**内存管理——请求调页存储管理方式模拟**

**项目设计报告**

Designer：胡峻玮 2153393

**一、项目背景**

**1.1 项目概述**

本项目旨在模拟请求调页存储管理方式，模拟一个作业的执行过程。该作业共有320条指令，每个页面可以存放10条指令，分配给作业的内存块为4个。模拟过程中记录并计算缺页次数和缺页率，并提供两种页面置换算法（FIFO和LRU）供选择。同时，模拟作业的指令访问顺序可以按照顺序执行和随机跳转相结合的原则进行。

**1.2 系统需求**

本系统的主要需求如下：

1. 指令存放：每个页面可存放10条指令，整个作业共有320条指令，即地址空间为32页。
2. 页面置换：分配给作业的内存块为4个，需实现页面置换算法。当所访问指令不在内存中时，发生缺页，需要记录缺页次数，并将其调入内存。
3. 页面置换算法：支持FIFO（先进先出）和LRU（最近最少使用）两种页面置换算法。
4. 指令执行顺序：50%的指令按顺序执行，25%的指令随机跳转到前地址部分，25%的指令随机跳转到后地址部分。
5. 缺页统计：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。
6. 用户交互：提供用户界面供用户选择执行模式（单步执行、连续执行、暂停、重置）和页面置换算法，并展示执行过程和结果。

**二、开发环境**

**2.1 硬件和软件环境**

1. 操作系统：Windows 11
2. 开发工具：PyCharm 2023.3.3
3. 编程语言：Python 3.11

**2.2 主要依赖库**

1. PyQt5：用于创建图形用户界面（GUI），提供了丰富的控件和布局管理器。
2. queue：Python标准库中的队列模块，用于实现FIFO页面置换算法。
3. random：Python标准库中的随机数生成模块，用于模拟指令的随机跳转。

**三、系统设计**

**3.1 系统架构**

本项目采用模块化设计，主要分为以下几个部分：

1. 界面模块：负责用户界面显示和交互。
2. 模拟线程模块：负责模拟作业的执行过程，包括指令执行和页面置换。
3. 管理器模块：实现页面置换算法和内存管理。
4. 任务模块：生成任务并获取指令。

系统架构图如下：

MainWindow

│

└── SimulationThread

├── run\_task

├── run\_task\_fifo

└── run\_task\_lru

│

└── MyManager

├── page\_swap

├── allocate\_empty\_page

├── update\_unused\_time

├── run\_task

├── run\_task\_fifo

└── run\_task\_lru

│

└── Task

├── get\_current\_code\_id

└── generate\_next\_code\_id

│

└── Pcb

├── \_\_init\_\_

**3.2 功能模块**

1. **界面模块（MainWindow）：**

* 提供用户界面，用户可以通过界面选择执行模式（单步执行、连续执行、暂停、重置）和页面置换算法。
* 显示指令执行过程，包括指令序号、地址、是否缺页、换出页、换入页等信息。
* 显示缺页次数和缺页率。

1. **模拟线程模块（SimulationThread）：**

* 负责模拟作业的执行过程，根据用户选择的页面置换算法和指令执行顺序执行指令，并进行页面置换。
* 更新界面信息。

1. **管理器模块（MyManager）：**

* 实现页面置换算法，包括FIFO和LRU两种算法。
* 分配和置换页面，记录页面分配情况。

1. **任务模块（Task）：**

* 生成模拟任务，随机获取当前执行的指令ID。
* 根据不同状态（顺序执行、随机跳转）生成下一条指令ID。

1. **任务控制块（Pcb）：**

* 记录任务的页面表信息。
* 初始化页面表，根据页面大小和代码数量生成页面表，记录页面在内存中的分配情况。

**3.3 算法介绍**

1. **FIFO（先进先出）算法：**
2. 算法简介

* 基本思想：FIFO算法根据页面进入内存的顺序进行页面置换，最先进入内存的页面最先被置换出去。
* 实现方式：使用队列（queue）来记录页面进入内存的顺序。每当需要置换页面时，从队列头部取出最早进入的页面进行置换，并将新页面加入队列尾部。
* 优点：实现简单，易于理解。
* 缺点：可能会将一些仍然频繁使用的页面置换出去，导致缺页频率较高。

