《面向对象设计与构造》Lec08-第二单元总结分析

OO2022课程组

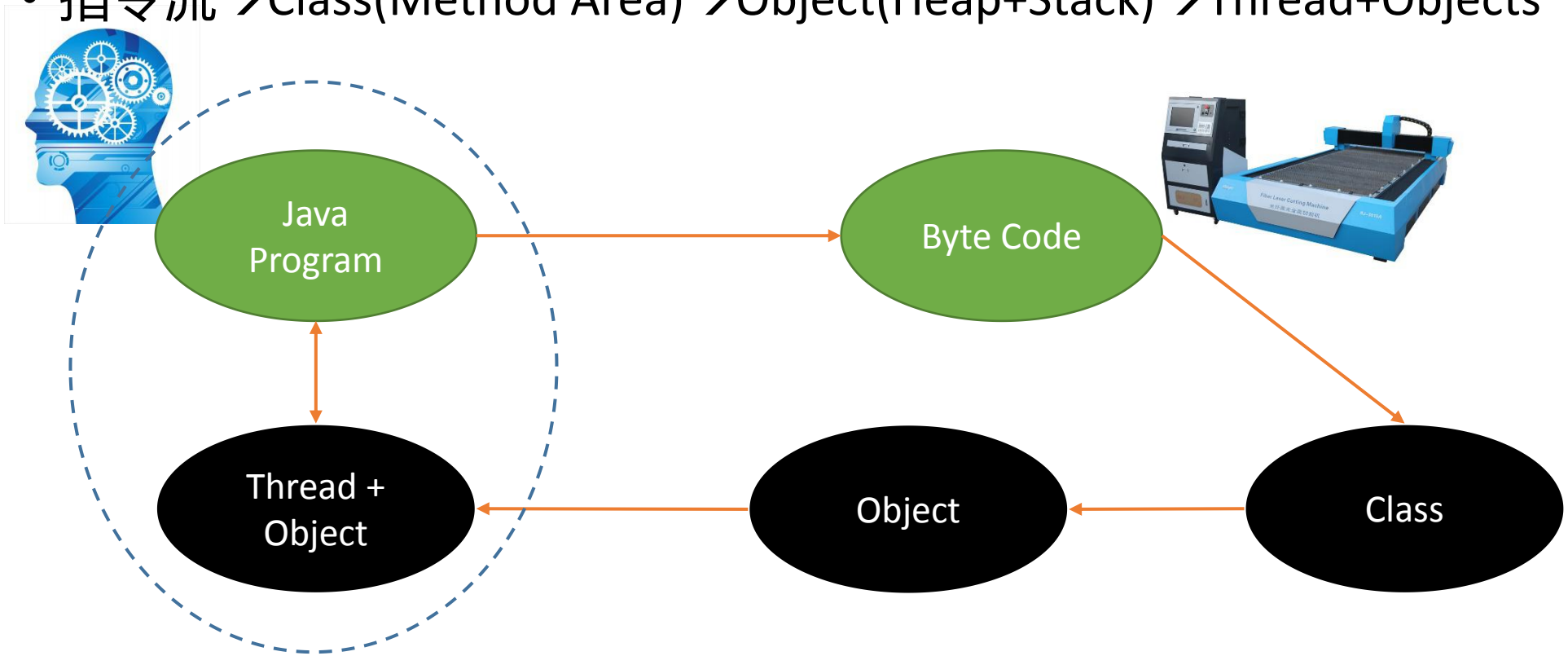
北京航空航天大学计算机学院

单元知识点回顾

- 对象：逻辑设计+运行时状态
- 逻辑设计
  - 类定义其规格：方法、属性
  - 程序通过对象引用来访问
  - 可以使用不同类型的对象引用来访问
  - 对象创建类型不会变化
- 运行时状态
  - 对象存储在堆区
  - 对象引用类型与对象创建类型的关系决定了能访问哪些属性和方法
  - 对象状态只有一份！

单元知识点回顾

- 介绍JVM的基本机理是为了学习和理解对象的运行时特性
  - Java程序→class文件：byte[]
  - 指令流→Class(Method Area)→Object(Heap+Stack)→Thread+Objects



单元知识点回顾

- 线程是在静态层面刻画对象运行时特性的机制
  - Thread.start是个异步方法
  - Thread.run不让我调用(?)
  - Thread似乎失控了：你无法在代码逻辑中确定其状态，并对其进行确定性的控制
- Thread的双重职责
  - 业务职责：根据业务功能管理所需对象及其访问
  - 业务职责：根据系统设计要求，与其他线程交互（等待/唤醒/...）
  - 系统职责：JVM进行调度控制的抓手

单元知识点回顾

- 线程在执行控制上具有独立性
  - 不具有互相调用关系
  - Q：如果我在一个线程方法中调用了另一个线程对象的方法会发生什么？
- 线程之间的关系
  - 父子关系：创建和启动线程
  - 协作关系：生产者-消费者
    - 同步关系：通过共享对象，或者主动状态控制进行同步(wait,notify)
- 访问共享对象有风险
  - 稍有不慎，程序状态失控
  - 措施
    - 不共享
    - 不得不共享，同步控制

单元知识点回顾

- 生产者与消费者协作的多种形态
  - 1个生成者---1个消费者
  - 1个生产者---多个消费者
  - 多个生产者---1个消费者
  - 多个生产者---多个消费者
  - 单一产品的生产与消费
  - 多种产品的生产与消费
- 一种多线程协同的设计模式
  - Producer
  - Consumer
  - Queue<something>

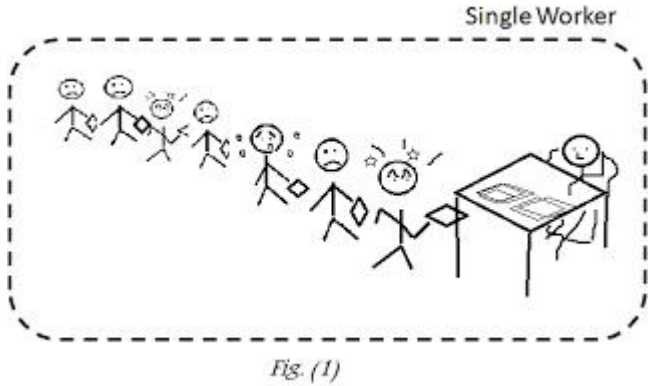


Fig. (1)

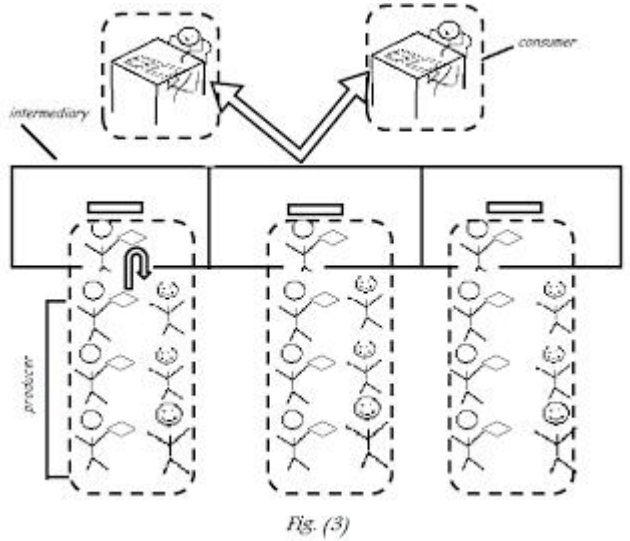


Fig. (3)

http://www.javacodegeeks.com/2012/02/concurrency-pattern-producer-and.html

单元知识点回顾

- 同步块是”房间”，monitor是锁
  - 每个对象有且只有一把物理锁，可守护多个同步块
  - 逻辑相关是“房间”和锁配合好的基本设计策略
- 可以使用JVM内置提供的物理锁，也可以自行构造逻辑锁
  - 逻辑锁可实现更加灵活的进出房间机制
- 如何设置<同步块，锁>并不是一个琐碎问题
  - 线程体代码随意设置同步块
  - 方法按需设置同步块
  - **只在共享对象中设置同步块**
  - **仅使用同步块中的共享对象来守护该同步块**

单元知识点回顾

- 线程安全
  - “线程**，你不用担心什么，只管做你的事情，保证不给你添乱…”
- 如何导致线程不安全
  - 读写冲突
    - Check-then-act
    - Read-modify-write
  - 写写冲突
- 线程安全类的特征
  - 自己管理好“房间”，不烦“房客”
    - 任意时候都假设会有多个“房客”(线程)等待进入使用“房间”
  - 技术手段
    - 细分“房间”功能：单一职责，提高房间使用效率，减少等待时间
    - 针对“房间”配备相应的锁

单元知识点回顾

- 我什么时候知道该用多线程？
  - 分析/设计时识别出了并发行为模式
- 锁机制的选择
  - 物理锁，严格，JVM内置，好理解
  - 逻辑锁，灵活，可扩展出自己想要的更多特性
- 继承、接口、多态形成了面向**数据和行为**抽象的层次设计结构
  - 关注数据抽象和行为抽象
- 线程、共享、交互形成了面向**并发和协同**抽象的层次设计结构
  - 关注并发行为的安全和效率

单元知识点回顾

- 线程交互结构
  - 线程交互需借助于共享对象，数据流交互
  - 三种交互结构：主从、流水线、事件驱动
  - 要围绕线程的业务目标和业务交互关系来选择交互结构
- 线程管理
  - 线程创建与销毁消耗资源
  - 重用相关对象，构建线程池
  - 还是要服从于业务设计
    - 工人有上下班
    - 工人可以转岗
    - 工人有离职和新加入

单元知识点回顾

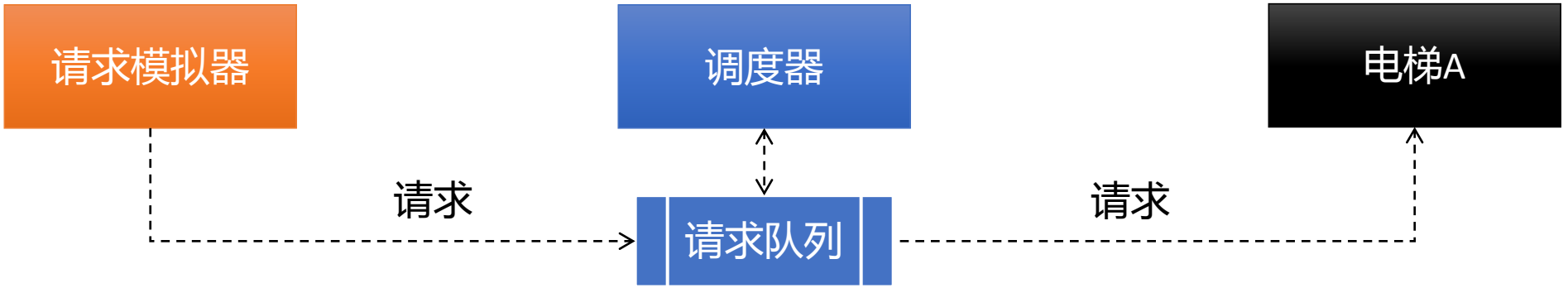
- 保持架构的扩展能力是不变的追求
  - 仍然要强调层次化设计
  - 稳定住大结构：电梯、队列、调度器
  - 把易变的按照层次来展开和逐步细化
    - 楼层的特殊性（是否停靠）
    - 电梯的特殊性（横向电梯、纵向电梯、运动速度、载客量）
    - 调度的特殊性（换乘、捎带、队列平衡、等待时间平衡）
- 细节仍然是决定成败的关键
  - 线程安全
  - 数据可配置性
  - 控制逻辑的简洁性

