**12-数据结构与算法**

**数据结构**

1. **二叉平衡树**

* 查找（Search）: 平均时间复杂度为 O(log n)，最坏情况下为 O(n)。
* 插入（Insert）: 平均时间复杂度为 O(log n)，最坏情况下为 O(n)。
* 删除（Delete）: 平均时间复杂度为 O(log n)，最坏情况下为 O(n)。

二叉搜索树的时间复杂度是稳定的，意味着在相同的输入规模下，操作的执行时间不会随着执行次数的增加而变化。

2. **AVL树**

高度计算log2(*n*)

2.1 **二叉平衡搜索树原理**

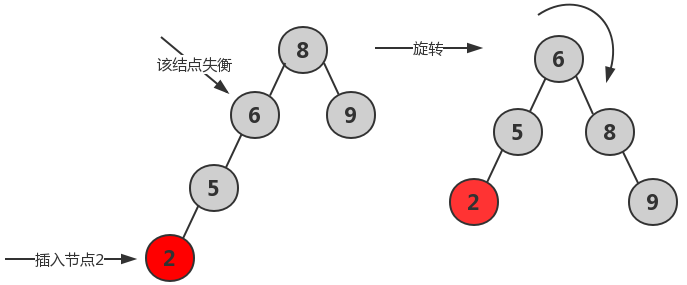
**平衡因子 : 树中某结点其左子树的高度和右子树的高度之差**

AVL树中的任意一个结点, 其平衡因子的**绝对值小于2**

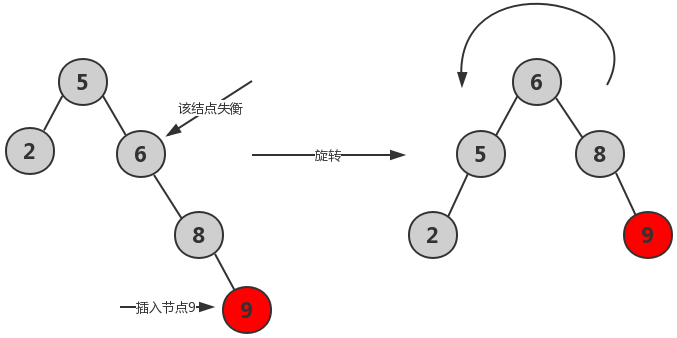
AVL树是一种特殊的二叉搜索树 (BST树), 相对于数据极端情况下, 二叉搜索树会退化成为单链表, AVL树定义了**旋转操作**, 在平衡因子大于等于2时, AVL树会旋转来调整树的结构, 来重新满足平衡因子小于2

2.2 **AVL树不平衡的情况**

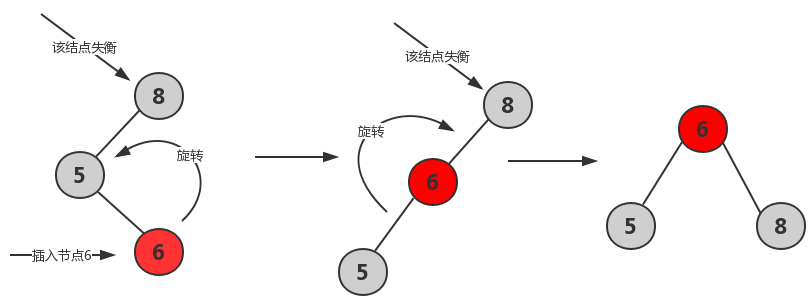
**2.1 左子树的左子树插入节点（左左）**



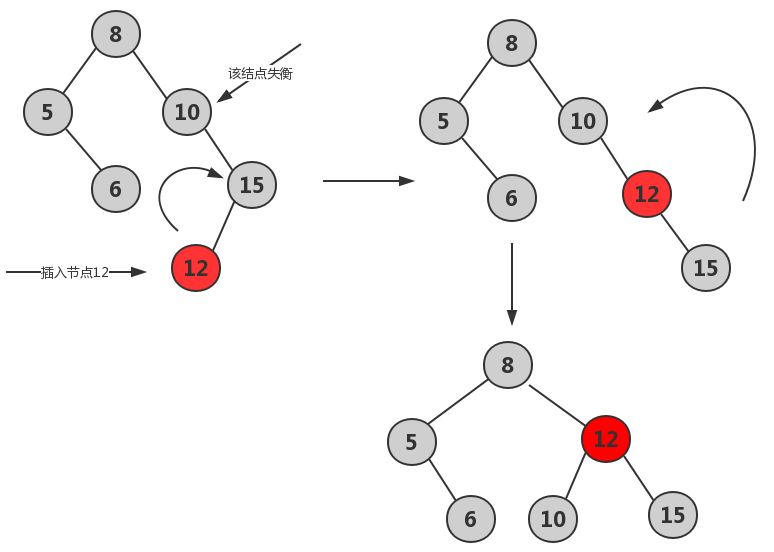
**2.2 右子树的右子树插入节点（右右）**



**2.3 左子树的右子树插入节点（左右）**



**2.4 右子树的左子树插入节点（右左）**



3. **红黑树**

**红黑树是一种含有红黑结点并能自平衡的二叉查找树。它必须满足下面性质：**

* 性质1：每个节点要么是黑色，要么是红色。
* 性质2：根节点是黑色。
* 性质3：每个叶子节点（NIL）是黑色。
* 性质4：每个红色结点的两个子结点一定都是黑色。
* 性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。

**插入详情（插入节点一定是红色）**

* 如果红黑树是空树，插入为根节点，设置为黑色
* 插入节点已经存在，跟新节点的值
* 插入的节点父节点是黑色，直接插入
* 插入节点的父节点是红色
* 叔叔节点存在并且为红色：将父亲节点和叔叔节点全部设置为黑色，将祖父节点设置为红色，
* 叔叔节点不存在或者为黑色：等于AVL树的四种情况，记得变色

**红黑树和平衡二叉树的不同之处**

平衡二叉树(AVL树)和红黑树都是一种自平衡的二叉树。AVL树是严格平衡的二叉树，要求每个节点的左右子树的高度差不超过1，而红黑树则要求宽松一些，不追求完全平衡，只要求任何一条路径的长度不超过其路径长度的2倍（黑高性质的限制），任何不平衡都会在三次旋转之内解决。正是由于这个差别，AVL树的查找效率会更高，但平衡调整的成本也更大。需要频繁查找时用AVL树更合适，在需要频繁插入和删除数时，选用红黑树更适合！

4. **B、B+树**

**B树是一种平衡多分树，我们常说的m阶的B树必须满足以下条件：**

* 每个节点最多只有m个子节点
* 每个非叶子节点（除了根）至少有m/2向上取整个子节点
* 如果根不是叶节点，则根至少有两个子节点
* 具有K个子节点的非叶节点至少包含K-1个键
* 叶子节点都出现在同一水平，没有任何信息
* 当节点个数等于m会引起分裂

