****

# 内存管理项目

姓 名：李翠琪

学 号：1751022

所在院系：软件学院

学科专业：软件工程

指导教师：王冬青

二〇二〇年五月

1. **项目目的**

* 动态分区分配方式
  + 数据结构、分配算法
  + 加深对动态分区存储管理方式及其实现过程的理解。
* 请求分区分配方式
  + 页面、页表、地址转换
  + 页面置换过程
  + 加深对请求调页系统的原理和实现过程的理解

1. **项目需求**
2. **动态分区分配方式的模拟**

* 要求:

假设初始态下，可用内存空间为640K，并有下列请求序列，请分别用首次适应算法和最佳适应算法进行内存块的分配和回收，并显示出每次分配和回收后的空闲分区链的情况来。

* 请求序列:

作业1申请130K

作业2申请60K

作业3申请100k

作业2释放60K

作业4申请200K

作业3释放100K

作业1释放130K

作业5申请140K

作业6申请60K

作业7申请50K

作业6释放60K

1. **请求调页存储管理方式模拟**

* 内容

假设每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。

* 模拟过程
* 在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。
* 所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。
* 置换算法可以选用FIFO或者LRU算法
* 作业中指令访问次序可以按照下面原则形成:50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分
* 具体实施方法(参考):
* 在0－319条指令之间，随机选取一个起始执行指令，如序号为m
* 顺序执行下一条指令，即序号为m+1的指令
* 通过随机数，跳转到前地址部分0－m-1中的某个指令处，其序号为m1
* 顺序执行下一条指令，即序号为m1+1的指令
* 通过随机数，跳转到后地址部分m1+2~319中的某条指令处，其序号为m2
* 顺序执行下一条指令，即m2+1处的指令。
* 重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到执行完320条指令。

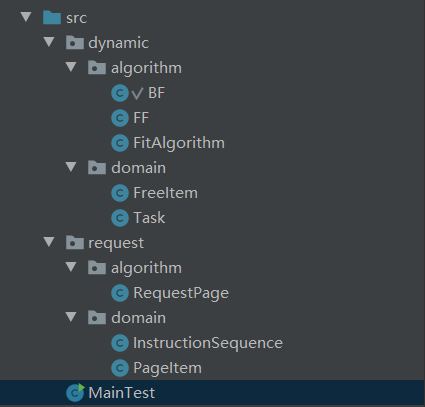
1. **项目设计及算法**

**3.1项目整体结构**

本项目是采用java语言开发的，通过命令行互动的内存管理项目。主要分为2个包(dynamic,request)和一个主类MainTest。通过主类的main函数选择相应的算法进行模拟。

其中的dynamic包主要是完成对动态分区分配的模拟，他有两个子包，一个是存储实体的domain包，里面包括了要执行的作业（Task类），以及空闲分区链中的每一项(FreeItem)。而另一个子包algorithm中，有一个类FitAlgorithm,抽取了FF和BF的整个流程里面重复的代码片段。并且他可以根据选择的参数，来决定调用首次适应算法（BF类）还是最佳适应算法（FF类）中的方法。

而request包主要是完成请求调页方式的模拟，也有两个子包domain以及algorithm。在domain包中有两个类，PageItem类主要负责存储页表中的项，InstructionSequence则负责提供一组特定规则的随机序列，模拟指令。而algorithm包中，只有一个类requestPage，负责完成请求调页方式模拟的逻辑以及展示页面。

****

**3.2首次适应算法**

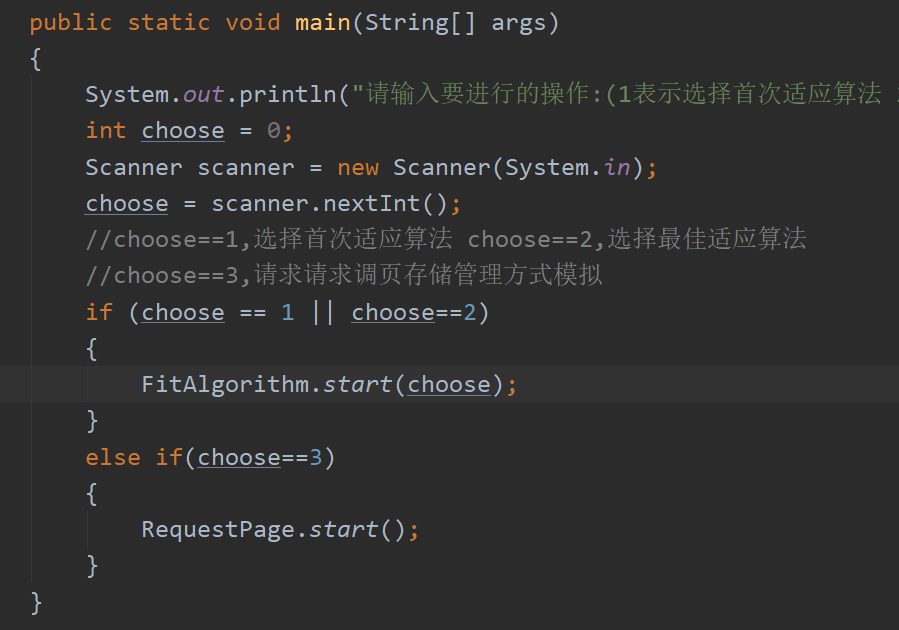
* + 实现：根据空闲分区链中地址从低到高的顺序，从空闲分区链的第一个地址找起，把最先能够满足要求的空闲区分配给作业。如果满足要求的空闲分区大小正好等于该任务所需内存大小，则需要在空闲分区链中删除该空闲分区，否则就改变该空闲分区的相应参数（始址，长度）。
  + 优点:查找时间短，算法开销小，回收分区后不需要对空闲分区队列重新排序，综合性能最好。
  + 缺点:低址部分不断被划分，会留下许多难以利用的，很小的空闲分区，称为碎片。而每次查找又都是从低址部分开始的，这无疑又会增加查找可用空闲分区时的开销。
  1. **最佳适应算法**
  + 实现：通常是采用空闲分区以容量递增的顺序排列，每次选择容量最小的能满足需求的空闲分区。但我的具体实现是从按地址递增的空闲分区链中,先筛选出符合条件的空闲分区,再从中找到最小的一个空闲分区，如果该空闲分区大小正好等于该任务所需内存大小，则需要在空闲分区链中删除该空闲分区，否则就改变该空闲分区的相应参数（始址，长度）。因为这样可以不用对空闲分区按照容量递增排序，可以复用FF中的分区回收算法。
  + 优点:优先使用更小的分区，保留更多的大分区，所以会有更多的大分区保留下来，更能满足大进程的需求。
  + 缺点:会留下许多难以利用的，很小的空闲分区，称为碎片碎片。每次都要找到最小的可用的空闲分区，算法的开销大。
  1. **LRU置换算法**
  + 实现：在每一个页表项的访问字段中记录了上次访问时间（用一个不断递增的变量实现）。当内存块中已经满了，需要置换时， 就查看内存块里面的四个页面，根据访问字段选取出最近最久未使用的页面，并且模拟调出内存，再把所需要的页面调入内存（实际上就是在对应的内存块数组中放入该页面的索引），并且把该页面访问时间改为最新的。
  + 优点:性能很好，缺页率较低，是局部性原理的合理近似，
  + 缺点:算法开销较大，需要记录页面使用时间的先后关系，硬件开销太大。

