基于python3.8的文件系统实现：

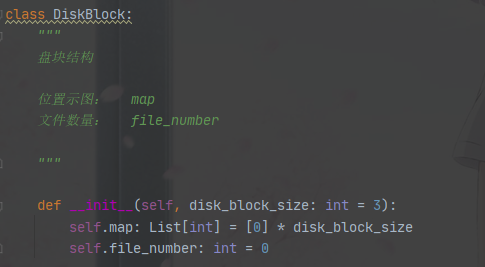
①定义位示图，文件目录，数据块的数据结构：

1.文件目录



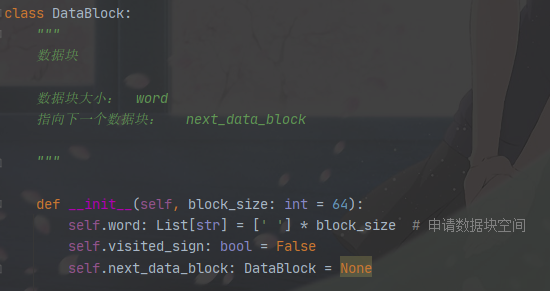
文件目录除了基本的文件目录信息，还定义了设置文件目录，删除文件目录这两个函数，是为了精简在创建文件，复制文件和删除文件过程的重复代码。

