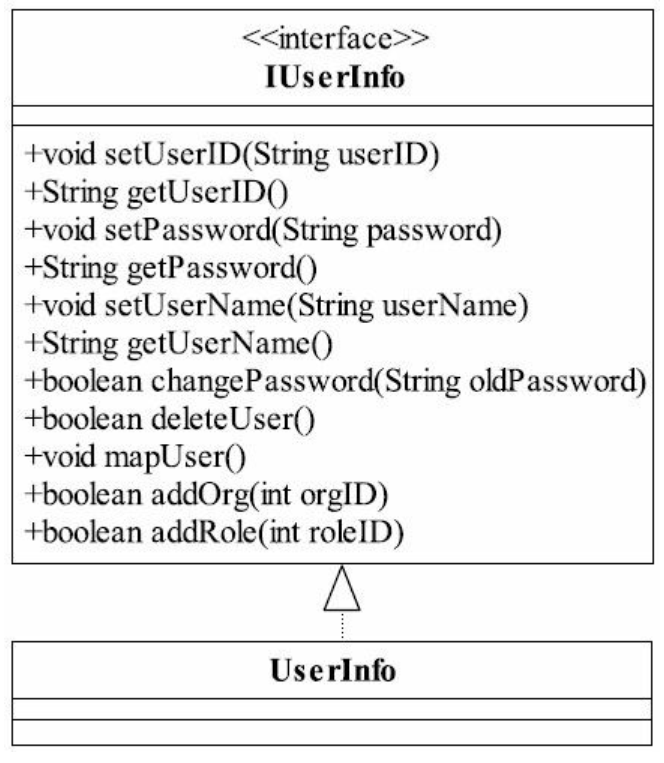
# 六大设计原则

## 单一职责原则

Single Responsibility Principle ，简称SRP**。就一个类而言，应该仅有一个引起它变化的原因。**当然对于职责的定义，什么是类的职责，以及怎么划分类的职责这几点在实际项目开发中很难确定。要灵活应对。

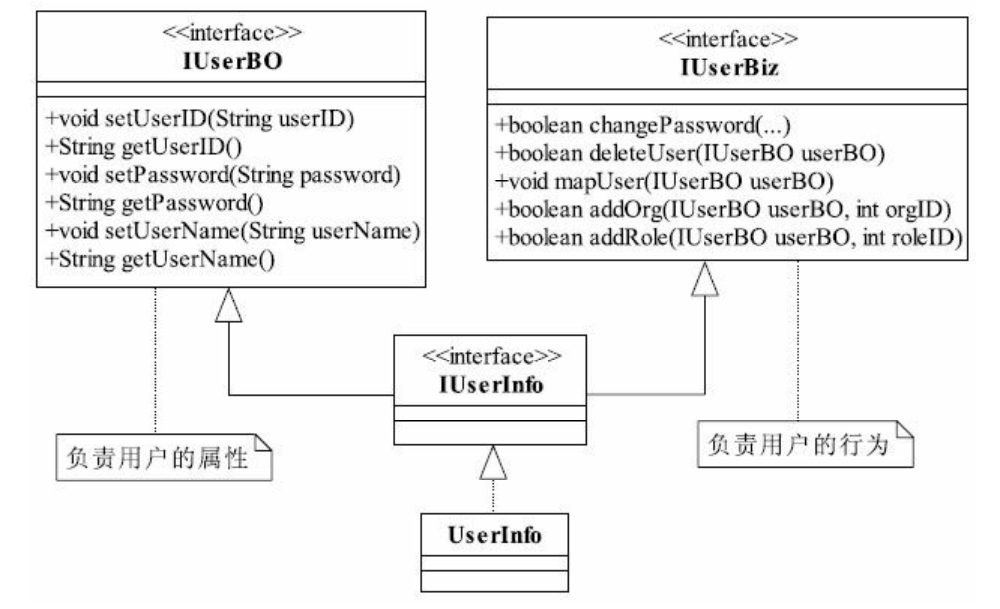
### 举例1：用户角色管理

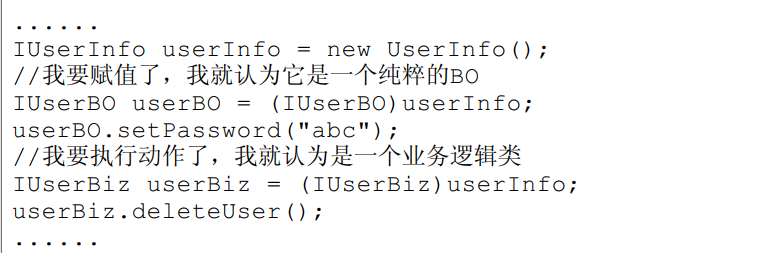


这个类设计的怎么样？

违反单一职责，**用户的属性和用户的行为没有分开，这是个严重的错误。**

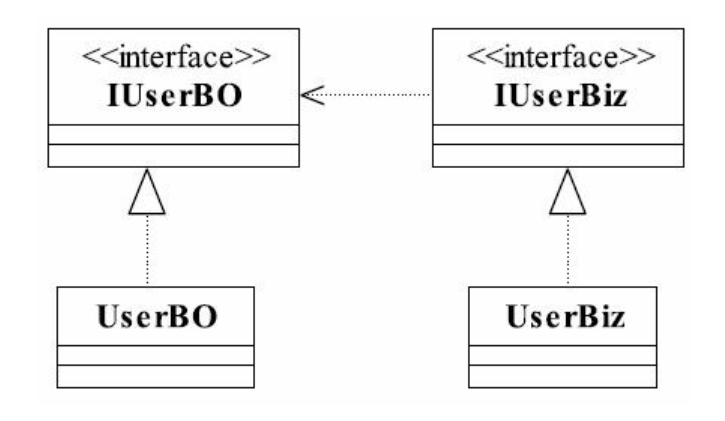
调整一下，把用户熟悉感信息抽成一个业务对象，把行为抽取成一个业务逻辑模块，修正后如下



是不是觉得有点别扭，最终的实现类还是拥有了“两类”不同的职责。再来看看使用的代码写法

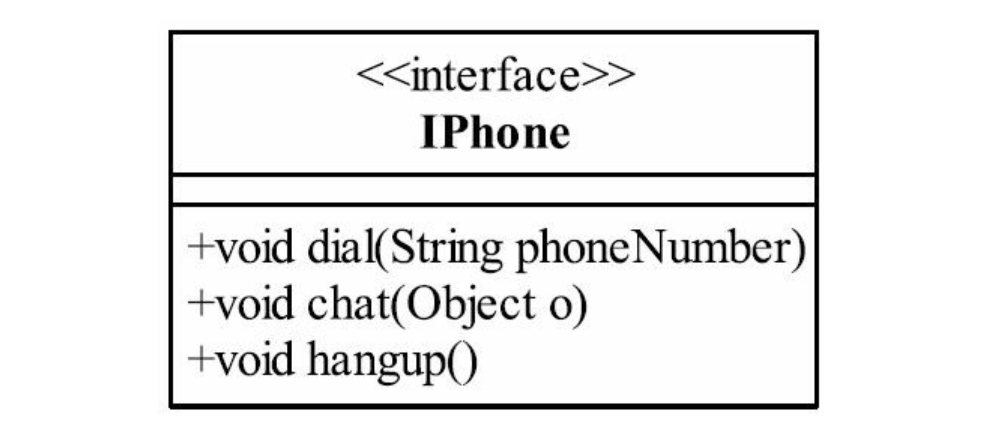
虽然接口实现确实职责分离了，但是实现类和功能写法都有问题。

**为什么要把一个接口拆分成2个，并且实现类又同时拥有了2个不同的职责？更好的处理方式是使用两个不同的类和接口。**

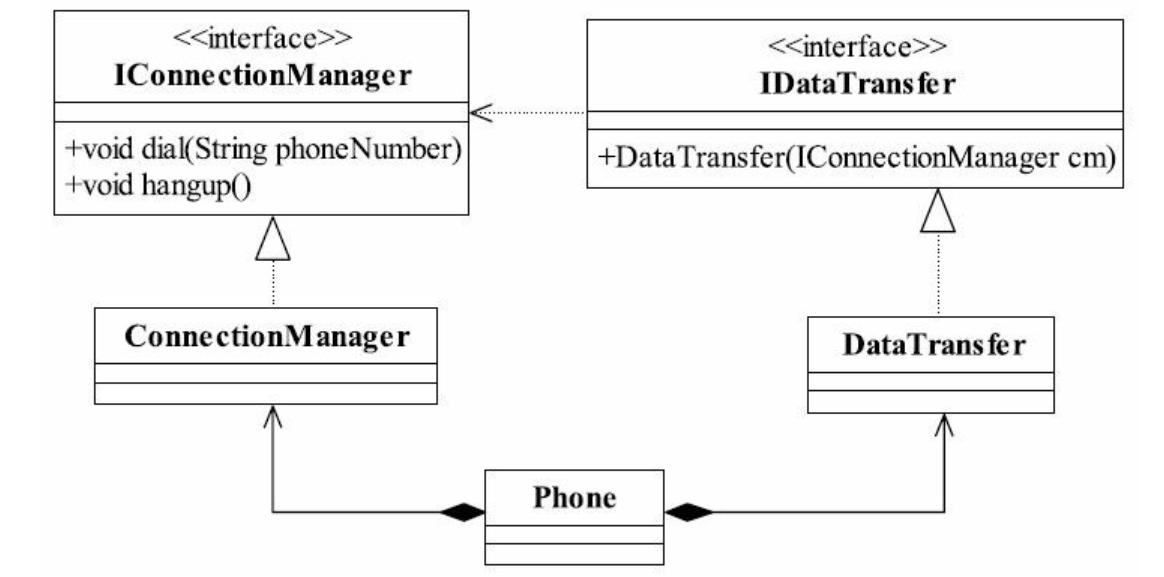


**注意行为和属性的依赖关系。**

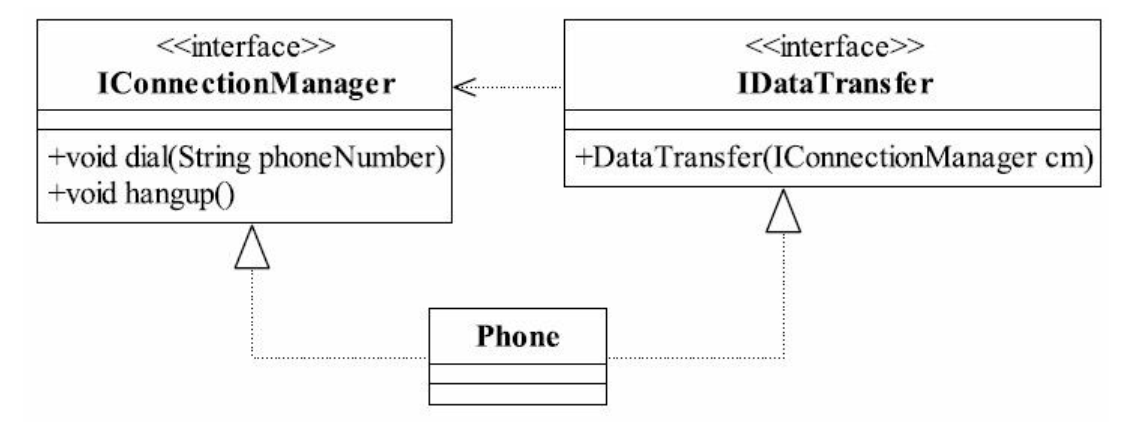
### 举例2：电话



有问题吗？从单一职责角度来讲的话，这个接口包含了拨打，挂断等的协议管理模块和通话过程中的数据传递模块。且协议模块不关心数据的内容（不关心是移动，联通还是电信）。拆分成2个接口：



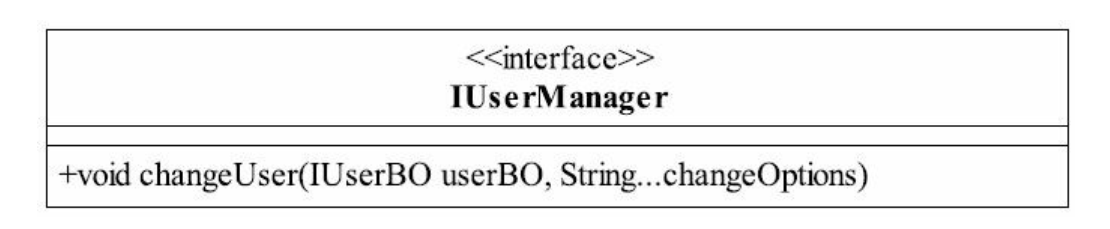
职责是明确了，但是增加了2个**实现类，并且这个两个实现类在phone中是组合到一起用的。组合是一种强耦合关系。**可以直接改成接口



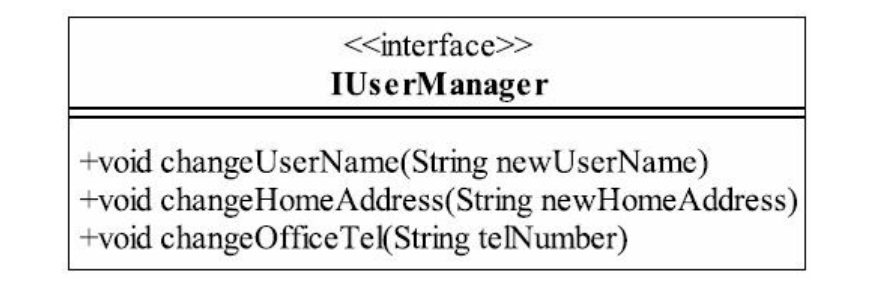
你可能会问Phone也是有两个原因一起变化呀。是的，**别忘了我们面向接口编程，对外公布的是接口而不是实现类。**如果真要实现类的单一职责，就必须用到上面的组合方式，类的数量会增加，并且耦合严重，**人为增加了设计的复杂性。**

**回到现实，单一职责提供了一个编写程序的标准，用“职责”，“变化原因”来衡量接口或类设计是否优良，但这个标准因环境和项目而异。所以最开始IPhone设计其实实际项目中并不需要修改。**

**单一职责适用于接口，类，同时也适用于方法。**一个方法尽可能做一件事情，比如修改用户密码方法，不要把这个方法放到修改用户信息方法中，这个方法颗粒度很粗。



方法职责不清晰，不单一，不要让别人猜测这个额方法可能是用来处理什么逻辑的。



### 总结：

对于单一职责，建议**接口，方法一定要做到单一职责，类的设计尽量做到只有一个原因引起变化**

类的单一职责的好处：  
1，复杂性降低，职责清晰明了

2，可读性提高，复杂性降低，可读性就提高了

3，可维护性提高，可读性降低，维护更容易

4，变更引起的风险降低

举例3：ImageLoader

1. */\*\**
2. \* 图片加载类
3. \*/
4. public class ImageLoader {
5. LruCache<String, Bitmap> mImageCache;
6. ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
7. Handler mUiHandler = new Handler(Looper.getMainLooper()) {
8. @Override
9. public void handleMessage(@NonNull Message msg) {
10. super.handleMessage(msg);
11. }
12. };
13. public ImageLoader() {
14. initImageCache();
15. }
16. private void initImageCache() {
17. *//j计算可使用的最大内存*
18. final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
19. final int cache = maxMemory / 4;
20. mImageCache = new LruCache<String, Bitmap>(cache) {
21. @Override
22. protected int sizeOf(String key, Bitmap value) {
23. return value.getRowBytes() \* value.getHeight() / 1024;
24. }
25. };
26. }
27. public void disPlayImage(final String url, final ImageView imageView) {
28. imageView.setTag(url);
29. mExecutorService.submit(new Runnable() {
30. @Override
31. public void run() {
32. Bitmap bitmap = downLoadImage(url);
33. if (bitmap == null) {
34. return;
35. }
36. if (imageView.getTag().equals(url)) {
37. updateImageView(imageView, bitmap);
38. }
39. mImageCache.put(url, bitmap);
40. }
41. });
42. }
43. private void updateImageView(final ImageView imageView, final Bitmap bmp) {
44. mUiHandler.post(new Runnable() {
45. @Override
46. public void run() {
47. imageView.setImageBitmap(bmp);
48. }
49. });
50. }
51. public Bitmap downLoadImage(String url) {
52. Bitmap bitmap = null;
53. try {
54. URL bitmapUrl = new URL(url);
55. HttpURLConnection connection = (HttpURLConnection) bitmapUrl.openConnection();
56. bitmap = BitmapFactory.decodeStream(connection.getInputStream());
57. connection.disconnect();
58. } catch (Exception e) {
59. e.printStackTrace();
60. }
61. return bitmap;
62. }
63. }

