**C++编程语言部分**

《C++ primer》 第十二章 动态内存

1. 智能指针的循环引用问题
2. **class** B;
3. **class** A
4. {
5. **public**:
6. shared\_ptr<B> m\_b;
7. };
9. **class** B
10. {
11. **public**:
12. shared\_ptr<A> m\_a;
13. };
15. **int** main()
16. {
17. **while** (**true**)
18. {
19. shared\_ptr<A> a(**new** A); //new出来的A的引用计数此时为1
20. shared\_ptr<B> b(**new** B); //new出来的B的引用计数此时为1
21. a->m\_b = b; //B的引用计数增加为2
22. b->m\_a = a; //A的引用计数增加为2
23. }
24. //b先出作用域，B的引用计数减少为1，不为0，所以堆上的B空间没有被释放，且B持有
25. //的A也没有机会被析构，A的引用计数也完全没减少。
26. //a后出作用域，同理A的引用计数减少为1，不为0，所以堆上A的空间也没有被释放。
27. }

所以在使用基于引用计数的智能指针时，要特别小心循环引用带来的内存泄漏，循环引用不只是两方的情况，只要引用链成环都会出现问题。当然循环引用本身就说明设计上可能存在一些问题，如果特殊原因不得不使用循环引用，那可以让引用链上的一方持用普通指针（或弱智能指针weak\_ptr）即可。 weak\_ptr通常与shared\_ptr配合使用，shared\_ptr是指该指针对资源拥有所有权，而weak\_ptr没有所有权，对象的生命周期与弱指针没有关联。

《C++ primer》 第十三章 拷贝控制

1. 拷贝初始化何时发生？

使用=初始化变量时；将一个对象作为实参传递给非引用类型的形参；从一个返回类型为非引用类型的函数返回一个对象；用花括号列表初始化一个数组中的元素或一个聚合类中的成员；向一个容器中添加新元素时。

1. 何时调用析构函数？

变量在离开其作用域时被销毁；当一个对象被销毁时，其成员被销毁；容器被销毁时，其元素被销毁；对于动态分配的对象，当对指向它的指针应用delete运算符时被销毁；对于临时对象，当创建它的完整表达式结束时被销毁。

1. 编译器何时将合成的拷贝控制成员定义为删除的？

类的某个成员的析构函数是删除的或不可访问的（private），则类的合成析构函数和合成拷贝构造函数被定义为删除的；如果类的某个成员的析构函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的；类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个const的或引用成员，则类的合成拷贝赋值运算符被定义为删除的；类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器，或是类有一个const成员，它没有类内初始化器且器类型未显示定义默认构造函数，则该类的默认构造函数被定义为删除的。

**总结：如果一个类有数据成员不能默认构造、构造、复制或销毁，则对应的成员函数将被定义为删除的。**

1. 编译器何时将合成的移动操作定义为删除的函数？

移动操作永远不会被隐式定义为删除的函数。但是如果我们显示地要求编译器生成=default的移动操作，且编译器不能移动所有成员，则编译会将移动操作定义为删除的函数。具体有以下几种情况。

有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数，移动赋值运算符情况类似；有类成员的移动操作被定义为删除的或不可访问的，则类的移动操作被定义为删除的；类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的；有类成员是const的或是引用，则类的移动赋值运算符被定义为删除的。

**注意：定义了一个移动构造函数和移动赋值运算符的类必须定义自己的拷贝操作，否则，这些成员默认地被定义为删除的。**

《C++ primer》 第十五章 面向对象程序设计

1. 简述存在继承关系的类型之间的转换规则？

从派生类向基类的类型转换只针对指针或引用类型有效；基类向派生类不存在隐士类型的转换；和任何其他成员一样，派生类向基类的类型转换也可能由于访问受限而不可行；通常将一个派生类对象拷贝、移动或赋值给一个基类对象时，只处理派生类对象中的基类部分。

1. 介绍一个类中受保护成员的作用？

和私有成员类似，受保护成员对于类的用户来说是不可访问的；和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的；派生类的成员和友元通过类对象的形式访问成员时，只能访问派生类对象中基类受保护的部分，而不能访问普通基类对象中的受保护成员。

1. 简述派生类向基类转换的可访问性？

假设D继承自B。（1）当D公有继承B时，用户代码才能使用派生类向基类的转换；（2）无论D如何继承B，D的成员函数和友元都能使用派生类向基类的转换；（3）D公有继承或受保护继承B，D的派生类成员和友元可以使用D向B的类型转换；否则不能使用。

1. 派生类中删除的拷贝控制与基类的关系是什么？

如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的或是不可访问的，则派生类中对应的成员是被删除的；如果基类中有一个不可访问的或者是删掉的析构函数，则派生类中中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分；当使用=default请求一个移动操作时，如果基类对应的操作是删除或不可访问的，则派生类的移动构造函数也将是删除的。

1. 虚函数表问题

关于虚函数表，大概浏览下这篇文章，要注意的是这篇文章中有一个错误，类里面保存的是虚函数表的地址，所以关于虚函数的地址，应该是(int \*)\*(int \*)(&b)，第一个函数地址应该是(int \*)\*((int \*)\*(int \*)(&b))，可以在编译器上验证一下。文章地址：

<https://blog.csdn.net/haoel/article/details/1948051/>

虚函数表里面存的地址应该独立于对象，放在静态存储区中，因为所有的类对象共享同一个虚函数表，在比较对象的大小时，如果类里面有虚函数，应考虑上这个虚表的4字节大小的指针，具体的类对象的大小问题可以浏览下面的内容。

<https://bbs.csdn.net/topics/390308992>

<https://blog.csdn.net/longjialin93528/article/details/80160467>

更详细的讲解：

<https://www.jianshu.com/p/46acf45ee795>

隐藏成员如this指针不影响其他部分的字节对齐，那么虚表指针4字节应该按照额外添加来计算（存疑）

<https://www.cnblogs.com/sunbines/p/9257981.html>

《C++ primer》 第十六章 模板与泛型编程

1. 模板参数的类型转换

能在调用中应用于函数模板的自动类型转换只有以下两种情况。

（1）const转换：可以将一个非const对象的引用或指针传递给一个const的引用或指针形参；（2）数组或函数指针转换：如果函数的形参不是引用类型，可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。类似的，一个函数实参可以转换为一个该函数类型的指针。

其他的类型转换，如算术转换、派生类向基类的转换以及用户定义的转换都不能应用于函数模板。

1. 关于函数模板接受右值引用的函数参数

如果通过类型别名或模板类型参数间接创建了一个引用的引用，这些引用会形成折叠。从而会导致以下两个重要结果。

如果一个函数参数是一个指向模板类型参数的右值引用（T &&），则它可以被绑定到一个左值；并且，如果实参是一个左值，则推断出的模板实参类型将是一个左值引用，而且函数参数将被实例化为一个左值引用参数（T &）。

一般编程实践中，为了避免普通类型和引用类型受到影响，通常需要进行重载，比如以下的方式

template<typename T> void f(T &&);//绑定到非const右值。

template<typename T> void f(const T &);//左值和const右值。

1. 函数模板的匹配规则

(1)对于一个调用，其候选函数包括所有的模板实参推断成功的函数模板实例；

(2)候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板；

(3)与往常一样，可行函数（模板与非模板）按类型转换来排序。但可用于函数模板调用的类型转换非常有限；

(4)与往常一样，如果恰好有一个函数提供了比任何其他函数都更好的匹配，则选择此函数，但是如果有多个函数提供同样好的匹配，则根据以下几个规则：同样好的函数中只有一个是非模板函数，则选择此函数；同样好的函数没有非模板函数，而有多个模板，如果其中一个模板比其他模板更加特例化，选择此模板；除此外，此调用是有歧义的。

4.

