**java23种设计模式全解析**

**一、设计模式的分类**

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：



**二、设计模式的六大原则**

**总原则：开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类等，后面的具体设计中我们会提到这点。

**1、单一职责原则**

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，如若不然，就应该把类拆分。

**2、里氏替换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里 氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现 抽象化的具体步骤的规范。

历史替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、 方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量首先使用合成/聚合的方式，而不是使用继承。

**三、Java的23中设计模式**

**A、创建模式**

从这一块开始，我们详细介绍Java中23种设计模式的概念，应用场景等情况，并结合他们的特点及设计模式的原则进行分析。

首先，简单工厂模式不属于23中涉及模式，简单工厂一般分为：普通简单工厂、多方法简单工厂、静态方法简单工厂。

**0、简单工厂模式**

简单工厂模式模式分为三种：

***01、普通***

就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。首先看下关系图：



举例如下：（我们举一个发送邮件和短信的例子）

首先，创建二者的共同接口：

1. **public** **interface** Sender {
2. **public** **void** Send();
3. }

其次，创建实现类：

1. **public** **class** MailSender **implements** Sender {
2. @Override
3. **public** **void** Send() {
4. System.out.println("this is mailsender!");
5. }
6. }
7. **public** **class** SmsSender **implements** Sender {
9. @Override
10. **public** **void** Send() {
11. System.out.println("this is sms sender!");
12. }
13. }

最后，建工厂类：

1. **public** **class** SendFactory {
3. **public** Sender produce(String type) {
4. **if** ("mail".equals(type)) {
5. **return** **new** MailSender();
6. } **else** **if** ("sms".equals(type)) {
7. **return** **new** SmsSender();
8. } **else** {
9. System.out.println("请输入正确的类型!");
10. **return** **null**;
11. }
12. }
13. }

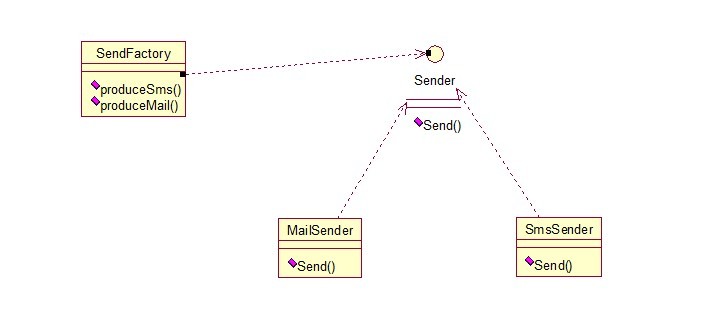
我们来测试下：

1. **public** **class** FactoryTest {
3. **public** **static** **void** main(String[] args) {
4. SendFactory factory = **new** SendFactory();
5. Sender sender = factory.produce("sms");
6. sender.Send();
7. }
8. }

输出：this is sms sender!

***02、多个方法***

是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。关系图：



将上面的代码做下修改，改动下SendFactory类就行，如下：

1. public class SendFactory {
2. public Sender produceMail(){
3. return new MailSender();
4. }
6. **public** Sender produceSms(){
7. **return** **new** SmsSender();
8. }
9. }

测试类如下：

1. **public** **class** FactoryTest {
3. **public** **static** **void** main(String[] args) {
4. SendFactory factory = **new** SendFactory();
5. Sender sender = factory.produceMail();
6. sender.Send();
7. }
8. }

输出：this is mailsender!

***03、多个静态方法***

将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

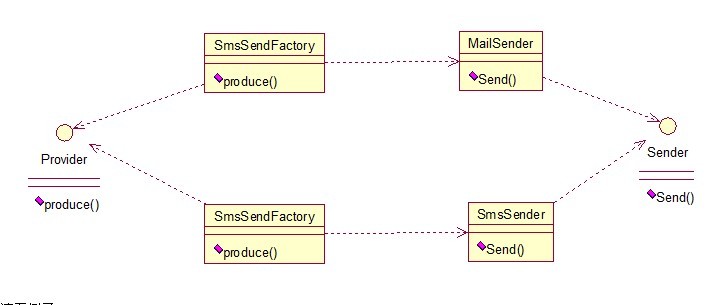
1. **public** **class** SendFactory {
3. **public** **static** Sender produceMail(){
4. **return** **new** MailSender();
5. }
7. **public** **static** Sender produceSms(){
8. **return** **new** SmsSender();
9. }
10. }
11. **public** **class** FactoryTest {
13. **public** **static** **void** main(String[] args) {
14. Sender sender = SendFactory.produceMail();
15. sender.Send();
16. }
17. }

输出：this is mailsender!

总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字 符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。

**1、工厂方法模式（Factory Method）**

简单工厂模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到工厂方法模式，创建一个工厂接口和创建多个工厂实现类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前 的代码。



请看例子：

1. **public** **interface** Sender {
2. **public** **void** Send();
3. }

两个实现类：

1. **public** **class** MailSender **implements** Sender {
2. @Override
3. **public** **void** Send() {
4. System.out.println("this is mailsender!");
5. }
6. }
7. **public** **class** SmsSender **implements** Sender {
9. @Override
10. **public** **void** Send() {
11. System.out.println("this is sms sender!");
12. }
13. }

两个工厂类：

1. **public** **class** SendMailFactory **implements** Provider {
3. @Override
4. **public** Sender produce(){
5. **return** **new** MailSender();
6. }
7. }
8. **public** **class** SendSmsFactory **implements** Provider{
10. @Override
11. **public** Sender produce() {
12. **return** **new** SmsSender();
13. }
14. }

在提供一个接口：

1. **public** **interface** Provider {
2. **public** Sender produce();
3. }

测试类：

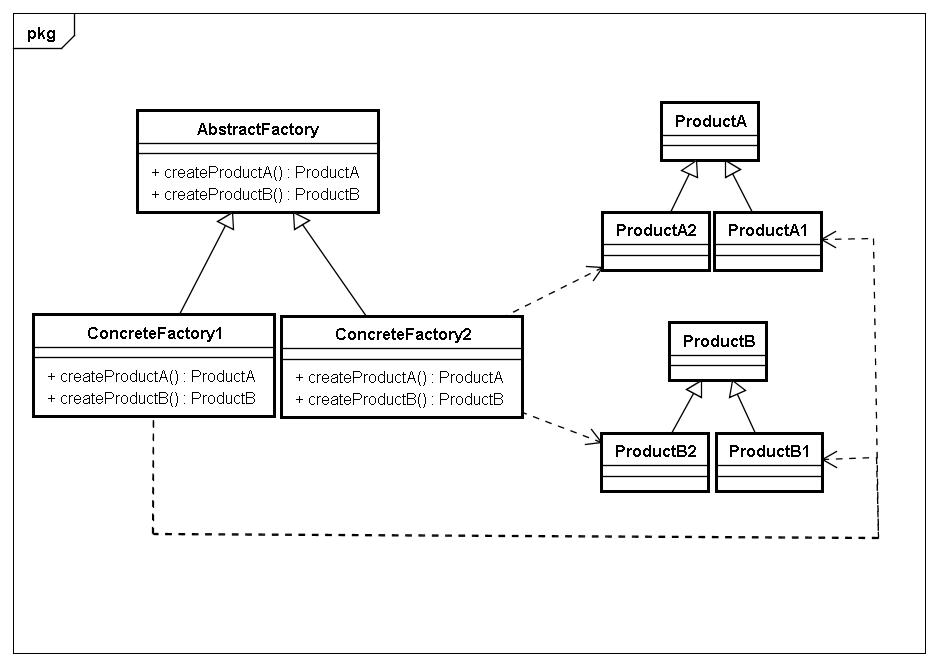
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Provider provider = **new** SendMailFactory();
5. Sender sender = provider.produce();
6. sender.Send();
7. }
8. }

其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！

**2、抽象工厂模式(abstractFactory)**

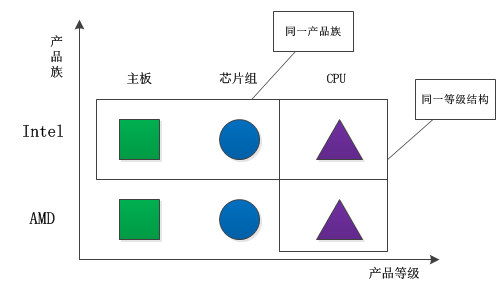
提供一个创建一系列或相互依赖对象的借口，而无须指定它们具体的类。

模式的UML类图：



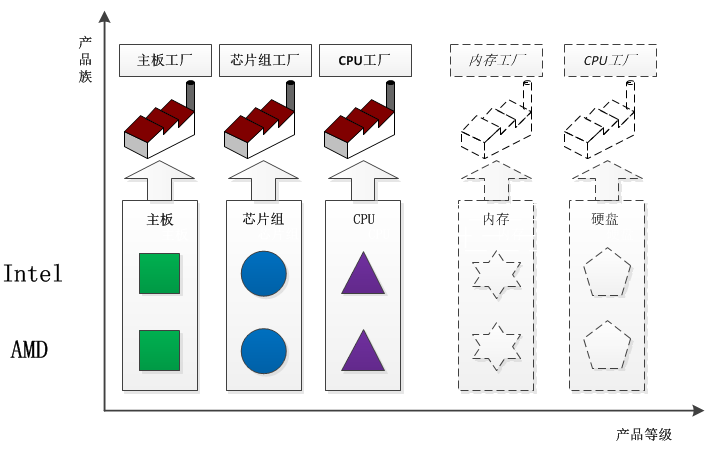
每一个模式都是针对一定问题的解决方案。抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。

在学习抽象工厂具体实例之前，应该明白两个重要的概念：产品族和产品等级。所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。比如AMD的主板、芯片组、CPU组成一个家族，Intel的主板、芯 片组、CPU组成一个家族。而这两个家族都来自于三个产品等级：主板、芯片组、CPU。一个等级结构是由相同的结构的产品组成，示意图如下：

