电梯调度设计方案

处理机管理之项目⼀

|  |  |
| --- | --- |
| 姓名 | 学号 |
| 王蔚达 | 2151300 |

[⼀、项目简介](#_bookmark1)

[二、系统架构](#_bookmark2)

[三、核心算法](#_bookmark3)

[四、线程同步与互斥](#_bookmark4)

[4.1 项目中用到的全局常量与变量](#_bookmark5)

[4.2 线程同步的关键部分](#_bookmark6)

[4.3 使用锁时的注意事项](#_bookmark7)

[4.4 多线程编程技术在本题中的体现](#_bookmark8)

[五、用户界面设计](#_bookmark9)

[5.1 初始状态](#_bookmark10)

[5.2 运行状态](#_bookmark11)

[5.3 报警状态](#_bookmark12)

[六、心得体会](#_bookmark13)

[6.1 设计过程中遇到的问题及解决办法](#_bookmark14)

[6.2 经验](#_bookmark15)

⼀ 、项目简介

1. 项目内容 ：本项目旨在实现⼀个电梯调度系统 ，模拟调度⼀栋20层楼的5部电梯 ，能够根据用户 的上行和下行需求自动分配电梯并控制电梯运行。该系统采用多线程技术 ，有效地提高了程序的 运行效率。系统通过图形用户界面(GUI)展示电梯的实时状态 ，并允许用户通过点击楼层按钮来 发出上行和下行请求。

2. 开发环境

 系统 ：Windows 11 家庭中文版

 IDE ：PyCharm Community Edition 2022.2

 Python 解释器 ：Python版本为3.8.16 ，且已通过pip安装了PyQt5

二、系统架构

根据任务需求 ，将电梯调度程序分为以下几个模块 (类) ：

1. 主程序入口 ：负责初始化变量、创建对象和启动线程。

2. 图形用户界面 ( MainWindow 类) ：展示电梯的实时状态 ，并允许用户通过点击电梯内外部楼层按 钮来发出上行和下行请求 ，同时可以通过警报键来紧急制动电梯。 同时 ，该类每隔⼀段时间会刷 新⼀次界面 ，实现界面的实时更新。

在开发时 ，在QtDesigner上进行设计 ，之后使用PyUIC导出至ui\_mainwindow中 ，并在该类中直 接调用。

3. 电梯线程 ( Elevator 类) ：负责根据任务队列控制电梯的运行状态 (上行、下行或停止) 以及开

关门操作。

4. 外部任务处理线程 ( Outer 类) ：负责接收和处理用户的上行和下行请求 ，通 过寻找距离最近的

电梯来分配任务。

|  |  |
| --- | --- |
| 类名 | 作用 |
| MainWindow | 继承自QtWidgets.QMainWindow ，并实现了⼀个电梯调度系统的用户界面。这个界面包含了 电梯内部按钮、 电梯楼层显示、 电梯运行状态指示、外部上下楼请求按钮以及电梯报警按钮 等组件。 |
| Elevator | 继承自**QThread** ，使得电梯可以作为⼀个独立的线程运行。类中包含了多个方法 ，分别用于 处理电梯的运行状态、楼层移动、 门的开关以及故障处理。 |
| Outer | 继承自 **QThread** ，是⼀个外部任务处理线程。这个类主要负责处理电梯系统中外部任务 (即 用户在楼层按上行或下行按钮时产生的任务) ，并将任务分配给最合适的电梯。 |

三、核心算法

系统的核心算法为寻找距离最近的电梯 (调度算法) ，其思路如下 ：

为了寻找距离最近的电梯 ，我们需要考虑电梯的运行状态、 当前楼层和任务队列。综合FCFS (先来 先服务算法) 算法与LOOK算法进行设计 ，其具体算法如下 ：

1. 遍历所有电梯 ，跳过故障状态的电梯。

2. 根据电梯运行状态和任务队列计算到目标楼层的距离 ：

a. 如果电梯没有任务 ，直接计算当前楼层与目标楼层的绝对值。

b. 如果电梯运行方向与任务需求相同 ，且任务楼层在电梯运行方向上 ，计算当前楼层与目标楼 层的绝对值。

c. 其他情况下 ，计算当前楼层与最远任务楼层的绝对值 ，再加上目标楼层与最远任务楼层的绝 对值。

c. 选择距离最小的电梯作为目标电梯。

该算法优先考虑运行方向与任务需求相同的电梯 ，并在多个电梯之间选择距离最近的电梯以减少等待 时间。具体代码如下 ：

def find\_best\_elevator(outer\_task) :

min\_distance = FLOOR\_NUM + 1

# 初始化分配电梯

target\_id = -1

# 依次访问每⼀个电梯

for i in range(ELEVATOR\_NUM) :

# 如果电梯处于故障状态 ，则跳过它

if elevator\_states[i] == ELEVATOR\_STATE .FAULT :

continue

# 如果已经上行/下行了 ，则上/下移动⼀层

origin = elevator\_cur\_floor[i]

if elevator\_states[i] == ELEVATOR\_STATE .UP :

origin += 1

elif elevator\_states[i] == ELEVATOR\_STATE .DOWN :

origin -= 1

if elevator\_move\_states[i] == MOVE\_STATE .UP :

targets = elevator\_up\_target\_list[i]

else : # down

targets = elevator\_down\_target\_list[i]

