# 进程管理 - 电梯调度项目设计方案报告

2051498 储岱泽

## 项目需求

该项目需要为一栋大楼的电梯编写电梯调度算法程序，让电梯能够尽可能高效的运转。该大楼内一共20层，有5部互联的电梯，我们可以基于线程的思想编写一个电梯调度的程序。

## 功能描述

1. 每个电梯内部设置有必要功能键：如楼层数字键、关门键、开门键、电梯故障按钮、以及有一个LED显示屏显示当前电梯的楼层数。

2. 每层楼的每部电梯门口，应该有上行和下行按钮和当前电梯状态的数码显示器。

3. 所有电梯初始状态都在第一层。每个电梯如果在它的上层或者下层没有相应请求情况下，则应该在原地保持不动。

4.为了方便测试电梯的性能，界面的右边编写有一个电梯控制台，可以随机生成任务给电梯执行，并且可以在下面的输出框中看见电梯的运行情况。

## **开发环境**

•开发环境: Windows 10，MacOS

（在MacOS下编写程序，在Windows下打包成exe文件）

•**开发软件**: Visual Studio Code 1.77.3 (Universal)

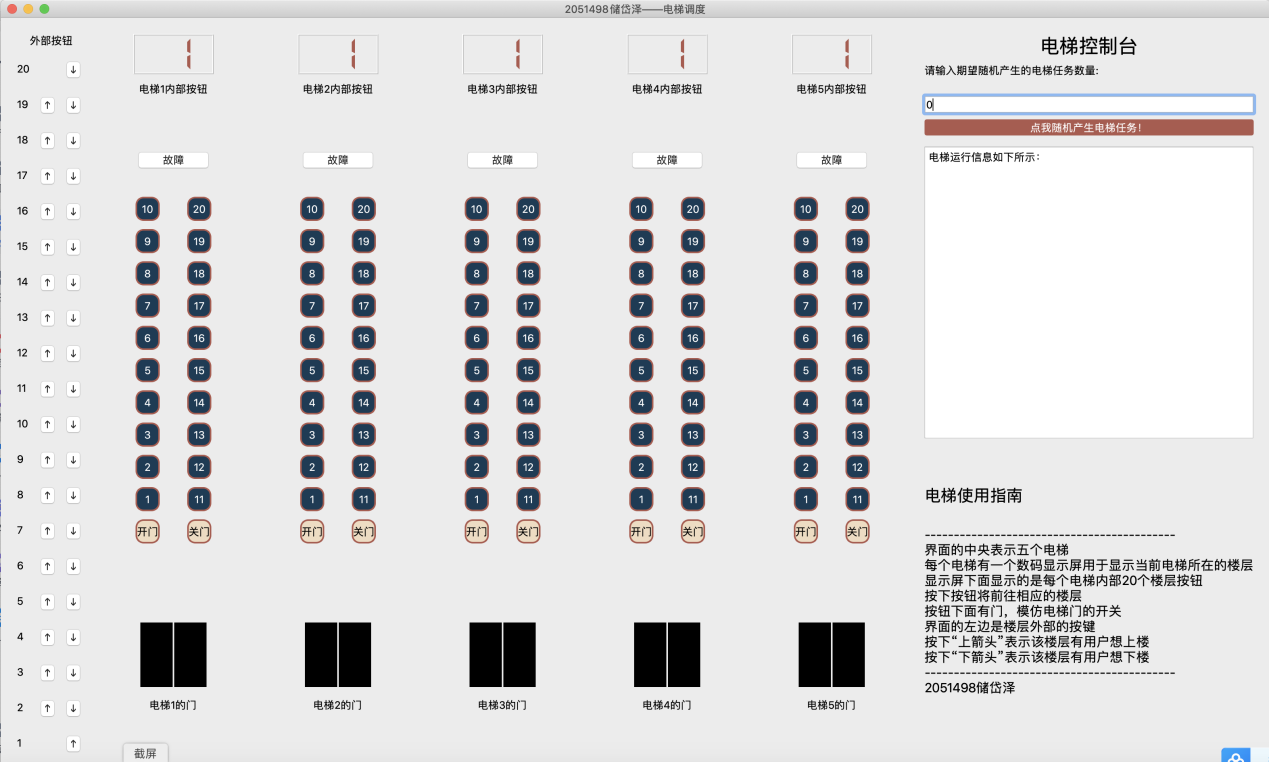
•**开发语言:** python 3.8.2

•**主要引用块内容:**

1. PyQt5（QtCore，QtWidgets，QThread，QMutex, QTimer）
2. Enum
3. partial

## 4.操作说明

•双击运行elevator.exe，或者打开编辑器运行python文件可以看见如下所示的界面：



这个界面主要分成了三个部分：外部按钮部分，五部电梯内部按钮部分以及电梯控制台部分。

