# 作业讲解

2024年5月10日 9:33

**Leetcode 98. 验证二叉搜索树**

给你一个二叉树的根节点 root ， 判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

节点的左子树只包含 小于 当前节点的数。
节点的右子树只包含 大于 当前节点的数。
所有左子树和右子树自身必须也是二叉搜索树。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool isValidBST(struct TreeNode* root) {

}
```

max

中序遍历是有序的

L < D < R

citations[r] < (n-r)

citations[l] ≥ (n-l)

① h = n-l

② h = n-l

③ h = n-l

[0, 0, 0, ---, 0]

# 归并排序

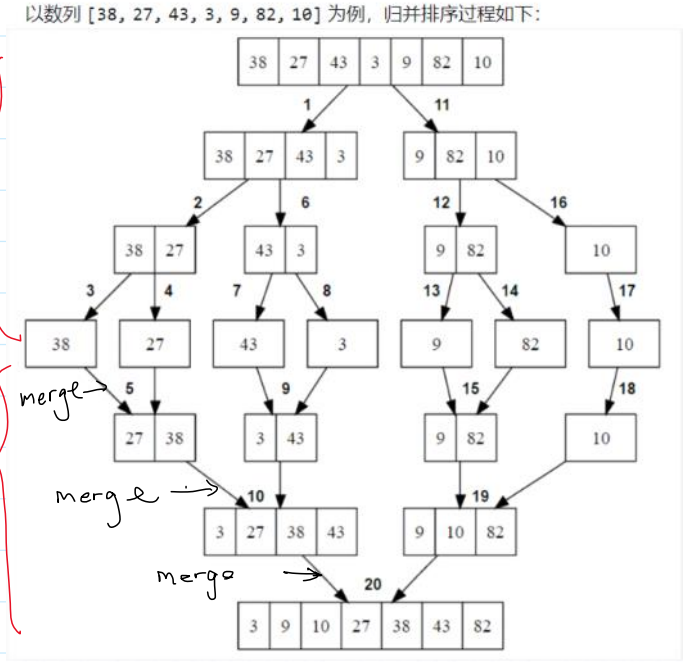
2024年5月10日 10:56

分治思想，算法设计思想

⇓

递归，代码的实现方式

## 1. 算法步骤




## 2. 实现

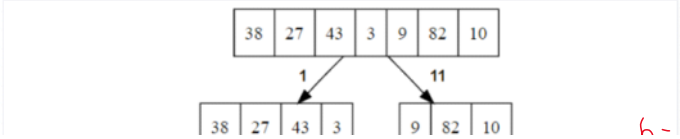
```c
void m_sort(int arr[], int left, int right) {
    // 边界条件
    if (left >= right) return;
    // 递归公式
    int mid = left + (right - left >> 1);
    // 对左边区间排序
    m_sort(arr, left, mid);
    // 对右边区间排序
    m_sort(arr, mid + 1, right);
    // 归并
    merge(arr, left, mid, right);

