

设计模式 (Design Pattern, DP)

设计模式的产生背景

“设计模式”这个术语最初并不是出现在软件设计中，而是被用于建筑领域的设计中。

直到 1990 年，软件工程界才开始研讨设计模式的话题。

1995年，“四人组” (Gang of Four, GoF) 合作出版了《设计模式：可复用面向对象软件的基础》 (Design Patterns: Elements of Reusable Object-Oriented Software) 一书，在书籍中收录了 23 个设计模式，这是设计模式领域里程碑的事件，导致了软件设计模式的突破。

直到今天，狭义的设计模式还是该书中所介绍的23种经典设计模式。

设计模式的概念

软件设计模式 (Software Design Pattern)，又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。其目的是为了提

高代码的可重用性、代码的可读性和代码的可靠性。

1、设计模式的六大设计原则

1.1 开闭原则：Open Closed Principle, OCP

1.1.1 开闭原则的定义

开闭原则由勃兰特·梅耶 (Bertrand Meyer) 提出，他在 1988 年的著作《面向对象软件构造》 (Object Oriented Software Construction) 中提出：软件实体应当对扩展开放，对修改关闭 (Software entities should be open for extension, but closed for modification)，这就是开闭原则的经典定义。简单点说就是：一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现变化。那么什么是软件实体呢？

这里的软件实体包括以下几个部分：

- 项目中划分出的模块
- 类与接口
- 方法

一个软件产品在它的生命周期内一般都会发生变化，开闭原则视为软件实体的未来事件而制定的对现行开发设计进行约束的一个原则。

举个例子：以书店销售书籍为例：

```
public interface IBook {
    public String getName();
    public int getPrice();
    public String getAuthor();
}
public class NovelBook implements IBook{
    private String name;
```

```

private int price;
private String author;

public NovelBook(String name, int price, String author) {
    this.name = name;
    this.price = price;
    this.author = author;
}

@Override
public String getName() {
    return this.name;
}

@Override
public int getPrice() {
    return this.price;
}

@Override
public String getAuthor() {
    return this.author;
}
}

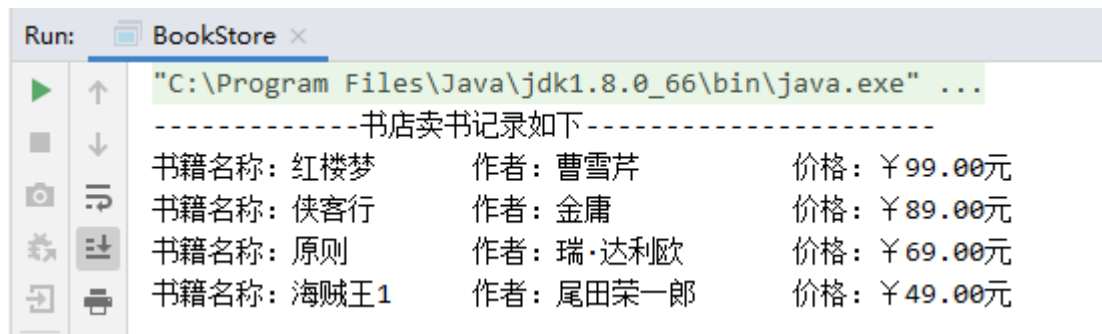
public class BookStore {
    private final static ArrayList<IBook> bookList=new ArrayList();

    static {
        bookList.add(new NovelBook("红楼梦", 9900, "曹雪芹 "));
        bookList.add(new NovelBook("侠客行", 8900, "金庸 "));
        bookList.add(new NovelBook("原则", 6900, "瑞·达利欧"));
        bookList.add(new NovelBook("海贼王1", 4900, "尾田荣一郎"));
    }

    public static void main(String[] args) {
        NumberFormat format=NumberFormat.getCurrencyInstance();
        format.setMaximumFractionDigits(2);
        System.out.println("-----书店卖书记录如下-----");
        for (IBook book : bookList) {
            System.out.println("书籍名称: "+book.getName()+"\t\t作者: "+book.getAuthor()+"\t\t价
格: "+format.format(book.getPrice()/100.0)+"元");
        }
    }
}

```

运行结果:



项目上线，书籍能正常销售了。但是现在双十一要到了，书店决定要来一波促销活动，70元以上的书籍9折销售，70元以下的书籍8折销售。对该项目来说，这就是一个变化，我们该怎么应该如下变化呢？有3中解决方法：

(1) 修改接口

在IBook接口里新增getOffPrice()方法，专门用于进行打折，所有的实现类都实现该方法。但这样修改的后果就是，实现类NovelBook要修改，书店类BookStore中的main方法也要修改，同时，IBook作为接口应该是稳定且可靠的，不应该经常发生变化，因此，该方案被否定。

(2) 修改实现类

修改NovelBook类中的方法，直接在getPrice()方法中实现打折处理，这个方法可以是可以，但如果采购书籍的人员要看价格怎么办，由于该方法已经进行了打折处理，因此采购人员看到的也是打折后的价格，会因信息不对称出现决策失误的情况。因此，该方案也不是一个最优的方案。

(3) 通过扩展实现变化

增加一个子类OffNovelBook，覆写getPrice方法，高层次的模块（也就是BookStore中static静态块中）通过OffNovelBook类产生新的对象，完成业务变化对系统的最小开发。这样修改也少，风险也小。

```
public class OffNovelBook extends NovelBook {
    public OffNovelBook(String name, int price, String author) {
        super(name, price, author);
    }

    @Override
    public int getPrice() {
        int selPrice=super.getPrice();
        int offPrice=0;
        if(selPrice>7000){
            offPrice=selPrice*90/100;
        }else{
            offPrice=selPrice*80/100;
        }
        return offPrice;
    }
}

public class BookStore {
    private final static ArrayList<IBook> bookList=new ArrayList();

    static {
        bookList.add(new OffNovelBook("红楼梦", 9900, "曹雪芹 "));
        bookList.add(new OffNovelBook("侠客行", 8900, "金庸 "));

        bookList.add(new OffNovelBook("原则", 6900, "瑞·达利欧"));
    }
}
```

```

        bookList.add(new OffNovelBook("海贼王1", 4900, "尾田荣一郎"));
    }

    public static void main(String[] args) {
        NumberFormat format=NumberFormat.getCurrencyInstance();
        format.setMaximumFractionDigits(2);
        System.out.println("-----书店卖书记录如下-----");
        for (IBook book : bookList) {
            System.out.println("书籍名称: "+book.getName()+"\t\t作者: "+book.getAuthor()+"\t\t价
            格: "+format.format(book.getPrice()/100.0)+"元");
        }
    }
}

```

运行结果:

```

Run: BookStore x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
-----书店卖书记录如下-----
书籍名称: 红楼梦      作者: 曹雪芹      价格: ￥89.10元
书籍名称: 侠客行      作者: 金庸        价格: ￥80.10元
书籍名称: 原则        作者: 瑞·达利欧   价格: ￥55.20元
书籍名称: 海贼王1     作者: 尾田荣一郎  价格: ￥39.20元

```

1.1.2 开闭原则的作用

开闭原则是面向对象程序设计的终极目标，它使软件实体拥有一定的适应性和灵活性的同时具备稳定性和延续性。具体来说，其作用如下。

1. 对软件测试的影响

软件遵守开闭原则的话，软件测试时只需要对扩展的代码进行测试就可以了，因为原有的测试代码仍然能够正常运行。

2. 可以提高代码的可复用性

粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。

3. 可以提高软件的可维护性

遵守开闭原则的软件，其稳定性高和延续性强，从而易于扩展和维护。

1.2 单一职责原则：Single responsibility principle, SRP

这是一个备受争议的原则，跟人吵架的时候这个是屡试不爽的一个梗。

为什么会备受争议呢？怎么就能吵起来呢？主要就是对职责如何定义，什么是类的职责，以及怎么划分类的职责。

举个栗子：我们新职课的老师对学生有很多的工作要做：例如了解个人信息、每天的学习情况、记录考勤；回答学生问题，帮助解决bug,重难点串讲；行业经验分享等。

如果将这些工作交给一位老师负责显然不合理，正确的做法是现在我们新课的模式：班主任负责日常工作，技术辅导老师负责技术辅导；企业师傅负责行业经验分享等。

1.2.1 单一职责原则的定义

单一职责原则（Single Responsibility Principle, SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。这里的职责是指类变化的原因，单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）。

该原则提出对象不应该承担太多职责，如果一个对象承担了太多的职责，至少存在以下两个缺点：

1. 一个职责的变化可能会削弱或者抑制这个类实现其他职责的能力；
2. 当客户端需要该对象的某一个职责时，不得不将其他不需要的职责全都包含进来，从而造成冗余代码或代码的浪费。

再举一个例子：

```
public interface IPhone{
    //拨通电话
    public void dial(String phoneNumber);
    //通话
    public void chat(Object o);
    //通话完毕，挂断电话
    public void hangup();
}
```

以上是符合单一职责原则的吗？说白了是一个接口只负责一件事情吗？是只有一个原因引起变化么？

好像不是哦！

其实他负责了两个内容：1、协议管理，2、数据传送。

dial()和hangup()两个方法实现的是协议管理；chat()方法负责的是数据的传送。那么协议的改变可能引起接口或者实现类的变化；同样数据传送（电话不仅可以打电话，还能上网）的变化也可能会引起接口或实现类的变化。两个原因都能引起变化，而两个职责直接是互不影响的，所以可以考虑拆分为两个接口。

```
public interface IPhone{
}
public interface IConnectionManager extends IPhone{
    //拨通电话
    public void dial(String phoneNumber);
    //通话完毕，挂断电话
    public void hangup();
}
public interface IDataTransfer extends IPhone{
    //通话
    public void chat(IConnectionManager con);
}
```

1.2.2 单一职责原则的优点

单一职责原则的核心就是控制类的粒度大小、将对象解耦、提高其内聚性。如果遵循单一职责原则将有以下优点。

- 降低类的复杂度。一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多。
- 提高类的可读性。复杂性降低，自然其可读性会提高。
- 提高系统的可维护性。可读性提高，那自然更容易维护了。
- 变更引起的风险降低。变更是必然的，如果单一职责原则遵守得好，当修改一个功能时，可以显著降低对其他功能的影响。

单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，再封装到不同的类或模块中。而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

PS：单一职责同样也适用于方法。一个方法应该尽可能做好一件事情。如果一个方法处理的事情太多，其颗粒度会变得很粗，不利于重用。

但是原则是死的，人是活的。所以有些时候我们可以为了效率，牺牲一定的原则性。

1.3 里氏替换原则：Liskov Substitution Principle, LSP

这是一个爱恨纠葛的父子关系的故事。该原则可以理解为：**子类可以替换父类。**

父子类实在我们学习继承这个知识点的时候学习到的概念。我们先来回忆一下继承的优缺点：

优点：

1. 代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性；
2. 提高代码的重用性；
3. 提高代码的可扩展性，子类可形似于父类，但异于父类，保留了自己独特的个性；其实很多开源框架的扩展都是通过继承父类实现的。
4. 提供产品或者项目的开放性。

缺点：

1. 继承是侵入性的，只要继承就必须拥有父类的所有方法和属性；
2. 降低了代码的灵活性。子类必须拥有父类的属性和方法，让子类中多了约束
3. 增加了耦合，当父类的常量、变量或者方法被修改了，需要考虑子类的修改，所以一旦父类有了变动，很可能造成非常糟糕的结果，要重构大量的代码。

java中使用extends关键字来实现继承，采用的是单一继承的规则，C++则采用了多重继承的规则，即一个子类可以继承多个父类。从整体上上看，利大于弊，怎么才能更大的发挥“利”的作用呢？

解决方案就是**引入里氏替换原则**。什么是里氏替换原则呢？

1.3.1 里氏替换原则的定义

该原则有两个定义：

- 第一种：If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T. (如果每一个类型S的对象o1，都有一个类型T的对象o2，在以T定义的所有程序P中将所有的对象o2都替换为o1，而程序P的行为没有发生变化，那么S是T的子类。)
- 第二种：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. (所有引用基类的地方必须能透明地使用其子类对象。)

第一个定义看起来有点难，其实意思就是说：在一个程序中，如果可以将一个类T的对象全部替换为另一个类S的对象，而程序的行为没有发生变化，那么S是T的子类。

第二个定义明显要比第一个定义更容易理解，非常的清晰明确。通俗地讲，就是任何一个使用父类的地方，你都可以把它替换成它的子类，而不会发生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必可以替换。

里氏替换原则是继承复用的基石，它为良好的继承定义了一个规范，定义中包含了4层含义：

1. 子类必须完全实现父类的方法。

我们以前做过的项目中，经常定义一个接口或者抽象类，然后编码实现，调用类则直接传入接口或者抽象类，其实这就是已经在使用历史替换原则了。

举个栗子：是不是很多人玩过CS(真人版cs或者游戏都算，没玩过也没关系，他就是一个比较火爆的第一人称射击游戏。你就知道用到了很多枪就行了)?

```
//枪的抽象类
public abstract class AbstractGun {
    //杀敌
    public abstract void shoot();
}
//手枪：携带方便但是射程短
public class Handgun extends AbstractGun{
    @Override
    public void shoot() {
        System.out.println("手枪射击-----");
    }
}
//步枪：威力大射程远
public class Rifle extends AbstractGun{
    @Override
    public void shoot() {
        System.out.println("步枪射击-----");
    }
}
//机枪：威力更大连续发射
public class MachineGun extends AbstractGun{
    @Override
    public void shoot() {
        System.out.println("机枪射击-----");
    }
}
//士兵：使用枪支
public class Soldier {
    //士兵使用的枪支
    private AbstractGun gun;
    //通过set方法给士兵配枪
    public void setGun(AbstractGun gun){
        this.gun=gun;
    }
    public void killEnemy(){
        System.out.println("士兵杀敌: ");
        gun.shoot();
    }
}
public class Client {
```

```

    public static void main(String[] args){
        //定义一个士兵许三多
        Soldier xuSanDuo=new Soldier();
        //给许三多配枪：参数可以是任何一把枪：机枪、步枪都可以
        xuSanDuo.setGun(new Handgan());
        xuSanDuo.killEnemy();
    }
}

```

运行结果：



在场景类中，给士兵配枪的时候可以是三种枪中的任何一个，其实士兵类可以不用知道是哪种的枪（子类）被传入。

PS：在类中调用其他类时务必要使用父类或接口，如果不能使用父类或者接口，说明类的设计违背了里氏替换原则。

2. 子类中可以增加自己特有的方法。

类都有自己的属性和方法，子类当然也不例外。除了从父类继承过来的，可以有自己独有的内容。为什么要单独列出来，是因为里氏替换原则是不可以反过来用的。也就是子类出现的地方，父类未必可以胜任。

我们继续上面的案例：步枪下面还有几个比较知名的种类：例如AK47和AUG狙击步枪。

```

//AUG狙击枪
public class AUG extends Rifle{
    //狙击枪都携带了精准望远镜
    public void zoomOut(){
        System.out.println("通过望远镜观察敌人：");
    }
    @Override
    public void shoot() {
        System.out.println("AUG射击-----");
    }
}
//狙击手
public class Sniper extends Soldier{
    public void killEnemy(AUG aug) {
        //先观察
        aug.zoomOut();
        //射击
        aug.shoot();
    }
}
//场景类：
public class Client {
    public static void main(String[] args){
        //定义一个狙击手韩光
        Sniper hanGuang=new Sniper();
        //给韩光配枪
        hanGuang.setGun(new AUG());
        hanGuang.killEnemy();
    }
}

```



```
}  
}
```

运行结果：



场景类中我们可以直接使用子类，狙击手是依赖枪支的，别说换一个型号的枪，就是同一个型号的枪都会影响射击，所以这里直接传递子类。

如果我们直接使用父类传递进来可以吗？

```
//使用父类作为参数  
public class Client {  
    public static void main(String[] args){  
        //定义一个狙击手韩光  
        Sniper hanGuang=new Sniper();  
        //给韩光配枪  
        hanGuang.setGun((AUG)new Rifle());  
        hanGuang.killEnemy();  
    }  
}
```

运行结果：



会在运行的时候抛出异常，这就是我们经常说的额向下转型是不安全的。从里氏替换原则来看：子类出现的地方，父类未必可以出现。

3. 当子类覆盖或实现父类的方法时，方法的输入参数（方法的形参）要比父类方法的输入参数更宽松。

```
public class LSP {  
    class Parent {  
        public void fun(HashMap map){  
            System.out.println("父类被执行...");  
        }  
    }  
  
    class Sub extends Parent{  
        public void fun(Map map){  
            System.out.println("子类被执行...");  
        }  
    }  
  
    public static void main(String[] args){  
        System.out.print("父类的运行结果：");  
        LSP lsp =new LSP();  
        Parent a= lsp.new Parent();  
        HashMap<Object, Object> map=new HashMap<Object, Object>();  
        a.fun(map);  
        //父类存在的地方，可以用子类替代  
        //子类B替代父类A
```

```

        System.out.print("子类替代父类后的运行结果: ");
        LSP.Sub b=lsp.new Sub();
        b.fun(map);
    }

}

```

运行结果:



ps:这里子类并非重写了父类的方法,而是重载了父类的方法。因为子类和父类的方法的输入参数是不同的。子类方法的参数Map比父类方法的参数HashMap的范围要大,所以当参数输入为HashMap类型时,只会执行父类的方法,不会执行父类的重载方法。这符合里氏替换原则。

4. 当子类的方法实现父类的抽象方法时,方法的后置条件(即方法的返回值)要比父类更严格。

```

public class LSP1 {
    abstract class Parent {
        public abstract Map fun();
    }

    class Sub extends Parent{
        @Override
        public HashMap fun(){
            HashMap b=new HashMap();
            b.put("b","子类被执行...");
            return b;
        }
    }

    public static void main(String[] args){
        LSP1 lsp =new LSP1();
        LSP1.Parent a=lsp.new Sub();
        System.out.println(a.fun());
    }
}

```

运行结果:



1.3.2 里氏替换原则的作用

里氏替换原则的主要作用如下。

1. 里氏替换原则是实现开闭原则的重要方式之一。
2. 它克服了继承中重写父类造成的可复用性变差的缺点。

3. 它是动作正确性的保证。即类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

里氏替换原则通俗来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。

如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

如果程序违背了里氏替换原则，则继承类的对象在基类出现的地方会出现运行错误。这时其修正方法是：取消原来的继承关系，重新设计它们之间的关系。

1.4 依赖倒置原则：Dependence Inversion Principle, DIP

1.4.1 依赖倒置原则的定义

依赖倒置原则的原始定义为:High level modules shouldnot depend upon low level modules.Both should depend upon abstractions.Abstractions should not depend upon details. Details should depend upon abstractions.

