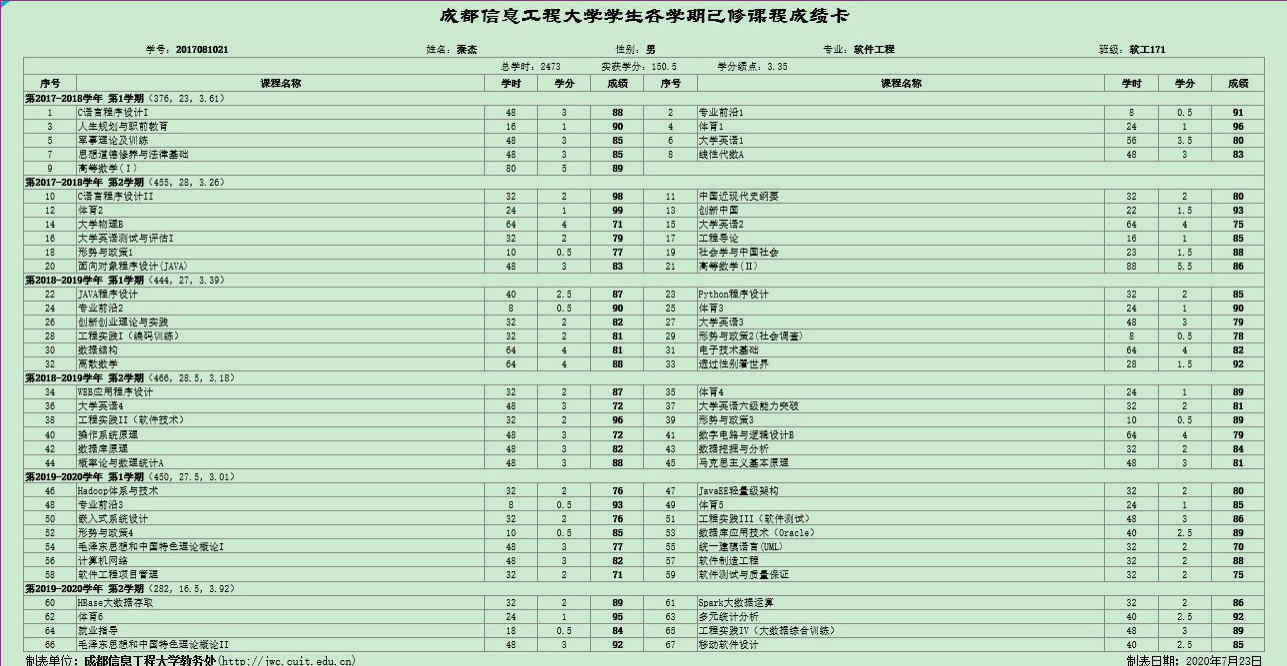
矩阵



# 遍历

## 前序遍历

## 中序遍历

## 后续遍历

# 搜索

## 深度优先搜索

深度优先搜索用栈（stack）来实现

深度优先遍历图的方法是，从图中某顶点v出发：

（1）访问顶点v；

（2）依次从v的未被访问的邻接点出发，对图进行[深度优先遍历](https://baike.baidu.com/item/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E9%81%8D%E5%8E%86)；直至图中和v有路径相通的顶点都被访问；

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行[深度优先遍历](https://baike.baidu.com/item/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E9%81%8D%E5%8E%86)，直到图中所有顶点均被访问过为止。　当然，当人们刚刚掌握深度优先搜索的时候常常用它来走迷宫.事实上我们还有别的方法，那就是[广度优先搜索](https://baike.baidu.com/item/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2)(BFS).

## 广度优先搜索

深度优先搜索用栈（stack）来实现，整个过程可以想象成一个倒立的树形：

1、把根节点压入栈中。

2、每次从栈中弹出一个元素，搜索所有在它下一级的元素，把这些元素压入栈中。并把这个元素记为它下一级元素的前驱。

3、找到所要找的元素时结束程序。

4、如果遍历整个树还没有找到，结束程序。

广度优先搜索使用队列（queue）来实现，整个过程也可以看做一个倒立的树形：

1、把根节点放到队列的末尾。

2、每次从队列的头部取出一个元素，查看这个元素所有的下一级元素，把它们放到队列的末尾。并把这个元素记为它下一级元素的前驱。

3、找到所要找的元素时结束程序。

4、如果遍历整个树还没有找到，结束程序。 [1]

# 查找

## 二分查找

每个元素的平均查找长度为：O(log2n)

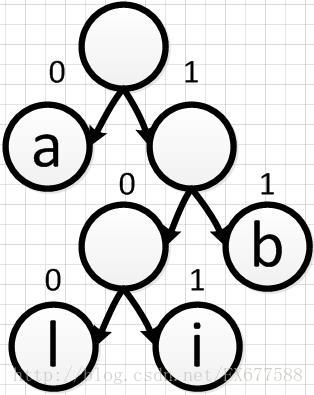
二分查找的时间复杂度 O(Log n)

# 编码

## 哈夫曼编码

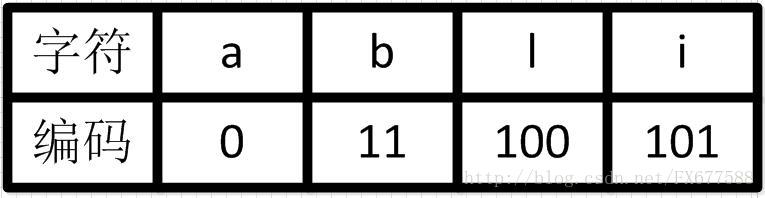
　　这题是考察哈夫曼的编码方式，它是根据字符出现频率构建的带权重二叉树确定每个字符编码的。首先我们统计“alibaba”各个字符出现频率：a-3，b-2，l-1，i-1。由出现的频率我们有以下哈夫曼二叉树：

这里写图片描述



　　对应每个字符编码为：

这里写图片描述



　　所以最终“alibaba”整个字符串的编码为0 100 101 11 0 11 0。也就是说该字符串二进制哈夫曼编码位数为13位。

# 排序

## 拓扑排序

对一个[有向无环图](https://baike.baidu.com/item/%E6%9C%89%E5%90%91%E6%97%A0%E7%8E%AF%E5%9B%BE/10972513)(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边<u,v>∈E(G)，则u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。简单的说，由某个集合上的一个[偏序](https://baike.baidu.com/item/%E5%81%8F%E5%BA%8F/2439087)得到该集合上的一个[全序](https://baike.baidu.com/item/%E5%85%A8%E5%BA%8F/10577699)，这个操作称之为拓扑排序。

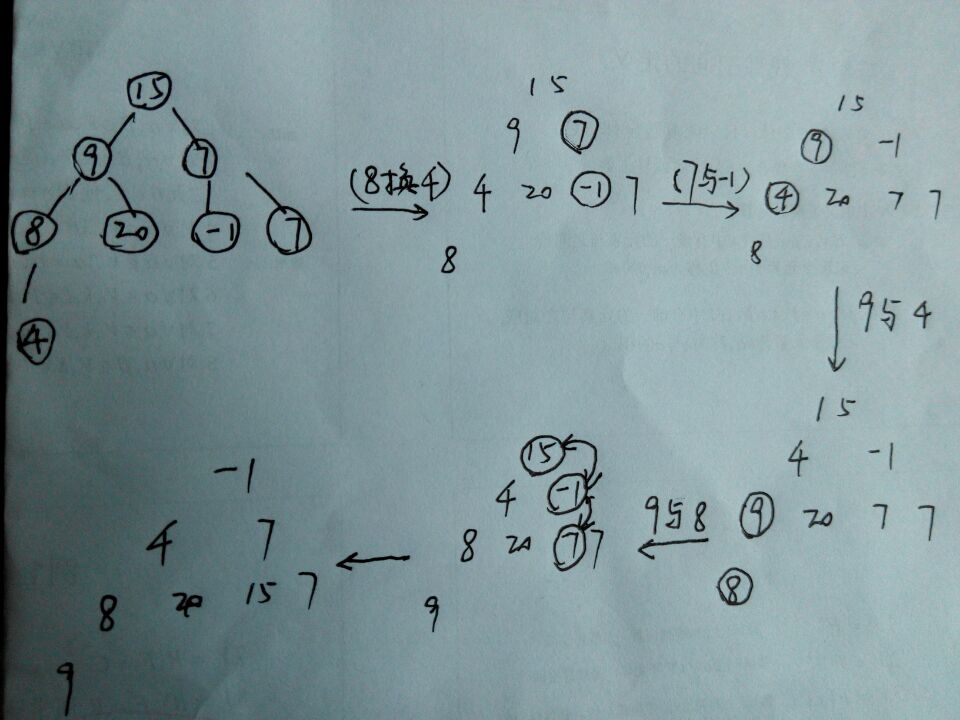
拓扑序列是顶点活动网中将活动按发生的先后次序进行的一种排列。

该排列满足：如果图中有一条从u到v的路径，则顶点v必须出现在顶点u之后。找出顶点活动网中的拓扑序列称“拓扑排序”