# 根据到outer\_task的距离计算优先级

# 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 .floor)

# 若电梯朝着按键所在楼层运行 ，且运动方向与外部请求相同

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

(outer\_task .move\_state == MOVE\_STATE .UP and outer\_task .floor >= origin) or

(outer\_task .move\_state == MOVE\_STATE .DOWN and outer\_task .floor <= origin)) :

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

# 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 .floor - targets[-1])

# 寻找最小值

if distance < min\_distance :

min\_distance = distance

target\_id = i

return target\_id

此外 ，**elevator\_up\_target\_list** 与 **elevator\_down\_target\_list** 是两个优先级队列 ，分别为升序和降 序排列 ，在访问这两个二维队列时 ，通过pop便可以得知下⼀任务的位置。

四、线程同步与互斥

在本项目中 ，使用了多个线程来处理不同任务 ，例如外部任务处理线程 (Outer) 和电梯运行线程 ( Elevator) 。在这种情况下 ，线程同步就显得尤为重要 ，以防止数据竞争和保持系统的正确性。我 们主要通过使用互斥锁 (mutex) 来实现线程同步。

互斥锁是⼀种同步原语 ，用于解决多线程中的临界区问题。 当⼀个线程拥有互斥锁时 ，其他线程必须 等待 ，直到锁被释放。通过使用互斥锁 ，我们可以确保同⼀时间只有⼀个线程能够访问和修改共享数 据。

**4.1** 项目中用到的全局常量与变量

全局常量

# 基本参数

ELEVATOR\_NUM = 5 # 电梯数量

FLOOR\_NUM = 20 # 电梯层数

MOVE\_TIME = 1000 # 上升下降时间

DOOR\_OPEN\_AND\_CLOSE\_TIME = 2000 # 电梯开关门时间

# 颜色设置

BUTTON\_COLOR = (255, 255, 255) # 按钮未被按下的颜色

BUTTON\_CLICKED\_COLOR = (255, 255, 0) # 按钮按下的颜色(黄色)

ELEVATOR\_COLOR = (127, 255, 170) # 电梯运行中的颜色(绿色)

DOOR\_OPERATION\_COLOR = (255, 255, 0) # 电梯开关门时的中间色

WARNING\_BUTTON\_COLOR = (250, 128, 114) # 报警按钮颜色(粉红色)

全局变量

|  |  |
| --- | --- |
| 变量名 | 作用 |
| **elevator\_up\_target\_list** | 每台电梯需要向上运行处理的目标楼层列表 |
| **elevator\_down\_target\_list** | 每台电梯需要向下运行处理的目标楼层列表 |
| **elevator\_cur\_floor** | 每台电梯的当前楼层 |
| **elevator\_states** | 每台电梯的状态 |

|  |  |
| --- | --- |
| 变量名 | 作用 |
| **outer\_tasks\_list** | 用户产生的外部任务列表 |

枚举类

此外 ，为了使得代码的可读性更强 ，使用ELEVATOR\_STATE、 MOVE\_STATE、TASK\_STATE三个 枚举类分别表示电梯状态、 电梯移动状态和任务状态。

**4.2** 线程同步的关键部分

以下是涉及线程同步的关键部分 ：

1. **Outer** 线程

Outer 线程负责处理外部任务 ，如用户在楼层按下上行或下行按钮。在这个线程中 ，需要访问和 修改以下共享数据 ：outer\_tasks\_list、elevator\_states、elevator\_cur\_floor、 elevator\_up\_target\_list、elevator\_down\_target\_list

在访问和修改这些共享数据时 ，需要使用互斥锁来确保线程同步。具体实现如下 ：

def run(self) :

while True :

# 加锁

mutex .lock()

# 访问和修改共享数据的代码

# 解锁

mutex .unlock()

在这个示例中 ， 当线程进入 **run** 方法时 ，首先锁定互斥锁。在互斥锁被锁定期间 ，其他试图访 问共享资源的线程将被阻塞。接下来 ，线程执行⼀系列操作 ，如找到距离最短的电梯编号、将任 务加入队列等。在完成这些操作之后 ，线程解锁互斥锁 ，允许其他线程访问共享资源。

2. **Elevator** 线程

Elevator 线程负责控制每台电梯的运行。在这个线程中 ，同样需要访问和修改共享数据 ，如 **elevator\_states**、**elevator\_cur\_floor**、**elevator\_up\_target\_list**、**elevator\_down\_target\_list** 等。

其使用方法与Outer类似。

3. **MainWindow.update** 中

在UI界面上要实现实时更新 ，需要通过互斥锁实现了对共享资源的同步访问。在对电梯状态、楼 层信息等共享数据进行读取和更新时 ，我们都会先获取互斥锁 ，从而确保数据的准确性。

