实现

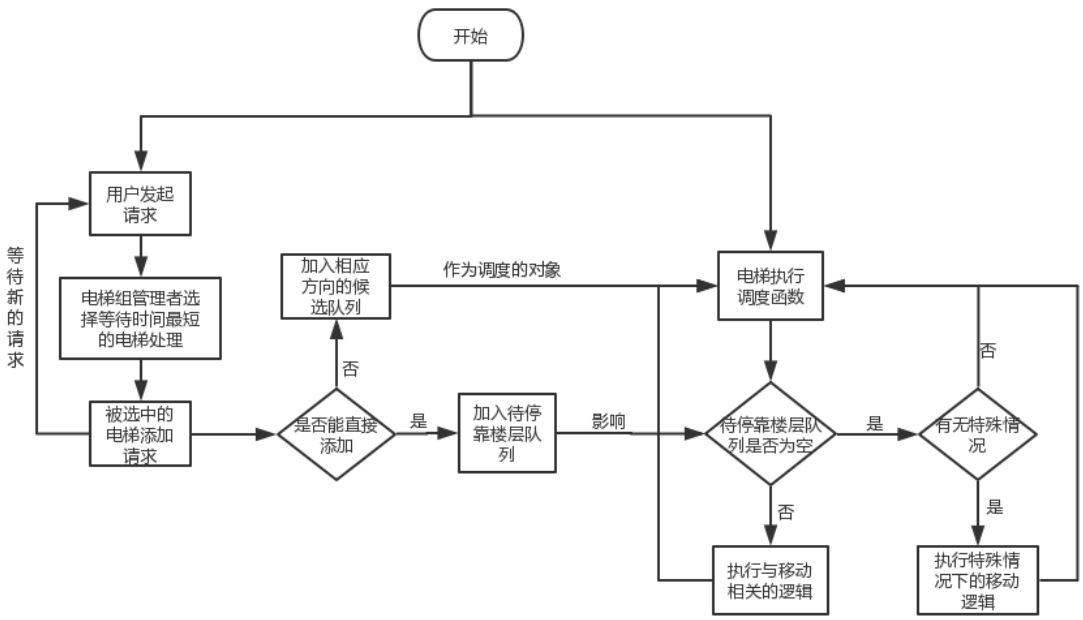
1.简述

本次电梯项目采用C++语言编写并在Visual Studio 2017上开发，图形界面基于Qt平台。基于多线程思想，程序为每一个电梯分配一个线程，**电梯组管理者**也单独有一个线程，当然Qt图形界面也拥有一个线程。本电梯程序的基本运行机制如下：

- 当电梯静止时，优先响应**先请求**的楼层方向
- 当电梯运行时，任何时候电梯**只响应并优先响应**在电梯运动方向上的请求，其他的均加入**候选**停靠楼层队列
- 当一个方向上再无请求时即**调转方向**
- 整体上电梯的运行状态是呈上下**扫描趋势**
- 当接到请求时，**电梯组管理者**会选择在当前状态下使等待时间最小的电梯去处理该请求

每一个电梯在启动后即进入无限循环，检测**当前待停靠楼层队列**，若不为空则开始运作，否则保持静止，而电梯的**当前待停靠队列**是有**电梯组管理者**控制的。每次改变方向时由**电梯调度函数**更新电梯的**当前待停靠楼层队列**，以此电梯系统正常运行。