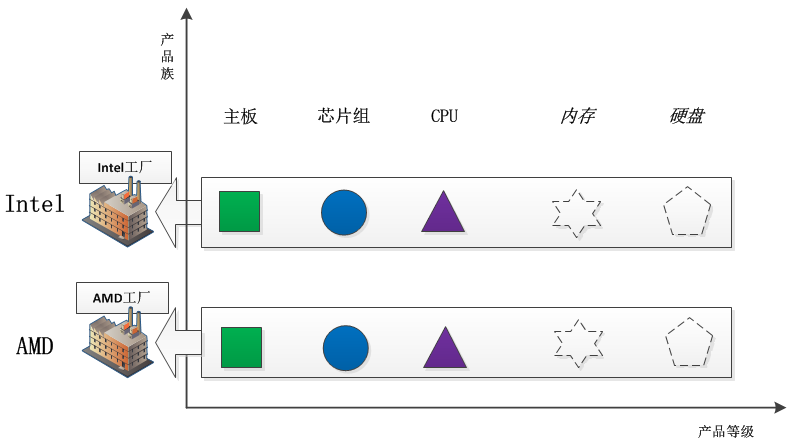


显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构与产品族将产品按照不同方向划分，形成一个二维的坐标系。横轴表示产品的等级结构，纵轴表示产品族，上图共有两个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。 如下图：



那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建。

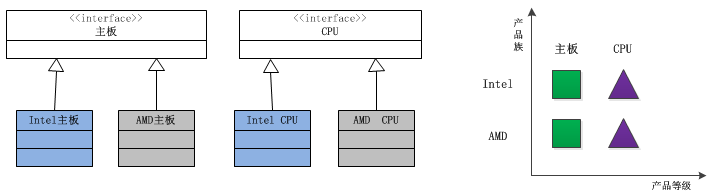


可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。显然，这时候抽象工厂模式比简单工厂模式、工厂方法模式更有效率。对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。

抽象工厂模式是对象的创建模式，它是工厂方法模式的进一步推广。

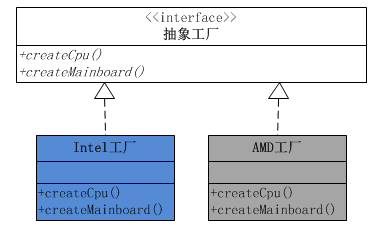
假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构中的多个产品族中的产品对象的创建问题。如下图所示：

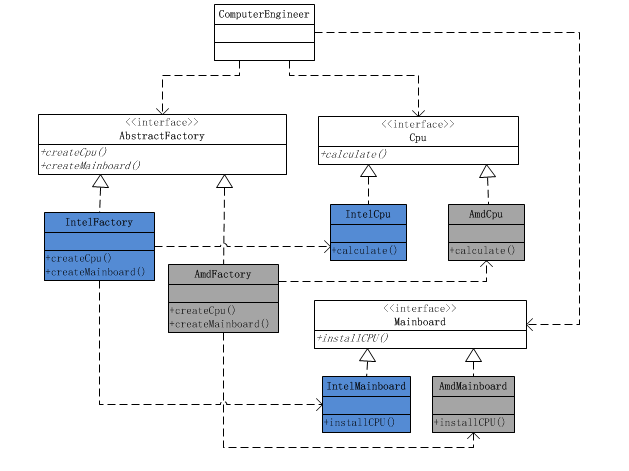


由于这两个产品族的等级结构相同，因此使用同一个工厂族也可以处理这两个产品族的创建问题，这就是抽象工厂模式。

根据产品角色的结构图，就不难给出工厂角色的结构设计图。



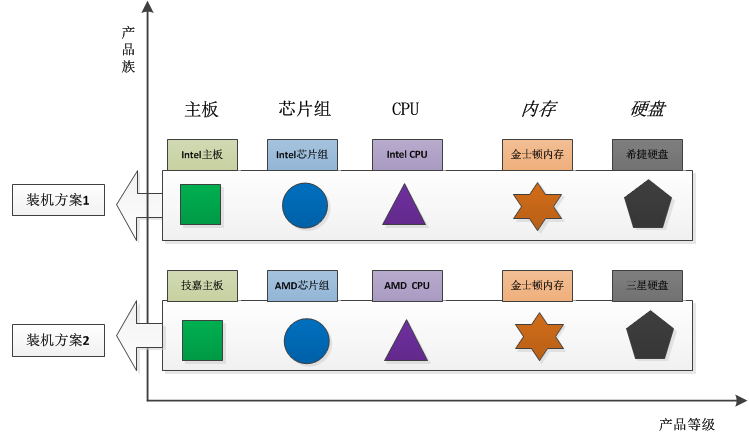
可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。



代码

1. cpu接口
2. public interface Cpu {
3. public void calculate();
4. }
5. Inter型号的cpu
6. public class IntelCpu implements Cpu {
7. /\*\*
8. \* CPU的针脚数
9. \*/
10. private int pins = 0;
11. public IntelCpu(int pins){
12. this.pins = pins;
13. }
14. @Override
15. public void calculate() {
16. // TODO Auto-generated method stub
17. System.out.println("Intel CPU的针脚数：" + pins);
18. }
19. AMD型号的cpu
20. public class AmdCpu implements Cpu {
21. /\*\*
22. \* CPU的针脚数
23. \*/
24. private int pins = 0;
25. public AmdCpu(int pins){
26. this.pins = pins;
27. }
28. @Override
29. public void calculate() {
30. // TODO Auto-generated method stub
31. System.out.println("AMD CPU的针脚数：" + pins);
32. }
33. }
34. }
35. 主板接口
36. public interface Mainboard {
37. public void installCPU();
38. }
39. Inter主板
40. public class IntelMainboard implements Mainboard {
41. /\*\*
42. \* CPU插槽的孔数
43. \*/
44. private int cpuHoles = 0;
45. /\*\*
46. \* 构造方法，传入CPU插槽的孔数
47. \* @param cpuHoles
48. \*/
49. public IntelMainboard(int cpuHoles){
50. this.cpuHoles = cpuHoles;
51. }
52. @Override
53. public void installCPU() {
54. // TODO Auto-generated method stub
55. System.out.println("Intel主板的CPU插槽孔数是：" + cpuHoles);
56. }
57. }
58. AMD的主板
59. public class AmdMainboard implements Mainboard {
60. /\*\*
61. \* CPU插槽的孔数
62. \*/
63. private int cpuHoles = 0;
64. /\*\*
65. \* 构造方法，传入CPU插槽的孔数
66. \* @param cpuHoles
67. \*/
68. public AmdMainboard(int cpuHoles){
69. this.cpuHoles = cpuHoles;
70. }
71. @Override
72. public void installCPU() {
73. // TODO Auto-generated method stub
74. System.out.println("AMD主板的CPU插槽孔数是：" + cpuHoles);
75. }
76. }
77. 抽象工厂接口：负责生产全部品种的设备
78. public interface AbstractFactory {
79. /\*\*
80. \* 创建CPU对象
81. \* @return CPU对象
82. \*/
83. public Cpu createCpu();
84. /\*\*
85. \* 创建主板对象
86. \* @return 主板对象
87. \*/
88. public Mainboard createMainboard();
89. }
90. Inter的工厂，可以生产Inter的全部产品
91. public class IntelFactory implements AbstractFactory {
92. @Override
93. public Cpu createCpu() {
94. // TODO Auto-generated method stub
95. return new IntelCpu(755);
96. }
97. @Override
98. public Mainboard createMainboard() {
99. // TODO Auto-generated method stub
100. return new IntelMainboard(755);
101. }
102. }
103. AMD工厂，可以生产AMD的全套设备
104. public class AmdFactory implements AbstractFactory {
105. @Override
106. public Cpu createCpu() {
107. // TODO Auto-generated method stub
108. return new AmdCpu(938);
109. }
110. @Override
111. public Mainboard createMainboard() {
112. // TODO Auto-generated method stub
113. return new AmdMainboard(938);
114. }
115. }
116. 装机工程师类
117. public class ComputerEngineer {
118. /\*\*
119. \* 定义组装机需要的CPU
120. \*/
121. private Cpu cpu = null;
122. /\*\*
123. \* 定义组装机需要的主板
124. \*/
125. private Mainboard mainboard = null;
126. public void makeComputer(AbstractFactory af){
127. /\*\*
128. \* 组装机器的基本步骤
129. \*/
130. //1:首先准备好装机所需要的配件
131. prepareHardwares(af);
132. //2:组装机器
133. //3:测试机器
134. //4：交付客户
135. }
136. private void prepareHardwares(AbstractFactory af){
137. //这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个
138. //可是，装机工程师并不知道如何去创建，怎么办呢？
140. //直接找相应的工厂获取
141. this.cpu = af.createCpu();
142. this.mainboard = af.createMainboard();
144. //测试配件是否好用
145. this.cpu.calculate();
146. this.mainboard.installCPU();
147. }
148. }
149. 客户端类
150. public class Client {
151. public static void main(String[]args){
152. //创建装机工程师对象
153. ComputerEngineer cf = new ComputerEngineer();
154. //客户选择并创建需要使用的产品对象
155. AbstractFactory af = new IntelFactory();
156. //告诉装机工程师自己选择的产品，让装机工程师组装电脑
157. cf.makeComputer(af);
158. }
159. }

抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法。比如上面例子中的主板和CPU，都是为了组装一台电脑的相关对象。不同的装机方案，代表一种具体的电脑系列。



**3、单例模式（Singleton）**

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

首先我们写一个简单的单例类：

1. **public** **class** Singleton {
3. /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/
4. **private** **static** Singleton instance = **null**;
6. /\* 私有构造方法，防止被实例化 \*/
7. **private** Singleton() {
8. }
10. /\* 静态工程方法，创建实例 \*/
11. **public** **static** Singleton getInstance() {
12. **if** (instance == **null**) {
13. instance = **new** Singleton();
14. }
15. **return** instance;
16. }
18. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
19. **public** Object readResolve() {
20. **return** instance;
21. }
22. }

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

1. **public** **static** **synchronized** Singleton getInstance() {
2. **if** (instance == **null**) {
3. instance = **new** Singleton();
4. }
5. **return** instance;
6. }

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

1. **public** **static** Singleton getInstance() {
2. **if** (instance == **null**) {
3. **synchronized** (instance) {
4. **if** (instance == **null**) {
5. instance = **new** Singleton();
6. }
7. }
8. }
9. **return** instance;
10. }

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创 建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也 就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间， 然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例：

a>A、B线程同时进入了第一个if判断

b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();

c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。

d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。

e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

1. **private** **static** **class** SingletonFactory{
2. **private** **static** Singleton instance = **new** Singleton();
3. }
4. **public** **static** Singleton getInstance(){
5. **return** SingletonFactory.instance;
6. }

实际情况是，单例模式使用内部类来维护单例的实现，JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用 getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不 用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

