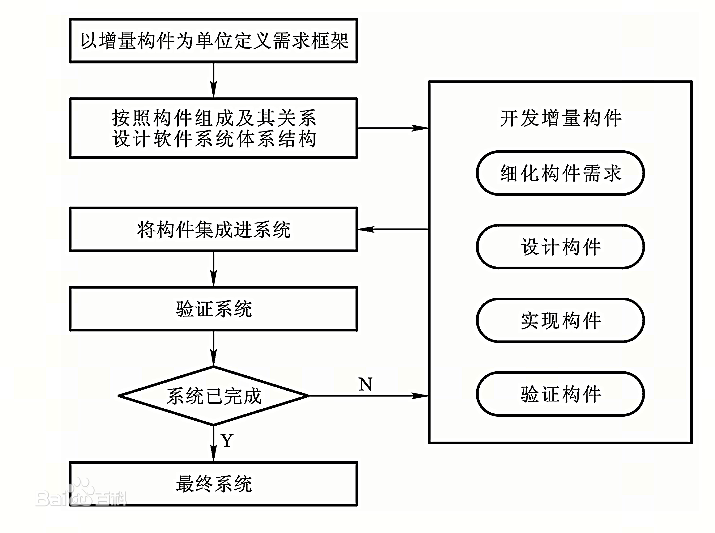
# 增量模型

增量模型是把待开发的软件系统模块化，将每个模块作为一个增量组件，从而分批次地分析、设计、编码和测试这些增量组件。运用增量模型的软件开发过程是递增式的过程。相对于瀑布模型而言，采用增量模型进行开发，开发人员不需要一次性地把整个软件产品提交给用户，而是可以分批次进行提交。

## 基本介绍

增量模型又称为渐增模型，也称为有计划的产品改进模型，它从一组给定的需求开始，通过构造一系列可执行中间版本来实施开发活动。第一个版本纳入一部分需求，下一个版本纳入更多的需求，依此类推，直到系统完成。每个中间版本都要执行必需的过程、活动和任务。

增量模型是瀑布模型和原型进化模型的综合，它对软件过程的考虑是：在整体上按照瀑布模型的流程实施项目开发，以方便对项目的管理；但在软件的实际创建中，则将软件系统按功能分解为许多增量构件，并以构件为单位逐个地创建与交付，直到全部增量构件创建完毕，并都被集成到系统之中交付用户使用。

如同原型进化模型一样，增量模型逐步地向用户交付软件产品，但不同于原型进化模型的是，增量模型在开发过程中所交付的不是完整的新版软件，而只是新增加的构件。 右图是增量模型的工作流程，它被分成以下三个阶段：

1、在系统开发的前期阶段，为了确保所建系统具有优良的结构，仍需要针对整个系统进行需求分析和概要设计，需要确定系统的基于增量构件的需求框架，并以需求框架中构件的组成及关系为依据，完成对软件系统的体系结构设计。

2、在完成软件体系结构设计之后，可以进行增量构件的开发。这个时候，需要对构件进行需求细化，然后进行设计、编码测试和有效性验证。

3、在完成了对某个增量构件的开发之后，需要将该构件集成到系统中去，并对已经发生了改变的系统重新进行有效性验证，然后再继续下一个增量构件的开发。

## 增量模型的特点

增量模型的最大特点就是将待开发的软件系统模块化和组件化。基于这个特点，增量模型具有以下优点。

1、将待开发的软件系统模块化，可以分批次地提交软件产品，使用户可以及时了解软件项目的进展。

2、以组件为单位进行开发降低了软件开发的风险。一个开发周期内的错误不会影响到整个软件系统。

3、开发顺序灵活。开发人员可以对组件的实现顺序进行优先级排序，先完成需求稳定的核心组件。当组件的优先级发生变化时，还能及时地对实现顺序进行调整。

增量模型的缺点是要求待开发的软件系统可以被模块化。如果待开发的软件系统很难被模块化，那么将会给增量开发带来很多麻烦。

增量模型适用于具有以下特征的软件开发项目：

1、软件产品可以分批次地进行交付。

2、待开发的软件系统能够被模块化。

3、软件开发人员对应用领域不熟悉，难以一次性地进行系统开发。

4、项目管理人员把握全局的水平较高。 [2]

## 增量模型的作用

增量模型带来了以下几方面的作用：

1、开发初期的需求定义只是用来确定软件的基本结构，这使得开发初期，用户只需要对软件需求进行大概的描述，而对于需求的细节性描述，则可以延迟到增量构件开发时进行，以增量构件为单位逐个地进行需求补充。这种方式有利于用户需求的逐渐明朗，能够有效适应用户需求的变更。

2、软件系统可以按照增量构件的功能安排开发的优先顺序，并逐个实现和交付使用。这不仅有利于用户尽早地用上系统，能够更好地适应新的软件环境，而且用户在以增量方式使用系统的过程中，还能够获得对软件系统后续构件的需求经验。这样能使软件需求定义越往后越顺利。

3、软件系统是逐渐扩展的，因此，开发者可以通过对诸多构件的开发，逐步积累开发经验。实际上，增量式开发还有利于技术复用，前面构件中设计的算法、采用的技术策略、编写的源码等，都可以应用到后面将要创建的增量构件中去。

4、增量式开发还有利于从总体上降低软件项目的技术风险。个别的构件或许不能使用，但这一般不会影响到整个系统的正常工作。

5、实际上，在采用增量模型时，具有最高优先权的核心增量构件将会被最先交付，而随着后续构件不断被集成进系统，这个核心构件将会受到最多次数的测试。这意味着软件系统最重要的心脏部分将具有最高的可靠性，这将使得整个软件系统更具健壮性。

比较瀑布模型、原型进化模型，增量模型具有非常显著的优越性。但是，增量模型对软件设计有更高的技术要求，特别是对软件体系结构，要求它具有很好的开放性与稳定性，能够顺利地实现构件的集成。在把每个新的构件集成到已建软件系统的结构中的时候，一般要求这个新增的构件应该尽量少地改变原来已建的软件结构。因此增量构件要求具有相当好的功能独立性，其接口应该简单，以方便集成时与系统的连接。

# 值类型和引用类型

其它多数语言：某些类型一定是值类型，某些类型一定是引用类型

C++：所有类型默认都是值类型，都能加个引用得到相应的引用类型

值类型变量声明后，不管是否已经赋值，编译器为其分配内存。此时该值存储于栈上；int i=10;执行之前为默认一个值0。

引用类型定义时在栈上开辟一个空间用来存放其在堆上的地址，当赋值或者实例化时候就会在堆上开辟一个空间，然后把堆中的地址存放在栈中，这时候栈就存放了其地址。

# 死锁

## 定义

线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。当线程进入对象的synchronized代码块时，便占有了资源，直到它退出该代码块或者调用wait方法，才释放资源，在此期间，其他线程将不能进入该代码块。当线程互相持有对方所需要的资源时，会互相等待对方释放资源，如果线程都不主动释放所占有的资源，将产生死锁。

## 必要条件

1.互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放

2.请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。

3.不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用

4.循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

## 预防

打破四个必要条件之一就能有效预防死锁的发生。

打破互斥条件：各个进程可以同时使用资源。需要注意的是互斥条件是不能摒弃的，因为一旦摒弃互斥条件，有可能使访问临界资源的进程出现执行结果的错误。

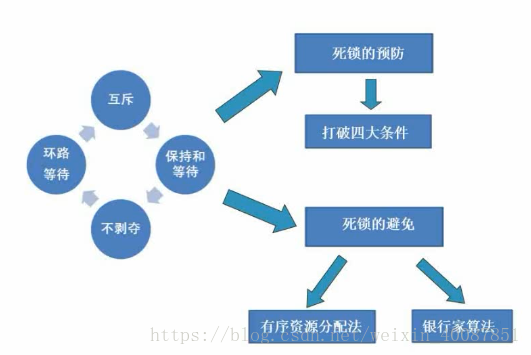
打破保持等待条件：当进程因请求不到资源处于请求阻塞状态时，主要将自己拥有的资源分配给其他资源。

打破不可剥夺条件：当进程缺少资源时，可从其他进行剥夺相应资源使自己运行。

打破环路等待条件：实现资源有序分配策略，A进程运行完再分配资源给B进程，B进程运行完再分配资源给C进行…

## 避免

1.有序资源分配法:A进程运行完再分配资源给B进程，B进程运行完再分配资源给C进行…，可以有效避免死锁，但资源利用率低。

2.银行家算法。

## 银行家算法

利用银行家算法避免死锁的基本思想是：一个进程提出资源请求后，系统先进行资源的试分配，然后检查本次的试分配是否使系统处于安全状态，若安全则按试分配方案分配资源，否则不分配资源。

银行家算法缺少实用价值：很少有进程能够在运行之前就知道其所需资源的最大值，而且进程数不是固定的，往往在不断变化，况且原本可用的资源也可能突然间变得不可用。

安全状态转不安全状态

# 递归和递推

递推就是从前往后推,递归还有个回溯的过程

## 一、递推算法基本思想：

     递推算法是一种理性思维模式的代表，其根据已有的数据和关系，逐步推导而得到结果。递推算法的执行过程如下：

     1）根据已有的·结果和关系，求解中间结果

     2）判定是否达到要求，如果没有达到，则继续根据已知结果和关系求解中间结果；如果满足要求，则表示寻找到一个正确的答案。

递推算法往往需要用户知道答案和问题之间的逻辑关系。在许多数学问题中，都有着明确的计算公式可以遵循，因此往往可以采用递推来实现。

2.递推算法示例

     如果一对两个月大的兔子以后每个月可以生一对兔子，而一对新生的兔子出生两个月后才可以生兔子。也就是说1月份出生的3月份才能生子。假定一年内兔子没有死亡事件，那么一年后共有多少对兔子。

    分析产仔问题，逐月分析每月的兔子对数：

    第一个月：1对兔子

   第二个月：1对兔子

   第三个月：2对兔子

   第四个月：3对兔子

   第五个月：5对兔子

   第六个月：8对兔子

  。。。。。。。

可以看出，从第三个月开始，每个月的兔子总队数是前两个月的对数之和，公式如下：

  第n个月兔子总数Fn = F(n-1)+F(n-2)

 代码示例：

[复制代码](javascript:void(0);)

public static int Fibonacci(int n){ //月数

int t1,t2;

if(n == 1 || n ==2){

return 1;

}else{

t1 = Fibonacci(n-1);

t2 = Fibonacci(n-2);

return t1+t2;

}

}

[复制代码](javascript:void(0);)

## 二、递归算法思想：

 递归算法即在程序中不断反复调用自身来达到求解问题的方法。编写递归算法时，必须使用if语句强制方法在未执行递归调用前返回。如果不这样做，在调用方法后，它将永远不会返回。

缺点：递归算法比非递归形式运行速度要慢一些。

算法示例：

  计算阶乘，就是从1到指定数之间的所有自然数相乘的结果，n的阶乘为：

   n! = n\*(n-1)\*(n-2)\*.......\*2\*1;

   代码：

[复制代码](javascript:void(0);)

int fact(int n){

if(n <= 1)

return n;

else

return n\*fact(n-1); //递归

}

# 字符串比较

比较两个字符串的算法是：逐个比较两个串中对应的字符，字符大小按照ASCII码值确定，从左向右比较，如果遇到不同字符，所遇第一对不同字符的大小关系就确定了两个字符串的大小关系（字典序），如果未遇到不同字符而某个字符串首先结束，那么这个字符串是较小的，否则两个字符串相等。

