

**2021年春季学期  
计算学部《软件构造》课程**

**Lab 3实验报告**

|  |  |
| --- | --- |
| 姓名 | 林搏海 |
| 学号 | 1190202128 |
| 班号 | 1903002 |
| 电子邮件 | 1172798125@qq.com |
| 手机号码 | 13452053282 |

**目录**

[1 实验目标概述 1](#_Toc76287764)

[2 实验环境配置 1](#_Toc76287765)

[3 实验过程 1](#_Toc76287766)

[3.1 待开发的三个应用场景 1](#_Toc76287767)

[3.2 面向可复用性和可维护性的设计：IntervalSet<L> 2](#_Toc76287768)

[3.2.1 IntervalSet<L>的共性操作 2](#_Toc76287769)

[3.2.2 局部共性特征的设计方案 2](#_Toc76287770)

[构造测试代码如下： 4](#_Toc76287771)

[3.2.3 面向各应用的IntervalSet子类型设计（个性化特征的设计方案） 5](#_Toc76287772)

[3.3 面向可复用性和可维护性的设计：MultiIntervalSet<L> 10](#_Toc76287773)

[3.3.1 MultiIntervalSet<L>的共性操作 10](#_Toc76287774)

[3.3.2 局部共性特征的设计方案 10](#_Toc76287775)

[3.3.3 面向各应用的MultiIntervalSet子类型设计（个性化特征的设计方案） 12](#_Toc76287776)

[3.4 面向复用的设计：L 14](#_Toc76287777)

[3.5 可复用API设计 24](#_Toc76287778)

[3.5.1 计算相似度 24](#_Toc76287779)

[3.5.2 计算时间冲突比例 24](#_Toc76287780)

[3.5.3 计算空闲时间比例 25](#_Toc76287781)

[3.6 应用设计与开发 25](#_Toc76287782)

[3.6.1 排班管理系统 25](#_Toc76287783)

[3.6.2 操作系统的进程调度管理系统 26](#_Toc76287784)

[3.6.3 课表管理系统 27](#_Toc76287785)

[3.7 基于语法的数据读入 28](#_Toc76287786)

[3.8 应对面临的新变化 29](#_Toc76287787)

[3.8.1 变化1 29](#_Toc76287788)

[3.8.2 变化2 29](#_Toc76287789)

[3.9 Git仓库结构 30](#_Toc76287790)

[4 实验进度记录 30](#_Toc76287791)

[5 实验过程中遇到的困难与解决途径 30](#_Toc76287792)

[6 实验过程中收获的经验、教训、感想 30](#_Toc76287793)

[6.1 实验过程中收获的经验和教训 30](#_Toc76287794)

[6.2 针对以下方面的感受 30](#_Toc76287795)

# 实验目标概述

本实验需要设计并实现抽象数据类型 IntervalSet，对现实中各类“在特定 时间段进行的特定任务”进行抽象，并在三个具体应用中使用它。 在设计该 ADT 时，需要对其进行泛型化，从而使其抽象能力更强、适应现 实中不同情况需求的能力更强。

# 实验环境配置

在这里给出你的GitHub Lab3仓库的URL地址（Lab3-学号）。

https://github.com/ComputerScienceHIT/Lab3-HIT-1190202128

# 实验过程

请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## 待开发的三个应用场景

三个应用场景具有一定的共性：他们都是针对一段时间轴上的不同时间段的信息记录，我们可以简单地设想他们都可以通过一个线性数据结构记录不同时间段的不同事件。

而这三个场景也存在这一定的差异。

首先，第一个场景需要我们在完成了时间段分配之后不能出现时间间隙，因此需要我们对其进行检测，此外，还需要注意的事每一个员工只能被分配在一个时间段，不同时间段时间不能发生重叠。

第二个场景中，则允许存在空隙，而且针对同一个进程，我们也允许其出现在不同的时间段，其限制条件相对于第一个场景有所放宽。

在第三个场景中，不仅允许时间段发生重叠，即两个不同的课程共同使用一个时间段，而且还加入了一个新的时间限制：可重复的时间段。因此，我们需要构思新的思路去设计第三个ADT，而且，我们很容易针对其重复时间段的特点想到：我们可以仅仅存储一个时间段即可，这样也提高了运行效率，降低了实现难度。

## 面向可复用性和可维护性的设计：IntervalSet<L>

在本人的构思中，我实现了一个抽象接口IntervalSet<L>，并且实现了CommonIntervalSet和MultiIntervalSet两个具体类，通过在两个具体的实现类中加入不同的属性和方法，达成了共性和特性的统一。

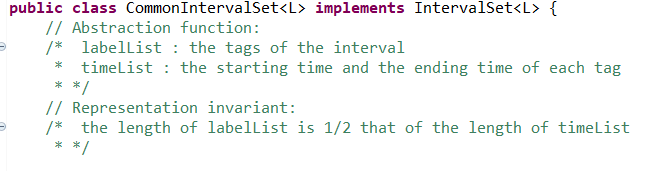
### IntervalSet<L>的共性操作

|  |  |
| --- | --- |
| 函数声明 | 功能概述 |
| **public** **void** insert​(**long** start, **long** end, L label); | 插入一个时间段，如果时间输入不合法或者与已存在时间段重叠，就不修改然后返回 |
| **public** Set<L> labels(); | 返回标签集合 |
| **public** **boolean** remove​(L label); | 移除一个标签，如果标签存在则返回true，否则返回false |
| **public** **long** start​(L label); | 返回当前标签的起始时间 |
| **public** **long** end(L label); | 返回当前标签的结束时间 |

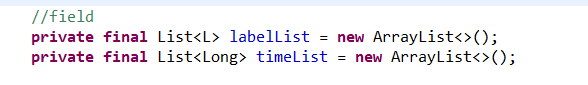
### 局部共性特征的设计方案

**CommonIntervalSet**

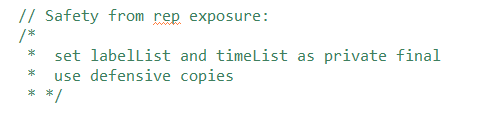
首先我们可以给出这个局部共性类的AF和RI



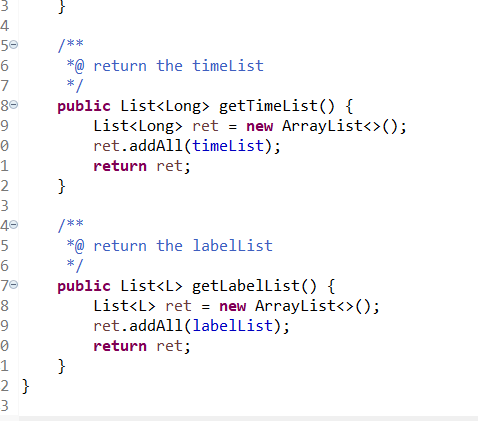
以及该类的数据结构：



然后给出防止数据泄漏的方法：



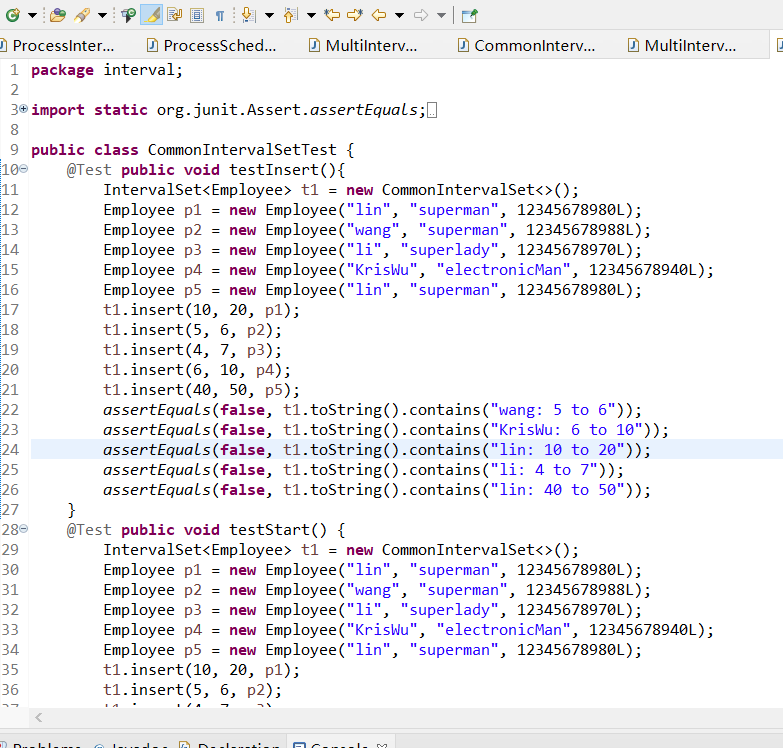
其中，关于防御性拷贝的具体事例如下：



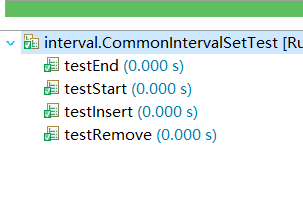
在此基础上，我们可以给出该类的一些重写的接口继承方法和新加入的方法：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** **void** insert​(**long** start, **long** end, L label); | 插入一个时间段，如果时间输入不合法或者与已存在时间段重叠，就不修改然后返回 | 是 |
| **public** Set<L> labels(); | 返回标签集合 | 否 |
| **public** **oolean** remove​(L label); | 移除一个标签，如果标签存在则返回true，否则返回false | 是 |
| **public** **long** start​(L label); | 返回当前标签的起始时间 | 否 |
| **public** **long** end(L label); | 返回当前标签的结束时间 | 否 |
| **public** **int** getAssignedLength() | 返回已经分配tags的时间长度 | 否 |
| **public** **void** clear() | 将list中的数据清空 | 是 |
| **public** List<Long> getTimeList() | 返回分配时间点列表 | 否 |
| **public** List<L> getLabelList() | 返回标签列表 | 否 |