    print_array(arr, N);
}
```

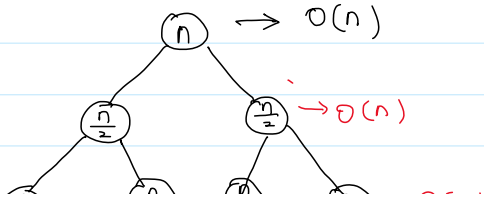
## 3. 分析

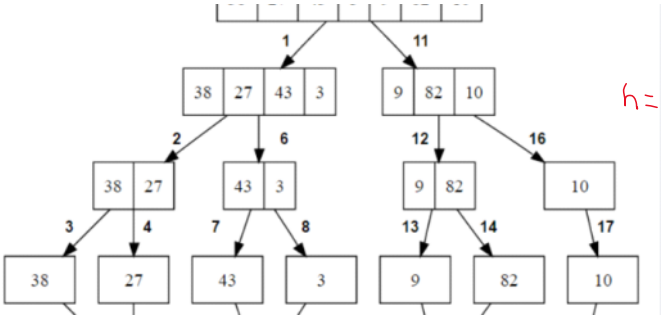
$O(n\log n)$

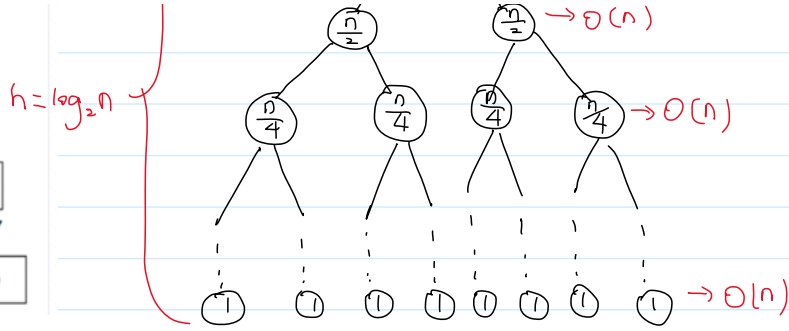
$T(n) = 2T(\frac{n}{2}) + O(n)$

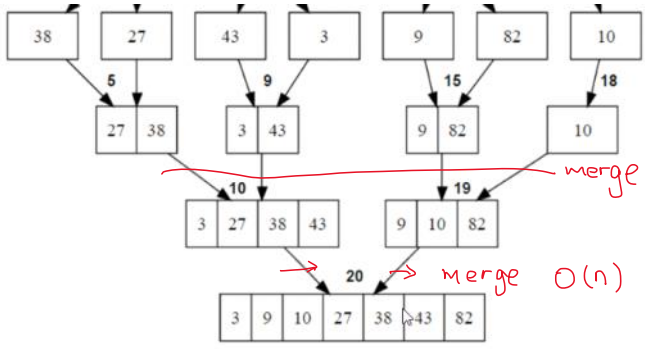



递归树






$T(N) = \log_2 n \cdot O(n)$

$= \log_2 n \quad c \cdot n$

$= O(n \log_2 n)$

$= O(n \log n)$

对数据无敏感

换底公式：

$$\log_b n = \log_b a^{\log_a n} = \log_a n \cdot \log_b a$$

$$\frac{\log_b n}{\log_a n} = \log_b a$$

#2. 空间复杂度

$O(\log n) + O(n) = O(n)$

太高了.

#3. 稳定性

稳定

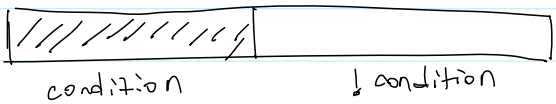
# 快速排序

2024年5月10日 14:23

分治思想

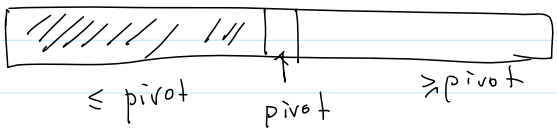
## 1. 算法步骤

① 分区 partition




condition

选取一个基准值：pivot




condition：≤ pivot

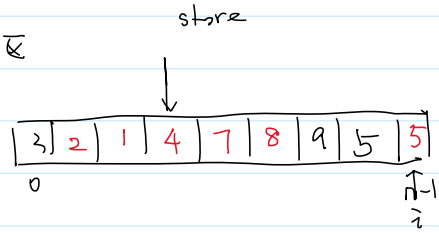
② 对左边进行快速排序

③ 对右边进行快速排序

## 2. 实现

如何分区：

a. 单向分区




pivot = 4

store：下一个 ≤ pivot 的元素应该置于的位置

不变式

[0, store)：≤ pivot 满足条件

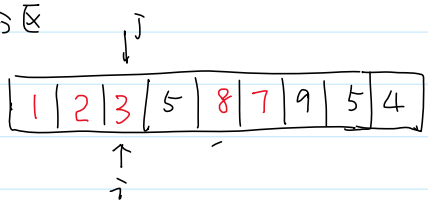
[store, i)：> pivot 不满足条件

[i, n-1)：？

终止：i == n-1

平均：$\frac{n}{2} \times 3 = 1.5n$ （赋值）

b. 双向分区




i：下一个 ≤ pivot 置于的位置

j：下一个 ≥ pivot 置于的位置

[0, i)：≤ pivot

pivot = 3

[0, i): ≤ pivot
(j, n-1]: ≥ pivot

pivot = 3

分析: $\frac{n}{2} \times 1 = 0.5n$ (赋值)

c. 三向分区

| < pivot | = pivot | > pivot |
|---|---|---|

相同元素比较多的情况下.

