编 程 学 习 笔 记

数据结构与算法

Data Stucture and Algorithm

作 者 姓 名： 颜佳

目 录

[1 五大算法 1](#_Toc76210168)

[1.1 分治 1](#_Toc76210169)

[1.1.1 问题特点 1](#_Toc76210170)

[1.1.2 解题套路 1](#_Toc76210171)

[1.1.3 适用问题 1](#_Toc76210172)

[1.1.4 注意事项 1](#_Toc76210173)

[1.2 贪心算法 2](#_Toc76210174)

[1.2.1 问题特点 2](#_Toc76210175)

[1.2.2 解题套路 2](#_Toc76210176)

[1.2.3 适用问题 2](#_Toc76210177)

[1.2.4 注意事项 2](#_Toc76210178)

[1.3 动态规划 2](#_Toc76210179)

[1.3.1 问题特点 2](#_Toc76210180)

[1.3.2 解题套路 2](#_Toc76210181)

[1.3.3 适用问题 3](#_Toc76210182)

[1.3.4 注意事项 6](#_Toc76210183)

[1.4 回溯法 7](#_Toc76210184)

[1.4.1 问题特点 7](#_Toc76210185)

[1.4.2 解题套路 7](#_Toc76210186)

[1.4.3 适用问题 8](#_Toc76210187)

[1.4.4 注意事项 8](#_Toc76210188)

[1.5 分支限界法 11](#_Toc76210189)

[1.5.1 问题特点 11](#_Toc76210190)

[1.5.2 适用问题 11](#_Toc76210191)

[1.5.3 解题套路 11](#_Toc76210192)

[1.5.4 注意事项 11](#_Toc76210193)

[2 数据结构 11](#_Toc76210194)

[2.1 数组与链表 11](#_Toc76210195)

[2.1.1 左右指针 12](#_Toc76210196)

[2.1.2 快慢指针——成环问题 12](#_Toc76210197)

[2.1.3 滑动窗口——子串问题 13](#_Toc76210198)

[2.2 二分法 14](#_Toc76210199)

[2.2.1 解题套路 14](#_Toc76210200)

[2.2.2 注意事项 15](#_Toc76210201)

[2.3 字符串 15](#_Toc76210202)

[2.3.1 字符串函数 15](#_Toc76210203)

[2.3.2 模式匹配 18](#_Toc76210204)

[2.4 树 18](#_Toc76210205)

[2.4.1 深度优先搜索 18](#_Toc76210206)

[2.4.2 广度优先搜索 18](#_Toc76210207)

[2.5 深度优先搜索和回溯法的区别 20](#_Toc76210208)

[2.5.1 深度优先搜索 20](#_Toc76210209)

[2.5.2 回溯法 20](#_Toc76210210)

[2.6 图 21](#_Toc76210211)

[2.7 前缀和和查分数组 21](#_Toc76210212)

[3 UTHASH的使用 21](#_Toc76210213)

[3.1 3个数据结构 21](#_Toc76210214)

[3.1.1 UT\_hash\_handle 21](#_Toc76210215)

[3.1.2 UT\_hash\_table 21](#_Toc76210216)

[3.1.3 UT\_hash\_bucket 21](#_Toc76210217)

[3.2 自定义结构体 22](#_Toc76210218)

[3.3 7个函数 22](#_Toc76210219)

[3.3.1 以int类型为key 22](#_Toc76210220)

[3.3.2 以char[]类型为key 24](#_Toc76210221)

[3.3.3 以char \*类型为key 26](#_Toc76210222)

[3.3.4 以地址void\*类型为key 28](#_Toc76210223)

[3.3.5 以任意类型为key 30](#_Toc76210224)

[4 技巧 33](#_Toc76210225)

[4.1 m\*n的二维数组的下标i和j与展开后的一维数组的下标k的关系： 33](#_Toc76210226)

[4.1.1 i = k / n; 33](#_Toc76210227)

[4.1.2 j = k %n; 33](#_Toc76210228)

[4.2 求余数的加法与乘法分配律 33](#_Toc76210229)

[4.3 添加哑结点 33](#_Toc76210230)

[4.4 判断一个数是不是2的次幂 33](#_Toc76210231)

[5 精选题目： 33](#_Toc76210232)

[5.1 378. 有序矩阵中第 K 小的元素 33](#_Toc76210233)

[5.2 1712. 将数组分成三个子数组的方案数 33](#_Toc76210234)

[5.3 395. 至少有 K 个重复字符的最长子串 33](#_Toc76210235)

[6 Linux内核无锁缓冲队列kfifo原理 34](#_Toc76210236)

[6.1 kfifo概述 34](#_Toc76210237)

[6.2 kfifo数据结构定义 34](#_Toc76210238)

[6.2.1 kfifo结构体定义 34](#_Toc76210239)

[6.2.2 kfifo内存分配和初始化——kfifo\_alloc() 34](#_Toc76210240)

[6.2.3 kfifo入队操作——\_\_kfifo\_put() 36](#_Toc76210241)

[6.2.4 kfifo出队操作——\_\_kfifo\_get() 38](#_Toc76210242)

[6.3 kfifo无锁并发与内存屏障 41](#_Toc76210243)

[6.3.1 无锁并发 41](#_Toc76210244)

[6.3.2 内存屏障 41](#_Toc76210245)

[6.4 Linux内核实现的kfifo的特点： 42](#_Toc76210246)

[6.4.1 使用内存屏障 Memory Barrier 42](#_Toc76210247)

[6.4.2 初始化缓冲区空间时要保证缓冲区的大小为2的幂次方 42](#_Toc76210248)

[6.4.3 使用无符号整数保存in和out（输入输出的指针），并且在放入取出数据的时候不做模运算，让其自然溢出。 42](#_Toc76210249)

[6.5 kfifo结构的优点 43](#_Toc76210250)

[6.6 编程技巧总结 43](#_Toc76210251)

[6.6.1 判断一个整数n是不是2的幂次方 43](#_Toc76210252)

[6.6.2 将一个整数n向上圆整为2的幂次方 43](#_Toc76210253)

[6.6.3 对2的幂次方取余可以转化为与运算 44](#_Toc76210254)

[6.6.4 循环队列 44](#_Toc76210255)

[6.6.5 无符号整数溢出 45](#_Toc76210256)

# 五大算法

## 分治

待解决的复杂问题能够简化为几个若干个小规模相同的问题，且各子问题间独立存在，即子问题之间不包含公共的子问题，子问题的解可以合并为该问题的解。

### 问题特点

1. 子问题之间不包含公共的子问题，即子问题之间相互独立；
2. 子问题的解可以合并为原问题的解。

### 解题套路

### 适用问题

1. 二分搜索
2. 合并排序
3. 快速排序
4. 二叉树的生成
5. 大整数乘法
6. Strassen矩阵乘法
7. 棋盘覆盖
8. 线性时间选择
9. 最接近点对问题
10. 循环赛日程表
11. 汉诺塔

