目录

基于 Android 多道作业批处理调度模拟实现

软件工程 2003 杜睿

1 设计目的

软件体系结构课程设计是在学生系统地学习了《软件设计与体系结构》课程后，按照软件设计模式的基本原理，综合运用所学的知识，设计开发一个应用场景，至少使用三种设计模式解决实际问题。通过对一个实际问题的分析、设计与实现，将原理与应用相结合，使学生学会如何把书本上学到的知识用于解决实际问题，培养学生的动手能力；另一方面，使学生能深入理解和灵活掌握教学内容。

2 系统设计

2.1 设计内容

模拟实现一个单（多）道批处理系统及其多级调度策略，输入作业序列（提交时间、服务时间、优先级 [可选]），输出作业进入内存时间、结束时间、平均周转时间，展示各作业的执行时间线，并对多种调度算法进行性能比较。

1. 模拟实现作业调度中先来先服务（FCFS）、短作业优先（SJF）和最高响应比调度（HRRN）算法。每次作业调度时，显示各作业的执行情况（开始执行时间，结束时间，周转时间）；最后，计算并列出平均周转时间并对相同情况下不同调度算法进行性能分析比较。

2. 模拟实现进程调度中基于优先级的时间片轮转调度算法（PRR）和多级反馈队列轮转调度算法（MFQ）。每次进行进程切换时，显示各进程执行情况（剩余时间，进程状态，进程排队情况），最后列出各进程的开始执行时间和结束时间。

2.2 需求分析

一个作业从用户提交开始到占有处理机被执行，一般来要由系统三级调度才能实现，即作业调度、内存调度、进程调度。其中最为重要的:

作业调度 主要是完成作业[1]从*后备状态*到*可执行状态*的转变，即按照某原则，从*外存后备队列*中挑选作业，将其装入*内存*并为其创建进程;

进程调度 主要是完成进程从*就绪状态*到*执行（完成）状态*的转变，即按照某策略，从*内存就绪队列*中选取进程，为其分配处理机使其运行。

表 1: 常用的处理机调度策略

算法名称	主要适用范围	默认调度方式
先来先服务	作业调度 & 进程调度	非抢占式
短作业（进程）优先	作业调度 & 进程调度	非抢占式
高响应比优先	作业调度	非抢占式
时间片轮转	进程调度	抢占式（不抢时间片）
多级反馈队列	进程调度	抢占式（抢占时间片）

* 调度策略也就是调度算法

基于上述理论常识，可以做出如下基本假设:

1. 作业调度和进程调度是互补共存的层级关系。不考虑内存调度的前提下，实现任何算法，至少要实现两级调度：作业调度及进程调度。
2. 作业和进程可以合二为一，统一抽象为任务[2]。不考虑阻塞的前提下，一个任务大致可归为未提交态、收容态、就绪态、运行态及销毁态。
3. 调度是种将何资源分配给何任务的决策行为。从各级调度的交互上说，调度可以分为：本级进行调度、移交上层调度及委托下层调度。

2.3 概要设计

要模拟批处理系统，就须要模拟时钟、任务及其状态、分级调度。

[1]用户向计算机提交任务的任务实体，《计算机操作系统教程（第四版）》，清华大学出版社

[2]作业是提交层面的任务，进程是执行层面的任务，作者注

时钟 标定时间流逝的基准，以便执行作业调度、进程调度、性能分析等。每流逝一个单位时间，系统至少要进行一次调度。所以，可以建立“时钟”与“调度器”之间的依赖关系，当时钟发生改变时应该自动通知调度器，调度器接受信息并做出响应。时钟，被称为观察目标；调度器，被称为观察者。一个观察目标可以对应对个观察者，可以根据需要增加和删除观察者。这种设计模式被称为*观察者模式*或*发布-订阅模式*。

任务 每个任务都有提交时间、到达时间等属性；运行中任务的剩余时间随时间流逝而减少

从这种意义说，运行中进程应当观察时间流逝，并自主改变剩余时间字段。换言之，理论上说，任务应当与调度类共同作为时钟的*观察者*。

任务揉杂作业和进程，集成两者的属性，即 id、优先级、状态、时钟，提交时间、预计需要时间，到达时间、开始时间、剩余时间、完成时间。

在模拟逻辑上，状态包含五种类型：未提交、收容、就绪、运行、结束。未提交，代表用户输入作业序列中的提交时间晚于当前时钟时刻；收容，代表作业已提交，进入外存收容队列（外存一般没有容量限制）；就绪，代表进程已创建，进入内存就绪队列（内存一般有容量限制）；运行，代表进程获得处理机资源，随时间流逝剩余时间递减。

调度 任务状态由调度器进行切换；调度是分层级的。

作业提交与调度细节应当是解耦的，同时为较好地实现“分级”调度，可以将调度器链式组织，各司其职，又保证了作业在各级调度中的传递，直至进程销毁并记录日志。这种思想来源于*职责链模式*。

机器 整合时钟、任务、调度，并接受测试样例，提供日志。不同策略在各层级上排列，可以衍生成为多种具体“机器”，例如，以先来先服务作为作业调度、以抢占式多级反馈队列作为进程调度，内存无限制的多道批处理机。为将对象创建和对象使用解耦，可以使用*工厂方法模式*。

对于各算法性能分析而言，各个“处理机”，由于模拟时减少任务的剩余运行时间，应仅仅保有用户提交序列的副本。即通过拷贝一个已经存在的实例来返回新的实例，而不是新建实例。被复制的实例就是我们所称的“原型”，这个原型是可定制的，例如，只拷贝标识符、提交时间、服务时间和优先级。这种设计模式被称为*原型模式*。

