OO第五次作业解析

一.基本概念

三四月春仍就夹杂着凉意，OO的第二单元也如同寒冷的幽魂一般悄然而来。在本单元，我们首次接触到了**多线程**的概念，并将在三次作业中迭代一个多线程电梯系统，实现对于不同需求的人员的调动。很多人认为JAVA中最大的难点就是并发编程，即多线程，我们在如此短暂的时间了解清楚并熟练掌握多线程程序的编写方法显然是不可能的，在作业中的确也不太需要太过深入复杂的知识(但是有可能出一些奇怪的Bug导致不得不去了解一些深入的知识)，但是首先有一些基本概念必须要搞懂，笔者在此再次强调。

1.synchronized

直译即同步，主要解决的是多个线程之间访问资源的同步性，可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行，是一种**锁**。某线程必须得到这把**锁**才可进入相应的代码块。具体而言又分为两种锁。

- **对象锁**：在 Java 中，每个对象都会有一个 monitor 对象，这个对象其实就是 Java 对象的锁，通常会被称为“内置锁”或“对象锁”。可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。类的对象可以有多个，所以每个对象有其独立的对象锁，互不干扰（类的不同对象之间异步）。
- **类锁**：在 Java 中，针对每个类也有一个锁，可以称为“类锁”，每个类只有一个 Class 对象，所以每个类只有一个类锁（类的各个对象之间同步），本次作业不会用到。

syn关键字在代码中的实现几种情况:

- **syn修饰方法或类**: synchronized修饰静态方法等价于修饰类，是获取了类锁;synchronized修饰非静态方法等价于修饰该实例对象，是获取了对象锁
- **syn修饰代码块**: 被代码块包围的区域称作**临界区**，一次只能由一个线程在临界区内，具体锁类型根据syn后锁的目标依据上述规则(对象or类锁)来判断，在本次作业中会出现一些因此而不好调和的矛盾。

使用syn后，产生了**一把**能将锁的目标打开的钥匙，一个线程必须得到这把钥匙才能进入相应的方法or代码块，而没有得到钥匙的线程只能在临界区外等待着钥匙的出现。

当线程在临界区执行完全部代码后，会交出钥匙

2.wait，notify,notifyAll

这些方法必须写在syn的里面

两个基本概念

- 争夺池: 其他线程运行到临界区，却没有钥匙时，会进入争夺池中。
- 等待池: 只有wait方法能让该线程进入等待区，等待被唤醒。

每个被锁的目标都有独立的等待池与争夺池， 一定一定要注意锁的目标的统一

(1)wait

使用方法: 锁的目标.wait(),意思是当前拥有钥匙获取到锁的目标的线程等待挂起，**并将钥匙交出去**。该线程会进入**等待池**中，等待着相应的notify方法对该线程进行唤醒，被唤醒后该线程会进入**争夺池**(自己编的)中，与所有在争夺池中的线程争夺下一次钥匙的使用权，若该线程成功得到钥匙，则会从上次等待的地方开始继续运行。