### 注意事项

## 贪心算法

### 问题特点

### 解题套路

### 适用问题

### 注意事项

## 动态规划

每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种**多阶段最优化决策解决问题**的过程就称为动态规划。

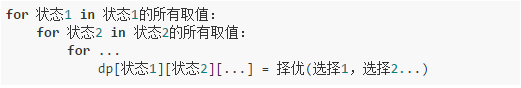
重叠子问题、最优子结构、状态转移方程是动态规划三要素。

### 问题特点

1. 子问题之间包含**公共的子问题**
2. 子问题之间相互独立；
3. 最优子结构，即问题的最优解所包含的子问题的解也是最优的；

### 解题套路

1. 明确「状态」，也就是原问题和子问题中变化的变量，
2. 定义 dp 数组/函数的含义
3. 明确「选择」
4. 明确 base case



### 适用问题

动态规划问题的一般形式就是求最值，

#### 背包问题

##### 二维数组形式

*int* coinChange(*int*\* *coins*, *int* *coinsSize*, *int* *amount*)

{

    if (coins == NULL || coinsSize <= 0 || amount < 0) {

        return 0;

    }

*int* \*\*dp = (*int* \*\*)malloc((coinsSize + 1) \* sizeof(*int* \*));

    for (*int* i = 0; i < coinsSize + 1; i++) {

        dp[i] = (*int* \*)malloc((amount + 1) \* sizeof(*int*));

    }

    for (*int* i = 0; i <= coinsSize; i++) {

        dp[i][0] = 0;

        for (*int* j = 1; j <= amount; j++) {

            dp[i][j] = amount + 1;

        }

    }

    for (*int* i = 1; i <= coinsSize; i++) {

        for (*int* j = 1; j <= amount; j++) {

            if (j >= coins[i - 1]) {

                dp[i][j] = fmin(dp[i - 1][j], dp[i][j - coins[i - 1]] + 1);

            } else {

                dp[i][j] = dp[i - 1][j];

            }

        }

    }

*int* ans = dp[coinsSize][amount];

    if (ans == amount + 1) {

        return -1;

    }

    return ans;

}

##### 一维数组形式

*int* coinChange(*int*\* *coins*, *int* *coinsSize*, *int* *amount*)

{

    if (coins == NULL || coinsSize <= 0 || amount < 0) {

        return 0;

    }

*int* \*dp = (*int* \*)malloc((amount + 1) \* sizeof(*int*));

    if (dp == NULL) {

        return 0;

    }

    dp[0] = 0;

    for (*int* i = 1; i <= amount; i++) {

        dp[i] = amount + 1;

    }

    for (*int* j = 0; j < coinsSize; j++) {

        for (*int* i = coins[j]; i <= amount; i++) {

            dp[i] = fmin(dp[i], dp[i - coins[j]] + 1);

        }

    }

    return dp[amount] == amount + 1 ? -1 : dp[amount];

}

##### 公式

###### 组合问题公式

dp[i] = dp[i] + dp[i - num]

###### 真假问题公式

dp[i] = dp[i] | dp[i - num]

###### 最大最小问题公式

dp[i] = min(dp[i], dp[i - num] + 1)

或

dp[i] = max(dp[i], dp[i - num] + 1)

##### 循环方向

###### 0-1背包问题

数组中的元素不可重复使用，nums放在外循环，target在内循环，且内循环倒序。

*void* ZeroOneBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    for (*int* i = 0; i < *numsSize*; i++) {

        for (*int* j = *target*; j >= *nums*[i]; j--) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

###### 完全背包问题

数组中的元素可重复使用，nums放在外循环，target在内循环。且内循环正序。

*void* CompleteBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    for (*int* i = 0; i < *numsSize*; i++) {

        for (*int* j =  *nums*[i]; j <= *target*; j++) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

###### 组合问题

需考虑元素之间的顺序，需将target放在外循环，将nums放在内循环。

*void* CompleteBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    dp[0] = 0;

    for (*int* j =  1; j <= *target*; j++) {

        for (*int* i = 0; i < *numsSize*; i++) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

#### 子序列子数组问题

#### 买卖股票问题

[121. 买卖股票的最佳时机](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/)

[122. 买卖股票的最佳时机 II](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/)

[123. 买卖股票的最佳时机 III](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iii/)

[188. 买卖股票的最佳时机 IV](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/)

[309. 最佳买卖股票时机含冷冻期](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/)

[714. 买卖股票的最佳时机含手续费](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/)

<https://mp.weixin.qq.com/s/61CU8PcT2z4Po7cVdwTprg>

#### 打家劫舍问题

### 注意事项

1. base case多试几次，自己认为的不一定是对的

## 回溯法

回溯法是一种搜索算法，从根节点出发，按照**深度优先搜索的策略**进行搜索，到达某一节点后 ，探索该节点是否包含该问题的解，如果包含则进入下一个节点进行搜索，若是不包含则回溯到父节点选择其他支路进行搜索。**回溯法一般来说是遍历整个解空间，获取问题的所有解。**

DFS 是一个劲的往某一个方向搜索，而回溯算法建立在 DFS 基础之上的，但不同的是在搜索过程中，达到结束条件后，恢复状态，回溯上一层，再次搜索。因此回溯算法与 DFS 的区别就是**有无状态重置。**

回溯法与深度优先搜索不同点

* **访问顺序**

深度优先遍历：

目的是“遍历”，本质是无序的。也就是说访问次序不重要，重要的是都被访问过了。因此在实现上，只需要对于每个位置记录是否被visited就足够了。

回溯法：

目的是“求解过程”，本质是有序的。也就是说必须每一步都是要求的次序。因此在实现上，不能使用visited记录，因为同样的内容不同的序访问就会造成不同的结果，而不是仅仅“是否被访问过”这么简单。要使用访问状态来记录，也就是对于每个点记录已经访问过的邻居方向，回溯之后从新的未访问过的方向去访问邻居。至于这点点之前有没有被访问过并不重要，重要的是没有以当前的序进行访问。