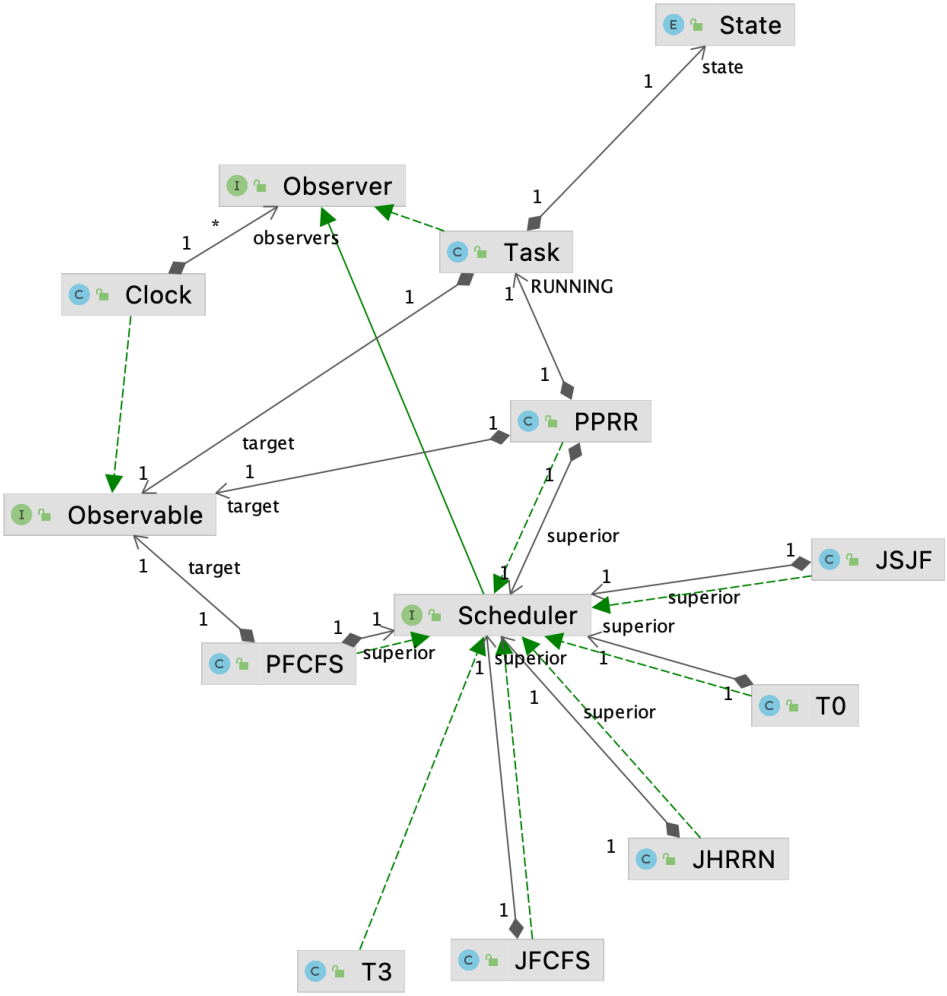
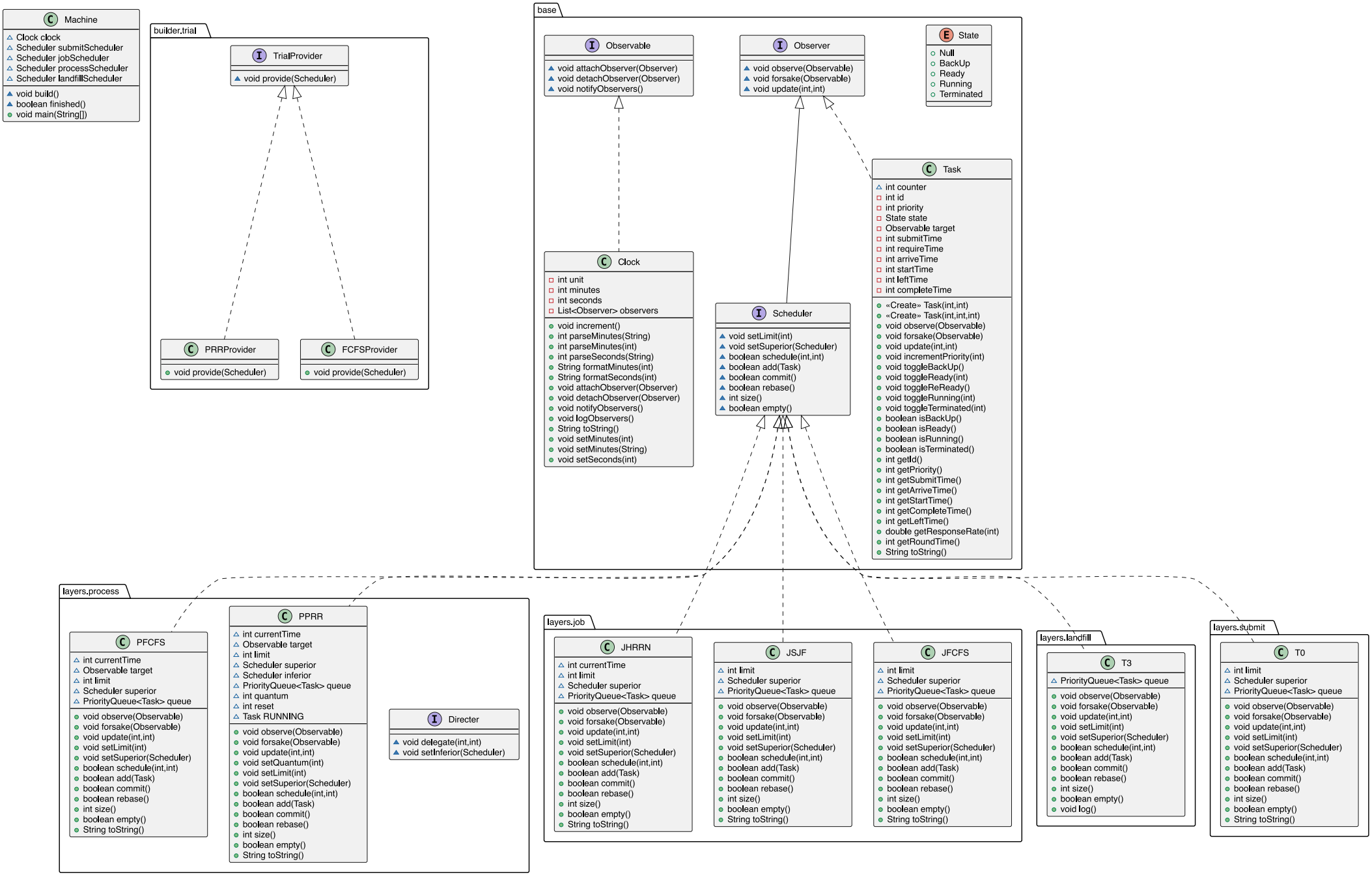

State
state
1
1
Observer
*
1
observers
Task
Clock
1
1
RUNNING
1
1
PPRR
target
1
1
1
target
Observable
1
superior
1
JSJF
target
1
1
1
superior
Scheduler
1
1
1
superior
1
superior
1
1
PFCFS
superior
1
T0
1
superior
1
JHRRN
1
1
T3
JFCFS

图 1: UML 类图（第一版）


Machine
Clock clock
Scheduler submitScheduler
Scheduler jobScheduler
Scheduler processScheduler
Scheduler landfillScheduler
void build()
boolean finished()
void main(String[])
builder.trial
TrialProvider
void provide(Scheduler)
PRRProvider
void provide(Scheduler)
FCFSProvider
void provide(Scheduler)
base
Observable
void attachObserver(Observer)
void detachObserver(Observer)
void notifyObservers()
Observer
void observe(Observable)
void forsake(Observable)
void update(int,int)
State
Null
BackUp
Ready
Running
Terminated
Clock
int unit
int minutes
int seconds
List<Observer> observers
void increment()
int parseMinutes(String)
int parseMinutes(int)
int parseSeconds(String)
String formatMinutes(int)
String formatSeconds(int)
void attachObserver(Observer)
void detachObserver(Observer)
void notifyObservers()
void logObservers()
String toString()
void setMinutes(int)
void setMinutes(String)
void setSeconds(int)
Scheduler
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
Task
int counter
int id
int priority
State state
Observable target
int submitTime
int requireTime
int arriveTime
int startTime
int leftTime
int completeTime
«Create» Task(int,int)
«Create» Task(int,int,int)
void observe(Observable)
void forsake(Observable)
void update(int,int)
void incrementPriority(int)
void toggleBackUp()
void toggleReady(int)
void toggleReReady()
void toggleRunning(int)
void toggleTerminated(int)
boolean isBackUp()
boolean isReady()
boolean isRunning()
boolean isTerminated()
int getId()
int getPriority()
int getSubmitTime()
int getArriveTime()
int getStartTime()
int getCompleteTime()
int getLeftTime()
double getResponseRate(int)
int getRoundTime()
String toString()
layers.process
PFCFS
int currentTime
Observable target
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
PPRR
int currentTime
Observable target
int limit
Scheduler superior
Scheduler inferior
PriorityQueue<Task> queue
int quantum
int reset
Task RUNNING
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setQuantum(int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
Directer
void delegate(int,int)
void setInferior(Scheduler)
layers.job
JHRRN
int currentTime
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
JSJF
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
JFCFS
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
layers.landfill
T3
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
void log()
layers.submit
T0
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()

