在电梯月到来之前，我便从很多学长和同学那里听闻了电梯的恶心与折磨。所以我之前每每想到电梯月，总是会想起鲁迅先生所说的“***不在沉默中死亡，就在沉默中爆发***”，总是幻想会有某种日夜debug而不得果的悲壮亦或者一路斩将过关的豪爽。

但是当电梯月真的到来时，我却只是感到平淡，甚至有些无聊。当然绝不是说多线程简单，更不是说我天资聪慧，事实上U2于我来说确实是要比U1要难，我也绝不处于聪慧的人的队列。只是感觉我似乎丧失了当初U1时的斗志，那种追求极致的痴迷，而是多走了很多捷径，草草了事。

0 前言

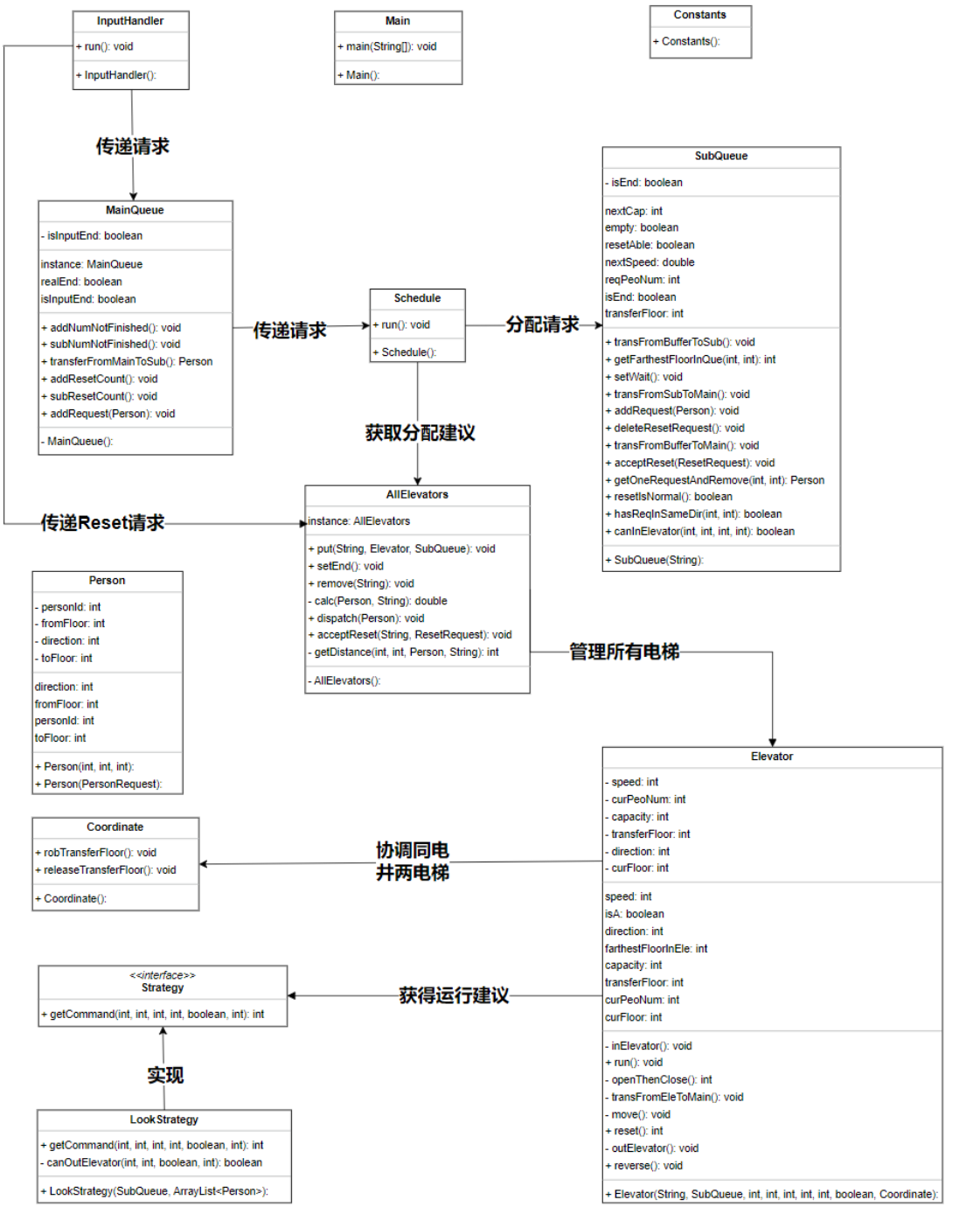
言归正传，第二单元的主题是模拟多线程实时电梯系统。之前在某个接手的任务之中我曾经略微接触过多线程编程，但是那个多线程并行计算不需要任何线程安全方面的考虑，所以只是让我知道了多线程并行计算的优势。在这一单元，我真正熟悉线程的创建、运行等基本操作，掌握线程之间的交互方法，学习到了一些基本的线程之间的协同设计层次架构。

