**进程管理——电梯调度项目设计报告**

Designer：胡峻玮 2153393

**一、项目背景**

**1.1 项目概述**

本项目旨在开发一个电梯调度系统，该系统管理一栋楼宇内的5部电梯，服务于20层楼。系统需要处理和响应楼内外用户的请求，优化电梯运行效率，减少用户等待时间，并确保电梯运行的安全性。

**1.2 系统需求**

系统应具备以下功能：

内部控制：电梯内部配备楼层按钮、开门按钮、关门按钮、故障按钮和LED显示屏。

外部请求：每层楼设有上行和下行按钮，用户可通过这些按钮发送电梯调用请求。

调度算法：设计高效的电梯调度算法，合理分配电梯响应内部外部请求。

**二、开发环境**

**2.1 硬件和软件环境**

操作系统：Windows 11

开发工具：PyCharm 2023.3.3

编程语言：Python 3.11

**2.2 主要依赖库**

PyQt5：用于创建图形用户界面，包含QtCore、QtWidgets、QThread、QMutex、QTimer等模块。

Enum：为电梯状态和任务状态提供枚举支持。

functools.partial：用于部分应用函数，以便于创建可重用的按钮响应函数。

**三、系统设计**

**3.1 系统架构**

本系统采用基于事件驱动的多线程模型，每部电梯由一个独立的线程控制，以模拟电梯的实际运行和调度。每个电梯线程负责处理该电梯内的所有操作请求，如楼层移动、门的开关操作及故障恢复。系统中使用互斥锁（QMutex）确保电梯状态更新的原子性，避免多线程操作中的数据冲突。外部请求由一个中央控制器（OuterTaskController）处理，该控制器也作为一个独立线程运行，负责收集和分配外部楼层按钮的请求到各个电梯，优化任务分配。

**3.2 功能模块**

3.2.1 内部控制模块

按钮和显示屏：

楼层按钮：用户通过按下对应的楼层按钮发出到达该楼层的请求。

开门/关门按钮：允许用户控制电梯门的开关。电梯到达指定楼层时，用户可以选择开门或延迟关门时间。

故障按钮：用于模拟电梯发生故障的情况。按下故障按钮后，电梯将进入故障模式，停止所有正常运行，直到故障被重置。

LED显示屏：实时显示当前电梯的楼层状态，以及电梯运行状态（如上升、下降、停止等）。

3.2.2 外部请求处理

上行和下行按钮：用户在电梯外的楼层按下上行或下行按钮后，系统根据电梯当前运行状况和位置来决定哪部电梯响应该请求。系统优先选择空闲或最接近用户且方向一致的电梯，以减少用户等待时间和电梯空驶。