1. **项目实现**

**4.1 MainTest类实现**

通过命令行输入的参数，选择对应的算法进行模拟。其中，1表示选择首次适应算法，2表示选择最佳适应算法，3表示进行请求调页存储管理方式模拟。

其中，选择1和2将使用FitAlgorithm类并且传入该参数来选择首次适应算法，或者最佳适应算法。选择3则使用RequestPage类进行请求调页方式模拟。

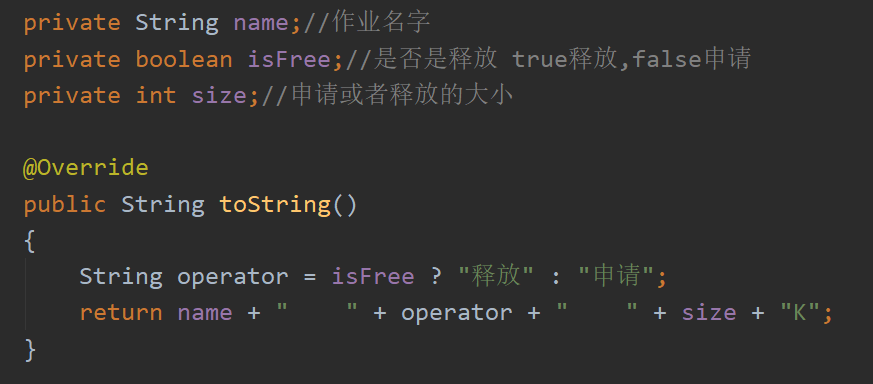


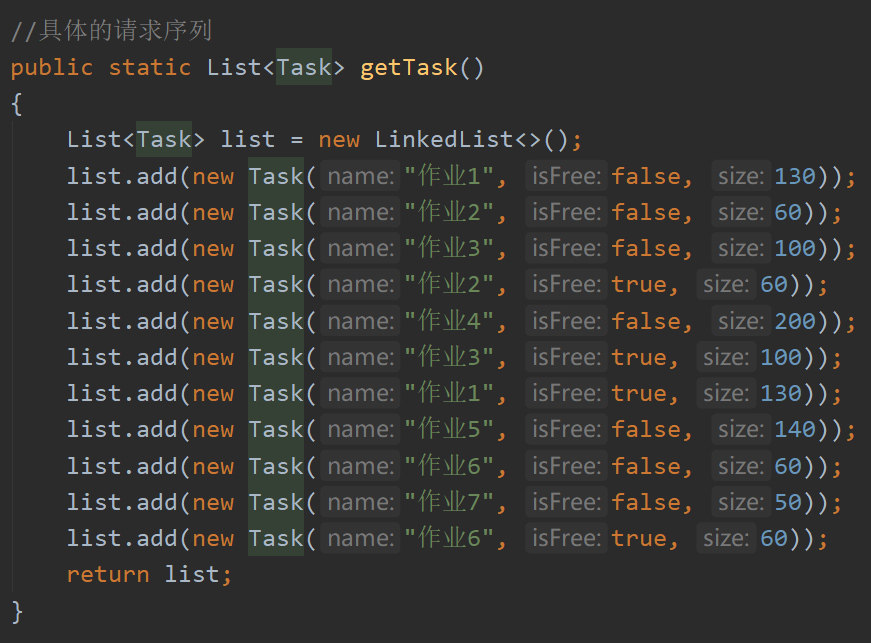
**4.2 dynamic包下类实现**

**4.2.1 Task类实现**

实体类，字段包含了作业名称，进行的操作（释放/申请），释放/申请的内存大小。并且重写了toString方法，使得展示更符合阅读规范。

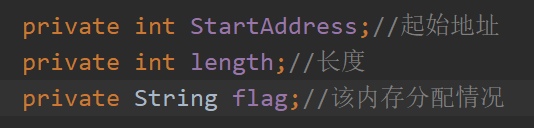
该类有一个静态方法，getTask，将存储了一系列的任务列表封装成链表并返回。





**4.2.2 FreeItem类实现**

该类也是实体类，有三个字段，分别是空闲分区的起始地址，长度，以及分配情况（未分配，或者分配给了哪一个作业）。同样重写了toString()方法，使得格式更加整齐。