# 运算符优先级

## C语言优先级

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 优先级 | 运算符 | 名称或含义 | 使用形式 | 结合方向 | 说明 |
| 1 | [] | 数组下标 | 数组名[整型表达式] | 左到右 |  |
|  | () | 圆括号 | （表达式）/函数名(形参表) |  |  |
|  | . | 成员选择（对象） | 对象.成员名 |  |  |
|  | -> | 成员选择（指针） | 对象指针->成员名 |  |  |
| 2 | - | 负号运算符 | -算术类型表达式 | 右到左 | 单目运算符 |
|  | (type) | 强制类型转换 | (纯量数据类型)纯量表达式 |  |  |
|  | ++ | 自增运算符 | ++纯量类型可修改左值表达式 |  | 单目运算符 |
|  | -- | 自减运算符 | --纯量类型可修改左值表达式 |  | 单目运算符 |
|  | \* | 取值运算符 | \*指针类型表达式 |  | 单目运算符 |
|  | & | 取地址运算符 | &表达式 |  | 单目运算符 |
|  | ! | 逻辑非运算符 | !纯量类型表达式 |  | 单目运算符 |
|  | ~ | 按位取反运算符 | ~整型表达式 |  | 单目运算符 |
|  | sizeof | 长度运算符 | sizeof 表达式  sizeof(类型) |  |  |
| 3 | / | 除 | 表达式/表达式 | 左到右 | 双目运算符 |
|  | \* | 乘 | 表达式\*表达式 |  | 双目运算符 |
|  | % | 余数（取模） | 整型表达式%整型表达式 |  | 双目运算符 |
| 4 | + | 加 | 表达式+表达式 | 左到右 | 双目运算符 |
|  | - | 减 | 表达式-表达式 |  | 双目运算符 |
| 5 | << | 左移 | 整型表达式<<整型表达式 | 左到右 | 双目运算符 |
|  | >> | 右移 | 整型表达式>>整型表达式 |  | 双目运算符 |
| 6 | > | 大于 | 表达式>表达式 | 左到右 | 双目运算符 |
|  | >= | 大于等于 | 表达式>=表达式 |  | 双目运算符 |
|  | < | 小于 | 表达式<表达式 |  | 双目运算符 |
|  | <= | 小于等于 | 表达式<=表达式 |  | 双目运算符 |
| 7 | == | 等于 | 表达式==表达式 | 左到右 | 双目运算符 |
|  | != | 不等于 | 表达式!= 表达式 |  | 双目运算符 |
| 8 | & | 按位与 | 整型表达式&整型表达式 | 左到右 | 双目运算符 |
| 9 | ^ | 按位异或 | 整型表达式^整型表达式 | 左到右 | 双目运算符 |
| 10 | | | 按位或 | 整型表达式|整型表达式 | 左到右 | 双目运算符 |
| 11 | && | 逻辑与 | 表达式&&表达式 | 左到右 | 双目运算符 |
| 12 | || | 逻辑或 | 表达式||表达式 | 左到右 | 双目运算符 |
| 13 | ?: | 条件运算符 | 表达式1? 表达式2: 表达式3 | 右到左 | 三目运算符 |
| 14 | = | 赋值运算符 | 可修改左值表达式=表达式 | 右到左 |  |
|  | /= | 除后赋值 | 可修改左值表达式/=表达式 |  |  |
|  | \*= | 乘后赋值 | 可修改左值表达式\*=表达式 |  |  |
|  | %= | 取模后赋值 | 可修改左值表达式%=表达式 |  |  |
|  | += | 加后赋值 | 可修改左值表达式+=表达式 |  |  |
|  | -= | 减后赋值 | 可修改左值表达式-=表达式 |  |  |
|  | <<= | 左移后赋值 | 可修改左值表达式<<=表达式 |  |  |
|  | >>= | 右移后赋值 | 可修改左值表达式>>=表达式 |  |  |
|  | &= | 按位与后赋值 | 可修改左值表达式&=表达式 |  |  |
|  | ^= | 按位异或后赋值 | 可修改左值表达式^=表达式 |  |  |
|  | |= | 按位或后赋值 | 可修改左值表达式|=表达式 |  |  |
| 15 | , | 逗号运算符 | 表达式,表达式,… | 左到右 | 从左向右顺序结合 |

## C++

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 描述 | 例子 | 可重载性 |
| **第一级别** |  |  |  |
| :: | 作用域解析符 | Class::age = 2; | 不可重载 |
| **第二级别** |  |  |  |
| () | 函数调用 | isdigit('1') | 可重载 |
| () | 成员初始化 | c\_tor(int x, int y) : \_x(x), \_y(y\*10){}; | 可重载 |
| [] | 数组数据获取 | array[4] = 2; | 可重载 |
| -> | 指针型成员调用 | ptr->age = 34; | 可重载 |
| . | 对象型成员调用 | obj.age = 34; | 不可重载 |
| ++ | 后自增运算符 | for( int i = 0; i < 10; i++ ) cout << i; | 可重载 |
| -- | 后自减运算符 | for( int i = 10; i > 0; i-- ) cout << i; | 可重载 |
| const\_cast | 特殊属性转换 | const\_cast<type\_to>(type\_from); | 不可重载 |
| dynamic\_cast | 特殊属性转换 | dynamic\_cast<type\_to>(type\_from); | 不可重载 |
| static\_cast | 特殊属性转换 | static\_cast<type\_to>(type\_from); | 不可重载 |
| reinterpret\_cast | 特殊属性转换 | reinterpret\_cast<type\_to>(type\_from); | 不可重载 |
| typeid | 对象类型符 | cout &laquo; typeid(var).name();  cout &laquo; typeid(type).name(); | 不可重载 |
| **第三级别**(具有右结合性) |  |  |  |
| ! | 逻辑取反 | if( !done ) … | 可重载 |
| not | ! 的另一种表达 |  |  |
| ~ | 按位取反 | flags = ~flags; | 可重载 |
| compl | ~的另一种表达 |  |  |
| ++ | 预自增运算符 | for( i = 0; i < 10; ++i ) cout << i; | 可重载 |
| -- | 预自减运算符 | for( i = 10; i > 0; --i ) cout << i; | 可重载 |
| - | 负号 | int i = -1; | 可重载 |
| + | 正号 | int i = +1; | 可重载 |
| \* | 指针取值 | int data = \*intPtr; | 可重载 |
| & | 值取指针 | int \*intPtr = &data; | 可重载 |
| new | 动态元素内存分配 | long \*pVar = new long;  MyClass \*ptr = new MyClass(args); | 可重载 |
| new [] | 动态数组内存分配 | long \*array = new long[n]; | 可重载 |
| delete | 动态析构元素内存 | delete pVar; | 可重载 |
| delete [] | 动态析构数组内存 | delete [] array; | 可重载 |
| (type) | 强制类型转换 | int i = (int) floatNum; | 可重载 |
| sizeof | 返回类型内存 | int size = sizeof floatNum;  int size = sizeof(float); | 不可重载 |
| **第四级别** |  |  |  |
| ->\* | 类指针成员引用 | ptr->\*var = 24; | 可重载 |
| .\* | 类对象成员引用 | obj.\*var = 24; | 不可重载 |
| **第五级别** |  |  |  |
| \* | 乘法 | int i = 2 \* 4; | 可重载 |
| / | 除法 | float f = 10.0 / 3.0; | 可重载 |
| % | 取余数(模运算) | int rem = 4 % 3; | 可重载 |
| **第六级别** |  |  |  |
| + | 加法 | int i = 2 + 3; | 可重载 |
| - | 减法 | int i = 5 - 1; | 可重载 |
| **第七级别** |  |  |  |
| << | 位左移 | int flags = 33 << 1; | 可重载 |
| >> | 位右移 | int flags = 33 >> 1; | 可重载 |
| **第八级别** |  |  |  |
| < | 小于 | if( i < 42 ) … | 可重载 |
| <= | 小于等于 | if( i <= 42 ) ... | 可重载 |
| > | 大于 | if( i > 42 ) … | 可重载 |
| >= | 大于等于 | if( i >= 42 ) ... | 可重载 |
| **第九级别** |  |  |  |
| == | 恒等于 | if( i == 42 ) ... | 可重载 |
| eq | == 的另一种表达 |  |  |
| != | 不等于 | if( i != 42 ) … | 可重载 |
| not\_eq | !=的另一种表达 |  |  |
| **第十级别** |  |  |  |
| & | 位且运算 | flags = flags & 42; | 可重载 |
| bitand | &的另一种表达 |  |  |
| **第十一级别** |  |  |  |
| ^ | 位异或运算 | flags = flags ^ 42; | 可重载 |
| xor | ^的另一种表达 |  |  |
| **第十二级别** |  |  |  |
| | | 位或运算 | flags = flags | 42; | 可重载 |
| bitor | |的另一种表达 |  |  |
| **第十三级别** |  |  |  |
| && | 逻辑且运算 | if( conditionA && conditionB ) … | 可重载 |
| and | &&的另一种表达 |  |  |
| **第十四级别** |  |  |  |
| || | 逻辑或运算 | if( conditionA || conditionB ) ... | 可重载 |
| or | ||的另一种表达 |  |  |
| **第十五级别**(具有右结合性) |  |  |  |
| ? : | 条件运算符 | int i = (a > b) ? a : b; | 不可重载 |
| **第十六级别**(具有右结合性) |  |  |  |
| = | 赋值 | int a = b; | 可重载 |
| += | 加赋值运算 | a += 3; | 可重载 |
| -= | 减赋值运算 | b -= 4; | 可重载 |
| \*= | 乘赋值运算 | a \*= 5; | 可重载 |
| /= | 除赋值运算 | a /= 2; | 可重载 |
| %= | 模赋值运算 | a %= 3; | 可重载 |
| &= | 位且赋值运算 | flags &= new\_flags; | 可重载 |
| and\_eq | &= 的另一种表达 |  |  |
| ^= | 位异或赋值运算 | flags ^= new\_flags; | 可重载 |
| xor\_eq | ^=的另一种表达 |  |  |
| |= | 位或赋值运算 | flags |= new\_flags; | 可重载 |
| or\_eq | |=的另一种表达 |  |  |
| <<= | 位左移赋值运算 | flags <<= 2; | 可重载 |
| >>= | 位右移赋值运算 | flags >>= 2; | 可重载 |
| **第十七级别** |  |  |  |
| throw | 异常抛出 | throw EClass(“Message”); | 不可重载 |
| **第十八级别** |  |  |  |
| , | 逗号分隔符 | for( i = 0, j = 0; i < 10; i++, j++ ) … | 可重载 |

# 唯一确定一棵二叉树

给出中序遍历之后再给一个其他的遍历就能够确定了，前序和后续不能确定。因为一个跟在最前，一个根在最后，无法区分左右孩子，不能递归的将孩子分开。

## 中序遍历

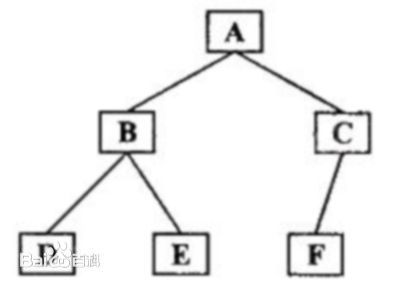
中序遍历（LDR）是[二叉树遍历](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91%E9%81%8D%E5%8E%86/9796049)的一种，也叫做[中根遍历](https://baike.baidu.com/item/%E4%B8%AD%E6%A0%B9%E9%81%8D%E5%8E%86/1703566)、中序周游。在二叉树中，中序遍历首先遍历左子树，然后访问根结点，最后遍历右子树。中序遍历首先遍历左子树，然后访问根结点，最后遍历右子树。若二叉树为空则结束返回，否则：

（1）中序遍历左子树

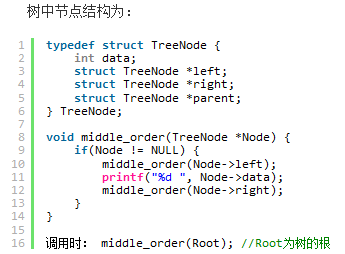
（2）访问根结点

（3）中序遍历右子树

如右图所示二叉树，中序遍历结果：DBEAFC



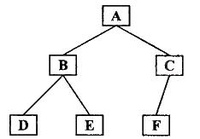
### c++版本



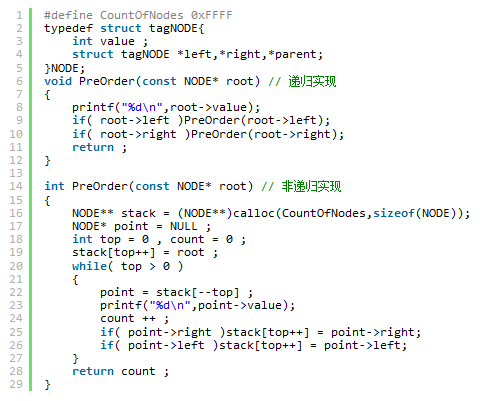
## 先序遍历(DBF)

先序遍历也叫做先根遍历、前序遍历，可记做根左右（二叉树父结点向下先左后右）。

首先访问根结点然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树，如果二叉树为空则返回。例如，下图所示二叉树的遍历结果是：ABDECF