### 构造测试代码如下：



代码执行情况如下：



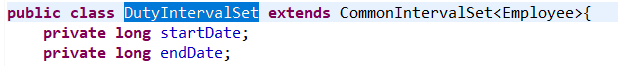
代码覆盖率如下：



### 面向各应用的IntervalSet子类型设计（个性化特征的设计方案）

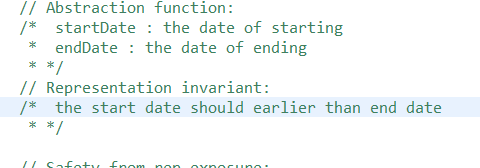
**DutyIntervalSet**

首先，我们给出这个子类的数据结构：

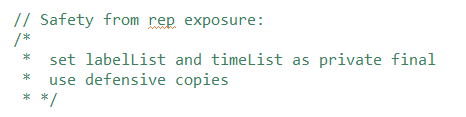


由于这个类继承了之前实现的子类CommonIntervalSet，因此，我们只需要增加两个数据用于记录开始和结束时间即可。

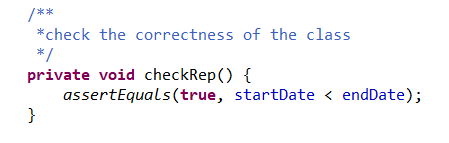
接着，我们可以给出函数的AF和RI



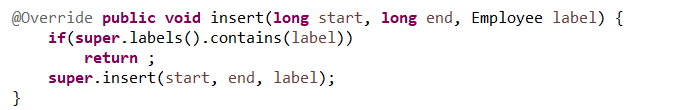
然后我们给出防止数据泄漏的方法：



给出checkrep函数的实现：



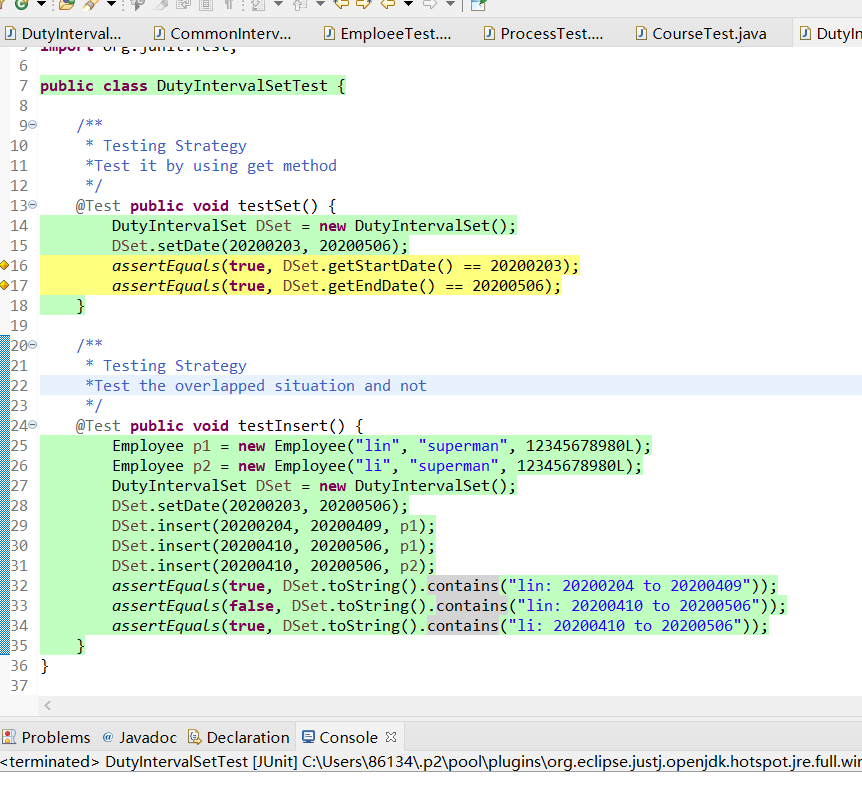
需要注意的是，我们在重写insert方法的时候使用了父进程的方法：



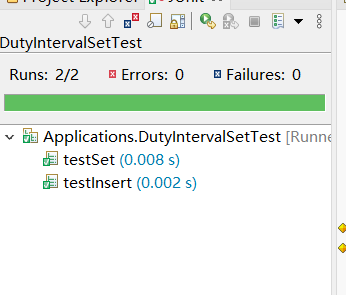
接着我们可以给出该类新增或者是重写的一些方法：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** **void** setDate(**long** start, **long** end) | 设置起止日期 | 是 |
| **public** **void** autoAssign(Set<Employee> waitingSet) | 自动分配时间 | 否 |
| **public** **double** calculateAssignedRate() | 计算分配百分比 | 否 |
| **public** **long** getStartDate () | 返回开始日期 | 否 |
| **public** **long** getEndDate () | 返回结束日期 | 否 |
| @Override **public** **void** insert​(**long** start, **long** end, Employee label) | 插入时间段 | 否 |
| @Override **public** String toString() | 返回输出相关信息的字符串 | 否 |

测试代码如下：



测试结果如下：

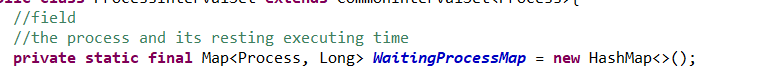


代码覆盖率如下：

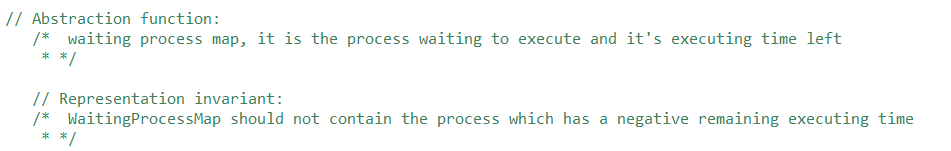


**ProcessIntervalSet**

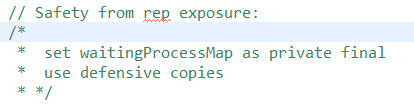
首先，我们给出其数据结构：



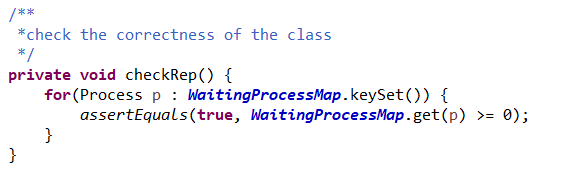
然后，我们给出对应的AF和RI等信息：



给出防止数据泄漏的策略：



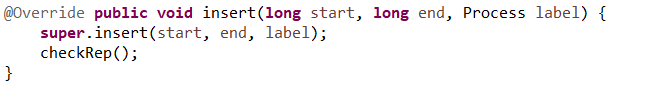
给出checkRep函数：



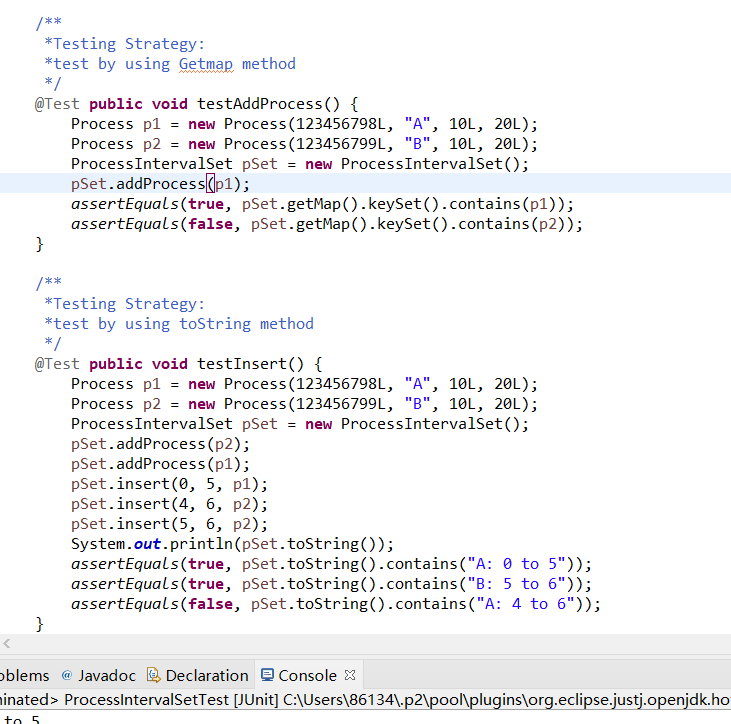
然后我们可以给出关于其中的用到的方法的声明：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** **void** addProcess(Process a) | 将进程a加入进程集合中 | 是 |
| @Override **public** **void** insert​(**long** start, **long** end, Process label) | 设置进程的时间段 | 否 |
| **public** Map<Process, Long> getMap() | 返回进程集合的地图 | 否 |
| **public** **boolean** isAllTerminated() | 是否所有进程都结束 | 否 |
| **public** **void** resetMap() | 清空等待集合地图 | 是 |
| **public** **void** shortestProcessFirst() | 按照最短进程原则随机运行进程 | 是 |
| **public** **void** randomlyRun() | 完全随机运行进程 | 是 |
| @Override **public** String toString() | 返回包含当前时间表信息的字符串 | 否 |