**数据结构**

1. 简要描述红黑树

红黑树采用适度平衡，可大致表述为：任意节点左右子树的高度，相差不得超过两倍。

为了保证红黑树的平衡特性，组成红黑树的二叉搜索树满足如下特点：

（1）树根始终为黑色；（2）外部节点均为黑色；（3）其余节点若为红色，则其孩子节点比为黑色，同样的，父节点也为黑色；（4）从任意外部节点到根节点的沿途，黑节点数目相等，从任意节点通往其任一后代的外部节点沿途，黑节点总数亦必相等。以上特点保证了红黑树最短节点高度一定大于全树高的1/2,即任意节点左右子树的高度相差不得超过两倍。

红黑树插入节点调整：

红黑树插入一个节点后，则将新节点刷红，这样可能导致与父节点同为红色，需要做双红修正。情况1：插入节点的父节点的兄弟节点为黑色，需要互换爷爷节点和紧邻节点的颜色，做一次旋转，即可完成重平衡。情况2：插入节点的父节点的兄弟节点为红色，则需要将红节点转黑，黑节点转红，这时会引发上溢，做递归双红修正即可。

红黑树删除节点调整：

红黑树删除一个节点后，如果实际删除的节点和接替的节点都为黑色，则可能局部失衡，需要做双黑修正，此时需要考察被删除节点的兄弟节点s。情况1：s为黑色且有红孩子，做一次旋转，s为重平衡后的根节点并继承之前的颜色，s的两个孩子转为黑色。情况2：s为黑色且其孩子均为黑色，s父节点p为红色，则将s转红，其父节点p转黑。情况3：s为黑节点且其孩子均为黑色，s父节点p为黑色，将s转红，局部恢复但引发上溢，对p递归作双黑修正。情况4：s节点为红色，s转黑，父节点p转红，做一次右旋，对p进行双黑修正，此时会转为情况1和情况2。

红黑树在节点插入和节点删除操作上均能保证常数次的调整，而递归修正也能保证不超过O（logn），而AVL树只能保证节点插入的常数次调整，节点删除需要O（logn）次的旋转操作以解决失衡传播的问题。

1. 堆排序

（1）Floyd建堆

对于一个vector中的所有数据，看作是完全二叉树的各个节点，首先找到根据二叉树层次遍历排序的第一个内部节点内部节点，从这个内部节点开始，对每个内部节点进行下滤操作，最终可以得到一个完全二叉堆。因为一个完全二叉树中，越向下节点越多，所以先调整下面的节点，时间复杂度为o（n）。

（2）下滤操作

如果删除了一个堆顶元素，往往将堆尾元素替换到堆顶位置，则堆序可能被破坏。为了重新恢复堆序，则将堆顶元素与其（最多）两个孩子中的大者相比较，如果大于其孩子，则交换二者元素，下降一层后，还有可能不满足堆序，则继续与其新孩子进行比较，直到堆序满足为止。同样的，如果插入一个元素，一般插入在堆尾，可能会不满足堆序，这时候要做上滤操作，与其父亲元素比较大小以及元素交换，直到满足堆序。

（3）堆排序

堆排序不需要额外辅助空间，可以实现就地排序。一般堆会维持一个堆元素的数量，这样可以把堆分为两个部分，排序和未排序的部分。首先对vector中的所有元素实行建堆操作，然后循环执行删除堆顶操作，将堆顶元素放到向量的尾端，这样不断扩大已排序部分，缩小未排序部分，直到堆空，这样就实现了排序。

1. **class** Solution
2. {
3. **private**:
4. **bool** inheap(**int** len, **int** index)
5. {
6. **return** (index > -1) && (index < len);
7. }
8. **bool** parentvalid(**int** index)
9. {
10. **return** (index>0);
11. }
12. **bool** lchildvalid(**int** len,**int** index)
13. {
14. **return** inheap(len, lchild(index));
15. }
16. **bool** rchildvalid(**int** len, **int** index)
17. {
18. **return** inheap(len, rchild(index));
19. }
20. **int** parent(**int** index)
21. {
22. **return** (index - 1) >> 1;
23. }
24. **int** lastinternal(**int** len)
25. {
26. **return** parent(len - 1);
27. }
28. **int** lchild(**int** index)
29. {
30. **return** ((index << 1) + 1);
31. }
32. **int** rchild(**int** index)
33. {
34. **return** ((1 + index) << 1);
35. }
36. **int** properparent(vector<**int**> &vec,**int** len,**int** index)
37. {
38. **if** (rchildvalid(len, index))
39. {
40. **int** left = lchild(index), right = rchild(index);
41. **return** (vec[index] < vec[left]) ? (vec[left] < vec[right] ? right : left) : (vec[index] < vec[right] ? right : index);
42. }
43. **else** **if** (lchildvalid(len, index))
44. {
45. **int** left = lchild(index);
46. **return** (vec[index] < vec[left]) ? left : index;
47. }
48. **else**
49. {
50. **return** index;
51. }
52. }
54. **private**:
55. **int** percolatedown(vector<**int**> &vec, **int** len, **int** index)
56. {
57. **int** j = -1;
58. **while** (index != (j = properparent(vec, len, index)))
59. {
60. swap(vec[index], vec[j]);
61. index = j;
62. }
63. **return** index;
64. }
65. **void** heapify(vector<**int**> &vec)
66. {
67. **int** len = vec.size();
68. **for** (**int** i = lastinternal(len); inheap(len, i); --i)
69. {
70. percolatedown(vec, len, i);
71. }
72. }
73. **int** delmax(vector<**int**> &vec, **int** len)
74. {
75. **int** maxNum = vec[0];
76. vec[0] = vec[--len];
77. percolatedown(vec, len, 0);
78. **return** maxNum;
79. }
81. **public**:
82. **void** heapsort(vector<**int**> &vec)
83. {
84. **int** heapLen = vec.size(); //heapLen记录堆的长度。
85. heapify(vec);
86. **while** (heapLen != 0)
87. {
88. **int** maxNum = delmax(vec, heapLen);
89. vec[--heapLen] = maxNum;
90. }
91. }
92. };
93. 快速排序

快速排序采用分治策略，通过快速划分算法取得轴点位置，然后以轴点为界，分别递归的对前后子向量实施快速排序。快速划分算法通过随机选择候选轴点，然后比较每个元素，得到真正的轴点位置并返回。

分治策略高效实现有三个必要条件：子问题划分的高效性；子问题相互之间的独立性；子任务的规模接近。快排有时难以保证第三点，所以快排在最坏情况下的时间复杂度为o(n^2)，但在大多数情况下，快速排序平均效率依然可以达到o(nlogn)，而且较之其它排序算法，其时间复杂度中的常系数更小。

1. 归并排序

归并排序采用分治策略，一般作二路归并，以中点为界，划分两个子向量，分别作归并排序，直到总体有序。归并算法可采取自顶而下的递归，也可以自底而上的循环，时间复杂度为o(nlogn)，空间复杂度o(logn)，而对于链表结构，循环实现归并可以实现常数空间复杂度，链表使用归并实现高效的排序，具体方法可以参考leetcode题目148：sort list。

1. 希尔排序

希尔排序是插入排序的改进，又称为缩小增量排序。实质上是一种分组插入的方法，取一个小于数组长度的步长，将一个待排序的数组分成若干个子数组，所有下标之间的距离为步长的倍数的数字在一个分组中。然后对各组内的元素直接插入排序，这一趟排序完成后，每个组的元素都是有序的，然后减小步长的值，重复执行上述的分组和排序操作，直到步长为1，使整个数组整体有序。

希尔排序的时间复杂度与增量(即步长)的选取有关。例如，当增量为1时，希尔排序退化成了直接插入排序，此时的时间复杂度为o()，而Hibbard增量的希尔排序的时间复杂度为o()。希尔排序是不稳定的算法。

1. //希尔排序
2. **class** Solution
3. {
4. **public**:
5. //对单个组的插入排序，i是分组的起始位置，gap是步长。
6. **void** groupSort(vector<**int**> &vec, **int** i, **int** gap)
7. {
8. **int** len = vec.size();
9. **for** (**int** j = i + gap; j < len; ++j)
10. {
11. **if** (vec[j] < vec[j - gap])
12. {
13. **int** temp = vec[j];
14. **int** k = j - gap;
15. **while** (k >= 0 && vec[k]>temp)
16. {
17. vec[k + gap] = vec[k];
18. k -= gap;
19. }
20. vec[k + gap] = temp;
21. }
22. }
23. }
24. **void** shellSort(vector<**int**> &vec)
25. {
26. **int** len = vec.size();
27. **for** (**int** gap = len / 2; gap > 0; gap /= 2)
28. {
29. **for** (**int** i = 0; i < gap; ++i)
30. groupSort(vec, i, gap);
31. }
32. }
33. };
34. 基数排序

基数排序是桶排序的扩展，将整数按位数切割为不同的数字，然后按照每个位数分别比较排序。具体做法是：将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列。

1. //基数排序
2. **class** Solution
3. {
4. **public**:
5. //对数字的每个位桶排序。
6. **void** countSort(vector<**int**> &vec, **int** exp)
7. {
8. **int** len = vec.size();
9. vector<**int**> output(len);
10. vector<**int**> bucket(10, 0);//表示0-9这十个数字。
11. **int** i = 0;
12. //存储数字出现的次数。
13. **for** (i = 0; i < len; ++i)
14. ++bucket[(vec[i] / exp) % 10];
15. //将次数转换为索引值。
16. **for** (i = 1; i < 10; ++i)
17. bucket[i] += bucket[i - 1];
18. //将数据存到output数组合适的位置中。
19. **for** (i = len - 1; i >= 0; --i)
20. {
21. //vec[i]所在的相对位置应该保持不变。
22. output[bucket[(vec[i] / exp) % 10] - 1] = vec[i];
23. --bucket[(vec[i] / exp) % 10];
24. }
25. **for** (i = 0; i < len; ++i)
26. vec[i] = output[i];
27. }
28. **int** getMax(vector<**int**> &vec)
29. {
30. **int** len = vec.size();
31. **int** maxNum = vec[0];
32. **for** (**int** i = 1; i < len; ++i)
33. maxNum = max(maxNum, vec[i]);
34. **return** maxNum;
35. }
36. //基数排序。
37. **void** radixSort(vector<**int**> &vec)
38. {
39. **int** maxNum = getMax(vec);
40. **for** (**int** exp = 1; maxNum / exp>0; exp \*= 10)
41. countSort(vec, exp);
42. }
43. };
44. 排序算法总结