### C版本



## 后序遍历

后序遍历（LRD）是[二叉树遍历](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91%E9%81%8D%E5%8E%86/9796049)的一种，也叫做[后根遍历](https://baike.baidu.com/item/%E5%90%8E%E6%A0%B9%E9%81%8D%E5%8E%86/8545391)、后序周游，可记做左右根。后序遍历有[递归算法](https://baike.baidu.com/item/%E9%80%92%E5%BD%92%E7%AE%97%E6%B3%95/4323575)和非递归算法两种。在二叉树中，先左后右再根，即首先遍历左子树，然后遍历右子树，最后访问根结点。

后序遍历首先遍历左子树，然后遍历右子树，最后访问根结点，在遍历左、右子树时，仍然先遍历左子树，然后遍历右子树，最后遍历根结点。即：

若二叉树为空则结束返回，

否则：

（1）后序遍历左子树

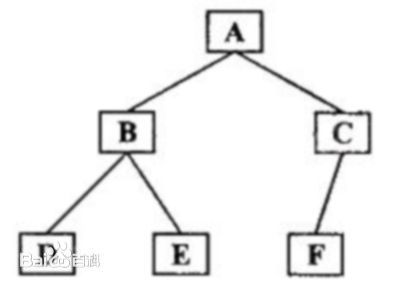
（2）后序遍历右子树

（3）访问根结点

如右图所示二叉树

后序遍历结果：DEBFCA

已知前序遍历和中序遍历，就能确定后序遍历。 [1]



首先要搞清楚先序、中序、后序的非递归算法共同之处：用栈来保存先前走过的路径，以便可以在访问完子树后,可以利用栈中的信息,回退到当前节点的双亲节点,进行下一步操作。

后序遍历的非递归算法是三种顺序中最复杂的，原因在于，后序遍历是先访问左、右子树,再访问根节点，而在非递归算法中，利用栈回退到时，并不知道是从左子树回退到根节点，还是从右子树回退到根节点，如果从左子树回退到根节点，此时就应该去访问右子树，而如果从右子树回退到根节点，此时就应该访问根节点。所以相比前序和后序，必须得在压栈时添加信息，以便在退栈时可以知道是从左子树返回，还是从右子树返回进而决定下一步的操作。

# 哈夫曼树（最优二叉搜索树）

## 一、哈夫曼树的概念和定义

**什么是哈夫曼树？**

让我们先举一个例子。

判定树：**在很多问题的处理过程中，需要进行大量的条件判断，这些判断结构的设计直接影响着程序的执行效率。例如，编制一个程序，将百分制转换成五个等级输出。大家可能认为这个程序很简单，并且很快就可以用下列形式编写出来：**

**[cpp]** [view plaincopyprint?](https://www.cnblogs.com/13224ACMer/p/4706174.html)

1. if(score<60)
2. cout<<"Bad"<<endl;
3. else if(score<70)
4. cout<<"Pass"<<endl
5. else if(score<80)
6. cout<<"General"<<endl;
7. else if(score<90)
8. cout<<"Good"<<endl;
9. else
10. cout<<"Very good!"<<endl;

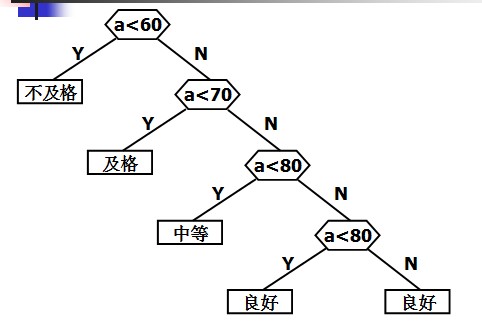
**若考虑上述程序所耗费的时间，就会发现该程序的缺陷。在实际中，学生成绩在五个等级上的分布是不均匀的。当学生百分制成绩的录入量很大时，上述判定过程需要反复调用，此时程序的执行效率将成为一个严重问题。**

**但在实际应用中，往往各个分数段的分布并不是均匀的。下面就是在一次考试中某门课程的各分数段的分布情况：**

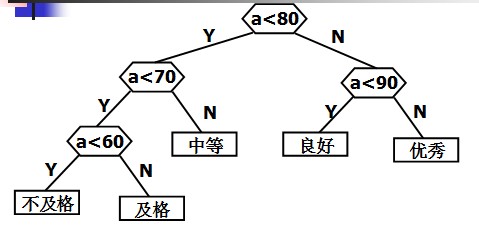
**http://hi.csdn.net/attachment/201203/11/0_1331430021L3WB.gif**

**下面我们就利用哈夫曼树寻找一棵最佳判定树，即总的比较次数最少的判定树。**

**第一种构造方式：**



**第二种构造方式：**



这两种方式，显然后者的判定过程的效率要比前者高。在也没有别地判定过程比第二种方式的效率更高。

**我们称判定过程最优的二叉树为哈夫曼树，又称最优二叉树**

===================================================================================================

**定义哈夫曼树之前先说明几个与哈夫曼树有关的概念：**

**路径： 树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。**

**路径长度：路径上的分枝数目称作路径长度。**

**树的路径长度：从树根到每一个结点的路径长度之和。**

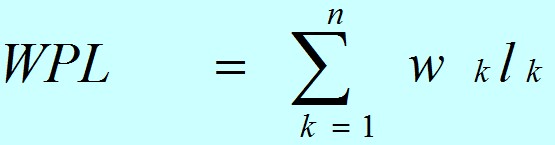
**结点的带权路径长度：在一棵树中，如果其结点上附带有一个权值，通常把该结点的路径长度与该结点上的权值之积称为该结点的带权路径长度（weighted path length）**

**什么是权值?( From 百度百科 )计算机领域中（**[**数据结构**](http://baike.baidu.com/view/9900.htm)**）**

 　　权值就是定义的路径上面的值。可以这样理解为节点间的距离。通常指字符对应的二进制编码出现的概率。至于[霍夫曼](http://baike.baidu.com/view/145803.htm)树中的权值可以理解为：权值大表明出现概率大！一个结点的权值实际上就是这个结点子树在整个树中所占的比例.

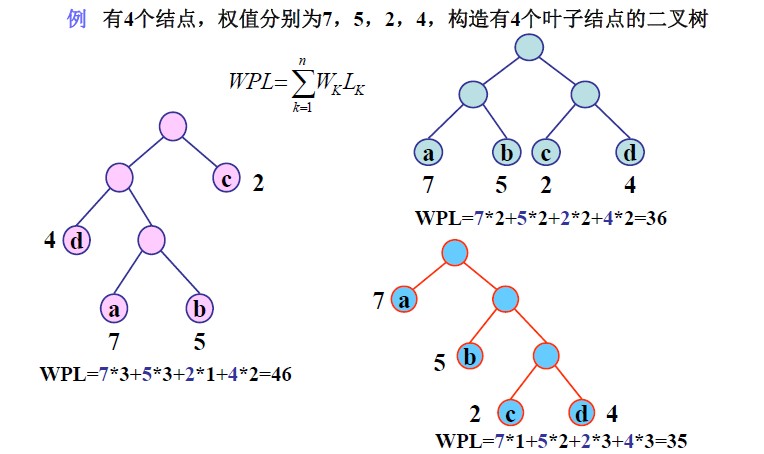
　　abcd四个[叶子结点](http://baike.baidu.com/view/2335663.htm)的权值为7,5,2,4. 这个7,5,2,4是根据实际情况得到的,比如说从一段文本中统计出abcd四个字母出现的次数分别为7,5,2,4. 说a结点的权值为7,意思是说a结点在系统中占有7这个份量.实际上也可以化为百分比来表示,但反而麻烦,实际上是一样的.

**树的带权路径长度：如果树中每个叶子上都带有一个权值，则把树中所有叶子的带权路径长度之和称为树的带权路径长度。设某二叉树有n个带权值的叶子结点，则该二叉树的带权路径长度记为：**

**公式中，Wk为第k个叶子结点的权值；Lk为该结点的路径长度。**

 示例：



======================================================================================================

**一般来说，用n（n>0）个带权值的叶子来构造二叉树，限定二叉树中除了这n个叶子外只能出现度为2的结点。**

**那么符合这样条件的二叉树往往可构造出许多颗，**

**其中带权路径长度最小的二叉树就称为哈夫曼树或最优二叉树**

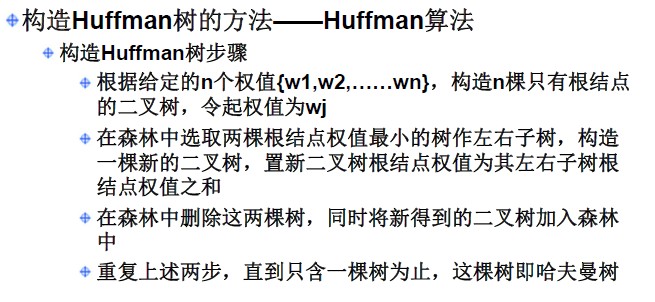
===============================================================================

## 二、哈夫曼树的构造

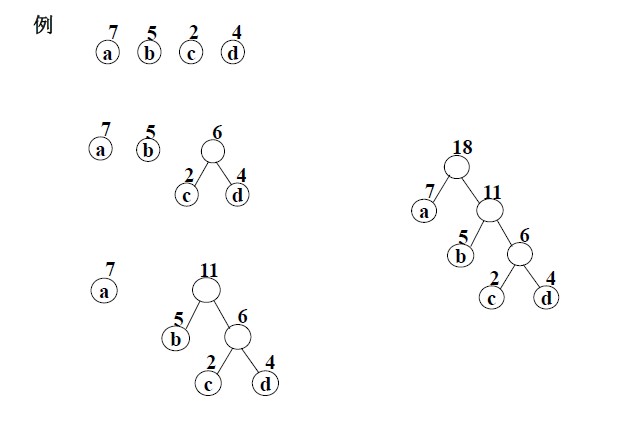
**根据哈弗曼树的定义，一棵二叉树要使其WPL值最小，必须使权值越大的叶子结点越靠近根结点，而权值越小的叶子结点**

**越远离根结点。**

**哈弗曼依据这一特点提出了一种构造最优二叉树的方法，其基本思想如下：**



**下面演示了用Huffman算法构造一棵Huffman树的过程：**



## 三、哈夫曼树的在编码中的应用

**在电文传输中，需要将电文中出现的每个字符进行二进制编码。在设计编码时需要遵守两个原则：**

**（1）发送方传输的二进制编码，到接收方解码后必须具有唯一性，即解码结果与发送方发送的电文完全一样；**

**（2）发送的二进制编码尽可能地短。下面我们介绍两种编码的方式。**

**1. 等长编码**

**这种编码方式的特点是每个字符的编码长度相同（编码长度就是每个编码所含的二进制位数）。假设字符集只含有4个字符A，B，C，D，用二进制两位表示的编码分别为00，01，10，11。若现在有一段电文为：ABACCDA，则应发送二进制序列：00010010101100，总长度为14位。当接收方接收到这段电文后，将按两位一段进行译码。这种编码的特点是译码简单且具有唯一性，但编码长度并不是最短的。**

**2. 不等长编码**

**在传送电文时，为了使其二进制位数尽可能地少，可以将每个字符的编码设计为不等长的，使用频度较高的字符分配一个相对比较短的编码，使用频度较低的字符分配一个比较长的编码。例如，可以为A，B，C，D四个字符分别分配0，00，1，01，并可将上述电文用二进制序列：000011010发送，其长度只有9个二进制位，但随之带来了一个问题，接收方接到这段电文后无法进行译码，因为无法断定前面4个0是4个A，1个B、2个A，还是2个B，即译码不唯一，因此这种编码方法不可使用。**