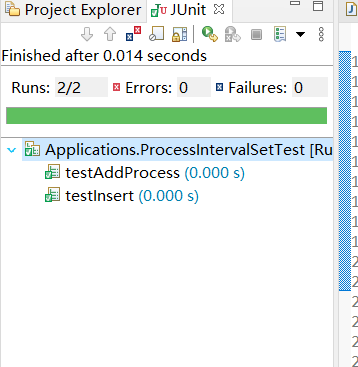
值得注意的是，我们同样在重写次insert方法的时候使用了父进程的方法：



测试代码如下：



测试结果如下：



测试覆盖率如下：



## 面向可复用性和可维护性的设计：MultiIntervalSet<L>

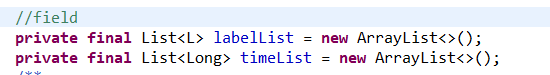
### MultiIntervalSet<L>的共性操作

我们将MultiIntervalSet作为了IntervalSet接口的实现类，因此它的共性操作即为IntervalSet的共性操作。

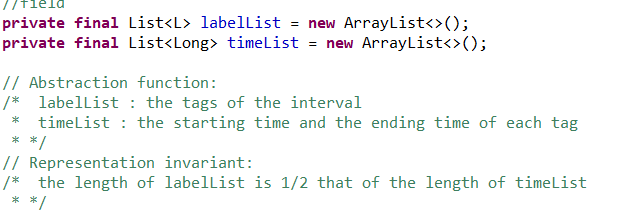
### 局部共性特征的设计方案

**MultiIntervalSet**

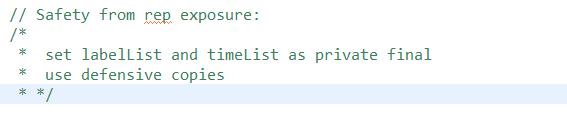
我们需要在这个类中实现重叠的时间段记录，因此我们需要对其方法和规约作出一些改变，首先，我们给出这个类的一些数据结构：



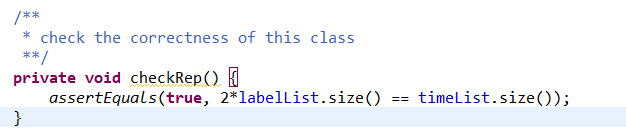
给出相应的AF和RI等信息：



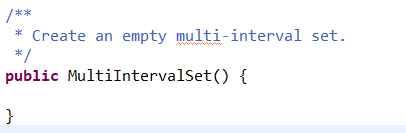
给出防止数据泄漏的方法：



给出checkRep函数：



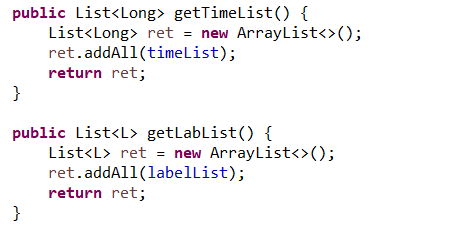
给出构造器：



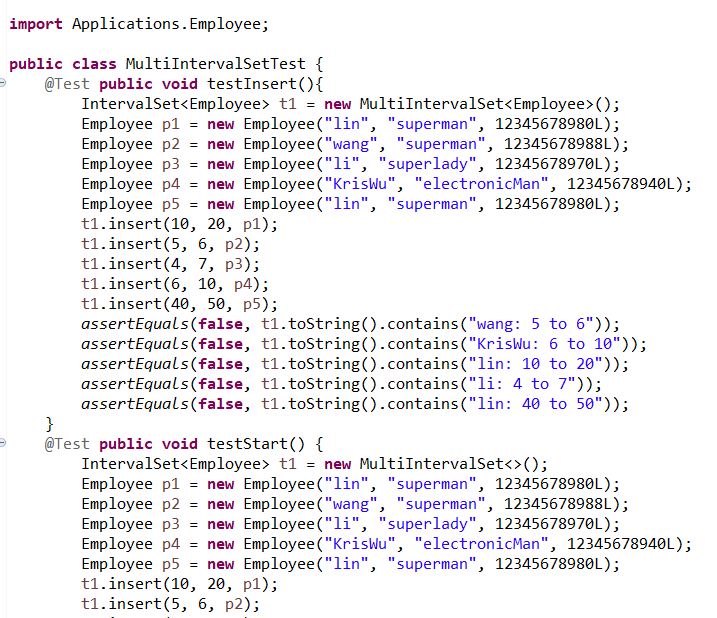
在此基础上，我们可以给出相关的函数声明以及实现概述：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| @Override **public** **void** insert​(**long** start, **long** end, L label) | 为标签设置一个时间段 | 是 |
| @Override **public** **boolean** remove​(L label) | 移除一个标签和对应时间段 | 是 |
| @Override **public** **long** start​(L label) | 返回标签的开始时间 | 否 |
| @Override **public** **long** end(L label) | 返回标签的结束时间 | 否 |
| @Override **public** String toString() | 返回关于集合的信息的字符串 | 否 |
| @Override **public** Set<L> labels() | 返回标签集合 | 否 |
| **public** List<Long> getTimeList() | 返回tineList列表 | 否 |
| **public** List<L> getLabList() | 返回labelList列表 | 否 |

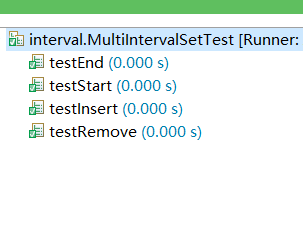
该类中关于防御性拷贝的实例如下：



代码构造如下：



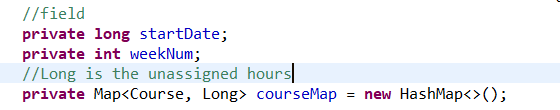
代码测试如下：



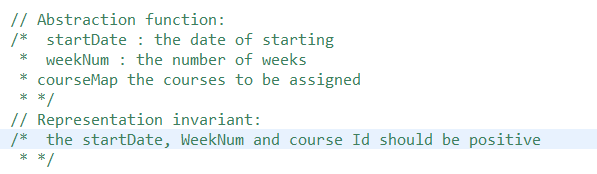
### 面向各应用的MultiIntervalSet子类型设计（个性化特征的设计方案）

**CourseIntervalSet**

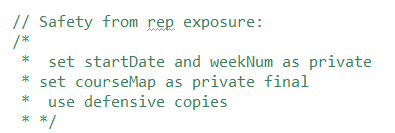
首先，我们可以给出其数据域：



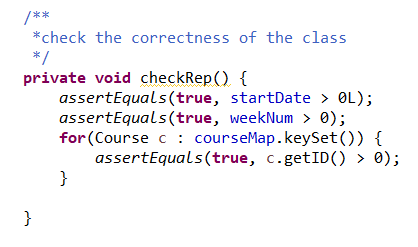
其次，我们可以给出其AF和RI等信息：



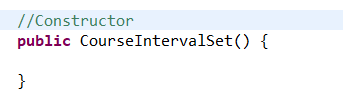
然后我们可以给出关于防止数据泄漏的策略：



给出checkRep函数的实现：



给出构造器：



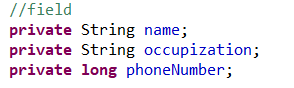
然后，在此基础上，我们可以给出该类用到的相关函数的具体声明和功能概述：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **void** init(**long** start, **int** wNum) | 初始化集合，设置开始时间和周数 | 是 |
| **public** **void** addCourse(Course c, **long** weekHours) | 加入一个课程并且记录周学习时间 | 是 |
| **public** **void** printCourseInformation() | 打印所有待分配课程以及剩余分配时间 | 否 |
| **public** **boolean** isCourseAvailable(String name) | 判断该课程是否还可以被分配时间，如果等待课程集合中有这门课且还有剩余时间则返回TRUE，否则返回false | 否 |
| **public** **void** assignCourse(String name, **long** day, **long** time) | 为一个课程分配在周几上课和巨日上课开始时间 | 是 |
| **public** **double** getBlankTimeRate() | 返回空闲时间比例 | 否 |
| **public** **double** getRepeatedTimeRate() | 返回重叠时间比例 | 否 |
| **private** **long** getRepeatedTimeLength() | 返回重叠时间长度 | 否 |
| **public** **void** printDateCourses(**long** date) | 打印当天课程的信息 | 否 |