1 整体架构

本单元作业我并没有重构过，hw6与hw7均是在上一次作业的基础上进行的迭代开发，所以在此我仅仅给出迭代的最终版本——hw7代码的UML图与时序图。

1.1 UML图

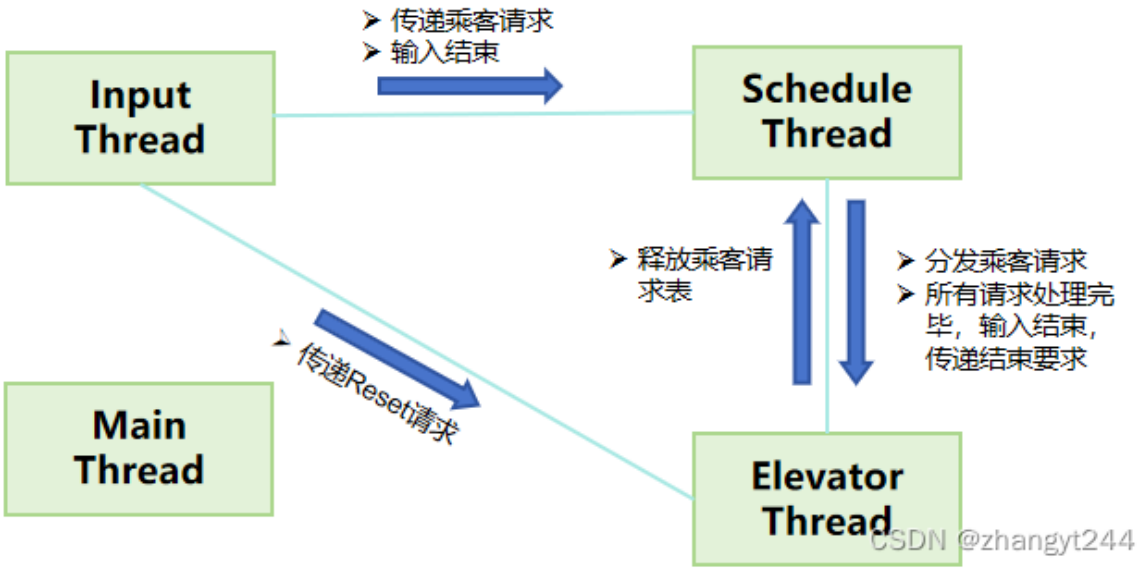
下面是第七次作业的UML类图：



其中第六次作业主要扩展了调度策略的实现方法；第七次作业新增了AllElevator类用于对电梯类进行统一管理，将调度策略的具体实现方法迁移到了AllElevator类中以便于实现线程之间的互斥，并且还新建了Coordinate类以用于协调同一个电井下两轿厢之间的工作。