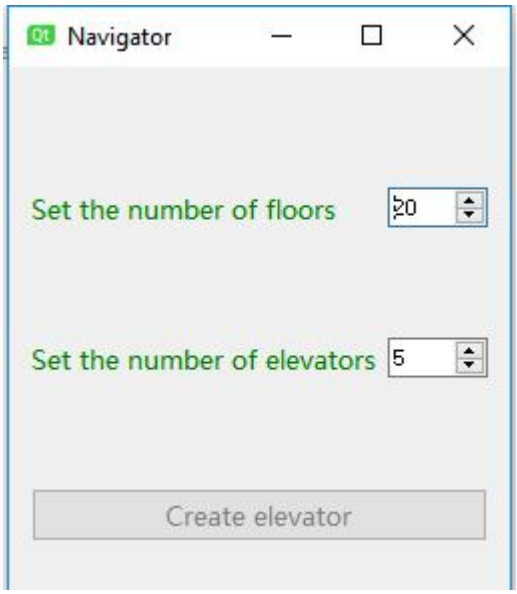
2.流程图



界面

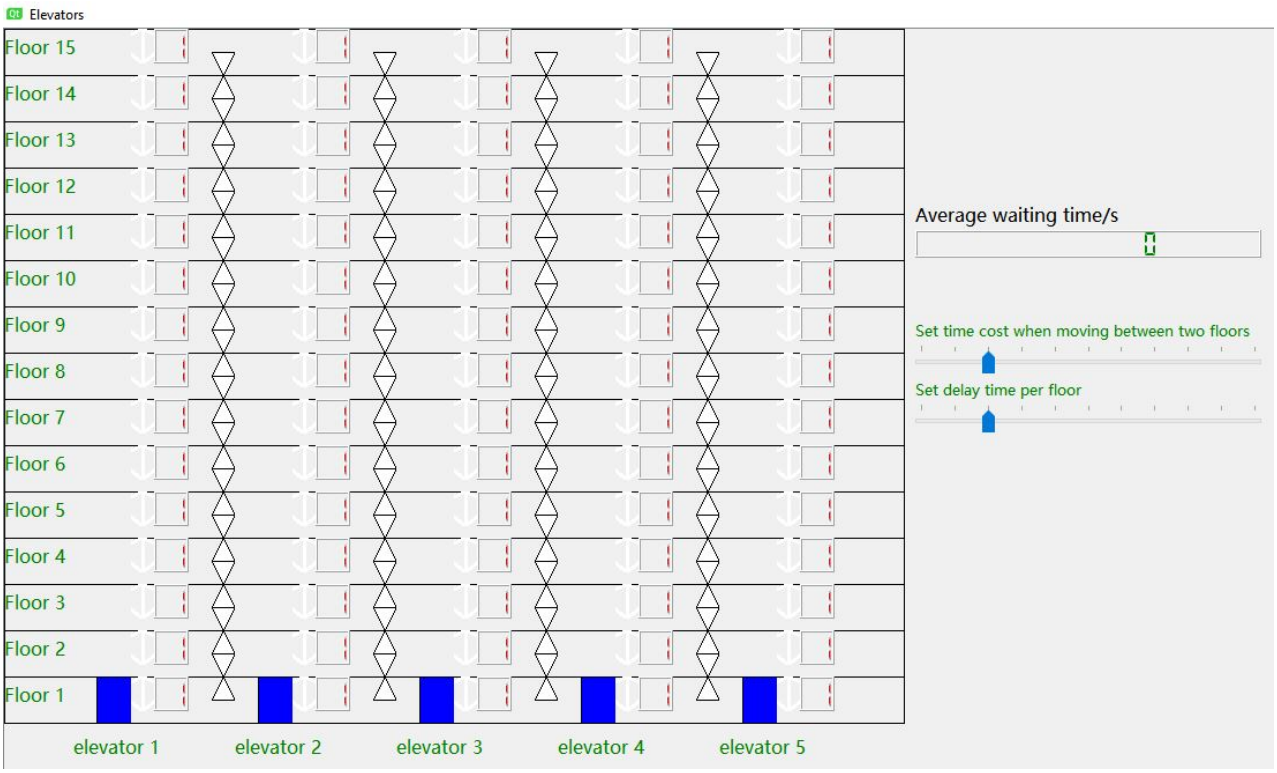
本项目界面分为三个部分：**导航界面**、**电梯界面**、**按钮界面**。导航界面用于设置电梯数量和最高楼层数，电梯界面是在外界控制下电梯运动状态的界面，按钮界面是正乘坐电梯的人看到的按钮盘。

1. 导航界面



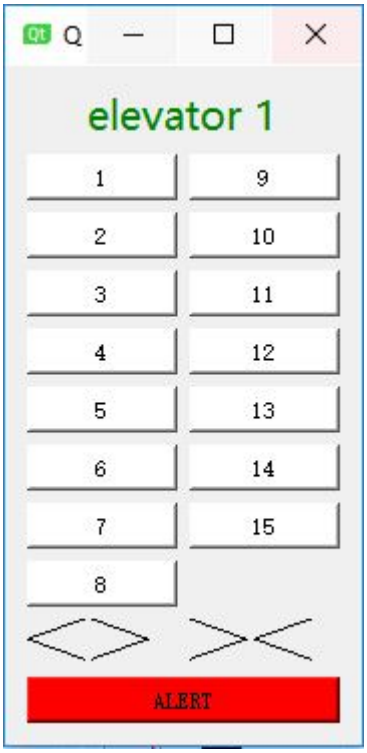
可以看出导航界面默认电梯数和最高楼层数为5和20，在运行过程中随时可以改变设置，生成多个电梯。

2. 电梯界面



这里**蓝色方块**代表电梯，**白色三角形**代表电梯门外的请求方向按钮，右边显示了当前所有请求的平均等待时间。另外在右边可以设置电梯运行的速度，适应不同的用途。

3. 按钮界面



这里有各楼层的按钮、开关门按钮以及报警按钮。由于本程序是按照**自动开关门**设计，没有开关门的相应实现，所以开关门键其实没有作用。

算法

本程序综合了**扫描调度算法**和**最短平均等待时间调度算法**。对每个电梯采用**扫描调度算法**，一次性处理完同一个方向上的所有请求；对所有电梯整体采用**最短平均等待时间调度算法**，选择当前状态下**等待时间预测值最小**的电梯处理当前请求。

- 最短平均等待时间调度算法

```
void elevatorGroupManager::addRequest(int selfFloor, Direction direction)
{
    if (direction == Direction::still)
        return;
    int bestIndex = 0;
    clock_t minWaitingTime = -1;
    for (int i = 0; i < elevatorGroup.size(); i++) //找到当前电梯群中可以使该请求等待时间最小的
    {
        clock_t waitingTime = elevatorGroup[i]->getWaitingTime(selfFloor, direction);
        if (i == 0)
            minWaitingTime = waitingTime;
        if (waitingTime < minWaitingTime)
        {
            minWaitingTime = waitingTime;
            bestIndex = i;
        }
    }
    elevatorGroup[bestIndex]->addRequest(selfFloor, direction);
```

```
    emit(requestHandled(bestIndex, selfFloor));
}
```

