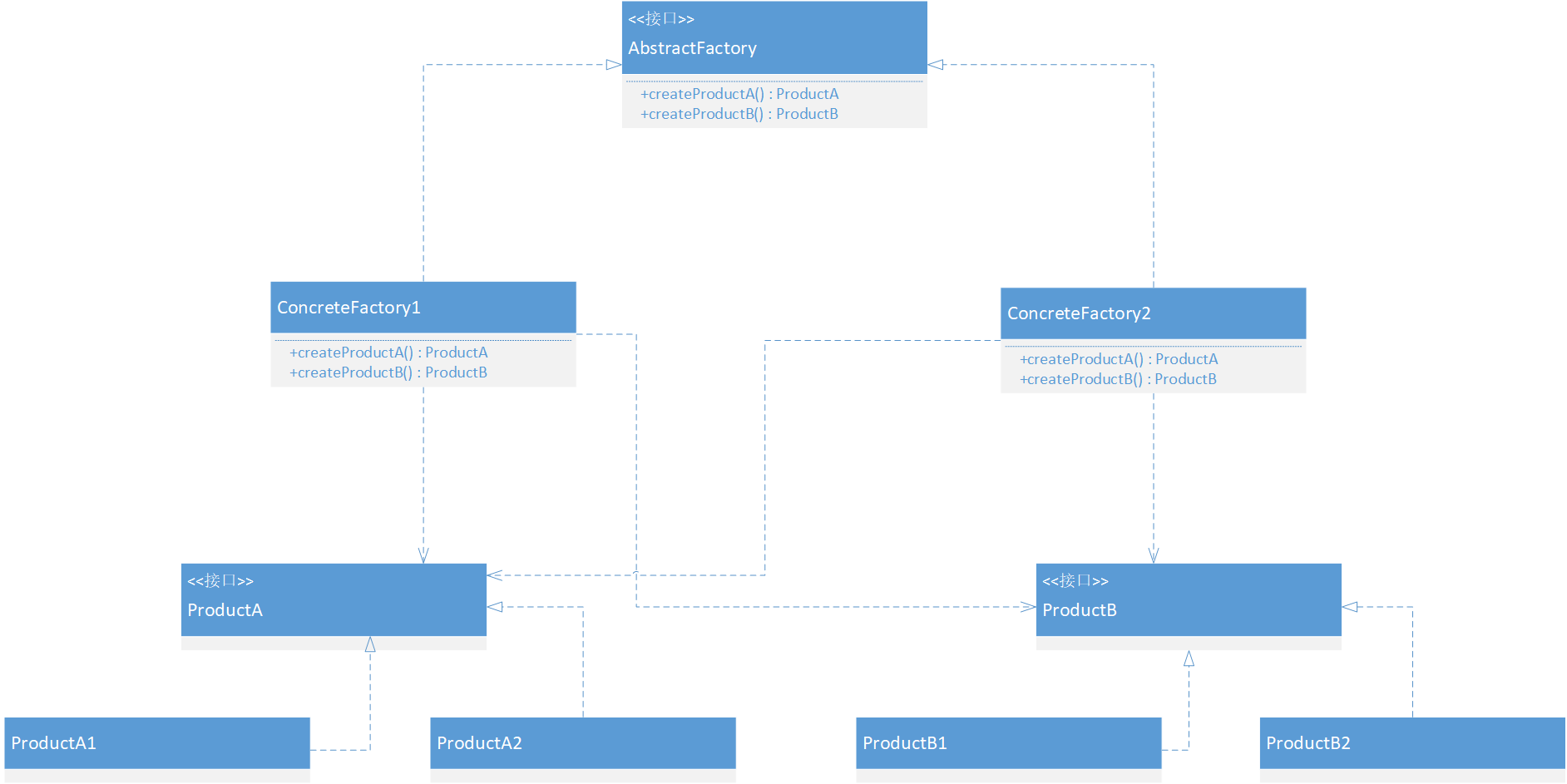
（2）由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

## 1.4 工厂模式-抽象工厂模式

### 1.4.1 定义

相比较工厂方法模式，抽象工厂模式将产品也抽象化。它可以说是是工厂方法模式的升级版，当需要创建的产品有多个产品线（产品族）时使用抽象工厂模式是比较好的选择。

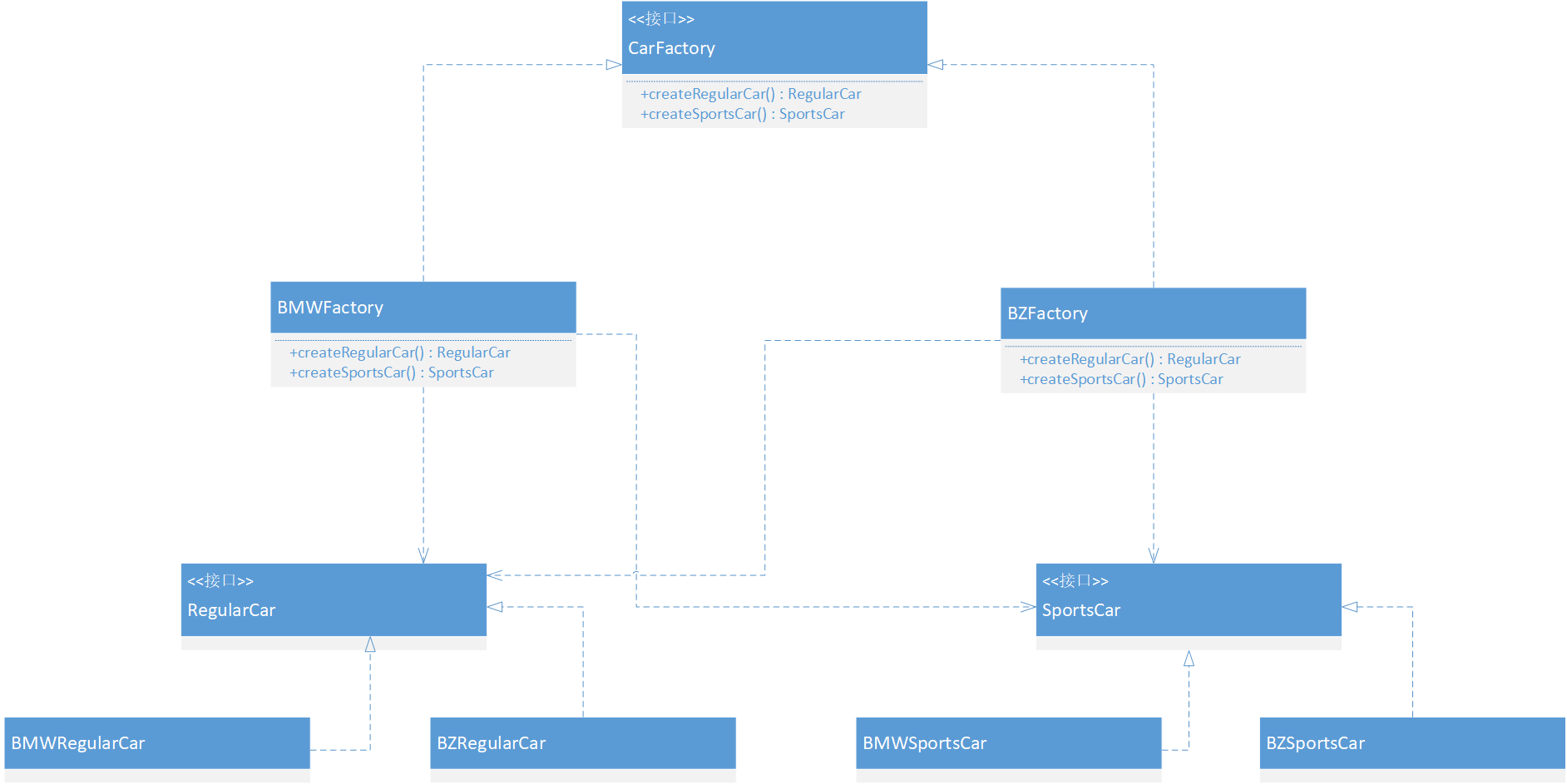
### 1.4.2 一般模式



### 1.4.3 举例

#### 例1 汽车厂

假设现在有两个汽车厂，奔驰厂和宝马厂，都可以生产两种类型的汽车，分别是普通车和跑车。即系统中有两个产品等级结构：普通车和跑车；两个产品族：奔驰和宝马。



各个类的定义如下：

抽象产品类：

1. **package** com.cz.product;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 11:39
8. \*/
9. **public** **interface** RegularCar
10. {
11. }
12. **package** com.cz.product;
14. /\*\*
15. \* @author CHANG Zheng
16. \* @version 1.0.0
17. \* @projectName design-patterns
18. \* @date 2020/1/4 11:39
19. \*/
20. **public** **interface** SportsCar
21. {
22. }

具体产品类：

1. **package** com.cz.product.concrete;
3. **import** com.cz.product.RegularCar;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 11:40
10. \*/
11. **public** **class** BMWRegularCar **implements** RegularCar
12. {
13. **public** BMWRegularCar()
14. {
15. System.out.println("BMWRegularCar produced");
16. }
17. }
18. **package** com.cz.product.concrete;
20. **import** com.cz.product.SportsCar;
22. /\*\*
23. \* @author CHANG Zheng
24. \* @version 1.0.0
25. \* @projectName design-patterns
26. \* @date 2020/1/4 11:41
27. \*/
28. **public** **class** BMWSportsCar **implements** SportsCar
29. {
30. **public** BMWSportsCar()
31. {
32. System.out.println("BMWSportsCar produced");
33. }
34. }
35. **package** com.cz.product.concrete;
37. **import** com.cz.product.RegularCar;
39. /\*\*
40. \* @author CHANG Zheng
41. \* @version 1.0.0
42. \* @projectName design-patterns
43. \* @date 2020/1/4 11:41
44. \*/
45. **public** **class** BZRegularCar **implements** RegularCar
46. {
47. **public** BZRegularCar()
48. {
49. System.out.println("BZRegularCar produced");
50. }
51. }
52. **package** com.cz.product.concrete;
54. **import** com.cz.product.SportsCar;
56. /\*\*
57. \* @author CHANG Zheng
58. \* @version 1.0.0
59. \* @projectName design-patterns
60. \* @date 2020/1/4 11:42
61. \*/
62. **public** **class** BZSportsCar **implements** SportsCar
63. {
64. **public** BZSportsCar()
65. {
66. System.out.println("BZSportsCar produced");
67. }
68. }

抽象工厂类：

1. **package** com.cz.factory;
3. **import** com.cz.product.RegularCar;
4. **import** com.cz.product.SportsCar;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 11:38
11. \*/
12. **public** **interface** CarFactory
13. {
14. RegularCar createRegularCar();
16. SportsCar createSportsCar();
17. }

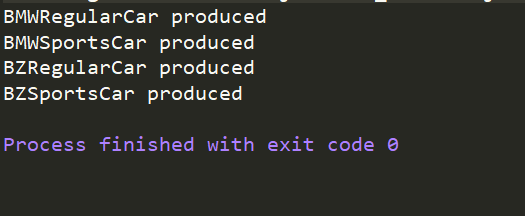
具体工厂类：

1. **package** com.cz.factory.concrete;
3. **import** com.cz.factory.CarFactory;
4. **import** com.cz.product.RegularCar;
5. **import** com.cz.product.SportsCar;
6. **import** com.cz.product.concrete.BMWRegularCar;
7. **import** com.cz.product.concrete.BMWSportsCar;
9. /\*\*
10. \* @author CHANG Zheng
11. \* @version 1.0.0
12. \* @projectName design-patterns
13. \* @date 2020/1/4 11:43
14. \*/
15. **public** **class** BMWFactory **implements** CarFactory
16. {
17. **public** RegularCar createRegularCar()
18. {
19. **return** **new** BMWRegularCar();
20. }
22. **public** SportsCar createSportsCar()
23. {
24. **return** **new** BMWSportsCar();
25. }
26. }
27. **package** com.cz.factory.concrete;
29. **import** com.cz.factory.CarFactory;
30. **import** com.cz.product.RegularCar;
31. **import** com.cz.product.SportsCar;
32. **import** com.cz.product.concrete.BZRegularCar;
33. **import** com.cz.product.concrete.BZSportsCar;
35. /\*\*
36. \* @author CHANG Zheng
37. \* @version 1.0.0
38. \* @projectName design-patterns
39. \* @date 2020/1/4 11:44
40. \*/
41. **public** **class** BZFactory **implements** CarFactory
42. {
43. **public** RegularCar createRegularCar()
44. {
45. **return** **new** BZRegularCar();
46. }
48. **public** SportsCar createSportsCar()
49. {
50. **return** **new** BZSportsCar();
51. }
52. }