**问题：所有功能都放在一个类，极其不利于修改，扩展！**

1. */\*\**
2. \* 图片加载类
3. \*/
4. public class ImageLoader {
5. private ImageCache imageCache = new ImageCache();
6. private ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
7. private Handler mUiHandler = new Handler(Looper.getMainLooper()) {
8. @Override
9. public void handleMessage(@NonNull Message msg) {
10. super.handleMessage(msg);
11. }
12. };
13. public ImageLoader() {
14. }
15. public void disPlayImage(final String url, final ImageView imageView) {
16. Bitmap bitmap = imageCache.get(url);
17. if (bitmap != null) {
18. imageView.setImageBitmap(bitmap);
19. return;
20. }
21. imageView.setTag(url);
22. mExecutorService.submit(new Runnable() {
23. @Override
24. public void run() {
25. Bitmap bitmap = downLoadImage(url);
26. if (bitmap == null) {
27. return;
28. }
29. if (imageView.getTag().equals(url)) {
30. updateImageView(imageView, bitmap);
31. }
32. imageCache.put(url, bitmap);
33. }
34. });
35. }
36. private void updateImageView(final ImageView imageView, final Bitmap bmp) {
37. mUiHandler.post(new Runnable() {
38. @Override
39. public void run() {
40. imageView.setImageBitmap(bmp);
41. }
42. });
43. }
44. public Bitmap downLoadImage(String url) {
45. Bitmap bitmap = null;
46. try {
47. URL bitmapUrl = new URL(url);
48. HttpURLConnection connection = (HttpURLConnection) bitmapUrl.openConnection();
49. bitmap = BitmapFactory.decodeStream(connection.getInputStream());
50. connection.disconnect();
51. } catch (Exception e) {
52. e.printStackTrace();
53. }
54. return bitmap;
55. }
56. }
58. public class ImageCache {
59. private LruCache<String, Bitmap> mImageCache;
60. public ImageCache() {
61. initImageCache();
62. }
63. private void initImageCache() {
64. *//j计算可使用的最大内存*
65. final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
66. final int cache = maxMemory / 4;
67. mImageCache = new LruCache<String, Bitmap>(cache) {
68. @Override
69. protected int sizeOf(String key, Bitmap value) {
70. return value.getRowBytes() \* value.getHeight() / 1024;
71. }
72. };
73. }
74. public void put(String url, Bitmap bitmap) {
75. mImageCache.put(url, bitmap);
76. }
77. public Bitmap get(String url) {
78. return mImageCache.get(url);
79. }
80. }

V2：将缓存拆出来，单独成类

## 里氏替换原则

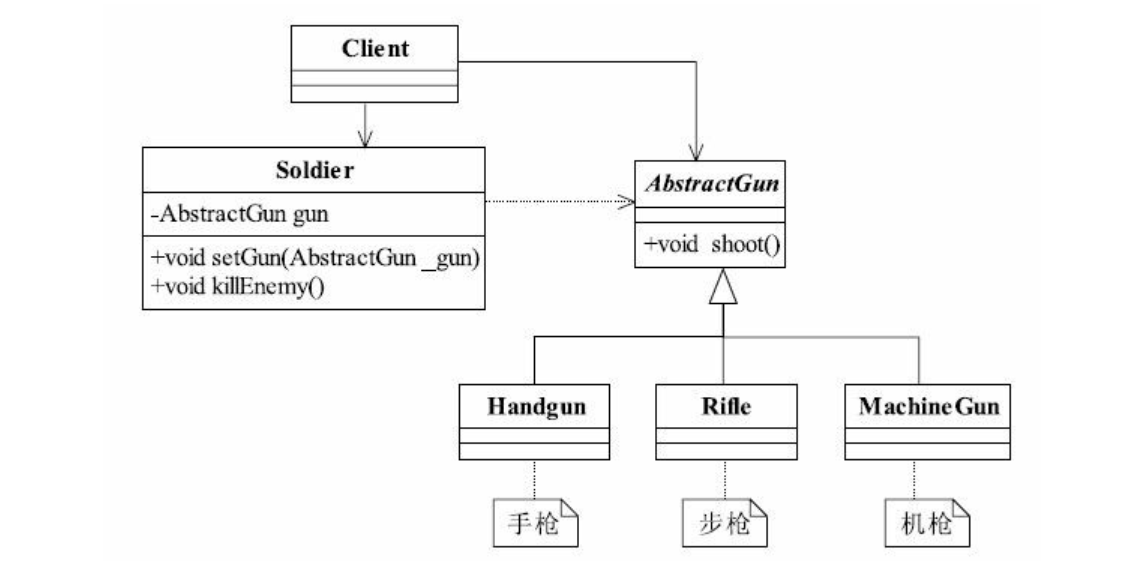
所有引用基类的地方必须能透明的使用其子类的对象。

里氏替换原则为良好的继承定义了一个规范，包含4曾含义：

### （1）子类必须完全实现父类的方法

系统设计时，经常会定义一个接口或抽象类，然后编码实现。其实这里使用了里氏替换。**在类中调用其他类时无比使用父类或接口，如果不能使用父类或接口，则说明设计已经违背了LSP**

举例：设计游戏中的枪



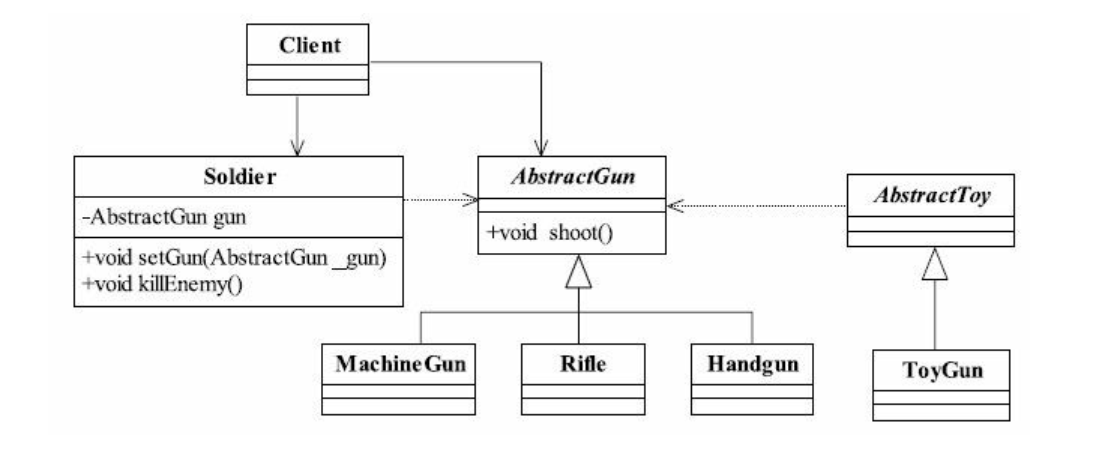
手枪，步枪，机枪都能实现射击操作。这几种枪

需求扩展：添加一个玩具枪

玩具枪不具备射击功能怎么办？有两种解决方法：第一，在Soldier类中增加instanceof的判断，如果时玩具枪则不能用来射击。

但是这种方法每增加一个类就要新增一下判断，非常不可取

第二，玩具枪脱离继承，创建一个独立的父类，为了代码复用，可以与真枪的抽象类建立委托关系



按照继承原则，玩具枪继承AbstractGun是绝对没有问题的，玩具枪也是枪嘛。但是具体应用场景中要考虑：**子类是否能够完整的实现父类的业务。如果不能，或者父类的某些方法在子类中已经发生“畸变”，则建议断开继承关系，采用依赖，聚集，组合灯关系代替继承。**

### （2）子类可以有自己的个性

一句话：子类能用的地方，父类不一定能用

### （3）覆盖或实现父类的方法时输入参数可以被放大

1. public class Father {
2. public Collection doSomeThing(HashMap map){
3. System.out.println("父类被执行");
4. return map.values();
5. }
6. }
7. public class Son extends Father {
8. *//放大输入类型*
9. public Collection doSomeThing(Map map){
10. System.out.println("子类被执行");
11. return map.values();
12. }
13. }
14. public class Test {
15. public static void main(String[] args) {
16. HashMap hashMap = new HashMap();
17. Father father = new Father();
18. father.doSomeThing(hashMap);
19. Son son = new Son();
20. son.doSomeThing(hashMap);
21. }
22. }



子类方法不能用override，会抱错。因为方法参数不一样。这其实时重载，因为子类继承了父类的方法。

**两次都时执行了父类的方法，这是对的！因为没有重载的情况下就应该执行父类的方法，这样才是里氏替换，良好的代码规范。**

上面的例子子类的方法参数范围扩大了父类的参数范围，如果反过来呢？

1. public class Father {
2. public Collection doSomeThing(Map map){
3. System.out.println("父类被执行");
4. return map.values();
5. }
6. }
7. public class Son extends Father {
8. *//缩小输入类型*
9. public Collection doSomeThing(HashMap map){
10. System.out.println("子类被执行");
11. return map.values();
12. }
13. }
14. public class Test {
15. public static void main(String[] args) {
16. HashMap hashMap = new HashMap();
17. Father father = new Father();
18. father.doSomeThing(hashMap);
19. Son son = new Son();
20. son.doSomeThing(hashMap);
21. }
22. }

这种情况下override也是会报错的。执行结果为：  


子类缩小了前置条件（方法参数范围），**子类在没有覆盖父类的前提下执行了子类的方法，这会引起业务逻辑混乱，这样的代码设计歪曲了父类的意图，不符合代码设计规范。所以子类中方法的前置条件必须与超类被覆写的方法的前置条件相同或者更宽松**

### （4）覆写或实现父类的方法时输出结果可以相同或者被缩小

### 总结：

采用里氏替换时，尽量避免子类的“个性”，一旦子类有“个性”，这个子类或父类的关系就难调和。把子类当作父类用，子类的个性被抹杀，失去了意义。把子类当作一个业务来使用，则会让代码间的耦合关系变得扑簌迷离，缺乏类替换的标准。

优点：

（1）代码宠用，减少创建类成本，每个子类自动拥有父类属性和方法

（2）子类和父类基本相似，又可以有所区别

（3）提高代码可扩展性

缺点：

（1）继承时侵入式的，只要继承就必须拥有父类属性和方法

（2）可能造成子类代码冗余，灵活性降低

（3）父类修改的时候，所有子类都要修改，成本大

## 依赖倒置原则

1，高层模块不应该依赖低层模块

2，抽象不应该依赖细节

3，细节应该依赖抽象

在Java中，抽象就是接口或抽象类；细节就是实现类。依赖倒置原则在Java语言中表现为

1，模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系时通过接口或抽象类产生的

2，接口或抽象类不依赖于实现类

3，实现类依赖接口或抽象类

**更精简的第一就是面向对象的精髓之一：面向接口编程**

举例：Imageloder 库缓存应该使用接口，而不是直接写实现类，否则想替换缓存类型的时候都要涉及到imageloder

## 接口隔离原则（ISP）

客户端不应该依赖它不需要的接口。另一种定义：类间的依赖关系应该建立在最小的接口上。

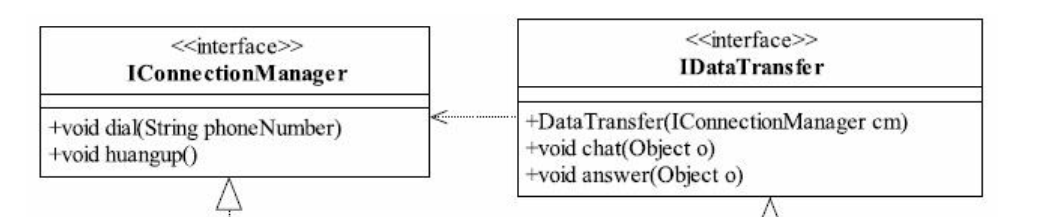
**通过分散定义多个接口，可以预防未来变更的扩散，提供系统的灵活性和可维护性**

接口隔离原色对接口进行规范约束，包含4层含义：

### 1，接口要尽量小

**一个接口只服务于一个子模块或业务逻辑**

不出现臃肿的接口，但是小是有限度的，首**先就是不违反单一职责原则。**



挂断电话接口如果要符合尽量小的原则的话，可以分为正常挂断，异常挂断。如果要这样就得把Iconnection拆分成拨打和挂断两个接口。再细分还可以对业务和协议拆分，是否要考虑3G协议，中继服务器等。拆还是不拆呢？答案是不拆，因为业务逻辑来讲，通信的建立和关闭已经是最小的业务单位了，如果拆分了就不符合单一职责原则了！

### 2，接口要高内聚

提高接口，类，模块的处理能力，减少对外的交互。在接口中尽量少公布public方法，接口时对外的承诺，承诺越少对系统的开发越有利，变更风险越小。

### 3，定制服务

系统内的模块必然存在交互，耦合。之间可能是通过接口调用或者单纯的数据交换。针对不同的调用方要针对性服务，绝不能为了省事暴露给他们不需要的接口。

### 4，接口设计时有限度的

越小越灵活，但是越小也带来了结构的复杂化。所以度要斟酌，要结合实际业务，环境做不同标准

举例：inputstream，outputstream的关闭

1. public class CloseUtil {
2. public static void closeQuietly(Closeable closeable){
3. if(null!=closeable){
4. try {
5. closeable.close();
6. } catch (IOException e) {
7. e.printStackTrace();
8. }
9. }
10. }
11. }

关闭的时候只需要确定支持close接口的就行，不用关心write，read等方法。这也是某种意义上的接口隔离

## 迪米特法则

也称最小知识原则。一个对象应该对其他对象有最少的了解。通俗讲：一个类应该对自己需要耦合或者调用的类知道的最少。包含4层含义：

### 1，只和朋友交流

**只与直接的朋友通信！**

举例：老师要求清点女生

1. public class GroupLeader {
2. *//  清点数量*
3. public void countGirls(List<Girl> girlList){
4. System.out.println("女神数量为："+girlList.size());
5. }
6. }
7. public class Teacher {
8. *//老师发布清点v女生命令*
9. public void commond(GroupLeader leader){
10. List<Girl> girls = new ArrayList<>();
11. for(int i=0;i<20;i++){
12. girls.add(new Girl());
13. }
14. leader.countGirls(girls);
15. }
16. }
17. public class Test {
18. public static void main(String[] args) {
19. Teacher teacher = new Teacher();
20. GroupLeader leader = new GroupLeader();
21. teacher.commond(leader);
22. }
23. }