2.位示图



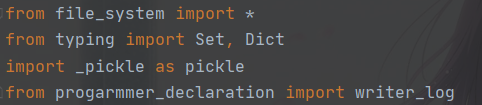
位示图的数据结构定义有位示图和文件数量。位示图为数组结构。

3.数据块



数据块定义有文本内容，访问标志（检测这块数据块是否有数据写入），指向下一个数据块的指针，数据块为一个链式结构。

②文件管理系统

1.引入第三方库  


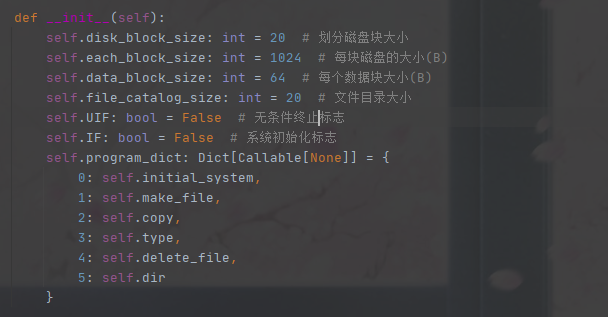
1.本次实验需要的数据结构全部定义在file\_system中

2.typing 扩充python的type-hint

3.\_pickle 变量序列化工具

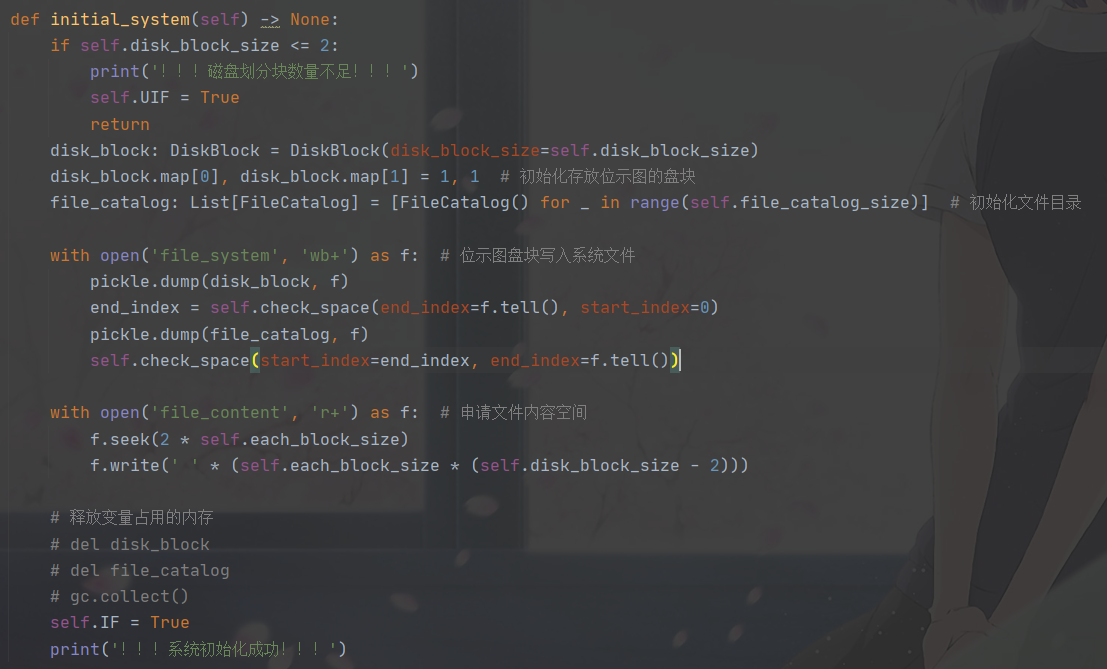
4.programmer\_declaration 打印作者信息的修饰器

2.构造函数



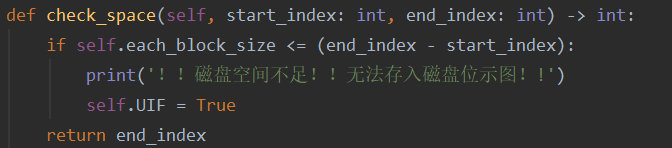
在构造函数中声明了划分磁盘块的大小，每块分区的大小，每个数据块的大小，文件目录大小。无条件终止标志是为了防止每块分区的大小不足以存放位示图，文件目录而启用的中断标志（搭配下文的检查函数使用），系统初始化标志是为了防止用户没有初始化就运行程序导致程序崩溃而启用的。

3.初始化函数



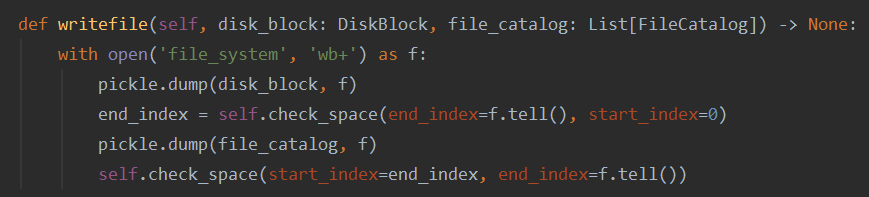
初始化函式首先检查了磁盘的分区是否能放入位示图和文件目录，若不足则置无条件标志符置True，阻止程序继续执行。接下来将位示图实例化，并将位示图的数组前两位置1表示在磁盘前两个分区分别存入位示图和文件目录。接下来将二者写入file\_system文件，同时打开file\_content文件根据磁盘区域大小为文本内容申请空间。最后将初始化标志置True，完成程序初始化。