**•外部按钮部分：**这一部分主要由两种类型的按钮组成，一种是“上箭头”的按钮，一种是“下箭头”的按钮。在各个楼层的用户如果按下“上箭头”的按钮则表示请求向上运行，如果按下“下箭头”的按钮则表示请求向下运行。

•**电梯内部按钮部分：**该栋楼一共有五部电梯，每部电梯内部按钮的布局都是一样的。在内部按钮的顶部是一个电子显示屏，用来显示当前电梯到达的楼层数；下面有一个“故障”按钮，当电梯运行故障的时候可以点击该按钮，锁定电梯；然后下面有20个按钮分别表示20个楼层，电梯内的乘客可以按下这些按钮前往对应的楼层；除此之外，下面还有“开门”和“关门”的按钮；以及最底下有“电梯的门”，当用户按下“开门”键或者电梯到达指定楼层时，门就会打开。

•**电梯控制台部分：**电梯控制台部分用户可以直接在输入框中输入期望产生任务的数量来随机给电梯和外部按钮分配任务，然后测试电梯的运行情况是否符合预期。同时，在电梯控制台还可以看见电梯实时的运行状况以及对于使用该电梯的说明。

**电梯运行过程截图：**



## 5.调度算法分析与设计

**5.1 常见的电梯调度算法**

常见的电梯调度算法大概有以下几种：

**•先来先服务算法（FCFS）：**该算法根据乘坐电梯的先后次序调度，虽然公平、简 单，但是一旦载荷较大，就会严重增加等待时间

**•最短寻找楼层时间优先算法（SSTF）：**该算法每次寻找可以最快满足的楼层，这 样可以减少等待时间，但部分楼层（如顶楼和1楼）可能长时间无法响应，出现 类似于操作系统中“饥饿”（starvation）的现象

**•扫描算法（SCAN）：**该算法使电梯在最底层和最顶层之间连续往返，所有方向相 同的请求在一趟中完成，以确保总体等待时间的稳定。

**•LOOK 算法：**该算法对扫描算法改进，当电梯发现移动方向上不再有目标时立即 改变方向。

**本项目的电梯调度算法基本基于LOOK算法。**区别在于，在本项目中，电梯发现移动方向上无目标时，不会立即改变运行方向，而是等到反方向产生任务后方改变运行方向。

该电梯需要调度的任务主要有两个来源——**外部电梯按钮产生的任务以及内部电梯按钮产生的任务**，因此接下来我们分别对这两个任务来源进行分析，以及给出相应期望的处理方案。