其实里面包含了三层含义：

- 高层模块不应该依赖低层模块，两者都应该依赖其抽象；
- 抽象不应该依赖细节，
- 细节应该依赖抽象。

核心思想：要面向接口编程，不要面向实现编程。

依赖倒置原则是实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合。

由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。

使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给它们的实现类去完成。

1.4.2 依赖倒置原则的作用

- 依赖倒置原则可以降低类间的耦合性。
- 依赖倒置原则可以提高系统的稳定性。
- 依赖倒置原则可以减少并行开发引起的风险。
- 依赖倒置原则可以提高代码的可读性和可维护性。

我们来举个栗子说明上面的这些作用。（我们反向证明一下）

现在汽车越来越便宜，所有人们出行的时候开车出行的越来越多。在此场景下，有了汽车，开车的人就是司机。

我们使用程序来描述一下：

```
//奔驰车
public class Benz {
    public void run(){
        System.out.println("奔驰汽车飞驰-----");
    }
}

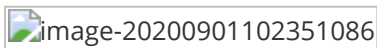
//司机
```

```

public class Driver {
    //司机驾驶汽车
    public void drive(Benz benz){
        System.out.print("司机开车: ");
        benz.run();
    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        Driver james=new Driver();
        Benz benz=new Benz();
        james.drive(benz);
    }
}

```

运行结果：



司机开奔驰车这个项目没有问题了。业务需求变更的时候更能发觉我们的设计或程序是否是松耦合。所以现在我们填个需求：司机不仅能开奔驰车，还能开宝马车，该怎么实现呢？

不管怎么实现，先把宝马车产出出来：

```

//宝马车
public class BMW {
    public void run(){
        System.out.println("宝马车飞驰-----");
    }
}

```

车有了，可是我们的司机james竟然没办法开动，james没有开动宝马的方法啊！拿了驾照只能开奔驰也不太合理吧！现实世界都不这样，更何况程序还是对现实世界的抽象呢。

很显然，我们的设计不合理：司机类和奔驰车类之间是紧耦合的关系，结果就是系统的可维护性降低、可阅读性降低，两个相似的类要阅读两个类；这里增加了一个车类就需要修改司机类，让程序变得不稳定。这样我们就证明了不使用依赖导致原则就没有前两个好处。

继续证明“减少并行开发的风险”，什么是并行开发风险？本来只是一段程序的错误或者异常，逐步波及一个功能、一个模块，甚至最后毁掉整个项目。为什么并行开发有这也有风险？假如一个团队有20个人，各人负责不同的功能模块，A负责汽车类，B负责司机类.....在A未完成的情况下，B不能完全地编写代码，因为缺少汽车类编译器编译根本通不过，就更不用说单元测试了！在这种不使用依赖倒置原则的情况下，所有的开发都是“单线程”的，只能是A做完B再做.....在早期的小型项目中还可以，但在现在的中大型项目中就不合适了，需要团队人员同时并行开发，所以这个时候依赖原则的作用就体现出来了。因为根据上面的案例已经说明不使用依赖倒置原则就会增加类直接的耦合性，降低系统的稳定性、可读性和维护性，增加了并行开发的风险。

咱们将上面的案例引入依赖倒置原则：

```

//汽车接口
public interface ICar {
    void run();
}

```

```

}
//奔驰车
public class Benz implements ICar{
    @Override
    public void run(){
        System.out.println("奔驰汽车飞驰-----");
    }
}
//宝马车
public class BMW implements ICar{
    @Override
    public void run(){
        System.out.println("宝马车飞驰-----");
    }
}
//司机接口
public interface IDriver {
    //司机驾驶汽车:通过传入ICar接口实现了抽象之间的依赖关系
    void drive(ICar car);
}
//司机类实现司机接口
public class Driver implements IDriver {
    //司机驾驶汽车:实现类也传入ICar接口,至于到底是哪个型号的车,需要在高层模块中声明
    @Override
    public void drive(ICar car){
        System.out.print("司机开车: ");
        car.run();
    }
}
//场景类:属于高层业务逻辑,他对底层的依赖都建立在抽象上
public class Client {
    public static void main(String[] args) {
        IDriver james=new Driver();
        ICar benz=new Benz();
        james.drive(benz);
    }
}

```

james和benz表明的类型都是接口，是抽象的，虽然在实例化对象的时候调用了低层模块，但是后续所有操作中，james都是以IDriver类型进行操作，屏蔽了细节对抽象的影响。

如果我们此时再新增一个低层模块，只修改业务场景类，也就是高层模块，对其它低层模块不需要做任何修改，业务依然可以运行，把变更引起的风险扩散降到最低。

依赖倒置对并行开发的影响。只要定义好了接口，即使负责Car开发的程序员工作滞后，我们依然可用进行测试。引入了JMock工具，根据抽象虚拟一个对象进行测试（不了解该测试工具也没关系，以后有机会再了解）。

```

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JUnit4Mockery;
import junit.framework.TestCase;
import org.junit.Test;

```

```

public class DriverTest extends TestCase {
    Mockery context=new JUnit4Mockery();

    @Test
    public void test1(){
        final ICar car=context.mock(ICar.class);
        IDriver driver=new Driver();
        context.checking(new Expectations(){
            {
                oneOf(car).run();
            }
        });
        driver.drive(car);
    }
}

```

1.4.3 依赖倒置原则的实现方法

依赖倒置原则的目的是通过要面向接口的编程来降低类间的耦合性，所以我们在实际编程中只要遵循以下4点，就能在项目中满足这个规则。

1. 每个类尽量提供接口或抽象类，或者两者都具备。
2. 变量的声明类型尽量是接口或者是抽象类。
3. 任何类都不应该从具体类派生。
4. 尽量不要覆写基类的方法
5. 使用继承时结合里氏替换原则。

1.5 接口隔离原则:Interface Segregation Principle, ISP

1.5.1 接口隔离原则的定义

接口隔离原则要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。

2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法（Clients should not be forced to depend on methods they do not use）。该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上（The dependency of one class to another one should depend on the smallest possible interface）。

以上两个定义的含义是：要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

接口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想，但两者是不同的：

- 单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离。
- 单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。

我们举个栗子说明一下：现在有个特别流行的词:小姐姐。我觉得这是对美女的别称。什么是美女呢？一般来说：首先长得好看，其次身材窈窕，最后还有气质。我们按照这三种要求去寻求美女(也过一把星探的瘾)。

```

//美女接口
public interface IPettyGirl {

```

```

//天使面孔
void goodLooking();
//魔鬼身材
void niceFigure();
//有气质
void greatTemperament();
}
//美女实现类
public class PettyGirl implements IPettyGirl{

    private String name;

    public PettyGirl(String name) {
        this.name = name;
    }

    @Override
    public void goodLooking() {
        System.out.println(this.name+"面容：倾国倾城比天仙");
    }

    @Override
    public void niceFigure() {
        System.out.println(this.name+"身材：窈窕淑女赛西施");
    }

    @Override
    public void greatTemperament() {
        System.out.println(this.name+"气质：亭亭玉立似贵妃");
    }
}
//星探抽象类
public abstract class AbstractSearcher {
    protected IPettyGirl girl;

    public AbstractSearcher(IPettyGirl girl) {
        this.girl = girl;
    }
    //搜索美女并展示信息
    public abstract void search();
}
//星探类
public class Searcher extends AbstractSearcher{
    public Searcher(IPettyGirl girl) {
        super(girl);
    }

    @Override
    public void search() {
        System.out.println("搜索到的美女展示如下：");
        super.girl.goodLooking();
        super.girl.niceFigure();

        super.girl.greatTemperament();
    }
}

```

```

    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        //定义一个美女
        IPettyGirl reBa=new PettyGirl("迪丽热巴");
        AbstractSearcher searcher=new Searcher(reBa);
        searcher.search();
    }
}

```

运行结果：



我们的审美观点一直在改变，美女的定义也在变化。例如看到一个女孩：身材面容一般，但是气质特别好，我们也会把她成为美女，于是产生了气质美女，但是我们的接口中却定义了美女都必须具备三个条件，按照这个标准，气质美女就不是美女了，怎么办呢？

重新扩展一个美女类，只实现气质方法，其他两个方法置空就好啦！可是星探类AbstractSearcher依赖的是IPettyGirl接口，它有3个方法，星探的方法是不是要修改？

分析到这里，我们发现其实是IPettyGirl的设计有缺陷，过于庞大，容纳了一些可变的因素，根据接口隔离的原则，星探AbstractSearcher应该依赖具有部分特质的女孩子，而我们却把这些都放到了一个接口中。

找到问题原因，接下来就解决问题：我们把臃肿的接口分为两个：一个面容身材好接口，一个气质好接口。

```

public interface IGoodBodyGirl {
    //天使面孔
    void goodLooking();
    //魔鬼身材
    void niceFigure();
}
public interface IGreatTemperamentGirl {
    //有气质
    void greatTemperament();
}
//美女实现类:可以根据需求自行选择实现一个或者多个接口
public class PettyGirl implements IGoodBodyGirl,IGreatTemperamentGirl{

    private String name;

    public PettyGirl(String name) {
        this.name=name;
    }

    @Override
    public void goodLooking() {
        System.out.println(this.name+"面容：倾国倾城比天仙");
    }

    @Override

```



```
public void niceFigure() {  
    System.out.println(this.name+"身材：窈窕淑女赛西施");  
}  
  
@Override  
public void greatTemperament() {  
    System.out.println(this.name+"气质：亭亭玉立似贵妃");  
}  
}
```

把一个臃肿的接口变为两个独立的接口所依赖的原则就是接口隔离原则。

1.5.2 接口隔离原则的优点

接口隔离原则是为了约束接口、降低类对接口的依赖性，遵循接口隔离原则有以下 5 个优点。

1. 将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
2. 接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。
3. 如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。
4. 使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。
5. 能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

1.5.3 接口隔离原则的实现方法

在具体应用接口隔离原则时，应该根据以下几个规则来衡量。

- 接口尽量小，但是要有限度。一个接口只服务于一个子模块或业务逻辑。
- 为依赖接口的类定制服务。只提供调用者需要的方法，屏蔽不需要的方法。
- 了解环境，拒绝盲从。每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。
- 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

1.6 迪米特法则：Law of Demeter, LoD

1.6.1 迪米特法则的定义

迪米特法则（Law of Demeter, LoD）又叫作最少知识原则（Least Knowledge Principle, LKP），产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，它要求**一个对象应该对其他对象有最少的了解**。通俗的说，一个类应该对自己需要耦合或调用的类知道的最少，被耦合或调用的类的内部是如何复杂都与我无关，我就知道你提供的public方法就好。

迪米特法则还是在讲如何减少耦合的问题，类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。也就是说，信息的隐藏促进了软件的复用。

迪米特法则还有一个定义是：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

什么叫做直接的朋友呢？每个对象都必然会和其他对象有耦合关系，两个对象之间的耦合就 成为朋友关系，这种关系有很多比如组合、聚合、依赖等等。包括以下几类：

1. 当前对象本身 (this)
2. 当前对象的方法参数 (以参数形式传入到当前对象方法中的对象)
3. 当前对象的成员对象
4. 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友
5. 当前对象所创建的对象

1.6.2 迪米特法则的优点

- 降低了类之间的耦合度，提高了模块的相对独立性。
- 由于亲合度降低，从而提高了类的可复用率和系统的扩展性。

但是，过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。所以，在采用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。

1.6.3 迪米特法则的实现方法

从迪米特法则的定义和特点可知，它强调以下两点：

1. 从依赖者的角度来说，只依赖应该依赖的对象。
2. 从被依赖者的角度说，只暴露应该暴露的方法。

所以，在运用迪米特法则时要注意以下 6 点。

1. 在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标。
2. 在类的结构设计上，尽量降低类成员的访问权限。
3. 在类的设计上，优先考虑将一个类设置成不变类。
4. 在对其他类的引用上，将引用其他对象的次数降到最低。
5. 不暴露类的属性成员，而应该提供相应的访问器 (set 和 get 方法)。
6. 谨慎使用序列化 (Serializable) 功能。

举个栗子：明星平时档期都很满，例如拍电影、演出、粉丝见面会等等，那么他们的这些日程是怎么来的呢？一般都是由经纪人负责处理。这里的经纪人是明星的朋友，而见面会上的粉丝和拍电影或举办演出的媒体公司是陌生人，所以适合使用迪米特法则。

```
//场景类
public class Client{
    public static void main(String[] args){
        Agent agent=new Agent();
        agent.setStar(new Star("迪丽热巴"));
        agent.setFans(new Fans("小九"));
        agent.setCompany(new Company("荔枝电视台"));
        agent.meeting();
        agent.business();
    }
}
//经纪人
class Agent{
    private Star star;
    private Fans fans;

    private Company company;
```

```

        public void setStar(Star star) {
            this.star = star;
        }

        public void setFans(Fans fans) {
            this.fans = fans;
        }

        public void setCompany(Company company) {
            this.company = company;
        }

        public void meeting(){
            System.out.println(this.fans.getName()+"与明星"+this.star.getName()+"见面了。");
        }
        public void business() {
            System.out.println(this.company.getName()+"与明星"+this.star.getName()+"商谈合作。");
        }
    }
    //明星
    class Star{
        private String name;
        public Star(String name){
            this.name=name;
        }
        public String getName(){
            return name;
        }
    }
    //粉丝
    class Fans{
        private String name;
        public Fans(String name){
            this.name=name;
        }
        public String getName(){
            return name;
        }
    }
    //公司
    class Company{
        private String name;
        public Company(String name){
            this.name=name;
        }
        public String getName(){
            return name;
        }
    }
}

```

运行结果：

2、设计模式之创建型模式

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。这样可以降低系统的耦合度，使用者不需要关注对象的创建细节，对象的创建由相关的工厂来完成。就像我们去商场购买商品时，不需要知道商品是怎么生产出来一样，因为它们由专门的厂商生产。

创建型模式分为以下几种。



2.1 单例模式

2.1.1 什么是单例模式？（概念的引入）

案例：我是皇帝，独此一家。

中国自从秦始皇确立了皇帝这个职位之后，同一个时期基本上就只有一个人孤零零的坐在皇位上啦。这种情况的好处就是大家好办事，大家讨论或者汇报大事的时候只要提及皇帝，每个人都知道指的是谁，不需要在皇帝面前加上特定的称呼。这种过程反应到软件设计领域就是：一个类只能产生一个对象（皇帝），大家对他的依赖都是相同的。我们把皇帝这种特殊的职位通过程序来实现。

皇帝类：

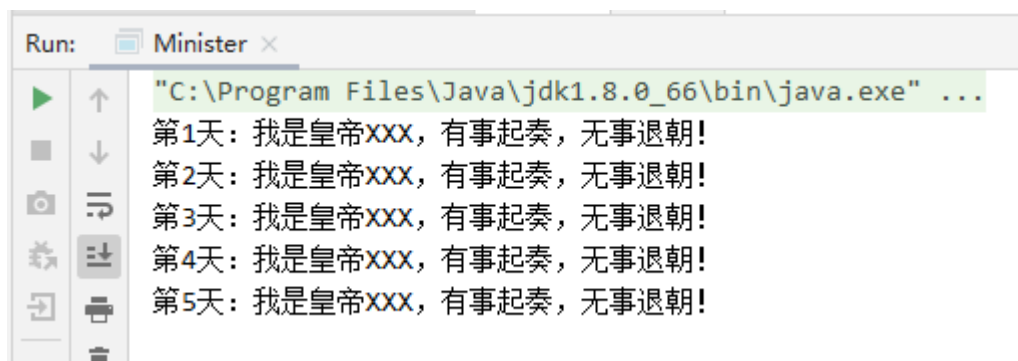
```
public class Emperor {
    private static Emperor emperor=null;
    //构造方法私有，避免在类的外部创建对象
    private Emperor() {
    }
    public static Emperor getInstance(){
        if(emperor==null) {
            emperor=new Emperor();
        }
        return emperor;
    }
    //皇帝办公
    public void work(){
        System.out.println("我是皇帝xxx，有事起奏，无事退朝！");
    }
}
```

```
}
```

大臣类：

```
public class Minister {  
    public static void main(String[] args) {  
        //每天见面的都是同一个皇帝  
        for (int i=1;i<=5;i++){  
            Emperor emperor=Emperor.getInstance();  
            System.out.print("第"+i+"天: ");  
            emperor.work();  
        }  
    }  
}
```

运行结果：



```
Run: Minister x  
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...  
第1天: 我是皇帝XXX, 有事起奏, 无事退朝!  
第2天: 我是皇帝XXX, 有事起奏, 无事退朝!  
第3天: 我是皇帝XXX, 有事起奏, 无事退朝!  
第4天: 我是皇帝XXX, 有事起奏, 无事退朝!  
第5天: 我是皇帝XXX, 有事起奏, 无事退朝!
```

大臣每天上朝汇报国事的对象都是同一个皇帝，这就是单例模式！

2.1.2 单例模式的定义以及特点

定义：

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。
Ensure a class has only one instance, and provide a global point of access to it.

特点：

- 单例类只有一个实例对象；
- 该单例对象必须由单例类自行创建；
- 单例类对外提供一个访问该单例的全局访问点。

2.1.3 单例模式的分类

上面的单例模式，在低并发的情况下可能不会出现问题，如果并发量增大，内存中就会出现多个实例，就不是真正意义上的单例。为什么会出现这种情况呢？（视频录制中语言解释）

解决线程不安全的方式有多种。我们先将上面代码的单例模式修改为线程安全的：

懒汉式单例：

```

public class LazyEmperor {
    //保证emperor在所有线程中同步
    private static volatile LazyEmperor emperor=null;
    //构造方法私有，避免在类的外部创建对象
    private LazyEmperor() {
    }
    public static synchronized LazyEmperor getInstance(){
        if(emperor==null) {
            emperor=new LazyEmperor();
        }
        return emperor;
    }
    //皇帝办公
    public void work(){
        System.out.println("我是皇帝xxx，有事起奏，无事退朝！");
    }
}

```

该模式的特点是类加载时没有生成单例，只有当第一次调用 `getInstance` 方法时才去创建这个单例。

关键字 `volatile` 和 `synchronized`，能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源，这是懒汉式单例的缺点。

饿汉式单例：

```

public class HungryEmperor {
    //实例化一个皇帝
    private static final HungryEmperor emperor=new HungryEmperor();
    //构造方法私有，避免在类的外部创建对象
    private HungryEmperor(){}

    public static HungryEmperor getInstance(){
        return emperor;
    }

    //皇帝办公
    public void work(){
        System.out.println("我是皇帝xxx，有事起奏，无事退朝！");
    }
}

```

该模式的特点是类一旦加载就创建一个单例，保证在调用 `getInstance` 方法之前单例已经存在了。而且该方式是线程安全的。

2.1.4 单例模式的使用场景

- 某类只要求生成一个对象的时候，如一个航班的机长、每个人的身份证号等。
- 当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。
- 当某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

- 在计算机系统中，Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

2.1.5 单例模式的优缺点

优点：

- 在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。
- 避免对资源的多重占用（比如写文件操作）。
- 单例模式可以在系统设置全局的访问点，优化和共享资源访问。

缺点：

- 单例模式一般没有接口，扩展很困难。如果要扩展，只能修改代码。
- 与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

2.1.6 单例模式的扩展

单例模式可扩展为有限的多例（Multitcm）模式，这种模式可生成有限个实例并保存在 ArrayList 中，客户需要时可随机获取。

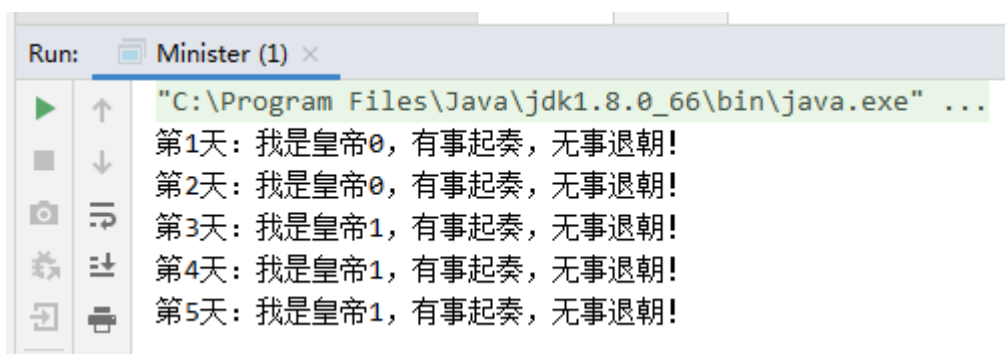
```
public class Emperor {
    //皇帝的姓名属性
    private String name;
    //定义最多能产生的实例的数量
    private static final int maxNum=2;
    //定义盛放所有皇帝实例的列表
    private static ArrayList<Emperor> list=new ArrayList<Emperor>(2);
    //产生所有的皇帝对象
    static {
        for (int i=0;i<maxNum;i++){
            list.add(new Emperor("皇帝"+i));
        }
    }
    //构造方法私有，避免在类的外部创建对象
    private Emperor() {
    }
    private Emperor(String name) {
        this.name=name;
    }
    //随机获取一个皇帝对象
    public static Emperor getInstance(){
        int index=new Random().nextInt(maxNum);
        return list.get(index);
    }

    //皇帝办公
    public void work(){
        System.out.println("我是"+name+", 有事起奏，无事退朝！");
    }
}
```

