项目说明文档

操作系统

——进程管理之电梯调度

作 者 姓 名： 杨煜

学 号： 1850217

指 导 教 师： 张惠娟

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目背景 1](#_Toc40428823)

[1.1 项目简介 1](#_Toc40428824)

[1.2 项目目的 1](#_Toc40428825)

[1.3 项目要求 1](#_Toc40428826)

[2 需求分析 2](#_Toc40428827)

[2.1 上下行请求 2](#_Toc40428828)

[2.2 数字按钮请求 2](#_Toc40428829)

[2.3 开关门请求 2](#_Toc40428830)

[2.4 报警请求 2](#_Toc40428831)

[3 算法分析 2](#_Toc40428832)

[3.1 单部电梯 2](#_Toc40428833)

[3.1.1 FCFS算法 2](#_Toc40428834)

[3.1.2 SSTF算法 3](#_Toc40428835)

[3.1.3 SCAN算法 3](#_Toc40428836)

[3.2 多部电梯 3](#_Toc40428837)

[4 设计实现 4](#_Toc40428838)

[4.1 电梯类 4](#_Toc40428839)

[4.2 调度类 5](#_Toc40428840)

[4.2.1 初始化数据 5](#_Toc40428841)

[4.2.2 上行请求 6](#_Toc40428842)

[4.2.3 下行请求 8](#_Toc40428843)

[4.2.4 数字按钮请求 9](#_Toc40428844)

[4.2.5 上行后尚未处理的请求 12](#_Toc40428845)

[4.2.6 下行后尚未处理的请求 13](#_Toc40428846)

[4.2.7 开关门请求 14](#_Toc40428847)

[4.2.8 报警请求 15](#_Toc40428848)

[4.3 界面类 16](#_Toc40428849)

[5 附录 17](#_Toc40428850)

[5.1 文件说明 17](#_Toc40428851)

[5.2 开发环境 18](#_Toc40428852)

# 1 项目背景

## 1.1 项目简介

操作系统的进程的调度算法的思想在生活的方方面面都有体现。从磁盘寻道到电梯调度再到进程管理，这些看似不同的事物之间本质的算法思想却是相近的。本项目的目标是通过电梯调度算法对于理解进程管理更进一步。电梯是人们出行的重要工具之一，在人们日常生活中扮演着重要的角色。电梯调度算法是决定电梯运行效率的关键，也是可以用来熟悉操作系统中的进程调度算法的良好借鉴。本项目通过研究构建电梯调度算法及程序来达到熟练掌握进程调度思想。

## 1.2 项目目的

1、通过控制电梯调度，实现操作系统调度过程；

2、学习特定环境下多线程编程方法

3、学习调度算法。

## 1.3 项目要求

某一层楼20层，有五部互联的电梯。基于线程思想，编写一个电梯调度程序。

功能描述：

1、电梯应有一些按键，如：数字键、关门键、开门键、上行键、下行键、报警键等；

2、有数码显示器指示当前电梯状态；

3、每层楼、每部电梯门口，有上行、下行按钮、数码显示。

4、五部电梯相互联结，即当一个电梯按钮按下去时，其它电梯相应按钮同时点亮，表示也按下去了。

5、所有电梯初始状态都在第一层；每个电梯没有相应请求情况下，则应该在原地保持不动；

6、电梯调度算法自行设计。

# 2 需求分析

## 2.1 上下行请求

对于上下行请求，这是独立于电梯的外部的请求。每一楼层可以发出两种外部请求，即上下行的请求，这些请求对于五部电梯来说是相互联结的，而不是作用于每一部电梯。通过电梯调度的算法，在五部互相联结的电梯中，找到一部最合适的电梯将该请求加入其响应队列。按下的按钮直到被响应前一直保持按下状态，得到响应后恢复初始状态。

## 2.2 数字按钮请求

对于数字按钮请求，这是每一部电梯内部的请求。每一电梯内部有二十个数字按钮，分别对应着想要前往的楼层。这些请求对于五部电梯来说是相互孤立的，一旦出现了楼层请求，就将其加入响应队列。按下的按钮直到被响应前一直保持按下状态，得到响应后恢复初始状态。

## 2.3 开关门请求

对于开关门请求，这也是每一部电梯内部的请求。对于开关门请求，这也是每一部电梯内部的请求。对于开门按钮，当该电梯到达相应楼层后，会延长继续响应其他请求的等待时间。按下的按钮直到被响应前一直保持按下状态，响应结束后恢复初始状态。对于关门按钮，会将开门按钮所带来的响应延迟消除，并将按钮恢复初始状态。

## 2.4 报警请求

对于报警请求，这也是每一部电梯内部的请求。当按下时，报警键会常亮，并进行相应的操作，直到再次按下按钮时。恢复其初始状态。

# 3 算法分析

## 3.1 单部电梯

### 3.1.1 FCFS算法

先来先服务（FCFS-First Come First Serve）算法，是一种随即服务算法，它不仅仅没有对寻找楼层进行优化，也没有实时性的特征，它是一种最简单的电梯调度算法。它根据乘客请求乘坐电梯的先后次序进行调度。此算法的优点是公平、简单，且每个乘客的请求都能依次地得到处理，不会出现某一乘客的请求长期得不到满足的情况。这种方法在载荷较轻松的环境下，性能尚可接受，但是在载荷较大的情况下，这种算法的性能就会严重下降，甚至恶化。该算法可以作为衡量其他算法的标准。

### 3.1.2 SSTF算法

最短寻找楼层时间优先（SSTF-Shortest Seek Time First）算法，它注重电梯寻找楼层的优化。最短寻找楼层时间优先算法选择下一个服务对象的原则是最短寻找楼层的时间。这样请求队列中距当前能够最先到达的楼层的请求信号就是下一个服务对象。最短寻找楼层时间优先算法的平均响应时间较短，但响应时间的方差较大，原因是队列中的某些请求可能长时间得不到响应，出现所谓的“饥饿”现象。

