BUAA_OO_第二单元总结

前言

本次面向对象第一单元总结文档将从以下几个部分展开：

1. 基本思路和框架
2. UML类图
3. 代码规模和复杂度分析
4. 互测与hack攻略
5. 心得体会

第五次作业

目标是要熟悉 Java 多线程的基本操作，包括线程对象与共享对象、线程的创建、线程通信、互斥与锁机制，熟悉**产销模型**。需求是搭建一个包含6个纵向电梯的调度系统。和往届的不同是每名乘客新增了优先级属性，表示该乘客到达终点楼层的急迫程度。

基本思路和框架

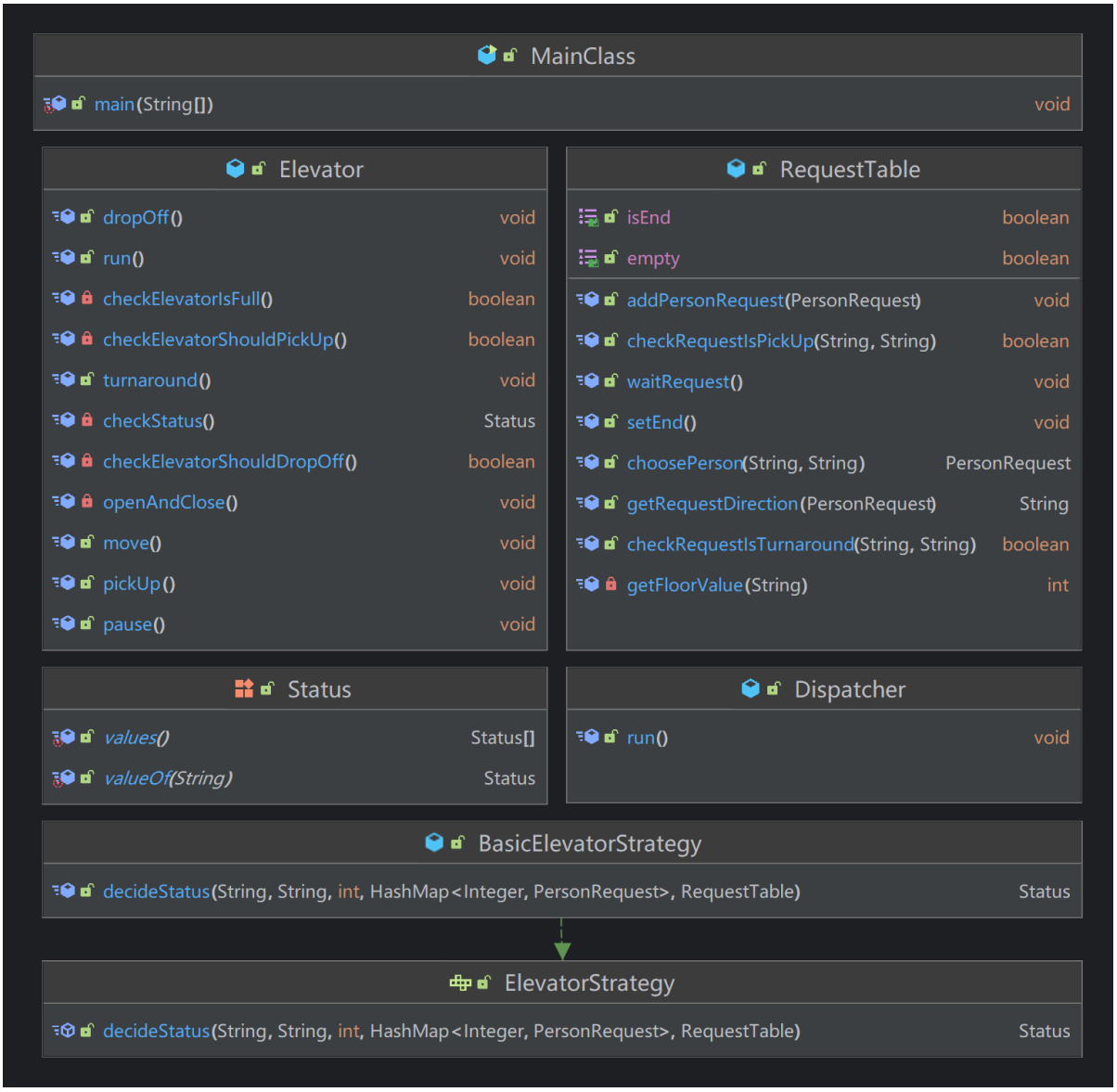
1. **产销模型。**为了保证良好的工程架构，一开始就参考实验情景构造**产销模型**。输入线程作为生产者，电梯作为消费者。初步分析得到至少需要以下几个独立类：
   - **电梯类。**方法有：开门-上客-下客-关门、楼层间移动、停留等待、调头、终止。
   - **候乘表类。**每个电梯都需要有自己独立的候乘表。该候乘表管理自己对应电梯上的乘客请求，并根据当前候乘表提供一些方法，例如添加乘客请求、判断是否需要上下客或移动等待调头终止等。
   - **策略类。**为了实现**单一职责原则**、**单例模式**和**监视器模式**，形成良好的架构，必须要把电梯类和策略类相分离。策略类的核心作用是**根据电梯和候乘表双方提供的状态信息，综合判断电梯的下一步行为。**实现电梯类和策略类相分离的好处在于，方便封装不同的策略（继承一个策略接口），在后续作业**不同情况下选取不同的策略。**
   - **分配器类。**作用是承接输入接口，把从输入接口获得的乘客请求分配到不同的电梯候乘表中。

