OO第二单元总结

一、概述

OO第二单元、三次作业眼看着就结束了，~~还想再来一遍~~。

本单元的重心在于对于多线程的编写和应用，以及对于多线程中的三种语法：`wait()`，`notifyAll()`，`sychronized(lock)`的使用；同时考察了一定的设计策略（不限于观察者模式和单例模式），本单元均采用生产者消费者模式。

通过本单元的学习，逐步了解了线程同步问题以及针对同步灵活使用锁并避免产生死锁的尝试和思考。

本单元的三次作业完成度较为糟糕，但是相对第一单元已经有了一定的进步，希望能在第三单元中再接再厉。 **本次总结的内容分布如下：**

- 针对每一次作业中锁的分析和使用
- 针对电梯运行采用的调度算法
- 基于度量分析程序并分析其可扩展性
- 程序bug分析
- 互测分析
- 心得体会

二、同步块设置和锁选择

第一次作业

第一次作业是通过**生产者-消费者模式**进行设计的，而且并没有设计`scheduler`类，只包括生产者`InputThread`和消费者`Elevator`，以及用于中转的楼层类`Floor`（非线程）。

- 针对于中转`Floor`，必须锁住每一个方法

```
public synchronized void setIsEnd(boolean isEnd) {
    Floor.isEnd = isEnd;
    this.notifyAll();
}

public synchronized boolean getIsEnd() {}

public synchronized Map<Integer, Vector<PersonRequest>>
getFloors() {}

```

```
public synchronized void adds(PersonRequest request) {
    add(request);
    this.notifyAll();
}

public synchronized void removes(PersonRequest request) {}

public synchronized boolean isEmpty() {}
```

其中在添加新的请求以及输入结束时需要使用`notifyAll()`。

- 针对于`Elevator`
  - `run()`中通过锁住`Floor`和`wait()`进行读取运行。
  - 不需要其他加锁

```
while (true) {
    synchronized (floor) {
        while (floor.isEmpty()) {
            if (floor.getIsEnd()) {
                return;
            }
            floor.wait();
            if (floor.getIsEnd()) {
                return;
            }
        }
    }
}
```

- 针对于`InputThread`
- 不需要加任何锁。
- 分析

  由于实际上电梯和输入线程是分离的，因此任何时候都只需要对Floor静态类进行读取操作，**在实际运行时将Floor锁住从而不同时被两个线程同时调用，即可实现题目要求。同时也能够满足两个线程同步进行，不产生冲突。**

第二次-第三次作业

由于第二次作业和第三次作业中增加了调度器`Scheduler`而删除了中转`Floor`类，因此同步和锁方法较第一次作业有区别，但第二次、第三次作业类似。第二三次作业均没有用到设计模式，直接根据需求进行的设计。

- `InputThread`读取------`Scheduler`响应----------`Elevator`分配

  这三个连续的过程需要在后两者中使用`wait()`，`notifyAll()`进行联系，同样不仅需要在添加请求和结束请求的时候加锁，**同时在访问自身的队列时必须加锁。**

  - `Scheduler`

```
    public synchronized void setEnd(boolean end) {
        isEnd = end;
        this.notifyAll();
    }

    public synchronized void setElevatorQueue(Elevator e)
{
        synchronized (elevatorQueue) {
            elevatorQueue.add(e);
        }
        //队列锁住
        e.start();
        //this.notifyAll();这里并不需要notify
    }//方法锁住

    public synchronized void setRequestQueue(Person p) {
        synchronized (requestQueue) {
            requestQueue.add(p);
        }
        this.notifyAll();
    }
    public void run() {
        while (true) {
            synchronized (this) {
                if (...) {
                    wait();
                }
            }
            allocElevator();//该方法中也必须时时刻刻锁住队列
        }
        //do_something
    }
```

  - `Elevator`

```
public synchronized void setRequest(Person p) {
```

```
        synchronized (outside) {
            //System.out.println(p.getId() + "-ADD");
            outside.add(p);
        }
        this.state = "work";
        this.notifyAll();
    }

    public synchronized void setEnd(boolean end) {
        this.isEnd = end;
        this.notifyAll();
    }

    public void run() {
        while (true) {
            synchronized (this) {
                while (outside.isEmpty() &&
inside.isEmpty() && !isEnd) {
                    wait();
                }
                if (END) {return;}
            }
            //do_something
        }
    }
```