**B+树**

高度计算 B+树的高度可以通过公式h=⌈log⁡m+1N⌉*h*=⌈log*m*+1*N*⌉来计算，其中m*m*是每个节点的子节点数（即分支因子），N*N*是数据项的数量。

* 每个元素不保存数据，只用来索引
* 叶子节点包含了全部关键字信息，以及指向含有这些关键字记录的指针
* 非叶子节点中的信息属于索引信息，仅包含索引部分的值

**为什么B+树比B树更适合数据库索引**

* B+树的磁盘读写代价更低：B+树内节点比B树小，可以降低IO的读写次数
* B+树查询效率更加稳定：所有关键字查询的路径长度相同
* B+树便于范围查询：b+树的叶子节点都是连接起来的，找到某一个范围起点即可执行范围查找，而数据库对于范围查找是非常频繁的

5. **哈希表**

**哈希表的基本思想**

建立记录关键码字与其存储位置的对应关系，或者说，由关键码的值决定数据的存储地址。这样，不经过比较，一次存取就能得到所查元素的查找方法

**哈希函数构造的方法**

* 直接定址法
* 除留余数法
* 乘余取整法
* 数字分析法
* 平方取中法
* 折叠法
* 随机数法

**哈希冲突处理的方法**

* 开放定址法（开地址法）
* 链地址法（拉链法）
* 再哈希法（双哈希函数法）
* 建立一个公共溢出区

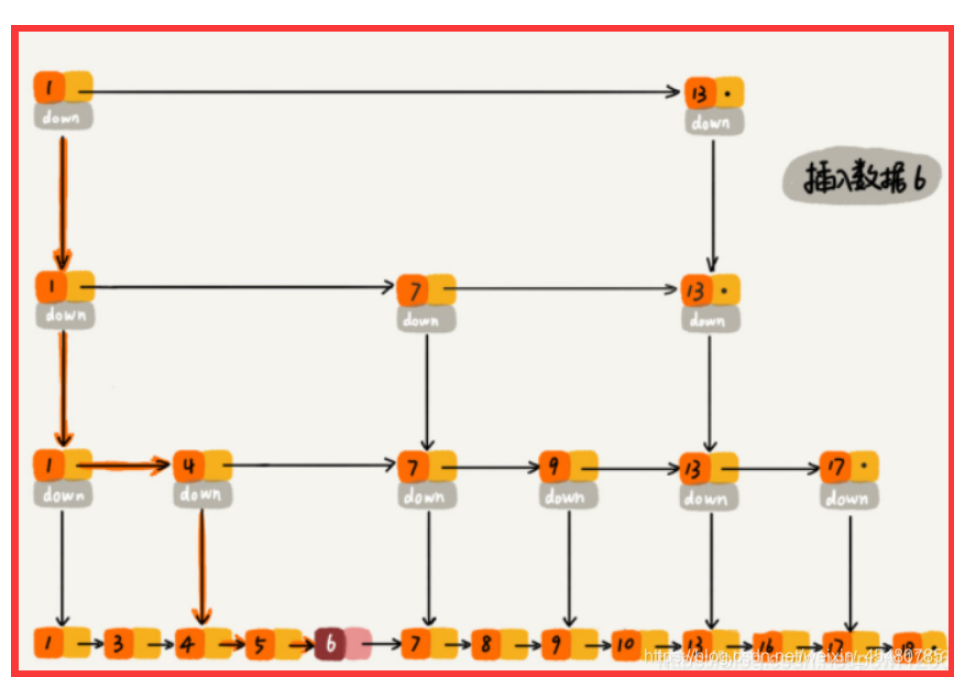
6. **跳跃表**

* 跳表，又叫做跳跃表、跳跃列表，在有序链表的基础上增加了“跳跃”的功能
* 跳表在原来的有序链表上加上了多级索引，通过索引来快速查找；可以支持快速的删除、插入和查找操作。
* 跳表实际上是一种增加了前向指针的链表，是一种随机化的数据结构
* Redis中 的 SortedSet、LevelDB 中的 MemTable 都用到了跳表
* 对比平衡树, 跳表的实现和维护会更加简单, 跳表的搜索、删除、添加的平均时间复杂度是 O(logn)

**跳表搜素**

1. 从顶层链表的首元素开始，从左往右搜索，直至找到一个大于或等于目标的元素，或者到达当前层链表的尾部
2. 如果该元素等于目标元素，则表明该元素已被找到
3. 如果该元素大于目标元素或已到达链表的尾部，则退回到当前层的前一个元素，然后转入下一层进行搜索

**跳表插入**



和跳表搜素一致

**跳表删除**

在跳表中删除某个结点时，如果这个结点在索引中也出现了，我们除了要删除原始链表中的结点，还要删除索引中的。因为单链表中的删除操作需要拿到删除结点的前驱结点，然后再通过指针操作完成删除。所以在查找要删除的结点的时候，一定要获取前驱结点（双向链表除外）。因此跳表的删除操作时间复杂度即为O(logn)。

**跳表索引动态更新**

当我们在跳表中插入数据的时候，我们通过选择同时将这个数据插入到部分索引层中，如何选择索引层，可以通过一个随机函数来决定这个节点插入到哪几级索引中，比如随机生成了k，那么就将这个索引加入到，第一级到第k级索引中。

**查询效率低的情况**

在理想情况下，查询效率接近于二分查找，时间复杂度为O(logN)。然而，如果skiplist的层数非常深，或者是在极端的情况下，如果skiplist的元素数量极少，或者是查询的key离散度很高，那么查询效率可能会下降到O(N)，此时就是查询效率的最低时刻。

以下是一个可能导致skiplist查询效率下降的情况：

1. 如果skiplist的元素分布非常不均匀，导致最上面的几层链表长度极长，那么查找某个特定元素可能需要遍历所有层。
2. 如果skiplist的层数非常深，比如有上百层，同样可能导致查询效率下降。
3. 如果skiplist中只存在几个元素，无法充分利用skiplist的优势，查询效率可能接近于链表的O(N)。

解决方法：

* 尽量保证数据的均匀分布，使得skiplist的层数和每层的元素数量都保持合理。
* 在插入数据时，可以使用一定策略动态调整skiplist的层数，以保持查询效率。
* 如果发现查询效率下降，可以考虑重建或者优化skiplist，比如进行数据迁移或者重新分配层数等。