1. 算法步骤
2. 初始化页面分配队列。
3. 判断当前指令是否在内存中。
4. 若在内存中，直接执行下一条指令。
5. 若不在内存中，判断内存是否已满。
6. 若内存未满，分配空页面并记录。
7. 若内存已满，置换最早分配的页面，更新页面分配队列。
8. 重复上述步骤直至所有指令执行完毕。
9. 算法示例

def run\_task\_fifo(self, task):  
 *# 获取当前要执行的代码信息* current\_code\_id, memory\_page\_for\_code, code\_page\_id = task.get\_current\_code\_id()  
 *# 初始化log，记录可视化界面所需信息* log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page = 0, 0, False, -1, -1, -1  
  
 if memory\_page\_for\_code != -1: *# 在内存中* log\_page\_missing, log\_old\_page, log\_memory\_page = False, -1, memory\_page\_for\_code  
 else: *# 不在内存中* if self.page\_allocated\_amount < self.task\_memory\_page\_amount: *# 内存没有分配满* for i in range(self.task\_memory\_page\_amount):  
 if self.task\_page[i] is None: *# 找到空位置* self.allocate\_empty\_page(i, code\_page\_id, task) *# 分配空的模拟内存页面* self.page\_allocate\_queue.put(i) *# 插入FIFO算法队列* log\_page\_missing, log\_old\_page, log\_memory\_page = True, -1, i  
 break  
 else: *# 内存已满，进行页面调换* dst\_memory\_page\_id = self.page\_allocate\_queue.get() *# 从队列中取出最早分配的页序号* old\_page = self.page\_swap(dst\_memory\_page\_id, code\_page\_id, task) *# 页面调换* self.page\_allocate\_queue.put(dst\_memory\_page\_id) *# 新分配的页序号插入队尾* log\_page\_missing, log\_old\_page, log\_memory\_page = True, old\_page, dst\_memory\_page\_id  
  
 log\_code\_num, log\_cur\_code, log\_code\_page = self.code\_num, current\_code\_id, code\_page\_id  
 self.code\_num += 1  
 return log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page

1. **LRU（最近最少使用）算法：**
2. 算法简介

* 基本思想：LRU算法根据页面最近的使用情况进行页面置换，置换最近最少使用的页面。
* 实现方式：使用列表记录每个页面的未使用时间。每次访问页面时，更新其未使用时间为0，并增加其他页面的未使用时间。当需要置换页面时，选择未使用时间最长的页面进行置换。
* 优点：更符合实际使用情况，可以减少缺页频率。
* 缺点：实现较复杂，需要额外的时间和空间记录页面的未使用时间。

1. 算法步骤
2. 初始化页面未使用时间列表。
3. 判断当前指令是否在内存中。
4. 若在内存中，更新该页面的未使用时间。
5. 若不在内存中，判断内存是否已满。
6. 若内存未满，分配空页面并更新未使用时间列表。
7. 若内存已满，置换最近最少使用的页面，更新未使用时间列表。
8. 重复上述步骤直至所有指令执行完毕。
9. 算法示例

def run\_task\_lru(self, task):  
 *# 获取当前要执行的代码信息* current\_code\_id, memory\_page\_for\_code, code\_page\_id = task.get\_current\_code\_id()  
 *# 初始化log，记录可视化界面所需信息* log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page = 0, 0, False, -1, -1, -1  
  