看出来有什么问题吗？Teacher类有几个朋友类，一个：GroupLeader。为什么girls不是呢，Tearcher也对它产生了依赖。**出现在成员变量，方法输入输出参数的类称为成员朋友，而出现在方法体内部的类不是朋友类。**方法是类的一个行为，类竟然不知道自己的类和其他类产生了依赖关系，这是不允许的，严重违反了迪米特法则。

1. public class GroupLeader {
2. private List<Girl> girls;
3. public GroupLeader(List<Girl> girlList) {
4. girls = girlList;
5. }
6. *//  清点数量*
7. public void countGirls() {
8. System.out.println("女神数量为：" + girls.size());
9. }
10. }
11. public class Teacher {
12. *//老师发布清点v女生命令*
13. public void commond(GroupLeader leader){
14. leader.countGirls();
15. }
16. }
17. public class Test {
18. public static void main(String[] args) {
19. Teacher teacher = new Teacher();
20. List<Girl> girls = new ArrayList<>();
21. for (int i = 0; i < 20; i++) {
22. girls.add(new Girl());
23. }
24. GroupLeader leader = new GroupLeader(girls);
25. teacher.commond(leader);
26. }
27. }

将Teacher中对List的初始化移到场景类中，同时在GroupLeader中增加了对Girl的注入，避开了Teacher类对list girl的耦合，提供了系统的健壮性

### 2，朋友之间也是有距离的

刺猬取暖，太远取不到暖，太近就相互刺伤。

1. public class Wizard {
2. private Random rand = new Random(System.currentTimeMillis());
3. public int first() {
4. System.out.println("执行第一步");
5. return rand.nextInt(100);
6. }
7. public int second() {
8. System.out.println("执行第二步");
9. return rand.nextInt(100);
10. }
11. public int third() {
12. System.out.println("执行第三步");
13. return rand.nextInt(100);
14. }
15. }
16. public class InstallSoftware {
17. public void installWizard(Wizard wizard) {
18. int first = wizard.first();
19. if (first > 50) {
20. int second = wizard.second();
21. if (second > 50) {
22. wizard.third();
23. }
24. }
25. }
26. }
27. public class Test {
28. public static void main(String[] args) {
29. InstallSoftware invoke = new InstallSoftware();
30. invoke.installWizard(new Wizard());
31. }
32. }

有什么问题？

Wizard把太多方法暴露给InstallSoftware类了，如果Wizard中间安装的逻辑改了，那么InstallSoftware也得跟着改。

修改方法：把public方法改为private，具体的安装细节交给Wizard

1. public class Wizard {
2. private Random rand = new Random(System.currentTimeMillis());
3. private int first() {
4. System.out.println("执行第一步");
5. return rand.nextInt(100);
6. }
7. private int second() {
8. System.out.println("执行第二步");
9. return rand.nextInt(100);
10. }
11. private int third() {
12. System.out.println("执行第三步");
13. return rand.nextInt(100);
14. }
15. public void installWizard() {
16. int first = first();
17. if (first > 50) {
18. int second = second();
19. if (second > 50) {
20. third();
21. }
22. }
23. }
24. }
25. public class InstallSoftware {
26. public void installWizard(Wizard wizard) {
27. wizard.installWizard();
28. }
29. }

**迪米特法则要求尽量不要对外公布太多的public方法和静态public变量，尽量内敛，多使用private，protected，final关键字。**

### 3，是自己的就是自己的

如何决定方法放在这个类还是哪个类？**如果一个方法放在本类中，既不增加类间关系，也对本类不产生负面影响，那就放在本类中。**

### 4，谨慎使用Serializable

Private 改为public 容易导致客户端和服务端反序列化异常

### 总结

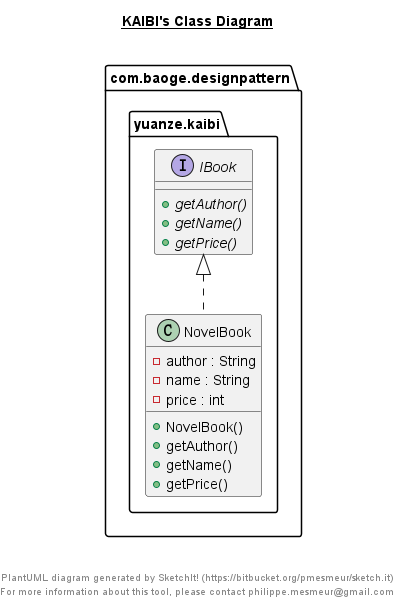
迪米特法则核心观念就是类间解耦，弱耦合，只有弱耦合类的复用率才可以提高。其要求的结果就是产生了大量的中转和跳转类，导致复杂性提高，维护困难。所以要反复权衡，及做到结构清晰，又做到高内聚低耦合。纪要减少暴露又要避免“路径”过长。

## 开闭原则

**对扩展开放，对修改关闭！**

**面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。**

举例：买书



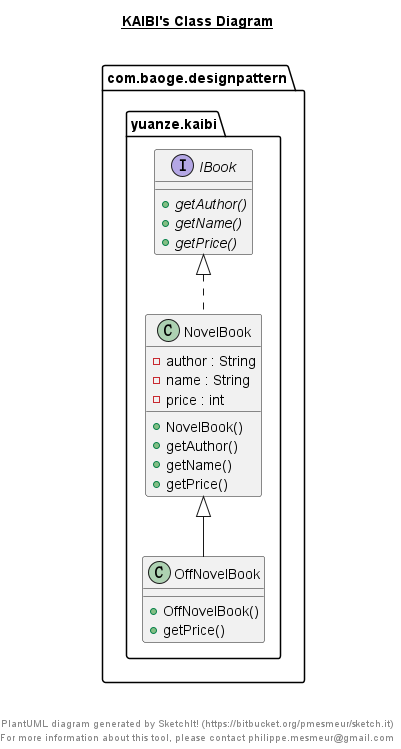
现在发生需求变更，大于50的书打8折，其他打9折

方法一：在IBook接口上增加方法getOffPrice()。这种方案的后果是所有实现类都要变，调用方也得改。同时IBook作为接口应该是稳定且可靠的，不应该经常变动，否则就失去了**契约的效能**

方法二：修改实现类，在实现类getprice方法中直接做价格打折逻辑。这是常见的方法，某些情况下这种方案是不错的方法，但是这里某些管理员是需要知道原价的，所以不能直接改

方案三：**通过扩展实现变化**

增加OffNovelBook打折类继承NovelBook，然后覆盖getprice方法，不同的角色调用不同的类



### 为什么要采用开闭原则

**1，开闭原则对测试的影响**

保持之前的代码不动，保证代码质量

**2，开闭原则可以提高复用性**

**3，可以提高可维护性**

**4，面向对象开发的要求**

### 如何使用开闭原则

**1，抽象约束**

通过接口或抽象类可以约束一组可能变化的行为，并且能够实现对扩展开放。包含3层含义：

（1）通过接口或者抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法；

（2）类型参数，引用对象尽量使用接口或抽象类，而不是实现类；

（3）抽象层尽量保持稳定，一旦确定即不允许修改

**2，元数据控制模块行为**

元数据：用来描述环境和数据的数据。通俗的说就是配置参数，通过参数的修改实现业务的调整

**3，执行项目章程**

规则统一，风格统一

**4，封装变化**

将相同的变化封装到一个接口或抽象类中；将不同的变化封装到不同的接口或抽象类中

# 设计模式

单例模式：某个类只能有一个实例，提供一个全局的访问点。

简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

原型模式：通过复制现有的实例来创建新的实例。

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

组合模式：将对象组合成树形结构以表示“”部分-整体“”的层次结构。

装饰模式：动态的给对象添加新的功能。

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

观察者模式：对象间的一对多的依赖关系。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

中介者模式：用一个中介对象来封装一系列的对象交互。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

## 创建型设计模式

单例模式，原型模式，工厂方法模式，抽象工厂模式，建造者模式

### 单例模式

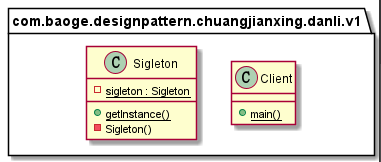
#### 定义：

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。

#### 使用场景

要求只有一个对象的场景

#### UML类图



#### V1：恶汉模式

1. public class Sigleton {
2. private static Sigleton instance = new Sigleton();
3. private Sigleton() {
4. }
5. public static Sigleton getInstance() {
6. return instance;
7. }
8. }

优点：在类加载的时候就实例化了对象，类加载只会有一次，所以线程安全

缺点：没有用的时候就实例化了，浪费。

#### 懒汉模式

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

优点：使用的时候才会创建  
缺点：线程不安全

#### V2，加锁

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

getInstance方法上加锁，可以保证单例安全性。  
优点：第一次使用的时候才会初始化。

缺点：每次调用这个方法都会进行同步开销，造成不必要的开销。

#### 再优化->Double Check Lock (DCL) 双重检查锁定-双重锁懒汉模式

1. public class Sigleton {
2. private static Sigleton instance;
3. private Sigleton(){}
4. public static  Sigleton getInstance(){
5. if(instance == null){
6. synchronized (Sigleton.class) {
7. if(instance == null) {
8. instance = new Sigleton();
9. }
10. }
11. }
12. return instance;
13. }
14. }

进行了两次判空，**第一次判空避免了不必要的同步，第二次判断则是为了在null情况下创建实例。**这就万无一失了吗？

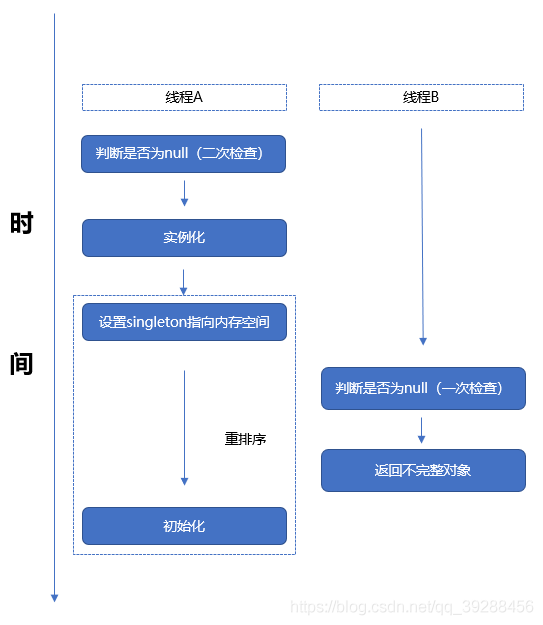
Instance = new Sigleton();最终会被编译成多条汇编指令，大概做3件事情：

1. 给Sigleton的实例分配内存；
2. 调用Sigleton构造函数
3. 将instance对象指向分配的内存空间（此时instance就不是null了）

对象初始化初略分为2个部分：实例化和初始化。实例化指系统为对象分配内存；初始化指实例化后的赋值操作。

**对象的初始化不是一个原子操作，复合操作，有发生重排序的可能。同步块能够保证多个线程有序的（即同步）执行同步块中的代码，但并不能避免块中代码发生重排序。**

上面执行环节有可能是 1-2-3 ，也有可能是1-3-2。所以当A线程按照1-3-2执行到第二次检查步骤3的时候，线程B执行第一次检查的时候发现不为null，返回。但是这个对象地址不为空，**但是没有经过构造函数，是一个不完整的对象。**



Volatile变量是java语言提供的一种稍弱的同步机制。有两个重要的特点：不会被重排序；不会被缓存到工作线程的本地内存。基于这两点，变量增加volatile关键字，就能避免DCL失效问题

1. public class Sigleton {
2. private volatile static Sigleton instance;
3. private Sigleton(){}
4. public static Sigleton getInstance(){
5. if(instance == null){
6. synchronized (Sigleton.class) {
7. if(instance == null) {
8. instance = new Sigleton();
9. }
10. }
11. }
12. return instance;
13. }
14. }

优点：资源利用率高，第一次执行才会初始化

缺点：第一次加载反应稍慢，volatile关键字稍微影响一些性能

#### 静态内部类

1. public class Sigleton {
2. private Sigleton() {
3. }
4. public static Sigleton getInstance() {
5. return SingleonHolder.instance;
6. }
7. private static class SingleonHolder {
8. private static final Sigleton instance = new Sigleton();
9. }
10. }

**静态内部类避讳随着外部类的加载而加载。所以：**

第一次加载Singleton类时并不会初始化instance，只有调用getInstance方法虚拟机才会加载SingleonHolder 类，执行构造函数初始化。线程安全&实例化延迟，推荐。

#### 枚举类型

1. public enum SingletonEnum {
2. INSTANCE;
3. public void doSomting(){}
4. }
5. *//调用方式*
6. SingletonEnum.INSTANCE.doSomting();

默认枚举实例的创建是线程安全的，并且任何情况下都是一个单例。实际上：  
**枚举类隐藏了构造器**

https://blog.csdn.net/ningmengshuxiawo/article/details/115594979

<https://blog.csdn.net/qq_43778308/article/details/110873569>

枚举：Y隐藏构造，怎么防止放射，克隆，序列化？

#### 破坏单例模式的三种方式

1，反射

解决方案：定义一个全局变量，当第二次创建时抛出异常

2，克隆

重写clone，直接返回单例对象

3，序列化

添加readRFesolve()，返回object对象

#### 枚举单例是怎么解决的？

**枚举单例不仅能避免多线程问题，也可以防止放序列化和反射的破坏。**

通过反编译枚举单例，发现  
1，枚举中各个枚举项都是通过static代码块来定义和初始化的，他们会在类加载时完成初始化，所以线程安全

2，防止反序列化的破坏

Java的序列化对枚举做了规定，序列化时，只是将枚举对象的name属性输出到结果中；反序列化时通过java.lang.Enum的valueOf方法根据名字查找对象，而不是新建一个新的对象，所以防止了序列化对单例的破坏。

1. private Enum<?> readEnum(boolean unshared) throws IOException {
2. if (this.bin.readByte() != 126) {
3. throw new InternalError();
4. } else {
5. ObjectStreamClass desc = this.readClassDesc(false);
6. if (!desc.isEnum()) {
7. throw new InvalidClassException("non-enum class: " + desc);
8. } else {
9. int enumHandle = this.handles.assign(unshared ? unsharedMarker : null);
10. ClassNotFoundException resolveEx = desc.getResolveException();
11. if (resolveEx != null) {
12. this.handles.markException(enumHandle, resolveEx);
13. }
14. String name = this.readString(false);
15. Enum<?> result = null;
16. Class<?> cl = desc.forClass();
17. if (cl != null) {
18. try {
19. Enum<?> en = Enum.valueOf(cl, name);
20. result = en;
21. } catch (IllegalArgumentException var9) {
22. throw (IOException)(new InvalidObjectException("enum constant " + name + " does not exist in " + cl)).initCause(var9);
23. }
24. if (!unshared) {
25. this.handles.setObject(enumHandle, result);
26. }
27. }
28. this.handles.finish(enumHandle);
29. this.passHandle = enumHandle;
30. return result;
31. }
32. }
33. }

3，防止反射的破坏

反射在通过newInstance创建对象时会检查这个类是否是枚举类，如果是枚举类，就会抛出异常：

1. public T newInstance(Object... initargs) throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {
2. if (!this.override) {
3. Class<?> caller = Reflection.getCallerClass();
4. this.checkAccess(caller, this.clazz, this.clazz, this.modifiers);
5. }
6. if ((this.clazz.getModifiers() & 16384) != 0) {
7. throw new IllegalArgumentException("Cannot reflectively create enum objects");
8. } else {
9. ConstructorAccessor ca = this.constructorAccessor;
10. if (ca == null) {
11. ca = this.acquireConstructorAccessor();
12. }
13. T inst = ca.newInstance(initargs);
14. return inst;
15. }
16. }