排序算法的选择需要考虑多种因素，平均时间复杂度低的算法不一定是最优的。有时平均时间复杂度高的算法可能更适合某些特殊情况。同时，选择算法还应考虑可读性，以利于软件的维护。一般而言需要考虑四点因素：待排序的记录数目n的大小；记录本身数据量的大小，也就是记录中除关键字外其他信息量的大小；关键字的结构及其分布情况；对排序稳定性的要求。

设待排序的记录数目为n，分析以下具体情况。

(1)当n较大，应采用时间复杂度为o(nlogn)的排序方法，比如快速排序，归并排序，和堆排序。快速排序是目前基于比较的内部排序中最好的方法，当待排序的关键字随机分布，且不要求稳定性，快速排序的平均时间最短。归并排序的稳定性最好，对于海量数据可以存储在外存中，并且要求稳定性，选择归并排序。堆排序的优势在于空间复杂度最低，对于十分在乎内存使用量的环境，选择堆排序。

(2)当n较小，一般采用简单的排序算法，比如选择排序，插入排序。当元素分布有序时，可以采用插入排序，可大大减少比较次数和移动记录的次数。选择排序中，元素的移动次数相对较少，对不要求稳定性的待排元素来说，元素本身数据量大，可以使用选择排序。

(3)基数排序是一种稳定的排序算法，但有一定的局限性：关键字可分解；记录的关键字位数较少，如果密集更好；如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。

1. 二叉搜索树的简单实现

最简单的二叉树结点结构：

1. **struct** TreeNode
2. {
3. **int** val;
4. **struct** TreeNode \*left;
5. **struct** TreeNode \*right;
6. TreeNode(**int** x) :
7. val(x), left(NULL), right(NULL) {}
8. };

二叉搜索树查找，插入，删除操作：

1. **class** Solution
2. {
3. **public**:
4. TreeNode \*& search(TreeNode \*&pNode, **int** num) //查找。
5. {
6. **if** (!pNode || pNode->val == num) **return** pNode;
7. **return** search((pNode->val > num ? pNode->left : pNode->right), num);
8. }
9. TreeNode \* insert(TreeNode \*pRoot, **int** num) //插入。
10. {
11. TreeNode \*&node = search(pRoot, num);
12. **if** (node) **return** node;
13. node = **new** TreeNode(num);
14. **return** node;
15. }
16. TreeNode \* remove(TreeNode \*pRoot, **int** num) //删除。
17. {
18. TreeNode \*&node = search(pRoot, num);
19. **if** (!node) **return** nullptr;
20. **return** removeAt(node);
21. }
22. TreeNode \* removeAt(TreeNode \*&pNode)
23. {
24. TreeNode \*delnode = pNode;
25. TreeNode \*succnode = nullptr;
26. **if** (!pNode->left)
27. {
28. succnode = pNode = pNode->right;
29. }
30. **else** **if** (!pNode->right)
31. {
32. succnode = pNode = pNode->left;
33. }
34. **else**
35. {
36. TreeNode \*mem = delnode;
37. delnode = delnode->right;
38. **while** (delnode->left)
39. {
40. mem = delnode;
41. delnode = delnode->left;
42. }
43. swap(pNode->val, delnode->val);
44. ((mem == pNode) ? mem->right : mem->left) = succnode = delnode->right;
45. }
46. **delete** delnode;
47. **return** succnode;
48. }
49. };

二叉搜索树查找的迭代实现：

1. TreeNode \*& search(TreeNode \*&pNode, **int** num) //查找（迭代实现）。
2. {
3. **if** (!pNode || pNode->val == num) **return** pNode;//退化情况，在树根处命中。
4. TreeNode \*hot = pNode;
5. **while** (**true**)
6. {
7. TreeNode \*&c = (num < hot->val) ? hot->left : hot->right;
8. **if** (!c || c->val == num) **return** c;//命中返回或深入一层。
9. hot = c;//hot始终指向最后一个失败节点。
10. }
11. }
12. B树

可将大数据集组织为B树存放于外存，对于活跃的B树，根节点常驻内存，此外，任何时刻通常只有另一节点留驻于内存。对于一个m阶B树，各个内部节点的分支数最少为（m+1）/2,最多为m，关键码应该比分支少一个。

B树的查找算法共需访问o() 个节点，相应的需要做o()次外存读取操作，多路搜索树将极其耗时的的I/O操作次数大致缩减为常规平衡二叉搜索树的1/，提高实际访问效率。

关键码的插入。首先通过查找得到合适的插入位置，将关键码插入到对应位置，同时创建一个空子树指针。由于节点内增加了一个关键码，节点内关键码的总数可能会发生上溢，需要做分裂处理。m阶搜索树上溢时关键码的数量应该正好为m，这时以m/2为轴点，新建一个节点，将轴点右侧的孩子和关键码添加到新节点当中。将轴点位置的关键码上升到父节点中，同时父节点将新建的节点添加到孩子中。由于父节点增加了一个关键码，可能造成上溢传播，所以应该递归做分裂处理。

关键码的删除。首先通过查找确定关键码的所在的节点，然后找出关键码在节点中的秩。如果该节点非叶子节点，那么需要不断寻找关键码的后继，在右子树中不断向左直到某叶节点，然后关键码与其交换，删除掉该关键码和一个空孩子节点。删除一个关键码后，可能会造成分支的数量少于(m+1)/2，这时出现了下溢，需要做合并处理。发生下溢时，下溢的节点刚好包含(m+1)/2-1个分支，需要处理三种情况：（1）下溢节点的左兄弟存在，而且至少包含(m+1)/2个节点，则其分支多一个，可以借给下溢节点，下溢节点向父亲借一个关键码，父亲再向左兄弟借一个关键码。(2)左兄弟为空或节点数不足，而右兄弟满足条件。下溢节点向父亲借一个关键码，父亲再向右兄弟借一个关键码。(3)左兄弟或者右兄弟不存在，或者节点数都不足。由于左右兄弟一定会存在一个，所以先假定左兄弟存在，则下溢节点向父亲借一个关键码，然后与左兄弟合并成一个节点。如果左兄弟不存在，则与有右兄弟合并。

1. 求一个二叉树中任意两个节点间的最大距离

这个题目与leetcode 124: binary tree maximum path sum 异曲同工，基本思想都是通过树的后序遍历得到每一个节点的最大高度，然后通过这个值求出当前的路径最大值。

1. **class** Solution
2. {
3. **public**:
4. **int** findMaxDis(TreeNode \*root)
5. {
6. **if** (!root) **return** 0;
7. **int** maxLength = INT\_MIN;
8. travPost(root, maxLength);
9. **return** maxLength;
10. }
11. **int** travPost(TreeNode \*node, **int** &maxLength)
12. {
13. **if** (!node) **return** 0;
14. **int** l = 0, r = 0;
15. l = travPost(node->left, maxLength);
16. r = travPost(node->right, maxLength);
17. maxLength = max(maxLength, l + r + 1);
18. **return** 1 + max(l, r);
19. }
20. };
21. 判断一棵二叉树是否为完全二叉树

对于一颗完全二叉树采用层次遍历，从根节点开始，入队列，如果队列不为空，循环。遇到第一个没有左孩子或者右孩子的节点，设置标志位，如果之后再遇到有左/右孩子的节点，那么这不是一颗完全二叉树。这个方法需要遍历整棵树，复杂度为O(N)，N为节点的总数。

1. **class** Solution {
2. **public**:
3. //设置标志位，意为已经到达最后一个内部节点或第一个叶节点，剩下的节点不能再有孩子
4. //节点，否则不是完全二叉树。
5. **bool** leftMost = **false**;
6. queue<TreeNode \*> q;
7. **bool** isCompleteBinarytree(TreeNode \*root)
8. {
9. **if** (!root) **return** **true**;
10. q.push(root);
11. **while** (!q.empty())
12. {
13. TreeNode \*temp = q.front();
14. q.pop();
15. **if** (!ProcessChild(temp->left))
16. {
17. **return** **false**;
18. }
19. **if** (!ProcessChild(temp->right))
20. {
21. **return** **false**;
22. }
23. }
24. **return** **true**;
25. }
26. **bool** ProcessChild(TreeNode \*child)
27. {
28. **if** (child)
29. {
30. **if** (!leftMost)
31. {
32. q.push(child);
33. }
34. **else**
35. {
36. **return** **false**;
37. }
38. }
39. **else**
40. {
41. leftMost = **true**;
42. }
43. **return** **true**;
44. }
45. };
46. 最长公共子串和最长公共子序列问题