测试类：

1. **package** com.cz;
3. **import** com.cz.factory.CarFactory;
4. **import** com.cz.factory.concrete.BMWFactory;
5. **import** com.cz.factory.concrete.BZFactory;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 11:38
12. \*/
13. **public** **class** Main
14. {
15. **public** **static** **void** main(String[] args)
16. {
17. **final** CarFactory bmwFactory = **new** BMWFactory();
18. **final** CarFactory bzFactory = **new** BZFactory();
20. bmwFactory.createRegularCar();
21. bmwFactory.createSportsCar();
23. bzFactory.createRegularCar();
24. bzFactory.createSportsCar();
25. }
26. }

运行结果：



### 1.4.4 应用场景

在以下情况下可以考虑使用抽象工厂模式：

（1）一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是很重要的，用户无须关心对象的创建过程，将对象的创建和使用解耦。

（2）系统中有多于一个的产品族，而每次只使用其中某一产品族。可以通过配置文件等方式来使得用户可以动态改变产品族，也可以很方便地增加新的产品族。

（3）属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。同一个产品族中的产品可以是没有任何关系的对象，但是它们都具有一些共同的约束，如同一制作水果蛋糕用的水果——草莓和芒果，草莓和芒果之间没有直接关系，但它们都是属于水果。

（4）产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

### 1.4.5 优缺点

#### 优点

（1）抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。

（2）当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。这对一些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。

（3）增加新的具体工厂和产品族很方便，比如例1中再增加一个大众车厂，然后增加大众车的产品族，无须修改已有系统，符合“开闭原则”。

#### 缺点

（1）在添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品，这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便。比如例1中要不同的产品族中增加老爷车这种汽车产品等级结构的话就必须修改工厂了。

（2）开闭原则的倾斜性

(a) 增加产品族：对于增加新的产品族，抽象工厂模式很好地支持了“开闭原则”，只需要增加具体产品并对应增加一个新的具体工厂，对已有代码无须做任何修改。

(b) 增加新的产品等级结构：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，违背了“开闭原则”。

正因为抽象工厂模式存在“开闭原则”的倾斜性，它以一种倾斜的方式来满足“开闭原则”，为增加新产品族提供方便，但不能为增加新产品结构提供这样的方便，因此要求设计人员在设计之初就能够全面考虑，不会在设计完成之后向系统中增加新的产品等级结构，也不会删除已有的产品等级结构，否则将会导致系统出现较大的修改，为后续维护工作带来诸多麻烦。

# 2 结构型

## 2.1 代理模式

### 2.1.1 定义

代理模式为其它对象提供一种代理以控制对这个对象的访问。它提供了对目标对象额外的访问方式，即通过代理对象访问目标对象，这样可以在不修改原目标对象的前提下，提供额外的功能操作，扩展目标对象的功能。代理对象在客户端和目标对象之间起到中介的作用。

代理模式也叫做委托模式，它是一项基本设计技巧，许多其它的模式，如状态模式、策略模式、访问者模式本质上是在更特殊的场合采用了委托模式，而且在日常的应用中，代理模式可以提供非常好的访问控制。

我们有多种不同的方式来实现代理。如果按照代理创建的时期来进行分类的话，可以分为两种：静态代理、动态代理。静态代理是由程序员创建或特定工具自动生成源代码，在对其编译。静态代理中，每个代理类只能为一个接口服务，这样程序开发中必然会产生许多的代理类。所以我们想办法希望通过一个代理类完成全部的代理功能，那么我们就需要用动态代理。动态代理是在运行时，通过反射机制实现动态代理，并且能够代理各种类型的对象。

代理模式 vs 装饰器模式：

从类图中我们看到代理模式和装饰器模式非常相似，都实现了接口，可以对对象进行包装。他们的区别在于：

（1）代理模式，注重对对象某一功能的流程把控和辅助。代理包装对象是为了控制对对象的访问。它可以控制对象做某些事，重心是为了借用对象的功能完成某一流程，而非对象功能如何。

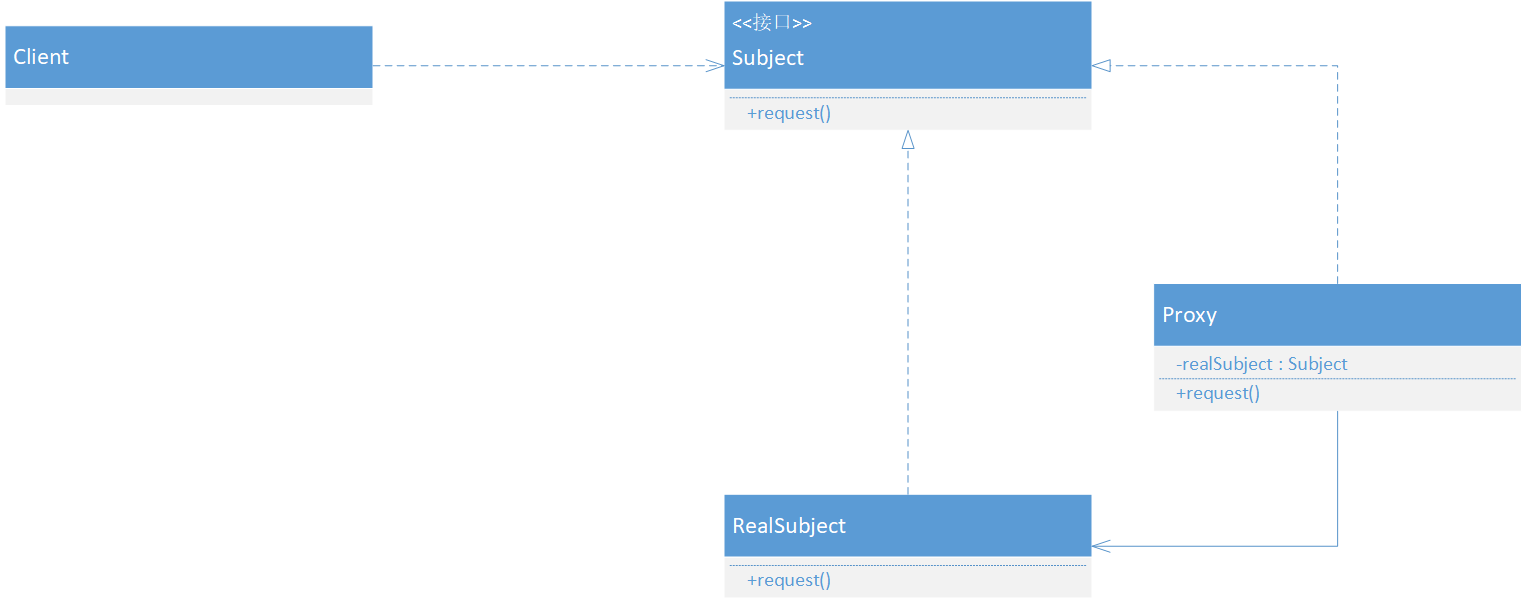
（2）装饰模式，注重对对象功能的扩展，装饰者是为了给对象添加装饰。它不关心外界如何调用，只注重对对象功能的加强，装饰后还是对象本身。

（3）对于代理类，如何调用对象的某一功能是思考重点，而不需要兼顾对象的所有功能；

（4）对于装饰类，如何扩展对象的某一功能是思考重点，同时也需要兼顾对象的其它功能，因为再怎么装饰，本质也是对象本身，要担负起对象应有的职责。

例2将体现出这两种设计模式的区别。

### 2.1.2 一般模式



（1）Subject：抽象主题角色

抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。

（2）RealSubject：真实主题角色

也叫做被委托角色、被代理角色，它才是冤大头，是业务逻辑的具体执行者。

1. Proxy：代理主题角色

也叫做委托类、代理类，它负责对真实角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题处理完毕前后做预处理和善后处理工作。

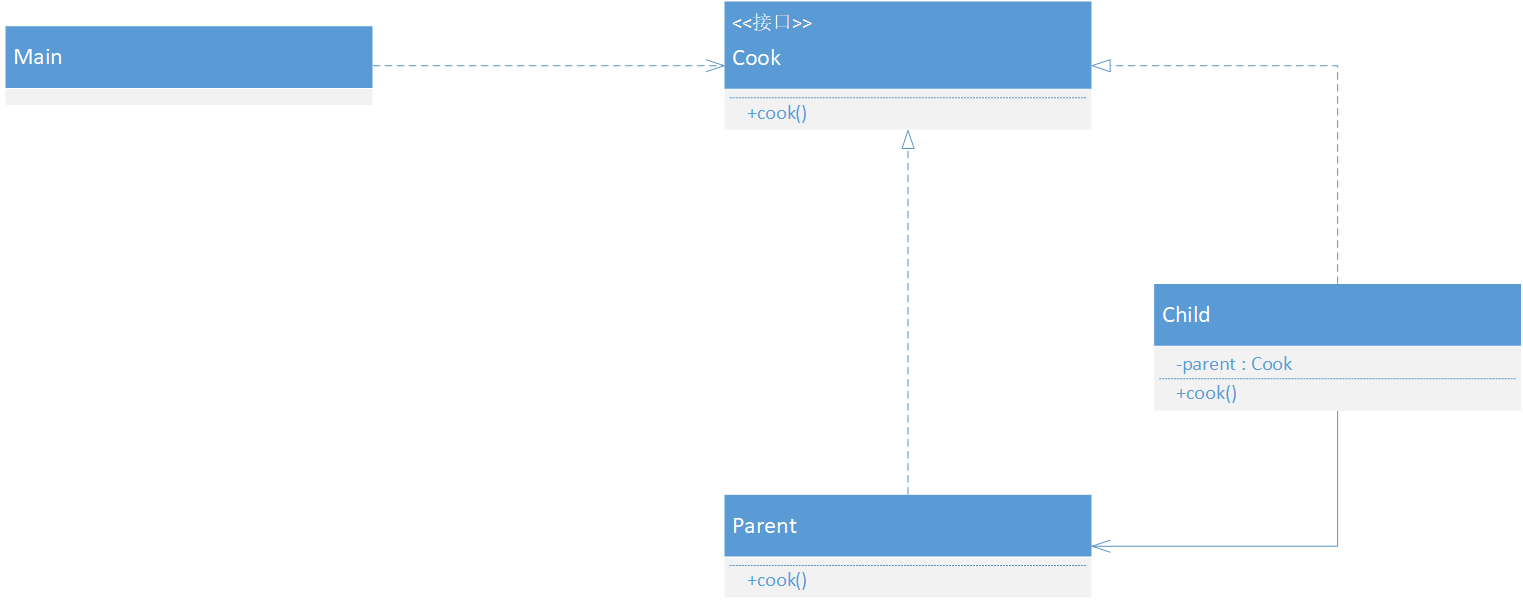
1. Client 客户端

使用代理类和主题接口完成一些工作。

### 2.1.3 举例

#### 例1 孩子做饭-静态代理

学校布置了一项家庭作业，让每个学生回家做一顿饭。但是实际上做饭的还是父母，孩子打酱油。即孩子代理了父母去做饭。



各个类的定义如下：

主题接口——厨师：

1. **package** com.cz.subject;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 20:33
8. \*/
9. **public** **interface** Cook
10. {
11. **void** cook();
12. }

真是主题类——作为厨师的父母：

1. **package** com.cz.subject.concrete;
3. **import** com.cz.subject.Cook;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 20:33
10. \*/
11. **public** **class** Parent **implements** Cook
12. {
13. @Override
14. **public** **void** cook()
15. {
16. System.out.println("父母做饭");
17. }
18. }

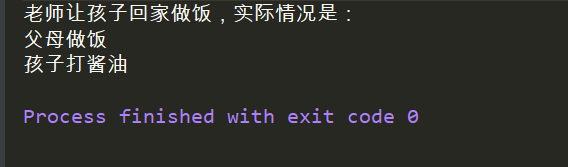
代理类——孩子：

1. **package** com.cz.proxy;
3. **import** com.cz.subject.Cook;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 20:34
10. \*/
11. **public** **class** Child **implements** Cook
12. {
13. **private** Cook parent;
15. **public** Child(Cook parent)
16. {
17. **this**.parent = parent;
18. }
20. @Override
21. **public** **void** cook()
22. {
23. parent.cook();
24. System.out.println("孩子打酱油");
25. }
26. }

客户端——老师，即测试类：

1. **package** com.cz;
3. **import** com.cz.proxy.Child;
4. **import** com.cz.subject.Cook;
5. **import** com.cz.subject.concrete.Parent;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 20:32
12. \*/
13. **public** **class** Main
14. {
15. **public** **static** **void** main(String[] args)
16. {
17. System.out.println("老师让孩子回家做饭，实际情况是：");
19. Cook child = **new** Child(**new** Parent());
20. child.cook();
21. }
22. }