大臣类：

```
public class Minister {  
    public static void main(String[] args) {  
        //每天见面的都是随机的皇帝  
        for (int i=1;i<=5;i++){  
            Emperor emperor= Emperor.getInstance();  
            System.out.print("第"+i+"天: ");  
            emperor.work();  
        }  
    }  
}
```

运行结果：PS：每个人的运行结果都不一样，因为随机数的产生不一致



```
Run: Minister (1) x  
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...  
第1天: 我是皇帝0, 有事起奏, 无事退朝!  
第2天: 我是皇帝0, 有事起奏, 无事退朝!  
第3天: 我是皇帝1, 有事起奏, 无事退朝!  
第4天: 我是皇帝1, 有事起奏, 无事退朝!  
第5天: 我是皇帝1, 有事起奏, 无事退朝!
```

2.2 工厂方法模式

2.2.1 什么是工厂模式？

引入案例：话说前一阵疫情期间，居家的时间有点长，于是我也跟广大网友一样，开始好好修炼自己的厨艺，有人做凉皮，有人做锅巴。我也加入了霍霍面粉的大军--做面包。

于是和面、发面、捏成我想要的布朗熊的样子，开心放入烤箱，等待我的布朗熊面包出炉。然而，步骤没错，烤箱没错，可能是大厨不对：

第一次：时间稍短了点，没太烤熟，于是布朗熊面包变成了北极熊面包；

再来一次：时间长点肯定能熟了，烤箱中多靠一会，出炉发现烤焦啦，与布朗熊面包变成了黑熊面包；

第三次：吸取教训，别跟时间死磕了，一直盯着烤箱中的面包好了，等到面包微微发焦，终于成功的做出了我想要的布朗熊面包。

好在家人给面子，每一种面包都有人吃掉哈。

在这个过程中，我的职业病就犯了，是不是可以通过软件开发来实现这个过程呢？

在面向对象的思想中，万物皆对象。是对象我们就可以通过软件设计来实现。来分析一下烤面包的过程，该过程涉及三个对象：大厨（也就是我哈）、烤箱、三种不同成果的面包（我称他们为北极熊、黑熊、布朗熊）。

大厨可以使用场景类Client来表示，烤箱类似一个工厂，负责生产产品（即面包），三种不同成果的面包都是一个接口下不同实现类，因为好不好吃好不好看也都是面包啊。

面包接口：

```
public interface Bread {  
    //每个面包都有自己的颜色  
    public void getColor();  
}
```

北极熊面包：

```
public class PolarBearBread implements Bread{  
    public void getColor() {  
        System.out.println("烤的时间有点短的-----北极熊面包!!!");  
    }  
}
```

黑熊面包：

```
public class BlackBearBread implements Bread{  
    public void getColor() {  
        System.out.println("烤的时间有点长的-----黑熊面包!!!");  
    }  
}
```

布朗熊面包：

```
public class BrownBearBread implements Bread{  
    public void getColor() {  
        System.out.println("烤的时间刚刚好的-----布朗熊面包!!!");  
    }  
}
```

抽象面包创建工厂：

```
public abstract class AbstractBreadFactory {  
    public abstract Bread createBread(Class cls);  
}
```

具体面包创建工厂：

```
public class BreadFactory extends AbstractBreadFactory {  
    public Bread createBread(Class cls){  
        //定义一个生产的面包  
        Bread bread=null;  
        try {  
            //产生一个面包  
            bread= (Bread) Class.forName(cls.getName()).newInstance();  
        } catch (Exception e) {
```

```

        e.printStackTrace();
        System.out.println("烤面包出错了? ! ");
    }
    return bread;
}
}

```

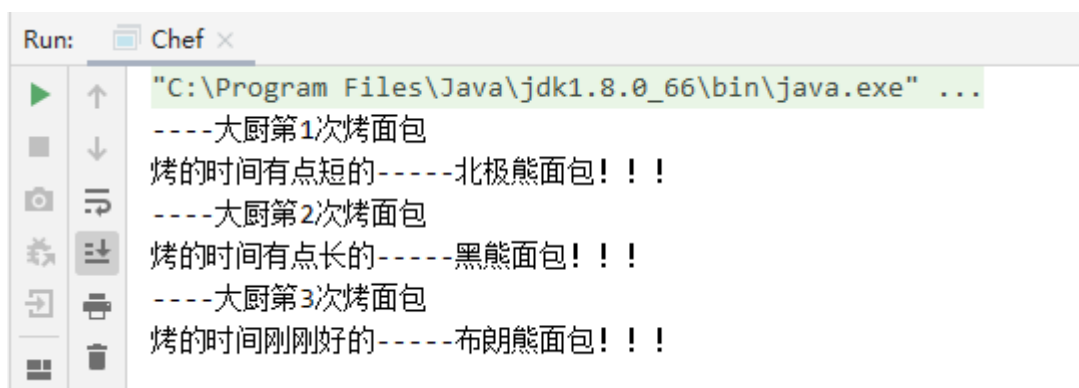
大厨：

```

public class Chef {
    public static void main(String[] args) {
        //来一个helloKitty烤箱
        AbstractBreadFactory kitty=new BreadFactory();
        //大厨第1次烤面包稍显急躁，时间不足，于是产生了北极熊面包
        System.out.println("----大厨第1次烤面包");
        Bread polarBearBread=kitty.createBread(PolarBearBread.class);
        polarBearBread.getColor();
        //大厨第2次烤面包耐心十足，时间过长，于是产生了黑熊面包
        System.out.println("----大厨第2次烤面包");
        Bread blackBearBread=kitty.createBread(BlackBearBread.class);
        blackBearBread.getColor();
        //大厨第3次烤面包机智无比，时间刚刚好，于是产生了布朗熊面包
        System.out.println("----大厨第3次烤面包");
        Bread brownBearBread=kitty.createBread(BrownBearBread.class);
        brownBearBread.getColor();
    }
}

```

万事俱备，见证奇迹的时刻：



```

Run: Chef x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
----大厨第1次烤面包
烤的时间有点短的-----北极熊面包!!!
----大厨第2次烤面包
烤的时间有点长的-----黑熊面包!!!
----大厨第3次烤面包
烤的时间刚刚好的-----布朗熊面包!!!

```

2.2.2 工厂方法模式的定义

定义：

定义一个用户创建对象的接口，让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子类。

工厂方法模式的主要角色如下：

1. 抽象工厂 (Abstract Factory) : 提供了创建产品的接口, 调用者通过它访问具体工厂的工厂方法 newProduct() 来创建产品。
2. 具体工厂 (ConcreteFactory) : 主要是实现抽象工厂中的抽象方法, 完成具体产品的创建。
3. 抽象产品 (Product) : 定义了产品的规范, 描述了产品的主要特性和功能。
4. 具体产品 (ConcreteProduct) : 实现了抽象产品角色所定义的接口, 由具体工厂来创建, 它同具体工厂之间一一对应。

我们可以将工厂方法模式抽取出一个实用的通用代码:

```
//抽象产品: 提供了产品的接口
public interface Product{
    public void method();
}
//具体的产品可以有多个, 都实现抽象产品接口
public class ConcreteProduct1 implements Product{
    public void method(){
        //具体业务逻辑处理, 例如
        System.out.println("具体产品1显示...");
    }
}
public class ConcreteProduct2 implements Product{
    public void method(){
        //具体业务逻辑处理, 例如
        System.out.println("具体产品2显示...");
    }
}
//抽象工厂: 负责定义产品对象的产生
public abstract class AbstractFactory{
    //创建一个产品对象, 输入的参数类型可以自行设置
    public abstract <T extends Product>T createProduct(Class<T> tClass);
}
//具体工厂: 具体如何生产一个产品的对象, 是由具体的工厂类实现的
public class ConcreteFactory implements AbstractFactory{
    public <T extends Product> T createProduct(Class<T> tClass) {
        Product product=null;
        try {
            product=(T)Class.forName(tClass.getName()).newInstance();
        } catch (Exception e) {
            //异常处理
        }
        return (T)product;
    }
}
//场景类:
public class Client {
    public static void main(String[] args) {
        AbstractFactory factory=new ConcreteFactory();
        Product product=factory.createProduct(ConcreteProduct1.class);
        //继续其他业务处理
    }
}
```

2.2.3 工厂方法模式的应用场景

工厂方法模式通常适用于以下场景。

- 客户只知道创建产品的工厂名，而不知道具体的产品名。如 TCL 电视工厂、海信电视工厂等。
- 创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口。
- 客户不关心创建产品的细节，只关心产品的品牌。

2.3 抽象工厂模式

2.3.1 引入案例：

上回书说到我的烤面包副业已经小有成就，做的多了发现我的面包似乎缺少了灵魂，面包怎么能没有馅儿呢？！于是打算将自己最爱的水果菠萝和芒果放入面包中。

大厨烤面包之前也是做了很多的准备工作的，所以想在不浪费现有资源的情况下继续完成新产品的制作。之前做的面包中虽然大厨自己觉得布朗熊面包才是最成功的，但是每个人的口味不一样，有人偏爱烤的过火的，有的偏爱稍欠火候的。所以我决定继续满足所有人的口味：即将做出三种火候的菠萝面包和芒果面包。

即将要做的产品分析完了，可是我的工厂（面包机）真真只有一个烤面包的功能，于是为了做出蛋糕，又专门买了一个烘焙蛋糕的烤箱。于是乎，可以准备干活啦！

```
public interface Bread {
    //每个面包根据烤的时间长短都有自己的颜色
    public void getColor();
    //每个面包都有不同的馅儿
    public void getType();
}

public abstract class AbstractPolarBearBread implements Bread{
    public void getColor() {
        System.out.println("烤的时间有点短的-----北极熊面包!!!");
    }
}

public abstract class AbstractBlackBearBread implements Bread {
    public void getType() {
        System.out.println("烤的时间有点长的-----黑熊面包!!!");
    }
}

public abstract class AbstractBrownBearBread implements Bread{
    public void getColor() {
        System.out.println("烤的时间刚刚好的-----布朗熊面包!!!");
    }
}

public interface BreadFactory{
    //创建北极熊面包
    public Bread createPolarBearBread();
    //创建黑熊面包
    public Bread createBlackBearBread();
    //创建布朗熊面包
    public Bread createBrownBearBread();
}
```

```

public class MangoFactroy implements BreadFactory {
    //生产一个北极熊芒果面包
    @Override
    public Bread createPolarBearBread() {
        return new MangoPolarBearBread();
    }
    //生产一个黑熊芒果面包
    @Override
    public Bread createBlackBearBread() {
        return new MangoBlackBearBread();
    }
    //生产一个布朗熊芒果面包
    @Override
    public Bread createBrownBearBread() {
        return new MangoBrownBearBread();
    }
}

public class PineappleFactroy implements BreadFactory {
    //生产一个北极熊菠萝面包
    @Override
    public Bread createPolarBearBread() {
        return new PineapplePolarBearBread();
    }
    //生产一个黑熊菠萝面包
    @Override
    public Bread createBlackBearBread() {
        return new PineappleBlackBearBread();
    }
    //生产一个布朗熊菠萝面包
    @Override
    public Bread createBrownBearBread() {
        return new PineappleBrownBearBread();
    }
}

//大厨
public class Chef {
    public static void main(String[] args) {
        //来一个专门烤菠萝面包的烤箱
        BreadFactory pineapple=new PineappleFactroy();
        //来一个专门烤芒果面包的烤箱
        BreadFactory mango=new PineappleFactroy();
        //两个烤箱准备完毕，开始生产面包啦：
        System.out.println("生产的第1批面包：");
        Bread polarBearBread1=pineapple.createPolarBearBread();
        Bread polarBearBread2=mango.createPolarBearBread();
        polarBearBread1.getColor();
        polarBearBread1.getType();
        polarBearBread2.getColor();
        polarBearBread2.getType();
        System.out.println("生产的第2批面包：");
        Bread polarBearBread21=pineapple.createPolarBearBread();

        Bread polarBearBread22=mango.createPolarBearBread();
    }
}

```

```

        polarBearBread21.getColor();
        polarBearBread21.getType();
        polarBearBread22.getColor();
        polarBearBread22.getType();
        System.out.println("生产的第3批面包: ");
        Bread polarBearBread31=pineapple.createPolarBearBread();
        Bread polarBearBread32=mango.createPolarBearBread();
        polarBearBread31.getColor();
        polarBearBread32.getType();
        polarBearBread31.getColor();
        polarBearBread32.getType();
    }
}

```

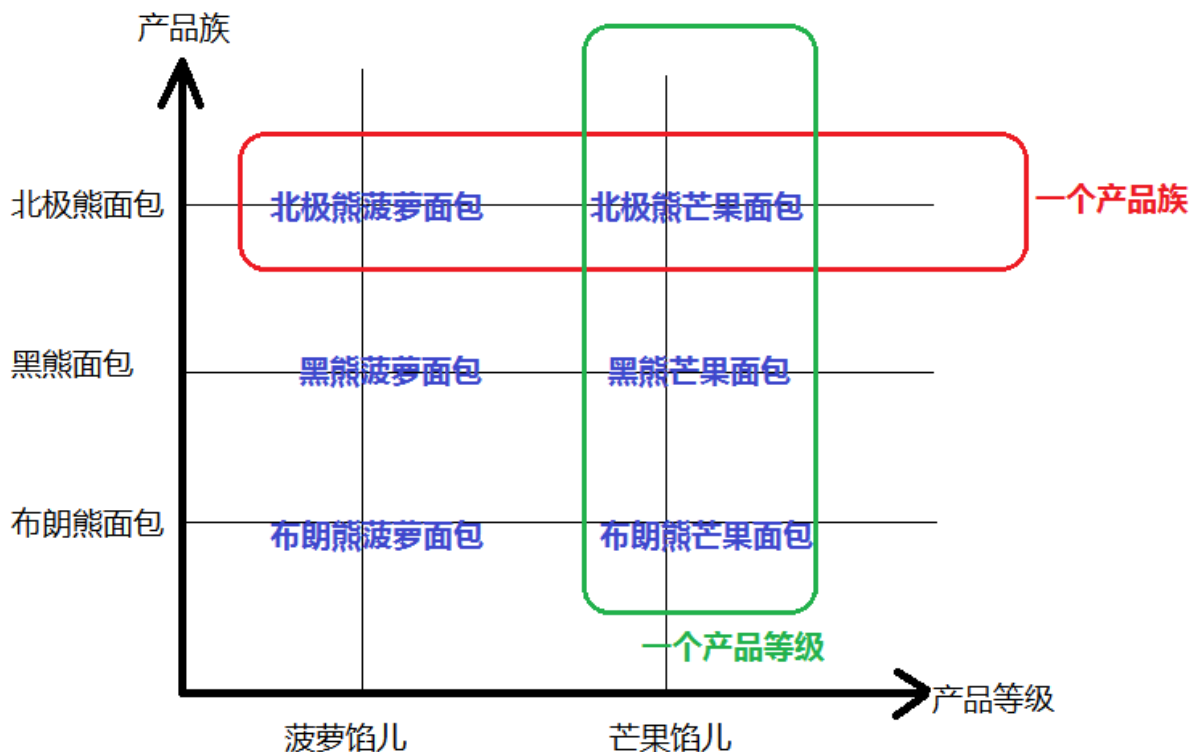
运行结果:

```

Run: Chef x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
生产的第1批面包:
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包
生产的第2批面包:
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包
生产的第3批面包:
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包
烤的时间有点短的-----北极熊面包!!!
北极熊---菠萝面包

```

所有产品都出炉啦!



工厂方法模式只考虑生产同等级的产品，抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族。

2.3.2 模式的定义与特点

抽象工厂（AbstractFactory）模式的定义：为创建一组相关或者相互依赖的对象提供一个接口，而且无须指定他们的具体类。

抽象工厂模式是工厂方法模式的升级版，在有多多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方案。工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

使用抽象工厂模式一般要满足以下条件。

- 系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品。
- 系统一次只可能消费其中某一族产品，即同族的产品一起使用。

2.3.3 抽象工厂的通用代码

```
//抽象产品类(只写了一个AbstractProductA, AbstractProductB省略):
public abstract class AbstractProductA{
    //每个产品的共有方法
    public void sharedMthod(){
    }
    //每个产品相同方法, 不同实现
    public abstract void doSomething();
}
具体产品类:
public class ProductA1 extends AbstractProductA{
    public abstract void doSomething(){
        System.out.println("产品A1的实现方法");
    }
}
```

```

    }
}
public class ProductA2 extends AbstractProductA{
    public abstract void doSomething(){
        System.out.println("产品A2的实现方法");
    }
}
//抽象工厂类:
public abstract class AbstractCreator{
    //创建A产品家族
    public abstract AbstractProductA createProductA();
    //创建B产品家族
    public abstract AbstractProductB createProductB();
    //如果有N个产品族, 该类中应该有N个创建方法
}
//产品等级实现类:
//有M个产品等级就应该有M个工厂的实现类, 在每个实现工厂中, 实现不同产品族的生产业务。
public class Creator1 extends AbstractCreator{
    //只生成产品等级为1的A产品
    public AbstractProductA createProductA(){
        return new ProductA1();
    }
    //只生成产品等级为1的B产品
    public AbstractProductB createProductB(){
        return new ProductB1();
    }
}
public class Creator2 extends AbstractCreator{
    //只生成产品等级为2的A产品
    public AbstractProductA createProductA(){
        return new ProductA2();
    }
    //只生成产品等级为2的B产品
    public AbstractProductB createProductB(){
        return new ProductB2();
    }
}
//场景类
public class Client{
    public static void main(String[] args){
        //定义两个工厂
        AbstractCreator creator1=new Creator1();
        AbstractCreator creator2=new Creator2();
        //产生A1对象
        AbstractProductA a1=creator1.createProductA();
        //产生A2对象
        AbstractProductA a2=creator2.createProductA();
        //产生B1对象
        AbstractProductA a1=creator1.createProductB();
        //产生B2对象
        AbstractProductA a2=creator2.createProductB();
        //按需求自己实现其他
    }
}

```



```
}
```

在场景类中，没有任何一个方法与实现类有关系，对于一个产品，我们只需要知道它的工厂方法就可以直接生产一个产品对象，无须关系它的实现类。

2.3.4 抽象工厂模式的优缺点：

优点：

- 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
- 当增加一个新的产品族时不需要修改原代码，满足开闭原则。

其缺点是：当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

2.3.5 抽象工厂模式的应用场景

1. 适合于产品之间相互关联、相互依赖且相互约束的地方
2. 需要动态切换产品族的地方

2.4 建造者 (Builder) 模式

2.4.1 案例引入

老板：又签订了一个新合同，XX公司将宝马和奔驰两款车辆模型都交给我们公司制作了。不过这次有了新的需求：汽车的启动、停止、鸣笛、引擎声音都由客户自己控制，他们想要什么顺序就什么顺序，OK吗？

我：OK!