(1)最长公共子串问题要求得到两个字符串的公共子字符串，这个子字符串在原串中是连续的。例如：

输入abcdefghijklmnop和abcsafjklmnopqrstuvw

输出jklmnop

note: 若有多个，输出在较短串中最先出现的那个。

1. **class** Solution
2. {
3. **public**:
4. string LCSubstring(string s1, string s2)
5. {
6. **if** (s1.size() > s2.size()) swap(s1, s2);
7. **int** len1 = s1.size(), len2 = s2.size();
8. **int** maxLen = 0, start = 0;
9. vector<vector<**int**>> dp(len1 + 1, vector<**int**>(len2 + 1, 0));
10. **for** (**int** i = 1; i <= len1; ++i)
11. {
12. **for** (**int** j = 1; j <= len2; ++j)
13. {
14. **if** (s1[i - 1] == s2[j - 1])
15. {
16. dp[i][j] = dp[i - 1][j - 1] + 1;
17. **if** (dp[i][j] > maxLen)
18. {
19. maxLen = dp[i][j];
20. start = i - maxLen;
21. }
22. }
23. }
24. }
25. **return** s1.substr(start, maxLen);
26. }
27. };

(2)最长公共子序列则是指两个字符串的公共子串在原串中可以是不连续的。

例如：“abcfbc”和“abfcab”，其中“abc”同时出现在两个字符串中，因此“abcb”是它们的最长公共子序列。此外，“ab”、“af”等都是它们的子序列。

输入 abcfbc和abfcab

输出 4

1. **class** Solution {
2. **public**:
3. **int** LCSequence(string word1, string word2)
4. {
5. **int** len1 = word1.size(), len2 = word2.size();
6. vector<vector<**int**>> dp(len1 + 1, vector<**int**>(len2 + 1, 0));
7. **for** (**int** i = 1; i <= len1; ++i)
8. {
9. **for** (**int** j = 1; j <= len2; ++j)
10. {
11. **if** (word1[i - 1] == word2[j - 1])
12. dp[i][j] = dp[i - 1][j - 1] + 1;
13. **else**
14. dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
15. }
16. }
17. **return** dp[len1][len2];
18. }
19. };
20. Union Find算法：并查集

参考文章：并查集(Union-Find)算法介绍

<https://blog.csdn.net/dm_vincent/article/details/7655764>

1. 最长重复子串(LeetCode 1044)

该题目可以使用后缀法，简单容易理解，但该方法内存占用过高，无法通过。所以使用了二分查找+ Rabin-Karp算法，该题目的原理部分可以参考官方题解：

[**https://leetcode-cn.com/problems/longest-duplicate-substring/solution/zui-chang-zhong-fu-zi-chuan-by-leetcode/**](https://leetcode-cn.com/problems/longest-duplicate-substring/solution/zui-chang-zhong-fu-zi-chuan-by-leetcode/)

1. **class** Solution {
2. **public**:
3. string longestDupSubstring(string S)
4. {
5. **int** len = S.size();
6. vector<**int**> v;
7. **for** (**int** i = 0; i < len; ++i) v.push\_back(S[i] - 'a');
8. **int** cnt = 26;
9. //假设重复字符串长度最短为1，最长为len,执行二分搜索。
10. **int** lo = 1, hi = len;
11. **int** subLen = 0;
12. **while** (lo < hi)
13. {
14. subLen = lo + ((hi - lo) >> 1);
15. **if** (searchCommon(subLen, cnt, len, v, S) != -1)
16. lo = subLen + 1;
17. **else**
18. hi = subLen;
19. }
20. **int** start = searchCommon(lo - 1, cnt, len, v, S);
21. **return** start != -1 ? S.substr(start, lo - 1) : "";
22. }
23. **int** searchCommon(**int** subLen, **int** cnt, **int** len, vector<**int**> &v,string &S)
24. {
25. **int** prime = pow(10, 9) + 7;
26. **long** **long** RK = 0;
27. **for** (**int** i = 0; i < subLen; ++i)
28. RK = (RK\*cnt + v[i]) % prime;
29. unordered\_map<**int**, vector<**int**>> m;
30. m[RK] = vector<**int**>(1, 0);
31. **long** **long** aL = 1;
32. **for** (**int** i = 0; i < subLen; ++i)
33. aL = (aL \* cnt) % prime;
34. **for** (**int** start = 1; start < len - subLen + 1; ++start)
35. {
36. RK = (RK \* cnt - v[start - 1] \* aL % prime + prime) % prime;
37. RK = (RK + v[start + subLen - 1]) % prime;
38. //可能会出现不同子串相同编码值的情况，所以要比较一下。
39. **if** (m.count(RK))
40. {
41. **for** (auto it : m[RK])
42. {
43. **if** (S.substr(it, subLen) == S.substr(start, subLen))
44. **return** start;
45. }
46. m[RK].push\_back(start);
47. }
48. **else**
49. {
50. m[RK] = vector<**int**>(1, start);
51. }
52. }
53. **return** -1;
54. }
55. };
56. 01背包问题

有n件物品和一个容量是c的背包。每件物品只能使用一次。第i件物品的体积是wi，价值是vi。求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

1. //自上而下记忆法
2. **class** Solution1
3. {
4. **public**:
5. **int** knapSack0\_1(vector<**int**> &w, vector<**int**> &v, **int** c)
6. {
7. **if** (w.empty() || v.empty() || c == 0) **return** 0;
8. **int** len = w.size();
9. vector<vector<**int**>> memo(len, vector<**int**>(c + 1, -1));
10. **return** helper(w, v, memo, len - 1, c);
11. }
12. **int** helper(vector<**int**> &w, vector<**int**> &v, vector<vector<**int**>> &memo, **int** index, **int** c)
13. {
14. **if** (index < 0 || c <= 0) **return** 0;
15. **if** (memo[index][c] != -1) **return** memo[index][c];
16. **if** (w[index]>c) **return** helper(w, v, memo, index - 1, c);
17. **else**
18. {
19. **int** tmp1 = helper(w, v, memo, index - 1, c);//不放当前的物品
20. **int** tmp2 = helper(w, v, memo, index - 1, c - w[index]) + v[index];//放当前的物品
21. memo[index][c] = max(tmp1, tmp2);
22. **return** memo[index][c];
23. }
24. }
25. };
26. //自底而上动态规划
27. **class** Solution2
28. {
29. **public**:
30. **int** knapSack0\_1(vector<**int**> &w, vector<**int**> &v, **int** c)
31. {
32. **if** (w.empty() || v.empty() || c == 0) **return** 0;
33. **int** len = w.size();
34. vector<vector<**int**>> dp(len + 1, vector<**int**>(c + 1, 0));
35. **for** (**int** i = 1; i <= len; ++i)
36. {
37. **for** (**int** j = 1; j <= c; ++j)
38. {
39. **if** (w[i - 1] > j) dp[i][j] = dp[i - 1][j];
40. **else** dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i - 1]] + v[i - 1]);
41. }
43. }
44. **return** dp[len][c];
45. }
46. };
47. //优化为一维数组
48. **class** Solution3
49. {
50. **public**:
51. **int** knapSack0\_1(vector<**int**> &w, vector<**int**> &v, **int** c)
52. {
53. **if** (w.empty() || v.empty() || c == 0) **return** 0;
54. **int** len = w.size();
55. vector<**int**> dp(c + 1, 0);
56. **for** (**int** i = 0; i < len; ++i)
57. {
58. **for** (**int** j = c; j >= w[i]; --j)
59. dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
60. }
61. **return** dp[c];
62. }
63. };

完全背包问题

有n种物品和一个容量是c的背包，每种物品都有无限件可用。第i种物品的体积是wi，价值是vi。求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