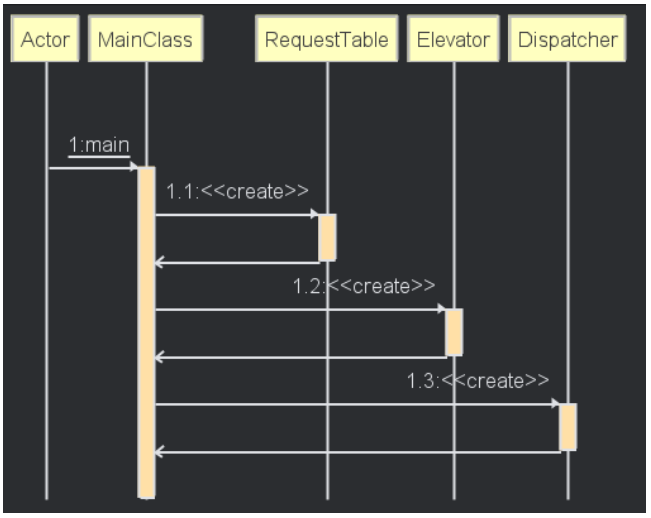
   其中调度器类、电梯类是线程对象，做到互不干扰。我专门设置了一个枚举类用于存放电梯的所有状态。

2. **捎带策略。**参考指导书提供的 ALS 策略和往届广泛使用的 LOOK 策略，我最终采用 **PRI-LOOK 算法**（模仿LOOK的核心思想并增加优先级权重）。该算法的具体实现如下：
   - 主请求选择规则：
     - 在候乘楼层存在多个请求时，优先选择**乘客目标方向与电梯当前运行方向一致**的请求；
     - 在方向一致的条件下，从中选择**最高优先级**的请求；
     - 若优先级相同，则选择**目标楼层最近**的乘客（即目标楼层距离当前楼层最小，在大规模数据下能有效减少绕路）。
   - 被捎带请求选择规则：
     - 电梯当前未满（未达到最大承载量）；
     - 只捎带**与电梯当前方向一致**的请求；

