《Head First设计模式》笔记

**Eirc Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates著**

**中国电力出版社**

书评： 写书的范例，写得特别好

**设计原则**（Page 9）

|  |
| --- |
| 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。 |

**设计原则**（Page 11）

|  |
| --- |
| 针对接口编程，而不是针对实现编程。 |

“针对接口编程”真正的意思是“针对超类型（supertype）编程”

“针对超类型编程”这句话，可以更明确地说成“变量的声明类型应该是超类型，通常是一个抽象类或者是一个接口，如此，只要是具体实现此超类型的类所产生的对象，都可以指定给这个变量。这也意味着，声明类时不用理会以后执行时的真正对象类型！”

**设计原则**（Page 23）

|  |
| --- |
| 多用组合，少用继承。 |

# 一、策略模式（Strategy Pattern）

|  |
| --- |
| **策略模式**定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。 |

书中示例关键词：鸭子、不同的样子、不同的飞行方式、不同的鸣叫方式

**批注：**

指对象有某种行为，但是在不同的场景下，该行为有不同的实现算法。比如每个人都要交“个人所得税”，但是“在美国交个人所得税”和“在中国交个人所得税”就是不同的算税方法。

良好的OO设计必须具备**可复用、可扩充、可维护**三个特性。

# 二、观察者模式（Observer Pattern）

出版者 + 订阅者 = 观察者模式

|  |
| --- |
| **观察者模式**定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。 |

书中例子关键词：气象站、不同的布告板、有数据更新立即通知布告板

**设计原则**（Page53）

|  |
| --- |
| 为了交互对象之间的松耦合设计而努力。 |

**松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的互相依赖降到了最低。**

Java的java.util包内包含了最基本的Observer**接口**与Observable**类**。

java.util.Observable的黑暗面

* Observable是一个类：因为Observable是一个“类”，你必须设计一个类继承它。如果某类想同时具有Observable类和另一个超类的行为，就会陷入两难。
* Observable将关键的方法保护起来：Observable中的setChanged()方法被保护了起来（被定义成protected），这样依赖，除非你继承自Observable，否则你无法创建Observable实例并组合到你自己的对象中来。

有多个观察者时，不可以依赖特定的通知次序。

# 三、装饰者模式（Decorator Pattern）

**设计原则**（Page86）

|  |
| --- |
| 类应该对扩展开放，对修改关闭。 |

在选择需要被扩展的代码部分时要小心。每个地方都采用开放-关闭原则，是一种浪费，也没必要，还会导致代码变得负责且难以理解。

* 装饰者和被装饰对象有相同的超类型。
* 你可以用一个或多个装饰者包装一个对象。
* 既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象（被包装的）场合，可以用装饰过的对象代替它。
* **装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的。**
* 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰对象。

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

书中例子关键词：星巴克咖啡、不同的咖啡、在咖啡中添加不同数量、不同种类的调料、计算咖啡的最终价格

    /\*

     \* 饮料基类

     \*/

    public abstract class Beverage {

        String description = "Unknown Beverage";

        public String getDescription() {

            return description;

        }

        public abstract double cost();

    }

    /\*

     \*调料

     \*/

    public abstract class CondimentDecorator extends Beverage {

        public abstract String getDescription();

    }

    public class Espresso extends Beverage {

        public Espresso() {

            description = "Espresso";

        }

        public double cost(){

            return 1.99;

        }

    }

    public class HouseBlend extends Beverage {

        public HouseBlend(){

            description = "House Blend Coffee";

        }

        public double cost(){

            return .89;

        }

    }

    /\*

     \*摩卡

     \*/

    public class Mocha extends CondimentDecorator {

        Beverage beverage;

        public Mocha(Beverage beverage){

            this.beverage = beverage;

        }

        public String getDescription(){

            return beverage.getDescription() + "， Mocha";

        }

        public double cost(){

            return .20 + beverage.cost();

        }

    }

    public class StarbuzzCoffee {

        public static void main(String args[]){

            Beverage beverage = new Espresso();

            System.out.println(beverage.getDescription() + " $" + beverage.cost());

            Beverage beverage2 = new DarkRoast();

            beverage2 = new Mocha(beverage2);

            beverage2 = new Mocha(beverage2);

            beverage2 = new Whip(beverage2);

            System.out.println(beverage2.getDescription() + " $" + beverage2.cost());

            Beverage beverage3 = new HouseBlend();

            beverage3 = new Soy(beverage3);

            beverage3 = new Mocha(beverage3);

            beverage3 = new Whip(beverage3);

            System.out.println(beverage3.getDescription() + " $" + beverage3.cost());

