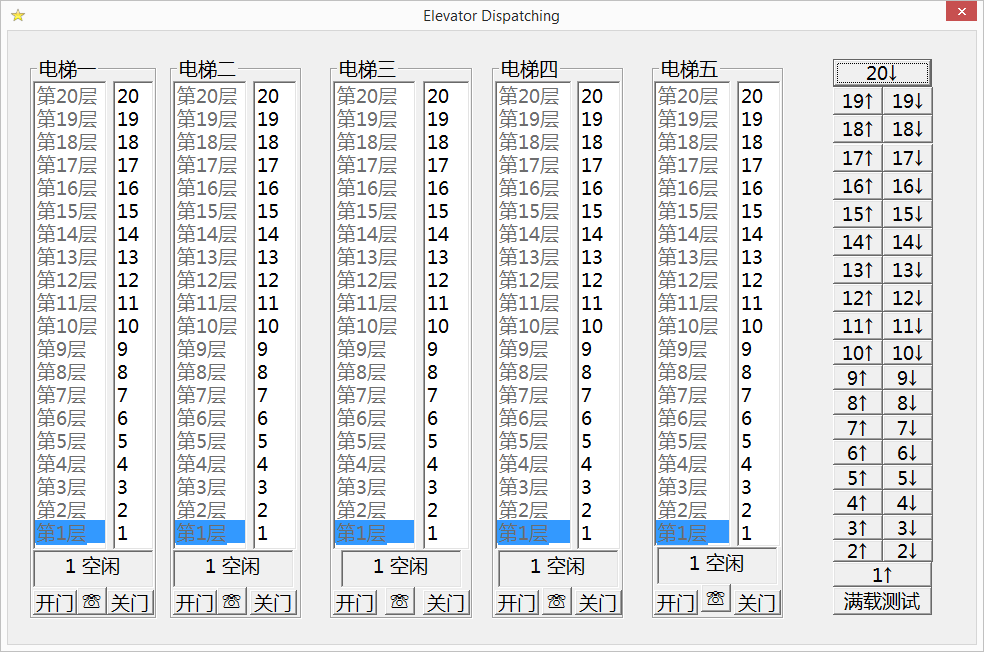
电梯调度项目文档

1. 软件使用介绍。

这个是软件初始状态下的界面：



用户按下报警键之后，该部电梯的全部上下行需求将被清空，电梯基址在原地（模拟电梯出现故障），此时电梯处于报警状态，不再接受上下行任务。只有当用户再次按下该键，解除报警状态，电梯才能恢复正常。