        - 在当前楼层，按前述优先级策略逐个筛选进电梯，**不考虑同一楼层请求之间的时间差**，仅在**方向一致 + 优先级 + 距离近**这三个维度上**依次**进行筛选。
3. **线程通信。** 锁机制和频繁的唤醒能有效保障线程安全，但会降低程序性能。**不要让一个线程对象访问另一个线程对象，线程对象只能访问共享对象。** 电梯、调度器都是线程对象，相互之间不能访问。他们呢均访问共享对象：候乘表。
    - `Elevator` 类中显然不需要 `synchronized` 方法，因为内部状态**全是私有的**、只被自己线程访问，**不会被其他线程共享或并发访问。**
    - `RequestTable` 不是线程类，但方法必须要加 `synchronized`。这是**典型的"监视器模式"**：**不是线程类的类**，反而是**线程之间共享的数据类**，需要用 synchronized 来保护它的状态。因为**多个线程会同时访问它**（例如 `Dispatcher` 类调用 `addPersonRequest()`、`Elevator` 类调用 `choosePerson()`），所以它需要通过 `synchronized` 来加锁，保证每次只有一个线程能访问其内部数据结构。
    - `notifyAll()` 也是广泛使用于 `RequestTable` 中，为了减少不必要的唤醒，只在真正会改变线程状态时使用（例如选到人时、`boolean` 方法为 `true` 时）。
4. **优化策略。**除了捎带策略的优化，主要做了 "抢跑"（也可以认为是**量子电梯**）。第一个数据通常是0.4秒之后（互测要求1秒之后），假如第一条乘客请求不在 F1 的化，等到第一条乘客请求发出电梯再启动取接人，太慢了。因此可以通过**动态修改**电梯 `move` 方法的 `waitingTime`（而不是卡死在400ms）来实现以下行为：假如第一个乘客请求时间戳为 [1.0]，出发楼层为 B1，则电梯在 1s 时刻可以直接 `ARRIVE` 在 B1，相当于在0.6s 时就 "**预判**" 了该请求。本质上就是：**电梯的等待和线程的等待是两码事，线程需要取决于输入进行等待，但电梯的行为可以灵活处理。**然而，量子电梯不是绝对优化，在某些路线上也会产生严重弊端：路线繁杂多绕路。

UML 类图与协作图


MainClass
main(String[])
void
Elevator
dropOff()
void
run()
void
checkElevatorIsFull()
boolean
checkElevatorShouldPickUp()
boolean
turnaround()
void
checkStatus()
Status
checkElevatorShouldDropOff()
boolean
openAndClose()
void
move()
void
pickUp()
void
pause()
void
RequestTable
isEnd
boolean
empty
boolean
addPersonRequest(PersonRequest)
void
checkRequestIsPickUp(String, String)
boolean
waitRequest()
void
setEnd()
void
choosePerson(String, String)
PersonRequest
getRequestDirection(PersonRequest)
String
checkRequestIsTurnaround(String, String)
boolean
getFloorValue(String)
int
Status
values()
Status[]
valueOf(String)
Status
Dispatcher
run()
void
BasicElevatorStrategy
decideStatus(String, String, int, HashMap<Integer, PersonRequest>, RequestTable)
Status
ElevatorStrategy
decideStatus(String, String, int, HashMap<Integer, PersonRequest>, RequestTable)
Status


Actor
MainClass
RequestTable
Elevator
Dispatcher
1:main
1.1:<<create>>
1.2:<<create>>
1.3:<<create>>

代码规模

Source File	Total Lines	Source Code Lines	Source Code Line...	Comment Lines	Comment Lines [...	Blank Lines	Blank Lines [%]
BasicElevatorStrategy.java	28	24	86%	0	0%	4	14%
Dispatcher.java	41	36	88%	1	2%	4	10%
Elevator.java	173	143	83%	10	6%	20	12%
ElevatorStrategy.java	8	6	75%	0	0%	2	25%
MainClass.java	20	16	80%	0	0%	4	20%
RequestTable.java	123	103	84%	3	2%	17	14%
Status.java	3	3	100%	0	0%	0	0%
Total:	**396**	**331**	**84%**	**14**	**4%**	**51**	**13%**