### 3.1.3 SCAN算法

扫描算法（SCAN） 是一种按照楼层顺序依次服务请求，它让电梯在最底层和最顶层之间连续往返运行，在运行过程中响应处在于电梯运行方向相同的各楼层上的请求它进行寻找楼层的优化，效率比较高，它是一个非实时算法。扫描算法较好地解决了电梯移动的问题，在这个算法中，每个电梯响应乘客请求使乘客获得服务的次序是由其发出请求的乘客的位置与当前电梯位置之间的距离来决定的。所有的与电梯运行方向相同的乘客的请求在一次电向上运行或向下运行的过程中完成，免去了电梯频繁的来回移动。扫描算法的平均响应时间比最短寻找楼层时间优先算法长，但是响应时间方差比最短寻找楼层时间优先算法小。

## 3.2 多部电梯

本项目采用的多部电梯算法，基于单部电梯的SCAN算法。对于每部电梯，为其设计一对上下行序列数组，其中上行序列数组正序排列，下行序列数组倒序排列，电梯根据目前状态以及两数组中内容执行动作。当某楼层有上行或下行请求发出时，先考虑每部电梯在完成当前序列的过程中能否响应该请求，计算出符合此条件的电梯的响应距离，再考虑剩余电梯从其当前位置到序列终点与终点到该请求位置的响应距离之和，最后比较每部电梯的响应距离，将该请求分配给具有最短响应距离一部电梯。

# 4 设计实现

## 4.1 电梯类

仿照操作系统进程管理的进程控制块（PCB）,建立存储电梯状态信息的类。该类用于记录电梯运行时的各种状态。其中PID标记是哪一部电梯。电梯状态有三种，上行，下行以及静止。该类同样存储了电梯当前的位置，电梯是否被开关门，电梯的上下行请求序列，电梯尚未处理的请求序列等。

class PCB():

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

self.PID=PID

self.state=0 #当前状态，上行，静止或下行

self.last\_state=0#前一刻状态，上行，静止或下行

self.location=1#当前电梯所在的位置

self.downfloor=[0]\*42#记录下还没有来得及处理的下行请求

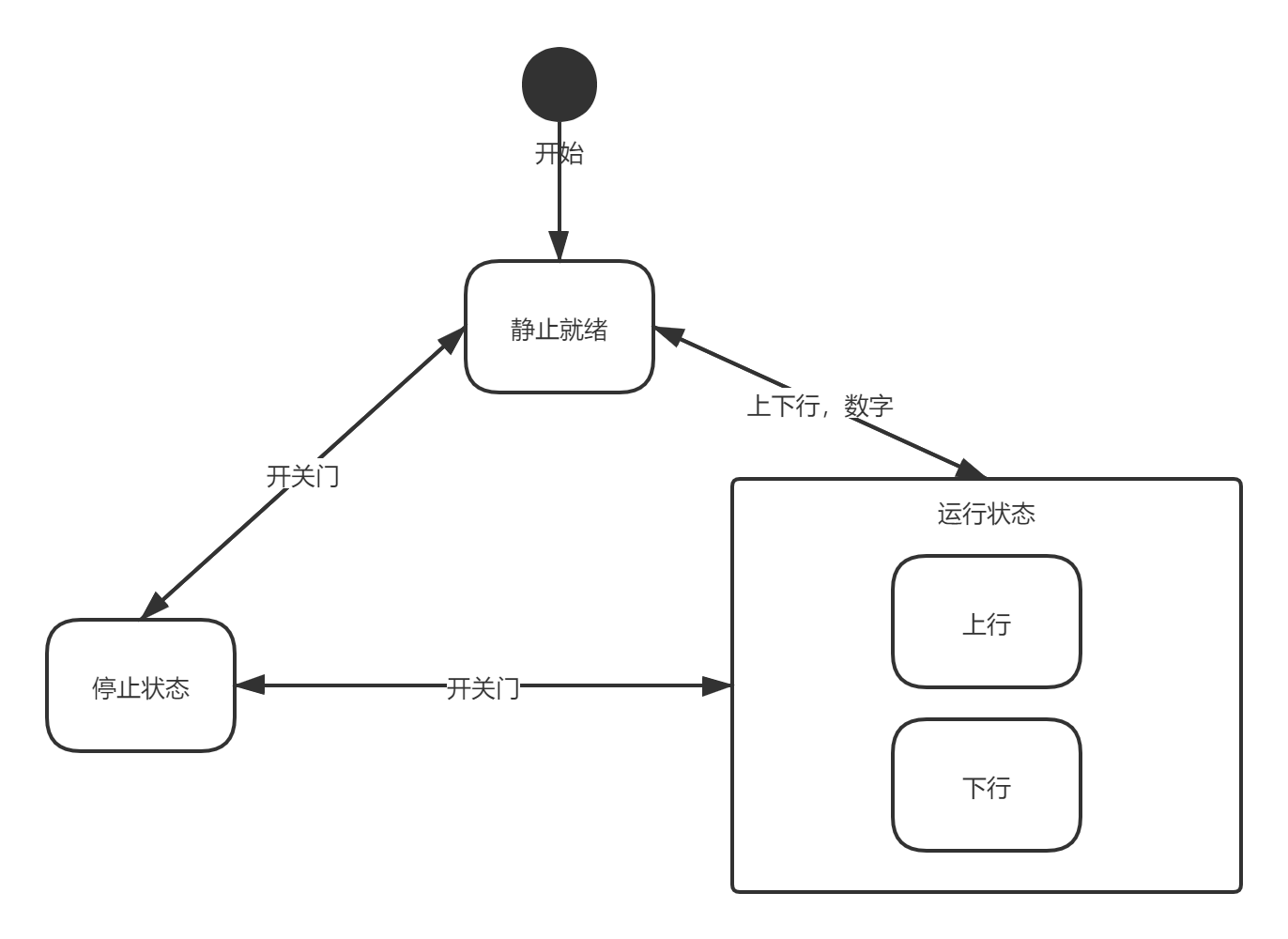
self.upfloor=[0]\*42#记录下还没有来得及处理的上行请求

self.upSequence=[]#上行请求的序列

self.downSequence=[]#下行请求的序列

self.isPause=0#判断开关门按钮是否被按下

self.alert=0#判断警报



## 4.2 调度类

### 4.2.1 初始化数据

对数据进行初始化操作，初始化界面类。产生五部电梯的电梯类。将监听函数与按钮进行绑定。

class elevator\_schedul():