完成以上基本架构，这样整个同步和锁的方法就实现了，之后在**类之间需要调用**或者**需要访问类中的队列时，加锁。**

- 分析

  后面两次作业专门提到：**针对类自身的队列在进行修改、访问、调用时必须加锁。**关键原因**在于自身的队列可能随时有新的请求加入，也就是**`setRequest`**等方法，从而出现**`ConcurrentModificationException`

三、三次作业的调度策略

第一次作业

第一次作业没有使用调度器，最多设置了可捎带策略，因此简单分析。

```
if (visualPeople < 6 && !floor.isEmpty()) {
    //对每个楼层中的人进行检测
   for (int i = curFloor; i <= 20; i++) {
       for (PersonRequest p : floor.getFloors().get(i)) {
```

```
            //人数大于6时停止
            if (visualPeople >= 6) {break;}
            //关键在于修改destination，从而达到可捎带目的
            //因为对于每一层都会根据请求的方向来分析人员是否可以进入
            if (Math.abs(p.getFromFloor() - curFloor) >= max)
{
                visualPeople++;
                des = p.getFromFloor();
                max = Math.abs(p.getToFloor() - curFloor);
             }
        }
        if (visualPeople >= 6) {break;}
    }
}
```

对于目标楼层进行处理，从而使其能够满足可捎带策略。

第二次作业

第二次作业使用的是一个相对简单的调度策略。分析如下：

- 先判断有没有处于空闲等待状态的电梯。如果有，则优先加入空闲电梯
- 如果没有空闲等待电梯，则根据可捎带策略为其分配电梯
- 如果可捎带策略不成立，直接为其分配离其最近的电梯（无论方向和人数）

显然这样的调度是有很大缺陷的：

- 直接分配最近的电梯存在很大缺陷，有可能会出现较长的等待。

伪代码如下：

```
public void setElevator(Vector<PersonRequest> waitQueue) {
        for (int k = 0; k < waitQueue.size(); k++) {
            PersonRequest p = waitQueue.get(k);
            //如果存在空闲电梯
            if (availableElevatorIndex(p)) {ADD-TO-ELEVATOR};
            //如果存在可捎带电梯
            else if (takenableInFive(p)) {ADD-TO-ELEVATOR};
            //加入离其最近的
            else {ADD-TO-latestElevator};
            remove(p);
        }
    }
```

第三次作业

第三次由于涉及到换乘，因此仔细考虑之后构造了一个满足换乘的调度器。分析如下：

- 每一个电梯自身存在一个`getFloorTime()`方法，该方法返回一个数组，表示电梯到达每一楼层的所需时间，如果该层无法达到直接设置成`30000`。
- 调度器分配电梯时首先获取`getFloorTime()`，根据`A`，`B`，`C`，三种类型的电梯分别得到属于该类电梯到达指定楼层（`Person.getFromFloor`）时间最小值对应的`index`。
- 首先分配空闲电梯，并进行`CheckElevatorValid(Person p, Elevator e)`来分析是否可乘、是否需要换乘。
- 否则分配最近的电梯，并进行`CheckElevatorValid`，如果不满足检查条件，就加入`A`类`Elevator`。

同样这类算法也有缺陷：

- `getFloorTime()`函数写的并不是很好，不一定就能反映出确切能够到达的时间（没有考虑到电梯内人的数量的因素）
- 调度器的函数很多，比较辛苦
- 算法也做不到时间最短。在没有考虑换乘的情况下花费时间可能会少于通过换乘算法需要的时间。（以弱测1为例：前后差5s的时间）

不过个人认为也有其优点：

- 换乘可以自动实现，不需要为换乘算法苦恼；并在算法中尽量对各个情况考虑周全了。

部分代码如下：

```
public synchronized void allocElevator() {
    for (Elevator e : elevatorAQueue) {
        atime.add(e.getFloorTime());
    }
    for (Elevator e : elevatorBQueue) {
        btime.add(e.getFloorTime());
    }
    for (Elevator e : elevatorCQueue) {
        ctime.add(e.getFloorTime());
    }
    //获得时间
    String s = evenlyAllocMethod();
    int enum1 = elevatorAQueue.size();
    int enum2 = elevatorBQueue.size();
    int enum3 = elevatorCQueue.size();
    elevatorSet(s, enum1, enum2, requestQueue);
    elevatorSet(s, enum1, enum2, tmpQueue);
    atime.clear();
```

```
19      btime.clear();
20      ctime.clear();
21  }
```