7. **图**

图G由顶点集V和边集E组成，记为G=(V,E)，其中V(G)表示图G中顶点的有限非空集；E(G)表示图G中顶点之间的关系(边)的集合。

**图的基本概念**

* **有向图**
* **无向图**
* **简单图**

1）不存在重复边

2）不存在顶点到自身的边

3）有向图和无向图都是简单图

* **完全图**

无向图中任意两点之间都存在边，称为无向完全图。

有向图中任意两点之间都存在方向向反的两条弧，称为有向完全图。

* **子图**

若有两个图G=(V,E),G1=(V1,E2)，若V1是V的子集且E2是E的子集，称G1是G的子图。

* **连通、连通图、连通分量**

**连通**：在无向图中，两顶点有路径存在，就称为连通的。

**连通图**：若图中任意两顶点都连通，同此图为连通图。

**连通分量**：无向图中的极大连通子图称为连通分量。

**极大连通子图**：假设已经有了一个连通子图G，其顶点集为V，边集为E。如果E包含了在原图中和所有和V有关的边，那我们就认为它是极大连通子图。

* **强连通图、强连通分量**

**强连通图：**在有向图中，两顶点两个方向都有路径，两顶点称为强连通。

**强连通分量：**若任一顶点都是强连通的，称为强连通。有向图中极大强连通子图为有向图的强连通分量。

* **生成树和生成森林**

**生成树：**连通图的生成树是包含图中全部顶点的一个极小连通子图，若图中有n个顶点，则生成树有n-1条边。所以对于生成树而言，若砍去一条边，就会变成非连通图。

**生成森林**：在非连通图中，连通分量的生成树构成了非连通图的生成森林。

* **顶点的度、入度和出度**

对于无向图，顶点的边数为度，度数之和是顶点边数的两倍。

对于有向图，入度是以顶点为终点，出度相反。有向图的全部顶点入度之和等于出度之和且等于边数。顶点的度等于入度与出度之和。

* **边的权和网**

图中每条边上标有某种含义的数值，该数值称为该边的权值。这种图称为带权图，也称作网。

* **路径、路径长度和回路**

两顶点之间的路径指顶点之间经过的顶点序列，经过路径上边的数目称为路径长度。若有n个顶点，且边数大于n-1，此图一定有环。

* **简单路径、简单回路**

顶点不重复出现的路径称为简单路径。

除第一个顶点和最后一个顶点外，其余顶点不重复出现的回路称为简单回路。

* **距离**

若两顶点存在路径，其中最短路径长度为距离。

* **有向树**

有一个顶点的入度为0，其余顶点的入度均为1的有向图称作有向树。

**图的存储结构**

* **邻接矩阵**

图的邻接矩阵的表示方式需要两个数组来表示图的信息，一个一维数组表示每个数据元素的信息，一个二维数组（邻接矩阵）表示图中的边或者弧的信息。

**优点：** 直观、容易理解，可以很容易的判断出任意两个顶点是否有边，最大的优点就是很容易计算出各个顶点的度。

**缺点：** 但是对于稀疏矩阵，由于它边少，但是顶点多，这样就会造成空间的浪费。

* **邻接表**

邻接表是图的一种链式存储结构。主要是应对于邻接矩阵在顶点多边少的时候，浪费空间的问题。

**优点：** 对于，稀疏图，邻接表比邻接矩阵更节约空间。

**缺点：** 不容易判断两个顶点是有关系（边），顶点的出度容易，但是求入度需要遍历整个邻接表。

* **十字链表**

十字链表和邻接表一样，他会有两个结构来表示图：其中一个结构用于保存顶点信息，另外一个结构是用于保存每条边的信息。但是它只是为了有向图设计的，其中data主要是保存顶点的信息（如顶点的名称），firstin是保存第一个入度的边的信息，firstout保存第一个出度的边的信息。其中，表结点就是记录每条边的信息，其中tailvex是记录这条边弧头的顶点的在顶点表中的下标（不是箭头那个），headvex则是记录弧尾对应的那个顶点在顶点表中的下标（箭头的那个），hlink是指向具有下一个具有相同的headvex的表结点，tlink指向具有相同的tailvex的表结点，weight是表示边的权重（网图才需要使用）。

* **邻接多重表**

邻接多重表同样是对邻接表的一个改进得到来的结构，它同样需要一个头结点保存每个顶点的信息和一个表结点，保存每条边的信息，主要解决邻接表存储无向图时同一条边要存储两次的问题。其中mark为标志域，例如标志是否已经访问过，ivex和jvex代表边的两个顶点在顶点表中的下标，ilink指向下一个依附在顶点ivex的边，jlink指向下一个依附在顶点jvex的边，weight在网图的时候使用，代表该边的权重。

**图的应用**

* **图的遍历**

**（1）深度优先遍历（DFS）**

假设初始状态是图中所有顶点均未被访问，则从某个顶点v出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和v有路径相通的顶点都被访问到。若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

**（2）广度优先遍历 （BFS）**

从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

* **图的最短路径**

（1）深度或广度优先搜索算法（解决单源最短路径）

（2）弗洛伊德算法（解决多源最短路径）：时间复杂度O(n^3),空间复杂度O(n^2)

（3）迪杰斯特拉算法（解决单源最短路径）

* **判断图是否有环**

拓扑排序

**排序算法**

**1、冒泡排序**

|  |
| --- |
| C++ void bubble\_sort(vector<int>& array) {  int boder = array.size()-1;  int lastExchange = 0;  for(int i = 0; i<array.size()-1; i++)  {  bool is\_sorted = true;  for(int j = 0; j<boder; j++)  {  if(array[j] > array[j+1])  {  int tmp = array[j];  array[j] = array[j+1];  array[j+1] = tmp;  is\_sorted = false;  lastExchange = j;  }  }  boder = lastExchange;  if(is\_sorted)  {  break;  }  } } |

**2、鸡尾酒排序**

|  |
| --- |
| C++ void cocktail\_sort(vector<int>& array) {  int boder1 = array.size()-1;  int boder2 = 0;  int lastExchange1 = 0;  int lastExchange2 = 0;  for(int i = 0; i<array.size()-1; i++)  {  bool is\_sorted = true;  for(int j = 0; j<boder1; j++)  {  if(array[j] > array[j+1])  {  int tmp = array[j];  array[j] = array[j+1];  array[j+1] = tmp;  is\_sorted = false;  lastExchange1 = j;  }  }  boder1 = lastExchange1;  for(int k = array.size()-1; k>boder2; k--)  {  if(array[k-1] > array[k])  {  int tmp = array[k];  array[k] = array[k-1];  array[k-1] = tmp;  is\_sorted = false;  lastExchange2 = k;  }  }  boder2 = lastExchange2;  if(is\_sorted)  {  break;  }  } } |

