进程管理项目——电梯调度 Elevator Dispatch

进程管理项目——电梯调度 Elevator Dispatch

- 项目需求
  - 基本任务
  - 功能描述
- 开发环境
- 系统分析
  - 电梯内部请求
  - 电梯外部请求
- 系统设计
  - 界面设计
  - 类设计
  - 电梯状态设计
  - 消息传递机制设计
- 系统实现
  - 内部请求处理
  - 外部请求处理
  - 调度算法
  - 电梯状态更新
  - 多线程编程
- 操作说明
- 功能演示

学号：1850250

姓名：赵浠明

班级：操作系统（42036901） 周二1-2节、周四1-2节（双）

教师：张慧娟

邮箱：2440720776@qq.com

项目需求

基本任务

某一层楼20层，有五部互联的电梯。基于线程思想，编写一个电梯调度程序。

功能描述

- 设有不同功能的按键
  - 电梯内部设有**数字键**、**开门键**、**关门键**和**报警键**
  - 每层楼的每部电梯外部设有**上行键**和**下行键**
- 有数码显示器指示当前电梯状态

开发环境

- **操作系统平台**：Windows 10
- **开发语言**：Java SE
  - **JDK版本**：jdk1.8.0_152
- **开发软件**：Eclipse IDE *2020-03 (4.15.0)*

系统分析

电梯内部请求

- 乘客按下**开\关门**按钮
  - 若电梯处于运行状态，乘客按下开\关门按钮将不会引起响应
  - 若电梯处于停靠状态，电梯将在固定时间后自动关门，

    a. 若乘客按下开门键，电梯将保持开门状态，并重新进行关门倒计时

    b. 若乘客按下关门键，电梯将立即关门
- 乘客按下**警报**按钮

  电梯将转变为**故障**状态，所有已经被调度给此电梯的外部请求将被重新调度
- 乘客按下**楼层**按钮（设对应楼层为`i`）

  此电梯将添加一个目标楼层为`i`的调度请求，
  - 如果第`i`层在电梯当前的运行方向上，且在电梯当前楼层与目标楼层之间，那么电梯将在该方向运行过程中在第`i`层停靠
  - 如果第`i`层在电梯当前的运行方向上，且比当前电梯目标楼层更远，那么电梯将把目标楼层修改为`i`
  - 如果第`i`层不在电梯当前的运行方向上，那么电梯将先前往当前的目标楼层，再重新设置目标楼层，改变运行方向，前往第`i`层
  - 如果第`i`层恰好为电梯的当前楼层，考虑到电梯实际运行时，需要一段减速时间才能停靠，因此电梯不应立即响应此请求，而是将这种情况看作**情况3**

电梯外部请求

- 乘客在某一楼层按下**上行键**或**下行键**（设楼层为`i`）
    1. 系统将创建一个目标楼层为`i`的上行\下行请求
    2. 根据调度算法，计算每部电梯响应此调度请求的权值，并选出接受此请求的电梯