2.4 质量评审

基于 2.3 实现的系统，将存在一个致命缺陷：延时。之所以会延时，主要是因为各个时间观察者对于时间流逝的感知顺序是固定、单向的。

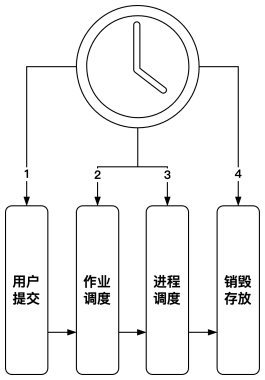


图 2: 通知顺序

虽然从层级上说，作业调度高于（先于）进程调度，但是从顺序上说，在单位时间流逝后，有可能出现先作业调度、再进程调度、再作业调度的情况。

表 2: 错误案例（单道批处理系统）

时刻	收容队列	就绪队列	备注
09:59	{}	{[运行，余 1 分钟]}	初始状态
10:00	{}	{[运行，余 1 分钟]}	用户提交新作业
10:00	{[收容，余 1 分钟]}	{[运行，余 1 分钟]}	内存已满 | 无作业调度
10:00	{[收容，余 1 分钟]}	{[运行，余 1 分钟]}	正在运行 | 无进程调度
10:00	{[收容，余 1 分钟]}	{[结束，余 0 分钟]}	进程自主修改剩余时间
10:01	{[收容，余 1 分钟]}	{[结束，余 0 分钟]}	内存已满 | 无作业调度
10:01	{[收容，余 1 分钟]}	{}	释放进程（进程调度）
10:02	{}	{[就绪，余 1 分钟]}	掉入内存（作业调度）
10:02	{}	{[运行，余 1 分钟]}	运行进程（进程调度）
10:02	{}	{[结束，余 0 分钟]}	进程自主修改剩余时间
...	...	...	...

本错误案例中，10:00～10:01 处理机竟然处于空闲状态。这是因为：

- 进程运行不受进程调度控制，进程调度没有及时将结束进程向上提交；
- 进程调度即使向上提交，作业调度也无法预知未来，并为其创建进程。

下面提出两种可能的改进方案，并加以分析：

调整时间单位 承认并接受延时的存在，并将时间继续细分。例如，倘若单位设定为“秒”，那么在“分钟”维度上看，延时仿佛并不存在。不过，只要没有无限细分，延时累积最终必将影响结果。

优化通信机制 扭转各层对时间流逝固定、单向的感知顺序。拟人地说，各层消息传递机制不能只包含“向上告知”，还要包含“向下问责”。进程调度结束后，倘若可以通知作业调度：“你那里还有什么要我签字处理的吗？”；那么，延时问题说不定就可以迎刃而解。

此外，在上一节的基础上，添加以下前提假设，以便系统实现：时钟的最小颗粒单位为分钟，所有作业均在 23:59 前提交并结束。倘若用户要求输入时间单位不统一，可以尝试添加*适配器*。

3 系统实现

3.1 核心实现

以“多级队列反馈队列”为例，我们需要实现（类图见下页）：

- 核心组件：
  - 任务、状态
  - 时钟（观察对象）
  - 提交“调度”、作业调度、进程调度、销毁“调度”（严格意义上说，提交和销毁不属于调度，它们是对定时提交和进程销毁的一种模拟）
- 边缘组件：
  - 处理机工厂，用不同算法构建各级调度
  - 测试样例工厂，创建任务并添加至提交“调度”

多级调度各司其职，共同作为批处理系统的重要组成部分发挥作用：

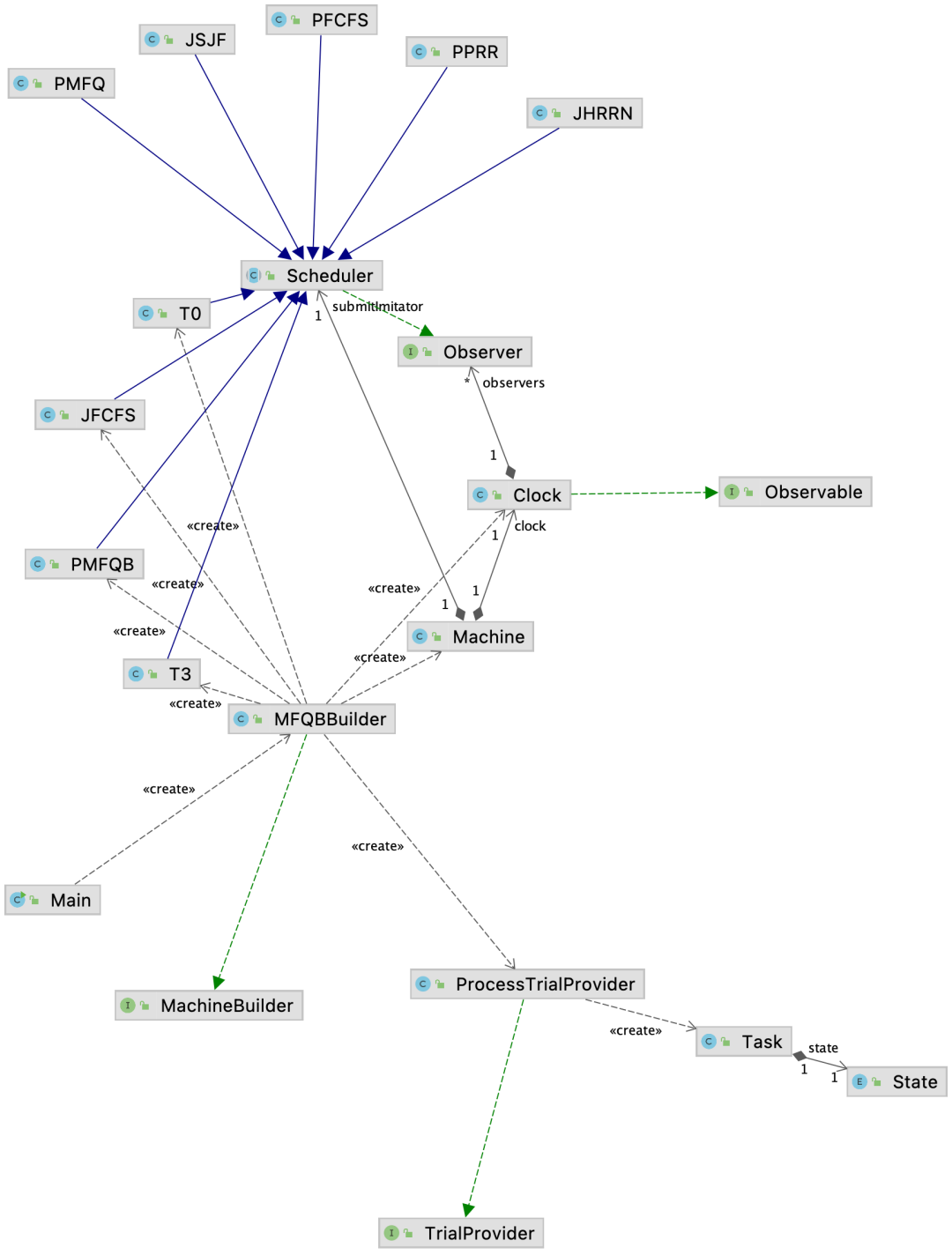

PFCFS
JSJF
PPRR
PMFQ
JHRRN
Scheduler
T0
submitImitator
1
Observer
*
observers
JFCFS
1
Clock
Observable
«create»
1
clock
PMFQB
«create»
«create»
1
1
«create»
Machine
«create»
T3
«create»
MFQBBuilder
«create»
«create»
Main
MachineBuilder
ProcessTrialProvider
«create»
Task
state
1
1
State
TrialProvider