**4.3** 使用锁时的注意事项

此外 ，在本项目的其他函数中也有用到mutex ，下面对于其使用的注意事项进行总结如下 ：

 避免死锁 ：死锁是指两个或多个线程在等待对方释放资源的情况。在使用互斥锁时 ，需要确保在

适当的时机解锁互斥锁 ，防止死锁的发生。例如 ，本项目中 ，我们在执行完共享资源操作后立即 解锁互斥锁 ，以避免死锁。

 尽量减小锁的粒度 ：为了提高系统的并发性能 ，应尽量减小锁的粒度 ，即尽量缩小需要互斥访问

的资源范围。在本项目中 ，我们尽量在访问共享资源时使用互斥锁 ，而在执行其他操作时不使用 互斥锁。

 注意锁的使用范围 ：在使用互斥锁时 ，需要注意确保所有需要互斥访问的资源都在锁的范围内。

**4.4** 多线程编程技术在本题中的体现

1. 使用了**QThread**类 ，用于创建线程 ，这使得多个线程可以同时运行 ，提高了程序的并发性。在这 个程序中 ，创建了⼀个用于处理外部任务的线程**Outer** ，和多个用于模拟电梯内部运行的线程 **Elevator**。

2. 使用了互斥锁**QMutex**来确保数据在多线程环境中的安全访问。这个程序中的全局变量可能被多 个线程同时访问 ，例如**elevator\_states**、**elevator\_cur\_floor**、**elevator\_up\_target\_list**等 ，通 过互斥锁可以确保在同⼀时刻只有⼀个线程可以访问这些资源 ，避免数据竞争。

3. 使用了**lock()**和**unlock()**方法来对互斥锁进行加锁和解锁。在涉及访问共享资源的关键部分 ，先 调用**lock()**加锁 ，然后在访问完成后调用**unlock()**解锁。这样可以确保在操作共享资源时不会发 生数据冲突。

4. 程序中实现了多个线程间的协作。例如 ，**Outer**线程负责处理外部任务 ，将任务分配给合适的电 梯线程 ；**Elevator**线程则负责根据任务执行电梯的运动。在这个过程中 ，线程之间需要共享和访 问全局变量 ，通过互斥锁来实现数据安全访问。

五、用户界面设计

用户界面采用 Qt 库实现 ，界面设计简洁直观 ，包括以下部分 ：

1. 电梯实时状态显示区 ：在主窗⼝中展示每个电梯的实时状态 ，包括 ：  当前楼层 ：用数字表示 ，方便用户直观地了解电梯所在楼层。

 运行方向 ：用箭头表示 ，指示电梯当前的运行方向 (上行、下行或停止) 。

 电梯的状态 ：用颜色表示 ，配有文字 ，如绿色正在运行中 (包括等待指令、 向上运行、 向下

运行) ，从绿变黄的过程表明电梯在开门-等待- 关门 ，红色表示警报状态。

2. 楼层按钮区 ：每个楼层都有上行和下行按钮 ，用户可以通过点击这些按钮发出上行和下行请求。 按钮设计应考虑易用性 ，如使用明确的箭头图标和醒目的颜色。

3. 电梯内部按钮区 ：展示每个电梯内部的楼层按钮 ，允许用户在电梯内部选择目标楼层。按钮应为 数字形式 ，方便用户识别。 ( 关门按钮 ：考虑到”安慰按钮“的说法 ，并未对其添加功能)