        }

    }

利用装饰者模式的Java API

InputStream

FileInputStream、StringBufferInputStream、ByteArrayInputStream、FilterInputStream

PushbackInputStream、BufferedInputStream、DataInputStream、LineNumberInputStream

Java I/O在使用装饰者模式的同时，也引入了一个“缺点”：利用装饰者模式，常常造成设计中有大量的小类，数量实在太多，可能会造成使用此API程序员的困扰。

**装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。**

# 四、工厂模式（Factory Pattern）

静态工厂不需要使用创建对象的方法来实例化对象，但不能通过继承来改变创建方法的行为。

**在设计模式中，所谓的“实现一个接口”并“不一定”表示“写一个类，并利用implements关键词来实现某个Java接口”。“实现一个接口”泛指实现某个超类型（可以是类或接口）的某个方法。**

书中例子关键词：披萨店、相同的制作流程、不同地域的披萨店、创建不同口味的披萨（工厂）

    public abstract class PizzaStore {

        public Pizza orderPizza(String type){

            Pizza pizza;

            pizza = createPizza(type);

            pizza.prepare();

            pizza.bake();

            pizza.cut();

            pizza.box();

            return pizza;

        }

        abstract Pizza createPizza(String type);

    }

    public class NYPizzaStore extends PizzaStore{

        Pizza createPizza(String item){

            if(item.equals("cheese")){

                return new NYStyleCheesePizza();

            } else if(item.equals("veggie")){

                return new NYStyleVeggiePizza();

            } else if(item.equals("clam")){

                return new NYStyleClamPizza();

            } else if(item.equals("pepperoni")){

                return new NYStylePepperoniPizza();

            } else {

                return null;

            }

        }

    }

    public abstract class Pizza {

        String name;

        String dough;

        String sauce;

        ArrayList toppings = new ArrayList();

        void prepare(){

            System.out.println("Preparing " + name);

            System.out.println("Tossing dough...");

            System.out.println("Adding sauce...");

            System.out.println("Adding toppings: ");

            for(int i = 0; i < toppings.size(); i++){

                System.out.println("   " + toppings.get(i));

            }

        }

        void bake(){

            System.out.println("Bake for 25 minutes at 350");

        }

        void cut(){

            System.out.println("Cutting the pizza into diagonal slices");

        }

        void box(){

            System.out.println("Place pizza in official PizzaStore box");

        }

        public String getName(){

            return name;

        }

    }

    public class NYStyleCheesePizza extends Pizza {

        public NYStyleCheesePizza(){

            name = "NY Style Sauce and Cheese Pizza";

            dough = "Thin Crust Dough";

            sauce = "Marinara Sauce";

            toppings.add("Grated Reggiano Cheese");

        }

    }

    public class ChicagoStyleCheesePizza extends Pizza {

        public ChicagoStyleCheesePizza(){

            name = "Chicago Style Deep Dish Cheese Pizza";

            dough = "Extra Thick Crust Dough";

            sauce = "Plum Tomato Sauce";

            toppings.add("Shredded Mozzarella Cheese");

        }

        void cut(){

            System.out.println("Cutting the pizza into square slices");

        }

    }

    public class PizzaTestDrive{

        public static void main(String[] args){

            PizzaStore nyStore = new NYPizzaStore();

            PizzaStore chicagoStore = new ChicagoPizzaStore();

            Pizza pizza = nyStore.orderPizza("cheese");

            System.out.println("Ethan orderd a " + pizza.getName() + "\n");

            pizza = chicagoStore.orderPizza("cheese");

            System.out.println("Joel orderd a " + pizza.getName() + "\n");

        }

    }

|  |
| --- |
| **工厂方法模式**定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。 |

工厂方法不一定是抽象的，可以定义一个默认的工厂方法来产生某些具体的产品，这么一来，即使创建者没有任何子类，依然可以创建产品。

工厂方法带参数时，被称为“参数化工厂方法”。

如何限制传入工厂方法的参数，可通过创建代表参数类型的对象和使用静态常量或者Java5所支持的enum。

依赖倒置原则（Dependency Inversion Principle）

**设计原则**（Page 139）

|  |
| --- |
| 要依赖抽象，不用依赖具体类。 |

**不能让高层组件依赖低层组件，而且，不管高层或者低层组件，“两者”都应该依赖于抽象。**

PizzaStore

Pizza

NYStyleCheesePizza、NYStylePepperoniPizza、……

依赖倒置：低层组件依赖高层的抽象。高层组件也依赖相同的抽象。

如何避免在OO设计中违反依赖倒置原则：

* **变量不可以持有具体类的引用。**如果使用new，就会持有具体类的引用。你可以改用工厂来避开这样的做法。
* **不要让类派生自具体类。**如果派生自具体类，你就会依赖具体类。请派生自一个抽象（接口或抽象类）
* **不要覆盖基类中已实现的方法。**如果覆盖基类已实现的方法，那么你的基类就不是一个真正适合被基础的抽象。基类中已实现的方法，应该由所有的子类共享。