图 3: UML 类图（第二版）

表 3: 常用的处理机调度策略

多级调度	调度方式	调度时机	调度次序	任务状态
用户提交	↑	提交时刻 ≤ 当前时刻	提交时间	未提交
作业调度	↑	时间流逝 & 上层命令	*	收容
进程调度	↑、↔、↓	时间流逝	**	就绪、运行
进程销毁				销毁

[1] ↑ 移交上层调度，↔ 本层级调度，↓、委托下层调度
[2] * 提交时间、剩余时间、响应比
[3] ** 优先级（多级队列优先次序）

3.1.1 时钟

接口:

```
public interface Observable {
    void attachObserver(Observer o);

    void detachObserver(Observer o);

    void notifyObservers();
}
```

具体实现:

```
public class Clock implements Observable {
    public static final int UNIT = 1;
    // main attribute
    public static int minutes = -1;
    // subscribers
    private final List<Observer> observers = new ArrayList<>();

    // TODO timeline logger

    public String getMilestones() {
```

```
11          return null;
12      }
13
14      // increment by one minute
15      public void incrementByUnit() {
16          minutes += UNIT;
17          notifyObservers();
18          Log.e("time-line", this + "");
19          for (Observer o : observers) {
20              Log.e("time-line", o + "");
21          }
22      }
23
24      // helpers
25      public static int parsePatternIntoMinutes(String pattern) {
26          String[] parts = pattern.split(":");
27          return Integer.parseInt(parts[0]) * 60 + Integer.parseInt(parts[1]);
28      }
29
30      public static String generatePatternFromMinutes(int minutes) {
31          return String.format(Locale.CHINA, "%02d:%02d", minutes / 60, minutes % 60);
32      }
33
34
35      // constructors & setters
36      public Clock(int minutes) {
37          Clock.minutes = minutes - 1;
38      }
39
40      public Clock(String pattern) {
41          minutes = parsePatternIntoMinutes(pattern) - 1;
42      }
43
```

```
44      // observable implementation
45      @Override
46      public void attachObserver(Observer o) {
47          observers.add(o);
48      }
49
50      @Override
51      public void detachObserver(Observer o) {
52          observers.remove(o);
53      }
54
55      @Override
56      public void notifyObservers() {
57          for (Observer o : observers) {
58              o.update(minutes, UNIT);
59          }
60      }
61
62      // logger
63      @Override
64      public String toString() {
65          return generatePatternFromMinutes(minutes);
66      }
67  }
68
```

3.1.2 任务

```java
public class Task implements Cloneable {
    // auto-increment
    private static int COUNTER = 0;

    // universal attribute
    private final int id;
    private int priority = 0;
    private State state = State.NULL;

    // exclusive for job scheduling
    private final int jobSubmitTime;
    private final int estimatedRequireTime;

    // exclusive for process scheduling
    private int processLeftTime;
    private int processArriveTime = -1;
    private int processStartTime = -1;
    private int processCompleteTime = -1;

    public Task(int id, int priority, int jobSubmitTime, int estimatedRequireTime) {
        this.id = id;
        this.priority = priority;
        this.jobSubmitTime = jobSubmitTime;
        this.estimatedRequireTime = estimatedRequireTime;
        this.processLeftTime = estimatedRequireTime;
    }

    public Task(int priority, int jobSubmitTime, int estimatedRequireTime) {
        this.id = COUNTER++;
        this.priority = priority;
        this.jobSubmitTime = jobSubmitTime;
```

```
32          this.estimatedRequireTime = estimatedRequireTime;
33          this.processLeftTime = estimatedRequireTime;
34      }
35
36      public Task(int jobSubmitTime, int estimatedRequireTime) {
37          this.id = COUNTER++;
38          this.jobSubmitTime = jobSubmitTime;
39          this.estimatedRequireTime = estimatedRequireTime;
40          this.processLeftTime = estimatedRequireTime;
41      }
42
43      public void decrementLeftTime(int currentTime, int elapseUnit) {
44          if (isRunning()) {
45              this.processLeftTime -= elapseUnit;
46              if (processLeftTime <= 0) {
47                  toggleTerminated(currentTime);
48              }
49          }
50      }
51
52      public void incrementPriority(int diff) {
53          this.priority += diff;
54      }
55
56      public void toggleBackup() {
57          this.state = State.BACKUP;
58      }
59
60      public void toggleReady(int arriveTime) {
61          this.state = State.READY;
62          this.processArriveTime = arriveTime;
63      }
64
```

```
65      public void toggleReReady() {
66          this.state = State.READY;
67      }
68
69      public void toggleTerminated(int completeTime) {
70          this.state = State.TERMINATED;
71          this.processCompleteTime = completeTime;
72      }
73
74      public void toggleRunning(int startTime) {
75          this.state = State.RUNNING;
76          if (processStartTime == -1) {
77              this.processStartTime = startTime;
78          }
79      }
80
81      public boolean isBackup() {
82          return this.state == State.BACKUP;
83      }
84
85      public boolean isReady() {
86          return this.state == State.READY;
87      }
88
89      public boolean isRunning() {
90          return this.state == State.RUNNING;
91      }
92
93      public boolean isTerminated() {
94          return this.state == State.TERMINATED;
95      }
96
97      public int getId() {
```

```
98          int factor = (isRunning()) ? (-1) : (1);
99          return id * factor;
100     }
101
102     public int getJobSubmitTime() {
103         int factor = (isRunning()) ? (-1) : (1);
104         return jobSubmitTime * factor;
105     }
106
107     public int getPriority() {
108         int factor = (isRunning()) ? (-1) : (1);
109         return priority * factor;
110     }
111
112     public int getProcessLeftTime() {
113         int factor = (isRunning()) ? (-1) : (1);
114         return processLeftTime * factor;
115     }
116
117     public int getProcessArriveTime() {
118         int factor = (isRunning()) ? (-1) : (1);
119         return processArriveTime * factor;
120     }
121
122     public int getProcessStartTime() {
123         int factor = (isRunning()) ? (-1) : (1);
124         return processStartTime * factor;
125     }
126
127     public int getProcessCompleteTime() {
128         int factor = (isRunning()) ? (-1) : (1);
129         return processCompleteTime * factor;
130     }
```

```
131
132     public int getEstimatedRequireTime() {
133         return estimatedRequireTime;
134     }
135
136     public double getResponseRate(int currentTime) {
137         int factor = (isRunning()) ? (-1) : (1);
138         return (1 + ((1.0 * currentTime - jobSubmitTime)
139         / estimatedRequireTime)) * factor;
140     }
141
142     public int getRoundTime() {
143         int factor = (isRunning()) ? (-1) : (1);
144         return (processCompleteTime - jobSubmitTime) * factor;
145     }
146
147     public String toString() {
148         return String.format("{%d,%s,%s}", id, state,
149             Clock.generatePatternFromMinutes(processLeftTime));
150     }
151
152     public Object clone() {
153         return new Task(this.id, this.priority, this.jobSubmitTime,
154             this.estimatedRequireTime);
155     }
156 }
```