**3、快速排序**

1. [通用排序需求](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E9%80%9A%E7%94%A8%E6%8E%92%E5%BA%8F%E9%9C%80%E6%B1%82&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：快速排序适用于各种类型的数据，无论是整数、浮点数还是字符串，都能高效地进行排序。
2. [数据库索引构建](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95%E6%9E%84%E5%BB%BA&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在数据库中，快速排序常用于构建索引，通过对数据库表中的数据进行排序，可以加速对数据的检索操作，提高数据库查询性能。
3. [文件系统排序](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F%E6%8E%92%E5%BA%8F&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在文件系统中，需要对文件或目录进行排序时，快速排序是一个快速且常用的算法，例如，按照文件名、文件大小或创建时间排序。
4. [编译器优化](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%BC%96%E8%AF%91%E5%99%A8%E4%BC%98%E5%8C%96&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在编译器优化过程中，快速排序用于对代码中的符号表、变量等信息进行排序，以加速这类排序操作。
5. [财务系统中的交易排序](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E8%B4%A2%E5%8A%A1%E7%B3%BB%E7%BB%9F%E4%B8%AD%E7%9A%84%E4%BA%A4%E6%98%93%E6%8E%92%E5%BA%8F&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在财务系统中，对交易记录进行排序是一项常见的任务，快速排序能够快速对交易记录按照日期、金额等进行排序。
6. [搜索引擎中的排名](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E%E4%B8%AD%E7%9A%84%E6%8E%92%E5%90%8D&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：搜索引擎需要对网页、文档等进行排序以提供相关性较高的搜索结果，快速排序在这种情境下的效率表现优异。
7. [大数据处理](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E5%A4%A7%E6%95%B0%E6%8D%AE%E5%A4%84%E7%90%86&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在大数据处理领域，快速排序因其平均时间复杂度为O(n log n)，在大数据场景下的性能优势非常明显。
8. [网络协议](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE&rsv_pq=a72c81190265c9f7&oq=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=d7b6mDOMBq1jgwCdzQx6cv2/YYb1OAyiLds3XKTe8l9Al7Rhj1X7E9ALbhs&tn=baidu&ie=utf-8)‌：在某些网络协议的实现中，快速排序可以应用于对数据包按照特定规则进行排序的场景。

|  |
| --- |
| C++  int doubleBorderPartition(vector<int>& array, int start, int end) {  int tmp = array[start];  int left = start+1;  int right = end;  while(left!=right)  {  while(left<right && array[left]<tmp)  {  left++;  }  while(left<right && array[right]>=tmp)  {  right--;  }  if(left<right)  {  int p = array[right];  array[right] = array[left];  array[left] = p;  }  }  array[start] = array[left];  array[left] = tmp;  return left; }  int singleBorderPartition(vector<int>& array, int start, int end) {  int tmp = array[start];  int mark = start;  for(int i = start+1; i<end; i++)  {  if(array[i] < tmp)  {  mark++;  int p = array[mark];  array[mark] = array[i];  array[i] = p;  }  }  array[start] = array[mark]  array[mark] = tmp;  return mark;   }  void quick\_sort(vector<int>& array, int start, int end) {  if(start >= end)  {  return;  }  int pivot = doubleBorderPartition(array,start,end);  quick\_sort(array,start,pivot-1);  quick\_sort(array,pivot+1,end); }  void quick\_sort(vector<int>& array, int start, int end) {  stack<map<string,int>> s;  map<string,int> tmp;  tmp["start"] = start;  tmp["end"] = end;  s.push(tmp);  while(!s.empty())  {  map<string,int> t = s.top();  s.pop();  if(tmp["start"]<tmp["end"])  {  int pivot = doubleBorderPartition(array,tmp["start"],tmp["end"]);  if(pivot-1 > tmp["start"])  {  map<string,int> t1;  t1["start"] = start;  t1["end"] = pivot-1;  s.push(t1);  }  if(pivot+1 < tmp["end"])  {  map<string,int> t2;  t2["start"] = pivot+1;  t2["end"] = end;  s.push(t2);  }  }  } } |

**4、堆排序**

* 大数据处理‌：在大数据时代，堆排序因其O(nlogn)的时间复杂度和只需维护部分数据的堆结构优势，特别适合处理无法一次性加载到内存中的大规模数据集。例如，在搜索引擎中，堆排序可以用于快速找到与查询最相关的K个文档；在推荐系统中，堆排序帮助筛选出用户最可能感兴趣的物品列表。
* 优先级队列实现‌：堆排序通过大顶堆或小顶堆的实现方式，适用于需要按照优先级进行排序的场景。在任务调度系统中，可以根据任务的优先级使用堆排序来安排任务的执行顺序；在操作系统中，堆排序也可以用于管理进程或线程的优先级。
* 图算法优化‌：在图的最短路径算法中，如[Dijkstra算法](https://www.baidu.com/s?sa=re_dqa_generate&wd=Dijkstra%E7%AE%97%E6%B3%95&rsv_pq=db270d710420b90f&oq=%E5%A0%86%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=706d53rzf9nzQKE618Pi6jnMfUikgIPLd8+Dz/eKo2zlu4M3QjQkj/G+qD8&tn=baidu&ie=utf-8)和[Prim算法](https://www.baidu.com/s?sa=re_dqa_generate&wd=Prim%E7%AE%97%E6%B3%95&rsv_pq=db270d710420b90f&oq=%E5%A0%86%E6%8E%92%E5%BA%8F%20%E5%BA%94%E7%94%A8%E5%9C%BA%E6%99%AF%E6%9C%89%E5%93%AA%E4%BA%9B&rsv_t=706d53rzf9nzQKE618Pi6jnMfUikgIPLd8+Dz/eKo2zlu4M3QjQkj/G+qD8&tn=baidu&ie=utf-8)，堆排序通过维护一个最小堆来存储待处理的节点及其当前最短距离，高效地找到下一个需要处理的节点，从而加速算法的执行。
* 实时网络流量监控与分析‌：堆排序适用于对实时数据进行排序和分析，帮助快速识别和处理大量数据中的关键信息。
* 实时排行榜系统‌：在需要实时更新排名的场景中，如体育比赛得分排名、在线投票排名等，堆排序可以有效地对数据进行排序，确保排名的实时性和准确性。

|  |
| --- |
| C++ //当新增元素的时候向上调整 void upAdjustHeap(vector<int>& array) {  int child = array.size()-1;  int parent = (child - 1)/2;  int tmp = array[child];  int c = 0;  while(parent > 0 && tmp > array[parent])  {  array[child] = array[parent];  child = parent;  parent = (child-1)/2;  c=1;  }  if(c)  {  array[parent] = tmp;  }  }  //当删除一个元素的时候，把最后一个元素与要删除的元素兑换，然后使用向下调整 void downAdjustHeap(vector<int>& array, int parent, int length) {  int child = parent\**2 + 1;*  *int tmp = array[parent];*  *while(child < length)*  *{*  *if(child+1 < length && array[child+1] > array[child])*  *{*  *child++;*  *}*  *if(tmp >= array[child])*  *{*  *break;*  *}*  *array[parent] = array[child];*  *parent = child;* \* child = parent\*2 + 1;  }  array[parent] = tmp; }  void heap\_sorted(vector<int>& array) {  for(int i = 0; i<(array.size()-2)/2; i++)  {  downAdjustHeap(array,i,array.size())  }  for(int i = 0; i<array.size()-1; i++)  {  int tmp = array[0];  array[0] = array[array.size()-i-1];  array[array.size()-i-1] = tmp;  downAdjustHeap(array,0,array.size()-i-1)  } } |