**因此，为了设计长短不等的编码，以便减少电文的总长，还必须考虑编码的唯一性，即在建立不等长编码时必须使任何一个字符的编码都不是另一个字符的前缀，这宗编码称为前缀编码（prefix  code）**

**（1）利用字符集中每个字符的使用频率作为权值构造一个哈夫曼树；**

**（2）从根结点开始，为到每个叶子结点路径上的左分支赋予0，右分支赋予1，并从根到叶子方向形成该叶子结点的编码**

**例题：**

假设一个文本文件TFile中只包含7个字符{A，B，C，D，E，F，G}，这7个字符在文本中出现的次数为{5，24，7，17，34，5，13}

利用哈夫曼树可以为文件TFile构造出符合前缀编码要求的不等长编码

具体做法：

1. 将TFile中7个字符都作为叶子结点，每个字符出现次数作为该叶子结点的权值

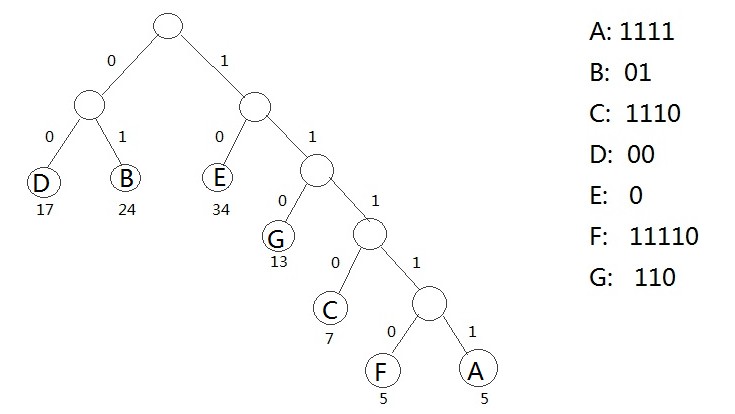
2. 规定哈夫曼树中所有左分支表示字符0，所有右分支表示字符1,将依次从根结点到每个叶子结点所经过的分支的二进制位的序列作为该

     结点对应的字符编码

3. 由于从根结点到任何一个叶子结点都不可能经过其他叶子，这种编码一定是前缀编码，哈夫曼树的带权路径长度正好是文件TFile编码

    的总长度

通过哈夫曼树来构造的编码称为**哈弗曼编码（huffman code）**



**[**

[复制代码](javascript:void(0);)

#include<iostream>

#include<cstdio>

#include<cstring>

using namespace std;

#define N 10 // 带编码字符的个数，即树中叶结点的最大个数

#define M (2\*N-1) // 树中总的结点数目

class HTNode{ // 树中结点的结构

public:

unsigned int weight;

unsigned int parent,lchild,rchild;

};

class HTCode{

public:

char data; // 待编码的字符

int weight; // 字符的权值

char code[N]; // 字符的编码

};

void Init(HTCode hc[], int \*n){

// 初始化，读入待编码字符的个数n，从键盘输入n个字符和n个权值

int i;

printf("input n = ");

scanf("%d",&(\*n));

printf("\ninput %d character\n",\*n);

fflush(stdin);

for(i=1; i<=\*n; ++i)

scanf("%c",&hc[i].data);

printf("\ninput %d weight\n",\*n);

for(i=1; i<=\*n; ++i)

scanf("%d",&(hc[i].weight) );

fflush(stdin);

}//

void Select(HTNode ht[], int k, int \*s1, int \*s2){

// ht[1...k]中选择parent为0，并且weight最小的两个结点，其序号由指针变量s1，s2指示

int i;

for(i=1; i<=k && ht[i].parent != 0; ++i){

; ;

}

\*s1 = i;

for(i=1; i<=k; ++i){

if(ht[i].parent==0 && ht[i].weight<ht[\*s1].weight)

\*s1 = i;

}

for(i=1; i<=k; ++i){

if(ht[i].parent==0 && i!=\*s1)

break;

}

\*s2 = i;

for(i=1; i<=k; ++i){

if(ht[i].parent==0 && i!=\*s1 && ht[i].weight<ht[\*s2].weight)

\*s2 = i;

}

}

void HuffmanCoding(HTNode ht[],HTCode hc[],int n){

// 构造Huffman树ht，并求出n个字符的编码

char cd[N];

int i,j,m,c,f,s1,s2,start;

m = 2\*n-1;

for(i=1; i<=m; ++i){

if(i <= n)

ht[i].weight = hc[i].weight;

else

ht[i].parent = 0;

ht[i].parent = ht[i].lchild = ht[i].rchild = 0;

}

for(i=n+1; i<=m; ++i){

Select(ht, i-1, &s1, &s2);

ht[s1].parent = i;

ht[s2].parent = i;

ht[i].lchild = s1;

ht[i].rchild = s2;

ht[i].weight = ht[s1].weight+ht[s2].weight;

}

cd[n-1] = '\0';

for(i=1; i<=n; ++i){

start = n-1;

for(c=i,f=ht[i].parent; f; c=f,f=ht[f].parent){

if(ht[f].lchild == c)

cd[--start] = '0';

else

cd[--start] = '1';

}

strcpy(hc[i].code, &cd[start]);

}

}

int main()

{

int i,m,n,w[N+1];

HTNode ht[M+1];

HTCode hc[N+1];

Init(hc, &n); // 初始化

HuffmanCoding(ht,hc,n); // 构造Huffman树，并形成字符的编码

for(i=1; i<=n; ++i)

printf("\n%c---%s",hc[i].data,hc[i].code);

printf("\n");

return 0;

}

[复制代码](javascript:void(0);)

# 数据库

## 关系型数据库

**关系型数据库：指采用了关系模型来组织数据的数据库。**  
**关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。**

### 关系模型中常用的概念

关系：一张二维表，每个关系都具有一个关系名，也就是表名  
元组：二维表中的一行，在数据库中被称为记录  
属性：二维表中的一列，在数据库中被称为字段  
域：属性的取值范围，也就是数据库中某一列的取值限制  
关键字：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组成  
关系模式：指对关系的描述。其格式为：关系名(属性1，属性2， ... ... ，属性N)，在数据库中成为表结构

### 关系型数据库的优点

**1.容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解  
2.使用方便：通用的SQL语言使得操作关系型数据库非常方便  
3.易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率**

### 关系型数据库存在的问题

**1.网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈  
2.网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的  
3.在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。当需要对数据库系统进行升级和扩展时，往往需要停机维护和数据迁移。  
4.性能欠佳：在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。**

### ****ACID特性****

**数据库事务必须具备ACID特性，ACID分别是Atomic原子性，Consistency一致性，**  
**Isolation隔离性，Durability持久性。**

**原子性**

**整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。**

**一致性**

**一个事务可以封装状态改变（除非它是一个只读的）。事务必须始终保持系统处于一致的状态，不管在任何给定的时间并发事务有多少。**

**也就是说：如果事务是并发多个，系统也必须如同串行事务一样操作。其主要特征是保护性和不变性(Preserving an Invariant)，以转账案例为例，假设有五个账户，每个账户余额是100元，那么五个账户总额是500元，如果在这个5个账户之间同时发生多个转账，无论并发多少个，比如在A与B账户之间转账5元，在C与D账户之间转账10元，在B与E之间转账15元，五个账户总额也应该还是500元，这就是保护性和不变性。**

**隔离性**

**隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。**

**持久性**

**在事务完成以后，该事务对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。**

**由于一项操作通常会包含许多子操作，而这些子操作可能会因为硬件的损坏或其他因素产生问题，要正确实现ACID并不容易。ACID建议数据库将所有需要更新以及修改的资料一次操作完毕，但实际上并不可行。**

**目前主要有两种方式实现ACID：第一种是Write ahead logging，也就是日志式的方式(现代数据库均基于这种方式)。第二种是Shadow paging。**

**相对于WAL（write ahead logging）技术，shadow paging技术实现起来比较简单，消除了写日志记录的开销恢复的速度也快(不需要redo和undo)。shadow paging的缺点就是事务提交时要输出多个块，这使得提交的开销很大，而且以块为单位，很难应用到允许多个事务并发执行的情况——这是它致命的缺点。**

**WAL 的中心思想是对数据文件的修改（它们是表和索引的载体）必须是只能发生在这些修改已经记录了日志之后 -- 也就是说，在日志记录冲刷到永久存储器之后。如果我们遵循这个过程，那么我们就不需要在每次事务提交的时候都把数据页冲刷到磁盘，因为我们知道在出现崩溃的情况下，我们可以用日志来恢复数据库：任何尚未附加到数据页的记录 都将先从日志记录中重做（这叫向前滚动恢复，也叫做 REDO） 然后那些未提交的事务做的修改将被从数据页中删除 （这叫向后滚动恢复 - UNDO）。**

### 当今十大主流的关系型数据库

[**Oracle**](https://www.oracle.com/database/index.html)**，**[**Microsoft SQL Server**](https://www.microsoft.com/en-us/sql-server/)**，**[**MySQL**](https://www.mysql.com/)**，**[**PostgreSQL**](https://www.postgresql.org/)**，**[**DB2**](https://www.ibm.com/analytics/us/en/db2/)**，**[**Microsoft Access**](https://products.office.com/zh-cn/access)**，** [**SQLite**](https://www.sqlite.org/)**，**[**Teradata**](https://www.teradata.com.cn/)**，**[**MariaDB**](https://mariadb.org/)**(MySQL的一个分支)，**[**SAP**](https://www.sap.com/)

## 非关系型数据库

NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。

### NoSQL数据库的四大分类

键值(Key-Value)存储数据库

这一类数据库主要会使用到一个哈希表，这个表中有一个特定的键和一个指针指向特定的数据。Key/value模型对于IT系统来说的优势在于简单、易部署。但是如果DBA只对部分值进行查询或更新的时候，Key/value就显得效率低下了。举例如：Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB.

列存储数据库。

这部分数据库通常是用来应对分布式存储的海量数据。键仍然存在，但是它们的特点是指向了多个列。这些列是由列家族来安排的。如：Cassandra, HBase, Riak.

文档型数据库

文档型数据库的灵感是来自于Lotus Notes办公软件的，而且它同第一种键值存储相类似。该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。文档型数据库可 以看作是键值数据库的升级版，允许之间嵌套键值。而且文档型数据库比键值数据库的查询效率更高。如：CouchDB, MongoDb. 国内也有文档型数据库SequoiaDB，已经开源。

图形(Graph)数据库

图形结构的数据库同其他行列以及刚性结构的SQL数据库不同，它是使用灵活的图形模型，并且能够扩展到多个服务器上。NoSQL数据库没有标准的查询语言(SQL)，因此进行数据库查询需要制定数据模型。许多NoSQL数据库都有REST式的数据接口或者查询API。如：Neo4J, InfoGrid, Infinite Graph.

因此，我们总结NoSQL数据库在以下的这几种情况下比较适用：1、数据模型比较简单；2、需要灵活性更强的IT系统；3、对数据库性能要求较高；4、不需要高度的数据一致性；5、对于给定key，比较容易映射复杂值的环境。

### 适用场景

NoSQL数据库在以下的这几种情况下比较适用：

1、数据模型比较简单；

2、需要灵活性更强的IT系统；

3、对数据库性能要求较高；

4、不需要高度的数据一致性；

5、对于给定key，比较容易映射复杂值的环境。

## 总结

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库 类型 | 特性 | 优点 | 缺点 |
| 关系型数据库 SQLite、Oracle、mysql | 1、关系型数据库，是指采用了关系模型来组织 数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型， 而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。 | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。 | 1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写； |
| 非关系型数据库 MongoDb、redis、HBase | 1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好； |

注1：数据库事务必须具备ACID特性，ACID是Atomic原子性，Consistency一致性，Isolation隔离性，Durability持久性。

注2：数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库。

## 相关资料

<https://blog.csdn.net/clover_lily/article/details/79991300>

# 范式

数据库范式是数据库设计中必不可少的知识，没有对范式的理解，就无法设计出高效率、优雅的数据库。甚至设计出错误的数据库。而想要理解并掌握范式却并不是那 么容易。教科书中一般以关系代数的方法来解释数据库范式。这样做虽然能够十分准确的表达数据库范式，但比较抽象，不太直观，不便于理解，更难以记忆。

## 基础知识

  理解范式，首先必须对知道什么是关系数据库，如果你不知道，我可以简单的不能再简单的说一下：关系数据库就是用二维表来保存数据。表和表之间可以……（省略10W字）。  
然后你应该理解以下概念：

* 实体：现实世界中客观存在并可以被区别的事物。比如“一个学生”、“一本书”、“一门课”等等。值得强调的是这里所说的“事物”不仅仅是看得见摸得着的“东西”，它也可以是虚拟的，不如说“老师与学校的关系”。
* 属性：教科书上解释为：“实体所具有的某一特性”，由此可见，属性一开始是个逻辑概念，比如说，“性别”是“人”的一个属性。在关系数据库中，属性又是个物理概念，属性可以看作是“表的一列”。
* 元组：表中的一行就是一个元组。
* 分量：元组的某个属性值。在一个关系数据库中，它是一个操作原子，即关系数据库在做任何操作的时候，属性是“不可分的”。否则就不是关系数据库了。
* 码：表中可以唯一确定一个元组的某个属性（或者属性组），如果这样的码有不止一个，那么大家都叫候选码，我们从候选码中挑一个出来做老大，它就叫主码。
* 全码：如果一个码包含了所有的属性，这个码就是全码。
* 主属性：一个属性只要在任何一个候选码中出现过，这个属性就是主属性。
* 非主属性：与上面相反，没有在任何候选码中出现过，这个属性就是非主属性。
* 外码：一个属性（或属性组），它不是码，但是它别的表的码，它就是外码。

