# 2025华为软件精英挑战赛

我的参赛代码：

https://github.com/Fengxingzhe666/huawei\_codecraft2025

<https://github.com/DuYangzi/2025_HW_Code_Craft>

在本次竞赛中，官方要求选手设计一个分布式对象存储系统，接受外界的写入、读取和删除对象请求。选手需要根据题目提供的对象标签、对象大小等信息，将具有相似特征的对象尽可能聚合写入，降低硬盘上数据的碎片化程度。在读取时，选手需要合理规划磁头的动作，提高系统读取对象的效率。

我选择的参赛语言是C++，接下来简单介绍一下我的写入、删除和读取的逻辑与具体实现方法。

首先比赛采用的硬盘是一个高度虚拟化的抽象概念，有存储空间、磁头位置、存储单元等概念，因此非常契合于C++的类（class）的定义，为了代码的可读性，我定义了一个表示硬盘状态的类（这里以复赛为例）：

class DISK {

private:

const int Len; //硬盘大小，使用size()接口

int remain; //硬盘当前剩余空闲位置数，使用remaining()接口

vector<int> data; //存储的数据，使用[]运算符重载访问

const int M; //tag值的种类数，该值必须在构造函数中初始化

const int G; //单次操作最多消耗的tooken数，必须初始化

int tooken\_[2]; //表示当前硬盘剩余的tooken

int head\_pos[2] = {0,0}; //表示当前硬盘的磁头位置，注意索引从0开始

int consecutive\_read[2] = {0,0}; //表示磁头连读的次数

unordered\_multimap<int, partition> table; //哈希表，使用tabled()接口

string c\_str[2] = { "#","#" }; //用于保存硬盘动作的字符串

/\*一个1\* Len的数组，表示硬盘上每个存储单元存储的数据。如果第i个存储单元当前存储了数据，则save\_pos[i] = {第i个存储单元存储的obj编号，该obj的block编号，该obj的副本编号}，如果第i个存储单元当前不存在读请求则save\_pos[i] = {0,0,0}\*/

vector<UNIT> save\_pos;

/\*一个1\* Len的数组，表示硬盘上第i个存储单元当前是否存在读请求，如存在则request\_id[i] = vector<int>{(该位置的所有请求编号...)}，元素request\_id[i][j]表示第i个存储单元当前第j个读请求的编号\*/

vector<vector<int>> request\_id;

bool time\_flag[2] = {true,true};

public:

……(接下来是方法的定义，包含写入、删除、读取，磁头的三种动作以及和判题器的互动输出等)

};

## 写入逻辑

根据任务书提供的信息，具有相同tag的object更有概率在同一时间片被读取，这条关键信息指引我们，如果能尽可能的把tag相同的object在写入时就写在相邻的存储单元中，会有利于磁头进行顺序读取。

为此，在类DISK中再抽象出“分区”的概念，为每个tag创立分区，创立一个哈希表存放分区的tag值、起始位置、长度。在写入时，优先选择该tag值的“分区”，这样实现相同tag值的object尽可能存放在相邻的存储单元。

但当时间片逐渐增大，会有一部分分区被均匀的删除，而其他tag值无法找到合适的分区的情况，这种情况下我建立了一套写入机制。

首先引入共享分区的概念，当某个object在写入硬盘时，找不到有剩余空间的对应tag分区，也找不到足够的空间来创建分区，这时候会先尝试切割分区，寻找磁盘中是否有某分区满足后X个存储单元均为空，如果有则将这X个存储单元从原分区剔除，分配给当前tag然后写入（X是一个const常量参数）。如果找不到满足条件的分区，于是就会尝试创建共享分区，即该分区由2种tag值共享，2种tag都可以向该分区写入。

在硬盘的选择上，我优先选择拥有待写入object的tag分区的且分区有足够空间的硬盘进行写入，多个满足则优先选择分区余量大者。其次选择硬盘上拥有超过X个连续空闲存储单元的，多个满足则优先选择连续空闲存储单元较大的，最后选择硬盘整体余量大的。绝对禁止选择余量小于object.size的硬盘。