**3.3 调度算法介绍**

3.3.1 常见的电梯调度算法

电梯调度算法主要包括以下几种：

先来先服务算法（FCFS）：该算法按乘客请求的顺序进行调度。虽然操作公平且简单，但在高负载情况下，等待时间可能显著增加。

最短寻找楼层时间优先算法（SSTF）：此算法选择最近的请求优先服务，可以减少等待时间，但可能导致远端楼层（如顶楼和底层）长时间未得到响应，引发“饥饿”现象。

扫描算法（SCAN）：电梯在最底层和最顶层之间往返，连续处理同一方向上的请求，确保总体等待时间的稳定性。

LOOK算法：此算法为SCAN的改进版本，电梯在检测到移动方向上无进一步目标时立即改变方向。

本项目的电梯调度基本基于LOOK算法，但具有如下区别：当电梯移动方向上无目标时，不立即改变方向，而是等到反方向产生任务后才改变运行方向。

3.3.2 每部电梯内部的任务请求与响应机制

Ⅰ 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”按钮控制门的开闭。

3. 用户点击“故障”按钮报告电梯故障。

Ⅱ 任务响应机制

假设电梯当前楼层为 cur\_floor，用户在电梯内点击了第 i 层的按钮：

如果 i = cur\_floor，电梯自动开门，保持开门状态一段时间后自动关闭。

如果电梯处于静止空闲状态，电梯直接向第 i 层移动。

如果 i > cur\_floor 且电梯运行方向向上，则将该任务加入向上的任务列表，并按楼层从小到大排序。

如果 i > cur\_floor 但电梯运行方向向下，则将该任务加入向下的任务列表，并按楼层从大到小排序。

如果 i < cur\_floor 且电梯运行方向向下，则将该任务加入向下的任务列表，并按楼层从大到小排序。

如果 i < cur\_floor 但电梯运行方向向上，则将该任务加入向下的任务列表，并按楼层从大到小排序。

Ⅲ 开门和关门按钮响应

开门按钮：

如果电梯处于静止状态，则直接开门，并在一定时间后自动关闭。

如果电梯正在运行，则忽略开门请求，保持电梯门关闭。

关门按钮：

如果电梯门已开，点击关门按钮则立即关闭门。

3.3.4 故障处理

用户点击“故障”按钮后，系统将清除该电梯此时的所有内部任务，并尝试将影响的外部任务重新分配给其他正常的电梯。

故障发生后，电梯的内部按钮将被锁定，直到故障被修复。

3.3.5 电梯外部任务的请求与响应机制

Ⅰ 外部任务的请求

用户在其对应的楼层按下“上行”或“下行”按钮，发出向上或向下的移动请求。

Ⅱ 任务响应机制

当系统检测到用户按下某一层的“上行”或“下行”按钮后，首先扫描所有五部电梯，寻找没有故障的电梯。

在可用的电梯中，选择能最快且方向一致到达请求楼层的电梯。任务分配逻辑如下：

* 计算距离：根据每部电梯当前的位置 (cur\_floor) 和请求楼层 (task\_floor) 计算距离 (distance)。
* 静止状态电梯：如果电梯未移动，distance = abs(cur\_floor - task\_floor)。
* 顺路电梯：如果电梯当前运行方向与请求方向一致且冲着 task\_floor 方向运行，distance = abs(cur\_floor task\_floor)。
* 反向或不一致方向电梯：如果电梯运行方向与请求方向不一致，或虽然冲着 task\_floor 方向但运行方向与按钮方向不一致，distance = abs(max\_target - cur\_floor) + abs(max\_target - task\_floor)，其中 max\_target 是该方向上当前任务中最远的楼层。
* 选择计算出 distance 最小的电梯，将前往 task\_floor 的任务加入该电梯的任务列表，并根据距离优先级重新排序任务列表。

**3.4 Look调度算法综合阐述**

Look调度算法是该电梯系统的核心，负责合理地分配电梯以响应内部和外部的请求。该算法通过优化响应时间和运行效率，以最大程度减少用户等待时间和电梯的能耗。以下是具体实现细节：

3.3.1 动态任务调度

任务队列：每部电梯维护两个任务列表，一个用于上行任务，一个用于下行任务。这些列表根据电梯当前的移动方向动态更新，确保电梯在尽可能完成当前方向上的所有任务后，才改变运动方向。

任务分配：当外部按钮被按下时，OuterTaskController 控制器计算每部电梯到请求楼层的距离，并考虑当前的移动方向和状态，选择最合适的电梯来响应请求。这种方法减少了电梯的无效运行，并优化了电梯的响应时间。

3.3.2 优化运行策略

楼层优先访问：在向一个方向移动时，电梯尽可能先访问离当前位置最近的楼层，然后逐步移动到更远的楼层。这一策略有效减少了电梯的停顿次数和运行时间。

应急调度：在电梯发生故障时，系统会立即重新分配该电梯中的任务给其他电梯，同时故障电梯停止接收新的请求直到故障被修复。

3.3.3 故障处理机制

故障检测与响应：电梯在执行任务过程中会不断检查自身状态。一旦检测到故障（如按钮长时间未恢复默认状态或电梯长时间未离开某楼层），Elevator 类中的故障处理函数会被触发。此时，电梯将取消当前所有任务，清空任务队列，并通知控制器重新分配这些任务。

故障恢复：一旦故障得到修复，电梯重新加入系统，等待来自 OuterTaskController 的新任务分配。

3.3.4 门操作与时间控制

门操作管理：电梯门的开启和关闭通过时间控制来模拟。电梯到达指定楼层后，将执行一系列的门操作，包括开门和维持开门状态一段时间，然后自动关闭门。门的开闭状态通过一个进度变量 door\_open\_status 进行跟踪，确保门操作的流畅性和时效性。