## **6个范式**

好了，上面已经介绍了我们掌握范式所需要的全部基础概念，下面我们就来讲范式。首先要明白，范式的包含关系。一个数据库设计如果符合第二范式，一定也符合第一范式。如果符合第三范式，一定也符合第二范式…  
  
第一范式（1NF）

属性不可分  
在前面我们已经介绍了属性值的概念，我们说，它是“不可分的”。而第一范式要求属性也不可分。那么它和属性值不可分有什么区别呢？给一个例子：

|  |  |  |  |
| --- | --- | --- | --- |
| name | tel | | age |
| 大宝 | 13612345678 | | 22 |
| 小明 | 13988776655 | 010－1234567 | 21 |

Ps：这个表中，属性值“分”了。

|  |  |  |  |
| --- | --- | --- | --- |
| name | tel | | age |
| 手机 | 座机 |
| 大宝 | 13612345678 | 021－9876543 | 22 |
| 小明 | 13988776655 | 010－1234567 | 21 |

Ps：这个表中，属性 “分”了。  
这两种情况都不满足第一范式。不满足第一范式的数据库，不是关系数据库！所以，我们在任何关系数据库管理系统中，做不出这样的“表”来。  
  
第二范式（2NF）

符合1NF，并且，非主属性完全依赖于码。  
听起来好像很神秘，其实真的没什么。  
一 个候选码中的主属性也可能是好几个。如果一个主属性，它不能单独做为一个候选码，那么它也不能确定任何一个非主属性。给一个反例：我们考虑一个小学的教务 管理系统，学生上课指定一个老师，一本教材，一个教室，一个时间，大家都上课去吧，没有问题。那么数据库怎么设计？（学生上课表）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 学生 | 课程 | 老师 | 老师职称 | 教材 | 教室 | 上课时间 |
| 小明 | 一年级语文（上） | 大宝 | 副教授 | 《小学语文1》 | 101 | 14：30 |

一个学生上一门课，一定在特定某个教室。所以有（学生，课程）－>教室  
一个学生上一门课，一定是特定某个老师教。所以有（学生，课程）－>老师  
一个学生上一门课，他老师的职称可以确定。所以有（学生，课程）－>老师职称  
一个学生上一门课，一定是特定某个教材。所以有（学生，课程）－>教材  
一个学生上一门课，一定在特定时间。所以有（学生，课程）－>上课时间  
因此（学生，课程）是一个码。  
然而，一个课程，一定指定了某个教材，一年级语文肯定用的是《小学语文1》，那么就有课程－>教材。（学生，课程）是个码，课程却决定了教材，这就叫做不完全依赖，或者说部分依赖。出现这样的情况，就不满足第二范式！  
有什么不好吗？你可以想想：  
1、校长要新增加一门课程叫“微积分”，教材是《大学数学》，怎么办？学生还没选课，而学生又是主属性，主属性不能空，课程怎么记录呢，教材记到哪呢? ……郁闷了吧?(插入异常)  
2、下学期没学生学一年级语文（上）了，学一年级语文（下）去了，那么表中将不存在一年级语文（上），也就没了《小学语文1》。这时候，校长问：一年级语文（上）用的什么教材啊？……郁闷了吧?(删除异常)  
3、校长说：一年级语文（上）换教材，换成《大学语文》。有10000个学生选了这么课，改动好大啊！改累死了……郁闷了吧？（修改异常）  
那应该怎么解决呢？投影分解，将一个表分解成两个或若干个表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学生 | 课程 | 老师 | 老师职称 | 教室 | 上课时间 |
| 小明 | 一年级语文（上） | 大宝 | 副教授 | 101 | 14：30 |

学生上课表新

|  |  |
| --- | --- |
| 课程 | 教材 |
| 一年级语文（上） | 《小学语文1》 |

### 第三范式（3NF）

符合2NF，并且，消除传递依赖上面的“学生上课表新”符合2NF，可以这样验证：两个主属性单独使用，不用确定其它四个非主属性的任何一个。但是它有传递依赖！在哪呢？问题就出在“老师”和“老师职称”这里。一个老师一定能确定一个老师职称。有什么问题吗？想想：

1、老师升级了，变教授了，要改数据库，表中有N条，改了N次……（修改异常）  
2、没人选这个老师的课了，老师的职称也没了记录……（删除异常）  
3、新来一个老师，还没分配教什么课，他的职称记到哪？……（插入异常）  
那应该怎么解决呢？和上面一样，投影分解：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 学生 | 课程 | 老师 | 教室 | 上课时间 |
| 小明 | 一年级语文（上） | 大宝 | 101 | 14：30 |

|  |  |
| --- | --- |
| 老师 | 老师职称 |
| 大宝 | 副教授 |

### BC范式（BCNF）

符合3NF，并且，主属性不依赖于主属性  
若关系模式属于第一范式，且每个属性都不传递依赖于键码，则R属于BC范式。  
通常BC范式的条件有多种等价的表述：每个非平凡依赖的左边必须包含键码；每个决定因素必须包含键码。  
  
BC范式既检查非主属性，又检查主属性。当只检查非主属性时，就成了第三范式。满足BC范式的关系都必然满足第三范式。

还可以这么说：若一个关系达到了第三范式，并且它只有一个候选码，或者它的每个候选码都是单属性，则该关系自然达到BC范式。  
  
一般，一个数据库设计符合3NF或BCNF就可以了。在BC范式以上还有第四范式、第五范式。  
  
第四范式

要求把同一表内的多对多关系删除。  
  
第五范式

从最终结构重新建立原始结构。

但在绝大多数应用中不需要设计到这种程度。并且，某些情况下，过于范式化甚至会对数据库的逻辑可读性和使用效率起到阻碍。数据库中一定程度的冗余并不一定是坏事情。如果你对第四范式、第五范式感兴趣可以看一看专业教材，从头学起，并且忘记我说的一切，以免对你产生误导

数据库设计中的一些技巧

|  |
| --- |
| 1. 原始单据与实体之间的关系 　可以是一对一、一对多、多对多的关系。在一般情况下，它们是一对一的关系：即一张原始单据对应且只对应一个实体。在特殊情况下，它们可能是一对多或多对一的关系，即一张原始单证对应多个实体，或多张原始单证对应一个实体。这里的实体可以理解为基本表。明确这种对应关系后，对我们设计录入界面大有好处。 　　〖例1〗：一份员工履历资料，在人力资源信息系统中，就对应三个基本表：员工基本情况表、社会关系表、工作简历表。这就是“一张原始单证对应多个实体”的典型例子。 2. 主键与外键 　一般而言，一个实体不能既无主键又无外键。在E—R 图中, 处于叶子部位的实体, 可以定义主键，也可以不定义主键(因为它无子孙), 但必须要有外键(因为它有父亲)。 　　主键与外键的设计，在全局数据库的设计中，占有重要地位。当全局数据库的设计完成以后，有个美国数据库设计专家说：“键，到处都是键，除了键之外，什么也没有”，这就是他的数据库设计经验之谈，也反映了他对信息系统核心(数据模型)的高度抽象思想。因为：主键是实体的高度抽象，主键与外键的配对，表示实体之间的连接。 3. 基本表的性质 　　基本表与中间表、临时表不同，因为它具有如下四个特性： 　　 (1) 原子性。基本表中的字段是不可再分解的。 　　 (2) 原始性。基本表中的记录是原始数据（基础数据）的记录。 　　 (3) 演绎性。由基本表与代码表中的数据，可以派生出所有的输出数据。 　　 (4) 稳定性。基本表的结构是相对稳定的，表中的记录是要长期保存的。 　　理解基本表的性质后，在设计数据库时，就能将基本表与中间表、临时表区分开来。 4. 范式标准 　　基本表及其字段之间的关系, 应尽量满足第三范式。但是，满足第三范式的数据库设计，往往不是最好的设计。为了提高数据库的运行效率，常常需要降低范式标准：适当增加冗余，达到以空间换时间的目的。 　　〖例2〗：有一张存放商品的基本表，如表1所示。“金额”这个字段的存在，表明该表的设计不满足第三范式，因为“金额”可以由“单价”乘以“数量”得到，说明“金额”是冗余字段。但是，增加“金额”这个冗余字段，可以提高查询统计的速度，这就是以空间换时间的作法。 　　在Rose 2002中，规定列有两种类型：数据列和计算列。“金额”这样的列被称为“计算列”，而“单价”和“数量”这样的列被称为“数据列”。 　　表1 商品表的表结构 　　商品名称 商品型号 单价 数量 金额 　　电视机 29吋 2,500 40 100,000 　　 　　 5. 通俗地理解三个范式 　　通俗地理解三个范式，对于数据库设计大有好处。在数据库设计中，为了更好地应用三个范式，就必须通俗地理解三个范式(通俗地理解是够用的理解，并不是最科学最准确的理解)： 　　第一范式：1NF是对属性的原子性约束，要求属性具有原子性，不可再分解； 　　第二范式：2NF是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性； 　　第三范式：3NF是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余。 　　没有冗余的数据库设计可以做到。但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是：在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余。 　　 6. 要善于识别与正确处理多对多的关系 　　若两个实体之间存在多对多的关系，则应消除这种关系。消除的办法是，在两者之间增加第三个实体。这样，原来一个多对多的关系，现在变为两个一对多的关系。要将原来两个实体的属性合理地分配到三个实体中去。这里的第三个实体，实质上是一个较复杂的关系，它对应一张基本表。一般来讲，数据库设计工具不能识别多对多的关系，但能处理多对多的关系。 　　〖例3〗：在“图书馆信息系统”中，“图书”是一个实体，“读者”也是一个实体。这两个实体之间的关系，是一个典型的多对多关系：一本图书在不同时间可以被多个读者借阅，一个读者又可以借多本图书。为此，要在二者之间增加第三个实体，该实体取名为“借还书”，它的属性为：借还时间、借还标志(0表示借书，1表示还书)，另外，它还应该有两个外键(“图书”的主键，“读者”的主键)，使它能与“图书”和“读者”连接。 　　 7. 主键PK的取值方法 　　 PK是供程序员使用的表间连接工具，可以是一无物理意义的数字串, 由程序自动加1来实现。也可以是有物理意义的字段名或字段名的组合。不过前者比后者好。当PK是字段名的组合时，建议字段的个数不要太多，多了不但索引占用空间大，而且速度也慢。 　　 8. 正确认识数据冗余 　　主键与外键在多表中的重复出现, 不属于数据冗余，这个概念必须清楚，事实上有许多人还不清楚。非键字段的重复出现, 才是数据冗余！而且是一种低级冗余，即重复性的冗余。高级冗余不是字段的重复出现，而是字段的派生出现。 　　〖例4〗：商品中的“单价、数量、金额”三个字段，“金额”就是由“单价”乘以“数量”派生出来的，它就是冗余，而且是一种高级冗余。冗余的目的是为了提高处理速度。只有低级冗余才会增加数据的不一致性，因为同一数据，可能从不同时间、地点、角色上多次录入。因此，我们提倡高级冗余(派生性冗余)，反对低级冗余(重复性冗余)。 　　 9. E--R图没有标准答案 　　信息系统的E--R图没有标准答案，因为它的设计与画法不是惟一的，只要它覆盖了系统需求的业务范围和功能内容，就是可行的。反之要修改E--R图。尽管它没有惟一的标准答案，并不意味着可以随意设计。好的E—R图的标准是：结构清晰、关联简洁、实体个数适中、属性分配合理、没有低级冗余。 　　 10. 视图技术在数据库设计中很有用 　　与基本表、代码表、中间表不同，视图是一种虚表，它依赖数据源的实表而存在。视图是供程序员使用数据库的一个窗口，是基表数据综合的一种形式, 是数据处理的一种方法，是用户数据保密的一种手段。为了进行复杂处理、提高运算速度和节省存储空间, 视图的定义深度一般不得超过三层。 若三层视图仍不够用, 则应在视图上定义临时表, 在临时表上再定义视图。这样反复交迭定义, 视图的深度就不受限制了。 　　对于某些与国家政治、经济、技术、军事和安全利益有关的信息系统，视图的作用更加重要。这些系统的基本表完成物理设计之后，立即在基本表上建立第一层视图，这层视图的个数和结构，与基本表的个数和结构是完全相同。并且规定，所有的程序员，一律只准在视图上操作。只有数据库管理员，带着多个人员共同掌握的“安全钥匙”，才能直接在基本表上操作。请读者想想：这是为什么？ 　　 11. 中间表、报表和临时表 　　中间表是存放统计数据的表，它是为数据仓库、输出报表或查询结果而设计的，有时它没有主键与外键(数据仓库除外)。临时表是程序员个人设计的，存放临时记录，为个人所用。基表和中间表由DBA维护，临时表由程序员自己用程序自动维护。 　　 12. 完整性约束表现在三个方面 　　域的完整性：用Check来实现约束，在数据库设计工具中，对字段的取值范围进行定义时，有一个Check按钮，通过它定义字段的值城。 　　参照完整性：用PK、FK、表级触发器来实现。 　　用户定义完整性：它是一些业务规则，用存储过程和触发器来实现。 　　 13. 防止数据库设计打补丁的方法是“三少原则” 　　 (1) 一个数据库中表的个数越少越好。只有表的个数少了，才能说明系统的E--R图少而精，去掉了重复的多余的实体，形成了对客观世界的高度抽象，进行了系统的数据集成，防止了打补丁式的设计； 　　 (2) 一个表中组合主键的字段个数越少越好。因为主键的作用，一是建主键索引，二是做为子表的外键，所以组合主键的字段个数少了，不仅节省了运行时间，而且节省了索引存储空间； 　　 (3) 一个表中的字段个数越少越好。只有字段的个数少了，才能说明在系统中不存在数据重复，且很少有数据冗余，更重要的是督促读者学会“列变行”，这样就防止了将子表中的字段拉入到主表中去，在主表中留下许多空余的字段。所谓“列变行”，就是将主表中的一部分内容拉出去，另外单独建一个子表。这个方法很简单，有的人就是不习惯、不采纳、不执行。 　　数据库设计的实用原则是：在数据冗余和处理速度之间找到合适的平衡点。“三少”是一个整体概念，综合观点，不能孤立某一个原则。该原则是相对的，不是绝对的。“三多”原则肯定是错误的。试想：若覆盖系统同样的功能，一百个实体(共一千个属性) 的E--R图，肯定比二百个实体(共二千个属性) 的E--R图，要好得多。 　　提倡“三少”原则，是叫读者学会利用数据库设计技术进行系统的数据集成。数据集成的步骤是将文件系统集成为应用数据库，将应用数据库集成为主题数据库，将主题数据库集成为全局综合数据库。集成的程度越高，数据共享性就越强，信息孤岛现象就越少，整个企业信息系统的全局E—R图中实体的个数、主键的个数、属性的个数就会越少。 　　提倡“三少”原则的目的，是防止读者利用打补丁技术，不断地对数据库进行增删改，使企业数据库变成了随意设计数据库表的“垃圾堆”，或数据库表的“大杂院”，最后造成数据库中的基本表、代码表、中间表、临时表杂乱无章，不计其数，导致企事业单位的信息系统无法维护而瘫痪。 　　 “三多”原则任何人都可以做到，该原则是“打补丁方法”设计数据库的歪理学说。“三少”原则是少而精的原则，它要求有较高的数据库设计技巧与艺术，不是任何人都能做到的，因为该原则是杜绝用“打补丁方法”设计数据库的理论依据。 　　 14. 提高数据库运行效率的办法 　　在给定的系统硬件和系统软件条件下，提高数据库系统的运行效率的办法是： 　　 (1) 在数据库物理设计时，降低范式，增加冗余, 少用触发器, 多用存储过程。 　　 (2) 当计算非常复杂、而且记录条数非常巨大时(例如一千万条)，复杂计算要先在数据库外面，以文件系统方式用C++语言计算处理完成之后，最后才入库追加到表中去。这是电信计费系统设计的经验。 　　 (3) 发现某个表的记录太多，例如超过一千万条，则要对该表进行水平分割。水平分割的做法是，以该表主键PK的某个值为界线，将该表的记录水平分割为两个表。若发现某个表的字段太多，例如超过八十个，则垂直分割该表，将原来的一个表分解为两个表。 　　 (4) 对数据库管理系统DBMS进行系统优化，即优化各种系统参数，如缓冲区个数。 　　 (5) 在使用面向数据的SQL语言进行程序设计时，尽量采取优化算法。 　　总之，要提高数据库的运行效率，必须从数据库系统级优化、数据库设计级优化、程序实现级优化，这三个层次上同时下功夫。 　　上述十四个技巧，是许多人在大量的数据库分析与设计实践中，逐步总结出来的。对于这些经验的运用，读者不能生帮硬套，死记硬背，而要消化理解，实事求是，灵活掌握。并逐步做到：在应用中发展，在发展中应用 |