## 堆排序

O (nlgn) 堆排序中建堆过程的时间复杂度O（n） top-k问题，常用堆排序

堆是一个近似[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91)的结构，并同时满足**堆积的性质**：即子结点的键值或索引总是小于（或者大于）它的父节点。



首先根据序列构建一个完全二叉树。

在完全二叉树的基础上，从最后一个非叶结点开始调整：比较三个元素的大小–自己，它的左孩子，右孩子。分为三种情况:

自己最大，不用调整

左孩子最大，交换该非叶结点与其左孩子的值，并考察以左孩子为根的子树是否满足大顶堆的要求，不满足递归向下处理

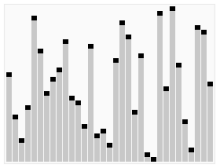
右孩子最大，交换该非叶结点与其右孩子的值，并考察以右孩子为根的子树是否满足大顶堆的要求，不满足递归向下处理

## 快速排序

O (nlogn)~O(n平方) 无序时是最快的内部排序算法

是对[冒泡排序](https://baike.baidu.com/item/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F/4602306)的一种改进

设要排序的[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)是A[0]……A[N-1]，首先任意选取一个数据（通常选

[](https://baike.baidu.com/pic/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95/369842/0/b7003af33a87e950707fdf2110385343fbf2b416?fr=lemma&ct=single)快排图

用数组的第一个数）作为关键数据，然后将所有比它小的数都放到它左边，所有比它大的数都放到它右边，这个过程称为一趟快速排序。值得注意的是，快速排序不是一种稳定的[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)，也就是说，多个相同的值的相对位置也许会在算法结束时产生变动。 [1] 

一趟快速排序的算法是： [1]

1）设置两个变量i、j，[排序](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F)开始的时候：i=0，j=N-1； [1]

2）以第一个数组元素作为关键数据，赋值给**key**，即**key**=A[0]； [1]

3）从j开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于**key**的值A[j]，将A[j]和A[i]的值交换； [1]

4）从i开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于**key**的A[i]，将A[i]和A[j]的值交换； [1]

5）重复第3、4步，直到i=j； (3,4步中，没找到符合条件的值，即3中A[j]不小于**key**,4中A[i]不大于**key**的时候改变j、i的值，使得j=j-1，i=i+1，直至找到为止。找到符合条件的值，进行交换的时候i， j指针位置不变。另外，i==j这一过程一定正好是i+或j-完成的时候，此时令循环结束）。 [1]

## 归并排序

O(n log n)

归并操作(merge)，也叫归并算法，指的是将两个顺序序列合并成一个顺序序列的方法。

如　设有数列{6，202，100，301，38，8，1}

初始状态：6,202,100,301,38,8,1

第一次归并后：{6,202},{100,301},{8,38},{1}，比较次数：3；

第二次归并后：{6,100,202,301}，{1,8,38}，比较次数：4；

第三次归并后：{1,6,8,38,100,202,301},比较次数：4；

总的比较次数为：3+4+4=11；

逆序数为14；

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 最好时间复杂度 | 最坏时间复杂度 | 平均时间复杂度 | 空间复杂度 | 稳定性 |
| 传统归并排序 | O(*n*log*n*) | O(*n*log*n*) | O(*n*log*n*) | T(n) | 稳定 |
| 改进归并排序 [1] | O(*n*) | O(*n*log*n*) | O(*n*log*n*) | T(n) | 稳定 |
| TimSort | O(*n*) | O(*n*log*n*) | O(*n*log*n*) | T(n) | 稳定 |

## 拓扑排序

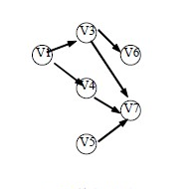
O（n+e）

对一个[有向无环图](https://baike.baidu.com/item/%E6%9C%89%E5%90%91%E6%97%A0%E7%8E%AF%E5%9B%BE/10972513)(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边<u,v>∈E(G)，则u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。简单的说，由某个集合上的一个[偏序](https://baike.baidu.com/item/%E5%81%8F%E5%BA%8F/2439087)得到该集合上的一个[全序](https://baike.baidu.com/item/%E5%85%A8%E5%BA%8F/10577699)，这个操作称之为拓扑排序。

由AOV网构造拓扑序列的拓扑排序算法主要是循环执行以下两步，直到不存在入度为0的顶点为止。

(1) 选择一个入度为0的顶点并输出之；

(2) 从网中删除此顶点及所有[出边](https://baike.baidu.com/item/%E5%87%BA%E8%BE%B9)。

[](https://baike.baidu.com/pic/%E6%8B%93%E6%89%91%E6%8E%92%E5%BA%8F/5223807/0/adaf2edda3cc7cd9565490a03401213fb80e914a?fr=lemma&ct=single)

循环结束后，若输出的顶点数小于网中的顶点数，则输出“有[回路](https://baike.baidu.com/item/%E5%9B%9E%E8%B7%AF/35792)”信息，否则输出的顶点序列就是一种拓扑[序列](https://baike.baidu.com/item/%E5%BA%8F%E5%88%97/1302588)。 [2]

## 希尔排序

O（nlogn）～O（n2）

希尔排序(Shell's Sort)是[插入排序](https://baike.baidu.com/item/%E6%8F%92%E5%85%A5%E6%8E%92%E5%BA%8F/7214992)的一种又称“缩小增量排序”（Diminishing Increment Sort），是直接插入[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95/5399605)的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因D.L.Shell于1959年提出而得名。

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95/209025)便终止。 [1]

比较相隔较远距离（称为[增量](https://baike.baidu.com/item/%E5%A2%9E%E9%87%8F)）的数，使得数移动时能跨过多个元素，则进行一次比较就可能消除多个元素交换。D.L.shell于1959年在以他名字命名的[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)中实现了这一思想。算法先将要排序的一组数按某个[增量](https://baike.baidu.com/item/%E5%A2%9E%E9%87%8F)d分成若干组，每组中记录的下标相差d.对每组中全部元素进行排序，然后再用一个较小的增量对它进行，在每组中再进行排序。当[增量](https://baike.baidu.com/item/%E5%A2%9E%E9%87%8F)减到1时，整个要排序的数被分成一组，排序完成。

一般的初次取序列的一半为[增量](https://baike.baidu.com/item/%E5%A2%9E%E9%87%8F)，以后每次减半，直到增量为1。

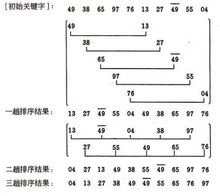
**给定实例的shell排序的排序过程**

假设待排序文件有10个记录，其关键字分别是：

49，38，65，97，76，13，27，49，55，04。

[增量](https://baike.baidu.com/item/%E5%A2%9E%E9%87%8F)序列的取值依次为：

5，2，1

[](https://baike.baidu.com/pic/%E5%B8%8C%E5%B0%94%E6%8E%92%E5%BA%8F/3229428/0/dbb44aed2e738bd4423393fead8b87d6267ff9eb?fr=lemma&ct=single)希尔排序

## 插入排序

O(N^(1~2)) 直接插入排序是数据越有序越快，最快时间复杂度可达到O(n) .

