第八次博客作业

15061200 李明轩

一， 总结梳理推荐的线程及其协同、同步设计结构

1. 线程之间的关系可以分为三种，父子关系，协作关系，同步关系。
   a) 父子关系：当两个线程之间为创建者和被创建关系的时候，可以说他们属于父子关系。最直接的例子就是我们在 main 方法里调用 Thread.start()，在这里 main 就是父线程，启动的 thread 就是子线程。

      JVM 的线程机制和 windows 并不一样，JVM 中的父子线程并没有生命周期关系。首先，我们需要知道，JVM 在结束运行的时候会唤醒一个名叫 DestroyJavaVM 的线程，它会卸载 JVM 以结束运行释放资源。当父进程结束时若子进程还没有结束，也即父进程不是 JVM 中的最后一个非守护（daemon）进程，它会调用 thread->exit(false)表示不能退出。一直到了子进程结束的时候，由于是最后一个非守护进程，子进程会调用 before_exit 方法并触发线程结束事件和 JVM 结束事件。

      子线程若要访问父线程中的数据可以采用传入引用并加锁（如使用共同的阻塞队列）的方法来实现。

   b) 协作关系：协作关系最典型的例子就是生产者——消费者模型了。实质上生产者消费者模型主要处理的是生产数据和消费数据的线程之间的强耦合关系，为了解耦，我们引入一个第三者——也就是缓冲区，来协调生产和消费能力的不平衡问题。实现该模型有很多种方法，私以为使用阻塞队列应该是最直截了当的一种，生产者线程将产品直接放入队列，消费者从队列中取出，对于队列的读写操作是保证互斥的。

      在近几次作业中对于该模型最直接的使用例子应该是线程池的概念了，我们在声明一个线程池并进行启动的时候，可以限定线程池的大小，如果要运行的任务数量超过了线程池大小，那么多出来的任务就会被放进阻塞队列等待。其余的任务就会直接运行。这里提交任务的用户就是生产者，线程池就是消费者，阻塞队列扮演了缓冲区的角色。或许有的同学会发现，这个例子里并不是生产者先存储，然后消费者取的模式，而是直接给消费者。那么我们也由此引申出一个更快的生产者消费者模型，我们可以直接让生产者供给产品，只有当用不完的时候才由消费者放进阻塞队列。这样和先放进队列再取得的方法相比就会更快一些。

   c) 同步关系：按照 ppt 的解释，同步关系指的是两个或多个线程之间持有和等待的是同一个锁，他们之间存在互斥关系。为了实现这种同步，我们可以将需要同步的多个线程用一个父进程统一调用，然后将父进程传入这些子进程中，从而使子进程之间可以获得并等待互相之间的锁。

      对于实际的应用来说，获取控制台输入和在控制台上输出信息应该是两个互斥的操作，我相信大家也碰到过控制台输出过程中如果强行输入内容就会引发输出内容显示混乱且不定期卡死的现象。这个时候，如果我们在代码中把输入和输出的代码块使用同一个锁进行同步，就可以避免了。

2. 花式锁在线程协同设计中的作用。
   a) PPT 中介绍了三种锁，分别是线程锁（应该指的是类锁？），控制块锁和对象

锁。这三种锁分别有不同的同步范围，线程锁是最大也是最直截了当的解决方案，“你只管忙你的，没有人来打搅”。诚然，这样做解决了线程安全问题，但是也大大降低了并发性，某种意义上来说，过多的线程锁和对象锁会把预期的多线程变成一个单线程程序。因此我们需要控制块锁，尽量减少临界区大小。

在这里我们需要区分的是类锁和对象锁，类锁将对该类的所有实例化对象有效，而对象锁只对一个具体的实例化对象内部各方法有效。

b) 但实际上在应用中，与其说锁是一种线程同步的方法，不如说锁是一种线程安全的设计的思维。因为锁的实现千差万别，java 为我们已经提供了丰富的操作封装了相关的锁，接下来我简要介绍几个这几次作业比较有用的小东西。

i. Atomic 类：比如需要记录两个线程同时对一个变量自增操作，就可以使用 AtomicInteger ，用原子方式更新 int 的值，我们直接声明一个此类型的变量然后++即可。

ii. 读写锁：允许多个读者或者一个写者，但不能同时存在读写。相关实现在 ReentrantReadWriteLock 中。

iii. CountDownLatch 类：通过 countdown()和 await()方法，当计数值不为零的时候阻塞。一个线程(或者多个)，等待另外多个线程完成某个事情之后才能执行。