用户交互：用户可以通过内部的开门和关门按钮实时控制门的状态。如果用户在门开启期间请求延长开门时间，系统将重置保持门开启的计时器。

**四、用户界面设计**

**4.1 界面布局**

本系统的用户界面(UI)设计简洁直观，旨在提供清晰的操作和实时信息反馈。整体界面分为三大部分：

总控台：位于界面左侧，包括任务生成区和电梯运行信息输出区。

电梯控制面板：界面中部，为每部电梯提供独立的控制面板。

外部按钮面板：界面右侧，展示每层楼的上行和下行按钮。

4.1.1 总控台

总控台包括以下元素：

任务输入：用户可输入希望生成的随机任务数量，通过“产生随机任务”按钮激活任务生成。

电梯运行信息显示：文本框实时显示电梯的运行状态、任务处理情况及任何故障信息。

4.1.2 电梯控制面板

每部电梯的控制面板包括：

楼层显示屏：显示电梯当前楼层。

楼层按钮：内部楼层按钮允许用户选择目标楼层。

开门/关门按钮：控制电梯门的开闭。

故障按钮：用于模拟电梯故障状态。

门状态显示：通过改变按钮颜色和显示来模拟电梯门的开闭状态。

4.1.3 外部按钮面板

每层楼设有上行和下行按钮，用于发出电梯调用请求。按钮旁边的楼层标签显示按钮对应的楼层。

**4.2 交互设计**

系统设计考虑到了易用性和直观性，确保用户可以快速理解和操作：

4.2.1 电梯操作

楼层选择：点击对应的楼层按钮即可发送到达该楼层的请求。

门控制：开门和关门按钮允许用户在电梯到达后控制门的开闭。

故障模拟：故障按钮使电梯进入故障状态，再次点击可恢复正常。

4.2.2 任务生成

随机任务：输入希望生成的任务数量，系统将自动产生内部请求或外部呼叫。

4.2.3 实时信息反馈

信息输出区：系统实时输出电梯状态变化、任务处理结果及故障信息，帮助用户了解电梯当前状况。

4.2.4 界面反馈

按钮反馈：按钮在激活时改变颜色，提供视觉反馈。

楼层显示：电梯当前楼层通过LCD显示屏实时更新。

门状态模拟：门状态通过四个方形按钮模拟，颜色和可见性变化反映实际门状态。

**五、代码实现**

**5.1 代码结构**

本项目的代码主要分为以下几个模块：

主界面模块 (UI\_MainWindow)：负责整个应用的界面布局和用户交互逻辑。

电梯控制模块 (Elevator)：处理单个电梯的运动逻辑、门控制以及故障管理。

任务控制模块 (OuterTaskController)：负责处理和分配从外部按钮产生任务到适合的电梯。

数据模型和状态管理：包含电梯状态、任务状态、电梯位置等数据的管理。

辅助工具类：如枚举类 (Enum) 用于定义电梯和按钮状态，以及全局设置。

**5.2 关键代码解析**

以下是几个核心功能的代码片段及其详细解释：

5.2.1 电梯状态更新和门控制

电梯的基本运行逻辑和门的控制是通过 Elevator 类的 move\_one\_floor 方法和 door\_operation 方法实现的。这两个方法处理电梯按计划移动楼层和门的自动开关。

class Elevator(QThread):

def move\_one\_floor(self, move\_state):  
 self.update\_elevator\_status(move\_state)  
 if not self.check\_for\_faults(move\_state):  
 return  
 self.update\_current\_floor(move\_state)  
 if elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.break\_down:  
 self.handle\_fault()  
  
# 一次门的操作 包括开门和关门  
def door\_operation(self):  
 opening\_time = 0.0 # 记录门打开所需的累积时间  
 open\_time = 0.0 # 记录门保持开启的累积时间  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.opening\_door # 初始设置为门正在打开  
  
 while True:  
 # 检查电梯是否处于故障状态  
 if elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.break\_down:  
 self.handle\_fault() # 处理故障  
 break  
  
 # 用户请求开门  
 if open\_button\_clicked[self.elevator\_id]:  
 # 如果门正在关，重置为开门状态  
 if elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.closing\_door:  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.opening\_door  
  
 # 如果门已经是开启状态，重置保持开启的时间  
 if elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.open\_door:  
 open\_time = 0  
  
 # 重置开门按钮状态  
 open\_button\_clicked[self.elevator\_id] = False  
  
 # 用户请求关门  
 if close\_button\_clicked[self.elevator\_id]:  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.closing\_door # 设置为门正在关闭  
 open\_time = 0 # 重置门开启时间  
  
 # 重置关门按钮状态  
 close\_button\_clicked[self.elevator\_id] = False  
  
 # 门正在打开的逻辑处理  
 if elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.opening\_door:  
 mutex.unlock() # 允许其他线程运行  
 self.msleep(self.rest\_time) # 等待一个时间段  
 mutex.lock() # 重新锁定  
 opening\_time += self.rest\_time # 累积增加开门时间  
 door\_open\_status[self.elevator\_id] = opening\_time / DOOR\_OPENING\_TIME # 更新门的打开进度  
  
 # 如果达到了完全开门的时间  
 if opening\_time >= DOOR\_OPENING\_TIME:  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.open\_door # 设置状态为门已开  
  
 # 门已经完全开启的处理  
 elif elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.open\_door:  
 mutex.unlock()  
 self.msleep(self.rest\_time)  
 mutex.lock()  
 open\_time += self.rest\_time # 累积门开启时间  
 if open\_time >= DOOR\_OPEN\_TIME:  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.closing\_door # 时间到，开始关闭门  
  