1. **public** **class** Singleton {
3. /\* 私有构造方法，防止被实例化 \*/
4. **private** Singleton() {
5. }
7. /\* 此处使用一个内部类来维护单例 \*/
8. **private** **static** **class** SingletonFactory {
9. **private** **static** Singleton instance = **new** Singleton();
10. }
12. /\* 获取实例 \*/
13. **public** **static** Singleton getInstance() {
14. **return** SingletonFactory.instance;
15. }
17. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
18. **public** Object readResolve() {
19. **return** getInstance();
20. }
21. }

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适 合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加 synchronized关键字，也是可以的：

1. **public** **class** SingletonTest {
3. **private** **static** SingletonTest instance = **null**;
5. **private** SingletonTest() {
6. }
8. **private** **static** **synchronized** **void** syncInit() {
9. **if** (instance == **null**) {
10. instance = **new** SingletonTest();
11. }
12. }
14. **public** **static** SingletonTest getInstance() {
15. **if** (instance == **null**) {
16. syncInit();
17. }
18. **return** instance;
19. }
20. }

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

**补充：采用"影子实例"的办法为单例对象的属性同步更新**

1. **public** **class** SingletonTest {
3. **private** **static** SingletonTest instance = **null**;
4. **private** Vector properties = **null**;
6. **public** Vector getProperties() {
7. **return** properties;
8. }
10. **private** SingletonTest() {
11. }
13. **private** **static** **synchronized** **void** syncInit() {
14. **if** (instance == **null**) {
15. instance = **new** SingletonTest();
16. }
17. }
19. **public** **static** SingletonTest getInstance() {
20. **if** (instance == **null**) {
21. syncInit();
22. }
23. **return** instance;
24. }
26. **public** **void** updateProperties() {
27. SingletonTest shadow = **new** SingletonTest();
28. properties = shadow.getProperties();
29. }
30. }

通过单例模式的学习告诉我们：

1、单例模式理解起来简单，但是具体实现起来还是有一定的难度。

2、synchronized关键字锁定的是对象，在用的时候，一定要在恰当的地方使用（注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁）。

到这儿，单例模式基本已经讲完了，结尾处，笔者突然想到另一个问题，就是采用类的静态方法，实现单例模式的效果，也是可行的，此处二者有什么不同？

首先，静态类不能实现接口。（从类的角度说是可以的，但是那样就破坏了静态了。因为接口中不允许有static修饰的方法，所以即使实现了也是非静态的）

其次，单例可以被延迟初始化，静态类一般在第一次加载是初始化。之所以延迟加载，是因为有些类比较庞大，所以延迟加载有助于提升性能。

再次，单例类可以被继承，他的方法可以被覆写。但是静态类内部方法都是static，无法被覆写。

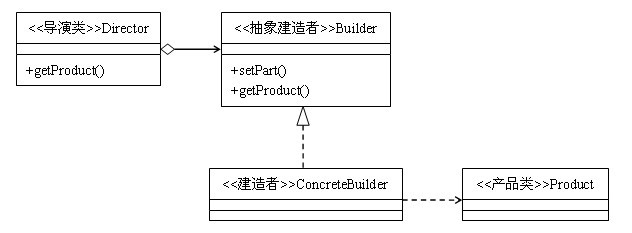
最后一点，单例类比较灵活，毕竟从实现上只是一个普通的Java类，只要满足单例的基本需求，你可以在里面随心所欲的实现一些其它功能，但是静态类不行。 从上面这些概括中，基本可以看出二者的区别，但是，从另一方面讲，我们上面最后实现的那个单例模式，内部就是用一个静态类来实现的，所以，二者有很大的关联，只是我们考虑问题的层面不同罢了。两种思想的结合，才能造就出完美的解决方案，就像HashMap采用数组+链表来实现一样，其实生活中很多事情都是这样，单用不同的方法来处理问题，总是有优点也有缺点，最完美的方法是，结合各个方法的优点，才能最好的解决问题！

**4、建造者模式（Builder）**

**定义：**将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

**类型：**创建类模式

**类图：**

****

**四个要素**

* **产品类：**一般是一个较为复杂的对象，也就是说创建对象的过程比较复杂，一般会有比较多的代码量。在本类图中，产品类是一个具体的类，而非抽象类。实际编程中，产品类可以是由一个抽象类与它的不同实现组成，也可以是由多个抽象类与他们的实现组成。
* **抽象建造者：**引入抽象建造者的目的，是为了将建造的具体过程交与它的子类来实现。这样更容易扩展。一般至少会有两个抽象方法，一个用来建造产品，一个是用来返回产品。
* **建造者：**实现抽象类的所有未实现的方法，具体来说一般是两项任务：组建产品；返回组建好的产品。
* **导演类：**负责调用适当的建造者来组建产品，导演类一般不与产品类发生依赖关系，与导演类直接交互的是建造者类。一般来说，导演类被用来封装程序中易变的部分。

**代码实现**

1. class Product {
2. private String name;
3. private String type;
4. public void showProduct(){
5. System.out.println("名称："+name);
6. System.out.println("型号："+type);
7. }
8. public void setName(String name) {
9. this.name =name;
10. }
11. public void setType(String type) {
12. this.type =type;
13. }
14. }
16. abstract class Builder {
17. public abstract void setPart(String arg1, String arg2);
18. public abstract Product getProduct();
19. }
20. class ConcreteBuilder extends Builder {
21. private Product product = new Product();
23. public Product getProduct() {
24. return product;
25. }
27. public void setPart(String arg1, String arg2) {
28. product.setName(arg1);
29. product.setType(arg2);
30. }
31. }
33. public class Director {
34. private Builder builder = new ConcreteBuilder();
35. public Product getAProduct(){
36. builder.setPart("宝马汽车","X7");
37. return builder.getProduct();
38. }
39. public Product getBProduct(){
40. builder.setPart("奥迪汽车","Q5");
41. return builder.getProduct();
42. }
43. }
44. public class Client {
45. public static void main(String[] args){
46. Director director = new Director();
47. Product product1 = director.getAProduct();
48. product1.showProduct();
50. Product product2 = director.getBProduct();
51. product2.showProduct();
52. }
53. }

**建造者模式的优点**

   首先，建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在导演类中对整体而言可以取得比较好的稳定性。

  其次，建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。

**建造者模式与工厂模式的区别**

我们可以看到，建造者模式与工厂模式是极为相似的，总体上，建造者模式仅仅只比工厂模式多了一个“导演类”的角色。在建造者模式的类图中，假如把这个导演类看做是最终调用的客户端，那么图中剩余的部分就可以看作是一个简单的工厂模式了。

  与工厂模式相比，建造者模式一般用来创建**更为复杂的对象**，因为对象的创建过程更为复杂，因此将对 象的创建过程独立出来组成一个新的类——导演类。也就是说，工厂模式是将对象的全部创建过程封装在工厂类中，由工厂类向客户端提供最终的产品；而建造者模 式中，建造者类一般只提供产品类中各个组件的建造，而将具体建造过程交付给导演类。由导演类负责将各个组件按照特定的规则组建为产品，然后将组建好的产品 交付给客户端。

**总结** 建造者模式与工厂模式类似，他们都是建造者模式，适用的场景也很相似。一般来说，如果产品的建造**很复杂**，那么请用工厂模式；如果产品的建造**更复杂**，那么请用建造者模式。

**5、原型模式（Prototype）**

原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。本小结会通过对象的复制，进行讲解。在Java中，复制对象是通过clone()实现的，先创建一个原型类：

1. **public** **class** Prototype **implements** Cloneable {
3. **public** Object clone() **throws** CloneNotSupportedException {
4. Prototype proto = (Prototype) **super**.clone();
5. **return** proto;
6. }
7. }

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空 接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone() 调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，我会在另一篇文章中，关于解读 Java中本地方法的调用，此处不再深究。在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

此处，写一个深浅复制的例子：

1. **public** **class** Prototype **implements** Cloneable, Serializable {
3. **private** **static** **final** **long** serialVersionUID = 1L;
4. **private** String string;
6. **private** SerializableObject obj;
8. /\* 浅复制 \*/
9. **public** Object clone() **throws** CloneNotSupportedException {
10. Prototype proto = (Prototype) **super**.clone();
11. **return** proto;
12. }
14. /\* 深复制 \*/
15. **public** Object deepClone() **throws** IOException, ClassNotFoundException {
17. /\* 写入当前对象的二进制流 \*/
18. ByteArrayOutputStream bos = **new** ByteArrayOutputStream();
19. ObjectOutputStream oos = **new** ObjectOutputStream(bos);
20. oos.writeObject(**this**);
22. /\* 读出二进制流产生的新对象 \*/
23. ByteArrayInputStream bis = **new** ByteArrayInputStream(bos.toByteArray());
24. ObjectInputStream ois = **new** ObjectInputStream(bis);
25. **return** ois.readObject();
26. }
28. **public** String getString() {
29. **return** string;
30. }
32. **public** **void** setString(String string) {
33. **this**.string = string;
34. }
36. **public** SerializableObject getObj() {
37. **return** obj;
38. }
40. **public** **void** setObj(SerializableObject obj) {
41. **this**.obj = obj;
42. }
44. }
46. **class** SerializableObject **implements** Serializable {
47. **private** **static** **final** **long** serialVersionUID = 1L;
48. }

要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。

**B、结构模式（7种）**

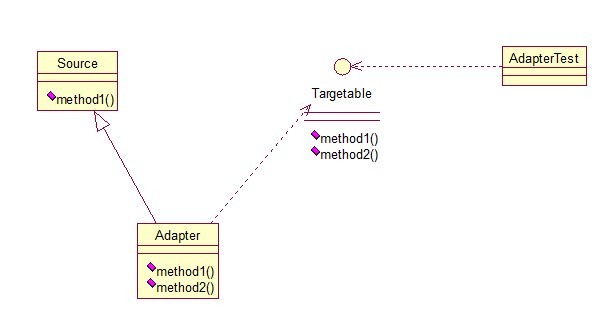
我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：



**6、适配器模式**

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

**01、类的适配器模式**



核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口是Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

1. **public** **class** Source {
3. **public** **void** method1() {
4. System.out.println("this is original method!");
5. }
6. }
7. **public** **interface** Targetable {
9. /\* 与原类中的方法相同 \*/
10. **public** **void** method1();
12. /\* 新类的方法 \*/
13. **public** **void** method2();
14. }
15. **public** **class** Adapter **extends** Source **implements** Targetable {
17. @Override
18. **public** **void** method2() {
19. System.out.println("this is the targetable method!");
20. }
21. }

Adapter类继承Source类，实现Targetable接口，下面是测试类：

1. **public** **class** AdapterTest {
3. **public** **static** **void** main(String[] args) {
4. Targetable target = **new** Adapter();
5. target.method1();
6. target.method2();
7. }
8. }

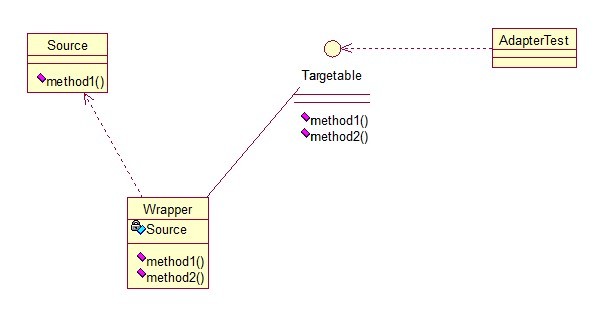
输出：

this is original method!  
this is the targetable method!