1. //自顶向下的记忆法
2. **class** Solution1
3. {
4. **public**:
5. **int** ksComp(vector<**int**> &w, vector<**int**> &v, **int** c)
6. {
7. **if** (w.empty() || v.empty() || c == 0) **return** 0;
8. **int** len = w.size();
9. vector<vector<**int**>> memo(len, vector<**int**>(c + 1, -1));
10. **return** helper(w, v, memo, len - 1, c);
11. }
12. **int** helper(vector<**int**> &w, vector<**int**> &v, vector<vector<**int**>> &memo, **int** index, **int** c)
13. {
14. **if** (index < 0 || c == 0) **return** 0;
15. **if** (memo[index][c] != -1) **return** memo[index][c];
16. **if** (c < w[index]) **return** helper(w, v, memo, index - 1, c);
17. **else**
18. {
19. **int** res = 0;
20. **for** (**int** k = 0; k\*w[index] <= c; ++k)
21. {
22. **int** tmp = helper(w, v, memo, index - 1, c - w[index] \* k) + v[index] \* k;
23. res = max(res, tmp);
24. }
25. memo[index][c] = res;
26. **return** res;
27. }
28. }
29. };
30. //自下而上的动态规划
31. **class** Solution2
32. {
33. **public**:
34. **int** ksComp(vector<**int**> &w, vector<**int**> &v, **int** c)
35. {
36. **if** (w.empty() || v.empty() || c == 0) **return** 0;
37. **int** len = w.size();
38. vector<vector<**int**>> dp(len + 1, vector<**int**>(c + 1, 0));
39. **for** (**int** i = 1; i <= len; ++i)
40. {
41. **for** (**int** j = 1; j <= c; ++j)
42. {
43. **for** (**int** k = 0; k \* w[i - 1] <= j; ++k)
44. dp[i][j] = max(dp[i][j], dp[i - 1][j - k \* w[i - 1]] + k \* v[i - 1]);
45. }
46. }
47. **return** dp[len][c];
48. }
49. };
50. //优化为一维数组
51. **class** Solution3
52. {
53. **public**:
54. **int** ksComp(vector<**int**> &w, vector<**int**> &v, **int** c)
55. {
56. **if** (w.empty() || v.empty() || c == 0) **return** 0;
57. **int** len = w.size();
58. vector<**int**> dp(c + 1, 0);
59. **for** (**int** i = 0; i < len; ++i)
60. {
61. **for** (**int** j = c; j >= w[i]; --j)
62. {
63. **for** (**int** k = 0; k\*w[i] <= j; ++k)
64. dp[j] = max(dp[j], dp[j - k\*w[i]] + k\*v[i]);
65. }
66. }
67. **return** dp[c];
68. }
69. };

这里提供另一种更好的思路，再来看一下第一个状态转移函数：

F[i,c]=max{F[i−1,c−k\*wi]+k\*vi} ; (0≤k\*wi≤c)

仔细观察，在求解F[i,c]的过程中，用到了若干个F[i−1，0...c]。这个方程的核心在于：若要求解当前F[i,c]，我们遍历所有的第i−1层中需要的部分F[i−1,c−k\*wi]，找出在该子问题的基础上，我们k取何值时所得到的F[i,c]最大。

这个算法核心的缺陷就是：对于F[i][c]的求解，我们仅仅利用了第i−1层的数据F[i−1,c−k\*wi]，而忽略了本层的数据F[i][0...c−1]。在多层循环的过程中，可以保证前面的F[][]一定是正确的，不管它使用了几个物品。所以同样的，F[i][c]之前的子问题，包括F[i−1,c−k\*wi]和F[i][0...c−1]都是正确的结果，都要用到我们的状态转移函数中去。

如何利用本层的数据？F[i][0...c−1]与F[i][c]的共同点在于，它们都有可能包含有物品i，所以考虑将问题转化为“每次添加一个物品i的状态转移函数”。如果F[i][c]包含物品i，那么F[i][c−wi]中物品i的数量一定比F[i][c]少1。此时F[i,c]=F[i][c−wi]+vi。如果F[i][c]不包含物品i，那么F[i,c]=F[i−1,c]。当然，子问题F[i][c-wi]也可能包含有物品i，但是这个问题在前面的循环中已经被解决了，这里我们只需考虑在从F[i][c−wi]到F[i][c]的过程中，物品i是否被再一次添加。

新的状态转移函数：

F[i,c]=max{F[i−1,c],F[i][c−wi]+vi}

注意到该函数与“01背包问题”好像，但是需要注意：对于二维数组实现的01背包问题，第二层循环（遍历背包容量）可以正序，也可以逆序。 一维数组的01背包问题，第二层循环必须逆序。 对于完全背包问题，无论二维还是一维数组实现，都必须正序。具体留给大家思考。顺便写出一维情况下的状态转移函数：

F[c]=max{F[c],F[c−wi]+vi}

1. //一种更好的优化方法
2. **class** Solution4
3. {
4. **public**:
5. **int** ksComp(vector<**int**> &w, vector<**int**> &v, **int** c)
6. {
7. **if** (w.empty() || v.empty() || c == 0) **return** 0;
8. **int** len = w.size();
9. vector<**int**> dp(c + 1, 0);
10. **for** (**int** i = 0; i < len; ++i)
11. {
12. **for** (**int** j = w[i]; j <= c; ++j)
13. {
14. dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
15. }
16. }
17. **return** dp[c];
18. }
19. };

多重背包问题

有n种物品和一个容量是c的背包，每种物品都有无限件可用。第i种物品最多有ai件，体积是wi，价值是vi。求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

1. //自上而下的记忆法
2. **class** Solution1
3. {
4. **public**:
5. **int** ksMul(vector<**int**> &w, vector<**int**> &v, vector<**int**> &a, **int** c)
6. {
7. **if** (w.empty() || v.empty() || a.empty() || c == 0) **return** 0;
8. **int** len = w.size();
9. vector<vector<**int**>> memo(len + 1, vector<**int**>(c + 1, -1));
10. **return** helper(w, v, a, memo, len - 1, c);
11. }
12. **int** helper(vector<**int**> &w, vector<**int**> &v, vector<**int**> &a, vector<vector<**int**>> &memo, **int** index, **int** c)
13. {
14. **if** (index < 0 || c == 0) **return** 0;
15. **if** (memo[index][c] != -1) **return** memo[index][c];
16. **if** (w[index] > c) **return** helper(w, v, a, memo, index - 1, c);
17. **else**
18. {
19. **int** res = 0;
20. **for** (**int** j = 0; j <= a[index] && j\*w[index] <= c; ++j)
21. {
22. **int** tmp = helper(w, v, a, memo, index - 1, c - w[index] \* j) + j\*v[index];
23. res = max(res, tmp);
24. }
25. memo[index][c] = res;
26. **return** res;
27. }
28. }
29. };
30. //自下而上的动态规划
31. **class** Solution2
32. {
33. **public**:
34. **int** ksMul(vector<**int**> &w, vector<**int**> &v, vector<**int**> &a, **int** c)
35. {
36. **if** (w.empty() || v.empty() || a.empty() || c == 0) **return** 0;
37. **int** len = w.size();
38. vector<vector<**int**>> dp(len + 1, vector<**int**>(c + 1, 0));
39. **for** (**int** i = 1; i <= len; ++i)
40. {
41. **for** (**int** j = 1; j <= c; ++j)
42. {
43. **for** (**int** k = 0; k <= a[i - 1] && k\*w[i - 1] <= j; ++k)
44. {
45. dp[i][j] = max(dp[i][j], dp[i - 1][j - k\*w[i - 1]] + k\*v[i - 1]);
46. }
47. }
48. }
49. **return** dp[len][c];
50. }
51. };
52. //优化为一维数组
53. **class** Solution3
54. {
55. **public**:
56. **int** ksMul(vector<**int**> &w, vector<**int**> &v, vector<**int**> &a, **int** c)
57. {
58. **if** (w.empty() || v.empty() || a.empty() || c == 0) **return** 0;
59. **int** len = w.size();
60. vector<**int**> dp(c + 1, 0);
61. **for** (**int** i = 0; i < len; ++i)
62. {
63. **if** (w[i] \* a[i] >= c)//可以转为完全背包问题
64. {
65. **for** (**int** j = w[i]; j <= c; ++j)
66. dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
67. }
68. **else**//这里其实还可以转为01背包问题，继续优化
69. {
70. **for** (**int** j = c; j >= w[i]; --j)
71. {
72. **for** (**int** k = 0; k <= a[i] && k \* w[i] <= j; ++k)
73. {
74. dp[j] = max(dp[j], dp[j - w[i] \* k] + k \* v[i]);
75. }
76. }
77. }
78. }
79. **return** dp[c];
80. }
81. };

背包问题相关的讲解：

1. <https://www.cnblogs.com/mfrank/p/10849505.html>

这个讲解中，多重背包优化为一维数组时，代码存在问题，重点主要是看状态转移方程和递归方法

2. [[每日算法DP]背包九讲（其三）多重背包问题](https://quqi.gblhgk.com/s/516996/Iriy2USfH6L3Fclv)

1. 补充数组的插入排序和选择排序的简单实现
2. //插入排序
3. **void** insertSort(vector<**int**> &nums)
4. {
5. **int** len = nums.size();
6. **if** (len == 0 || len == 1) **return**;
7. **for** (**int** i = 1; i < len; ++i)
8. {
9. **if** (nums[i] < nums[i - 1])
10. {
11. **int** temp = nums[i];
12. **int** j = i - 1;
13. **while** (j >= 0 && nums[j]>temp)
14. {
15. nums[j + 1] = nums[j];
16. --j;
17. }
18. nums[j + 1] = temp;
19. }
20. }
21. }
23. //选择排序
24. //每次找最小的元素
25. **void** selectSort(vector<**int**> &nums)
26. {
27. **int** len = nums.size();
28. **for** (**int** i = 0; i < len - 1; ++i)
29. {
30. **int** j = i;
31. **for** (**int** k = i + 1; k < len; ++k)
32. {
33. **if** (nums[k] < nums[j]) j = k;
34. }
35. **if** (j != i) swap(nums[i], nums[j]);
36. }
37. }
38. 汉诺塔问题