 # 门正在关闭的逻辑处理  
 elif elevator\_status[self.elevator\_id] == ELEVATOR\_STATUS.closing\_door:  
 mutex.unlock()  
 self.msleep(self.rest\_time)  
 mutex.lock()  
 opening\_time -= self.rest\_time # 减少开门时间  
 door\_open\_status[self.elevator\_id] = opening\_time / DOOR\_OPENING\_TIME # 更新门的关闭进度  
  
 # 如果门完全关闭  
 if opening\_time <= 0:  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.normal # 设置电梯状态为正常  
 break # 退出循环，操作结束

5.2.2 用户界面与交互

UI\_MainWindow 类负责创建和管理用户界面，包括按钮、显示屏、输入框等组件的布局和事件处理。

def setup\_ui(self):  
 self.setWindowTitle("Elevator scheduling system designed by Hu Junwei 2153393 ")  
 self.setGeometry(WINDOW\_SIZE)  
  
 h1 = QHBoxLayout()  
 self.setLayout(h1)  
 v1 = QVBoxLayout()  
 h1.addLayout(v1)  
 title1 = QLabel("电梯总控台")  
 title1.setStyleSheet("font-size:40px;""font-weight:bold;")  
 v1.addWidget(title1)  
 v1.setAlignment(title1, Qt.AlignHCenter)  
 # 接收用户输入的产生任务的数量  
 instruction1 = QLabel("请输入随机产生的电梯任务数量:")  
 v1.addWidget(instruction1)  
 self.generate\_num\_edit = QLineEdit()  
 self.generate\_num\_edit.setText("0")  
 self.generate\_num\_edit.setStyleSheet("font-size:40px;""font-weight:bold;")  
 v1.addWidget(self.generate\_num\_edit)  
 button = QPushButton()  
 button.setText("产生随机任务")  
 button.setStyleSheet("background-color :rgb(148, 0, 211);""border-style: solid;"  
 "border-width: 20px;"  
 "border-color: rgb(148, 0, 211);"  
 "border-radius:20px;"  
 "color:white;")  
 button.clicked.connect(self.\_\_generate\_tasks)  
 v1.addWidget(button)  
  
 # 输出电梯信息  
 self.output = QTextEdit()  
 self.output.setText("电梯运行信息如下所示：\n")  
 v1.addWidget(self.output)  
 h2 = QHBoxLayout()  
 h1.addLayout(h2)  
  
 # 对每一个电梯都进行相同的设置  
 for i in range(ELEVATOR\_NUMS):  
 v2 = QVBoxLayout() # 竖直布局  
 h2.addLayout(v2)  
  
 # 电梯上方的LCD显示屏  
 floor\_display = QLCDNumber() # 定义了一个LCD显示屏，用来显示电梯当前所在的楼层数  
 # 将显示屏的位数设置为 2  
 floor\_display.setNumDigits(2)  
 # 将段的样式设置为 Flat，使数字居中显示  
 floor\_display.setSegmentStyle(QLCDNumber.Flat)  
 # 设置样式表，将数字颜色设为红色  
 floor\_display.setStyleSheet("color: rgb(255, 0, 0);")  
 floor\_display.setFixedSize(100, 50) # LCD显示屏的大小  
 self.\_\_floor\_displayers.append(floor\_display)  
 v2.addWidget(floor\_display) # 将该LCD添加到v2布局中  
  
 # 添加文字提示  
 Text = QLabel("电梯" + str(i + 1) + "内部按钮", self)  
 v2.addWidget(Text)  
 v2.addStretch()  
  
 # 故障按钮  
 fault\_button = QPushButton("故障")  
 fault\_button.setFixedSize(120, 40)  
 fault\_button.clicked.connect(partial(self.\_\_inner\_fault\_button\_clicked, i))  
 self.\_\_inner\_fault\_buttons.append(fault\_button)  
 v2.addWidget(fault\_button)  
  
 # 设置每一个电梯的内部按钮  
 self.\_\_inner\_num\_buttons.append([])  
 elevater\_button\_layout = QHBoxLayout() # 用来水平的排列每排按钮  
 # 创建电梯按钮  
 button\_group1 = QVBoxLayout() # 前10层按钮  
 for j in range(1, int(FLOORS / 2 + 1)):  
 button = QPushButton(str(int(FLOORS / 2 + 1 - j)))  
 button.setFixedSize(35, 35)  
  
 # 绑定点击每一个楼层的按钮后的事件  
 button.clicked.connect(partial(self.\_\_inner\_num\_button\_clicked, i, int(FLOORS / 2 + 1 - j)))  
 button.setStyleSheet("background-color : rgb(255,255,255);""border-style: solid;"  
 "border-width: 2px;"  
 "border-color: rgb(165,93,81);"  
 "border-radius:10px;"  
 "color:black;")  
 self.\_\_inner\_num\_buttons[i].append(button)  
 button\_group1.addWidget(button)  
 # 增大元素之间的竖直方向距离  
 button\_group1.setSpacing(10)  
  