互测与hack策略

本单元 hack 的唯一策略是大规模评测机。在A屋中，经过一天一夜超过10000个顶格数据的轰炸，找出了天璇星的一个 bug。天璇星还是我的同学。

第六次作业

乘客请求中不再指定想要乘坐的电梯；新增电梯的临时调度功能，电梯拿到临时调度任务时，应以新的速度抵达目标楼层，开门清空，等待1s后关门；修改了 OUT 输出格式。

基本思路和框架

1. `RECEIVE` 的设计。主要修改的是 `Dispatcher` 类中的分配策略。实现了平均分配（模6加1）和影子电梯两种分配策略，广泛测试后发现在某一时刻积压大量请求时，影子电梯表现明显劣于平均分配。故采用**平均分配**策略。
2. `SCHE` 方法的设计。把临时调度看作电梯的一个行为（类似于开关门、上下行）。一旦检测到电梯处于 `SCHE` 状态（SCHE-ACCEPT之后，SCHE-BEGIN之前），则立即依次进行下列操作：
   1. 在当前楼层立即开门，清空轿厢内所有乘客，清空候乘表。
   2. 重置速度，以指定速度前往目标楼层。
   3. 在目标楼层开门，持续1000ms，关门，输出SCHE-END。

   这里有几个关键点需要说明：
   - 题目中说前往目标楼层时可以携带乘客，但是为了减少乘客绕路，我们选择**接到临时调度请求后就地下客**，下来的这些乘客和候乘表中的请求交由其他电梯善后。
   - 若清空轿厢内乘客时该乘客还没有到达终点，那该乘客重新返回候乘表。中途下电梯的乘客和从起始位置出发的乘客对程序而言没有任何区别。
   - 为了实现中途下电梯的乘客可以由其他电梯接管，要实现一个总候乘表。因此有必要**分离输入线程和分配线程**，在输入线程中，我们把所有乘客请求先全部放到总表中，临时调度请求直接分发给电梯。在分配线程中，从总表中取出一个请求，通过平均分配的方式分配给一个电梯执行。对于要中途下电梯的乘客，只需把该乘客请求重新返回总表中即可**打通各个电梯的接管**。

代码规模

Source File	Total Lines	Source Code Lines	Source Code Lin...	Comment Lines	Comment Lines ...	Blank Lines	Blank Lines [%]
BasicStrategy.java	34	29	85%	0	0%	5	15%
DispatchThread.java	45	40	89%	1	2%	4	9%
ElevatorThread.java	291	234	80%	26	9%	31	11%
InputThread.java	43	38	88%	0	0%	5	12%
MainClass.java	28	21	75%	0	0%	7	25%
RequestTable.java	170	143	84%	3	2%	24	14%
Status.java	3	3	100%	0	0%	0	0%
Strategy.java	8	6	75%	0	0%	2	25%
Total:	**622**	**514**	**83%**	**30**	**5%**	**78**	**13%**

影子电梯

该方法最终目的是**把一个乘客请求最优分配出去。**方法是为**每个不在** `SCHE` **中的电梯**创建一个副本。当一个乘客请求来临时，我们假设6个电梯都不在 `SCHE` 中，那么理论上这个请求可以分配给6个电梯中的每一个。故有6种分配方式。下面选取最优方式如下：

- 首先，把6种分配方式看成6批。每批有6个电梯，因此我们需要建立36个电梯副本。
- 然后，**对每一批**，模拟该批6个电梯每个的运行。注意在这种情况下，不会真正 `sleep` 以耽误整个真实系统运行，而是为每个电梯建立一个总运行时间属性，然后例如移动就每次加 400ms 即可。这样，该批次每个电梯都拥有了自己的模拟运行时间。
- 接着，**对每一批**，选取该批6个电梯的**最长运行时间**作为该批次的运行时间。
- 最后，**遍历批次**，找到最短运行时间的批次。该批次对应的分配的电梯号就是我们理想中的电梯。