插入排序，一般也被称为直接插入排序。对于少量元素的排序，它是一个有效的算法 [1]  。插入排序是一种最简单的[排序](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F/1066239)方法，它的基本思想是将一个记录插入到已经排好序的有序表中，从而一个新的、记录数增1的有序表。在其实现过程使用双层循环，外层循环对除了第一个元素之外的所有元素，内层循环对当前元素前面有序表进行待插入位置查找，并进行移动 [2]  。

插入排序的工作方式像许多人排序一手扑克牌。开始时，我们的左手为空并且桌子上的牌面向下。然后，我们每次从桌子上拿走一张牌并将它插入左手中正确的位置。为了找到一张牌的正确位置，我们从右到左将它与已在手中的每张牌进行比较。拿在左手上的牌总是排序好的，原来这些牌是桌子上牌堆中顶部的牌 [1]  。

插入排序是指在待排序的元素中，假设前面n-1(其中n>=2)个数已经是排好顺序的，现将第n个数插到前面已经排好的序列中，然后找到合适自己的位置，使得插入第n个数的这个序列也是排好顺序的。按照此法对所有元素进行插入，直到整个序列排为有序的过程，称为插入排序 [3]  。

## 冒泡排序

O(n²)

冒泡排序（Bubble Sort），是一种[计算机科学](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)领域的较简单的[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95/5399605)。

它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序（如从大到小、首字母从Z到A）错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。

这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。

冒泡[排序算法](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95/5399605)的原理如下： [1]

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。 [1]
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。 [1]
3. 针对所有的元素重复以上的步骤，除了最后一个。 [1]
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。 [1]

## 直接选择排序

O(n平方)

直接选择排序(Straight Select Sorting) 也是一种简单的排序方法，它的基本思想是：第一次从R[0]~R[n-1]中选取最小值，与R[0]交换，第二次从R[1]~R[n-1]中选取最小值，与R[1]交换，....，第i次从R[i-1]~R[n-1]中选取最小值，与R[i-1]交换，.....，第n-1次从R[n-2]~R[n-1]中选取最小值，与R[n-2]交换，总共通过n-1次，得到一个按排序码从小到大排列的有序序列。

## 基数排序

基数排序（radix sort）属于“分配式排序”（distribution sort），又称“桶子法”（bucket sort）或bin sort，顾名思义，它是透过键值的部份资讯，将要排序的[元素分配](https://baike.baidu.com/item/%E5%85%83%E7%B4%A0%E5%88%86%E9%85%8D/2107419)至某些“桶”中，藉以达到排序的作用，基数排序法是属于稳定性的排序，其[时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/1894057)为O (nlog(r)m)，其中r为所采取的基数，而m为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。

### 第一步

以LSD为例，假设原来有一串数值如下所示：

73, 22, 93, 43, 55, 14, 28, 65, 39, 81

首先根据个位数的数值，在走访数值时将它们分配至编号0到9的桶子中：

0

1 81

2 22

3 73 93 43

4 14

5 55 65

6

7

8 28

9 39

### 第二步

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

81, 22, 73, 93, 43, 14, 55, 65, 28, 39

接着再进行一次分配，这次是根据十位数来分配：

0

1 14

2 22 28

3 39

4 43

5 55

6 65

7 73

8 81

9 93

### 第三步

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

14, 22, 28, 39, 43, 55, 65, 73, 81, 93

这时候整个数列已经排序完毕；如果排序的对象有三位数以上，则持续进行以上的动作直至最高位数为止。

LSD的基数排序适用于位数小的数列，如果位数多的话，使用MSD的效率会比较好。MSD的方式与LSD相反，是由高位数为基底开始进行分配，但在分配之后并不马上合并回一个[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)中，而是在每个“桶子”中建立“子桶”，将每个桶子中的数值按照下一数位的值分配到“子桶”中。在进行完最低位数的分配后再合并回单一的[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)中。

## 内排序

内排序是被排序的数据元素全部存放在计算机内存中的排序算法。

内部排序是指待排的记录全部在内存中完成排序的过程，内部排序也称为内排序。若待排序记录的数量庞大，在排序的过程中需要使用到外部存储介质如磁盘等，这种涉及内外存储器数据交换的排序过程称为外部排序，又称为外排序。内排序是外排序的基础，外排序算法的原理和内排序算法的原理在很多方面都类似，但因内存的读写速度与外存的读写速度存在很大差别，因而实际操作中仍有不同。 [1]

## 总结



# 数据库范式

## [第一范式](https://baike.baidu.com/item/%E7%AC%AC%E4%B8%80%E8%8C%83%E5%BC%8F/3193590)（1NF）是指数据库表的每一列都是不可分割的基本[数据项](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%A1%B9/3227309)

## 第二范式(Second Normal Form,2nd NF)是指每个表必须有[主关键字](https://baike.baidu.com/item/%E4%B8%BB%E5%85%B3%E9%94%AE%E5%AD%97/1239455)(Primary key),其他数据元素与主关键字一一对应

## 第三范式(Third Normal Form,3rd NF)就是指表中的所有数据元素不但要能唯一地被主关键字所标识,而且它们之间还必须相互独立,不存在其他的函数关系

# 数据库关系规范化中异常

插入失败：该插入的没插入；

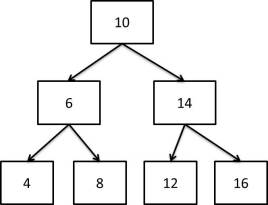
插入异常：不该插入的被插入；

删除失败：该删除的没删除；

删除异常：不该删除的被删除；

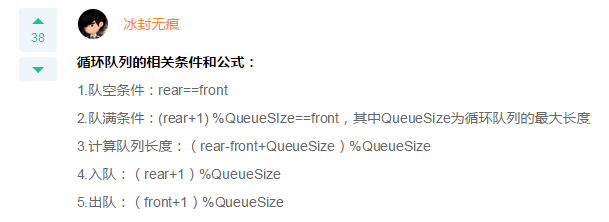
# 数据结构

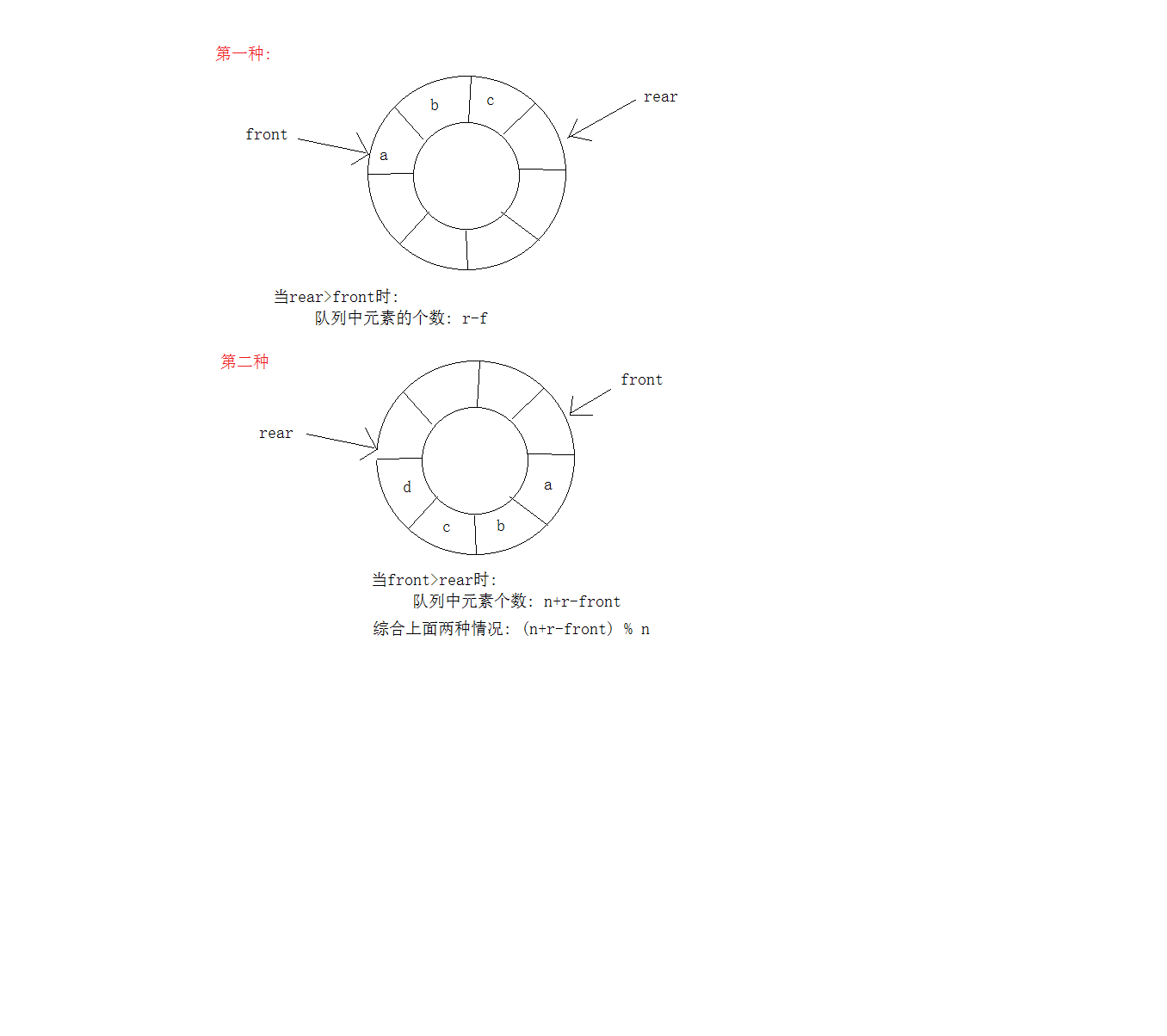
## 二叉链表

二叉链表是树的二叉链表实现方式。

## 队列

### 循环队列





### 优先队列

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （first in, largest out）的行为特征。通常采用堆数据结构来实现。

