# 六大设计准则

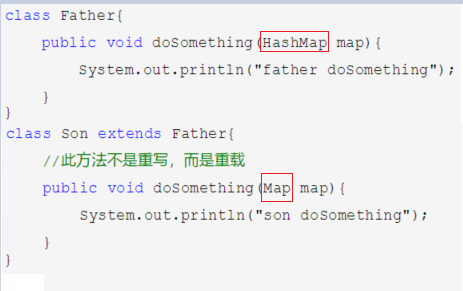
## 单一职责原则：

接口： 一个接口只实现一个功能

类： 尽可能做到只有一个原因可以引起类的改变

## 里氏替换原则

1. 子类必须实现父类的virtual函数
2. 子类可以有自己的个性
3. 覆盖或实现父类的方法时，形参可以被放大



1. 覆盖或实现父类的方法时，返回值可以被缩小

## 接口隔离原则

客户端不应该依赖它不需要的接口；

一个类对另一个类的依赖，应该建立在最小的接口上

## 迪米特法则：最少知道原则

一个类应该尽可能少的知道另一个类的细节

## 开闭原则

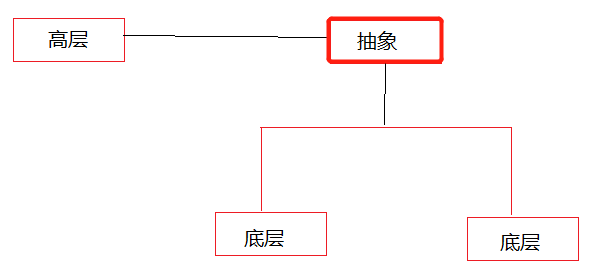
定义：对扩展开放，对修改封闭

简单的说：软件实体应该在尽量不要修改原有代码的基础上，进行扩展

实现技巧：动态绑定，运行时的多态

## 依赖倒置原则

1. 抽象不依赖于细节，细节应当依赖于抽象
2. 高层（业务逻辑）不依赖于底层（业务逻辑具体实现），二者都依赖于抽象



# 01. 单例模式

**特点：**

（1）确保类只有一个实例：private static Singleton\* m\_singleton;

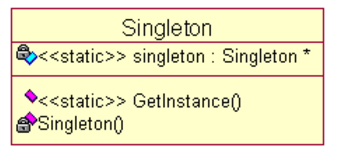
（2）自行实例化:

（3）提供外界可以获得该实例的方法: public static Singleton\* GetInstance()

**应用场景：**

每台计算机可以有若干个**打印机**，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干**通信端口**，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

**代码实现：**





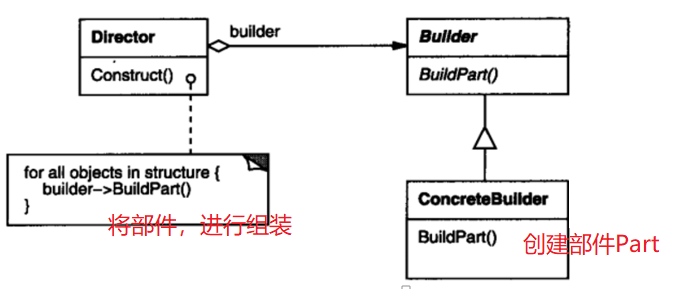
# Builder模式

将一个复杂对象的**构建（组装）**和**表示（部分创建）**分离🡪相同的构建过程，获得不同的构建结果。

**举例：建房子**

1. 房子的墙，门，窗户的规格不一样，在不同的厂子builder中生产
2. 但是，建造房子时，组装墙，门，窗户的顺序都是一致的

最后生产出来的房屋，组装顺序虽然相同，但是由于不同工厂生产的墙，门，窗户的规格不同，导致最后建造的房屋也具有很多样貌。





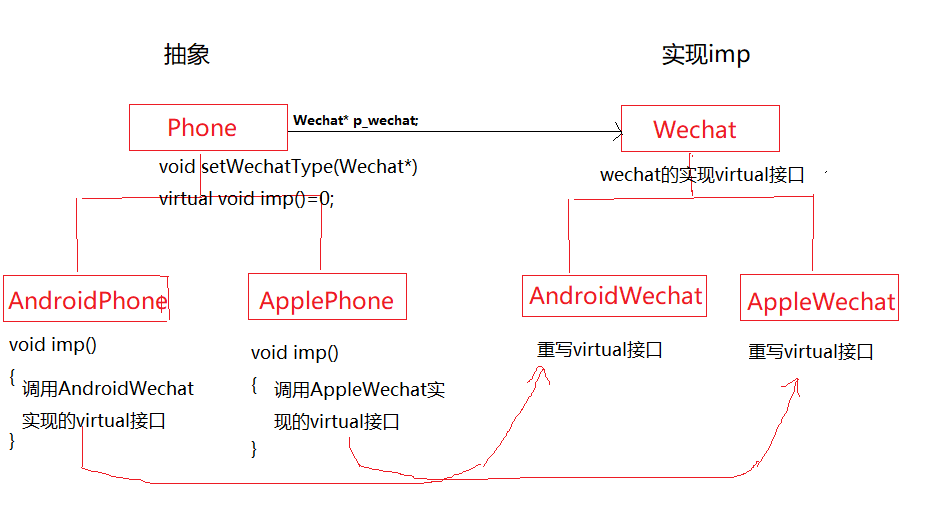
# 桥接模式

将**抽象部分**和**它的实现部分**相分离，使得它们能独立的变化。

举例：Apple手机平台和Android手机平台；二者都实现wechat程序，其中wechat程序对应也有两个版本，AppleWechat和AndroidWechat🡪要实现Apple手机运行AppleWechat程序，Android手机运行AndroidWechat程序🡪

方案一（NO）：采用继承，会造成类的膨胀

方案二（Yes）：可以将运行Wechat程序抽象成接口；供手机Phone类调用🡪从而实现二者都可以独立的变化





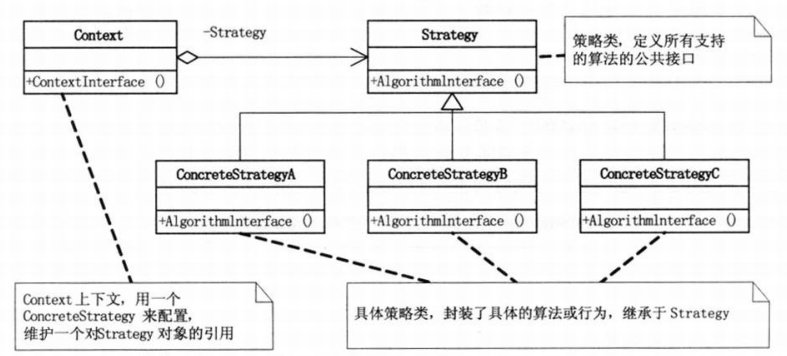
罗克韦尔实习中使用过Bridge模式：

平台Client/Server使用不同的加锁方式IPCBase(CriticalSection,Mutext,Semphore,Event等)

# 策略模式

**使用场景：当使用一个算法的判断条件过于复杂时，有大量的if..else…**

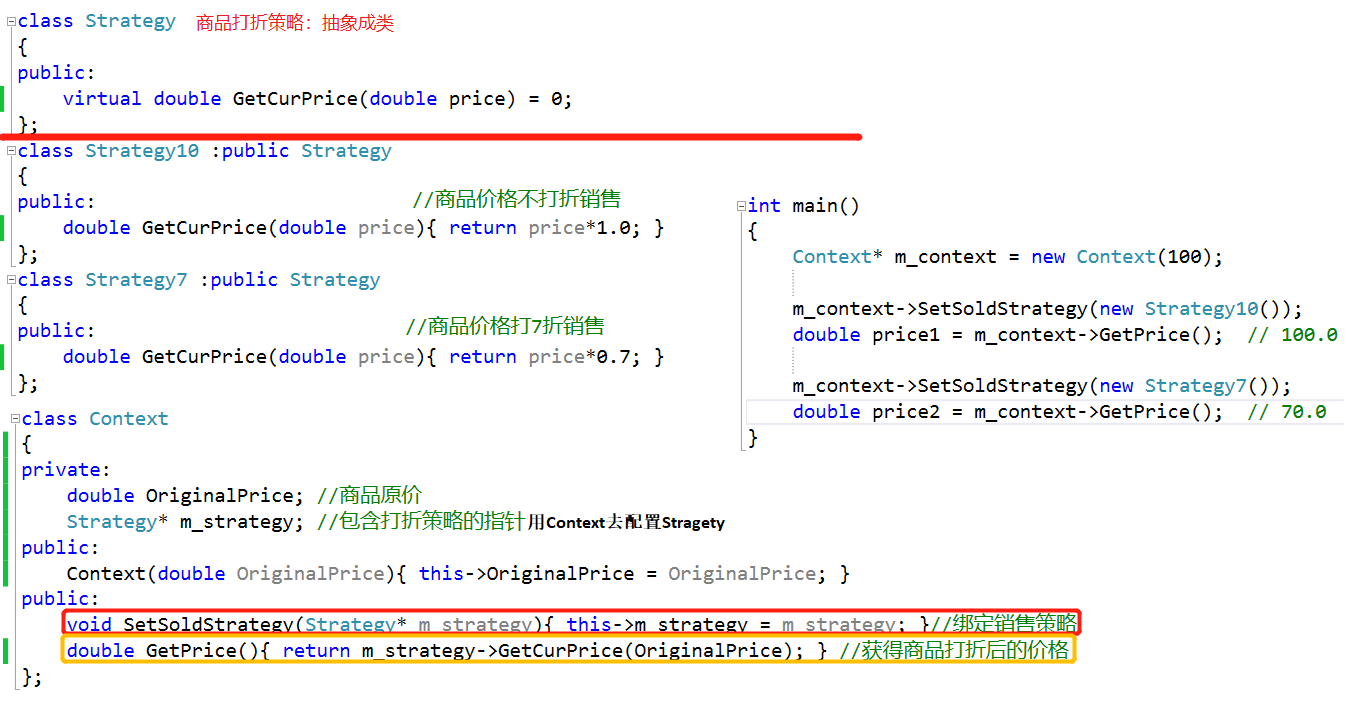
将Alogrithm抽象成接口封装到一个类中Strategy，用Context去配置Stragety，从而使Client端只需要知道Context，就能调用相应的Strategy。