4. 故障和恢复按钮 ：为每台电梯提供故障和恢复按钮 ，以便在电梯出现故障时进行模拟和处理。故 障按钮可以用红色表示 ，恢复按钮可以用绿色表示。

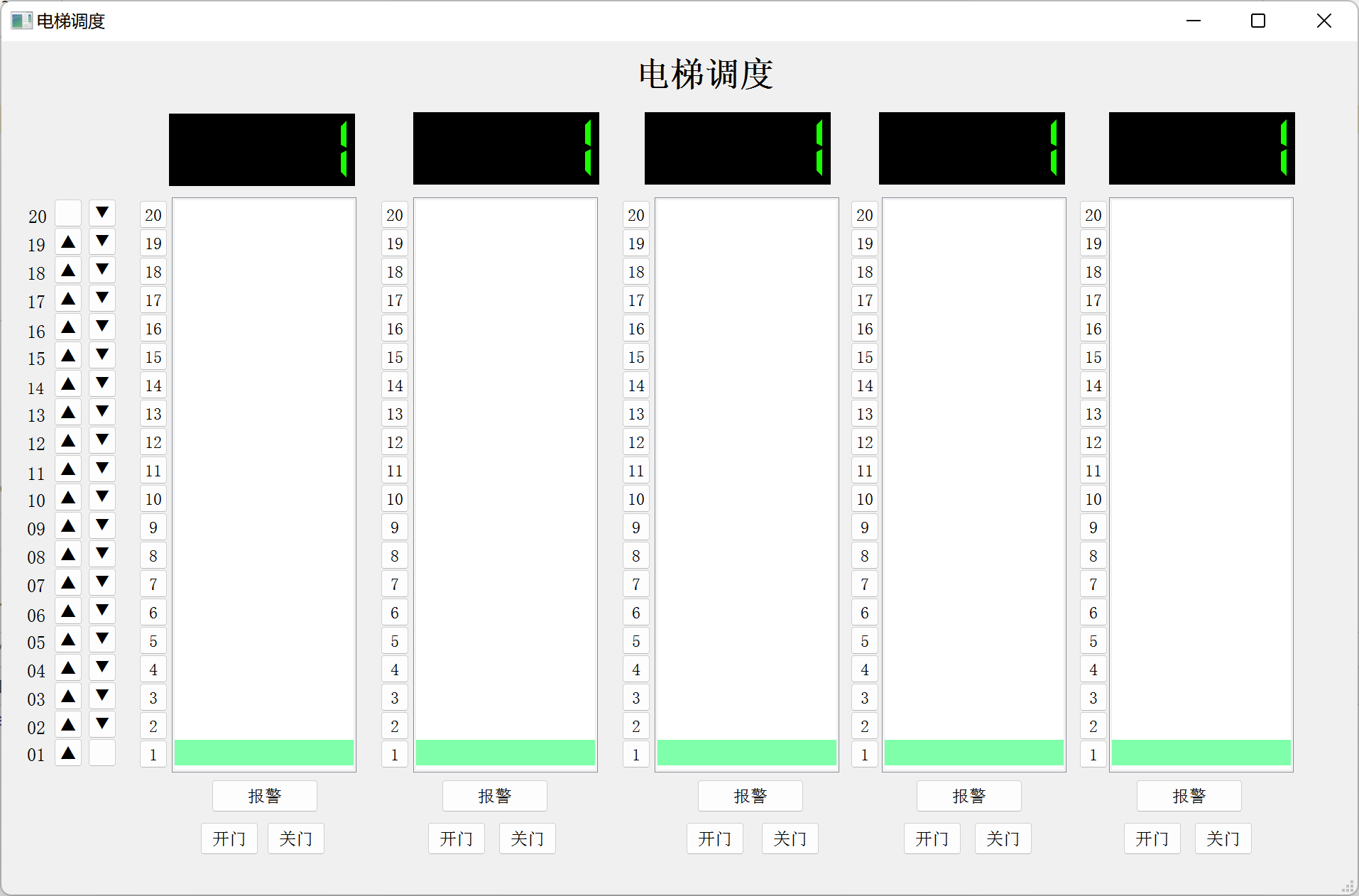
通过以上设计 ，用户可以轻松地了解电梯的实时状态 ，并通过点击按钮发出上行、下行和内部请求。

整体界面设计应保持简洁明了 ，易于用户操作和理解。

以下是程序运行时的截图 ：

**5.1** 初始状态

电梯均在⼀楼 ，LCD示数为“1”