应尽量遵守上述原则。

**确保原料的一致**

    public interface PizzaIngredientFactory{

        public Dough createDough();

        public Sauce createSauce();

        public Cheese createCheese();

        public Veggies[] createVeggies();

        public Pepperoni createPepperoni();

        public Clams createClam();

    }

    public class NYPizzaIngredientFactory implements PizzaIngredientFactory{

        public Dough createDough(){

            return new ThinCrustDough();

        }

        public Sauce createSauce(){

            return new MarinaraSauce();

        }

        public Cheese createCheese(){

            return new ReggianoCheese();

        }

        public Veggeis[] createVeggeis(){

            Veggies veggeis[] = {new Garlic(), new Onion(), new Mushroom(), new RedPepper()};

            return veggies;

        }

        public Pepperoni createPepperoni(){

            return new SlicedPeppernoi();

        }

        public Clams createClam(){

            return new FreshClams();

        }

    }

    public abstract class Pizza{

        String name;

        Dough dough;

        Sauce sauce;

        Veggies veggeis[];

        Cheese cheese;

        Pepperoni pepperoni;

        Clams clam;

        abstract void prepare();

        void bake(){

            System.out.println("Bake for 25 minutes at 350");

        }

        void cut(){

            System.out.println("Cutting the pizza into diagonal slices");

        }

        void setName(String name){

            this.name = name;

        }

        String getName(){

            return name;

        }

        public String toString(){

            //这里是打印披萨的代码

        }

    }

    public class CheesePizza extend Pizza{

        PizzaIngredientFactory ingredientFactory;

        public CheesePizza(PizzaIngreientFactory ingredientFactory){

            this.ingredientFactory = ingredientFactory;

        }

        void prepare(){

            System.out.println("Preparing " + name);

            dough = ingredientFactory.createDough();

            sauce = ingredientFactory.createSauce();

            cheese = ingredientFactory.createCheese();

        }

    }

    public class ClamPizza extends Pizza{

        PizzaIngredientFactory ingredientFactory;

        public ClamPizza(PizzaIngredientFactory ingredientFactory){

            this.ingredientFactory = ingredientFactory;

        }

        void prepare(){

            System.out.println("Preparing " + name);

            dough = ingredientFactory.createDough();

            sauce = ingredientFactory.createSauce();

            cheese = ingredientFactory.createCheese();

            clam = ingredientFactory.createClam();

        }

    }

    public class NYPizzaStore extends PizzaStore{

        protected Pizza createPizza(String item){

            Pizza pizza = null;

            PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();

            if(item.equals("cheese")){

                pizza = new CheesePizza(ingredientFactory);

                pizza.setNmae("Ney York Style Cheese Pizza");

            } else if(item.equals("veggie")){

                pizza = new VeggiePizza(ingredientFactory);

                pizza.setName("New York Style Veggie Pizza");

            } else if(item.equals("clam")){

                pizza = new ClamPizza(ingredientFactory);

                pizza.setName("New York Style Clam Pizza");

            } else if(item.equals("pepperoni")){

                pizza = new PepperoniPizza(ingredientFactory);

                pizza.setName("New York Style Pepperoni Pizza");

            }

        }

    }

**抽象工厂模式**

|  |
| --- |
| **抽象工厂模式**提供一个接口，用于创建相关或依赖对象的**家族**，而不需要明确指定具体类。 |

抽象工厂的任务是定义一个负责创建一组产品的接口。这个接口内的每个方法都负责创建一个具体产品，同时我们利用实现抽象工厂的子类来提供这些具体的做法。

**工厂模式与抽象工厂模式的区别**

待梳理...