1. 检查空间大小



通过输入文件指针的初始位置和终止位置判断磁盘分区大小是否足以存储位示图和文件目录。

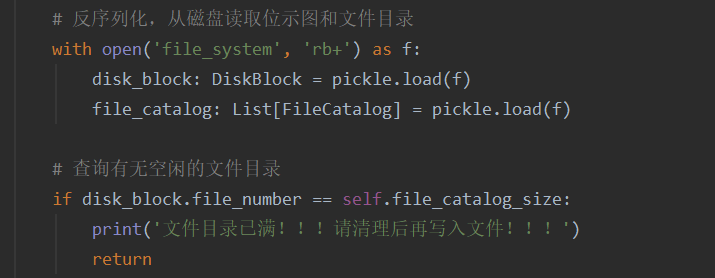
1. 写入磁盘



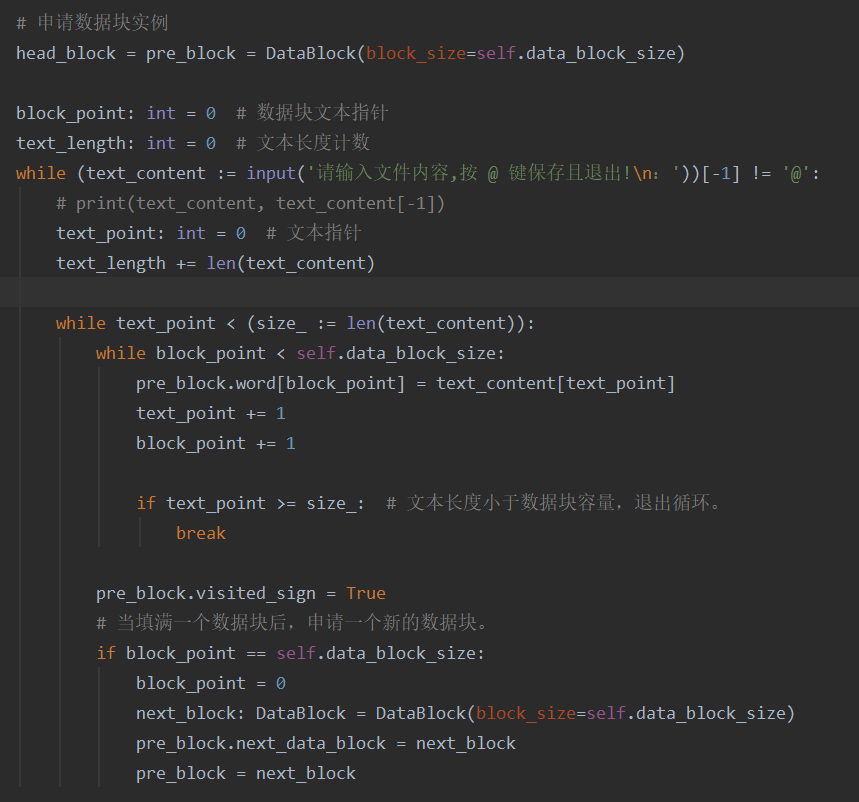
实现把位示图和文件目录写入磁盘

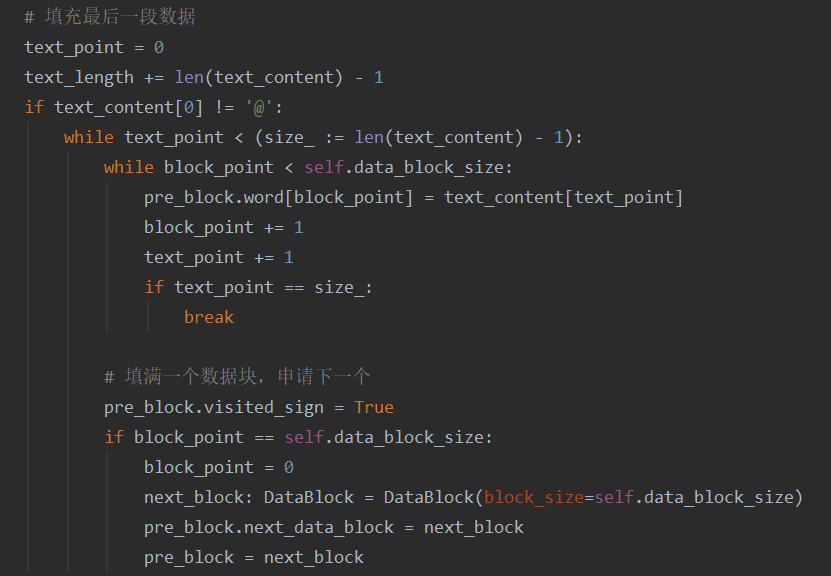
1. 创建新文件

（1）从磁盘中载入位示图和文件目录，并判断文件目录是否已经装满：

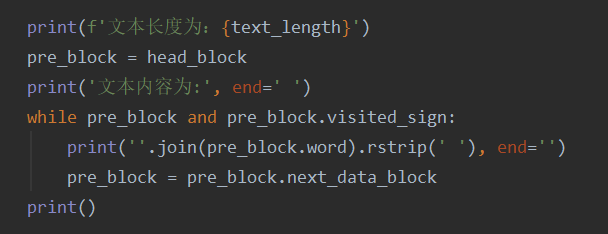


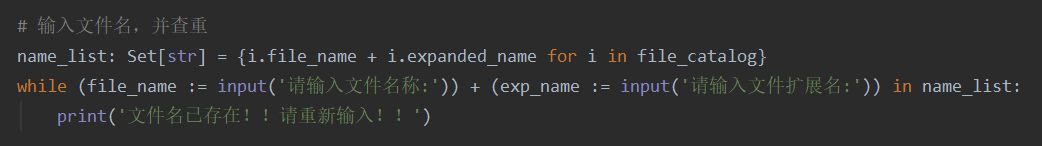
（2）申请数据块，把用户输入的内容写入数据块中：

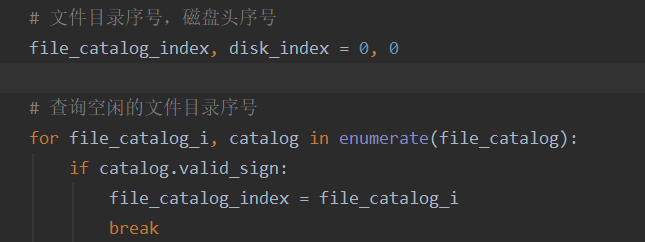