这里需要注意，由于分区余量是一个需要频繁访问的值，如果每次都花费O(patition.length)去遍历，复杂度会很高，不利于通过5分钟的限制，因此需要维护一个remain变量。但问题在于对于共享分区的remain不容易维护，于是我采用了循环链表的思想，让共享分区的两个partition对象互相指向对方，由此不但可以以固定时间复杂度判定一个分区是否被共享，还可以在一个分区的remain发生变动时使它的共享分区的对象同步变化，大致的思路如下：

struct partition {

int key; //该分区代表的tag值

int begin; //该分区在硬盘的起始位置索引

int length; //该分区的长度

int remain; //该分区的余量

partition\* shared; //共享的分区的对象地址，默认为空

//默认构造函数（供编译器生成哈希表用，实际不调用）

partition() : key(0), begin(0), length(0), remain(0), shared(nullptr) {}

//构造函数

partition(DISK& d, int k, int beg, int len) : key(k), begin(beg), length(len),remain(len), shared(nullptr) {

//构造分区，data值写为-1

for (int i = beg;i < beg + len;i++) {

d.data[i] = -1;

}

}

//改变余量+x，如果有共享分区需要同步改变

void maintain\_remain(int x) {

remain += x;

if (shared != nullptr) {

shared->remain += x;

//起始位置索引和长度也一起同步，防止分割小分区时产生bug

shared->length = length;

shared->begin = begin;

}

}

……（其他函数定义）

};

其中partition作为一个DISK的私有成员结构体，注意我维护的那个DISK成员变量哈希表结构unordered\_multimap<int, partition> table;

它就表示当前该硬盘上的分区信息，key表示分区代表的tag，而value就是包含分区的信息的对象，同时通过指针shared可以迅速同步具有相同起始位置和长度但tag不同的共享分区的各项参数，也可以通过shared是否为空判断该分区是否与其他tag共享了。

## 删除逻辑

删除逻辑比较简单，仅需要更新DISK中的data变量，并检测该位置是否含有未完成的读请求即可（得益于我一直维护的变量request\_id），向判题器上报被取消的读请求编号，并维护全局变量将这些请求踢出队列即可。注意删除操作需要维护DISK的remain变量，同时需要遍历哈希表找到该位置所在的分区，将分区余量-1

## 读取逻辑

读取逻辑是整个比赛的核心，任务书中明确磁头有3种动作：r（读取当前存储单元），p（跳过当前存储单元），j（跳跃至某存储单元），并且规定了每个时间片消耗的最大tooken值。在这种情况下，明显按照读请求的到来时间进行处理远远不如按照object在硬盘中存储的位置进行处理（即顺序读取更优），且尽可能的保证磁头连续读取（这样消耗的tooken会递减，在同一时间片就能读更多）

注意，并不是当前位置不存在读请求，磁头p就一定优于r，虽然在当前位置p只消耗1tooken，而r至少消耗16tooken或更多，但p会破坏磁头的连读状态，在接下来的读取中还需要消耗更多tooken，有些情况反而不划算。

于是在我的代码中设计了一个两个函数互相调用的递归模式，在每次磁头运动到不含读请求的位置时，调用函数判定是否需要连续r到下一个有读请求的位置

/\*

\* @brief 判断连读空位置到下一个有读请求的位置是否划算

\* @param head\_pos 磁头位置索引（从0开始）

\* @param consecutive 起始时的连读次数

\* @param tok 起始时的剩余tooken

\* @return 布尔值，如果连读划算就返回true；如果pass划算就返回false

\*/

bool do\_we\_read\_null(int head\_pos, int consecutive, int tok){……}

在函数do\_we\_read\_null()中调用另一个函数read\_successufl\_nums()计算能成功读取的block个数，以成功读取的block个数作为判定是否连续r的标准。read\_successufl\_nums()函数中磁头遇到下一个没有读请求的位置同样调用do\_we\_read\_null()判断是否需要连续r到下一个有读请求的位置。这样两个函数构成一个递归结构。