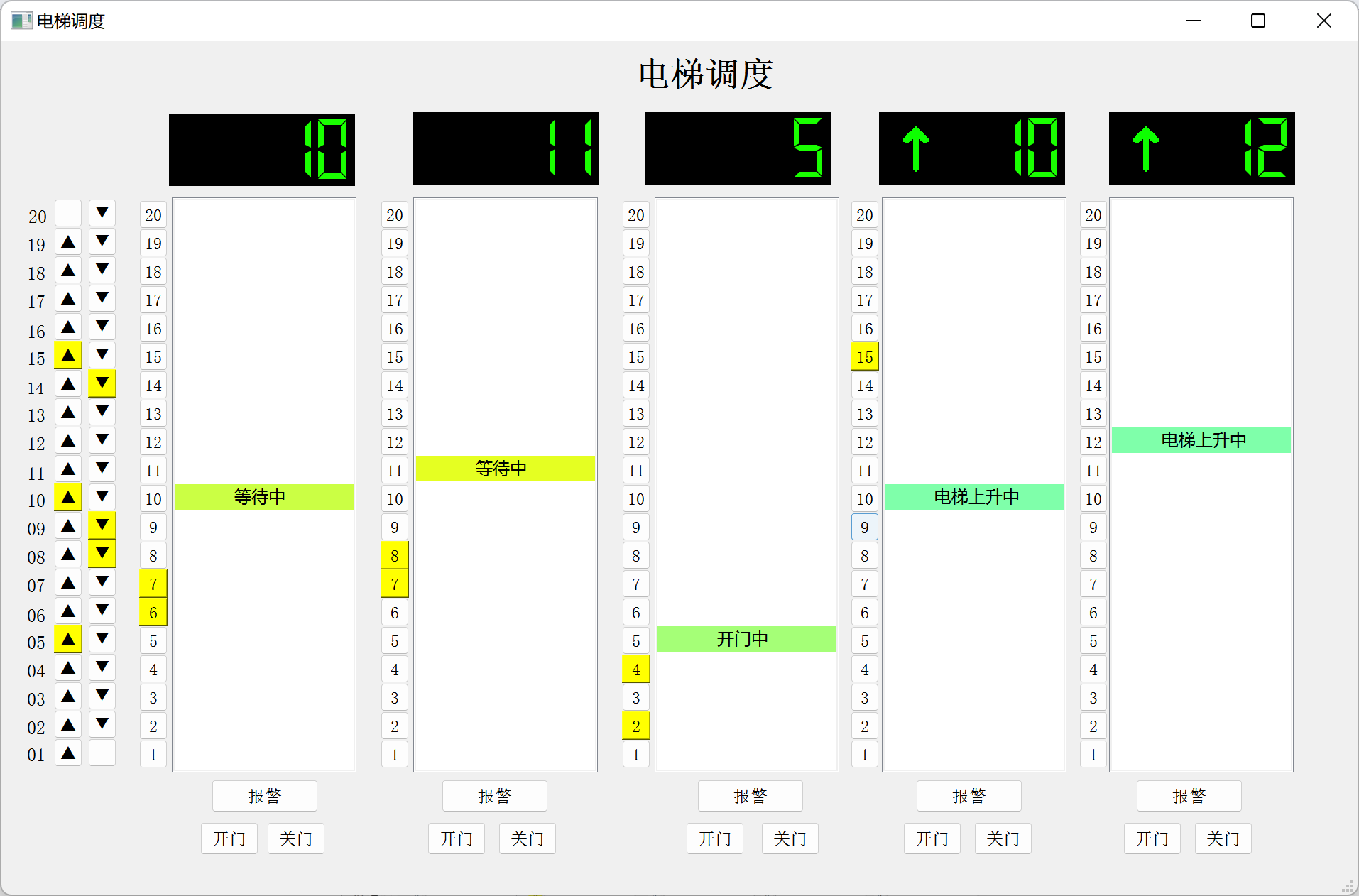


**5.2** 运行状态

 按钮按下会变成黄色 ，任务完成后黄色消去

 LCD上会展示电梯所在楼层即运行方向

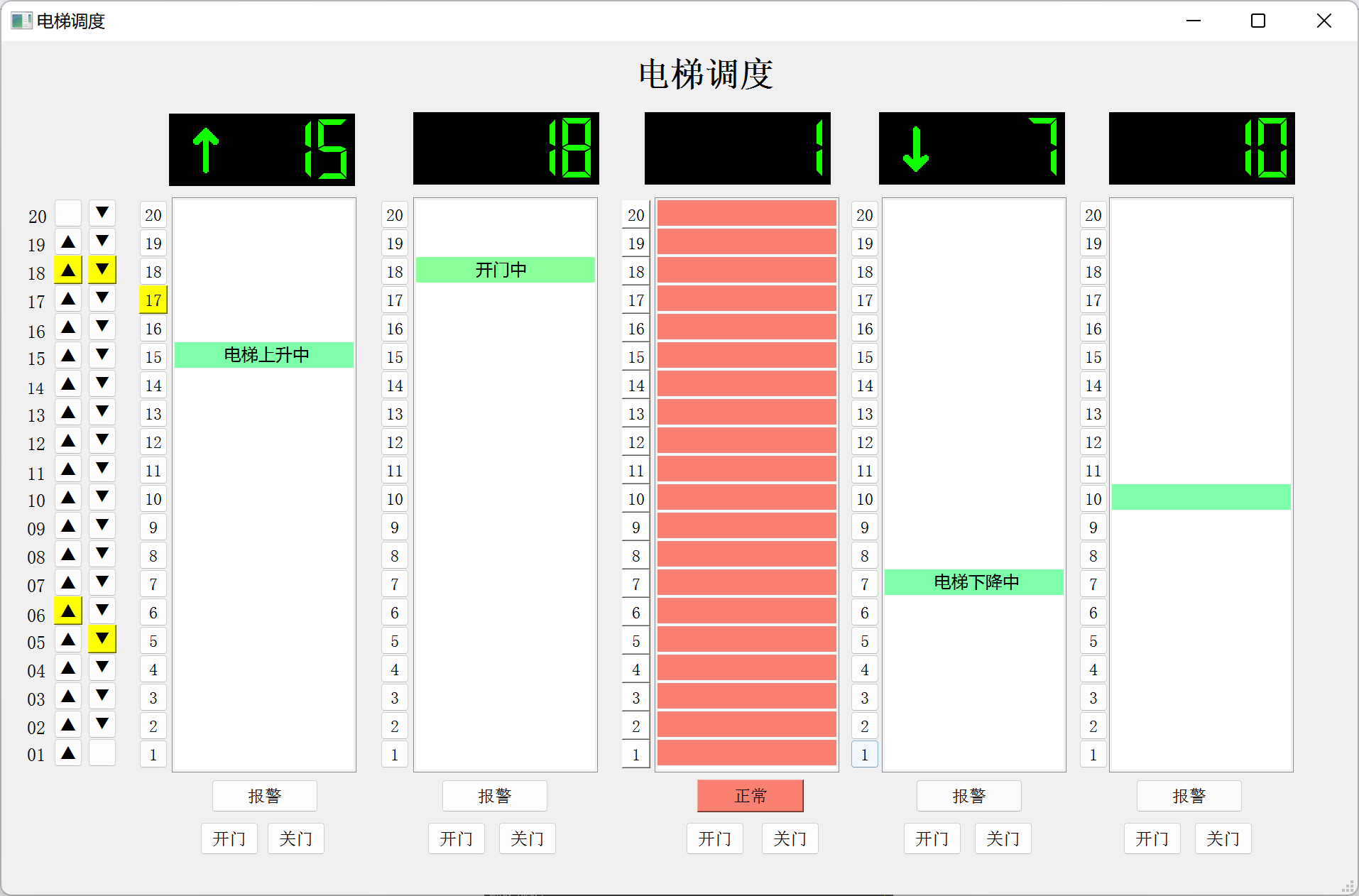
 电梯在运行 (上升或下降) 时为绿色 ，开门-等待- 关门动作时 ，颜色会从绿变黄再变回绿色