Head First：会飞的鸭子

举例：商场售卖商品，有的商品原价出售，有的商品打折（7折）销售；

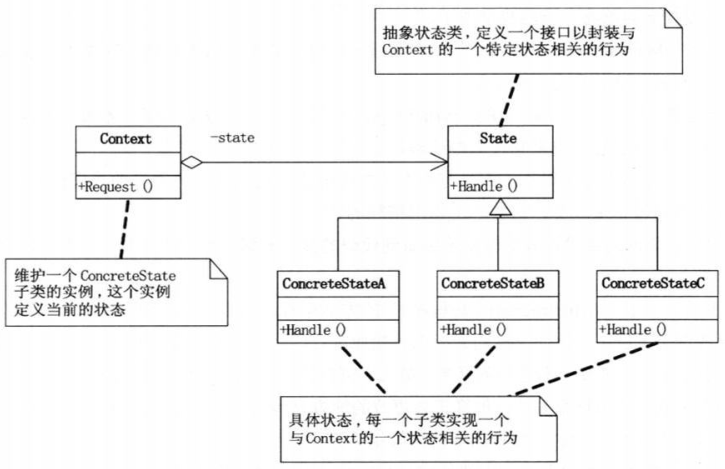
对于商品有很多销售策略🡪🡪因此可以将销售策略抽象成strategy interface，在Context中包含strategy\*变量，client通过Context绑定strategy，进而执行不同的strategy。



# 状态模式

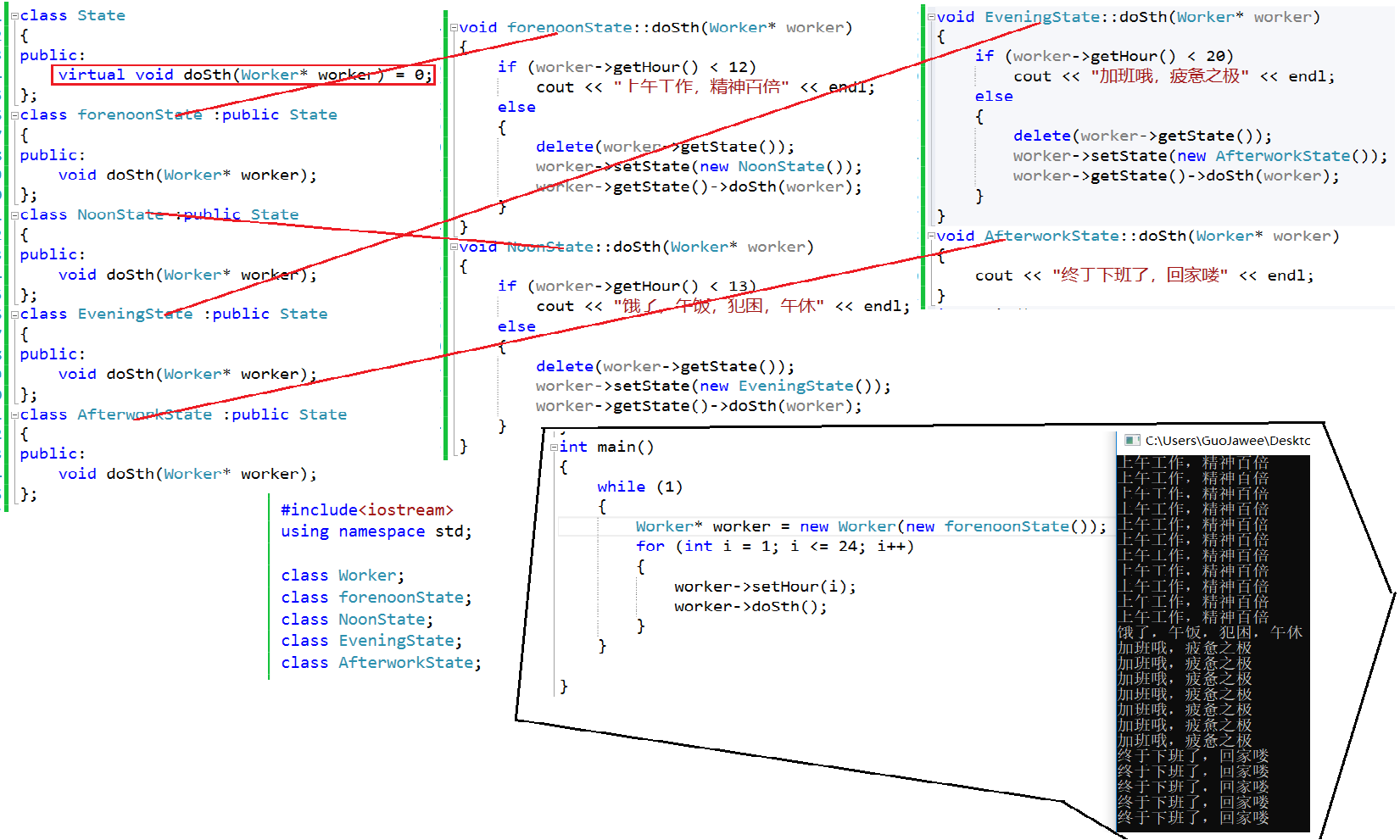
**使用场景：当控制一个对象状态的判断条件过于复杂时，有大量的if..else…**

State模式很好地实现了对象的状态逻辑和动作实现的分离，状态逻辑分布在State的派生类中实现，而动作实现则可以放在Context类中实现（这也是为什么State派生类需要拥有一个指向Context的指针）。这使得两者的变化相互独立，改变State的状态逻辑可以很容易复用Context的动作，也可以在不影响State派生类的前提下创建Context的子类来更改或替换动作实现。



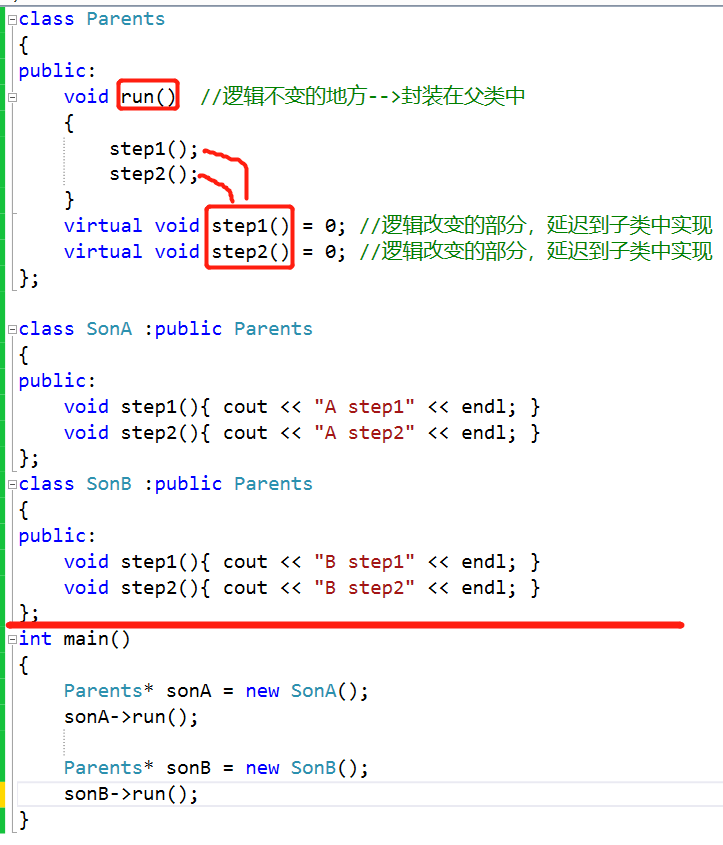
举例：一个Worker从早上起床到晚上下班，要经过以下时间（每个时间段都对应一个状态）：forenoonState,NoonState,EveningState,AfterworkState🡪🡪将worker的state从worker中抽象出来，使得worker与state解耦合。