这样Targetable接口的实现类就具有了Source类的功能。

**02、对象的适配器模式**

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。看图：



只需要修改Adapter类的源码即可：

1. **public** **class** Wrapper **implements** Targetable {
3. **private** Source source;
5. **public** Wrapper(Source source){
6. **super**();
7. **this**.source = source;
8. }
9. @Override
10. **public** **void** method2() {
11. System.out.println("this is the targetable method!");
12. }
14. @Override
15. **public** **void** method1() {
16. source.method1();
17. }
18. }

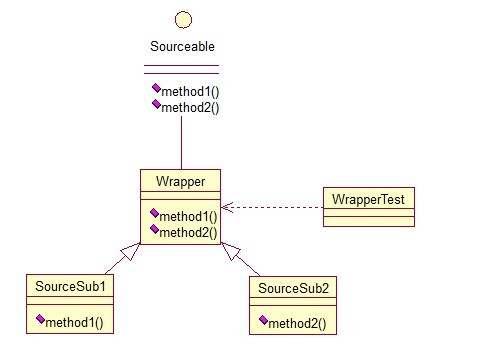
测试类：

1. **public** **class** AdapterTest {
3. **public** **static** **void** main(String[] args) {
4. Source source = **new** Source();
5. Targetable target = **new** Wrapper(source);
6. target.method1();
7. target.method2();
8. }
9. }

输出与第一种一样，只是适配的方法不同而已。

**03、接口的适配器模式**

第三种适配器模式是**接口的适配器模式**， 接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所 有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

1. **public** **interface** Sourceable {
3. **public** **void** method1();
4. **public** **void** method2();
5. }

抽象类Wrapper2：

1. **public** **abstract** **class** Wrapper2 **implements** Sourceable{
3. **public** **void** method1(){}
4. **public** **void** method2(){}
5. }
6. **public** **class** SourceSub1 **extends** Wrapper2 {
7. **public** **void** method1(){
8. System.out.println("the sourceable interface's first Sub1!");
9. }
10. }
11. **public** **class** SourceSub2 **extends** Wrapper2 {
12. **public** **void** method2(){
13. System.out.println("the sourceable interface's second Sub2!");
14. }
15. }
16. **public** **class** WrapperTest {
18. **public** **static** **void** main(String[] args) {
19. Sourceable source1 = **new** SourceSub1();
20. Sourceable source2 = **new** SourceSub2();
22. source1.method1();
23. source1.method2();
24. source2.method1();
25. source2.method2();
26. }
27. }

测试输出：

the sourceable interface's first Sub1!  
the sourceable interface's second Sub2!

达到了我们的效果！

 讲了这么多，总结一下三种适配器模式的应用场景：

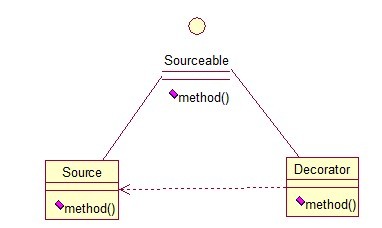
类的适配器模式：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

**7、装饰模式（Decorator）**

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }
4. **public** **class** Source **implements** Sourceable {
6. @Override
7. **public** **void** method() {
8. System.out.println("the original method!");
9. }
10. }
11. **public** **class** Decorator **implements** Sourceable {
13. **private** Sourceable source;
15. **public** Decorator(Sourceable source){
16. **super**();
17. **this**.source = source;
18. }
19. @Override
20. **public** **void** method() {
21. System.out.println("before decorator!");
22. source.method();
23. System.out.println("after decorator!");
24. }
25. }

测试类：

1. **public** **class** DecoratorTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source = **new** Source();
5. Sourceable obj = **new** Decorator(source);
6. obj.method();
7. }
8. }

输出：

before decorator!  
the original method!  
after decorator!

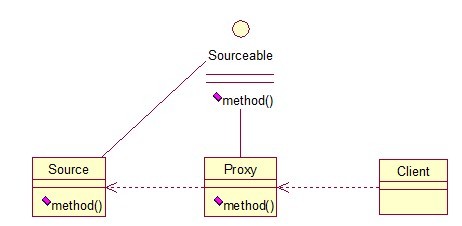
装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

**8、代理模式（Proxy）**

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有 专长，可以替我们进行操作，表达我们的想法。先来看看关系图：

根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }
4. **public** **class** Source **implements** Sourceable {
6. @Override
7. **public** **void** method() {
8. System.out.println("the original method!");
9. }
10. }
11. **public** **class** Proxy **implements** Sourceable {
13. **private** Source source;
14. **public** Proxy(){
15. **super**();
16. **this**.source = **new** Source();
17. }
18. @Override
19. **public** **void** method() {
20. before();
21. source.method();
22. atfer();
23. }
24. **private** **void** atfer() {
25. System.out.println("after proxy!");
26. }
27. **private** **void** before() {
28. System.out.println("before proxy!");
29. }
30. }

测试类：

1. **public** **class** ProxyTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source = **new** Proxy();
5. source.method();
6. }
8. }

输出：

before proxy!  
the original method!  
after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

**JDK原生动态代理和CGLIB动态代理**。

动态代理在Java中有着广泛的应用，比如Spring AOP，Hibernate数据查询、测试框架的后端mock、RPC，Java注解对象获取等。静态代理的代理关系在编译时就确定了，而动态代理的代理关系是在运行时确定的。静态代理实现简单，适合于代理类较少且确定的情况，而动态代理则给我们提供了更大的灵活性。今天我们来探讨Java中两种常见的动态代理方式：

JDK原生动态代理

先从直观的示例说起，假设我们有一个接口Hello和一个简单实现HelloImp：

1. // 接口
2. **interface** Hello{
3. String sayHello(String str);
4. }
5. // 实现
6. **class** HelloImp **implements** Hello{
7. @Override
8. **public** String sayHello(String str) {
9. **return** "HelloImp: " + str;
10. }
11. }

这是Java种再常见不过的场景，使用接口制定协议，然后用不同的实现来实现具体行为。假设你已经拿到上述类库，如果我们想通过日志记录对sayHello()的调用，使用静态代理可以这样做：

1. // 静态代理方式
2. **class** StaticProxiedHello **implements** Hello{
3. ...
4. **private** Hello hello = **new** HelloImp();
5. @Override
6. **public** String sayHello(String str) {
7. logger.info("You said: " + str);
8. **return** hello.sayHello(str);
9. }
10. }

上例中静态代理类StaticProxiedHello作为HelloImp的代理，实现了相同的Hello接口。用Java动态代理可以这样做：

首先实现一个InvocationHandler，方法调用会被转发到该类的invoke()方法。

然后在需要使用Hello的时候，通过JDK动态代理获取Hello的代理对象。

1. // Java Proxy
2. // 1. 首先实现一个InvocationHandler，方法调用会被转发到该类的invoke()方法。
3. **class** LogInvocationHandler **implements** InvocationHandler{
4. ...
5. **private** Hello hello;
6. **public** LogInvocationHandler(Hello hello) {
7. **this**.hello = hello;
8. }
9. @Override
10. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
11. **if**("sayHello".equals(method.getName())) {
12. logger.info("You said: " + Arrays.toString(args));
13. }
14. **return** method.invoke(hello, args);
15. }
16. }
17. // 2. 然后在需要使用Hello的时候，通过JDK动态代理获取Hello的代理对象。
18. Hello h=new HelloImp();
19. LogInvocationHandler l=new LogInvocationHandler**(h);**
20. Hello hello = (Hello)Proxy.newProxyInstance(
21. h.getClass().getClassLoader(), // 1. 类加载器
22. h.getClass().getInterfaces(), // 2. 代理需要实现的接口，可以有多个
23. l);// 3. 方法调用的实际处理者
24. System.out.println(hello.sayHello("I love you!"));

运行上述代码输出结果：

1. 1.  日志信息: You said: [I love you!]
2. HelloImp: I love you

上述代码的关键是Proxy.newProxyInstance( h.getClass().getClassLoader(),

h.getClass().getInterfaces())该方法会根据指定的参数动态创建代理对象。三个参数的意义如下：

loader，指定代理对象的类加载器；

interfaces，代理对象需要实现的接口，可以同时指定多个接口；

handler，方法调用的实际处理者，代理对象的方法调用都会转发到这里（\*注意1）。

newProxyInstance()会返回一个实现了指定接口的代理对象，对该对象的所有方法调用都会转发给InvocationHandler.invoke()方法。理解上述代码需要对Java反射机制有一定了解。动态代理神奇的地方就是：

代理对象是在程序运行时产生的，而不是编译期；

**对代理对象的所有接口方法调用都会转发到InvocationHandler.invoke()方法**，在invoke()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；之后我们通过某种方式执行真正的方法体，示例中通过反射调用了Hello对象的相应方法，还可以通过RPC调用远程方法。

*注意1：对于从Object中继承的方法，JDK Proxy会把hashCode()、equals()、toString()这三个非接口方法转发给InvocationHandler，其余的Object方法则不会转发。详见*[*JDK Proxy官方文档*](https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html)*。*

如果对JDK代理后的对象类型进行深挖，可以看到如下信息：

1. superClass=**class** java.lang.reflect.Proxy
2. interfaces:
3. **interface** jdkproxy.Hello
4. invocationHandler=jdkproxy.LogInvocationHandler@a09ee92

代理对象的类型是jdkproxy.$Proxy0，这是个动态生成的类型，类名是形如$ProxyN的形式；父类是java.lang.reflect.Proxy，所有的JDK动态代理都会继承这个类；同时实现了Hello接口，也就是我们接口列表中指定的那些接口。