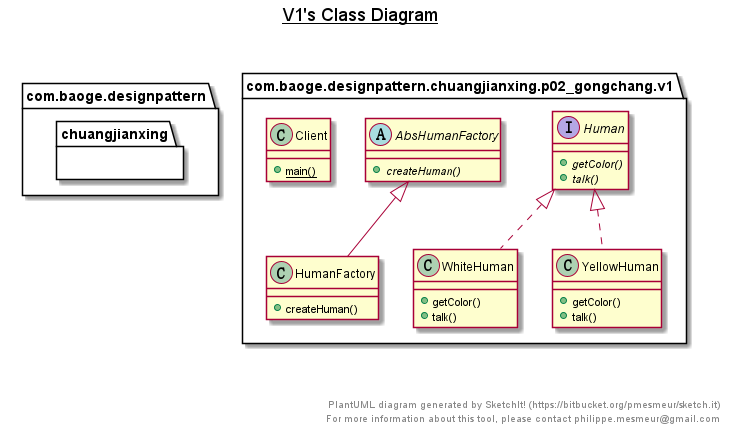
### 工厂方法模式

#### 定义

定义一个用于创建对象的接口，让子类决定实例化哪个类。工厂方法时一个类的实例化延迟到其子类。

#### UML类图

1. public interface Human {
2. public void getColor();
3. public void talk();
4. }
5. public class WhiteHuman implements Human {
6. @Override
7. public void getColor() {
8. System.out.println("白色");
9. }
10. @Override
11. public void talk() {
12. System.out.println("白说");
13. }
14. }
15. public class YellowHuman implements Human {
16. @Override
17. public void getColor() {
18. System.out.println("黄色");
19. }
20. @Override
21. public void talk() {
22. System.out.println("黄说");
23. }
24. }
25. public abstract class AbsHumanFactory {
26. public abstract <T extends Human> T createHuman(Class<T> tClass);
27. }
28. public class HumanFactory extends AbsHumanFactory {
29. @Override
30. public <T extends Human> T createHuman(Class<T> tClass) {
31. Human human = null;
32. try {
33. human = (T) Class.forName(tClass.getName()).newInstance();
34. } catch (Exception e) {
35. e.printStackTrace();
36. }
37. return (T) human;
38. }
39. }
40. public class Client {
41. public static void main(String[] args) {
42. AbsHumanFactory factory = new HumanFactory();
43. Human yellowHuman = factory.createHuman(YellowHuman.class);
44. yellowHuman.getColor();
45. yellowHuman.talk();
46. WhiteHuman whiteHuman = factory.createHuman(WhiteHuman.class);
47. whiteHuman.getColor();
48. whiteHuman.talk();
49. }
50. }



#### 工厂方法扩展

##### 缩小为简单工厂模式（静态方法模式）

1. public class HumanFactory   {
2. public static <T extends Human> T createHuman(Class<T> tClass) {
3. Human human = null;
4. try {
5. human = (T) Class.forName(tClass.getName()).newInstance();
6. } catch (Exception e) {
7. e.printStackTrace();
8. }
9. return (T) human;
10. }
11. }
12. public class Client {
13. public static void main(String[] args) {
14. Human yellowHuman = HumanFactory.createHuman(YellowHuman.class);
15. yellowHuman.getColor();
16. yellowHuman.talk();
17. WhiteHuman whiteHuman = HumanFactory.createHuman(WhiteHuman.class);
18. whiteHuman.getColor();
19. whiteHuman.talk();
20. }
21. }

HumanFactory类有个地方发生变化：去掉抽象类，并将创建方法改为static关键字；调用房直接调用静态方法即可。

##### 升级多个工厂类

1. public abstract class AbsHumanFactory {
2. public abstract Human createHuman();
3. }
4. public class WhiteHumanFactory extends AbsHumanFactory {
5. @Override
6. public Human createHuman() {
7. Human human = new WhiteHuman();
8. return  human;
9. }
10. }
11. public class YellowHumanFactory extends AbsHumanFactory {
12. @Override
13. public Human createHuman() {
14. Human human = new YellowHuman();
15. return  human;
16. }
17. }
18. public class Client {
19. public static void main(String[] args) {
20. Human yellowHuman = new YellowHumanFactory().createHuman();
21. yellowHuman.getColor();
22. yellowHuman.talk();
23. Human whiteHuman = new WhiteHumanFactory().createHuman();
24. whiteHuman.getColor();
25. whiteHuman.talk();
26. }
27. }

变化点：AbsHumanFactory 抽象类不在是泛型；增加了工厂模式的实现类；调用方式改变，不再需要传入类参数。

#### 使用场景

工厂方法是new一个对象的替代，所以在需要生成对象的地方都可以使用，但是要慎重考虑是否要增加一个工厂类来进行管理，增加代码复杂度。**一般复杂对象适合使用工厂模式，用new就可以完成创建的对象无需使用工厂模式**

map的迭代器，Activity的oncreate等生命周期回调。

### 抽象工厂模式

#### 定义

为创建一组相关或者相互依赖的对象提供一个接口，而不需要指定他们的具体类

#### UML图

1. public interface IBrake {
2. void brake();*//制动系统*
3. }
4. public class NormalBrake implements IBrake {
5. @Override
6. public void brake() {
7. System.out.println("普通制动");
8. }
9. }
10. public class SeniorBrake implements IBrake {
11. @Override
12. public void brake() {
13. System.out.println("高级制动");
14. }
15. }
16. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
17. public interface IEngine {
18. void engine();*//发动机*
19. }
20. public class DomesticEngine implements IEngine {
21. @Override
22. public void engine() {
23. System.out.println("国产发动机");
24. }
25. }
26. public class ImportEngine implements IEngine {
27. @Override
28. public void engine() {
29. System.out.println("进口发动机");
30. }
31. }
32. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
33. public interface ITire {
34. void tire();*//轮胎*
35. }
36. public class NormalTire implements ITire {
37. @Override
38. public void tire() {
39. System.out.println("普通轮胎");
40. }
41. }
42. public class SuvTire implements ITire {
43. @Override
44. public void tire() {
45. System.out.println("越野轮胎");
46. }
47. }
48. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
49. public abstract class CarFactory {
50. public abstract ITire createTire();
51. public abstract IEngine createEngine();
52. public abstract IBrake createBrake();
53. }
54. public class AudiQ3Factory extends CarFactory {
55. @Override
56. public ITire createTire() {
57. return new NormalTire();
58. }
59. @Override
60. public IEngine createEngine() {
61. return new DomesticEngine();
62. }
63. @Override
64. public IBrake createBrake() {
65. return new NormalBrake();
66. }
67. }
68. public class AudiQ7Factory extends CarFactory {
69. @Override
70. public ITire createTire() {
71. return new SuvTire();
72. }
73. @Override
74. public IEngine createEngine() {
75. return new ImportEngine();
76. }
77. @Override
78. public IBrake createBrake() {
79. return new SeniorBrake();
80. }
81. }
82. public class Client {
83. public static void main(String[] args) {
84. CarFactory audiQ3 = new AudiQ3Factory();
85. audiQ3.createBrake().brake();
86. audiQ3.createEngine().engine();
87. audiQ3.createTire().tire();
88. CarFactory audiQ7 = new AudiQ7Factory();
89. audiQ7.createBrake().brake();
90. audiQ7.createEngine().engine();
91. audiQ7.createTire().tire();
92. }
93. }

#### 优缺点

优点：分离接口和实现，调用方只是面向产品接口编程，从具体的产品实现中解耦，抽象该工厂方法模式在切换产品类时更加灵活容易

缺点：类文件爆炸性增加；不太容易扩展新的产品类，因为增加一个产品类就需要修改抽象工厂，那么所有具体工厂类均被修改

### 原型模式

#### 定义

用圆形实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象；

1，性能优良。原型模式是在内存二进制流的拷贝，大多数情况下要比new一个对象性能要好，特别是一个循环体内产生大量对象时，效果更明显

2，逃避构造函数约束；**内存中拷贝，不执行拷贝函数。**既是优点也是缺点。

#### 使用场景

1，资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据，硬件资源等

2，性能和安全要求的场景

通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式

3，一个对象多个修改者场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象共调用者使用

#### 注意事项

1，不会执行构造函数

2，浅拷贝。数组，对象都指向同一个地址

3，clone和final 冲突，会报错。

### 建造者（build）模式

#### 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**

Build模式是一步步创建要给复杂对象的创建型模式，它允许用户在**不知道内部构件细节**的情况下，可以更精细的控制对象的构造流程。该模式是为了将构建复杂对象的过程和他的部件解耦，是的构建过程和部件的表示隔离开来。**为了在构建过程中对外隐藏细节，就可以使用build模式将部件和组装过程分离，使得构建过程和部件都可以自由扩展，两者之间的耦合也降到最低。**

#### 使用场景

1，相同的方法，不同的执行顺序，产生不同的事件结果

2，多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同

3，产品类非常复杂，或者产品类中的调用顺序不同产生了不同的作用，这个时候使用建造者模式非常合适

4，当初始化一个对象特别复杂，如参数多，且很多参数都具有默认值时。

#### UML图

Product产品类：产品的抽象类

Builder ：抽象Build类，规范产品的组件，一般是由子类实现具体的组件过程；

ConcreateBuilder：具体的Build类；

Director：统一组装过程

1. */\*\**
2. \* 抽象产品类
3. \*/
4. public abstract class Computer {
5. protected String board;
6. protected String display;
7. protected String os;
8. protected Computer(){}
9. public String getBoard() {
10. return board;
11. }
12. public void setBoard(String board) {
13. this.board = board;
14. }
15. public String getDisplay() {
16. return display;
17. }
18. public void setDisplay(String display) {
19. this.display = display;
20. }
21. public abstract void setOs() ;
22. }
23. */\*\**
24. \* 具体产品类
25. \*/
26. public class MacBook extends Computer {
27. protected MacBook(){}
28. @Override
29. public void setOs( ) {
30. os = "Mac OS X 10.10";
31. }
32. }
33. public abstract class Builder {
34. public abstract void buildBoard(String board);
35. public abstract void buildDisplay(String display);
36. public abstract void buildOs();
37. public abstract Computer create();
38. }
39. public class MacBookBuilder extends Builder {
40. private Computer computer = new MacBook();
41. @Override
42. public void buildBoard(String board) {
43. computer.setBoard(board);
44. }
45. @Override
46. public void buildDisplay(String display) {
47. computer.setDisplay(display);
48. }
49. @Override
50. public void buildOs() {
51. computer.setOs();
52. }
53. @Override
54. public Computer create() {
55. return computer;
56. }
57. }
58. public class Director {
59. Builder builder = null;
60. public Director(Builder builder){
61. this.builder = builder;
62. }
63. public void construct(String board,String display){
64. builder.buildBoard(board);
65. builder.buildDisplay(display);
66. builder.buildOs();
67. }
68. }
69. public class Client {
70. public static void main(String[] args) {
71. Builder builder = new MacBookBuilder();
72. Director director = new Director(builder);
73. director.construct("英特尔主板","三星显示器");
74. builder.create();
75. }
76. }

**通过具体的MacBuilder（持有macbook对象）来构建Macbook对象。而Diretor封装了调用过程，隐藏了创建细节。Builder和Director一起将复杂对象的创建与它的表示分离，使得同样的构建过程可以创建不同的对象。**

实际开发中Director常被省略。而直接使用一个Builder来进行组装，Builder通常为链式调用，set返回自身。

Android典型用法就是AlertDialog

### 工厂方法VS建造者模式

工厂方法模式注重的是整体对象的创建方法，而建造者模式注重的是部件构建的过程。比如要制造一个超人，如果使用工厂方法模式，直接产生出来的就是一个力大无穷，能够飞翔，内裤外穿的超人；如果使用建造者模式，则需要组装手，头，脚，躯干等部分。

意图不同：

工厂方法模式关注的是一个产品整体，无须关心产品各个部分是如何创建出来的；但在建造者模式中，一个具体产品的产生是依赖各个部件的产生以及封装顺序。简单来说，工厂模式是一个对象创建的粗线条应用，建造者模式则是通过细线条勾勒出一个复杂对象，关注的是产品组成部分创建过程

产品复杂度不同：

工厂方法模式创建的产品一般都是单一性质产品，而建造者模式创建的则是一个复合产品，它由各个部件复合而成，部件不同产品对象不同。一般来说，工厂方法模式对象颗粒度比较粗，建造者模式颗粒度比较细。

如果需要详细关注一个产品的生产，安装步骤，则选择建造者模式，否则工厂方法模式。

## 行为型设计模式

### 模板方法

#### 定义

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

知道算法所需的关键步骤，并且确定了步骤的执行顺序，但是步骤的具体实现未知。

#### 使用场景

1，多个子类有公有的方法，并且逻辑基本相同

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

3，重构时把相同的代码抽取到父类中，通过钩子函数约束行为

#### UML图

1. public abstract class AbsComputer {
2. protected void powerOn() {
3. System.out.println("开启电源");
4. }
5. protected void checkHardware() {
6. System.out.println("硬盘检查");
7. }
8. protected void loadOs() {
9. System.out.println("载入操作系统");
10. }
11. protected void login() {
12. System.out.println("登录成功后，进入系统");
13. }
14. protected boolean isNeedLogin() {
15. return true;
16. }
17. public final void startUp() {
18. powerOn();
19. checkHardware();
20. loadOs();
21. if (isNeedLogin()) {
22. login();
23. }
24. }
25. }
26. public class CoderComputer extends AbsComputer {
27. @Override
28. protected void login() {
29. System.out.println("程序员登录，权限最大");
30. }
31. }
32. public class NormalComputer extends AbsComputer {
33. @Override
34. protected void checkHardware() {
35. System.out.println("不需要检测");
36. }
37. @Override
38. protected boolean isNeedLogin() {
39. return false;
40. }
41. }
42. public class Client {
43. public static void main(String[] args) {
44. AbsComputer corderComputer = new CoderComputer();
45. corderComputer.startUp();
46. AbsComputer normalComputer = new NormalComputer();
47. normalComputer.startUp();
48. }
49. }

1，抽象类中方法的限定符，被调用的时public，其他的时protect

2，钩子函数可以改变行为

3，被调用的public方法用final修饰，防止被子类覆写

Android源码中的模板方法AsyncTask。生命周期，

#### 小结

优点：

1，封装不变化部分，扩展可变部分

2，提取公共部分代码，便于维护

3，**行为由父类控制，子类实现**

缺点：  
模板方法会带来阅读的难度，难以理解

### 策略模式

#### 定义

策略模式定义了一序列的算法，并将每个算法封装起来，而且使他们可以相互替换。策略模式让算法独立于使用它的客户端而独立变化

软件开发经常遇到的情况：实现某一个功能可以有很多算法或者策略，我们根据实际情况选择不同的算法或者策略来完成该功能。例如，排序算法，可以使用插入排序，归并排序等。

针对这种情况，可以把多种算法都放到一个类中，一个方法对应一个算法。或者放到同一个方法，通过参数来判断使用哪一个算法。这两种方式都会导致类变得臃肿，维护成本高，并且当需要新增一个算法时，需要修改封装算法类的源代码。明显违反了OCP原则和单一职责原则。