## 树

**度：节点有几个分叉就有几个度**

**任一棵树中，节点总数=总分支数目+1**

因为任一棵树中，节点总数=总分支数目+1

14+15+14+5+10+叶节点个数=14\*5+15\*4+14\*3+5\*2+10\*1+1

叶节点个数为135

### 最小生成树

最小生成树其实是**最小权重生成树**的简称

最小权重生成树不唯一

### 二叉树

二叉树的性质

**性质1**：在二叉树的第i层上至多有2i-1 个结点（i≥1）。（数学归纳法可证）

**性质2**：深度为k的二叉树最多有2k-1个结点（k≥1）。（由性质1，通过等比数列求和可证）

**性质3**：一棵二叉树的叶子结点数为n0 ，度为2的结点 数为n2 ，则n0 = n2 + 1。

**性质4**：具有n个结点的完全二叉树的深度为floor(log2n) + 1 。

**性质5**：如果对一棵有n个结点的完全二叉树（其深度为floor(log2n) + 1 ）的结点按层序编号，则对任一结点i（1≤i≤n）有：

（1） 如果i = 1，则结点i是二叉树的根，无双亲；如果i > 1，则其双亲PARENT(i)是结点 floor((i)/2)。

（2）如果2i > n，则结点i无左孩子；否则其左孩子LCHILD(i)是结点2i。

（3）如果2i + 1 > n，则结点i无右孩子；否则其右孩子RCHILD(i)是结点2i + 1

### 哈夫曼树

给定N个权值作为N个[叶子结点](https://baike.baidu.com/item/%E5%8F%B6%E5%AD%90%E7%BB%93%E7%82%B9/3620239)，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

树的路径长度是从树根到树中每一结点的路径长度之和。在结点数目相同的二叉树中，完全二叉树的路径长度最短。

由权值分别为 3,8,6,2 的叶子生成一棵哈夫曼树，它的带权路径长度为 (      ) 。

根据哈夫曼树的定义画出哈夫曼树，得到带权路径长度为WPL = 2\*3+3\*3+6\*2+8\*1=35

构建方法：<https://blog.csdn.net/dyingstraw/article/details/88093064>

### 完全二叉树

N为总节点数

叶节点个数：(n+1)/2

深度log2n+1

### 线索二叉树

在[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91/1602879)的结点上加上线索的二叉树称为线索二叉树，对二叉树以某种遍历方式（如先序、中序、后序或层次等）进行遍历，使其变为线索二叉树的过程称为对二叉树进行线索化。 [1]

[离散数学](https://www.baidu.com/s?wd=%E7%A6%BB%E6%95%A3%E6%95%B0%E5%AD%A6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)里关于树的性质应该还记得吧？一棵n结点树包含n-1条边，而每个结点有两个指针域即总共2n个指针，减去表示边的指向关系（即左右子树）的n-1条边，剩下n+1条边即为线索。

## 图

### 无向图

无向图边数的两倍等于各顶点度数的总和。由于其他顶点的度均小于3，可以设它们的度都为2，设它们的数量是x，可列出这样的方程4\*3+3\*4+2\*x=16\*2，解得x=3。4+3+3=11，B正确。

度：顶点的度指的是与该顶点相关联的边的总数。

两个顶点相邻指的是该两个顶点之间有边连接，这条边依附于这两个顶点存在。

顶点数为*N*（*N*≥1）的无向完全图中不存在度为1的顶点

# 计算机网络

## Tcp/ip

TCP/IP传输协议，即传输控制/网络协议，也叫作网络通讯协议。它是在网络的使用中的最基本的通信协议。TCP/IP传输协议对互联网中各部分进行通信的标准和方法进行了规定。并且，TCP/IP传输协议是保证网络数据信息及时、完整传输的两个重要的协议。TCP/IP传输协议是严格来说是一个四层的体系结构，应用层、传输层、网络层和数据链路层都包含其中。

TCP/IP协议是Internet最基本的协议,其中应用层的主要协议有[Telnet](https://baike.baidu.com/item/Telnet/810597)、[FTP](https://baike.baidu.com/item/FTP/13839)、[SMTP](https://baike.baidu.com/item/SMTP/175887)等，是用来接收来自传输层的数据或者按不同应用要求与方式将数据传输至传输层；

1. 传输层的主要协议有[UDP](https://baike.baidu.com/item/UDP/571511)（用户数据包协议）、TCP（传输控制协议），是使用者使用平台和计算机信息网内部数据结合的通道，可以实现数据传输与数据共享；
2. 网络层的主要协议有ICMP（Internet控制报文协议。它是TCP/IP协议簇的一个子协议，用于在IP主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达）、IP（IP是Internet Protocol（网际互连协议）的缩写，是TCP/IP体系中的网络层协议）、IGMP（Internet 组管理协议称为IGMP协议（Internet Group Management Protocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间），主要负责网络中数据包的传送等；
3. 而网络访问层，也叫网路接口层或数据链路层，主要协议有ARP、[RARP](https://baike.baidu.com/item/RARP/610685)，主要功能是提供链路管理错误检测、对不同通信媒介有关信息细节问题进行有效处理等。
4. 应用层：应用层是TCP/IP协议的第一层，是直接为应用进程提供服务的。

（1）对不同种类的应用程序它们会根据自己的需要来使用应用层的不同协议，邮件传输应用使用了[SMTP](https://baike.baidu.com/item/SMTP/175887)协议、万维网应用使用了[HTTP](https://baike.baidu.com/item/HTTP/243074)协议、远程登录服务应用使用了有[TELNET](https://baike.baidu.com/item/TELNET/810597)协议。 [1]

（2）应用层还能加密、解密、格式化数据。 [1]

（3）应用层可以建立或解除与其他节点的联系，这样可以充分节省网络资源。 [1]

运输层：作为TCP/IP协议的第二层，运输层在整个TCP/IP协议中起到了中流砥柱的作用。且在运输层中，TCP和UDP也同样起到了中流砥柱的作用。 [1]

网络层：网络层在TCP/IP协议中的位于第三层。在TCP/IP协议中网络层可以进行网络连接的建立和终止以及IP地址的寻找等功能。 [1]

网络接口层：在TCP/IP协议中，网络接口层位于第四层。由于网络接口层兼并了[物理层](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%B1%82/4329158)和[数据链路层](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%93%BE%E8%B7%AF%E5%B1%82/4329290)所以，网络接口层既是传输数据的物理媒介，也可以为网络层提供一条准确无误的线路。 [1]

# Java基础

## Static执行顺序

、 Java中静态变量只能在类主体中定义，不能在方法中定义。 静态变量属于类所有而不属于方法。

      2、 静态块：用static申明，JVM加载类时执行，仅执行一次

　　      构造块：类中直接用{}定义，每一次创建对象时执行

             执行顺序优先级：静态块>main()>构造块>构造方法

     3、类的加载顺序

　　     (1) 父类静态对象和静态代码块

　　     (2) 子类静态对象和静态代码块

　　     (3) 父类非静态对象和非静态代码块

　　     (4) 父类构造函数

　　     (5) 子类 非静态对象和非静态代码块

　　     (6) 子类构造函数

## [sleep()和write()的简答题](https://www.cnblogs.com/Javafirst/archive/2011/02/22/1960347.html)

sleep是线程类（Thread）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用sleep不会释放对象锁。

wait是Object类的方法，对此对象调用wait方法导致本线程放弃对象锁，进入等待此对象的等待锁定池。只有针对此对象发出notify方法（或notifyAll）后，本线程才进入对象锁定池准备获得对象锁进入运行状态。