 # 开门按钮  
 open\_button = QPushButton("开")  
 open\_button.setFixedSize(35, 35)  
 open\_button.clicked.connect(partial(self.\_\_inner\_open\_button\_clicked, i))  
 self.\_\_inner\_open\_buttons.append(open\_button)  
 open\_button.setStyleSheet("background-color :rgb(237,220,195);""border-style: solid;"  
 "border-width: 2px;"  
 "border-color: rgb(192, 192, 192);"  
 "border-radius:10px;"  
 "color:black;")  
 button\_group1.addWidget(open\_button)  
  
 button\_group2 = QVBoxLayout() # 后10层按钮  
 for j in range(1, int(FLOORS / 2 + 1)):  
 button = QPushButton(str(FLOORS + 1 - j))  
 button.setFixedSize(35, 35)  
  
 # 绑定点击每一个楼层的按钮后的事件  
 button.clicked.connect(partial(self.\_\_inner\_num\_button\_clicked, i, FLOORS + 1 - j))  
 button.setStyleSheet("background-color : rgb(255,255,255);""border-style: solid;"  
 "border-width: 2px;"  
 "border-color: rgb(165,93,81);"  
 "border-radius:10px;"  
 "color:black;")  
 self.\_\_inner\_num\_buttons[i].append(button)  
 button\_group2.addWidget(button)  
 button\_group2.setSpacing(10)  
  
 # 关门按钮  
 close\_button = QPushButton("关")  
 close\_button.setFixedSize(35, 35)  
 close\_button.clicked.connect(partial(self.\_\_inner\_close\_button\_clicked, i))  
 close\_button.setStyleSheet("background-color :rgb(237,220,195);""border-style: solid;"  
 "border-width: 2px;"  
 "border-color: rgb(192, 192, 192);"  
 "border-radius:10px;"  
 "color:black;")  
 self.\_\_inner\_close\_buttons.append(close\_button)  
 button\_group2.addWidget(close\_button)  
  
 # 将 button\_group1 添加到 elevater\_button\_layout 中  
 button\_group1\_widget = QWidget()  
 button\_group1\_widget.setLayout(button\_group1) # 嵌套一下 QVBoxLayout  
 elevater\_button\_layout.addWidget(button\_group1\_widget)  
 # 将 button\_group1 添加到 elevater\_button\_layout 中  
 button\_group2\_widget = QWidget()  
 button\_group2\_widget.setLayout(button\_group2) # 嵌套一下 QVBoxLayout  
 elevater\_button\_layout.addWidget(button\_group2\_widget)  
  
 elevater\_button\_layout\_widget = QWidget()  
 elevater\_button\_layout\_widget.setLayout(elevater\_button\_layout) # 嵌套一下 QHBoxLayout  
 v2.addWidget(elevater\_button\_layout\_widget)  
 v2.addStretch()  
 # 接下来给v2添加门  
 door = []  
 # 创建四个充当门的按钮的水平布局  
 hbox1 = QHBoxLayout()  
 for d in range(4):  
 DoorTimer = QTimer()  
 self.door\_timer.append(DoorTimer)  
 button = QPushButton('', self)  
 button.setFixedSize(20, 20)  
 door.append(button)  
 hbox1.addWidget(button)  
 door[0].setStyleSheet('background-color: transparent;')  
 door[1].setStyleSheet('background-color: black;')  
 door[2].setStyleSheet('background-color: black;')  
 door[3].setStyleSheet('background-color: transparent;')  
 door\_each\_elevator.append(door) # 将这扇门添加进Doors中  
 hbox1\_widget = QWidget()  
 hbox1\_widget.setLayout(hbox1)  
 v2.addWidget(hbox1\_widget)  
  
 # 添加文字提示  
 Text1 = QLabel("电梯" + str(i + 1) + "的门", self)  
 v2.addWidget(Text1)  
 # v2.addStretch()  
  
 v2.addStretch()  
 # 设置布局中的组件水平居中  
 v2.setAlignment(floor\_display, Qt.AlignHCenter)  
 v2.setAlignment(fault\_button, Qt.AlignHCenter)  
 v2.setAlignment(elevater\_button\_layout\_widget, Qt.AlignHCenter)  
 v2.setAlignment(hbox1\_widget, Qt.AlignHCenter)  
 v2.setAlignment(Text, Qt.AlignHCenter)  
 v2.setAlignment(Text1, Qt.AlignHCenter)  
  