作业5训练要点分析

- 从控制流程来看待电梯的运行过程
  - 请求模拟器通过控制台获得用户请求
  - 调度器调度电梯来响应请求
  - 电梯根据调度所分配请求来提供服务
- 在单线程场景下
  - 调度器→请求模拟器，同步调用关系，获得请求对象
  - 调度器→电梯，同步调用关系，分配请求
- 在多线程场景下
  - 几个线程？
  - 交互关系？请求模拟器不断的发送请求

作业5训练要点分析

- 两个线程方案
  - 请求模拟器、电梯是线程
- 三个线程方案
  - 请求模拟器、调度器、电梯是线程
- 请求队列交给谁来管理维护
  - 必然是共享对象
  - 管理：新请求如何放入、分配哪个/哪些请求、…



作业5训练要点分析

- 方案A：由请求模拟器线程来管理请求队列
  - 模拟器**不断**把获得的新请求插入队列
  - 调度器**不断**从队列提取请求，分配给电梯
    - 方案1：等待电梯执行完毕后，再取下一个并分配
    - 方案2：不等待电梯执行完毕，只要有新请求就分配给电梯
  - 电梯**不断**响应分配过来的请求
  - 问题分析
    - 调度器是否需要一个局部队列，否则**调度**如何体现？
    - 不论是否等待电梯执行完毕，都需要了解**电梯状态**，那么关心哪些状态？
      - 如果调度器不等待电梯执行完毕，电梯是否需要一个局部队列？

作业5训练要点分析

- 方案B：由调度器来管理请求队列
  - 模拟器**不断**将请求推送到请求队列
  - 调度器**不断**扫描队列，按照请求关系来**重排序**
  - 电梯**不断**从该队列获得请求
    - 方案1：执行完毕再去获得下一个请求
    - 方案2：先存放到内部请求队列，然后选择请求来响应
  - 问题分析
    - 调度器调整请求顺序的依据是什么？
    - 电梯内部请求队列如何管理？

作业5训练要点分析

- 方案C：还是让调度器管理请求队列
  - 模拟器**不断**将请求推送到请求队列
  - 调度器维护两个队列
    - **全局队列**，接收模拟器发来的请求
    - **电梯局部队列**，存放分配给电梯的请求
    - 调度器是个搬运工
  - 电梯**不断**从局部队列来获得请求执行
  - 问题分析
    - 电梯在执行一个请求的过程中是否能够捎带其他请求？如何做？
- 方案D：让电梯来管理请求队列
  - 需要讨论吗？

作业5训练要点分析

- 本次作业的重点在于从业务场景入手分析需要哪些线程，以及线程如何协作
  - 生产者/消费者协作模式是入门必备
- 调度器的设计存在诸多灵活性
  - 作为共享对象来管理请求队列
  - 作为线程对象来管理请求队列
- 如何捎带处理多个请求，提高性能
  - 在哪里进行捎带处理？
  - 捎带调度需要考虑电梯状态和请求队列状态

作业5训练要点分析

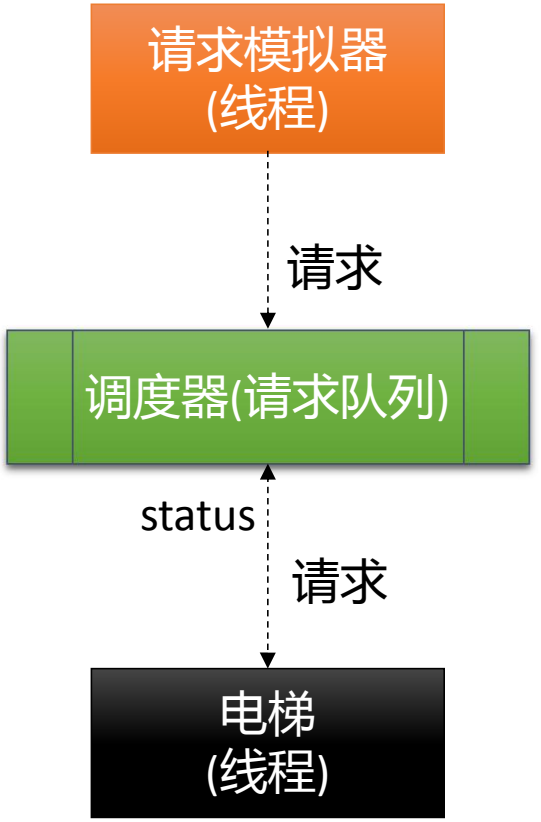
- 如何支持捎带？
  - 方案A：调度器作为**共享对象**
    - 设置status board，电梯把自己的状态发布到board，调度器从中读取电梯状态
  - 电梯何时更新board？
    - 取请求时更新board
    - **随时**更新board(调度器提供一个新的接口)
  - 调度器的请求提取接口
    - 方案1：一次性取出所有可能的可捎带请求
      - 把当前同一个方向的请求都取走
    - 方案2：先取走一个，然后更新状态时取可捎带请求
      - fetch(提取主请求), fetchPass(提取捎带请求)

作业5训练要点分析

- 如何支持捎带？
  - 方案B：调度器作为**线程**，无需了解电梯状态。按照一般性策略把请求按照时间序转交给电梯，由电梯自身进行捎带调度
    - 电梯无需对外公布其状态变化
    - 电梯随时根据状态变化扫描local queue来选取其中的可捎带请求
  - 思考1：捎带策略由调度器执行或电梯执行带来什么差异？
  - 思考2：如果将来不同电梯有不同的捎带策略呢？
- 捎带调度的动态性要求
  - 运行过程中不断出现新请求，调度器和电梯都无法提前确定可捎带请求
  - 考虑实际工程扩展：中心调度器关注请求到电梯的分配（如运动量或负载均衡），电梯局部调度器关注捎带处理（请求响应效率）

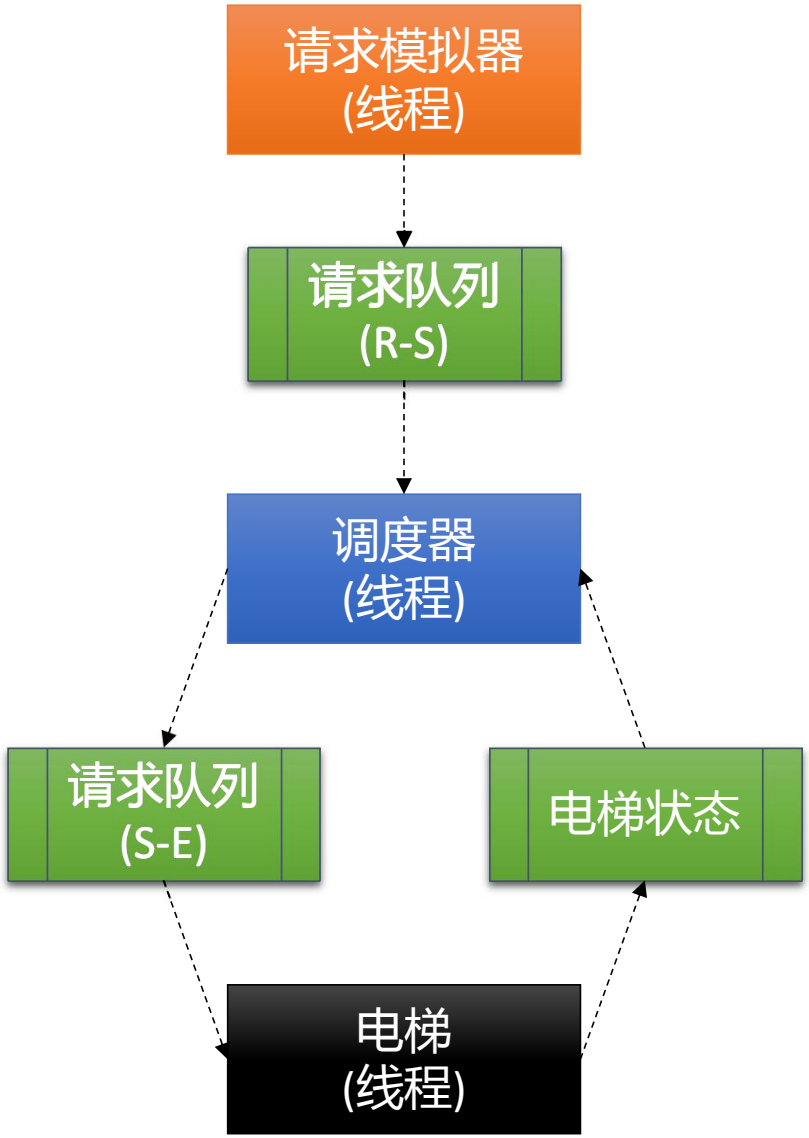
作业5训练要点分析

- 调度器作为共享对象的设计架构
  - 调度器提供请求推入和请求取出接口
  - fetch接口：按照简单策略来决定电梯取走哪个请求
    - 取走请求队列头元素，不考虑电梯状态
  - fetchPass接口：按照电梯状态变化准备好所有可捎带的请求
    - 电梯状态是接口输入参数
    - 按照电梯状态扫描请求队列
    - 如果fetchPass未发现可捎带请求，电梯则继续当前请求的执行，直至结束再调用fetch接口
  - 调度体现于请求取出接口调用所获得的请求序列
    - 不考虑电梯状态的队列头调度
    - 考虑电梯状态的捎带调度