**5、选择排序**

|  |
| --- |
| C++ void select\_sorted(vector<int>& array) {  for(int i = 0; i<array.size()-2; i++)  {  int min = i;  for(j=i+1; j<array.size()-1; j++)  {  if(array[min] > array[j])  {  min = j;  }  }  int tmp = array[min];  array[min] = array[i];  array[i] = tmp;  } } |

**6、插入排序**

|  |
| --- |
| C++ void insert\_sorted(vector<int>& array) {  for(int i =1; i < array.size()-1; i++)  {  int tmp = array[i];  for(int j = i-1; j>=0; j--)  {  if(array[j] > array[i])  {  array[j+1] = array[j]; }  else  {  array[j+1] = tmp;  break;  }  }  } } |

**7、希尔排序**

|  |
| --- |
| C++ void insertE(vector<int>& array, int gap, int i) {  int tmp = array[i];  int k = 0;  for(k = i-gap; k>0&&array[k]>tmp; k-=gap)  {  array[k+gap] = array[k];  }  array[k+gap] = tmp; }  void shell\_sorted(vector<int>& array) {  int N = array.size();  for(int i=N/2; i>0; i=i/2)  {  for(int j = i; j<N; j++)  {  insertE(array,i,j)  }  } } |

**8、归并排序**

|  |
| --- |
| C++ void merge(vector<int>& array, int low, int mid, int hight) {  int i = low;  int j = mid+1;  int k = 0;  vector<int> tmp(hight-low+1);  while(i<=mid && j<=hight)  {  if(array[i]<=array[j])  {  tmp[k++] = array[i++];  }  else  {  tmp[k++] = array[j++];  }  }   while(i<=mid)  {  tmp[k++] = array[i++];  }  while(j<=hight)  {  tmp[k++] = array[j++];  }  for(int z = low; z<=hight; z++)  {  array[z] = tmp[z-low];  } } void merge\_sorted(vector<int>& array, int low, int hight) {  if(low>=hight) return;  int mid = (low + hight)/2;  merge\_sorted(array,low,mid);  merge\_sorted(array,mid+1,hight);  if(array[mid]<=array[mid+1]) return;  merge(array,low,mid,hight); } void merge\_sorted(vector<int>& array) {  int N = array.size();  for(int s = 1; s<N; s\*=2)  {  for(int i = 0; i+s<N; i+=2*s)*  *{*  *int low = i;*  *int mid = i+s-1;* \* int hight = i+2\*s-1;  if(hight>=N) hight = N-1;  merge(array,low,mid,hight);  }  } } |

**9、计数排序**

|  |
| --- |
| C++ void count\_sorted(vector<int>& array) {  int max = array[0];  int min = array[0];  for(int i=0; i<array.size(); i++)  {  if(array[i]>max)  {  max = array[i];  }  if(array[i]<min)  {  min = array[i];  }  }   vector<int> tmp(max-min+1,0);  vector<int> copy\_a(array);  for(int i=0; i<array.size(); i++)  {  tmp[array[i]-min]++;  }  for(int j = 1; j<tmp.size(); j++)  {  tmp[j]=tmp[j]+tmp[j-1];  }  for(int t = 0; t<array.size(); t++)  {  array[tmp[copy\_a[t]]--] = copy\_a[t];  } } |

**10、桶排序**

|  |
| --- |
| C++ void bucket\_sorted(vector<int>& array) {  vector<int> tmp[array.size()];  int max = array[0];  int min = array[0];  for(int i=0; i<array.size(); i++)  {  if(array[i]>max)  {  max = array[i];  }  if(array[i]<min)  {  min = array[i];  }  }  int size = (max-min)/array.size();  for(int i=0; i<array.size(); i++)  {  tmp[(array[i]-min)/size].push\_back(array[i]);  }  int k =0;  for(int i=0; i<array.size(); i++)  {  if(tmp[i].size())  {  sort(tmp[i].begin(),tmp[i].end());  for(int j=0;j<tmp[i].size();j++)  {  array[k++]=tmp[i][j];   }   }  } } |

**11、基数排序**

|  |
| --- |
| C++  void radixSort(vector<int>& array) {  int max = array[0];  int min = array[0];  for(int i = 0; i < array.size(); i++)  {  if(array[i]>max)  {  max = array[i];  }  if(array[i]<min)  {  min = array[i];  }  }  int n = 0;  while(max)  {  max = max / 10;  n++;  }  int extra = 0;  if(min<0)  {  extra = -min;  }  for(int j = 1; j <= n; j++)  {  vector<int> t(10,0);  vector<int> copy\_a(array);  int size = 1;  for(int i=1; i<j; i++)  {  size\*=10;  }  for(int k = 0; k < array.size(); k++)  {  t[(array[k]+min)/size%10]++;  }  for(int k = 1; k < 10; k++)  {  k[i]+=k[i-1];  }  for(int k = 0; k < array.size(); k++)  {  array[t[(copy\_a[k]+min)/size%10]--] = copy\_a[k];  }  } } |

**12、各类排序算法分析**



**其它算法**

**单例模式**

**饿汉模式**

饿汉方式表明只要提出需要，则立即用该类实例化对象，并且当前进程只能有一个该类型的对象。程序开始执行时系统就自动实例化了一个Singleton的data对象，因为Singleton类的构造函数被执行，说明构造了一个对象。

|  |
| --- |
| C++ #include <iostream> #include <unistd.h>   using namespace std;   class Singleton {  Singleton()//该类不能在外被构造  {  cout<<"程序运行时就打印该信息"<<endl;  }  ~Singleton(){}//该类不能在外被析构  Singleton(const Singleton& )=delete;//该类不能在外被拷贝  Singleton& operator=(const Singleton& )=delete;//该类不能在外被赋值    static Singleton data;//该进程下唯一Singleton对象，即单例 public:  static Singleton \*GetInstance()//外部只能通过调用该函数拿到data  {  return &data;  } };   Singleton Singleton::data;//data的初始化   int main() {  //以下两种实例化方式都无法实例化出对象  // Singleton s;  // Singleton s1(s);    cout<<Singleton::GetInstance()<<endl;  return 0; } |