关门键按下之后，电梯的线程将会启动，如果楼层中有上下楼的需求，那么电梯将运动起来，否则电梯将静止在原地。

当用户按下开门键时，如果当前楼层恰好为有需求的楼层，那么电梯将会在该楼层停下（模拟开门操作）。如果当前楼层为无需求层，那么直接忽略开门动作。

同时代表电梯内部，以及每层楼每部电梯门口的数码显示器，指示当前电梯状态。（指示格式为：%d [上升中|下降中|空闲]）

这个是每层楼每部电梯门口的按钮，电梯的外部按钮

代表楼层，蓝色为有电梯停靠的楼层

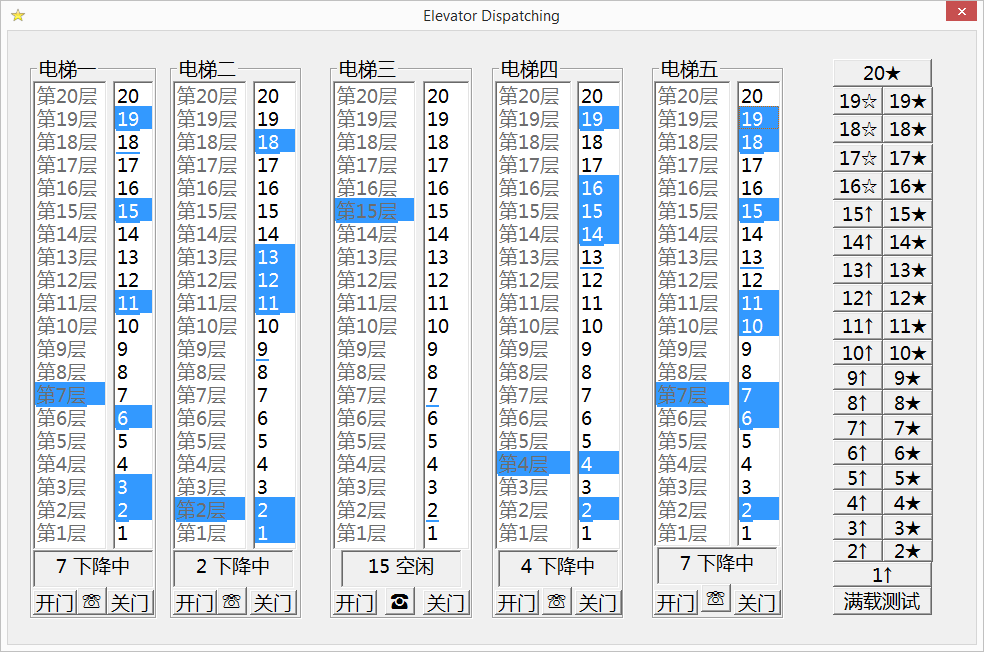
代表电梯的内部按钮，有20层楼，则有20个按钮。用户点击某一项代表需到达哪一层。

该按钮按下之后，电梯的外部按钮将全部点亮，这时我们可以看到满载情况下电梯的调度情况（满负荷情况下的调度）。

在该程序中，用户的外部需求会转化为某一部电梯的内部需求（这就是该程序中的电梯调度）

这个是软件在运行时的界面：

空心星星代表有人在电梯外部按了上行的按钮，且该需求未分配给电梯。



向下箭头的意义由此类推。

向上的箭头表示（1）该楼层没有电梯外部的上行需求；（2）或者有上行的外部需求，但是该需求已分配给五部电梯中的某一部。

实心的星星代表有人在电梯外按了下行的按钮，且该需求未分配给电梯。

实时显示电梯当前状态

表示当前电梯所处的位置

蓝色的格子就代表当前有需求的楼层，当电梯到达需求楼层后，格子的蓝色将被取消，表示需求已被满足。

电梯到达需求层后将会暂停1.5秒，模拟用户进出电梯。

电梯处于报警状态，不接受任何上行下行需求（电梯线程（计时器）暂停）。当电梯解除报警状态时，电梯线程（计时器）自动恢复。

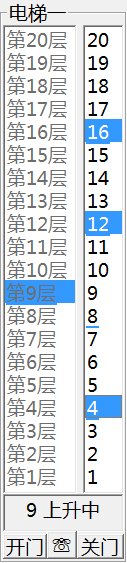
1. 具体实现思路。

程序里封装了一个Elevator类，类中包含电梯运行的一些基本信息，在类中有一个指向长度为21的bool数组的指针（Elevator类的私有成员），该数组中记录了该电梯需要到达那些楼层。楼层与数组下标一一对应（楼层下标从一开始），电梯需到达楼层对应的数组下标处的值将被置为true，否则为false。当电梯到达需求层后会将该层对应bool数组下标中的值置为false，表示该层需求已被满足。

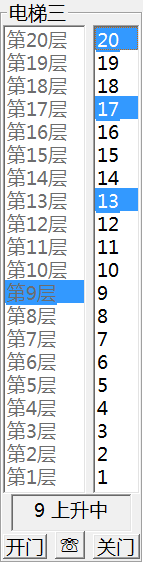
类似的，在Elevator的静态成员中有两个指向长度为21的bool数组的指针，一个代表的是电梯外部的上行需求，另一个是电梯外部的下行需求。同样的值为true表示该楼层有外部需求，否则值为false，当某一层的外部需求分配给某一部电梯之后，数组中对应处的值将置为false，表示该外部需求已分配出去了。