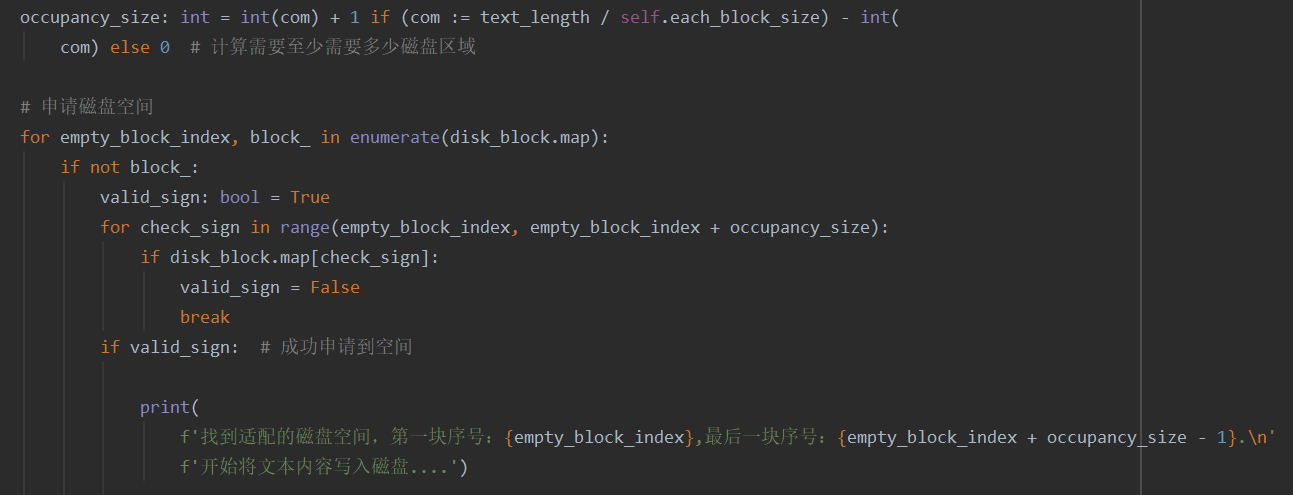
1. 输入完成，统计并展示文本信息，包括文本长度，文本内容：



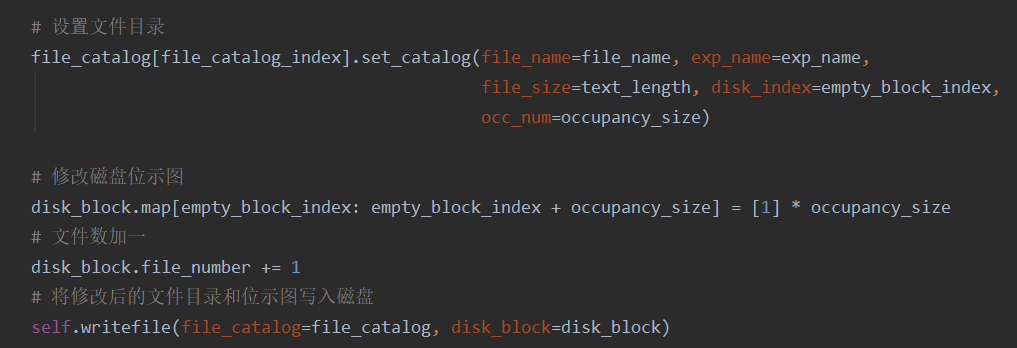
1. 输入并检查文件完整名称是否重复：  
    
2. 查询空的文件目录序号（由于之前已经判断文件目录是否已经装满了，所以这里不做判断）：



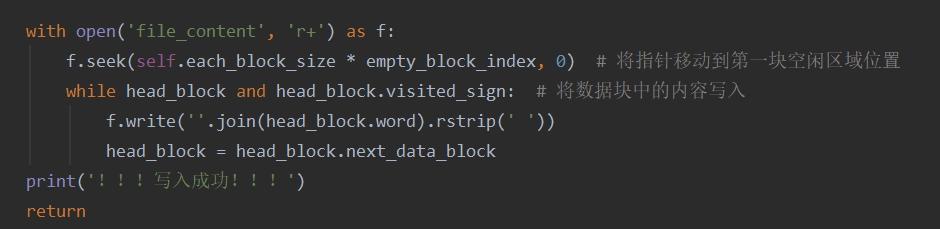
1. 计算文本需要的磁盘块数，并申请空间（若申请失败，则跳转到（9））：



1. 修改文件目录和位示图并写入磁盘：



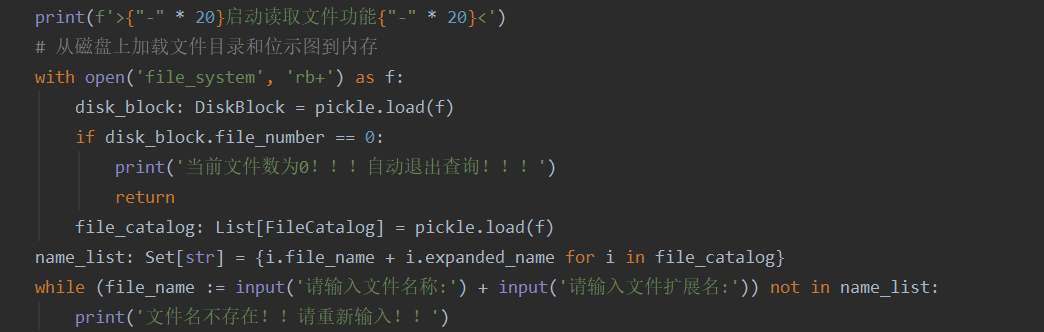
1. 将文本内容写入磁盘，结束程序）：



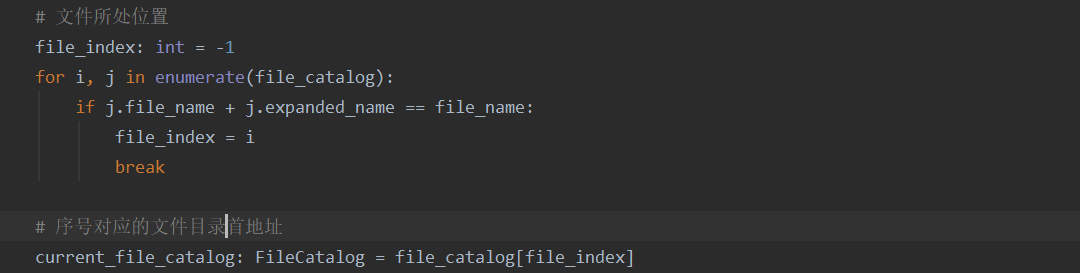