3.1.3 机器

```
1  public class Machine {
2      private String TAG;
3      private Clock clock;
4      private Scheduler submitImitator;
5      private Scheduler jobScheduler;
6      private Scheduler processScheduler;
7      private Scheduler landfilImitator;
8
9      public void run() {
10         build();
11         while (!(submitImitator.isEmpty()
12             && jobScheduler.isEmpty() && processScheduler.isEmpty())) {
13             clock.incrementByUnit();
14         }
15
16         ((T3) landfilImitator).getFinalSnapShot();
17     }
18
19     // getters
20     public float getAverageRoundTime() {
21         return ((T3) landfilImitator).getAverageRoundTime();
22     }
23
24     public String getTAG() {
25         return TAG;
26     }
27
28     public String getFinalSnapShot() {
29         return ((T3) landfilImitator).getFinalSnapShot();
30     }
31
```

```
    public Object[] getFinalSnapShotTasks() {
        return ((T3) landfilImitator).getFinalSnapShotTasks();
    }

    // setters
    public Machine setQuantum(int quantum) {
        if (processScheduler.getClass().equals(PPRR.class)) {
            ((PPRR) processScheduler).setQuantum(quantum);
        }
        return this;
    }

    public Machine setFactor(int factor) {
        if (processScheduler.getClass().equals(PMFQ.class)) {
            ((PMFQ) processScheduler).setFactor(factor);
        } else if (processScheduler.getClass().equals(PMFQB.class)) {
            ((PMFQB) processScheduler).setFactor(factor);
        }
        return this;
    }

    public Machine setTrial(List<Task> tasks) {
        int starter = Integer.MAX_VALUE;
        for (Task task : tasks) {
            // FIXME prototype
            submitImitator.add((Task) task.clone());
            starter = Math.min(starter, task.getJobSubmitTime());
        }
        this.clock = new Clock(starter);
        return this;
    }

    public Machine setTAG(String TAG) {
```

```
65          this.TAG = TAG;
66          return this;
67      }
68
69      public Machine setClock(Clock clock) {
70          this.clock = clock;
71          return this;
72      }
73
74      public Machine setSubmitImitator(Scheduler submitImitator) {
75          this.submitImitator = submitImitator;
76          return this;
77      }
78
79      public Machine setJobScheduler(Scheduler jobScheduler) {
80          this.jobScheduler = jobScheduler;
81          return this;
82      }
83
84      public Machine setProcessScheduler(Scheduler processScheduler) {
85          this.processScheduler = processScheduler;
86          return this;
87      }
88
89      public Machine setLandfilImitator(Scheduler landfilImitator) {
90          this.landfilImitator = landfilImitator;
91          return this;
92      }
93
94
95
96
97
```

```
98      // Observer & Chain of Responsibility
99      private void build() {
100         submitImitator.observe(clock);
101         jobScheduler.observe(clock);
102         processScheduler.observe(clock);
103         landfilImitator.observe(clock);
104         submitImitator.setSuperior(jobScheduler);
105         jobScheduler.setSuperior(processScheduler);
106         processScheduler.setSuperior(landfilImitator);
107         landfilImitator.setInferior(processScheduler);
108         processScheduler.setInferior(jobScheduler);
109         jobScheduler.setInferior(submitImitator);
110     }
111 }
112
```

3.1.4 机器建造者

接口:

```
public interface MachineBuilder {
    Machine build();
}
```

具体实现，以先来先服务为例:

```
public class FCFSBuilder implements MachineBuilder {
public Machine build() {
    return new Machine()
            .setTAG("FCFS")
            .setClock(new Clock("00:00"))
            .setSubmitImitator(new T0())
            .setJobScheduler(new JFCFS())
            .setProcessScheduler(new PFCFS().setLIMIT(1))
            .setLandfilImitator(new T3());
    }
}
```

3.1.5 作业调度

实现单道批处理系统，作业调度分别使用“先来先服务算法”、“短作业优先算法”、“高响应比优先算法”，进程调度统一使用“先来先服务算法”。

图 4: 作业调度层

作业调度的职责是按照一定的策略，向进程调度提交任务（填入内存）。FCFS、HRRN、SJF 的不同主要体现于调度次序，即优先队列的比较方式。本节主要以高响应比优先实现为例，介绍抽象调度类与作业调度层。

```
1 public abstract class Scheduler implements Observer {
2     protected int LIMIT = Integer.MAX_VALUE;
3     protected Scheduler superior = null;
4     protected Scheduler inferior = null;
5
6     public void setLIMIT(int LIMIT) {this.LIMIT = LIMIT;}
7
8     public void setSuperior(Scheduler superior) {this.superior = superior;}
9
```

```
10      public void setInferior(Scheduler inferior) {this.inferior = inferior;}
11
12      @Override
13      public void observe(Observable target) {target.attachObserver(this);}
14
15      @Override
16      public void forsake(Observable target) {target.detachObserver(this);}
17
18      public abstract boolean add(Task t);
19
20      public abstract boolean reportToSuperior();
21
22      public abstract boolean notifyToInferior();
23
24      public abstract boolean rebase();
25
26      public abstract int size();
27
28      public abstract boolean isEmpty();
29
30      public abstract void shuffle();
31  }
```

下面对于各个接口，分别解释具体实现：

Listing 1: 收容队列（基于响应比的优先队列）

```
1  private PriorityQueue<Task> queue = new PriorityQueue
       <>(new Comparator<Task>() {
2      @Override
3      public int compare(Task o1, Task o2) {
4          return (o2.getResponseRate(Clock.minutes) - o1
               .getResponseRate(Clock.minutes) > 0) ? (1)
               : (-1);
5      }
```

```
});
```

Listing 2: 调度方式（向上提交），调度时机（时间流逝）

```
@Override
public void update(int currentTime, int elapseUnit) {
    boolean flag = true;
    while (queue.peek() != null && flag) {
        flag = reportToSuperior();
    }
}
```

Listing 3: 添加作业（收容状态）

```
@Override
public boolean add(Task t) {
    if (queue.size() >= LIMIT)
        return false;

    t.toggleBackup();
    queue.offer(t);
    return true;
}
```

Listing 4: 模拟为作业创建进程，装填进入内存

```
@Override
public boolean reportToSuperior() {
    if (superior == null || queue.isEmpty())
        return false;

    shuffle();
    if (!superior.add(queue.peek()))
        return false;

    return rebase();
```

```
11 }
```

Listing 5: 在本层队列中移除任务

```
1 @Override
2 public boolean rebase() {
3     if (queue.isEmpty())
4         return false;
5
6     queue.poll();
7     return true;
8 }
```

Listing 6: 响应比动态变化，刷新队列顺序

```
1 @Override
2 public void shuffle() {
3     if (queue.size() >= 2) {
4         queue.offer(queue.poll());
5     }
6 }
```

需要特别说明的是，短作业优先（SJF, Shortest Job First）也有抢占式的版本，通常被称为最短剩余时间优先。即，每当就绪队列改变时都需要进行调度，如果新到达的进程剩余时间比当前运行进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。

3.1.6 进程调度

实现多道批处理系统（容量无限），作业调度统一使用“先来先服务”，进程调度分别使用“基于动态优先级的时间片轮转法”、“多级反馈队列法”。
本节主要介绍四种算法的实现异同。共性，例如，只有进程调度会通知下层。