 # 上下按钮  
 v3 = QVBoxLayout() # 创建一个垂直布局  
 h1.addLayout(v3)  
 # 标题  
 title\_outer = QLabel("外侧按钮")  
 v3.addWidget(title\_outer)  
 v3.setAlignment(title\_outer, Qt.AlignHCenter)  
  
 for i in range(FLOORS): # 对于每一层楼  
 h4 = QHBoxLayout() # 创建一个水平布局  
 v3.addLayout(h4)  
 label = QLabel(str(FLOORS - i))  
 h4.addWidget(label)  
 if i != 0:  
 # 给2楼到顶楼放置上行按钮  
 up\_button = QPushButton("▲")  
 up\_button.setFixedSize(30, 30)  
 up\_button.clicked.connect(  
 partial(self.\_\_outer\_direction\_button\_clicked, FLOORS - i, MOVING\_STATUS.up))  
 self.\_\_outer\_up\_buttons.append(up\_button) # 从顶楼往下一楼开始..  
 h4.addWidget(up\_button)  
  
 if i != FLOORS - 1:  
 # 给1楼到顶楼往下一楼放置下行按钮  
 down\_button = QPushButton("▼")  
 down\_button.setFixedSize(30, 30)  
 down\_button.clicked.connect(  
 partial(self.\_\_outer\_direction\_button\_clicked, FLOORS - i, MOVING\_STATUS.down))  
 self.\_\_outer\_down\_buttons.append(down\_button) # 从顶楼开始..到2楼  
 h4.addWidget(down\_button)  
  
 # 设置定时  
 self.timer.setInterval(30)  
 self.timer.timeout.connect(self.update)  
 self.timer.start()  
  
 self.show()

5.2.3 任务分配逻辑

OuterTaskController 类通过监听外部按钮的事件，分析当前各电梯的状态和位置，决定将任务分配给哪部电梯。

class OuterTaskController(QThread):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_() # 初始化父类 QThread  
  
 def run(self):  
 while True:  
 mutex.lock()  
 self.assign\_tasks()  
 self.cleanup\_finished\_tasks()  
 mutex.unlock()  
  
 def assign\_tasks(self):  
 global outer\_button\_task  
 for outer\_task in outer\_button\_task:  
 if outer\_task.state == OUTER\_BUTTON\_STATUS.unassigned:  
 target\_id = self.find\_closest\_elevator(outer\_task)  
 if target\_id != -1:  
 self.assign\_task\_to\_elevator(outer\_task, target\_id)  
  
 def find\_closest\_elevator(self, outer\_task):  
 min\_distance = FLOORS + 1  
 target\_id = -1  
 for i in range(ELEVATOR\_NUMS):  
 if elevator\_status[i] == ELEVATOR\_STATUS.break\_down:  
 continue  
 distance = self.calculate\_distance(i, outer\_task)  
 if distance < min\_distance:  
 min\_distance = distance  
 target\_id = i  
 return target\_id

5.2.4 故障处理逻辑

故障处理是电梯系统中一个至关重要的功能，它确保在电梯发生故障时能迅速识别并进行相应处理，以维护系统的安全和效率。Elevator 类中的 handle\_fault 方法负责处理电梯的故障状态。

def handle\_fault(self):  
 # 设置电梯状态为故障状态  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.break\_down  
 # 初始化门的开启状态为0，表示门完全关闭  
 door\_open\_status[self.elevator\_id] = 0.0  
 # 重置开门按钮状态，防止在故障处理期间误操作  
 open\_button\_clicked[self.elevator\_id] = False  
 # 重置关门按钮状态，同样防止误操作  
 close\_button\_clicked[self.elevator\_id] = False  
 # 再次确认设置电梯状态为故障，以确保处理逻辑的一致性  
 elevator\_status[self.elevator\_id] = ELEVATOR\_STATUS.break\_down  
 # 遍历所有外部按钮任务  
 for outer\_task in outer\_button\_task:  
 # 检查任务是否处于等待状态  
 if outer\_task.state == OUTER\_BUTTON\_STATUS.waiting:  
 # 如果任务目标楼层在上行或下行任务列表中，将其状态设置为未分配  
 if outer\_task.target in up\_task\_remains[self.elevator\_id] or outer\_task.target in down\_task\_remains[self.elevator\_id]:  
 outer\_task.state = OUTER\_BUTTON\_STATUS.unassigned # 使得这些任务可以被重新分配  
 # 清空当前电梯的上行任务列表  
 up\_task\_remains[self.elevator\_id] = []  
 # 清空当前电梯的下行任务列表  
 down\_task\_remains[self.elevator\_id] = []