对于传统的汉诺塔游戏我们做一个拓展，我们有从大到小放置的n个圆盘，开始时所有圆盘都放在左边的柱子上，按照汉诺塔游戏的要求我们要把所有的圆盘都移到右边的柱子上，请实现一个函数打印最优移动轨迹。

题目：给定一个int n，表示有n个圆盘。请返回一个string数组，其中的元素依次为每次移动的描述。描述格式为： move from [left/mid/right] to [left/mid/right]。

测试样例： 1

返回：move from left to right

1. **class** Hanoi {
2. **public**:
3. vector<string> getSolution(**int** n)
4. {
5. vector<string> res;
6. hanoi(n, "left", "mid", "right",res);
7. **return** res;
8. }
9. **void** hanoi(**int** n, string left, string mid, string right, vector<string> &res)
10. {
11. **if** (n == 0) **return**;
12. hanoi(n - 1, left, right, mid,res);
13. move(left, right, res);
14. hanoi(n - 1, mid, left, right, res);
15. }
16. **void** move(string from, string to, vector<string> &res)
17. {
18. string str = "move from " + from + " to " + to;
19. res.push\_back(str);
20. }
21. };

1. <https://blog.csdn.net/qq_41705423/article/details/82025409>

2. <https://blog.csdn.net/qq_33530753/article/details/80549842>

3. <https://www.zhihu.com/question/24385418>

**网络编程**

TCP协议

1. TCP提供可靠的数据传输，为实现这个功能，它提供了差错检测、重传、累计确认、定时器以及用于序号和确认号的首部字段。
2. TCP协议只在端系统中运行，提供的是全双工服务，TCP连接是点对点的即单个发送方与单个接受方之间的连接。
3. 最大报文段长度MSS是TCP报文段中应用层数据的长度，这个长度不包括首部信息。MSS的最大值取决于最大链路层帧长度MTU，一般以太网链路层帧为1500字节，去掉TCP/IP首部长度40字节（一般而言），所以MSS典型值为1460字节。
4. TCP报文段中的序号是对数据流中的每一个字节进行编号，确认号是主机期望收到的下一个字节的编号。TCP只确认该流中至第一个丢失字节为止的字节，所以TCP提供累积确认。
5. TCP处理失序报文段时一般有两个选择。（1）接收方立即丢弃失序报文段；（2）接收方保留失序的字节，并等待缺少的字节以填补该间隔。一般而言，第二种选择对网络带宽更为有效，是实践中采取的方法。
6. TCP定时器设置超时时间必须大于该连接的往返时间（RTT），大多数TCP的实现仅在某个时刻做一次样本RTT的测量，当获得一个新的样本RTT值时，更新RTT均值。一般会根据如下公式维护RTT均值。

Estimated=（1-α）\*Estimated+α\*SampleRTT（α一般取0.125）

TCP使用的是单一的重传定时器，发生如下三个事件时，TCP会启动定时器。（1）从上层应用程序中接收到数据，启动定时器；（2）发生超时，重启定时器；（3）接收到ACK时更新已确认的序号，如果还存在未确认的报文段，则重启定时器。

1. TCP的重传策略是回退N步和选择重传的结合，TCP的发送方仅需维持已发送过但未被确认的字节的最小序号，在这种意义下，TCP是GBN风格协议。另一方面，TCP接收方有选择的确认失序报文段，则TCP发送端最多重传一个报文段，若分组n丢失，则重传分组n，如果对报文段n+1的确认在报文段n超时之前到达，TCP不会重传报文段n。TCP跳过重传已被接收方选择确认过的报文段，这样类似于SR协议。所以说TCP的差错恢复机制是GBN协议与SR协议的混合体。
2. TCP发起连接经过三次握手的过程。第一步：客户端向服务器端发送SYN报文段，设置了客户端数据初始序号；第二步：服务器端收到SYN报文段，分配TCP缓存和变量（此操作容易受到SYN洪泛攻击），选择自己的初始序号，然后向客户端发送SYNACK报文段；第三步：客户端收到SYNACK报文段后，为该连接分配缓存和变量，同时向服务器发送确认报文段，此时连接已经建立，SYN位置应该0。

TCP断开连接经过四次挥手。客户端发出一个关闭连接命令，服务器向发送方回送一个确认报文段。然后服务器发送它自己的终止报文段，最后，该客户端对服务器的终止报文段进行确认。这之后客户端进入TIME\_WAIT状态，因为最后的ACK可能丢失，所以会等待一段时间，处理ACK丢失的情况，以重传ACK确认报文段。

1. SYN洪泛攻击，攻击者发送大量的TCP SYN报文段，而不完成第三次握手的步骤，服务器不断为这些半开连接分配资源，却永不会使用，导致服务器的连接资源被消耗殆尽。有一种有效的防御系统称为SYN cookie。当服务器收到一个SYN请求报文时，服务器不会分配资源，而是生成一个初始序列号，判断ACK返回值是否与该序列号匹配，如果匹配，则会生成一个全开连接。
2. TCP拥塞控制提出了三个问题。（1）TCP发送方如何限制它发送流量的速率。TCP发送方向网络发送的流量受到两个窗口的控制，接收窗口rwnd和拥塞窗口cwnd，它们限制了TCP向网络中发送流量的速率。（2）TCP如何感知它到目的地之间的路径存在拥塞。TCP发送端如果出现丢包事件则意味着拥塞，即出现超时或者收到来自接收方的3个冗余ACK。（3）当发送端感知到端到端的拥塞，采用何种算法改变发送速率。TCP拥有一套拥塞控制算法。其原则包括以下三个部分：出现丢包意味着拥塞，此时降低TCP发送方的速率；如果收到确认报文段，增加发送方的速率；通过带宽探测调节传输速率。