**懒汉模式**

懒汉方式指的是当提出需求后，程序开始执行时系统不会立即实例化出对象，而是先实例化一个指针，当真正需要用到该对象时，系统才会主动的去申请一个对象，并让该指针指向这个对象。

|  |
| --- |
| C++ #include <iostream> #include <thread> #include <mutex> using namespace std; //如果用户使用懒汉式就需要加锁 //但是回收指针同样存在问题，存在多重释放或者不知道哪个指针释放的问题。 class SingleAuto { private:  SingleAuto() //无参构造  {   }  SingleAuto(const SingleAuto&) = delete; //禁止拷贝构造  SingleAuto& operator=(const SingleAuto&) = delete; //禁止拷贝复制 public:  ~SingleAuto()  {  cout << "Single auto delete success" << endl;  }  static std::shared\_ptr<SingleAuto> GetInst() {  if (single != nullptr)  {  return single; //已被初始化，直接返回  }  s\_mutex.lock();  if (single != nullptr)  {  s\_mutex.unlock();  return single;  }  single = std::shared\_ptr<SingleAuto>(new SingleAuto); //智能指针，自动回收内存  s\_mutex.unlock();  return single;  } private:  static std::shared\_ptr<SingleAuto> single;  static std::mutex s\_mutex; }; //饿汉式初始化 std::shared\_ptr<SingleAuto> SingleAuto::single = nullptr; std::mutex SingleAuto::s\_mutex;  void thread\_singleauto(int i) {   } void test\_singleauto() {  for (int i = 0; i < 3; i++) //3个子线程  {  std::thread tid(thread\_singleauto,i);  tid.join();  } } int main() {  test\_singleauto();  return 0; } |

**GO 语言版本**

|  |
| --- |
| Go var once sync.Once var instance interface{} func GetInstance() \*singleton {  once.Do(func() {  instance = &amp;amp;singleton{}  })  return instance } |

**生产者与消费者**

|  |
| --- |
| C++ **#include <iostream>** **#include <thread>** **#include <mutex>** **#include <condition\_variable>** **#include<queue>** using namespace std;  **const** **int** BUFFER\_SIZE = 15; *// 缓冲区大小* std::queue<**int**> buffer; *// 缓冲区* mutex mtx; std::condition\_variable cond; *// 条件变量*  **void** **producer\_do**() {  **int** count = 1;  **while** (1) {  std::unique\_lock<std::mutex> **locker**(mtx); *// 声明即加锁*  **while**(buffer.size() >= BUFFER\_SIZE){  cond.wait(locker); *//缓冲区已满，等待消费者线程取走数据*  }  buffer.push(count); *// 往缓冲区放入数据*  std::cout << "生产者线程 " << this\_thread::get\_id();   locker.unlock();  cond.notify\_one(); *// // 相当于V(full)*  *// std::this\_thread::sleep\_for(std::chrono::seconds(1));*  count++;  } }  **void** **consumer\_do**() {  **int** data = 0;  **while** (1) {  std::unique\_lock<std::mutex> **locker**(mtx); *// 声明即加锁*  **while**(buffer.empty())  cond.wait(locker); *// 等待缓冲区出现数据*  data = buffer.front();  buffer.pop(); *// 从缓冲区取走数据*  std::cout << "消费者线程 " << this\_thread::get\_id();   locker.unlock();  cond.notify\_one(); *// 相当于V(free)*  } }  **int** **main**() {  **const** **int** p\_num = 3;  **const** **int** c\_num = 2;  std::thread producers[p\_num];  std::thread consumers[c\_num];  **for**(**int** i = 0;i < p\_num;i ++){  producers[i] = std::thread(producer\_do);  }  **for**(**int** i = 0;i < c\_num;i ++){  consumers[i] = std::thread(consumer\_do);  }  **for**(**int** i = 0;i < p\_num;i ++){  producers[i].join();  }  **for**(**int** i = 0;i < p\_num;i ++){  consumers[i].join();  }  **return** 0; } |

|  |
| --- |
| C++ package main  import (  "fmt"  "sync"  "time" )  var counter int = 0 var mu sync.Mutex  type Consumer struct {  Buffer chan int }  func (c \*Consumer) Consume(wg \*sync.WaitGroup) {  defer wg.Done()  for {  select {  case n := <-c.Buffer:  if n == -1 {  fmt.Println("receive -1, stop consume")  return  } else {  fmt.Println("consume ", n)  }  default:  fmt.Println("still not receive channel message, sleep 1 second ... ")  time.Sleep(1 \* time.Second)  }  } }  type Producer struct {  Buffer chan int }  func (p \*Producer) Produce(wg \*sync.WaitGroup) {  defer wg.Done()  startTime := time.Now()   for i := 0; i < 10000; i++ {  // counter++ 是非原子性操作，  // 其可以拆解为 counter = counter + 1  // 多个协程同时修改counter变量时，可能读到相同的counter，做完累加后再写回。   // 比如 当前counter为1， 两个协程同时操作counter++后，预期counter应该为3，但实际可能发生如下情况  // 协程1读到 counter = 1，  // 协程2读到 counter = 1，  // 协程1累加后写回 counter = 2，  // 协程2累加后写回 counter = 2  // 最终counter = 2，而不是预期的3  // 为了解决这个问题，可以使用互斥锁来保护临界区  // 保护临界区  mu.Lock()  counter++  mu.Unlock()   p.Buffer <- i  }   endTime := time.Now()  fmt.Println("time cost: ", endTime.Sub(startTime)) }  func main() {  var wg sync.WaitGroup  fmt.Println("start")  ch := make(chan int, 10000000)  defer close(ch)   c := Consumer{Buffer: ch}  p := Producer{Buffer: ch}   // 启动协程1 开始生产  wg.Add(1)  go p.Produce(&wg)  // 启动协程2 开始生产  wg.Add(1)  go p.Produce(&wg)   wg.Add(1)  go c.Consume(&wg)   // 等待生产和消费完毕  time.Sleep(3 \* time.Second)  // 生产者生产完毕后，向通道发送-1，通知消费者停止消费  ch <- -1  wg.Wait()  fmt.Println("produce counter ", counter)  fmt.Println("end") } |