Listing 7: 命令下层再次进行调度

```
1 @Override
2 public boolean notifyToInferior() {
```

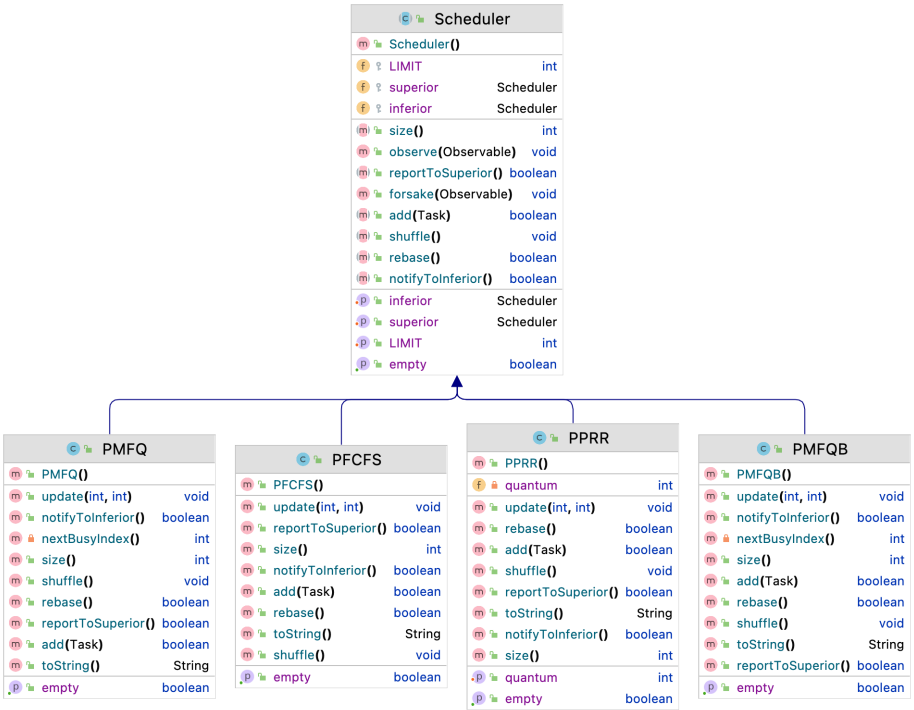

Scheduler
Scheduler()
LIMIT int
superior Scheduler
inferior Scheduler
size() int
observe(Observable) void
reportToSuperior() boolean
forsake(Observable) void
add(Task) boolean
shuffle() void
rebase() boolean
notifyToInferior() boolean
inferior Scheduler
superior Scheduler
LIMIT int
empty boolean
PMFQ
PMFQ()
update(int, int) void
notifyToInferior() boolean
nextBusyIndex() int
size() int
shuffle() void
rebase() boolean
reportToSuperior() boolean
add(Task) boolean
toString() String
empty boolean
PFCFS
PFCFS()
update(int, int) void
reportToSuperior() boolean
size() int
notifyToInferior() boolean
add(Task) boolean
rebase() boolean
toString() String
shuffle() void
empty boolean
PPRR
PPRR()
quantum int
update(int, int) void
rebase() boolean
add(Task) boolean
shuffle() void
reportToSuperior() boolean
toString() String
notifyToInferior() boolean
size() int
quantum int
empty boolean
PMFQB
PMFQB()
update(int, int) void
notifyToInferior() boolean
nextBusyIndex() int
size() int
add(Task) boolean
rebase() boolean
shuffle() void
toString() String
reportToSuperior() boolean
empty boolean

图 5: 进程调度层

```
3       if (inferior == null)
4           return false;
5
6       return inferior.reportToSuperior();
7   }
```

此外，进程调度的大体框架如下：

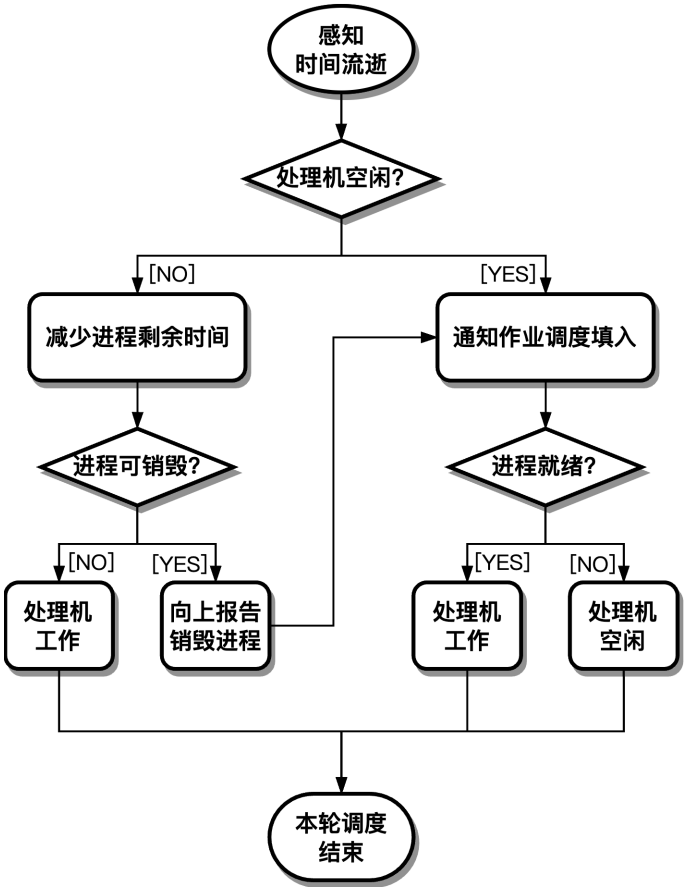


图 6: 进程调度（以先来先服务为例）

不过，不同算法实现细节有所差别，下面结合代码进行分析。这里采用改进通信机制的方式避免延时问题，即，进程调度后，若向上提交销毁进程导致内存存在空余，则通知作业调度并允许其提交新的进程。

Listing 8: 先来先服务

```
1  private boolean leisure = true;
2  private PriorityQueue<Task> queue = new PriorityQueue
       <>(Comparator.comparingInt(Task::
       getProcessArriveTime));
3
4  @Override
5  public void update(int currentTime, int elapseUnit) {
6      shuffle();
7
8      // existing a running process
9      if (!leisure) {
10         queue.peek().decrementLeftTime(currentTime,
               elapseUnit);
11         if (queue.peek().isTerminated()) {
12             reportToSuperior();
13             leisure = true;
14         } else {
15             leisure = false;
16         }
17     }
18
19     // needy for a ready process
20     if (leisure) {
21         notifyToInferior();
22         if (queue.peek() != null) {
23             queue.peek().toggleRunning(currentTime);
24             leisure = false;
25         }
26     }
27 }
```

基于动态优先级的时间片轮转要求对时间片动态判定修改。此外，程序没有记录运行进程的引用，所以必须通过 tricks 保证运行进程优先级永远最高。

Listing 9: Getter（以优先级为例）

```
public int getPriority() {
    int factor = (isRunning()) ? (-1) : (1);
    return priority * factor;
}
```

Listing 10: 基于动态优先级的时间片轮转

```
private boolean leisure = true;
private int quantum = 2;
private int dynamicTimeSlice = quantum;
private PriorityQueue<Task> queue = new PriorityQueue
    <>(new Comparator<Task>() {
    @Override
    public int compare(Task o1, Task o2) {
        return (o1.getPriority() - o2.getPriority() !=
            0) ? (o1.getPriority() - o2.getPriority())
            : (o1.getJobSubmitTime() - o2.
            getJobSubmitTime());
        // return o1.getPriority() - o2.getPriority();
    }
});

public void setQuantum(int quantum) {
    this.quantum = quantum;
}

public void update(int currentTime, int elapseUnit) {
    shuffle();

    // existing a running process within the last
        minute
    if (!leisure) {
        queue.peek().decrementLeftTime(currentTime,
```

```
            elapseUnit);
22          dynamicTimeSlice -= elapseUnit;
23          // terminated
24          if (queue.peek().isTerminated()) {
25              queue.peek().incrementPriority(2);
26              reportToSuperior();
27              // reset time slice
28              leisure = true;
29          } else if (queue.peek().isRunning()) {
30              // brute-force offline
31              if (dynamicTimeSlice == 0) {
32                  queue.peek().incrementPriority(2);
33                  queue.peek().toggleReReady();
34                  shuffle();
35                  leisure = true;
36              }
37              // slice is still adequate for continuous
                    running
38              else {
39                  leisure = false;
40              }
41          }
42      }
43
44      // needy for a ready process
45      if (leisure) {
46          dynamicTimeSlice = quantum;
47          notifyToInferior();
48          if (queue.peek() != null) {
49              queue.peek().toggleRunning(currentTime);
50              leisure = false;
51          }
52      }
```

```
53
54 }
```