经过实验，发现这整个过程系统运行时间在几十毫秒级别。这是忍不了的，因此影子电梯一定要配合量子电梯使用。

然而，开头说过，影子电梯在**某一时刻积压大量请求时**，表现明显劣于平均分配。为什么会这样呢？原因是影子电梯是**局部优先策略**，是对每一个请求都根据当前的情况分配性能最优的电梯。然而，假如某时刻积压大量乘客请求，此时有5个电梯全在临时调度，那么显而易见影子电梯会将该时刻所有积压的乘客请求直接分配给剩下的最后一个电梯。这比均分明显差很多，因为等临时调度的电梯结束之后，它们显然仍然可以参与调度。

bug分析

由于使用量子电梯过度，我在所有的 `sleep` 中并不固定时间（如400ms），而是固定时间加上当前时刻减去上次电梯启动时刻，想抢占一些程序运行时间。结果导致强测中每个点的 `move time` 都少一点点，没有进入互测。

第七次作业

新增双轿厢改造功能。

基本思路和框架

双轿厢改造功能从行为描述上和临时调度十分相似，因此仍然可以当作电梯的一个行为（指 `UPDATE` 是一个行为）。从宏观角度，本次作业有两种大思路：

- 思路一：对现有电梯类增加 `maxfloor`、`minfloor`、`transFloor` 属性，通过限制电梯的楼层范围来隐式实现"双轿厢"。思路一的优势在于代码扩展量较小，但是电梯线程类会混合两种功能，不利于**单一职责原则**。

- 思路二：新建一个双轿厢电梯类，在其中实现特定的功能。思路二的优势在于可以实现单一职责原则，但是双轿厢电梯类和电梯类中有许多相似的行为，造成一些重复性代码。

经过权衡考虑，采用思路二。

难点与细节

本次作业主题是"线程交互"，因此线之间的联系与交互是实现的难点。具体而言，要巧妙实现：

- 两个电梯均在清空轿厢结束并关门后才能输出 `UPDATE-BEGIN`，且只输出一遍
- 两个电梯均在输出 `UPDATE-BEGIN` 后，别的电梯才能 `RECEIVE` 他们的乘客等待请求，而**两个电梯释放候乘表是在 `UPDATE-BEGIN` 之前还是之后，没有规定。**

于是，我采用了一个机器精妙的架构：借鉴OS中进程同步（PV操作）的知识，构建了如下 `crossNotify()` 的小结构：

```
public void crossNotifyOut(Object lock) {
    synchronized (lock) {
        lock.notifyAll();
        try {
            lock.wait();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    synchronized (lock) {
        lock.notifyAll();
    }
}

public void crossNotifyClear(UpdateRequest updateRequest, String abType) {
    synchronized (updateRequest) {
        // 更新双轿厢开始
        if (abType.equals("B")) {
            TimableOutput.println("UPDATE-BEGIN-" +
updateRequest.getElevatorAId()
                + "-" + updateRequest.getElevatorBId());
        }
        updateRequest.notifyAll();
        try {
            updateRequest.wait();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    synchronized (updateRequest) {
        updateRequest.notifyAll();
    }
}
```