5.2每部电梯内部的任务请求

**•内部任务请求**

1. 用户点击电梯内部的楼层按钮请求前往对应楼层。
2. 用户点击“开门”和“关门”按钮请求开门和关门。
3. 用户点击“故障”按钮，上报电梯故障。

**•任务响应机制**

假设电梯此时所在的楼层是cur\_floor，用户在电梯内点击了第i层的按钮

1. **若用户点击楼层按钮**

**•**如果**i=cur\_foor**，则电梯自动开门，并且打开一段时间之后自动关门。

**•**如果电梯处于**静止的空闲状态**，则电梯直接向第i层的方向运动。

**•**如果**i>cur\_floor且电梯运行方向是向上的**，则将该任务加入电梯向上需要

处理的任务数组中，并且将该任务数组中的任务按照楼层大小从小到大排序，越往前优先级越高。

**•**如果**i>cur\_floor但电梯运行方向是向下的**，则将该任务加入电梯向下需要

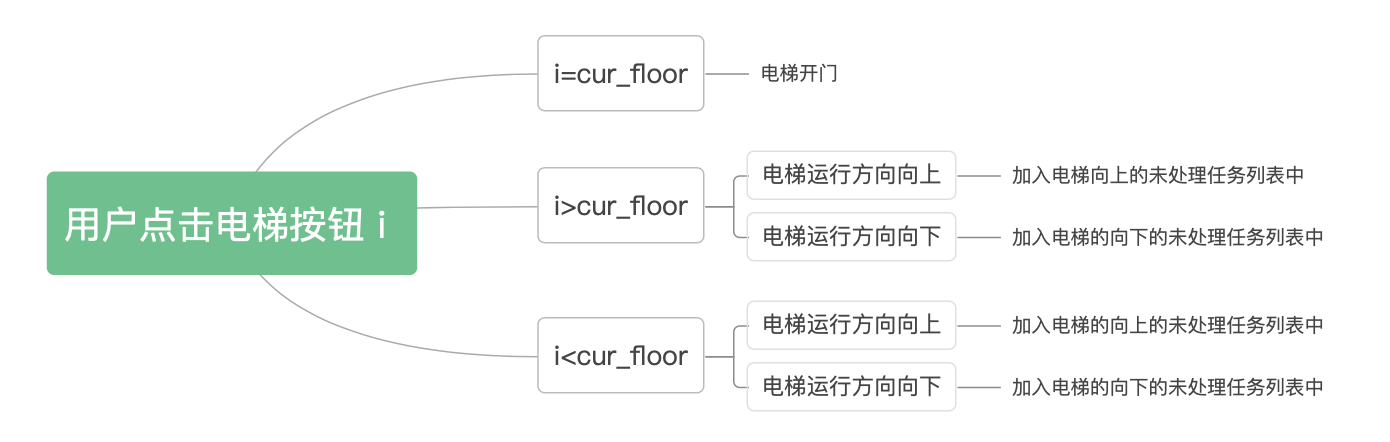
处理的任务数组中，并且将该向下任务数组中的任务按照楼层大小从大到小排序，越往前优先级越高。

**•**如果**i<cur\_floor且电梯运行方向是向下的，**则将该任务加入电梯向下需要

处理的任务数组中，并且将该向下任务数组中的任务按照楼层大小从大到小排序，越往前优先级越高。

**•**如果**i<cur\_floor但电梯运行方向是向上**的，则将该任务加入电梯向下需要

处理的任务数组中，并且将该向下任务数组中的任务按照楼层大小从大到小排序，越往前优先级越高。

****

1. **若用户点击“开门”或者“关门”按钮**

**•**点击“开门”按钮，则当电梯处于静止状态直接开门，且在一定时间后自动关闭；当电梯正在运行时，忽略这一请求保持电梯门的关闭状态。

**•**点击“关门”按钮，则当电梯门开着的时候，直接关门。

1. **若用户点击“故障”**

**•**清除该电梯此时所有的内部任务，并将外部任务分配给其他正常的电梯。然后将其内部按钮锁定。

**5.3电梯外部任务的请求**

**•外部任务的请求**

用户在其对应的楼层按下“上行”按钮或者“下行”按钮，发送上楼或者下楼的请求。

**•任务响应机制**

**•**当监测到用户按下了某一层的“上行”或者“下行”按钮之后，首先扫描五个电梯，寻找没有故障的电梯。

**•**在没有故障的电梯当中选择最快的那个能以和用户请求的运行方向一致到达该楼层的电梯，并将到达该楼层的任务塞入该电梯的任务列表之中，具体方法如下所示：

可以通过计算请求楼层与每个电梯之间的“距离”，然后比较这些“距离”，选择“距离”最小的电梯，将该楼层的任务加入该电梯的任务列表中。

假设distance为电梯以与外部按下的按钮相同的方向运行到目标楼层时，电梯运行的距离；cur\_floor为电梯此时在的楼层;task\_floor为按下的外部按钮所在楼层;max\_target表示当前电梯运行方向上，该电梯所需要处理的任务中最远的那个，则：

•若电梯本身处于静止状态，没有运动。则**distance=abs(cur\_floor-task\_floor)**

**•**若电梯刚好冲着task\_floor来，顺路，且电梯此时运行的方向与按钮的方向一致，则**distance=abs(cur\_floor-task\_floor)。**

•若电梯本身运行的方向刚好背离task\_floor的方向，或者虽然也是冲着task\_floor来，但是按钮本身代表的用户向前往的方向与电梯本身此时运行的方向不一致，则电梯此时需要先把其运行方向上最远的一个任务先处理完才会回来响应这个task\_floor处的外部任务，此时**distance=abs(max\_target-cur\_floor)+abs(max\_target-task\_floor)**。