```c
void q_sort(int arr[], int left, int right) {
    // 边界条件
    if (left >= right) return;
    // 递归公式
    // 1. 分区
    int idx = partition(arr, left, right);

    print_array(arr, 9); // 调试信息
    // 2. 对左边区间排序
    q_sort(arr, left, idx - 1);
    // 3. 对右边区间排序
    q_sort(arr, idx + 1, right);
}
```

#3. 分析

A. 时间复杂度

最好: 每次分区, 基准值都位于中间. $O(n \log n)$

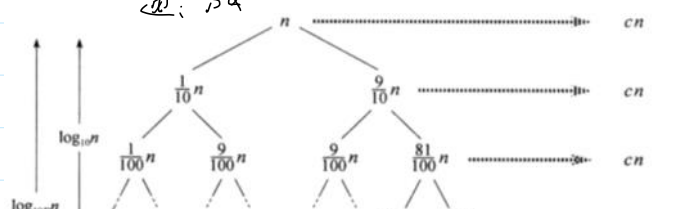
$$T(n) = O(n) + T(\frac{n}{2}) + T(\frac{n}{2})$$
$$= 2T(\frac{n}{2}) + O(n)$$

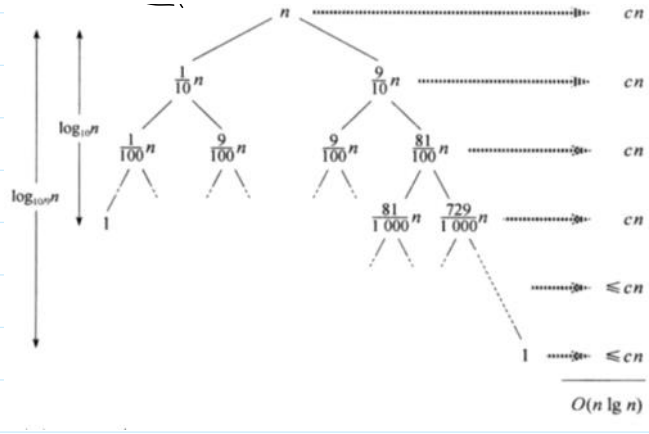
最坏: 每次分区, 基准值都位于两边. $O(n^2)$

$$T(n) = c \cdot n + T(n-1)$$
$$= c \cdot n + c(n-1) + T(n-2) = \cdots$$
$$= c \cdot n + c \cdot (n-1) + \cdots + c \cdot 2 + c = c \cdot \frac{n(n+1)}{2} = O(n^2)$$

平均: 假定每次分区, 都会成大致为 9:1 的区间 $O(n \log n)$

递: 分成






$$T(n) \leq \log_{\frac{10}{9}} n \cdot cn = O(n \log n)$$

B. 空间, $O(\log n)$

C. 稳定: 不稳定

#4. 改进策略

① 选取基准值.

随机选取

选取3到5个元素, 选取其中位数

② 当区间长度小于某个值 ($\leq 32, \leq 64$), 改用插入排序

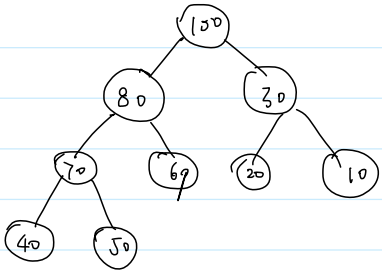
③ 分区算法.

| <pivot | = pivot | > pivot |
|---|---|---|

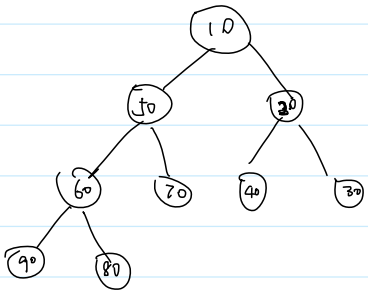
# 堆排序

2024年5月10日 14:23

## #1. 二叉树



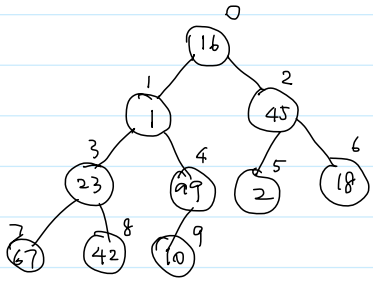

大顶堆




小顶堆