# C++语法

## 友元

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend。**

class Box

{

double width;

public:

double length;

friend void printWidth( Box box );

void setWidth( double wid );

};

### 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元函数不能被继承；友元函数没有this指针；友元函数破坏了类的封装性。

#include <iostream>

using namespace std;

class Box

{

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// 成员函数定义

void Box::setWidth( double wid )

{

width = wid;

}

// 请注意：printWidth() 不是任何类的成员函数

void printWidth( Box box )

{

/\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/

cout << "Width of box : " << box.width <<endl;

}

// 程序的主函数

int main( )

{

Box box;

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth( box );

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Width of box : 10

#### **友元函数的使用**

因为友元函数没有this指针，则参数要有三种情况：

1.要访问非static成员时，需要对象做参数；

2.要访问static成员或全局变量时，则不需要对象做参数；

3.如果做参数的对象是全局对象，则不需要对象做参数.

4.可以直接调用友元函数，不需要通过对象或指针

## C++基类和派生类指针的相互赋值和转换

在公有派生的情况下，派生类的[指针](http://c.biancheng.net/c/80/)和引用可以直接赋值给基类指针。但即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有而派生类中有的成员。基类的指针不能赋值给派生类的指针。但是通过强制类型转换，也可以将基类指针强制转换成派生类指针后再赋值给派生类指针。只是在这种情况下，程序员需要保证被转换的基类指针本来就指向一个派生类的对象，这样才是安全的，否则很容易出错。

#include <iostream>

using namespace std;

class shape

{public:

        virtual int area()=0;

};

class rectangle:public shape

{public:

        int a, b;

        void setLength (int x, int y) {a=x;b=y;}

        int area() {return a\*b;}

};

若有语句定义rectangle r; r.setLength(3,5); 则编译时无语法错误的语句是（   AB   ）

A shape \*s1=&r;

B shape &s2=r;

C shape s3=r;

D shape s4[3];

## lambda函数

lambda函数的用处：https://www.cnblogs.com/lidabo/p/3908663.html

微软官方解释：<https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2017>

lambda在c++14后开始可以支持泛型

## 构造函数

### 构造函数是干什么的

该类对象被创建时，编译系统对象分配内存空间，并自动调用该构造函数，由构造函数完成成员的初始化工作，故：构造函数的作用：初始化对象的数据成员。

### 构造函数的种类

1 class Complex

2 {

3

4 private :

5 double m\_real;

6 double m\_imag;

7

8 public:

9

10 // 无参数构造函数

11 // 如果创建一个类你没有写任何构造函数,则系统会自动生成默认的无参构造函数，函数为空，什么都不做

12 // 只要你写了一个下面的某一种构造函数，系统就不会再自动生成这样一个默认的构造函数，如果希望有一个这样的无参构造函数，则需要自己显示地写出来

13 Complex(void)

14 {

15 m\_real = 0.0;

16 m\_imag = 0.0;

17 }

18

19 // 一般构造函数（也称重载构造函数）

20 // 一般构造函数可以有各种参数形式,一个类可以有多个一般构造函数，前提是参数的个数或者类型不同（基于c++的重载函数原理）

21 // 例如：你还可以写一个 Complex( int num)的构造函数出来

22 // 创建对象时根据传入的参数不同调用不同的构造函数

23 Complex(double real, double imag)

24 {

25 m\_real = real;

26 m\_imag = imag;

27 }

28

29 // 复制构造函数（也称为拷贝构造函数）

30 // 复制构造函数参数为类对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在对象的数据成员的值复制一份到新创建的对象中

31 // 若没有显示的写复制构造函数，则系统会默认创建一个复制构造函数，但当类中有指针成员时，由系统默认创建该复制构造函数会存在风险，具体原因请查询有关 “浅拷贝” 、“深拷贝”的文章论述

32 Complex(const Complex & c)

33 {

34 // 将对象c中的数据成员值复制过来

35 m\_real = c.m\_real;

36 m\_img = c.m\_img;

37 }

38

39 // 类型转换构造函数，根据一个指定的类型的对象创建一个本类的对象

40 // 例如：下面将根据一个double类型的对象创建了一个Complex对象

41 Complex::Complex(double r)

42 {

43 m\_real = r;

44 m\_imag = 0.0;

45 }

46

47 // 等号运算符重载

48 // 注意，这个类似复制构造函数，将=右边的本类对象的值复制给等号左边的对象，它不属于构造函数，等号左右两边的对象必须已经被创建

49 // 若没有显示的写=运算符重载，则系统也会创建一个默认的=运算符重载，只做一些基本的拷贝工作

50 Complex &operator=(const Complex &rhs)

51 {

52 // 首先检测等号右边的是否就是左边的对象本，若是本对象本身,则直接返回

53 if ( this == &rhs )

54 {

55 return \*this;

56 }

57

58 // 复制等号右边的成员到左边的对象中

59 this->m\_real = rhs.m\_real;

60 this->m\_imag = rhs.m\_imag;

61

62 // 把等号左边的对象再次传出

63 // 目的是为了支持连等 eg: a=b=c 系统首先运行 b=c

64 // 然后运行 a= ( b=c的返回值,这里应该是复制c值后的b对象)

65 return \*this;

66 }

67 };

下面使用上面定义的类对象来说明各个构造函数的用法：

[复制代码](javascript:void(0);)

1 void main()

2 {

3 // 调用了无参构造函数，数据成员初值被赋为0.0

4 Complex c1，c2;

5

6 // 调用一般构造函数，数据成员初值被赋为指定值

7 Complex c3(1.0,2.5);

8 // 也可以使用下面的形式

9 Complex c3 = Complex(1.0,2.5);

10

11 // 把c3的数据成员的值赋值给c1

12 // 由于c1已经事先被创建，故此处不会调用任何构造函数

13 // 只会调用 = 号运算符重载函数

14 c1 = c3;

15

16 // 调用类型转换构造函数

17 // 系统首先调用类型转换构造函数，将5.2创建为一个本类的临时对象，然后调用等号运算符重载，将该临时对象赋值给c1

18 c2 = 5.2;

19

20 // 调用拷贝构造函数( 有下面两种调用方式)

21 Complex c5(c2);

22 Complex c4 = c2; // 注意和 = 运算符重载区分,这里等号左边的对象不是事先已经创建，故需要调用拷贝构造函数，参数为c2

23

24 }

### 复制构造函数

几个原则：

C++ primer p406 ：复制构造函数是一种特殊的构造函数，具有单个形参，该形参（常用const修饰）是对该类类型的引用。当定义一个新对象并用一个同类型的对象对它进行初始化时，将显示使用复制构造函数。当该类型的对象传递给函数或从函数返回该类型的对象时，将隐式调用复制构造函数。

 C++支持两种初始化形式：复制初始化（int a = 5;）和直接初始化（int a(5);）对于其他类型没有什么区别，对于类类型直接初始化直接调用实参匹配的构造函数，复制初始化总是调用复制构造函数，也就是说：

A x(2);　　//直接初始化，调用构造函数  
A y = x;　　//复制初始化，调用复制构造函数

 必须定义复制构造函数的情况：

只包含类类型成员或内置类型（但不是指针类型）成员的类，无须显式地定义复制构造函数也可以复制；有的类有一个数据成员是指针，或者是有成员表示在构造函数中分配的其他资源，这两种情况下都必须定义复制构造函数。

什么情况使用复制构造函数：

类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：  
（1）一个对象以值传递的方式传入函数体   
（2）一个对象以值传递的方式从函数返回   
（3）一个对象需要通过另外一个对象进行初始化。

### 深拷贝和浅拷贝

所谓浅拷贝，指的是在对象复制时，只对对象中的数据成员进行简单的赋值，默认拷贝构造函数执行的也是浅拷贝。在“深拷贝”的情况下，对于对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间