 if memory\_page\_for\_code != -1: *# 在内存中* self.update\_unused\_time(used\_page\_id=memory\_page\_for\_code)  
 log\_page\_missing, log\_old\_page, log\_memory\_page = False, -1, memory\_page\_for\_code  
 else: *# 不在内存中* if self.page\_allocated\_amount < self.task\_memory\_page\_amount: *# 内存没有分配满* for i in range(self.task\_memory\_page\_amount):  
 if self.task\_page[i] is None: *# 找到空位置* self.allocate\_empty\_page(i, code\_page\_id, task) *# 分配空的模拟内存页面* self.update\_unused\_time(used\_page\_id=i) *# 更新LRU算法记录的未使用时间* log\_page\_missing, log\_old\_page, log\_memory\_page = True, -1, i  
 break  
 else: *# 内存已满，进行页面调换* dst\_memory\_page\_id = self.unused\_time.index(max(self.unused\_time)) *# 找到最久未使用的页序号* old\_page = self.page\_swap(dst\_memory\_page\_id, code\_page\_id, task) *# 页面调换* self.update\_unused\_time(used\_page\_id=dst\_memory\_page\_id) *# 更新LRU算法记录的未使用时间* log\_page\_missing, log\_old\_page, log\_memory\_page = True, old\_page, dst\_memory\_page\_id  
  
 log\_code\_num, log\_cur\_code, log\_code\_page = self.code\_num, current\_code\_id, code\_page\_id  
 self.code\_num += 1  
 return log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page

**四、用户界面设计**

**4.1 界面布局**

用户界面采用图形化界面设计，主要包括以下几个部分：

1. **页面展示区：**

* 位置：界面的左侧部分。
* 功能：显示内存中各页面的分配情况。每个页面用一个按钮表示，当页面内容变化时，按钮的背景颜色会随之变化。

1. **指令执行过程区：**

* 位置：界面的中间部分。
* 功能：显示指令执行的详细过程，包括指令序号、地址、是否缺页、换出页、换入页等信息。

1. **控制区：**

* 位置：界面的右侧部分。
* 功能：提供用户控制模拟执行的按钮，包括“连续执行”、“单步执行”、“暂停”、“重置”四个按钮，以及总指令数和页面置换算法选择。
* 按钮：
* 连续执行：点击后开始连续执行指令。
* 单步执行：点击后执行下一条指令。
* 暂停：点击后暂停指令执行。
* 重置：点击后重置所有数据和界面。
* 下拉菜单：选择页面置换算法（FIFO或LRU）。

1. **缺页信息区：**

* 位置：界面的底部部分。
* 功能：显示当前缺页次数和缺页率。

1. **界面布局示意图如下：**



**4.2 交互设计**

1. **按钮交互：**

* 连续执行：用户点击后，模拟程序会自动连续执行所有指令，直到完成或用户点击暂停。
* 单步执行：用户点击后，模拟程序只执行当前指令，并等待用户的下一步操作。
* 暂停：用户点击后，暂停当前的指令执行。用户可以继续点击“单步执行”或“连续执行”按钮继续操作。
* 重置：用户点击后，重置所有数据和界面，缺页次数和缺页率清零，所有按钮恢复初始状态。

1. **页面置换算法选择：**

* 用户可以通过下拉菜单选择页面置换算法（FIFO或LRU），并在重置后生效。

1. **缺页信息实时更新：**

* 缺页次数和缺页率会在每次指令执行后实时更新，并在界面底部显示。

1. **页面展示区交互：**

* 每次页面内容变化时，页面按钮的背景颜色会随之变化，以直观展示页面分配情况。

1. **指令执行过程显示：**

* 指令执行过程区实时显示指令的详细执行情况，包括当前指令、物理地址、缺页情况、换出页和换入页等信息。

**五、代码实现**

**5.1 核心代码解析**

1. **SimulationThread 类**

* 描述：SimulationThread 负责模拟作业的执行过程。它继承自 QThread，用于在后台线程中运行模拟逻辑，以保持界面响应。
* 主要方法：

1. check\_for\_reset：检查是否需要重置。如果需要重置，发出 reset\_signal 信号，并返回 True。
2. wait\_for\_initial\_mode：等待用户第一次设定运行模式（单步执行或连续执行）。
3. wait\_for\_mode\_change：在模拟过程中等待用户改变运行模式（单步执行、连续执行或暂停）。
4. run：线程的主逻辑，根据用户选择的页面置换算法执行任务，并更新界面信息

* 代码：

class SimulationThread(QThread):  
 *# 定义信号* update\_signal = pyqtSignal(int, int, bool, int, int, int, int, int, int)  
 reset\_signal = pyqtSignal()  
  
 def \_\_init\_\_(self):  
 super(SimulationThread, self).\_\_init\_\_()  
 self.tmp\_reset\_flag = None  
 *# 连接信号与槽* self.update\_signal.connect(update\_ui)  
 self.reset\_signal.connect(reset\_ui)  
  