作业5训练要点分析

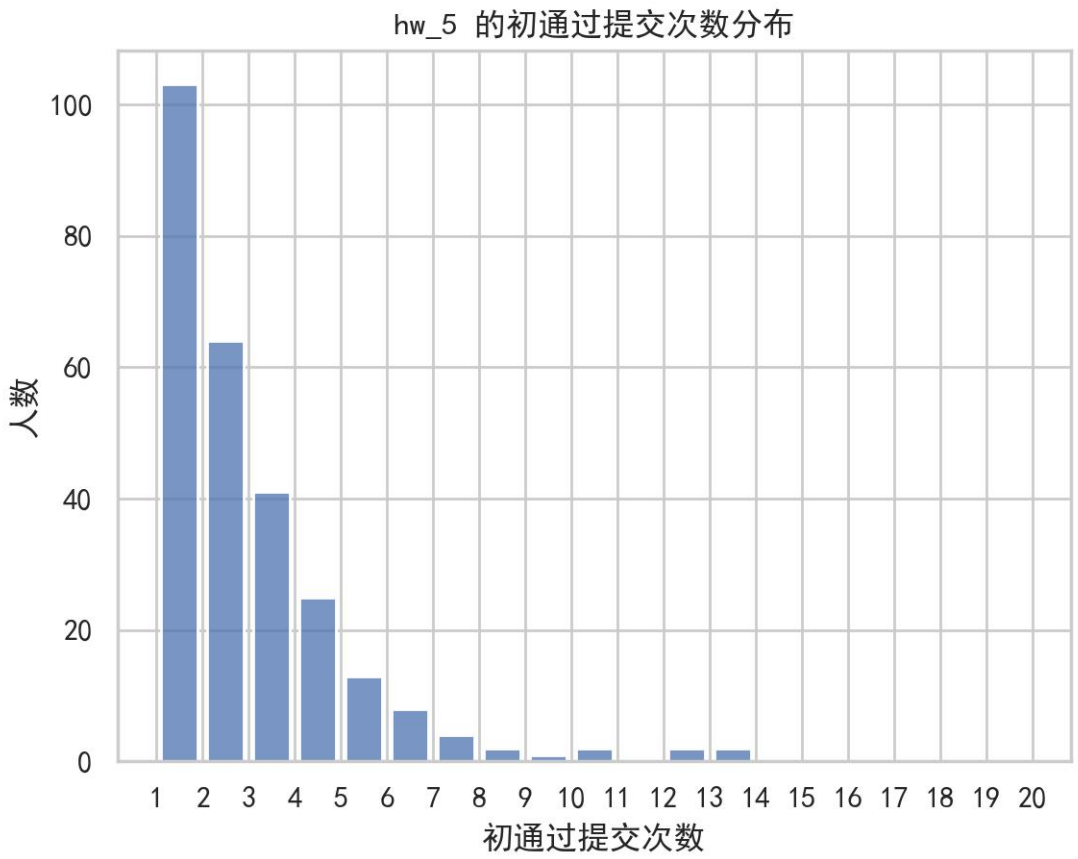
- 调度器作为线程的设计架构
  - 现实中的调度器运行于一个控制器(嵌入式系统)中，持续运行
- 调度器与请求模拟器、电梯的交互都基于共享对象
  - 调度器把R-S队列中的请求进行重排序，并放入到S-E队列
  - 根据电梯状态的变化，从S-E队列中挑选可捎带的请求
- 三个线程类之间不再有控制耦合关系

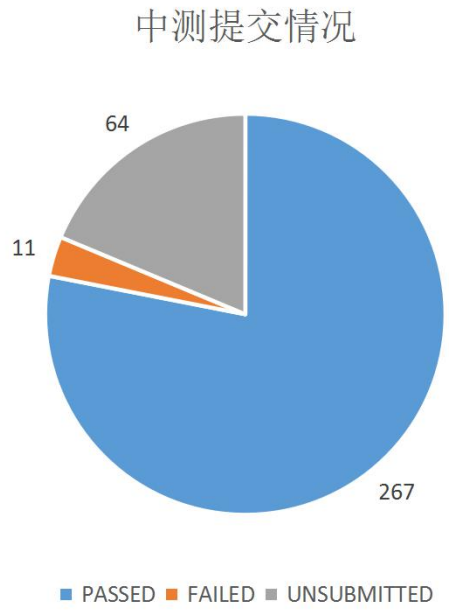


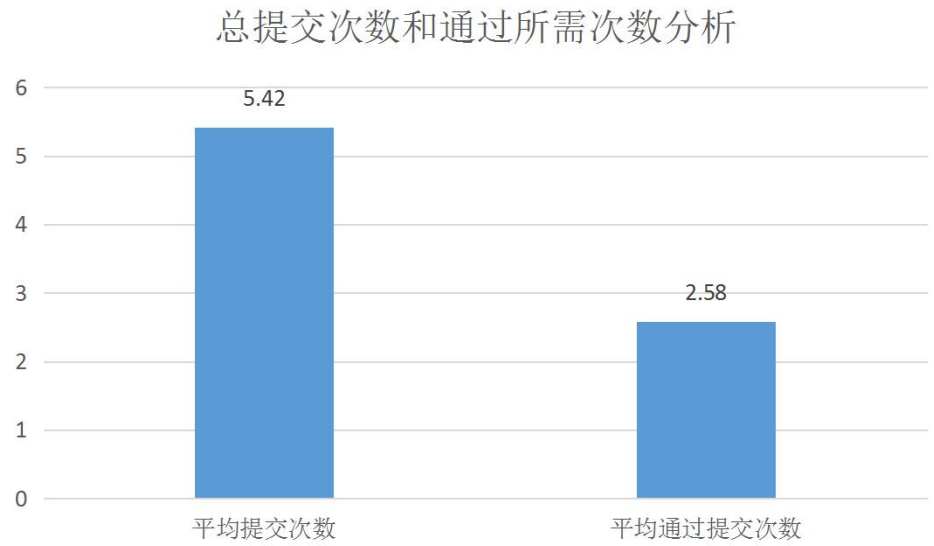
作业5的主要设计问题

- 没有或错误的线程安全控制
  - 何处加锁
  - 使用什么锁
  - sleep, wait, notify, notifyAll的使用
- 在线程类代码中直接进行同步访问控制
- 在线程类的run方法中展开细节
  - run方法应保持简洁，只描述顶层处理流程

作业5的中测情况分析

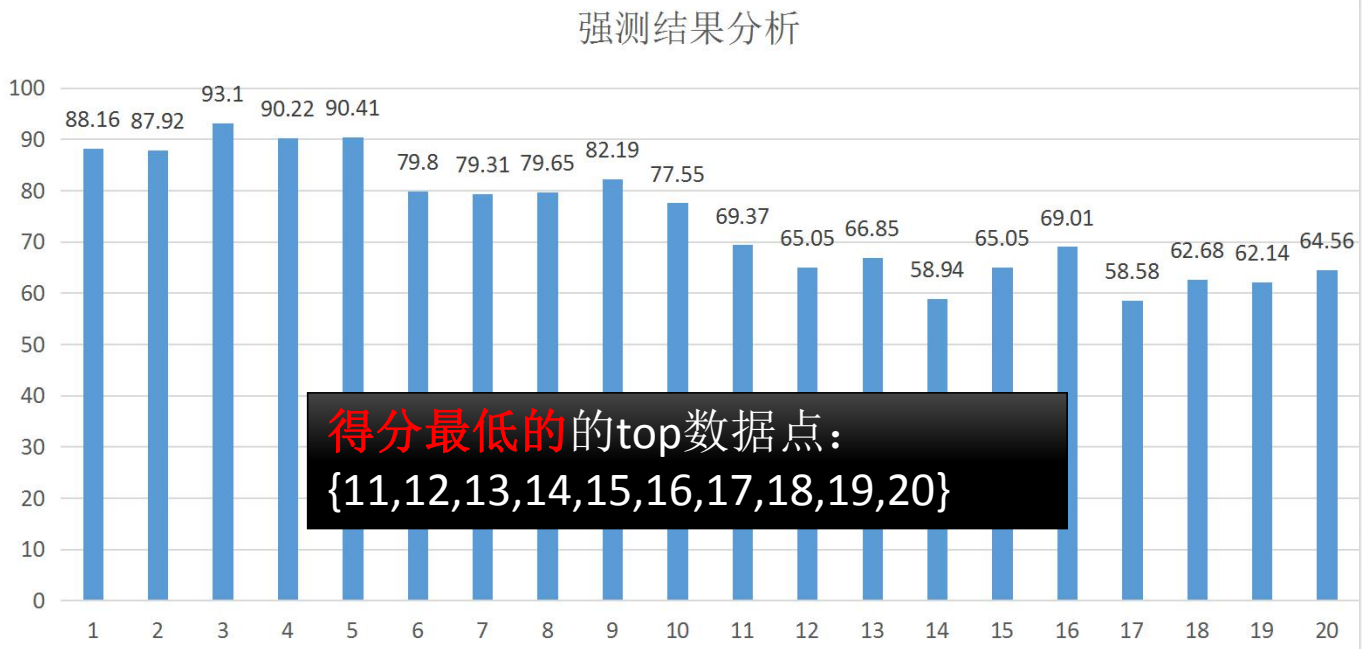

hw_5 的初通过提交次数分布
人数
100
80
60
40
20
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
初通过提交次数


