中国科学技术大学计算机学院
《计算机系统概论实验报告》

实验题目：电梯模拟

学生姓名：叶子昂

学生学号：PB20020586

完成时间：2021年12月29日

实验要求

0.1 基本要求

（1）模拟某校五层教学楼的电梯系统。该楼有一个自动电梯，能在每层停留。五个楼层由下至上依次称为地下层、第一层、第二层、第三层和第四层，其中第一层是大楼的进出层，即是电梯的“本垒层”，电梯“空闲”时，将来到该层候命。

（2）乘客可随机地进出于任何层。对每个人来说，他有一个能容忍的最长等待时间，一旦等候电梯时间过长，他将放弃。

（3）模拟时钟从 0 开始，时间单位为 0.1 秒。人和电梯的各种动作均要耗费一定的时间单位(简记为 t)，比如：

有人进出时，电梯每隔 $40t$ 测试一次，若无人进出，则关门；

关门和开门各需要 $20t$；

每个人进出电梯均需要 $25t$；

如果电梯在某层静止时间超过 $300t$，则驶回 1 层候命。

（4）按时序显示系统状态的变化过程：发生的全部人和电梯的动作序列。

0.2 完成的要求

- 基本要求
- 双电梯以及任意指定楼层的模拟

0.2.1 输入：待程序提示输入时对应输入模拟时间数，楼层数

0.2.2 输出：程序依时间顺序输出电梯和乘客动作

设计思路

0.3 代码设计思路

0.3.1 时间驱动

time做为计数器，每次循环time++,在循环过程中当time与某事件发生时间相同则执行该事件。

```
while (time < MRuningtime) {
    if (Co.Ctime.intertime == time) {//生成乘客到来事件。
        ID++;
        p = Co.peoarrive(ID, Co.Ctime.intertime);
             Co.Inquene(Co.List[p->get_InFloor()][p->get_arrow()], p, Co.Call
            [p->get_InFloor()][p->get_arrow()]);
    }
    Co.LiftH();//电梯调度
    Co.LiftRun(time, 0);//一号电梯运行
    Co.LiftRun(time, 1);//二号电梯运行
    Co.peplgiveup_check(time);//乘客放弃事件
    time+=1;
}
```

0.3.2 代码结构

整个模拟系统由CoCenter Co类代表，由mian函数通过time时间驱动CoCenter。

- CoCenter中有以下子类：

```
Time Ltime[2];//电梯事件时间表记录电梯事件发生时间
Time Ctime;//系统事件时间表记录系统事件发生时间
LiftN Lift[2];//两个电梯
QueneList** List;//乘客排队队列根据楼层动态分配
LiftStack* Stack[2];//电梯中的乘客栈根据楼层动态分配
LiftROrder RunOrder[2];//电梯目的楼层表
int** Call;//楼层请求根据楼层动态分配
```

- 电梯类：

```
class LiftN {
private:
	int Floor;
	// int D1;
	// int D2;
	// int D3;
	int State;//IDLE, GOINGUP,GOINGDOWN
	int arState;//电梯优先运行方向
	int Rstate;//上升或下降用于判断加速减速平稳运行的状态
	int Ostate;//待机与否状态
public:
	int waitstate;//到达目的楼层运行状态用于IDLE
	// int upfloor;//将上行目的楼层
	// int downfloor;//将下行目的楼层
	LiftN();
	int Liftret(int i);
	int& get_Floor() { return Floor; }
	int& get_state() { return State; }
	int& get_arState() { return arState;}
	int& get_Rstate() { return Rstate; }
	int& get_Ostate() { return Ostate; }
	Status change_state(int s) { State = s; return OK; }
	Status change_arState(int s) { arState = s; return OK;}
	Status change_Rstate(int s) { Rstate = s; return OK; }
	Status change_Ostate(int s) { Ostate = s; return OK; }
	//
	Status UpFloor(){Floor++; return OK;}
	Status DownFloor(){Floor--; return OK;}
	Status Back(){Floor=0; State=Idle; return OK;}
	Status Waitin() { State = Idle; return OK; }
};
```