 *# 检查重置状态* def check\_for\_reset(self):  
 global reset\_flag  
 semaphore\_acquire(reset\_flag\_semaphore)  
 if reset\_flag:  
 reset\_flag = 0  
 semaphore\_release(reset\_flag\_semaphore)  
 self.reset\_signal.emit()  
 return True  
 semaphore\_release(reset\_flag\_semaphore)  
 return False  
  
 *# 等待用户第一次设定运行模式* def wait\_for\_initial\_mode(self):  
 global run\_mode  
 while True:  
 semaphore\_acquire(run\_mode\_semaphore)  
 if run\_mode in [1, 2]:  
 semaphore\_release(run\_mode\_semaphore)  
 break  
 semaphore\_release(run\_mode\_semaphore)  
  
 *# 等待用户更改运行模式* def wait\_for\_mode\_change(self):  
 global run\_mode  
 while True:  
 semaphore\_acquire(run\_mode\_semaphore)  
 if run\_mode == 1:  
 run\_mode = 0  
 semaphore\_release(run\_mode\_semaphore)  
 break  
 elif run\_mode == 2:  
 semaphore\_release(run\_mode\_semaphore)  
 break  
 semaphore\_release(run\_mode\_semaphore)  
  
 semaphore\_acquire(reset\_flag\_semaphore)  
 if reset\_flag:  
 semaphore\_release(reset\_flag\_semaphore)  
 self.tmp\_reset\_flag = True  
 break  
 semaphore\_release(reset\_flag\_semaphore)  
  
 def run(self):  
 while True:  
 global run\_mode  
 semaphore\_acquire(run\_mode\_semaphore)  
 run\_mode = 0  
 semaphore\_release(run\_mode\_semaphore)  
  
 self.wait\_for\_initial\_mode()  
  
 global reset\_flag  
 semaphore\_acquire(reset\_flag\_semaphore)  
 reset\_flag = 0  
 semaphore\_release(reset\_flag\_semaphore)  
  
 page\_size = 10  
 code\_amount = ui.spinBox.value()  
 page\_swap\_algo = ui.comboBox.currentText()  
  
 task = Task(page\_size=page\_size, code\_amount=code\_amount)  
 manager = MyManager(page\_size=page\_size, algo=page\_swap\_algo)  
 page\_missing = 0  
  
 self.tmp\_reset\_flag = False  
  
 while True:  
 if self.check\_for\_reset():  
 break  
  
 self.wait\_for\_mode\_change()  
 if self.tmp\_reset\_flag:  
 self.tmp\_reset\_flag = False  
 continue  
  
 log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page = manager.run\_task(task)  
  
 if log\_page\_missing:  
 page\_missing += 1  
 if log\_code\_num >= (code\_amount - 1):  
 semaphore\_acquire(reset\_flag\_semaphore)  
 reset\_flag = 1  
 semaphore\_release(reset\_flag\_semaphore)  
  
 self.update\_signal.emit(log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page,log\_memory\_page, page\_missing, page\_size, code\_amount)  
 time.sleep(0.1)

1. **MyManager 类**

* 描述：MyManager 负责管理内存页面的分配和置换，实现FIFO和LRU两种页面置换算法。
* 主要方法：

1. page\_swap：执行页面调换，将新的作业页加载到内存中。
2. allocate\_empty\_page：分配空页面，将作业页加载到未使用的内存页面中。
3. update\_unused\_time：更新页面的未使用时间，用于LRU算法。
4. run\_task：根据选定的算法（FIFO或LRU）运行任务。
5. run\_task\_fifo：实现FIFO页面置换算法。
6. run\_task\_lru：实现LRU页面置换算法。