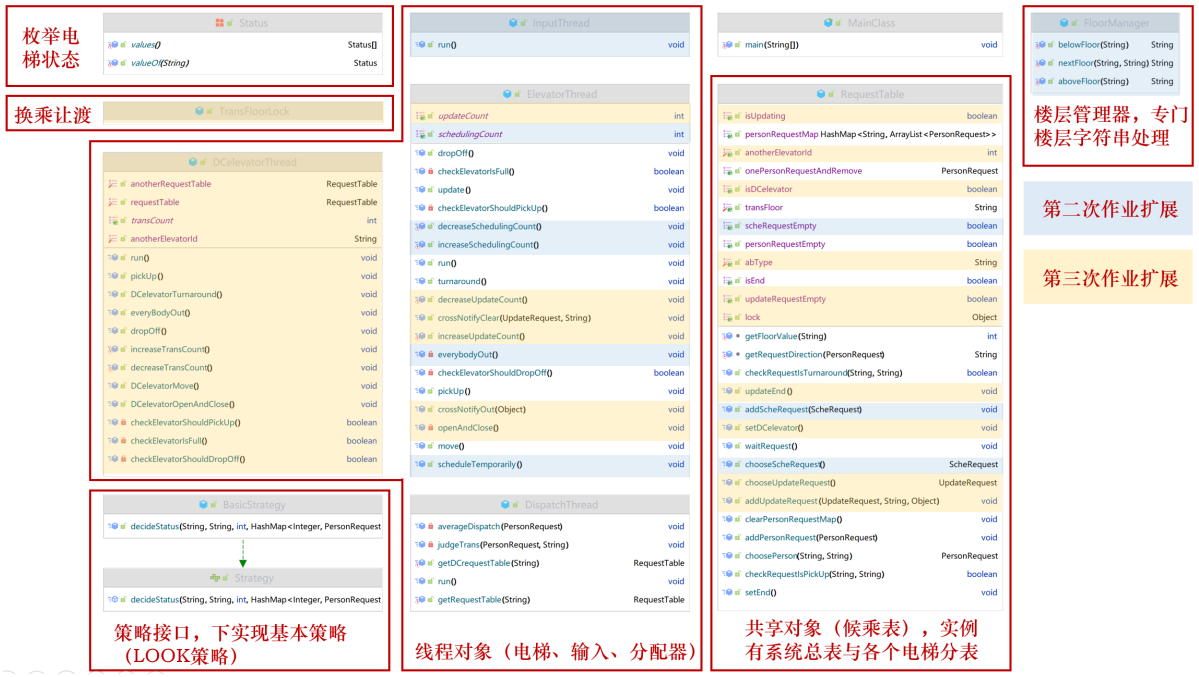
在 `update` 方法中进行如下设计，即可进行同步：

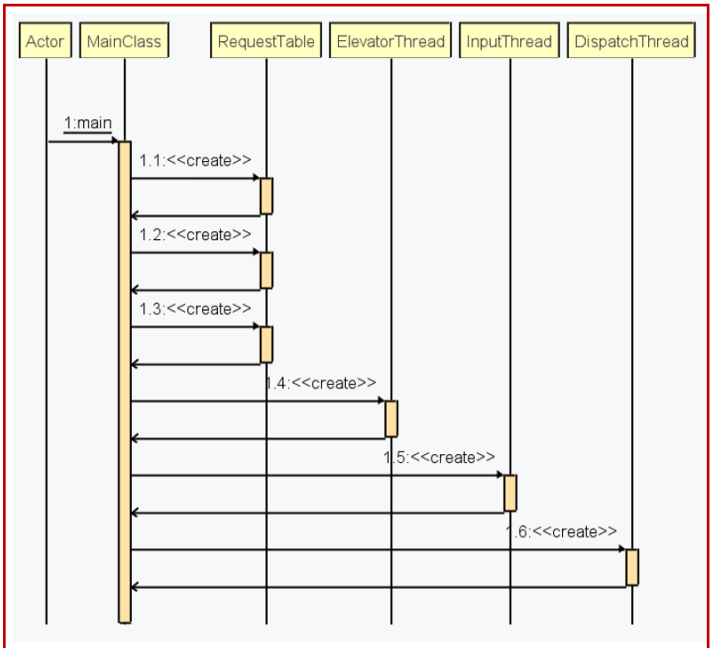
```
everybodyOut();
crossNotifyOut(requestTable.getLock());      // 同步清空电梯
crossNotifyClear(updateRequest, abType);     // 两个电梯均清空后由一个电梯输出 BEGIN
synchronized (globalRequestTable) {
    // 清空当前候乘表
}
if (abType.equals("B")) {
    // 新建2个双轿厢电梯线程
}
```