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

self.ui=ui

self.elevator\_PCB=[]

for i in range(1, 6):#产生5部电梯

self.elevator\_PCB.append(PCB(i))

for i in range(1, 20): # 上行按钮与监听函数绑定

self.ui.up\_btn[i].clicked.connect(partial(self.upbutton\_listen, i))

for i in range(2, 21): # 下行按钮与监听函数绑定

self.ui.down\_btn[i].clicked.connect(partial(self.downbutton\_listen, i))

for i in range(1, 6):

for j in range(1, 21): # 数字按钮与监听函数绑定

self.ui.number\_btn[i][j].clicked.connect(partial(self.numberbtn\_listen, i, j))

for i in range(1, 6):# 打开按钮与监听函数绑定

self.ui.open\_btn[i].clicked.connect(partial(self.openbtn\_listen, i))

for i in range(1, 6):# 关闭按钮与监听函数绑定

self.ui.close\_btn[i].clicked.connect(partial(self.closebtn\_listen, i))

for i in range(1, 6):# 警报按钮与监听函数绑定

self.ui.alert\_btn[i].clicked.connect(partial(self.alertbtn\_listen, i))

for i in range(1, 6): # 加载线程

self.thread(i)

### 4.2.2 上行请求

对于上行请求来说，首先记录五部电梯的运行状态。随后根据电梯运行状态的不同来确定到达请求位置的距离。如果电梯处于上行状态，且请求的位置在上方或者本层。则距离记为目标位置减去当前位置。否则记为当前位置的终点距离加终点距离位置到目标位置。如果电梯处于静止状态，则是目标位置减去当前位置。如果电梯处于下行状态，记为当前位置的终点距离加终点距离位置到目标位置。随后找出最短距离的电梯。如果电梯处于上行状态，请求在上方或本层，则将其加入上行队列，否则加入还未处理的上行请求。静止状态同样如此。对于下行状态，直接记录为还未处理的上行请求。

def upbutton\_listen(self, btn\_number):

self.ui.up\_btn[btn\_number].setStyleSheet("QPushButton{border-image: url(icon/up\_pressed.png)}")#设置图标

upbtn\_distance = [100, 100, 100, 100, 100, 100] # 该列表用于记录某楼层发出上行请求时,五部电梯到达楼层所要走的距离，初始化为一个足够大的数

requestIsUp = {} # 记录上行请求是否在电梯上方或本层

for i in range(1, 6):

if (btn\_number - self.elevator\_PCB[i-1].location) >= 0:

requestIsUp[i] = True

else:

requestIsUp[i] = False

for i in range(1, 6):

if self.elevator\_PCB[i-1].state == 1: # 上行状态

if requestIsUp[i]: # 请求在上方或本层

upbtn\_distance[i] = abs(btn\_number - self.elevator\_PCB[i-1].location) # 当前位置距请求位置距离

else: # 请求在下方

upbtn\_distance[i] = abs(self.elevator\_PCB[i-1].location - self.elevator\_PCB[i-1].upSequence[len(self.elevator\_PCB[i-1].upSequence) - 1]) \

+ abs(btn\_number - self.elevator\_PCB[i-1].upSequence[len(self.elevator\_PCB[i-1].upSequence) - 1]) # 当前位置距终点距离 + 终点距请求位置距离

elif self.elevator\_PCB[i-1].state == 0: # 静止状态

upbtn\_distance[i] = abs(btn\_number - self.elevator\_PCB[i-1].location) # 当前位置距请求位置距离

elif self.elevator\_PCB[i-1].state == -1: # 下行状态

upbtn\_distance[i] = abs(self.elevator\_PCB[i-1].location - self.elevator\_PCB[i-1].downSequence[len(self.elevator\_PCB[i-1].downSequence) - 1]) \

+ abs(btn\_number - self.elevator\_PCB[i-1].downSequence[len(self.elevator\_PCB[i-1].downSequence) - 1]) # 当前位置距终点距离 + 终点距请求位置距离

elevator\_number = upbtn\_distance.index(min(upbtn\_distance)) # 记录距离最短的电梯

if self.elevator\_PCB[elevator\_number-1].state == 1: # 上行状态

if requestIsUp[elevator\_number]: # 请求在上方或本层

self.append\_upSequence(elevator\_number,btn\_number)

else: # 请求在下方

self.elevator\_PCB[elevator\_number-1].upfloor[btn\_number] = 1 # 记录电梯将处理但未处理的楼层上行请求