(2)notify, notifyAll

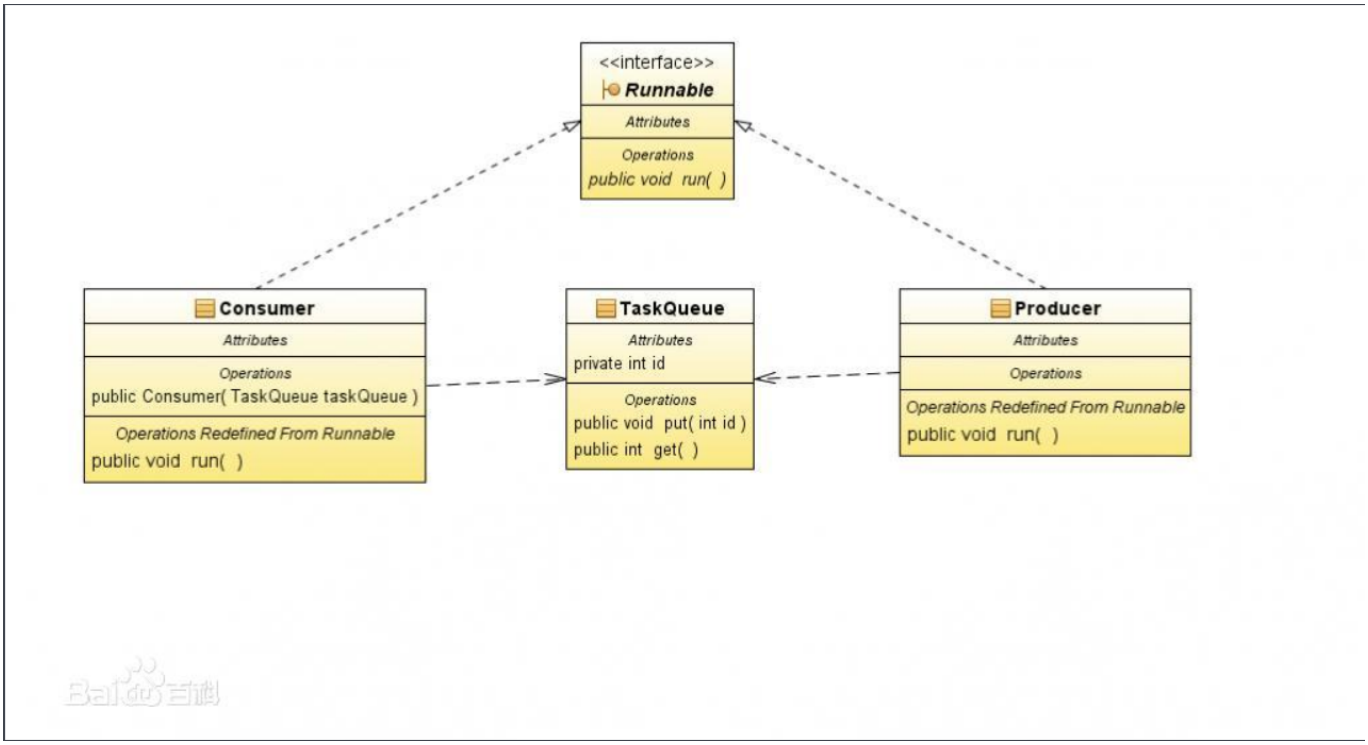
notify是随机唤醒一个等待池中的线程，notifyAll是唤醒全部等待池中的线程。在notify后，当前线程不会马上交出钥匙，要等到当前线程被synchronized修饰的代码执行完，才会释放。

3.sleep

令当前线程睡觉一段时间(即挂起)，可自行设置时间，该方法与syn无关，可在任意地方执行，因此挂起后即使有钥匙也不会交出。

4.生产者消费者模式

本次作业中可能还需用到**观察者模式**，**单例模式**等，但这些较好理解，重点介绍生产消费模式。 在该模式中，组成包括生产者，消费者，托盘(缓冲区), 每个生产者各是一个线程，负责生产对象并将对象运送到托盘中，每个消费者各是一个线程，负责从托盘中拿走对象并消费对象，那么在多线程运行时，托盘会被多个线程访问并修改，若同时发生则会出现问题。因此托盘是共享对象，一次只能允许一个线程进入托盘对象中进行增删的修改，那么托盘中的方法便需要加上syn修饰。进一步说，消费者在托盘中没有产品时应选择等待，而不是白白一遍又一遍轮询占用资源，而当生产者向托盘补充产品时要唤醒等待的消费者，若产品满足消费者的需求，则进行消费，若不满足，应继续进入等待状态。



二.题目分析

本次作业要求实现有六台电梯的多线程电梯系统，完成对人员的调度，在正确调度的基础上，还应保证总用时尽量短，电梯运行层数尽量少，人员等待时间尽量短。由于涉及到多台电梯协同工作，因此从中找到一种完美的调度的方案是不可能的，一方面原因是电梯系统无法预知未来的人员请求，另一方面则是因为要令上述三种