- 乘客类

```
typedef class passanger {
private:
    int ID;//身份
    int InFloor;//进入楼层
    int OutFloor;//目的楼层
    float GiveupTime;//放弃时间
    int arrow;//目的方向
public:
    passanger();
    passanger(int a,float t);
    int& get_arrow() { return arrow; }
    int& get_InFloor() { return InFloor; }
    int& get_OutFloor() { return OutFloor; }
    int& get_ID(){return ID;}
    int timein_check(float t);
    friend class waitingquene;
    friend class Head;
    friend class LiftStack;
}Person;
```

- 乘客队列

```
typedef class waitquene//节点
{
public:
    Person* Pe;
    class waitquene* next;
    waitquene();
}QNode,*Pquene;

typedef class Head{
    Pquene front;
    Pquene rear;
public:
    Head();
    int checknull() { if (front->next == NULL)return 0; else return 1; }
    Status Enquene(Person* &P);//入队
    Person* Dequene();//出队
    Status Giveup();
    Status timeout_check(float t);//处理乘客放弃事件
}QueneList;
```

- 乘客栈

```
typedef class Nstack {//节点
public:
```

```
    Person* p;
    class Nstack* next;
}*NS;

class LiftStack {
public:
    NS base;
    NS top;
    int   num;
    LiftStack();
    Status Push(Person*& P);
    Status Pop();
};
```

- 时间表类

```
class Time
{
public:
    float intertime;//乘客进入放弃记录
    float givetime;

    float inouttime;//进出时间记录

    float utime;//上升下降事件记录
    float dtime;

    float opendotime;//IDLE处理时间记录
    float checdotime;
    float clodotime;

    float backtime;//电梯久无任务返回时间记录
    Time();
    Status timeret()
{intertime=0;givetime=0;inouttime=0;utime=0;dtime=0;opendotime=0;checdotime=0;back
time=0;return OK;}
};
```

- 电梯目的楼层表

```
typedef struct H {//指令节点
    int F;
    int arrow;
    struct H* next;
    H();
}*pHNode;

struct LiftROrder {//指令链表
```

```
        pHNode head;
        LiftROrder();
        int& get_arrow() { return head->next->arrow; }
        int& get_Ofloor() { return head->next->F; }
        Status change_arrow(int ar){ head->next->arrow = ar; return OK; }
        Status OrderInsert(int n,int ar );//根据楼层方向插入指令链表
        Status OrderDone();
        Status OrderClear();
        int OrderNull();
    };
```

关键代码分析

0.4 乘客到来事件

0.4.1 时间等于下一位乘客到来时触发，生成乘客同时进入等待队列

```
if (Co.Ctime.intertime == time) {
            ID++;
            p = Co.peoarrive(ID, Co.Ctime.intertime);
                 Co.Inquene(Co.List[p->get_InFloor()][p->get_arrow()], p, Co.Call[p-
  >get_InFloor()][p->get_arrow()]);
       }//进入队列等待
```

0.4.2 乘客生成（随机生成乘客各项属性和下一位乘客到来时间加入系统时间表）

```
Person* ConCenter::peoarrive(int i,float &t)
{
    Person* p=NULL;
    p = CrPerson(i, t);
    Call[p->get_InFloor()][p->get_arrow()]=1;
    printline();
    cout<<"第"<<p->get_ID()<<"号乘客将在第"
        <<p->get_InFloor()<<"层排队等待乘坐电梯，准备前往第"
       <<p->get_OutFloor()<<"层楼"<<endl;
    cout<<"下一位乘客将在"<<t<<"时前来"<<endl;
    return p;
}
Person* CrPerson(int i,float &t)
{
    Person* per = new Person(i, t);
    if (!per)exit(OVERFLOW);
    t += CrRandom(10, 15) * timewide;
    // t+= 11;调试使用
    return per;
}
```

```
passanger::passanger(int a,float t)
{
    ID = a;
    // InFloor=0;     //调试使用
    // OutFloor = 3;
    InFloor = CrRandom(0,tofloor-1);
    do
        OutFloor = CrRandom(0, tofloor-1);
      while (OutFloor == InFloor);//防止目的楼层与进入楼层相同
    GiveupTime = t + CrRandom(8, 20) * timewide;
    if (OutFloor > InFloor)arrow = up;
    else arrow = down;
}
```