## 设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。简单说：

**模式：在某些场景下，针对某类问题的某种通用的解决方案。**

场景：项目所在的环境

问题：约束条件，项目目标等

解决方案：通用、可复用的设计，解决约束达到目标。

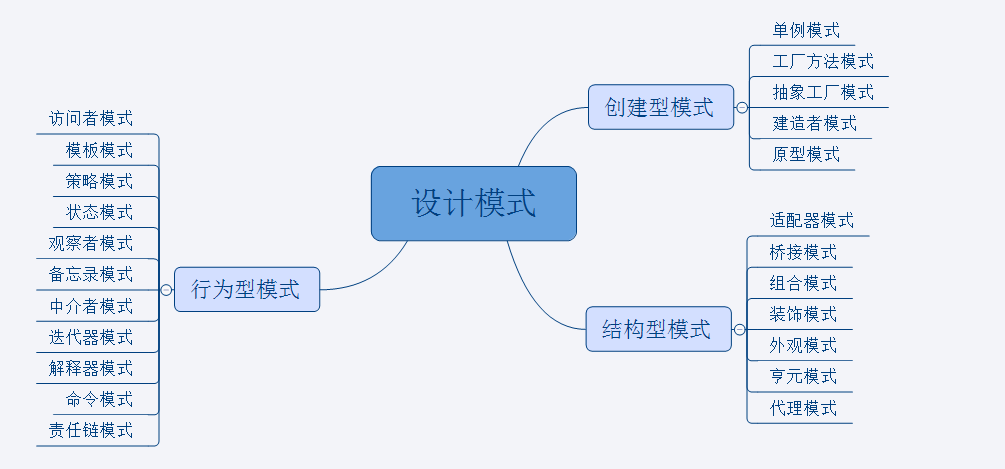
### 设计模式的三个分类

**创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。**

**结构型模式：把类或对象结合在一起形成一个更大的结构。**

**行为型模式：类和对象如何交互，及划分责任和算法。**

**如下图所示：**

****

### **各分类中模式的关键点**

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。

### **概说23种设计模式**

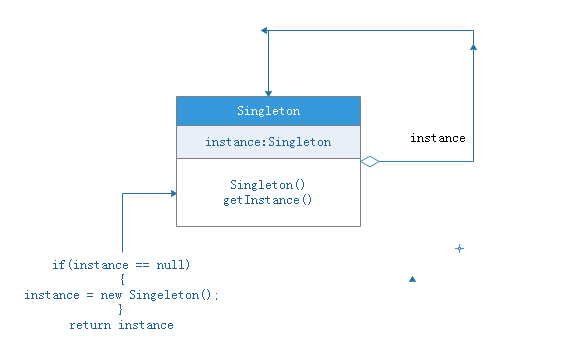
#### 单例模式

单例模式，它的定义就是确保某一个类只有一个实例，并且提供一个全局访问点。

单例模式具备典型的3个特点：1、只有一个实例。 2、自我实例化。 3、提供全局访问点。

 因此当系统中只需要一个实例对象或者系统中只允许一个公共访问点，除了这个公共访问点外，不能通过其他访问点访问该实例时，可以使用单例模式。

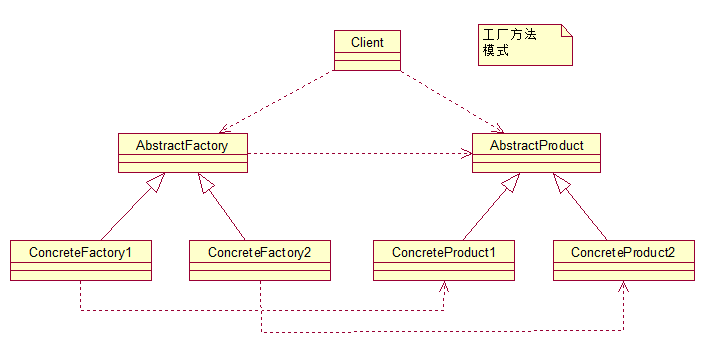
单例模式的主要优点就是节约系统资源、提高了系统效率，同时也能够严格控制客户对它的访问。也许就是因为系统中只有一个实例，这样就导致了单例类的职责过重，违背了“单一职责原则”，同时也没有抽象类，所以扩展起来有一定的困难。其UML结构图非常简单，就只有一个类，如下图：



#### 工厂方法模式

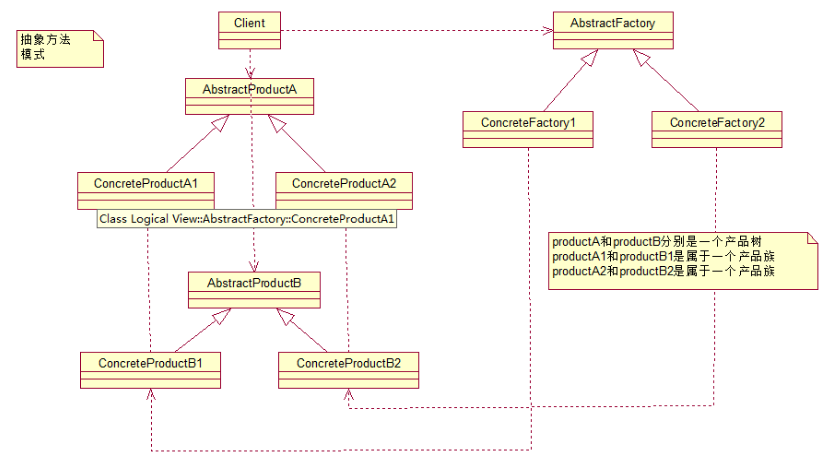
作为抽象工厂模式的孪生兄弟，工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，也就是说工厂方法模式让实例化推迟到子类。

工厂方法模式非常符合“开闭原则”，当需要增加一个新的产品时，我们只需要增加一个具体的产品类和与之对应的具体工厂即可，无须修改原有系统。同时在工厂方法模式中用户只需要知道生产产品的具体工厂即可，无须关系产品的创建过程，甚至连具体的产品类名称都不需要知道。虽然他很好的符合了“开闭原则”，但是由于每新增一个新产品时就需要增加两个类，这样势必会导致系统的复杂度增加。其UML结构图：



#### 抽象工厂模式

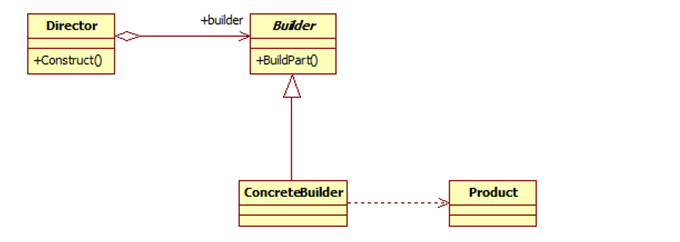
所谓抽象工厂模式就是提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类。他允许客户端使用抽象的接口来创建一组相关的产品，而不需要关系实际产出的具体产品是什么。这样一来，客户就可以从具体的产品中被解耦。它的优点是隔离了具体类的生成，使得客户端不需要知道什么被创建了，而缺点就在于新增新的行为会比较麻烦，因为当添加一个新的产品对象时，需要更加需要更改接口及其下所有子类。其UML结构图如下：



#### 建造者模式

对于建造者模式而已，它主要是将一个复杂对象的构建与表示分离，使得同样的构建过程可以创建不同的表示。适用于那些产品对象的内部结构比较复杂。

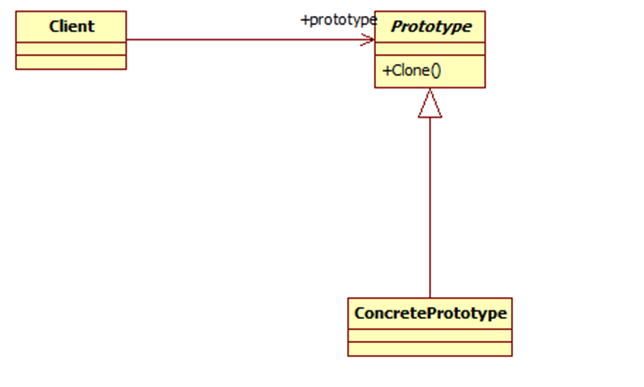
建造者模式将复杂产品的构建过程封装分解在不同的方法中，使得创建过程非常清晰，能够让我们更加精确的控制复杂产品对象的创建过程，同时它隔离了复杂产品对象的创建和使用，使得相同的创建过程能够创建不同的产品。但是如果某个产品的内部结构过于复杂，将会导致整个系统变得非常庞大，不利于控制，同时若几个产品之间存在较大的差异，则不适用建造者模式，毕竟这个世界上存在相同点大的两个产品并不是很多，所以它的使用范围有限。其UML结构图：