       调度权值： *（调度算法具体分析在**系统实现**部分）*

        1. 故障电梯

           处于**故障**状态的电梯无法调度

        2. 静止电梯

           电梯当前楼层到第`i`层的**层数差**

        3. 顺路

           电梯当前楼层到第`i`层的**层数差** + 到达第`i`层路程中的**停靠时间**

        4. 非顺路

           电梯绕路后到第`i`层的**运行总层数** + 到达第`i`层路程中的**停靠时间**
    3. 将此调度请求添加到最优电梯的调度请求队列中

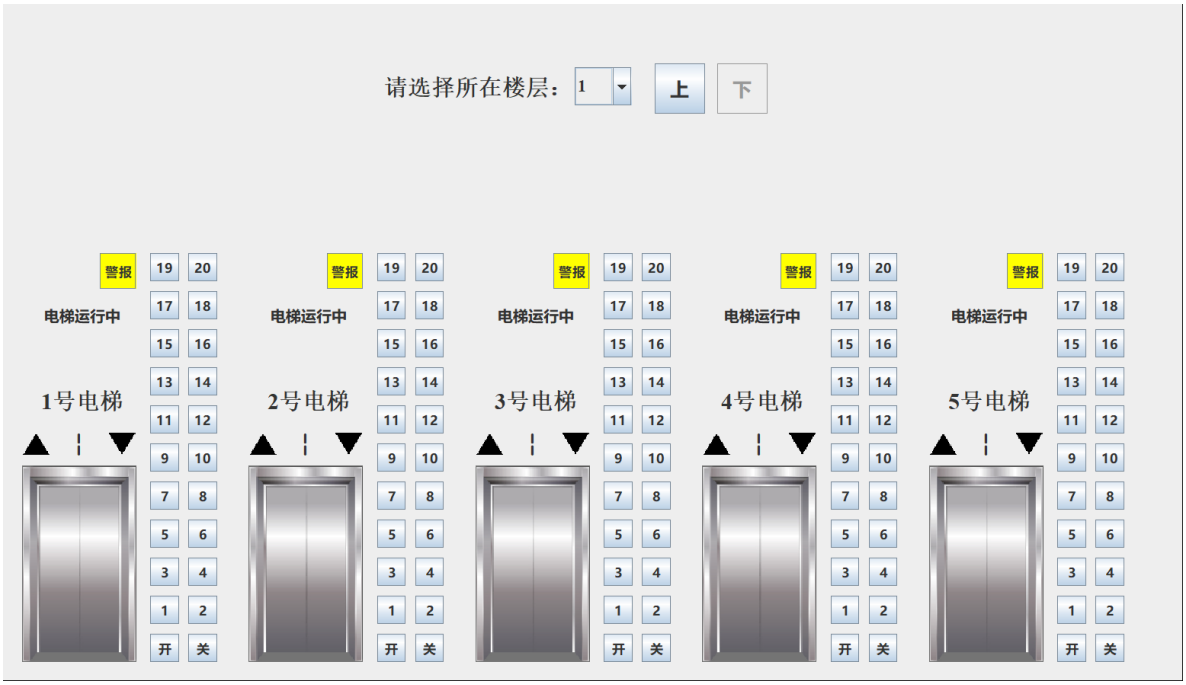
系统设计

界面设计

1. 电梯内部

- 按钮
  - 楼层键1-20 *floorJb[20]*
  - 开门键、关门键 *openJb closeJb*
  - 警报键 *alarmJb*
- 标签
  - 电梯运行信息标签 *infoJl*
  - 电梯编号标签 *IDJl*
  - 楼层显示标签 *floorJl*
  - 运行方向显示标签 *upJl downJl*
  - 电梯门图片 *doorJl*