0.5 电梯的调度

0.5.1 电梯调度原则

- 单电梯调度原则
  - 当前运行方向为请求优先接受方向。
  - 电梯内人员请求具有较高优先级。
  - 电梯根据请求优先接受方向获取最高/低楼层请求，并向其运行。
  - 顺路原则，在向3中请求运行中时，中间楼层的请求会被顺路响应。
  - 在一方向上的请求完成时，切换请求优先接受方向，若无请求，则进入等待返回状态。
  - 等待返回状态结束进入返回待命状态后，回到初始楼层前不响应任何请求。
- 双电梯调度原则
  - 请求分配到处于待命的电梯，则电梯启动并响应该请求。
  - 由调度函数轮询电梯，若请求在电梯顺路的中间楼层上则不予干涉。若不在电梯电梯顺路的中间楼层上则判断其他电梯是否空闲，若空闲则分配任务。若无电梯空闲，则等待电梯空闲在分配该请求。
  - 电梯可能顺路完成另一电梯的任务，此时另一电梯不在响应该请求。

0.5.2 代码实现

```
Status ConCenter::LiftH()
{
    int ar=-1;
    int u=FindCalluphighest(Lift[0].get_Floor());
    int d=FindCalldownlowest(Lift[0].get_Floor());
    int u1=-1, d1=-1;
    if(Lift[0].get_arState()==up){
        u1=FindCalluphighest(Lift[1].get_Floor(),0,Lift[0].get_Floor()-1);
        d1=FindCalldownlowest(Lift[1].get_Floor(),0,Lift[1].get_Floor()-1);
    }
    else{
        u1=FindCalluphighest(Lift[1].get_Floor(),Lift[0].get_Floor()+1,tofloor-1);
        d1=FindCalldownlowest(Lift[1].get_Floor(),Lift[0].get_Floor()+1,tofloor-1);
    }
    if(Lift[0].get_state()!=reset){
```

```
    if(u==-1&&d==-1)return OK;
    if(Lift[0].get_Ostate()==WAIT){
        Lift[0].change_Ostate(RUN);
        if(u!=-1){//启动一号电梯
            ar=arrow_conculate(0,u);
            RunOrder[0].OrderInsert(u,ar);
            Lift[0].change_state(arrow_conculate(0,u));
            Lift[0].change_arState(up);
        }
        cout<<"一号电梯启动！"<<endl;
    }
    else{
        if(Lift[0].get_arState()==up){
            if(RunOrder[0].OrderNull()){
                if(u == -1 || u == Lift[0].get_Floor()){
                    Lift[0].change_arState(down);//转变请求响应方向
                    if(u == Lift[0].get_Floor()){
                    RunOrder[0].OrderInsert(Lift[0].get_Floor(),down);
                    }
                }
                else RunOrder[0].OrderInsert(u,up);//分配请求
            }
        }
        else {
            if(RunOrder[0].OrderNull()){
                if((d == -1) || (d == Lift[0].get_Floor())){
                    Lift[0].change_arState(up);//转变请求响应方向
                    if(d == Lift[0].get_Floor()){
                    RunOrder[0].OrderInsert(Lift[0].get_Floor(),up);
                    }
                }
                else RunOrder[0].OrderInsert(d,down);//分配请求
            }
        }
    }
}
if(Lift[1].get_state()==reset)return OK;
else{
    if(u1==-1&&d1==-1)return OK;
    if(Lift[1].get_Ostate()==WAIT){
        Lift[1].change_Ostate(RUN);
        if(u!=-1){
            ar=arrow_conculate(0,u1);
            RunOrder[1].OrderInsert(u1,ar);
            Lift[1].change_state(arrow_conculate(0,u1));
            Lift[1].change_arState(up);
        }
        cout<<"二号电梯启动！"<<endl;
```

```
        }
        else{
            if(Lift[1].get_arState()==up){
                if(RunOrder[1].OrderNull()){
                    if(u1 == -1 || u1 == Lift[1].get_Floor()){
                        Lift[1].change_arState(down);//转变请求响应方向
                        if(u1 == Lift[1].get_Floor()){
                        RunOrder[1].OrderInsert(Lift[1].get_Floor(),down);
                        }
                    }
                    else RunOrder[1].OrderInsert(u1,up);//分配请求
                }
            }
            else {
                if(RunOrder[1].OrderNull()){
                    if(d1 == -1 || d1 == Lift[1].get_Floor()){
                        Lift[1].change_arState(up);//转变请求响应方向
                        if(d1 == Lift[1].get_Floor()){
                        RunOrder[1].OrderInsert(Lift[1].get_Floor(),up);
                        }
                    }
                    else RunOrder[1].OrderInsert(d1,down);//分配请求
                }
            }
        }
    }
    scerror0();
    return OK;
}
```