**5.3** 报警状态

 按下报警键后 ，报警键自身会变成红色 ， 电梯会回到⼀层

 点击正常键后 ，警报接触 ， 电梯可以继续正常运行

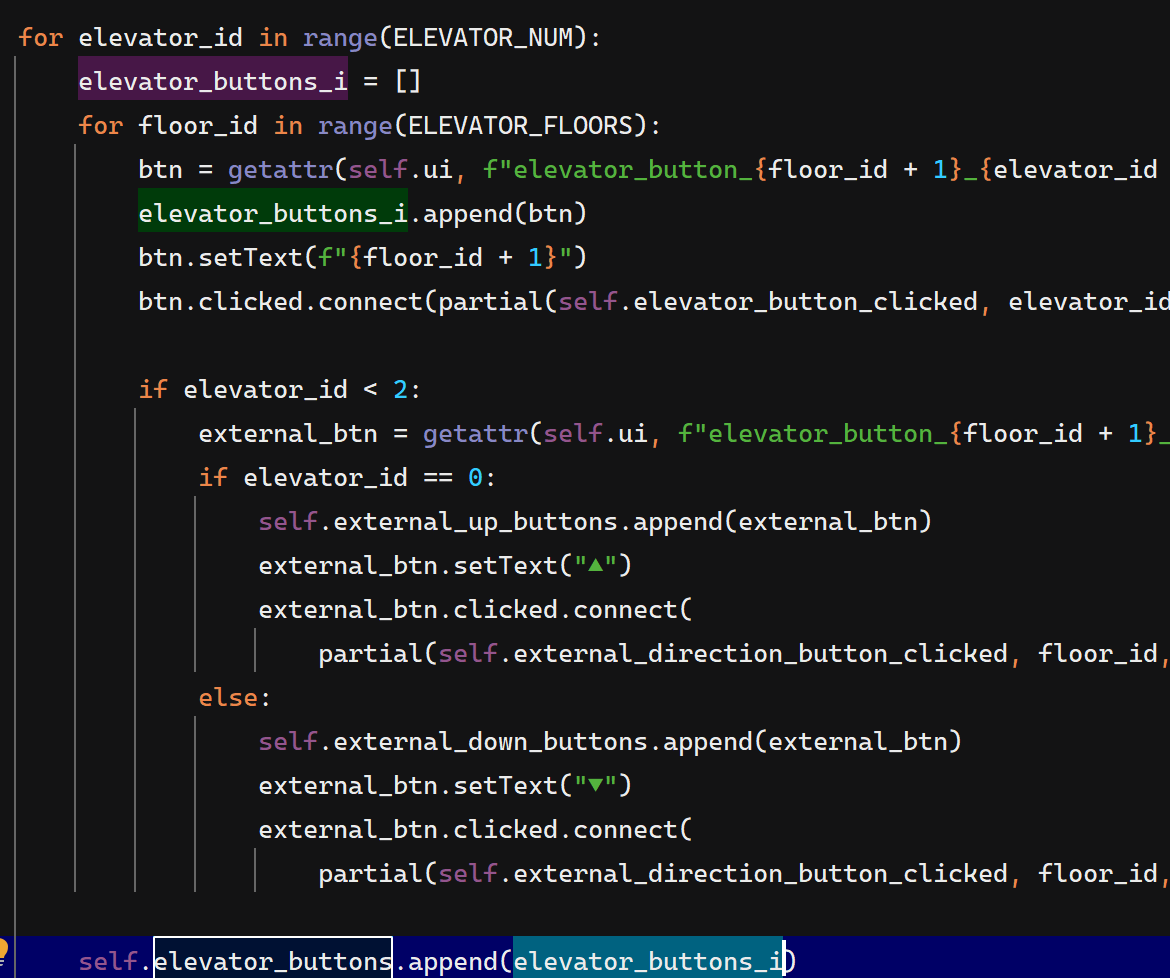


六、心得体会

**6.1** 设计过程中遇到的问题及解决办法

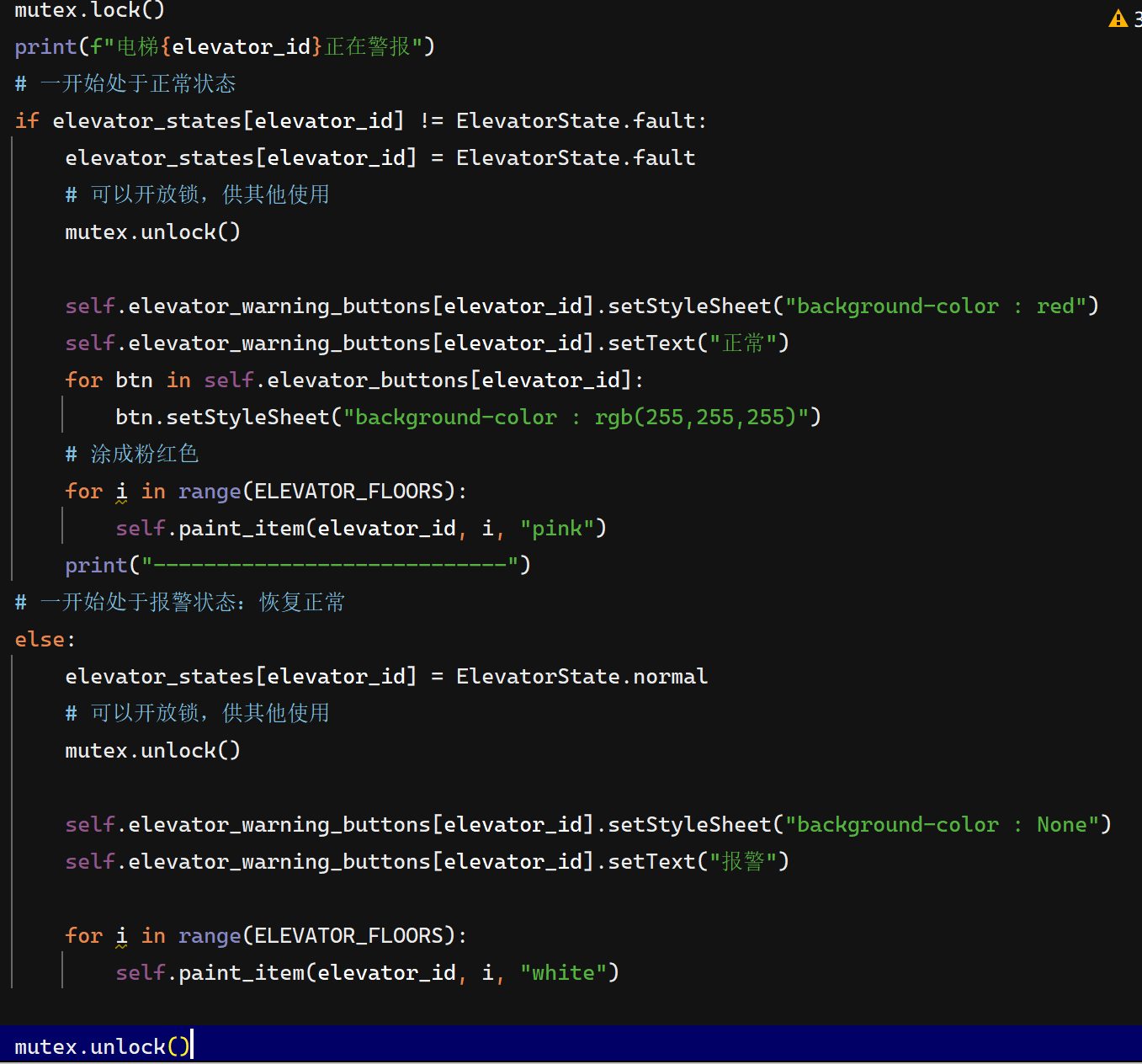
 在UI设计调试时 ，发现按下(1,2)的按钮结果(2,1)的按键亮起了

解决方案 ：在导入UI时 ，注意两个for的顺序 ，以便正确控制电梯按钮的亮灭



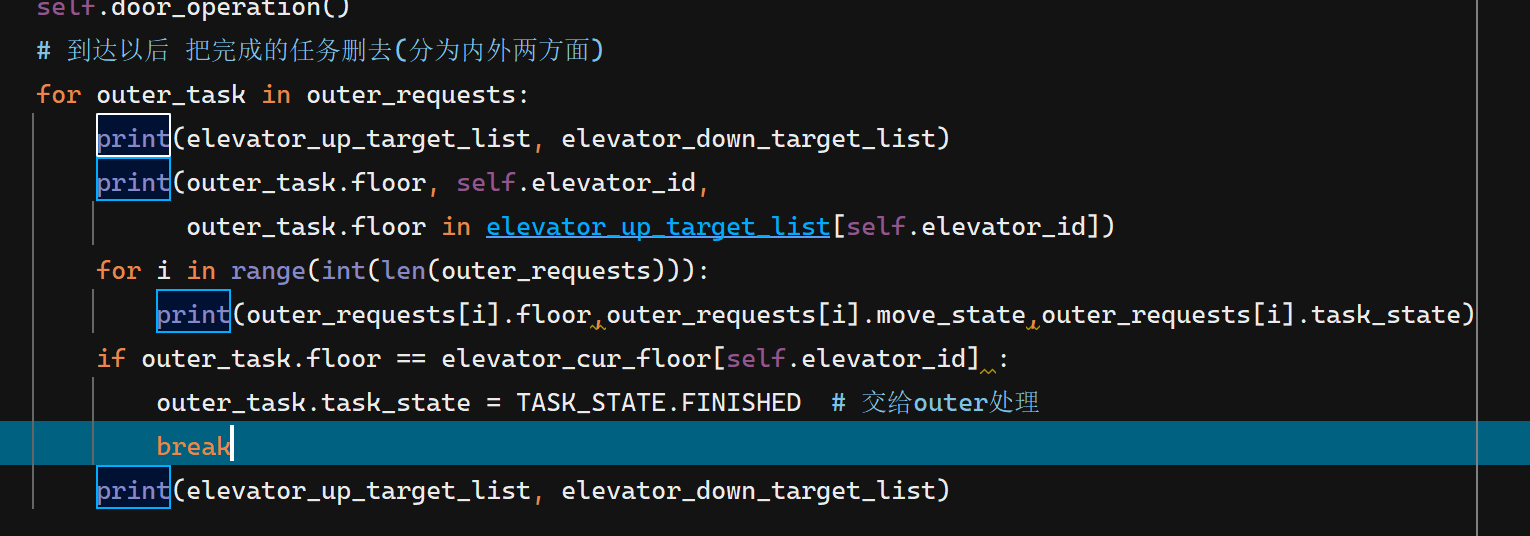
在代码设计过程中 ，点击“报警”按钮后 ， 出现了程序闪退的情况 ，且检查了源代码 ，发现数据处 理方面没有问题。

解决方案 ：检查锁的状态 ，发现在点击按钮之后 ，并未进行解锁操作 ，使得线程都被卡在这个阶 段 ，故总结为“注意信号量的变化 ，不要解锁多次”。