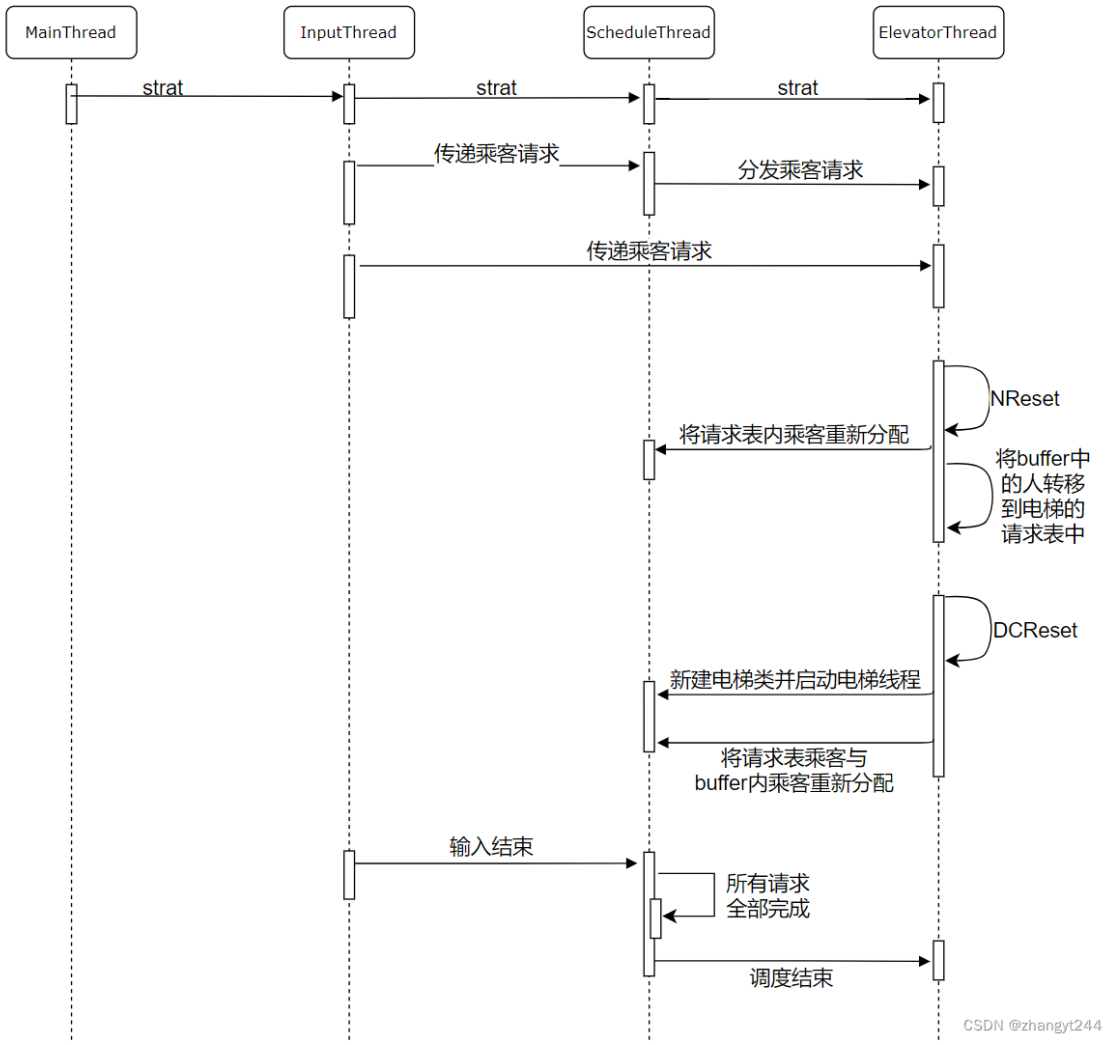
1.2 协作图

下面是MainThread、InputThread、Schedule、Elevator三个线程之间的协作图：



1.3 时序图

下面是作业中出现的四个线程之间的时序图：



2 hw5

2.1 架构设计

本次作业要求我们模拟六部电梯的运行，竟然如此，我想我们当然应该从电梯开始想起，每部电梯仅有`RECEIVE`,`MOVE`,`OPEN`,`CLOSE`,`IN`,`OUT`等可供输出的操作，除此之外，我们还比较容易想到实际上电梯还应当由有`WAIT`与`END`这两个操作，以便防止轮询的产生以及终止电梯线程。、