可以将算法或者策略抽象出来，提供一个统一接口，不同的算法或者策略有不同的实现类，客户端调用的时候可以注入不同的实现对象来实现算法或者策略的动态切换。

#### 使用场景

1，针对同一类型的多种处理方式，仅仅时具体行为有差别时

2，需要安全地封装多种同一类型的操作时

3，出现同一抽象类有多个子类，而又需要使用if else来选择具体子类时

#### UML图

策略模式三个角色：  
1，Context封装角色：承上启下，屏蔽高层模块对策略，算法的直接访问，封装可能存在的变化。

2，Strategy抽象策略角色：策略，算法家族的抽象，通常为接口，定义每个策略或算法必须具有的方法和属性。

3，ConcreteStrategy具体策略角色：实现抽象策略中的操作，含有具体的算法。

1. */\*\**
2. \* 抽象策略角色
3. \*/
4. public interface CalculateStrategy {
5. */\*\**
6. \* 按举例来计算价格
7. \* @param km 公里
8. \* @return 价格
9. \*/
10. int calculatePrice(int km);
11. }
12. public class BusCalculateStrategy implements CalculateStrategy {
13. @Override
14. public int calculatePrice(int km) {
15. int price = 100;
16. *//省略公交车计算逻辑*
17. return price;
18. }
19. }
20. public class TaxiCalculateStrategy implements CalculateStrategy {
21. @Override
22. public int calculatePrice(int km) {
23. int price = 50;
24. *//省略出租车计算逻辑*
25. return price;
26. }
27. }
28. */\*\**
29. \* 公交方式价格计算类，扮演Context上下文封装类角色
30. \*/
31. public class TranficCalculator {
32. private CalculateStrategy strategy;
33. public void setStrategy(CalculateStrategy strategy){
34. this.strategy = strategy;
35. }
36. public int calculate(int km){
37. return strategy.calculatePrice(km);
38. }
39. }
40. public class Client {
41. public static void main(String[] args) {
42. TranficCalculator tranficCalculator = new TranficCalculator();
43. tranficCalculator.setStrategy(new BusCalculateStrategy());
44. tranficCalculator.calculate(10);
45. }
46. }

Android中的策略模式：动画的插值器

#### 小结

很好的演示了开闭原则，也就是定义抽象，注入不同的实现。从而达到很好的可扩展性。

**优点**

1，结构清晰明了，使用简单直观；

2，耦合度较低，扩展方便

3，操作封装彻底，数据安全

缺点：

1，算法增多，实现类也要跟着多

2，所有的策略都得暴露出去，并且要调用方自己选择

### 状态模式

#### 定义

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

状态模式中的行为是由状态决定的，不同的状态下有不同的行为。状态模式和策略模式的结构几乎完全一致，但他们的目的，本质却完全不一样。状态模式的意图是让一个对象在其内部状态改变的时候，行为随之改变

等会注意下面的例子，发现状态模式和策略模式都是可以通过方法设置实现类，**但是状态模式比策略模式增加了状态改变的时候实现类也跟着变化的。**

#### 使用场景

1，一个对象的行为与取决于它的状态，并且它必须在运行时更具状态改变它的行为

2，代码中包含大量与对象状态有关的条件语句

#### UML图

Context环境类：定义**客户**感兴趣的接口，维护一个State子类的实例，这个实例定义了对象的当前状态。

State抽象状态类或者状态接口，定义一个或者一组接口，表示该状态下的行为

ConcreteState 具体状态类，每一个具体的状态类实现抽象State中定义的接口，从而达到不同状态下的不同行为

电视遥控器类：

1. public class TvController01 {
2. private final static int POWER\_ON = 1;
3. private final static int POWER\_OFF = 2;
4. private int state = POWER\_OFF;
5. public void powerOn(){
6. state = POWER\_ON;
7. if(state == POWER\_OFF){
8. System.out.println("开机了");
9. }
10. }
11. public void powerOff(){
12. state = POWER\_OFF;
13. if(state == POWER\_ON){
14. System.out.println("关机了");
15. }
16. }
17. public void nextChannel(){
18. if(state == POWER\_ON){
19. System.out.println("下一个频道");
20. }else {
21. System.out.println("关机状态，无法控制");
22. }
23. }
24. public void turnUp(){
25. if(state == POWER\_ON){
26. System.out.println("调高音量");
27. }else {
28. System.out.println("关机状态，无法控制");
29. }
30. }
31. }

问题：

1，每个方法都要经过ifelse判断

2，这些代码充斥在一个类中，无法提取，难以维护

1. public interface TvState {
2. void nextChannel();
3. void turnUp();
4. }
5. public class PowerOnState implements TvState {
6. @Override
7. public void nextChannel() {
8. System.out.println("下一个频道");
9. }
10. @Override
11. public void turnUp() {
12. System.out.println("调高音量");
13. }
14. }
15. public class PowerOffState implements TvState {
16. @Override
17. public void nextChannel() {
18. System.out.println("关机状态，无法控制");
19. }
20. @Override
21. public void turnUp() {
22. System.out.println("关机状态，无法控制");
23. }
24. }
25. public interface PowerController {
26. void powerOn();
27. void powerOff();
28. }
29. public class TvController implements PowerController {
30. private TvState tvState;
31. public TvState getTvState() {
32. return tvState;
33. }
34. public void setTvState(TvState tvState) {
35. this.tvState = tvState;
36. }
37. @Override
38. public void powerOn() {
39. setTvState(new PowerOnState());
40. System.out.println("开机了");
41. }
42. @Override
43. public void powerOff() {
44. setTvState(new PowerOffState());
45. System.out.println("开机了");
46. }
47. public void nextChannel(){
48. tvState.nextChannel();
49. }
50. public void turnUp(){
51. tvState.turnUp();
52. }
53. }
54. public class Client {
55. public static void main(String[] args) {
56. TvController tvController = new TvController();
57. tvController.powerOn();
58. tvController.nextChannel();
59. tvController.turnUp();
60. tvController.powerOff();
61. tvController.nextChannel();
62. tvController.turnUp();
63. }
64. }

跟策略模式对比，TvController 环境类除了可以设置实现类，也在状态变化方法中自动替换实现类。

Android中的应用 wifi管理类

#### 小结

优点：state模式将所有与一个特定状态相关的行为都放入到一个状态对象中，它提供了一个更好的方法来组织与特定状态相关的代码，**将繁琐的状态判断转换为结构清晰的状态类族。**

结构清晰：避免了if else判断，提高可维护性

遵循设计原则：复合开闭原则和单一职责原则。要修改状态只需要修改一个子类就行

封装性好：状态变换放置到类的内部来实现，外部调用不用知道类内部如何实现状态和行为的变换。

缺点：类对象变多。

### 中介者模式

#### 定义

中介者模式包装了一序列对象的次昂胡作用的方式，使得这些对象**不必相互明显作用**。从而使得他们松散耦合。某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立变化。**中介者模式将多对多的相互作用转化为一对多的相互作用。**中介者模式将对象的行为和协作抽象化，**把对象在小尺度的行为上与其他对象的相互作用分开处理。**

#### 使用场景

当对象之间的交互操作很多且对象的行为操作都依赖彼此时，为防止在修改一个对象的行为时，同时涉及修改很多其他对象的行为，可采用中介者模式来解决紧耦合问题。该模式将对象之间的多对多关系变成一对多关系，**中介者对象将系统从网状结构变成以调停者为中心的星形结构，**达到降低系统复杂度，提高可扩展性的作用。

#### UML图

角色介绍：  
1，Mediator：抽象中介者，**定义统一的接口，用于各个同事角色之间的交流**，一般以抽象类的方式实现。

2，ConcreteMediator：具体中介者角色，继承抽象中介者，实现了父类定义的方法。它从具体的同事对象接收消息，向具体同事对象发出命令。**协调各同事角色协作行为**

3，Colleague：抽象同事类角色，定义了中介者对象的接口，**它只知道中介者而不知道其他的同事对象**

每个同事角色都知道中介者角色，而且与其他同事角色通信的时候，一定要通过中介者协作。每个同事类行为分为两种：同事本身的行为，改变自身的状态叫做自发行为；第二种必须依赖中介者才能完成的行为，叫做依赖方法；

4，ConcreteColleagueA/B：具体同事类角色，继承于抽象同事类，每个具体同事类都知道本身在小范围内的行为，而不知道它在大范围内的目的。

1. public abstract class Mediator {
2. *//定义同事类*
3. protected ConcreteColleagueA concreteColleagueA;
4. protected ConcreteColleagueB concreteColleagueB;
5. public abstract void doSomething() ;
6. public ConcreteColleagueA getConcreteColleagueA() {
7. return concreteColleagueA;
8. }
9. public void setConcreteColleagueA(ConcreteColleagueA concreteColleagueA) {
10. this.concreteColleagueA = concreteColleagueA;
11. }
12. public ConcreteColleagueB getConcreteColleagueB() {
13. return concreteColleagueB;
14. }
15. public void setConcreteColleagueB(ConcreteColleagueB concreteColleagueB) {
16. this.concreteColleagueB = concreteColleagueB;
17. }
18. }
19. public class ConcreteMediator extends Mediator {
20. @Override
21. public void doSomething() {
22. super.concreteColleagueA.selfMethod();
23. super.concreteColleagueB.selfMethod();
24. }
25. }
26. public abstract class Colleague {
27. protected Mediator mediator;
28. public Colleague(Mediator mediator){
29. this.mediator = mediator;
30. }
31. }
32. public class ConcreteColleagueA extends Colleague {
33. public ConcreteColleagueA(Mediator mediator) {
34. super(mediator);
35. }
36. */\*\**
37. \* 自有方法
38. \*/
39. public void selfMethod(){
40. System.out.println("selfMethod A");
41. }
42. */\*\**
43. \* 依赖方法
44. \*/
45. public void depMethod(){
46. System.out.println("depMethod A");
47. *//1，处理自己的业务逻辑*
48. *//2，委托中介者处理其他同事业务逻辑*
49. super.mediator.doSomething();
50. }
51. }
52. public class ConcreteColleagueB extends Colleague {
53. public ConcreteColleagueB(Mediator mediator) {
54. super(mediator);
55. }
56. */\*\**
57. \* 自有方法
58. \*/
59. public void selfMethod(){
60. System.out.println("selfMethod B");
61. }
62. */\*\**
63. \* 依赖方法
64. \*/
65. public void depMethod(){
66. System.out.println("depMethod B");
67. *//1，处理自己的业务逻辑*
68. *//2，委托中介者处理其他同事业务逻辑*
69. super.mediator.doSomething();
70. }
71. }
72. public class Client {
73. public static void main(String[] args) {
74. ConcreteMediator mediator = new ConcreteMediator();
75. ConcreteColleagueA concreteColleagueA = new ConcreteColleagueA(mediator);
76. ConcreteColleagueB concreteColleagueB = new ConcreteColleagueB(mediator);
77. mediator.setConcreteColleagueA(concreteColleagueA);
78. mediator.setConcreteColleagueB(concreteColleagueB);
79. concreteColleagueA.depMethod();
80. }
81. }

**为什么同事类需要使用构造函数注入中介者，而中介者使用get/set方式注入同事类呢？因为同事类必须有中介者，而中介者却可以只有部分同事类**

Demo2：电脑

1. public abstract class Mediator {
2. public abstract void changed(Colleague colleague);
3. }
4. public class MainBoard extends Mediator {
5. private CDDevice cdDevice;
6. private Cpu cpu;
7. private SoundCard soundCard;
8. private GraphicsCard graphicsCard;
9. public CDDevice getCdDevice() {
10. return cdDevice;
11. }
12. public void setCdDevice(CDDevice cdDevice) {
13. this.cdDevice = cdDevice;
14. }
15. public Cpu getCpu() {
16. return cpu;
17. }
18. public void setCpu(Cpu cpu) {
19. this.cpu = cpu;
20. }
21. public SoundCard getSoundCard() {
22. return soundCard;
23. }
24. public void setSoundCard(SoundCard soundCard) {
25. this.soundCard = soundCard;
26. }
27. public GraphicsCard getGraphicsCard() {
28. return graphicsCard;
29. }
30. public void setGraphicsCard(GraphicsCard graphicsCard) {
31. this.graphicsCard = graphicsCard;
32. }
33. @Override
34. public void changed(Colleague colleague) {
35. if (colleague == cdDevice) {
36. cpu.decode(cdDevice.read());
37. } else if (colleague == cpu) {
38. soundCard.soundPlay(cpu.getDataSound());
39. graphicsCard.videoPlay(cpu.getDataVideo());
40. }
41. }
42. }
43. public abstract class Colleague {
44. protected Mediator mediator;
45. public Colleague(Mediator mediator){
46. this.mediator = mediator;
47. }
48. }
49. */\*\**
50. \* CD设别负责读取光盘数据，并将数据提供给主板
51. \*
52. \*/
53. public class CDDevice extends Colleague {
54. private String data;*//视频数据*
55. public CDDevice(Mediator mediator) {
56. super(mediator);
57. }
58. public String read(){
59. return data;
60. }
61. public void load(){
62. data = "视频数据，音频数据";
63. mediator.changed(this);
64. }
65. }
66. */\*\**
67. \* CPU负责对主板传递来的音视频解码
68. \*
69. \*/
70. public class Cpu extends Colleague {
71. private String dataVideo, dataSound;*//视频音频数据*
72. public Cpu(Mediator mediator) {
73. super(mediator);
74. }
75. public String getDataVideo() {
76. return dataVideo;
77. }
78. public void setDataVideo(String dataVideo) {
79. this.dataVideo = dataVideo;
80. }
81. public String getDataSound() {
82. return dataSound;
83. }
84. public void setDataSound(String dataSound) {
85. this.dataSound = dataSound;
86. }
87. public void decode(String data) {
88. String[] tmp = data.split("，");
89. dataSound = tmp[0];
90. dataVideo = tmp[1];
91. *//告知中介者自身状态改变*
92. mediator.changed(this);
93. }
94. }
95. public class GraphicsCard extends Colleague {
96. public GraphicsCard(Mediator mediator) {
97. super(mediator);
98. }
99. public void videoPlay(String data){
100. System.out.println("播放："+data);
101. }
102. }
103. public class SoundCard extends Colleague {
104. public SoundCard(Mediator mediator) {
105. super(mediator);
106. }
107. public void soundPlay(String data){
108. System.out.println("播放："+data);
109. }
110. }
111. public class Client {
112. public static void main(String[] args) {
113. MainBoard mainBoard = new MainBoard();
114. CDDevice cdDevice = new CDDevice(mainBoard);
115. Cpu cpu = new Cpu(mainBoard);
116. GraphicsCard graphicsCard = new GraphicsCard(mainBoard);
117. SoundCard soundCard = new SoundCard(mainBoard);
118. mainBoard.setCdDevice(cdDevice);
119. mainBoard.setCpu(cpu);
120. mainBoard.setGraphicsCard(graphicsCard);
121. mainBoard.setSoundCard(soundCard);
122. *//开始播放*
123. cdDevice.load();
124. }
125. }