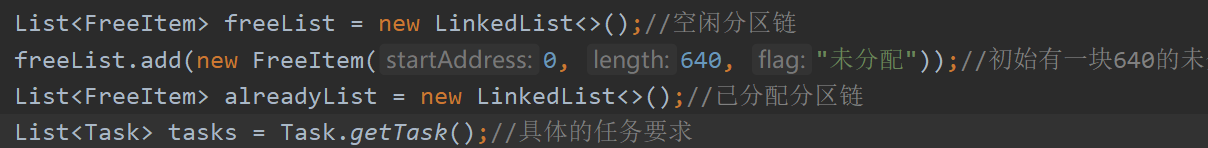




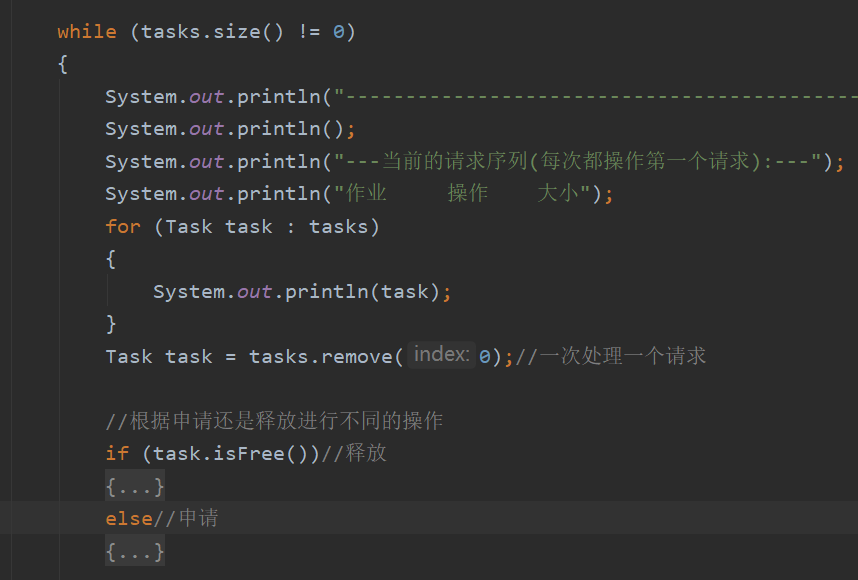
**4.2.3 FitAlgorithm类实现**

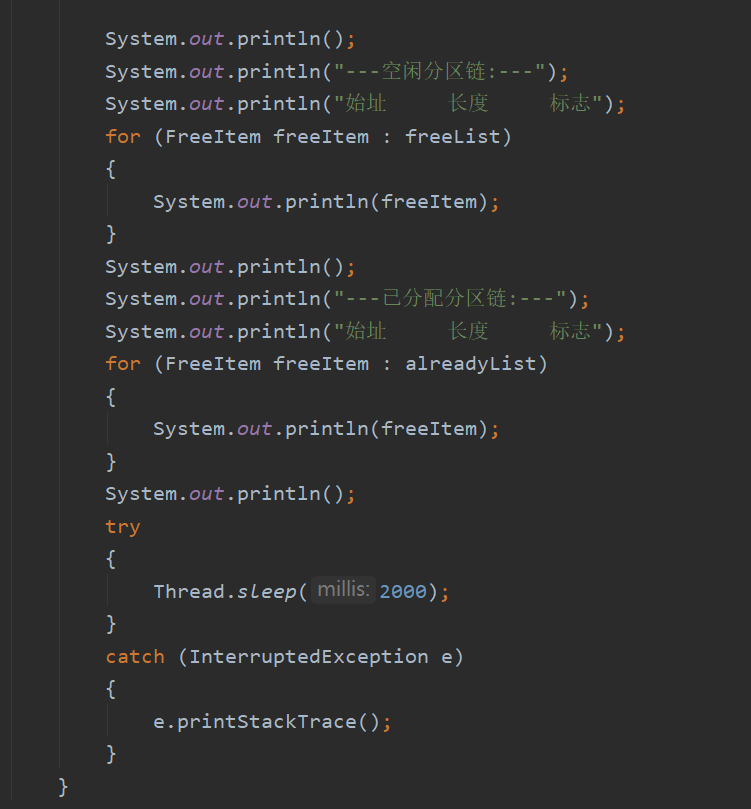
该类负责实现动态分区分配的具体逻辑，并且通过传入参数选择FF或者BF算法进行内存分配。

首先，初始化空闲分区链，已分配分区链，并且从Task类中拿到具体的任务要求。



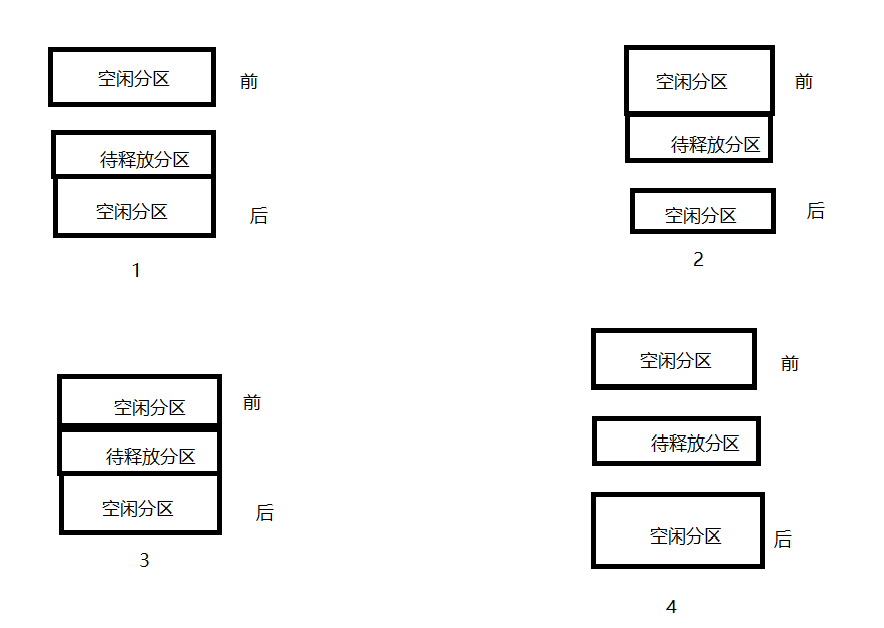
然后每次从任务列表中取出第一条任务，根据该任务需要申请/释放内存进行相应操作（下面会讲），然后打印出空闲分区链及已分配分区链此时的情况。每间隔两秒循环一次，直到任务列表为空。

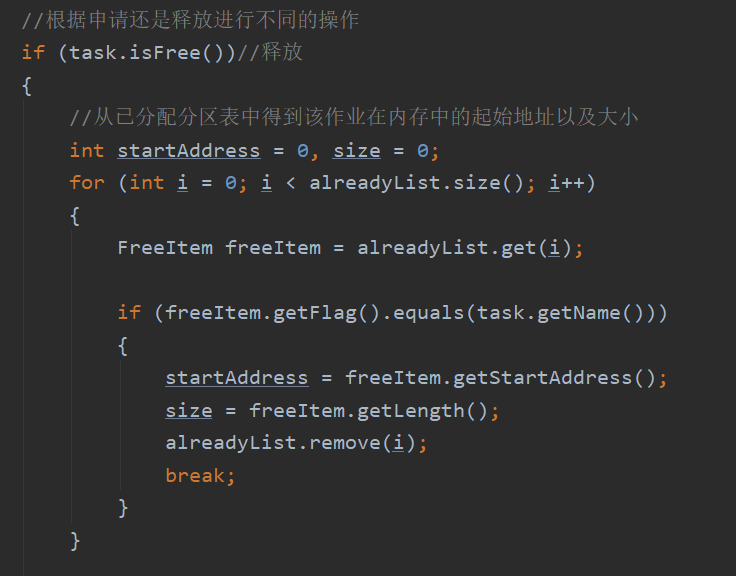




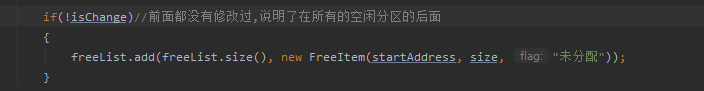
任务如果进行的是释放操作，则首先从已分配分区表中得到该作业在内存中的起始地址以及大小，随后在空闲分区表中释放该区域内存。但是这样一共有四种情况： 1.释放该作业内存后和后面的空闲分区相连,2.释放该作业内存后和前面的空闲分区相连,3. 释放该作业内存后和前面以及后面的空闲分区连成一大块,4释放的该作业内存和前后空闲分区都不相连。(需要考虑是否存在前后空闲分区的情况)。