* 访问次数

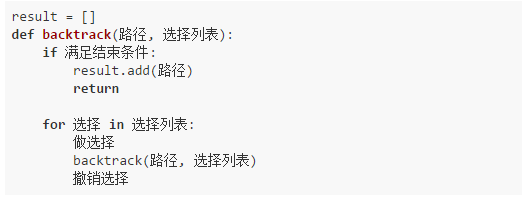
深度优先遍历：已经访问过的节点不再访问，所有点仅访问一次。

回溯法：已经访问过的点可能再次访问，也可能存在没有被访问过的点。

### 问题特点

### 解题套路

* 1. 路径：记录已经做出的选择。
  2. 选择列表：当前可以做的选择。
  3. 结束条件：到达决策树底层，无法再做选择的条件。



### 适用问题

#### 子集问题

#### 组合问题

#### 排列问题

#### 八皇后问题

#### 迷宫问题

#### 数独问题

### 注意事项

1. 定义宏不能太大或者太小；
2. 去重需要先排序，并传入下标的start值；
3. 子集问题，没有结束条件；

void BackTrack(int\* candidates, int candidatesSize, int target, int \*returnSize, int \*\*returnColumnSizes, int \*\*ans, int \*path, int curDepth, int start)

{

    if (target < 0) {

        return;

    }

    if (target == 0) {

        memcpy(ans[\*returnSize], path, curDepth \* sizeof(int));

        (\*returnColumnSizes)[\*returnSize] = curDepth;

        (\*returnSize)++;

        return;

    }

    for (**int i = start**; i < candidatesSize; i++) {

        if (i > start && candidates[i] == candidates[i - 1]) {

            continue;

        }

        path[curDepth] = candidates[i];

        BackTrack(candidates, candidatesSize, target - candidates[i], returnSize, returnColumnSizes, ans, path, curDepth + 1, **i + 1**);

        path[curDepth] = 0;

    }

}

#define M 10000

bool \*g\_selected;

int g\_count;

int \*g\_path;

int g\_numsSize;

void dfs(int \*nums, int \*\*ans, int curDepth)

{

    if (curDepth == g\_numsSize) {

        memcpy(ans[g\_count], g\_path, g\_numsSize \* sizeof(int));

        (g\_count)++;

        return;

    }

    for (int i = 0; i < g\_numsSize; i++) {

        if (g\_selected[i] == 0) {

            g\_selected[i] = 1;

            g\_path[curDepth] = nums[i];

            dfs(nums, ans, curDepth + 1);

            g\_selected[i] = 0;

        }

    }

}

int\*\* permute(int \*nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes)

{

    \*returnSize = 0;

    if (nums == NULL || numsSize <= 0) {

        \*returnColumnSizes = NULL;

        return NULL;

    }

    int \*\*ans = (int \*\*)malloc(M \* sizeof(int \*));

    if (ans == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    for (int i = 0; i < M; i++) {

        ans[i] = (int \*)malloc(numsSize \* sizeof(int));

    }

    g\_path = (int \*)malloc(numsSize \* sizeof(int));

    if (g\_path == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    g\_selected = (int \*)malloc(numsSize \* sizeof(int));

    if (g\_selected == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    g\_count = 0;

    g\_numsSize = numsSize;

    memset(g\_selected, 0, numsSize \* sizeof(int));

    memset(g\_path, 0, numsSize \* sizeof(int));

    dfs(nums, ans, 0);

    \*returnSize = g\_count;

    \*returnColumnSizes = (int \*)malloc((\*returnSize) \* sizeof(int));

    if ((\*returnColumnSizes) == NULL) {

        free(ans);

        return NULL;

    }

    for (int i = 0; i < \*returnSize; i++) {

        (\*returnColumnSizes)[i] = numsSize;

    }

    return ans;

}

## 分支限界法

### 问题特点

和回溯法相似，也是一种搜索算法，但回溯法是找出问题的许多解，而分支限界法是**找出原问题的一个解**。或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解

在当前节点（扩展节点）处，先生成其所有的儿子节点（分支），然后再从当前的活节点（当前节点的子节点）表中选择下一个扩展节点。为了有效地选择下一个扩展节点，加速搜索的进程，在每一个活节点处，计算一个函数值（限界），并根据函数值，从当前活节点表中选择一个最有利的节点作为扩展节点，使搜索朝着解空间上有最优解的分支推进，以便尽快地找出一个最优解。

分支限界法：

1）FIFO分支限界法

3）优先队列分支限界法：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

### 适用问题

#### 装载问题

#### 旅行售货员问题

### 解题套路

### 注意事项

# 数据结构

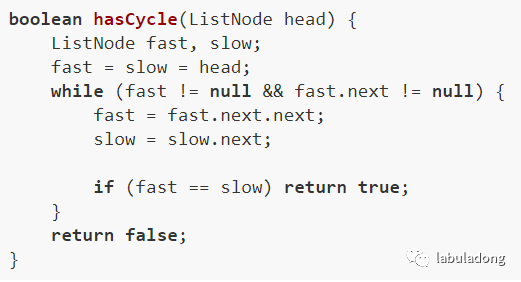
数据结构的存储方式只有两种：顺序存储和链式存储。对于任何数据结构，其基本操作无非遍历 + 访问，也就是增删查改。各种数据结构的遍历 + 访问无非两种形式：线性的和非线性的。线性就是 for/while 迭代为代表，非线性就是递归为代表。

## 数组与链表

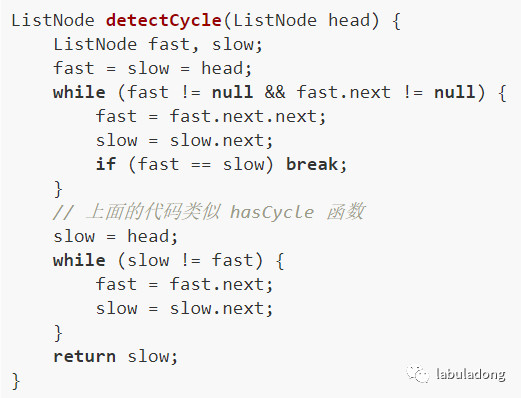
### 左右指针

### 快慢指针——成环问题

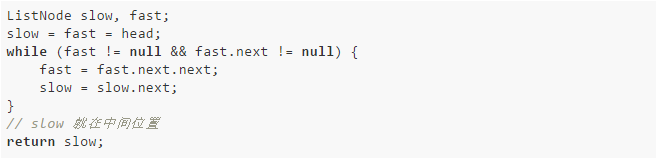
#### 判定链表中是否含有环



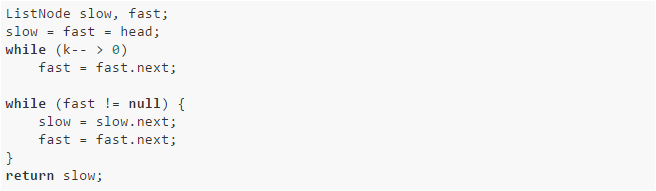
#### 已知链表中含有环，返回这个环的起始位置



#### 寻找链表的中点



#### 寻找链表的倒数第 k 个元素



### 滑动窗口——子串问题

双指针（滑动窗口）利用了二段性质：当一个指针确定在某个位置，另外一个指针能够落在某个明确的分割点，**使得左半部分满足，右半部分不满足。**

#define M 128

bool IsSatisfy(int \*need, int \*window)