要注意的是，两次同步行为必须要用不同的锁，把两个配对电梯共享的某个对象当作锁就可以。这样就实现了**不会在一个电梯输出 `UPDATE-BEGIN` 后另一个电梯还在下人，也不会出现还没输出 `UPDATE-BEGIN` 之前就有其他电梯 `RECEIVE` 到这两个电梯候乘表中的人。**

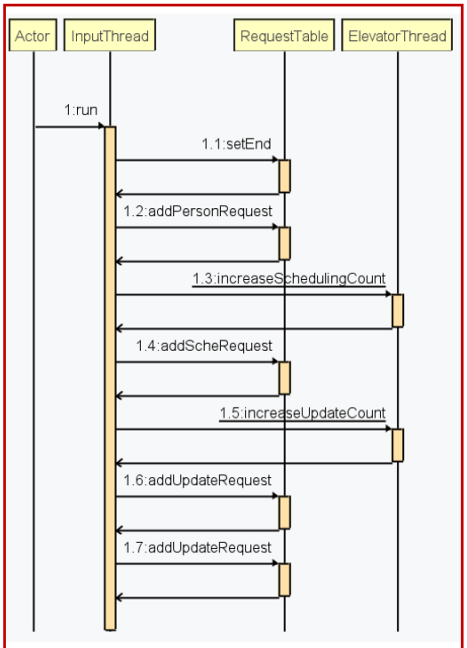
同样的思路可以用到两个电梯如何在换乘楼层不相撞，而是相互避让上。可以设置一个专门的换乘楼层类（并创建对象，单例模式），在双轿厢电梯类中的 `move` 方法时，判断下一楼层是不是换乘楼层。如果是，就加锁，**使电梯井中同一时刻最多仅有一个电梯持有该锁。**这就实现了相互避让（互斥访问）。