二， 作业程序结构分析

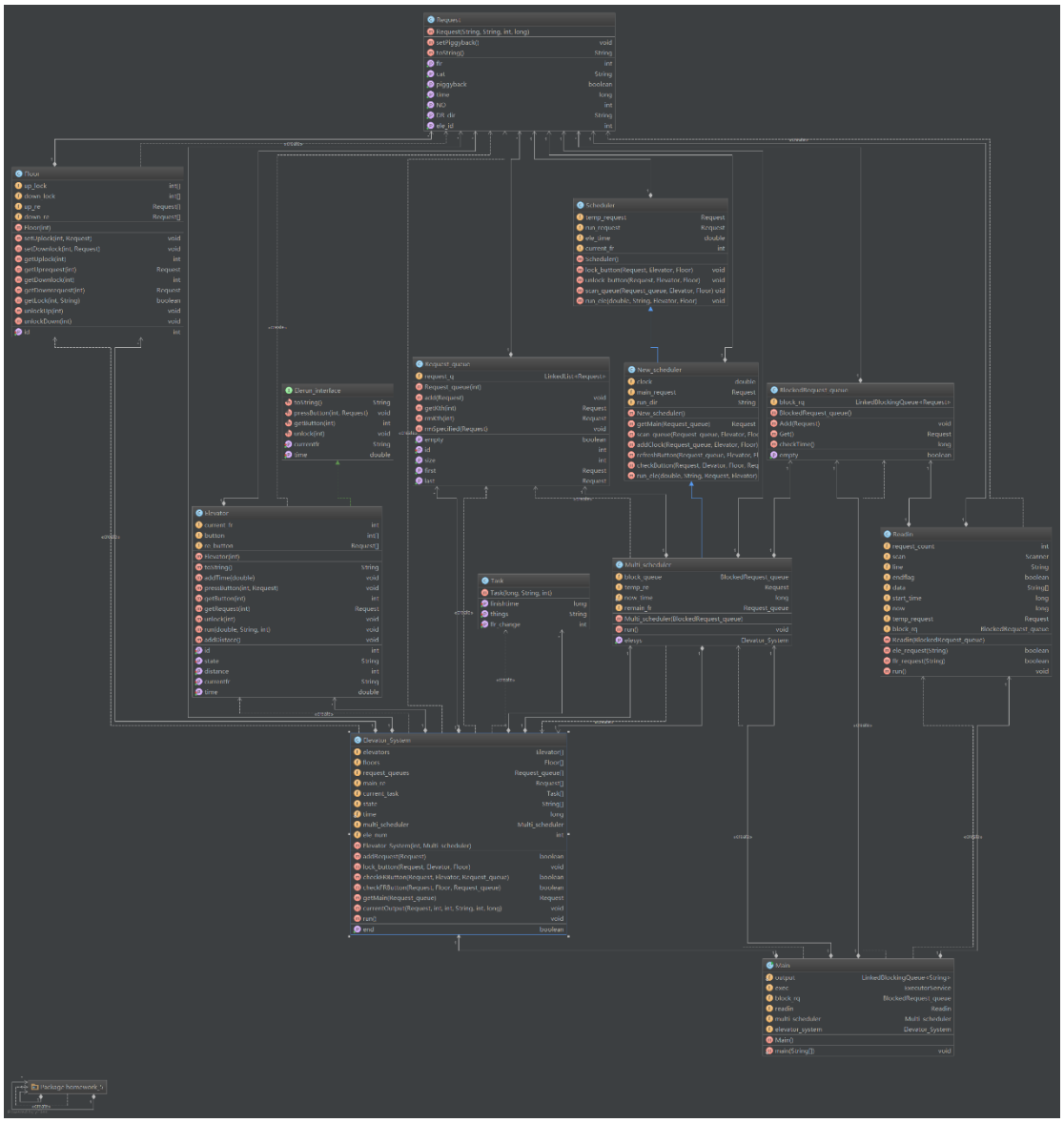
（类的方法细节统计中忽略了各种 getter setter）

1. 第五次作业

a)