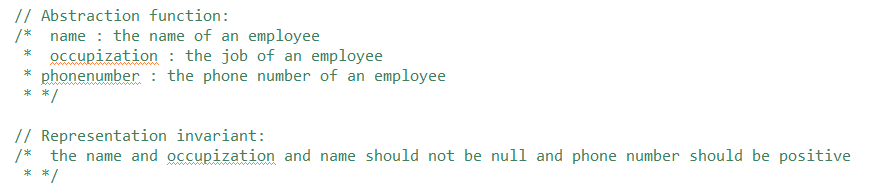
## 面向复用的设计：L

**Employee**

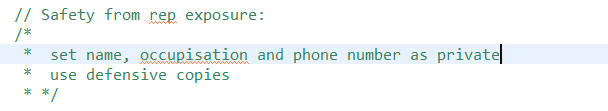
我们可以给出其数据域：



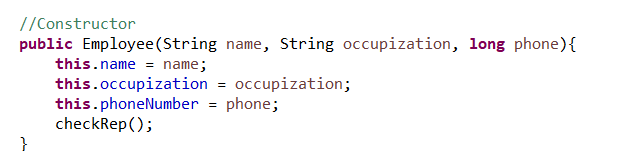
然后我们可以给出其具体的AF和RI等信息：



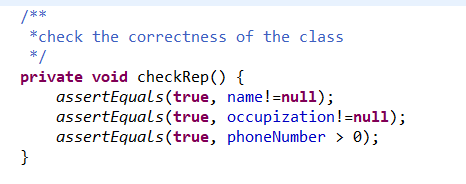
然后我们可以给出防止数据泄漏的策略：



给出构造器的实现：



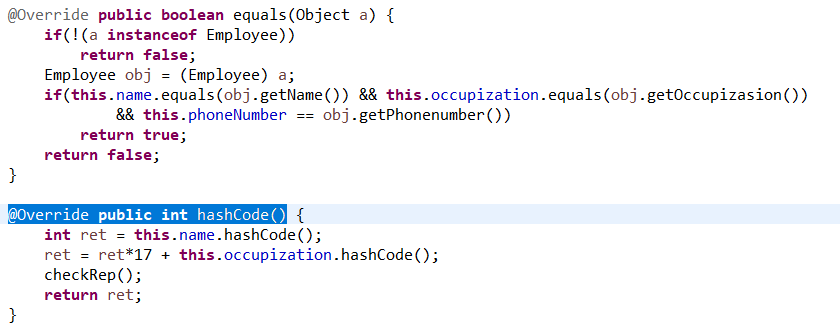
给出关于checkRep的具体实现：



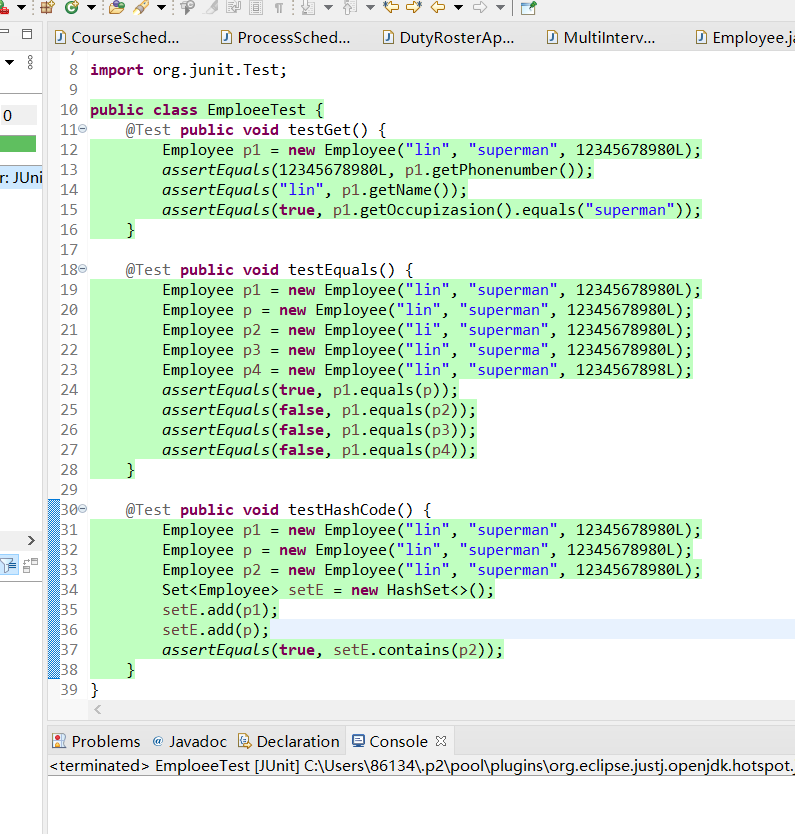
在此基础上，我们可以给出关于该类的所有方法的声明和功能简介：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** Employee(String name, String occupization, **long** phone) | 构造函数，构造一个新的Employee类的成员 | 是 |
| **public** String getName() | 返回姓名 | 否 |
| **public** String getOccupizasion() | 返回职位 | 否 |
| **public** **long** getPhonenumber() | 返回电话号码 | 否 |
| @Override **public** **boolean** equals(Object a) | 判断两个employee是否相等 | 否 |
| @Override **public** **int** hashCode() | 返回一个employee的hashcode | 否 |

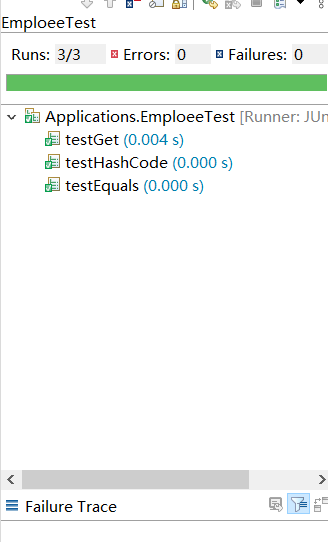
其中，为了方便判别两个employee成员的等价性，重写的equals和还是从的方法如下：



测试代码构造如下：



测试结果如下:

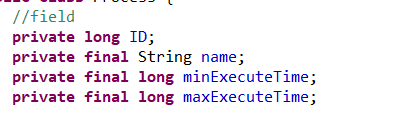


分支覆盖情况：

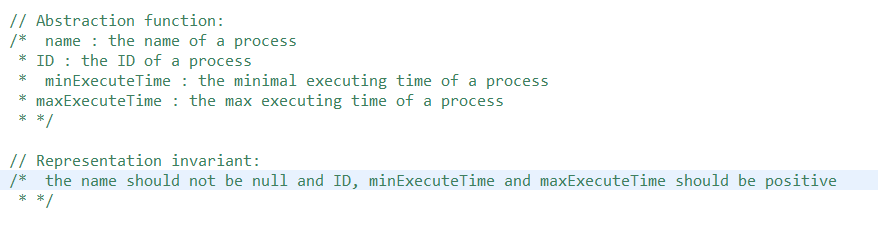


**Process**

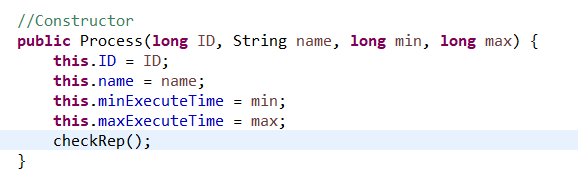
首先，我们可以给出该类的数据结构：



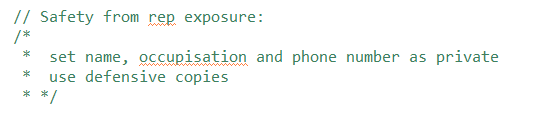
其次我们可以给出相关的AF和RI等信息：



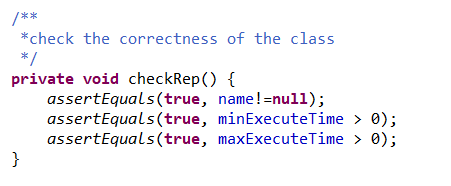
然后给出构造函数：



给出防止数据泄漏的方法：



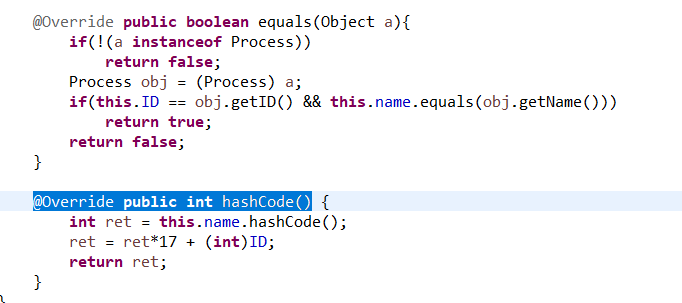
给出checkRep模块：



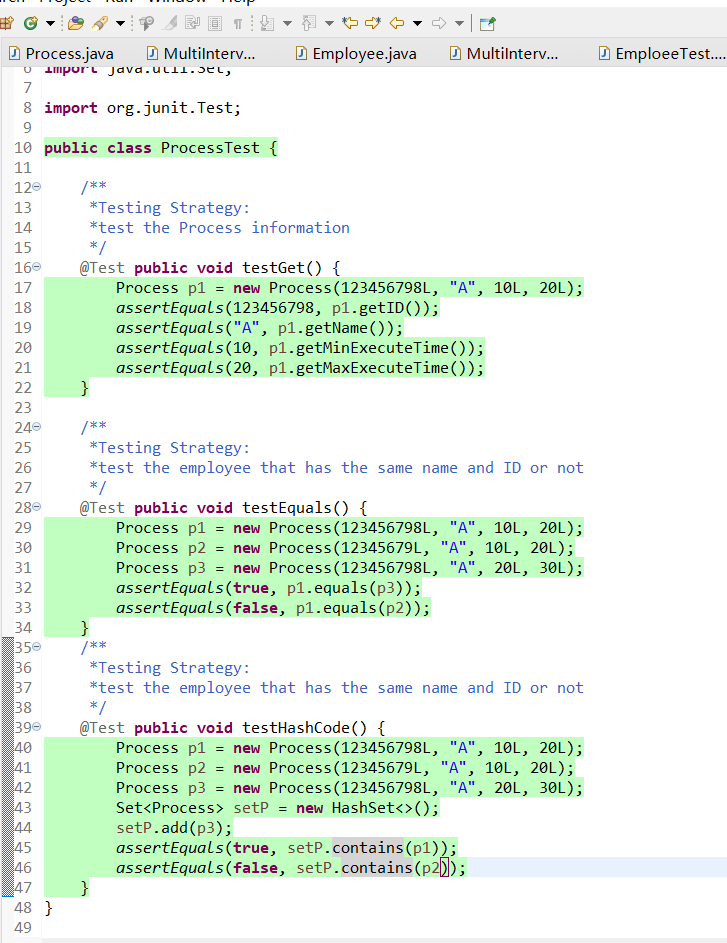
根据以上的内容，我们可以给出关于此函数的具体功能的简介和声明：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** Process(**long** ID, String name, **long** min, **long** max) | 构造函数，构造一个新的Process类的成员 | 是 |
| **public** **long** getID() | 返回进程ID | 否 |
| **public** String getName() | 返回进程名字 | 否 |
| **public** **long** getMaxExecuteTime() | 返回进程最长执行时间 | 否 |
| **public** **long** getMinExecuteTime() | 返回进程最短执行时间 | 否 |
| @Override **public** **boolean** equals(Object a) | 判断2个进程是否相等 | 否 |
| @Override **public** **int** hashCode() | 返回该进程的hashcode | 否 |