elif self.elevator\_PCB[elevator\_number-1].state == 0: # 静止状态

if requestIsUp[elevator\_number]: # 请求在上方或本层

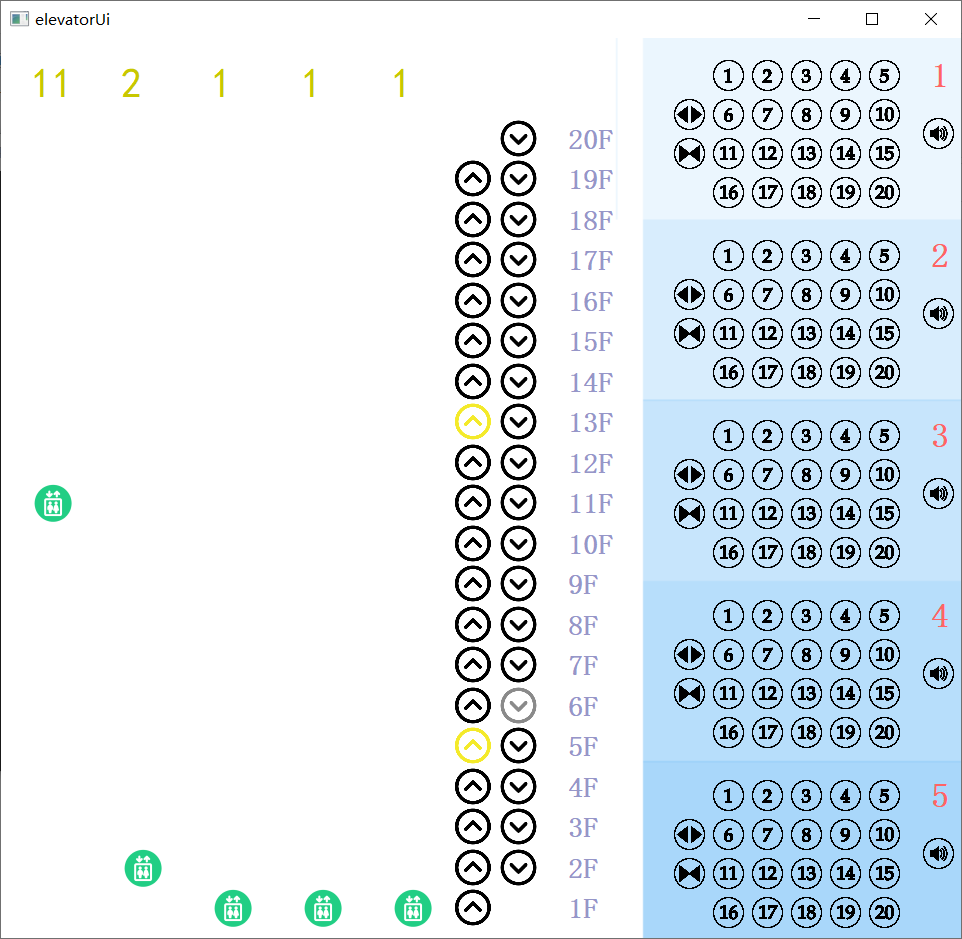
self.append\_upSequence(elevator\_number,btn\_number)

else: # 请求在下方

self.elevator\_PCB[elevator\_number-1].upfloor[btn\_number] = 1 # 记录电梯将处理但未处理的楼层上行请求

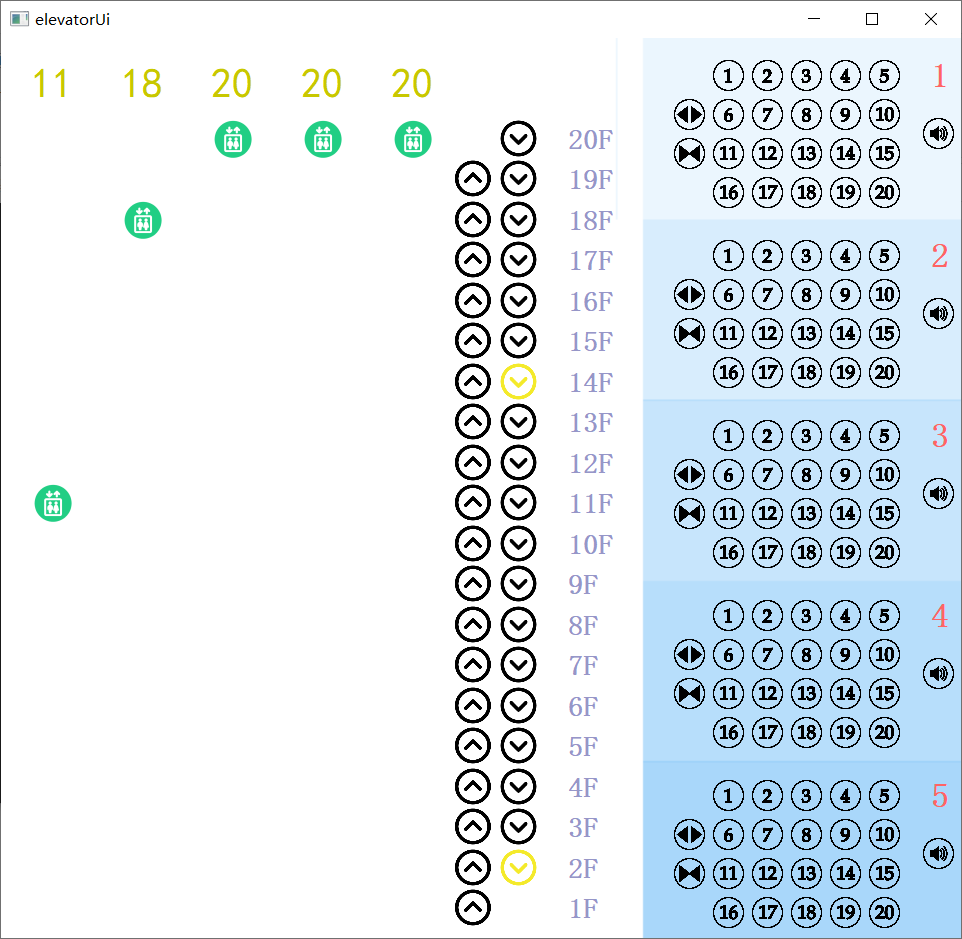
elif self.elevator\_PCB[elevator\_number-1].state == -1: # 下行状态

self.elevator\_PCB[elevator\_number-1].upfloor[btn\_number] = 1 # 记录电梯将处理但未处理的楼层上行请求