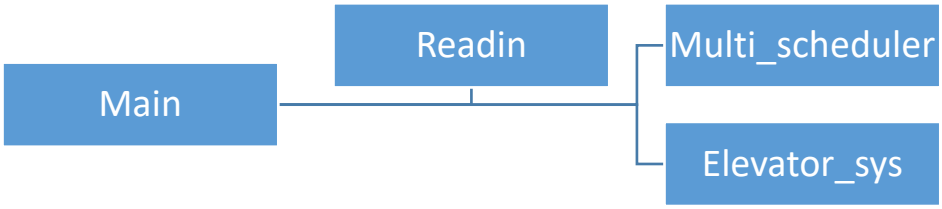
类名	属性个数	方法个数	分支个数	总行数
BlockedRequest_queue	2	4	0	36
Elevator	8	9		135
Main	6	2		74
Multi_scheduler	5	2		60
New_scheduler	3	7		156
Readin	9	4		143
Request_queue	6	5		60
Request	7	3		80
Scheduler	4	5		80
Task	3	1		37
Elevator_System	10	8		277
Floor	5	2		87

b) 通过类图来分析类之间的关系



c) 通过线程协作示意图来分析线程关系

本次作业一共使用了四个线程。他们的协作关系如下：主线程创建了三个子线程，Readin 负责读取控制台输入并将输入加入阻塞队列，Multi_Scheduler 负责从阻塞队列中去除请求并分配给 Elevator_system 进行调度，如果当前请求无法分配则先加入一个等待队列。



d) 本次作业的设计我认为优点在于很好地发掘出了请求的分配机制。我恰好采用了和老师后来所讲一致的设计思路，包括逐层分配请求，使用阻塞队列，建立 state 数组，建立分电梯的请求队列等等。但是在继承机制和子类父类的交互

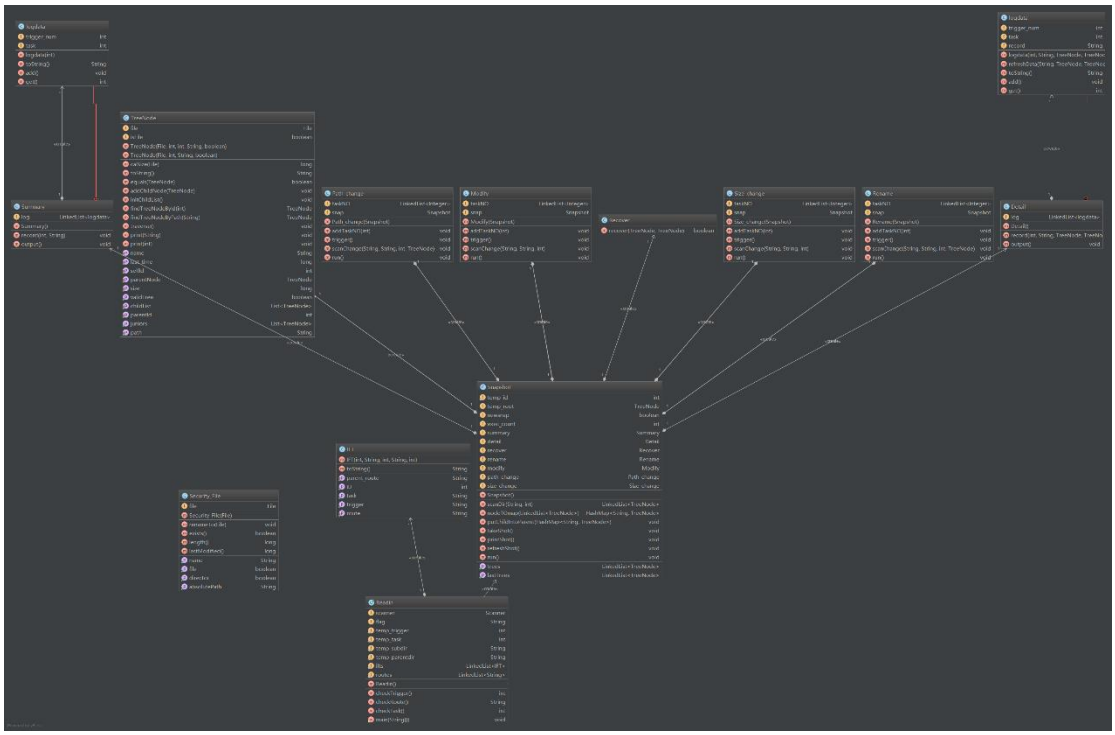
使用上我做的不是很好，我在代码中保留了前两次的调度器，逐层继承，但实际上只是为了继承而继承，真正复用了的方法只有两三个。这一点需要在后期优化结构中做进一步调整，同时先期顶层设计也应考虑范围更加广阔的继承代码。