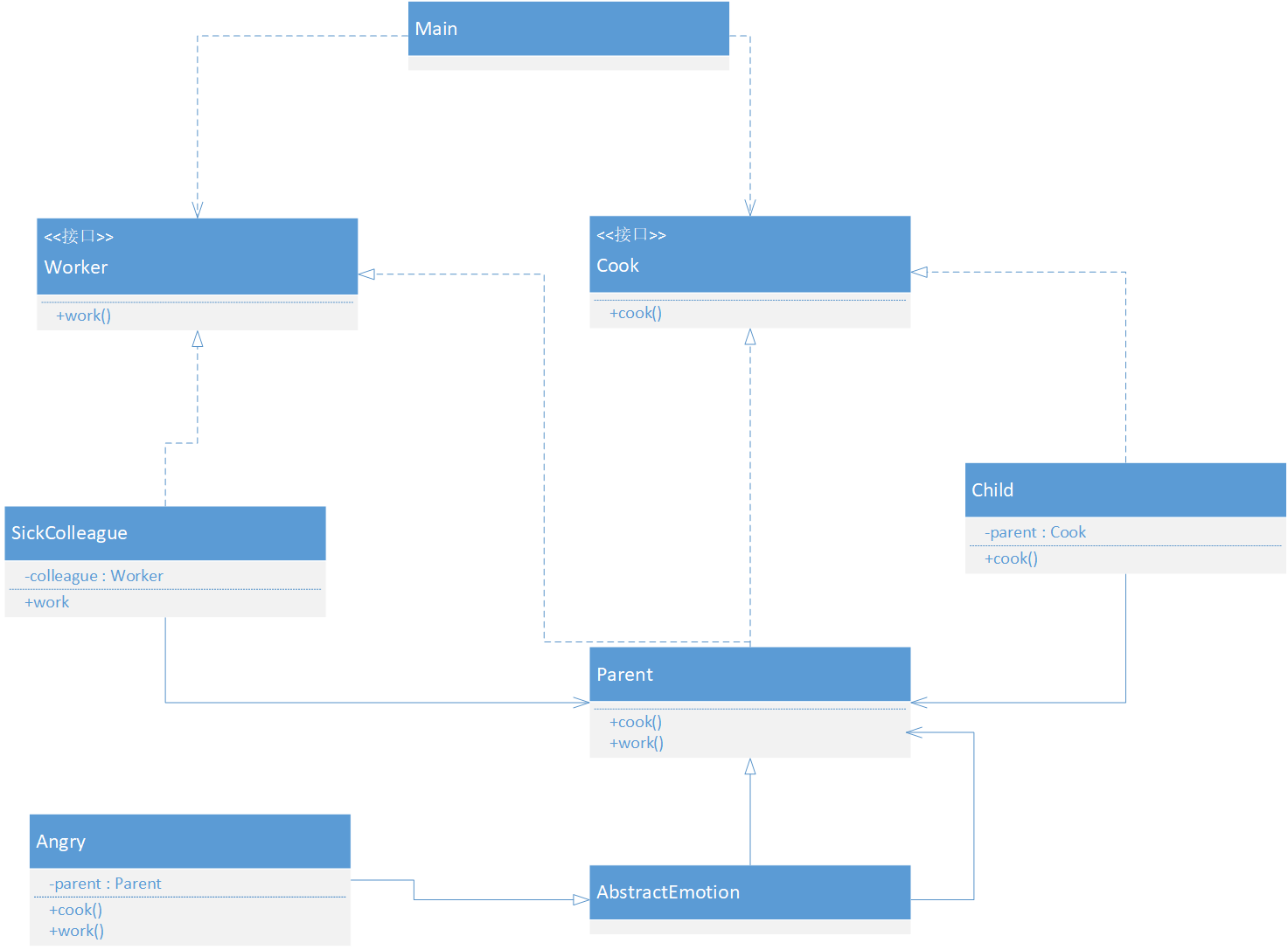
运行结果：



#### 例2 忙碌的父母-静态代理+装饰器

如果父母不仅要帮孩子做饭，还要帮请假的同事工作。那么显然孩子不能代理父母去工作。对于工作这一个主题，代理是请假的同事，客户端是老板，因为老板不管真实情况是谁在工作，只管找这个同事安排任务即可。

另外，父母很忙碌就会产生情绪。这里我们再用情绪装饰器装饰父母。



对于这种情况，各个类的定义如下：

主题接口——厨师，和例1一样。主题接口——工作者：

1. package com.cz.subject;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 22:07
8. \*/
9. **public** interface Worker
10. {
11. **void** work();
12. }

实际执行者——父母：

1. package com.cz.subject.concrete;
3. import com.cz.subject.Cook;
4. import com.cz.subject.Worker;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 20:33
11. \*/
12. **public** **class** Parent implements Cook, Worker
13. {
14. @Override
15. **public** **void** cook()
16. {
17. System.out.println("父母做饭");
18. }
20. @Override
21. **public** **void** work()
22. {
23. System.out.println("父母帮同事工作");
24. }
25. }

做饭代理类——孩子：

1. package com.cz.proxy;
3. import com.cz.subject.Cook;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 22:07
10. \*/
11. **public** **class** Child implements Cook
12. {
13. **private** Cook parent;
15. **public** Child(Cook parent)
16. {
17. **this**.parent = parent;
18. }
20. @Override
21. **public** **void** cook()
22. {
23. parent.cook();
24. System.out.println("孩子打酱油");
25. }
26. }

工作代理——生病的同事：

1. package com.cz.proxy;
3. import com.cz.subject.Worker;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 22:07
10. \*/
11. **public** **class** SickColleague implements Worker
12. {
13. **private** Worker colleague;
15. **public** SickColleague(Worker colleague)
16. {
17. **this**.colleague = colleague;
18. }
20. @Override
21. **public** **void** work()
22. {
23. colleague.work();
24. System.out.println("同事请假养病");
25. }
26. }

抽象装饰器——由于做饭和工作而产生的情绪：

1. package com.cz.decorator;
3. import com.cz.subject.concrete.Parent;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 22:11
10. \*/
11. **public** abstract **class** AbstractEmotion extends Parent
12. {
13. }

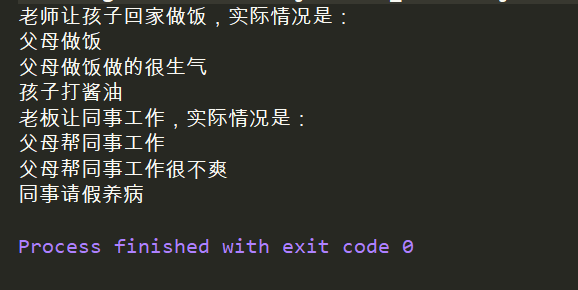
具体装饰器：生气的

1. package com.cz.decorator.concrete;
3. import com.cz.decorator.AbstractEmotion;
4. import com.cz.subject.concrete.Parent;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 22:12
11. \*/
12. **public** **class** Angry extends AbstractEmotion
13. {
14. **private** Parent parent;
16. **public** Angry(Parent parent)
17. {
18. **this**.parent = parent;
19. }
21. @Override
22. **public** **void** cook()
23. {
24. parent.cook();
25. System.out.println("父母做饭做的很生气");
26. }
28. @Override
29. **public** **void** work()
30. {
31. parent.work();
32. System.out.println("父母帮同事工作很不爽");
34. }
35. }

客户端（老师、老板）测试类：

1. package com.cz;
3. import com.cz.decorator.concrete.Angry;
4. import com.cz.proxy.Child;
5. import com.cz.proxy.SickColleague;
6. import com.cz.subject.Cook;
7. import com.cz.subject.Worker;
8. import com.cz.subject.concrete.Parent;
10. /\*\*
11. \* @author CHANG Zheng
12. \* @version 1.0.0
13. \* @projectName design-patterns
14. \* @date 2020/1/4 22:05
15. \*/
16. **public** **class** Main
17. {
18. **public** **static** **void** main(String[] args)
19. {
20. Parent parent = **new** Parent();
21. Angry angryParent = **new** Angry(parent);
23. System.out.println("老师让孩子回家做饭，实际情况是：");
24. Cook cookingchild = **new** Child(angryParent);
25. cookingchild.cook();
27. System.out.println("老板让同事工作，实际情况是：");
28. Worker workingColleague = **new** SickColleague(angryParent);
29. workingColleague.work();
30. }
31. }

运行结果：



#### 例3 孩子做饭-动态代理

老师不仅要求孩子回家要做饭，还要洗碗。那么就需要新的抽象的Subject和具体的Subject，同时还需要再写一个的代理类（代理类变成做饭的孩子和洗碗的孩子两个）。如果要代理的对象一多，显然很琐碎。于是我们采用动态代理，省去写新的代理类的步骤。

动态代理类（孩子）不用我们自己去写，由JVM去生成。具体类的定义如下：

做饭Subject，即Cook没有变。新增一个抽象主题，即洗碗类接口：

1. **package** com.cz.subject;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 21:01
8. \*/
9. **public** **interface** Washer
10. {
11. **void** wash();
12. }

我们把父母（真实主题类）拆成两个类，让母亲做饭，父亲洗碗：

1. **package** com.cz.subject.concrete;
3. **import** com.cz.subject.Cook;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 20:33
10. \*/
11. **public** **class** Mother **implements** Cook
12. {
13. @Override
14. **public** **void** cook()
15. {
16. System.out.println("母亲做饭");
17. }
18. }
19. **package** com.cz.subject.concrete;
21. **import** com.cz.subject.Washer;
23. /\*\*
24. \* @author CHANG Zheng
25. \* @version 1.0.0
26. \* @projectName design-patterns
27. \* @date 2020/1/4 21:09
28. \*/
29. **public** **class** Father **implements** Washer
30. {
31. @Override
32. **public** **void** wash()
33. {
34. System.out.println("父亲洗碗");
35. }
36. }

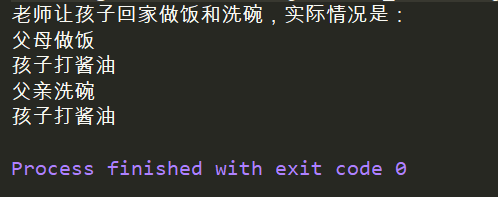
对于动态代理，不同的主题不需要写不同的代理类，只需要写一个InvocationHandler即可：

1. **package** com.cz;
3. **import** com.cz.subject.Cook;
4. **import** java.lang.reflect.InvocationHandler;
5. **import** java.lang.reflect.Method;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 21:44
12. \*/
13. **public** **class** SubjectHandler **implements** InvocationHandler
14. {
15. **private** Object abstractSubject;
17. **public** **void** setAbstractSubject(Object realSubject)
18. {
19. **this**.abstractSubject = realSubject;
20. }
22. @Override
23. **public** Object invoke(Object obj, Method method, Object[] args) **throws** Throwable
24. {
25. //利用反射机制将请求分派给委托类处理。Method的invoke返回Object对象作为方法执行结果。
26. //因为示例程序没有返回值，所以这里忽略了返回值处理
27. method.invoke(abstractSubject, args);
28. System.out.println("孩子打酱油");
29. **return** **null**;
30. }
32. }

客户端——老师，即测试类：

1. **package** com.cz;
3. **import** com.cz.subject.Washer;
4. **import** com.cz.subject.Cook;
5. **import** com.cz.subject.concrete.Father;
6. **import** com.cz.subject.concrete.Mother;
7. **import** java.lang.reflect.Proxy;
9. /\*\*
10. \* @author CHANG Zheng
11. \* @version 1.0.0
12. \* @projectName design-patterns
13. \* @date 2020/1/4 20:32
14. \*/
15. **public** **class** Main
16. {
17. **public** **static** **void** main(String[] args)
18. {
19. System.out.println("老师让孩子回家做饭和洗碗，实际情况是：");
21. Mother mother = **new** Mother();
22. Father father = **new** Father();
24. SubjectHandler handler = **new** SubjectHandler();
26. //生成代理类对象
27. handler.setAbstractSubject(mother);
28. Cook cookingChild = (Cook) Proxy
29. .newProxyInstance(ClassLoader.getSystemClassLoader(), **new** Class[]{Cook.**class**},
30. handler);
31. cookingChild.cook();
33. handler.setAbstractSubject(father);
34. Washer washeringChild = (Washer) Proxy
35. .newProxyInstance(ClassLoader.getSystemClassLoader(), **new** Class[]{Washer.**class**},
36. handler);
37. washeringChild.wash();
38. }
39. }