四、基于度量和可扩展性分析

- 第一次作业
  - UML类图

  - 类的属性表格和复杂度分析

  类复杂度分析：

class	OCavg	OCmax	WMC
Elevator	3.85	16.0	50.0
Floor	1.43	3.0	10.0
Input	2.0	3.0	4.0
Main	1.0	1.0	1.0
Total			65.0
Average	2.83	5.75	16.25

  方法复杂度分析：

method	Cogc	ev(G)	iv(G)	v(G)
Elevator.close()	0.0	1.0	1.0	1.0
Elevator.Elevator(Floor)	0.0	1.0	1.0	1.0
Elevator.in()	10.0	4.0	6.0	8.0
Elevator.isIn()	5.0	3.0	4.0	6.0
Elevator.isOut()	3.0	1.0	2.0	3.0
Elevator.move()	1.0	1.0	2.0	2.0
Elevator.open()	0.0	1.0	1.0	1.0
Elevator.out()	5.0	1.0	5.0	5.0
Elevator.run()	32.0	7.0	12.0	17.0
Elevator.setDestinationA()	3.0	1.0	3.0	3.0
Elevator.setDestinationB()	19.0	6.0	6.0	8.0
Elevator.setDestinationC()	3.0	1.0	3.0	3.0
Elevator.setType(String)	0.0	1.0	1.0	1.0
Floor.adds(PersonRequest)	0.0	1.0	1.0	1.0
Floor.Floor()	1.0	1.0	2.0	2.0
Floor.getFloors()	0.0	1.0	1.0	1.0
Floor.getIsEnd()	0.0	1.0	1.0	1.0
Floor.isEmpty()	3.0	3.0	2.0	3.0
Floor.removes(PersonRequest)	0.0	1.0	1.0	1.0
Floor.setIsEnd(boolean)	0.0	1.0	1.0	1.0
Input.Input(Floor)	0.0	1.0	1.0	1.0
Input.run()	5.0	3.0	4.0	4.0
Main.main(String[])	0.0	1.0	1.0	1.0
Total	90.0	43.0	62.0	75.0
Average	3.91	1.87	2.70	3.26

- 第二次作业
  - UML类图

  - 类的属性表格和复杂度分析

    类复杂度分析：

class	OCavg	OCmax	WMC
Elevator	2.72	14.0	49.0
InputThread	5.0	5.0	5.0
Main	1.0	1.0	1.0
Scheduler	2.67	6.0	32.0
Total			87.0
Average	2.72	6.5	21.75

    方法复杂度分析：

method	Cogc	ev(G)	iv(G)	v(G)
Elevator.changeState(String)	0.0	1.0	1.0	1.0
Elevator.close()	0.0	1.0	1.0	1.0
Elevator.Elevator(int,Scheduler)	0.0	1.0	1.0	1.0
Elevator.getCurFloor()	0.0	1.0	1.0	1.0
Elevator.getDirection()	0.0	1.0	1.0	1.0
Elevator.getPeople()	0.0	1.0	1.0	1.0
Elevator.getSumTime(PersonRequest)	0.0	1.0	1.0	1.0
Elevator.getTheState()	0.0	1.0	1.0	1.0
Elevator.in()	5.0	1.0	6.0	6.0
Elevator.isIn()	11.0	4.0	6.0	7.0
Elevator.isOut()	3.0	3.0	2.0	3.0
Elevator.move()	1.0	1.0	2.0	2.0
Elevator.open()	0.0	1.0	1.0	1.0
Elevator.out()	4.0	1.0	4.0	4.0
Elevator.run()	31.0	3.0	15.0	16.0
Elevator.setEnd(boolean)	0.0	1.0	1.0	1.0
Elevator.setRequest(PersonRequest)	0.0	1.0	1.0	1.0
Elevator.setTargetFloor()	26.0	3.0	13.0	14.0
InputThread.run()	9.0	3.0	6.0	6.0
Main.main(String[])	0.0	1.0	1.0	1.0
Scheduler.addElevator(int)	0.0	1.0	1.0	1.0
Scheduler.addRequest(PersonRequest)	0.0	1.0	1.0	1.0
Scheduler.availableElevatorIndex(PersonRequest)	1.0	2.0	1.0	2.0
Scheduler.availableInFive(PersonRequest)	6.0	1.0	3.0	5.0
Scheduler.getType()	0.0	1.0	1.0	1.0
Scheduler.latestInFive(PersonRequest)	3.0	1.0	3.0	3.0
Scheduler.note()	0.0	1.0	1.0	1.0
Scheduler.run()	14.0	3.0	9.0	9.0
Scheduler.Scheduler(String)	1.0	1.0	2.0	2.0

method	Cogc	ev(G)	iv(G)	v(G)
Scheduler.setElevator(Vector)	5.0	4.0	4.0	4.0
Scheduler.setEnd(boolean)	0.0	1.0	1.0	1.0
Scheduler.takenableInFive(PersonRequest)	9.0	3.0	6.0	8.0
Total	129.0	51.0	99.0	108.0
Average	4.03	1.59	3.09	3.38