2. 第六次作业

a)

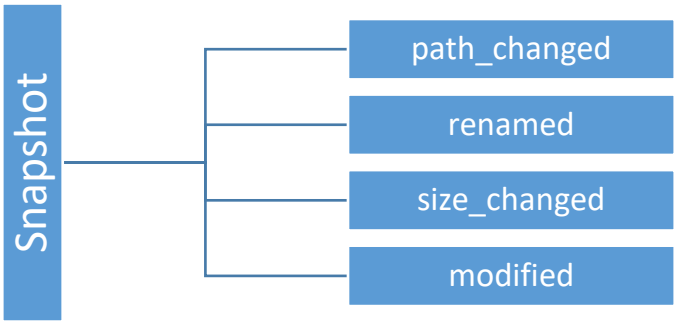
类名	属性个数	方法个数	分支个数	总行数
Readin	8	5	4	215
Detail	1	3	5	99
IFT	2	5	3	50
Modify	2	5	2	115
Path_change	2	5	4	134
Recover	0	1	3	49
Rename	2	5	2	135
Security_File	5	5	0	45
Size_change	2	5	2	120
Snapshot	13	8	6	190
Summary	1	3	1	78
TreeNode	12	12	4	247

b) 通过类图来分析类之间的关系



c) 通过线程协作示意图来分析线程关系

在控制台读取监控路径完成后，启动 Snapshot 线程，然后启动 snapshot 中的四个触发器线程。每隔一定的时间段，Snapshot 就会更新然后唤醒各个触发器执行相应的操作。



d) 自我点评设计的优点和缺点：第六次作业我的程序采用多叉树存储文件快照，一个统一的 SnapShot 类按照时间间隔获取快照，并依次分发给不同触发器进行操作。总的来看，我遵循了一个触发器一个线程的做法，同时采用父子协作的方法来加锁，对快照和触发器之间进行同步。现在看来，我应该采用读写者模式，将快照类设置为写者即可，这将大大简化设计。最大的缺点就是一开始考虑了过多的实现细节导致作业复杂度成幂级增长，最终留下了很多问题。同时判断两次快照有何变化的逻辑分支做的不是很好，有逻辑冗余的情况。