运行结果：



### 2.1.4 应用场景

（1）当我们想要隐藏某个类时，可以为其提供代理类

　　（2）当一个类需要对不同的调用者提供不同的调用权限时，可以使用代理类来实现（代理类不一定只有一个，我们可以建立多个代理类来实现，也可以在一个代理类中金进行权限判断来进行不同权限的功能调用）

（3）当我们要扩展某个类的某个功能时，可以使用代理模式，在代理类中进行简单扩展（只针对简单扩展，可在引用委托类的语句之前与之后进行）

### 2.1.5 优缺点

#### 优点

（1）实现了访问者与访问对象之间的解耦。

（2）代理模式在客户端与对象之间起到中介作用，保护了对对象的访问。

（3）代理模式可以在访问过程中增加逻辑（aop）。

#### 缺点

（1）由于在客户端和真实对象之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢；

（2）实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

（3）静态代理在委托类变多的情况时会显的非常臃肿，不方便阅读与使用

## 2.2 适配器模式

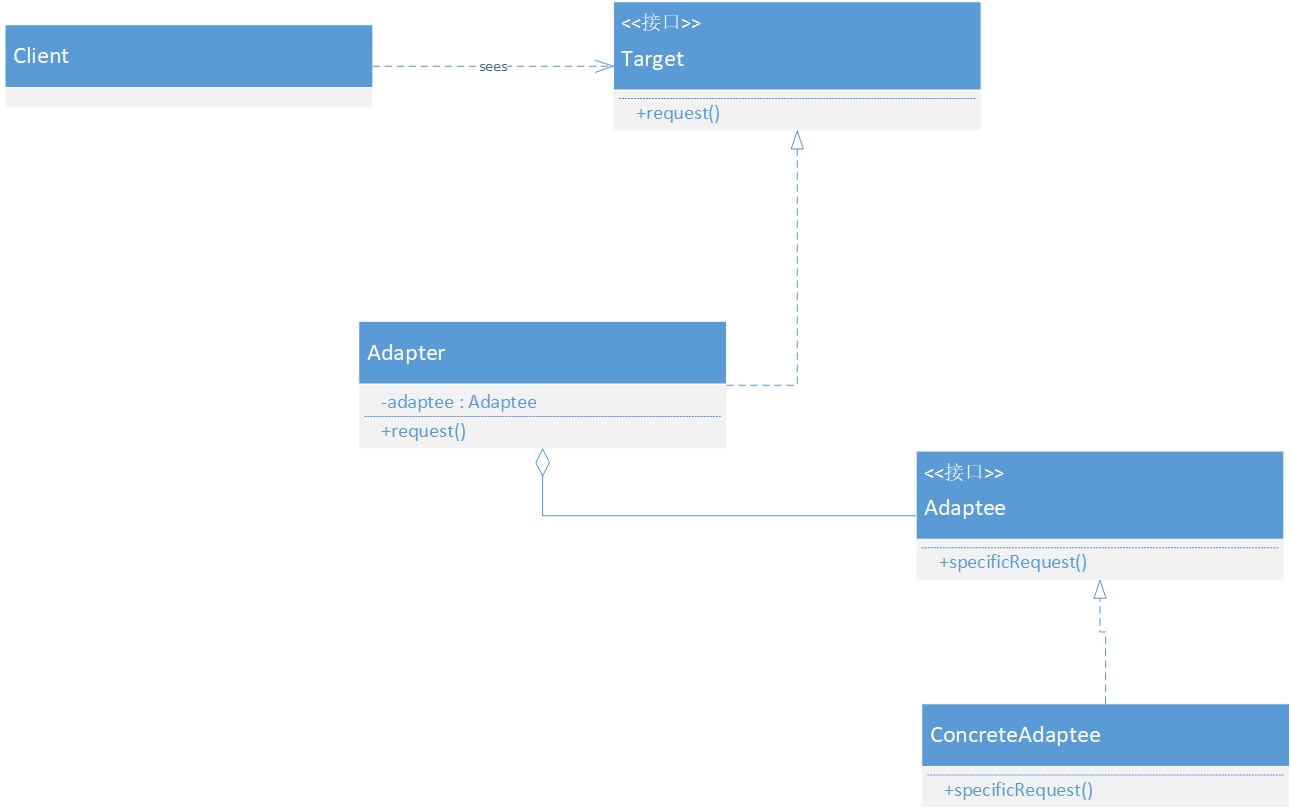
### 2.2.1 定义

适配器是将一个类的接口转换成客户希望的另外一个接口这样使得原本由于接口不兼容而不能一起工作的那些类可以一起工作，采用适配器的模式称为适配器模式。

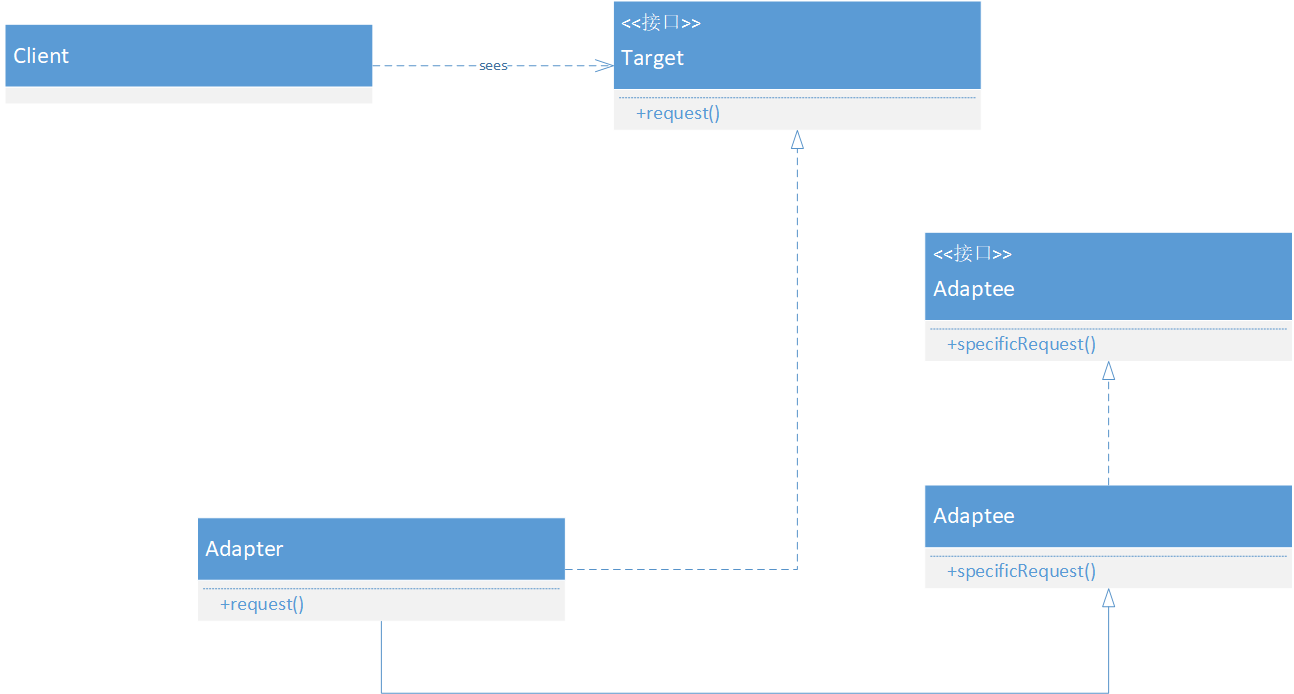
适配器分为类适配器和对象适配器。

### 2.2.2 一般模式

对象适配器：



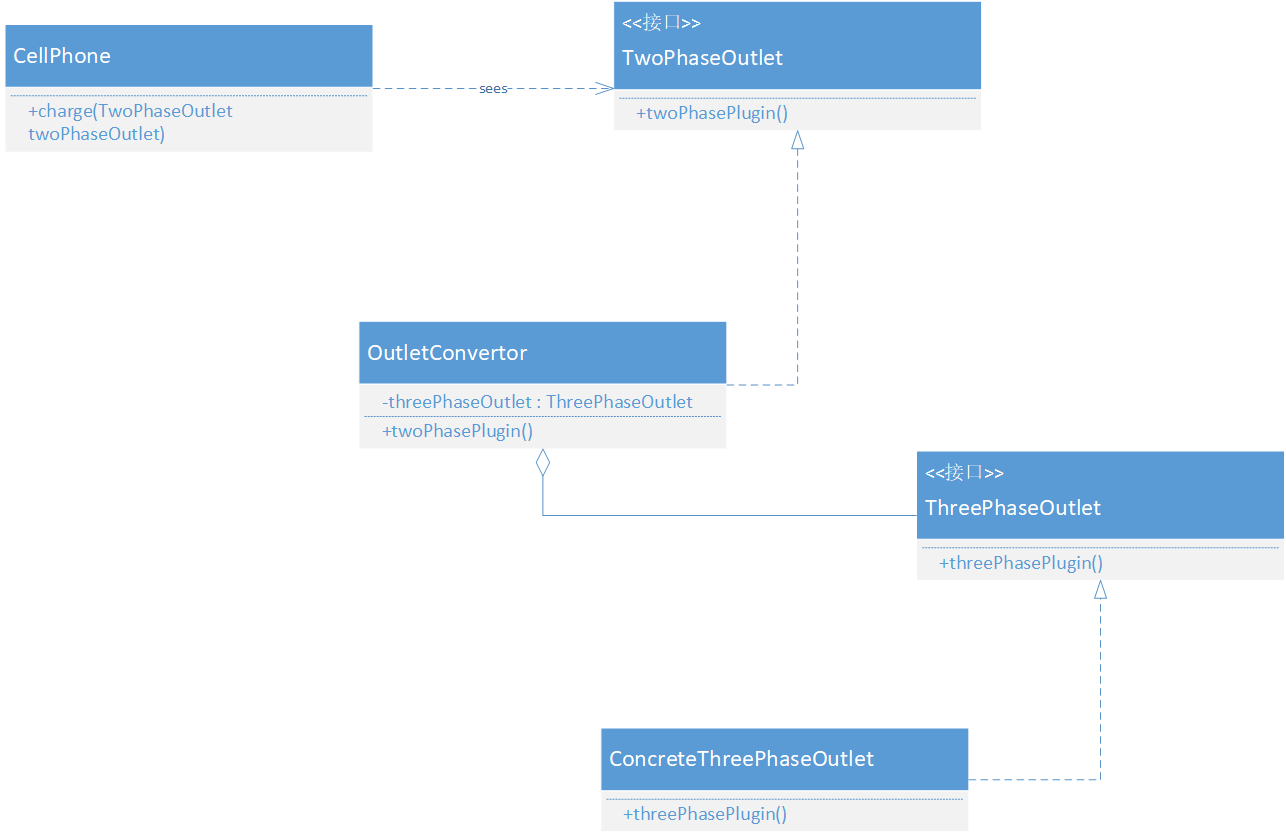
类适配器：



### 2.2.3 举例

#### 例1 插口转换-对象适配器

你在网上买了一个手机，但是买家给你发回来了一个2接头的充电器，但是恰好你又没有2接头的插槽，只有3个接口的插槽，于是你很直然地便会想到去找你个3接口转2接口的转换器。简单的分析下这个转换器便是我们这里的适配器Adapter。三相插口便是我们要适配的Adaptee，其中具体的三相插口就是我们现在有个3插口的插槽。



各个类的定义如下：

被适配的接口——三相插孔：

1. **package** com.cz.adaptee;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 15:17
8. \*/
9. **public** **interface** ThreePhaseOutlet
10. {
11. **void** threePhasePlugin();
12. }
13. **package** com.cz.adaptee.concrete;
15. **import** com.cz.adaptee.ThreePhaseOutlet;
17. /\*\*
18. \* @author CHANG Zheng
19. \* @version 1.0.0
20. \* @projectName design-patterns
21. \* @date 2020/1/4 15:26
22. \*/
23. **public** **class** ConcreteThreePhaseOutlet **implements** ThreePhaseOutlet
24. {
25. **public** **void** threePhasePlugin()
26. {
27. System.out.println("this three phase outlet plugined");
28. }
29. }

目标——两相插孔：

1. **package** com.cz.target;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 15:18
8. \*/
9. **public** **interface** TwoPhaseOutlet
10. {
11. **void** twoPhasePlugin();
12. }