性能同时最优显然是不可能的(除非有大佬用给出的性能公式每种情况去遍历计算，我相信这是办不到的)，因此我们的调度算法应完成某种**tradeoff**，正所谓鱼和熊掌不可兼得。这正是本单元率先教会我的道理。

分析从两方面考虑:

1.架构

显然，不难发现这次的电梯模型很像是生产者消费者模式，输入器作为一个线程，在时间线里向请求队列(托盘)中投放人员需求，而每台电梯作为一个线程，从请求队列中取出请求，不过当然没有那么简单，因为电梯还需要去对应的楼层找到请求，并将其送往应去的地点。我们认为**电梯是没有脑子**的，而幻想出来一个**长脑子的调度器**，他会告诉每一台电梯应该怎么做，电梯只需要根据调度器的指示完成相应的操作即可，如上下楼，开关门等。我们的核心算法将在调度器内实现。由于想更合理的规划六台电梯的运行，笔者的调度器里包含着请求队列以及每台电梯的全部信息，而电梯并不直接与请求队列相连，而选择通过调度器统一完成信息之间的传递。

2.算法

首先我们先思考单台电梯的调度算法，LOOK算法显然是最稳妥的做法，并且再新增一条规则，当无请求的时候选择原地等待减少电量损耗。那么六台电梯呢，是否跟一台电梯有某种共同之处，笔者和身边人大概分为了三种思路。

- 基于LOOK的自由竞争法: 对每一台电梯统一使用LOOK法调度，对于每个请求而言哪台电梯先抢到算谁的，可以预想到将会有许多台一直在做无用功，电梯耗电量极高，这正是本方法最大的缺点，同时这六台电梯某种意义上称得上是一个**整体**，失去了多台电梯有更多可能性的优势。而优点在于方法简单，只需实现单一电梯的LOOK算法即可。

- 基于LOOK的分派法: 将每一台电梯单独开出一个请求队列，在每个请求来临时，按照某种方式分配到某一台电梯的请求队列中，接着每一台电梯根据自己的请求队列执行LOOK算法即可。分配的方式可以是随机数模6，或者平均分配，或者一些更牛的算法。归根到底就是在请求来临时就已经决定了承载他的电梯，像是命中注定，生死纠缠。这种方法的解决了电梯空跑的问题，让每台电梯都能投入其中，发挥了多台电梯的优势，然而不足之处在于这种命中注定的模式，缺乏变通性，并且随机分配等方法导致这一切太随机了，这次性能很好下次可能就很差。

- 宏观调控/动态规划法(瞎起的): 这种方法正如其名，是动态的实时的根据每台电梯的状态来分配请求，每台电梯没有一个明确的目标，且每台电梯的行为不仅取决于请求队列，还取决于其他电梯的状态，例如当一台空闲电梯在1楼，另一台在10楼，面临一个来自11楼的请求时，会让10楼的电梯去接，而1楼电梯保持不动。事实上这种方法一定不是完备的，当请求数变多变复杂时，难免会出现没有考虑到的情况，且算法思路有很多很多，针对某个逻辑判断或者说算法，某些情况下更好而某些情况会更烂，因此设计的思路将千奇百怪且没有最优解，笔者也仅是给出自己的思路抛砖引玉。这种方法的优点很显然可以省电，一定令电梯尽可能少动，相对的运行时间和人员等待时间**也许会**更多，属于是资本阶级压榨人民了。

三.类分析

 UML

1.Input 输入类:

继承Runnable接口，是一个线程，用于定时的输入，需要注意的在结束时应往请求队列设置一个结束标记，来结束各个电梯线程，类似**观察者模式。**

2.Elevator 电梯类:

继承Runnable接口，是一个线程，他会不断执行run内的getOperation方法，向调度器征询当前应该执行的操作，分别为枚举类里的**WAIT**,**OVER**,**MOVE**,**OPEN**,**TURN**，即原地等待，线程终止，向当前方向前进一层，开门并执行后续进出及关门操作，转向。

3.RequestQueue 请求队列类:

共享对象，其中方法皆上锁处理，里面用HashMap存储每一层楼的(fromFloor)的请求，并存储停止标记。一共有两处需notifyAll解放全场，一处是执行完set方法后notifyAll，大赦天下告诉电梯们有新请求该去处理了，去选一台电梯去处理吧，另一处是在判断是否结束为true时需要大赦天下，令苦苦原地等待的线程终止，不然会超时。

4.Scheduler 调度器类:

本次作业核心类，内部存有请求队列和电梯的全部信息，主要通过实现getOperation方法来为各台电梯提供指导，并在该类实现判断电梯该进行哪一步骤的算法。总体流程逻辑为:

```
st=>start: Start
op=>operation: getOperation
cond1=>condition: 电梯是否需要开门
st->op->cond1
cond2=>condition: 电梯内是否有人
cond3=>condition: 请求队列是否为空
cond4=>condition: 输入是否已经结束
cond5=>condition: 电梯能否找到合适接的人
cond6=>condition: 电梯后方能否找到合适接的人
move=>operation: MOVE
move1=>operation: MOVE
open=>operation: OPEN
over=>operation: OVER
wait=>operation: WAIT
turn=>operation: TURN
waittime=>operation: WAIT(a period time)
cond1(yes)->open
cond1(no)->cond2
cond2(yes)->move
cond2(no)->cond3
cond3(yes)->cond4
cond4(yes)->over
cond4(no)->wait
cond3(no)->cond5
cond5(yes)->move1
cond5(no)->cond6
cond6(yes)->waittime
cond6(no)->turn
```

判断顺序很关键，可以说是精髓中的精髓，而相较于单台电梯的算法，笔者加入了宏观调控机制。在笔者看来，一台电梯一旦有人，那么他的行为就已经确定，优先满足电梯中的乘客即可，能捎带的捎带，不需要考虑和其他电梯的配合；而开关门取决于来临的乘客是否同向以及电梯内乘客是否下车，因为此时既然面临开关门，那么该电梯就一定针对当前层的请求是**最近**的，因此开关门逻辑也不应该有其他电梯的影响。而当一台电梯内部为空而请求队列有需求时，此刻便是**宏观调控**发挥作用地方，笔者选择从远及近扫描请求(应该符合现实)，由于当前的电梯未必是接该请求最优的电梯，正如我上文举得例子，因此需要将该台电梯的状态与其他五台电梯的状态进行对比判断，这不仅取决于其他每台电梯的楼层，也取决于其他电梯此刻是在**找寻猎物**还是在**送乘客**(根据其他电梯内部有无人，笔者又分了两种情况讨论，因为找寻猎物时方向可以不固定，而送乘客时有内部乘客目的地的限制)，一旦发现有一台会更好的接上乘客，那么本台电梯便会放弃这个需求去扫描下一个需求，如果该方向的所有需求皆不需要它去接，则它会反方向以同样的方式扫描其他需求并判断当前层是否有正向接不了反向能接的请求，若有，则选择转向，若同样也无，则便让当前咨询的电梯原地等待一段时间，**注意千万不要令其无限期wait**，因为其实此时请求队列中仍然是有请求的，而我们只有来新需求时才会解放全场，这可能导致后面的某一时刻当前电梯针对某一请求成了最优解，而此刻因无新请求的到来而无法唤醒，因此我们需要定期的唤醒它，使得它时不时就进行判断(之所以要wait是因为CPU的限制，如果CPU没有限制则轮询会更好)。至于如何判断哪台电梯是较优解的算法，笔者所想仍有较大局限性，应付不了复杂场面，因此就不多说了。

5.Main

用于构造对象与启动线程。

四.建议

- 笔者建议还是别用第三种方法了，人要想晕了的同时性能也许不如第二种方法，学有余力的同学可以试试。
- 不要随意notifyAll，想好应当notify的时机。
- 多线程并发，空指针等**玄学**Bug会层出不穷，遍历删除容器元素应用迭代器实现并使用迭代器的remove方法。