2. 整体设计

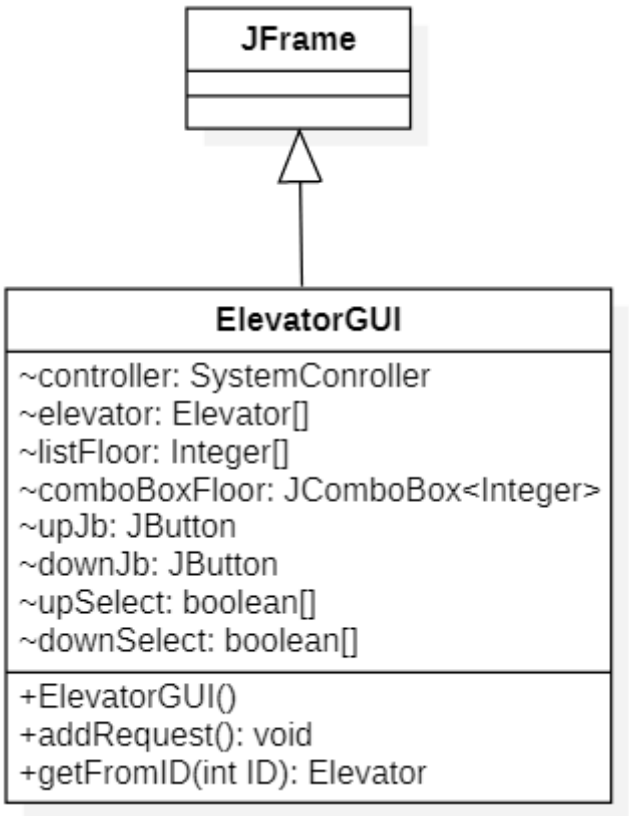


- 电梯组 *elevator[5]*
- 楼层选项框 *comboBoxFloor*
- 上下行按键 *upJb downJb*

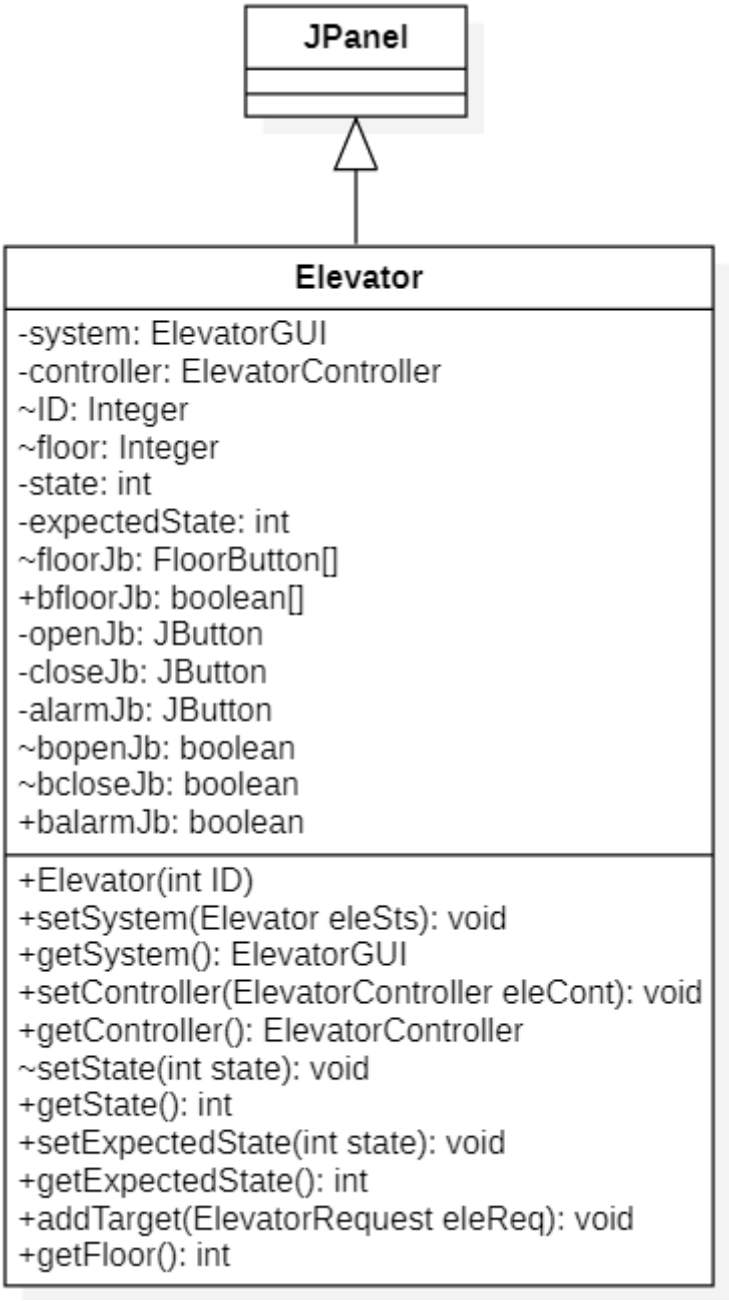
  由于五部电梯是相互连结的，用户通过在楼层选项框中选择楼层，并按下上下行按键，可以模拟乘客在某一楼层外按下了上下行按键，即模拟电梯外部请求。