四种情况示意图如下：

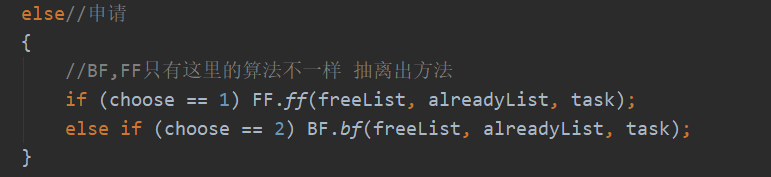






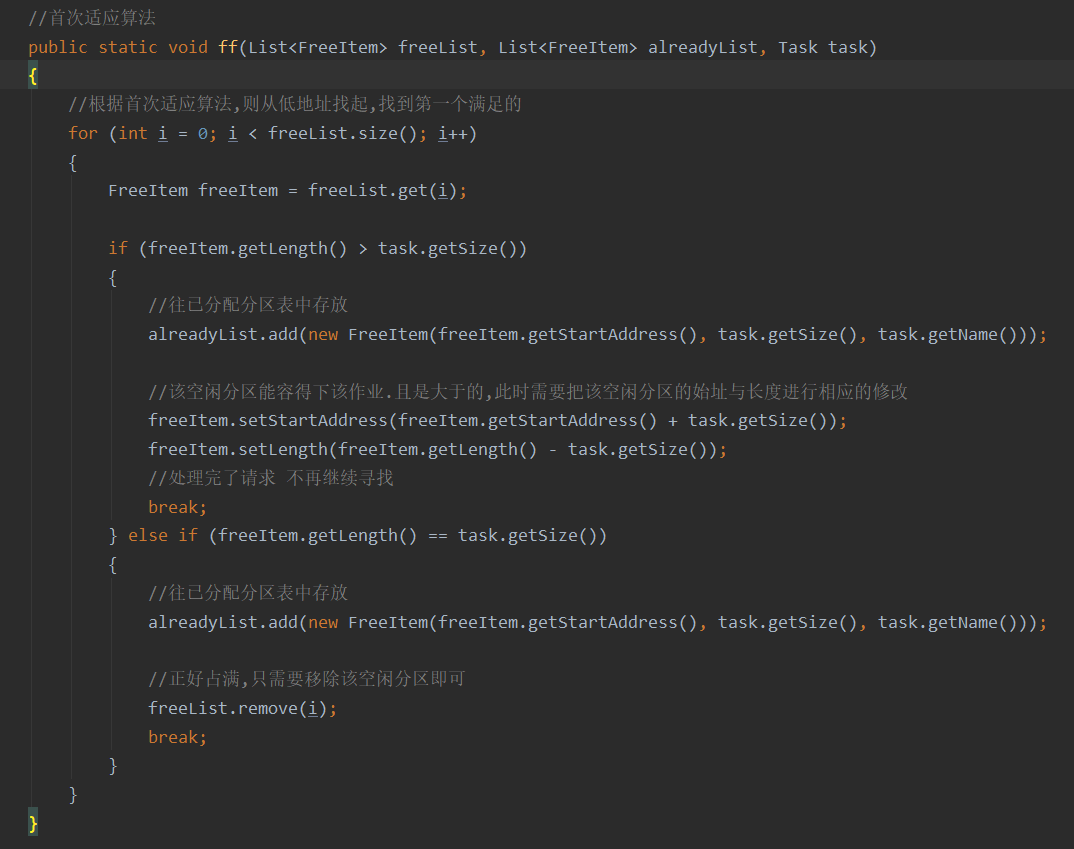


任务如果是申请操作，则根据之前在main函数选择的数字来采取对应的算法（FF或者BF）。



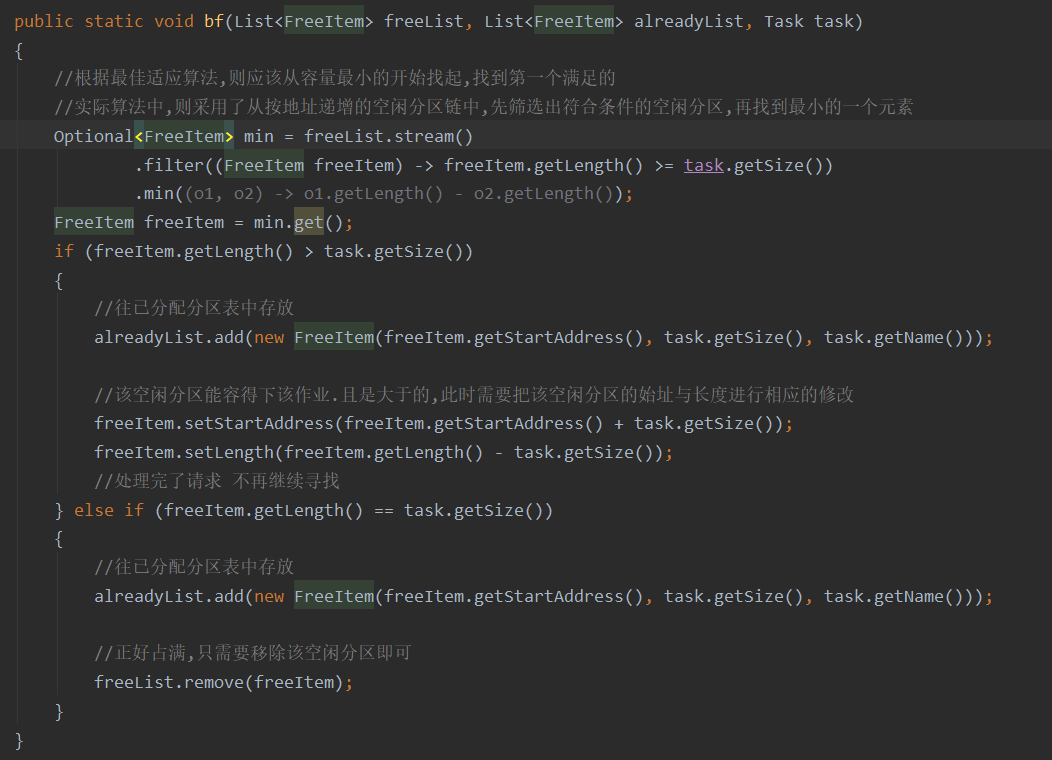
**4.2.4 FF类实现**

采取首次适应算法进行内存分配，主要实现在静态函数ff中。根据空闲分区链中地址从低到高的顺序，从空闲分区链的第一个地址找起，把最先能够满足要求的空闲区分配给作业。如果满足要求的空闲分区大小正好等于该任务所需内存大小，则需要在空闲分区链中删除该空闲分区，否则就改变该空闲分区的相应参数（始址，长度）。

****

**4.2.5 BF类实现**

采取最佳适应算法进行内存分配，主要实现在静态函数bf中。从按地址递增的空闲分区链中,先筛选出符合条件的空闲分区,再从中找到最小的一个空闲分区，如果该空闲分区大小正好等于该任务所需内存大小，则需要在空闲分区链中删除该空闲分区，否则就改变该空闲分区的相应参数（始址，长度）。



**4.3 request包下类实现**

**4.3.1** **InstructionSequence类实现**

主要作用是根据ppt的方式,得到指令序列.并且是通过伪随机数,每次都得到同样的一组随机数,便于测试与复现。

大致先随机选取作业中的一条指令，并且按顺序得到下一条指令，随后不断循环，重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到得到320条指令（最后两条指令由于数组大小原因，选择了随机某条指令以及按顺序的下一条指令）。