### 4.2.3 下行请求

下行请求效果同上，实现方式和上行请求相似，故不再赘述。



### 4.2.4 数字按钮请求

对于数字按钮请求。首先会确定是哪一部电梯所发出的请求。因为数字请求是孤立的，互不影响的。如果当前电梯处于静止状态，请求如果在电梯上方则将数字加入上行队列，否则加入下行队列。如果电梯处于上行状态，请求位置在电梯之上，则将其加入上行队列，否则加入还未处理的下行请求。同理，对于下行状态相似的处理。

def numberbtn\_listen(self, elevator\_number, btn\_number):

if self.elevator\_PCB[elevator\_number-1].state == 0: # 电梯处于静止状态

if self.elevator\_PCB[elevator\_number-1].location > btn\_number: # 电梯当前位置在请求楼层之上

self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.append\_downSequence(elevator\_number,btn\_number)

if self.elevator\_PCB[elevator\_number-1].location < btn\_number: # 电梯当前位置在请求楼层之下

self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.append\_upSequence(elevator\_number,btn\_number)

elif self.elevator\_PCB[elevator\_number-1].state == 1: # 电梯处于上行状态

if self.elevator\_PCB[elevator\_number-1].location < btn\_number: # 电梯当前位置在请求楼层之下

self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.append\_upSequence(elevator\_number,btn\_number)

else:

self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.elevator\_PCB[elevator\_number-1].downfloor[btn\_number] = 1 #记录电梯将处理但未处理的楼层下行请求

elif self.elevator\_PCB[elevator\_number-1].state == -1: # 电梯处于下行状态

if self.elevator\_PCB[elevator\_number-1].location > btn\_number: # 电梯当前位置在请求楼层之上

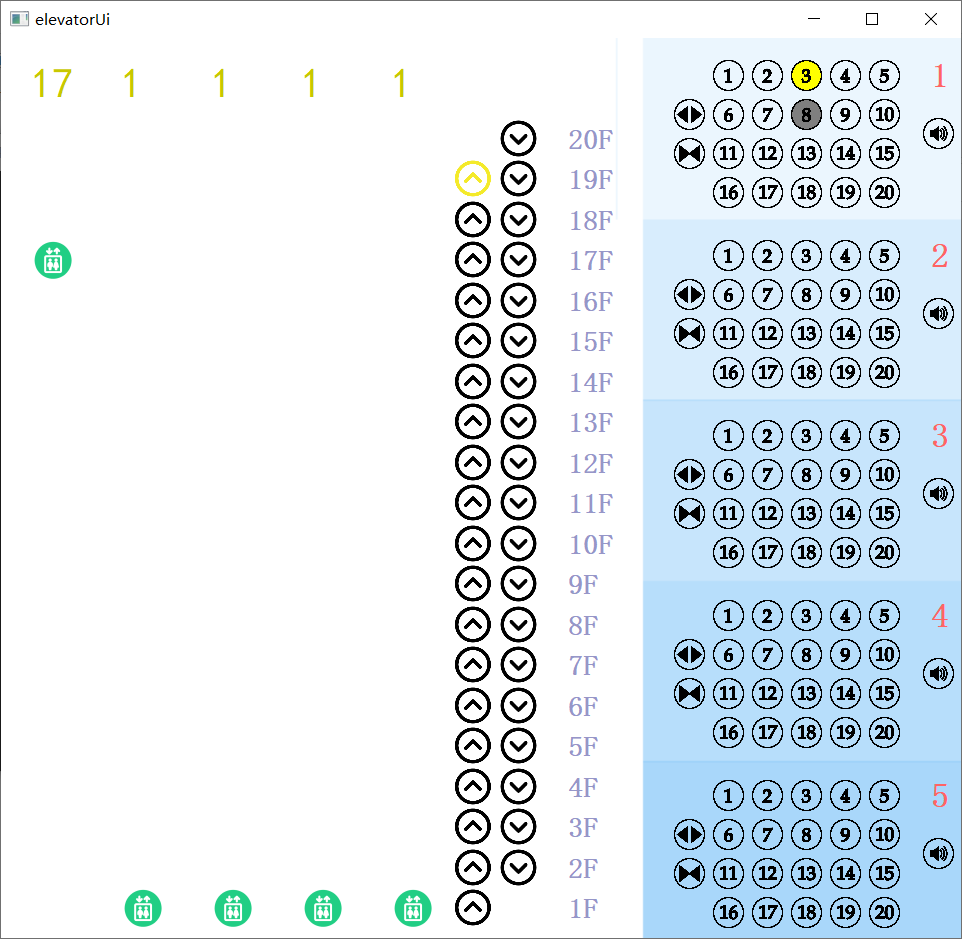
self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.append\_downSequence(elevator\_number,btn\_number)

else:

self.ui.number\_btn[elevator\_number][btn\_number].setStyleSheet("QPushButton{border-image: url(icon/" + str(btn\_number) + "\_pressed.png)}")

self.elevator\_PCB[elevator\_number-1].upfloor[btn\_number] = 1 #记录电梯将处理但未处理的楼层上行请求



### 4.2.5 上行后尚未处理的请求

上行动作结束后，对于尚未处理的请求进行相应。首先倒序处理下行请求，如果请求在上方则加入上行队列并实行动画。否则加入下行队列。随后正序处理上行请求。将最下方的上行请求取出并放置如下行队列。