那么我们接下来需要做的就应当是确定如何实现以及何时执行这些操作了。显然，我们需要建立输入类`InputThread`用于获得输入请求，建立总请求表类`MainQueue`用于存储所有乘客请求，建立分请求表类`SubQueue`用于存储每部电梯的乘客请求。基于这些类中存储的属性，我们可以意识到确定电梯如何接收请求、如何运行应当是基于这几个类中所存储的属性。那么具体该如何实现呢，还需要一个调度器类`Schedule`用于分发乘客请求，一个策略类`Strategy`用于给出运行建议或者说命令。

显然地，对于一个刚刚接触多线程的人而言，解决问题的关键有二：

- 如何确定调度策略和运行策略；
- 如何设置同步块、实现线程安全；

我们将在下文给出分析。

2.2 调度策略和运行策略

对于第一次作业，由于乘客请求本身就含有目标电梯的信息，所以在调度器类中直接分发即可：

```
Person person = mainQueue.transferFromMainToSub(
if (person == null) {
    continue;
}
subQueues.get(person.getElevatorId()).addRequest(person);
```

> 我们当时可能会产生如下疑问：既然直接分发即可，那么为什么还需要设置调度器类呢？直接从输入类中分发给各电梯的请求表不是更好吗？
> 事实上不是这样的，我们纵观往年三届OO课程的U2指导书，可以发现今年的作业中新增了Receive操作，这样的话也就基本堵死了自由竞争这一个低投入高回报的方法的实现可能性，那么可以预想在之后的作业中调度策略的实现将会是重中之重，事实也的确如此

至于运行策略，我选择了往届学长大力推荐的`LOOK`策略，另外还采用了一些小trick，也就是同学们口中所说的量子电梯。在此我不想介绍量子电梯这种奇技淫巧的实现方式，因为我认为这是与实际情况相悖的，也与面向对象的知识没有太多关系，是不应当提倡的，~~希望明年OO助教能够ban掉这种歪门邪道~~。下面我们介绍一下`LOOK`策略的实现逻辑：

```
    public Com getCommand(int curPeoNum, int curFloor, int direction) {
        // 判断是否要开门，判断的依据是是否有人想出电梯或者是否有人想进电梯
        if (canOutElevator(curPeoNum, curFloor) || canInElevator(curPeoNum,
curFloor, direction)) {
            return Com.OPEN;
        }
        // 无人出且无人进 或  无人出且电梯满
        // 电梯内有人
        if (curPeoNum != 0) {
            return Com.MOVE;
        }
        // 电梯内有人还未出电梯
        // 电梯请求队列不为空
        if (requestQueue.getNum() != 0) {
```

```
            // 有人在电梯当前运行方向前方等待电梯
            if (hasReqInSameDir(curFloor, direction)) {
                return Com.MOVE;
            }
            // 电梯当前运行方向前方没有任何请求
            return Com.REVERSE;
        }
        // 电梯请求队列为空
        else {
            if (requestQueue.isEnd()) {
                return Com.END;
            }
            else {
                return Com.WAIT;
            }
        }

    }
```

2.3 同步块与锁

我选择了课上讲到的`synchronized`关键字用来加锁，当然也可以选择读写锁等来实现更加灵活美观地上锁，但是考虑到有且仅有六部电梯并且可以预见将来也不会有太多电梯，所以感觉`synchronized`足够了。