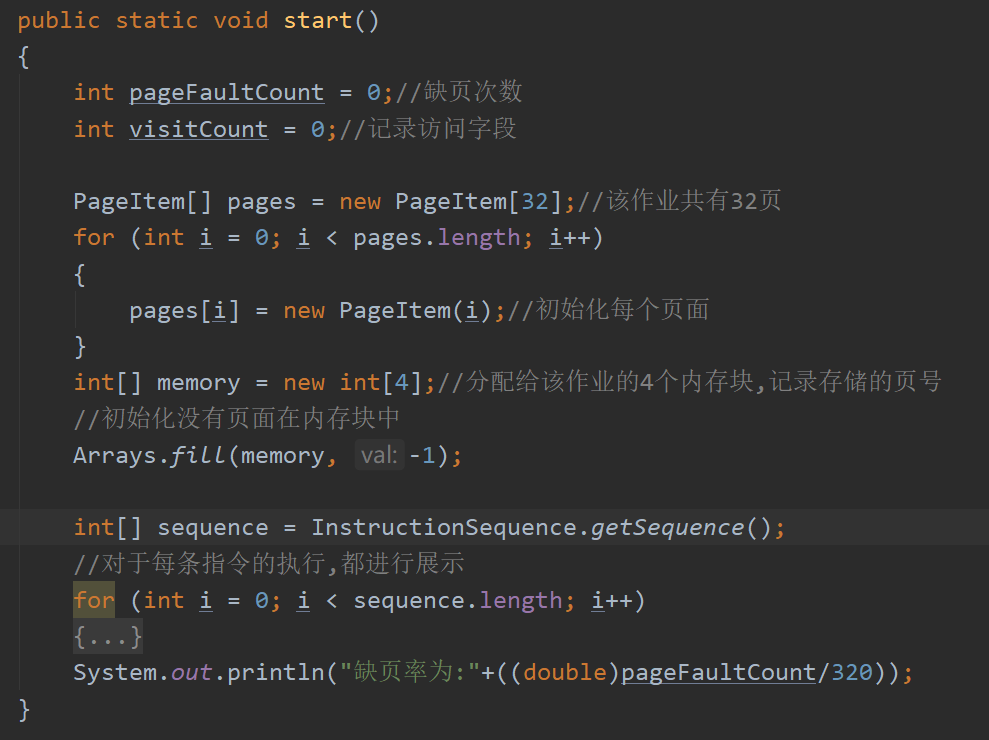
**4.3.2PageItem类实现**

该类是实体类，代表了页表中的页表项，有四个字段：页号，内存块号（-1代表不在内存中），状态（true代表在内存中，false代表不在内存中），访问字段（记录上次访问时间，以实现LRU算法）。初始化中，所有页面都不在内存中，最久访问字段也设置为-1。

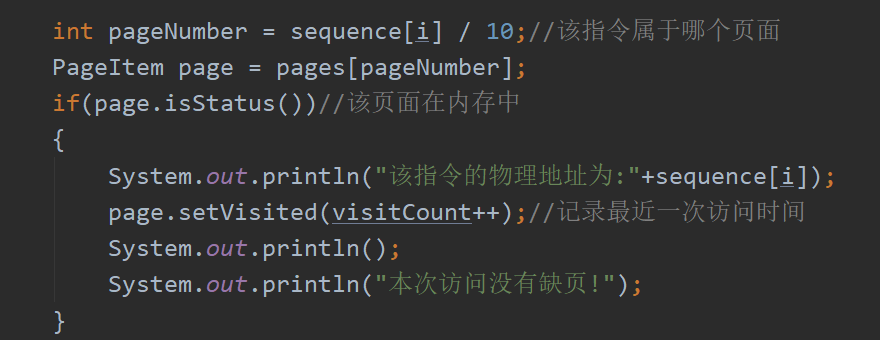


**4.3.3RequestPage类实现**

该类实现请求调页的主要功能。首先定义了缺页次数字段，已经最近访问时间visitCount(初始化为0,每次访问页面后赋值到页面的访问字段并且+1)，并且初始化了该作业的32个页面。并且初始化分配给该作业的4个内存块（开始内存块中无页面）。随后通过之前的InstructionSequence类得到了指令序列。随后就对指令序列的每条指令进行执行（随后会说），最后打印缺页率。



在每条指令的执行过程中，首先先找到该指令所属的页号。如果该页面在内存块中，则打印出指令的物理地址，并且将访问时间设为最近的（visitCount++）。



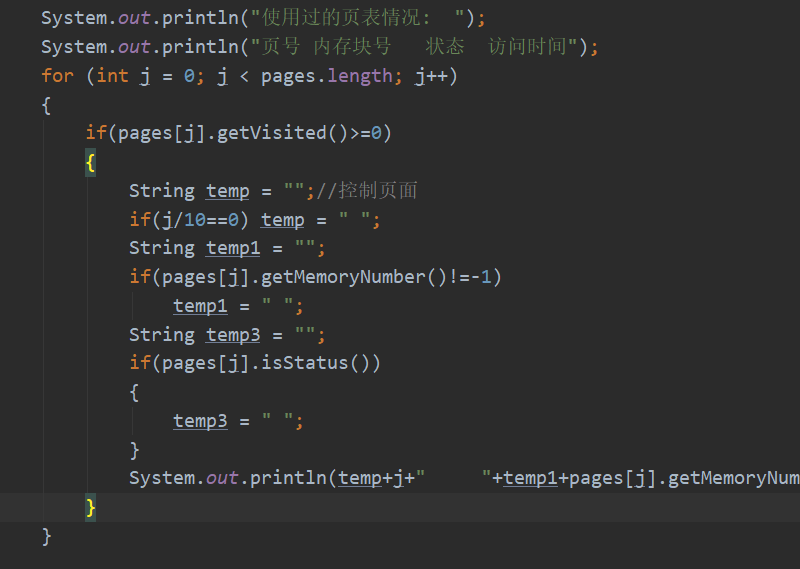
如果缺页了，但是内存块中还有空余位置，则将页面直接放入空闲的内存块中，并且让缺页次数+1，并且设置最近访问时间，而且不需要页面置换。



如果缺页了，并且4个内存块中都装满了作业，则使用LRU算法进行页面置换。首先查看内存块里面的四个页面，根据访问字段选取出最近最久未使用的页面，并且模拟调出内存，再把所需要的页面调入内存（实际上就是在对应的内存块数组中放入该页面的索引），并且把该页面访问时间改为最新的。



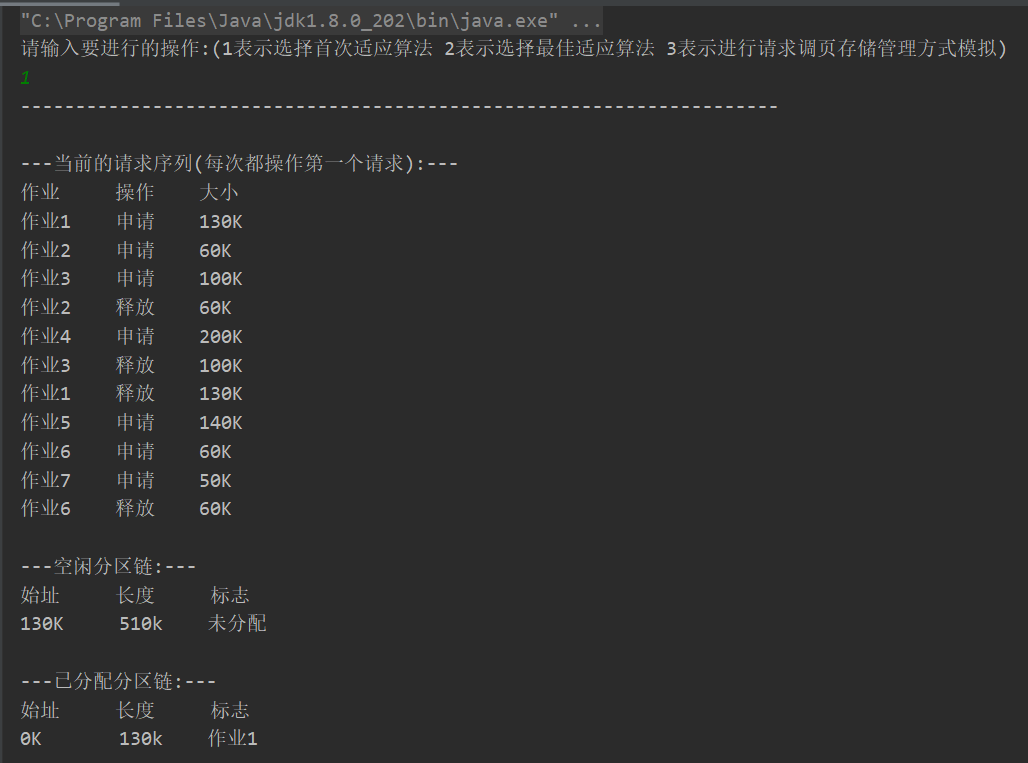
最后，展示页表目前的情况，但是由于作业的总页数太大，所以只展示了曾经放入内存块中的页面（通过判断访问时间是否>=0实现）。并且还展示了4个内存块里面存储的页面号。