来分析一下需求：宝马和奔驰两款车辆模型都是产品，他们有共有的属性，XX公司关心的是每个模型的运行过程，期待奔驰A模型先有引擎声音再鸣笛，奔驰B模型是先启动再有引擎声音。老板的意思就是满足客户所有要求，要什么顺序立刻就生成什么顺序的模型出来，而且能批量生成出来。

使用程序模拟实现这一需求：

```
//车辆模型的抽象类
public abstract class CarModel {
    //基本方法的执行属性
    private ArrayList<String> sequence=new ArrayList<>();
    //设置方法的执行顺序
    final public void setSequence(ArrayList<String> sequence){
        this.sequence=sequence;
    }
    //启动
    protected abstract void start();
    //停止
    protected abstract void stop();
    //鸣笛
    protected abstract void alarm();
    //引擎轰鸣
    protected abstract void engineBoom();
    //汽车能跑
    final public void run(){
        for (String s : sequence) {
            if("start".equalsIgnoreCase(s)){
                this.start();
            }
        }
    }
}
```

```

        }else if("stop".equalsIgnoreCase(s)){
            this.stop();
        }
        else if("alarm".equalsIgnoreCase(s)){
            this.alarm();
        }
        else if("engineBoom".equalsIgnoreCase(s)){
            this.engineBoom();
        }
    }
}

```

//奔驰车模型

```

public class BenzModel extends CarModel {
    @Override
    protected void start() {
        System.out.println("奔驰车启动-----");
    }

    @Override
    protected void stop() {
        System.out.println("奔驰车停止-----");
    }

    @Override
    protected void alarm() {
        System.out.println("奔驰车鸣笛-----");
    }

    @Override
    protected void engineBoom() {
        System.out.println("奔驰车引擎轰鸣-----");
    }
}

```

//宝马车模型

```

public class BMWModel extends CarModel {
    @Override
    protected void start() {
        System.out.println("宝马车启动-----");
    }

    @Override
    protected void stop() {
        System.out.println("宝马车停止-----");
    }

    @Override
    protected void alarm() {
        System.out.println("宝马车鸣笛-----");
    }

    @Override
    protected void engineBoom() {

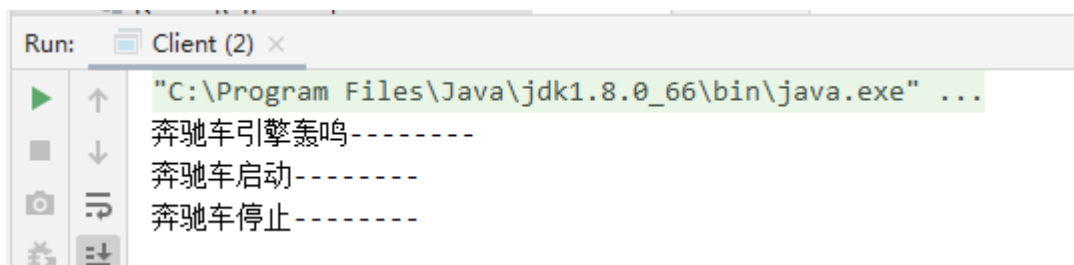
```

```

        System.out.println("宝马车引擎轰鸣-----");
    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        BenzModel benz=new BenzModel();
        //存放run的顺序
        ArrayList<String> sequence=new ArrayList<>();
        sequence.add("engineBoom");
        sequence.add("start");
        sequence.add("stop");
        benz.setSequence(sequence);
        benz.run();
    }
}

```

运行结果:



看到运行结果，满足了一个需求，如果还有更多的不同顺序的需求怎么办呢？不停的写场景类来满足吗？很显然这是一个问题，所以我们就要想一种方案来解决这个问题。

我们为每种产品模型定义一个建造者，要创建什么顺序直接通知建造者，由建造者来建造。使用程序模拟一下：

```

//抽象汽车组装者
public abstract class CarBuilder {
    //给我一个组装顺序，按照要求的顺序建造模型
    public abstract void setSequence(ArrayList<String> sequence);
    //设置好顺序后直接获取到车辆模型
    public abstract CarModel getCarModel();
}
//奔驰车组装者
public class BenzBuilder extends CarBuilder {
    private BenzModel benz=new BenzModel();
    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.benz.setSequence(sequence);
    }

    @Override
    public CarModel getCarModel() {
        return benz;
    }
}
//宝马车组装者

```

```

public class BMWBuilder extends CarBuilder {
    private BMWModel bmw=new BMWModel();
    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.bmw.setSequence(sequence);
    }

    @Override
    public CarModel getCarModel() {
        return bmw;
    }
}
//修改后的场景类
public class Client {
    public static void main(String[] args) {
        //存放run的顺序
        ArrayList<String> sequence=new ArrayList<>();
        sequence.add("engineBoom");
        sequence.add("start");
        sequence.add("stop");
        //来一个奔驰模型:
        BenzBuilder benzBuilder=new BenzBuilder();
        //设置顺序
        benzBuilder.setSequence(sequence);
        //生产一个奔驰模型
        BenzModel benz= (BenzModel) benzBuilder.getCarModel();
        benz.run();
        //按照同样的顺序，再来一个宝马
        BMWBuilder bmwBuilder=new BMWBuilder();
        benzBuilder.setSequence(sequence);
        BMWModel bmw= (BMWModel) bmwBuilder.getCarModel();
        bmw.run();
    }
}

```

运行结果：

```

Run: Client (2) x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
奔驰车引擎轰鸣-----
奔驰车启动-----
奔驰车停止-----
宝马车启动-----
宝马车引擎轰鸣-----
宝马车停止-----

```

同样运行顺序的宝马车也出来了，而且代码比第一版直接访问产品类简单清晰。

我们在做项目的时候要知道：需求不可能一成不变的。我们案例中的4个过程（start stop alarm engineboom）按照组合有很多种。客户可以随意组合，它是上帝，想要什么顺序我就要生成什么顺序的车模。怎么办呢？我们就需要封装一下，找个导演来指挥各个事件的先后顺序，然后为每种顺序指定一个代码，你要什么我们立刻就可以提供。

//导演类

```
public class Director {
    private ArrayList<String> sequence=new ArrayList<>();
    private BenzBuilder benzBuilder=new BenzBuilder();
    private BMWBuilder bmwBuilder=new BMWBuilder();

    /**
     * 奔驰模型A, 先start,再stop, 其他啥都没有
     * @return
     */
    public BenzModel getABenzModel(){
        this.sequence.clear();
        this.sequence.add("start");
        this.sequence.add("stop");
        this.benzBuilder.setSequence(sequence);
        return (BenzModel) this.benzBuilder.getCarModel();
    }

    /**
     * 奔驰模型B, 先发动引擎,然后start,再stop, 其他啥都没有
     * @return
     */
    public BenzModel getBBenzModel(){
        this.sequence.clear();
        this.sequence.add("engineboom");
        this.sequence.add("start");
        this.sequence.add("stop");
        this.benzBuilder.setSequence(sequence);
        return (BenzModel) this.benzBuilder.getCarModel();
    }

    /**
     * 宝马模型C, 先按下喇叭,然后start,再stop
     * @return
     */
    public BMWModel getCBMWzModel(){
        this.sequence.clear();
        this.sequence.add("alarm");
        this.sequence.add("start");
        this.sequence.add("stop");
        this.bmwBuilder.setSequence(sequence);
        return (BMWModel) this.bmwBuilder.getCarModel();
    }

    /**
     * 宝马模型D, 先start,再引擎响, 然后鸣笛, 再stop
     * @return
     */
    public BMWModel getDBMWzModel(){
        this.sequence.clear();
        this.sequence.add("start");
        this.sequence.add("engineboom");
        this.sequence.add("alarm");
        this.sequence.add("stop");
        this.bmwBuilder.setSequence(sequence);
    }
}
```

```

        return (BMWModel) this.bmwBuilder.getCarModel();
    }
    //你还需要其他顺序的再自定义方法，导演类就是按照什么顺序导演说了算
}
//修改后的场景类
public class Client {
    public static void main(String[] args) {
        Director director=new Director();
        //各个类型的模型车
        System.out.println("奔驰A~~~~~");
        director.getABenzModel().run();
        System.out.println("奔驰B~~~~~");
        director.getBBenzModel().run();
        System.out.println("宝马C~~~~~");
        director.getCBMWzModel().run();
        System.out.println("宝马D~~~~~");
        director.getDBMWzModel().run();
    }
}

```

运行结果：



有了这样的导演类之后，我们的场景类就更容易处理了。而且代码变得简单清晰。

其实我们上面用的就是建造者模式！

其实生活中还有更多的这样的案例。例如，计算机是由 OPU、主板、内存、硬盘、显卡、机箱、显示器、键盘、鼠标等部件组装而成的，采购员不可能自己去组装计算机，而是将计算机的配置要求告诉计算机销售公司，计算机销售公司安排技术人员去组装计算机，然后再交给要买计算机的采购员。

再例如游戏中的不同角色，其性别、个性、能力、脸型、体型、服装、发型等特性都有所差异；还有汽车中的方向盘、发动机、车架、轮胎等部件也多种多样；每封电子邮件的发件人、收件人、主题、内容、附件等内容也各不相同。

以上所有这些产品都是由多个部件构成的，各个部件可以灵活选择，但其创建步骤都大同小异。这类产品的创建无法用前面介绍的工厂模式描述，只有建造者模式可以很好地描述该类产品的创建。

2.4.2 模式的定义与结构

2.4.2.1 建造者模式的定义：

指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

2.4.2.2 模式的结构与实现

1. 模式的结构

建造者模式的主要角色如下。

1. 产品（Product）类：它是包含多个组成部件的复杂对象，由具体建造者来创建其各个部件。

2. 抽象建造者 (Builder) : 它是一个包含创建产品各个子部件的抽象方法的接口, 通常还包含一个返回复杂产品的方法 getResult()。
3. 具体建造者(Concrete Builder) : 实现 Builder 接口, 完成复杂产品的各个部件的具体创建方法。
4. 导演 (Director) 类: 它调用建造者对象中的部件构造与装配方法完成复杂对象的创建, 在指挥者中不涉及具体产品的信息。

2. 模式的实现

(1) 产品类: 包含多个组成部件的复杂对象。

```
public class Product
{
    private String partA;
    private String partB;
    private String partC;
    public void setPartA(String partA){
        this.partA=partA;
    }
    public void setPartB(String partB){
        this.partB=partB;
    }
    public void setPartC(String partC){
        this.partC=partC;
    }
    public void doSomething(){
        //独立业务处理
    }
}
```

(2) 抽象建造者: 包含创建产品各个子部件的抽象方法。

```
public abstract class Builder{
    //创建产品的不同部分, 以获取不同产品
    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();
    //返回产品对象
    public abstract Product buildProduct();
}
```

(3) 具体建造者: 实现了抽象建造者接口。

```
public class ConcreteBuilder extends Builder{
    private Product product=new Product();
    public void buildPartA(){
        product.setPartA("建造 PartA");
    }
    public void buildPartB(){
        product.setPartA("建造 PartB");
    }
    public void buildPartC(){
```

```

        product.setPartA("建造 PartC");
    }
    //组件一个产品
    public Product buildProduct(){
        return product;
    }
}

```

(4) 指挥者：调用建造者中的方法完成复杂对象的创建。

```

public class Director
{
    private Builder builder;
    public Director(Builder builder){
        this.builder=builder;
    }
    //产品构建与组装方法:设置不同的零件，生成不同的产品
    public Product constructA(){
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.buildProduct();
    }
    public Product constructB(){
        builder.buildPartB();
        builder.buildPartA();
        builder.buildPartC();
        return builder.buildProduct();
    }
}

```

(5) 场景类

```

public class Client{
    public static void main(String[] args){
        Builder builder=new ConcreteBuilder();
        Director director=new Director(builder);
        Product product=director.construct();
        product.doSomething();
    }
}

```

2.4.3 建造者模式的优缺点

优点：

1. 各个具体的建造者相互独立，有利于系统的扩展。
2. 客户端不必知道产品内部组成的细节，便于控制细节风险。

缺点：

1. 产品的组成部分必须相同，这限制了其使用范围。

2. 如果产品的内部变化复杂，该模式会增加很多的建造者类。

建造者（Builder）模式和工厂模式的关注点不同：建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。

2.4.4 模式的应用场景

建造者（Builder）模式创建的是复杂对象，其产品的各个部分经常面临着剧烈的变化，但将它们组合在一起的算法却相对稳定，所以它通常在以下场合使用。

- 相同的方法，不同的执行顺序，产生不同的实践结果
- 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。
- 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

2.5 原型模式

3、设计模式之结构型模式

3.1 代理（Proxy）模式

3.1.1 引入案例：

前阵子疫情期间，为了打发时间竟然入了游戏的坑。一个多月的时间内我竟然打怪、升级、砍人、被砍，沉迷游戏，不可自拔，陷入打怪、升级、打怪、升级.....的死循环中无法逃脱。不过升级还是挺快的哈，小有成就感。这段时间真真儿的体会了什么叫苦乐参半。参与工会攻城胜利之后超开心，觉得自己还真厉害（可能只是队友厉害哈），但是苦的就是为了升级就要不停的打怪、做任务（毕竟游戏外挂管的也太紧了，怕被封号不敢用哈），升级基本靠自己，梦中还在和大BOSS进行PK。

有了这样一段经历也比较可贵。咱们作为程序员，能不能把这段打游戏的过程系统化呢？

说来就来：分析、动手。

接口：IGamePlayer,所有网游的玩家（作者也是其中一个哈，包括你吗？）

实现类：实现游戏爱好者为了玩游戏要执行的功能

```
public interface IGamePlayer {
    //登录游戏
    public void login(String username,String password);
    //打怪
    public void killBoss();
    //升级
    public void upgrade();
}
public class GamePlayer implements IGamePlayer{
    //昵称
    private String petName="";

    public GamePlayer(String petName) {
        this.petName = petName;
    }
}
```

```

@Override
public void login(String username, String password) {
    System.out.println("登录账号: "+username+", 昵称: "+this.petName+"登录成功!!! ");
}

@Override
public void killBoss() {
    System.out.println(this.petName+"打怪ing-----");
}

@Override
public void upgrade() {
    System.out.println(this.petName+"又升级啦-----撒花庆祝-----");
}
}
//场景类
public class Client {
    public static void main(String[] args) {
        //定义一个玩家
        IGamePlayer player=new GamePlayer("宋仲基");
        //开始打游戏, 记录下时间
        System.out.println("开始时间: 2020-2-2 13:00");
        player.login("songsong", "123456");
        //打怪
        player.killBoss();
        //升级
        player.upgrade();
        //结束打游戏, 记录下时间
        System.out.println("结束时间: 2020-2-2 23:00");
    }
}

```

运行结果:

```

Run: Client x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
开始时间: 2020-2-2 13:00
登录账号: songsong, 昵称: 宋仲基登录成功!!!
宋仲基打怪ing-----
宋仲基又升级啦-----撒花庆祝-----
结束时间: 2020-2-2 23:00

```

运行结果记录了我的网游生涯。打游戏结束后网游成瘾综合征体现的淋漓尽致, 不想放弃游戏账号, 又不想这样打游戏精疲力尽, 怎么办呢?

找代练啊! 让他们帮我去打怪、去升级。

定义代练类: 代练也不能作弊哦, 也是手动打怪升级呢!

```

public class GamePlayerProxy implements IGamePlayer{

    //被代理的玩家

```

```

private IGamePlayer player=null;

public GamePlayerProxy(IGamePlayer player) {
    this.player = player;
}

@Override
public void login(String username, String password) {
    this.player.login(username,password);
}


@Override
public void killBoss() {
    this.player.killBoss();
}

@Override
public void upgrade() {
    this.player.upgrade();
}
}

//修改一下场景类:
public class Client {
    public static void main(String[] args) {
        //定义一个玩家
        IGamePlayer player=new GamePlayer("宋仲基");
        //定义一个代练者
        IGamePlayer proxyPlayer=new GamePlayerProxy(player);
        //开始打游戏,记录下时间
        System.out.println("开始时间: 2020-2-2 13:00");
        proxyPlayer.login("songsong", "123456");
        //打怪
        proxyPlayer.killBoss();
        //升级
        proxyPlayer.upgrade();
        //结束打游戏,记录下时间
        System.out.println("结束时间: 2020-2-2 23:00");
    }
}

```

运行结果:



```

Run: Client x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
开始时间: 2020-2-2 13:00
登录账号: songsong,昵称: 宋仲基登录成功!!!
宋仲基打怪ing-----
宋仲基又升级啦-----撒花庆祝-----
结束时间: 2020-2-2 23:00

```

看到结果了吧: 没有任何改变! 你没干活, 但是有人帮你干了, 你的游戏已经不知不觉中升级啦, 躺赢变大佬!

这就是代理模式！

3.1.2 代理模式的定义与结构

3.1.2.1 代理模式的定义：

为其他对象提供一种代理以控制这个对象的访问。这是一个使用频率非常高的模式。

3.1.2.2 代理模式的结构

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问。

1、代理模式的主要角色如下。

1. 抽象主题（Subject）角色：抽象主题类可以是接口或抽象类，是一个普通的业务类型定义，声明真实主题和代理对象实现的业务方法，无特殊要求。
2. 真实主题（Real Subject）角色：真实主题角色类也叫作被委托角色、被代理角色，实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象，是业务逻辑的具体执行者。
3. 代理（Proxy）角色：也叫做委托类、代理类。他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，并且在真实主题角色处理完毕前后做预处理和善后的工作。提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。



图1 代理模式的结构图

2. 模式的实现

代理模式的实现代码如下：

```
//抽象主题类
public interface Subject
{
    void request();
}
//真实主题类
public class RealSubject implements Subject{
    public void request(){
        //业务逻辑处理
    }
}
//代理类：代理模式的核心就在代理类上
public class Proxy implements Subject{
    //要代理哪个实现类
    private Subject subject=null;
    //通过构造方法传入被代理对象（也可以有其他方式）
    public Proxy(Subject subject){
        this.subject=subject;
    }
    public void request(){
        preRequest();
        resubjectalSubject.request();
        postRequest();
    }
}
```

```

//预处理
public void preRequest(){
    System.out.println("访问真实主题之前的预处理。");
}
//善后工作
public void postRequest(){
    System.out.println("访问真实主题之后的善后。");
}
}
//场景类
public class Client
{
    public static void main(String[] args){
        Proxy proxy=new Proxy();
        proxy.request();
    }
}

```

一个代理类可以代理多个被委托或者被代理者，因此一个代理类具体代理哪个真实主题角色是由场景类决定的。最简单的情况就是一个主题类和一个代理类，这是最简单的代理模式。所以上面的结构中通过构造方法传入被代理对象就是最简单的方式。

3.1.3 代理模式的优缺点

代理模式的主要优点有：

- 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
- 代理对象可以扩展目标对象的功能；
- 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；

其主要缺点是：

- 在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
- 增加了系统的复杂度；

3.1.4 代理模式的应用场景

在有些情况下，一个客户不能或者不想直接访问另一个对象，这时需要找一个中介帮忙完成某项任务，这个中介就是代理对象。例如，购买火车票不一定要去火车站买，可以通过 12306 网站或者去火车票代售点买。又如买房子、找保姆、找工作等都可以找中介完成。

在软件设计中，使用代理模式的例子也很多，例如，spring框架中就使用了动态代理模式！

- 远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。
- 虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。
- 安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。
- 智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。

- 延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，[Hibernate](#) 中就存在属性的延迟加载和关联表的延时加载。