- 扫描调度算法

```
void Elevator::schedule()
{
    if (moveDirection == Direction::still)//只在电梯静止时检查是否有候选请求
    {
        if (candidateDownDest.empty() && candidateUpDest.empty())//没有候选请求则不做什么
            return;
        else if (!candidateDownDest.empty())//处理向上的候选请求
        {
            int floor = candidateDownDest.back().floor;
            if (floor > currentFloor)//处理特殊情况
            {
                setMoveDirection(Direction::up);
                targetFloor = candidateDownDest.back().floor;
                targetDirection = Direction::down;
                targetStartTime = candidateDownDest.back().startTime;
            }
            else
            {
                copyDests(candidateDownDest);
                if(floor < currentFloor)
                    setMoveDirection(Direction::down);//改变方向
            }
        }
        else if (!candidateUpDest.empty())//处理向下的候选请求
        {
            int floor = candidateUpDest.front().floor;
            if (floor < currentFloor)//处理特殊情况
            {
                setMoveDirection(Direction::down);
                targetFloor = candidateUpDest.front().floor;
                targetDirection = Direction::up;
                targetStartTime = candidateUpDest.front().startTime;
            }
            else
            {
                copyDests(candidateUpDest);
                if (floor > currentFloor)
                    setMoveDirection(Direction::up);//改变方向
            }
        }
    }
}
```

- 等待时间预测

```
//这里利用当前待停靠楼层队列以及候选停靠楼层队列预测当前请求可能需要等待的最少时间，根据方向的不同和相对位置分为多种情况，下面一一计算了相应的等待时间预测值
clock_t Elevator::getWaitingTime(int selfFloor, Direction direction)
{
    clock_t waitingTimePrediction = 0;
    std::list<dest>::iterator iter;
    if (!destQueue.empty())
    {
        for (iter = destQueue.begin(); iter != destQueue.end(); iter++)
        {
            waitingTimePrediction += delayPerFloor;
        }
    }
    if (moveDirection == Direction::still)
    {
        waitingTimePrediction += abs(currentFloor - selfFloor) * timePerFloor;
    }
    else if (moveDirection == direction)
    {
        if (direction == Direction::down)
        {
            if (selfFloor < currentFloor)
            {
                waitingTimePrediction += (currentFloor - selfFloor) * timePerFloor;
            }
            else
            {
                int finalFloor = destQueue.empty() ? targetFloor :
destQueue.front().floor;
                waitingTimePrediction += (currentFloor - finalFloor) * timePerFloor;
                waitingTimePrediction += (selfFloor - finalFloor) * timePerFloor;
            }
        }
        else
        {
            if (selfFloor > currentFloor)
            {
                waitingTimePrediction += (selfFloor - currentFloor) * timePerFloor;
            }
            else
            {
                int finalFloor = destQueue.empty() ? targetFloor :
destQueue.back().floor;
                waitingTimePrediction += (finalFloor - currentFloor) * timePerFloor;
                waitingTimePrediction += (finalFloor - selfFloor) * timePerFloor;
            }
        }
    }
    else
    {
        if (moveDirection == Direction::down)
        {
```

```
            int finalFloor = destQueue.empty() ? targetFloor :
destQueue.front().floor;
            waitingTimePrediction += (currentFloor - finalFloor + abs(selfFloor -
finalFloor))* timePerFloor;
        }
        else
        {
            int finalFloor = destQueue.empty() ? targetFloor : destQueue.back().floor;
            waitingTimePrediction += (finalFloor - currentFloor + abs(finalFloor -
selfFloor)) * timePerFloor;
        }
    }
    return waitingTimePrediction;
}
```

可行性

- 在电梯静止时，由于不需要处理请求，其无限循环几乎不耗费时间，所以可能会造成图形界面**异常卡顿**。所以在每次循环前让线程**睡眠很小一段时间**可以很好地解决这个问题，本程序设置的睡眠时间为4ms。
- 为了简便，本程序采用**一次移动一层**的方式，加之每次移动前睡眠一小段时间，会使得电梯运动有些生硬，但是还是可以很好地呈现电梯的运动状态。
- 在前面提到过，由于本程序没有设置开关门的动画，所以其实按钮界面的开关门按钮并没有作用。
- 在计算等待时间预测值时，我们采用了调用相应函数那个时刻的状态，但是电梯状态是**动态变化**的，所以可能当前选择处理请求的电梯以后不见得是最好的。但是，在**电梯数量较多，楼层不太高，同时使用人数不太多**的情况下，程序这样处理也是可以接受的。当然，最好可以是动态判断，但是相应的**开销与复杂度的提升**和带来的**改善**是否匹配是值得考虑的问题。

注意事项

- 项目源文件编译需要**Qt环境**
- 尽量**不要生成多个电梯组实例**，会造成卡顿