#### 原型模式

在我们应用程序可能有某些对象的结构比较复杂，但是我们又需要频繁的使用它们，如果这个时候我们来不断的新建这个对象势必会大大损耗系统内存的，这个时候我们需要使用原型模式来对这个结构复杂又要频繁使用的对象进行克隆。所以原型模式就是用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。

它主要应用与那些创建新对象的成本过大时。它的主要优点就是简化了新对象的创建过程，提高了效率，同时原型模式提供了简化的创建结构。UML结构图：

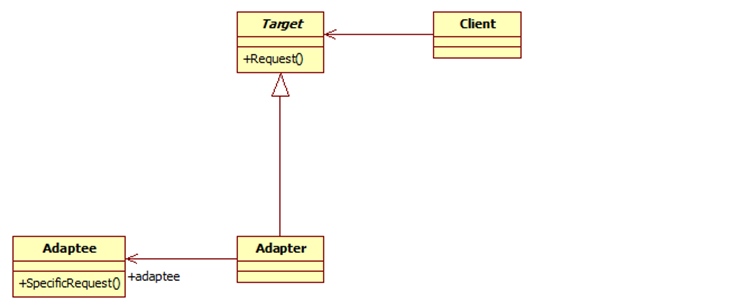


**模式结构**  
原型模式包含如下角色：  
Prototype：抽象原型类  
ConcretePrototype：具体原型类  
Client：客户类

6.适配器模式

在我们的应用程序中我们可能需要将两个不同接口的类来进行通信，在不修改这两个的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。

作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

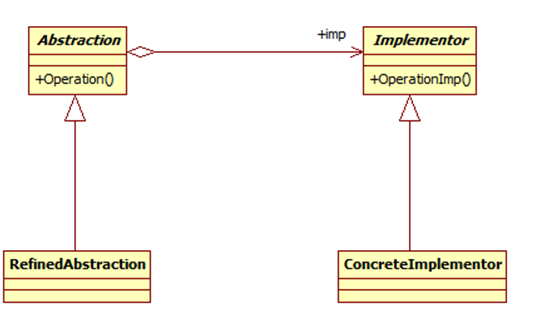


适配器模式包含如下角色：  
Target：目标抽象类  
Adapter：适配器类  
Adaptee：适配者类  
Client：客户类

7.桥接模式

如果说某个系统能够从多个角度来进行分类，且每一种分类都可能会变化，那么我们需要做的就是讲这多个角度分离出来，使得他们能独立变化，减少他们之间的耦合，这个分离过程就使用了桥接模式。所谓桥接模式就是讲抽象部分和实现部分隔离开来，使得他们能够独立变化。

桥接模式将继承关系转化成关联关系，封装了变化，完成了解耦，减少了系统中类的数量，也减少了代码量。

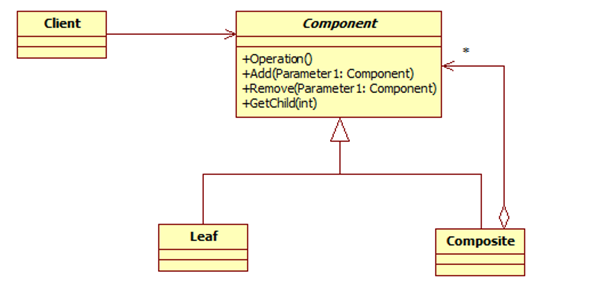


桥接模式包含如下角色：  
Abstraction：抽象类  
RefinedAbstraction：扩充抽象类  
Implementor：实现类接口  
ConcreteImplementor：具体实现类

8.组合模式

组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。它定义了如何将容器对象和叶子对象进行递归组合，使得客户在使用的过程中无须进行区分，可以对他们进行一致的处理。在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。

虽然组合模式能够清晰地定义分层次的复杂对象，也使得增加新构件也更容易，但是这样就导致了系统的设计变得更加抽象，如果系统的业务规则比较复杂的话，使用组合模式就有一定的挑战了。

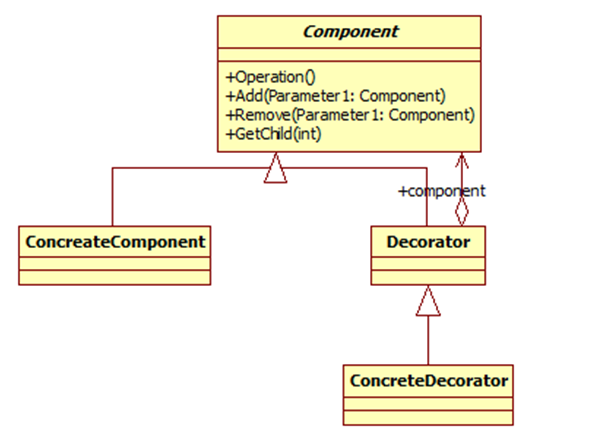


**模式结构**  
组合模式包含如下角色：  
Component: 抽象构件  
Leaf: 叶子构件  
Composite: 容器构件  
Client: 客户类

#### 装饰模式

我们可以通过继承和组合的方式来给一个对象添加行为，虽然使用继承能够很好拥有父类的行为，但是它存在几个缺陷：一、对象之间的关系复杂的话，系统变得复杂不利于维护。二、容易产生“类爆炸”现象。三、是静态的。在这里我们可以通过使用装饰者模式来解决这个问题。

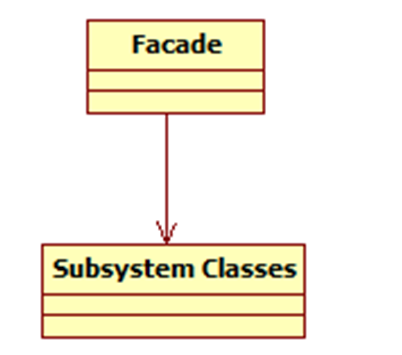
装饰者模式，动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更加有弹性的替代方案。虽然装饰者模式能够动态将责任附加到对象上，但是他会产生许多的细小对象，增加了系统的复杂度。



**模式结构**  
装饰模式包含如下角色：  
Component: 抽象构件  
ConcreteComponent: 具体构件  
Decorator: 抽象装饰类  
ConcreteDecorator: 具体装饰类

#### 外观模式

我们都知道类与类之间的耦合越低，那么可复用性就越好，如果两个类不必彼此通信，那么就不要让这两个类发生直接的相互关系，如果需要调用里面的方法，可以通过第三者来转发调用。外观模式非常好的诠释了这段话。外观模式提供了一个统一的接口，用来访问子系统中的一群接口。它让一个应用程序中子系统间的相互依赖关系减少到了最少，它给子系统提供了一个简单、单一的屏障，客户通过这个屏障来与子系统进行通信。通过使用外观模式，使得客户对子系统的引用变得简单了，实现了客户与子系统之间的松耦合。但是它违背了“开闭原则”，因为增加新的子系统可能需要修改外观类或客户端的源代码。



外观模式包含如下角色：  
Facade: 外观角色  
SubSystem:子系统角色

#### 亨元模式

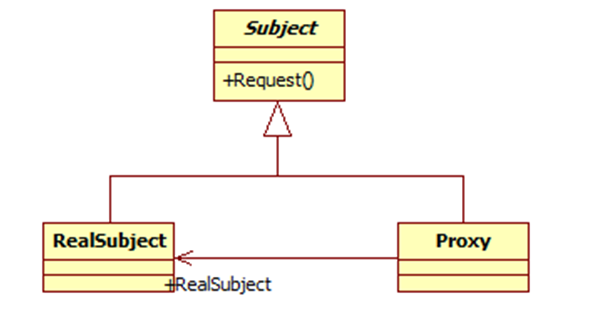
在一个系统中对象会使得内存占用过多，特别是那些大量重复的对象，这就是对系统资源的极大浪费。享元模式对对象的重用提供了一种解决方案，它使用共享技术对相同或者相似对象实现重用。享元模式就是运行共享技术有效地支持大量细粒度对象的复用。系统使用少量对象,而且这些都比较相似，状态变化小，可以实现对象的多次复用。这里有一点要注意：享元模式要求能够共享的对象必须是细粒度对象。享元模式通过共享技术使得系统中的对象个数大大减少了，同时享元模式使用了内部状态和外部状态，同时外部状态相对独立，不会影响到内部状态，所以享元模式能够使得享元对象在不同的环境下被共享。同时正是分为了内部状态和外部状态，享元模式会使得系统变得更加复杂，同时也会导致读取外部状态所消耗的时间过长。