类设计

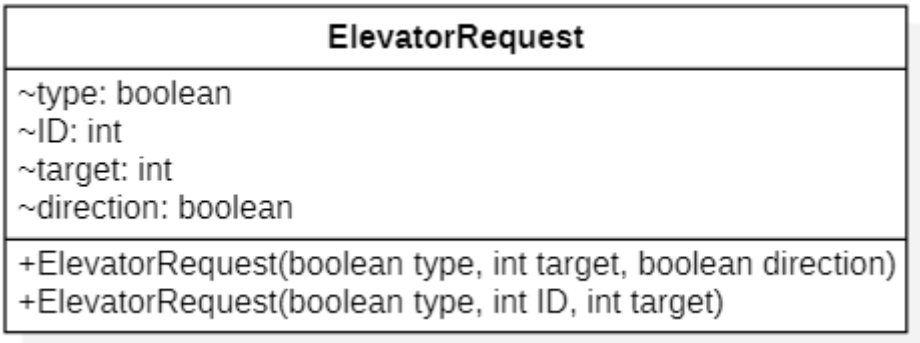
1. ElevatorGUI类：电梯调度系统与用户之间的交互



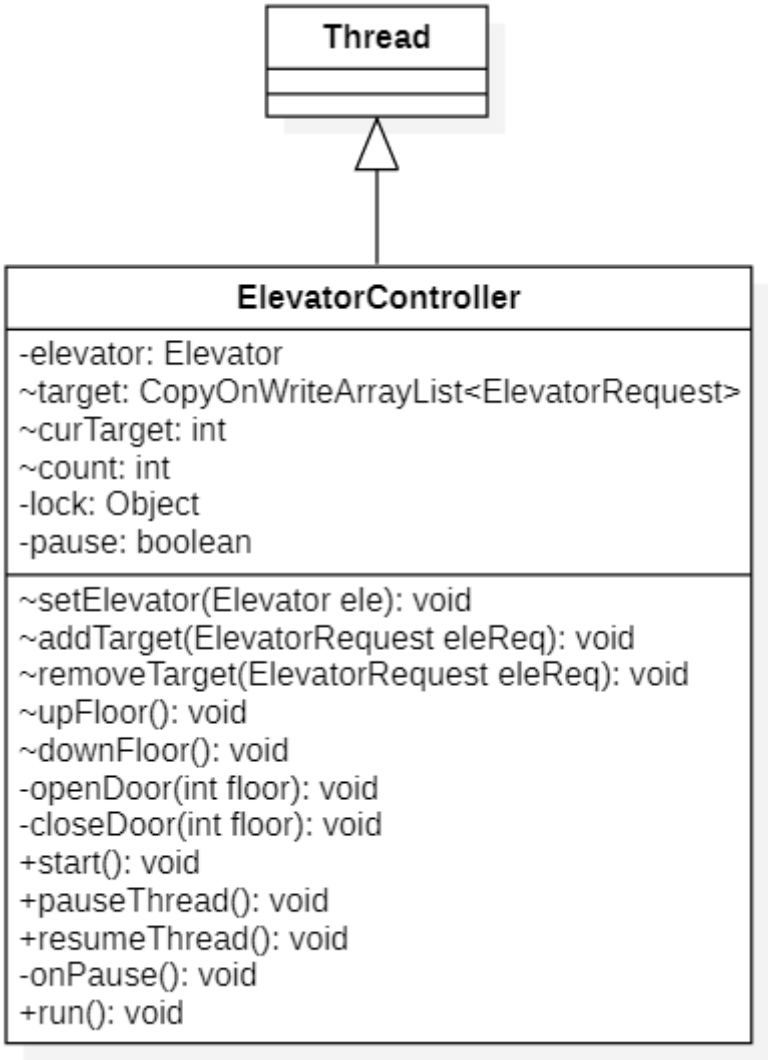
2. Elevator类：封装电梯相关的属性及交互响应逻辑



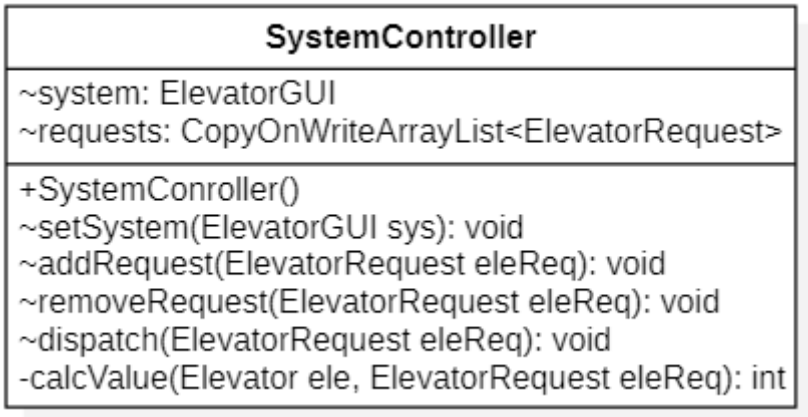
3. ElevatorRequest类：封装电梯的调度请求（内部调度请求、外部调度请求）