{

    for (int i = 0; i < M; i++) {

        if (need[i] > window[i]) {

            return false;

        }

    }

    return true;

}

bool checkInclusion(char \* s1, char \* s2)

{

    if (s1 == NULL || s2 == NULL) {

        return false;

    }

    int \*need = (int \*)malloc(M \* sizeof(int));

    int \*window = (int \*)malloc(M \* sizeof(int));

    memset(need, 0, M \* sizeof(int));

    memset(window, 0, M \* sizeof(int));

    for (int i = 0; i < strlen(s1); i++) {

        need[s1[i]]++;

    }

    int left = 0;

    int right = 0;

    while (right < strlen(s2)) {

        char c = s2[right];

        right++;

        if (need[c] != 0) {

            window[c]++;

        }

        while (IsSatisfy(need, window) == true) {

            if (right - left == strlen(s1)) {

                return true;

            }

            char c = s2[left];

            left++;

            if (need[c] != 0) {

                window[c]--;

            }

        }

    }

    return false;

}

## 二分法

前提：有序

### 解题套路

*int* BinarySearch(*int* \**nums*, *int* *numsSize*){

    if (nums == NULL) {

        return 0;

    }

*int* left = 0;

*int* right = numsSize - 1;

    while (left < right) {

*int* mid = left + (right - left) / 2;

        if (nums[mid] > nums[right]) {

            left = mid + 1;

        } else if (nums[mid] < nums[right]) {

            right = mid;

        } else {

            right--;

        }

    }

    return nums[left];

}

### 注意事项

1. left的初始值为0，right的初始值为numsSize – 1，**两者是能取到的左右边界；**

while语句中的判定条件：当while循环中出现left = mid或者right = mid时，写为left < right，否者可以写为left <= right；**建议一律写为left < right；**

while循环中if语句建议写成if…else if…else的形式；

while循环中if语句是否加减1：当确定mid或者nums[mid]不是最终答案时，可以加减1，否者不能；

是left = mid + 1; right = mid 还是left = mid; right = mid – 1：建议取left = mid + 1; right = mid，即寻找是从右边开始的第一个满足条件的，而不是从左边开始最后一个不满足条件的。

最后的返回值：如果while循环中有return语句，需要对最后的nums[left]作特殊处理；**如果按照第5条，在[left, right]中不存在满足条件情况的，需要对最后的返回值作判断。**

## 字符串

### 字符串函数

#### 长度不受限的字符串函数

strcpy()

strcat()

strcmp()

#### 长度受限的字符串函数

strncpy(char \*dst, char const \*src, size\_t len)

向dst写入len个字符：

1）strlen(src) < len, dst数组使用NULL填充到len长度；

2）strlen(src) >= len, len个字符复制到dst中；

3）**结果不会以NULL结尾。**

strncat(char \*dst, char const \*src, size\_t len)

1）strlen(src) < len, 将strlen(src)个字符连接到dst后面，并在后面添加NULL；

2）strlen(src) >= len, 将len个字符连接到dst后面，并在后面添加NULL；

3）**不管目标参数的剩余空间够不够，不够的话会覆盖掉已有的数据，总是在字符串后面添加NULL**

strncmp(char const \*s1, char const \*s2, size\_t len)

#### 字符串拆分函数 char \*strtok(char \*src, char const \*splits)

int main(int argc, char \*argv[]) {

    char src[] = "A,B,C,D E F G";

    char splits[] = " ,";

    char \*token = NULL;

    token = strtok(src, splits);

    while (token != NULL) {

        puts(token);

        token = strtok(NULL, splits);

    }

    getchar();

    return 0;

}

#### 查找字符串函数 char \*strstr(char \*const s1, char const \*s2)

1. 查找s2在s1中第一次出现的位置，并返回指向该位置的指针；
2. 如果没有找到，返回NULL；
3. 如果s2 == NULL，返回s1。

#### 查找字符函数 char \*strstr(char \*const str, char ch)

1. 查找ch在str中第一次出现的位置，并返回指向该位置的指针；
2. 如果没有找到，返回NULL；

### 模式匹配

#### BF算法

时间复杂度：N\*M

#### KMP算法

## 树

### 深度优先搜索

1. 基本情况；

* 做事；
* 递归；

前序遍历

#### Top Down DFS

* 把值通过参数的形式从上往下传；
* 一般dfs()本身不返回值；

#### Bottom Up DFS

* 把值从下往上传；
* 当前递归层利用子问题传上来的值计算当前层的新值并返回；
* 一定会有返回值；

**一般流程**

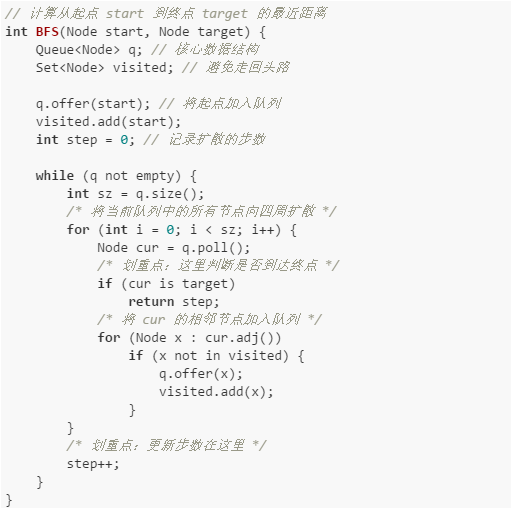
1. 基本情况；
2. 向子问题要答案；
3. 利用子问题答案构建当前问题答案；
4. 返回答案给父问题。

#### 是否需要双重递归：路径不需要从根节点开始，需要双重递归。

### 广度优先搜索

本质上就是一幅「图」，让你从一个起点，走到终点，问最短路径。

#### C语言套路



## 深度优先搜索和回溯法的区别

深度优先搜索传递值，不需要撤销选择。

### 深度优先搜索

void BackTrace(struct TreeNode \*root, int targetSum, int \*returnSize, int \*\*returnColumnSizes, int \*\*ans, int \*path, int curDepth, int sum)

{

    if (root == NULL) {

        return;

    }

    sum += root->val;

    path[curDepth] = root->val;

    curDepth++;

    if (targetSum == sum && root->left == NULL && root->right == NULL) {

        memcpy(ans[\*returnSize], path, curDepth \* sizeof(int));

        (\*returnColumnSizes)[\*returnSize] = curDepth;

        (\*returnSize)++;

        return;

    }

    BackTrace(root->left, targetSum, returnSize, returnColumnSizes, ans, path, curDepth, sum);

    BackTrace(root->right, targetSum, returnSize, returnColumnSizes, ans, path, curDepth, sum);

    return;