* 代码：

class MyManager:  
 def \_\_init\_\_(self, page\_size, algo):  
 self.task\_memory\_page\_amount = 4 *# 分配给任务的页面数* self.page\_size = page\_size *# 页面尺寸* self.task\_page = [None] \* self.task\_memory\_page\_amount *# 分配的页面 记录页面号* self.code\_num = 0 *# 记录执行到第几条代码* self.algo = algo *# 记录设定的管理器的算法* self.page\_allocated\_amount = 0  
  
 if self.algo == 'FIFO': *# 根据不同的算法配置不同的参数* self.page\_allocate\_queue = queue.Queue() *# 记录页面分配顺序* elif self.algo == 'LRU':  
 self.unused\_time = [None] \* self.task\_memory\_page\_amount *# 记录页面未被使用的时间* def page\_swap(self, dst\_memory\_page\_id, code\_page\_id, task): *# 页面调换* old\_page = self.task\_page[dst\_memory\_page\_id] *# 获取旧页面* self.task\_page[dst\_memory\_page\_id] = code\_page\_id *# 写入新页面* task.pcb.page\_table[old\_page] = -1 *# 更新旧页面页表* task.pcb.page\_table[code\_page\_id] = dst\_memory\_page\_id *# 更新新页面页表* return old\_page  
  
 def allocate\_empty\_page(self, empty\_page\_id, code\_page\_id, task): *# 分配空页面* self.task\_page[empty\_page\_id] = code\_page\_id  
 task.pcb.page\_table[code\_page\_id] = empty\_page\_id  
 self.page\_allocated\_amount += 1  
  
 def update\_unused\_time(self, used\_page\_id): *# 用于LRU算法，更新页面未被使用的时间* for i in range(self.task\_memory\_page\_amount):  
 if self.unused\_time[i] is not None:  
 self.unused\_time[i] += 1  
 self.unused\_time[used\_page\_id] = 0 *# 被使用的页面重置未使用时间* def run\_task(self, task):  
 if self.algo == 'FIFO':  
 return self.run\_task\_fifo(task)  
 elif self.algo == 'LRU':  
 return self.run\_task\_lru(task)  
  
 def run\_task\_fifo(self, task):  
 *# 获取当前要执行的代码信息* current\_code\_id, memory\_page\_for\_code, code\_page\_id = task.get\_current\_code\_id()  
 *# 初始化log，记录可视化界面所需信息* log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page = 0, 0, False, -1, -1, -1  
  
 if memory\_page\_for\_code != -1: *# 在内存中* log\_page\_missing, log\_old\_page, log\_memory\_page = False, -1, memory\_page\_for\_code  
 else: *# 不在内存中* if self.page\_allocated\_amount < self.task\_memory\_page\_amount: *# 内存没有分配满* for i in range(self.task\_memory\_page\_amount):  
 if self.task\_page[i] is None: *# 找到空位置* self.allocate\_empty\_page(i, code\_page\_id, task) *# 分配空的模拟内存页面* self.page\_allocate\_queue.put(i) *# 插入FIFO算法队列* log\_page\_missing, log\_old\_page, log\_memory\_page = True, -1, i  
 break  
 else: *# 内存已满，进行页面调换* dst\_memory\_page\_id = self.page\_allocate\_queue.get() *# 从队列中取出最早分配的页序号* old\_page = self.page\_swap(dst\_memory\_page\_id, code\_page\_id, task) *# 页面调换* self.page\_allocate\_queue.put(dst\_memory\_page\_id) *# 新分配的页序号插入队尾* log\_page\_missing, log\_old\_page, log\_memory\_page = True, old\_page, dst\_memory\_page\_id  
  
 log\_code\_num, log\_cur\_code, log\_code\_page = self.code\_num, current\_code\_id, code\_page\_id  
 self.code\_num += 1  
 return log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page  
  
 def run\_task\_lru(self, task):  
 *# 获取当前要执行的代码信息* current\_code\_id, memory\_page\_for\_code, code\_page\_id = task.get\_current\_code\_id()  
 *# 初始化log，记录可视化界面所需信息* log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page = 0, 0, False, -1, -1, -1  
  