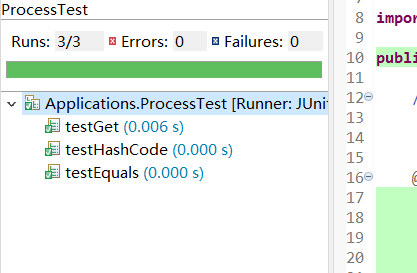
其中，关于两个重写方法的具体实现如下：



测试代码构造如下：



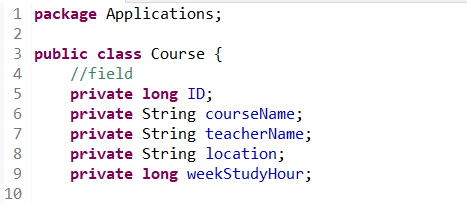
测试结果如下：



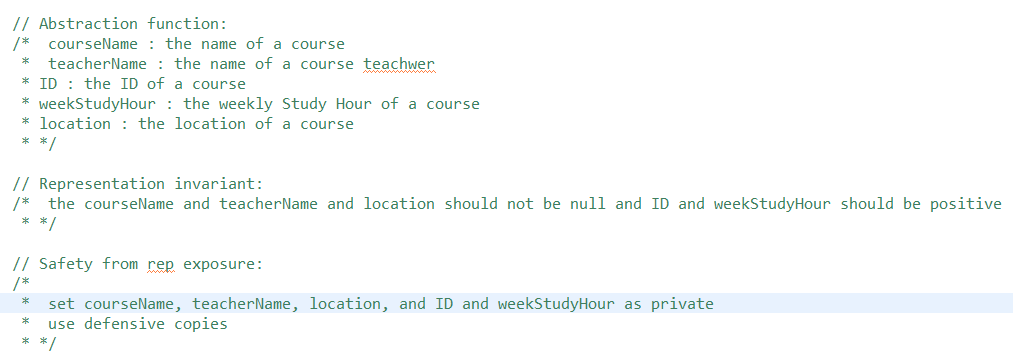
代码覆盖率如下：

**Course**

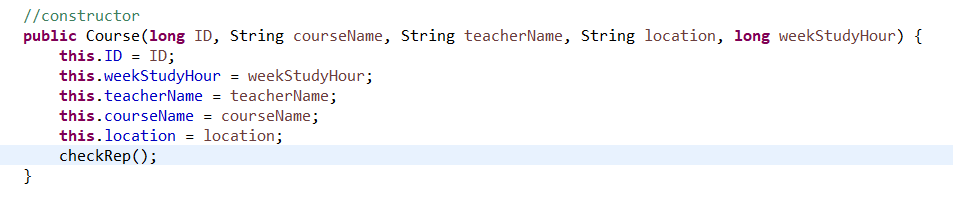
首先，我们可以给出该类的数据结构：



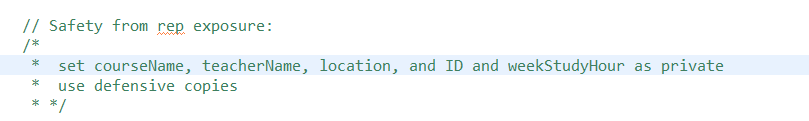
其次我们可以给出相关的AF和RI等信息：



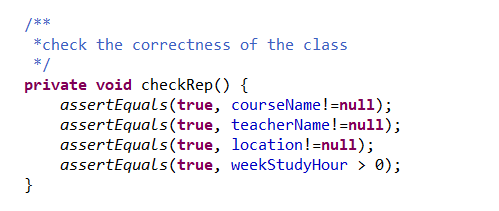
然后给出构造函数：



给出防止数据泄漏的方法：



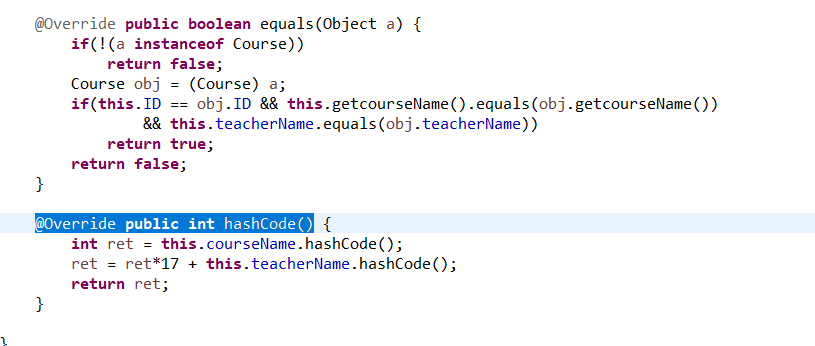
给出checkRep模块：



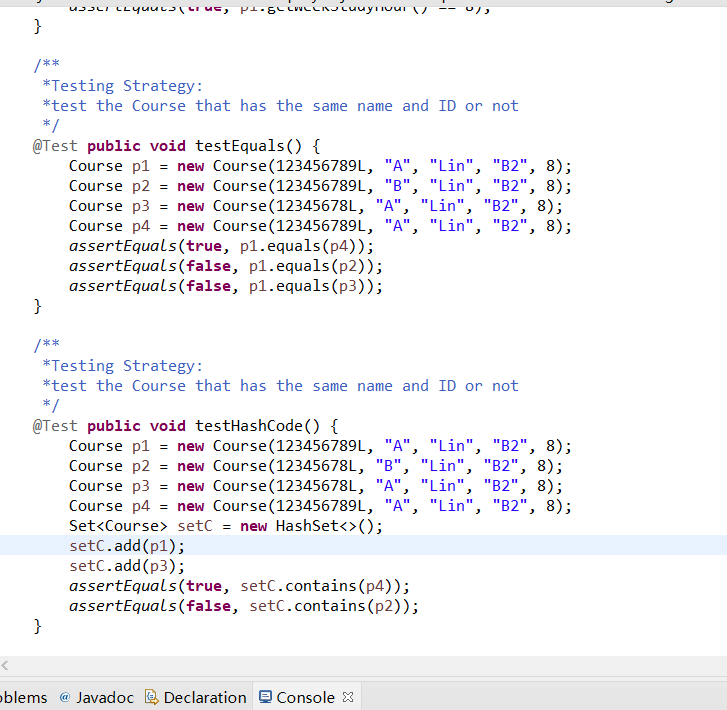
根据以上的内容，我们可以给出关于此函数的具体功能的简介和声明：

|  |  |  |
| --- | --- | --- |
| 函数声明 | 功能概述 | 是否调用checkRep |
| **public** Course(**long** ID, String courseName, String teacherName, String location, **long** weekStudyHour) | 构造函数，返回一个新的Course类的成员 | 是 |
| **public** **long** getID() | 返回课程ID | 否 |
| **public** String getcourseName() | 返回课程名字 | 否 |
| **public** String getTeacherName() | 返回授课教师姓名 | 否 |
| **public** String getLocation() | 返回课程地点 | 否 |
| **public** **long** getWeekStudyHour() | 返回课程周学习时间 | 否 |
| @Override **public** **boolean** equals(Object a) | 判断两个课程是否相等 | 否 |
| @Override **public** **int** hashCode() | 返回当前course成员的hashcode | 否 |