- 第三次作业
  - UML类图

  - 类的属性表格和复杂度分析

类的复杂度分析：

class	Ocavg	OCmax	WMC
Elevator	3.95	22.0	79.0
InputThread	5.0	5.0	5.0
Main	1.0	1.0	1.0
Person	1.0	1.0	9.0
Scheduler	6.0	23.0	84.0
Total			178.0
Average	3.97	10.4	35.6

方法复杂度分析：

	Cogc	ev(G)	iv(G)	v(G)
Elevator.close()	0.0	1.0	1.0	1.0
Elevator.Elevator(int,String,Scheduler)	1.0	1.0	1.0	4.0
Elevator.getCurFloor()	0.0	1.0	1.0	1.0
Elevator.getFloorTime()	42.0	1.0	12.0	20.0
Elevator.getMaxPeople()	0.0	1.0	1.0	1.0
Elevator.getOutside()	0.0	1.0	1.0	1.0
Elevator.getPeople()	0.0	1.0	1.0	1.0
Elevator.getTheState()	0.0	1.0	1.0	1.0
Elevator.getType()	0.0	1.0	1.0	1.0
Elevator.in()	5.0	1.0	6.0	6.0
Elevator.isEmpty()	1.0	1.0	2.0	2.0
Elevator.isIn()	11.0	4.0	6.0	7.0
Elevator.isOut()	3.0	3.0	2.0	3.0
Elevator.move()	1.0	1.0	2.0	2.0
Elevator.open()	0.0	1.0	1.0	1.0
Elevator.out()	7.0	1.0	5.0	5.0
Elevator.run()	30.0	3.0	14.0	15.0
Elevator.setEnd(boolean)	0.0	1.0	1.0	1.0
Elevator.setRequest(Person)	0.0	1.0	1.0	1.0
Elevator.setTargetFloor()	36.0	3.0	13.0	16.0
InputThread.run()	9.0	3.0	6.0	6.0
Main.main(String[])	0.0	1.0	1.0	1.0
Person.getFromFloor()	0.0	1.0	1.0	1.0
Person.getId()	0.0	1.0	1.0	1.0
Person.getOriginToFloor()	0.0	1.0	1.0	1.0
Person.getStatus()	0.0	1.0	1.0	1.0
Person.getToFloor()	0.0	1.0	1.0	1.0
Person.Person(PersonRequest)	0.0	1.0	1.0	1.0
Person.setFromFloor(int)	0.0	1.0	1.0	1.0
Person.setStatus(String)	0.0	1.0	1.0	1.0
Person.setToFloor(int)	0.0	1.0	1.0	1.0
Scheduler.allocElevator()	3.0	1.0	4.0	4.0
Scheduler.checkElevatorValid(Person,Elevator)	14.0	5.0	10.0	10.0
Scheduler.elevatorSet(String,int,int,Vector)	23.0	1.0	13.0	13.0

	Cogc	ev(G)	iv(G)	v(G)
Scheduler.evenlyAllocMethod()	8.0	3.0	4.0	8.0
Scheduler.getNearestElevator(Person,String)	45.0	1.0	23.0	31.0
Scheduler.getType()	0.0	1.0	1.0	1.0
Scheduler.getWaitElevator(Person)	12.0	1.0	7.0	9.0
Scheduler.personAnalyse(Person,Elevator)	17.0	6.0	12.0	12.0
Scheduler.run()	28.0	3.0	21.0	21.0
Scheduler.Scheduler(String)	0.0	1.0	1.0	1.0
Scheduler.setElevatorQueue(Elevator)	1.0	1.0	1.0	4.0
Scheduler.setEnd(boolean)	0.0	1.0	1.0	1.0
Scheduler.setRequestQueue(Person)	0.0	1.0	1.0	1.0
Scheduler.setTmpQueue(Person)	1.0	1.0	2.0	2.0
Total	298.0	69.0	190.0	224.0
Average	6.62	1.5333333333333334	4.22	4.98