拥塞控制算法的内容包括三个部分，（1）慢启动，指数增长。（2）拥塞避免，线性增长

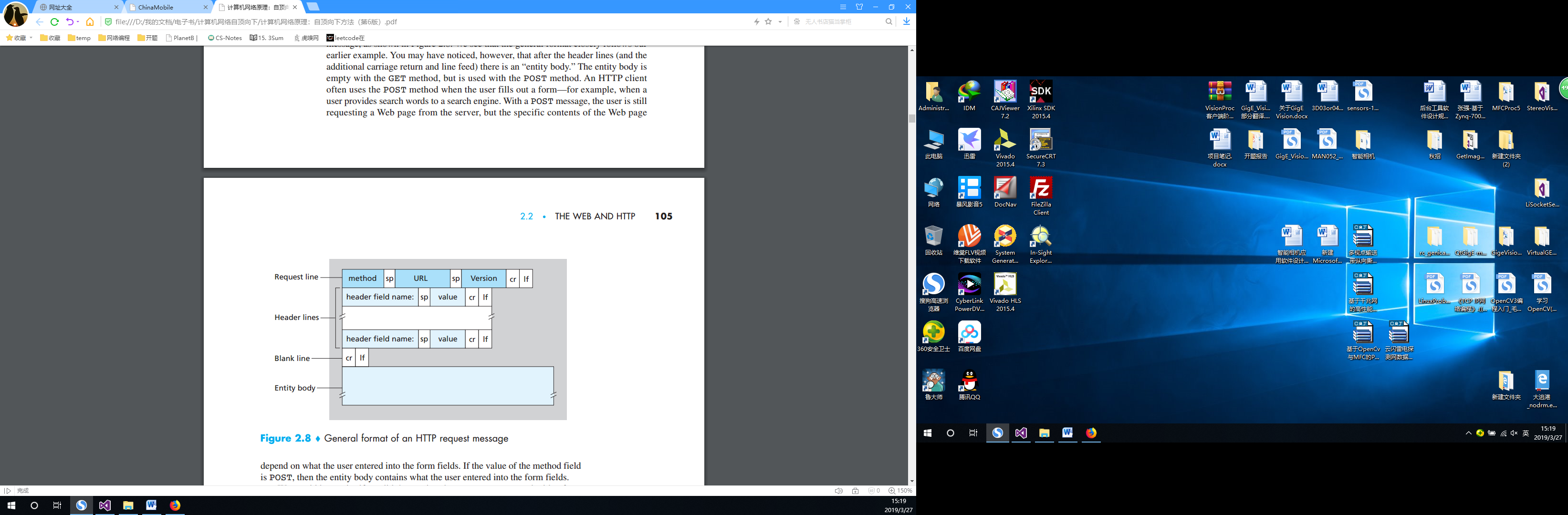
（3）快速恢复。收到ACK转为拥塞避免，超时转入慢启动。

1. 关于TCP粘包问题的讨论。

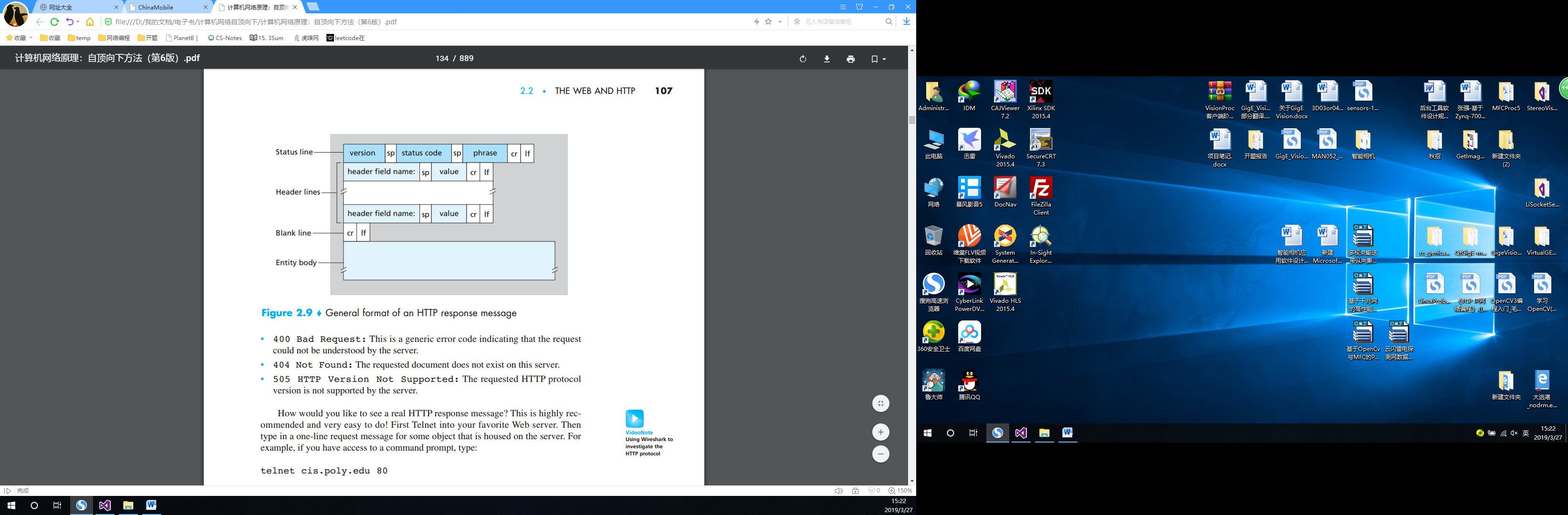
<https://www.zhihu.com/question/20210025/answer/358326299>

HTTP协议

1. Web服务器向客户发送被请求的文件，而不存储任何关于该客户的状态信息。服务器不会因为刚刚为该客户提供了该对象就不再做出反应，而是重新发送该对象。因为HTTP服务器并不保存关于客户的任何信息，所以HTTP是一个无状态协议。
2. HTTP对每个请求-响应对经过一个单独的TCP连接发送，称为非持续连接；对所有请求-响应对经相同的TCP连接发送称为持续连接。一般来说，对于持续连接，如果一条连接经过一定的时间间隔（一个可配置的超时间隔）仍未被使用，HTTP服务器就关闭该连接。
3. HTTP请求报文：



HTTP响应报文：



1. 因为HTTP服务器是无状态的，所以用户与服务器可以通过cookie进行交互，它允许站点对用户进行跟踪。cookie技术有四个组件：(1)在HTTP响应报文中包含一个cookie首部行；(2)HTTP请求报文中有一个cookie首部行；(3)在用户端系统中保留有一个cookie文件，并由用户的浏览器进行管理；(4)位于Web站点有一个后端数据库，记录了用户在该站点的活动。

select与epoll

1. select函数的调用过程

（1）设置文件描述符，将需要监视的文件描述符注册。（2）指定监视范围，linux中需要确定文件描述符的数量。（3）设置超时。（4）调用select函数，此时除了发生变化的文件描述符的对应位之外，剩下的所有位将初始化为0。（5）查找发生状态变化的文件描述符，并进行相应的处理。

1. 基于select的I/O复用速度慢的原因

基于select的I/O复用有两处存在效率低的情况。（1）调用select函数后需要针对所有文件描述符进行循环检查；（2）每次调用select函数时，都需要把监视对象信息传递给操作系统。相比于循环语句，每次传递监视对象的信息更加阻碍服务器的性能。

1. epoll的实现

epoll函数具有如下优点：无需编写以监视状态变化为目的针对所有文件描述符的循环语句；调用对应于select函数的epoll\_wait函数时无需每次传递监视对象信息。

epoll服务器端实现需要三个函数：epoll\_create，创建保存epoll文件描述符的空间；epoll\_ctl,向空间注册并注销文件描述符；epoll\_wait，与select函数类似，等待文件描述符发生变化。

1. 条件触发与边缘触发

条件触发方式中，只要输入缓冲中有数据就会一直通知该事件。边缘触发方式中，接收到数据时，仅会通知一次该事件。鉴于边缘触发的这种特性，需要将套接字更改为非阻塞特性，这样在接收到数据时，需要循环调用读取函数将输入缓冲中的所有数据读取出来，直到读取函数返回-1时且errno的值为EAGAIN，意味着读取了输入缓冲中的全部数据，此时跳出读取循环即可。

同步I/O与异步I/O的概念

* 1. 参考知乎中的回答，区别主要在于调用I/O函数的返回方式

<https://www.zhihu.com/question/19732473>

* 1. CPU与I/O操作之间的关系

<https://www.zhihu.com/question/27734728>

* 1. 关于linux中的零拷贝

<https://www.jianshu.com/p/e76e3580e356>

DMA的含义应该是从硬件到内存中的I/O，而内存间的操作还是需要CPU的参与。

**设计模式**

1. 单例模式

单例模式的要点有三个：单例类有且仅有一个实例；单例类必须自行创建自己的唯一实例；单例类必须给所有其他对象提供这一实例。

从具体实现角度来说，可分为以下三点：提供一个 private 构造函数（防止外部调用而构造类的实例）；提供一个该类的 static private 对象；提供一个 static public 函数，用于创建或获取其本身的静态私有对象（例如：GetInstance()）。

除此之外，还有一些关键点（需要多加注意，很容易忽视）：线程安全（双检锁 - DCL，即：double-checked locking）；资源释放。

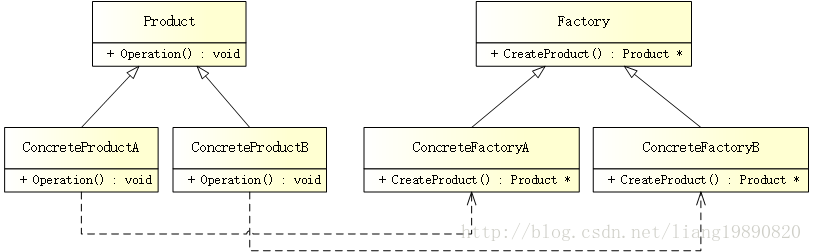
1. //单例模式
2. //局部静态变量方式,在单例对象消耗资源不大且不需要及时释放情况下比较常用。
3. //以下实现在C++ 11之前会存在线程安全的问题。
4. **class** Singleton1
5. {
6. **private**:
7. Singleton1() = **default**;
8. Singleton1(**const** Singleton1 &) = **delete**;
9. Singleton1 & operator=(**const** Singleton1 &) = **delete**;
10. **public**:
11. **static** Singleton1 & GetInstance()
12. {
13. **static** Singleton1 instance;
14. **return** instance;
15. }
16. **void** doSomething()
17. {
18. cout << "Do something" << endl;
19. }
20. };
22. //懒汉式，用于单例对象资源消耗较大或某种情况下需要及时释放的情况。
23. //优点：第一次调用才初始化，避免内存浪费。缺点：多线程下必须加锁才能保证单例，但//加锁会影响效率。
24. **class** Singleton2
25. {
26. **private**:
27. Singleton2() = **default**;
28. **static** Singleton2 \*m\_pSingleton2;
29. **static** mutex m\_mutex;
30. **public**:
31. **static** Singleton2 \* GetInstance();
32. **static** **void** DestoryInstance()//需要手动释放资源，有线程安全问题，最好 //使用智能指针。
33. {
34. **if** (m\_pSingleton2 != nullptr)
35. {
36. cout << "Here destroy the m\_pSingleton2..." << endl;
37. **delete** m\_pSingleton2;
38. m\_pSingleton2 = nullptr;
39. }
40. }
41. };
43. Singleton2 \*Singleton2::m\_pSingleton2 = nullptr;
44. mutex Singleton2::m\_mutex;
45. Singleton2 \* Singleton2::GetInstance()
46. {
47. **if** (m\_pSingleton2 == nullptr)
48. {
49. lock\_guard<mutex> lock(m\_mutex);//自解锁
50. **if** (m\_pSingleton2 == nullptr)
51. {
52. m\_pSingleton2 = **new** Singleton2();
53. }
54. }
55. **return** m\_pSingleton2;
56. }
58. //饿汉式。优点：没有加锁，执行效率会提高，缺点：类加载时就初始化，浪费内存。
59. **class** Singleton3
60. {
61. **private**:
62. Singleton3() = **default**;
63. **static** Singleton3 \*m\_pSingleton3;
64. **public**:
65. **static** Singleton3 \* GetInstance();
66. };
67. Singleton3 \*Singleton3::m\_pSingleton3 = **new** Singleton3();
68. Singleton3 \* Singleton3::GetInstance()
69. {
70. **return** m\_pSingleton3;
71. }