其中state类中，只有一个接口void doSth(Worker\* worker)



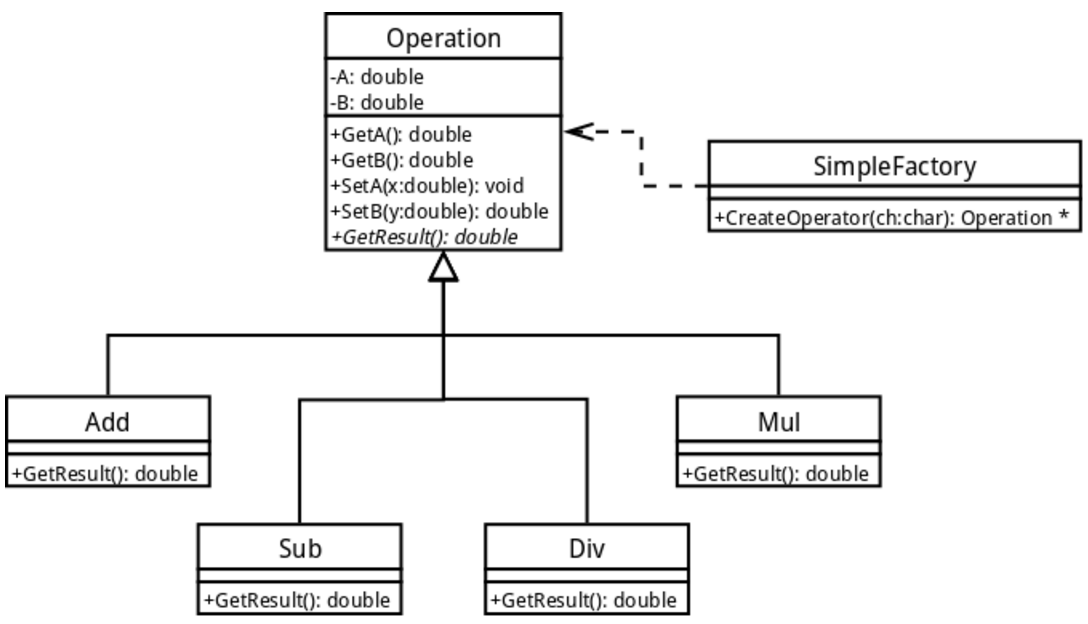
# 模板方法

本质：不变部分封装到父类，可变部分通过继承的方式在子类实现

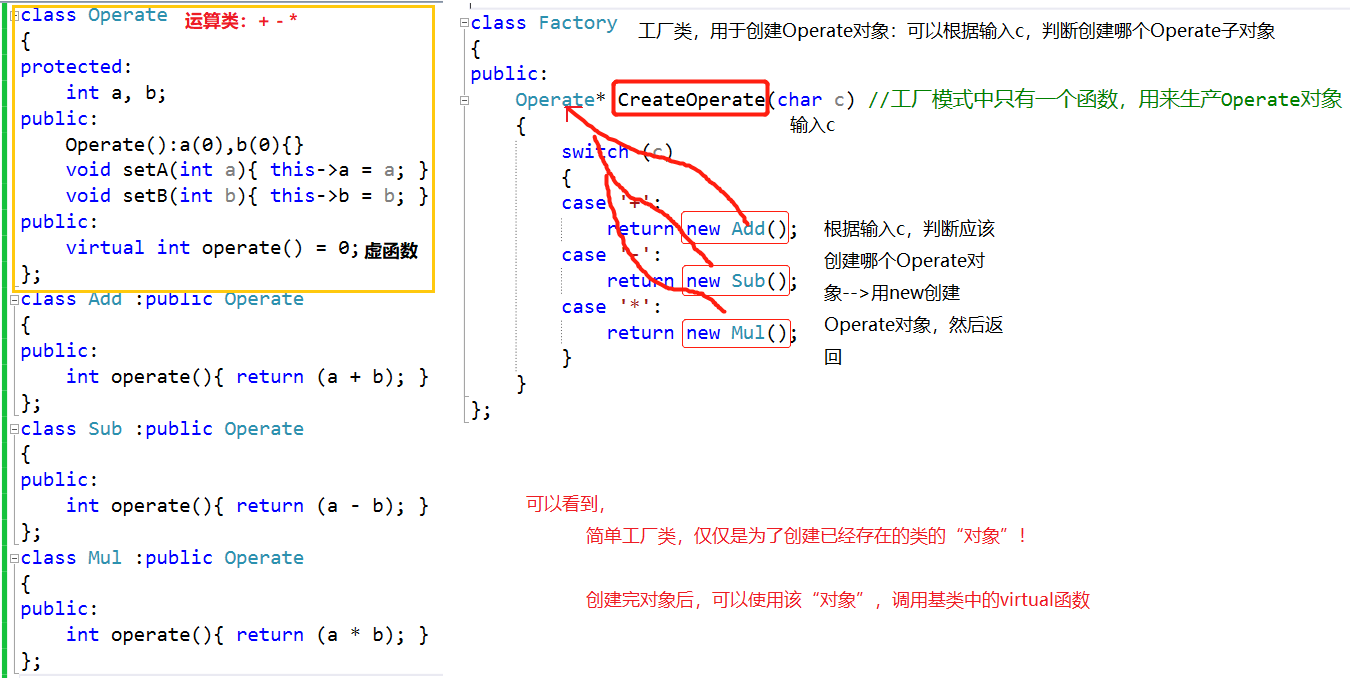


# 简单工厂模式

核心：对于一个父类的多个继承子类，工厂对象的工厂函数根据用户输入，自动new出一个子类对象并返回其父类的指针，这样利用父类的指针执行父类的虚函数，就可以动态绑定子类的重写函数，从而实现多态。



**代码实现：**



罗克韦尔实习中使用过simple facetory模式：

创建IPCBace对象时，使用switch…case…判断输入字符串的是CriticalSection，Mutext，Semphore，还是Event，进而通过Factory的IPCBase\* CreateIPCBase(string str)创建以上对象。

**用户需求在扩增：假如此时需要在原有的+ - \*运算的基础上，添加/运算**

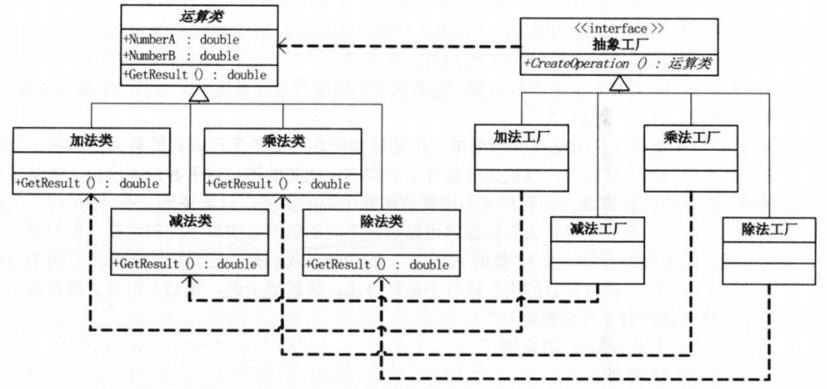
**那么如果使用简单工厂方法，需要在swtich…case…中加入case ‘/‘ 🡪🡪🡪这样破坏了“封闭原则”，程序的改动非常大🡪🡪🡪因此简单工厂模式在实际的应用中比较少，缺点很明显🡪🡪🡪引出工厂方法**