如果你还对jdkproxy.$Proxy0具体实现感兴趣，它大致长这个样子：

1. // JDK代理类具体实现
2. **public** **final** **class** $Proxy0 **extends** Proxy **implements** Hello
3. {
4. ...
5. **public** $Proxy0(InvocationHandler invocationhandler)
6. {
7. **super**(invocationhandler);
8. }
9. ...
10. @Override
11. **public** **final** String sayHello(String str){
12. ...
13. **return** **super**.h.invoke(**this**, m3, **new** Object[] {str});// 将方法调用转发给invocationhandler
14. ...
15. }
16. ...
17. }

这些逻辑没什么复杂之处，但是他们是在运行时动态产生的，无需我们手动编写。更多详情，可参考BrightLoong的[Java静态代理&动态代理笔记](https://www.jianshu.com/p/e2917b0b9614)

Java动态代理为我们提供了非常灵活的代理机制，但Java动态代理是基于接口的，如果对象没有实现接口我们该如何代理呢？CGLIB登场。

**CGLIB动态代理**

[CGLIB](https://github.com/cglib/cglib)(Code Generation Library)是一个基于[ASM](http://www.baeldung.com/java-asm)的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理。

来看示例，假设我们有一个没有实现任何接口的类HelloConcrete：

1. **public** **class** HelloConcrete {
2. **public** String sayHello(String str) {
3. **return** "HelloConcrete: " + str;
4. }
5. }

因为没有实现接口该类无法使用JDK代理，通过CGLIB代理实现如下：

首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。

然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。

1. // CGLIB动态代理
2. // 1. 首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。
3. **class** MyMethodInterceptor **implements** MethodInterceptor{
4. ...
5. @Override
6. **public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {
7. logger.info("You said: " + Arrays.toString(args));
8. **return** proxy.invokeSuper(obj, args);
9. }
10. }
11. // 2. 然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。
12. Enhancer enhancer = **new** Enhancer();
13. enhancer.setSuperclass(HelloConcrete.**class**);
14. enhancer.setCallback(**new** MyMethodInterceptor());
16. HelloConcrete hello = (HelloConcrete)enhancer.create();
17. System.out.println(hello.sayHello("I love you!"));

运行上述代码输出结果：

1. 日志信息: You said: [I love you!]
2. HelloConcrete: I love you!

上述代码中，我们通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，**对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法**，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。CGLIG中[MethodInterceptor](http://cglib.sourceforge.net/apidocs/net/sf/cglib/proxy/MethodInterceptor.html)的作用跟JDK代理中的InvocationHandler很类似，都是方法调用的中转站。

*注意：对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。*

如果对CGLIB代理之后的对象类型进行深挖，可以看到如下信息：

1. # HelloConcrete代理对象的类型信息
2. **class**=**class** cglib.HelloConcrete$$EnhancerByCGLIB$$e3734e52
3. superClass=**class** lh.HelloConcrete
4. interfaces:
5. **interface** net.sf.cglib.proxy.Factory
6. invocationHandler=not java proxy **class**

我们看到使用CGLIB代理之后的对象类型是cglib.HelloConcrete$$EnhancerByCGLIB$$e3734e52，这是CGLIB动态生成的类型；父类是HelloConcrete，印证了CGLIB是通过继承实现代理；同时实现了net.sf.cglib.proxy.Factory接口，这个接口是CGLIB自己加入的，包含一些工具方法。

注意，既然是继承就不得不考虑final的问题。我们知道final类型不能有子类，所以CGLIB不能代理final类型，遇到这种情况会抛出类似如下异常：

1. java.lang.IllegalArgumentException: Cannot subclass **final** **class** cglib.HelloConcrete

同样的，final方法是不能重载的，所以也不能通过CGLIB代理，遇到这种情况不会抛异常，而是会跳过final方法只代理其他方法。

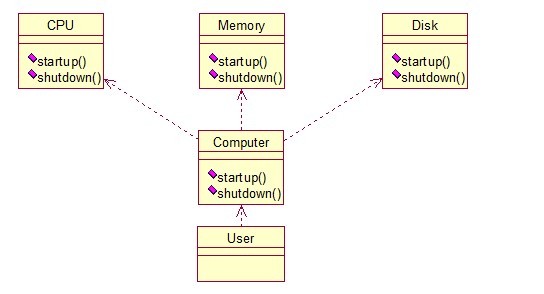
如果你还对代理类cglib.HelloConcrete$$EnhancerByCGLIB$$e3734e52具体实现感兴趣，它大致长这个样子：

1. // CGLIB代理类具体实现
2. **public** **class** HelloConcrete$$EnhancerByCGLIB$$e3734e52
3. **extends** HelloConcrete
4. **implements** Factory
5. {
6. ...
7. **private** MethodInterceptor CGLIB$CALLBACK\_0; // ~~
8. ...
10. **public** **final** String sayHello(String paramString)
11. {
12. ...
13. MethodInterceptor tmp17\_14 = CGLIB$CALLBACK\_0;
14. **if** (tmp17\_14 != **null**) {
15. // 将请求转发给MethodInterceptor.intercept()方法。
16. **return** (String)tmp17\_14.intercept(**this**,
17. CGLIB$sayHello$0$Method,
18. **new** Object[] { paramString },
19. CGLIB$sayHello$0$Proxy);
20. }
21. **return** **super**.sayHello(paramString);
22. }
23. ...
24. }

上述代码我们看到，当调用代理对象的sayHello()方法时，首先会尝试转发给MethodInterceptor.intercept()方法，如果没有MethodInterceptor就执行父类的sayHello()。这些逻辑没什么复杂之处，但是他们是在运行时动态产生的，无需我们手动编写。如何获取CGLIB代理类字节码可参考[Access the generated byte[] array directly](https://github.com/cglib/cglib/wiki/How-To#access-the-generated-byte-array-directly)。

**9、外观模式（Facade）**

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



我们先看下实现类：

1. **public** **class** CPU {
3. **public** **void** startup(){
4. System.out.println("cpu startup!");
5. }
7. **public** **void** shutdown(){
8. System.out.println("cpu shutdown!");
9. }
10. }
11. **public** **class** Memory {
13. **public** **void** startup(){
14. System.out.println("memory startup!");
15. }
17. **public** **void** shutdown(){
18. System.out.println("memory shutdown!");
19. }
20. }
21. **public** **class** Disk {
23. **public** **void** startup(){
24. System.out.println("disk startup!");
25. }
27. **public** **void** shutdown(){
28. System.out.println("disk shutdown!");
29. }
30. }
31. **public** **class** Computer {
32. **private** CPU cpu;
33. **private** Memory memory;
34. **private** Disk disk;
36. **public** Computer(){
37. cpu = **new** CPU();
38. memory = **new** Memory();
39. disk = **new** Disk();
40. }
42. **public** **void** startup(){
43. System.out.println("start the computer!");
44. cpu.startup();
45. memory.startup();
46. disk.startup();
47. System.out.println("start computer finished!");
48. }
50. **public** **void** shutdown(){
51. System.out.println("begin to close the computer!");
52. cpu.shutdown();
53. memory.shutdown();
54. disk.shutdown();
55. System.out.println("computer closed!");
56. }
57. }

User类如下：

1. **public** **class** User {
3. **public** **static** **void** main(String[] args) {
4. Computer computer = **new** Computer();
5. computer.startup();
6. computer.shutdown();
7. }
8. }

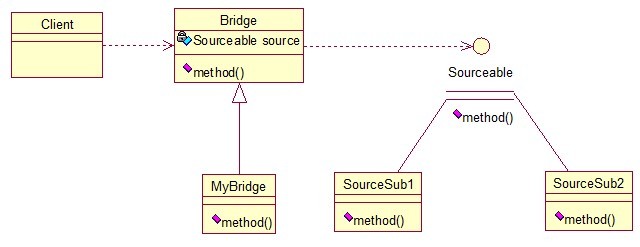
输出：

start the computer!  
cpu startup!  
memory startup!  
disk startup!  
start computer finished!  
begin to close the computer!  
cpu shutdown!  
memory shutdown!  
disk shutdown!  
computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会 带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观 模式！

**10、桥接模式（Bridge）**

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：**将抽象化与实现化解耦，使得二者可以独立变化**， 像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不 用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：



实现代码：

先定义接口：

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }

分别定义两个实现类：

1. **public** **class** SourceSub1 **implements** Sourceable {
3. @Override
4. **public** **void** method() {
5. System.out.println("this is the first sub!");
6. }
7. }
8. **public** **class** SourceSub2 **implements** Sourceable {
10. @Override
11. **public** **void** method() {
12. System.out.println("this is the second sub!");
13. }
14. }

定义一个桥，持有Sourceable的一个实例：

1. **public** **abstract** **class** Bridge {
2. **private** Sourceable source;
4. **public** **void** method(){
5. source.method();
6. }
8. **public** Sourceable getSource() {
9. **return** source;
10. }
12. **public** **void** setSource(Sourceable source) {
13. **this**.source = source;
14. }
15. }
16. **public** **class** MyBridge **extends** Bridge {
17. **public** **void** method(){
18. getSource().method();
19. }
20. }

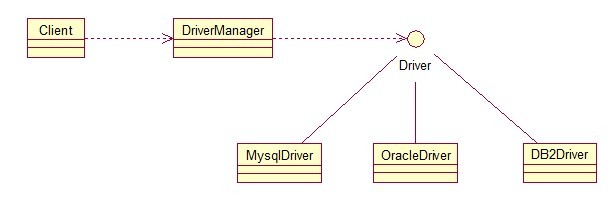
测试类：

1. **public** **class** BridgeTest {
3. **public** **static** **void** main(String[] args) {
5. Bridge bridge = **new** MyBridge();
7. /\*调用第一个对象\*/
8. Sourceable source1 = **new** SourceSub1();
9. bridge.setSource(source1);
10. bridge.method();
12. /\*调用第二个对象\*/
13. Sourceable source2 = **new** SourceSub2();
14. bridge.setSource(source2);
15. bridge.method();
16. }
17. }

output：

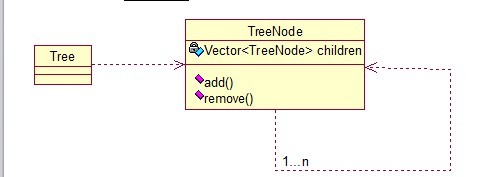
this is the first sub!  
this is the second sub!