然后选择五个电梯中distance计算下来最小的那个，将前往task\_floor的任务加入该电梯的任务列表，并重新根据距离的优先级对该电梯的任务列表进行排序。

## 具体实现

在了解了我的电梯调度算法的思路之后，接下来展示我将如何实现这个算法，并且展示在UI界面上。

###### 6.1状态设计

在我的程序中，我分别用三个枚举类来定义了在下面实现算法时我可能会用到的三类状态量：MoveState（当前电梯扫描的方向），ElevatorState（当前电梯的状态，包括运动方向，门的状态和是否故障），OuterButtonState（外部按钮可能的状态）。

# 电梯的扫描移动状态枚举，这个枚举中包含了电梯扫描的两种可能的状态，一个是向上一个是向下

class MoveState(Enum):

up = 2 #电梯在向上扫描的状态中

down = 3 #电梯在向下扫描的状态中

# 电梯状态的枚举，这个枚举中集合了电梯所有可能的状态

class ElevatorState(Enum):

normal = 0 #表示电梯状态是正常的

fault = 1 #表示电梯此时状态是故障的状态

opening\_door = 2 #表示电梯正在开门

open\_door = 3 #表示电梯门已经打开

closing\_door = 4 #表示电梯正在关门

going\_up = 5 #表示电梯正在上行

going\_down = 6 #表示电梯正在下行

# 外部按钮可能处在的状态，这个枚举中包含了电梯外部每个楼层的电梯按钮的状态

class OuterButtonState(Enum):

unassigned = 1 #表示该按钮没有被按下

waiting = 2 #表示该按钮已经被按下，正在等待被处理

finished = 3 #表示该按钮的任务已经被完成

###### 6.2类设计

以下是我的程序中所涉及到的一些类：

1. **Elevator类：**主要用来控制每一个电梯的行为。

class Elevator(QThread): # 继承Qthread

def \_\_init\_\_(self, elev\_id):

super().\_\_init\_\_() # 父类构造函数

self.elev\_id = elev\_id # 电梯编号

self.gap\_time = 10 # 时间间隔（单位：毫秒）

def lift\_up\_one\_floor(self, move\_state)#控制电梯向上运动一层

def door\_operation(self)# 一次门的操作 包括开门和关门

def fault\_tackle(self)# 当电梯发生故障时进行相应的处理

def run(self)# 处理电梯的运行

1. **OuterTaskController类：**主要用来处理对于电梯外部按钮任务的响应与处理。

class OuterTaskController(QThread):

def \_\_init\_\_(self):

super().\_\_init\_\_() # 父类构造函数

def run(self) #用来处理对于外部按钮任务的响应

1. **ElevatorUi类：**主要用来利用PyQT5画出电梯的界面显示。

# 图形化界面 同时处理画面更新和输入

class ElevatorUi(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_() # 父类构造函数

self.output = None

# 各种需要更新的按钮 显示屏等收集

self.\_\_floor\_displayers = [] #电梯上方的显示屏

self.\_\_inner\_num\_buttons = []#电梯内部的电梯按钮

self.\_\_inner\_open\_buttons = []#电梯内部的开门按钮

self.\_\_inner\_close\_buttons = []#电梯内部的关门按钮

self.\_\_outer\_up\_buttons = []#电梯外部的上楼按钮

self.\_\_outer\_down\_buttons = []#电梯外部的下楼按钮

self.\_\_inner\_fault\_buttons = []#电梯内部的故障按钮

# 定时器 用于定时更新UI界面

self.timer = QTimer()

#每个电梯的门的计时器

self.door\_timer=[]

# 设置UI

self.setup\_ui()

def setup\_ui(self)# 设置界面UI

def open\_the\_door(self,elevator\_id,choice)#用于开门

def close\_the\_door(self,elevator\_id)#用于关门

def \_\_generate\_tasks(self)#用于随机产生任务

def \_\_inner\_open\_button\_clicked(self, elevator\_id)#处理指定电梯的开门请求

def close\_after\_1s(self, elevator\_id)#开门按钮的槽函数

def \_\_inner\_close\_button\_clicked(self, elevator\_id)#处理电梯关门事件

def \_\_inner\_fault\_button\_clicked(self, elevator\_id)#处理电梯故障

def \_\_inner\_num\_button\_clicked(self, elevator\_id, floor)#如果按的是电梯内部的数字按钮，则执行下面的函数进行处理

def \_\_outer\_direction\_button\_clicked(self, floor, move\_state)#处理电梯外部每层楼的按钮点击事件

def update(self)#实时更新界面

###### 6.3其他一些全局变量

1. UI界面窗口大小设置

WINDOW\_SIZE = QRect(200, 200, 1100, 800)