如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝

上面提到，如果没有自定义复制构造函数，则系统会创建默认的复制构造函数，但系统创建的默认复制构造函数只会执行“浅拷贝”，即将被拷贝对象的数据成员的值一一赋值给新创建的对象，若该类的数据成员中有指针成员，则会使得新的对象的指针所指向的地址与被拷贝对象的指针所指向的地址相同，delete该指针时则会导致两次重复delete而出错。下面是示例：

[复制代码](javascript:void(0);)

1 #include <iostream.h>

2 #include <string.h>

3 class Person

4 {

5 public :

6

7 // 构造函数

8 Person(char \* pN)

9 {

10 cout << "一般构造函数被调用 !\n";

11 m\_pName = new char[strlen(pN) + 1];

12 //在堆中开辟一个内存块存放pN所指的字符串

13 if(m\_pName != NULL)

14 {

15 //如果m\_pName不是空指针，则把形参指针pN所指的字符串复制给它

16 strcpy(m\_pName ,pN);

17 }

18 }

19

20 // 系统创建的默认复制构造函数，只做位模式拷贝

21 Person(Person & p)

22 {

23 //使两个字符串指针指向同一地址位置

24 m\_pName = p.m\_pName;

25 }

26

27 ~Person( )

28 {

29 delete m\_pName;

30 }

31

32 private :

33 char \* m\_pName;

34 };

35

36 void main( )

37 {

38 Person man("lujun");

39 Person woman(man);

40

41 // 结果导致 man 和 woman 的指针都指向了同一个地址

42

43 // 函数结束析构时

44 // 同一个地址被delete两次

45 }

46

47

48 // 下面自己设计复制构造函数，实现“深拷贝”，即不让指针指向同一地址，而是重新申请一块内存给新的对象的指针数据成员

49 Person(Person & chs);

50 {

51 // 用运算符new为新对象的指针数据成员分配空间

52 m\_pName=new char[strlen(p.m\_pName)+ 1];

53

54 if(m\_pName)

55 {

56 // 复制内容

57 strcpy(m\_pName ,chs.m\_pName);

58 }

59

60 // 则新创建的对象的m\_pName与原对象chs的m\_pName不再指向同一地址了

61 }

### 重载赋值操作符

通过定义operate=的函数，可以对赋值进行定义。像其他任何函数一样，操作符函数有一个返回值和形参表。形参表必须具有与该操作符操作数书目相同的形参（如果操作符是一个成员，则包括隐式this形参）。赋值是二元运算，所以该操作符函数有两个形参：第一个形参（隐含的this指针）对应着左操作数，第二个形参对应右操作数。

 一个应用了对赋值号重载的拷贝构造函数的例子：

[复制代码](javascript:void(0);)

1 #include <iostream>

2

3 using namespace std;

4

5 class A

6 {

7 public:

8 A(int);//构造函数

9 A(const A &);//拷贝构造函数

10 ~A();

11 void print();

12 int \*point;

13 A &operator=(const A &);

14 };

15

16 A::A(int p)

17 {

18 point = new int;

19 \*point = p;

20 }

21

22 A::A(const A &b)

23 {

24 \*this = b;

25 cout<<"调用拷贝构造函数"<<endl;

26 }

27

28 A::~A()

29 {

30 delete point;

31 }

32

33 void A::print()

34 {

35 cout<<"Address:"<<point<<" value:"<<\*point<<endl;

36 }

37

38 A &A::operator=(const A &b)

39 {

40 if( this != &b)

41 {

42 delete point;

43 point = new int;

44 \*point = \*b.point;

45 }

46 }

47

48

49 int main()

50 {

51 A x(2);

52 A y = x;

53 x.print();

54 delete x.point;

55 y.print();

56

57 return 0;

58 }

### 关于构造函数私有的问题

构造函数私有只是不想在外部直接定义对象出来，并不是说不可以定义对象。**考虑实现一个单例类的例子：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class A{  public:      void show( ){          cout<<val<<endl;      }      static A\* GetInstance( )      {          static A instance;          return &instance;      }  private:      A() { }      static int val;  };  int A::val = 0; |

## 析构函数

### 定义

作用：对象消亡时，自动被调用，用来释放对象占用的空间  
特点:  
  1.函数名是在类名前加上~，无参数且无返回值。

2.一个类只能有且有一个析构函数，如果没有显式的定义，系统会生成一个缺省的析构函数（合成析构函数）。

3.析构函数不能重载。每有一次构造函数的调用就会有一次析构函数的调用。

在构造函数中，成员的在初始化是在函数体执行前完成的，并按照成员在类中出现的顺序进行初始化，而在析构函数中，首先执行函数体，然后再销毁成员，并且成员按照初始化的逆序进行销毁。

我们一直在说析构函数的作用是在你的类对象离开作用域后释放对象使用的资源，并销毁成员。那么到底这里所说的销毁到底是什么？那么继续往下看：

void test ()

{

int a=10;

int b=20;

}

回想我们在一个函数体内定义一个变量的情况，在test函数中定义了a和b两个变量，那么在出这个函数之后，a和b就会被销毁（栈上的操作）。那么如果是是一个指向动态开辟的一块空间的指针，我们都知道需要自己进行free，否则会造成内存泄漏。

说到这里，其实在类里面的情况和这是一样的，这就是合成析构函数体为空的原因，函数并不需要做什么，当类对象出作用域时系统会释放你的内置类型的那些成员。但是像上面说的一样，如果，我的成员里有一个指针变量并且指向了一块你动态开辟的内存，那么像以前那样也需要自己来释放，此时就需要在析构函数内部写你的释放代码，这样在调用析构函数的时候就可以把你所有的资源进行释放。（其实这才是析构函数有用的地方，对吗）

那么还有一点，当类类型对象的成员还有一个类类型对象，那么在析构函数里也会调用这个对象的析构函数。

### 析构函数来阻止该类型对象被销毁

我们如果不想要析构函数来对对象进行释放该怎么做呢，不显式的定义显然是不行的，因为编译器会生成默认的合成析构函数。之前我们知道了如果想让系统默认生成自己的构造函数可以利用default，那么其实还有一个东西叫做delete。

class Date

{

public:

Date(int year=1990,int month=1,int day=1)

: \_year(year),\_month(month), \_day(day)

{ }

~Date() = delete;

private:

int \_year=1990;

int \_month;

int \_day;

};

如果我这么写了，又在底下创建Date类型的对象，那么这个对象将是无法被销毁的，其实编译器并不允许这么做，直接会给我们报错。

但其实是允许我们动态创建这个类类型对象的，像这样：Date\* p = new Date;虽然这样是可行的，但当你delete p的时候依然会出错。

## 多态

多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。

C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

下面的实例中，基类 Shape 被派生为两个类，如下所示：

实例

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape{

public:

Rectangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape{

public:

Triangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// 程序的主函数

int main( )

{

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// 存储矩形的地址

shape = &rec;

// 调用矩形的求面积函数 area

shape->area();

// 存储三角形的地址

shape = &tri;

// 调用三角形的求面积函数 area

shape->area();

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Parent class area

Parent class area

导致错误输出的原因是，调用函数 area() 被编译器设置为基类中的版本，这就是所谓的静态多态，或静态链接 - 函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为 area() 函数在程序编译期间就已经设置好了。

但现在，让我们对程序稍作修改，在 Shape 类中，area() 的声明前放置关键字 virtual，如下所示：

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

virtual int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

修改后，当编译和执行前面的实例代码时，它会产生以下结果：

Rectangle class area

Triangle class area

此时，编译器看的是指针的内容，而不是它的类型。因此，由于 tri 和 rec 类的对象的地址存储在 \*shape 中，所以会调用各自的 area() 函数。

正如您所看到的，每个子类都有一个函数 area() 的独立实现。这就是多态的一般使用方式。有了多态，您可以有多个不同的类，都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。

### 虚函数

虚函数 是在基类中使用关键字 virtual 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要**静态链接**到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为**动态链接**，或**后期绑定**。

### 纯虚函数

您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

我们可以把基类中的虚函数 area() 改写如下：

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

= 0 告诉编译器，函数没有主体，上面的虚函数是纯虚函数。

这里面有一个误区，有人认为，virtual f()=0这种纯虚函数语法就是没有定义体的语义。

其实，这是不对的。这种语法只是表明这个函数是一个纯虚函数，因此这个类变成了抽象类，不能产生对象。我们完全可以为纯虚函数指定函数体 。通常的纯虚函数不需要函数体，是因为我们一般不会调用抽象类的这个函数，只会调用派生类的对应函数。这样，我们就有了一个纯虚析构函数的函数体，上面的代码需要改成：

http://images.csdn.net/syntaxhighlighting/OutliningIndicators/None.gifclass Base  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedBlockStart.gif{  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/InBlock.gifpublic:  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/InBlock.gif   Base()  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedSubBlockStart.gif   {  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedSubBlockEnd.gif   }  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/InBlock.gif   virtual ~Base() = 0; //pure virtual  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedBlockEnd.gif};  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/None.gif  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/None.gifBase::~Base()//function body  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedBlockStart.gif{  
http://images.csdn.net/syntaxhighlighting/OutliningIndicators/ExpandedBlockEnd.gif}

从语法角度来说，不可以将上面的析构函数直接写入类声明中（内联函数的写法）。这或许是一个不正交化的地方。但是这样做的确显得有点累赘

将一个函数定义为纯虚函数，实际上是将这个类定义为抽象类，不能实例化对象。

### 虚函数的定义要遵循以下重要规则

1.如果虚函数在基类与派生类中出现，仅仅是名字相同，而形式参数不同，或者是返回类型不同，那么即使加上了virtual关键字，也是不会进行滞后联编的。   
  
2.只有类的成员函数才能说明为虚函数，因为虚函数仅适合用与有继承关系的类对象，所以普通函数不能说明为虚函数。   
  
3.静态成员函数不能是虚函数,因为静态成员函数的特点是不受限制于某个对象。   
  
4.内联(inline)函数不能是虚函数，因为内联函数不能在运行中动态确定位置。即使虚函数在类的内部定义定义，但是在编译的时候系统仍然将它看做是非内联的。   
  
5.构造函数不能是虚函数，因为构造的时候，对象还是一片位定型的空间，只有构造完成后，对象才是具体类的实例。   
  
6.析构函数可以是虚函数,而且通常声名为虚函数。（析构函数可以是纯虚的，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的。）

### 从内存角度看虚函数

https://www.cnblogs.com/cxq0017/p/6074247.html

# 大端小端

**1)大端模式：Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。（其实大端模式才是我们直观上认为的模式，和字符串存储的模式差类似）拿16进制数0x12345678举例：**

低地址 --------------------> 高地址  
0x12  |  0x34  |  0x56  |  0x78

**2)小端模式：Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。**

低地址 --------------------> 高地址  
0x78  |  0x56  |  0x34  |  0x12

**3)下面是两个具体例子：**

16bit宽的数0x1234在Little-endian模式（以及Big-endian模式）CPU内存中的存放方式（假设从地址0x4000开始存放）为：

|  |  |  |
| --- | --- | --- |
| **内存地址** | **小端模式存放内容** | **大端模式存放内容** |
| **0x4000** | **0x34** | **0x12** |
| **0x4001** | **0x12** | **0x34** |

32bit宽的数0x12345678在Little-endian模式以及Big-endian模式）CPU内存中的存放方式（假设从地址0x4000开始存放）为：

|  |  |  |
| --- | --- | --- |
| **内存地址** | **小端模式存放内容** | **大端模式存放内容** |
| **0x4000** | **0x78** | **0x12** |
| **0x4001** | **0x56** | **0x34** |
| **0x4002** | **0x34** | **0x56** |
| **0x4003** | **0x12** | **0x78** |

**4)大端小端没有谁优谁劣，各自优势便是对方劣势：**

小端模式 ：强制转换数据不需要调整字节内容，1、2、4字节的存储方式一样。  
大端模式 ：符号位的判定固定为第一个字节，容易判断正负。

三、数组在大端小端情况下的存储：

　　以unsigned int value = 0x12345678为例，分别看看在两种字节序下其存储情况，我们可以用unsigned char buf[4]来表示value：  
　　Big-Endian: 低地址存放高位，如下：  
高地址  
        ---------------  
        buf[3] (0x78) -- 低位  
        buf[2] (0x56)  
        buf[1] (0x34)  
        buf[0] (0x12) -- 高位  
        ---------------  
        低地址  
Little-Endian: 低地址存放低位，如下：  
高地址  
        ---------------  
        buf[3] (0x12) -- 高位  
        buf[2] (0x34)  
        buf[1] (0x56)  
        buf[0] (0x78) -- 低位  
        --------------  
低地址