中测提交情况
64
11
267
PASSED
FAILED
UNSUBMITTED

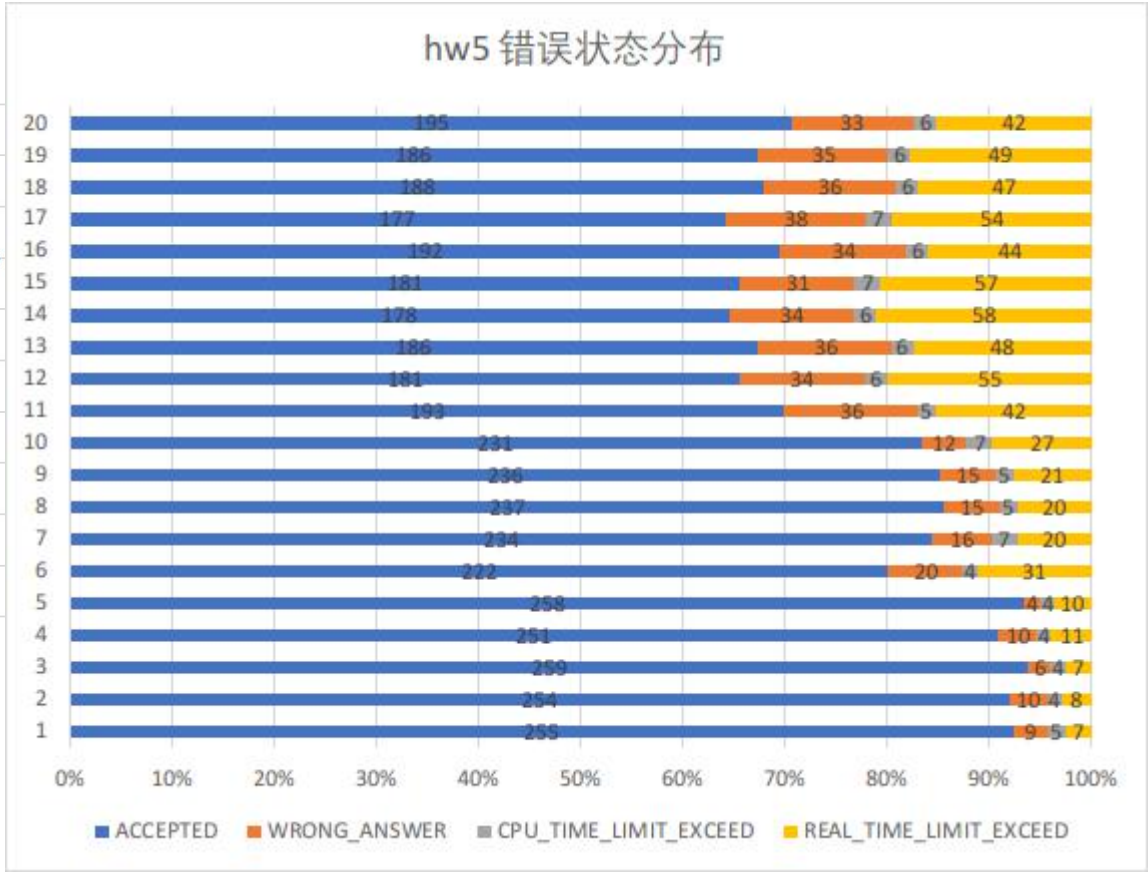

总提交次数和通过所需次数分析
6
5
4
3
2
1
0
5.42
2.58
平均提交次数
平均通过提交次数

作业5的强测情况分析

发现**线程安全问题**的top数据点：
{11,12,13,14,15,16,17,18,19,20}



得分最低的的top数据点：
{11,12,13,14,15,16,17,18,19,20}



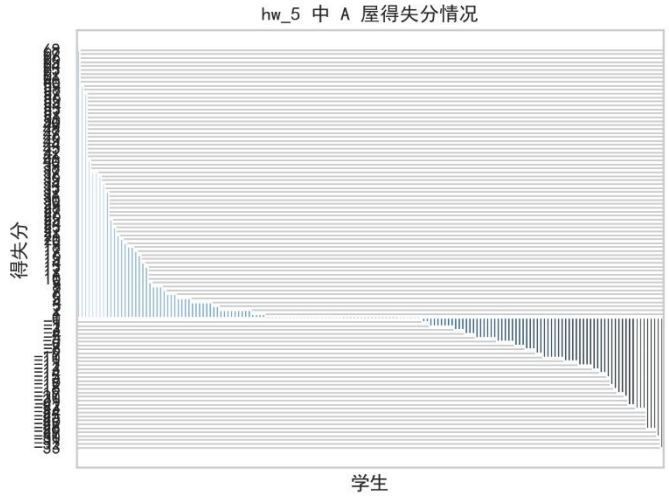
HW 5 强测数据得分较低的点特征：

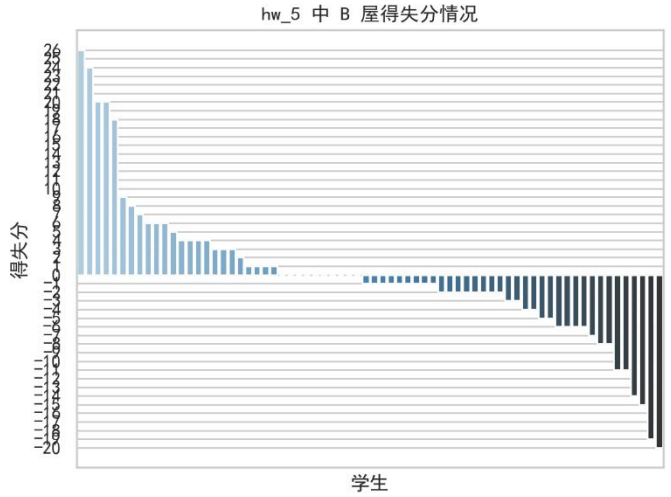
- 数据随机生成
- 指令数多，第一个乘客到达时间与最后一个乘客结束时间靠后，需要高效规划
- 跨度较大：纵向请求的间隔 >= 4
- 所有请求集中在同一楼座或同一楼层

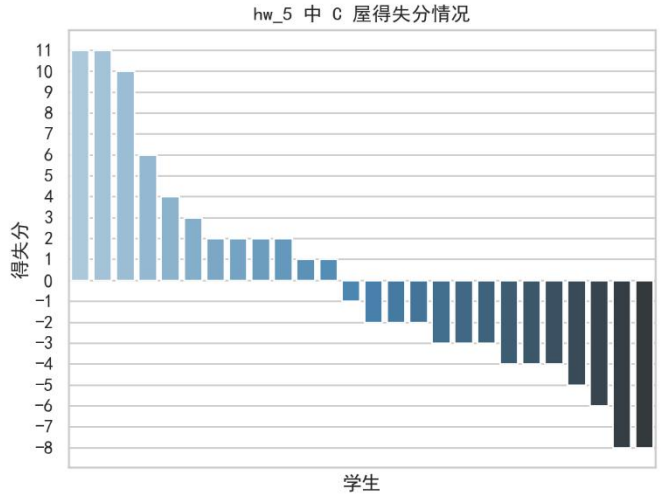
WA: 接送乘客出现问题
RTLE：线程安全问题
CTLE：线程安全或轮询
RE：线程安全

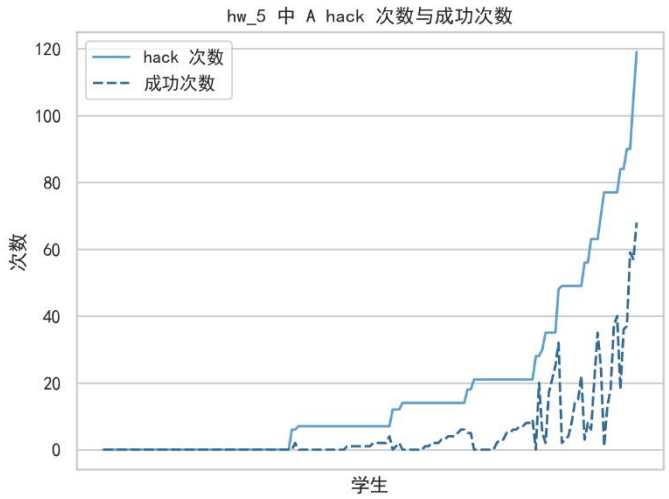
作业5的互测情况分析

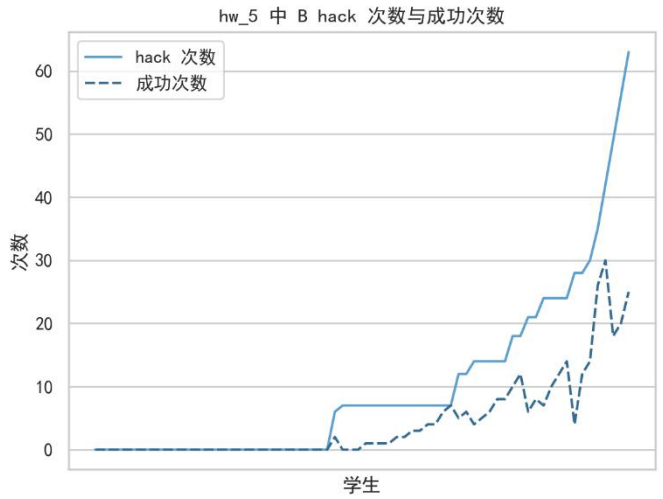
A级平均**17.67**次hack，成功率28.20%
B级平均**10.14**次hack，成功率43.24%
C级平均**5.54**次hack，成功率60.42%

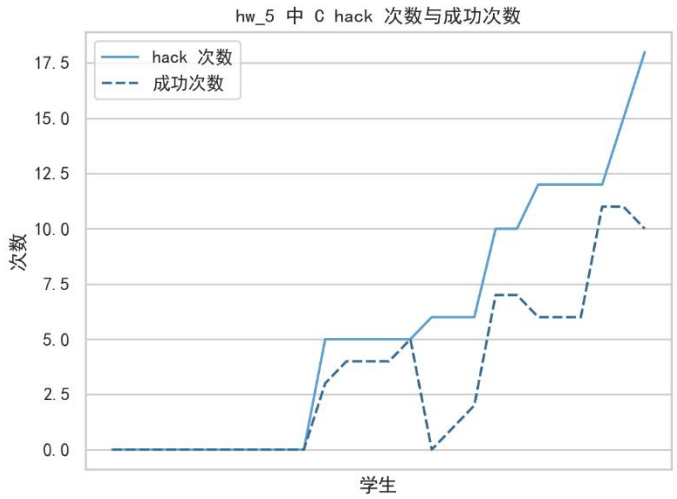












作业6训练要点分析

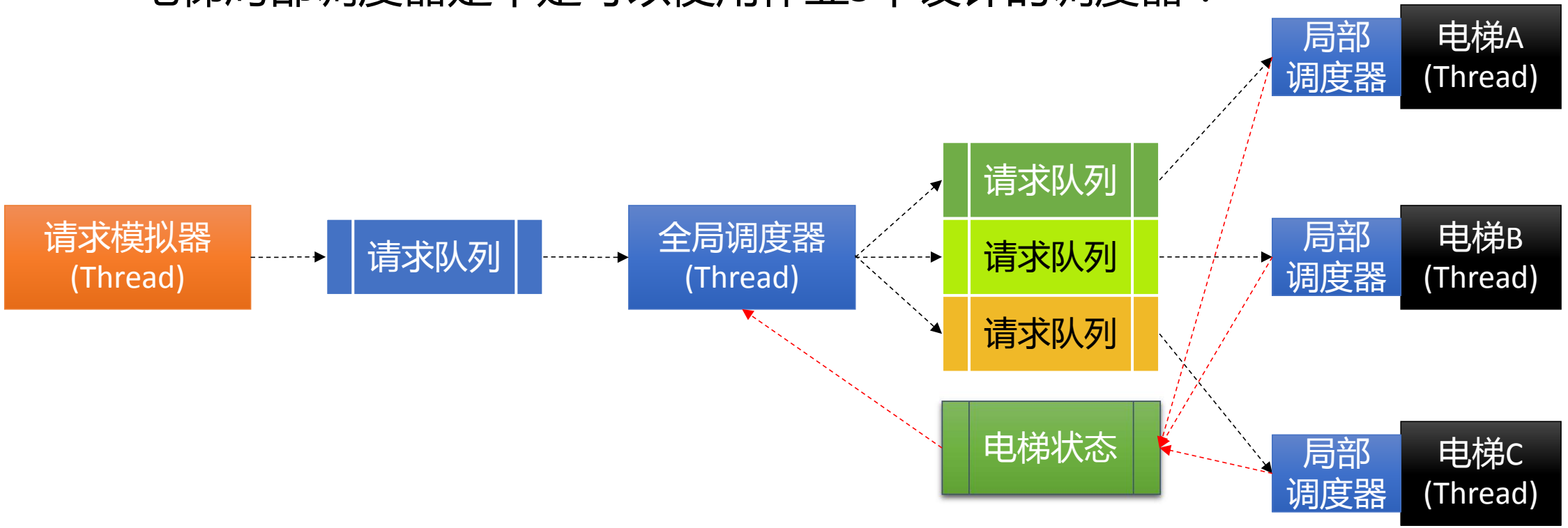
- 沿用第5次作业的架构设计
  - 调度器作为共享对象
  - 调度器作为线程
- 引入多部纵向电梯和横向电梯，调度需要分层
  - 顶层调度器负责在电梯间分配请求（电梯有载人容量限制）
  - 电梯局部调度器处理单个电梯的请求调度