这样，就通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们JDBC连接的原理，有数据库学习基础的，一结合就都懂了。



**11、组合模式（Composite）**

组合模式有时又叫**部分-整体**模式在处理类似树形结构的问题时比较方便，看看关系图：



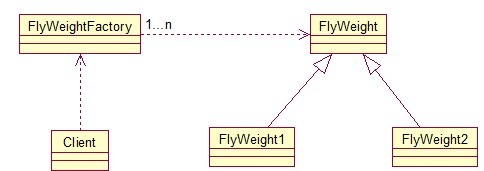
直接来看代码：

1. **public** **class** TreeNode {
3. **private** String name;
4. **private** TreeNode parent;
5. **private** Vector<TreeNode> children = **new** Vector<TreeNode>();
7. **public** TreeNode(String name){
8. **this**.name = name;
9. }
11. **public** String getName() {
12. **return** name;
13. }
15. **public** **void** setName(String name) {
16. **this**.name = name;
17. }
19. **public** TreeNode getParent() {
20. **return** parent;
21. }
23. **public** **void** setParent(TreeNode parent) {
24. **this**.parent = parent;
25. }
27. //添加孩子节点
28. **public** **void** add(TreeNode node){
29. children.add(node);
30. }
32. //删除孩子节点
33. **public** **void** remove(TreeNode node){
34. children.remove(node);
35. }
37. //取得孩子节点
38. **public** Enumeration<TreeNode> getChildren(){
39. **return** children.elements();
40. }
41. }
42. **public** **class** Tree {
44. TreeNode root = **null**;
46. **public** Tree(String name) {
47. root = **new** TreeNode(name);
48. }
50. **public** **static** **void** main(String[] args) {
51. Tree tree = **new** Tree("A");
52. TreeNode nodeB = **new** TreeNode("B");
53. TreeNode nodeC = **new** TreeNode("C");
55. nodeB.add(nodeC);
56. tree.root.add(nodeB);
57. System.out.println("build the tree finished!");
58. }
59. }

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

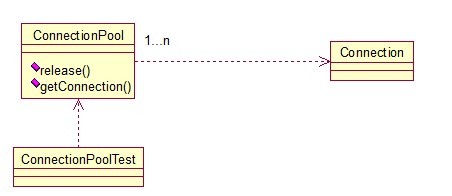
**12、享元模式（Flyweight）**

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。



FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、 password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

看个例子：



看下数据库连接池的代码：

1. **public** **class** ConnectionPool {
3. **private** Vector<Connection> pool;
5. /\*公有属性\*/
6. **private** String url = "jdbc:mysql://localhost:3306/test";
7. **private** String username = "root";
8. **private** String password = "root";
9. **private** String driverClassName = "com.mysql.jdbc.Driver";
11. **private** **int** poolSize = 100;
12. **private** **static** ConnectionPool instance = **null**;
13. Connection conn = **null**;
15. /\*构造方法，做一些初始化工作\*/
16. **private** ConnectionPool() {
17. pool = **new** Vector<Connection>(poolSize);
19. **for** (**int** i = 0; i < poolSize; i++) {
20. **try** {
21. Class.forName(driverClassName);
22. conn = DriverManager.getConnection(url, username, password);
23. pool.add(conn);
24. } **catch** (ClassNotFoundException e) {
25. e.printStackTrace();
26. } **catch** (SQLException e) {
27. e.printStackTrace();
28. }
29. }
30. }
32. /\* 返回连接到连接池 \*/
33. **public** **synchronized** **void** release() {
34. pool.add(conn);
35. }
37. /\* 返回连接池中的一个数据库连接 \*/
38. **public** **synchronized** Connection getConnection() {
39. **if** (pool.size() > 0) {
40. Connection conn = pool.get(0);
41. pool.remove(conn);
42. **return** conn;
43. } **else** {
44. **return** **null**;
45. }
46. }
47. }
48. /\*使用同步单例模式创建ConnectionPool\*/
49. **public static synchronized** ConnectionPool getInstance()
50. {
51. if (instance==null) {
52. **return** new ConnectionPool();
53. }
54. **return** instance;
55. }

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！

**C、关系模式（11种）**

先来张图，看看这11中模式的关系：

第一类：通过父类与子类的关系进行实现。

第二类：两个类之间。

第三类：类的状态。

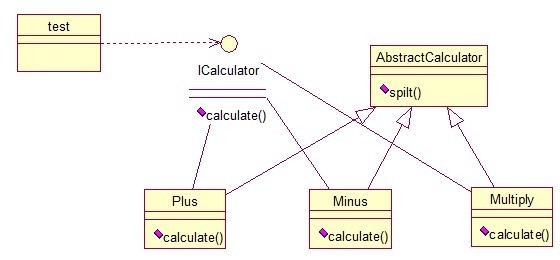
第四类：通过中间类



**父类与子类关系**

**13、策略模式（strategy）**

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数，关系图如下：



图中ICalculator提供同意的方法，  
AbstractCalculator是辅助类，提供辅助方法，接下来，依次实现下每个类：

首先统一接口：

1. **public** **interface** ICalculator {
2. **public** **int** calculate(String exp);
3. }

辅助类：

1. **public** **abstract** **class** AbstractCalculator {
3. **public** **int**[] split(String exp,String opt){
4. String array[] = exp.split(opt);
5. **int** arrayInt[] = **new** **int**[2];
6. arrayInt[0] = Integer.parseInt(array[0]);
7. arrayInt[1] = Integer.parseInt(array[1]);
8. **return** arrayInt;
9. }
10. }

三个实现类：

1. **public** **class** Plus **extends** AbstractCalculator **implements** ICalculator {
3. @Override
4. **public** **int** calculate(String exp) {
5. **int** arrayInt[] = split(exp,"\\+");
6. **return** arrayInt[0]+arrayInt[1];
7. }
8. }
9. **public** **class** Minus **extends** AbstractCalculator **implements** ICalculator {
10. @Override
11. **public** **int** calculate(String exp) {
12. **int** arrayInt[] = split(exp,"-");
13. **return** arrayInt[0]-arrayInt[1];
14. }
15. }
16. **public** **class** Multiply **extends** AbstractCalculator **implements** ICalculator {
18. @Override
19. **public** **int** calculate(String exp) {
20. **int** arrayInt[] = split(exp,"\\\*");
21. **return** arrayInt[0]\*arrayInt[1];
22. }
23. }

简单的测试类：

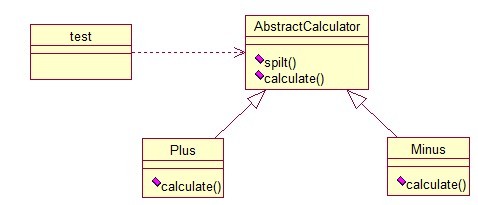
1. **public** **class** StrategyTest {
3. **public** **static** **void** main(String[] args) {
4. String exp = "2+8";
5. ICalculator cal = **new** Plus();
6. **int** result = cal.calculate(exp);
7. System.out.println(result);
8. }
9. }

输出：10

策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

**14、模板方法模式（Template Method）**

解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用，先看个关系图：



就是在AbstractCalculator类中定义一个主方法calculate，calculate()调用spilt()等，Plus和Minus 分别继承AbstractCalculator类，通过对AbstractCalculator的调用实现对子类的调用，看下面的例子：

1. **public** **abstract** **class** AbstractCalculator {
3. /\*主方法，实现对本类其它方法的调用\*/
4. **public** **final** **int** calculate(String exp,String opt){
5. **int** array[] = split(exp,opt);
6. **return** calculate(array[0],array[1]);
7. }
9. /\*被子类重写的方法\*/
10. **abstract** **public** **int** calculate(**int** num1,**int** num2);
12. **public** **int**[] split(String exp,String opt){
13. String array[] = exp.split(opt);
14. **int** arrayInt[] = **new** **int**[2];
15. arrayInt[0] = Integer.parseInt(array[0]);
16. arrayInt[1] = Integer.parseInt(array[1]);
17. **return** arrayInt;
18. }
19. }
20. **public** **class** Plus **extends** AbstractCalculator {
22. @Override
23. **public** **int** calculate(**int** num1,**int** num2) {
24. **return** num1 + num2;
25. }
26. }

测试类：

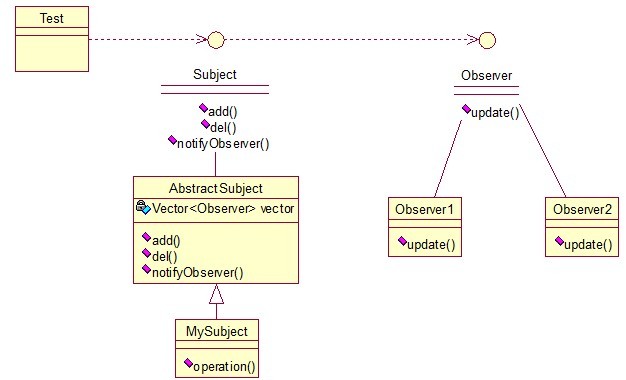
1. **public** **class** StrategyTest {
3. **public** **static** **void** main(String[] args) {
4. String exp = "8+8";
5. AbstractCalculator cal = **new** Plus();
6. **int** result = cal.calculate(exp, "\\+");
7. System.out.println(result);
8. }
9. }

我跟踪下这个小程序的执行过程：首先将exp和"\\+"做参数，调用AbstractCalculator类里的 calculate(String,String)方法，在calculate(String,String)里调用同类的split()，之后再调用 calculate(int ,int)方法，从这个方法进入到子类中，执行完return num1 + num2后，将值返回到AbstractCalculator类，赋给result，打印出来。正好验证了我们开头的思路。

**类之间的关系**

**15、观察者模式（Observer）**

包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！ 对象之间是一种一对多的关系。先来看看关系图：



我解释下这些类的作用：MySubject类就是我们的主对象，Observer1和Observer2是依赖于MySubject的对象，当 MySubject变化时，Observer1和Observer2必然变化。AbstractSubject类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当MySubject变化时，负责通知在列表内存在的对象。我们看实现代码：

一个Observer接口：

1. **public** **interface** Observer {
2. **public** **void** update();
3. }

两个实现类：

1. **public** **class** Observer1 **implements** Observer {
3. @Override
4. **public** **void** update() {
5. System.out.println("observer1 has received!");
6. }
7. }
8. **public** **class** Observer2 **implements** Observer {
10. @Override
11. **public** **void** update() {
12. System.out.println("observer2 has received!");
13. }
15. }