1. 电梯的全局基本量

ELEVATOR\_NUM = 5 # 电梯数量

ELEVATOR\_FLOORS = 20 # 电梯层数

TIME\_PER\_FLOOR = 800 # 运行一层电梯所需时间 单位 毫秒

OPENING\_DOOR\_TIME = 500 # 打开一扇门所需时间 单位 毫秒

OPEN\_DOOR\_TIME = 500 # 门打开后维持的时间 单位 毫秒

# 全局数组定义

elevator\_states = []# 每组电梯的状态

elevator\_move\_states = []# 每台电梯当前的扫描运行状态

elevator\_cur\_floor = []# 每台电梯的当前楼层

up\_remains = []# 电梯在向上扫描的过程中，还需要处理的任务

down\_remains = []# 电梯在向下扫描的过程中还需要处理的任务

open\_button\_clicked = []# 每台电梯内部的开门/关门键是否被按（True/False）

close\_button\_clicked = []

open\_progress = []# 每台电梯开门的进度条 范围为0-1的浮点数

Doors=[]#每个电梯的门

outer\_button\_events = []# 外部按钮产生的事件

mutex = QMutex()# mutex互斥锁

**6.4重点函数设计**

接下来将展示一些核心函数的设计思路与代码。（所有的代码具体请看py文件，这边仅解释部分比较重要的函数的代码和设计思路）

1. **电梯上/下楼**

在我的程序中，控制电梯上/下楼的函数是Elevator类中的lift\_up\_one\_floor函数。该函数有一个需要传的参数，即move\_state，来描述电梯当前扫描的方向。

如果move\_state等于MoveState.up，也就是电梯目前的运行方向是向上的，那么就给elevator\_states数组中该电梯的状态更新为向上运行going\_up；反之亦然。

然后接下来就是电梯上一层楼的部分，为了让用户感受到电梯是在运行的，我们这里使得该电梯直接静止一段时间，一段时间后再使得显示屏上的楼层数目发生变化即可。在这段时间中，需要每隔一小段时间检查电梯是否发生故障，如果发生故障则调用fault\_tackle()做错误处理。同时，为了使得五部电梯之间线程并行，所以需要使用mutex互斥锁。

“上楼”之后，根据电梯的扫描方向对楼层数进行更改，完成一层楼的运行。

def lift\_up\_one\_floor(self, move\_state):

# 修改电梯运行状态

#如果电梯当前的扫描运行状态是向上，那么就将当前电梯的运行状态改成向上运行

if move\_state == MoveState.up:

elevator\_states[self.elev\_id] = ElevatorState.going\_up

#如果当前电梯的扫描状态是向下，那么就将当前电梯的运行状态改成向下运行

elif move\_state == MoveState.down:

elevator\_states[self.elev\_id] = ElevatorState.going\_down

has\_slept\_time = 0

#每过一个gap\_time的时间就需要检查一下电梯是否故障，直到运行完一整个楼层

while has\_slept\_time != TIME\_PER\_FLOOR:

# 需要先放开锁 不然别的线程不能运行

mutex.unlock()

#休眠gap\_time毫秒，在用户看来就是这段时间电梯在向上运行

self.msleep(self.gap\_time) # 时间间隔（单位：毫秒）

has\_slept\_time += self.gap\_time

# 锁回来

mutex.lock()

# 如果此时出故障了，就需要进行相应的故障处理

if elevator\_states[self.elev\_id] == ElevatorState.fault:

self.fault\_tackle()

return

#电梯一层已经运行完毕，对相应的扫描方向做相应的所在楼层的更改

if move\_state == MoveState.up:

elevator\_cur\_floor[self.elev\_id] += 1

elif move\_state == MoveState.down:

elevator\_cur\_floor[self.elev\_id] -= 1

elevator\_states[self.elev\_id] = ElevatorState.normal

# print(self.elev\_id, "号现在在", elevator\_cur\_floor[self.elev\_id], "楼")

if elevator\_states[self.elev\_id] == ElevatorState.fault:

self.fault\_tackle()

if move\_state == MoveState.up:

elevator\_cur\_floor[self.elev\_id] += 1

elif move\_state == MoveState.down:

elevator\_cur\_floor[self.elev\_id] -= 1

elevator\_states[self.elev\_id] = ElevatorState.normal

# print(self.elev\_id, "号现在在", elevator\_cur\_floor[self.elev\_id], "楼")

if elevator\_states[self.elev\_id] == ElevatorState.fault:

self.fault\_tackle()