```
[16, 1, 45, 23, 99, 2, 18, 67, 42, 10]
```

| 16 | 1 | 45 | 23 | 99 | 2 | 18 | 67 | 42 | 10 |
|---|---|---|---|---|---|---|---|---|---|




$parent(i) = \frac{i-1}{2}$

$lchild(i) = 2i+1$

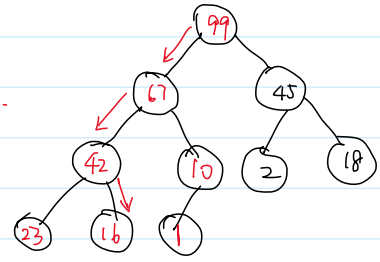
$rchild(i) = 2i+2$

## #2. 算法步骤

a. 构建大顶堆：

| 16 | 1 | 45 | 23 | 99 | 2 | 18 | 67 | 42 | 10 |
|---|---|---|---|---|---|---|---|---|---|

从后往前构建大顶堆（左右子树都是大顶堆）



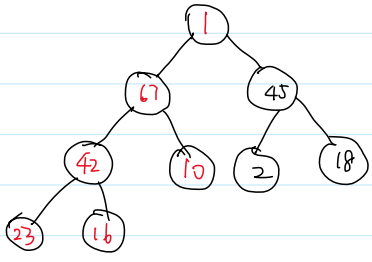

b 无序区 len = n;

c. 交换堆顶元素和无序区的最后一个元素

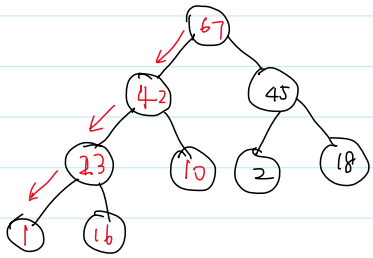
无序区的长度 len--;

重新调整成大顶堆

直到 len == 1




99




$O(\log n)$

#3. 实现