Subject接口及实现类：

1. **public** **interface** Subject {
3. /\*增加观察者\*/
4. **public** **void** add(Observer observer);
6. /\*删除观察者\*/
7. **public** **void** del(Observer observer);
9. /\*通知所有的观察者\*/
10. **public** **void** notifyObservers();
12. /\*自身的操作\*/
13. **public** **void** operation();
14. }
15. **public** **abstract** **class** AbstractSubject **implements** Subject {
17. **private** Vector<Observer> vector = **new** Vector<Observer>();
18. @Override
19. **public** **void** add(Observer observer) {
20. vector.add(observer);
21. }
23. @Override
24. **public** **void** del(Observer observer) {
25. vector.remove(observer);
26. }
28. @Override
29. **public** **void** notifyObservers() {
30. Enumeration<Observer> enumo = vector.elements();
31. **while**(enumo.hasMoreElements()){
32. enumo.nextElement().update();
33. }
34. }
35. }
36. **public** **class** MySubject **extends** AbstractSubject {
38. @Override
39. **public** **void** operation() {
40. System.out.println("update self!");
41. notifyObservers();
42. }
44. }

测试类：

1. **public** **class** ObserverTest {
3. **public** **static** **void** main(String[] args) {
4. Subject sub = **new** MySubject();
5. sub.add(**new** Observer1());
6. sub.add(**new** Observer2());
8. sub.operation();
9. }
11. }

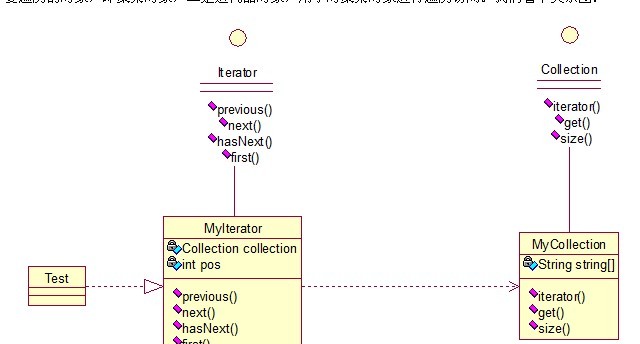
输出：

update self!  
observer1 has received!  
observer2 has received!

 这些东西，其实不难，只是有些抽象，不太容易整体理解，建议读者：**根据关系图，新建项目，自己写代码（或者参考我的代码）,按照总体思路走一遍，这样才能体会它的思想，理解起来容易！**

**16、迭代子模式（Iterator）**

顾名思义，迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见，如果对集合类比较熟悉的话，理解本模式会十分轻松。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。我们看下关系图：



这个思路和我们常用的一模一样，MyCollection中定义了集合的一些操作，MyIterator中定义了一系列迭代操作，且持有Collection实例，我们来看看实现代码：

两个接口：

1. **public** **interface** Collection {
3. **public** Iterator iterator();
5. /\*取得集合元素\*/
6. **public** Object get(**int** i);
8. /\*取得集合大小\*/
9. **public** **int** size();
10. }
11. **public** **interface** Iterator {
12. //前移
13. **public** Object previous();
15. //后移
16. **public** Object next();
17. **public** **boolean** hasNext();
19. //取得第一个元素
20. **public** Object first();
21. }

两个实现：

1. **public** **class** MyCollection **implements** Collection {
3. **public** String string[] = {"A","B","C","D","E"};
4. @Override
5. **public** Iterator iterator() {
6. **return** **new** MyIterator(**this**);
7. }
9. @Override
10. **public** Object get(**int** i) {
11. **return** string[i];
12. }
14. @Override
15. **public** **int** size() {
16. **return** string.length;
17. }
18. }
19. **public** **class** MyIterator **implements** Iterator {
21. **private** Collection collection;
22. **private** **int** pos = -1;
24. **public** MyIterator(Collection collection){
25. **this**.collection = collection;
26. }
28. @Override
29. **public** Object previous() {
30. **if**(pos > 0){
31. pos--;
32. }
33. **return** collection.get(pos);
34. }
36. @Override
37. **public** Object next() {
38. **if**(pos<collection.size()-1){
39. pos++;
40. }
41. **return** collection.get(pos);
42. }
44. @Override
45. **public** **boolean** hasNext() {
46. **if**(pos<collection.size()-1){
47. **return** **true**;
48. }**else**{
49. **return** **false**;
50. }
51. }
53. @Override
54. **public** Object first() {
55. pos = 0;
56. **return** collection.get(pos);
57. }
59. }

测试类：

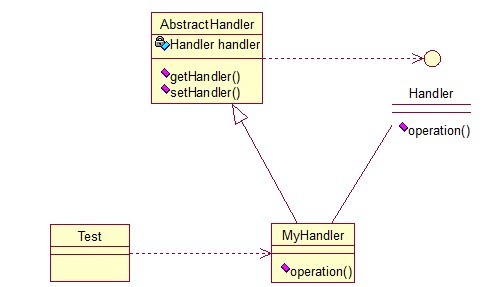
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Collection collection = **new** MyCollection();
5. Iterator it = collection.iterator();
7. **while**(it.hasNext()){
8. System.out.println(it.next());
9. }
10. }
11. }

输出：A B C D E

此处我们貌似模拟了一个集合类的过程，感觉是不是很爽？其实JDK中各个类也都是这些基本的东西，加一些设计模式，再加一些优化放到一起的，只要我们把这些东西学会了，掌握好了，我们也可以写出自己的集合类，甚至框架！

**17、责任链模式（Chain of Responsibility）**

接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。先看看关系图：



Abstracthandler类提供了get和set方法，方便MyHandle类设置和修改引用对象，MyHandle类是核心，实例化后生成一系列相互持有的对象，构成一条链。

1. **public** **interface** Handler {
2. **public** **void** operator();
3. }
4. **public** **abstract** **class** AbstractHandler {
6. **private** Handler handler;
8. **public** Handler getHandler() {
9. **return** handler;
10. }
12. **public** **void** setHandler(Handler handler) {
13. **this**.handler = handler;
14. }
16. }
17. **public** **class** MyHandler **extends** AbstractHandler **implements** Handler {
19. **private** String name;
21. **public** MyHandler(String name) {
22. **this**.name = name;
23. }
25. @Override
26. **public** **void** operator() {
27. System.out.println(name+"deal!");
28. **if**(getHandler()!=**null**){
29. getHandler().operator();
30. }
31. }
32. }
33. **public** **class** Test {
35. **public** **static** **void** main(String[] args) {
36. MyHandler h1 = **new** MyHandler("h1");
37. MyHandler h2 = **new** MyHandler("h2");
38. MyHandler h3 = **new** MyHandler("h3");
40. h1.setHandler(h2);
41. h2.setHandler(h3);
43. h1.operator();
44. }
45. }

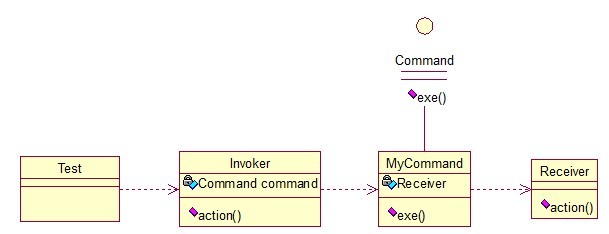
输出：

h1deal!  
h2deal!  
h3deal!

此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

**18、命令模式（Command）**

命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看 看关系图：



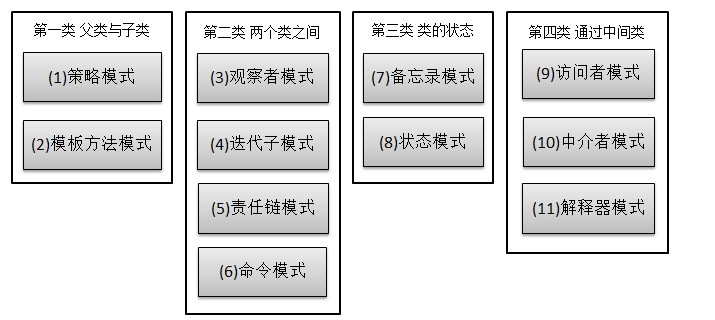
Invoker是调用者（司令员），Receiver是被调用者（士兵），MyCommand是命令，实现了Command接口，持有接收对象，看实现代码：

1. **public** **interface** Command {
2. **public** **void** exe();
3. }
4. **public** **class** MyCommand **implements** Command {
6. **private** Receiver receiver;
8. **public** MyCommand(Receiver receiver) {
9. **this**.receiver = receiver;
10. }
12. @Override
13. **public** **void** exe() {
14. receiver.action();
15. }
16. }
17. **public** **class** Receiver {
18. **public** **void** action(){
19. System.out.println("command received!");
20. }
21. }
22. **public** **class** Invoker {
24. **private** Command command;
26. **public** Invoker(Command command) {
27. **this**.command = command;
28. }
30. **public** **void** action(){
31. command.exe();
32. }
33. }
34. **public** **class** Test {
36. **public** **static** **void** main(String[] args) {
37. Receiver receiver = **new** Receiver();
38. Command cmd = **new** MyCommand(receiver);
39. Invoker invoker = **new** Invoker(cmd);
40. invoker.action();
41. }
42. }

输出：command received!