# 工厂方法Factory Meythod

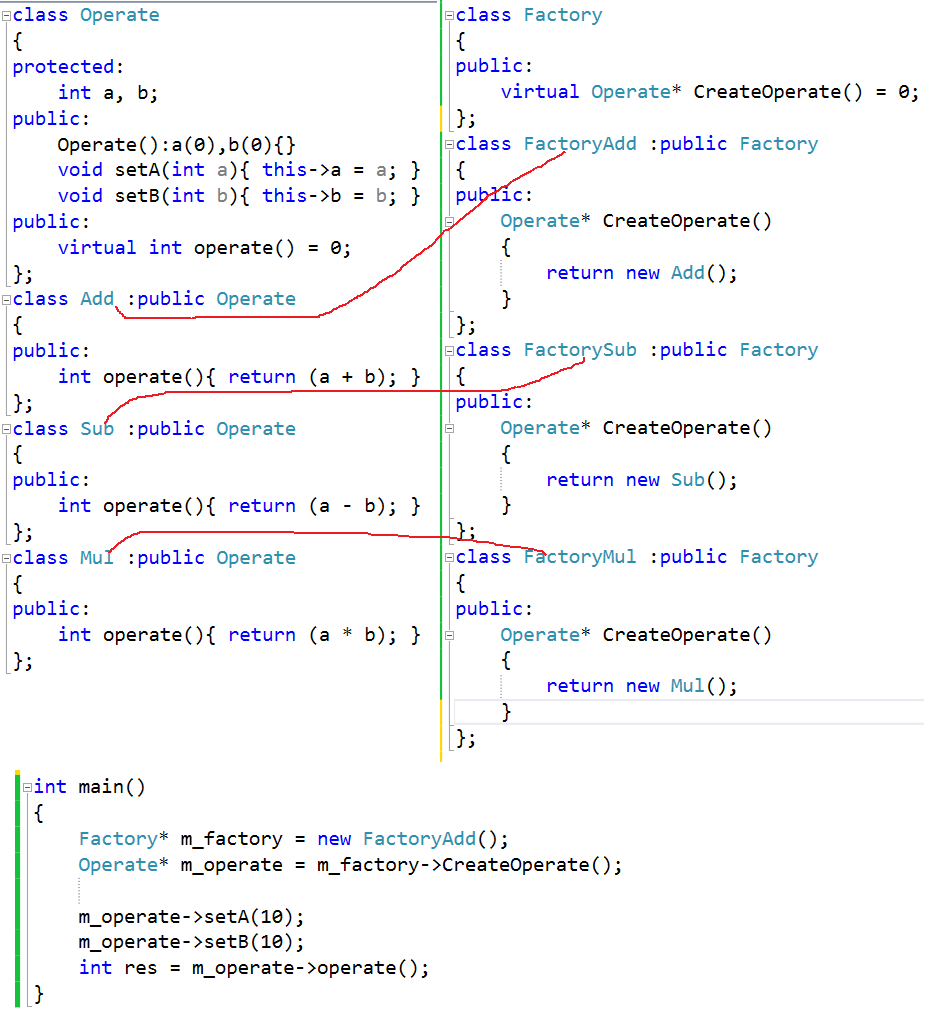
在简单工厂的基础上，进行下面修改

1. 将简单工厂中的工厂抽象出来
2. 抽象工厂类，每一个具体工厂都对应生产一个具体产品

好处：对于新增的产品，只需要在抽象工厂类下，创建新的工厂子类🡪🡪不破坏“开闭原则”



**代码实现：**



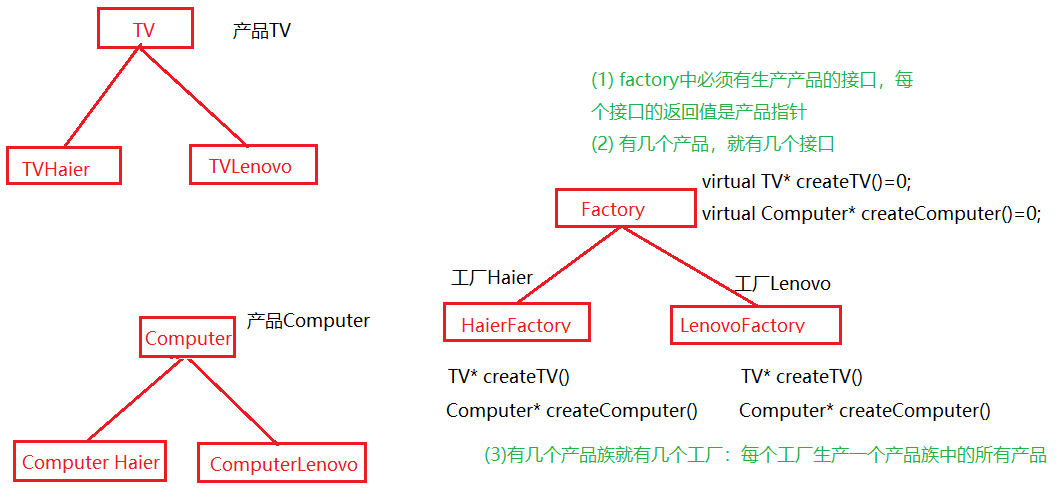
# 抽象工厂

举例：现在有两种产品TV和Computer；有两个公司Haier和Lenovo都生产TV和Computer；

分析：

(1)有2种产品：TV和Computer 🡪 因此Factory类中有两个接口

(2)有两个产品族：Haier和Lenovo 🡪 因此有Factory有两个子类，即有两个工厂



总结：

**简单工厂：**工厂类中，根据条件决定一个接口由哪个具体产品类来实现。

**工厂方法：**在简单工厂的基础上，将工厂抽象出来🡪每个工厂对应一个产品

生产TV和Computer，用两个工厂FactoryTV和FactoryComputer

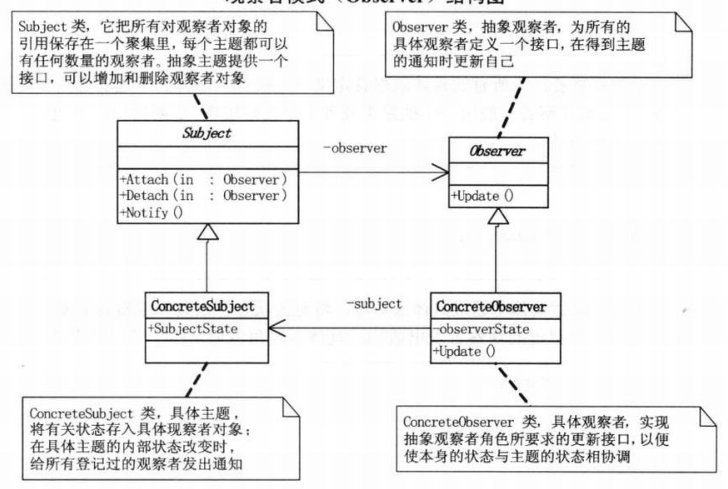
**抽象工厂：**

有几个产品，Factory中就有几个create方法；

有几个产品族，就有几个Factory工厂子类（每个工厂，生产产品族中的所有产品）

# Observer模式（消息发布-订阅模式）

观察者模式：observer订阅subject，当subject发出消息notify()时，所有的observer都进行update()🡪🡪**Subject:Observer=1:N**



**举例：**

（1）数据库Subject在进行更新时，将以饼状图/柱状图/线型图/表格等Observer形式呈现

（2）在工作生活中，有时经常会有员工开小差，比如：小王在看NBA，小明在打王者荣耀，小红在化妆等；此时有一个人（假设名字叫做subject）对老板进行监控，当老板来时，subject对小王/小明/小红发出通知notify()，与此同时小王/小明/小红收到通知后，立刻装作“努力工作”。

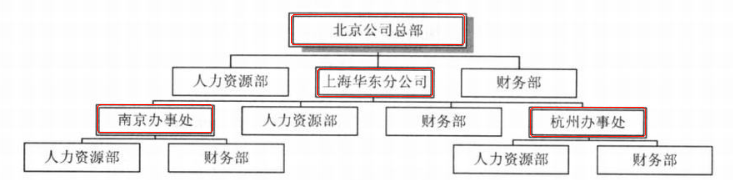


# 组合模式（树形结构）

**使用场景：组合模式让客户可以一致地使用组合对象和单个对象的接口**

当你发现需求中是体现**部分与整体**层次的结构时，以及希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时：用户不用关心到底是处理一个叶节点还是处理一个组合组件，也就不用写判断语句辨别是单一对象还是组合对象