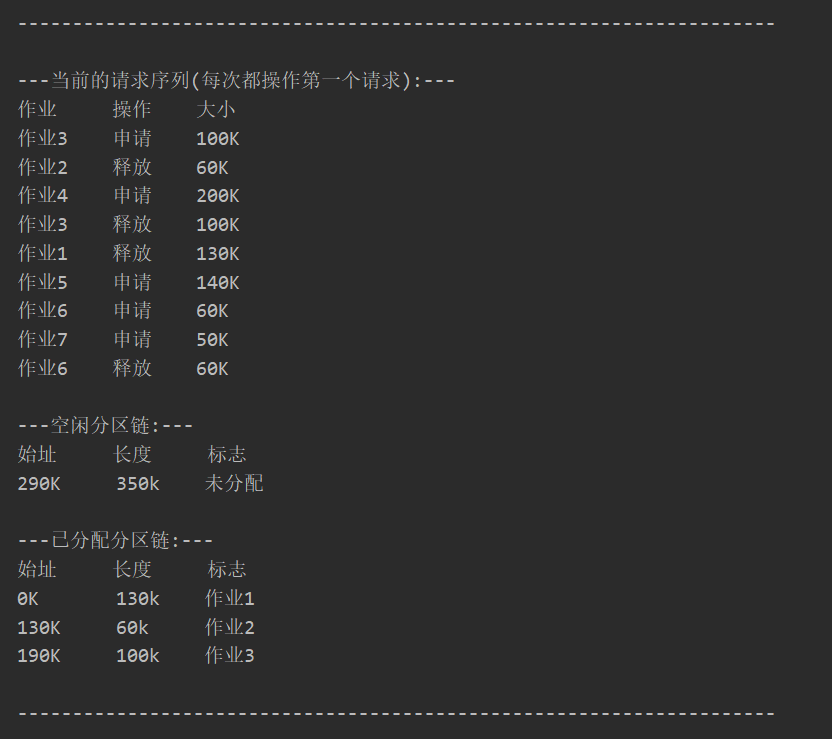
1. **项目界面展示**

**5.1动态分区分配-FF算法界面展示**

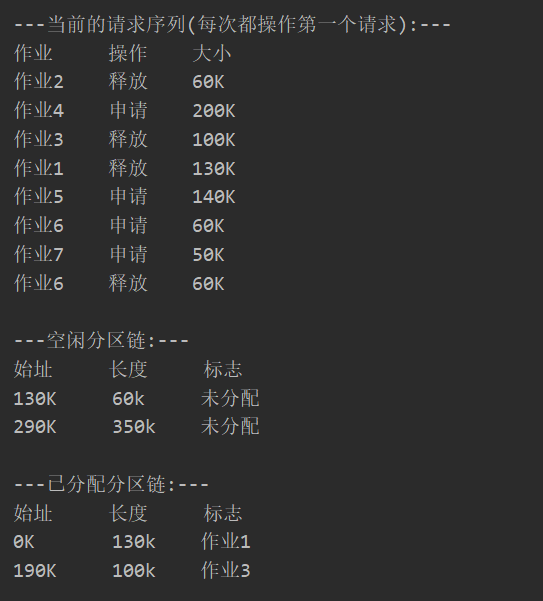
处理作业1后的情况：



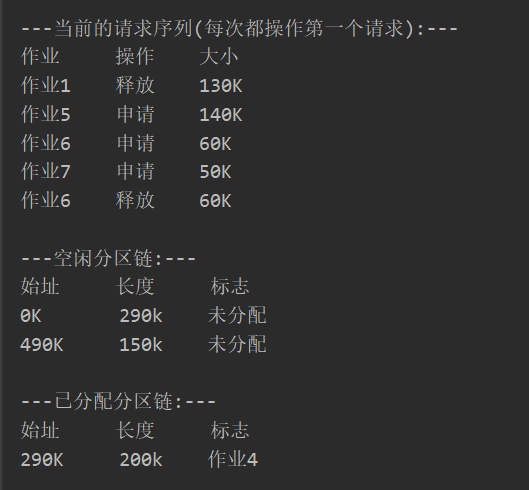
处理作业2，作业3情况也类似，只截处理完作业3的图:



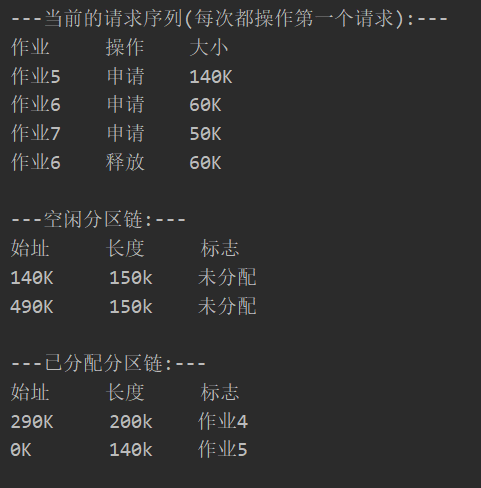
释放作业2内存：

****

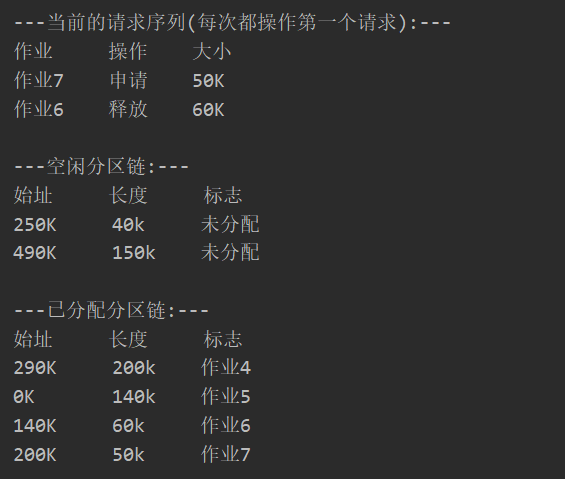
作业4申请，作业3，作业1释放后：



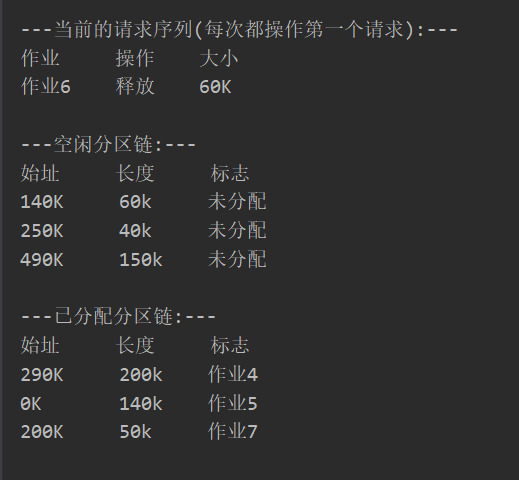
作业5申请，**这里是FF和BF算法在该作业序列里不一样的地方，**FF算法在这里按地址顺序寻找到前面的低地址空闲分区就分配了，但是BF会找到最后的那块容量最小的内存分区进行分配：



作业6，作业7申请后：

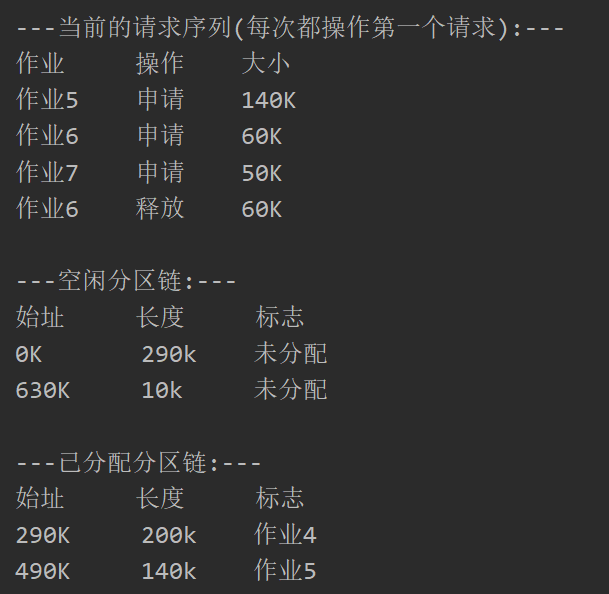


最后一个序列，作业6释放内存：

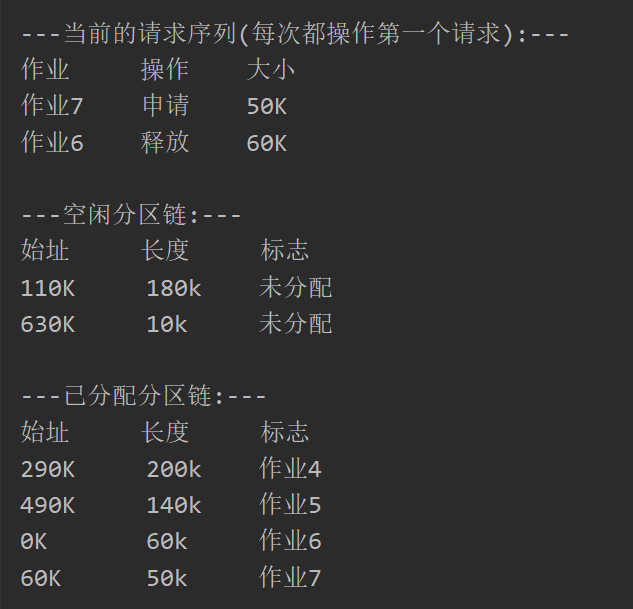


**5.2动态分区分配-BF算法界面展示**

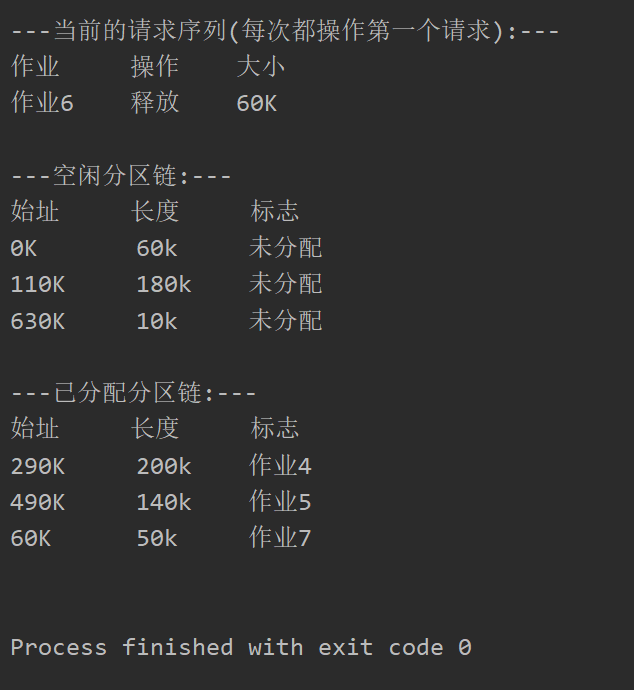
前面的都和FF算法一样，只是在作业5申请时有了变化，这里找到了容量较小的空闲分区进行分配：