适配器：

1. **package** com.cz.adapter;
3. **import** com.cz.adaptee.ThreePhaseOutlet;
4. **import** com.cz.target.TwoPhaseOutlet;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 15:33
11. \*/
12. **public** **class** OutletConvertor **implements** TwoPhaseOutlet
13. {
14. **private** ThreePhaseOutlet threePhaseOutlet;
16. **public** OutletConvertor(ThreePhaseOutlet threePhaseOutlet)
17. {
18. **this**.threePhaseOutlet = threePhaseOutlet;
19. }
21. **public** **void** twoPhasePlugin()
22. {
23. threePhaseOutlet.threePhasePlugin();
24. }
25. }

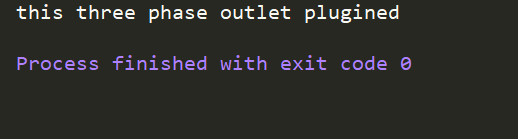
客户类——手机：

1. **package** com.cz.client;
3. **import** com.cz.target.TwoPhaseOutlet;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 15:30
10. \*/
11. **public** **class** CellPhone
12. {
13. **public** **void** charge(TwoPhaseOutlet twoPhaseOutlet)
14. {
15. twoPhaseOutlet.twoPhasePlugin();
16. }
17. }

测试类：

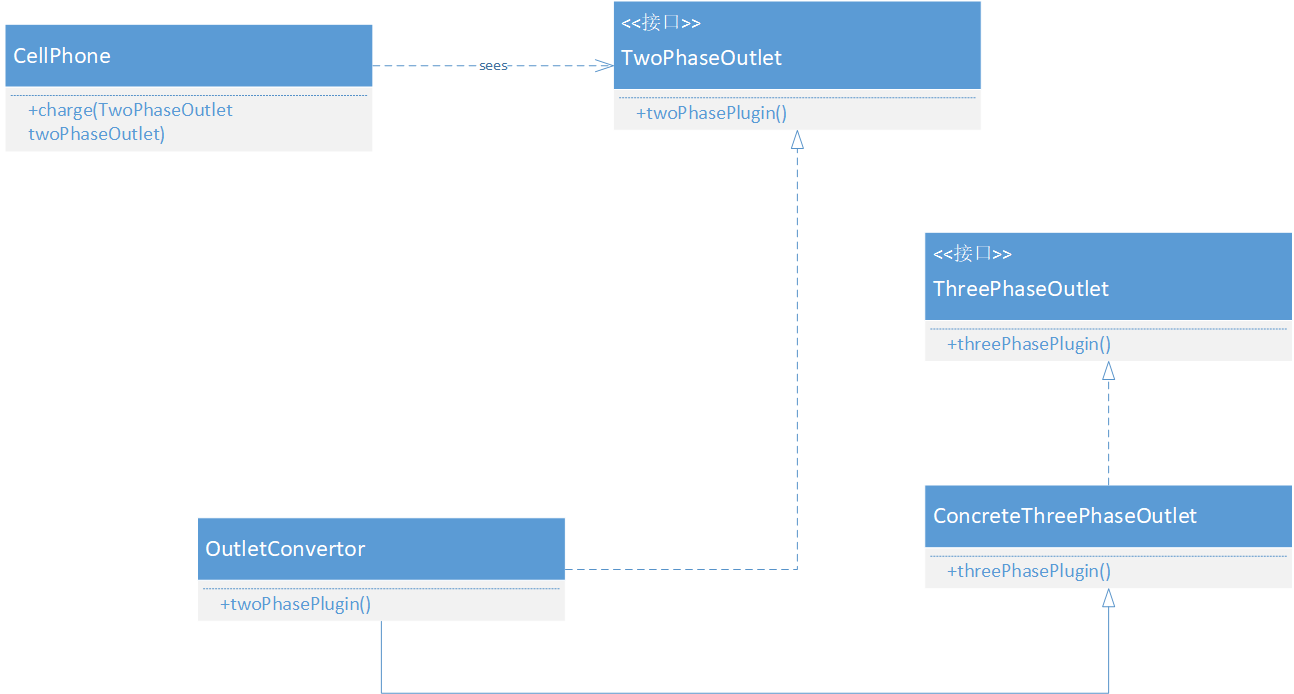
1. **package** com.cz;
3. **import** com.cz.adaptee.ThreePhaseOutlet;
4. **import** com.cz.adaptee.concrete.ConcreteThreePhaseOutlet;
5. **import** com.cz.adapter.OutletConvertor;
6. **import** com.cz.client.CellPhone;
8. /\*\*
9. \* @author CHANG Zheng
10. \* @version 1.0.0
11. \* @projectName design-patterns
12. \* @date 2020/1/4 15:18
13. \*/
14. **public** **class** Main
15. {
16. **public** **static** **void** main(String[] args)
17. {
18. CellPhone cellPhone = **new** CellPhone();
19. ThreePhaseOutlet concreteThreePhaseOutlet = **new** ConcreteThreePhaseOutlet();
20. OutletConvertor outletConvertor = **new** OutletConvertor(concreteThreePhaseOutlet);
21. cellPhone.charge(outletConvertor);
22. }
23. }

运行结果：



#### 例2 插口转换-类适配器

场景和例1一样，这回采用类适配器。



各个类的定义如下：

被适配的类、目标类、客户类和例1一样，适配器类：

1. **package** com.cz.adapter;
3. **import** com.cz.adaptee.concrete.ConcreteThreePhaseOutlet;
4. **import** com.cz.target.TwoPhaseOutlet;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName design-patterns
10. \* @date 2020/1/4 15:33
11. \*/
12. **public** **class** OutletConvertor **extends** ConcreteThreePhaseOutlet **implements** TwoPhaseOutlet
13. {
14. **public** **void** twoPhasePlugin()
15. {
16. threePhasePlugin();
17. }
18. }

测试类：

1. **package** com.cz;
3. **import** com.cz.adaptee.ThreePhaseOutlet;
4. **import** com.cz.adaptee.concrete.ConcreteThreePhaseOutlet;
5. **import** com.cz.adapter.OutletConvertor;
6. **import** com.cz.client.CellPhone;
8. /\*\*
9. \* @author CHANG Zheng
10. \* @version 1.0.0
11. \* @projectName design-patterns
12. \* @date 2020/1/4 15:18
13. \*/
14. **public** **class** Main
15. {
16. **public** **static** **void** main(String[] args)
17. {
18. CellPhone cellPhone = **new** CellPhone();
19. OutletConvertor outletConvertor = **new** OutletConvertor();
20. cellPhone.charge(outletConvertor);
21. }
22. }

运行结果和例1一样。

### 2.2.4 应用场景

在以下情况下可以使用适配器模式：

（1）系统需要使用现有的类，而这些类的接口不符合系统的需要。

（2）想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

### 2.2.5 优缺点

#### 优点

（1）将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。

（2）增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性

（3）灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。

类适配器模式还具有如下优点：

（4）由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。

对象适配器模式还具有如下优点：

（5）一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口。

#### 缺点

类适配器模式的缺点如下：

（1）对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，其使用有一定的局限性，不能将一个适配者类和它的子类都适配到目标接口。

对象适配器模式的缺点如下：

（2）与类适配器模式相比，要想置换适配者类的方法就不容易。如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

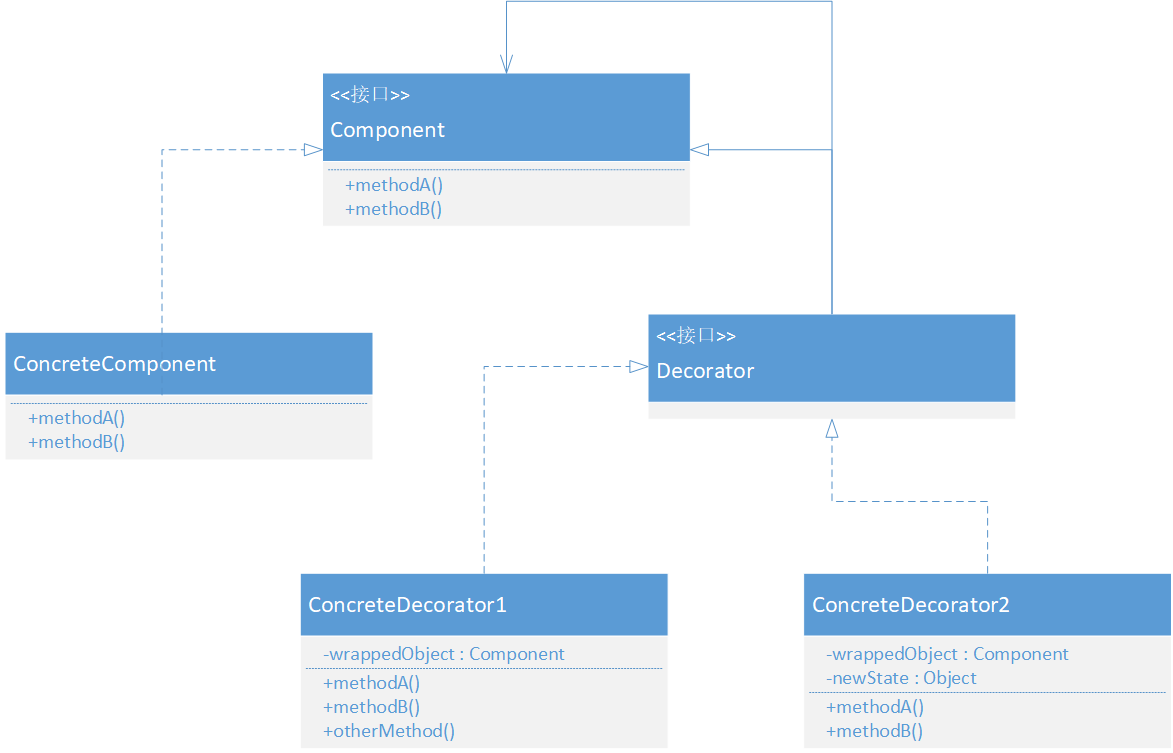
## 2.3 装饰器模式

### 2.3.1 定义

装饰器模式动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

和代理模式很相似，但在对被装饰的对象的控制程度是不同的；装饰器模式是对对象功能的加强，而代理模式是对对象施加控制，并不提供对对象本身功能的加强。

### 2.3.2 一般模式



Component：给出一个抽象接口或者抽象类，规范实现类的一些方法。

ConcreteComponent：具体的一个组件，实现了抽象方法。

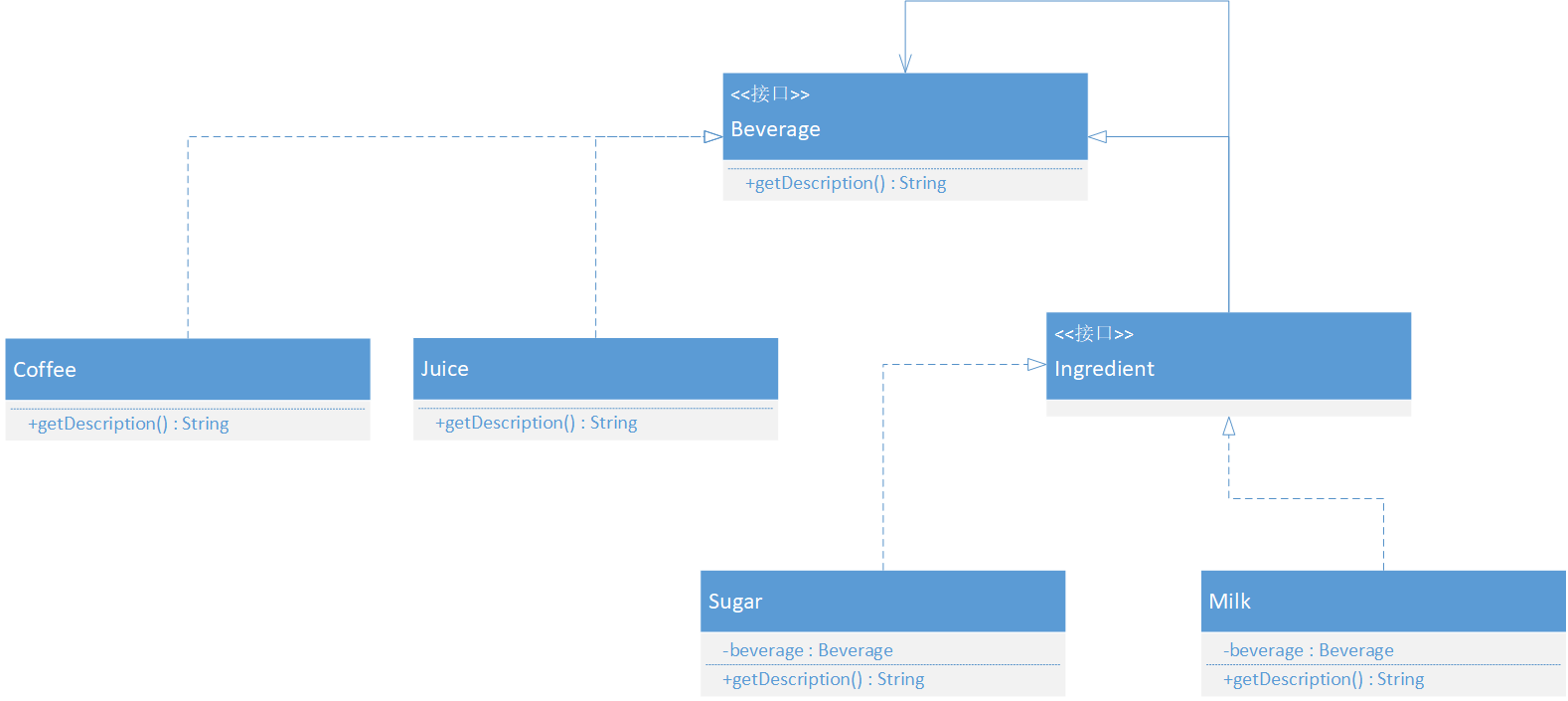
Decorator：抽象的装饰者，对抽象接口或者抽象类的一个引用，在method方法里面使用这个引用完成任务（代理模式需要实例化）。