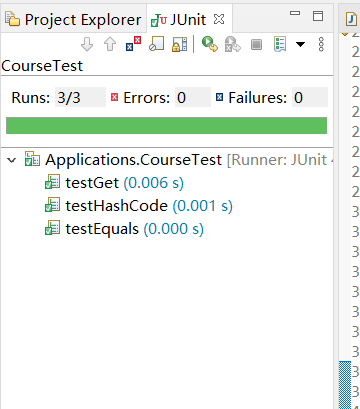
其中，关于两个重写方法的具体实现如下：



测试代码构造如下：



测试结果如下：



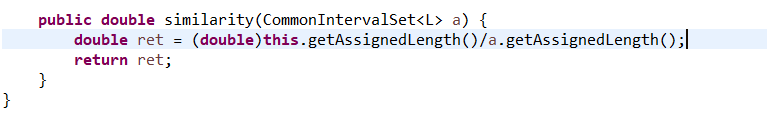
代码覆盖率如下：



## 可复用API设计

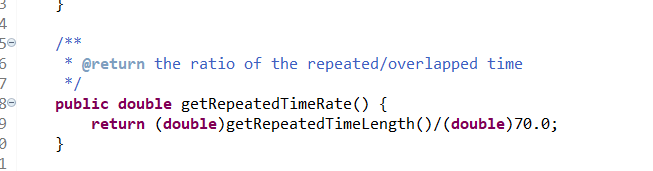
### 计算相似度

根据之前的数据结构很容易给出对应的方法：

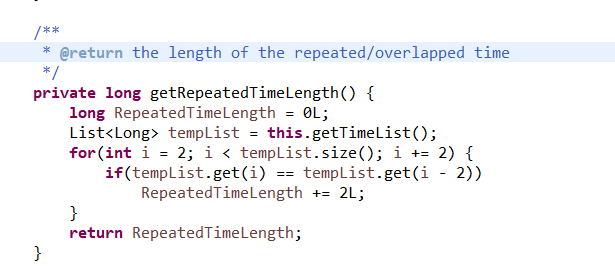


### 计算时间冲突比例

**getRepeatedTimeRate**

根据之前声明的一系列方法，可以完成该方法的设计如下：  


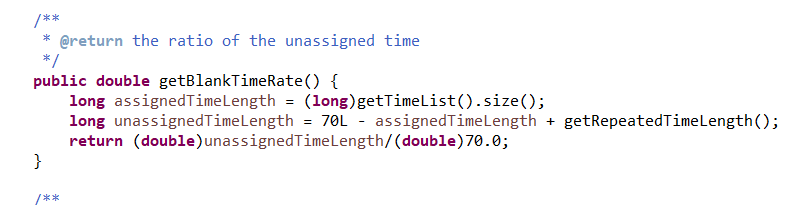
其中getrepeatedtimelength函数实现如下：



### 计算空闲时间比例

**getBlankTimeRate()**

其中具体实现细节如下：



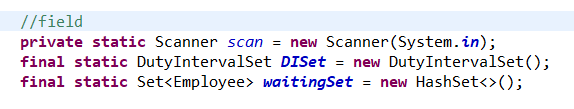
## 应用设计与开发

利用上述设计和实现的ADT，实现手册里要求的各项功能。

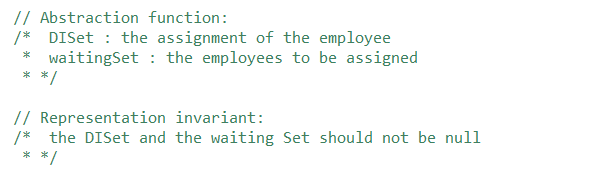
### 排班管理系统

**public class DutyRosterApp**

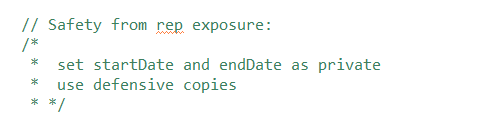
首先，我们可以给出该类的数据结构：



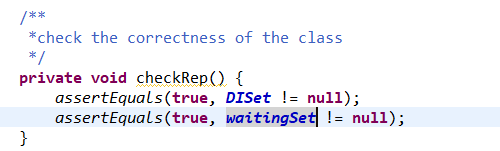
其次我们可以给出相关的AF和RI等信息：



给出防止数据泄漏的方法：



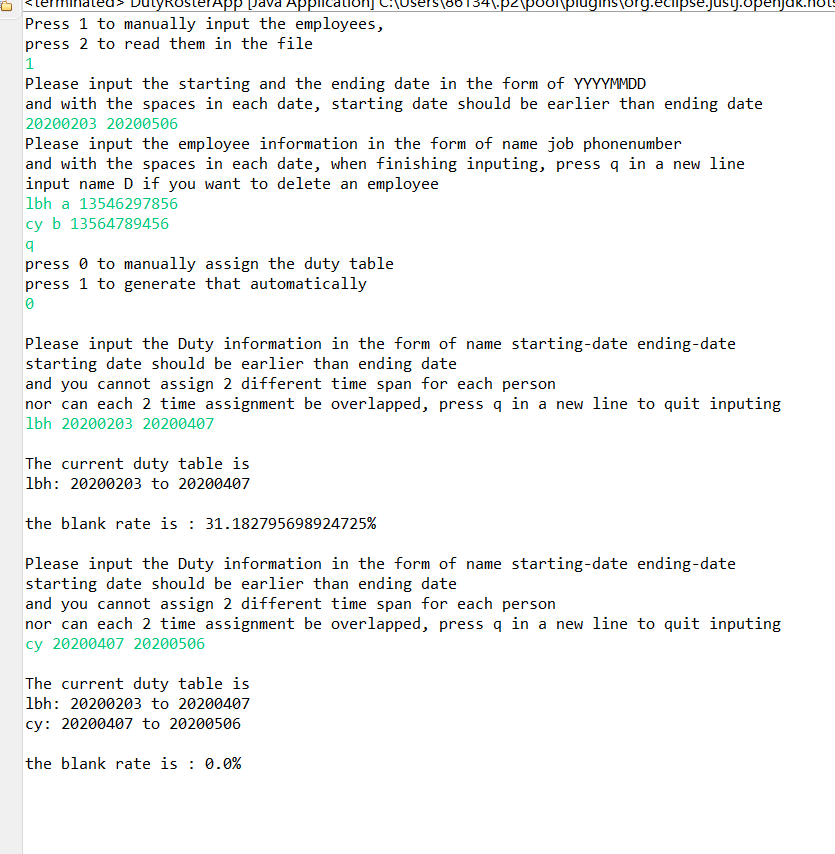
给出checkRep模块：



根据以上的内容，我们可以给出关于此函数的具体功能的简介和声明：

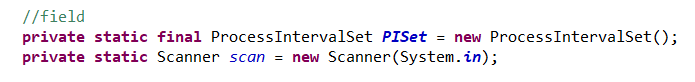
|  |  |
| --- | --- |
| 函数声明 | 功能概述 |
| **static** **private** **void** addEmployeeManual() | 手动添加一个Employee成员 |
| **private** **static** **void** manualWorkflow() | 手动输入状态时的主流程控制函数 |
| **private** **static** **void** manualAssign() | 手动为员工分配时间 |
| **private** **static** **void** setPeriodManual() | 手动设置起止时间 |
| **public** **static** **void** main(String[] args) **throws** IOException | 主函数，控制两种工作流运行 |

运行示意如下：

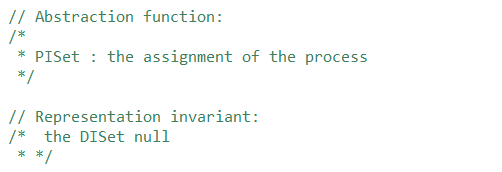


### 操作系统的进程调度管理系统

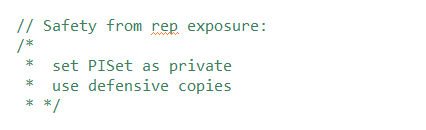
首先，我们可以给出该类的数据结构：



其次我们可以给出相关的AF和RI等信息：



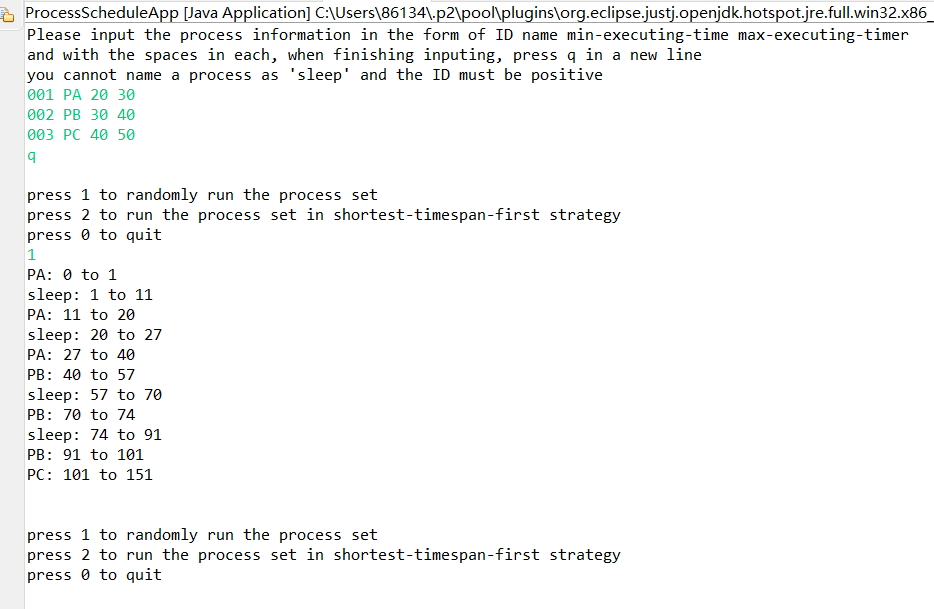
给出防止数据泄漏的方法：



根据以上的内容，我们可以给出关于此函数的具体功能的简介和声明：

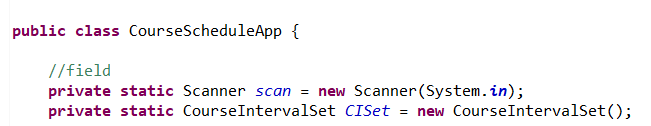
|  |  |
| --- | --- |
| 函数声明 | 功能概述 |
| **private** **static** **void** addProcess() | 将进程加入集合 |
| **public** **static** **void** main(String[] args) | 控制主函数流程 |