状态更新：首先，方法将电梯的状态设置为故障 (ELEVATOR\_STATUS.break\_down)。这是通知系统该电梯目前不可用的标志。

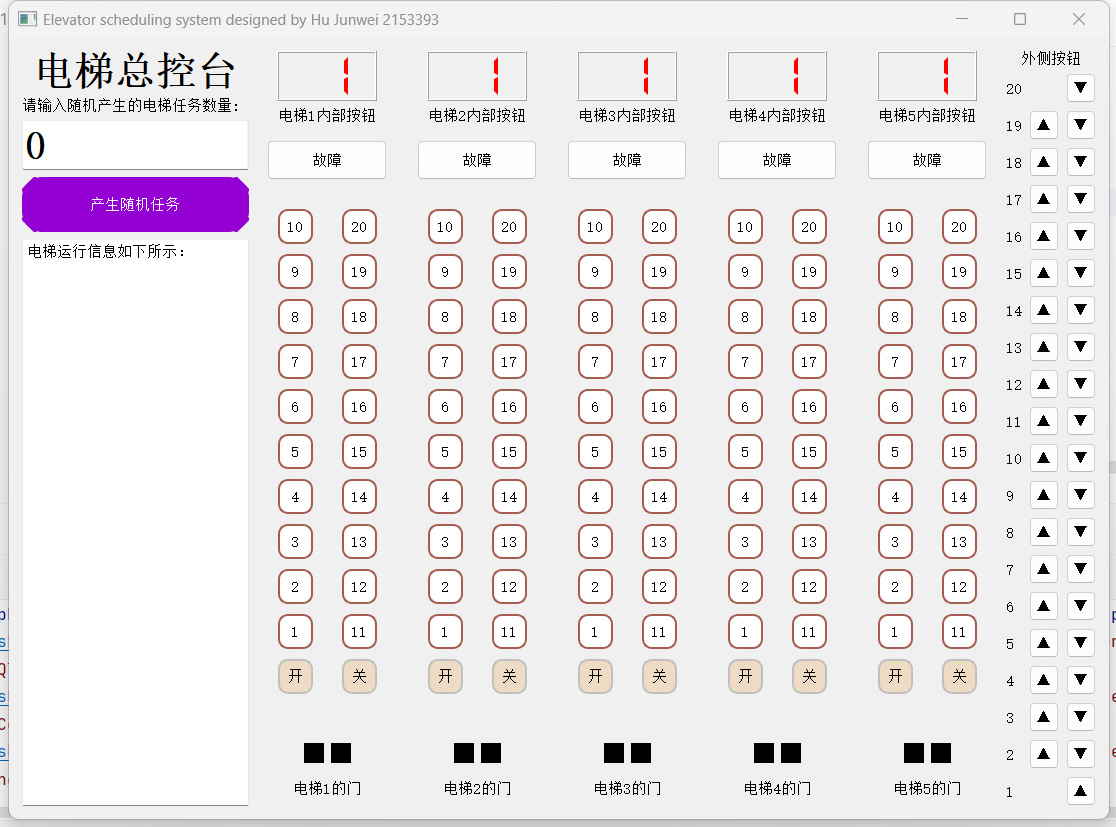
门状态和按钮重置：故障发生时，电梯门应确保处于关闭状态。此外，开门和关门按钮的状态被重置，避免在故障处理过程中被误操作。

任务队列清空：清空当前电梯的上行和下行任务队列。这一步骤是必要的，因为电梯在恢复正常之前不能执行这些任务。

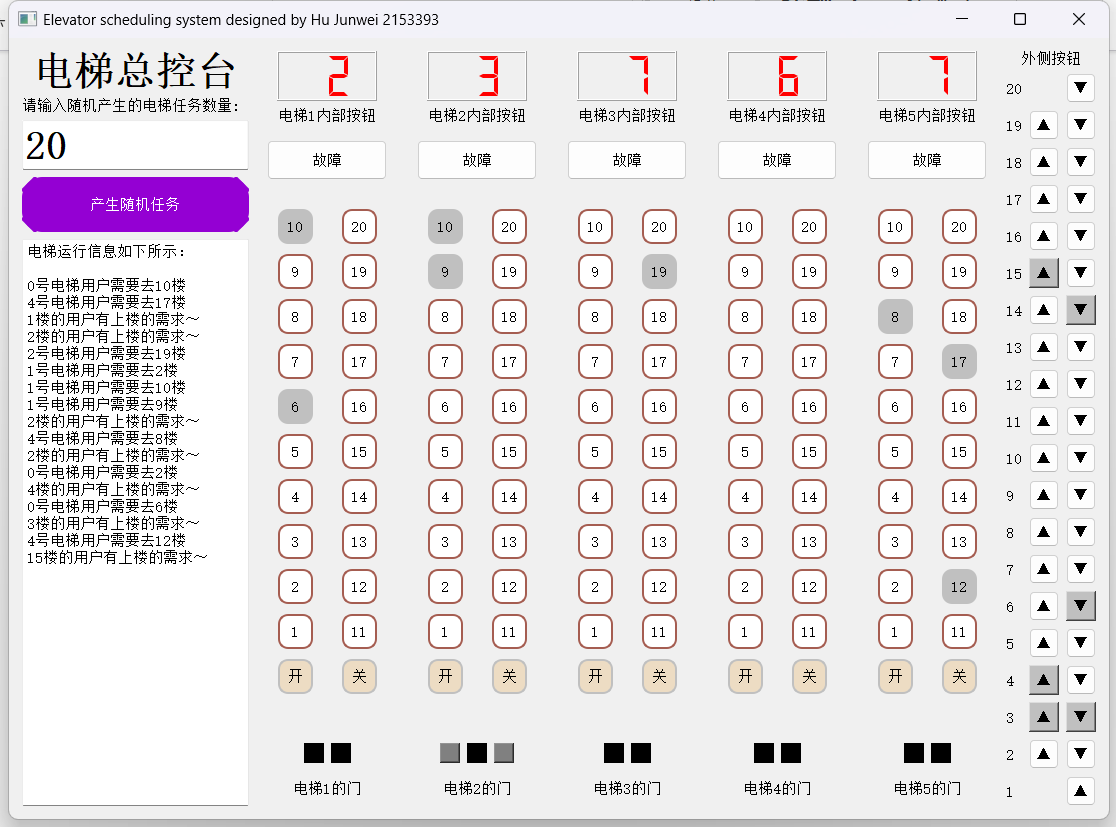
任务重新分配：遍历所有等待中的外部任务，检查这些任务是否被分配给了故障电梯。如果是，将这些任务的状态设置为未分配 (OUTER\_BUTTON\_STATUS.unassigned)，这样其他正常的电梯可以接管这些任务。