# 执行完上行动作恢复静止后，处理执行动作时产生的但未处理的请求

def elevator\_finish\_up(self, elevator\_number):

i = 20

while i >= 1: # 倒序处理执行动作时产生的但未处理的下行请求

if self.elevator\_PCB[elevator\_number-1].downfloor[i] == 1:

if i > self.elevator\_PCB[elevator\_number-1].location: # 上方存在执行动作时产生的但未处理的下行请求

self.elevator\_PCB[elevator\_number-1].upSequence.append(i) # 将最高楼层的下行请求加入上行序列,继续上行

self.elevator\_PCB[elevator\_number-1].upSequence = list(set(self.elevator\_PCB[elevator\_number-1].upSequence))

self.elevator\_PCB[elevator\_number-1].state = 1

break

# 上方不存在执行动作时产生的但未处理的下行请求

self.elevator\_PCB[elevator\_number-1].downfloor[i] = 0

self.append\_downSequence(elevator\_number,i)

self.elevator\_PCB[elevator\_number-1].state = -1

i = i - 1

# 不存在下行请求，处理执行动作时产生的但未处理的上行请求（该请求只可能在下方）

if self.elevator\_PCB[elevator\_number-1].state == 0:

for i in range(1, 21): # 正序处理

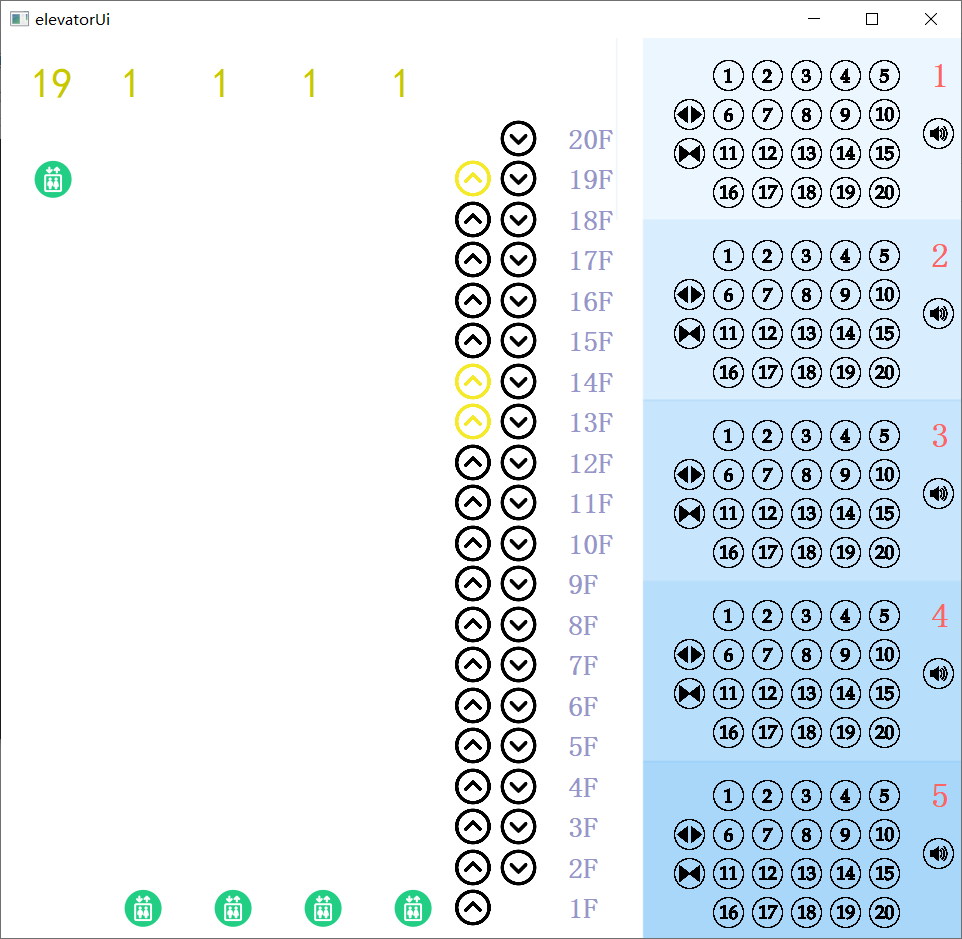
if self.elevator\_PCB[elevator\_number-1].upfloor[i] == 1:

self.elevator\_PCB[elevator\_number-1].downSequence.append(i) # 将最底楼层的下行请求加入下行序列,开始下行

self.elevator\_PCB[elevator\_number-1].downSequence = list(set(self.elevator\_PCB[elevator\_number-1].downSequence))

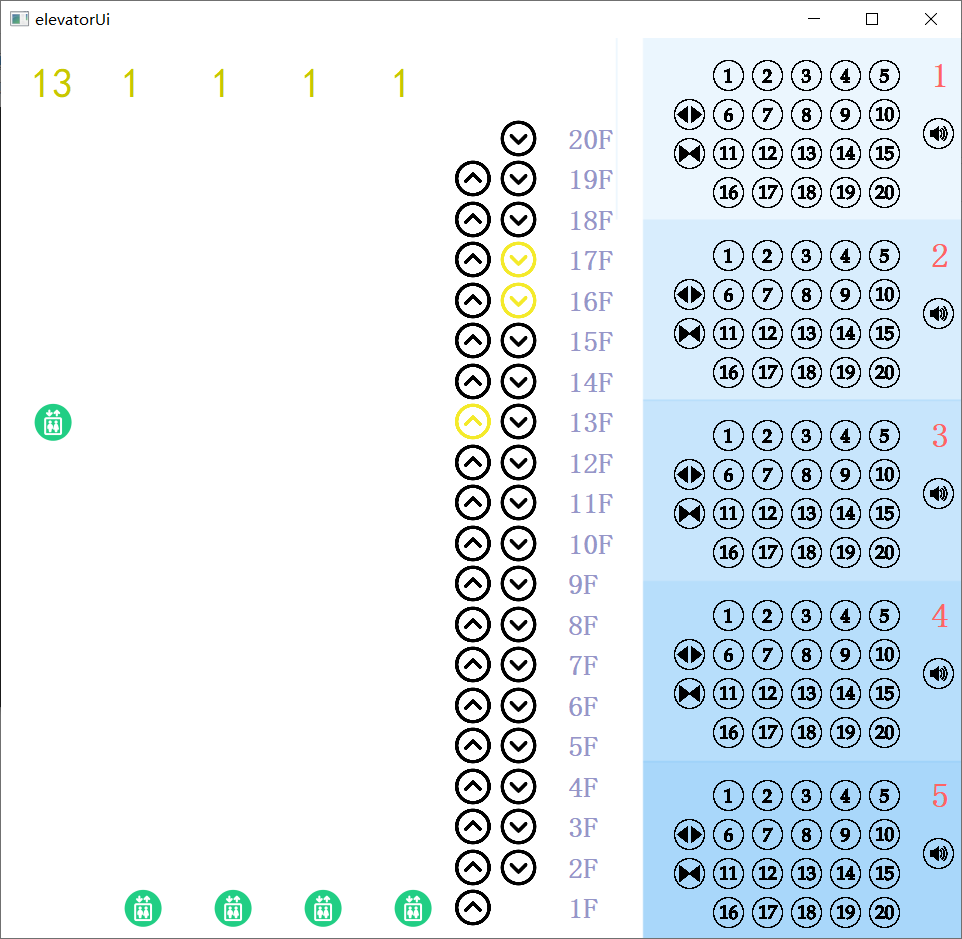
self.elevator\_PCB[elevator\_number-1].state = -1