作业6训练要点分析

- 调度的硬性约束
  - 任何时刻电梯不能出现超载、不能去不存在的楼层
  - 方案A：让顶层调度器进行处理
  - 方案B：让每个电梯进行处理
  - 需要跟踪的电梯状态：运动、方向、乘客人数
- 调度的优化约束
  - 整体运行时间尽可能短
  - 方案A：平衡电梯队列中的请求数
  - 方案B：平衡电梯队列中请求的估计响应时间ERT
    - 假设在没有访问冲突的情况下，完成所有请求的估计时间

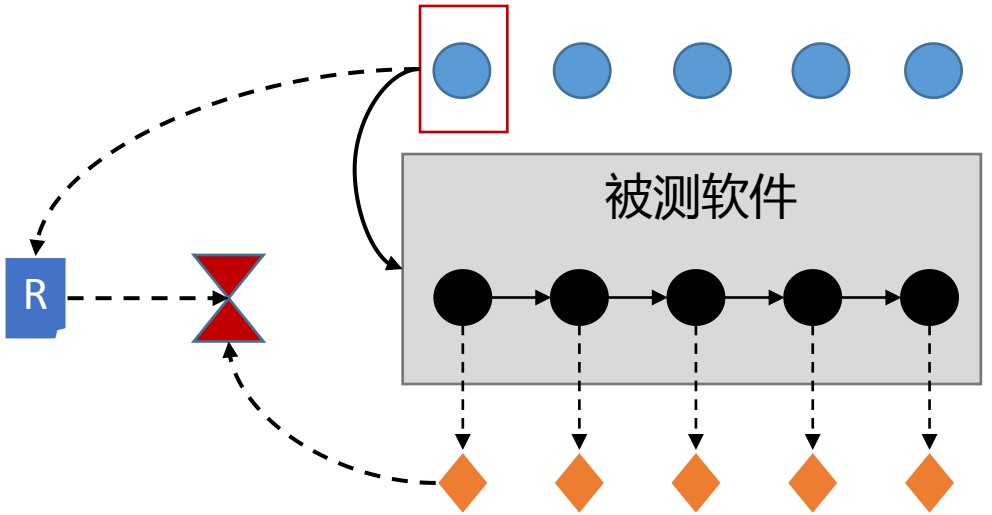
作业6训练要点分析

- 线程之间形成链状生产者-消费者关系
  - 电梯局部调度器是不是可以使用作业5中设计的调度器？



作业6训练要点分析

- 了解并掌握另一种测试形态
  - 第一单元作业测试：不关心求解过程的表达式状态变化
  - 第二单元作业测试：关心电梯和队列的状态变化
  - 如何针对电梯运行状态和队列状态来设计请求序列是重点，也是难点
- 线程测试
  - 不提供直接的调用交互机制
  - 通过共享对象：设置共享对象的状态进行测试



你是如何判断电梯系统的功能正确性？

作业6训练要点分析

- 基于锁的线程同步设计
  - 为什么锁会发生效果？
    - (1)多个线程会执行到相同的代码区域
    - (2)多个线程使用相同对象的锁(即相同的锁)
    - (3)JVM确保在任何时候只能有一个线程获得进入受控区域的锁
    - (4)线程可以主动放弃锁
    - (5)或者执行结束自动释放锁
  - 锁加在何处？
    - 线程类方法
    - 共享对象
  - 使用哪个锁？
    - 与受控区域计算相关的锁
    - 与受控区域计算无关的锁

再谈锁

- 假设有一种银行，里面有10个窗口，每个窗口都能办理存款，取款，房贷，外币兑换等各项业务
- 假设目前有100个这样的银行正在开门服务，有一个同学小明，他的4个家长都在银行同时给他存钱，其中两个家长在银行A，第三个家长在银行B，第四个家长在银行C

```
Class BankBranch
{
    private Cashier1;
    private Cashier2;
    …
    private Cashier10;

    public void Deposit(Account a, float amount){}
    public void Withdraw(Account a, float amount){}
    public void Mortgage(Account a){}
    public void ExchangeCurrency(Account a, float amount){}
}

Class Account
{
    String Name;
    String NationalID;
    float balance ;
    …
}
```

再谈锁

- 从办理业务的角度，用户排队，叫到号以后分配一个窗口Cashier
- 每个用户都可以看成一个线程，要竞争窗口这个资源（每个窗口都可以办理相应的业务）
- 使用synchronized void Deposit() 来解决同步控制问题
- 所有用户都在银行外面排队，当一个用户被分配到了某个窗口（比如5号窗口），用户进去以后就把银行门给锁上了，5号窗口自然不会有竞争发生了，但是其他窗口也不能服务用户了
- 办理业务的竞争资源应该细化到Cashier窗口
- Deposit方法使用同步块：synchronized(someCashier)

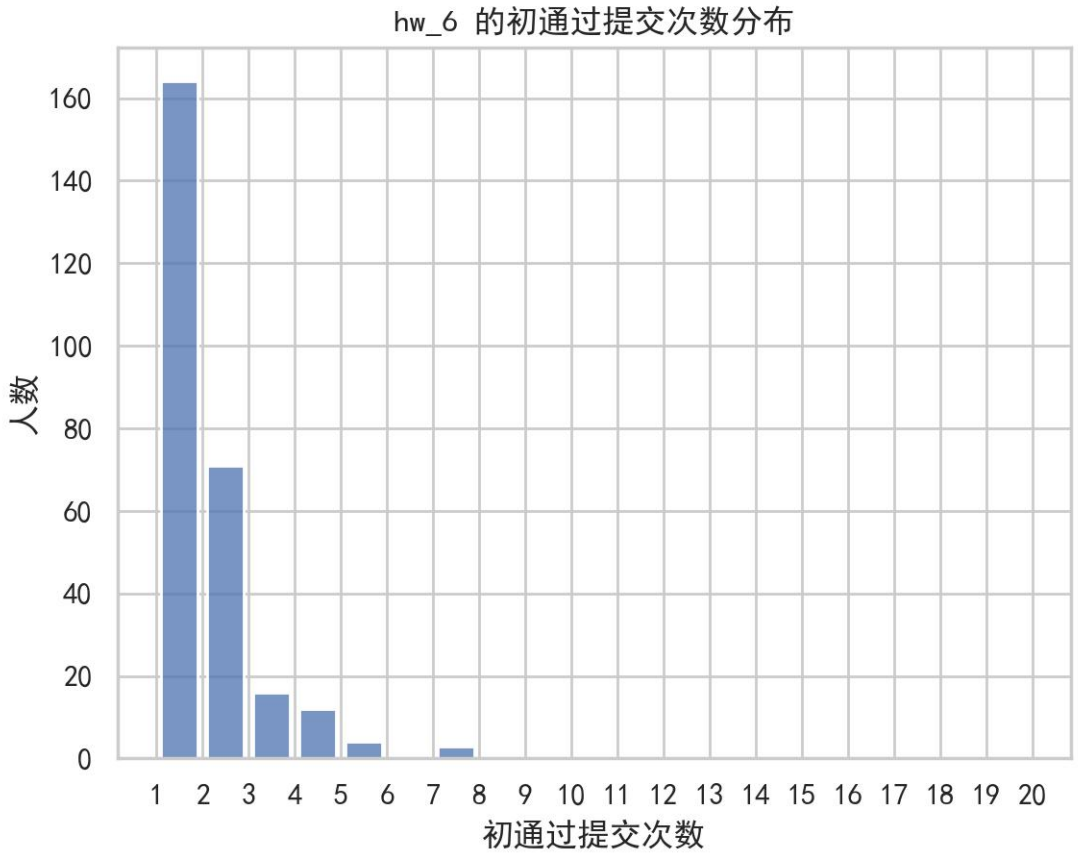
再谈锁

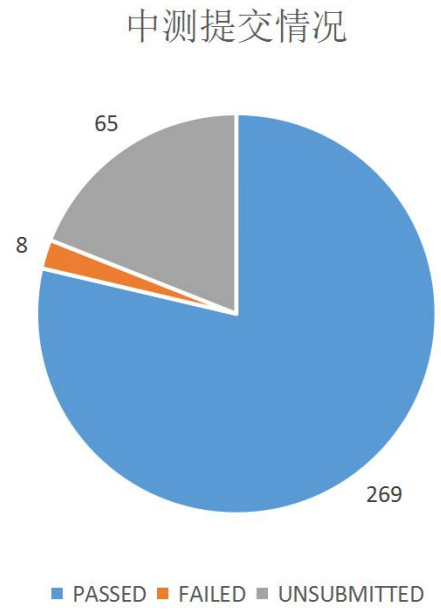
- 给小明办理存款，用户提供卡号和现金，银行把现金存入相应账户
  - 4个家长都有卡号，同时在**3个银行的4个窗口**给小明存钱
  - 4个家长可以看成4个线程，要竞争小明的账号这个资源
  - 小明很幸福
- Deposit使用cashier锁进行同步控制
- 在银行A的两个家长分别在不同的窗口存款，银行B和C的两位家长也分别通过相应的窗口在存款，假设同时都在给小明的银行账户存钱
  - 小明不傻
- Deposit还需要针对帐号进行同步控制：synchronized(xm_account)
- 很麻烦
  - 小明：为什么不把Cashier和Account设计成线程安全的类？！

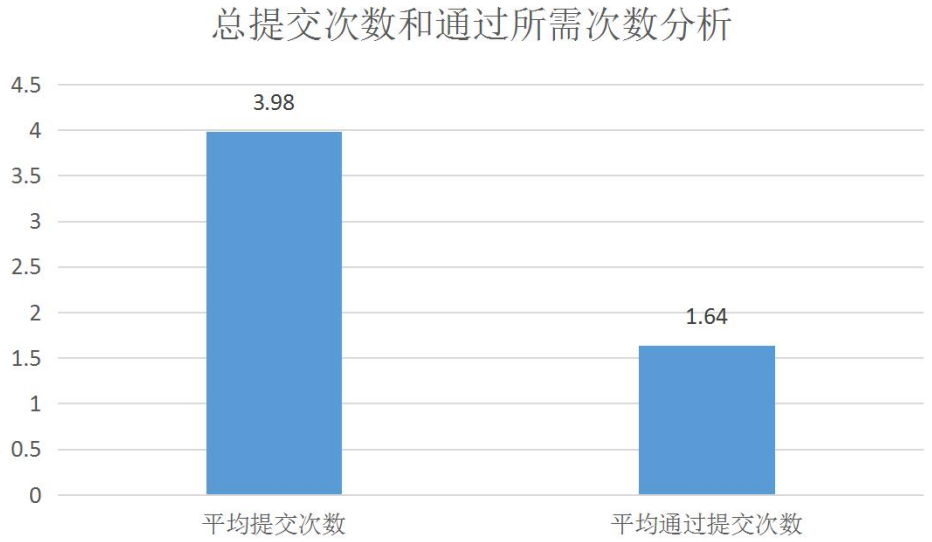
作业6训练要点分析

- 实践线程安全设计
  - 所有共享对象都要求设计为线程安全的类
- 编写代码来进行测试
  - 程序输入不是都通过显式的命令行或控制台
  - 很多软件都采取网络端口的监听等方式来获取外部输入，这时手工测试不仅仅是效率问题，而是个可行性问题
  - 测试程序的角色
    - 模拟产生所关注的输入
    - 调用被测模块进行”打靶式”测试
    - 调用相关模块获取信息做测试正确性判断