**多线程统计词频**

|  |
| --- |
| C++ #include <iostream> #include <fstream> #include <sstream> #include <map> #include <string> #include <thread> #include <mutex> #include <vector> #include <atomic> #include <filesystem>  namespace fs = std::filesystem;  std::mutex mtx; std::map<std::string, int> word\_freq; std::atomic<bool> done(false); std::vector<std::string> files;  void process\_file(const std::string& filename) {  std::ifstream file(filename);  std::string word;  while (file >> word) {  std::lock\_guard<std::mutex> lock(mtx);  word\_freq[word]++;  }  file.close();   // 删除文件  fs::remove(filename); }  void user\_input\_thread() {  std::string word;  while (!done) {  std::cout << "Enter a word to check its frequency (or 'exit' to quit): ";  std::cin >> word;  if (word == "exit") {  done = true;  break;  }   std::lock\_guard<std::mutex> lock(mtx);  auto it = word\_freq.find(word);  if (it != word\_freq.end()) {   } else {   }  } }  int main() {  // 填充文件列表  files = {"file1.txt", "file2.txt", "file3.txt"};   // 创建线程来处理文件  std::vector<std::thread> threads;  for (const auto& file : files) {  threads.emplace\_back(process\_file, file);  }   // 创建用户输入线程  std::thread input\_thread(user\_input\_thread);   // 等待所有文件处理线程完成  for (auto& thread : threads) {  thread.join();  }   // 通知用户输入线程结束  done = true;  input\_thread.join();   return 0; } |

|  |
| --- |
| Go package main  import (  "bufio"  "fmt"  "os"  "strings"  "sync" )  func main() {  files := []string{"file1.txt", "file2.txt", "file3.txt"}  var wg sync.WaitGroup  var mutex sync.Mutex  results := &sync.Map{}   // 读取和处理每个文件  for \_, file := range files {  wg.Add(1)  go func(file string) {  defer wg.Done()  countWordsInFile(file, results)  // 删除文件（在实际应用中，你可能希望先确保所有操作都完成了再删除文件）  mutex.Lock()  defer mutex.Unlock()  os.Remove(file)  }(file)  }   // 等待所有goroutine完成  wg.Wait()   // 用户输入查询词频  fmt.Println("输入单词查询词频，输入'exit'退出程序：")  scanner := bufio.NewScanner(os.Stdin)  for scanner.Scan() {  word := scanner.Text()  if word == "exit" {  break  }  // 查询并输出词频  if count, ok := results.Load(word); ok {  fmt.Printf("单词 '%s' 的词频是：%d\n", word, count)  } else {  fmt.Println("单词未找到。")  }  }   if err := scanner.Err(); err != nil {  fmt.Println("读取输入错误:", err)  } }  func countWordsInFile(filename string, results \*sync.Map) {  file, err := os.Open(filename)  if err != nil {  fmt.Println("打开文件错误:", err)  return  }  defer file.Close()   scanner := bufio.NewScanner(file)  for scanner.Scan() {  words := strings.Fields(scanner.Text())  for \_, word := range words {  results.Store(word, results.LoadOrStore(word, 0).(int)+1)  }  }   if err := scanner.Err(); err != nil {  fmt.Println("读取文件错误:", err)  } } |

**交替打印**

|  |
| --- |
| C++ #include <iostream> #include <thread> #include <mutex> #include <condition\_variable>  std::mutex mtx; std::condition\_variable cv; bool printEven = true; int count = 0; const int maxCount = 10;  void printEvenNumbers() {  while (true) {  std::unique\_lock<std::mutex> lock(mtx);  cv.wait(lock, [] { return printEven; });  if (count >= maxCount) {  lock.unlock();  cv.notify\_one();  break;  }  std::cout << count << " ";  count++;  printEven = false;  lock.unlock();  cv.notify\_one();  } }  void printOddNumbers() {  while (true) {  std::unique\_lock<std::mutex> lock(mtx);  cv.wait(lock, [] { return !printEven; });  if (count >= maxCount) {  lock.unlock();  cv.notify\_one();  break;  }  std::cout << count << " ";  count++;  printEven = true;  lock.unlock();  cv.notify\_one();  } }  int main() {  std::thread t1(printEvenNumbers);  std::thread t2(printOddNumbers);   t1.join();  t2.join();   return 0; } |

|  |
| --- |
| Go package main  import (  "fmt"  "sync" )  var (  mutex sync.Mutex  condVar sync.Cond  turn int // 用于指示哪个goroutine应该打印  count int // 用于跟踪打印的数字  maxCount int = 10 // 打印的最大次数 )  func printEven(wg \*sync.WaitGroup) {  defer wg.Done()  for {  mutex.Lock()  for turn != 0 { // 偶数goroutine等待  condVar.Wait()  }  if count >= maxCount {  mutex.Unlock()  break  }  fmt.Print(count, " ")  count++  turn = 1 // 切换到奇数goroutine  condVar.Broadcast()  mutex.Unlock()  } }  func printOdd(wg \*sync.WaitGroup) {  defer wg.Done()  for {  mutex.Lock()  for turn != 1 { // 奇数goroutine等待  condVar.Wait()  }  if count >= maxCount {  mutex.Unlock()  break  }  fmt.Print(count, " ")  count++  turn = 0 // 切换到偶数goroutine  condVar.Broadcast()  mutex.Unlock()  } }  func main() {  var wg sync.WaitGroup  condVar = sync.NewCond(&mutex)   wg.Add(2)  go printEven(&wg)  go printOdd(&wg)   // 开始时让偶数goroutine先打印  mutex.Lock()  turn = 0  condVar.Broadcast()  mutex.Unlock()   wg.Wait() } |

**十个子进程并发执行，父进程等待所有子进程结束状态之后，在返回结果**

|  |
| --- |
| C++ #include <iostream> #include <unistd.h> #include <sys/wait.h>  int main() {  int num\_children = 10;  pid\_t pid;   // 创建子进程  for (int i = 0; i < num\_children; ++i) {  pid = fork();   if (pid == -1) {  // fork失败   return 1;  } else if (pid == 0) {  // 子进程   // 模拟子进程工作  sleep(1);   exit(0); // 子进程退出  }  // 父进程继续循环创建下一个子进程  }   // 父进程等待所有子进程结束  int status;  while (num\_children > 0) {  pid = wait(&status);  if (pid > 0) {   --num\_children;  }  }   std::cout << "All child processes have exited. Parent process is exiting." << std::endl;  return 0; } |

|  |
| --- |
| Go package main  import (  "fmt"  "os/exec"  "sync" )  func main() {  var wg sync.WaitGroup  numChildren := 10   for i := 0; i < numChildren; i++ {  wg.Add(1)  go func(i int) {  defer wg.Done()  cmd := exec.Command("echo", fmt.Sprintf("Child process %d is running.", i))  if err := cmd.Run(); err != nil {  fmt.Printf("Error running child process %d: %v\n", i, err)  } else {  fmt.Printf("Child process %d has exited.\n", i)  }  }(i)  }   wg.Wait()  fmt.Println("All child processes have exited. Parent process is exiting.") } |