}

### 回溯法

void BackTrace(struct TreeNode \*root, int targetSum, int \*returnSize, int \*\*returnColumnSizes, int \*\*ans, int \*path, int curDepth, int \*sum)

{

    if (root == NULL) {

        return;

    }

    if (targetSum == \*sum + root->val && root->left == NULL && root->right == NULL) {

        path[curDepth] = root->val;

        curDepth++;

        memcpy(ans[\*returnSize], path, curDepth \* sizeof(int));

        (\*returnColumnSizes)[\*returnSize] = curDepth;

        (\*returnSize)++;

        return;

    }

    \*sum += root->val;

    path[curDepth] = root->val;

    BackTrace(root->left, targetSum, returnSize, returnColumnSizes, ans, path, curDepth + 1, sum);

    BackTrace(root->right, targetSum, returnSize, returnColumnSizes, ans, path, curDepth + 1, sum);

    path[curDepth] = 0;

    \*sum -= root->val;

    return;

}

## 图

## 前缀和和查分数组

前缀和的应用场景是，需要对某个区间[i...j]频繁查询累计和，避免每次查询都遍历这个区间。

差分数组的应用场景是，需要对某个区间[i...j]频繁地加或减某一值，避免每次都遍历这个区间。

# UTHASH的使用

#include “uthash.h”

## 3个数据结构

### UT\_hash\_handle

### UT\_hash\_table

### UT\_hash\_bucket

## 自定义结构体

typedef *struct* MyNode {

*int* key;

*char* value[10];

    UT\_hash\_handle hh;

} MyNode;

MyNode \*head = NULL;

* key的类型不一样，后面的插入、查找调用的接口函数就不一样，因此要求确保key的类型与uthash的接口函数一致。
* 必须提供UT\_hash\_handle变量hh，无需为其初始化。
* 定义一个hash结构的空指针users，用于指向保存数据的hash表，必须初始化为空，在后面的查、插等操作中，uthash内部会根据其是否为空而进行不同的操作

## 7个函数

add/replace

find

delete

count

iterate

sort

### 以int类型为key

typedef struct Node {

    int id;

    char name[10];

    UT\_hash\_handle hh;

} Node;

Node \*g\_head = NULL;

#### HASH\_FIND\_INT(head, findint, out)

#### HASH\_ADD\_INT(head, intfield, add)

void AddNode(int user\_id, char \*name)

{

    Node \*node = NULL;

    HASH\_FIND\_INT(g\_head, &user\_id, node);

    if (node == NULL) {

      node = (Node \*)malloc(sizeof(Node));

      node->id = user\_id;

      HASH\_ADD\_INT(g\_head, id, node);

    }

    strcpy(node->name, name);

}

#### HASH\_DEL(head, delptr)

#### HASH\_ITER(hh, head, cur, tmp)

void DeleteNode(int user\_id)

{

    Node \*node = NULL;

    HASH\_FIND\_INT(g\_head, &user\_id, node);

    if (node != NULL) {

      HASH\_DEL(g\_head, node);

      free(node);

    }

    return;

}

void DeleteAllNode(int user\_id)

{

    Node \*cur = NULL;

    Node \*next = NULL;

    HASH\_ITER(hh, g\_head, cur, next) {

        HASH\_DEL(g\_head, cur);

        free(cur);

    }

    return;

}

#### HASH\_SORT(head, cmpfcn)

int CompareByValue(Node \*a, Node \*b)

{

    return strcmp(a->name, b->name);

}

int CompareByKey(Node \*a, Node \*b)

{

    return (a->id - b->id);

}

void SortByValue()

{

    HASH\_SORT(g\_head, CompareByValue);

}

void SortByKey()

{

    HASH\_SORT(g\_head, CompareByKey);

}

#### HASH\_COUNT(head)

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

### 以char[]类型为key

typedef struct Node {

    int id;

    char name[10];

    UT\_hash\_handle hh;

} Node;

Node \*g\_head = NULL;

#### HASH\_FIND\_STR(head, findstr, out)

#### HASH\_ADD\_STR(head,strfield,add)

void AddNode(char \*name, int user\_id)

{

    Node \*node = NULL;

    HASH\_FIND\_STR(g\_head, name, node);

    if (node == NULL) {

      node = (Node \*)malloc(sizeof(Node));

      strcpy(node->name, name);

      node->id = user\_id;

      HASH\_ADD\_STR(g\_head, name, node);

    }

    return;

}

#### HASH\_DEL(head, delptr)

#### HASH\_ITER(hh, head, cur, tmp)

void DeleteNode(char \*name)

{

    Node \*node = NULL;

    HASH\_FIND\_STR(g\_head, name, node);

    if (node != NULL) {

      HASH\_DEL(g\_head, node);

      free(node);

    }

    return;

}

void DeleteAllNode()

{

    Node \*cur = NULL;

    Node \*next = NULL;

    HASH\_ITER(hh, g\_head, cur, next) {

        HASH\_DEL(g\_head, cur);

        free(cur);

    }

    return;

}

#### HASH\_SORT(head, cmpfcn)

int CompareByKey(Node \*a, Node \*b)

{

    return strcmp(a->name, b->name);

}

int CompareByValue(Node \*a, Node \*b)

{

    return (a->id - b->id);

}

void SortByValue()

{

    HASH\_SORT(g\_head, CompareByValue);

}

void SortByKey()

{

    HASH\_SORT(g\_head, CompareByKey);

}

#### HASH\_COUNT(head)

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

### 以char \*类型为key

typedef struct Node {

    int id;

    char name[10];

    UT\_hash\_handle hh;

} Node;

Node \*g\_head = NULL;

#### HASH\_FIND\_STR(head, findstr, out)

#### HASH\_ADD\_KEYPTR(hh, head, keyptr, keylen\_in, add)

void AddNode(char \*name, int user\_id)

{

    Node \*node = NULL;

    HASH\_FIND\_STR(g\_head, name, node);

    if (node == NULL) {

      node = (Node \*)malloc(sizeof(Node));

      node->name = name;

      node->id = user\_id;

      HASH\_ADD\_KEYPTR(hh,g\_head, node->name, strlen(node->name), node);

    }

    return;

}

#### HASH\_DEL(head, delptr)

#### HASH\_ITER(hh, head, cur, tmp)

void DeleteNode(char \*name)

{

    Node \*node = NULL;

    HASH\_FIND\_STR(g\_head, name, node);

    if (node != NULL) {

      HASH\_DEL(g\_head, node);

      free(node);

    }

    return;

}

void DeleteAllNode()

{

    Node \*cur = NULL;

    Node \*next = NULL;

    HASH\_ITER(hh, g\_head, cur, next) {

        HASH\_DEL(g\_head, cur);

        free(cur);

    }

    return;