```
void heap_sort(int arr[], int n) {
    // 1. 构建大顶堆
    build_heap(arr, n);  → O(n)
    print_array(arr, n);
    // 2. 初始化无序区的长度
    int len = n;
    // 3. 交换堆顶元素和无序区最后一个元素，直到 len==1
    while (len > 1) {
        SWAP(arr, 0, len - 1);
        len--;
O(logn)← heapify(arr, 0, len);
        print_array(arr, n);
    }
}
```

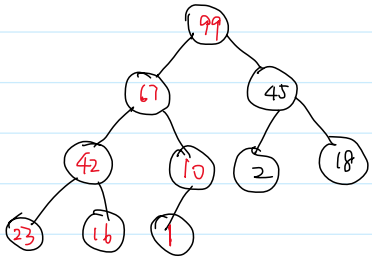
n-1 次 $O(n\log n)$

#4. 分析

时间复杂度：对数据不敏感 $O(n\log n)$

`build_heap(arr, n);` → $O(n)$

n：每个结点最多调整的次数




| | | |
|---|---|---|
| 0 | $2^0$ | h |
| 1 | $2^1$ | h-1 |
| 2 | $2^2$ | h-2 |
| ⋮ | ⋮ | ⋮ |
| h-1 | $2^{h-1}$ | 1 |

$$\sum = \underset{\triangle}{2^0}\cdot h + \underset{\triangle}{2^1}\cdot(h-1) + \underset{\triangle}{2^2}\cdot(h-2) + \cdots + \underset{\triangle}{2^{h-1}}\cdot 1$$

$$2\sum = 2^1\cdot h + 2^2\cdot(h-1) + \cdots + 2^{h-1}\cdot 2 + 2^h\cdot 1$$

$$\sum = -2^0\cdot h + (2^1 + 2^2 + \cdots + 2^{h-1} + 2^h)$$

$$= 2^{h+1} - 2 - h \quad \longleftarrow (h = \log_2 n)$$

$$= 2n - 2 - \log_2 n = O(n)$$

$$T(n) = \underline{O(n)} + (n-1)\cdot O(\log n) = O(n\log n)$$

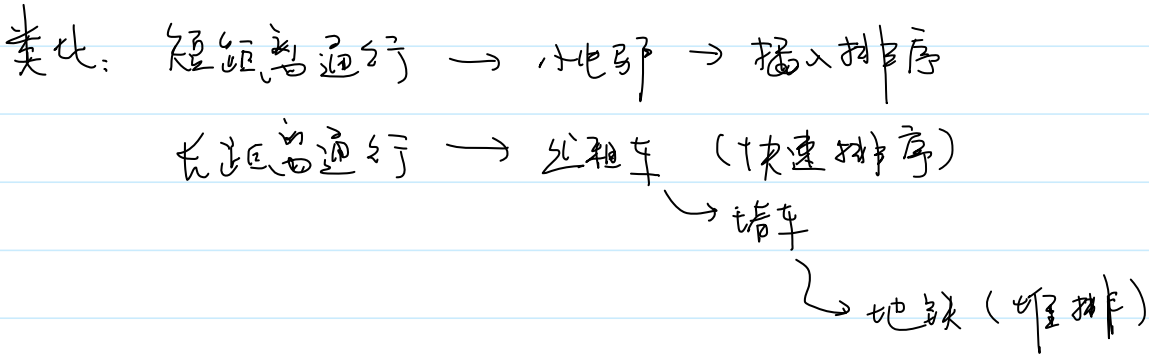
#5. 空间复杂度 $O(1)$

#2. 空间复杂度：O(1)

#3. 稳定性：不稳定

# 如何设计一个通用的排序算法

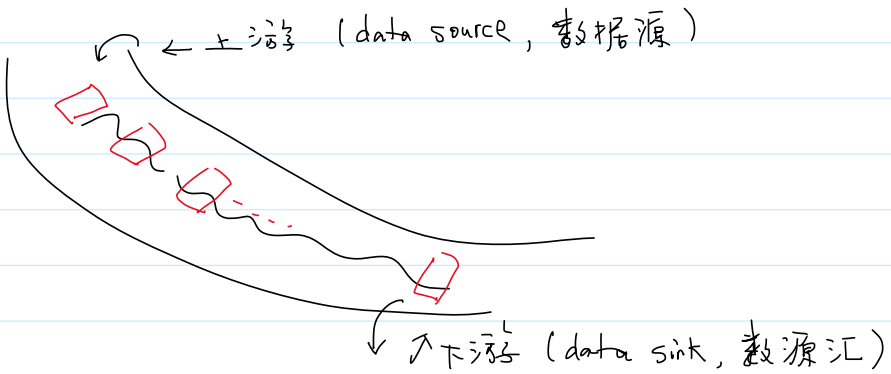
2024年5月10日 16:46



# 文件流

2024年5月10日 17:06

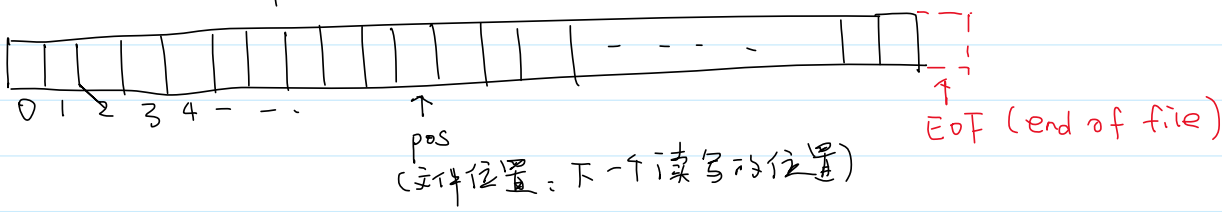
## #1、流模型




优点：①程序员读写文件时，不需要 care 文件的位置。

② 数据源和数据汇是解耦