1. **故障处理**

如果用户按下了“故障”按钮，则电梯不能运行，需要清空其目前所有的任务，并且锁住电梯内部的按钮。

清空目前所有的任务包括：

·对于“关门”和“开门”按钮，状态都重新更新为“False”即未点击。

·对于已经分配给该故障电梯的来自外部按钮的任务，重新分配给其他电梯。然后清空该故障电梯所有的上下楼任务。

# 当故障发生时 清除原先的所有任务

def fault\_tackle(self):

elevator\_states[self.elev\_id] = ElevatorState.fault

open\_progress[self.elev\_id] = 0.0

open\_button\_clicked[self.elev\_id] = False

close\_button\_clicked[self.elev\_id] = False

elevator\_states[self.elev\_id] = ElevatorState.fault

for outer\_task in outer\_button\_events:

if outer\_task.state == OuterButtonState.waiting:

if outer\_task.target in up\_remains[self.elev\_id] or outer\_task.target in down\_remains[self.elev\_id]:

outer\_task.state = OuterButtonState.unassigned # 把原先分配给它的任务交OuterTaskController重新分配

up\_remains[self.elev\_id] = []

down\_remains[self.elev\_id] = []

1. **控制电梯上下扫描的方向**

我们的电梯采用的是Look算法，当电梯在运行过程中，把一个方向上所有任务全部完成之后，就会掉头去处理另外一个方向的任务。当上下所有的任务都处理完成，up\_remains和down\_remains数组均为空的时候，电梯停留在原地不动。

def run(self):

while True:

mutex.lock()

if elevator\_states[self.elev\_id] == ElevatorState.fault:

self.fault\_tackle()

mutex.unlock()

continue

self.fault\_tackle()

mutex.unlock()

continue

# 向上扫描状态时

if elevator\_move\_states[self.elev\_id] == MoveState.up:

#如果上面还有未完成的任务，即电梯的上面还有请求没有满足

if up\_remains[self.elev\_id] != []:

#如果已经到达了目标楼层

if up\_remains[self.elev\_id][0] == elevator\_cur\_floor[self.elev\_id]:

self.door\_operation()

if up\_remains != []:

up\_remains[self.elev\_id].pop(0)

for outer\_task in outer\_button\_events:

if outer\_task.target == elevator\_cur\_floor[self.elev\_id]:

outer\_task.state = OuterButtonState.finished # 交给handler处理

elif up\_remains[self.elev\_id][0] > elevator\_cur\_floor[self.elev\_id]:

self.lift\_up\_one\_floor(MoveState.up)

# 当没有上行目标而出现下行目标时 更换状态

elif up\_remains[self.elev\_id] == [] and down\_remains[self.elev\_id] != []:

elevator\_move\_states[self.elev\_id] = MoveState.down

# 向下扫描状态时

elif elevator\_move\_states[self.elev\_id] == MoveState.down:

if down\_remains[self.elev\_id] != []:

if down\_remains[self.elev\_id][0] == elevator\_cur\_floor[self.elev\_id]:

self.door\_operation()

# 到达以后 把完成的任务删去

# 内部的任务

if down\_remains != []:

down\_remains[self.elev\_id].pop(0)

# 外部按钮的任务

for outer\_task in outer\_button\_events:

if outer\_task.target == elevator\_cur\_floor[self.elev\_id]:

outer\_task.state = OuterButtonState.finished

elif down\_remains[self.elev\_id][0] < elevator\_cur\_floor[self.elev\_id]:

self.lift\_up\_one\_floor(MoveState.down)

# 当没有下行目标而出现上行目标时 更换状态

elif down\_remains[self.elev\_id] == [] and up\_remains[self.elev\_id] != []:

elevator\_move\_states[self.elev\_id] = MoveState.up

mutex.unlock()

1. **任务响应算法**

在前面的调度算法分析与设计中，我们已经详细的解释了该项目中所用的调度算法任务分配机制：

如果是内部按钮产生的任务，直接放入对应电梯的任务列表之中，并重新对电梯的任务列表按照任务轻重缓急排序。

如果是外部按钮产生的任务，先计算distance，然后找到distance最小的电梯，即能够最快响应的那个电梯，将该任务放入该电梯的任务列表之中。

while True:

mutex.lock()

global outer\_button\_events

# 找到距离最短的电梯的id

for outer\_task in outer\_button\_events:

if outer\_task.state == OuterButtonState.unassigned: # 如果还没有把这个按钮的任务分配给任意的电梯

#先初始化最短距离为电梯楼层数+1

min\_distance = ELEVATOR\_FLOORS + 1

target\_id = -1

#寻找目前可以最快响应的电梯

for i in range(ELEVATOR\_NUM):