基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断地递归下去。**举例：**

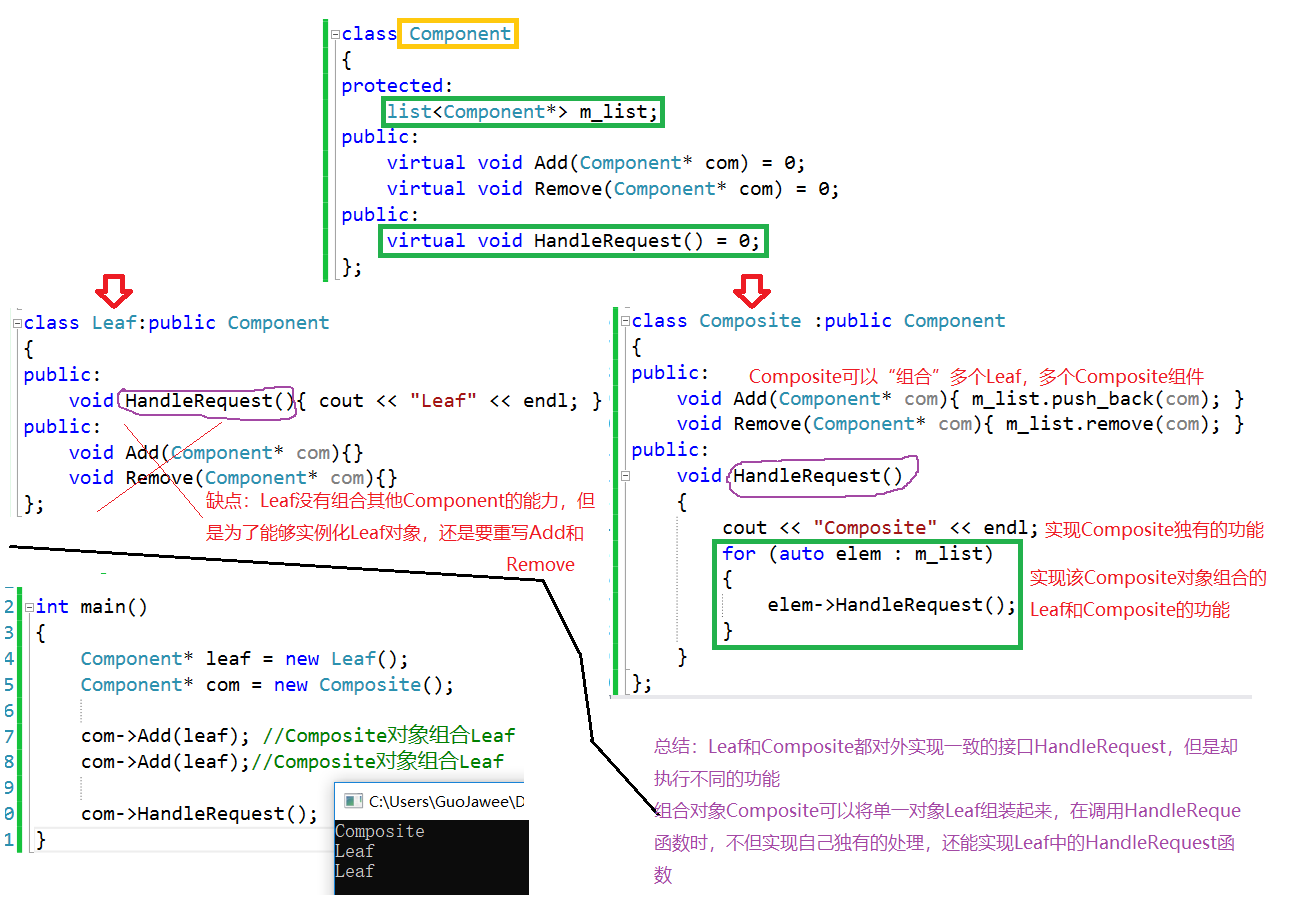


小菜已经开发出了人力资源部，财务部的办公管理系统，需求：某个大公司希望将这套办公管理系统在全公司进行推广。

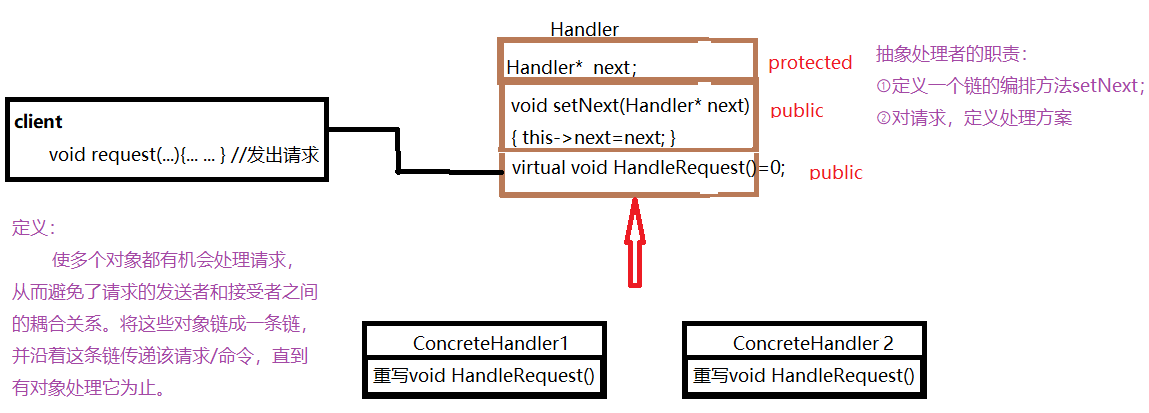
此时此刻，小菜已经开发好的人力资源部办公管理系统，财务部办公管理系统

解决方案1：对于每个分公司，都将人力资源部，财务部的管理系统都复制一份，工作量巨大。

解决方案2：将人力资源部，财务部的管理系统当作Leaf，将分公司当作Composite，因此Composite可以通过Add(Leaf)，实现Leaf功能的复用。



# 职责链模式（链表结构）



举例：小王申请“加薪水”，但是加薪水可不是那么简单的事情，操作流程那是相当的复杂：（1）如果申请增加的薪水小于500元，经理可以直接决定是否批准；

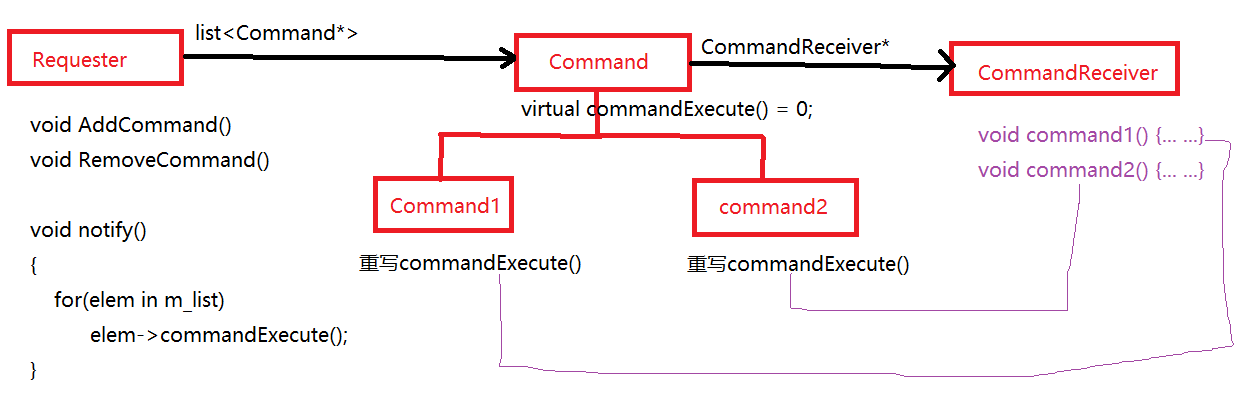
（2）如果申请增加的薪水大于500元，经理就没有权限决定是否批准，将会将消息向主管汇报，由主管决定是否批准

这种运作模式，形成一条链表，当前节点不能处理请求，就将请求转交给next节点，直到请求被处理。



# 命令模式

**将命令封装成对象，使得命令请求者和命令实现者解耦合**



**命令请求者Requester：**包含请求的命令列表list<Command\*>，添加/删除命令操作，通知命令列表中的命令执行的操作notify()

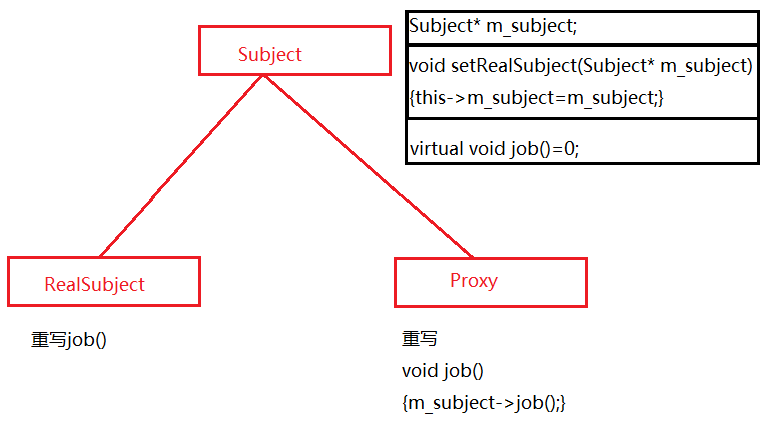
**命令Command：**包含命令接收者CommandReceiver；只有一个virtual函数命令执行接口