**MainBoard onchange方法中控制整个逻辑处理**

#### 小结

如果各种类的依赖关系如网状般错综复杂，那么必然会影响代码逻辑和执行效率。适当使用中介者可以让这种关系解耦，结构清晰。如果依赖关系并不复杂则不建议使用。

优点：减少类间依赖，把一对多变成一对一，减少依赖，降低耦合

缺点：中介者膨胀的很大，且逻辑复杂。

### 命令模式

#### 定义

将一个请求封装成一个对象，从而让用户使用不同的请求把客户端参数化；对请求排队或者记录请求日志，以及支持可撤销的操作

#### 使用场景

需要抽象出待执行的动作，然后以参数的形式提供出来

在不同的时刻指定，排列和执行请求。一个命令对象可以有与初始请求无关的生存期。

需要支持取消操作。

支持修改日志功能，这样当系统崩溃时，这些修改可以被重做一遍

需要支持事务操作。

#### UML类图

1. public class Receiver {
2. */\*\**
3. \* 真正执行具体命令逻辑的方法
4. \*/
5. public void action(){
6. System.out.println("执行具体操作");
7. }
8. }
9. public interface Command {
10. */\*\**
11. \* 执行具体操作的命令
12. \*/
13. void execute();
14. }
15. public class ConcreteCommand implements Command {
16. private Receiver receiver;
17. public ConcreteCommand(Receiver receiver){
18. this.receiver = receiver;
19. }
20. @Override
21. public void execute() {
22. receiver.action();
23. }
24. }
25. public class Invoke {
26. private Command command;
27. public Invoke(Command command){
28. this.command = command;
29. }
30. public void action(){
31. command.execute();
32. }
33. }
34. public class Client {
35. public static void main(String[] args) {
36. *//构造一个接收者*
37. Receiver receiver = new Receiver();
38. *//根据接收者对象构建一个命令对象*
39. Command command = new ConcreteCommand(receiver);
40. *//根据具体命令对象构造请求者对象*
41. Invoke invoke = new Invoke(command);
42. *//执行请求方法*
43. invoke.action();
44. }
45. }

Receiver：接收者角色。具体执行逻辑的角色

Command：命令角色。定义所有具体命令类的抽象接口

ConcreteCommand：具体命令角色

Invoke：请求者角色

Client：客户端角色

**一句话就是将行为调用者和实现者解耦。**

**为什么直接调用呢？当调用逻辑比较复杂或者调用行为有多种实现时，耦合在一起的写法就容易出问题**

#### 小结

优点：更弱的耦合性，更灵活的控制性，更好的扩展性

缺点：体现了所有设计模式的缺点：类膨胀。

### 观察者模式

#### 定义

也叫发布订阅模式。定义对象间一种一对多的依赖关系，使得当要给对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

#### 使用场景

关联行为场景，需要注意的时，关联行为时可拆分的，而不是“组合”关系

事件多级触发场景

跨系统的消息交换场景，如消息队列，事件总线的处理机制。

#### UML类图

Subject：抽象主题，也就是被观察（Observable）的角色。

ConcreteSubject：具体主题，又叫做具体被观察者角色

Observer：抽象观察则，观察者的抽象类

ConcreteObserver：具体观察者

1. public abstract class Subject {
2. private Vector<Observer> observers = new Vector<>();
3. public void addObserver(Observer observer){
4. observers.add(observer);
5. }
6. public void delObserver(Observer observer){
7. observers.remove(observer);
8. }
9. public void notifyObservers(){
10. for(Observer observer:observers){
11. observer.update();
12. }
13. }
14. }
15. public class ConcreteSubject extends Subject {
16. public void doSomething(){
17. *//1，处理自己的业务*
18. *//2，通知*
19. notifyObservers();
20. }
21. }
22. *//观察者*
23. public interface Observer {
24. void update();
25. }
26. public class ConcreteObserver implements Observer {
27. @Override
28. public void update() {
29. System.out.println("接收到消息，进行处理");
30. }
31. }
32. public class Client {
33. public static void main(String[] args) {
34. ConcreteSubject subject = new ConcreteSubject();
35. Observer observer = new ConcreteObserver();
36. subject.addObserver(observer);
37. subject.doSomething();
38. }
39. }

#### 小结

优点：

1，观察者和被观察者之间是抽象耦合。

如此设计，则不管是增加观察者还是被观察者都非常容易扩展。

2，建立一套触发机制。每个类的职责都是单一的

缺点：

1，默认是顺序执行，一个观察者卡壳，会影响后面的执行。这种情况一般考虑异步方式。

2，多级出发效率要注意

### 备忘录模式

#### 定义

在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样，以后就可以将该对象恢复到原先保存的状态。

#### 使用场景

1，需要保存一个对象在某一时刻的状态或部分状态

2，如果用一个接口来让其他对象得到这些状态，将会暴露对象的实现细节并破坏对象的封装性，一个对象不希望外界访问其内部状态，通过中间对象可以间接访问其内部状态。

#### UML类图

1，Originator：发起人。负责创建一个备忘录，可以记录，恢复自身的内部状态。同时Originator还可以根据需要决定Memento存储自身的哪些内部状态。

2，Memento：备忘录角色，用来存储Originator的内部状态，并且可以防止Originator意外对象访问Memento

3，CareTaker：负责存储备忘录，不能对被网络的内容进行操作和访问，只能够将备忘录传递给其他对象

1. \*\*
2. \* 使命召唤游戏，Originator发起人角色
3. \*/
4. public class CallOfDuty {
5. private int mCheckPoint = 1;
6. private int mLifeValue =100;
7. private String mWeapon = "沙漠之鹰";
8. public void play(){
9. System.out.println("玩游戏：");
10. mLifeValue -= 10;
11. mCheckPoint++;
12. }
13. public void quit(){
14. System.out.println("退出游戏：");
15. }
16. */\*\**
17. \* 创建备忘录(保存游戏进度)
18. \* @return
19. \*/
20. public Memoto createMemoto(){
21. Memoto memoto = new Memoto();
22. memoto.setmCheckPoint(mCheckPoint);
23. memoto.setmLifeValue(mLifeValue);
24. memoto.setmWeapon(mWeapon);
25. return memoto;
26. }
27. */\*\**
28. \* 恢复进度
29. \* @param memoto
30. \*/
31. public void restore(Memoto memoto){
32. mCheckPoint = memoto.getmCheckPoint();
33. mLifeValue = memoto.getmLifeValue();
34. mWeapon = memoto.getmWeapon();
35. }
36. }
37. \*\*
38. \* 备忘录角色
39. \*/
40. public class Memoto {
41. private int mCheckPoint;
42. private int mLifeValue;
43. private String mWeapon;
44. public int getmCheckPoint() {
45. return mCheckPoint;
46. }
47. public void setmCheckPoint(int mCheckPoint) {
48. this.mCheckPoint = mCheckPoint;
49. }
50. public int getmLifeValue() {
51. return mLifeValue;
52. }
53. public void setmLifeValue(int mLifeValue) {
54. this.mLifeValue = mLifeValue;
55. }
56. public String getmWeapon() {
57. return mWeapon;
58. }
59. public void setmWeapon(String mWeapon) {
60. this.mWeapon = mWeapon;
61. }
62. }
63. public class Caretaker {
64. private Memoto memoto;
65. public void archive(Memoto memoto){
66. this.memoto = memoto;
67. }
68. public Memoto getMemoto(){
69. return memoto;
70. }
71. }
72. public class Client {
73. public static void main(String[] args) {
74. *//构建游戏对象*
75. CallOfDuty game = new CallOfDuty();
76. *//打游戏*
77. game.play();
78. Caretaker caretaker = new Caretaker();
79. *//存档*
80. caretaker.archive(game.createMemoto());
81. *//退出游戏*
82. game.quit();
83. *//恢复游戏*
84. CallOfDuty newGame = new CallOfDuty();
85. newGame.restore(caretaker.getMemoto());
86. }
87. }

1，没有直接存储CallOfDuty对象，而是通过Memoto对CallOfDuty的数据进行存储，然后存储Memoto对象

2，存储操作不是在CallOfDuty，而是放在Caretaker对象。

角色清晰**单一，**对外屏蔽了CallOfDuty角色的直接访问。

Android中典型应用是onSaveInstanceState。

#### 小结

优点：

1，给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便的回到某个历史的状态

2，实现信息的封装，**使得用户不用关心状态保存的细节**

缺点：

消耗资源，如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

### 迭代器模式

#### 定义

提供一种方法顺序访问一个容器对象中的各个元素，**而又不需要暴露该对象的内部表示**。

#### 使用场景

遍历一个容器对象时

#### UML类图

1，Iterator：迭代器接口。负责定义访问和遍历元素的接口。基本上有固定3个方法：first()获得第一个元素，next()访问下一个元素；hasNext()是否访问到底部。

2，ConcreteIterator:具体迭代器类。实现迭代器接口，记录遍历的当前位置，完成容器遍历。

3，Aggregate:容器接口。容器接口负责提供创建具体迭代器角色的接口。在Java中一般是iterator()方法。

4，ConcreteAggregate：具体容器类。具体迭代器角色与该容器相关联。

举例：老板让A,B两员工统计两个部门的人数。

A用数组存储，B用链表存储。这两者都有各自办法来遍历出各个人员。但是需要注意的是，如果其他部门的人员都有各自的实现，那么**每个容器都要增加一个遍历逻辑，很不科学！**这个时候就可用到迭代器实现

1. public class Employee {
2. private String name;
3. private int age;
4. public Employee(String name, int age) {
5. this.age = age;
6. this.name = name;
7. }
8. }
9. public interface Iterator {
10. boolean hasNext();
11. Object next();
12. }
13. public class HuiIterator implements Iterator {
14. private Employee[] array;
15. private int position;
16. public HuiIterator(Employee[] array) {
17. this.array = array;
18. }
19. @Override
20. public boolean hasNext() {
21. return !(position > array.length - 1 || array[position] == null);
22. }
23. @Override
24. public Object next() {
25. Employee e = array[position];
26. position++;
27. return e;
28. }
29. }
30. public class MingIterator implements Iterator {
31. private List<Employee> list;
32. private int position;
33. public MingIterator(List<Employee> list) {
34. this.list = list;
35. }
36. @Override
37. public boolean hasNext() {
38. return !(position > list.size() - 1 || list.get(position) == null);
39. }
40. @Override
41. public Object next() {
42. Employee e = list.get(position);
43. position++;
44. return e;
45. }
46. }
47. public interface Depart {
48. Iterator iterator();
49. }
50. public class DepartHui implements Depart {
51. private Employee[] employees = new Employee[3];
52. public DepartHui(){
53. employees[0] = new Employee("cc",12);
54. employees[1] = new Employee("dd",122);
55. }
56. public Employee[] getEmployeeList(){
57. return employees;
58. }
59. @Override
60. public Iterator iterator() {
61. return new HuiIterator(employees);
62. }
63. }
64. public class DepartMing implements Depart {
65. private List<Employee> employeeList = new ArrayList<>();
66. public DepartMing(){
67. employeeList.add(new Employee("aa",12));
68. employeeList.add(new Employee("bb",13));
69. }
70. public List<Employee> getEmployeeList(){
71. return employeeList;
72. }
73. @Override
74. public Iterator iterator() {
75. return new MingIterator(employeeList);
76. }
77. }
78. public class Boss {
79. public static void main(String[] args) {
80. DepartMing ming = new DepartMing();
81. check(ming.iterator());
82. DepartHui hui = new DepartHui();
83. check(hui.iterator());
84. }
85. private static void check(Iterator iterator) {
86. while (iterator.hasNext()){
87. System.out.println(iterator.next().toString());
88. }
89. }
90. }

#### 小结

优点：支持以不同的方式遍历一个容器对象，也可以有多个遍历，弱化了容器类与遍历算法之间的关系。

### 访问者模式

#### 定义

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变这个数据结构的前提下定义作用域这些元素的新的操作。

基本思想是，软件系统中拥有一个由许多对象构成的，比较稳定的对象结构，这些对象的类都**拥有一个accept方法用来接受访问者对象的访问**。访问者是一个接口，它拥有一个visit方法，这个方法对访问到的对象中不同类型的元素作出不同的处理。在每一个accept方法中会调用访问者的visit方法，**可以针对对象结构设计不同的访问者类来完成不同的操作，达到区别对待的效果。**

#### 使用场景

1，对象结构比较稳定，但经常需要在此对象结构上定义新的操作。

-- **提供了一个切口，把自己交出去**。这样保证了自己的稳定性。

2，需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免这些操作“污染”这些对象的类，也不希望在增加新操作时修改这些类。

#### UML类图

1，Visitor：接口或者抽象类，它定义了对每一个元素（Element）访问的行为。**它的参数就是可以访问的元素（把自己交出去）。**所以这个方法个数理论上讲与元素个数是一样的，因此，访问者模式**要求元素的类族要稳定，如果经常添加，移除元素类，必然会导致频繁修改Visitor接口，如果出现这种情况则说明不适合使用访问者模式。**

2，ConcreteVisitor：具体访问者，实现对每一个元素类访问时所产生的具体行为。

3，Element：元素接口或者抽象类，定义了一个接收访问者(accept)的方法。**意义是指每一个元素都要可以被访问者访问。**

4，ElementABCD：具体元素类，提供接收访问方法的具体实现，而这个具体实现，通常情况下是使用访问者提供的访问该元素类的方法。

5，ObjectStructure：定义当中所提到的对象结构，对象结构是一个抽象表述，它内部管理了元素集合，并且可以迭代这些元素供访问者访问。

举例员工业绩考核。CEO和CTO对普通员工和经理的考核维度不同。

1. public interface Visitor {
2. void visit(Engineer engineer);
3. void visit(Manager manager);
4. }
5. public class CtoVisitor implements Visitor {
6. @Override
7. public void visit(Engineer engineer) {
8. System.out.println("工程师：" + engineer.name + ",代码行数:" + engineer.getCodeLines());
9. }
10. @Override
11. public void visit(Manager manager) {
12. System.out.println("经理：" + manager.name + ",KPI:" + manager.kpi + ",新产品数量：" + manager.getProducts());
13. }
14. }
15. public class CeoVisitor implements Visitor {
16. @Override
17. public void visit(Engineer engineer) {
18. System.out.println("工程师：" + engineer.name + ",KPI:" + engineer.kpi);
19. }
20. @Override
21. public void visit(Manager manager) {
22. System.out.println("经理：" + manager.name + ",KPI:" + manager.kpi + ",新产品数量：" + manager.getProducts());
23. }
24. }
25. public abstract class Staff {
26. public String name;
27. public int kpi;
28. public Staff(String name){
29. this.name = name;
30. kpi = new Random().nextInt(10);
31. }
32. public abstract void accept(Visitor visitor);
33. }
34. public class Engineer extends Staff {
35. public Engineer(String name){
36. super(name);
37. }
38. @Override
39. public void accept(Visitor visitor) {
40. *//切口中，把自己送出去了*
41. visitor.visit(this);
42. }
43. *//工程师的代码量*
44. public int getCodeLines(){
45. return new Random().nextInt(10\*10000);
46. }
47. }
48. public class Manager extends Staff {
49. private int products;*//产品数量*
50. public Manager(String name){
51. super(name);
52. products = new Random().nextInt(10);
53. }
54. @Override
55. public void accept(Visitor visitor) {
56. *//切口中，把自己送出去了*
57. visitor.visit(this);
58. }
59. public int getProducts(){
60. return products;
61. }
62. }
63. public class BusinessReport {
64. private List<Staff> mStaffs = new ArrayList<>();
65. public BusinessReport(){
66. mStaffs.add(new Manager("王经理"));
67. mStaffs.add(new Engineer("张工"));
68. }
69. public void showReport(Visitor visitor){
70. for(Staff staff:mStaffs){
71. staff.accept(visitor);
72. }
73. }
74. }
75. public class Client {
76. public static void main(String[] args) {
77. *//构建报表*
78. BusinessReport report = new BusinessReport();
79. *//给CEO看的报表*
80. report.showReport(new CeoVisitor());
81. *//给CTO看的报表*
82. report.showReport(new CtoVisitor());
83. }
84. }