1. 没有申请到空间，结束程序：



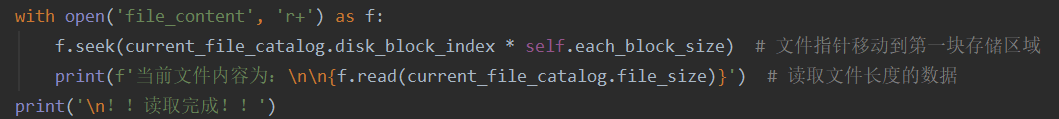
1. 查看文件内容：
2. 从磁盘载入位示图和文件目录，并判断文件目录中是否有文件，并判断生成一个散列表判断用户查询文件是否存在：



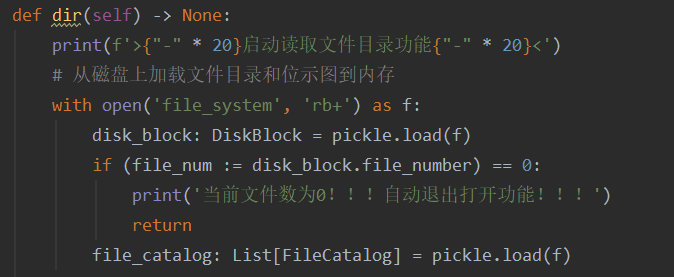
1. 查询用户查询文件在文件目录中的序号，并取出该序号对应的文件目录中存储的信息：



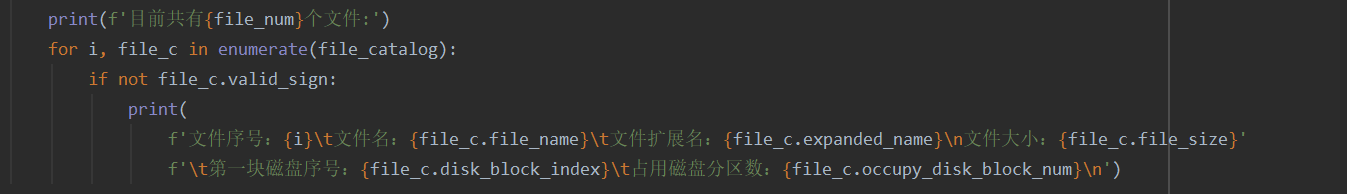
1. 根据文件目录存储的文件对应分区的第一块序号和文本长度，从磁盘上读取完整的文本内容，程序结束。



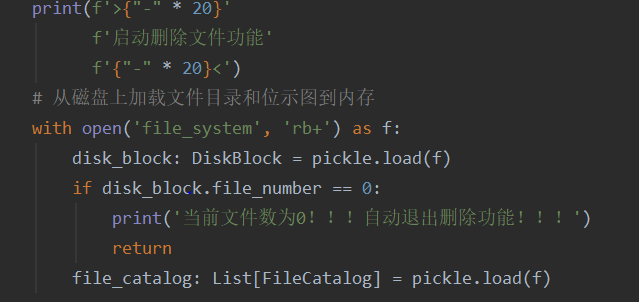
1. 查看完整的文件目录
2. 从磁盘上载入位示图和文件目录，并判断文件目录是否为空。