0.6 电梯的运行

0.6.1 运行过程中的状态

- 待命状态：无请求在初始楼层待命
- 上升状态
  - 加速状态
  - 平稳状态
  - 预进入目的楼层减速状态
- 下降状态
  - 加速状态
  - 平稳状态
  - 预进入目的楼层减速状态
- 楼层等待状态
  - 开门状态
  - 人员进出状态

- 电梯反向运行检查
- 关门状态同时根据调度电梯设置接下来如何运行状态
- 关门后无请求等待返回状态

- 返回待命指令执行状态
  - 加速状态
  - 平稳状态
  - 预进入目初始楼层减速状态
  - reset电梯状态和时间表

0.6.2 代码实现

```
Status ConCenter::LiftRun(float t,int i)
{
if(Lift[i].get_Ostate()==WAIT)return OK;
if (Lift[i].get_state() == GoingUp) {//执行上升指令
      if (Lift[i].get_Rstate() == preste) {
            Ltime[i].utime = t + uptime + prestetime;
            Lift[i].change_Rstate(steady);
      }
      if (abs(Lift[i].get_Floor() - RunOrder[i].get_Ofloor()) ==
1&&Lift[i].get_Rstate()==steady) {
            Ltime[i].utime += prestetime;
            Lift[i].change_Rstate(goingstop);
      }
      if (t == Ltime[i].utime) {
            Lift[i].UpFloor();
            Ltime[i].utime += uptime;
      }
      if (Lift[i].get_Floor() - RunOrder[i].get_Ofloor() ==
0&&Lift[i].get_Rstate()==goingstop) {
            Lift[i].change_state(Idle);
            Lift[i].change_Rstate(preste);
            Ltime[i].timeret();
            Lift[i].waitstate = 0;
      }
      else{
            if(Call[Lift[i].get_Floor()][Lift[i].get_arState()]==1){
                  Lift[i].change_state(Idle);
                  Lift[i].change_Rstate(preste);
                  Ltime[i].timeret();
                  Lift[i].waitstate = 0;
            }
      }
}
if(Lift[i].get_state() == GoingDown) {//执行下降指令
      if (Lift[i].get_Rstate() == preste) {
            Ltime[i].dtime = t + downtime + prestetime;
            Lift[i].change_Rstate(steady);
```

```
      }
       if (abs(Lift[i].get_Floor() - RunOrder[i].get_Ofloor()) ==
1&&Lift[i].get_Rstate()==steady) {
             Ltime[i].dtime += prestetime;
             Lift[i].change_Rstate(goingstop);
      }
       if (t == Ltime[i].dtime) {
             Lift[i].DownFloor();
             Ltime[i].dtime += downtime;
      }
       if (Lift[i].get_Floor() - RunOrder[i].get_Ofloor() ==
0&&Lift[i].get_Rstate()==goingstop) {
             Lift[i].change_state(Idle);
             Lift[i].change_Rstate(preste);
             Ltime[i].timeret();
             Lift[i].waitstate = 0;
      }
       else{
             if(Call[Lift[i].get_Floor()][Lift[i].get_arState()]==1){
                   Lift[i].change_state(Idle);
                   Lift[i].change_Rstate(preste);
                   Ltime[i].timeret();
                   Lift[i].waitstate = 0;
             }
      }
}
if (Lift[i].get_state() == Idle) {//执行电梯在楼层停顿及乘客进出指令
       switch (Lift[i].waitstate) {
       case 0: Lift[i].waitstate = 1; Ltime[i].opendotime = t+timeopen;
Ltime[i].inouttime = t + timeopen + 1;
                   Call[Lift[i].get_Floor()][Lift[i].get_arState()]=0; break;//开门
       case 1: if (t >= Ltime[i].opendotime) {//人员进出
             if (!peoinout(Stack[i][Lift[i].get_Floor()], List[Lift[i].get_Floor()]
[Lift[i].get_arState()], i, t) && (t - Ltime[i].inouttime >= 1)) {
                   Lift[i].waitstate = 2;
                   Ltime[i].clodotime = t + closetime;

  if(Lift[i].get_Floor()==RunOrder[i].get_Ofloor())RunOrder[i].OrderDone();
             }
      }break;
       case 2: if (t >= Ltime[i].clodotime) {//关门
             if (RunOrder[i].OrderNull()) {
                   Ltime[i].backtime = t + 5;
                   Lift[i].waitstate = 3;
             }
             else {
                   if (RunOrder[i].get_Ofloor()==Lift[i].get_Floor()){
                         Lift[i].waitstate=1;Ltime[i].inouttime =t + iotime;
```

```
                        Call[Lift[i].get_Floor()][Lift[i].get_arState()]=0;
                }
                 else{
                        if (RunOrder[i].get_arrow() == up)Lift[i].change_state(GoingUp);
                        if (RunOrder[i].get_arrow() ==
down)Lift[i].change_state(GoingDown);
                }
                 break;
           }
     } break;
      case 3: if (t == Ltime[i].backtime) {//等待返回
            if (Lift[i].get_Floor() == 0)
{Lift[i].Liftret(i);RunOrder[i].OrderClear();cout<<"当前时间为"<<t<<endl;}
            if (Lift[i].get_Floor() != 0) {
                  Lift[i].change_state(reset);
           }
     } break;
      default:exit(ERROR);
     }
}
if(Lift[i].get_state() == reset) {//执行返回待命请求
      if (Lift[i].get_Rstate() == preste) {
            Ltime[i].dtime = t + downtime + prestetime;
            Lift[i].change_Rstate(steady);
     }
      if (Lift[i].get_Floor() == 1&&Lift[i].get_Rstate()==steady) {
            Ltime[i].dtime += prestetime;
            Lift[i].change_Rstate(goingstop);
     }
      if (t == Ltime[i].dtime) {
            Lift[i].DownFloor();
            Ltime[i].dtime += downtime;
     }
      if (Lift[i].get_Floor() == 0&&Lift[i].get_Rstate()==goingstop) {
            Lift[i].Liftret(i);
            RunOrder[i].OrderClear();
            cout<<"当前时间为"<<t<<endl;
            Ltime[i].timeret();
     }
}
return OK;
}
```