}

#### HASH\_SORT(head, cmpfcn)

int CompareByValue(Node \*a, Node \*b)

{

    return strcmp(a->name, b->name);

}

int CompareByKey(Node \*a, Node \*b)

{

    return (a->id - b->id);

}

void SortByValue()

{

    HASH\_SORT(g\_head, CompareByValue);

}

void SortByKey()

{

    HASH\_SORT(g\_head, CompareByKey);

}

#### HASH\_COUNT(head)

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

### 以地址void\*类型为key

#### HASH\_ADD\_PTR(head, ptrfield, add)

#### HASH\_ADD\_PTR(head, ptrfield, add)

typedef struct Node {

    void \*key;

    int value;

    UT\_hash\_handle hh;

} Node;

Node \*g\_head = NULL;

void AddNode(void \*key, int value)

{

    Node \*node = NULL;

    HASH\_FIND\_PTR(g\_head, key , node);

    if (node == NULL) {

      node = (Node \*)malloc(sizeof(Node));

      node->key = key;

      node->value = value;

      HASH\_ADD\_PTR(g\_head, key, node);

    }

    return;

}

#### HASH\_DEL(head, delptr)

#### HASH\_ITER(hh, head, cur, tmp)

void DeleteNode(void \* key)

{

    Node \*node = NULL;

    HASH\_FIND\_PTR(g\_head, key , node);

    if (node != NULL) {

      HASH\_DEL(g\_head, node);

      free(node);

    }

    return;

}

void DeleteAllNode()

{

    Node \*cur = NULL;

    Node \*next = NULL;

    HASH\_ITER(hh, g\_head, cur, next) {

        HASH\_DEL(g\_head, cur);

        free(cur);

    }

    return;

}

#### HASH\_SORT(head, cmpfcn)

int CompareByKey(Node \*a, Node \*b)

{

    char \*stra = (char \*)(a->key);

    char \*strb = (char \*)(b->key);

    return strcmp(stra, strb);

}

int CompareByValue(Node \*a, Node \*b)

{

    return (a->value - b->value);

}

void SortByValue()

{

    HASH\_SORT(g\_head, CompareByValue);

}

void SortByKey()

{

    HASH\_SORT(g\_head, CompareByKey);

}

#### HASH\_COUNT(head)

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

### 以任意类型为key

typedef struct Key {

    char a;

    int b;

    UT\_hash\_handle hh;

} Key;

typedef struct Node {

    Key key;

    int value;

    UT\_hash\_handle hh;

} Node;

Node \*g\_head = NULL;

#### HASH\_FIND(hh, head, keyptr, keylen, out)

#### HASH\_ADD(hh, head, fieldname, keylen\_in, add)

void AddNode(Key key, int value)

{

    Node \*node = NULL;

    HASH\_FIND(hh, g\_head, &key, sizeof(Key), node);

    if (node == NULL) {

      node = (Node \*)malloc(sizeof(Node));

      node->key = key;

      node->value = value;

      HASH\_ADD(hh, g\_head, key, sizeof(Key), node);

    }

    return;

}

#### HASH\_DEL(head, delptr)

#### HASH\_ITER(hh, head, cur, tmp)

void DeleteNode(Key key)

{

    Node \*node = NULL;

    HASH\_FIND(hh, g\_head, &key, sizeof(Key), node);

    if (node != NULL) {

      HASH\_DEL(g\_head, node);

      free(node);

    }

    return;

}

void DeleteAllNode()

{

    Node \*cur = NULL;

    Node \*next = NULL;

    HASH\_ITER(hh, g\_head, cur, next) {

        HASH\_DEL(g\_head, cur);

        free(cur);

    }

    return;

}

#### HASH\_SORT(head, cmpfcn)

int CompareByKey(Node \*a, Node \*b)

{

    if (a->key.a != b->key.a) {

        return a->key.a - b->key.a;

    }

    return a->key.b, b->key.b;

}

int CompareByValue(Node \*a, Node \*b)

{

    return (a->value - b->value);

}

void SortByValue()

{

    HASH\_SORT(g\_head, CompareByValue);

}

void SortByKey()

{

    HASH\_SORT(g\_head, CompareByKey);

}

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

#### HASH\_COUNT(head)

int GetCount()

{

    return HASH\_COUNT(g\_head);

}

# 技巧

## m\*n的二维数组的下标i和j与展开后的一维数组的下标k的关系：

### i = k / n;

### j = k % n;

## 求余数的加法与乘法分配律

(a + b)% c = ((a % c) + (b % c)) % c

(a \* b )% c = ((a % c) 8 (b % c)) % c

## 添加哑结点

## 判断一个数是不是2的次幂

n != 0 && ((n & (n - 1)) == 0)

# 精选题目：

## [378. 有序矩阵中第 K 小的元素](https://leetcode-cn.com/problems/kth-smallest-element-in-a-sorted-matrix/)

## [1712. 将数组分成三个子数组的方案数](https://leetcode-cn.com/problems/ways-to-split-array-into-three-subarrays/)

## [395. 至少有 K 个重复字符的最长子串](https://leetcode-cn.com/problems/longest-substring-with-at-least-k-repeating-characters/)

# Linux内核无锁缓冲队列kfifo原理

## kfifo概述

kfifo是Linux内核（**Kernel**）里面的一个先进先出（**First In First Out**）数据结构。它采用**环形循环队列**的数据结构来实现，提供一个无边界的字节流服务；最重要的一点是，它使用**并行无锁编程技术**，即当它用于只有一个**入队线程**和一个**出队线程**的场景时，两个线程可以并发操作，而不需要任何加锁行为，就可以保证kfifo的线程安全。**鬼斧神工，匠心独运，简约高效，妙不可言，巧夺天工，让人叹为观止，拍手叫绝。**

## kfifo数据结构定义

### kfifo结构体定义

struct kfifo {

    unsigned char \*buffer; */\* the buffer holding the data \*/*

    unsigned int size;   */\* the size of the allocated buffer \*/*

    unsigned int in;     */\* data is added at offset (in % size) \*/*

    unsigned int out;    */\* data is extracted from off. (out % size) \*/*

    spinlock\_t \*lock;    */\* protects concurrent modifications \*/*

};

各个数据成员如下：

* buffer: 用于存放数据的缓存；
* size: buffer空间的大小，在初化时，将它向上扩展成2的幂；
* in, out: 和buffer一起构成一个循环队列。 in指向buffer中队头，而且out指向buffer中的队尾，它的结构如示图如下：
* lock: 如果使用不能保证任何时间最多只有一个读线程和写线程，需要使用该lock实施同步；