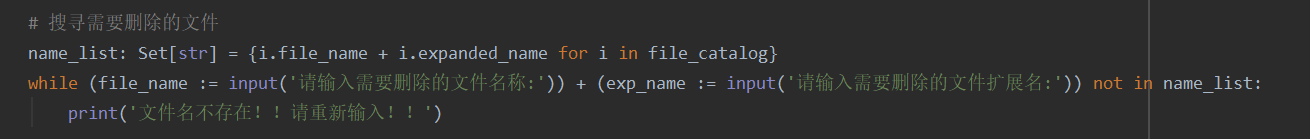
（2）遍历并打印文件目录全部信息



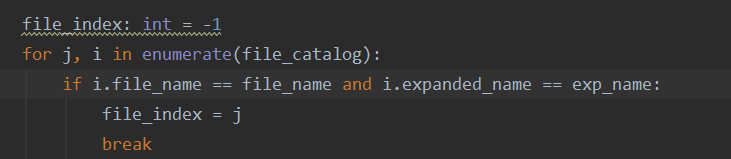
1. 删除文件
2. 载入位示图和文件目录，并判断文件数是否为0：



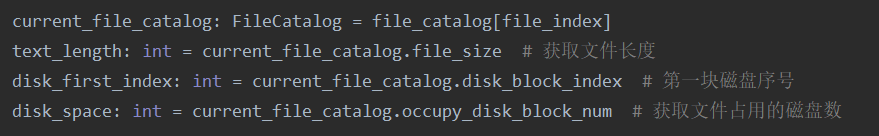
1. 建立一个临时的散列表判断用户输入的文件是否存在：



1. 定位需要删除的文件在文件目录中的位置



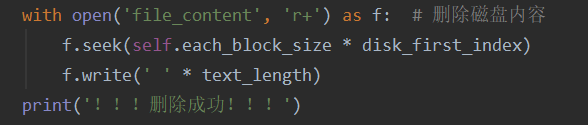
1. 获取文件存储的磁盘块序号，文本长度。



1. 修改位示图，调用文件目录的删除函数，并将修改后的内容写入磁盘：



1. 将磁盘上存储的文本内容覆盖删除：

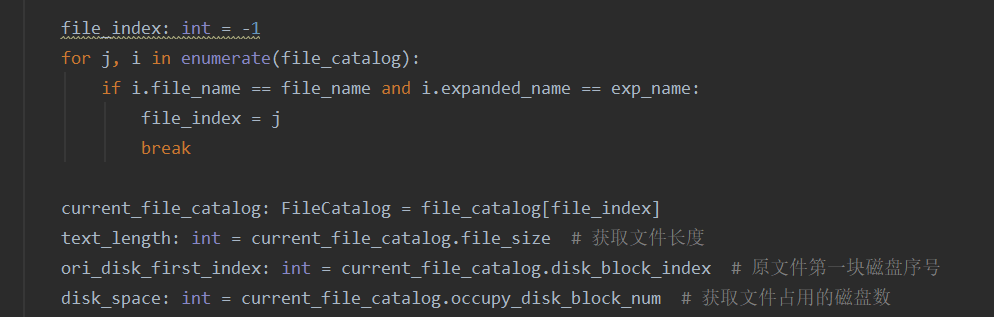


10.复制文件

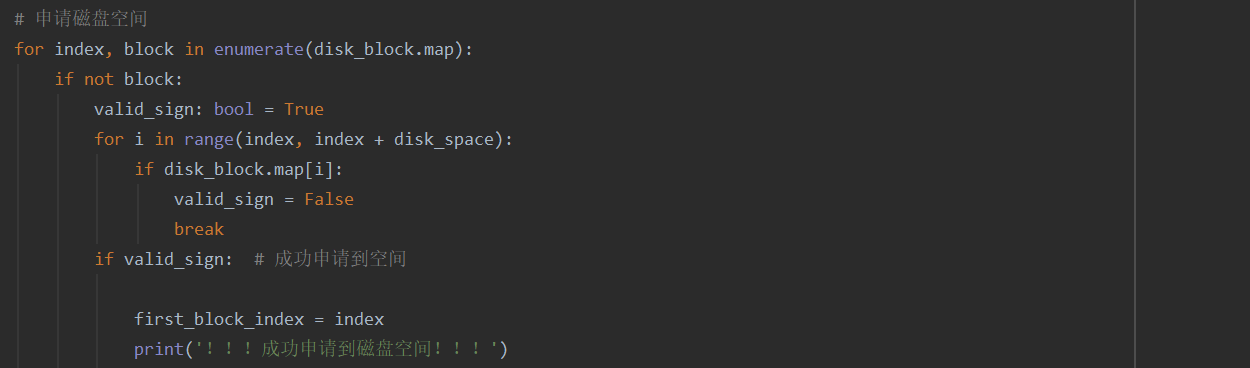
（1）从磁盘上读取文件目录和位示图，并判断文件数是否为空或文件是否已经装满



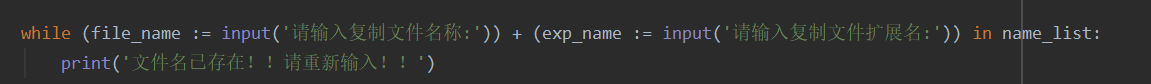
1. 判断用户输入文件是否存在，若存在则找出其在文件目录的索引号，获取被复制文件的在磁盘中的序号，文本长度，占用的磁盘数。