这个很哈理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

其实每个设计模式都是很重要的一种思想，看上去很熟，其实是因为我们在学到的东西中都有涉及，尽管有时我们并不知道，其实在Java本身的设计之中处处都 有体现，像AWT、JDBC、集合类、IO管道或者是Web框架，里面设计模式无处不在。因为我们篇幅有限，很难讲每一个设计模式都讲的很详细，不过我会 尽我所能，尽量在有限的空间和篇幅内，把意思写清楚了，更好让大家明白。本章不出意外的话，应该是设计模式最后一讲了，首先还是上一下上篇开头的那个图：

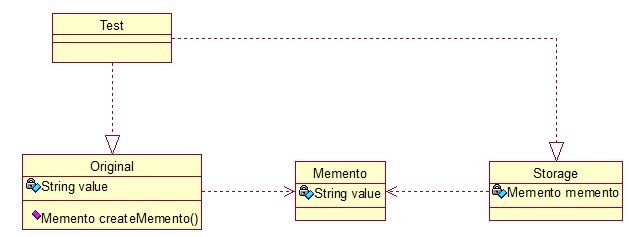


本章讲讲第三类和第四类。

**类的状态**

**19、备忘录模式（Memento）**

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需 要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下：



Original类是原始类，里面有需要保存的属性value及创建一个备忘录类，用来保存value值。Memento类是备忘录类，Storage类是存储备忘录的类，持有Memento类的实例，该模式很好理解。直接看源码：

1. **public** **class** Original {
3. **private** String value;
5. **public** String getValue() {
6. **return** value;
7. }
9. **public** **void** setValue(String value) {
10. **this**.value = value;
11. }
13. **public** Original(String value) {
14. **this**.value = value;
15. }
17. **public** Memento createMemento(){
18. **return** **new** Memento(value);
19. }
21. **public** **void** restoreMemento(Memento memento){
22. **this**.value = memento.getValue();
23. }
24. }
25. **public** **class** Memento {
27. **private** String value;
29. **public** Memento(String value) {
30. **this**.value = value;
31. }
33. **public** String getValue() {
34. **return** value;
35. }
37. **public** **void** setValue(String value) {
38. **this**.value = value;
39. }
40. }
41. **public** **class** Storage {
43. **private** Memento memento;
45. **public** Storage(Memento memento) {
46. **this**.memento = memento;
47. }
49. **public** Memento getMemento() {
50. **return** memento;
51. }
53. **public** **void** setMemento(Memento memento) {
54. **this**.memento = memento;
55. }
56. }

测试类：

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. // 创建原始类
6. Original origi = **new** Original("egg");
8. // 创建备忘录
9. Storage storage = **new** Storage(origi.createMemento());
11. // 修改原始类的状态
12. System.out.println("初始化状态为：" + origi.getValue());
13. origi.setValue("niu");
14. System.out.println("修改后的状态为：" + origi.getValue());
16. // 回复原始类的状态
17. origi.restoreMemento(storage.getMemento());
18. System.out.println("恢复后的状态为：" + origi.getValue());
19. }
20. }

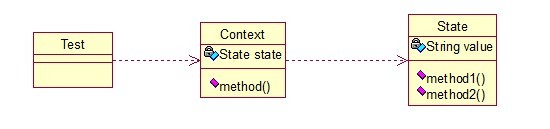
输出：

初始化状态为：egg  
修改后的状态为：niu  
恢复后的状态为：egg

简单描述下：新建原始类时，value被初始化为egg，后经过修改，将value的值置为niu，最后倒数第二行进行恢复状态，结果成功恢复了。其实我觉得这个模式叫“备份-恢复”模式最形象。

**20、状态模式（State）**

核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿QQ来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图：



State类是个状态类，Context类可以实现切换，我们来看看代码：

1. **package** com.xtfggef.dp.state;
3. /\*\*
4. \* 状态类的核心类
5. \* 2012-12-1
6. \* @author erqing
7. \*
8. \*/
9. **public** **class** State {
11. **private** String value;
13. **public** String getValue() {
14. **return** value;
15. }
17. **public** **void** setValue(String value) {
18. **this**.value = value;
19. }
21. **public** **void** method1(){
22. System.out.println("execute the first opt!");
23. }
25. **public** **void** method2(){
26. System.out.println("execute the second opt!");
27. }
28. }
29. **package** com.xtfggef.dp.state;
31. /\*\*
32. \* 状态模式的切换类   2012-12-1
33. \* @author erqing
34. \*
35. \*/
36. **public** **class** Context {
38. **private** State state;
40. **public** Context(State state) {
41. **this**.state = state;
42. }
44. **public** State getState() {
45. **return** state;
46. }
48. **public** **void** setState(State state) {
49. **this**.state = state;
50. }
52. **public** **void** method() {
53. **if** (state.getValue().equals("state1")) {
54. state.method1();
55. } **else** **if** (state.getValue().equals("state2")) {
56. state.method2();
57. }
58. }
59. }

测试类：

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. State state = **new** State();
6. Context context = **new** Context(state);
8. //设置第一种状态
9. state.setValue("state1");
10. context.method();
12. //设置第二种状态
13. state.setValue("state2");
14. context.method();
15. }
16. }

输出：

execute the first opt!  
execute the second opt!

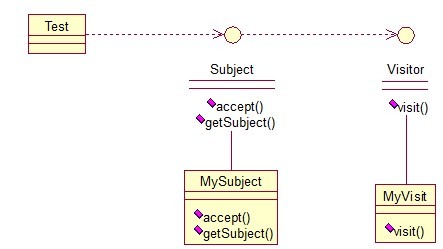
根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。

**通过中间类**

**21、访问者模式（Visitor）**

访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式 使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为 增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。—— From 百科

简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。简单关系图：



来看看原码：一个Visitor类，存放要访问的对象，

1. **public** **interface** Visitor {
2. **public** **void** visit(Subject sub);
3. }
4. **public** **class** MyVisitor **implements** Visitor {
6. @Override
7. **public** **void** visit(Subject sub) {
8. System.out.println("visit the subject："+sub.getSubject());
9. }
10. }

Subject类，accept方法，接受将要访问它的对象，getSubject()获取将要被访问的属性，

1. **public** **interface** Subject {
2. **public** **void** accept(Visitor visitor);
3. **public** String getSubject();
4. }
5. **public** **class** MySubject **implements** Subject {
7. @Override
8. **public** **void** accept(Visitor visitor) {
9. visitor.visit(**this**);
10. }
12. @Override
13. **public** String getSubject() {
14. **return** "love";
15. }
16. }

测试：

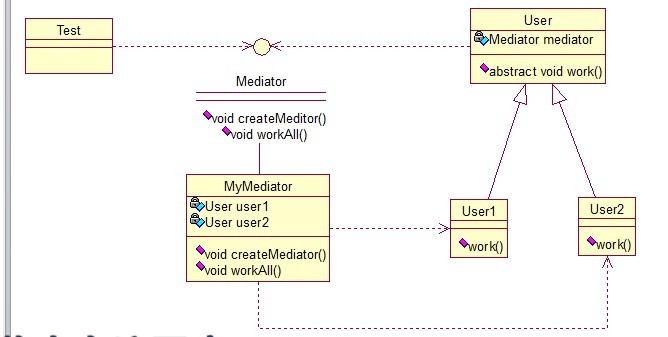
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. Visitor visitor = **new** MyVisitor();
6. Subject sub = **new** MySubject();
7. sub.accept(visitor);
8. }
9. }

输出：visit the subject：love

该模式适用场景：如果我们想为一个现有的类增加新功能，不得不考虑几个事情：1、新功能会不会与现有功能出现兼容性问题？2、以后会不会再需要添加？3、 如果类不允许修改代码怎么办？面对这些问题，最好的解决方法就是使用访问者模式，访问者模式适用于数据结构相对稳定的系统，把数据结构和算法解耦，

**22、中介者模式（Mediator）**

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。先 看看图：



User类统一接口，User1和User2分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高， 为了解耦，引入了Mediator类，提供统一接口，MyMediator为其实现类，里面持有User1和User2的实例，用来实现对User1和 User2的控制。这样User1和User2两个对象相互独立，他们只需要保持好和Mediator之间的关系就行，剩下的全由MyMediator类 来维护！基本实现：

1. **public** **interface** Mediator {
2. **public** **void** createMediator();
3. **public** **void** workAll();
4. }
5. **public** **class** MyMediator **implements** Mediator {
7. **private** User user1;
8. **private** User user2;
10. **public** User getUser1() {
11. **return** user1;
12. }
14. **public** User getUser2() {
15. **return** user2;
16. }
18. @Override
19. **public** **void** createMediator() {
20. user1 = **new** User1(**this**);
21. user2 = **new** User2(**this**);
22. }
24. @Override
25. **public** **void** workAll() {
26. user1.work();
27. user2.work();
28. }
29. }
30. **public** **abstract** **class** User {
32. **private** Mediator mediator;
34. **public** Mediator getMediator(){
35. **return** mediator;
36. }
38. **public** User(Mediator mediator) {
39. **this**.mediator = mediator;
40. }
42. **public** **abstract** **void** work();
43. }
44. **public** **class** User1 **extends** User {
46. **public** User1(Mediator mediator){
47. **super**(mediator);
48. }
50. @Override
51. **public** **void** work() {
52. System.out.println("user1 exe!");
53. }
54. }
55. **public** **class** User2 **extends** User {
57. **public** User2(Mediator mediator){
58. **super**(mediator);
59. }
61. @Override
62. **public** **void** work() {
63. System.out.println("user2 exe!");
64. }
65. }

测试类：

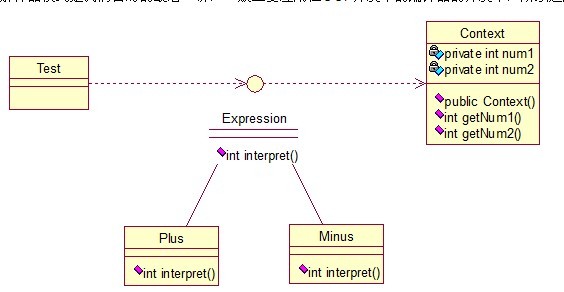
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Mediator mediator = **new** MyMediator();
5. mediator.createMediator();
6. mediator.workAll();
7. }
8. }

输出：

user1 exe!  
user2 exe!

**23、解释器模式（Interpreter）**

解释器模式是我们暂时的最后一讲，一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。



Context类是一个上下文环境类，Plus和Minus分别是用来计算的实现，代码如下：

1. **public** **interface** Expression {
2. **public** **int** interpret(Context context);
3. }
4. **public** **class** Plus **implements** Expression {
6. @Override
7. **public** **int** interpret(Context context) {
8. **return** context.getNum1()+context.getNum2();
9. }
10. }
11. **public** **class** Minus **implements** Expression {
13. @Override
14. **public** **int** interpret(Context context) {
15. **return** context.getNum1()-context.getNum2();
16. }
17. }
18. **public** **class** Context {
20. **private** **int** num1;
21. **private** **int** num2;
23. **public** Context(**int** num1, **int** num2) {
24. **this**.num1 = num1;
25. **this**.num2 = num2;
26. }
28. **public** **int** getNum1() {
29. **return** num1;
30. }
31. **public** **void** setNum1(**int** num1) {
32. **this**.num1 = num1;
33. }
34. **public** **int** getNum2() {
35. **return** num2;
36. }
37. **public** **void** setNum2(**int** num2) {
38. **this**.num2 = num2;
39. }

42. }
43. **public** **class** Test {
45. **public** **static** **void** main(String[] args) {
47. // 计算9+2-8的值
48. **int** result = **new** Minus().interpret((**new** Context(**new** Plus()
49. .interpret(**new** Context(9, 2)), 8)));
50. System.out.println(result);
51. }
52. }

最后输出正确的结果：3。

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！