ConcreteDecorator：具体的装饰者，对抽象装饰者的抽象部分进行实现。

### 2.3.3 举例

#### 例1 饮料加料

饮料有许多类型，比如咖啡、果汁，还可以加各种辅料，比如糖、牛奶等。另外，辅料不仅可以加一份，还可以加多份。最终的饮料的描述取决于饮料的种类和辅料。



各个类的定义如下：

Component——饮料类：

1. **package** com.cz.component;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 17:34
8. \*/
9. **public** **interface** Beverage
10. {
11. String getDescription();
12. }

具体饮料：

1. **package** com.cz.component.concrete;
3. **import** com.cz.component.Beverage;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 17:36
10. \*/
11. **public** **class** Coffee **implements** Beverage
12. {
14. @Override
15. **public** String getDescription()
16. {
17. **return** "Coffee";
18. }
19. }
20. **package** com.cz.component.concrete;
22. **import** com.cz.component.Beverage;
24. /\*\*
25. \* @author CHANG Zheng
26. \* @version 1.0.0
27. \* @projectName design-patterns
28. \* @date 2020/1/4 17:49
29. \*/
30. **public** **class** Juice **implements** Beverage
31. {
32. @Override
33. **public** String getDescription()
34. {
35. **return** "Juice";
36. }
37. }

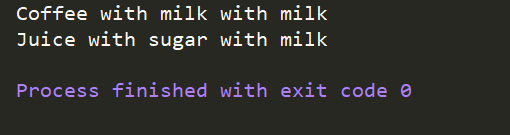
装饰器类——辅料：

1. **package** com.cz.decorator;
3. **import** com.cz.component.Beverage;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 17:50
10. \*/
11. **public** **interface** Ingredient **extends** Beverage
12. {
13. }
14. **package** com.cz.decorator.concrete;
16. **import** com.cz.component.Beverage;
17. **import** com.cz.decorator.Ingredient;
19. /\*\*
20. \* @author CHANG Zheng
21. \* @version 1.0.0
22. \* @projectName design-patterns
23. \* @date 2020/1/4 17:51
24. \*/
25. **public** **class** Milk **implements** Ingredient
26. {
27. **private** Beverage beverage;
29. **public** Milk(Beverage beverage)
30. {
31. **this**.beverage = beverage;
32. }
34. @Override
35. **public** String  getDescription()
36. {
37. **return** beverage.getDescription() + " with milk";
38. }
39. }
40. **package** com.cz.decorator.concrete;
42. **import** com.cz.component.Beverage;
43. **import** com.cz.decorator.Ingredient;
45. /\*\*
46. \* @author CHANG Zheng
47. \* @version 1.0.0
48. \* @projectName design-patterns
49. \* @date 2020/1/4 17:55
50. \*/
51. **public** **class** Sugar **implements** Ingredient
52. {
53. **private** Beverage beverage;
55. **public** Sugar(Beverage beverage)
56. {
57. **this**.beverage = beverage;
58. }
60. @Override
61. **public** String getDescription()
62. {
63. **return** beverage.getDescription() + " with sugar";
64. }
65. }

测试类：

1. **package** com.cz;
3. **import** com.cz.component.Beverage;
4. **import** com.cz.component.concrete.Coffee;
5. **import** com.cz.component.concrete.Juice;
6. **import** com.cz.decorator.concrete.Milk;
7. **import** com.cz.decorator.concrete.Sugar;
9. /\*\*
10. \* @author CHANG Zheng
11. \* @version 1.0.0
12. \* @projectName design-patterns
13. \* @date 2020/1/4 17:33
14. \*/
15. **public** **class** Main
16. {
17. **public** **static** **void** main(String[] args)
18. {
19. Beverage coffeeWithTwoMilk = **new** Milk(**new** Milk(**new** Coffee()));
20. System.out.println(coffeeWithTwoMilk.getDescription());
22. Beverage juickWithSugarAndMilk = **new** Milk(**new** Sugar(**new** Juice()));
23. System.out.println(juickWithSugarAndMilk.getDescription());
24. }
26. }

运行结果：



### 2.3.4 应用场景

因为装饰者模式的以下特点：

（1）装饰对象和真实对象有相同的接口。这样客户端对象就能以和真实对象相同的方式和装饰对象交互。

（2）装饰对象包含一个真实对象的引用（reference）。

（3）装饰对象接受所有来自客户端的请求。它把这些请求转发给真实的对象。

（4）装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。

所以在以下情况下可以使用装饰模式：

（1）在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

（2）需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。

（3）当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类定义不能继承（如final类）。

### 2.3.5 优缺点

#### 优点

（1）装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。

（2）可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。

（3）通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。

（4）具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”。

#### 缺点

（1）使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。

（2）这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

# 3 行为型

## 3.1 策略模式

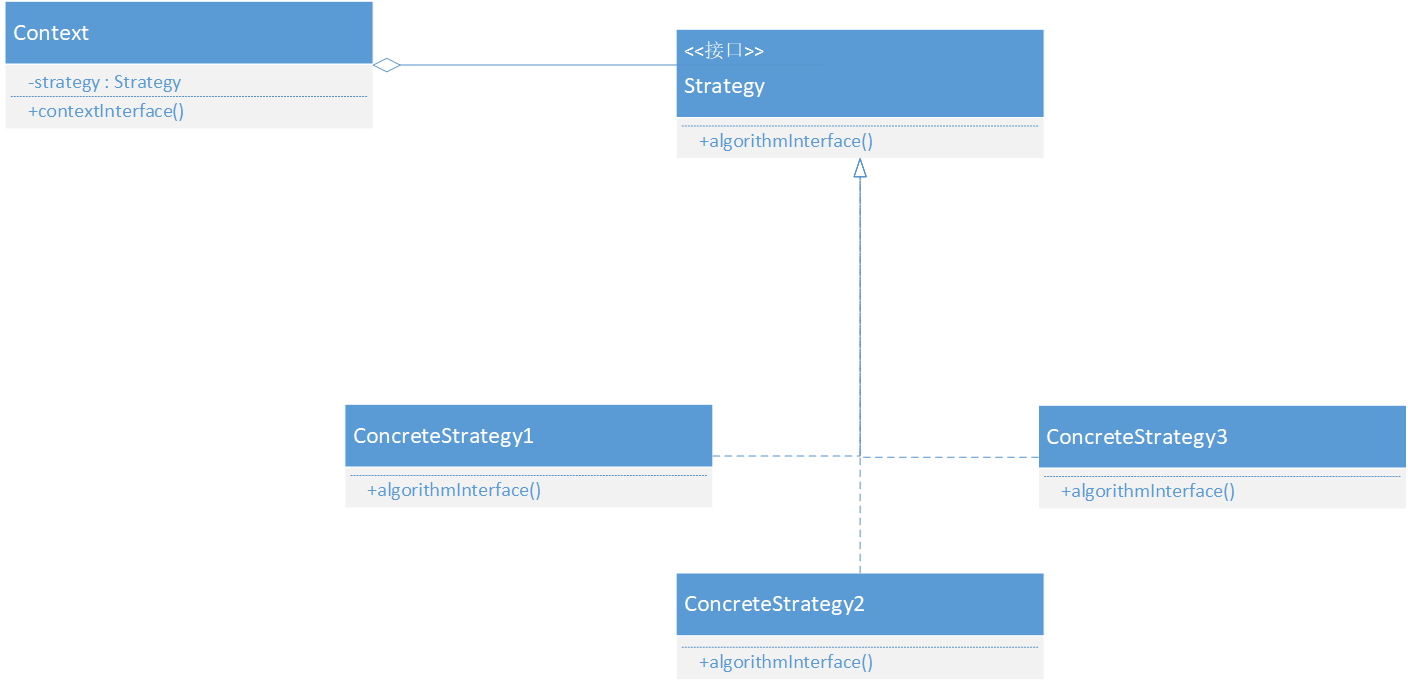
### 3.1.1 定义

策略模式，又叫算法簇模式，就是定义了不同的算法簇，并且之间可以互相替换，此模式算法的变化独立于使用算法的客户。

策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得他们可以互换。”

设计原则是把一个类中经常改变或者将来改变的部分提取出来，作为一个接口，然后在类中包含这个对象的实例，这样类的实例在运行时就可以随意调用实现这个接口的类的行为。

### 3.1.2 一般模式



（1）环境对象（Context）：该类中实现了对抽象策略中定义的接口或者抽象类的引用，管理具体算法要操作的数据，是用户访问的接口。

（2）抽象策略对象（Strategy）：算法的通用接口，它可由接口或者抽象类来实现。

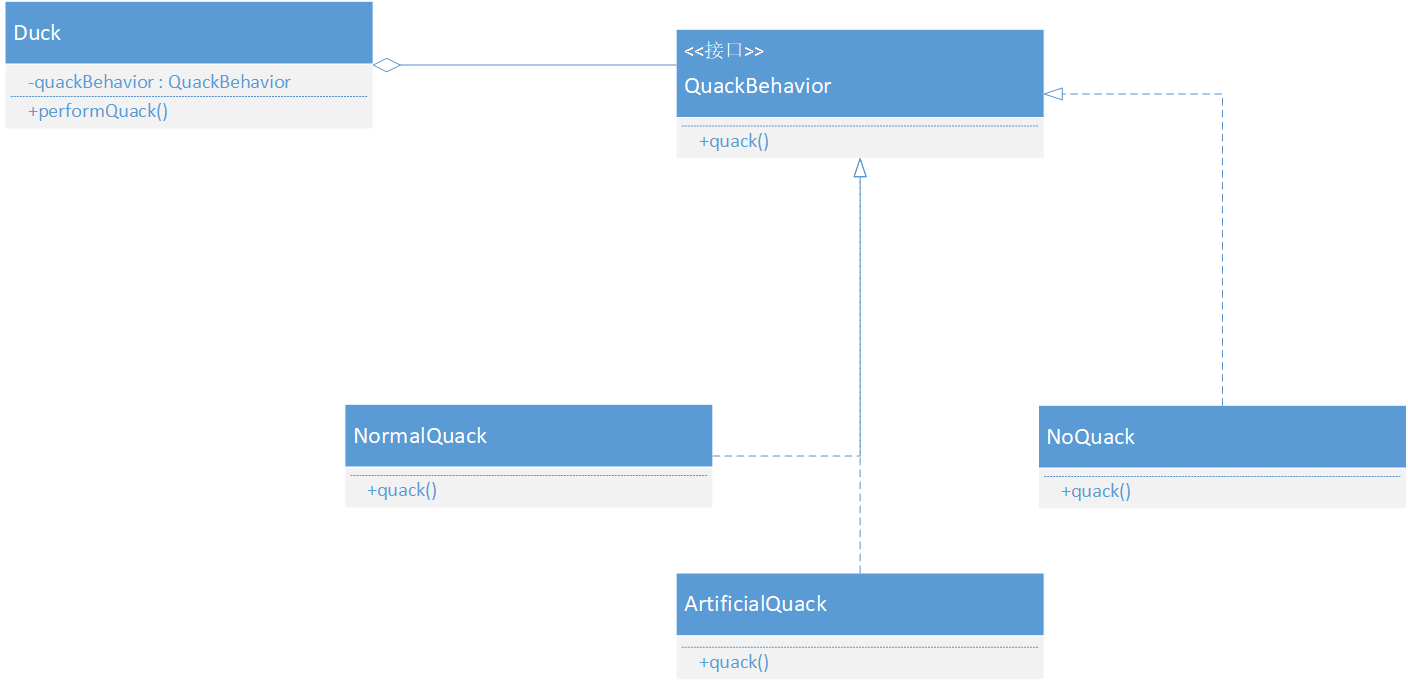
（3）具体策略对象（ConcreteStrategy）：它封装了实现不同功能的算法。