## #2、程序员视角的文件



# 前置知识

2024年5月10日 17:17

## #1. 缓冲区类型


缓冲区是以**先进先出**的方式管理数据的。缓冲区分为三种类型：

- 满缓冲。当缓冲区空时，从输入流中读取数据；当缓冲区满时，向输出流中写入数据。
- 行缓冲。每次从输入流中读取一行数据；每次向输出流中写入一行数据 (stdin，stdout)。← stdin，stdout
- 无缓冲。顾名思义，就是没有缓冲区 (stderr)。← stderr

输出错误信息.

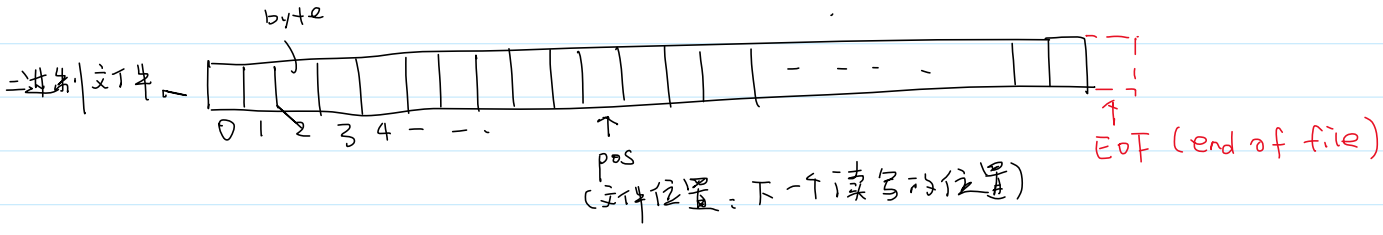
## #2. 标准流

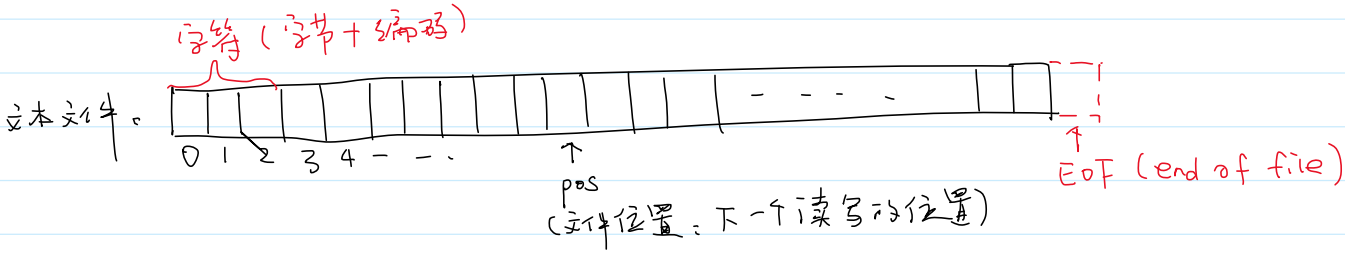
| | | |
|---|---|---|
| stdin | 标准输入 | 键盘 |
| stdout | 标准输出 | 屏幕 |
| stderr | 标准错误 | 屏幕 |

白嫖。不需要手动创建，也不需手动关闭。

## #3. 二进制文件 VS. 文本文件.

↳字节　↳字符




10086 ← 数据

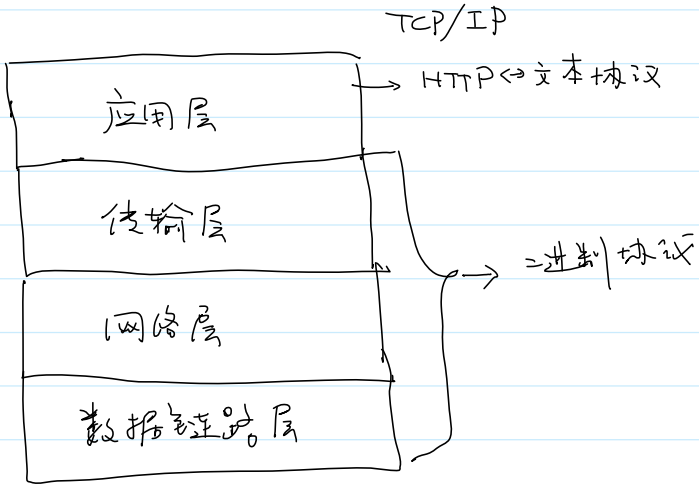
文本文件：'1','0','0','8','6'　5字节　人类可读，数据量大

文本文件：'1','0','0','8','6'　　5字节

二进制文件：　　2字节

人类可读，数据量大

人类不可读，数据量小。



# 文件流的接口 (API)

2024年5月10日 17:36

1. 打开文件流 → fopen

2. 读/写文件

统计、转换、加密、解密 ---

3. 关闭文件流 → fclose()

文件流

```
FILE* fopen(const char* filename, const char* mode);
```

filename：文件的路径。

- 绝对路径："C:/project/test.dat"，"/home/he/test.dat"
- 相对路径："test.dat"，"../test.data"
  - ↳ 当前工作目录

mode：打开模式 { 文件的类型 / 对文件的操作（r, w）

以文本文件方式打开 ⟶

| | | 文件存在 | 文件不存在 |
|---|---|---|---|
| "r" | 打开文件用于读 | → 要求文件必须存在，文件不存在，返回 NULL | |
| "w" | 打开文件用于写（文件不存在则创建） | 清空文件内容； | 创建 |
| "a" | 打开文件用于追加（文件不存在则创建） | | 创建 |

↳ append，追加写入

↳ 多次写入都在文件末尾追加

↳ 写的是文件

| | 文件存在 | 文件不存在 |
|---|---|---|
| r+ | x | NULL |
| w+ | 清空文件内容， | 创建 |
| a+ | x | 创建 |

↳ 追加写入

以二进制文件打开：

rb → (binary)
wb
ab

r+b，rb+
w+b，wb+
a+b，ab+

```
int fclose(FILE* stream);
```

↳ 文件流