作业6的中测情况分析

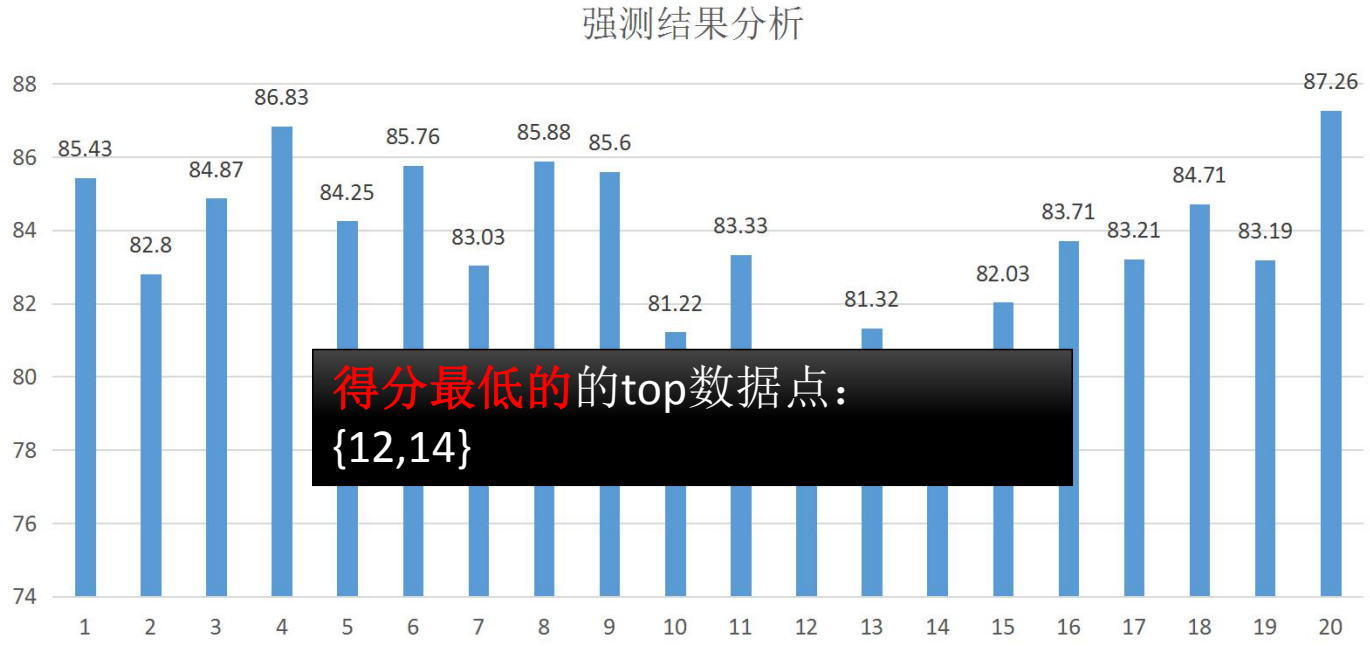

hw_6 的初通过提交次数分布
人数
160
140
120
100
80
60
40
20
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
初通过提交次数

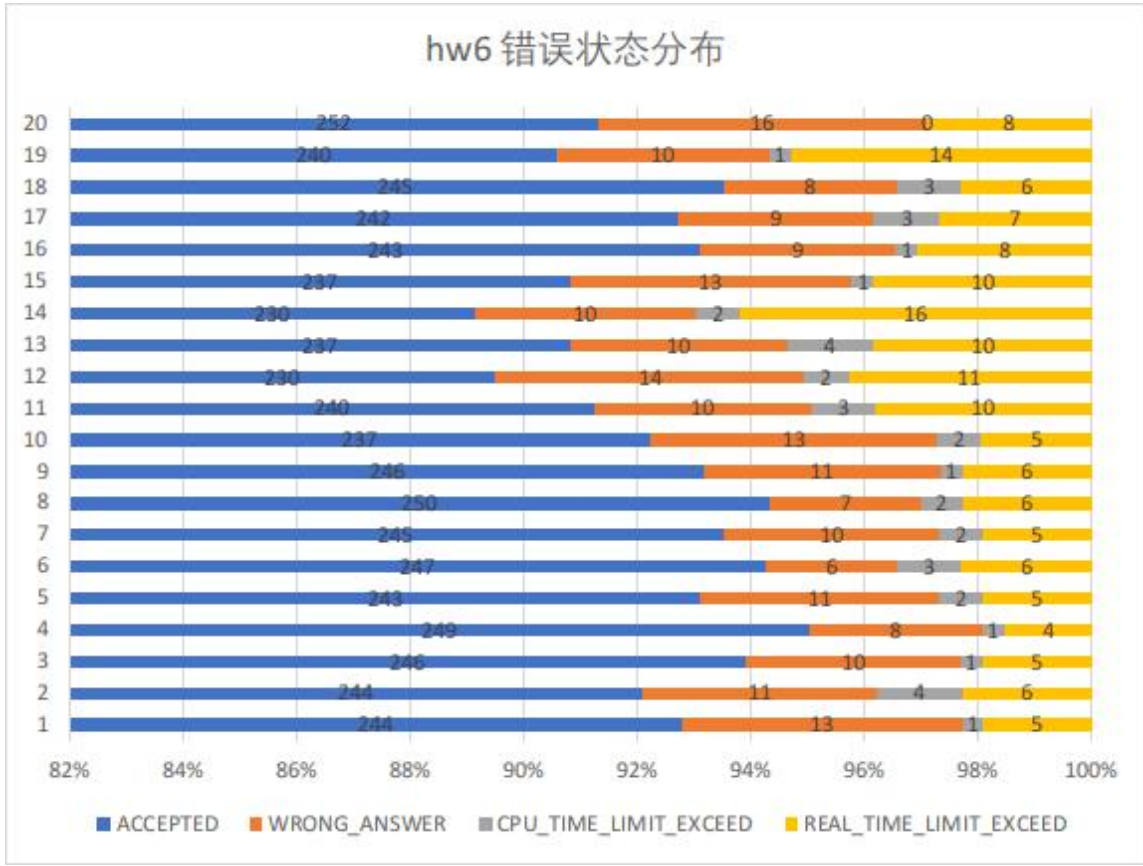

中测提交情况
65
8
269
PASSED
FAILED
UNSUBMITTED


总提交次数和通过所需次数分析
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
3.98
1.64
平均提交次数
平均通过提交次数

作业6的强测情况分析

发现**线程安全问题**的top数据点：
{2,11,12,13,14,15,17,19}





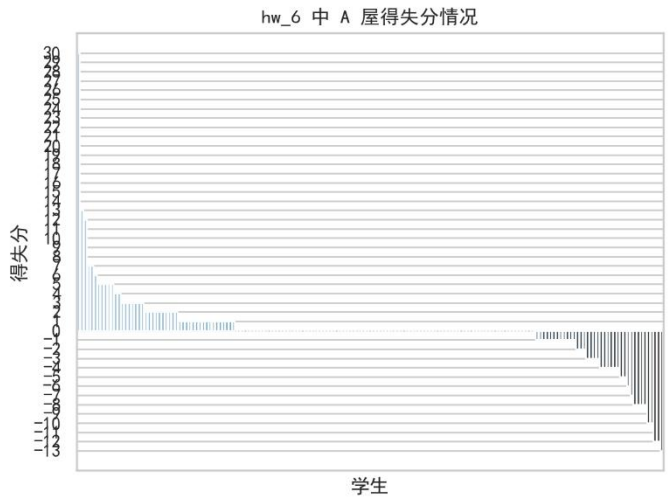
HW 6 强测数据得分较低的点特征：

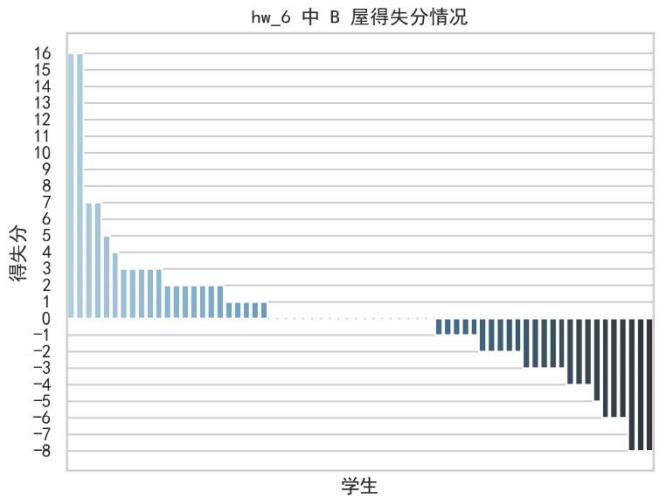
- 指令数多，需要高效规划
- 横向请求的间隔都最大，即两个楼座；纵向请求的间隔 >= 4
- 所有请求集中在同一楼座或同一楼层
- 一部分出发请求集中在同一楼座的同一楼层，一部分到达请求集中在同一楼座的同一楼层

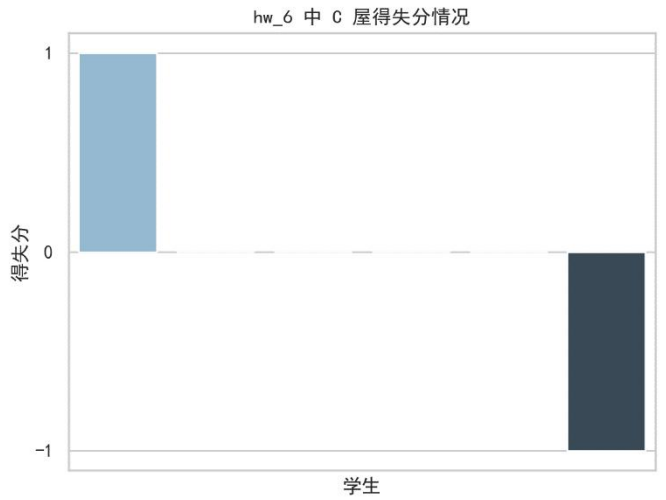
WA: 接送乘客出现问题
RTLE：线程安全问题
CTLE：线程安全或轮询
RE：线程安全