# 五、单例模式（Singleton Pattern）

应用场景：线程池（threadpool）、缓存（cache）、对话框、处理偏好设置和注册表（registry）的对象、日志对象，充当打印机、显卡等设备的驱动程序的对象。

关键要素：

构造器私有

静态方法获取实例

    public Class Singleton {

        //利用一个静态变量来记录Singleton类的唯一实例

        private static Singleton uniqueInstance;

        private Singleton(){}

        public static Singleton getInstance(){

            if(uniqueInstance == null){

                uniqueInstance = new Singleton();

            }

            return uniqueInstance;

        }

        //这里是其他的有用方法

    }

|  |
| --- |
| **单件模式**确保一个类只有一个实例，并提供一个全局访问点。 |

上述代码在多线程情况下，可能会实例化两个Singleton，这就导致了利用单例模式控制对象的实例化失败了。

    //同步确实可以解决问题，但是同步会降低性能，而且我们只需要在第一次执行此方法时，才真正需要同步。一旦设置好uniqueInstance变量，就不需要同步这个方法了。

    public Class Singleton {

        //利用一个静态变量来记录Singleton类的唯一实例

        private static Singleton uniqueInstance;

        private Singleton(){}

        public static synchronized Singleton getInstance(){

            if(uniqueInstance == null){

                uniqueInstance = new Singleton();

            }

            return uniqueInstance;

        }

        //这里是其他的有用方法

    }

如何改善多线程

1、如果getInstance()的性能对应用程序不是很关键，就什么都别做

同步一个方法可能造成程序执行效率下降100倍。因此，如果将getInstance()的程序使用在频繁运行的地方，就得重新考虑。

2、使用“急切”创建实例，而不是延迟实例化的做法

如果应用程序总是创建并使用单件实例，或者在创建和运行时方面的负担不太繁重，你可能想要急切（eagerly）创建此单件。

    //在静态初始化器（static initializen）中创建单件。这段代码保证了线程安全（thread safe）

    //JVM在加载这个类时马上创建此唯一的单件实例。JVM保证在任何线程访问uniqueInstance静态变量之前，一定先创建此实例。

    public class Singleton {

        private static Singleton uniqueInstance = new Singleton();

        private Singleton(){}

        public static Singleton getInstance(){

            return uniqueInstance;

        }

    }

3、用“双重检查加锁”，在getInstance()减少使用同步

利用双重检查加锁（double-checked locking），首先检查是否实例已经创建，如果尚未创建，“才”进行同步。这样一来，只有第一次会同步。

    public class Singleton {

        //volatile关键词确保：当uniqueInstance变量被初始化成Singleton实例时，多个线程正确地处理uniqueInstance变量

        private volatile static Singleton uniqueInstance;

        private Singleton(){}

        public static Singleton getInstance(){

            //检查实例，如果不存在，就进入同步区块

            if(uniqueInstance == null){

                synchronized(Singleton.class){

                    //进入区块后，再检查一次。如果仍是null，才创建实例

                    if(uniqueInstance == null){

                        uniqueInstance = new Singleton();

                    }

                }

            }

            return uniqueInstance;

        }

    }

> 双重检查加锁不适用于1.4及更早版本的Java！

> 在1.4及更早版本的Java中，许多JVM对于volatile关键词的实现会导致双重检查加锁的失败。

单件Q&A

|  |
| --- |
| 问：难道我不能创建一个类，把所有的方法和变量都定义为静态的，把类直接当做一个单件？  答：如果你的类自给自足，而且不依赖于复杂的初始化，那么你可以那么做。但是，以为静态初始化的控制权是在Java手上的，这么做有可能导致混乱，特别是当有许多类牵涉其中的时候。这么做常常会造成一些微妙的、不容易发现的和初始化的次序有关的bug。除非你有绝对的必要使用类的单件，否则还是建议使用对象的单件，比较保险。 |

|  |
| --- |
| 问：那么类加载器（classloader）呢？听说两个类加载器可能由机会各自创建自己的单件实例。  答：是的。每个类加载器都定义了一个命名空间，如果有两个以上的类加载器，不同的类加载器可能会加载同一个类，从整个程序来看，同一个类会被加载多次。如果这样的事情发生在单件上，就会产生多个单件并存的怪异现象。所以，如果你的程序有多个类加载器又同时使用单件模式，请小心，有一个解决方法：自行制定类加载器，并指定同一个类加载器。 |

**-------------------------------------------------------------要点-------------------------------------------------------------**

* 单件模式确保程序中一个类最多只有一个实例。
* 单件模式也提供访问这个实例的全局点。
* 在Java中实现单件模式需要私有的构造器、一个静态方法和一个静态变量。
* 确定在性能和资源上的限制，然后小心地选择适当的方案来实现单件，以解决多线程的问题（我们必须认定所有的程序都是多线程的）。
* 如果不是采用第五版的Java2，双重检查加锁实现会失效。
* 小心，你如果使用多个类加载器，可能导致单件失效而产生多个实例。
* 如果使用JVM1.2或之前的版本，你必须建立单件注册表，以免垃圾收集器将单件回收。

# 六、命令模式：封装调用

书签： 已看完Page190

2017-04-19 Page169-Page173

2017-06-27 Page174-Page177

2017-06-28 Page178-Page182

2017-07-05 Page183-Page190