Android中的典型应用就是ATP，ButterKnife

#### 小结

优点：

1，各角色职责分离，符合单一职责原则。

2，具有优秀的扩展性

3，使得数据结构和作用于结构上的操作解耦，使得操作集合可以独立变化

4，灵活性

缺点：

1，具体元素对访问者公布细节，违反了迪米特原则

2，具体元素变更时导致修改成本大

3，违反了依赖倒置原则，为了达到“区别对待“而依赖了具体类，并没有依赖抽象（否则就得用ifelse + instance判断了）

### 责任链模式

#### 定义

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，只到由对象处理它为止。

#### 使用场景

多个对象可以处理同一个请求，但具体由哪个对象处理则在运行时动态决定

在请求处理者不明确的情况下向多个对象中一个提交一个请求。

需要动态指定一组对象处理请求

#### UML类图

1. public class Level {
2. }
3. public class Request {
4. public Level getRequestLevel() {
5. return null;
6. }
7. }
8. public class Response {
9. }
10. public abstract class Handler {
11. private Handler nextHandler;
12. public final Response handleMessage(Request request) {
13. Response response = null;
14. if (this.getHandlerLevel().equals(request.getRequestLevel())) {
15. response = this.echo(request);
16. } else {
17. if (this.nextHandler != null) {
18. response = this.nextHandler.handleMessage(request);
19. } else {
20. }
21. }
22. return response;
23. }
24. public void setNext(Handler handler) {
25. nextHandler = handler;
26. }
27. protected abstract Level getHandlerLevel();
28. protected abstract Response echo(Request request);
29. }
30. public class ConcreteHandler1 extends Handler {
31. @Override
32. protected Level getHandlerLevel() {
33. return null;
34. }
35. @Override
36. protected Response echo(Request request) {
37. return null;
38. }
39. }
40. public class ConcreteHandler2 extends Handler {
41. @Override
42. protected Level getHandlerLevel() {
43. return null;
44. }
45. @Override
46. protected Response echo(Request request) {
47. return null;
48. }
49. }
50. public class ConcreteHandler3 extends Handler {
51. @Override
52. protected Level getHandlerLevel() {
53. return null;
54. }
55. @Override
56. protected Response echo(Request request) {
57. return null;
58. }
59. }
60. public class Client {
61. public static void main(String[] args) {
62. Handler handler1 = new ConcreteHandler1();
63. Handler handler2 = new ConcreteHandler2();
64. Handler handler3 = new ConcreteHandler3();
65. handler1.setNext(handler2);
66. handler2.setNext(handler3);
67. Response response = handler1.handleMessage(new Request());
68. }
69. }

#### 小结

优点：将请求和处理分开。解耦，提高灵活性

缺点：

1，性能问题。每个请求都是从头遍历到尾，链比较长的额时候，性能低下

2，调试不方便

### 解释器模式

用的比较少，不记录了

### 命令模式VS策略模式

#### 关注点不同

策略模式关注的是算法替换的问题，一个新的算法投产，旧算法退休，或者提供多种算法由调用者自己选择使用，**算法的自由替换是它实现的要点。**策略模式关注的是算法的完整性，封装性，只有具备了这两个条件才能额保证其可以自由切换。

命令模式则关注的是解耦问题，如何让请求者和执行者解耦是它首先考虑的，解耦的要求就是把请求内容封装成一个个命令，由接收者执行。由于封装成了命令，就同时可以对命令进行多种处理。

#### 角色功能不同

策略模式中的具体算法是负责一个完整算法逻辑，它是不可再拆分的原子业务单元，一旦变更就是对算法整体的变更。

命令关心的是功能的实现。接收者对命令负责，而与请求者无关。接收者变更，只影响到命令族的变更，对请求者没有影响。

#### 使用场景不同

策略模式适用于算法要求变换的场景，而命令模式适用于解耦两个由紧耦合关系的对象场合或者命令多撤销的场景。

### 策略模式VS状态模式

#### 环境角色的职责不同

两者都有一个叫做Context环境角色的类，策略模式环境角色只是一个委托作用，负责算法的替换；状态模式的环境角色不仅仅是委托行为，还具有登记状态变化的功能，与具体的状态类协作，共同完成状态切换行为随之切换的任务。

#### 解决的重点不同

策略模式旨在解决算法如何改变的问题，将内部算法的改变对外界的影响降低到最小，它保证的是算法的自由切换；而状态模式旨在解决内在状态改变引起行为改变的问题，出发点是事物的状态，封装状态而暴露行为，一个对象的状态改变，从外界看来就像是行为改变。

#### 解决问题的方法不同

策略模式只是确保算法可以自由切换，但是什么时候用什么算法它决定不了。

状态模式对暴露的是行为，状态的变化一般是由环境角色和具体状态共同完成，也就是说状态模式封装了状态的变化而暴露了不同的行为或行为结果。

#### 使用场景不同

算法必须是平行的。状态模式则要求由一序列状态发生变化的场景，它要求的是有状态且有行为的场景，也就是一个对象必须具有二维（状态和行为）描述才能采用状态模式，如果只有状态没有行为，状态的变化就失去了意义。

## 结构型设计模式

### 代理模式

#### 定义

代理模式就是代理对象具备真实对象的功能，并替代真实对象完成相应操作，并能在操作执行的前后，对操作进行增强处理。（为真实对象提供代理，把这个代理给调用方用来访问真实对象）

#### 静态代理

1. public interface IRentHouse {
2. void rentHouse();
3. }
4. public class RentHouse implements IRentHouse {
5. @Override
6. public void rentHouse() {
7. System.out.println("实现租房");
8. }
9. }
10. public class IntermediaryProxy implements IRentHouse {
11. private IRentHouse iRent;
12. public IntermediaryProxy(IRentHouse iRentHouse) {
13. iRent=iRentHouse;
14. }
15. @Override
16. public void rentHouse() {
17. dosomenThingBefore();
18. iRent.rentHouse();
19. dosomenThingAfter();
20. }
21. private void dosomenThingBefore(){
22. System.out.println("交中介费");
23. }
24. private void dosomenThingAfter(){
25. System.out.println("中介负责维修管理");
26. }
27. }
28. public class Client {
29. public static void main(String[] args) {
30. IRentHouse rentHouse = new RentHouse();
31. IRentHouse iRentHouseProxy = new IntermediaryProxy(rentHouse);
32. iRentHouseProxy.rentHouse();
33. }
34. }

#### 动态代理

**从静态代理的代码中可以发现，静态代理的缺点显而易见，那就是当真实类的方法越来越多的时候，这样构建的代理类的代码量是非常大的，所以就引进动态代理.**

**动态代理只能代理接口**

动态代理允许使用一种方法的单个类（代理类）为具有任意数量方法的任意类（真实类）的多个方法调用提供服务，看到这句话，可以容易的联想到动态代理的实现与反射密不可分。

JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

#### jdk动态代理（接口代理）

Jdk代理涉及到java.lang.reflect包中的InvocationHandler接口和Proxy类，核心方法是

1. public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

jdk动态代理过程中实际上代理的是**接口**，是因为在创建代理实例的时候，依赖的是java.lang.reflect包中Proxy类的newProxyInstance方法，该方法的生效就恰恰需要这个参数；

1. public static Object newProxyInstance(ClassLoader loader,
2. Class<?>[] interfaces,
3. InvocationHandler h)

**在执行过程中，为两个接口分别生成了编译以后的虚拟代理类$Proxy0.class 和 $Proxy1.class，下面以接口1的动态代理过程，讲述jdk动态代理的来龙去脉**

总结对比：

1.静态代理中，代理类和真实类实现的是同一个接口，重写同样的方法；jdk动态代理中，代理类和真实类关系不大，代理类实现无侵入式的代码扩展。

2.静态代理中当接口中方法增加的时候，在代理类代码量也会增加，显然是不妥的；jdk动态代理解决了这个问题，当业务增加，代理类的代码不会增加。

3.jdk动态代理实现的是jdk自带InvocationHandler接口，实现了这个接口的类也叫拦截器类，也叫代理类。

#### cglib动态代理

从上面可以看出，jdk动态代理的前提条件是，要有接口存在，那还有许多场景是没有接口的，这个时候就需要cglib动态代理了，CGLIB(Code Generation Library)是一个基于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。

以下案例中所用到的\*\*被代理类\*\*和和上面jdk动态代理一样

cglib动态代理和jdk动态代理的区别显而易见，但是实现逻辑差不多，cglib代理类是通过实现MethodInterceptor，重写intercept方法，通过生成被代理类的子类来达到代理增强代码的目的；而Jdk代理是通过实现InvocationHandler，重写invoke方法，通过生成接口的代理类来达到代码增强的目的，所以jdk动态代理的实现需要接口，cglib则不需要，spring5.0以上以及springboot2.0以上默认采用cglib动态来实现AOP。后面将专门介绍AOP以及spring中是如何实现AOP的！不当之处欢迎大家批评并指正！

### 组合模式

#### 定义

将对象组合成树状结构一表示“部分-整体“的层次结构，使得用户对单个对象和组合对象的使用具有一致性。也叫 合成模式或部分整体模式

#### 使用场景

1，表示对象的部分-整体层次结构时

2，从一个整体中能够独立出部分模块或功能的场景

#### UML类图

1，Component抽象构建角色

2，Composite树枝构件

3，Leaf 叶子构件

组合模式由两种不同的实现：透明模式和安全模式。透明模式是把用来组合使用的方法放到抽象类中。

1. public abstract class Component {
2. protected String name;
3. public Component(String name){
4. this.name = name;
5. }
6. public abstract void doSomething();
7. }
8. public class Composite extends Component {
9. private List<Component> components = new ArrayList<>();
10. public Composite(String name) {
11. super(name);
12. }
13. @Override
14. public void doSomething() {
15. System.out.println(name);
16. for(Component c:components){
17. c.doSomething();
18. }
19. }
20. public void addChild(Component component){
21. components.add(component);
22. }
23. public void removeChild(Component component){
24. components.remove(component);
25. }
26. public Component getChild(int index){
27. return components.get(index);
28. }
29. }
30. public class Leaf extends Component {
31. public Leaf(String name) {
32. super(name);
33. }
34. @Override
35. public void doSomething() {
36. System.out.println(name);
37. }
38. }
39. public class Client {
40. public static void main(String[] args) {
41. *//构建根节点*
42. Composite root = new Composite("root");
43. *//枝干*
44. Composite branch1 = new Composite("Branch1");
45. Composite branch2 = new Composite("Branch2");
46. *//叶子*
47. Leaf leaf1 = new Leaf("Leaf1");
48. Leaf leaf2 = new Leaf("Leaf2");
49. branch1.addChild(leaf1);
50. branch2.addChild(leaf2);
51. root.addChild(branch1);
52. root.addChild(branch2);
53. root.doSomething();
54. }
55. }

透明模式

1. public abstract class Component {
2. protected String name;
3. public Component(String name){
4. this.name = name;
5. }
6. public abstract void doSomething();
7. public abstract void addChild(Component component);
8. public abstract void removeChild(Component component);
9. public abstract Component getChild(int index);
10. }
11. public class Composite extends Component {
12. private List<Component> components = new ArrayList<>();
13. public Composite(String name) {
14. super(name);
15. }
16. @Override
17. public void doSomething() {
18. System.out.println(name);
19. for(Component c:components){
20. c.doSomething();
21. }
22. }
23. @Override
24. public void addChild(Component component){
25. components.add(component);
26. }
27. @Override
28. public void removeChild(Component component){
29. components.remove(component);
30. }
31. @Override
32. public Component getChild(int index){
33. return components.get(index);
34. }
35. }
36. public class Leaf extends Component {
37. public Leaf(String name) {
38. super(name);
39. }
40. @Override
41. public void doSomething() {
42. System.out.println(name);
43. }
44. @Override
45. public void addChild(Component component) {
46. throw new UnsupportedOperationException("没有子节点");
47. }
48. @Override
49. public void removeChild(Component component) {
50. throw new UnsupportedOperationException("没有子节点");
51. }
52. @Override
53. public Component getChild(int index) {
54. throw new UnsupportedOperationException("没有子节点");
55. }
56. }
57. public class Client {
58. public static void main(String[] args) {
59. *//构建根节点*
60. Composite root = new Composite("root");
61. *//枝干*
62. Composite branch1 = new Composite("Branch1");
63. Composite branch2 = new Composite("Branch2");
64. *//叶子*
65. Leaf leaf1 = new Leaf("Leaf1");
66. Leaf leaf2 = new Leaf("Leaf2");
67. branch1.addChild(leaf1);
68. branch2.addChild(leaf2);
69. root.addChild(branch1);
70. root.addChild(branch2);
71. root.doSomething();
72. }
73. }

#### 小结

优点：

组合模式可以清楚的定义分层次的复杂对象，表示对象的全部或者部分层次，它让高层模块忽略了层次的差异，方便对整个层次结构进行控制

高层模块可以一致的使用要给组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了高层模块的代码

在组合模式中增加新的枝干构建和叶子构件都很方便，无需对现有类库进行修改，复合开闭原则

缺点：

新增构件时不好对枝干中的构件类型进行限制，不能依赖类型系统来施加约束，因为大多数情况下，他们都来自于相同的抽象层，此时必须进行类型检查来实现。

### 适配器模式

#### 定义

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能在一起工作。

#### 使用场景

1，系统需要使用现有的类，而此类的接口不符合系统的需要，即接口不兼容

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

3，需要一个统一的输出接口，而输入端的类型不可预知

#### UML类图

**包含类适配器模式和对象适配器模式。**

1，Target：目标角色。也就是所期待得到的接口。如果是类是类适配器模式，目标不可以是类

2，Adaptee：需要适配的接口

3，适配器角色。将源接口转换成目标接口。

#### 类适配器

1. public interface Target {
2. void request();*//目标角色有自己的方法*
3. }
4. public class ConcreteTarget implements Target {
5. @Override
6. public void request() {
7. System.out.println("00000");
8. }
9. }
10. */\*\**
11. \* 待适配的 源角色
12. \*/
13. public class Adaptee {
14. *//原有的业务逻辑*
15. public void doSomething(){
16. System.out.println("1111");
17. }
18. }
19. public class Adapter extends Adaptee implements Target {
20. @Override
21. public void request() {
22. *//偷天换日*
23. super.doSomething();
24. }
25. }
26. public class Client {
27. public static void main(String[] args) {
28. *//原有的业务逻辑*
29. Target target = new ConcreteTarget();
30. target.request();
31. *//增加了适配器*
32. target = new Adapter();
33. target.request();
34. }
35. }

#### 对象适配器

对象适配器不是使用继承关系链接到Adaptee类，二十使用代理关系到的。

1. public class Adapter implements Target {
2. private Adaptee adaptee;
3. public Adapter(Adaptee adaptee) {
4. this.adaptee = adaptee;
5. }
6. @Override
7. public void request() {
8. *//偷天换日*
9. adaptee.doSomething();
10. }
11. }
12. public class Client {
13. public static void main(String[] args) {
14. *//原有的业务逻辑*
15. Target target = new ConcreteTarget();
16. target.request();
17. *//增加了适配器*
18. target = new Adapter(new Adaptee());
19. target.request();
20. }
21. }