### 3.1.3 举例

#### 例1 鸭子的行为

对于不同的鸭子，有着不同的叫声和不同的飞行行为。有的鸭子，例如橡皮鸭是不能飞的，诱饵鸭既不能飞也不能叫。

可以采用策略模式，以鸭子叫为例：



对于三种鸭子：红头鸭、橡皮鸭和诱饵鸭，他们三者叫的方式分别为正常地叫、人工方式叫和不能叫。其类定义如下：

抽象鸭子类：

1. **package** CZ.context;
3. **import** CZ.strategy.QuackBehavior;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName 策略模式-鸭子叫
9. \* @date 2020/1/3 20:41
10. \*/
11. **public** **abstract** **class** AbstractDuck
12. {
13. **private** QuackBehavior quackBehavior;
15. **public** **void** performQuack()
16. {
17. quackBehavior.quack();
18. }
20. **protected** **void** setQuackBehavior(QuackBehavior quackBehavior)
21. {
22. **this**.quackBehavior = quackBehavior;
23. }
24. }

红头鸭：

1. **package** CZ.context.concrete;
3. **import** CZ.context.AbstractDuck;
4. **import** CZ.strategy.concrete.NormalQuack;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName 策略模式-鸭子叫
10. \* @date 2020/1/3 20:44
11. \*/
12. **public** **class** RedHeadDuck **extends** AbstractDuck
13. {
14. **public** RedHeadDuck()
15. {
16. setQuackBehavior(**new** NormalQuack());
17. }
18. }

橡皮鸭：

1. **package** CZ.context.concrete;
3. **import** CZ.context.AbstractDuck;
4. **import** CZ.strategy.concrete.ArtificialQuack;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName 策略模式-鸭子叫
10. \* @date 2020/1/3 20:58
11. \*/
12. **public** **class** RubberDuck **extends** AbstractDuck
13. {
14. **public** RubberDuck()
15. {
16. setQuackBehavior(**new** ArtificialQuack());
17. }
18. }

诱饵鸭：

1. **package** CZ.context.concrete;
3. **import** CZ.context.AbstractDuck;
4. **import** CZ.strategy.concrete.NoQuack;
6. /\*\*
7. \* @author CHANG Zheng
8. \* @version 1.0.0
9. \* @projectName 策略模式-鸭子叫
10. \* @date 2020/1/3 20:57
11. \*/
12. **public** **class** DecoyDuck **extends** AbstractDuck
13. {
14. **public** DecoyDuck()
15. {
16. setQuackBehavior(**new** NoQuack());
17. }
18. }

抽象策略类——叫这种行为：

1. **package** CZ.strategy;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName 策略模式-鸭子叫
7. \* @date 2020/1/3 20:43
8. \*/
9. **public** **interface** QuackBehavior
10. {
11. **void** quack();
12. }

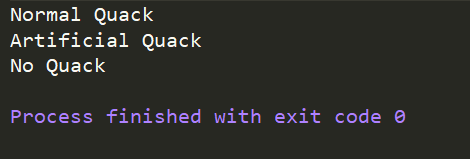
具体策略类——不同的叫法：

1. **package** CZ.strategy.concrete;
3. **import** CZ.strategy.QuackBehavior;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName 策略模式-鸭子叫
9. \* @date 2020/1/3 20:49
10. \*/
11. **public** **class** NormalQuack **implements** QuackBehavior
12. {
13. @Override
14. **public** **void** quack()
15. {
16. System.out.println("Normal Quack");
17. }
18. }
19. **package** CZ.strategy.concrete;
21. **import** CZ.strategy.QuackBehavior;
23. /\*\*
24. \* @author CHANG Zheng
25. \* @version 1.0.0
26. \* @projectName 策略模式-鸭子叫
27. \* @date 2020/1/3 20:50
28. \*/
29. **public** **class** ArtificialQuack **implements** QuackBehavior
30. {
31. @Override
32. **public** **void** quack()
33. {
34. System.out.println("Artificial Quack");
35. }
36. }
37. **package** CZ.strategy.concrete;
39. **import** CZ.strategy.QuackBehavior;
41. /\*\*
42. \* @author CHANG Zheng
43. \* @version 1.0.0
44. \* @projectName 策略模式-鸭子叫
45. \* @date 2020/1/3 20:50
46. \*/
47. **public** **class** NoQuack **implements** QuackBehavior
48. {
49. @Override
50. **public** **void** quack()
51. {
52. System.out.println("No Quack");
53. }
54. }

测试：

1. **package** CZ;
3. **import** CZ.context.AbstractDuck;
4. **import** CZ.context.concrete.DecoyDuck;
5. **import** CZ.context.concrete.RedHeadDuck;
6. **import** CZ.context.concrete.RubberDuck;
8. **public** **class** Main
9. {
11. **public** **static** **void** main(String[] args)
12. {
13. AbstractDuck redHeadDuck = **new** RedHeadDuck();
14. AbstractDuck rubberDuck = **new** RubberDuck();
15. AbstractDuck decoyDuck = **new** DecoyDuck();
17. redHeadDuck.performQuack();
18. rubberDuck.performQuack();
19. decoyDuck.performQuack();
20. }
21. }

运行结果：



### 3.1.4 应用场景

　　策略模式的好处在于你可以动态改变对象的行为。当存在以下情况时，使用策略模式：

　　（1）“策略”提供一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。

　　（2）需要使用一个算法的不同变体。例如，你可能会定义一些反应不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。

　　（3）算法使用了客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

（4）一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，复杂并且难以维护。将相关的条件分支移入他们各自的Strategy类中以代替这些条件语句。

（5）许多关联的类之间的区别仅仅是其行为。

### 3.1.5 优缺点

#### 优点

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

（2）使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

（3）策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便。

#### 缺点

（1）策略模式并不决定在何时使用何种算法，客户端（即例1中的具体的鸭子类）必须知道所有的策略类，并自行决定使用哪一个策略类。这虽然提高了系统的灵活性，但是意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

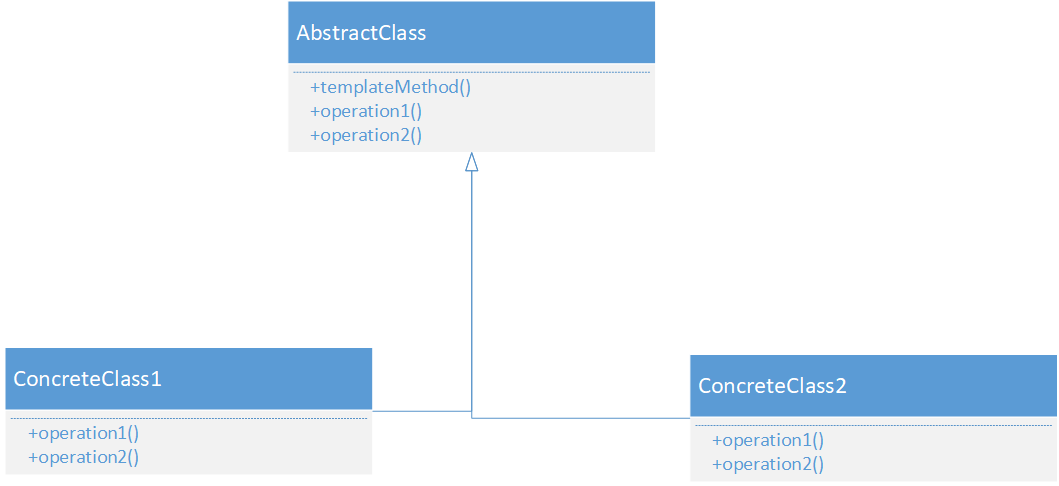
（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很多。

## 3.2 模板方法模式

### 3.2.1 定义

模板方法模式定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

### 3.2.2 一般模式



角色：

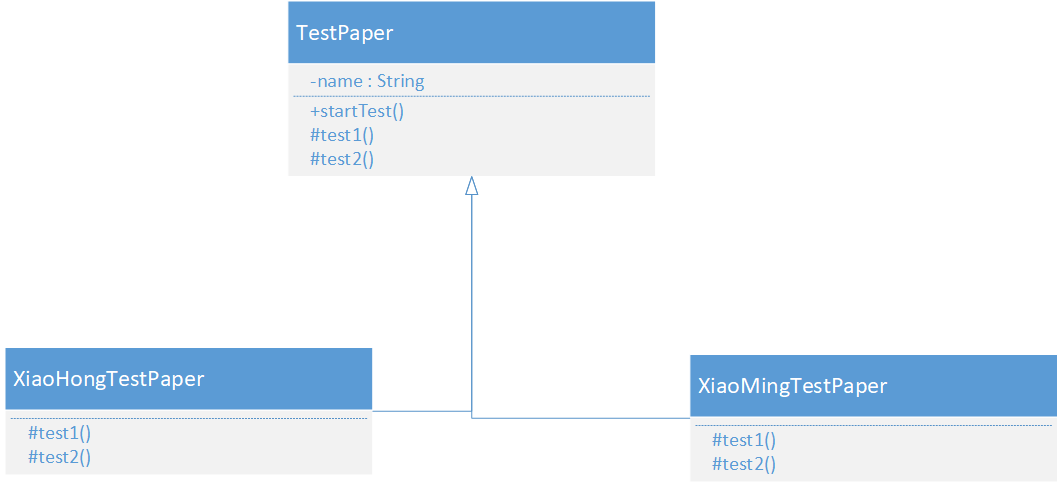
抽象类（AbstractClass）: 定义算法骨架。

具体子类 （ConcreteClass）: 重新定义特定的操作以完成算法中与特定子类相关的步骤，不改变算法骨架。

### 3.2.3 举例

#### 例1 考试答卷

班上有两位同学小红和小明进行考试，试卷上有两道题，两位同学分别对这两道题做出解答。



各个类的定义如下：

抽象模板类-试卷：

1. **package** com.cz.template;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 14:40
8. \*/
9. **public** **abstract** **class** TestPaper
10. {
11. **private** String name;
13. **public** TestPaper(String name)
14. {
15. **this**.name = name;
16. }
18. **protected** **abstract** **void** test1();
20. **protected** **abstract** **void** test2();
22. **public** **void** startTest()
23. {
24. System.out.println(name + " starts test.");
25. test1();
26. test2();
27. }
28. }

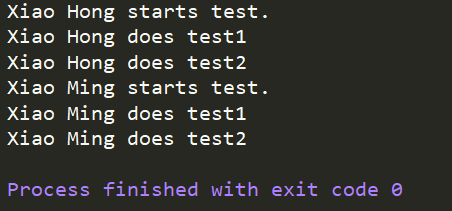
具体试卷类：

1. **package** com.cz.template.concrete;
3. **import** com.cz.template.TestPaper;
5. /\*\*
6. \* @author CHANG Zheng
7. \* @version 1.0.0
8. \* @projectName design-patterns
9. \* @date 2020/1/4 14:43
10. \*/
11. **public** **class** XiaoHongTestPaper **extends** TestPaper
12. {
14. **public** XiaoHongTestPaper()
15. {
16. **super**("Xiao Hong");
17. }
19. @Override
20. **protected** **void** test1()
21. {
22. System.out.println("Xiao Hong does test1");
23. }
25. @Override
26. **protected** **void** test2()
27. {
28. System.out.println("Xiao Hong does test2");
29. }
30. }
31. **package** com.cz.template.concrete;
33. **import** com.cz.template.TestPaper;
35. /\*\*
36. \* @author CHANG Zheng
37. \* @version 1.0.0
38. \* @projectName design-patterns
39. \* @date 2020/1/4 14:45
40. \*/
41. **public** **class** XiaoMingTestPaper **extends** TestPaper
42. {
43. **public** XiaoMingTestPaper()
44. {
45. **super**("Xiao Ming");
46. }
48. @Override
49. **protected** **void** test1()
50. {
51. System.out.println("Xiao Ming does test1");
52. }
54. @Override
55. **protected** **void** test2()
56. {
57. System.out.println("Xiao Ming does test2");
58. }
59. }