+--------------------------------------------------------------+

|            |<----------data---------->|                      |

+--------------------------------------------------------------+

             ^                          ^                      ^

             |                          |                      |

            out                        in                     size

### kfifo内存分配和初始化——kfifo\_alloc()

struct kfifo \*kfifo\_alloc(unsigned int size, gfp\_t gfp\_mask, spinlock\_t \*lock)

{

    unsigned char \*buffer;

    struct kfifo \*ret;

*/\**

*\* round up to the next power of 2, since our 'let the indices*

*\* wrap' tachnique works only in this case.*

*\*/*

    if (size & (size - 1)) {

        BUG\_ON(size > 0x80000000);

        size = roundup\_pow\_of\_two(size);

    }

    buffer = kmalloc(size, gfp\_mask);

    if (!buffer)

        return ERR\_PTR(-ENOMEM);

    ret = kfifo\_init(buffer, size, gfp\_mask, lock);

    if (IS\_ERR(ret))

        kfree(buffer);

    return ret;

}

在kfifo\_alloc函数中，**使用size & (size - 1)来判断size 是否为2幂**，如果条件为真，则表示size不是2的幂，然后调用roundup\_pow\_of\_two将之向上扩展为2的幂。

static inline unsigned int roundup\_pow\_of\_two(unsigned int size)

{

    if (size == 0)

        return 0;

    if ((size & (size - 1)) == 0) {

        return size;

    }

    unsigned int position = 0;

    for (int i = size; i != 0; i >>= 1)

        position++;

    return static\_cast<unsigned int>(1 << position);

}

kfifo->size的值总是在调用者传进来的size参数的基础上向2的幂扩展。这样做的目的是将kfifo->size取模运算可以转化为**与运算。**

kfifo->in % kfifo->size 可以转化为 kfifo->in & (kfifo->size – 1)

### kfifo入队操作——\_\_kfifo\_put()

unsigned int \_\_kfifo\_put(struct kfifo \*fifo, unsigned char \*buffer, unsigned int len)

{

    unsigned int l;

    len = min(len, fifo->size - fifo->in + fifo->out);

*/\**

*\* Ensure that we sample the fifo->out index -before- we*

*\* start putting bytes into the kfifo.*

*\*/*

    smp\_mb();

*/\* first put the data starting from fifo->in to buffer end \*/*

    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));

    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);

*/\* then put the rest (if any) at the beginning of the buffer \*/*

    memcpy(fifo->buffer, buffer + l, len - l);

*/\**

*\* Ensure that we add the bytes to the kfifo -before-*

*\* we update the fifo->in index.*

*\*/*

    smp\_wmb();

    fifo->in += len;

    return len;

}

6行，环形缓冲区的剩余容量为fifo->size - fifo->in + fifo->out，让写入的长度取len和剩余容量中较小的，避免写越界；

13行，加内存屏障，保证在开始放入数据之前，fifo->out取到正确的值（另一个CPU可能正在改写out值）

16行，前面讲到fifo->size已经2的次幂圆整，而且kfifo->in % kfifo->size 可以转化为 kfifo->in & (kfifo->size – 1)，所以fifo->size - (fifo->in & (fifo->size - 1)) 即位 fifo->in 到 buffer末尾所剩余的长度，l取len和剩余长度的最小值，即为需要拷贝l 字节到fifo->buffer + fifo->in的位置上。

17行，拷贝l 字节到fifo->buffer + fifo->in的位置上，如果l = len，则已拷贝完成，第20行len – l 为0，将不执行，如果l = fifo->size - (fifo->in & (fifo->size - 1)) ，则第20行还需要把剩下的 len – l 长度拷贝到buffer的头部。

27行，加写内存屏障，保证in 加之前，memcpy的字节已经全部写入buffer，如果不加内存屏障，可能数据还没写完，另一个CPU就来读数据，读到的缓冲区内的数据不完全，因为读数据是通过 in – out 来判断的。

29行，注意这里 只是用了 fifo->in +=  len而未取模，这就是kfifo的设计精妙之处，这里用到了unsigned int的溢出性质，当in 持续增加到溢出时又会被置为0，这样就节省了每次in向前增加都要取模的性能。

1. 往kfifo里面写一块长为len数据时：

a) kfifo ->size - (kfifo ->in % kfifo ->size) <= len

                                  kfifo\_put（写）空间开始地址

                                  |

                                  \\_/

                                  |XXXXXXXXXXXXXXXXXXX|

+--------------------------------------------------------------+

|              |<------data------>|                      |

+--------------------------------------------------------------+

               ^                  ^                            ^

               |                  |                            |

             out%size           in%size                       size

         ^

         |

       写空间结束地址

b) kfifo ->size - (kfifo ->in % kfifo ->size) > len

                                            kfifo\_put（写）空间开始地址

                                            |

                                            \\_/

                                            |XXXXXXXXXXXXXXXXXXX

XXXXXXXX|

+--------------------------------------------------------------+

|                        |<------data------>|           |

+--------------------------------------------------------------+

                         ^                  ^                  ^

                         |                  |                  |

                       out%size           in%size             size

        ^

        |

      写空间结束地址

第一块是：[kfifo->in % kfifo->size, kfifo->size]

第二块是：[0, len - (kfifo->size - kfifo->in % kfifo->size)]

### kfifo出队操作——\_\_kfifo\_get()

unsigned int \_\_kfifo\_get(struct kfifo \*fifo, unsigned char \*buffer, unsigned int len)

{

    unsigned int l;

    len = min(len, fifo->in - fifo->out);

*/\**

*\* Ensure that we sample the fifo->in index -before- we*

*\* start removing bytes from the kfifo.*

*\*/*

    smp\_rmb();

*/\* first get the data from fifo->out until the end of the buffer \*/*

    l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));

    memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);

*/\* then get the rest (if any) from the beginning of the buffer \*/*

    memcpy(buffer + l, fifo->buffer, len - l);

*/\**

*\* Ensure that we remove the bytes from the kfifo -before-*

*\* we update the fifo->out index.*

*\*/*

    smp\_mb();

    fifo->out += len;

    return len;

}

6行，可去读的长度为fifo->in – fifo->out，让读的长度取len和剩余容量中较小的，避免读越界；

13行，加读内存屏障，保证在开始取数据之前，fifo->in取到正确的值（另一个CPU可能正在改写in值）

16行，前面讲到fifo->size已经2的次幂圆整，而且kfifo->out % kfifo->size 可以转化为 kfifo->out & (kfifo->size – 1)，所以fifo->size - (fifo->out & (fifo->size - 1)) 即位 fifo->out 到 buffer末尾所剩余的长度，l取len和剩余长度的最小值，即为从fifo->buffer + fifo->in到末尾所要去读的长度。

17行，从fifo->buffer + fifo->out的位置开始读取l长度，如果l = len，则已读取完成，第20行len – l 为0，将不执行，如果l =fifo->size - (fifo->out & (fifo->size - 1)) ，则第20行还需从buffer头部读取 len – l 长。