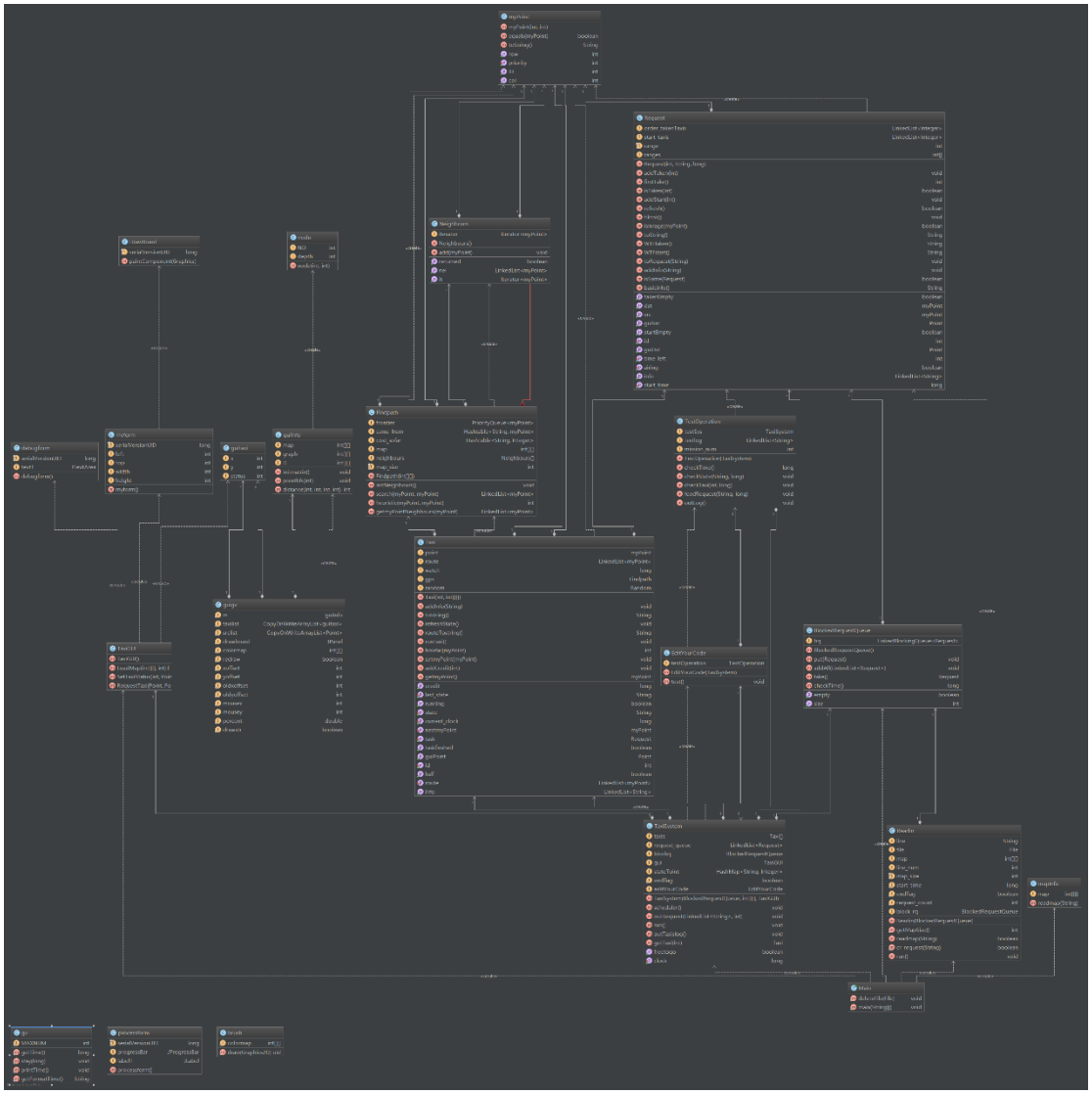
3. 第七次作业

a) （不算 GUI）

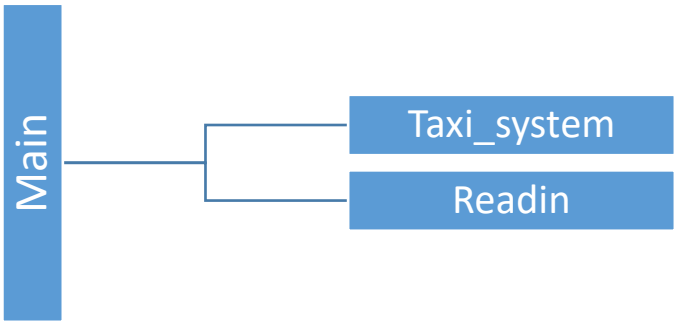
类名	属性个数	方法个数	分支个数	总行数
EditYourCode	1	1	0	23
BlockedRequestQueue	1	5	0	45
Findpath	6	5	2	150
Main	1	2	5	109
myPoint	3	1	0	42
TestOperation	3	6	6	96
TaxiSystem	8	6	18	259
Readin	9	5	12	202
Request	11	15	4	193
Taxi	13	10	13	266

b) 通过类图来分析类之间的关系

本次作业的线程设计较为简单，属于最基本的生产者消费者模型，读取输入然后从阻塞队列中分配给出租车调度系统。该调度系统模拟时钟运行，将输入的请求按照 100ms 时间片分配到每一个周期中进行处理。



c) 通过线程协作示意图来分析线程关系



d) 自我点评设计的优点和缺点：本次出租车在经历了两次多线程作业之后已经好了许多，不仅能够有机会考虑并实现更多指导书中的细节问题，也有时间优化逻辑上的设计。我将出租车设置为了一个数组作为共享变量访问，同时每次遍历数组刷新出租车状态。对于 SOLID 检查来说，除了继承尚未有可以验证的效果之外，其他四条均作了分析符合标准。缺点在于出现了一个 crash，一方面来说，稳妥起见，程序依旧需要 try-catch 语句来保证不异常退出；另一方面来说，我的测试还不够完备，逻辑推导阶段出现了问题——当前分配的出租车编号和数组索引+1 的关系没有正确处理。在错误的时刻-1 导致出现了-1 的数组