作业6，作业7申请：

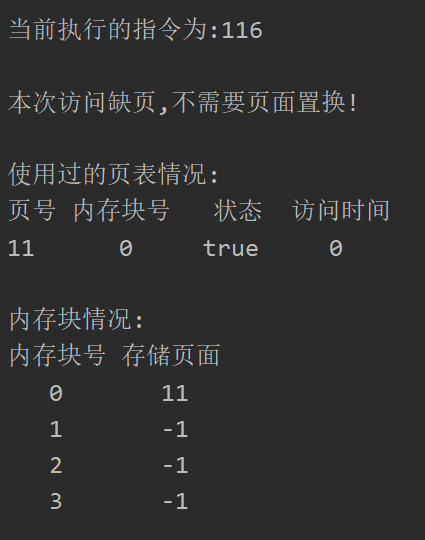


最后一条，作业6释放：

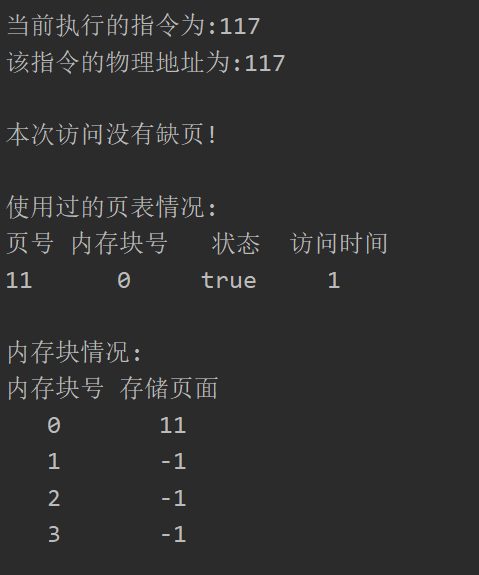


**5.3请求调页-LRU算法界面展示**

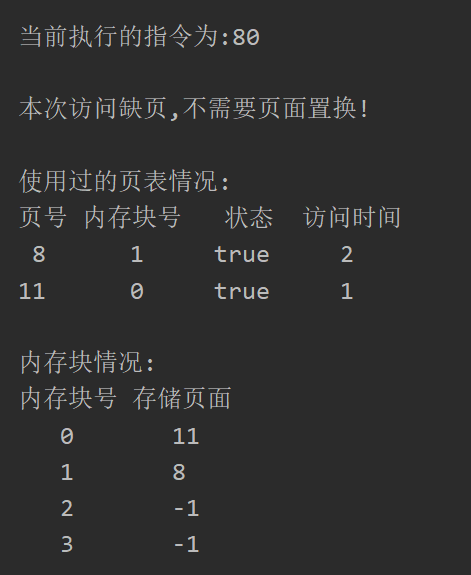
执行第一条指令，116：



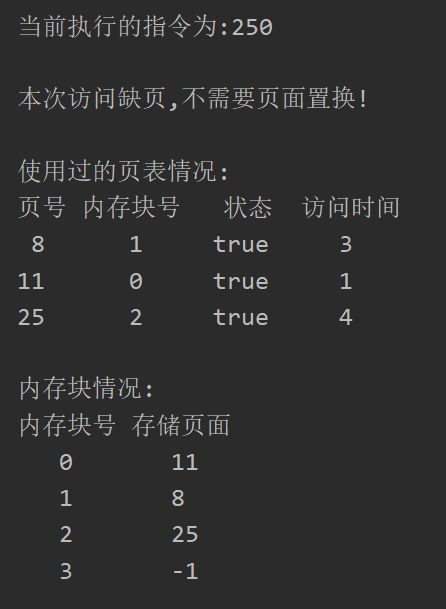
第二条，指令117（**之后的指令两两基本同一个页面，所以每隔2条指令截图一次**）：



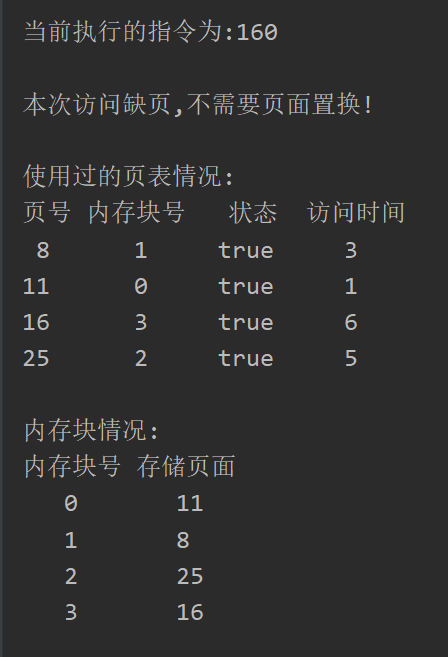
第三条，指令80：



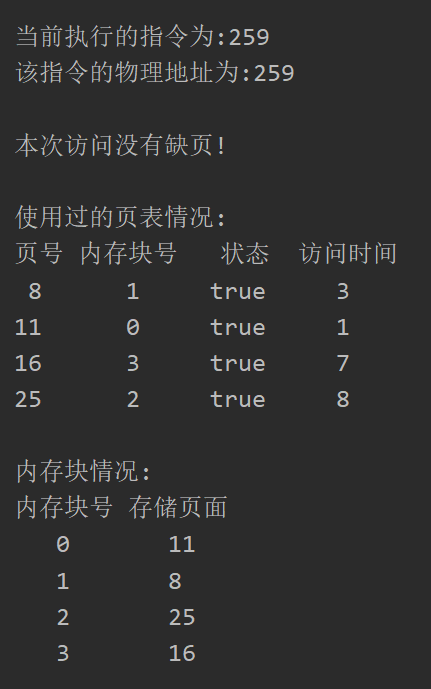
第五条，指令250：



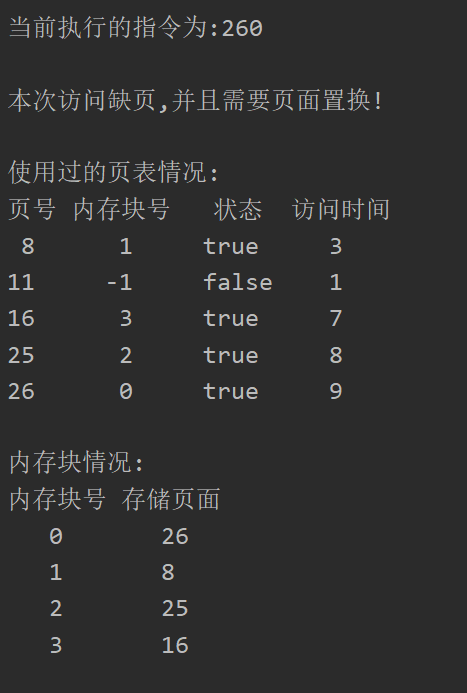
第七条，指令160：



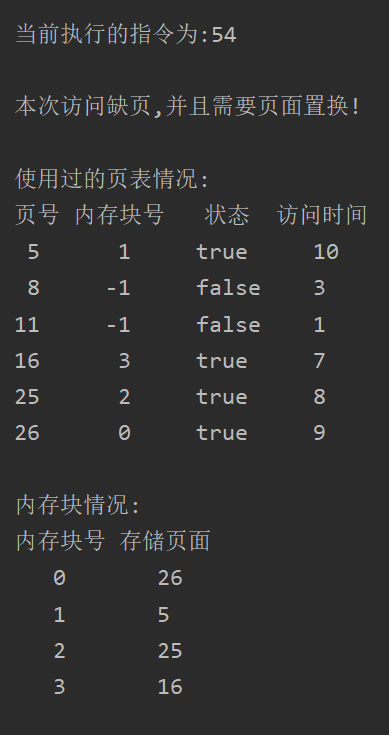
第九条，指令259（没有缺页，但是更新了访问时间）：



第十条，指令260（虽然和259相连，但是不在同一页上，所以缺页，并且发生页面置换。替换最近未访问的页号11，且页号11状态标志为不在内存块中）：

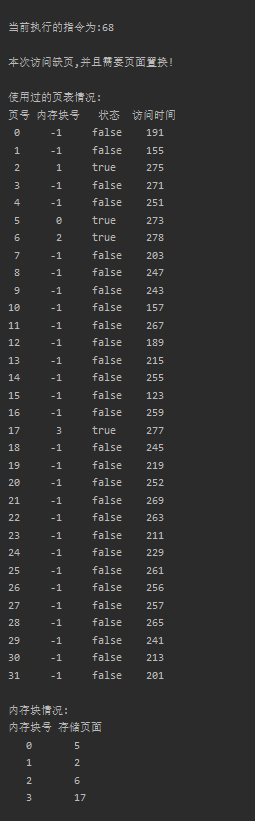


第十一条，指令54（缺页，页面置换。替换最久未使用的页面8）：



随后的指令基本类似，只截取最后两条指令：

指令81：



指令82：