享元模式包含如下角色：  
Flyweight: 抽象享元类  
ConcreteFlyweight: 具体享元类  
UnsharedConcreteFlyweight: 非共享具体享元类  
FlyweightFactory: 享元工厂类

#### 代理模式

 代理模式就是给一个对象提供一个代理，并由代理对象控制对原对象的引用。它使得客户不能直接与真正的目标对象通信。代理对象是目标对象的代表，其他需要与这个目标对象打交道的操作都是和这个代理对象在交涉。

代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了的作用和保护了目标对象的，同时也在一定程度上面减少了系统的耦合度。

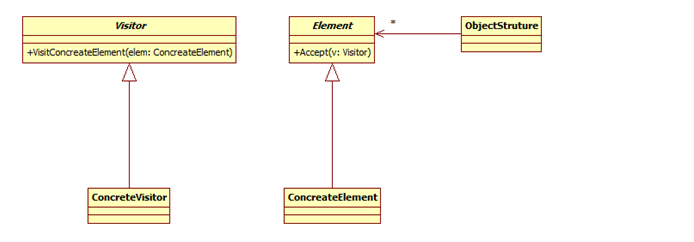


代理模式包含如下角色：  
ο Subject: 抽象主题角色  
ο Proxy: 代理主题角色  
ο RealSubject: 真实主题角色

#### 访问者模式

访问者模式俗称23大设计模式中最难的一个。除了结构复杂外，理解也比较难。在我们软件开发中我们可能会对同一个对象有不同的处理，如果我们都做分别的处理，将会产生灾难性的错误。对于这种问题，访问者模式提供了比较好的解决方案。访问者模式即表示一个作用于某对象结构中的各元素的操作，它使我们可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

访问者模式的目的是封装一些施加于某种数据结构元素之上的操作，一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。为不同类型的元素提供多种访问操作方式，且可以在不修改原有系统的情况下增加新的操作方式。同时我们还需要明确一点那就是访问者模式是适用于那些数据结构比较稳定的，因为他是将数据的操作与数据结构进行分离了，如果某个系统的数据结构相对稳定，但是操作算法易于变化的话，就比较适用适用访问者模式，因为访问者模式使得算法操作的增加变得比较简单了。



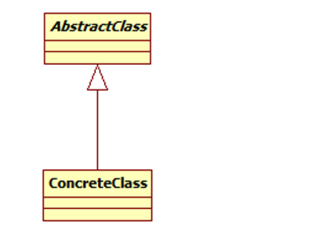
访问者模式包含如下角色：  
Vistor: 抽象访问者  
ConcreteVisitor: 具体访问者  
Element: 抽象元素  
ConcreteElement: 具体元素   
ObjectStructure: 对象结构

#### 模板模式

有些时候我们做某几件事情的步骤都差不多，仅有那么一小点的不同，在软件开发的世界里同样如此，如果我们都将这些步骤都一一做的话，费时费力不讨好。所以我们可以将这些步骤分解、封装起来，然后利用继承的方式来继承即可，当然不同的可以自己重写实现嘛！这就是模板方法模式提供的解决方案。

所谓模板方法模式就是在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

模板方法模式就是基于继承的代码复用技术的。在模板方法模式中，我们可以将相同部分的代码放在父类中，而将不同的代码放入不同的子类中。也就是说我们需要声明一个抽象的父类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法让子类来实现剩余的逻辑，不同的子类可以以不同的方式来实现这些逻辑。所以模板方法的模板其实就是一个普通的方法，只不过这个方法是将算法实现的步骤封装起来的。



模板方法模式包含如下角色：  
AbstractClass: 抽象类   
ConcreteClass: 具体子类

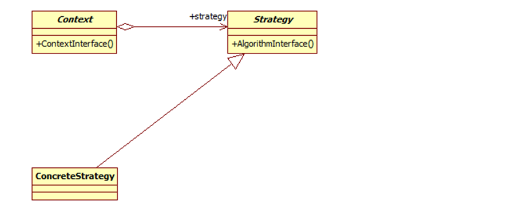
#### 策略模式

 我们知道一件事可能会有很多种方式来实现它，但是其中总有一种最高效的方式，在软件开发的世界里面同样如此，我们也有很多中方法来实现一个功能，但是我们需要一种简单、高效的方式来实现它，使得系统能够非常灵活，这就是策略模式。

所以策略模式就是定义了算法族，分别封装起来，让他们之前可以互相转换，此模式然该算法的变化独立于使用算法的客户。

在策略模式中它将这些解决问题的方法定义成一个算法群，每一个方法都对应着一个具体的算法，这里的一个算法我就称之为一个策略。虽然策略模式定义了算法，但是它并不提供算法的选择，即什么算法对于什么问题最合适这是策略模式所不关心的，所以对于策略的选择还是要客户端来做。客户必须要清楚的知道每个算法之间的区别和在什么时候什么地方使用什么策略是最合适的，这样就增加客户端的负担。

同时策略模式也非常完美的符合了“开闭原则”，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。但是一个策略对应一个类将会是系统产生很多的策略类。



策略模式包含如下角色：  
Context: 环境类  
Strategy: 抽象策略类  
ConcreteStrategy: 具体策略类

#### 状态模式

 在很多情况下我们对象的行为依赖于它的一个或者多个变化的属性，这些可变的属性我们称之为状态，也就是说行为依赖状态，即当该对象因为在外部的互动而导致他的状态发生变化，从而它的行为也会做出相应的变化。对于这种情况，我们是不能用行为来控制状态的变化，而应该站在状态的角度来思考行为，即是什么状态就要做出什么样的行为。这个就是状态模式。

所以状态模式就是允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

在状态模式中我们可以减少大块的if…else语句，它是允许态转换逻辑与状态对象合成一体，但是减少if…else语句的代价就是会换来大量的类，所以状态模式势必会增加系统中类或者对象的个数。

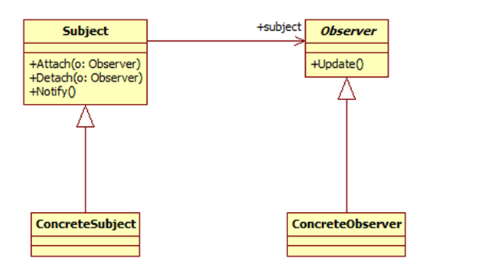
同时状态模式是将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。但是这样就会导致系统的结构和实现都会比较复杂，如果使用不当就会导致程序的结构和代码混乱，不利于维护。

 状态模式包含如下角色：  
Context: 环境类  
State: 抽象状态类  
ConcreteState: 具体状态类

#### 观察者模式

何谓观察者模式？观察者模式定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

在这里，发生改变的对象称之为观察目标，而被通知的对象称之为观察者。一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，所以么可以根据需要增加和删除观察者，使得系统更易于扩展。所以观察者提供了一种对象设计，让主题和观察者之间以松耦合的方式结合。

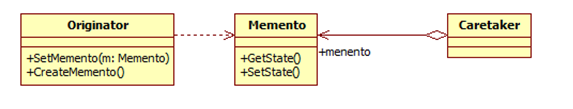


 观察者模式包含如下角色：  
Subject: 目标  
ConcreteSubject: 具体目标  
Observer: 观察者  
ConcreteObserver: 具体观察者

#### 备忘录模式

 后悔药人人都想要，但是事实却是残酷的，根本就没有后悔药可买，但是也不仅如此，在软件的世界里就有后悔药！备忘录模式就是一种后悔药，它给我们的软件提供后悔药的机制，通过它可以使系统恢复到某一特定的历史状态。

所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。它实现了对信息的封装，使得客户不需要关心状态保存的细节。保存就要消耗资源，所以备忘录模式的缺点就在于消耗资源。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。



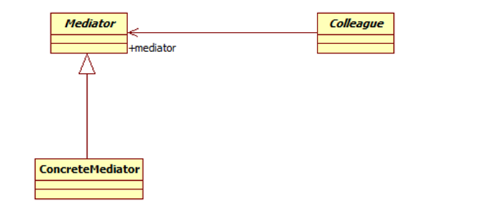
备忘录模式包含如下角色：  
Originator: 原发器  
Memento: 备忘录  
Caretaker: 负责人