测试类：

1. **package** com.cz;
3. **import** com.cz.template.TestPaper;
4. **import** com.cz.template.concrete.XiaoHongTestPaper;
5. **import** com.cz.template.concrete.XiaoMingTestPaper;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 14:40
12. \*/
13. **public** **class** Main
14. {
15. **public** **static** **void** main(String[] args)
16. {
17. TestPaper xiaoHongTestPaper = **new** XiaoHongTestPaper();
18. xiaoHongTestPaper.startTest();
20. TestPaper xiaoMingTestPaper = **new** XiaoMingTestPaper();
21. xiaoMingTestPaper.startTest();
22. }
23. }

运行结果：



### 3.2.4 应用场景

（1）多个子类有共有的方法，并且逻辑基本相同。

（2）重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

（3）重构时，模板方法是一个经常使用的方法，把相同的代码抽取到父类中，然后通过构造函数约束其行为。

### 3.2.5 优缺点

#### 优点

（1）模板方法模式在一个类中形式化地定义算法，而由它的子类实现细节的处理。

（2）模板方法是一种代码复用的基本技术。它们在类库中尤为重要，它们提取了类库中的公共行为。

（3）每个子类为一种实现，更加符合“单一职责原则”，使得类的内聚性得以提高。

#### 缺点

每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象。

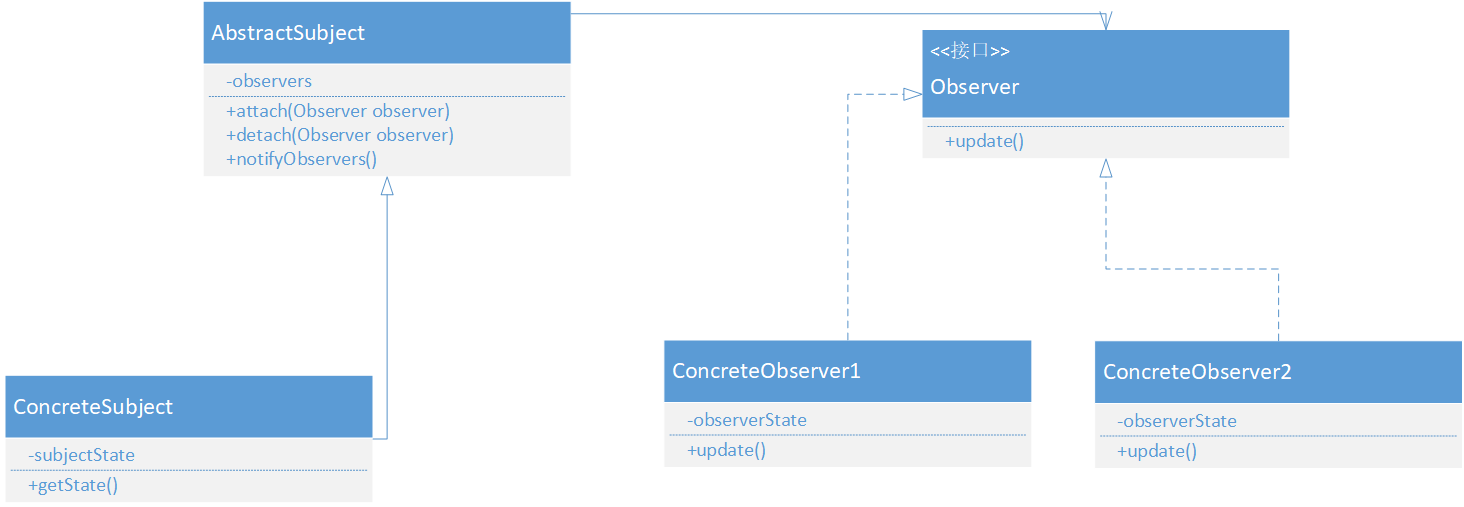
## 3.3 观察者模式

### 3.3.1 定义

观察者模式描述的是一种一对多的关系“一个被观察者对应多个观察者”，当被观察者的状态发生改变时，所有观察者都会得到通知。

观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

### 3.3.2 一般模式



Subject：接口或抽象类,也叫被观察者

ConcreteSubject：具体的被观察者，内部维护了观察者的列表

Observer：观察者接口或抽象类，抽象出观察者的接口

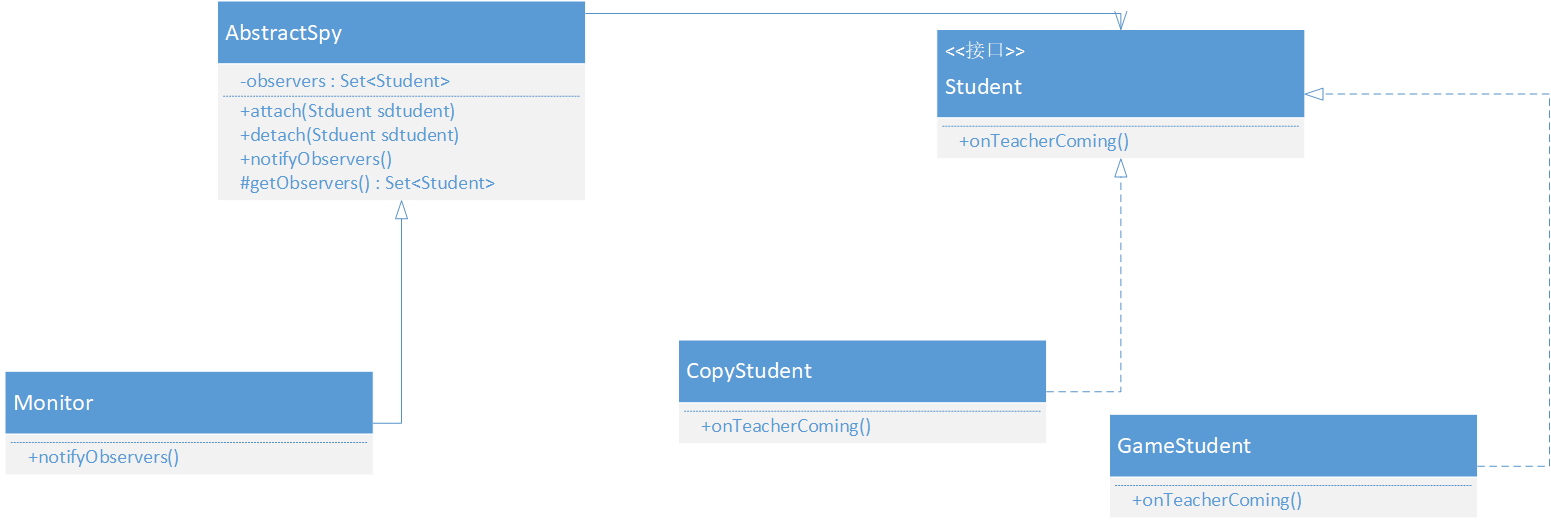
ConcreteObserver：具体的观察者，被观察者有更新，观察者立马响应更新

### 3.3.3 举例

#### 例1 班长卧底

在教室里老师还没有来，同学都在干着各的事情，小张正在打游戏，小李正在抄作业,现在同学们要求班长当卧底，监视老师，当老师来了通知大家一声。然后打游戏的马上停止，抄作业的也停止。

这里班长作为被观察者，干坏事的同学为观察者。一旦班长发现老师来了就通知所有观察的同学停止干坏事。



具体各个类的定义如下：

被观察者：

1. **package** com.cz.subject;
3. **import** com.cz.observer.Student;
4. **import** java.util.HashSet;
5. **import** java.util.Set;
7. /\*\*
8. \* @author CHANG Zheng
9. \* @version 1.0.0
10. \* @projectName design-patterns
11. \* @date 2020/1/4 18:50
12. \*/
13. **public** **abstract** **class** AbstractSpy
14. {
15. **private** Set<Student> observers = **new** HashSet<>();
17. **protected** Set<Student> getObservers()
18. {
19. **return** observers;
20. }
22. **public** **void** attach(Student student)
23. {
24. observers.add(student);
25. }
27. **public** **void** detach(Student student)
28. {
29. observers.remove(student);
30. }
32. **public** **abstract** **void** notifyObservers();
33. }
34. **package** com.cz.subject.concrete;
36. **import** com.cz.observer.Student;
37. **import** com.cz.subject.AbstractSpy;
38. **import** java.util.Set;
40. /\*\*
41. \* @author CHANG Zheng
42. \* @version 1.0.0
43. \* @projectName design-patterns
44. \* @date 2020/1/4 19:04
45. \*/
46. **public** **class** Monitor **extends** AbstractSpy
47. {
48. @Override
49. **public** **void** notifyObservers()
50. {
51. **final** Set<Student> observers = getObservers();
52. **for** (Student observer : observers)
53. {
54. observer.onTeacherComing();
55. }
56. }
57. }

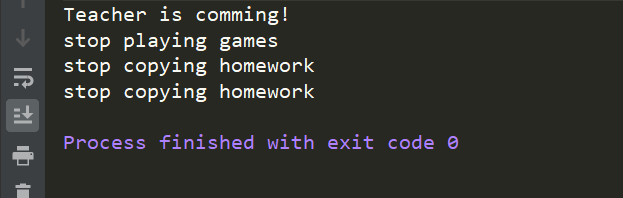
观察者：

1. **package** com.cz.observer;
3. /\*\*
4. \* @author CHANG Zheng
5. \* @version 1.0.0
6. \* @projectName design-patterns
7. \* @date 2020/1/4 18:47
8. \*/
9. **public** **interface** Student
10. {
11. **void** onTeacherComing();
12. }
13. **package** com.cz.observer.concrete;
15. **import** com.cz.observer.Student;
17. /\*\*
18. \* @author CHANG Zheng
19. \* @version 1.0.0
20. \* @projectName design-patterns
21. \* @date 2020/1/4 18:49
22. \*/
23. **public** **class** CopyStudent **implements** Student
24. {
25. @Override
26. **public** **void** onTeacherComing()
27. {
28. System.out.println("stop copying homework");
29. }
30. }
31. **package** com.cz.observer.concrete;
33. **import** com.cz.observer.Student;
35. /\*\*
36. \* @author CHANG Zheng
37. \* @version 1.0.0
38. \* @projectName design-patterns
39. \* @date 2020/1/4 18:49
40. \*/
41. **public** **class** GameStudent **implements** Student
42. {
43. @Override
44. **public** **void** onTeacherComing()
45. {
46. System.out.println("stop playing games");
47. }
48. }

测试类：

1. **package** com.cz;
3. **import** com.cz.observer.Student;
4. **import** com.cz.observer.concrete.CopyStudent;
5. **import** com.cz.observer.concrete.GameStudent;
6. **import** com.cz.subject.AbstractSpy;
7. **import** com.cz.subject.concrete.Monitor;
9. /\*\*
10. \* @author CHANG Zheng
11. \* @version 1.0.0
12. \* @projectName design-patterns
13. \* @date 2020/1/4 18:46
14. \*/
15. **public** **class** Main
16. {
17. **public** **static** **void** main(String[] args)
18. {
19. Student copyStudent1 = **new** CopyStudent();
20. Student copyStudent2 = **new** CopyStudent();
21. Student gameStudent = **new** GameStudent();
23. AbstractSpy monitor = **new** Monitor();
24. monitor.attach(copyStudent1);
25. monitor.attach(copyStudent2);
26. monitor.attach(gameStudent);
28. System.out.println("Teacher is comming!");
29. monitor.notifyObservers();
30. }
32. }

运行结果：



### 3.3.4 应用场景

（1）一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用。

（2）一个对象的改变将导致一个或多个其他对象也发生改变，而并不知道具体有多少对象将发生改变，也不知道这些对象是谁。

（3）需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

### 3.3.5 优缺点

#### 优点

（1）观察者模式可以实现表示层和数据逻辑层的分离，定义了稳定的消息更新传递机制，并抽象了更新接口，使得可以有各种各样不同的表示层充当具体观察者角色。

（2）观察者模式在观察目标和观察者之间建立一个抽象的耦合。观察目标只需要维持一个抽象观察者的集合，无须了解其具体观察者。由于观察目标和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。

（3）观察者模式支持广播通信，观察目标会向所有已注册的观察者对象发送通知，简化了一对多系统设计的难度。

（4）观察者模式满足 "开闭原则" 的要求，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便。

#### 缺点

（1）如果一个观察目标对象有很多直接和间接观察者，将所有的观察者都通知到会花费很多时间。

（2）如果在观察者和观察目标之间存在循环依赖，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

（3）观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

# 4 参考文献

[1] Mary Eleanor Wilkins Freeman. *Head First Design Patterns*. 2012

[2] 阿玛尼迪迪. <https://www.cnblogs.com/codingmengmeng/p>

[3] 凯玲之恋. <https://www.jianshu.com/p/91b7ed807093>

[4] Parallel\_Lines. <https://www.jianshu.com/p/c06a686dae39>