1. **代码测试**

**6.1 UI界面**

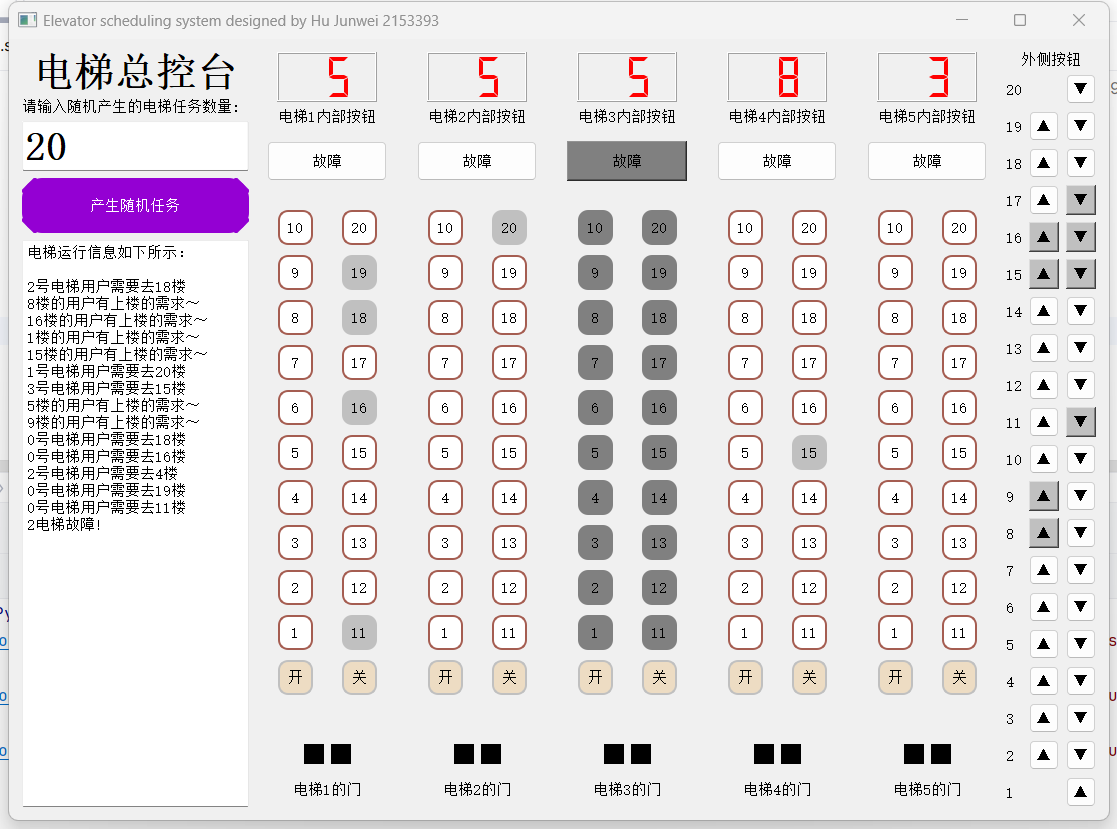
****

**6.2 随机任务测试**

****

**6.3 故障测试**

6.3.1 出现故障

****

6.3.2 故障解决