/\*

\* @brief 判断当前磁头状态向后读取的最多个数

\* @param head\_pos 磁头位置索引（从0开始）

\* @param consecutive 起始时的连读次数

\* @param tok 起始时的剩余tooken

\* @return int值，表示磁头向后理论能读取的有效最多个数

\*/

int read\_successufl\_nums(int head\_pos, int consecutive, int tok){……}

递归终止的条件是tooken被耗尽，在比赛数据集给出的tooken限制下，这样的算法的复杂度完全可以被接受，这样每次进行判定，得出的结果一定是当前时间片的使该磁头读取block尽可能多的最优解。

我设计的N个硬盘的磁头的读逻辑是一个贪心算法，假设每个磁头与最近的读请求位置的距离为d（以下成为d值），所有磁头各有一个时间标志位time\_flag，在时间片开始时time\_flag置为true，在因为tooken不足无法进行读取时time\_flag置为false，当某硬盘上没有待读取的object时，该硬盘上所有磁头的time\_flag置为false，我会选择time\_flag==true且d值最小的那个磁头执行顺序读取，当磁头碰到不含读请求的位置时，再次选择time\_flag==true且d值最小的磁头进行操作。当所有磁头的time\_flag均为false时，当前时间片的读逻辑结束，向判题器输出结果。

注意在初赛条件下，计算d值时，我会过滤掉超过105时间片未被处理的读请求，因为这些读请求即使读取也没有得分了。然而在复赛条件下，超过105时间片未被处理的读请求必须上报繁忙并倒扣分数，因此计算d值就不用额外过滤了。



如果只是这样的算法，还不足以支撑我走到复赛，于是我和队友在研究了数据集的特征以后发现，每个时间片都有一种或几种tag会被疯狂请求，而且这些热tag具有周期性，即一定时间内的热tag是保持不变的。

通过excel表统计预处理数据画出的各tag值在每1800时间片被请求的次数如下图：

横坐标：表示第i个1800时间片，例如1表示时间片1~1800，2表示时间片1801~3600，以此类推。纵坐标：表示在1800时间片内某tag被读请求的次数。

有了这种特征，于是我在时间片的读请求开始前，先进性一次check\_hot\_tag()逻辑，这个逻辑会检查前n\_hot个最热的tag分区，如果当前时间片读逻辑开始时，没有任何一个磁头位于某热tag分区内部，则找到一个ex值（ex是当前磁头满tooken向后顺序读取得到的最大得分）最小的磁头jump到该热tag分区的起点。如果有多个热tag分区可以jump，则选择评估jump过去后下一个时间片得分最多的那个热tag分区的起点。此外由于jump操作会导致当前时间片之后该磁头不能再有其他动作，该函数还会计算jump过去2个时间片的得分和当前位置顺序读取3个时间片的得分，如果得分有增益才会执行，确保一定是正优化。评估得分的函数是read\_most()，经过我的优化可以满足官方对时间复杂度的限制。

对于复赛每1800时间片有一次“垃圾回收”，允许选手在每个硬盘进行K次交换操作，交换操作会交换两个存储单元存储的block。对于这部分，受数学不等式：

的影响，我定义了一个*σ*值表示一个分区的整齐度，*σ*值的取值范围是(0,1]，值越大说明该分区的整齐度越高，去碎片化效果越佳。它的定义如下：

例如两个分区的存储情况如下：

data1={0,0,1,0,0,1,1,0,0,1,1,1,0,0,1}

data2={1,1,1,0,1,1,1,0,0,0,0,0,0,0,0}

则data1和data2对应的整齐度：

显然，直观感受也是data2所代表的分区更整齐。

在“垃圾回收”时，我会优先选择值较小的分区执行交换，交换逻辑采用双指针法，先计算两种tag值存储索引的平均值，平均值大的放在右侧，平均值小的放在左侧。左指针碰到一个要放在右侧的tag，右指针碰到一个要放在左侧的tag，就交换他们2个位置并继续直到两个指针碰到或者消耗次数达到K

很遗憾的是，由于实际K值给的太小，“垃圾回收”对于得分的提升实在是微不足道。但是有还是比没有强一点吧。