那么该怎么确定同步块呢，我选择OS课件中的Bernstein条件进行判断，这样的话就可以公式化地确定出哪里该加锁了。

3 hw6

本次作业新增了重置请求，并且不再指定电梯，所以需要自己去设计调度器将乘客请求分发给电梯。

3.1 调度策略

身边很多同学采用了有实现可能性的号称最强大的调度策略——影子电梯。可能是自身实力有限，亦或者清明节时花了许多时间在春游上，还有可能正如前言提到的那样——丧失了斗志，不管有什么理由，我没实现影子电梯却是事实，也是一个小小的遗憾。我采用了调参策略作为自己的调度策略，在我的调参策略之中，我考虑了电梯运行到乘客出发地所需要走的距离、电梯容量、电梯移动速度、电梯内人数、请求人数（这里的请求表人数不仅包括请求表内的人数还包括可能存在的`buffer`中的人数）等多个参数作为请求分发的依据，相对来说考虑因素比较全面，最终通过使用评测机进行大量测试并将运行时间开销与实现影子电梯的同学进行比较得出了如下公式：

```
double result = -1 * distance + 2 * capacity + 1 * dirIsSame - 2.5 * curPeoNum
                - 2.5 * reqPeoNum - (double)speed / 100;
```

在设计时，我和朋友讨论到是否要为正在reset的电梯分发乘客请求，最终我们得出了应当为正在reset的电梯分发请求的结论，原因有三：

1. reset时间仅有1.2秒，而电梯在reset初会将电梯内乘客全部out出去，那么此时与resetting的电梯处于同一层的乘客有很大概率选择等待1.2秒去做原电梯是最优的选择，而这些被赶出来的乘客绝不在少数。
2. 如果武断地不向reset电梯分发请求，那么当六个电梯同时reset时该如何办呢。
   3.如果不向reset电梯分发请求，当多数电梯在reset时就会向少数点题分发大量请求，有可能会出

现RTLE的可能，事实上也确实如此。
所以我们采取了向`SubQueue`类中加入一个用来在reset时存请求的`buffer`容器，然后在`reset`结束时统一将`buffer`内请求迁移到请求表里。

```
public synchronized void tackleBuffer() {
    Iterator<Person> iterator = buffer.iterator();
    while (iterator.hasNext()) {
        Person person = iterator.next();
        TimableOutput.println("RECEIVE-" + person.getPersonId() + "-" + id);
        persons.add(person);
        iterator.remove();
    }
    notifyAll();
}
```


3.2 同步块与锁

在上一次作业中，我使用了大量没有必要的`notifyAll`,本着对自己的学习负责的原则，我在这次作业中将`notifyAll`的数量减小到了最低限度。

另外，在这次作业之中，我还给获取建议类方法整体加上了对请求表的锁，以确保获得的判断结果是准确的。

本次作业与线程安全相关的最难的地方应该就是`Elevator`类中`reset()`方法，但是也只是需要认真思考一下各变量由何种进程读写即可，不必多言。

4 hw7

4.1 实现新增需求

第七次作业还是相当有意思的，新增了`DoubleCarResetRequest`请求类型。才开始我想不出来一个优雅的防止两电梯在换乘层相撞的方法，直到一个朋友提醒我可以学一下OS的信号量，看完之后果然获得了灵感，于是新建了一个`Coordinate()`的类，并将其实例化对象作为同一电井内两个轿厢的共享属性。

```
public class Coordinate {
  private boolean isOccupied;

  public Coordinate() {
      this.isOccupied = false;
  }

  // 抢夺换乘层
  public synchronized void robTransferFloor() {
      if (isOccupied) {
          try {
              wait();
          } catch (InterruptedException e) {
              throw new RuntimeException(e);
          }
      }
      isOccupied = true;
  }
```

```
    // 释放换乘层
    public synchronized void releaseTransferFloor() {
        isOccupied = false;
        notifyAll();
    }
}
```

4.2 调度策略

调度策略与上次作业基本相同，不再重述。唯一值得一提的是也会向正在`DCReset`的电梯分发请求，然后在`DCReset`结束后讲`buffer`中的请求全部放回总请求表之中。这样做虽然会损失一定的性能，但是却很好地规避了当多数电梯`DCReset`时调度器向少数电梯分发大量请求的可能，我认为是值得的。