break



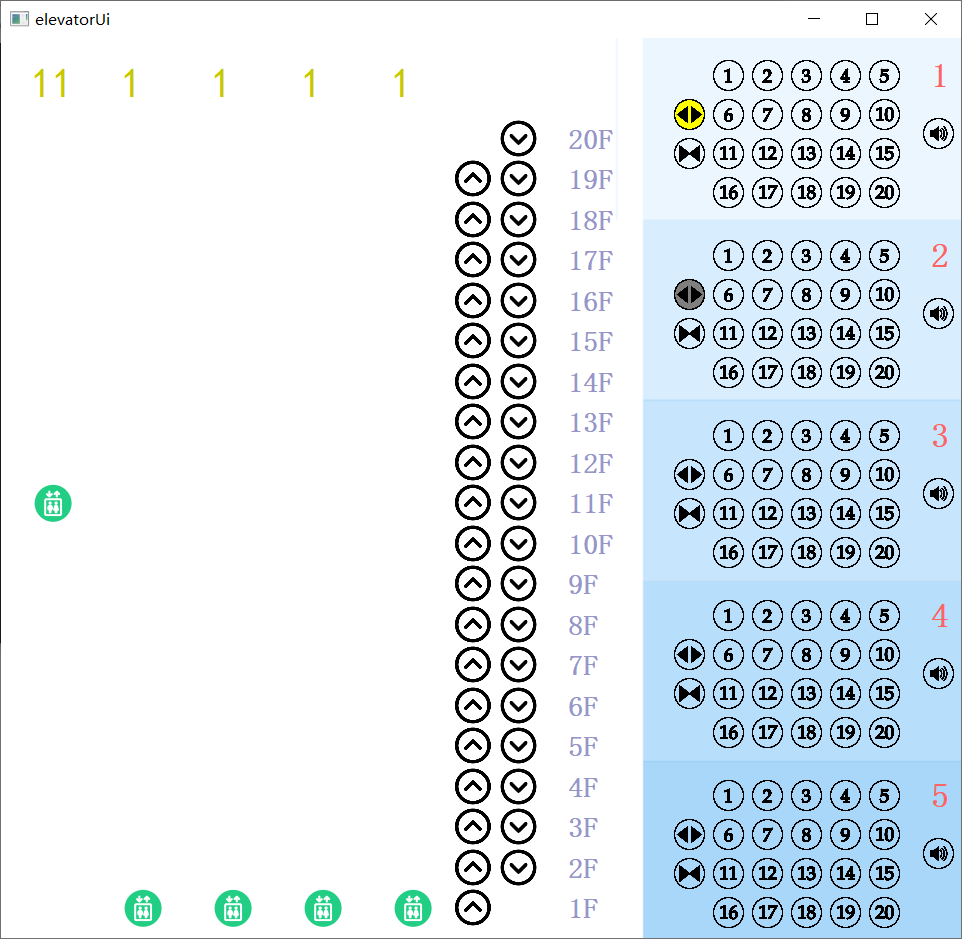
### 4.2.6 下行后尚未处理的请求

实现方案和上行后相似，故不在此赘述。



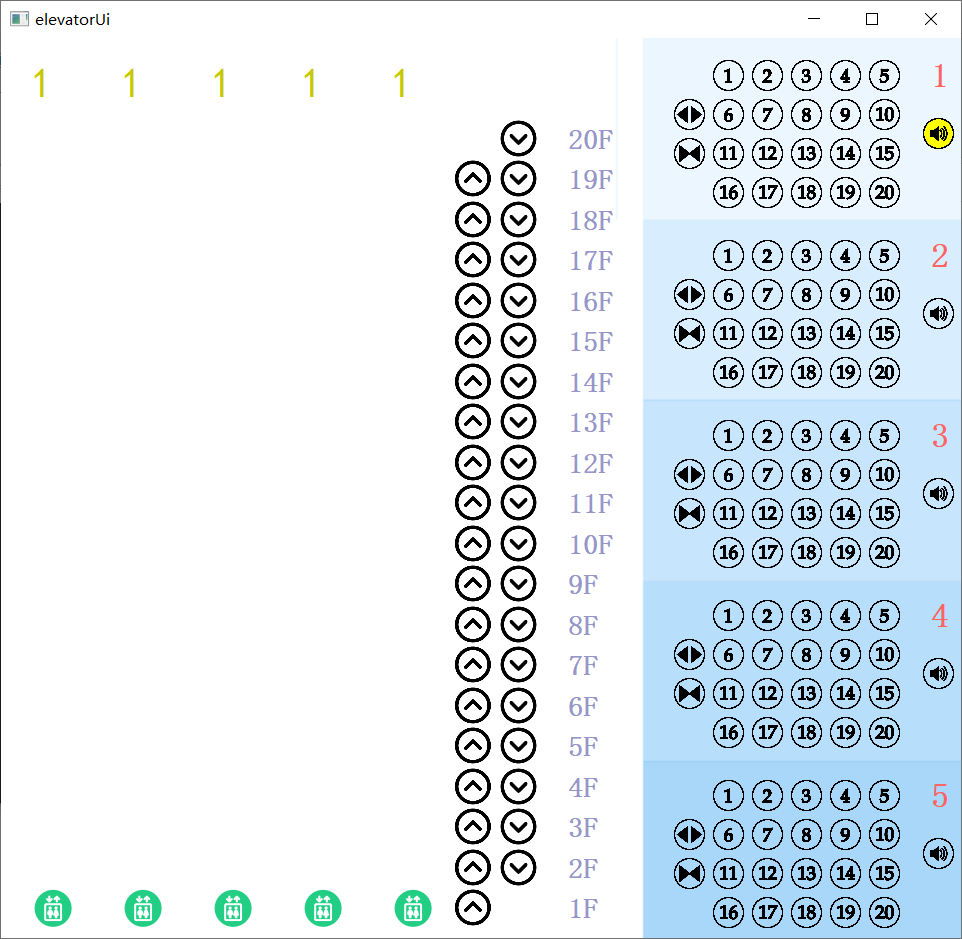
### 4.2.7 开关门请求

对于开门，将会设置是否暂停变量为1。对于关门则会将之设置为0，用于控制电梯是否暂停。



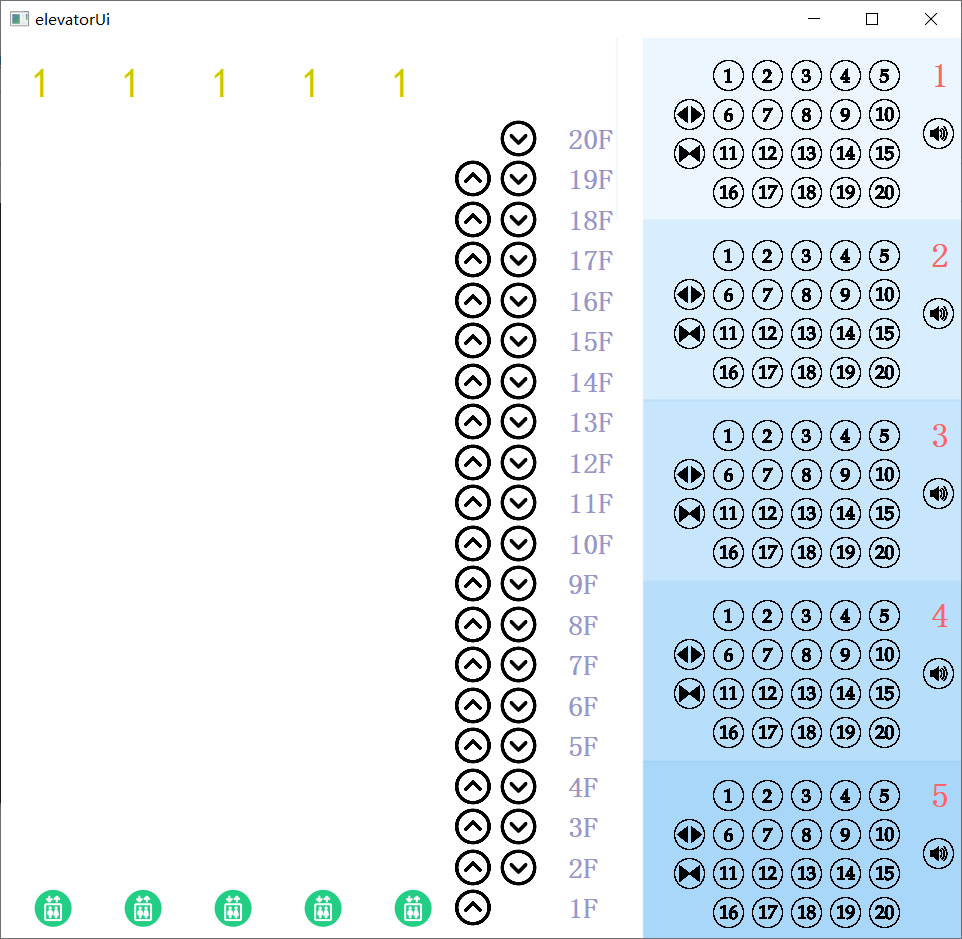
### 4.2.8 报警请求

对于报警，将会设置是否报警变量为1。再按一下，则会设置为0，取消报警。



## 4.3 界面类

通过PyQt5使得电梯调度算法的可视化。构建了相应的调度界面。左边为电梯运行的可视化，每一电梯的上方有显示楼层的数字。中间部分为上下行按钮。右侧为电梯内部按钮。每一电梯都有一排数字按钮。



# 5 附录

## 5.1 文件说明

eleavtorSchedule.py为调度算法的核心代码

elevatorUi.py为调度算法的界面代码

elevatorUi.exe为生成的可执行文件

icon文件夹内部存放了运行时所需要的素材

由于python环境问题，exe文件运行并不稳定（使用pyinstaller生成），所以使用时建议执行elevatorUi.py，所需环境为：PyQt5

## 5.2 开发环境

操作系统：windows10 1909 64bit

语言：python 3.7

相关python包：PyQt5