**（1）关于电梯处理自身的内部需求。**

在电梯满足自身的需求的时候（遍历电梯类里的内部需求数组）采用了类似look算法的思想，即按照楼层的顺序依次服务需求，使得电梯在最底层与最顶层之间往返运行（当发现电梯所移动的方向上不再有请求时立即改变运行方向）。



即如左图所示，此时电梯正处于上行状态，4,12，16层为需求层，那么该电梯将会先满足12,16层的请求，到达16层后，电梯发现16层之上已经没有需求，因此电梯改变方向为下行，满足4层的需求。电梯到达四层后若此时已没有内部需求，那么电梯将停止在第四层。电梯正处于下行的情况也类似，这时电梯会先满足当前楼层之下的需求，到达需求最底层后在转向为上行，满足其它层的需求（如果有的话）



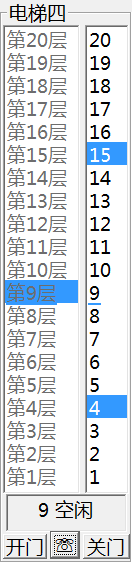
电梯原来处于静止态（静止在某一层），若此时来的需求都在当前楼层之上，那么电梯上行满足需求。类似的，电梯原来处于静止态（静止在某一层），这次来的需求都在当前楼层之下，那么电梯下行满足需求。

**如下是当电梯处于静止状态时，在当前楼层上下同时有请求出现时的情况。**模拟此情况的操作步骤：（1）先使所有电梯处于报警状态，在选择其中的某一个电梯进行试验可见当前电梯正处于第九层；

（2）再按下外部需求按钮（当前楼层9之上）（当前楼层9之上）（我选择的是外部下行需求按钮）；

（3）解除试验电梯的报警状态（但是并不解除其他电梯的报警状态），这是电梯线程（计时器将会）将会启动，15和4两个需求将会被同时分配到试验电梯（因为其他电梯都处于报警状态，不可用，那么任务只能由试验电梯——唯一可用的电梯接收）。

经过以上三步后我们可以得到如下状态（下图所示为程序中对此种情况的处理办法）：



目标：所有用户平均等待时间最短

T\_H\_wait T\_L\_wait T\_H T\_L为楼层距离

分析：

（1）电梯先满足H的需求

T\_H\_wait=H-C T\_L\_wait=H-C+H-L

T\_H= T\_H\_wait+ T\_L\_wait=3H-2C-L

（2）电梯先满足L的需求

T\_H\_wait=C-1+H-1 T\_L\_wait=C-L

T\_L= T\_H\_wait+ T\_L\_wait=H+2C-L-2

那么电梯要上行还是下行就取决于那个调度方案所需的时间少：

1. T\_H<T\_L 即 H-2C<-1时电梯要先满足当前楼层之上的需求；
2. T\_H>T\_L 即H-2C>-1时电梯应该满足当前楼层之下的需求；
3. T\_H==T\_L 即H-2C==-1这时电梯选择上行或下行无所谓，因为耗费的总时间相同。