多级反馈队列法：设置多级就绪队列，各级队列优先级从高到低，时间片由小到大。新进程到达时，先进入第一级队列，按“先来先服务”原则排队等待。若时间片用完进程仍未结束，则将该进程移入下级队列队尾（直至最后一级）。只有第 k 级队列为空，才会为 $k+1$ 级队头进程分配时间片。

Listing 11: 多级反馈队列（数据结构）

```
1  private List<Deque<Task>> queues = new ArrayList<>();
2  private boolean leisure = true;
3  private int busyIndex = 0;
4  private int quantum = 1;
5  private int dynamicTimeSlice = quantum;
6
7  public PMFQ() {
8      Deque<Task> top = new ArrayDeque<>();
9      queues.add(top);
10     Deque<Task> mid = new ArrayDeque<>();
11     queues.add(mid);
12     Deque<Task> btm = new ArrayDeque<>();
13     queues.add(btm);
```

多级反馈队列法同样可分为抢占式和非抢占式，这两种实现的主要区别在于“调度时机”。假设当前运行进程优先级并非顺位第一，即使时间片未运行完，抢占式调度也会收回剩余时间片使优先级更高的新进程获得处理机资源，并将被抢占处理机的进程重新放回原队列队尾。而非抢占式调度须要等时间片轮转才可以调度。

Listing 12: 多级反馈队列（非抢占式调度）

```
1  @Override
2  public void update(int currentTime, int elapseUnit) {
3      // existing a running process within the last
       minute
4      if (!leisure) {
```

```
5           Deque<Task> queue = queues.get(busyIndex);
6           queue.peek().decrementLeftTime(currentTime,
                elapseUnit);
7           dynamicTimeSlice -= elapseUnit;
8           // terminated
9           if (queue.peek().isTerminated()) {
10              reportToSuperior();
11              // reset time slice
12              leisure = true;
13          } else if (queue.peek().isRunning()) {
14              // brute-force offline
15              if (dynamicTimeSlice == 0) {
16                  queue.peek().toggleReReady();
17                  int nextIndex = Math.min(busyIndex +
                        1, 2);
18                  // next level or bottom
19                  queues.get(nextIndex).offer(queue.poll
                        ());
20                  leisure = true;
21              }
22              // slice is still adequate for continuous
                    running
23              else {
24                  leisure = false;
25              }
26          }
27      }
28
29      // needy for a ready process
30      if (leisure) {
31          notifyToInferior();
32          // somebody ready
33          if (nextBusyIndex() != -1) {
```

```
34                busyIndex = nextBusyIndex();
35                quantum = (int) Math.pow(2, busyIndex);
36                dynamicTimeSlice = quantum;
37
38                Deque<Task> queue = queues.get(busyIndex);
39                queue.peek().toggleRunning(currentTime);
40                leisure = false;
41            }
42        }
43    }
```

Listing 13: 多级反馈队列（抢占式调度）

```
1   @Override
2   public void update(int currentTime, int elapseUnit) {
3       // existing a running process within the last
            minute
4       if (!leisure) {
5           Deque<Task> queue = queues.get(busyIndex);
6           queue.peek().decrementLeftTime(currentTime,
                elapseUnit);
7           dynamicTimeSlice -= elapseUnit;
8           // terminated
9           if (queue.peek().isTerminated()) {
10              reportToSuperior();
11              // reset time slice
12              leisure = true;
13          } else if (queue.peek().isRunning()) {
14              // brute-force offline
15              if (dynamicTimeSlice == 0) {
16                  queue.peek().toggleReReady();
17                  int nextIndex = Math.min(busyIndex +
                        1, 2);
18                  // next level or bottom
```

```
19                  queues.get(nextIndex).offer(queue.poll
                        ());
20                  leisure = true;
21              }
22              // slice is still adequate for continuous
                    running
23              else {
24                  // nobody can forcibly occupy
25                  if (nextBusyIndex() == busyIndex) {
26                      leisure = false;
27                  } else {
28                      // FIXME
29                      queue.peek().toggleReReady();
30                      queue.offer(queue.poll());
31                      leisure = true;
32                  }
33              }
34          }
35      }
36
37      if (leisure) {
38          notifyToInferior();
39          if (nextBusyIndex() != -1) {
40              busyIndex = nextBusyIndex();
41              quantum = (int) Math.pow(2, busyIndex);
42              dynamicTimeSlice = quantum;
43
44              Deque<Task> queue = queues.get(busyIndex);
45              queue.peek().toggleRunning(currentTime);
46              leisure = false;
47          }
48      }
49  }
```

3.2 边缘实现

选择使用 Android 移动平台部署应用，以省去云服务器环境搭建。

3.2.1 依赖配置

```
android {
    namespace 'edu.wust.durui'
    compileSdk 32
    defaultConfig {
        vectorDrawables.useSupportLibrary = true
        multiDexEnabled true
        applicationId "edu.wust.durui"
        minSdk 29
        ...
    }
    ...
    viewBinding {
        enabled = true
    }
    dataBinding {
        enabled = true
    }
}
dependencies {
    ...
    implementation 'com.google.android.material:material:1.6.1'
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.4.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1"
    implementation "androidx.multidex:multidex:2.0.1"
    implementation "androidx.fragment:fragment:1.5.5"
    implementation 'com.github.blackfizz:eazegraph:1.2.2@aar'
    implementation 'com.nineoldandroids:library:2.4.0'
    implementation 'com.github.vipulasri:timelineview:1.1.5'
}
```

3.2.2 架构设计

谷歌在安卓开发者文档中强调，“应用架构定义了应用的各个部分之间的界限以及每个部分应承担的职责”，推荐按照下列原则设计应用架构：

1. 关注点分离。基于界面的类（Activity 或 Fragment）只应该包含界面和交互。这些类应尽可能保持精简，以避免产生与组件生命周期相关的问题，并提高可测试性。毕竟，操作系统可能会根据内存不足等系统条件随时销毁它们，所以，最好避免在 Activity 或 Fragment 编写所有代码。

2. 数据驱动界面。数据模型应独立于界面或其他组件。持久性模型是理想之选，因为如果 Android 操作系统销毁应用以释放资源，用户也不会丢失数据。

3. 单一数据源和单向数据流（Unidirectional Data Flow）。一定条件下，只有唯一的数据所有者（Signle Source Of Truth）可以修改数据。在离线优先应用中，应用数据的单一数据源通常是数据库。

基于上述架构原则，每个应用应至少有两个层：界面层和数据层。

界面层 负责展示数据，更新数据，一般对应 Activity 或 Fragment 类。

数据层 包含业务逻辑，包含应用创建、存储和更改数据的规则，向应用的其余部分公开数据，一般对应一个数据类。

网域层 位于界面与数据层之间的可选层，负责封装复杂的业务逻辑，一般该层中的类通常称为用例（UseCase），每个用例都应仅负责单个功能。

关系到核心架构的两个关键概念是“MVVM”和“MutableLiveData”。

MVVM MVVM（Model-View-ViewModel）是一种应用架构，它将应用程序分为三个部分：Model（模型层），View（视图层）和 ViewModel（视图模型层）。View 仅负责显示数据并接收用户输入，ViewModel 仅负责处理业务逻辑并将数据从 Model 传递到 View。

MutableLiveData LiveData 是生命周期感知型组件，它只会在组件处于激活状态时才会回调相应的方法，从而刷新界面。从软件设计模式的角度说，它原生支持*观察者模式*，能够在数据发生变化时通知观察者（Activity 或 Fragment 等）进行相应的处理。