- 可扩展性分析

五、程序bug分析

- 第一次作业的bug

  第一次作业中出现的bug如下：

  - 逻辑错误：电梯方法中`isIn()`判定是否有人能够进入和`In()`人进入方法出现了逻辑错误。在`In()`方法中无论人的行进方法是否为“上”还是为“下”，乘客都一视同仁地能够进入电梯，因此在之后判断目标楼层中出现了逻辑错误。如下：

```
Elevator().In():
for (PersonRequest p : floor.getFloors().get(curFloor))
{
    //没有判断该人的方向和电梯方向
    removable.add(p);
    requests.add(p);
    people++;
    TimableOutput.println("IN-" + p.getPersonId() + "-"
+ curFloor);
}

Elevator.setDestination():
for (PersonRequest p : requests) {
```

```
    //直接采用Math.abs，从而忽略了在In()中产生的人方向和电梯方向相反的情况
    if (Math.abs(p.getToFloor() - curFloor) > max) {
        des = p.getToFloor();
        max = Math.abs(p.getToFloor() - curFloor);
    }
}
return des;

//会在实际运行中产生以下问题：
//假如一个去9一个去12，电梯在11
ARRIVE-10
ARRIVE-11
ARRIVE-10
ARRIVE-11
//无限循环
```

- 情况考虑不足：在写`if`语句时由于没有考虑到所有的运行情况，因此`if`语句不完善

```
Elevator.isIn():
if (peopleDirect == direction || people == 0 || direction == 0) {
    stop = people < 6;
    break;
}
//没有考虑到Night模式
//people == 0 || direction == 0会导致一些方向相反的乘客进入

if (direct == direction || (people == 0 && direction == 0) ||
    type.equals("Night"))
{
    stop = people < 6;
    break;
}
```

- 第二次作业中的bug

  第二次作业最重要的在于设置锁和同步的问题，因为通过强测点判断有很严重的`Exception`。

  - 未初始化：给电梯初始化时未直接设置状态，而在之后调度中访问了这个不存在的状态抛出异常。

```
Elevator():
```

```
Elevator(int id, Scheduler scheduler) {
    this.scheduler = scheduler;
    this.id = id;
    this.setName("Elevator" + id);
    //这句话之前没有加上
    //this.state = "wait";
    /*但是实际加上的位置在run()中：
            while (personVector.isEmpty() &&
requestVector.isEmpty() && !isEnd) {
                            changeState("wait");
                            wait();
                        }
     */
}

Scheduler.availableInFive():

for (int i = 0; i < elevators.size(); i++) {
    //直接访问了
    state = elevators.get(i).getTheState();
    if (state.equals("wait")) {...}
}

```

但是遗憾的是本地测不出来，因为同步的问题在本地测试中极有可能是先在`Elevator.run()`中设定了`status`，而后才有`Schedulor`访问从而没有访问空指针，但是两者的执行是同步的，也有可能后来居上从而产生`Exception`，这大概也是线程的~~魅力所在~~吧。

- 未加锁：由于同步的问题，因此极有可能产生各种问题：比如**多次调用同一个`start`**。

```
Scheduler.run():
//未加锁时：
//synchronized (this) {
for (Elevator elevator : elevators) {
    elevator.start();
}
//        }

Scheduler.addElevator(PersonRequest p):
{
    Elevator elevator = new Elevator(p, this);
    elevators.add(elevator);
```

```
13      elevator.start();
14  }
15  //由于没有加锁，从而使得产生两种问题：同时访问List抛出异常；多次
    start抛出异常
```

通过第二次的严重错误，使得我对线程的锁与同步的理解更加深刻。

- 第三次作业中的bug

  细心问题：实际上前两次作业完成后第三次作业基本写下来顺风顺水，可以尝试多种调度。

  只修改了一行（当时看错了）。

```
1  //修改前：
2  amin = bmin = cmin = 20000;
3  //有几个点出现了问题
4  //修改后：
5  amin = bmin = cmin = 25000;
6  //成功AC
```

六、互测策略

互测主要通过构造一些极端数据来HACK，有以下两种思路：

- 构造循环往复：

  从最开始到最后，从最后到最开始这种方式尝试来卡人。

- 构造长延时

  可以等待一段时间之后进行输入。

七、心得体会

首先谈谈对本单元作业的满意程度：还需努力。本单元作业较于上一单元明显有了改善，但是依旧还有进步的空间，希望自己在后面两个单元争取做到更好。

这一单元主要考察线程思想。通过本单元自己能够以bug 的形式深刻了解到同步和锁的问题，并能够针对这些问题提出解决方案，这是最大的收获。

本单元就到此结束了，下一单元再见！