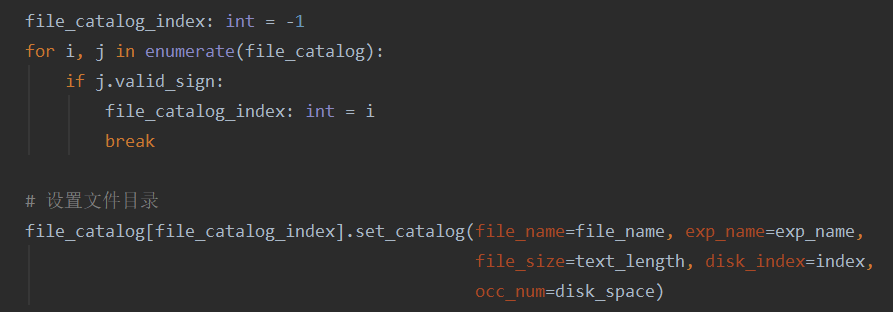
1. 申请磁盘空间（若失败就直接退出程序）：



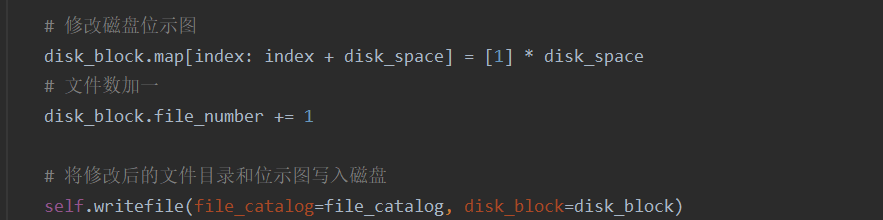
1. 输入文件名并判断是否已经存在：



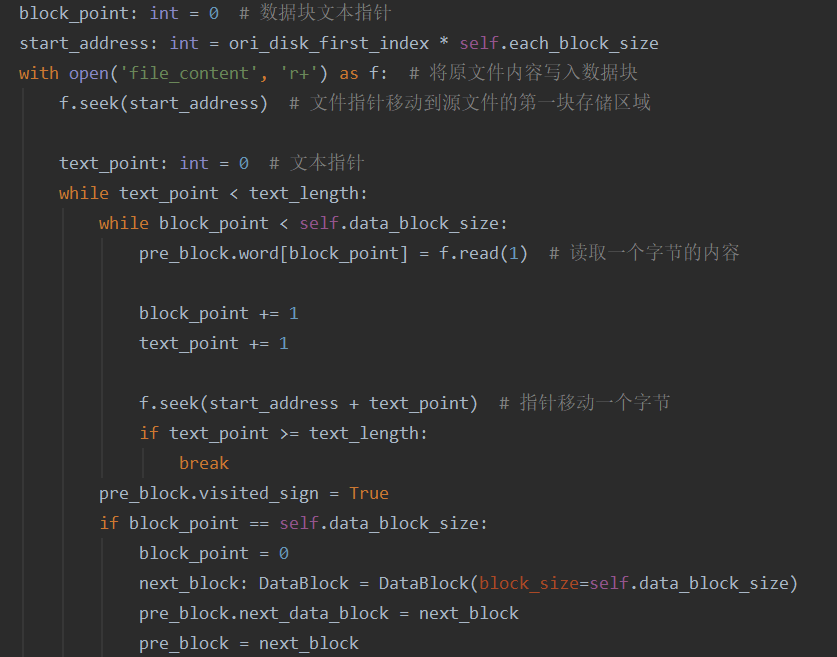
1. 申请文件目录并修改文件目录



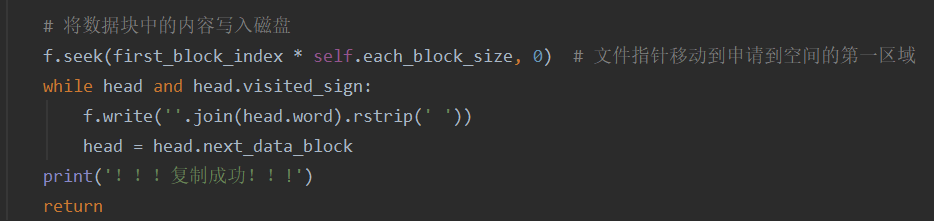
1. 修改位示图，并将修改后的文件目录和位示图存入磁盘



（7）读取被复制的文本内容，并写入数据块：



1. 将数据块的内容写入磁盘



③优化文件系统

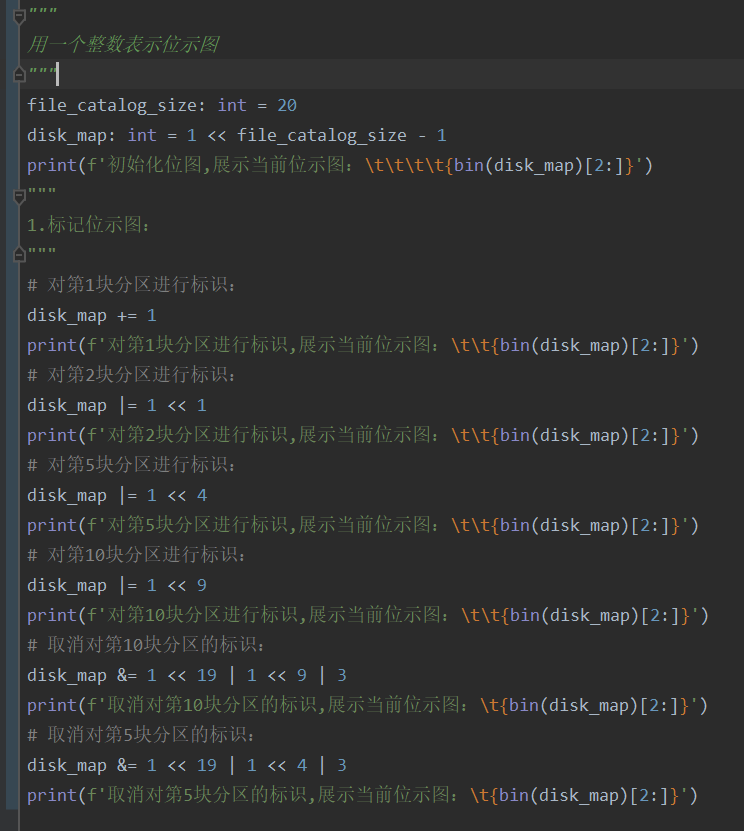
到此为止只是对原文件系统程序的复刻，下面进行一些优化：

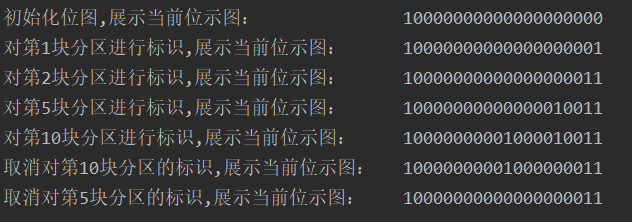
1. 对位示图的优化：

我们发现位示图中只是简单的对空闲的分区的进行标识。若分区过多，位示图数组过长，可能照成查找效率低和占用过多内存的问题。

①通过位运算进行存储表示。

1. 标记位示图：



我们发现，基于位运算的位示图虽然降低了存储开销，但是仍然没有对检索效率进行提升。

②下面我们采用一种改良后的双向链式结构：

链式结构中每一个块的数据结构如下图所示：

**Block-Data；**

**Next；**

**Prefer；**

**Cross-Next；**

**Block-Data：存储信息，在位示图中表示了分区的序号和该分区是否被使用。**

**Next：顺序指向下一个块。**

**Prefer：顺序指向上一个块。