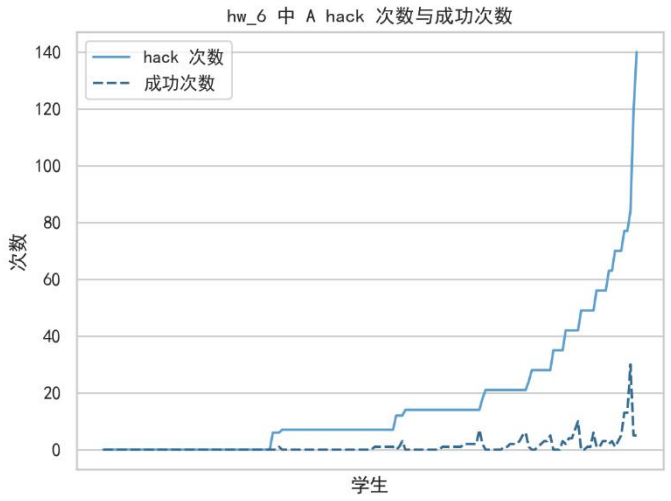
作业5的互测情况分析

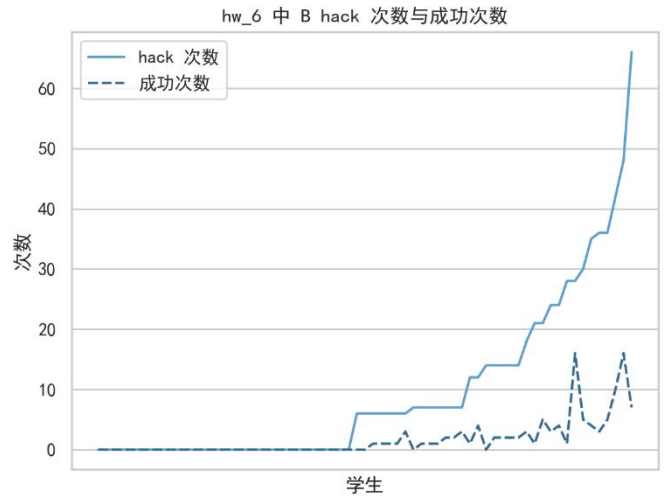
A级平均**16.36**次hack，成功率7.06%
B级平均**9.58**次hack，成功率17.60%
C级平均**2.5**次hack，成功率6.67%

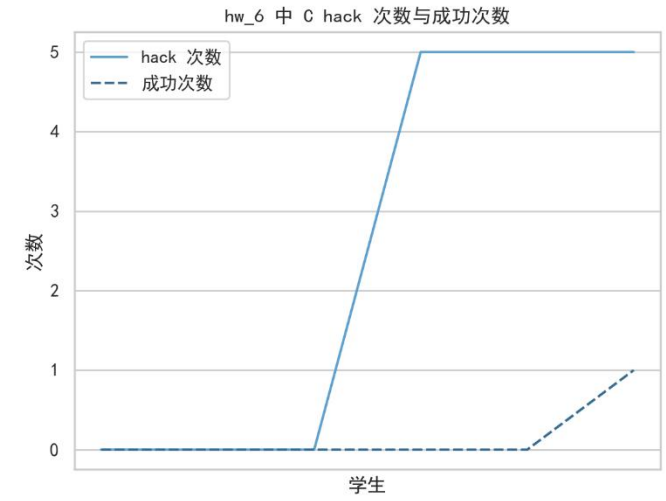










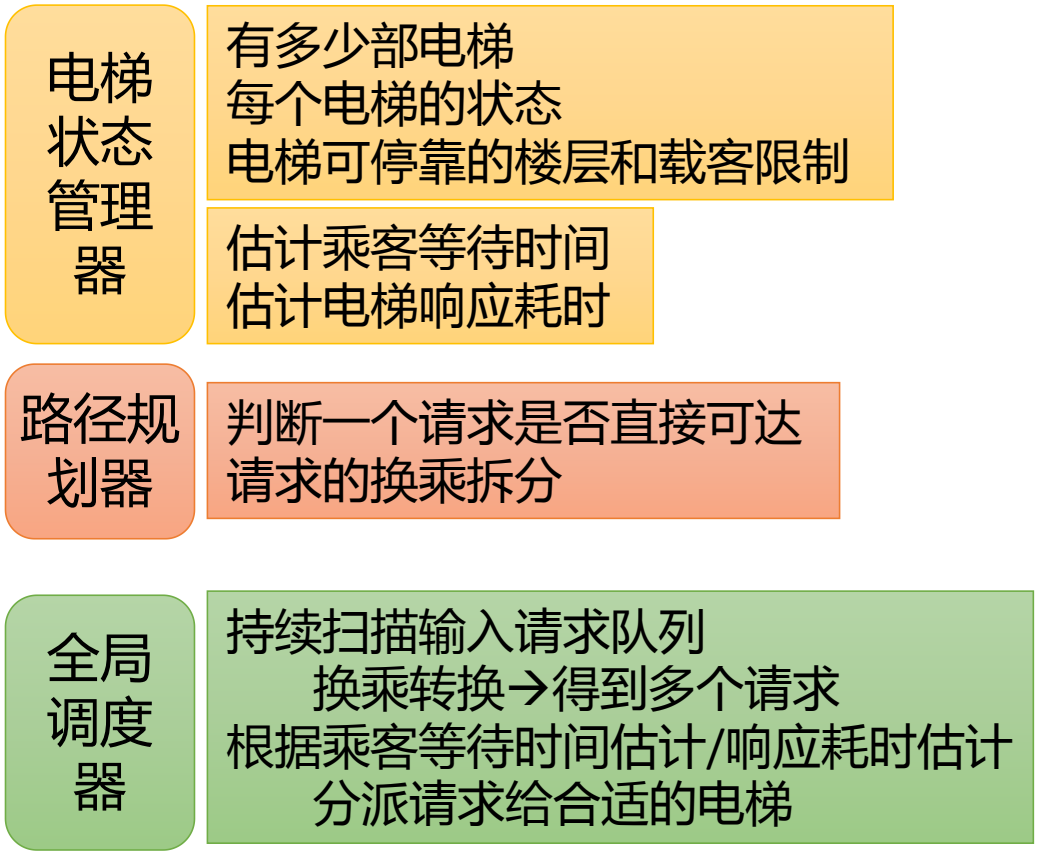


作业7训练要点分析

- 实践面向对象分析
  - 可配置电梯 ：可停靠楼层（横向电梯）、运行时间、载客限制
  - 乘客请求可以有多条路径来实现
    - 不得不换乘：路径可达性
    - 可以换乘：降低路径时间成本
  - 换乘
    - 一条请求转换成两个或更多请求
    - 本质上是路径搜索问题
    - 静态搜索、迭代式搜索

作业7训练要点分析

- 调度器需要减负（违背SRP、OCP）
  - 了解有多少部电梯
  - 了解每个电梯的状态
  - 了解电梯可停靠的楼层和载客限制
  - 扫描输入请求队列
  - 判断一个请求是否直接可达
  - 请求的换乘拆分
  - 分派请求给不同的电梯
  - 估计乘客等待时间
  - 估计电梯响应耗时



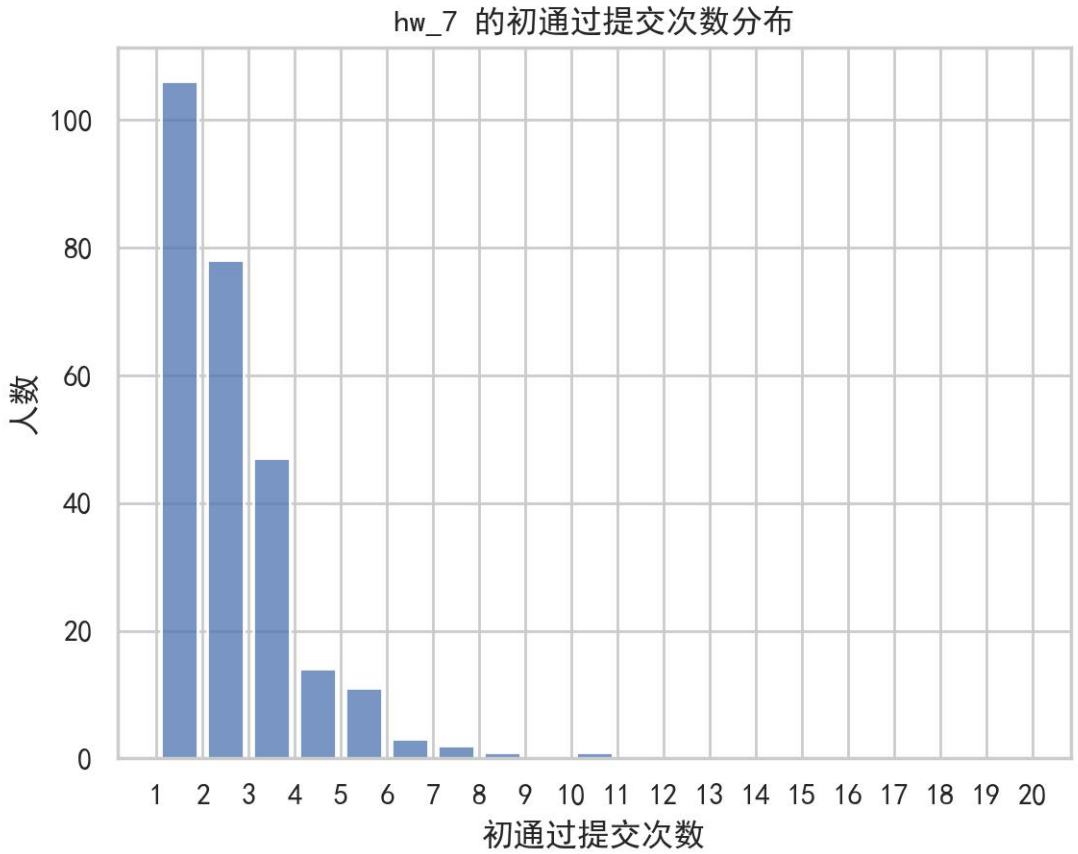
作业7训练要点分析

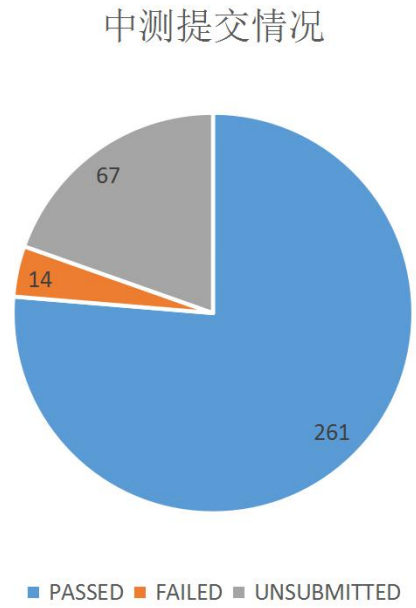
- 电梯对象的动态性涉及多个对象
  - 电梯的运行状态影响调度策略
  - 电梯的停靠状态受楼层停靠配置影响
  - 电梯的实际载客人数影响调度策略
  - 电梯的动态加入影响调度策略
- 优化目标虽然简单，但本质上是全局优化问题
  - 优化程度的考虑
  - 动态性的考虑