4.3 同步块与锁

本次作业中同步块基本没啥大的变化，只是新增了`Coordinate`类需要进行一些同步控制，但是我尝试使用了`ReentrantLock`来替换了`synchronized`,不过最后感觉可读性甚至不如`synchronized`,所以不又换了回来。可能是因为我的代码中很少存在叠加多个锁的情况，~~比如wxm同学的某处代码中就嵌套了12个`synchronized`doge~~。

5 稳定与易变

5.1 保持不变的内容

这三次作业运行策略几乎从未发生过大的变化，可能是因为`Look`策略足够经典，也足够自然，很难短时间相出比`Look`策略要好的策略。

另外，整体的架构也没有发生较大的变化，并未经历重构。可能是因为在动手写hw5之前查阅了大量的博客并与许多优秀的同学进行了交流，从而确定了一个比较好的架构。

5.2 易于变化的内容

这三次作业程序终止的条件都在不断发生变化，这是因为需求不断增加。比如，第一次作业只需要输入线程结束并且总表为空时便可以结束调度线程；第二次作业新增了`reset`请求，所以需要将调度线程终止条件修改成输入线程结束、总表为空并且`reset`全部结束；而第三次作业又新增了双轿厢电梯，所以没有处于reset的电梯也会中途放出乘客，那么就需要将调度线程终止条件修改成输入线程结束、`reset`全部结束并且所有乘客请求全部处理完毕。

```
// 线程结束条件，以Schedule线程为例
// hw5
    public synchronized boolean isRealEnd() {
        return (this.isInputEnd) & (this.num == 0);
    }
// hw6
    public synchronized boolean isRealEnd() {
        return (this.isInputEnd) & (this.num == 0) & (this.resetCount == 0);
    }`
// hw7
    public synchronized boolean isRealEnd() {
        return (this.isInputEnd) & (this.numNotFinished == 0) & (this.resetCount
== 0);
    }
```

6 bug分析

本单元作业在强测和互测之中均未出现过bug，但是在自己敲代码时遇到过一些bug，大多均为线程安全bug，不过通过采用print大法就很容易的分析出bug的类型，然后定位出产生bug的位置，从而修复bug。

除了print大法，我还要推荐一个非常好的debug方法，即寻找一个关系非常好的朋友，两人相互走查代码，亲测十分有效。

下面给出几个互测时成功的hack样例及分析：

```
// hack0
有一位同学没有意识到对容器的读写并非原子操作，所以会出现线程不安全的问题
// hack1
// input
[1.0]RESET-Elevator-1-3-0.6
[49.9]RESET-Elevator-2-3-0.6
[49.9]RESET-Elevator-3-3-0.6
[49.9]RESET-Elevator-4-3-0.6
[49.9]RESET-Elevator-5-3-0.6
[49.9]RESET-Elevator-6-3-0.6
[49.9]1-FROM-1-TO-11
[49.9]2-FROM-1-TO-11
[49.9]3-FROM-1-TO-11
[49.9]4-FROM-1-TO-11
[49.9]5-FROM-1-TO-11
[49.9]6-FROM-1-TO-11
[49.9]7-FROM-1-TO-11
......To be continued
// 分析
很多同学没有实现向正在reset的电梯分发请求的功能，所以当喂给他们这组数据后
，他们会将全部请求全部分发给1号电梯，那自然就会tle了。也正因如此，作者通
过这组数据hack掉了同房间的所有人。
```

7 心得体会

与第一单元总结一致，我最想说的还是层次化设计思想的重要性，我们需要在设计时能够清晰地意识到每个模块的职责，然后对代码进行抽象，比如当调用一个方法中不考虑代码的实现细节，只考虑代码的输入要求、输出形式及实现功能。

其次，通过这次作业，我对多线程有了更加深入的认识，从单线程到多线程，增加了程序的性能，那代价是什么呢，那应该是代码工程量的上升，具体也就体现在如何实现线程安全、如何实现线程协作、如何避免死锁等bug。

最后的最后，我还想说，多多思考，多多交流，这是最重要的。

> ***特别感谢q学长、xm、xf同学对我OO学习的帮助，特别感谢yzf同学对我的陪伴，感谢其他对我OO提供过帮助的所有人***