3.1.5 代理模式的扩展

在前面介绍的代理模式中，代理类中包含了对真实主题的引用，这种方式存在两个缺点。

1. 真实主题与代理主题一一对应，增加真实主题也要增加代理。
2. 设计代理以前真实主题必须事先存在，不太灵活。采用动态代理模式可以解决以上问题。

什么是动态代理？动态代理实在实现阶段不关心代理谁，而是在运行阶段才指定哪一个代理对象。

我们继续通过打游戏的案例来看动态代理如何实现。

定义一个实现了InvocationHandler接口的MyInvocationHandler类。其中InvocationHandler接口是JDK提供好的动态代理接口。

```
public class MyInvocationHandler implements InvocationHandler {
    //代理类中的真实对象
    Object obj=null;
    //通过构造函数给我们的真实对象赋值
    public MyInvocationHandler(Object obj) {
        this.obj = obj;
    }
    //调用被代理的方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result=method.invoke(this.obj,args);
        return result;
    }
}
//修改后的场景类
public class Client {

    public static void main(String[] args) {
        //定义一个玩家
        IGamePlayer player=new GamePlayer("宋仲基");
        //定义一个handler
        // //代理对象的调用处理程序，我们将要代理的真实对象传入代理对象的调用处理的构造函数中，最终代理
        //对象的调用处理程序会调用真实对象的方法
        InvocationHandler handler=new MyInvocationHandler(player);
        //开始打游戏，记录下时间
        System.out.println("开始时间: 2020-2-2 13:00");
        //获取类的Class loader
        ClassLoader loader=player.getClass().getClassLoader();
        //动态产生一个代理者
        IGamePlayer proxyPlayer= (IGamePlayer)
        Proxy.newProxyInstance(loader,player.getClass().getInterfaces(), handler);
        //登录
        proxyPlayer.login("songsong", "123456");
        //打怪
        proxyPlayer.killBoss();
        //升级
        proxyPlayer.upgrade();
    }
}
```

```

        //结束打游戏，记录下时间
        System.out.println("结束时间: 2020-2-2 23:00");
    }
}

```

运行结果:



```

Run: Client x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
开始时间: 2020-2-2 13:00
登录账号: songsong,昵称: 宋仲基登录成功!!!
宋仲基打怪ing-----
宋仲基又升级啦-----撒花庆祝-----
结束时间: 2020-2-2 23:00

```

还是代练在帮我打游戏，可是我们既没有创建代理类，也没有实现IGamePlayer接口，这就是动态代理。

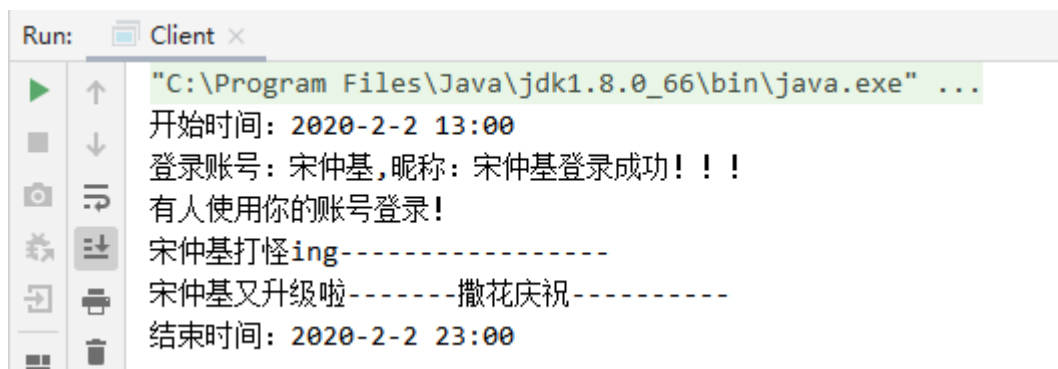
如果能让游戏登录后发个信息就更好了，怎么处理呢？

```

//修改了MyInvocationHandler类
public class MyInvocationHandler implements InvocationHandler {
    //代理类中的真实对象
    Object obj=null;
    //通过构造函数给我们的真实对象赋值
    public MyInvocationHandler(Object obj) {
        this.obj = obj;
    }
    //调用被代理的方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result=method.invoke(this.obj,args);
        if(method.getName().equalsIgnoreCase("login")){
            System.out.println("有人使用你的账号登录！");
        }
        return result;
    }
}

```

运行结果:



```

Run: Client x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
开始时间: 2020-2-2 13:00
登录账号: 宋仲基,昵称: 宋仲基登录成功!!!
有人使用你的账号登录！
宋仲基打怪ing-----
宋仲基又升级啦-----撒花庆祝-----
结束时间: 2020-2-2 23:00

```

如果有人用我的账号，就发送一个信息，看看自己的账号是不是被盗了，其实这就是AOP编程。

动态代理的实现：

```
//抽象主题类
public interface Subject{
    //业务操作
    void doSomething(String str);
}

//真实主题类
public class RealSubject implements Subject{
    //业务操作
    public void doSomething(String str){
        System.out.println("doSomething----->" + str);
    }
}

//动态代理的Handler类
public class MyInvocationHandler implements InvocationHandler {
    //被代理的对象
    private Object obj = null;
    //通过构造函数赋值
    public MyInvocationHandler(Object obj) {
        this.obj = obj;
    }
    //代理方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //执行被代理的方法
        return method.invoke(this.obj, args);
    }
}

//动态代理类
public class DynamicProxy<T>{
    public static <T> T newProxyInstance(ClassLoader loader, Class<?>[]
interfaces, InvocationHandler h){
        //寻找JoinPoint连接点，AOP框架使用元数据定义
        if(true){
            (new BeforeAdvice()).exec();
        }
        //执行目标并返回结果
        return (T) Proxy.newProxyInstance(loader, interfaces, h);
    }
}

//通知接口以及实现
public interface IAdvice{
    public void exec();
}

public class BeforeAdvice implements IAdvice{
    public void exec(){
        System.out.println("我是前置通知，执行完毕-----");
    }
}

//动态代理的场景类
public class Client {
```

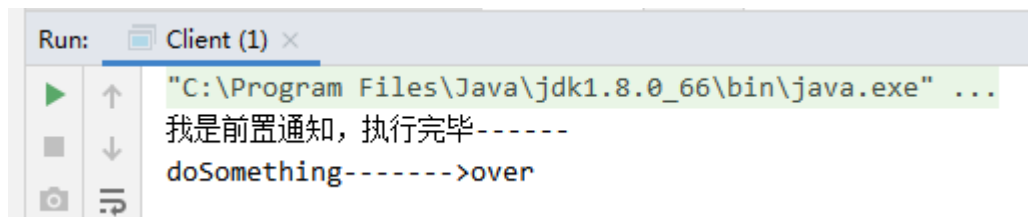


```

public static void main(String[] args) {
    //定义一个主题
    Subject subject=new RealSubject();
    //定义一个handler
    InvocationHandler handler=new com.wln.proxy.dongtai.MyInvocationHandler(subject);
    //定义主题的代理
    Subject proxy=DynamicProxy.newProxyInstance(subject.getClass().getClassLoader(),
subject.getClass().getInterfaces(), handler);
    //代理的行为
    proxy.doSomething("over");
}
}

```

运行结果:



上面的DynamicProxy类是一个通用的类, 不具有业务意义, 我们可以来一个更具体的类:

```

public class SubjectDynamainProxy extends DynamicProxy {
    public static <T> T newProxyInstance(Subject subject) {
        //获取loader
        ClassLoader loader=subject.getClass().getClassLoader();
        //获取接口数组
        Class<?>[] interfaces=subject.getClass().getInterfaces();
        //获取handler
        InvocationHandler handler=new MyInvocationHandler(subject);
        //执行目标并返回结果
        return newProxyInstance(loader, interfaces, handler);
    }
}

public class Client {
    public static void main(String[] args) {
        //定义一个主题
        Subject subject=new RealSubject();
        //定义主题的代理
        Subject proxy=SubjectDynamainProxy.newProxyInstance(subject);
        //代理的行为
        proxy.doSomething("over");
    }
}

```

写法更简洁了。该动态代理只是给出了一个通用的代理框架。大家可以根据自己的需求设计自己的AOP框架。

上面是基于JDK的动态代理, 还有基于CGLIB的动态代理。大家可以自行查阅。

3.2 适配器 (Adapter) 模式

在现实生活中，经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配。例如，讲中文的人同讲英文的人对话时需要一个翻译，用计算机访问照相机的 SD 内存卡时需要一个读卡器等。

在软件设计中也可能出现：需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决这些问题。

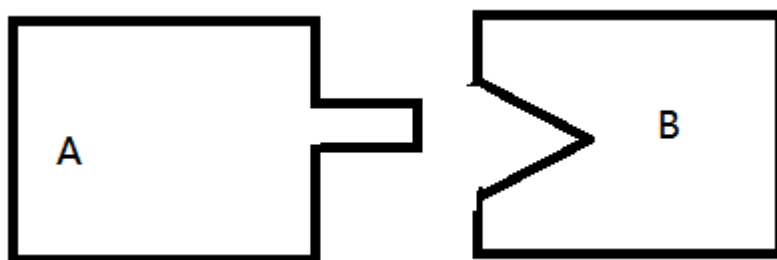
简单来说就是：让原来**不兼容**的两个接口**协同**工作。

3.2.1 适配器模式的定义与结构

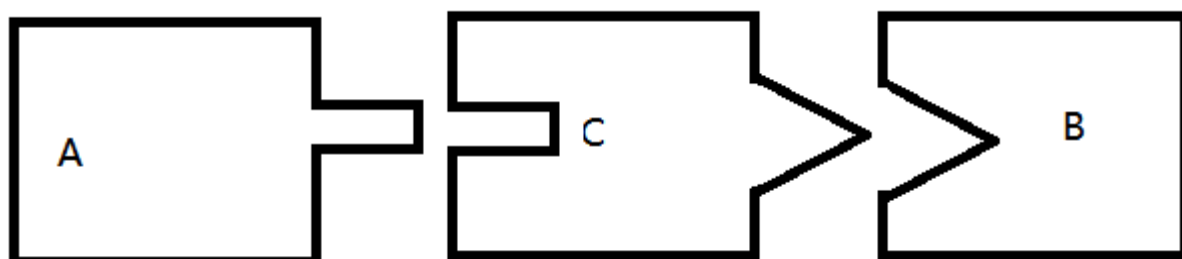
3.2.1.1 适配器模式的定义如下：

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

画个图表示一下原始的适配器：



AB两个物体接口不一致，不能安装在一起，这个时候引入物体C，要求C既能适应A也能适应B，这样三者就可以完美融合。



3.2.1.2 模式的结构

适配器模式包含以下主要角色：

1. 目标（Target）角色：该角色定义把其他类转换为何种接口，也就是我们的期望接口。它可以是抽象类或接口。
2. 源（Adaptee）角色：你想把谁转换为目标角色，这个“谁”就是源角色，他是已经存在的、运行良好的类或者对象，经过适配器角色的包装，他会成为一个新的角色。
3. 适配器（Adapter）角色：是适配器模式的核心角色，其他两个角色都是已经存在的角色，而适配器角色是需要新建立的，他的职责很简单：把原角色转换为目标角色。如何转换？通过继承或者类关联的方式。

我们可以按照分析抽取一下通用源码：

```
//目标角色
public interface Target{
    public void request();
}
```

```

}
//源角色
public class Adaptee{
    public void doSomething() {
        System.out.println("源角色的----doSomething! ");
    }
}
//适配器角色
public class Adapter extends Adaptee implements Target{
    public void request() {
        super.doSomething();
    }
}
//场景类:
public class Client{
    public static void main(String[] args) {
        Target target = new Adapter();
        target.request();
    }
}

```

程序的运行结果如下：

```
源角色的----doSomething!
```

3.2.2 适配器模式的优缺点

该模式的主要优点如下。

- 客户端通过适配器可以透明地调用目标接口。
- 复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类。
- 将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。

其缺点是：对类适配器来说，更换适配器的实现过程比较复杂。

3.2.3 适配器模式的应用实例

案例：话说前几年，买彩票中大奖啦！所以来了一次说走就走的旅行，去往梦想中的国度--希腊。这里有太多喜欢的风景啦！于是眼睛欣赏的时候，手机拍照根本停不下来。等到回到酒店的时候手机和充电宝都没电了！明天还要继续拍照呢，赶紧充电呗！拿出充电器发现一个问题，希腊的插座与中国不同，为欧式两圆孔插座，部分酒店为内嵌式插座，中国电器插头不能直接使用。没错，就是下图中这样的！



怎么办呢？询问前台之后发现他们很贴心的准备了转换器，于是借来之后顺利充电成功。



我们用代码实现这个转换过程：

```
// 给手机充电插头三项充电 (target)
public interface ThreePower {

    void powerByThree();
}
// 欧式两圆孔插座 Adapter
public class TwoPower {

    public void powerByTwo() {
        System.out.println("欧式两圆孔插座供电");
    }
}
// 二项转三项的适配器
public class TwoToThreeAdapter implements ThreePower{
    private TwoPower twoPower;
    public TwoToThreeAdapter(TwoPower twoPower) {
        this.twoPower = twoPower;
    }
    @Override
    public void powerByThree() {
        twoPower.powerByTwo();
        System.out.println("通过适配器转化三项电");
    }
}
```

```

public class Phone {

    private ThreePower threePower; //期待三项充电

    public Phone(ThreePower threePower) {
        this.threePower = threePower;
    }

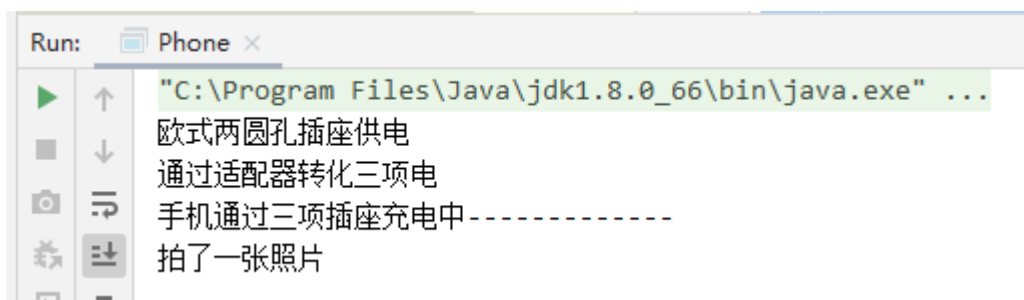
    public void recharge(){
        threePower.powerByThree();
        System.out.println("手机通过三项插座充电中-----");
    }

    public void takePicture(){
        System.out.println("拍了一张照片");
    }

    public static void main(String[] args) {
        //现在只有两项插座供电
        TwoPower twoPower = new TwoPower();
        //通过适配器转换
        ThreePower threePower = new TwoToThreeAdapter(twoPower);
        Phone phone = new Phone(threePower);
        // 充电
        phone.recharge();
        // 拍照
        phone.takePicture();
    }
}

```

运行结果：



The screenshot shows a Java IDE's Run console window. The title bar says 'Run: Phone x'. The command line shows the execution of 'C:\Program Files\Java\jdk1.8.0_66\bin\java.exe ...'. The output text is as follows:

```

欧式两圆孔插座供电
通过适配器转化三项电
手机通过三项插座充电中-----
拍了一张照片

```

3.2.4 适配器模式的使用场景

(1) 其中一个使用的场景是像上面所说的一样，有两个接口，你主动的想去连接着两个接口，写个适配器，感觉这种情况也不是很多，因为很多时候都是些一个实体类对象调用另一个实体类对象。

(2) 被动使用的情况，这种情况我可能见得比较多。举个栗子，比较极端的栗子，你和你同伴一起合作开发，你同伴写一个部分，你写一个部分，现在两个部分要对接。结过到对接时，你们发现两个人都自定义了接口，而且两个人都开发完了，都不想改，那怎么办，只能写一个适配器去适配两个接口。又或者说你开发新版本的时候重新定义了接口，要和旧版本写适配的时候，为了方便也可以使用适配器模式。

适配器模式 (Adapter) 通常适用于以下场景。

- 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
- 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

适配器模式是一个补偿模式，或者说是一个“补救”模式。通常用来解决接口不相容的问题。一般项目开始的时候用的偏少。大多数是在项目的需求不断变化的时候，技术为了业务服务的，因此业务在变化的时候，对技术也提出了要求，这些时候可能就需要这样的补救模式诞生。

3.3 装饰 (Decorator) 模式

3.3.1 案例引入

大家上学的时候有没有遇到过这样的情况：

老师：考试成绩出来了，大家把成绩单拿回家给家长看并请家长签字明天带回来。

我：.....(按惯例，一顿“竹笋炒肉”是少不了了)

我们用程序来描绘一下这个过程：

```
//抽象成绩单
public abstract class ScoreReport {
    //展示成绩
    public abstract void show();
    //家长签字
    public abstract void sign(String name);
}

//具体的成绩单
public class MyScoreReport extends ScoreReport{
    @Override
    public void show() {
        System.out.println("尊敬的家长：");
        System.out.println("\t以下是您孩子本次期末考试的成绩单，请阅读之后在后面的家长签名处签名：");
        System.out.println("\t语文 60 数学 61 自然 62 体育 90");
        System.out.println("\t\t\t家长签字:\t\t");
    }

    @Override
    public void sign(String name) {
        System.out.println("家长签名是：" + name);
    }
}

//家长查看成绩单
public class Parent {
    public static void main(String[] args) {
        //拿到成绩单
        ScoreReport report=new MyScoreReport();
        //查看成绩
        report.show();
        //签字
        System.out.println("这成绩还想签名？！");
    }
}
```

```
}
```

运行结果：

这样的成绩单肯定少不了一顿打，所以我就开始想办法，能不能把成绩单装饰一下，不让我的成绩看起来那么像挨揍的成绩呢？于是我想了一下办法（注意：修改成绩造假那是不可以滴）：

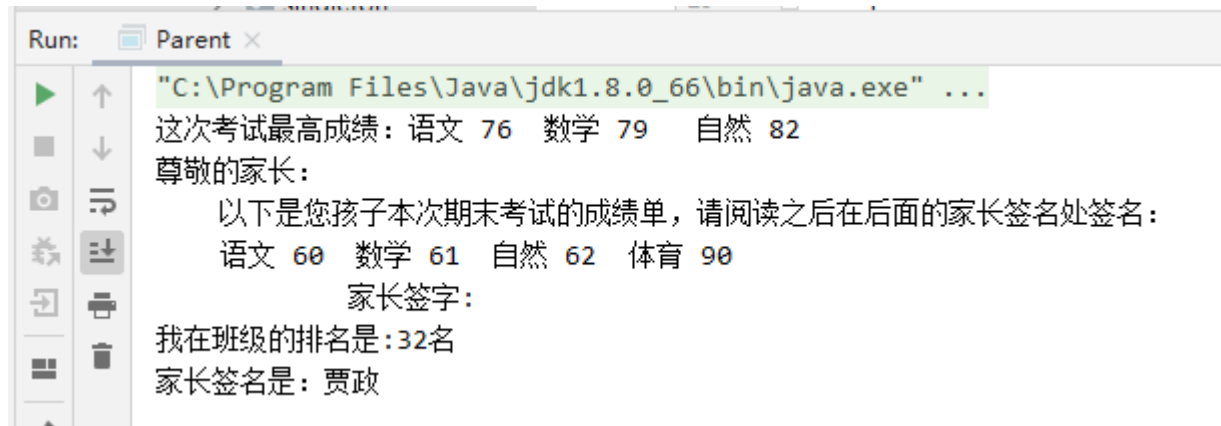
- 汇报一下最高成绩：其实是本次考试成绩都不好，最高分也都是七十多分，如果我报告了最高成绩，老爸一看我成绩跟最高成绩对比，就会觉得我这个分数还能接受；
- 汇报一下班级排名：告诉老爸我再全班排名32，当然我不会告诉他我们班考试人数也就是38个人，因为有好几个同学因为生病没参加考试。成绩单上没有这些信息，我得趁此机会免遭一顿打啊。