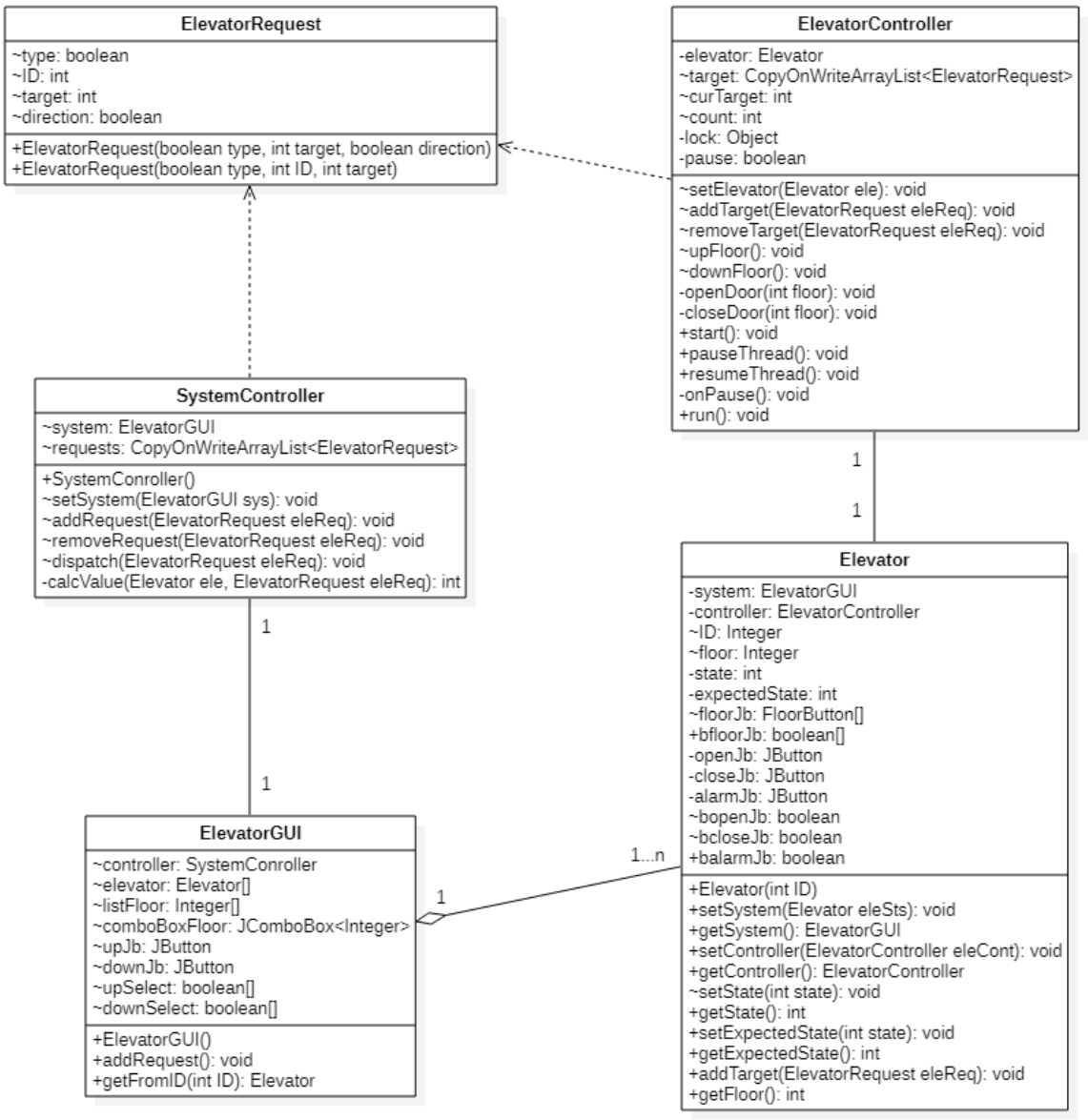
4. ElevatorController类：用于对单个电梯的调度，将电梯属性与调度逻辑控制分离，使架构更清晰



5. SystemController类：用于系统整体的调度，管理及调度系统内所有调度请求



6. 类关系



电梯状态设计

- 电梯当前状态 `state`
  - 静止状态 `STATE_STILL = 0`
  - 上升状态 `STATE_UP = 1`
  - 下降状态 `STATE_DOWN = 2`
  - 故障状态 `STATE_ERROR = 3`
- 电梯到达目标楼层后的期望状态 `expectedState`
  - 静止状态 `STATE_STILL = 0`
  - 上升状态 `STATE_UP = 1`
  - 下降状态 `STATE_DOWN = 2`

设计解释：

目标楼层 *curTarget*：一台电梯的调度请求队列中有多个调度请求，在电梯运行方向上最远的需停靠楼层即为目标楼层。在电梯到达目标楼层后，再根据请求队列中剩余的调度请求确定新的目标楼层。

1. `state`表示电梯的当前状态，根据`state`判断电梯接下来应该向上、向下还是静止
2. `expectedState`表示电梯到达目标楼层后的状态，与目标楼层对应的调度请求有关
    1. 如果目标楼层对应的调度请求为电梯内部请求，则`expectedState`为`STATE_STILL`
    2. 如果目标楼层对应的调度请求为向上的外部请求，则`expectedState`为`STATE_UP`
    3. 如果目标楼层对应的调度请求为向下的外部请求，则`expectedState`为`STATE_DOWN`
3. 在系统运行的过程中，需**动态修改**电梯的`state`、`expectedState`及目标楼层`curTarget`
4. 综合考虑`state`和`expectedState`才能在调度算法中判断电梯是否**顺路**

    eg1. 如果当前电梯在第1层，响应了14层的向下外部请求，此时18层有人按了向上键，这时电梯的`expectedState`为向下，因此是不顺路的

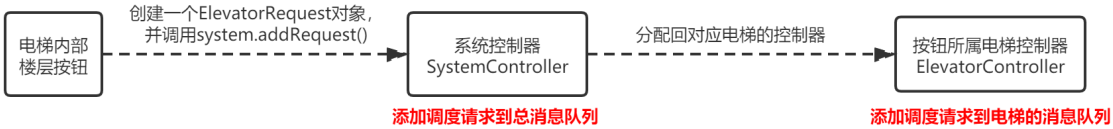
    eg2. 如果当前电梯在第1层，电梯内有人按了18层，而电梯在向上过程中，如果14层有人按了向下键，此调度请求对应的`expectdState`为向下，与当前的`state`不同，因此是不顺路的