**L**

**C**

**H**

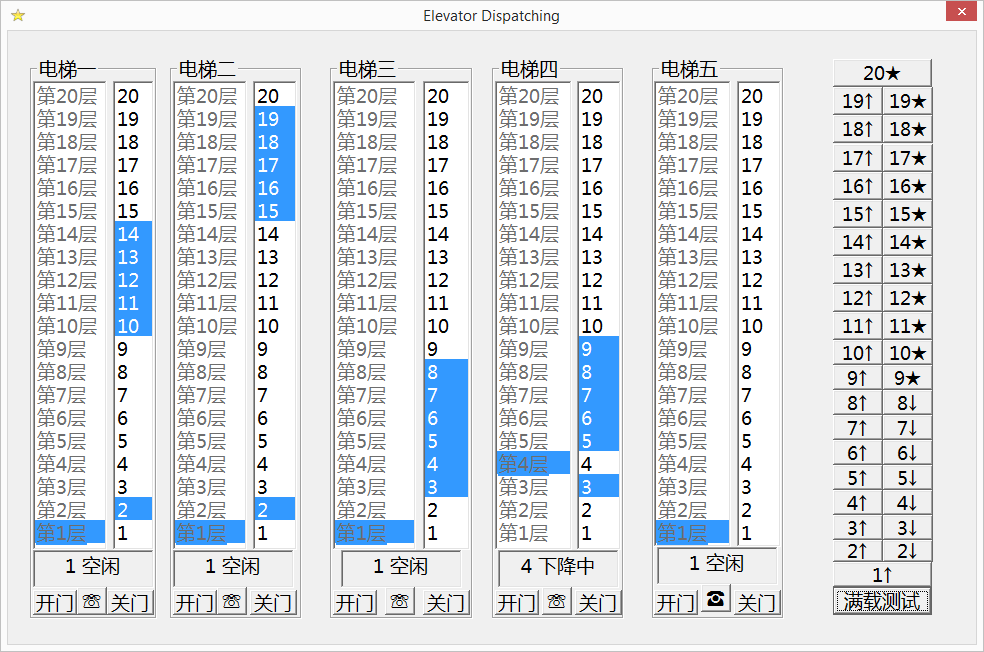
**（2）关于电梯外部需求的分配（将某一个外部需求如何分配给某一个电梯）**

在程序中将外部需求分配给某一个电梯 即为 将该外部需求转化为哪一部电梯的内部需求，在程序中我用**内部需求量来模拟电梯中现有的人数（我假定内部需求量越多，那么电梯中的人越多），在该程序中每个电梯的内部需求量不能超过六个，否则就会引起“超重”。程序中某一层外部需求转化为内部需求的规则如下：**

1. **优先考虑当前电梯中的内部需求总量**，当前内部需求总量小于等于6的电梯满足要求，如果没有电梯满足该要求，那么下次再分配该外部需求；
2. **在满足第一个条件的电梯范围内**，再考虑外部需求楼层与电梯当前楼层的距离因素：
3. 外部需求为上行需求：
4. 优先找当前正处于上行状态 并且 该电梯的当前楼层在外部需求楼层之下 的电梯，若有多部电梯满足要求，那么取距离最近的一个电梯接收该外部需求；
5. 若没有电梯满足第一个要求，那么从当前正处于静止状态的电梯中找出离外部需求楼层最近的电梯；从当前正处于下行状态（该下行状态电梯的当前楼层要在外部需求楼层之下） 并且 该电梯的最低内部需求楼层 离外部需求楼层最近的电梯；然后比较找到的两个电梯距 外部需求楼层的距离，取距离较小的那个电梯接收该外部需求。如果没有电梯满足要求，那么下次再分配该外部需求。
6. 外部需求为下行需求：
7. 优先找当前正处于下行状态 并且 该电梯的当前楼层在外部需求楼层之上 的电梯，若有多部电梯满足要求，那么取距离最近的一个电梯接收该外部需求；
8. 若没有电梯满足第一个要求，那么从当前正处于静止状态的电梯中找出离外部需求楼层最近的电梯；从当前正处于上行状态（该上行状态电梯的当前楼层要在外部需求楼层之上） 并且 该电梯的最顶内部需求楼层 离外部需求楼层最近的电梯；然后比较找到的两个电梯距 外部需求楼层的距离，取距离较小的那个电梯接收该外部需求。如果没有电梯满足要求，那么下次再分配该外部需求。