说做就做，于是修改一下原有的程序展示这个过程。最简单的办法就是添加一个子类，重写show方法：

```
//添加的修饰的成绩
public class BeautifyMyScoreReport extends MyScoreReport{
    //先汇报一下班级最高成绩
    public void reportHighestScore(){
        System.out.println("这次考试最高成绩：语文 76 数学 79 自然 82");
    }
    //汇报一下班级排名
    public void repoorSort(){
        System.out.println("我在班级的排名是:32名");
    }
    @Override
    public void show() {
        this.reportHighestScore();//先汇报一下班级最高成绩
        super.show();//然后展示成绩单
        this.repoorSort();//最后汇报一下班级排名
    }
}

//修改一下家长查看成绩单
public class Parent {
    public static void main(String[] args) {
        //拿到美化后的成绩单
        ScoreReport report=new BeautifyMyScoreReport();
        //查看成绩
        report.show();
        //看完觉得还凑合，没有那么差劲，签名
        report.sign("贾政");
    }
}
```

运行结果：



我们通过继承的方式解决了这个问题。老爸看后没揍我就签字了。但是现实的情况可能有很多种：

- 看完最高成绩和我的成绩，直接签名了，不看后面的排名
- 老爸要先看排名，再看我的成绩，再看最高成绩

现实中遇到不同的情况怎么办呢？要继续扩展多少个子类呢？这个还是需要装饰的条件比较少的情况，如果条件多了，不是2个，而是20个，那要有多少个子类啊？！

这就是继承解决该问题带来的问题，继承带来的类越多，后期维护的成本也会越高。那是不是应该优化一下咱们的设计呢？所有，咱们就定义一批专门负责装饰的类，然后根据实际情况来决定是否需要装饰。

程序实现：

```
//抽象成绩单
public abstract class ScoreReport {
    //展示成绩
    public abstract void show();
    //家长签字
    public abstract void sign(String name);
}

//修饰的抽象类
public class Decorator extends ScoreReport{
    //要装饰的成绩单
    private ScoreReport report;
    //构造函数传递成绩单
    public Decorator(ScoreReport report) {
        this.report = report;
    }
    //展示成绩单
    @Override
    public void show() {
        this.report.show();
    }
    //签名
    @Override
    public void sign(String name) {
        this.report.sign(name);
    }
}
```



```

}
//最高成绩修饰
public class HighestScoreDecorator extends Decorator{

    public HighestScoreDecorator(ScoreReport report) {
        super(report);
    }
    //先汇报一下班级最高成绩
    private void reportHighestScore(){
        System.out.println("这次考试最高成绩: 语文 76  数学 79  自然 82");
    }

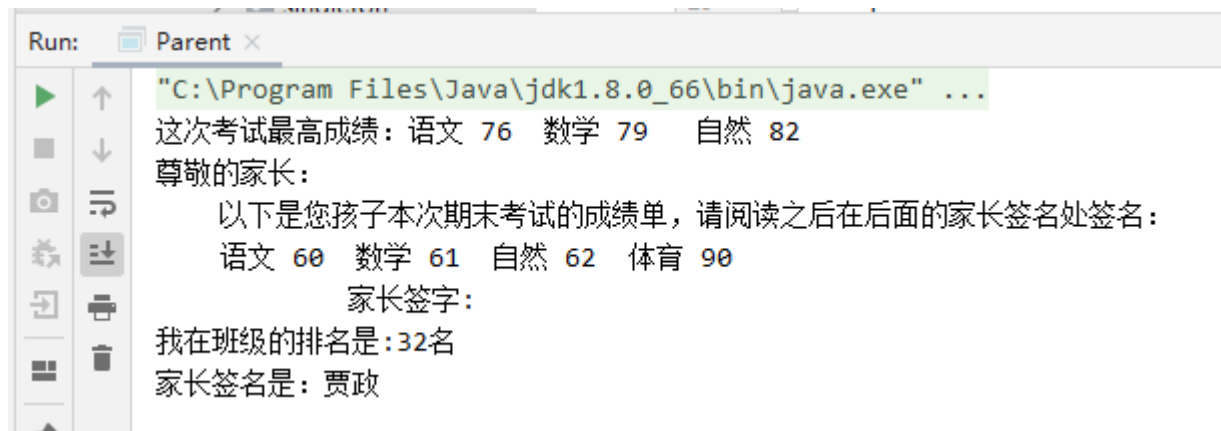
    //重写展示成绩单方法
    @Override
    public void show() {
        this.reportHighestScore();
        super.show();
    }
}
//排名修饰
public class SortDecorator extends Decorator{
    public SortDecorator(ScoreReport report) {
        super(report);
    }

    //汇报一下班级排名
    private void repoorSort(){
        System.out.println("我在班级的排名是:32名");
    }

    @Override
    public void show() {
        super.show();
        this.repoorSort();
    }
}
//家长查看成绩单
public class Parent {
    public static void main(String[] args) {
        //拿到原有成绩单
        ScoreReport report=new MyScoreReport();
        //加了最高成绩说明的成绩单
        report=new HighestScoreDecorator(report);
        //加了成绩排名的成绩单
        report=new SortDecorator(report);
        //查看成绩
        report.show();
        //看完觉得还凑合, 没有那么差劲, 签名
        report.sign("贾政");
    }
}

```

运行结果:



实现的结果一样! 而如果我还需要其他的修饰条件, 我们只需要实现Decorator类就可以啦! 这就是装饰模式!

3.3.2 装饰模式的定义与结构

3.3.2.1 装饰模式的定义:

指在不改变现有对象结构的情况下, 动态地给该对象增加一些职责 (即增加其额外功能) 的模式, 它属于对象结构型模式。

3.3.2.2 装饰模式的结构与实现

通常情况下, 扩展一个类的功能会使用继承方式来实现。但继承具有静态特征, 耦合度高, 并且随着扩展功能的增多, 子类会很膨胀。如果使用组合关系来创建一个包装对象 (即装饰对象) 来包裹真实对象, 并在保持真实对象的类结构不变的前提下, 为其提供额外的功能, 这就是装饰模式的目标。下面来分析其基本结构和实现方法。

1. 模式的结构

装饰模式主要包含以下角色。

1. 抽象构件 (Component) 角色: 是一个抽象类或者接口, 定义最核心的对象, 也就是最原始的对象, 例如上面的成绩单。
2. 具体构件 (Concrete Component) 角色: 实现抽象构件, 通过装饰角色为其添加一些职责。
3. 抽象装饰 (Decorator) 角色: 一般是一个抽象类, 继承抽象构件, 实现其抽象方法, 里面不一定有抽象的方法, 在他的属性里一般都会有一个private变量指向Component抽象构件。
4. 具体装饰 (ConcreteDecorator) 角色: 实现抽象装饰的相关方法, 并给具体构件对象添加附加的责任。

2. 模式的实现

装饰模式的实现代码如下:

```
//抽象构件
abstract class Component {
    public abstract void operation();
}
//具体构件
public class ConcreteComponent extends Component {
    @Override
    public void operation() {
        System.out.println("具体对象的操作");
    }
}
//抽象装饰者
```

```

public abstract class Decorator extends Component {
    private Component component = null;
    //通过构造函数传递给被修饰者
    public Decorator(Component component) {
        this.component = component;
    }
    //委托给被修饰者执行
    @Override
    public void operation() {
        if(component != null) {
            this.component.operation();
        }
    }
}
//具体装饰者
public class ConcreteDecoratorA extends Decorator {
    //定义被修饰者
    public ConcreteDecoratorA(Component component) {
        super(component);
    }
    //定义自己的修饰方法
    private void method1() {
        System.out.println("method1 修饰");
    }
    @Override
    public void operation() {
        this.method1();
        super.operation();
    }
}
public class ConcreteDecoratorB extends Decorator {
    //定义被修饰者
    public ConcreteDecoratorB(Component component) {
        super(component);
    }
    //定义自己的修饰方法
    private void method2() {
        System.out.println("method2 修饰");
    }
    @Override
    public void operation() {
        super.operation();
        this.method2();
    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        Component component = new ConcreteComponent();
        //第一次修饰
        component = new ConcreteDecoratorA(component);
        //第二次修饰
        component = new ConcreteDecoratorB(component);
    }
}

```

```
        //修饰后运行
        component.operation();
    }
}
```

3.3.3 装饰模式的优缺点

装饰（Decorator）模式的主要优点有：

- 采用装饰模式扩展对象的功能比采用继承方式更加灵活。
- 可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。

其主要缺点是：装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

3.3.4 装饰模式的应用场景

装饰模式通常在以下几种情况使用。

- 当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。
- 当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现。
- 当对象的功能要求可以动态地添加，也可以再动态地撤销时。
- 需要为一批的兄弟类进行改装或者加装功能的时候，可以首选装饰模式。

3.3.5 装饰模式在java中的应用

装饰模式在 java语言中的最著名的就是 Java I/O 标准库的设计了。

例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

下面代码是为 FileReader 增加缓冲区而采用的装饰类 BufferedReader 的例子：

```
BufferedReader in=new BufferedReader(new FileReader("filename.txt"));
String s=in.readLine();
```

3.4 亨元（Flyweight）模式

在面向对象程序设计过程中，有时会面临要创建大量相同或相似对象实例的问题。创建那么多的对象将会耗费很多的系统资源，它是系统性能提高的一个瓶颈。

例如：String常量池、数据库连接池、缓冲池等等都是亨元模式的应用，所以说亨元模式是池技术的重要实现方式。

比如我们每次创建字符串对象时，都需要创建一个新的字符串对象的话，内存开销会很大，所以如果第一次创建了字符串对象“adam”，下次再创建相同的字符串“adam”时，只是把它的引用指向“adam”，这样就实现了“adam”字符串再内存中的共享。

再举个例子：网络联机下棋的时候，一台服务器连接了多个玩家，如果我们每个棋子都要创建对象，那一盘棋可能就有上百个对象产生，玩家多点的话，因为内存空间有限，一台服务器就难以支持了，所以这里要使用享元模式，将棋子对象减少到几个实例。

3.4.1 享元模式的定义与结构

3.4.1.1 享元模式的定义：

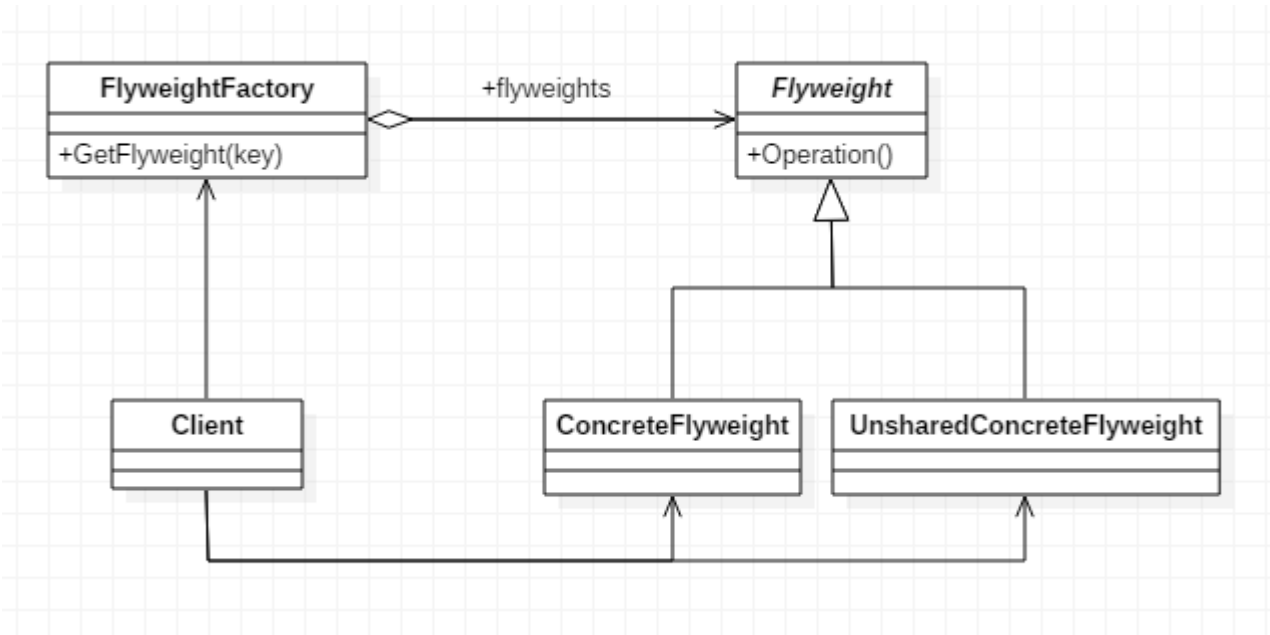
运用共享技术来有效地支持大量细粒度对象的复用。它通过共享已经存在的对象来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率。

3.4.1.2. 享元模式的结构

享元模式中存在以下两种状态：

- 内部状态，即不会随着环境的改变而改变的可共享部分；
- 外部状态，指随环境改变而改变的不可以共享的部分。享元模式的实现要领就是区分应用中的这两种状态，并将外部状态外部化。下面来分析其基本结构和实现方法。

1、享元模式的主要角色有如下。



- 抽象享元角色 (Flyweight) :简单理解就是一个产品的抽象类，同时定义出对象的外部状态和内部状态的接口或者实现。
- 具体享元 (Concrete Flyweight) 角色：实现抽象享元角色中所规定的接口。
- 非享元 (Unsharable Flyweight)角色：是不可以共享的外部状态，它以参数的形式注入具体享元的相关方法中。
- 享元工厂 (Flyweight Factory) 角色：负责创建和管理享元角色。当客户对象请求一个享元对象时，享元工厂检查系统中是否存在符合要求的享元对象，如果存在则提供给客户；如果不存在的话，则创建一个新的享元对象。

2. 享元模式的实现

享元模式的参考实现代码如下：

```
//抽象享元角色
```

```

public abstract class Flyweight {
    //内部状态
    public String intrinsic;
    //外部状态
    protected final String extrinsic;
    //要求享元角色必须接受外部状态
    public Flyweight(String extrinsic) {
        this.extrinsic = extrinsic;
    }
    //定义业务操作
    public abstract void operate(int extrinsic);
    public String getIntrinsic() {
        return intrinsic;
    }
    public void setIntrinsic(String intrinsic) {
        this.intrinsic = intrinsic;
    }
}
//具体享元角色
public class ConcreteFlyweight extends Flyweight {
    //接受外部状态
    public ConcreteFlyweight(String extrinsic) {
        super(extrinsic);
    }
    //根据外部状态进行逻辑处理
    @Override
    public void operate(int extrinsic) {
        System.out.println("具体Flyweight:" + extrinsic);
    }
}
//那些不需要共享的Flyweight子类。
public class UnsharedConcreteFlyweight extends Flyweight {

    public UnsharedConcreteFlyweight(String extrinsic) {
        super(extrinsic);
    }

    @Override
    public void operate(int extrinsic) {
        System.out.println("不共享的具体Flyweight:" + extrinsic);
    }
}
//享元工厂
public class FlyweightFactory {
    //定义一个池容器
    private static HashMap<String, Flyweight> pool = new HashMap<>();
    //享元工厂
    public static Flyweight getFlyweight(String extrinsic) {
        Flyweight flyweight = null;
        if(pool.containsKey(extrinsic)) { //池中有该对象
            flyweight = pool.get(extrinsic);

            System.out.print("已有 " + extrinsic + " 直接从池中取---->");
        }
    }
}

```

```

    } else {
        //根据外部状态创建享元对象
        flyweight = new ConcreteFlyweight(extrinsic);
        //放入池中
        pool.put(extrinsic, flyweight);
        System.out.print("创建 " + extrinsic + " 并从池中取出---->");
    }
    return flyweight;
}
}

```

3.4.2 享元模式的优缺点：

优点：

大大减少应用程序创建的对象，相同对象只要保存一份，，降低程序内存的占用，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

缺点：

1. 为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性。
2. 读取享元模式的外部状态会使得运行时间稍微变长。

3.4.3 享元模式的应用实例

实现一个生活中的场景：开部门会议

```

public interface Employee {
    void report();
}
public class Manager implements Employee{

    private String title = "部门经理";
    private String department;
    private String reportContent;

    @Override
    public void report() {
        System.out.println(reportContent);
    }

    public void setReportContent(String reportContent) {
        this.reportContent = reportContent;
    }

    public Manager(String department) {
        this.department = department;
    }
}
public class EmployeeFactory {
    private static final Map<String,Employee> EMPLOYEE_MAP = new HashMap<String,Employee>();

    public static Employee getManager(String department){

```

```

        Manager manager = (Manager) EMPLOYEE_MAP.get(department);

        if(manager == null){
            manager = new Manager(department);
            System.out.println("创建部门经理:"+department);
            String reportContent = department+"部门汇报:此次报告的主要内容是.....";
            manager.setReportContent(reportContent);
            System.out.println("\t创建报告:"+reportContent);
            EMPLOYEE_MAP.put(department,manager);
        }
        return manager;
    }
}

public class Client {
    private static final String departments[] = {"RD", "QA", "PM", "BD"};

    public static void main(String[] args) {
        for(int i=0; i<10; i++){
            String department = departments[(int)(Math.random() * departments.length)];
            Manager manager = (Manager) EmployeeFactory.getManager(department);
            manager.report();
        }
    }
}

```

运行结果：注意，因为有随机数的使用，结果每次都不同

```

Run: Client (1) x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
创建部门经理:QA
    创建报告:QA部门汇报:此次报告的主要内容是.....
QA部门汇报:此次报告的主要内容是.....
创建部门经理:RD
    创建报告:RD部门汇报:此次报告的主要内容是.....
RD部门汇报:此次报告的主要内容是.....
QA部门汇报:此次报告的主要内容是.....
RD部门汇报:此次报告的主要内容是.....
RD部门汇报:此次报告的主要内容是.....
创建部门经理:PM
    创建报告:PM部门汇报:此次报告的主要内容是.....
PM部门汇报:此次报告的主要内容是.....
RD部门汇报:此次报告的主要内容是.....
创建部门经理:BD
    创建报告:BD部门汇报:此次报告的主要内容是.....
BD部门汇报:此次报告的主要内容是.....
BD部门汇报:此次报告的主要内容是.....
BD部门汇报:此次报告的主要内容是.....