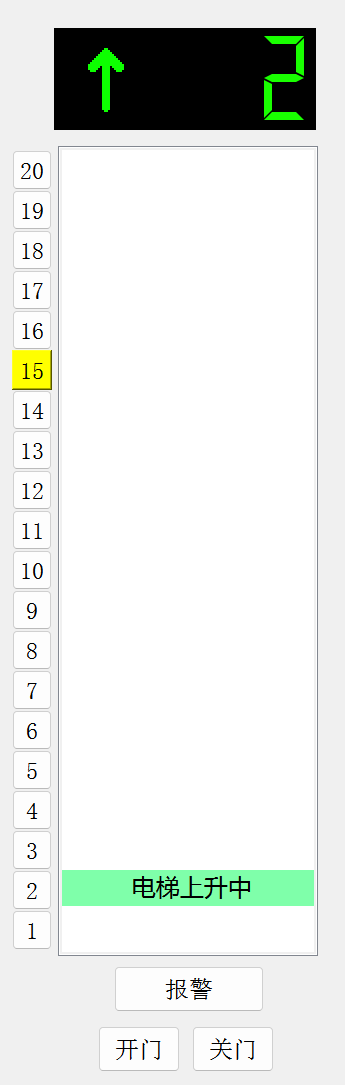
在电梯经过有外界上下楼请求的楼层 ，开关门之后 ，会将上下楼的按钮的颜色都进行变化 ，这样 下⼀个电梯 (接收到任务的) 到达该楼层时 ，会开关门 ，但此时按键是灭的 ，造成体验不佳。

解决方案 ：检测相应函数 ，使用TASK\_STATE.FINISHED来标定已经完成后的外界任务。在电梯 中遍历任务时 ，通过break让发现了任务后立即跳出 ，使得其⼀次性只能消去⼀个。



 将电梯的实时位置展示出来

解决方案 ：使用Qt中的QListWidget来实现 ，即使用表格的形式 ，使电梯没移动⼀层 ，即将之前 的格子中的内容抹去 ，同时在之前格子的上/下方展示⼀段文本。



**6.2** 经验

使用Qt Designer设计GUI ，再使用PyUIC将设计稿导出成.py文件 ，这样美观、方便、快捷。

可以使用print在命令行中输出当前信息进行调试 ：由于所展示的UI界面只能展示⼀部分的信息 ， 并不能将过程中的变量变化给输出 ，这样很难发现问题。故可以在变量变化的前后使用print ，在 命令行中输出⼀些共享资源的数值 ，看是否符合要求 ，再进行相应的调整。

 使用锁时 ，应当尽量避免死锁 ，在使用互斥锁时 ，需要确保在适当的时机解锁互斥锁 ，防止死锁

的发生。例如 ，本项目中 ，我们在执行完共享资源操作后立即解锁互斥锁 ，以避免死锁。 同时， 在处理长时间的功能 ，如在开关门时变化电梯按钮的颜色时间消耗3s ，此时将锁打开 ，允许并 行 ，使得其他线程能够在此期间获得锁 ，并进行相关操作。这样可以提高程序的并发性和响应 性 ，减少用户等待时间 ，提升用户体验。 同时 ，也要注意在多个线程之间协调好共享资源的访 问 ，避免出现数据竞争等并发问题。