0.7 人员的进出以及过长等待放弃

0.7.1 实现原则

- 人员先出后进
- 每个人的进出花费1s的时间
- 放弃检查针对所有请求不为空的队列进行

0.7.2 代码实现

0.7.2.1 人员进出

```
int ConCenter::peoinout(LiftStack& S, QueneList& L, int i, float t)
{
    int check = 0;
    if (Ltime[i].inouttime == t) {
        if (S.top != S.base) {//电梯内人出栈
            S.Pop();
            cout<<"当前时间为"<<t<<endl;
            Ltime[i].inouttime += iotime;
            check = 1;
        }
        else if (L.checknull()) {//先“出”后“进”
            Person* P=L.Dequene();
            Stack[i][P->get_OutFloor()].Push(P);
            cout<<"当前时间为"<<t<<endl;
            RunOrder[i].OrderInsert(P->get_OutFloor(), P->get_arrow());
            Ltime[i].inouttime += iotime;
            check = 1;
        }
    }
    return check;//返回进出是否已完成
}
```

0.7.2.2 放弃检查

```
Status ConCenter::peplgiveup_check(float& t){
    for(int i = 0; i <2;i++){
        for(int j = 0;j<tofloor-1; j++){
            if(Call[i][j]==1){
                List[i][j].timeout_check(t);
            }
        }
    }
    return OK;
}

Status Head::timeout_check(float t){
    if(!checknull())return OK;
    Pquene p=front->next;
    Pquene ptemp= front;
```

```
    for(;p;p=p->next,ptemp=ptemp->next){
        if(t>=p->Pe->GiveupTime){
            ptemp->next = p->next;
            cout<<"第"<<p->Pe->get_ID()<<"号乘客已放弃"<<endl;
            if(!ptemp)break;
            free(p);
            p=ptemp->next;
            if(!p)break;
        }

    }
    return OK;
}
```

0.8 任意楼层的实现(相对于源代码的调整)

0.8.1 动态分配

- 由于程序开始前无法得知楼层数则在CoCenter中固定声明如队列,栈的个数,所以在CoCenter中声明为指针,待用户指定楼层数后,通过构造函数动态分配.
- 调整各以来楼层数的函数的判断范围.