越界访问。

三， 作业程序 bug 分析

1. 第五次作业
   a) 第五次作业最主要的问题就在于随着运行，时间输出总会和真实时间相差 0.1-0.2s。对于这个 bug 主要出现在时钟同步上。因为我的 ele_system 里内置了一个全局模拟时钟，每次执行一轮请求之后睡眠固定的时间段，但是执行请求所花的计算时间理论上是瞬间发生，或者说应该在程序 100ms 的时间片粒度下可以忽略。但是由于输出请求的时候请求时间来自于真实时钟，导致模拟时钟和真实之间差距逐渐增大到不可忽略的地步。
2. 第六次作业
   a) 第六次作业主要问题在于线程加锁没有设计好，导致程序经常性崩溃；同时快照的变化检测需要精进，每次遍历找到父节点和它下面的子节点，效率地下，降低了程序并发性。
   b) 其中，经常奔溃就是由于多线程中 thread 里面 rename 和 path-change 两个同时访问了快照并在进行 recover 之类的操作时，没有对快照加锁。
3. 第七次作业
   a) 第七次作业只有一个 bug。在同一时间点发出的几个互相抢单范围交集很大的请求，如果有一辆车被选中了，程序就会出现数组越界情况。这个 bug 发生在 Taxi_system 的 108 行左右，是一个使用 allocID-1 作为数组索引的，一旦该请求别分配，就会出现数组越界的情况并导致一个 bug。

四， 互测作业 bug 分析

1. 一般性 bug 的构造和测试
   a) 首先第六次作业的时候由于个人原因没有仔细看指导书，也因此错过了很多评价机会。因此第一个构造 bug 的建议就是，认真阅读指导书并检查指导书中的设计细节是否实现，完成度如何。
   b) 在认真阅读指导书的基础上，查看指导书允许范围内的极端情况（合法输入），检查程序健壮性
   c) 检查健壮性之后根据个人写程序过程中的重点逻辑实现部分进行测试，包含有争议的逻辑处理
2. 线程安全类 bug 的构造和测试：首先我们需要明确，线程不安全发生的原因目前主要是由于数据竞争引发，比如读写同时发生，写写同时发生，触发器信号存在竞争性读写等等。故而我们直接去撞大运地测试肯定是不可取的。
   a) 首先找到测试代码中所有需要同步的地方（可以尝试直接搜索 Sychronized 或者 lock）
   b) 对于竞争某些数据的代码段进行分析，并逐条针对可疑的资源对象设计测试样例
   c) 设计测试样例并不一定能触发线程安全问题，我们是为了通过调试，比较直观地看出临界区的访问情况，并对安全性问题做出定性判断。

五， 心得体会

首先在设计阶段我们需要确定将使用多少个线程。而线程的使用数量则进一步决定了我们接下来的程序设计逻辑，包括线程间通信，抽象的数据处理，进一步产生对临界资源的访问安全性问题。

我们可以回忆在 OS 上学习的，线程是“CPU 调度的最小单位”，引入线程的目的是

“提高进程内部并发程度，共享资源”，那么从这两点考虑，我们就需要对指导书进行划分，*哪些是**资源**，哪些是**计算任务**，资源交给进程管理*（可以理解为主线程），*计算任务交给一个个小的线程并发执行*（尽量减小锁区域），多个线程计算的时候可以共享主线程的资源。这样我们就得到了一个基本的解决问题的多线程模型。

进一步地，我们考虑性能和资源的平衡问题。我们需要进一步划分计算任务并抽象成合理的线程数量来进行计算。比如，第六次作业的文件快照存储，如果不使用多叉树，则触发器需要大量分支来处理本应管理好的数据；或者采用一个监控对象一个线程，这样的话线程之间通信成为了问题而且线程数量过多，调度开销太大。因此我们需要合理划分计算任务和调度资源获取，尽量将资源获取和更新抽象为线程的输入输出，计算任务划分为可以并行的线程。

接下来我们来讨论线程安全问题。首先，在通过使用上文提到的策略初步划分资源之后，我们对于将会有多个线程访问的资源引入进一步的调度。一般来说我们分析问题情景，尽量套用 java 已有的类库中的线程安全类的实现，以减少自己设计锁的正确性风险。

总体来看，这三次作业是对多线程应用的一步步加深，而且代码重用的重要性也在不断提高。由此可见，我们接下来需要继续在代码复用的基础上设计出线程安全的可移植性强的模块，这样才能尽可能缓解最后两次出租车日益增长的需求和大家越来越紧凑的时间之间的冲突。（滑稽。。。