另外可参考：<https://blog.csdn.net/cjbct/article/details/79266057>

<https://bbs.csdn.net/topics/390648143>

尝试使用智能指针的实现：使用智能指针会要求用户也得使用智能指针，非必要不应该提出这种约束; 使用锁也有开销; 同时代码量也增多了，实现上我们希望越简单越好。还有更加严重的问题，在某些平台（与编译器和指令集架构有关），双检锁会失效，即CPU的乱序执行，可以考虑barrier命令。

1. **class** Singleton {
2. **public**:
3. **typedef** std::shared\_ptr<Singleton> Ptr;
4. ~Singleton()
5. {
6. std::cout << "destructor called!" << std::endl;
7. }
8. Singleton(Singleton&) = **delete**;
9. Singleton& operator=(**const** Singleton&) = **delete**;
10. **static** Ptr get\_instance()
11. {
12. // "double checked lock"
13. **if** (m\_instance\_ptr == nullptr)
14. {
15. std::lock\_guard<std::mutex> lk(m\_mutex);
16. **if** (m\_instance\_ptr == nullptr)
17. {
18. m\_instance\_ptr = std::shared\_ptr<Singleton>(**new** Singlet   on);
19. }
20. **return** m\_instance\_ptr;
21. }
22. }
23. **private**:
24. Singleton()
25. {
26. std::cout << "constructor called!" << std::endl;
27. }
28. **static** Ptr m\_instance\_ptr;
29. **static** std::mutex m\_mutex;
30. };
32. // initialization static variables out of class
33. Singleton::Ptr Singleton::m\_instance\_ptr = nullptr;
34. std::mutex Singleton::m\_mutex;
36. **int** main()
37. {
38. Singleton::Ptr instance = Singleton::get\_instance();
39. Singleton::Ptr instance2 = Singleton::get\_instance();
40. **return** 0;
41. }
42. 工厂方法模式



Factory（抽象工厂）：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。

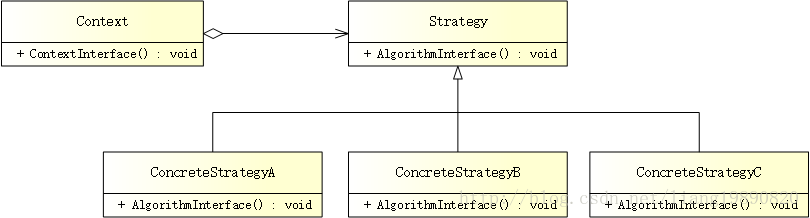
ConcreteFactory（具体工厂）：实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且被应用程序调用以创建产品对象。

Product（抽象产品）：所创建对象的基类，也就是具体产品的共同父类或共同拥有的接口。

ConcreteProduct（具体产品）：实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。

**工厂函数**：一个抽象接口类不能创建对象，而接口类的派生类可以创建实体，为了对接口类创建对象，通常调用一个特殊函数，这个函数用来构造真正被具现化的派生类的对象，这样的函数称为工厂函数。它们返回指针指向动态分配的对象，该对象支持接口类提供的接口，这样的函数往往在接口类中被声明为static。

1. //抽象接口。
2. **class** Car
3. {
4. **public**:
5. **virtual** string Name() = 0;
6. };
8. **class** Factory
9. {
10. **public**:
11. **virtual** Car \* CreateCar() = 0;
12. };
14. //汽车类的具体实现。
15. //奔驰汽车。
16. **class** BenzCar : **public** Car
17. {
18. **public**:
19. string Name()
20. {
21. **return** "Benz Car";
22. }
23. };
24. //宝马汽车。
25. **class** BmwCar : **public** Car
26. {
27. **public**:
28. string Name()
29. {
30. **return** "Bmw Car";
31. }
32. };
33. //奥迪汽车。
34. **class** AudiCar : **public** Car
35. {
36. **public**:
37. string Name()
38. {
39. **return** "Audi Car";
40. }
41. };
43. //工厂类的具体实现
44. //奔驰工厂。
45. **class** BenzFactory : **public** Factory
46. {
47. **public**:
48. Car \* CreateCar()
49. {
50. **return** **new** BenzCar();
51. }
52. };
53. //宝马工厂。
54. **class** BmwFactory : **public** Factory
55. {
56. **public**:
57. Car \* CreateCar()
58. {
59. **return** **new** BmwCar();
60. }
61. };
62. //奥迪工厂。
63. **class** AudiFactory : **public** Factory
64. {
65. **public**:
66. Car \* CreateCar()
67. {
68. **return** **new** AudiCar();
69. }
70. };
71. //生产奔驰。
72. Factory \*pFactory = **new** BenzFactory();
73. Car \*pCar = pFactory->CreateCar();
74. cout << "Benz Factory:" << pCar->Name() << endl;
75. **delete** pFactory;
76. **delete** pCar;
77. //生产宝马。
78. pFactory = **new** BmwFactory();
79. pCar = pFactory->CreateCar();
80. cout << "Bmw Factory:" << pCar->Name() << endl;
81. **delete** pFactory;
82. **delete** pCar;
84. pFactory = nullptr;
85. pCar = nullptr;
86. 策略模式



Context（环境角色）：持有一个对 Strategy 的引用，最终给客户端调用。

Strategy（抽象策略）：定义了一个公共接口，让不同的算法以不同的方式来实现。通过这个接口，Context 可以调用不同的算法。

ConcreteStrategy（具体策略）：实现 Strategy 定义的接口，提供具体算法的实现。

策略模式有以下优点：各自使用封装的算法，可以很容易地引入新的算法来满足相同的接口；由于实现的是同一个接口，所以策略之间可以自由切换；Strategy 使客户端能够选择所需的算法，而无需使用 switch/case 或 if/else 语句；算法的细节完全封装在 Strategy 类中，因此，可以在不影响 Context 类的情况下更改算法的实现。

1. //创建抽象策略，由IStrategy表示，它提供了一个travel()接口，用于提供出行方式。
2. **class** IStrategy
3. {
4. **public**:
5. **virtual** **void** travel() = 0;
6. };
8. //创建具体的策略供选择，比如骑自行车，开车，坐火车。
10. //骑自行车策略。
11. **class** BikeStrategy :**public** IStrategy
12. {
13. **public**:
14. **virtual** **void** travel() override
15. {
16. cout << "Travel by bike" << endl;
17. }
18. };
20. //开车的策略。
21. **class** CarStrategy :**public** IStrategy
22. {
23. **public**:
24. **virtual** **void** travel() override
25. {
26. cout << "Travel by car" << endl;
27. }
28. };
30. //坐火车的策略。
31. **class** TrainStrategy :**public** IStrategy
32. {
33. **public**:
34. **virtual** **void** travel() override
35. {
36. cout << "Travel by train" << endl;
37. }
38. };
40. //创建环境角色，环境角色对外提供了一个 Travel() 接口，最终由客户端调用。在内部，它//最终调用的是 IStrategy 的相应方法。
41. **class** Context
42. {
43. **public**:
44. Context(IStrategy \*strategy) :m\_pStrategy(strategy){}
45. **void** Travel()
46. {
47. m\_pStrategy->travel();
48. }
49. **private**:
50. IStrategy \*m\_pStrategy;
51. };
53. //创建客户端。
54. **int** main()
55. {
56. IStrategy \*bike = **new** BikeStrategy();
57. IStrategy \*car = **new** CarStrategy();
58. IStrategy \*train = **new** TrainStrategy();
60. //这里的意思是应该对客户隐藏具体的实现细节。
61. Context \*bikeContext = **new** Context(bike);
62. Context \*carContext = **new** Context(car);
63. Context \*trainContext = **new** Context(train);
65. bikeContext->Travel();
66. carContext->Travel();
67. trainContext->Travel();
69. **delete** bike,**delete** car,**delete** train;
70. **delete** bikeContext, **delete** carContext, **delete** trainContext;
71. }

有趣的题目

1. fork()产生的进程数量

<https://blog.csdn.net/hs794502825/article/details/10242091>

注意题目详解中下面给出的框图是指的在第一次fork()之后产生的进程执行的情况，仅针对那个位运算语句的。一个进程产生了4个新进程，之前fork（）产生的两个父子进程在执行该语句后即产生8个新进程。