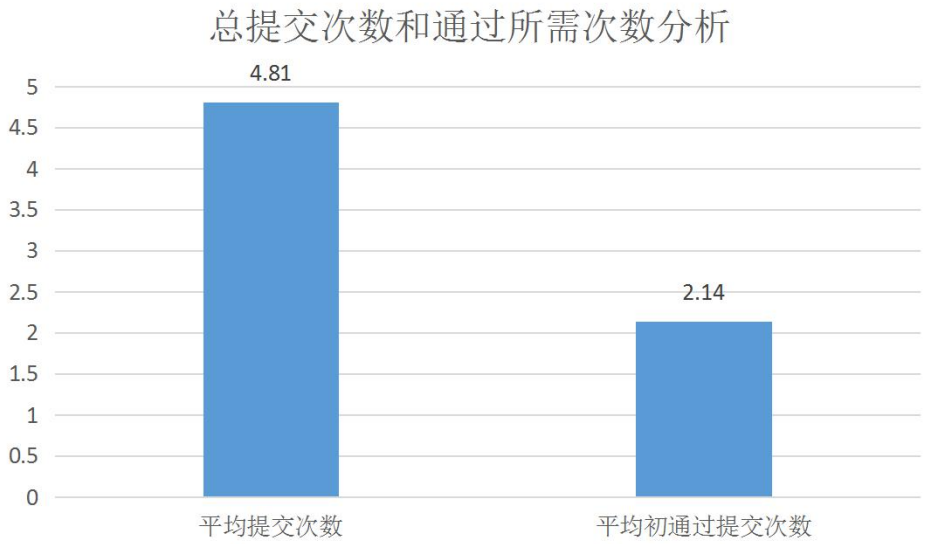
功能设计与性能设计的平衡

- 本单元三次作业的一个重要特点是影响性能的因素越来越多
  - 控制策略具有全局影响
  - 控制策略→改变对象状态→影响控制策略→…
- 性能不只是调度算法的问题
  - 分布式调度：全局调度、单电梯调度、换乘调度
  - 调度策略与电梯状态、乘客请求整体情况密切相关(耦合)
  - 实际执行时间(CPU)受线程调度的影响
- 乘客请求的到达模式会影响调度性能
  - 方向、楼层、间隔时间

作业7的中测情况分析

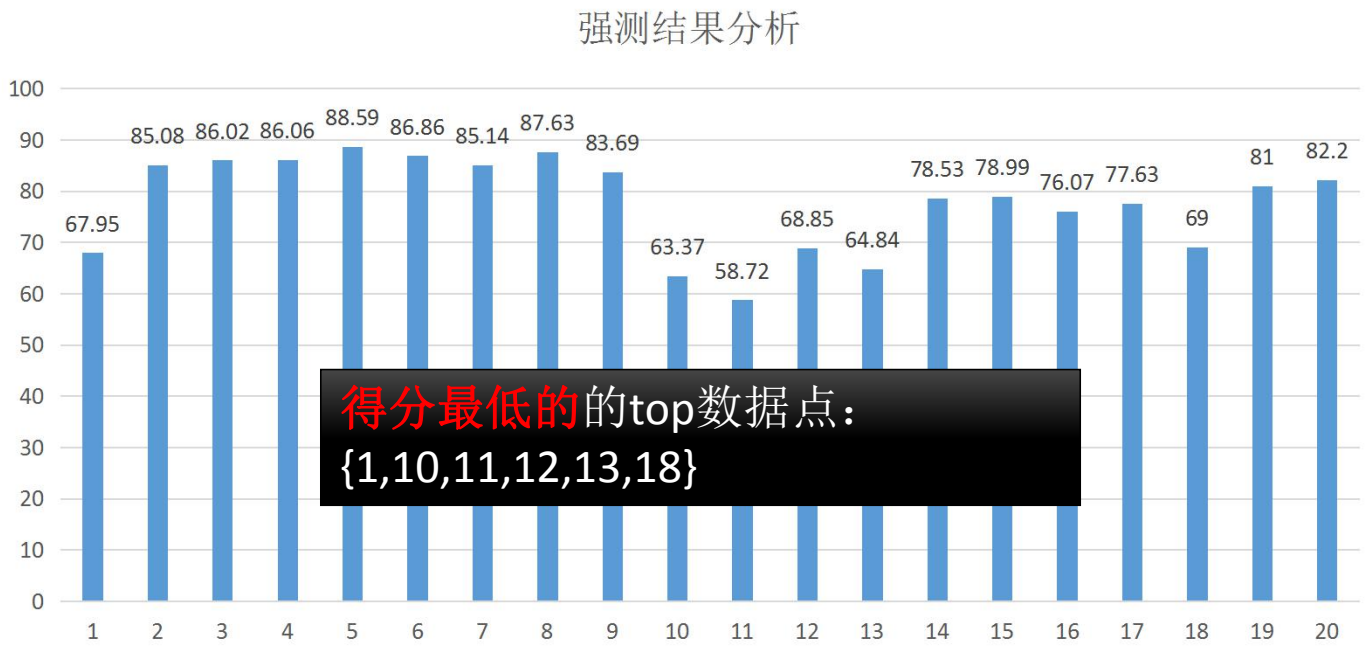

hw_7 的初通过提交次数分布
人数
初通过提交次数
0
20
40
60
80
100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

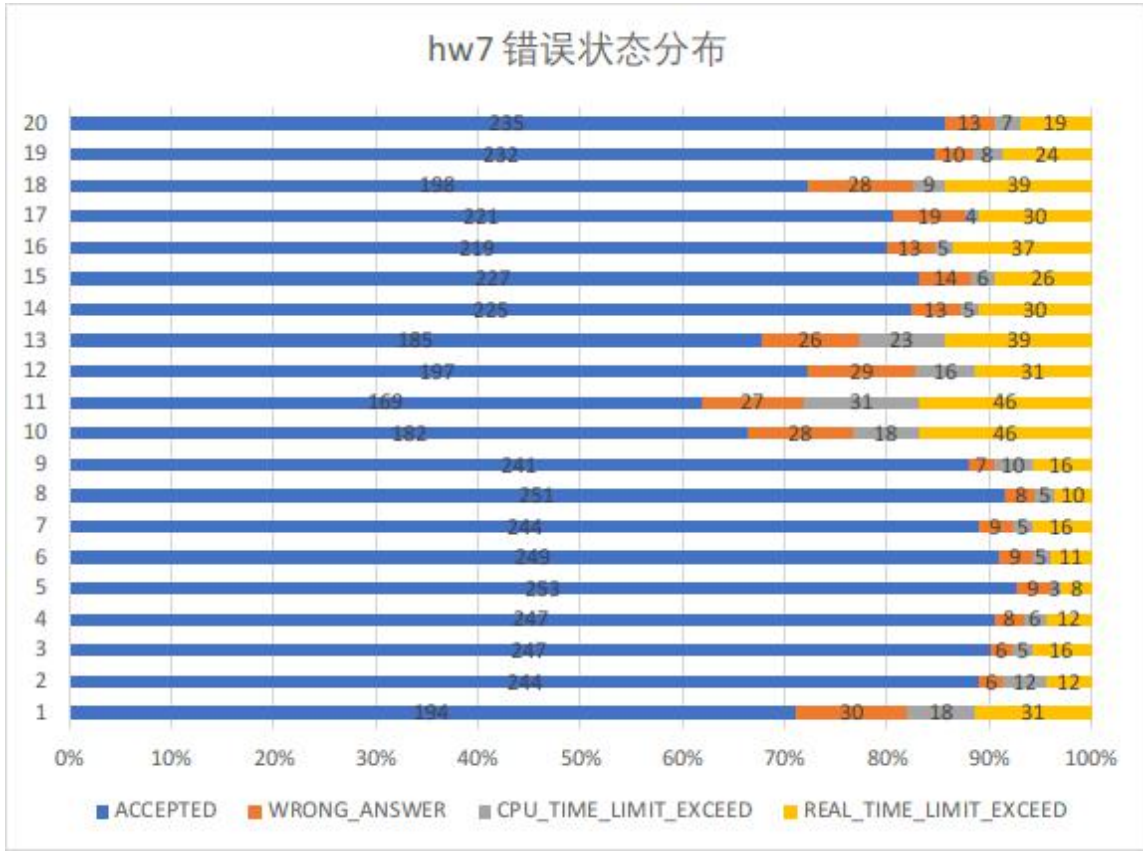

中测提交情况
67
14
261
PASSED
FAILED
UNSUBMITTED


总提交次数和通过所需次数分析
4.81
2.14
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
平均提交次数
平均初通过提交次数

作业7的强测情况分析

发现**线程安全问题**的top数据点：
{1,10,11,12,13,14,15,16,17,18,19}





HW 7 强测数据得分较低的点特征：

- 指令数多，需要高效规划
- 所有请求都需要至少进行一次横向与纵向电梯的换乘
- 所有到达请求集中在同一楼层
- 所有乘客在同一时间发出全部请求

WA: 接送乘客出现问题
RTLE：线程安全问题
CTLE：线程安全或轮询
RE：线程安全

作业

- 总结性博客作业
  - (1)总结分析三次作业中同步块的设置和锁的选择，并分析锁与同步快中处理语句直接的关系
  - (2)总结分析三次作业中的调度器设计，并分析调度器如何与程序中的线程进行交互
  - (3)从功能设计与性能设计的平衡方面，分析和总结自己**第三次作业**架构设计的可扩展性
    - 画UML类图
    - 画UML协作图(sequence diagram)来展示线程之间的协作关系（别忘记主线程）

作业

- (4)分析自己程序的bug
  - 分析未通过的公测用例和被互测发现的bug：特征、问题所在的类和方法
  - 特别注意分析那些与线程安全相关的问题（特别要注意死锁的分析）
- (5)分析自己发现别人程序bug所采用的策略
  - 列出自己所采取的测试策略及有效性
  - 分析自己采用了什么策略来发现线程安全相关的问题
  - 分析本单元的测试策略与第一单元测试策略的差异之处
- (6) 心得体会
  - 从线程安全和层次化设计两个方面来梳理自己在本单元三次作业中获得的心得体会