```


3.4.4 享元模式的应用场景

1. 系统中存在大量相同或相似的对象，这些对象耗费大量的内存资源。
2. 细粒度的对象都具备比较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定的身份。
3. 需要缓冲池的场景

3.5 外观模式

3.6 桥接模式

3.7 组合模式

4、设计模式之行为型模式

4.1 策略模式 (Strategy)

4.1.1 引入案例

我们上学到的时候每次考完试，老师都会讲试卷对吧？那么你的老师最喜欢说的一句话是什么呢？反正我印象中印象最深刻的一句话就是：这道题有多种解法，我们来依次讲解一下！然后黑板罗列出来第一种、第二种.....这是不是就是遇到题目的时候，可以根据当时的环境不同选择不同的算法或者叫策略来解决。

再举一个栗子：

我现在坐标：北京，放假了想去杭州看西湖看雷峰塔；我出行的方式有多种：飞机、高铁、开车。不管我使用哪一种出行方式，最终的目的都是要抵达一个地方。也就是选择不同的方式产生的结果都是一样的。

如果是你你怎么做呢？

最容易想到的是不是如下方式：

```
public class Travel {  
    public void travelType(String type){  
        if("飞机".equals(type)){  
            System.out.println("乘坐飞机出行-----");  
        }  
        else if("火车".equals(type)){  
            System.out.println("乘坐火车出行-----");  
        }  
        else if("开车".equals(type)){  
            System.out.println("开车出行-----");  
        }else{  
            System.out.println("暂时未提供该方式！");  
        }  
    }  
}
```

以上把所有出行的方式都写一个方法中，整个方法太庞大，很臃肿，我们这里的业务逻辑都用输出语句替代的，未来真实的业务逻辑复杂，代码就会很庞大。优化一下

```
public class Travel {
```

```

public void travelType(String type){
    if("飞机".equals(type)){
        byAir();
    }
    else if("火车".equals(type)){
        byTrain();
    }
    else if("开车".equals(type)){
        byCar();
    }else{
        System.out.println("暂时未提供该方式! ");
    }
}

public void byAir(){
    System.out.println("乘坐飞机出行-----");
}
public void byTrain(){
    System.out.println("乘坐火车出行-----");
}
public void byCar(){
    System.out.println("开车出行-----");
}
}

```

上面的代码比刚开始的时候要好一点，它把每个具体的算法都单独抽出来作为一个方法，当某一个具体的算法有了变动的时候，只需要修改相应的方法就可以了。

但是改进后的代码还是有问题的，那有什么问题呢？

1.当我们新增一种出行方式的时候，首先要添加一个该种出行方式的方法，然后再travelType方法中再加一个else if的分支，是不是感觉很是麻烦呢？而且这也违反了设计原则之一的开闭原则。还记得开闭原则吗？

开闭原则：

对于扩展是开放的。这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。也就是说，我们可以改变模块的功能。

对于修改是关闭的。对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。

2.我们经常会面临这样的情况，去不同的目的地使用不同的出行方式，按照上面的代码我们就得不停的修改if else里面的代码很是麻烦。

那有没有什么办法使得我们的出行方式即可扩展、可维护，又可以方便的响应变化呢？

当然有解决方案啦，就是我们下面要讲的策略模式。

4.1.2 策略模式的定义与结构

4.1.2.1 策略模式的定义：

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

4.1.2.2 策略模式的结构

策略模式的主要角色如下。

1. 抽象策略（Strategy）类：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。
2. 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现。
3. 策略上下文角色StrategyContext:策略上下文，负责和具体的策略实现交互，通常策略上下文对象会持有一个真正的策略实现对象，策略上下文还可以让具体的策略实现从其中获取相关数据，回调策略上下文对象的方法。

我们按照结构修改一下我们刚刚的案例：

```
public interface TravelStrategy {
    public void travelType();
}

public class AirStrategy implements TravelStrategy {
    @Override
    public void travelType() {
        System.out.println("乘坐飞机出行-----");
    }
}

public class TrainStrategy implements TravelStrategy {
    @Override
    public void travelType() {
        System.out.println("乘坐火车出行-----");
    }
}

public class CarStrategy implements TravelStrategy {
    @Override
    public void travelType() {
        System.out.println("开车出行-----");
    }
}

public class Traveler {
    //出行策略
    private TravelStrategy travelStrategy;

    //设置出行策略
    public Traveler(TravelStrategy travelStrategy) {
        this.travelStrategy = travelStrategy;
    }

    public void travelStyle(){
        travelStrategy.travelType();
    }
}

public class Client {
    public static void main(String[] args) {
        TravelStrategy travelStrategy=null;
        for(int i=0;i<3;i++) {
            System.out.println("要去杭州玩啦！可以有以下选择方式：");

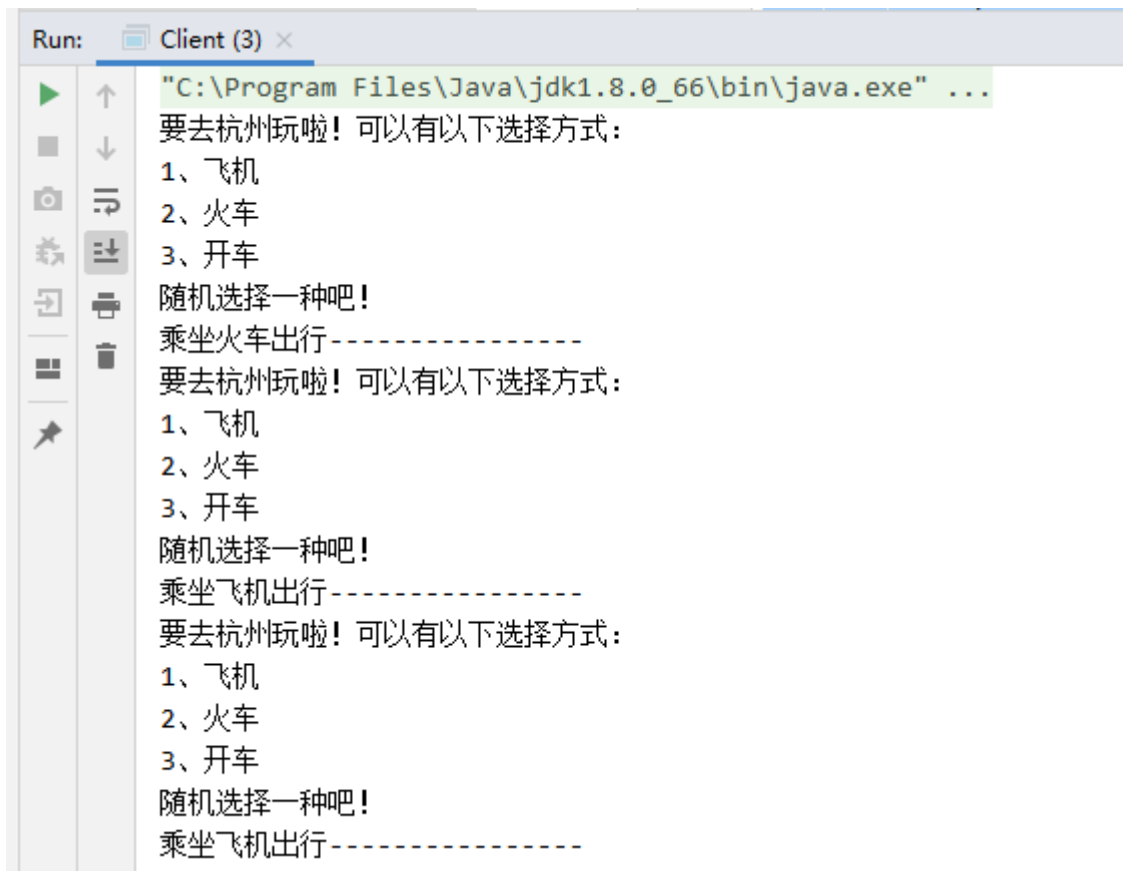
            System.out.println("1、飞机");
```

```

        System.out.println("2、火车");
        System.out.println("3、开车");
        System.out.println("随机选择一种吧! ");
        int num = new Random().nextInt(3) + 1;
        if (num == 1) {
            travelStrategy = new AirStrategy();
        } else if (num == 2) {
            travelStrategy = new TrainStrategy();
        } else {
            travelStrategy = new CarStrategy();
        }
        Traveler traveler = new Traveler(travelStrategy);
        traveler.travelStyle();
    }
}

```

运行结果:



```

Run: Client (3) x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
要去杭州玩啦! 可以有以下选择方式:
1、飞机
2、火车
3、开车
随机选择一种吧!
乘坐火车出行-----
要去杭州玩啦! 可以有以下选择方式:
1、飞机
2、火车
3、开车
随机选择一种吧!
乘坐飞机出行-----
要去杭州玩啦! 可以有以下选择方式:
1、飞机
2、火车
3、开车
随机选择一种吧!
乘坐飞机出行-----

```

4.1.3 策略模式的一般通用参考代码

```

//抽象策略类
public interface Strategy{
    void strategyMethod();    //策略方法
}
//具体策略类A
public class ConcreteStrategyA implements Strategy{
    public void strategyMethod(){

```

```

        System.out.println("具体策略A的策略方法被访问! ");
    }
}
//具体策略类B
public class ConcreteStrategyB implements Strategy{
    public void strategyMethod() {
        System.out.println("具体策略B的策略方法被访问! ");
    }
}
//环境类
class Context{
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy=strategy;
    }

    public void strategyMethod(){
        strategy.strategyMethod();
    }
}
public class Client{
    public static void main(String[] args){
        Strategy s=new ConcreteStrategyA();
        Context c=new Context(s);
        c.strategyMethod();
        System.out.println("-----");
        s=new ConcreteStrategyB();
        c.strategyMethod();
    }
}

```

4.1.4 策略模式的优缺点

优点:

1. 多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。
2. 策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。
3. 策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。
4. 策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。
5. 策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

缺点:

1. 客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类。
2. 策略模式造成很多的策略类。

4.1.5 策略模式的使用场景

生活中有很多场景都可以用策略模式来解决，例如不同客户不同折扣、大闸蟹的不同做法、一个跨国公司不同国家员工发放的工资使用的币种不一样。

简单来说就是满足以下内容的时候可以用策略模式：

- 多个类只有在算法或行为上稍有不同的场景
- 算法需要自由切换的场景
- 需要屏蔽算法规则的场景

4.2 观察者（Observer）模式

4.2.1 引入案例

现在是一个明星当道的时代，明星的一个行为能引起各种各样人群的不同反应。

我们如何用程序描述这一个现象呢？

```
//被观察者接口
public interface Observable {
    //添加一个观察者
    public void addObserver(Observer observer);
    //删除一个观察者
    public void deleteObserver(Observer observer);
    //当被观察者有所行动的是，通知观察者
    public void notifyObserver(String context);
}

public interface IStar {
    //明星也要吃饭啊
    public void eat();
    //明星的行动
    public void action();
}

public class ZhaoliYing implements IStar,Observable{

    private ArrayList<Observer> observerList=new ArrayList<>();
    @Override
    public void eat() {
        System.out.println("颖宝吃饭ing.....");
        this.notifyObserver("颖宝开始吃饭了!!!");
    }

    @Override
    public void action() {
        System.out.println("颖宝电视剧宣传中.....");
        this.notifyObserver("赵丽颖宣传电视剧!!!");
    }

    @Override
    public void addObserver(Observer observer) {
        this.observerList.add(observer);
    }

    @Override
    public void deleteObserver(Observer observer) {
        this.observerList.remove(observer);
    }
}
```

```

    }

    @Override
    public void notifyObserver(String context) {
        for (Observer observer : observerList) {
            observer.update(context);
        }
    }
}

//观察者接口
public interface Observer {
    //一旦发现别人有动静，自己就行动
    void update(String context);
}

//具体观察者：粉丝
public class Fans implements Observer{

    @Override
    public void update(String context) {
        System.out.println("粉丝：观察到明星活动");
        this.action(context);
    }

    private void action(String context){
        System.out.println(context+"----> 好可爱啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊.....");
    }
}

//具体的观察者：记者
public class Reporter implements Observer{

    @Override
    public void update(String context) {
        System.out.println("记者：观察到明星活动");
        this.action(context);
    }

    private void action(String context){
        System.out.println(context+"----> 演技高，敬业.....");
    }
}

//具体观察者：淘宝店主
public class TaobaoShop implements Observer {

    @Override
    public void update(String context) {
        System.out.println("淘宝店主：观察到明星活动");
        this.action(context);
    }

    private void action(String context){
        System.out.println(context+"----> 同款商品准备上架.....");
    }
}

public class Client {

    public static void main(String[] args) {

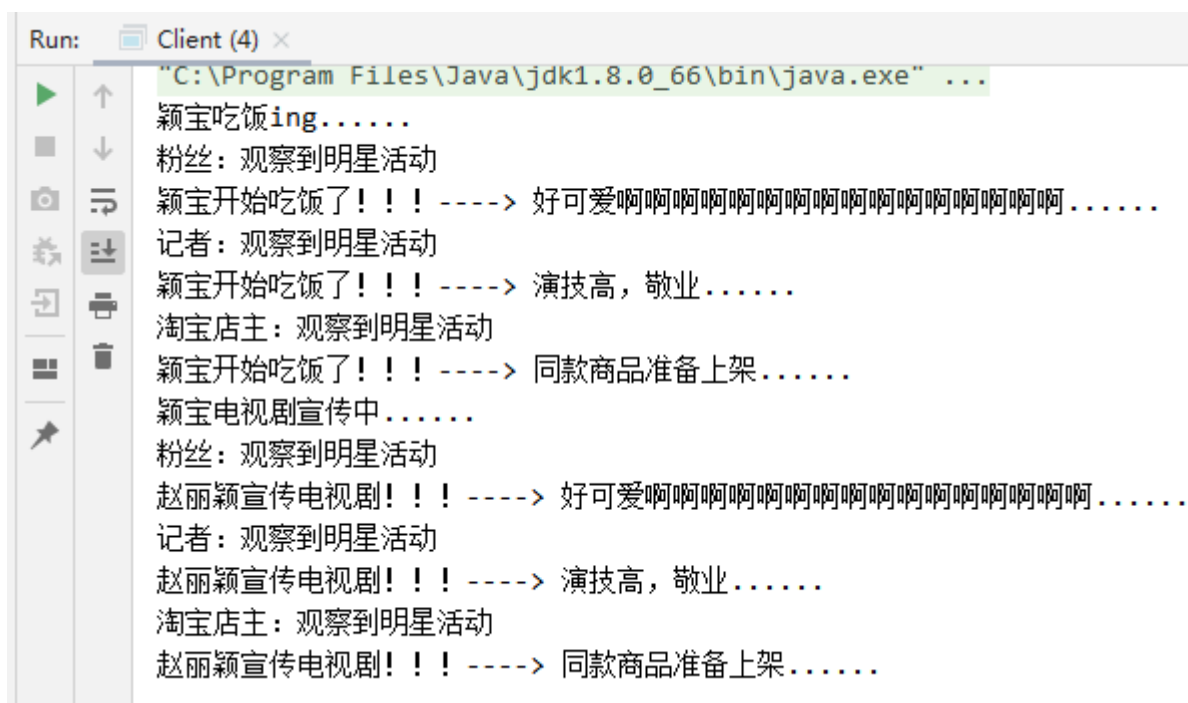
```

```

//三个具体的观察
Observer fan=new Fans();
Observer reporter=new Reportor();
Observer shop=new TaobaoShop();
//定义被观察者
ZhaoLiYing zhao=new ZhaoLiYing();
//三个观察者都在观察着赵丽颖
zhao.addObserver(fan);
zhao.addObserver(reporter);
zhao.addObserver(shop);
//赵丽颖行动
zhao.eat();
zhao.action();
}
}

```

运行结果：



```

Run: Client (4) x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
颖宝吃饭ing.....
粉丝：观察到明星活动
颖宝开始吃饭了!!! ----> 好可爱啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊.....
记者：观察到明星活动
颖宝开始吃饭了!!! ----> 演技高，敬业.....
淘宝店主：观察到明星活动
颖宝开始吃饭了!!! ----> 同款商品准备上架.....
颖宝电视剧宣传中.....
粉丝：观察到明星活动
赵丽颖宣传电视剧!!! ----> 好可爱啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊.....
记者：观察到明星活动
赵丽颖宣传电视剧!!! ----> 演技高，敬业.....
淘宝店主：观察到明星活动
赵丽颖宣传电视剧!!! ----> 同款商品准备上架.....

```

4.2.2 观察者模式的定义和结构

4.2.2.1 观察者模式的定义：

指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

4.2.2.2 观察者模式的结构：

1. 被观察者（Subject）：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。
2. 具体的被观察者（Concrete Subject）：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。
3. 观察者（Observer）：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

4. 具体的观察者（Concrete Observer）：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

4.2.2.3 观察者模式的通用的参考源码：

```
//被观察者
public abstract class Subject{
    protected List<Observer> observers=new ArrayList<Observer>();
    //增加观察者方法
    public void add(Observer observer){
        observers.add(observer);
    }
    //删除观察者方法
    public void remove(Observer observer){
        observers.remove(observer);
    }
    public abstract void notifyObserver(); //通知观察者方法
}

//具体被观察者
public class ConcreteSubject extends Subject{
    public void notifyObserver(){
        System.out.println("具体目标发生改变...");
        System.out.println("-----");
        for(Object obs:observers){
            ((Observer)obs).updateSelf();
        }
    }
}

//抽象观察者
public interface Observer{
    void updateSelf(); //反应
}

//具体观察者1
class ConcreteObserver1 implements Observer{
    public void updateSelf(){
        System.out.println("具体观察者1作出反应! ");
    }
}

//具体观察者2
class ConcreteObserver2 implements Observer{
    public void updateSelf(){
        System.out.println("具体观察者2作出反应! ");
    }
}

public class Client
{
    public static void main(String[] args){
        Subject subject=new ConcreteSubject();
        Observer obs1=new ConcreteObserver1();
        Observer obs2=new ConcreteObserver2();
        subject.add(obs1);
        subject.add(obs2);

        subject.notifyObserver();
    }
}
```

```
}  
}
```

4.2.3 观察者模式的优缺点

优点:

1. 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
2. 目标与观察者之间建立了一套触发机制。

缺点:

1. 目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。
2. 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

4.2.4 模式的应用场景

1. 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
2. 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

4.3 迭代器 (Iterator) 模式

4.3.1 引入案例

BOSS: 我现在看得报表中我们的项目费用很高，项目和人员情况也比较复杂，看得有点晕。你重新整理一下这些项目的信息给我，好查找问题在哪里。

我: 好的!!!

```
public interface IProject {  
    //Boss通过这里查看项目信息  
    String getProjectInfo();  
}  
  
public class Project implements IProject{  
    private String name;//项目名称  
    private int num=0;//项目的成员数量  
    private int cost=0;//项目费用  
  
    public Project(String name, int num, int cost) {  
        this.name = name;  
        this.num = num;  
        this.cost = cost;  
    }  
  
    @Override  
    public String getProjectInfo() {  
        String info="项目名称: "+this.name+"\t\t项目人数: "+this.num+"\t\t项目费用:"+this.cost;  
        return info;  
    }  
}  
  
public class Boss {  
  
    public static void main(String[] args) {
```

```

//存放所有项目的对象集合
List<IProject> list=new ArrayList<>();
//添加项目
list.add(new Project("疫情快递柜项目", 8, 10000));
list.add(new Project("XXX在线商城项目", 128, 1000000));
list.add(new Project("XXX综合在线办公系统", 1024, 10000000));
for(int i=4;i<105;i++){
    list.add(new Project("第"+i+"个项目", i+3, i*100000));
}
//遍历list, 取出所有的数据
for (IProject iProject : list) {
    System.out.println(iProject.getProjectInfo());
}
}
}

```

运行结果:

```

Run: Boss x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
项目名称: 疫情快递柜项目      项目人数: 8      项目费用:10000
项目名称: XXX在线商城项目    项目人数: 128      项目费用:1000000
项目名称: XXX综合在线办公系统 项目人数: 1024      项目费用:10000000
项目名称: 第4个项目          项目人数: 7      项目费用:400000
项目名称: 第5个项目          项目人数: 8      项目费用:500000
项目名称: 第6个项目          项目人数: 9      项目费用:600000
项目名称: 第7个项目          项目人数: 10     项目费用:700000
项目名称: 第8个项目          项目人数: 11     项目费用:800000
项目名称: 第9个项目          项目人数: 12     项目费用:900000
项目名称: 第10个项目         项目人数: 13     项目费用:1000000
项目名称: 第11个项目         项目人数: 14     项目费用:1100000

```

项目过多，运行结果后面的省略

出了结果给老板之后，回到工位上考虑：还有没有其它的方式呢？

老板要的就是所有项目中的部分信息，其实就是遍历啊！这个咱们是不是太熟悉了？Java中的很多集合类都提供了迭代器啊！对，就是java.util.iterator接口，作用就是遍历Collection集合下的元素，那么我也使用这种方式实验一下吧！动手！

```

public interface IProject {
    //Boss通过这里查看项目信息
    String getProjectInfo();
    //添加项目
    void add(IProject pro);
    //获取一个可用被遍历的对象
    IProjectIterator iterator();
}

public class Project implements IProject{
    //存放所有项目的对象集合
    private List<IProject> list=new ArrayList<>();

    private String name;//项目名称