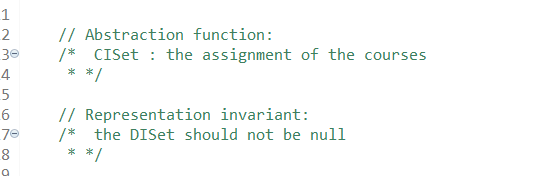
运行结果如下：



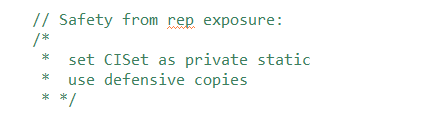
### 课表管理系统

首先，我们可以给出该类的数据结构：

其次我们可以给出相关的AF和RI等信息：



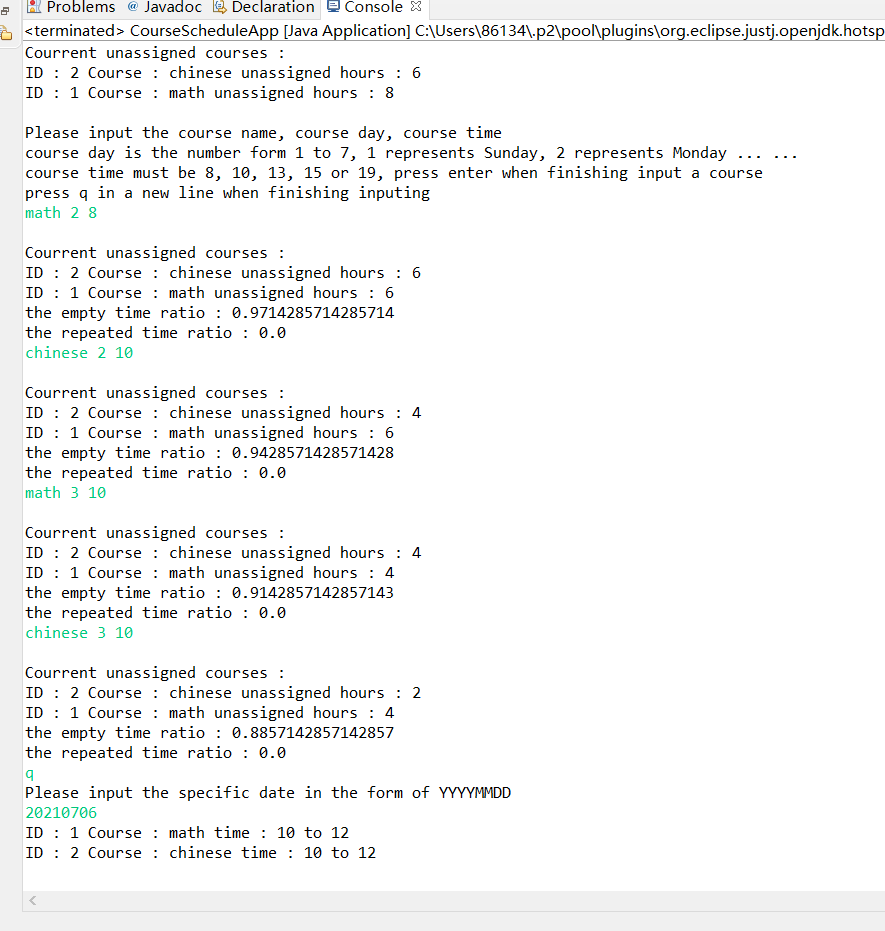
给出防止数据泄漏的方法：



根据以上的内容，我们可以给出关于此函数的具体功能的简介和声明：

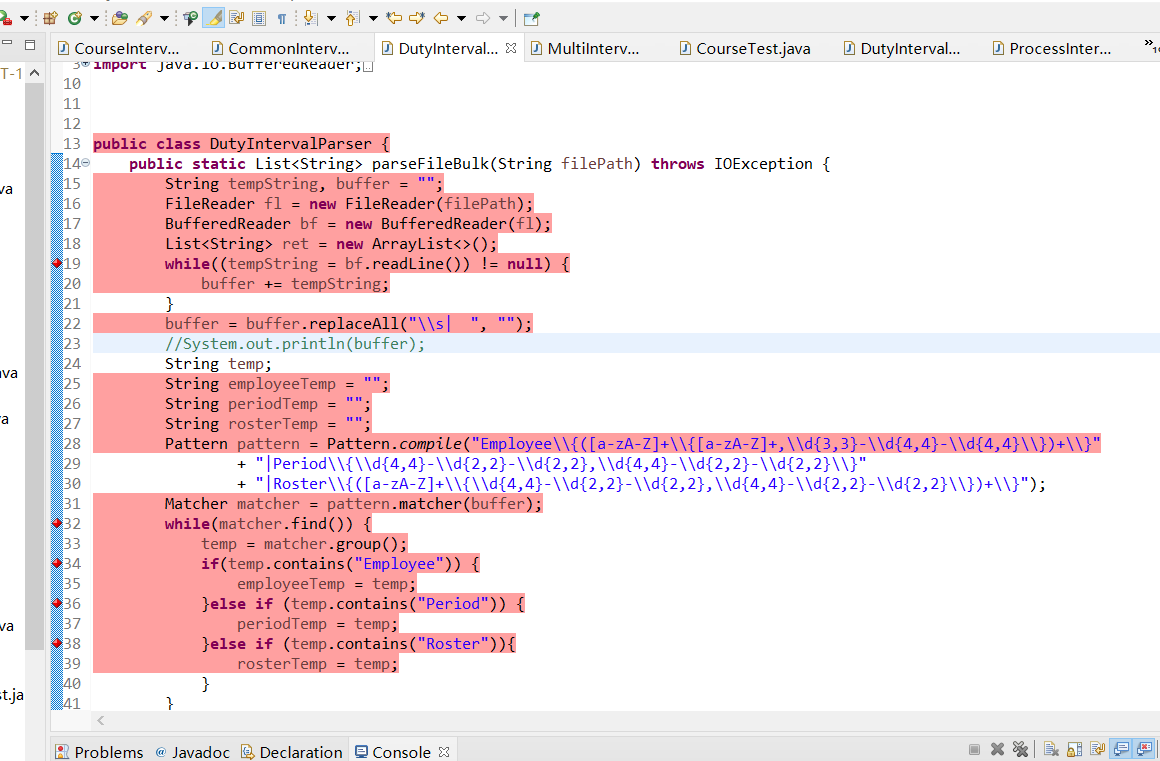
|  |  |
| --- | --- |
| 函数声明 | 功能概述 |
| **public** **static** **void** initial() | 初始化集合，设置上课时长和开始时间 |
| **public** **static** **void** addCourse() | 加入课程进入集合 |
| **public** **static** **void** assignCourse() | 分配课程时间 |
| **public** **static** **void** PrintBlankPeriodRate() | 打印空闲比例 |
| **public** **static** **void** PrintRepeatedPeriodRate() | 打印重复时间段比例 |
| **public** **static** **void** CheckSpecificDateCourse() | 查看指定的日期的课程 |
| **public** **static** **void** main(String[] args) | 控制工作流 |

具体运行如图：



## 基于语法的数据读入

我们可以针对输入文件的规则，构造一个针对该类型格式的文本文件的语法解析器parser，主要负责解析文本，主要构造代码如下：

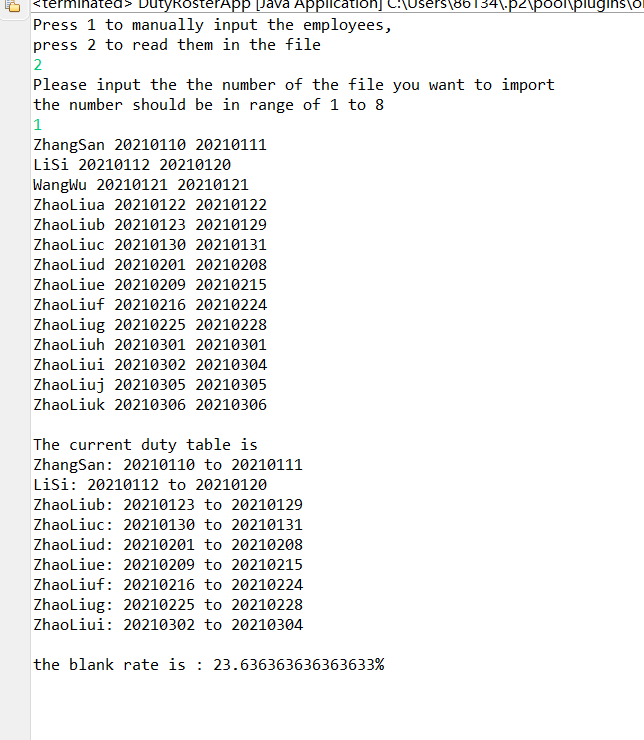


需要注意的是，在使用文本文件时，其中有处理不了的中文换行符，需要我们手动将其替换为英文换行符。

然后，我们可以向DutyRosterAPP中增加如下方法：

|  |  |
| --- | --- |
| 函数声明 | 功能概述 |
| **private** **static** **void** fileWorkflow() **throws** IOException | 控制文件读入等等流程的工作流函数 |
| **private** **static** **void** AssignPeriodByFile(String rosterString) | 根据输入文件自动分配时间段 |
| **private** **static** **void** SetEmployeeByFile(String employeeString) | 根据输入文件设置员工名单 |
| **private** **static** **void** SetPeriodByFile(String periodString) | 根据输入文件设置起止时间 |
| **private** **static** String getTargetFilePath() | 选择输入文件路径 |

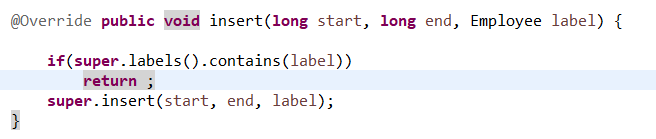
新增模块运行如下图所示：



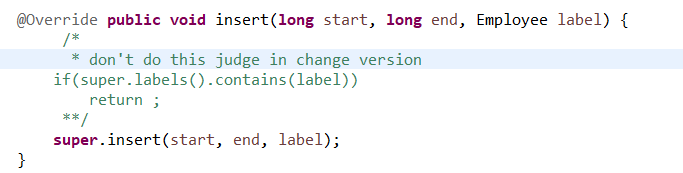
## 应对面临的新变化

### 变化1

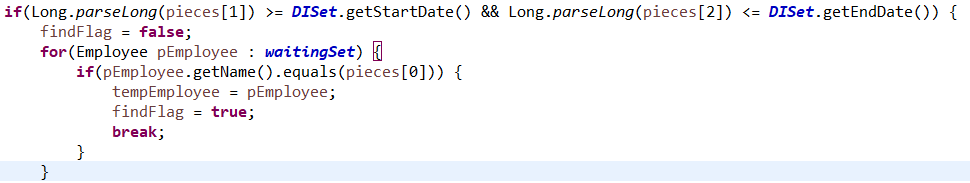
在此之前的master分支中的设计，在每次调用父类的插入函数的时候，完成检测：即将插入的标签label是否已经存在于父类的lablelList中，如果已经存在，就说明该label已经被分配时间，那么insert函数直接退出，如下图：