3.2.3 核心技术

Fragment Fragment 组件介于 Activity 和 View：一方面，它未继承 Context，无法独立存在，必须作为宿主视图的一部分；可另一方面，它又可以定义管理自己的布局，具有自己的生命周期（Lifecycle）。

RecyclerView 核心视图组件之一，用于显示大量数据。

DatePicker 核心视图组件之一。日期选择控件的主要功能是向用户提供包含年、月、日的日期数据并允许用户对其修改。

EazeGraph 核心视图组件之一。轻量级的图表库[3]，支持生成四种图表：条形图，层叠柱状图，饼状图，折线图。

TimelineView 核心视图组件之一。轻量级时间视图库[4]，支持呈现时间线，应用广泛，如物流追踪历史节点等。

3.3 原型设计

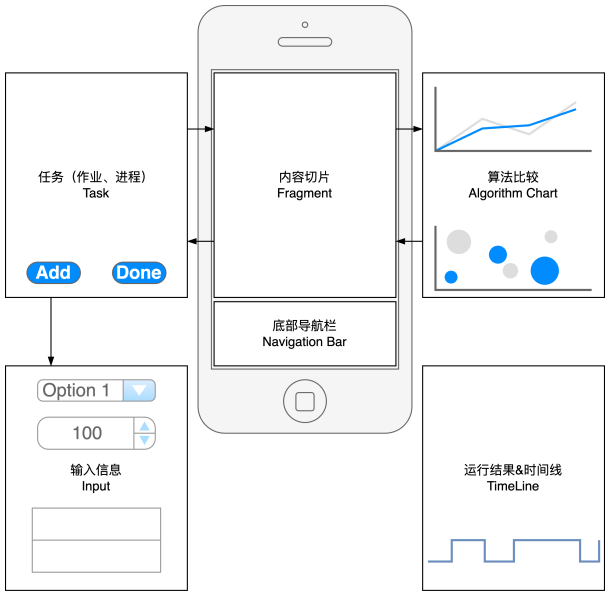


图 7: 界面原型设计图

[3]https://github.com/paulroehr/EazeGraph

[4]https://github.com/vipulasri/Timeline-View

4 系统测试

4.1 测试用例

表 4: 测试用例（短作业优先，平均周转时间 1.550h）

作业	进入时刻	运行时间（小时）	开始时刻	完成时刻	周转时间（小时）
1	08:00	2.00	08:00	10:00	2.00
2	08:30	0.50	10:18	10:48	2.30
3	09:00	0.10	10:00	10:06	1.10
4	09:30	0.20	10:06	10:18	0.80

表 5: 测试用例（高响应比优先，平均周转时间 1.625h）

作业	进入时刻	运行时间（小时）	开始时刻	完成时刻	周转时间（小时）
1	08:00	2.00	08:00	10:00	2.00
2	08:30	0.50	10:06	10:36	2.10
3	09:00	0.10	10:00	10:06	1.10
4	09:30	0.20	10:36	10:48	1.30

表 6: 测试用例（248 抢占式多级反馈队列）

作业	进入时刻	服务时间	完成时刻	周转时间
A	0	7	11	11
B	5	4	19	14
C	7	13	30	23
D	12	9	33	21

4.2 运行结果

见尾页。

5 设计总结

宏观而言，软件设计是针对软件的功能需求、可维护性和可复用性。例如，如果客户要求实现基于“短作业优先”的作业调度和基于“（抢占式）多级反馈队列”的两道批处理系统，本系统及其容易扩展。

微观而言，以下设计模式我深有体会：

观察者模式 定义了一种一对多的依赖关系，让多个观察者对象同时监听（订阅）某一个主题对象，当主题对象发生变化时，它的所有依赖者（观察者）都会自动收到通知并更新。

在这种情形下，我们认为观察者是平等的，更新顺序是固定的。在本系统设计中，倘若仅仅使用本模式，将导致延时。

职责链模式 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿该链传递处理请求。

作业调度是高级调度，负责辅存与内存间的调度；进程调度是低级调度，负责内存与处理器间的调度。两者天然具有职责链关系。

为了避免“延时”问题，我引入了“双向职责链模式”，即作业调度不止可以向进程调度提交作业，进程调度也可以通知作业调度允许提交作业。

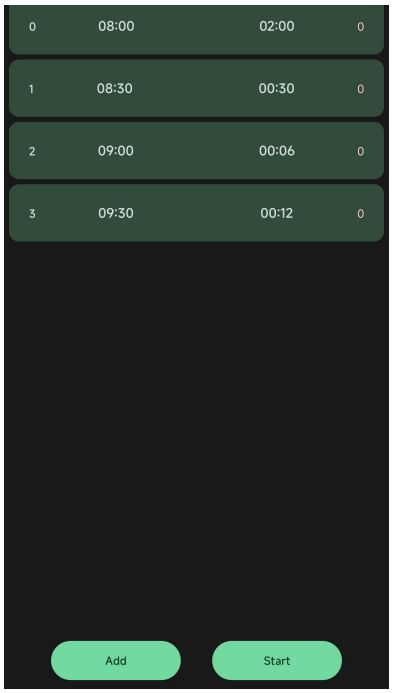
工厂方法模式 定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类。

机器多种多样，如果要进行各算法间的比较，就要创建多种机器。本系统使用工厂方法，实现创建与使用的结耦。

原型模式 拷贝这些原型创建新的对象。

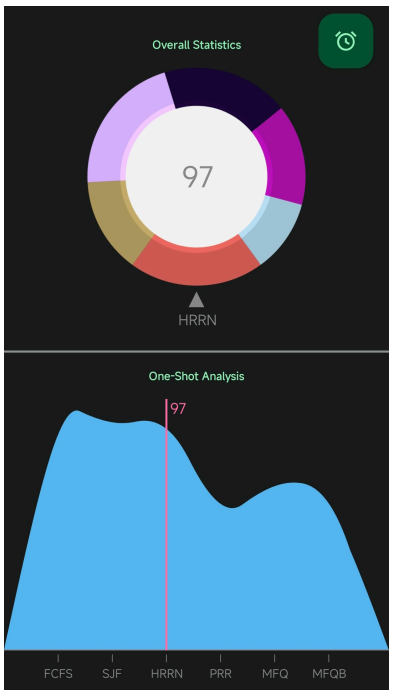
对于各算法性能分析而言，各个“处理机”，由于模拟时减少任务的剩余运行时间，应仅仅保有用户提交序列的副本。即通过拷贝一个已经存在的实例来返回新的实例，而不是新建实例。

总之，在使用“软件设计模式”时应该活学活用，切忌生搬硬套。最终目的都是让程序低耦合、高复用、高内聚、易扩展、易维护。

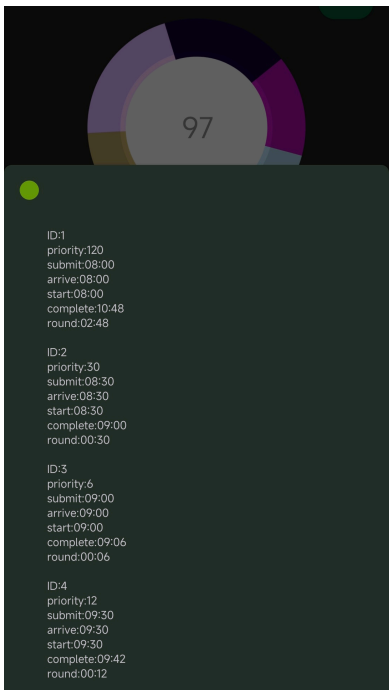


(a) 任务列表

(b) 时刻输入



(c) 性能比较



(d) 时间线

图 8: 运行截图