```

```

private int num=0;//项目的成员数量
private int cost=0;//项目费用

public Project() {
}

public Project(String name, int num, int cost) {
    this.name = name;
    this.num = num;
    this.cost = cost;
}

@Override
public String getProjectInfo() {
    String info="项目名称: "+this.name+"\t\t项目人数: "+this.num+"\t\t项目费用:"+this.cost;
    return info;
}

@Override
public void add(IProject pro) {
    this.list.add(pro);
}

@Override
public IProjectIterator iterator() {
    return new ProjectIterator(this.list);
}
}

public interface IProjectIterator extends Iterator {
}

public class ProjectIterator implements IProjectIterator {
    //存放所有项目的对象集合
    private List<IProject> list=null;
    private int currentIndex=0;

    //通过构造方法出入所有项目的对象集合
    public ProjectIterator(List<IProject> list) {
        this.list=list;
    }

    //判断是否还有元素，必须实现的方法
    @Override
    public boolean hasNext() {
        boolean flag=true;
        if(this.currentIndex>=list.size()||this.list.get(this.currentIndex)==null)
            flag=false;
        return flag;
    }

    //获取下一个值，必须实现
    @Override
    public IProject next() {
        return this.list.get(this.currentIndex++);
    }
}

```

```

//根据实际情况自行选择是否添加该方法
public void remove(IProject pro){
    this.list.remove(pro);
}
}
}
public class Boss {
    public static void main(String[] args) {
        IProject project=new Project();
        //添加项目
        project.add(new Project("疫情快递柜项目", 8, 10000));
        project.add(new Project("XXX在线商城项目", 128, 1000000));
        project.add(new Project("XXX综合在线办公系统", 1024, 10000000));
        for(int i=4;i<105;i++){
            project.add(new Project("第"+i+"个项目", i+3, i*100000));
        }
        //遍历list, 取出所有的数据
        IProjectIterator iterator=project.iterator();
        while(iterator.hasNext()){
            IProject p= (IProject) iterator.next();
            System.out.println(p.getProjectInfo());
        }
    }
}
}

```

运行结果:

```

Run: Boss x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
项目名称: 疫情快递柜项目      项目人数: 8      项目费用:10000
项目名称: XXX在线商城项目    项目人数: 128    项目费用:1000000
项目名称: XXX综合在线办公系统 项目人数: 1024   项目费用:10000000
项目名称: 第4个项目          项目人数: 7      项目费用:400000
项目名称: 第5个项目          项目人数: 8      项目费用:500000
项目名称: 第6个项目          项目人数: 9      项目费用:600000
项目名称: 第7个项目          项目人数: 10     项目费用:700000
项目名称: 第8个项目          项目人数: 11     项目费用:800000
项目名称: 第9个项目          项目人数: 12     项目费用:900000
项目名称: 第10个项目         项目人数: 13     项目费用:1000000
项目名称: 第11个项目         项目人数: 14     项目费用:1100000

```

项目过多，运行结果后面的省略

大家可以看出，运行结果完全一样！

结果一样，可是变复杂了啊！为啥要这样嘞？其实我们使用的是一种设计模式--迭代器模式！

4.3.2 模式的定义与结构

4.3.2.1 迭代器模式的定义：

提供一种方法访问一个容器对象中各个元素，而又不需要暴露该对象的内部细节表示。

迭代器是为容器服务的，什么是容器？能盛放对象的所有类型都可以认为是容器。例如我们学过的Collection集合类型等。

(迭代器模式是一个没落的模式，模式现在一般都用在产品性质的开发中。)

4.3.2.2 迭代器模式的结构

迭代器模式主要包含以下角色。

1. 抽象容器 (Aggregate) 角色：定义存储、添加、删除容器对象以及创建迭代器对象的接口。
2. 具体容器 (ConcreteAggregate) 角色：实现抽象容器类，返回一个具体迭代器的实例。
3. 抽象迭代器 (Iterator) 角色：定义访问和遍历容器元素的接口，通常包含 hasNext()、first()、next() 等方法。
4. 具体迭代器 (ConcreteIterator) 角色：实现抽象迭代器接口中所定义的方法，完成对容器对象的遍历，记录遍历的当前位置。

4.3.2.3 迭代器模式的通用参考代码

```
//抽象容器
public interface Aggregate{
    void add(Object obj);
    boolean remove(Object obj);
    Iterator iterator();
}
//具体容器
public class ConcreteAggregate implements Aggregate{
    //也可以选择使用Vector
    private List<Object> list=new ArrayList<Object>();
    public void add(Object obj){
        list.add(obj);
    }
    public boolean remove(Object obj){
        return list.remove(obj);
    }
    public Iterator iterator(){
        return (new ConcreteIterator(list));
    }
}
//抽象迭代器
public interface Iterator{
    //遍历到下一个元素
    Object next();
    //是否已经遍历到最后
    boolean hasNext();
    //获取第一个元素
    Object first();
    //删除指定元素
    boolean remove(Object obj);
}
//具体迭代器
class ConcreteIterator implements Iterator{
    //也可以选择使用Vector
    private List<Object> list=null;
    //定义当前位置
    private int index=0;

    public ConcreteIterator(List<Object> list){
```

```

        this.list=list;
    }
    public boolean hasNext(){
        if(index==list.size()){
            return false;
        }else{
            return true;
        }
    }
}
public Object first(){
    index=0;
    Object obj=list.get(index);
    return obj;
}
public Object next(){
    Object obj=null;
    if(this.hasNext()){
        obj=list.get(++index);
    }
    return obj;
}
public boolean remove(Object obj){
    return this.list.remove(obj);
}
}
public class Client{
    public static void main(String[] args){
        Aggregate ag=new ConcreteAggregate();
        ag.add("赵丽颖");
        ag.add("宋慧乔");
        ag.add("王菲");
        System.out.print("遍历容器中的内容如下: ");
        Iterator it=ag.iterator();
        while(it.hasNext()){
            Object ob=it.next();
            System.out.print(ob.toString()+"\t");
        }
        Object ob=it.first();
        System.out.println("\nFirst: "+ob.toString());
    }
}

```

4.3.3 迭代器模式的优缺点：

优点：

1. 访问一个聚合对象的内容而无须暴露它的内部表示。
2. 遍历任务交由迭代器完成，这简化了聚合类。
3. 它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。
4. 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
5. 封装性良好，为遍历不同的聚合结构提供一个统一的接口。

缺点：

增加了类的个数，这在一定程度上增加了系统的复杂性。

4.4 模板方法 (Template Method) 模式

4.4.1 案例引入

BOSS：来任务啦！两周时间内完成10万个车模的生产，悍马的H1和H2两个型号！

我：时间太短啦.....(一堆理由)

BOSS：只做出最基本的实现，不考虑太多的问题

我：.....好吧

```
public abstract class HummerModel {
    public abstract void start();
    public abstract void stop();
    public abstract void alarm();
    public abstract void engineBoom();
    public abstract void run();
}

public class HummerH1Model extends HummerModel{
    @Override
    public void start() {
        System.out.println("悍马H1启动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎轰鸣...");
    }

    @Override
    public void run() {
        //先启动车
        this.start();
        //引擎工作
        this.engineBoom();
        //开车上路，偶尔需要按喇叭
        this.alarm();
        //到达目的地停车
        this.stop();
    }
}
```



```

public class HummerH2Model extends HummerModel{
    @Override
    public void start() {
        System.out.println("悍马H2启动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H2停车...");
    }

    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎轰鸣...");
    }

    @Override
    public void run() {
        //先启动车
        this.start();
        //引擎工作
        this.engineBoom();
        //开车上路, 偶尔需要按喇叭
        this.alarm();
        //到达目的地停车
        this.stop();
    }
}

```

到这里大家发现问题了吧？两个实现类中的run方法都是完全相同的。那么这个方法的实现应该出现在抽象类中，不应该在实现类中，抽象是所有子类的共性封装。

那咱们就动手修改一下：

```

public abstract class HummerModel {
    public abstract void start();
    public abstract void stop();
    public abstract void alarm();
    public abstract void engineBoom();
    public void run() {
        //先启动车
        this.start();
        //引擎工作
        this.engineBoom();
        //开车上路, 偶尔需要按喇叭
        this.alarm();
        //到达目的地停车
        this.stop();
    }
}

```

```

    }
}

public class HummerH1Model extends HummerModel{
    @Override
    public void start() {
        System.out.println("悍马H1启动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎轰鸣...");
    }
}

public class HummerH2Model extends HummerModel{
    @Override
    public void start() {
        System.out.println("悍马H2启动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H2停车...");
    }

    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

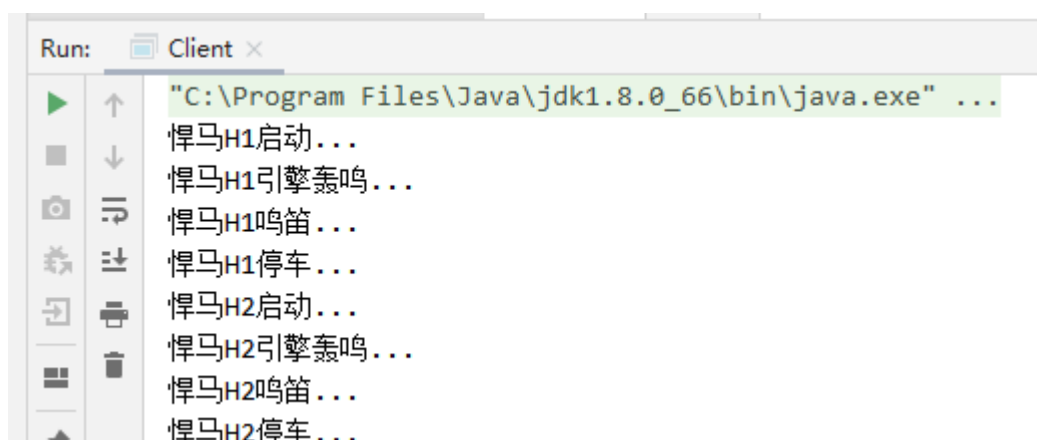
    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎轰鸣...");
    }
}

public class Client {
    public static void main(String[] args) {
        HummerModel h1=new HummerH1Model();
        h1.run();

        HummerModel h2=new HummerH2Model();
        h2.run();
    }
}

```

运行结果：



看到这里有没有觉得很熟悉？我原来也经常用啊，但是你不知道这就是模板方法模式。

4.4.2 模板方法模式的定义和结构

4.4.2.1、模板方法模式的定义如下：

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是应用非常广泛的模式，使用了java的继承机制。

4.4.2.2. 模板方法模式的结构

模板方法模式包含以下主要角色。

(1) 抽象模板类（Abstract Class）：负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下。

- 模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。
- 基本方法：是由子类实现的方法，并且在模板方法中被调用。一般包含以下三种类型的方法：
 - 抽象方法：在模板方法类中声明，由具体子类实现。
 - 具体方法：在模板方法类中已经实现，在具体子类中可以继承或重写它。
 - 钩子方法：在模板方法类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。（扩展中介绍）

(2) 具体模板类（Concrete Class）：实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。

```
//抽象模板类
public abstract class AbstractClass {
    //基本方法：抽象方法1
    protected abstract void doSomething();
    //基本方法：抽象方法2
    protected abstract void doAnything();
    //基本方法：具体方法
    protected void specificMethod(){
        //具体方法的业务逻辑
    }

    //模板方法
```

```

    public void templateMehtod() {
        //调用基本方法，完成相关逻辑
        this.specificMethod();
        this.doSomething();
        this.doAnything();
    }
}
//具体模板类
public class ConcreteClass1 extends AbstractClass{
    //实现基本方法
    protected abstract void doSomething(){
        //处理业务逻辑
    }
    protected abstract void doAnything(){
        //处理业务逻辑
    }
}
public class ConcreteClass2 extends AbstractClass{
    //实现基本方法
    protected abstract void doSomething(){
        //处理业务逻辑
    }
    protected abstract void doAnything(){
        //处理业务逻辑
    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        AbstractClass h1=new ConcreteClass1();
        AbstractClass h2=new ConcreteClass2();
        //调用模板方法
        h1.templateMehtod();
        h2.templateMehtod();
    }
}

```

4.4.3 模板方法模式的优缺点

该模式的主要优点如下。

1. 它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
2. 它在父类中提取了公共的部分代码，便于代码复用。
3. 部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

该模式的主要缺点如下。

1. 对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。
2. 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。

4.4.4 模板方法模式的扩展

到目前为止，两个模型都没啥问题。

突然BOOS又出现了：怎么设计的？车子一启动，喇叭就狂响！客户要求H1型号的喇叭自己控制，想让它响就响，不想让他响就不响；H2的就干脆不要有声音，赶快修改一下！

我：..... 自己的设计，跪着也得解决。

于是有了以下的解决方案：

```
public abstract class HummerModel {
    protected abstract void start();
    protected abstract void stop();
    protected abstract void alarm();
    protected abstract void engineBoom();
    final protected void run(){
        //先启动车
        this.start();
        //引擎工作
        this.engineBoom();
        //开车上路，要不要按喇叭由你决定
        if(this.isAlarm()){
            this.alarm();
        }
        //到达目的地停车
        this.stop();
    }
    //钩子方法：默认喇叭是会响的
    protected boolean isAlarm(){
        return true;
    }
}

public class HummerH1Model extends HummerModel{

    private boolean alarmFlag=true;//要鸣笛
    //要不要鸣笛，由客户决定
    public void setAlarm(boolean isAlarm){
        this.alarmFlag=isAlarm;
    }
    @Override
    protected boolean isAlarm(){
        return this.alarmFlag;
    }
    @Override
    protected void start() {
        System.out.println("悍马H1启动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H1停车...");
    }

    @Override
    protected void alarm() {
```

```

        System.out.println("悍马H1鸣笛...");
    }

    @Override
    protected void engineBoom() {
        System.out.println("悍马H1引擎轰鸣...");
    }

}

public class HummerH2Model extends HummerModel{
    @Override
    protected void start() {
        System.out.println("悍马H2启动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H2停车...");
    }

    @Override
    protected void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    protected void engineBoom() {
        System.out.println("悍马H2引擎轰鸣...");
    }

    //默认不鸣笛
    @Override
    protected boolean isAlarm(){
        return false;
    }

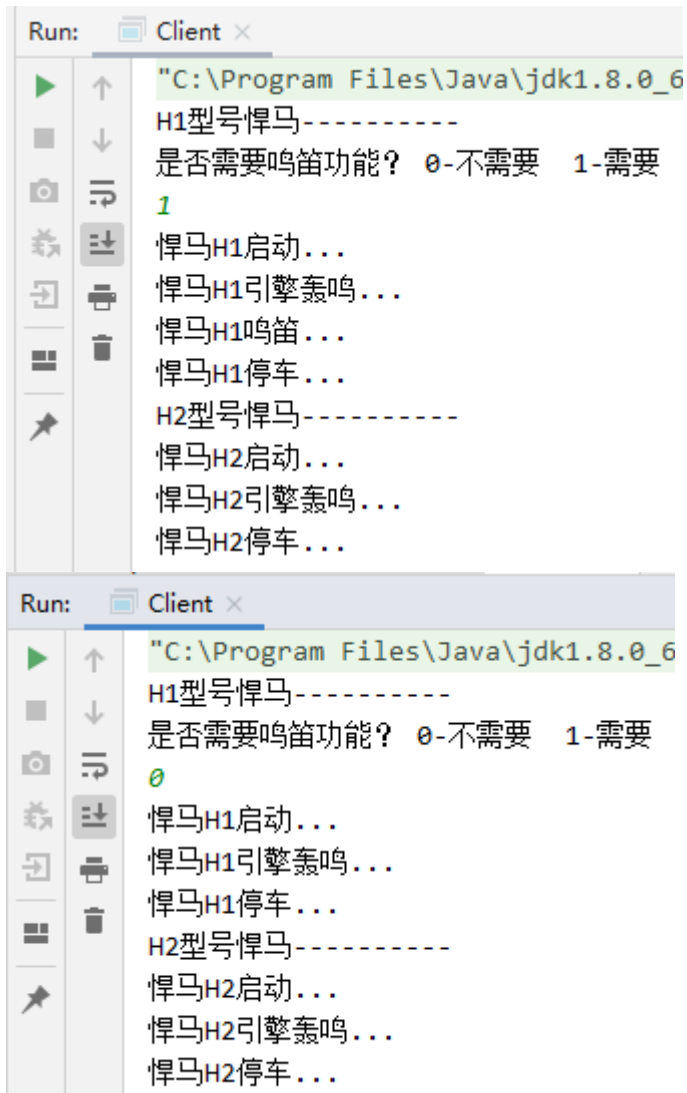
}

public class Client {
    public static void main(String[] args) {
        System.out.println("H1型号悍马-----");
        System.out.println("是否需要鸣笛功能? 0-不需要 1-需要");
        Scanner input=new Scanner(System.in);
        String type=input.next();
        HummerH1Model h1=new HummerH1Model();
        if("0".equals(type)){
            h1.setAlarm(false);
        }
        h1.run();
        System.out.println("H2型号悍马-----");
        HummerModel h2=new HummerH2Model();
        h2.run();
    }
}

```

```
}
```

运行结果：



```
Run: Client x
"C:\Program Files\Java\jdk1.8.0_6
H1型号悍马-----
是否需要鸣笛功能? 0-不需要 1-需要
1
悍马H1启动...
悍马H1引擎轰鸣...
悍马H1鸣笛...
悍马H1停车...
H2型号悍马-----
悍马H2启动...
悍马H2引擎轰鸣...
悍马H2停车...

Run: Client x
"C:\Program Files\Java\jdk1.8.0_6
H1型号悍马-----
是否需要鸣笛功能? 0-不需要 1-需要
0
悍马H1启动...
悍马H1引擎轰鸣...
悍马H1停车...
H2型号悍马-----
悍马H2启动...
悍马H2引擎轰鸣...
悍马H2停车...
```

H1型号的车是由客户自己控制是否鸣笛，也就是外界条件改变，影响到模板的执行方法。在抽象模板类中isAlarm方法的返回值影响了模板方法的执行结果，该方法就是钩子方法（Hook Method）。有了钩子方法的模板方法模式更完美。

修改一下模板方法模式的参考通用格式：（含有钩子方法）

```
//抽象模板类
public abstract class AbstractClass {
    //具体方法：钩子方法1
    protected void hookMethod1(){}
    //具体方法：钩子方法2
    protected boolean hookMethod2(){
        return true;
    }
    //基本方法：抽象方法1
    protected abstract void doSomething();
    //基本方法：抽象方法2
    protected abstract void doAnything();
}
```

```

//基本方法：具体方法
protected void specificMethod(){
    //具体方法的业务逻辑
}
//模板方法
public void templateMehtod() {
    //调用基本方法，完成相关逻辑
    this.doSomething();
    hookMethod1();
    if(hookMethod2()){
        this.specificMethod();
    }
    this.doAnything();
}
}
//具体模板类:含有钩子方法
public class ConcreteClass1 extends AbstractClass{
    //实现基本方法
    protected abstract void doSomething(){
        //处理业务逻辑
    }
    protected abstract void doAnything(){
        //处理业务逻辑
    }
    public void HookMethod1(){
        System.out.println("钩子方法1被重写...");
    }
    public boolean HookMethod2(){
        return false;
    }
}
public class ConcreteClass2 extends AbstractClass{
    //实现基本方法
    protected abstract void doSomething(){
        //处理业务逻辑
    }
    protected abstract void doAnything(){
        //处理业务逻辑
    }
}
//场景类
public class Client {
    public static void main(String[] args) {
        AbstractClass h1=new ConcreteClass1();
        AbstractClass h2=new ConcreteClass2();
        //调用模板方法
        h1.templateMehtod();
        h2.templateMehtod();
    }
}

```

4.4.5 模板方法模式的应用场景

- 1、多个子类有共有的方法，并且基本逻辑相同的时候
- 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可以由各个子类实现
- 3、重构的时候，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子方法约束其行为。