**

**Cross-Next：当顺序结构被破坏时，该指针负责保持链的连通。**

**初始化链表：**

**Idx：4**

**Idx：3**

**Idx：2**

**Idx：1**

**初始状态下块与块之间通过，Next和Prefer指针相连。**

**Idx：n**

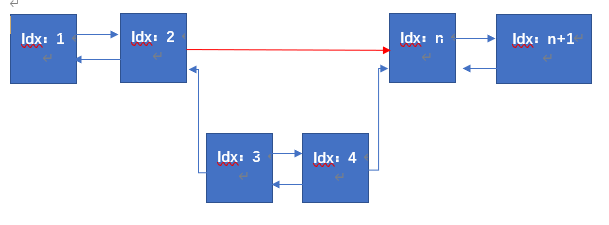
**移除3-4个块：**

**在移除第k到k+n个块时，进行一下操作：**

**①将k-1块的Next指针置None，Cross-Next指针指向k+n+1**

**②将第k+n+1个块的prefer指针置None**

**如此操作之后，就可以将已经被占用的块从顺序结构中移除，在进行遍历时就可以快速跳过被占用的块。同时通过下一个块是否通过Next指针连接还是Cross-Next指针连接判断是否时连续的块。**



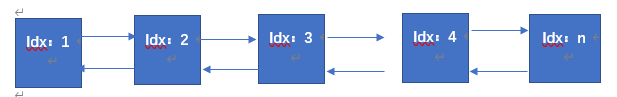
回收3-4号块：

回收3-4号块进行以下操作：

①指针移动到第2块，置第2块的Next指针为第3块的地址，Cross-Next置为None

②指针移动到第5块，置第5块的prefer指针为第4块的地址。

这样就完成了块的回收。



我们发现这样通过这种改良的链式结构的确加快了查找空闲区的速率，但是在进行块的回收时，时间复杂度又变成了O(n)如此看来，这样的链式结构多适用于查找频率大于回收频率的系统。但是我们实际可以再引入一种字典结构存储每一个块的地址，这样在回收块时，通过直接查找字典来代替指针移动，使得时间复杂度降位O（1）。

综上：

①基于位运算的数据结构适用于计算机资源紧张的系统，对查找效率没有要求的系统。

②基于链式的数据结构适用于对查找效率要求高的系统。