# 压栈出栈

先进后出，

1、可以全进再依次出，

2、也可以进一个出一个，

3、也可以进一部分，出一个，再进一部分

承接上一篇--O(1)实现“求取栈中最小值”，今天介绍栈的压入、弹出顺序相关的一个小puzzle

压入、弹出栈顺序

输入两个整数序列，第一个序列表示压入顺序，判断第二个序列是否为栈的弹出顺序。（假设栈中所有元素均不相同）

例：1、2、3、4、5是栈的压入顺序

那么，4、5、3、2、1可能是该栈的一个弹出序列

4、3、5、1、2则不可能是栈的弹出序列

值得引起注意的是：栈的压入顺序仅仅指压入的顺序，并不care其它操作，类似的理解弹出顺序

如果让我们直接用大脑来分析，这个问题其实很好解决，拿着第二个序列，对照第一个序列（即压入顺序），按照数字顺序逐个分析很快就能得到答案

更简单的办法是借助一个辅助栈：将第二个序列的数字按照栈操作的先后顺序要求压入辅助栈中

那么我们该如何将这一简单的思路转化为程序思维呢？

**思路分析**

压入顺序：1、2、3、4、5

以例子中的第一个弹出顺序4、5、3、2、1来进行分析：

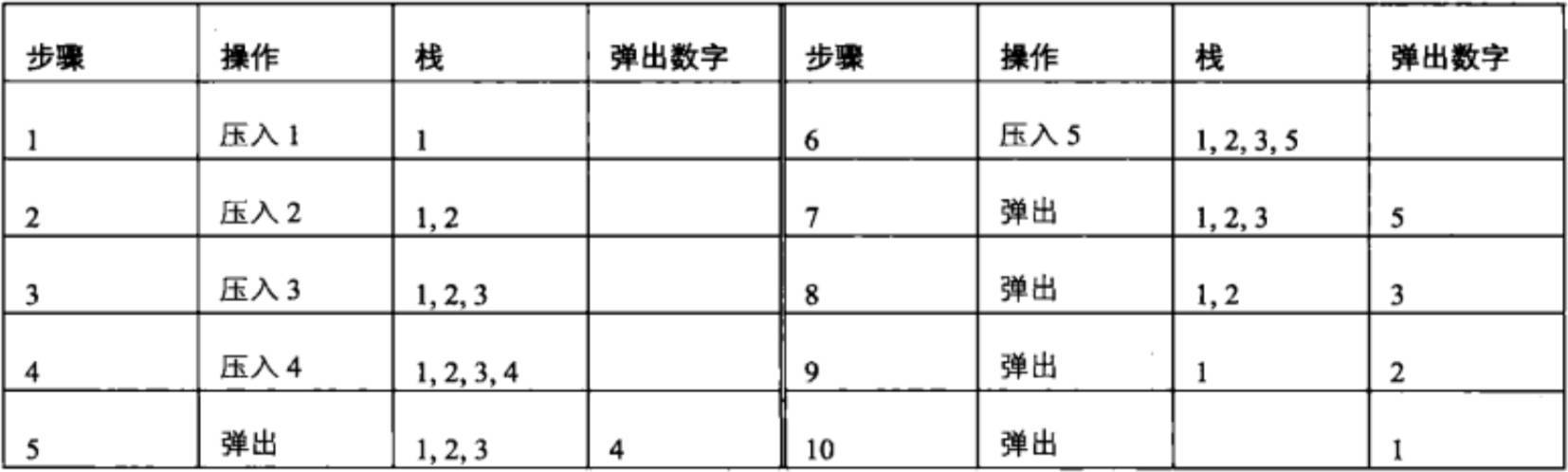
a. 第一个弹出数字为4，依据栈FILO特性，想要弹出4，必须先将它前面的数字都入栈：1、2、3、4

b. 第二个弹出数字为5，则在上一步的弹出4之后，紧接着压入5，再弹出5

c. 第三个数字是3，上一步之后栈内元素只剩1、2、3，直接弹出即可

d. 接下来的两个数字弹出情况与c类似

上述流程图示为：



可以得出这个弹出顺序是压入栈的弹出顺序

第二个例子：4、3、5、1、2弹出顺序的判定

a. 第一个弹出数字的处理与第一例一样

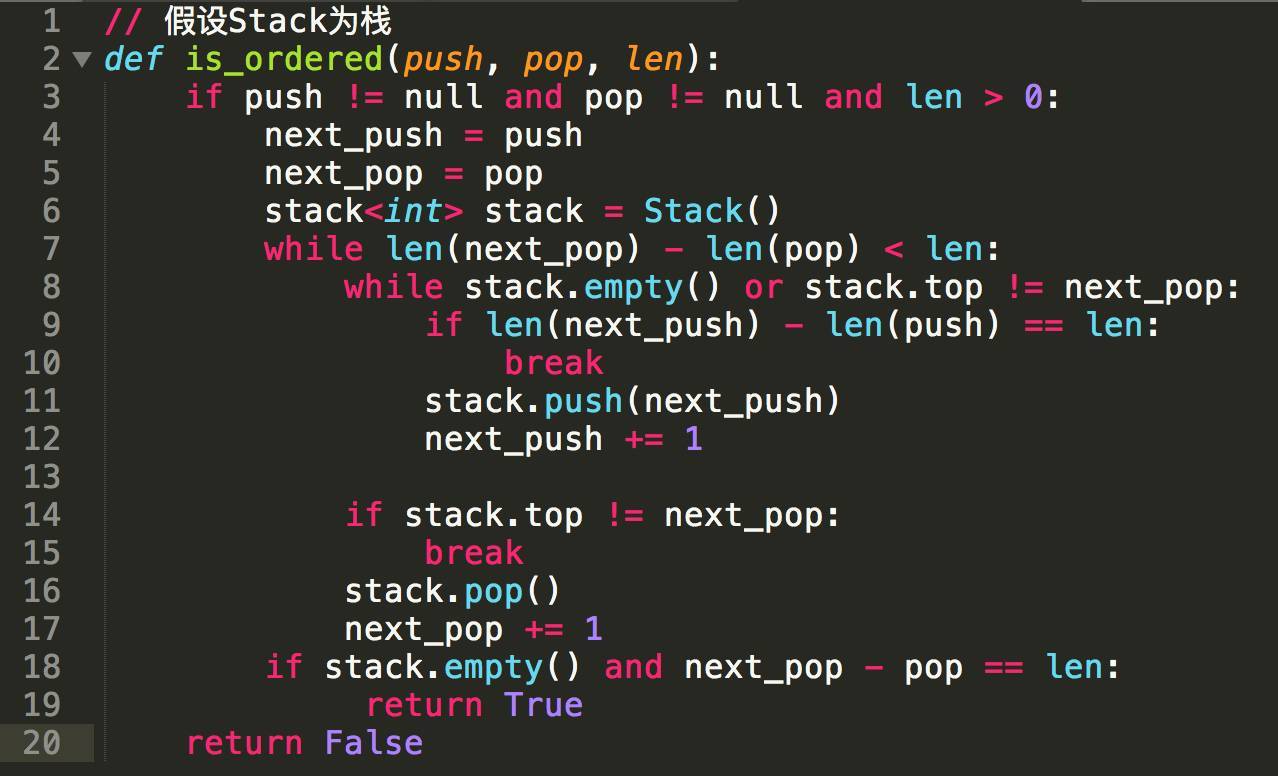
b. 第二个弹出数字是3，处理与上一例中的c类似

c. 第三个数字5情况也与上一步b类似

d. 第四个弹出数字是1，但实际此时栈内的内容是1、2，无法在2之前弹出1

可以得出这个弹出顺序并不是符合要求的弹出顺序(图略)

**伪代码**



# 网络编程

## 传输层

### TCP

TCP 详解（TCP机制）

https://blog.csdn.net/sinat\_36629696/article/details/80740678

TCP协议详解

<https://www.cnblogs.com/buxiangxin/p/8336022.html>

五分钟读懂TCP 协议——TCP协议简介

<https://blog.csdn.net/ningdaxing1994/article/details/73076795>

### 保证可靠性的机制

校验和

序列号(按序到达)

确认应答

超时重传

连接管理

流量控制

拥塞控制

### 提高性能的机制

滑动窗口

快速重传

延迟应答

捎带应答

### 定时器

超时重传定时器

保活定时器

TIME\_WAIT定时器

### UDP

## 应用层

### DNS

主要用UDP，但是当请求字节过长超过512字节时用TCP协议，将其分割成多个片段传输。

DNS协议默认端口号是53。

操作系统的DNS缓存：windows DNS缓存的默认值是 MaxCacheTTL，它的默认值是86400s，也就是一天。macOS 严格遵循DNS协议中的TTL。

**DNS（域名解析协议）详解：**https://blog.csdn.net/baidu\_37964071/article/details/80500825

**DNS协议详解及报文格式分析：**

https://blog.csdn.net/tianxuhong/article/details/74922454

# Linux

## 进程

### fork之后，子进程从父进程那继承了什么

#### 子进程继承父进程

* 用户号UIDs和用户组号GIDs
* 环境(表)Environment

https://blog.csdn.net/oqqHuTu12345678/article/details/75040810

环境表介绍https://www.cnblogs.com/kele-dad/p/9124047.html

* 堆栈
* 共享内存
* 打开文件的描述符(socket)
* 执行时关闭（Close-on-exec）标志
* 信号（Signal）控制设定
* 进程组号
* 当前工作目录
* 根目录
* 文件方式创建屏蔽字
* 资源限制
* 控制终端

 子进程独有

* 进程号PID
* 不同的父进程号
* 自己的文件描述符和目录流的拷贝
* 子进程不继承父进程的进程正文（text），数据和其他锁定内存（memory locks）
* 不继承异步输入和输出

 父进程和子进程拥有独立的地址空间和PID参数。

 子进程从父进程继承了用户号和用户组号，用户信息，目录信息，环境（表），打开的文件描述符，堆栈，（共享）内存等。

 经过fork()以后，父进程和子进程拥有相同内容的代码段、数据段和用户堆栈，就像父进程把自己克隆了一遍。事实上，父进程只复制了自己的PCB块。而代码段，数据段和用户堆栈内存空间并没有复制一份，而是与子进程共享。只有当子进程在运行中出现写操作时，才会产生中断，并为子进程分配内存空间。由于父进程的PCB和子进程的一样，所以在PCB中断中所记录的父进程占有的资源，也是与子进程共享使用的。这里的“共享”一词意味着“竞争”

## 线程

## fork，vfork，clone

<https://blog.csdn.net/gatieme/article/details/51417488>

## 文件系统

**软件常见23种设计模式**

## 创建型

### Singleton

单例模式

保证一个类只有一个实例，并提供一个访问它的全局访问点

### Abstract Factory

抽象工厂

提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类。

### Factory Method

工厂方法

定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到了子类。

### Builder

建造模式

将一个复杂对象的构建与他的表示相分离，使得同样的构建过程可以创建不同的表示。

Prototype

原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型来创建新的对象。

## 行为型

### Iterator

迭代器模式

提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。

Observer

观察者模式

定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知自动更新。

### Template Method

模板方法

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

Command

命令模式

将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。

### State

状态模式

允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类。

### Strategy

策略模式

定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户。

### Chain of Responsibility

职责链模式

使多个对象都有机会处理请求，从而避免请求的送发者和接收者之间的耦合关系

### Mediator

中介者模式

用一个中介对象封装一些列的对象交互。

Visitor

访问者模式

表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作。

### Interpreter

解释器模式

给定一个语言，定义他的文法的一个表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

### Memento

备忘录模式

在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

## 结构型

### Composite

组合模式

将对象组合成树形结构以表示部分整体的关系，Composite使得用户对单个对象和组合对象的使用具有一致性。

### Facade

外观模式

为子系统中的一组接口提供一致的界面，facade提供了一高层接口，这个接口使得子系统更容易使用。

### Proxy

代理模式

为其他对象提供一种代理以控制对这个对象的访问

### Adapter

适配器模式

将一类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。

### Decrator

装饰模式

动态地给一个对象增加一些额外的职责，就增加的功能来说，Decorator模式相比生成子类更加灵活。

### Bridge

桥模式

将抽象部分与它的实现部分相分离，使他们可以独立的变化。

### Flyweight

享元模式