**命令接收者CommandReceiver**：有几个命令子类，就对应有几个函数，真正的指向命令的内容

**举例：**客户(main函数)找到Waiter(命令请求者)进行点菜(命令)，Waiter将点的餐放入菜单列表，点完菜后，Waiter通知厨师做菜。



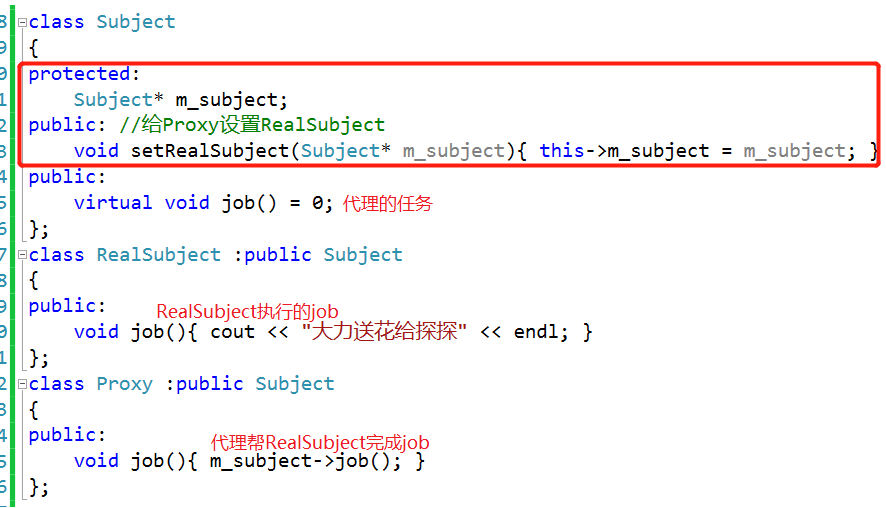
# 代理模式



代理模式非常的简单，举个通俗易懂的例子：假设大力RealSubject要送花给探探，但是由于自己没有时间，因此大力RealSubject将送花的任务委托给东东Proxy去做。这就是一个很简单的代理模式

分析：

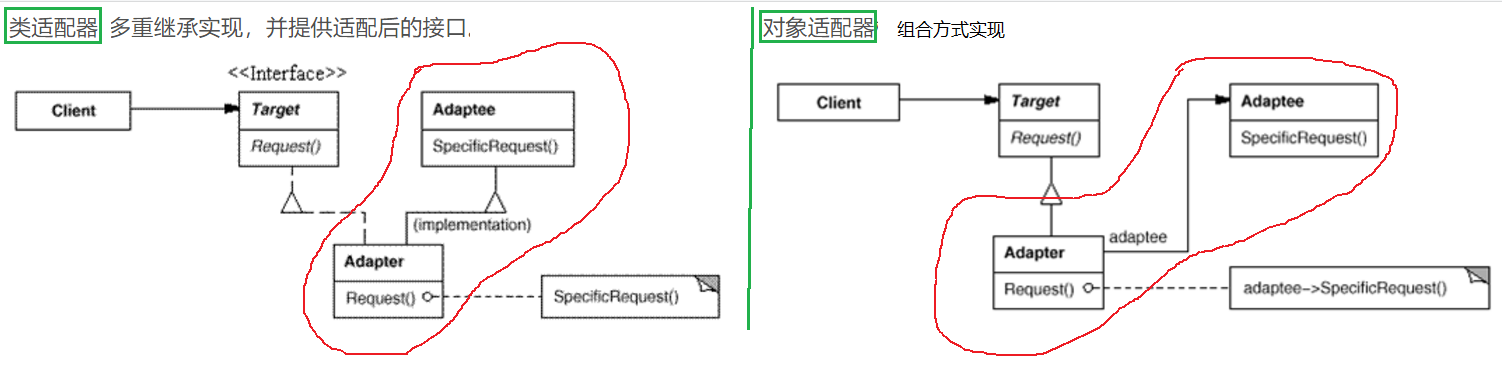
在代理模式中，有两个角色——任务委托者（大力），任务执行者（东东），东东是大力的代理。执行送花动作的人虽然是东东，但是实际上RealSubject实际主题确实大力。



# 适配器模式

说明：用的最多的结构型设计模式

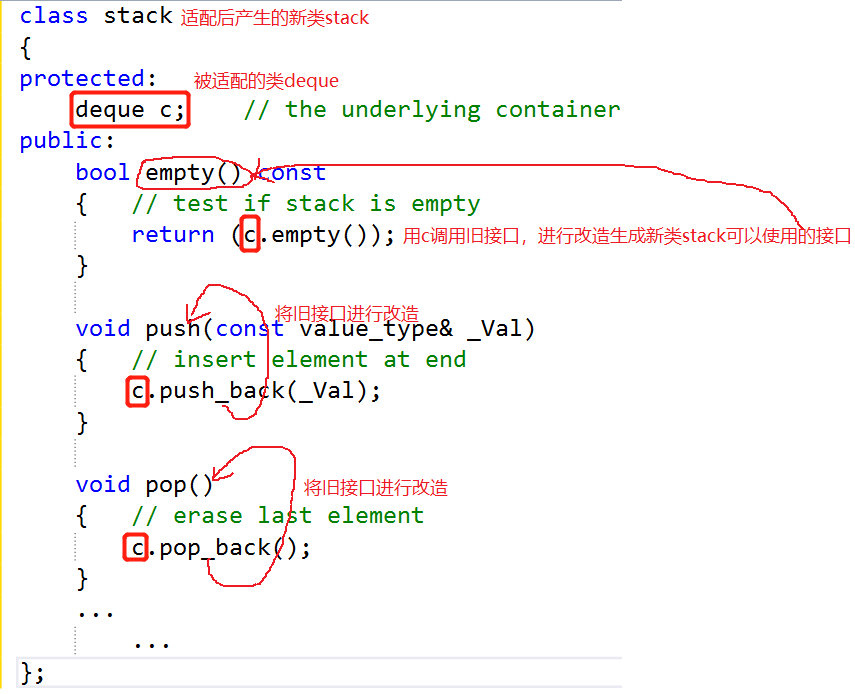
对已经存在的类，进行接口的改造，衍生出一个新类🡪🡪新类所有的接口，都是通过调用已经存在的类的接口实现的，例如：对deque进行适配，产生stack和queue。



应用场景：

STL源码剖析中，有三种适配器：容器适配器，迭代器适配器，仿函数适配器

**废话不多说，直接看STL源码部分stack：**



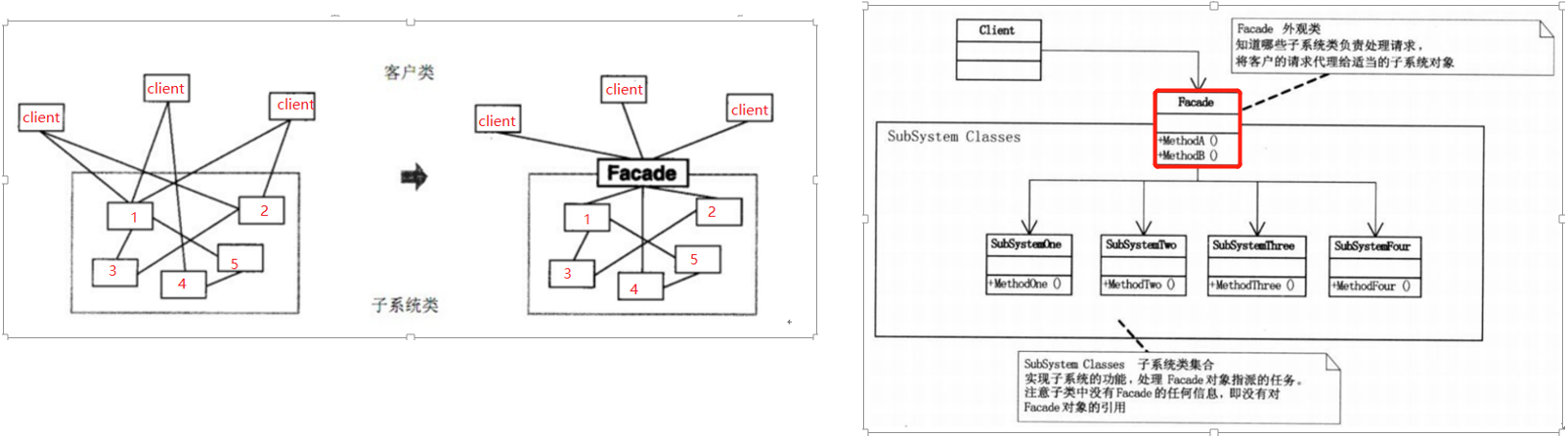
# 门面（外观）模式Facade

子系统内部和外部(client)之间的解耦合（用户只需要和Facede交互）

**解决的问题：**客户类不需要与子系统中的类直接打交道，而是通过Façade提供接口与子系统中的类打交道🡪🡪客户端不直到子系统中类的实现细节，只调用Façade提供的接口即可🡪🡪使得client与子系统中的类解耦合，系统维护更加简单。

**举例**：大力想组装电脑，如果自己组装电脑，那么他需要和cpu，主板，硬盘等各个厂商直接打交道，自己动手，丰衣足食；但是他嫌麻烦，因此大力直接找到组装电脑的厂家(facade)，只和组装电脑的厂家打交道，让厂家去和cpu，主板，硬盘等各个厂商打交道🡪🡪大大节省了时间。