消息传递机制设计

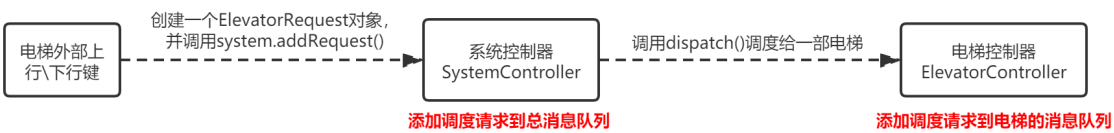
本系统的**调度**体现在将**电梯调度请求**根据调度算法分配给其中一部电梯，电梯作出响应，前往目标楼层。因此系统设计把内部请求和外部请求一同封装为`ElevatorRequest`类，通过创建、调度、移除各个`ElevatorRequest`对象实现电梯的调度。

同时在设计过程中，也设计了相应的**请求消息传递机制**，以此更系统地管理这些调度请求，使系统结构更清晰，增强系统的**健壮性**及**功能可拓展性**。

添加内部请求：



添加外部请求：



移除请求：

系统实现

内部请求处理

1. 楼层键

- 乘客按下电梯内部的楼层键，触发该按钮的监听器`ActionListener`
  - 获取该按钮所属的电梯`ele`，并向系统发送调度请求
  - 置`bfloor[i] = true`，即设置按钮为**按下状态**
- 设置按钮为**不可用状态**
- 调用系统控制器的`addRequest()`方法，将调度请求添加到总调度队列`SystemController.requests`中
- 调用`dispatch()`方法，内部请求不需要计算，直接调度给对应电梯即可

```
void dispatch(ElevatorRequest eleReq) throws Exception {
        /* 内部请求不用算法，直接调度 */
        if (eleReq.type == ElevatorRequest.REQUEST_INSIDE) {
            system.getFromID(eleReq.ID).addTarget(eleReq);
            return;
        }
        /* 外部请求部分省略 */
}
```

- 调用电梯控制器的`addTarget()`方法，将调度请求添加到电梯请求队列`target`中
  - 若满足电梯**顺路**条件，则判断是否需要更新电梯目标楼层`curTarget`，如果需要，由于此调度请求为内部请求，将`expectedState`修改为`STATE_STILL`
  - `target`在之前可能为空，需调用`resumeThread()`恢复电梯线程

2. 警报键

- 乘客按下电梯内部的警报键，触发该按钮的监听器`ActionListener`
  - 将电梯所有按键设置为**不可用状态**
- 调用`setState()`修改状态
  - 将电梯状态设置为`STATE_ERROR`
  - 将电梯的界面显示设置为"电梯故障"

- **遍历电梯的请求队列`controller.target`，将其中的外部请求进行重调度**

```
void setState(int state) {
    /* 其他部分省略 */
    if (state == Elevator.STATE_ERROR) {
        for (ElevatorRequest e : controller.target) {
            if (e.type == ElevatorRequest.REQUEST_INSIDE) {
                controller.removeTarget(e);
            } else {
                try {
                    system.controller.dispatch(e);
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
                controller.removeTarget(e);
            }
        }
    }
}
```

3. 开门键

- 当电梯处于停靠状态时，乘客按下开门键
- 将电梯关门倒计时`count`设置为5，重新开始倒计时，实现电梯保持开门的状态

4. 关门键

- 当电梯处于停靠状态时，乘客按下关门键
- 将电梯关门倒计时`count`设置为0，使电梯立即关门

外部请求处理

- 用户在楼层选项框选择所在楼层，并按下上行键或下行键，触发该按钮的监听器`ActionListener`
    - 更新`upSelect`或`downSelect`数组，设置该楼层的上行\下行键为**按下状态**
    - 设置该按钮为**不可用状态**
- 调用系统控制器的`addRequest()`方法，将调度请求添加到总调度队列`SystemController.requests`中
- 调用`dispatch()`方法，利用`calcValue()`计算各电梯接受此调度请求的权值，并将此调度请求调度给最优电梯

```
void dispatch(ElevatorRequest eleReq) throws Exception {
    /* 内部请求部分省略 */
    Elevator e = system.elevator[1];
    int value = Integer.MAX_VALUE;
    int cur;

    /* 外部请求根据caclValue()权衡各电梯的调度 */
```

```
    for (Elevator e1 : system.elevator) {
        cur = calcValue(e1, eleReq);
        System.out.println(e1.ID + " " + cur);
        if (cur < value) {
            value = cur;
            e = e1;
        }
    }
    eleReq.ID = e.ID;
    e.addTarget(eleReq);
}
```