19.中介者模式

 租房各位都有过的经历吧！在这个过程中中介结构扮演着很重要的角色，它在这里起到一个中间者的作用，给我们和房主互相传递信息。在外面软件的世界里同样需要这样一个中间者。在我们的系统中有时候会存在着对象与对象之间存在着很强、复杂的关联关系，如果让他们之间有直接的联系的话，必定会导致整个系统变得非常复杂，而且可扩展性很差！在前面我们就知道如果两个类之间没有不必彼此通信，我们就不应该让他们有直接的关联关系，如果实在是需要通信的话，我们可以通过第三者来转发他们的请求。同样，这里我们利用中介者来解决这个问题。

所谓中介者模式就是用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。在中介者模式中，中介对象用来封装对象之间的关系，各个对象可以不需要知道具体的信息通过中介者对象就可以实现相互通信。它减少了对象之间的互相关系，提供了系统可复用性，简化了系统的结构。

 在中介者模式中，各个对象不需要互相知道了解，他们只需要知道中介者对象即可，但是中介者对象就必须要知道所有的对象和他们之间的关联关系，正是因为这样就导致了中介者对象的结构过于复杂，承担了过多的职责，同时它也是整个系统的核心所在，它有问题将会导致整个系统的问题。所以如果在系统的设计过程中如果出现“多对多”的复杂关系群时，千万别急着使用中介者模式，而是要仔细思考是不是您设计的系统存在问题。



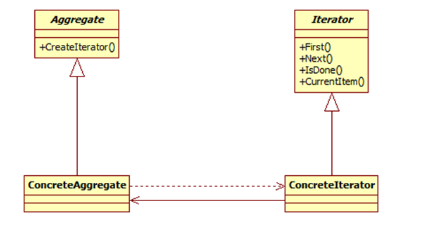
Mediator: 抽象中介者  
ConcreteMediator: 具体中介者  
Colleague: 抽象同事类  
ConcreteColleague: 具体同事类

20.迭代器模式

对于迭代在编程过程中我们经常用到，能够游走于聚合内的每一个元素，同时还可以提供多种不同的遍历方式，这就是迭代器模式的设计动机。在我们实际的开发过程中，我们可能会需要根据不同的需求以不同的方式来遍历整个对象，但是我们又不希望在聚合对象的抽象接口中充斥着各种不同的遍历操作，于是我们就希望有某个东西能够以多种不同的方式来遍历一个聚合对象，这时迭代器模式出现了。

何为迭代器模式？所谓迭代器模式就是提供一种方法顺序访问一个聚合对象中的各个元素，而不是暴露其内部的表示。迭代器模式是将迭代元素的责任交给迭代器，而不是聚合对象，我们甚至在不需要知道该聚合对象的内部结构就可以实现该聚合对象的迭代。

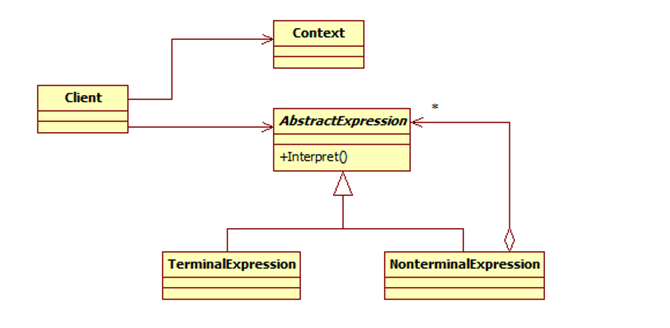
通过迭代器模式，使得聚合对象的结构更加简单，它不需要关注它元素的遍历，只需要专注它应该专注的事情，这样就更加符合单一职责原则了。



迭代器模式包含如下角色：  
Iterator: 抽象迭代器  
ConcreteIterator: 具体迭代器  
Aggregate: 抽象聚合类  
ConcreteAggregate: 具体聚合类

21.解释器模式

所谓解释器模式就是定义语言的文法，并且建立一个解释器来解释该语言中的句子。解释器模式描述了如何构成一个简单的语言解释器，主要应用在使用面向对象语言开发的编译器中。它描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。

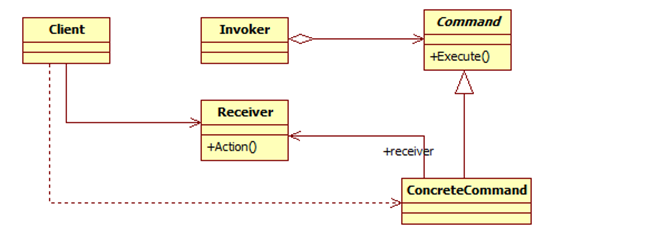


解释器模式包含如下角色：  
AbstractExpression: 抽象表达式  
TerminalExpression: 终结符表达式  
NonterminalExpression: 非终结符表达式  
Context: 环境类  
Client: 客户类

22.命令模式

 有些时候我们想某个对象发送一个请求，但是我们并不知道该请求的具体接收者是谁，具体的处理过程是如何的，们只知道在程序运行中指定具体的请求接收者即可，对于这样将请求封装成对象的我们称之为命令模式。所以命令模式将请求封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。同时命令模式支持可撤销的操作。

命令模式可以将请求的发送者和接收者之间实现完全的解耦，发送者和接收者之间没有直接的联系，发送者只需要知道如何发送请求命令即可，其余的可以一概不管，甚至命令是否成功都无需关心。同时我们可以非常方便的增加新的命令，但是可能就是因为方便和对请求的封装就会导致系统中会存在过多的具体命令类。

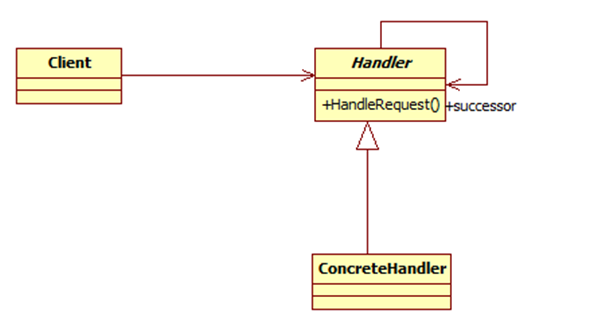


命令模式包含如下角色：  
Command: 抽象命令类  
ConcreteCommand: 具体命令类  
Invoker: 调用者  
Receiver: 接收者  
Client:客户类

23.责任链模式

职责链模式描述的请求如何沿着对象所组成的链来传递的。它将对象组成一条链，发送者将请求发给链的第一个接收者，并且沿着这条链传递，直到有一个对象来处理它或者直到最后也没有对象处理而留在链末尾端。

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止，这就是职责链模式。在职责链模式中，使得每一个对象都有可能来处理请求，从而实现了请求的发送者和接收者之间的解耦。同时职责链模式简化了对象的结构，它使得每个对象都只需要引用它的后继者即可，而不必了解整条链，这样既提高了系统的灵活性也使得增加新的请求处理类也比较方便。但是在职责链中我们不能保证所有的请求都能够被处理，而且不利于观察运行时特征。



职责链模式包含如下角色：  
Handler: 抽象处理者  
ConcreteHandler: 具体处理者  
Client: 客户类

# 常见题

## 判断一个单向链表中是否存在环的最佳方法是

让快慢指针都从链表表头开始，快指针一次向前移动两个位置，慢指针每次向前移动一个位置。如果快指针到达了NULL，说明不存在环，但如果快指针追上了慢指针，说明存在环

## 非空字串

N(n+1)/2

## 最大公约数，O(logn)，辗转相除法

**public static int** f(**int** a, **int** b){  
 **if**(b==0){  
 **return** a;  
 }**else**{  
 **return** *f*(b, a % b);  
 }  
}

## Hanoi塔 f（n）=f(n-1)\*2+1

## 将两个各有n个元素的有序表归并成一个有序表,其最多的比较次数是2n-1

## Next数组+NextVal

nextVal: https://blog.csdn.net/yuehailin/article/details/85323014

next数组的求解方法是：第一位的next值为0，第二位的next值为1，后面求解每一位的next值时，根据前一位进行比较。首先将前一位与其next值对应的内容进行比较，如果相等，则该位的next值就是前一位的next值加上1；如果不等，向前继续寻找next值对应的内容来与前一位进行比较，直到找到某个位上内容的next值对应的内容与前一位相等为止，则这个位对应的值加上1即为需求的next值；如果找到第一位都没有找到与前一位相等的内容，那么需求的位上的next值即为1。

对于本题，求next值的过程：

已知串S=′aaab′,其Next数组值为() 0123

## 卡特兰数（Catalan number，又称卡塔兰数）c(2n,n)/(n+1)。其中，n为节点的个数

## 一个含有n个顶点的连通且无环的简单无向图，在其临接矩阵存储结构中共有多少个零元素？

n^2 -2n +2