**举例：**东东找炒股公司炒股





# 中介者模式

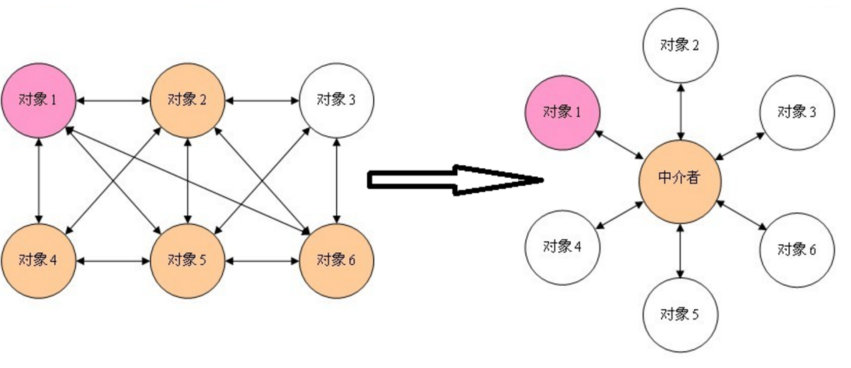
概念：用一个中介者Mediator对象，封装一系列的对象交互；使得对象之间不用强耦合

**实现技巧：Mediator有交互类的指针**

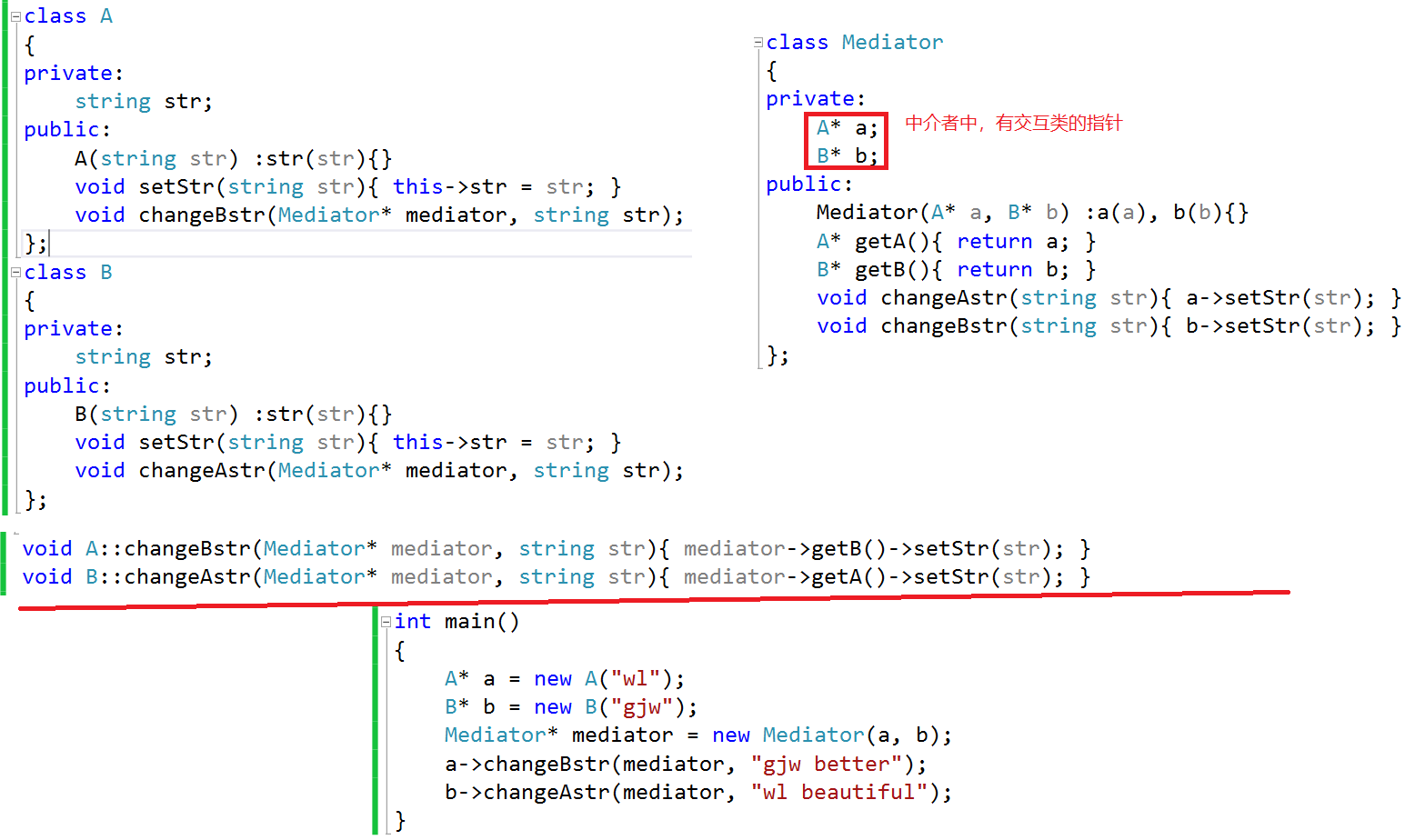
举例1：类A和类B交互，如A类要修改B类的内容，B类也要修改A类的内容

解决方案1：A类中组合B，B类中组合A，两个类可以直接修改🡪但是这种强耦合关系带来维护上的困难，尤其是在后来系统扩展时，交互类的个数增加并且各个类之间的交互变得复杂时，越来越力不从心

解决方案2：A类和B类都与中介类Mediator打交道，Mediator类中包含A和B



举例2：QQ和微信平台的使用——年轻人在进行聊天时，根本不用直到聊天工具内部子系统的设计，它们只需要直到如何使用聊天平台设计的接口(Facade)就可以了，至于剩下的消息传输的工作全部交给聊天平台。