0.8.2 构造函数代码(其他为涉及楼层变化范围判断函数（太多）不单独展示)

```
ConCenter::ConCenter(){
    List = new QueneList* [tofloor];
    Call = new int* [tofloor];
    for(int i=0;i<2; i++)Stack[i] = new LiftStack [tofloor];
    for(int i = 0;i<tofloor;i++){
        Call[i] = new int [2];
        List[i] = new QueneList [2];
    }
}
```

调试分析

0.9 代码的时间空间复杂度分析

> 本电梯模拟的问题规模为模拟时间，乘客移动操作的时空复杂度涉及的为数据结构中的乘客人数，电梯操作的时空复杂度涉及的为楼层数，乘客人数是随机生成的与随机方式相关，楼层数需输入，不便分析时空复杂度，此处仅对代码的时空效率做定性的分析。

0.9.1 空间复杂度

程序初运行所需内存外，仅需额外的空间存储乘客类，运行中仅涉及乘客类在各个数据结构中的转移，当乘客放弃或到达目的楼层后会被释放。

0.9.2 时间复杂度

程序实际上是在给出的模拟时间中的特定一些时间（在运行中由一个个随机事件生成）执行乘客异动操作，程序在大多数循环时仅在做逻辑判断和时间检查。涉及乘客移动的操作多在O(n)（n为人数）内完成。

0.10 遇到的一些问题

- 电梯模拟时乘客的放弃时间不能普遍太短，乘客按下请求后便不能取消，大量乘客的放弃会使电梯出现多次在有请求但没人的楼层停留浪费时间进而使更多的人放弃，一段时间后便几乎无法完成乘客请求。
- 电梯的位置只有整数层，这会带来如电梯在二三层之间时，状态只能设为二层，可能会因此被分配来自二层的请求出现不合实际情况的问题，为解决这些问题需要不少的其他状态判断，本实验暂未解决。

代码测试

输入

```
电梯模拟系统
请输入待运行时间:
500
请输入要模拟的楼层数:
10
```

部分输出

```
下一位乘客将在90时前来
5号乘客已在排队
第5号乘客进入电梯
当前时间为66
第5号乘客离开电梯
当前时间为78
**********************************
第6号乘客将在第4层排队等待乘坐电梯，准备前往第0层楼
下一位乘客将在102时前来
6号乘客已在排队
1号电梯复位!
当前时间为92
一号电梯启动!
**********************************
第7号乘客将在第0层排队等待乘坐电梯，准备前往第3层楼
下一位乘客将在117时前来
7号乘客已在排队
二号电梯启动!
第6号乘客进入电梯
当前时间为104
第7号乘客进入电梯
当前时间为104
第7号乘客离开电梯
当前时间为115
第6号乘客离开电梯
当前时间为116
**********************************
第8号乘客将在第3层排队等待乘坐电梯，准备前往第6层楼
下一位乘客将在135时前来
8号乘客已在排队
第8号乘客进入电梯
当前时间为121
1号电梯复位!
当前时间为124
第8号乘客离开电梯
当前时间为132
**********************************
第9号乘客将在第6层排队等待乘坐电梯，准备前往第3层楼
下一位乘客将在157时前来
```

实验总结

- 通过本次实验，我熟练掌握了包括线性表，栈，队列等数据结构的实现及应用，尽可能根据实际情况选择恰当的数据结构来组织数据。
- 离散事件模拟，关注事件产生，与系统对事件的响应，在本实验中主要体现在如何合理的进行电梯的调度，电梯在运行和停留过程中存在多种状态和多种任务需要处理，电梯的运行也是本实验的核心。
- 电梯是生活中非常常见的设备，在实际生活中乘客事件多样且难以预测，本实验尽量考虑周全但还有不少遗漏和与现实生活可能存在冲突的地方有待优化。

附录

（仅说明.h文件作用，.cpp不额外说明）

1. report PB20020586_叶子昂_lab01.pdf
2. main.cpp 主程序，主要以时间驱动CoCenter类。
3. class.h 声明乘客类，电梯类，目的楼层表类，时间表类及对应的方法。
4. datatype.h 数据结构的声明如乘客栈，等待队列及其方法
5. control.h CoCenter类的声明，及系统运行的控制函数如电梯调度，电梯运行。
6. user.h（无对应的user.cpp被所有其他文件include） 通用宏定义，别名，库文件包含，基本inline函数声明。