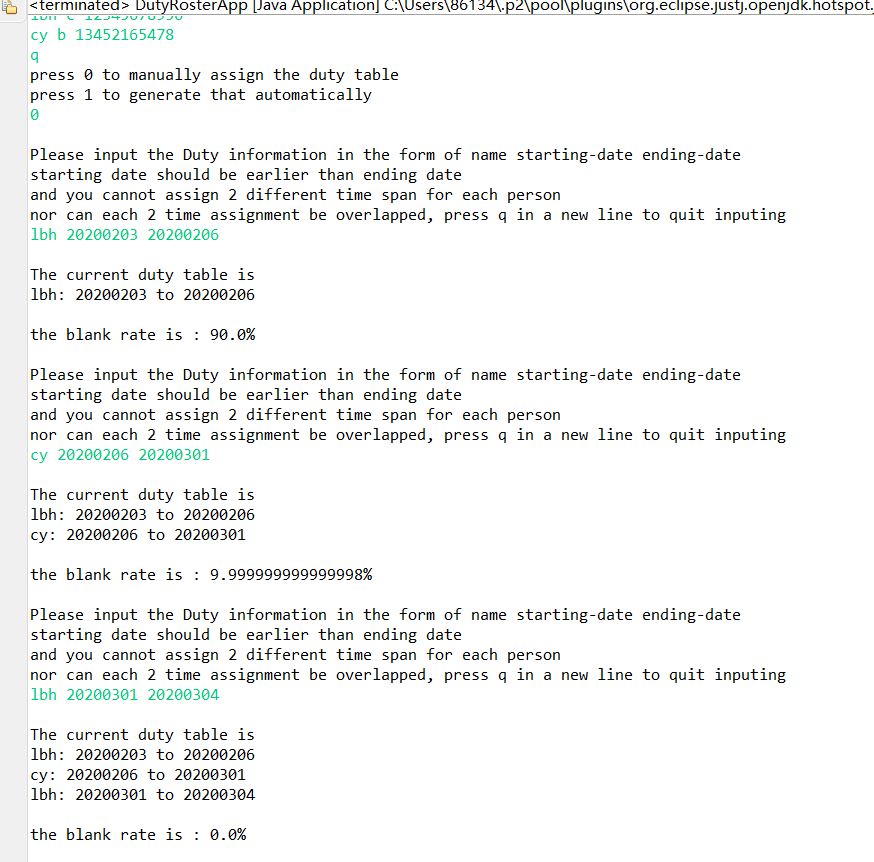
那么，在change版本中，只需要去除if判断即可：



同时，由于在APP类中进行增加之前，已经检测了次label是否在waitingSet中（如下图），因此，这样的改动简单而合理



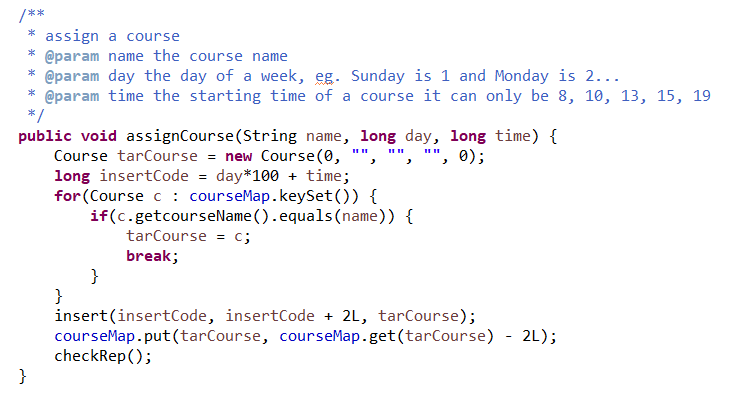
Change版本运行结果如下图：



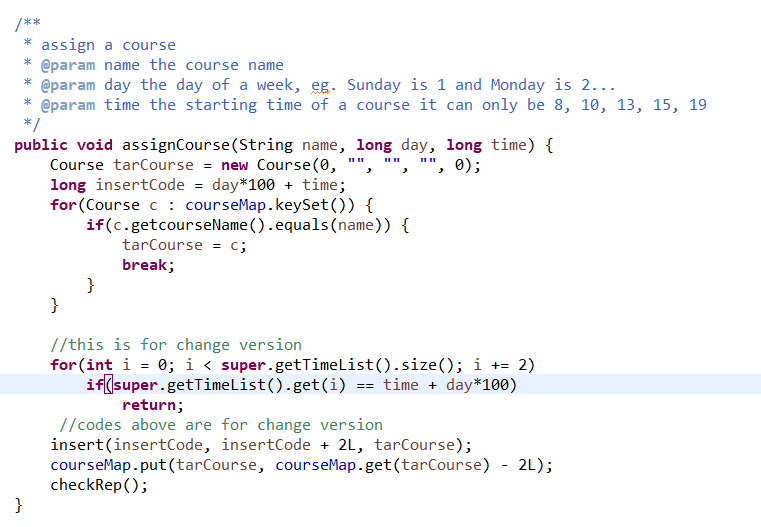
### 变化2

我们只需要在CourseIntervalSet类执行插入操作的时候检测一下目标时间点是否被占用即可。

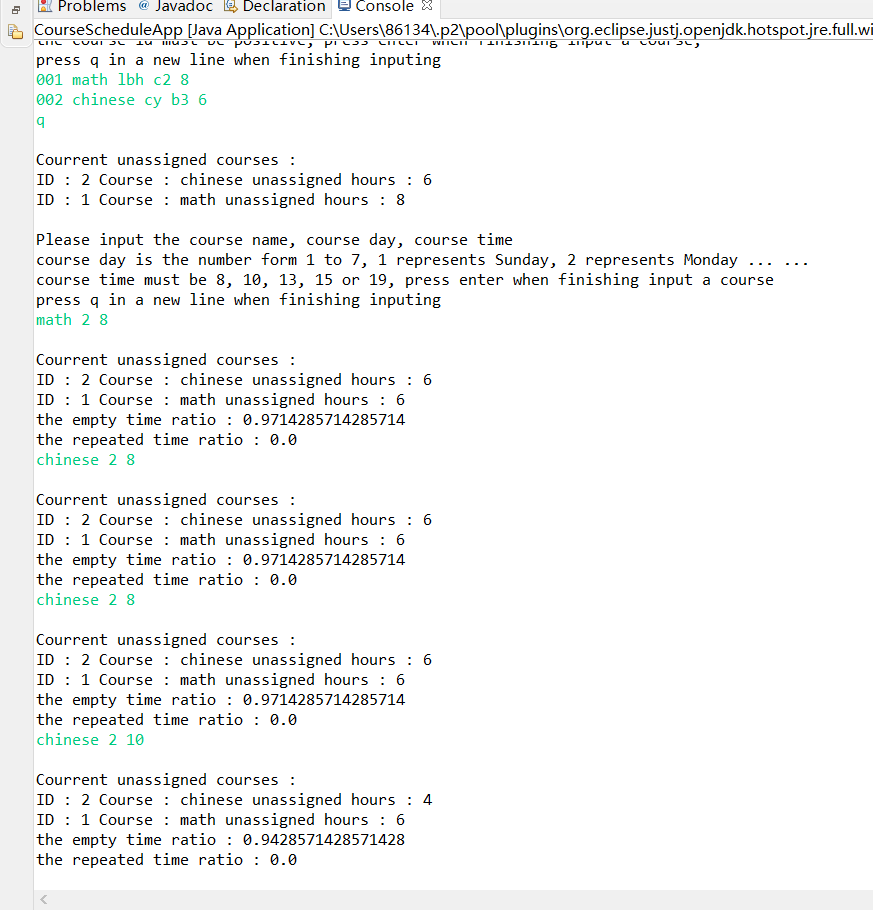
原本代码如下：



Change版本代码添加如下判断：

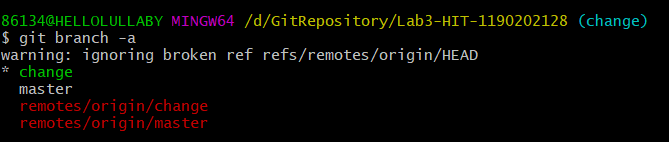


具体运行结果如下：



## Git仓库结构

请在完成全部实验要求之后，利用Git log指令或Git图形化客户端或GitHub上项目仓库的Insight页面，给出你的仓库到目前为止的Object Graph，尤其是区分清楚change分支和master分支所指向的位置。



# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 7.1 | 16:00-23:00 | APP1开发 | 完成 |
| 7.2 | 16:00-23:00 | APP2开发 | 完成 |
| 7.3 | 16:00-23:00 | APP3开发 | 完成 |
| 7.4 | 16:00-23:00 | Change开发 | 完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 读入字符串无法进行匹配 | 是由于文件读入出问题，原文件包含中文制表符，替换之后即可 |
| 完成APP2开发时无法运行控制流 | 控制函数写错，修改控制流程之后bug解决 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

更加了解了关于可复用性编程的意义，对于ADT的构造以及抽象类的意义，对于以后的软件编写有很好的帮助。

## 针对以下方面的感受

1. 重新思考Lab2中的问题：面向ADT的编程和直接面向应用场景编程，你体会到二者有何差异？本实验设计的ADT在五个不同的应用场景下使用，你是否体会到复用的好处？

ADT编程虽然复杂，但是有极好的复用性，应该多以其为主。

1. 重新思考Lab2中的问题：为ADT撰写复杂的specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后的编程中坚持这么做？

对于ADT复用时候的规范性有了更深的了解，更加清楚了对于规范编写的重要性，我会坚持。

1. 之前你将别人提供的API用于自己的程序开发中，本次实验你尝试着开发给别人使用的API，是否能够体会到其中的难处和乐趣？

对于其编写过程较为麻烦，但是复用性极好。

1. 你之前在使用其他软件时，应该体会过输入各种命令向系统发出指令。本次实验你开发了一个解析器，使用语法和正则表达式去解析输入文件并据此构造对象。你对语法驱动编程有何感受？

这样可以使得程序的运行更加方便，编写过程也非常有趣，其乐无穷