对象适配器采用组合形式实现更灵活，

被适配器对象中的方法不会暴露出来，类适配器由于继承了被适配对象，被适配对象类的方法在Adapter类中也含有，多余。增加使用成本

**小结**

优点：

1，更好的复用性。 系统需要使用现有的类，但此类的接口不符合系统的需要。通过适配器可以将这些功能更好的复用

2，更好的扩展性。实现适配器功能的时候，可以调用自己开发的功能，自然的扩展系统的功能

缺点：

过多使用适配器，会让系统零乱，不易整体把握。明明看到A接口，内部却调用的是B接口。

### 装饰者模式

#### 定义

动态的给对象添加一些额外的职责。就增加功能来说，**装饰模式比生成子类更为灵活。**

其使用一种对客户端透明的方式来动态地扩展对象的功能，同时它是继承关系的一种替代方案之一。

#### 使用场景

需要透明且动态地扩展类的功能时。

#### UML类图

1，Component抽象构件

时一个接口或者抽象类，就是定义我们最核心的对象，也就是最原始的对象。

在装饰模式中，必然有一个最基本，核心，原始的接口或抽象类充当Component抽象构件。

2，ConcreteComponent具体构件

实现类，我们要装饰的就是它

3，Decorator装饰角色

一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法，在它的属性里必然有一个private变量指向Component抽象构件

4，具体装饰角色

1. public abstract class Component {
2. public abstract void operate();
3. }
4. public class ConcreteComponent extends Component {
5. @Override
6. public void operate() {
7. System.out.println("ConcreteComponent operate");
8. }
9. }
10. public abstract class Decorator extends Component {
11. private Component component;
12. public Decorator(Component component){
13. this.component = component;
14. }
15. @Override
16. public void operate() {
17. component.operate();
18. }
19. }
20. public class ConcreteDecorator1 extends Decorator {
21. public ConcreteDecorator1(Component component) {
22. super(component);
23. }
24. @Override
25. public void operate() {
26. method1();
27. super.operate();
28. }
29. *//定义自己扩展的修饰方法*
30. private void method1(){
31. System.out.println("ConcreteDecorator1 method1");
32. }
33. }
34. public class ConcreteDecorator2 extends Decorator {
35. public ConcreteDecorator2(Component component) {
36. super(component);
37. }
38. @Override
39. public void operate() {
40. method2();
41. super.operate();
42. }
43. *//定义自己扩展的修饰方法*
44. private void method2(){
45. System.out.println("ConcreteDecorator2 method2");
46. }
47. }
48. public class Client {
49. public static void main(String[] args) {
50. Component component = new ConcreteComponent();
51. *//第一次修饰*
52. component = new ConcreteDecorator1(component);
53. *//第二次修饰*
54. component = new ConcreteDecorator2(component);
55. component.operate();
56. }
57. }

**这里要注意的是装饰类继承了抽象组件，并且内部又持有一个抽象组件。继承关系表明了装饰类is a 抽象构件。不管装饰了多少次，它还是is a 抽象将构件。客户端调用方式不用改变**

**内部持有则是为了功能的扩展**

Android应用 Contextimpl , 包含头尾的listview

#### 小结

优点：  
装饰类和被装饰类可以独立发展，不会相互耦合

装饰模式是继承关系的一个替代方案。返回的对象还是Component，实现的还是is a 的关系

装饰模式可以动态的扩展一个实现类的功能，

缺点：

多层装饰需要剥洋葱，增加了复杂度

### 享元模式

#### 定义

使用共享对象可有效的支持大量的细粒度的对象

将这些对象的信息分为两部分：内部状态和外部状态。

内部状态：是对象可共享出来的信息，存储在享元对象内部并且不会随着环境改变而改变，可以作为一个对象的动态附加细腻些，不必直接存储在某个对象中，属于可共享部分。

外部对象：是对象得以依赖的标记，随着环境改变而改变，不可共享的状态。它是一批对象的统一标识。例子中的出发点-目的地就是外部状态

#### 使用场景

1，系统中存在大量的相似对象

2，细粒度的对象都具备较相近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份

3，需要缓冲池的场景

#### UML类图

1，Flyweight：抽象享元角色。产品的抽象类，定义出对象的外部状态和内部状态的接口或实现

2，ConcreteFlyweight：具体享元角色。不应该出现一个操作改变了内部状态，同时修改了外部状态。

3，unsharedConcreteFlyweight：不可共享的享元角色

不存在外部状态或者安全要求不能够使用共享技术的对象

4，FlyweightFactory：享员工厂

1. public abstract class Flyweight {
2. *//内部状态*
3. private String intrinsic;
4. *//外部状态*
5. protected final String extrinsic;
6. *//要求享元角色必须接收外部状态*
7. public Flyweight(String extrinsic){
8. this.extrinsic = extrinsic;
9. }
10. *//定义业务操作*
11. public abstract void operate();
12. *//内部状态的访问*
13. public String getIntrinsic() {
14. return intrinsic;
15. }
16. public void setIntrinsic(String intrinsic) {
17. this.intrinsic = intrinsic;
18. }
19. }
20. public class ConcreteFlyweight1 extends Flyweight {
21. public ConcreteFlyweight1(String extrinsic) {
22. super(extrinsic);
23. }
24. @Override
25. public void operate() {
26. }
27. }
28. public class FlyweightFactory {
29. *//定义一个池容器*
30. private static HashMap<String,Flyweight> pool = new HashMap<>();
31. *//享员工厂*
32. public static Flyweight getFlyweight(String extrinsic){
33. Flyweight flyweight = null;
34. if(pool.containsKey(extrinsic)){
35. flyweight = pool.get(extrinsic);
36. }else {
37. flyweight = new ConcreteFlyweight1(extrinsic);
38. pool.put(extrinsic,flyweight);
39. }
40. return flyweight;
41. }
42. }

Android典型Message.obtain()

#### 小结

可以大大减少应用创建的对象，降低程序内存的占用，增加程序的性能。但同时提高了系统复杂度，需要分离外部状态和内部状态，而且外部状态具有固化特性，不应该随内部状态改变而改变。否则逻辑混乱

### 外观模式

#### 定义

要求一个子系统的外部与内部的通信必须通过一个统一的对象进行。外观模式提供一个高层次的接口，使得子系统更易于使用。也称门面模式

就是通过一个统一地接口对外提供服务，使得外部程序只通过一个类就可以实现系统内部地多种功能，而这些实现功能地内部子系统之间可能也有交互，或者说完成一个功能需要几个子系统之间进行协作，如果没有封装，那么用户就需要操作几个子系统地交互逻辑，容易出现错误。通过外观类对外屏蔽这些复杂地交互，降低用户使用成本。

#### 使用场景

1，为一个复杂子系统提供一个简单接口。子系统往往因为不断演化变得越来越复杂，甚至被替换。外观模式提供一个统一简单的接口，对外隐藏子系统的具体实现，隔离变化。

2，当你需要构件要给层次结构的子系统时，使用外观模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，额可以让他们仅通过外观模式进行通信，从而简化了他们之间的依赖关系。

#### UML类图

Façade：系统对外的统一接口，系统内部系统地工作

SystemABCD：子系统接口

1. public interface IPhone {
2. void dial();
3. void hangup();
4. }
5. public class PhoneImpl implements IPhone {
6. @Override
7. public void dial() {
8. System.out.println("打电话");
9. }
10. @Override
11. public void hangup() {
12. System.out.println("挂断");
13. }
14. }
15. public interface ICamera {
16. void open();
17. void takePicture();
18. void close();
19. }
20. public class HuaweiCamera implements ICamera {
21. @Override
22. public void open() {
23. System.out.println("打开相机");
24. }
25. @Override
26. public void takePicture() {
27. System.out.println("拍照");
28. }
29. @Override
30. public void close() {
31. System.out.println("关闭相机");
32. }
33. }
34. public class MobilePhone {
35. private IPhone phone = new PhoneImpl();
36. private ICamera camera = new HuaweiCamera();
37. public void dial(){
38. phone.dial();
39. }
40. public void videoChat(){
41. camera.open();
42. phone.dial();
43. }
44. public void hangup(){
45. phone.hangup();
46. }
47. public void takePicture(){
48. camera.open();
49. camera.takePicture();
50. }
51. public void closeCamera(){
52. camera.close();
53. }
54. }

#### 小结

优点：

1，对客户程序隐藏子系统细节，因而减少了客户对于子系统地耦合，能够拥抱变化。

2，外观类对子系统地接口封装，使得系统更易于使用

缺点：

1，外观类接口膨胀。由于子系统地接口都有外观类统一对外暴露，使得外观类API接口较多，增加使用成本

2，外观类没有遵循开闭原则，业务发生变更时，可能需要直接修改外观类

### 桥接模式

#### 定义

将抽象部分与实现部分分离，使得他们都可以独立地进行变化。

重点就是解耦，抽象角色引用实现角色，或者说抽象角色地部分实现是由实现角色完成的。

#### 使用场景

如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，可以通过桥接模式使他们在抽象层建立一个关联关系。

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，也可以考虑使用桥接模式

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展

#### UML类图

Abstraction：抽象部分。

该类保持一个对实现部分对象的引用，抽象部分中的方法需要调用实现部分的对象来实现，该类一般为抽象类。

RefinedAbstraction：优化的抽象部分。

抽象部分的具体实现，该类一般时对抽象部分的方法进行完善和扩展。

Implementor：实现部分。

可以为接口或抽象类，其方法不一定要与抽象部分中的一致，一般情况下是由实现部分提供基本操作，而抽象部分定义的则是基于实现部分这些基本操作的业务方法。

ConcreteImplementorABC：实现部分的具体实现

完善实现部分中方法定义的具体逻辑

举例：大杯小杯是否加糖 咖啡类

1. */\*\**
2. \* 抽象化角色
3. \*/
4. public abstract class Coffee {
5. *//抽象化角色中对实现化角色的引用 ！！*
6. protected CoffeeAdditives coffeeAdditives;
7. public Coffee(CoffeeAdditives additives) {
8. coffeeAdditives = additives;
9. }
10. */\*\**
11. \* 具体什么样子由子类决定
12. \*/
13. public abstract void makeCoffee();
14. }
15. public class LargeCoffee extends Coffee {
16. public LargeCoffee(CoffeeAdditives additives) {
17. super(additives);
18. }
19. @Override
20. public void makeCoffee() {
21. System.out.println("大杯的 "+coffeeAdditives.addSomething()+" 咖啡");
22. }
23. }
24. public class SmallCoffee extends Coffee {
25. public SmallCoffee(CoffeeAdditives additives) {
26. super(additives);
27. }
28. @Override
29. public void makeCoffee() {
30. System.out.println("小杯的 " + coffeeAdditives.addSomething() + " 咖啡");
31. }
32. }
33. */\*\**
34. \* 实现化角色
35. \*/
36. public abstract class CoffeeAdditives {
37. */\*\**
38. \* 加料
39. \* @return
40. \*/
41. public abstract String addSomething();
42. }
43. public class Ordinary extends CoffeeAdditives {
44. @Override
45. public String addSomething() {
46. return "原味";
47. }
48. }
49. public class Sugar extends CoffeeAdditives {
50. @Override
51. public String addSomething() {
52. return "加糖";
53. }
54. }
55. public class Client {
56. public static void main(String[] args) {
57. *//原汁原味*
58. Ordinary ordinary = new Ordinary();
59. *//加糖*
60. Sugar sugar = new Sugar();
61. LargeCoffee largeOrinary = new LargeCoffee(ordinary);
62. largeOrinary.makeCoffee();
63. SmallCoffee smallSugarCoffee = new SmallCoffee(sugar);
64. smallSugarCoffee.makeCoffee();
65. }
66. }

#### 小结

对于抽象与实现的分离的把握，是不是需要分离，如何分离？

### 代理模式VS装饰模式

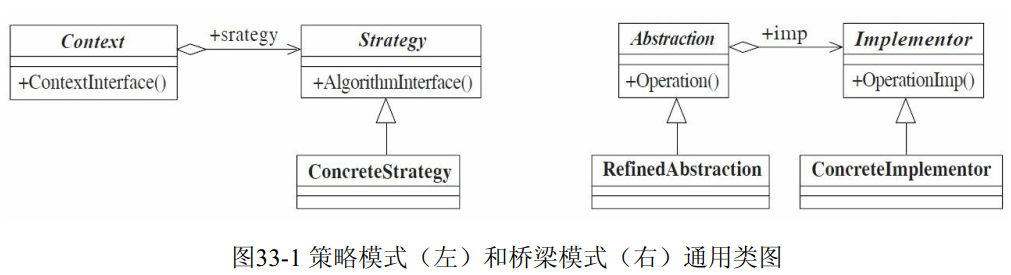
装饰模式就是代理模式的一个特殊应用，两者的共同点就是都具有相同的接口，不同点则是代理模式着重对代理过程的控制，而装饰模式则是对类的功能进行加强和减弱。它着重类的功能变化。

### 装饰模式VS适配器模式

类图上没有太多相似点，差别比较大，但是功能上有相似的地方：都是包装作用，都是通过委托方式实现其功能。不同点是：装饰模式包装的是自己的兄弟类，率属于同一个家族（相同接口或父类），适配器模式则是修饰非血缘关系类，把一个非本家族的对象伪装成本家族的对象。

## 跨战区PK

### 策略模式VS桥接模式



如此相似，只能从**意图**上分析。策略模式是一个行为模式，旨在封装一系列的行为，封装的算法不同，行为也就不同。而桥梁模式则是解决在不破坏封装的情况下如何抽取出它的抽象部分和实现部分，它的前提是不破坏封装，让抽象部分和实现部分都可以独立变化。

策略模式是使用继承和多态加你一套可以自由切换算法的模式，桥梁模式是在不破坏封装的前提下解决抽象和实现都可以独立扩展的模式。桥梁模式必然有两个桥墩：抽象化角色和实现化角色，而策略模式只有一个抽象角色，可以没有实现，也可以有很多实现。

### 外观模式VS中介者模式

外观模式以封装和隔离为主要任务，而中介者模式则以调和同事类之间的关系为主，因为要调和**所以具有了部分的业务逻辑控制。**

1，功能区别

外观模式只是增加了一个门面。它对子系统来说没有增加任何功能，子系统若脱离外观模式则完全可以独立运行。而中介模式增加了业务功能，把各个同事类中原有耦合关系移植到了中介者，同事类不可能脱离中介者独立存在，除非是想增加系统的复杂性和降低扩展性。

2，知晓状态不同

对于外观模式来说，子系统不知道外观模式存在，而对于中介者来说，每个同事类都知道中介者存在，因为要依靠中介者调和同事之间的关系，他们对中介者非常了解。

3，封装程度不同

外观模式是一种简单的封装，所有的请求处理都委托给子系统完成，而中介者模式则需要有一个中心，由中心协调同事类完成，并且中心本身也完成部分业务，它属于更进一步的业务功能封装。

### 包装模式PK

很多模式中的角色是不干具体的活的。他们只是充当黔首作用，你有问题找我，我不处理，我让其他人处理。这类模式成为包装模式。包括：装饰模式，适配器模式，门面模式，代理模式，桥梁模式

5个包装模式具有相似的特征：都是通过委托的方式对一个对象实行包装，有了包装系统才更加灵活，稳定。从实现的角度来看，他们都是代理的一种具体表现形式。看看使用场景上有什么区别：

代理模式主要用在不希望展示一个对象内部细节的场景中，增加一个代理类，可以轻松实现被代理类的功能封装。此外，代理模式还可以用在一个对象的访问需要限制的场景中，比如AOP

装饰模式是一种特殊的代理模式，倡导的是在不改变接口的前提下为对象增强功能，或者动态添加额外职责。

适配器模式主要是接口转换

桥梁模式是在抽象层产生耦合，解决的是自行扩展的问题，可以使两个有耦合关系的对象互不影响的扩展

门面模式是一个粗粒度的更壮，提供一个方便访问子系统的接口，不具有任何业务逻辑，仅仅是一个访问复杂系统的快速通道，没有它，子系统照常运行，有了它，只是更方便访问而已。

## 设计模式混编