**IP 转化为32位整数**

|  |
| --- |
| C++ #include <iostream> #include <sstream> #include <vector>  uint32\_t ipToInt(const std::string& ip) {  std::stringstream ss(ip);  uint32\_t result = 0;  uint32\_t shift = 24; // 因为是32位整数，所以开始转换时左移24位  uint32\_t part;   while (ss >> part) {  result |= part << shift;  shift -= 8;  if (ss.peek() == '.') {  ss.ignore();  }  }   return result; }  int main() {  std::string ip = "192.168.1.1";  uint32\_t ipInt = ipToInt(ip);    return 0; } |

**哲学家进餐问题**

|  |
| --- |
| C++ #include <iostream> #include <thread> #include <mutex> #include <condition\_variable>  class DiningPhilosophers { private:  std::mutex forks; // 五把叉子  std::condition\_variable cv; // 条件变量，用于唤醒等待的哲学家  public:  DiningPhilosophers() {}   void dine(int philosopher) {  int leftFork = philosopher;  int rightFork = (philosopher + 1) % 5;   // 思考  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));   // 获取左边的叉子  std::unique\_lock<std::mutex> leftLock(forks[leftFork]);  // 获取右边的叉子  std::unique\_lock<std::mutex> rightLock(forks[rightFork]);   // 进餐   std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));   // 放下叉子并唤醒其他等待的哲学家  leftLock.unlock();  rightLock.unlock();  cv.notify\_all();  } };  int main() {  DiningPhilosophers dp;  std::thread philosophers;   // 创建五位哲学家的线程  for (int i = 0; i < 5; ++i) {  philosophers[i] = std::thread(&DiningPhilosophers::dine, &dp, i);  }   // 等待所有哲学家完成进餐  for (auto& t : philosophers) {  t.join();  }   return 0; } |

**给一段英文，统计出现次数最多的单词**

在C++中，你可以使用以下步骤来统计一段英文文本中出现次数最多的单词：

1. 预处理文本：转换为小写，去除标点符号。
2. 分词：使用空格将文本分割成单词。
3. 统计单词频率：使用std::map或std::unordered\_map来记录每个单词的出现次数。
4. 找出高频单词：遍历map，找出出现次数最多的单词。

|  |
| --- |
| C++ #include <iostream> #include <sstream> #include <unordered\_map> #include <algorithm> #include <vector>  std::string preprocess(const std::string& text) {  std::string result;  for (char ch : text) {  if (isalpha(ch)) {  result += tolower(ch);  } else if (ch == ' ') {  result += ch;  }  }  return result; }  std::vector<std::string> split(const std::string& text) {  std::vector<std::string> words;  std::stringstream ss(text);  std::string word;  while (ss >> word) {  words.push\_back(word);  }  return words; }  int main() {  std::string text = "Hello, world! The world is not enough. Hello, C++!";  text = preprocess(text);  auto words = split(text);   std::unordered\_map<std::string, int> frequency;  for (const auto& word : words) {  ++frequency[word];  }   int maxFrequency = 0;  std::vector<std::string> mostFrequentWords;  for (const auto& pair : frequency) {  if (pair.second > maxFrequency) {  maxFrequency = pair.second;  mostFrequentWords = {pair.first};  } else if (pair.second == maxFrequency) {  mostFrequentWords.push\_back(pair.first);  }  }   std::cout << "Most frequent words:" << std::endl;  for (const auto& word : mostFrequentWords) {   }   return 0; } |

**写一个死锁**

|  |
| --- |
| C++ #include <iostream> #include <thread> #include <mutex>  std::mutex mutex1; std::mutex mutex2;  void threadFunction1() {  std::lock\_guard<std::mutex> lock1(mutex1);  std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  std::lock\_guard<std::mutex> lock2(mutex2);  std::cout << "Thread 1 is running" << std::endl; }  void threadFunction2() {  std::lock\_guard<std::mutex> lock1(mutex2);  std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  std::lock\_guard<std::mutex> lock2(mutex1);  std::cout << "Thread 2 is running" << std::endl; }  int main() {  std::thread t1(threadFunction1);  std::thread t2(threadFunction2);   t1.join();  t2.join();   return 0; } |

**链表实现快排**

|  |
| --- |
| C++ #include <iostream>  struct ListNode {  int val;  ListNode \*next;  ListNode(int x) : val(x), next(nullptr) {} };  ListNode\* getTail(ListNode\* cur) {  while (cur != nullptr && cur->next != nullptr) {  cur = cur->next;  }  return cur; }  ListNode\* partition(ListNode\* head, ListNode\* end, ListNode==&zwnj;\*\* newHead, ListNode\*\*&zwnj;== newEnd) {  ListNode \*pivot = end;  ListNode \*prev = nullptr;  ListNode \*cur = head;  ListNode \*tail = pivot;    while (cur != pivot) {  if (cur->val < pivot->val) {  if (\*newHead == nullptr) \*newHead = cur;  prev = cur;  cur = cur->next;  } else {  if (prev) {  prev->next = cur->next;  }  ListNode\* tmp = cur->next;  cur->next = nullptr;  tail->next = cur;  tail = cur;  cur = tmp;  }  }    if (\*newHead == nullptr) \*newHead = pivot;    \*newEnd = tail;  return pivot; }  ListNode\* quickSortRecur(ListNode\* head, ListNode\* end) {  if (!head || head == end) {  return head;  }    ListNode \*newHead = nullptr, \*newEnd = nullptr;  ListNode \*pivot = partition(head, end, &newHead, &newEnd);    if (newHead != pivot) {  ListNode\* temp = newHead;  while (temp->next != pivot) {  temp = temp->next;  }  temp->next = nullptr;    newHead = quickSortRecur(newHead, temp);    temp = getTail(newHead);  temp->next = pivot;  }    pivot->next = quickSortRecur(pivot->next, newEnd);    return newHead; }  ListNode\* quickSort(ListNode\* head) {  return quickSortRecur(head, getTail(head)); }  void printList(ListNode\* node) {  while (node != nullptr) {  std::cout << node->val << " ";  node = node->next;  }  std::cout << std::endl; }  int main() {  ListNode\* head = new ListNode(4);  head->next = new ListNode(2);  head->next->next = new ListNode(1);  head->next->next->next = new ListNode(3);   std::cout << "Original list: ";  printList(head);   ListNode\* sortedHead = quickSort(head);   std::cout << "Sorted list: ";  printList(sortedHead);   return 0; } |