三次作业整体UML类图与协作图

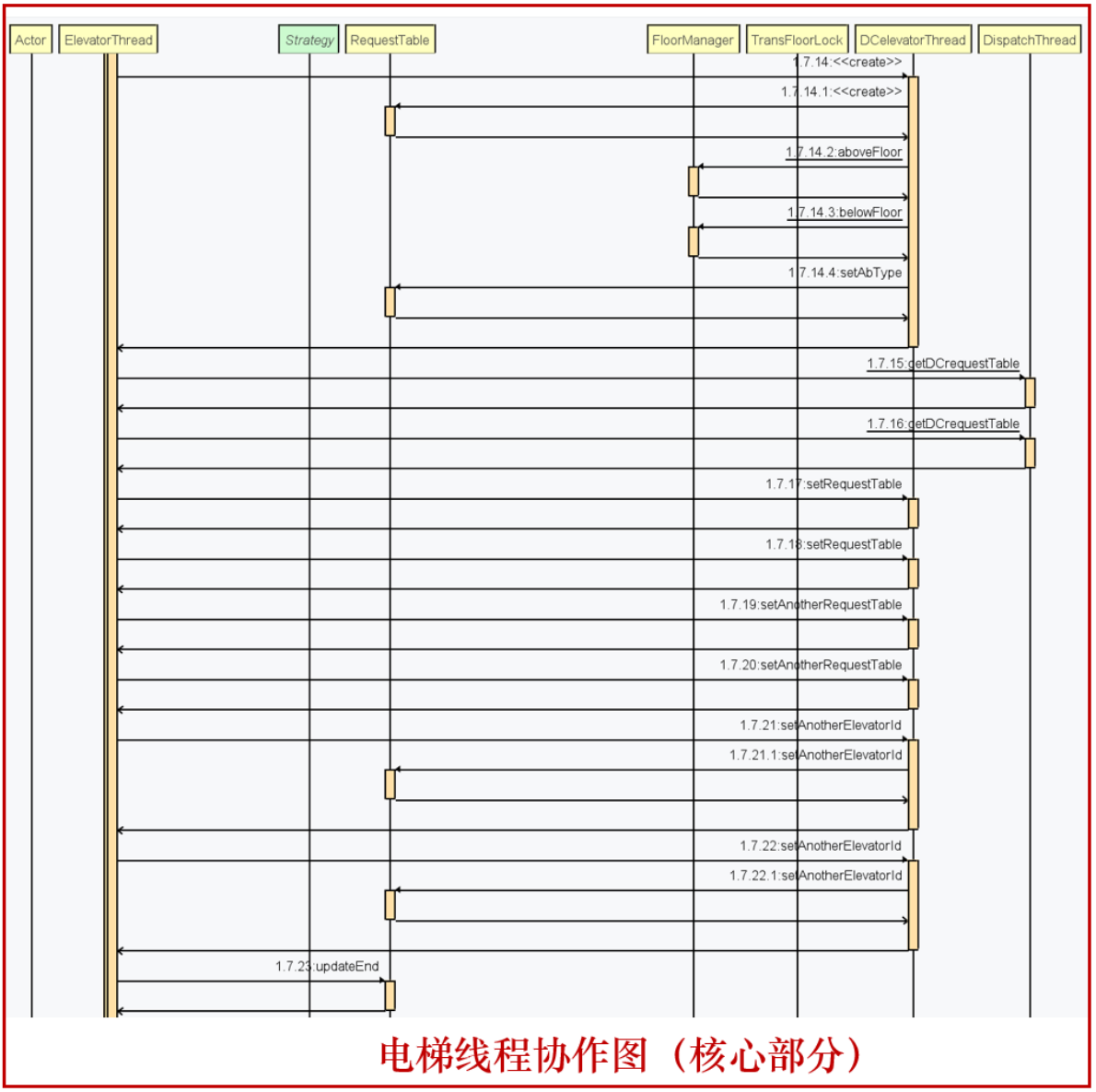




主线程协作图



分配器线程协作图



电梯线程协作图（核心部分）

总结体会

第二单元是难度最大的一单元，是第一次接触并行程序设计和多线程设计思想，也是我认为这门课的精华所在。从第一次作业初识多线程设计，设计独立的6个电梯线程互不影响独立工作，到第二次作业对一些线程进行特殊扩展，再到第三次作业涉及到线程之间的同步和互斥，我对线程的独立并行与通信协同有了实践的认识。

三次作业中我始终采用 **LOOK 调度策略和平均分配原则**，尝试了局部最优（影子电梯）和全局平衡（均分），最终我更推荐相对均衡普适的原则，因为**任何局部最优策略都有风险。**

- 在**同步块的设置和锁的选择**方面，我始终采用 `synchronized` 关键字，但在实验中对读写锁有了基本的了解。这使得一定程度上减少了程序的并发能力，但是提高了可读性和简洁性。
- 在**线程安全**方面，宏观来说共享对象的非只读方法必须加锁，细致来说要仔细分析各个线程之间的逻辑和访问区间，明确共享资源，对特定的需求和情景设计不同的措施（例如第七次作业那个巧妙地交叉同步）。线程安全要谨慎应对、**灵活处理。**
- 在**层次化设计**方面，我架构较为清晰，线程对象有：电梯、分配器、输入。共享对象有：候乘表。策略是一个接口，下面实现不同策略，根据不同情况选取最优策略（然而最后比较下来还是 LOOK 策略全局最优）。
- 在**性能优化**方面，出了量子电梯、影子电梯等方法，我认为以下捎带策略可以在大规模数据中有效减少绕路，实现时间和电量较优：每到一个楼层时，优先**依次**捎带该楼层**方向一致、优先级高、前往距离近**的乘客。
- 在 **debug** 方面，我增长了许多经验。对于能够稳定复现的 bug，首先通过**条件断点**进行单线程调试，看当前对象运行有没有逻辑上的错误或代码上的笔误。对于不能稳定复现的 bug 或涉及到线程通信的 bug，最好的方式就是在程序关键节点输出调试信息，获取现场情况。
- 三次作业**稳定**的内容是我的架构（线程对象、共享对象、策略类），**易变**的是其中具体的行为和需求，以及线程之间的协作。

第二单元非常精彩，第三单元加油 ~