以上便是将外部需求分配给某一个电梯的做法。下面来看一下效果，操作步骤：

1. 选择需要进行试验的电梯，将确保这几个电梯处于非报警状态，其他未参加试验的电梯全部处于报警状态（使他们不能接受外部需求）；
2. 然后按下程序窗口右下角的按钮，测试开始（效果图如下）：



已经被分配出去的外部需求

报警电梯不参与试验，不接受外部需求

参加试验的电梯内部需求最大为6

由于条件限制还未被分配出去的外部需求

程序的所有功能大致就是这样。这个程序里最主要的思想就是将电梯的外部需求与电梯的内部需求分开，每个电梯子线程专注于处理自身的内部需求，主线程则负责将电梯的外部需求转化为内部需求。

**三、简要代码实现。**

关于具体的代码实现，程序源码里有详细的注释。

（1）程序最主要的是Elevator类，其中bool\* lpbInnerRequirementFloors; 为存放电梯内部需求数组的指针；static bool\* lpbOuterRequestUp;//记录电梯外部需求中需要上楼的需求，true表示有该需求。

static int iOuterRequestUps;//外部上楼需求一共有多少个。

static bool\* lpbOuterRequestDown;//记录电梯外部需求中需要下楼的需求，true表示有该需求。

static int iOuterRequestDowns;//外部上楼需求一共有多少个。

这几个静态成员代表的是电梯的外部上下行需求的bool数组，以及上下行数组分别得总量。

（2）实现电梯对自身内部需求满足的主要为这三个函数（Elevator.h）

//三种状态下的信息运行时的不同动作。

//上升状态下。

bool UpingElevatorAction(Elevator\* lpElevator, int iInnerRequestCount);

//下降状态下。

bool DowningElevatorAction(Elevator\* lpElevator, int iInnerRequestCount);

//静止状态下。

bool StillElevatorAction(Elevator\* lpElevator, int iInnerRequestCount);

具体实现源码中有详细注释（就是按照上述思想实现的）。

（3）主线程向五个电梯子线程分配任务。

//分配外部需求，将电梯外部需求转化为电梯内部需求。

//对Elevator里面的外部需求数组进行扫描。分配任务。

bool DispatchOuterRequtst(Elevator\*\* lpMyElevator);

（ElevatorFunc.h中，最主要的分配外部需求函数，在其中会调用以下三个函数。）

//将外部需求发送给自己楼下正在上升的电梯。

//返回true表示任务成功分配。

//iRequestFloor 发出请求的楼层，iUpingCount当前已有的正在上升的电梯数，lpUpingElevators是那几部电梯。

bool SendToUnderFloorUping(Elevator\*\* lpMyElevator, int iRequestFloor, int iUpingCount, int\* lpUpingElevators);

//2.(1)找正在下行的，且下行底线离自己绝对值最小；(2)找正静止的，且离自己距离绝对值最小；取二者中值较小的那部电梯执行任务。

bool SendToDowningOrStill(Elevator\*\* lpMyElevator, int iRequestFloor, int iDowningCount, int\* lpDowningElevators, int iStillCount, int\* lpStillElevators);

//将外部需求发送给自己楼上正在下降的电梯。

bool SendToAboveFloorDowning(Elevator\*\* lpMyElevator, int iRequestFloor, int iDowningCount, int\* lpDowningElevators);

//2.(1)找正在上行的，且上行顶限离自己绝对值最小；(2)找静止的，且离自己绝对值最小；取二者中绝对值较小的那部电梯执行任务。

bool SendToUpingOrStill(Elevator\*\* lpMyElevator, int iRequestFloor, int iUpingCount, int\* lpUpingElevators, int iStillCount, int\* lpStillElevators);

（三个函数都在ElevatorFunc.h中，具体实现代码里有详细注释，它们与（二）中描述的电梯外部需求分配的思想相对应）

其他如报警键、开门键、关门键等等的实现代码中有详细注释。因为写起来字数太多，就不写在文档上了。