- 调用电梯控制器的`addTarget()`方法，将调度请求添加到电梯请求队列`target`中
    - 若满足电梯**顺路**条件，则判断是否需要更新电梯目标楼层`curTarget`，如果需要，由于此调度请求为外部请求，将`expectedState`修改为`STATE_UP`或`STATE_DOWN`
    - `target`在之前可能为空，需调用`resumeThread()`恢复电梯线程

调度算法

在进行**外部请求处理**时，需要使用权值函数来比较各电梯相应某一调度请求的优劣，这一部分将解释本系统调度算法中的权值函数`calcValue()`。

```
private int calcValue(Elevator ele, ElevatorRequest eleReq);
```

权值函数的返回值**越小**，表示此电梯响应此调度请求**越合适**。

1. 排除故障电梯

```
if (ele.getState() == Elevator.STATE_ERROR)
    return Integer.MAX_VALUE;
```

2. 静止状态电梯

对于静止状态的电梯，直接计算当前楼层到目标楼层的距离

```
if (ele.getState() == Elevator.STATE_STILL)
    return Math.abs(ele.getFloor() - eleReq.target);
```

3. 顺路电梯

- 判断是否**顺路**
    - 电梯向上运行，且电梯`expectedState`为`STATE_STILL`时，电梯可响应比当前楼层高且向上的外部指令

*eg1. 电梯位于1层，内部有人按了18层，电梯可响应14层**向上**的请求，但不能响应14层**向下**的请求*

*eg2. 电梯位于1层，响应了14层向上的请求，在前往14层过程中，不能响应其他楼层中**向下**的请求*

- 电梯向下运行，且电梯`expectedState`为`STATE_STILL`时，电梯可响应比当前楼层低且向下的外部指令

*eg3. 电梯位于20层，内部有人按了5层，电梯可响应10层**向下**的指令，但不能响应10层**向上**的指令*

*eg4. 电梯位于20层，响应了5层向上的请求，在前往5层的过程中，不能响应其他楼层**向上**的请求*

- 遍历当前楼层至该请求对应楼层之间，有多少层需要停靠，每停靠一层权值`ret += 6`（停靠时开门1s，关门5s）
- `ret`加上路程的层数即为总权值

```
int ret = 0;
    /* 电梯可以顺路接第floor层的乘客，需额外加上途中停靠楼层的时间开销 */
    if (eleReq.target >= ele.getFloor() && ele.getState() ==
Elevator.STATE_UP && ele.getExpectedState() == Elevator.STATE_STILL &&
(eleReq.direction || eleReq.target == 20)) {
        for (ElevatorRequest e : ele.getController().target) {
            if (e.target > ele.getFloor() && e.target < eleReq.target)
                ret += 6;
        }
        return eleReq.target - ele.getFloor() + ret;
    }
    if (eleReq.target <= ele.getFloor() && ele.getState() ==
Elevator.STATE_DOWN && ele.getExpectedState() == Elevator.STATE_STILL &&
(!eleReq.direction || eleReq.target == 1)) {
        for (ElevatorRequest e : ele.getController().target) {
            if (e.target < ele.getFloor() && e.target > eleReq.target)
                ret += 6;
        }
        return ele.getFloor() - eleReq.target + ret;
    }
```

4. 不顺路电梯

只考虑实际的时间耗费，不顺路电梯响应某调度请求的时间耗费可能在特殊情况下会比顺路电梯小，但由于系统内有5部电梯，有足够电梯可以调度，因此总是优先考虑顺路电梯。

```
return Integer.MAX_VALUE - 1;
```