 if memory\_page\_for\_code != -1: *# 在内存中* self.update\_unused\_time(used\_page\_id=memory\_page\_for\_code)  
 log\_page\_missing, log\_old\_page, log\_memory\_page = False, -1, memory\_page\_for\_code  
 else: *# 不在内存中* if self.page\_allocated\_amount < self.task\_memory\_page\_amount: *# 内存没有分配满* for i in range(self.task\_memory\_page\_amount):  
 if self.task\_page[i] is None: *# 找到空位置* self.allocate\_empty\_page(i, code\_page\_id, task) *# 分配空的模拟内存页面* self.update\_unused\_time(used\_page\_id=i) *# 更新LRU算法记录的未使用时间* log\_page\_missing, log\_old\_page, log\_memory\_page = True, -1, i  
 break  
 else: *# 内存已满，进行页面调换* dst\_memory\_page\_id = self.unused\_time.index(max(self.unused\_time)) *# 找到最久未使用的页序号* old\_page = self.page\_swap(dst\_memory\_page\_id, code\_page\_id, task) *# 页面调换* self.update\_unused\_time(used\_page\_id=dst\_memory\_page\_id) *# 更新LRU算法记录的未使用时间* log\_page\_missing, log\_old\_page, log\_memory\_page = True, old\_page, dst\_memory\_page\_id

log\_code\_num, log\_cur\_code, log\_code\_page = self.code\_num, current\_code\_id, code\_page\_id  
 self.code\_num += 1  
 return log\_code\_num, log\_cur\_code, log\_page\_missing, log\_old\_page, log\_code\_page, log\_memory\_page

1. **Task 类**

* 描述：Task 类负责生成模拟任务和获取当前执行的指令ID。任务包括320条指令，每个页面可存放10条指令。
* 主要方法：

1. \_\_init\_\_：初始化任务信息，包括页面大小、代码数量、当前代码序号和状态。
2. get\_current\_code\_id：获取当前指令ID，根据不同状态生成下一条指令ID。

* 代码：

class Task:  
 def \_\_init\_\_(self, page\_size, code\_amount):  
 self.code\_amount = code\_amount *# 代码数量* self.page\_size = page\_size *# 页尺寸* self.pcb = Pcb(page\_size, code\_amount) *# 记录任务信息* self.current\_code\_id = random.randint(0, self.code\_amount - 1) *# 当前代码序号* self.state = 0 *# 状态，用于生成下一条代码的序号* def get\_current\_code\_id(self):  
 tmp\_code\_id = self.current\_code\_id  
  
 if self.state == 0:  
 self.current\_code\_id = (self.current\_code\_id + 1) % self.code\_amount *# 顺序执行* elif self.state == 1:  
 self.current\_code\_id = random.randint(0, max(self.current\_code\_id - 1, 0)) *# 从当前代码之前的代码中随机取* elif self.state == 2:  
 self.current\_code\_id = (self.current\_code\_id + 1) % self.code\_amount *# 顺序执行* elif self.state == 3:  
 self.current\_code\_id = random.randint(min(self.current\_code\_id + 1, self.code\_amount - 1), self.code\_amount - 1) *# 从当前代码之后的代码中随机取* self.state = (self.state + 1) % 4 *# 在四种状态中循环，每次被调用切换到下一个状态* return tmp\_code\_id, self.pcb.page\_table[tmp\_code\_id // self.page\_size], tmp\_code\_id // self.page\_size

1. **Pcb 类**

* 描述：Pcb 类用于记录任务的页面表信息。每个任务都有一个对应的任务控制块（PCB），其中包含页表信息。
* 主要方法：

1. \_\_init\_\_：初始化页表信息，生成页面表并记录页面在内存中的分配情况。

* 代码：

class Pcb:  
 def \_\_init\_\_(self, page\_size, code\_amount):  
 self.task\_page\_table\_size = code\_amount // page\_size  
 if code\_amount % page\_size != 0:  
 self.task\_page\_table\_size += 1 *# 若不能整除，则增加一个页表页面* self.page\_table = [-1] \* self.task\_page\_table\_size *# 初始化页表，值为-1表示不在内存中*

**六、代码测试**

**6.1 UI界面**



**6.2 任务测试**

1. 完整执行320条指令
   1. FIFO算法



* 1. LRU算法



1. 单步执行



1. 执行过程中暂停执行



1. 重置