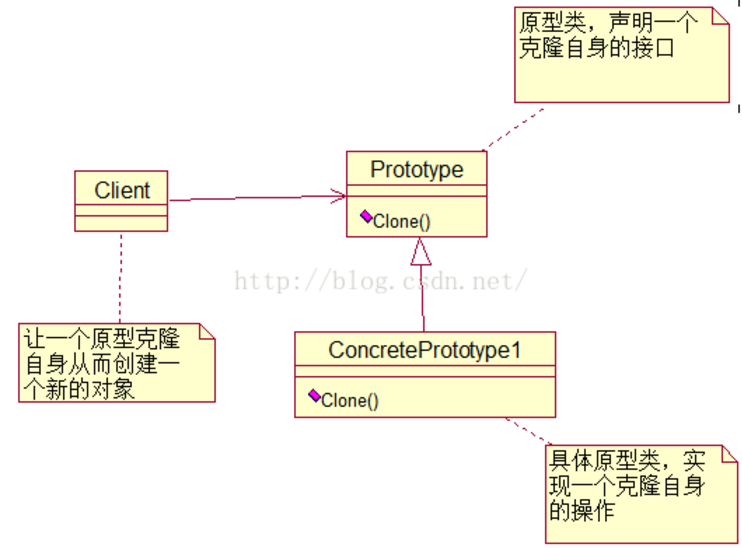


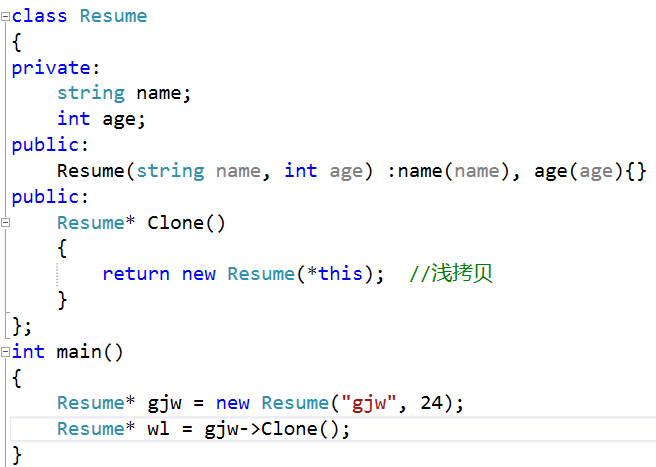
# 原型模式

原型模式，本质上就是实现一个Clone函数，使得能用一个原型对象克隆出另一个对象。

**注意：拷贝构造函数必须写对**

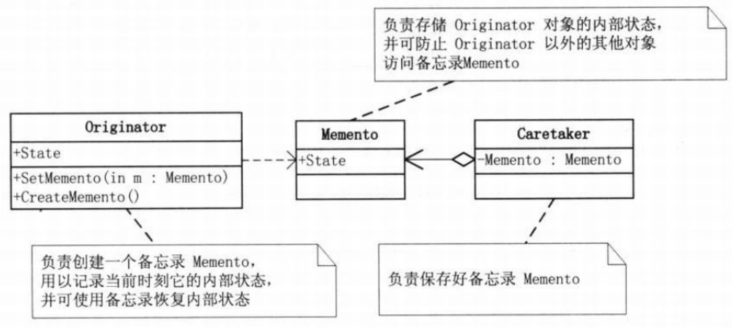
**举例：简历复制**





# 备忘录模式

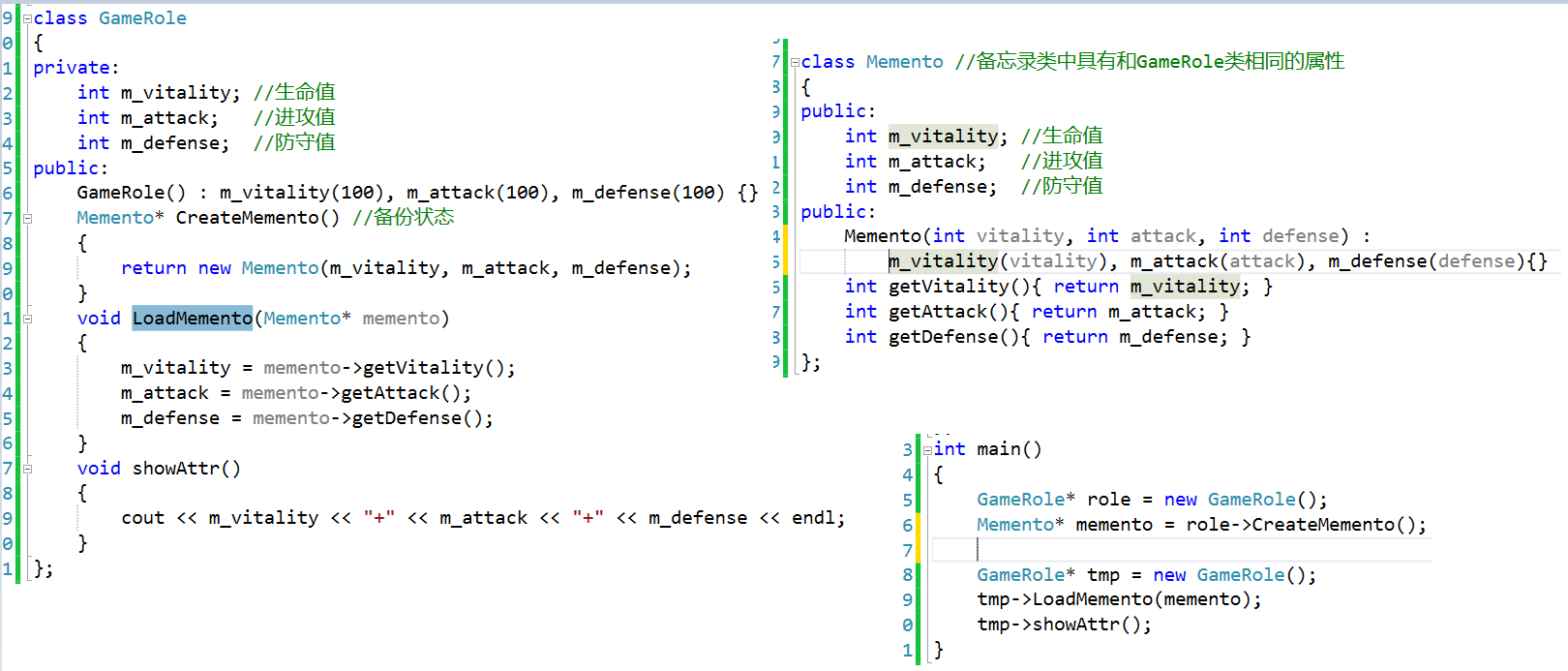
在不破坏封装性的前提下，捕获一个对象的内部状态，并在对象之外保存这个状态。这样以后就可将该对象恢复到原先的保存状态



**备忘录类：**必须具有Originator的所有的属性变量

**Originator类：**具有CreateMemento和LoadMemento接口

**为了简单理解：该程序，暂时不考虑备忘录管理者，只包括Original和Memento**



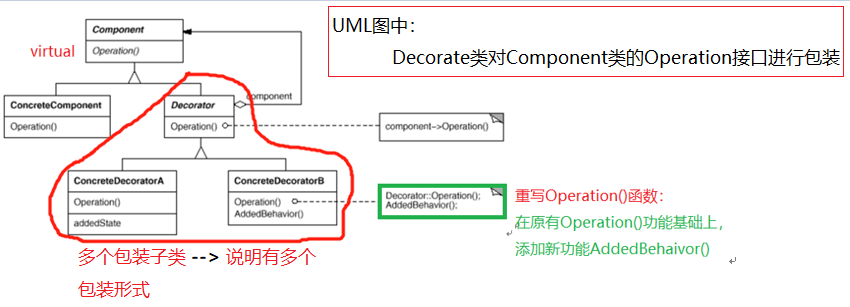
# 装饰模式

**核心：接口（函数）包装**

使用装饰者模式，可以动态的给**一个对象（而不是给整个类）**添加一些额外的职责。

**实现注意点：接口一致性**

**装饰对象的接口**必须与**它所装饰的类的接口**是一致的。体现了“装饰”过程不能改变对象的“本质”。🡪 因此只是对原有的接口函数进行功能的扩展，如：茶🡪柠檬茶🡪苹果柠檬茶，只是在茶的口味上进行包装，本质上是对口味这个接口进行功能扩展。





# 享元模式Flyweight

**定义：**对大量细粒度对象进行共享

**常见的例子：**在word文档中，有大量的文字，每个文字都是一个对象

**需求：修改word文字对象的字体格式**

**解决方案1：**给每一个文字对象都创建一个字体格式对象，供文字对象改变字体格式使用。这是十分占用内存资源的：试想一下，一篇文章又成千上万的文字，那么就对应有同样个数的字体对象，这是多么庞大的开销啊。

**思考：**由于字体格式的操作都是一样的，那为啥不让所有的文字对象都使用同一个字体格式对象呢？

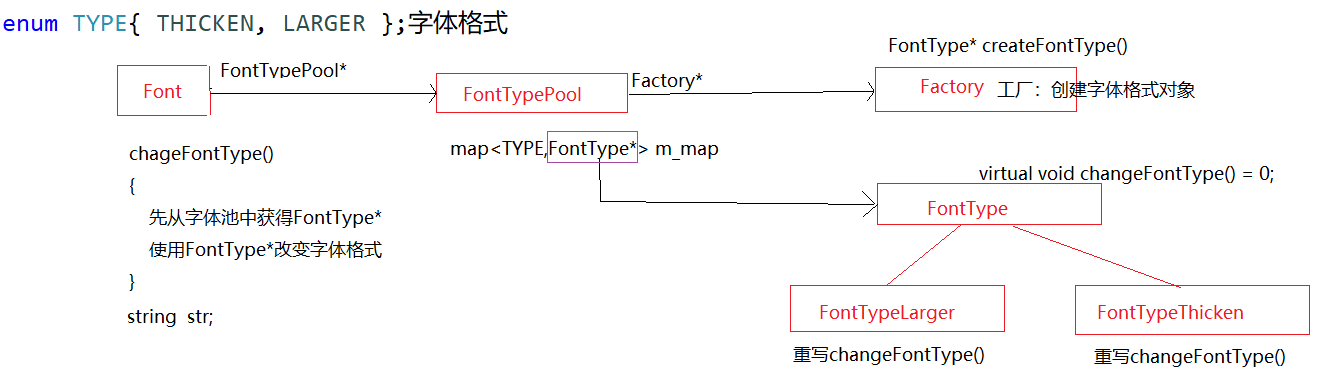
**解决方案2：**将字体格式类变成一个共享资源，让每个文本对象都使用这同一个字体格式对象，那么无论在多的字体出现，都只对应一个字体格式对象，这样的话，节省了巨大的开销。

**实现技巧：**

Flyweight功能对象类：实现享元类的功能

FactoryFlyweight类：创建享元对象

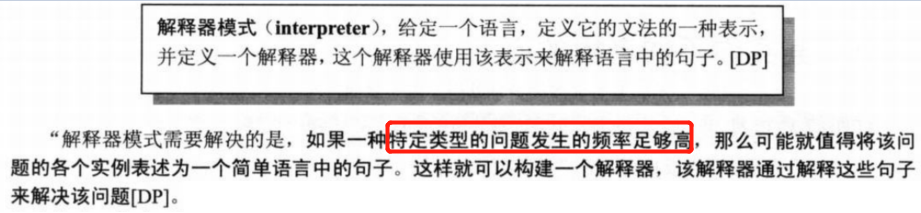
FlyweightPool：维护多个不同的享元对象的池子



**代码实现：享元模式，简单工厂模式，享元池（管理每种享元对象）**



# 解释器模式



# 迭代器模式

顺序访问集合中的每一个元素，而又无需暴露该对象的内部表示

**常见：**STL中的iterator