# 符合要求的电梯，必须没有故障，所以如果这个电梯是故障电梯，那么直接淘汰

if elevator\_states[i] == ElevatorState.fault:

continue

# 如果已经上行/下行了 就设成已经到达目的地的楼层了

origin = elevator\_cur\_floor[i]

if elevator\_states[i] == ElevatorState.going\_up:

origin += 1

elif elevator\_states[i] == ElevatorState.going\_down:

origin -= 1

#如果电梯的运行状态是向上的，那么就将该电梯上面的待处理的所有任务的一维数组赋值给targets

if elevator\_move\_states[i] == MoveState.up:

targets = up\_remains[i]

else: #如果下行，就把下面的待处理任务给targets

targets = down\_remains[i]

# 本身对某一种方向来说，根据这部电梯是否与它运行方向相同，是在上方还是下方，是否有任务，分为8种情况.

# 如果电梯运行方向无任务，则直接算绝对值

if targets == []:

distance = abs(origin - outer\_task.target)

# 如果电梯朝着按键所在楼层而来 且运行方向与理想方向相同 也是直接绝对值

elif elevator\_move\_states[i] == outer\_task.move\_state and \

((outer\_task.move\_state == MoveState.up and outer\_task.target >= origin) or(outer\_task.move\_state == MoveState.down and outer\_task.target <= origin)):

distance = abs(origin - outer\_task.target)

# 其余情况则算最远任务楼层到目标楼层的绝对值和最远楼层到当前电梯楼层的绝对值之和

else:

distance = abs(origin - targets[-1]) + abs(outer\_task.target - targets[-1])

# 寻找最小值

if distance < min\_distance:

min\_distance = distance

target\_id = i

# 假如找到了 对应添加任务

if target\_id != -1:

if elevator\_cur\_floor[target\_id] == outer\_task.target:

if outer\_task.move\_state == MoveState.up and outer\_task.target not in up\_remains[

target\_id] and elevator\_states[target\_id] != ElevatorState.going\_up:

up\_remains[target\_id].append(outer\_task.target)

up\_remains[target\_id].sort()

# 设为等待态

outer\_task.state = OuterButtonState.waiting

elif outer\_task.move\_state == MoveState.down and outer\_task.target not in down\_remains[

target\_id] and elevator\_states[target\_id] != ElevatorState.going\_down:

down\_remains[target\_id].append(outer\_task.target)

down\_remains[target\_id].sort(reverse=True) # 这里需要降序！ 例如，[20,19,..1]

# print(down\_targets)

# 设为等待态

outer\_task.state = OuterButtonState.waiting

elif elevator\_cur\_floor[target\_id] < outer\_task.target and outer\_task.target not in up\_remains[

target\_id]: # up

up\_remains[target\_id].append(outer\_task.target)

up\_remains[target\_id].sort()

# print(up\_remains)

# 设为等待态

outer\_task.state = OuterButtonState.waiting

elif elevator\_cur\_floor[target\_id] > outer\_task.target and outer\_task.target not in down\_remains[

target\_id]: # down

down\_remains[target\_id].append(outer\_task.target)

down\_remains[target\_id].sort(reverse=True) # 这里需要降序！ 例如，[20,19,..1]

# print(down\_targets)

# 设为等待态

outer\_task.state = OuterButtonState.waiting

# 查看哪些任务已经完成了 移除已经完成的

outer\_button\_events = [task for task in outer\_button\_events if task.state != OuterButtonState.finished]

mutex.unlock()

else:

distance = abs(origin - targets[-1]) + abs(outer\_task.target - targets[-1])