27行，加内存屏障，保证在修改out前，已经从buffer中取走了数据，如果不加屏障，可能先执行了增加out的操作，数据还没取完，令一个CPU可能已经往buffer写数据，将数据破坏，因为写数据是通过fifo->size - (fifo->in & (fifo->size - 1))来判断的 。

29行，注意这里 只是用了 fifo->out +=  len 也未取模，同样unsigned int的溢出性质，当out 持续增加到溢出时又会被置为0，如果in先溢出，出现 in  < out 的情况，那么 in – out 为负数（又将溢出），in – out 的值还是为buffer中数据的长度。

对于给定的kfifo：

1. kfifo每次入队或出队时，kfifo->in或kfifo->out只是+len，并没有对kfifo->size 进行取模运算，但它对应在kfifo->buffer空间的下标却是如下：

kfifo->in % kfifo->size kfifo->in & (kfifo->size - 1)

kfifo->out % kfifo->size kfifo->out & (kfifo->size - 1)

1. 虽然kfifo->in和kfifo->out一直增大，直到达到unsigned in最大值时自动溢出。利用溢出后的值参与运算，并且能够保证结果的正确。溢出运算保证了以下几点，这也是最能体现kfifo实现技巧的地方：

a) 缓冲区中的数据长度：kfifo->in - kfifo->out

b) 缓冲区中的空闲空间：kfifo->size - (kfifo->in - kfifo->out)

c) 缓冲区为空：kfifo->in == kfifo->out

d) 缓冲区已满：kfifo->size == (kfifo->in - kfifo->out)

## kfifo无锁并发与内存屏障

### 无锁并发

计算机科学家已经证明，当**只有一个读进程和一个写线程并发操作时**，不需要任何额外的锁，就可以确保是线程安全的，也即kfifo使用了**无锁编程技术**，以提高kernel的并发。

kfifo使用in和out两个指针来描述写入和读取游标，对于写入操作，只更新in指针，而读取操作，只更新out指针。

\_\_kfifo\_put是入队操作，它先将数据放入buffer里面，最后才修改in参数；\_\_kfifo\_get是出队操作，它先将数据从buffer中移走，最后才修改out参数

                                               |<--写入-->|

+--------------------------------------------------------------+

|                        |<----------data----->|               |

+--------------------------------------------------------------+

                         |<--读取-->|

                         ^                     ^               ^

                         |                     |               |

                        out                   in              size

### 内存屏障

为了避免读者看到写者预计写入，但实际没有写入数据的空间，写者必须保证以下的写入顺序：

往[kfifo->in, kfifo->in + len]空间写入数据

更新kfifo->in指针为 kfifo->in + len

在操作1完成时，读者是还没有看到写入的信息的，因为kfifo->in没有变化，认为读者还没有开始写操作，只有更新kfifo->in之后，读者才能看到。

那么如何保证1必须在2之前完成，秘密就是使用内存屏障：smp\_mb()，smp\_rmb(), smp\_wmb()，来保证对方观察到的内存操作顺序。

编译器编译源代码时，会将源代码进行优化，**将源代码的指令进行重排序**，以适合于CPU的并行执行。然而，内核同步必须**避免指令重新排序**，优化屏障（Optimization barrier）避免编译器的重排序优化操作，**保证编译程序时在优化屏障之前的指令不会在优化屏障之后执行。**

软件可通过读写屏障强制内存访问次序。读写屏障像一堵墙，**所有在设置读写屏障之前发起的内存访问，必须先于在设置屏障之后发起的内存访问之前完成**，确保内存访问按程序的顺序完成。Linux内核提供的内存屏障API函数说明如下表。内存屏障可用于多处理器和单处理器系统，如果仅用于多处理器系统，就使用smp\_xxx函数，在单处理器系统上，它们什么都不要。

|  |  |
| --- | --- |
| smp\_rmb | 适用于多处理器的读内存屏障。 |
| smp\_wmb | 适用于多处理器的写内存屏障。 |
| smp\_mb | 适用于多处理器的内存屏障。 |

## Linux内核实现的kfifo的特点：

### 使用内存屏障 Memory Barrier

### 初始化缓冲区空间时要保证缓冲区的大小为2的幂次方

### 使用无符号整数保存in和out（输入输出的指针），并且在放入取出数据的时候不做模运算，让其自然溢出。

通过限定写入的数据不能溢出和内存屏障实现在单线程写，单线程读的情况下不使用锁。因为锁是使用在共享资源可能存在冲突的情况下。还用设置buffer缓冲区的大小为2的幂次方，以简化求模运算，这样求模运算就演变为 (fifo->in & (fifo->size - 1))。通过使用unsigned int为kfifo的下标，可以不用考虑每次下标超过size时对下表进行取模运算赋值，这里使用到了无符号整数的溢出回零的特性。由于指示读写指针的下标一直在增加，没有进行取模运算，知道其溢出，在这种情况下写满和读完就是不一样的标志，写满是两者指针之差为fifo->size，读完的标志是两者指针相等。后面有一篇博客还介绍了VxWorks下的环形缓冲区的实现机制点击打开链接，从而可以看出linux下的fifo的灵巧性和高效性。

## kfifo结构的优点

1. 采用环形缓冲区，当一个数据元素被用掉后，其余数据元素不需要移动其存储位置，从而减少拷贝提高效率。
2. kfifo采用了并行无锁技术，kfifo实现的单生产/单消费模式的共享队列是不需要加锁同步的。多消费者和多生产者的时候还是需要加锁的。
3. 使用与运算in & (size-1)代替模运算。

## 编程技巧总结

### 判断一个整数n是不是2的幂次方

n != 0 && ((n & (n - 1)) == 0)

### 将一个整数n向上圆整为2的幂次方

### 对2的幂次方取余可以转化为与运算

kfifo->in % kfifo->size 可以转化为 kfifo->in & (kfifo->size – 1)

假设x为任意正整数，，证明。

1. ,









2. 









### 循环队列

一个进程A产生数据发给另外一个进程B，进程B需要对进程A传的数据进行处理并写入文件，如果B没有处理完，则A要延迟发送。为了保证进程A减少等待时间，可以在A和B之间采用一个缓冲区，A每次将数据存放在缓冲区中，B每次冲缓冲区中取。这是典型的生产者和消费者模型，缓冲区中数据满足FIFO特性，因此可以采用队列进行实现。

### 无符号整数溢出

在in先溢出的情况，size = 64， 写入前in = 4294967291，out = 4294967279，数据长度为 in - out = 12;

再写入16个字节数据，则 in + 16 = 4294967307，溢出为 11，此时 in - out = -4294967268，溢出为28，由kfifo->in - kfifo->out计算得到的数据长度仍然正确。