电梯状态更新

1. 添加调度请求

- 电梯处于上升状态，新请求对应楼层比当前目标楼层高，且电梯的`expectedState`为`STATE_STILL`
- 电梯处于下降状态，新请求对应楼层比当前目标楼层低，且电梯的`expectedState`为`STATE_STILL`
    - 若新请求为内部请求，则设置`expectedState`为`STATE_STILL`
    - 若新请求为外部请求，则设置`expectedState`为`STATE_UP`或`STATE_DOWN`

```
void addTarget(ElevatorRequest eleReq) {
    /* 当调度请求队列为空时，新加入的请求即为目标楼层 */

    target.add(eleReq);
    /*
     * 优先执行电梯当前运行方向上的调度请求
     */
    if (eleReq.target >= curTarget && elevator.getState() ==
Elevator.STATE_UP && elevator.getExpectedState() == Elevator.STATE_STILL)
{
        curTarget = eleReq.target;
        /* 设置expectedState */
        if (eleReq.type == ElevatorRequest.REQUEST_OUTSIDE)
            elevator.setExpectedState(eleReq.direction == true ?
Elevator.STATE_UP : Elevator.STATE_DOWN);
        else
            elevator.setExpectedState(Elevator.STATE_STILL);
    }

    if (eleReq.target <= curTarget && elevator.getState() ==
Elevator.STATE_DOWN && elevator.getExpectedState() ==
Elevator.STATE_STILL) {
        curTarget = eleReq.target;
        /* 设置expectedState */
        if (eleReq.type == ElevatorRequest.REQUEST_OUTSIDE)
            elevator.setExpectedState(eleReq.direction == true ?
Elevator.STATE_UP : Elevator.STATE_DOWN);
        else
            elevator.setExpectedState(Elevator.STATE_STILL);
    }
}
```

2. 移除调度请求

- 到达楼层停靠，移除调度请求后，判断楼层是否为当前目标楼层
    - 如果是，则将`expectedState`赋值给`state`，并置`expectedState`为`STATE_STILL`

```
void removeTarget(ElevatorRequest eleReq) {
    /* 更新系统控制器中的调度请求队列 */
    elevator.getSystem().controller.removeRequest(eleReq);
    if (elevator.getState() == Elevator.STATE_ERROR)
        return;
    if (curTarget == eleReq.target) {
        elevator.setState(elevator.expectedState);
        elevator.setExpectedState(Elevator.STATE_STILL);
        return;
    }
}
```

多线程编程

本系统利用了Java的多线程编程，ElevatorController继承自java.lang.Thread类

1. 开始线程

```
public synchronized void start() {
    super.start();
    target = new CopyOnWriteArrayList<ElevatorRequest>();

    if (elevator == null) {
        try {
            elevator = new Elevator(0);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. 暂停及恢复线程

电梯无调度请求时，为防止电梯线程忙等待，通过调用`pauseThread()`暂停线程

```
/* 利用两个变量实现线程的暂停及恢复，防止电梯线程忙等待 */
private final Object lock = new Object();
private boolean pause = false;

/**
 * 暂停线程
 */
public void pauseThread() {
        pause = true;
}
```

通过调用`resumeThread()`恢复线程

```
public void resumeThread() {
        pause = false;
        synchronized (lock) {
                lock.notify();
        }
}
```

`run()`函数中，当线程处于暂停状态，调用`onPause()`进行等待，实现线程的暂停

```
private void onPause() {
        synchronized (lock) {
                try {
                        lock.wait();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

3. 运行线程

```
public void run() {
        super.run();
        while (true) {
                /* 防止暂停时忙等待 */
                while (pause)
                        onPause();
                try {
                    while (elevator.floor != curTarget) {
                        if (curTarget > elevator.floor)
                            upFloor();
                        else
                            downFloor();
                        /* 检测电梯是否突然故障 */
                        if (elevator.getState() == Elevator.STATE_ERROR)
                            break;
                    }
                } catch (InterruptedException e) {
                            e.printStackTrace();
                }

                /* 无调度请求时，暂停进程 */
                if (target.isEmpty()) {
                        pauseThread();
                }
        }
}
```

操作说明

1. 运行程序

本程序需要用户电脑配置有**jre 1.8.0**环境才能运行

- 在命令行cmd中进入.jdk文件所在目录（如果放在桌面上，输入cd desktop即可）
- 输入命令 `java -jar elevator.jar`
- 成功进入程序

2. 模拟电梯调度

- 按下电梯内楼层键，模拟电梯内有乘客前往对应楼层
- 按下电梯内警报键，模拟电梯故障
- 按下电梯内开关门键，模拟电梯停靠时乘客想要开门或是关门
- 在楼层选项框选择楼层，并按下上行键或下行键，模拟某楼层电梯外有人想要搭乘电梯向上或向下

功能演示

注：1. 由于pdf中无法播放.gif文件，助教或老师可以在附件中查看

*2. 助教或老师也可以运行.jar文件直接运行程序，但需电脑上已配置**jdk/jre 1.8.0**环境*

1. 警报功能

（见resources/display/警报功能.gif）

2. 电梯运行

（见resources/display/电梯运行.gif）

3. 开关门

（见resources/display/开关门gif）

4. 外部请求调度

（见resources/display/外部请求调度.gif）