# 寻找最小值

if distance < min\_distance:

min\_distance = distance

target\_id = i

# 假如找到了 对应添加任务

if target\_id != -1:

if elevator\_cur\_floor[target\_id] == outer\_task.target:

if outer\_task.move\_state == MoveState.up and outer\_task.target not in up\_remains[target\_id] and elevator\_states[target\_id] != ElevatorState.going\_up:

up\_remains[target\_id].append(outer\_task.target)

up\_remains[target\_id].sort()

outer\_task.state = OuterButtonState.waiting

elif outer\_task.move\_state == MoveState.down and outer\_task.target not in down\_remains[target\_id] and elevator\_states[target\_id] != ElevatorState.going\_down:

down\_remains[target\_id].append(outer\_task.target)

down\_remains[target\_id].sort(reverse=True) # 这里需要降序！ 例如，[20,19,..1]

outer\_task.state = OuterButtonState.waiting

elif elevator\_cur\_floor[target\_id] < outer\_task.target and outer\_task.target not in up\_remains[target\_id]: # up

up\_remains[target\_id].append(outer\_task.target)

up\_remains[target\_id].sort()

outer\_task.state = OuterButtonState.waiting

elif elevator\_cur\_floor[target\_id] > outer\_task.target and outer\_task.target not in down\_remains[target\_id]: # down

down\_remains[target\_id].append(outer\_task.target)

down\_remains[target\_id].sort(reverse=True) # 这里需要降序！ 例如，[20,19,..1]

outer\_task.state = OuterButtonState.waiting

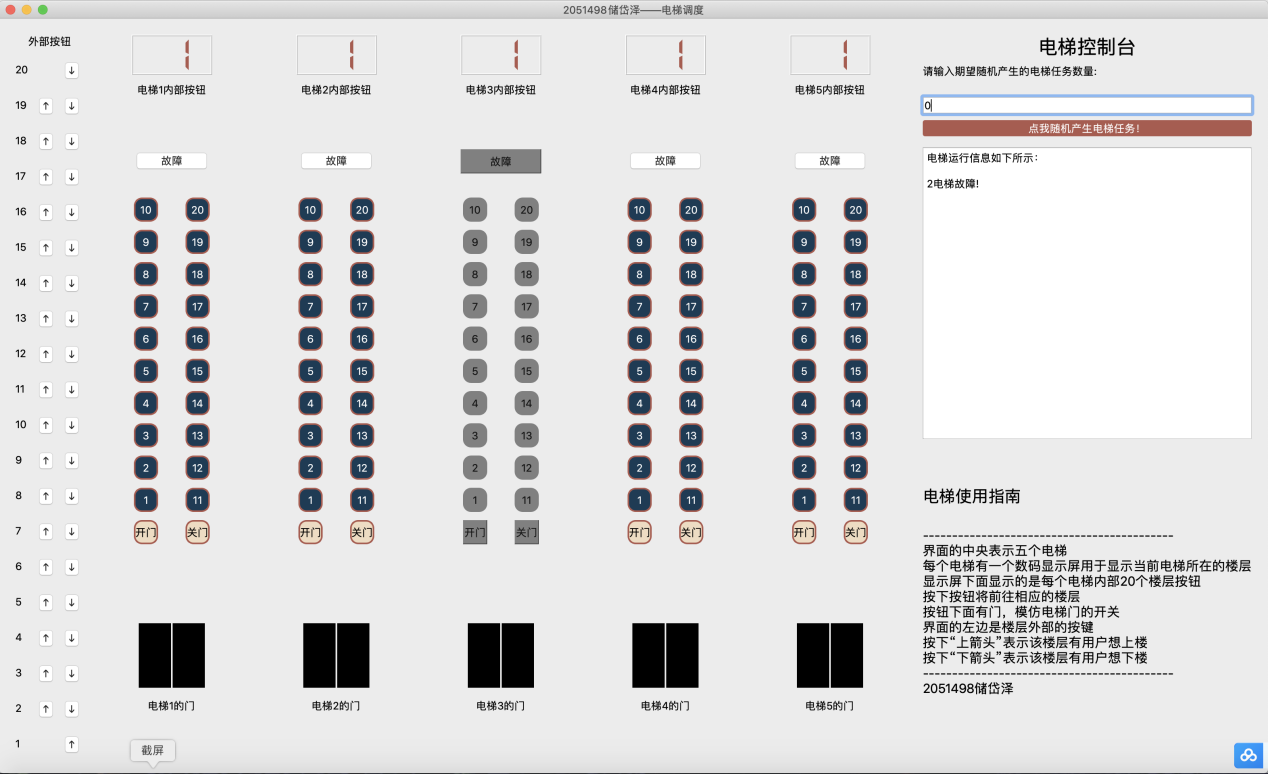
# 查看哪些任务已经完成了 移除已经完成的

outer\_button\_events = [task for task in outer\_button\_events if task.state != OuterButtonState.finished]

mutex.unlock()

### 运行截图

##### 电梯故障展示



##### **电梯开门展示**

