第四次博客作业

1. 基于度量分析作业架构
2. 第一次作业：

第一次作业UML类图如下:



以下为类的相关度量数据统计：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类名 | 属性个数 | 方法个数 | 分支个数 | 总行数 |  |  |  |
| match | 0 | 1 |  | 153 |  |  |  |
| term | 2 | 6 |  | 37 |  |  |  |

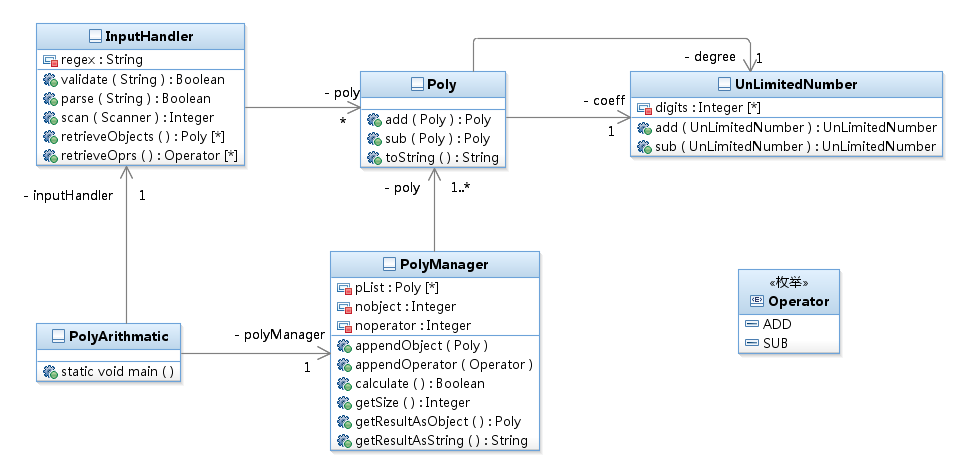
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 所属类 | 方法名 | 方法规模 |  |  |  |
| match | main | 144 |  |  |  |
| term | compareTo(term):int | 5 |  |  |  |
| term | getCoef():int | 3 |  |  |  |
| term | getExpo():int | 3 |  |  |  |
| term | getTermstring():String | 3 |  |  |  |
| term | setCoef(int):void | 3 |  |  |  |
| term | term(String, String, String) | 4 |  |  |  |

和推荐设计相比，我的结构区别在于：

1. 首先没有构造管理并计算多项式的类，而是直接以项为最小单位，同时计算方法封装在了main函数里；
2. 将多项式管理的相关操作放在了main里面通过一个term类型的数组来进行存储
3. 加减法的实现是通过读取输入过程中直接在系数上加上正负号，然后根据指数排序，合并同类项，直接求系数累加和得到结果，并输出output数组

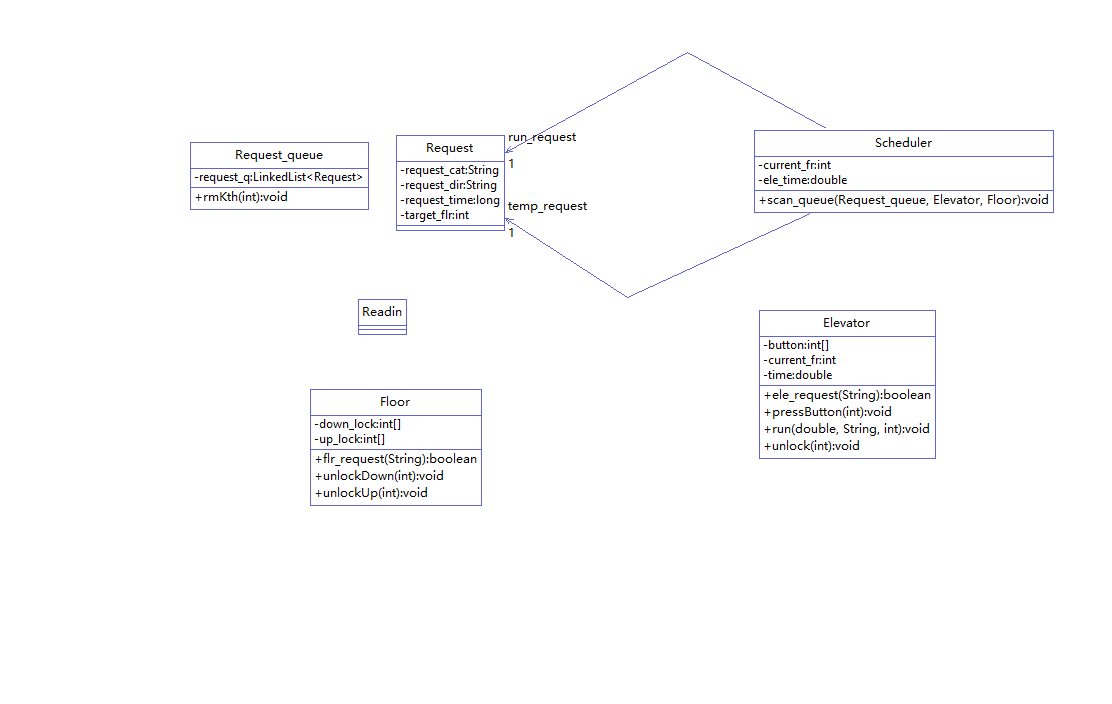
经分析，我的设计中缺少管理多项式和加减法的计算的功能，导致main方法规模过大不利于维护；同时多项式的管理输出倾向于过程式程序设计，也即读取输入，计算，输出；最后，没有拆分main函数的功能，不同功能之间耦合度太高，不利于后期扩展，可以预见的是，如果后期要求填加可视化的可交互的计算界面或者是加入乘除法功能的时候，我的main函数将需要重构。

虽然我的设计有诸多缺点，但对于第一次作业的任务来说，优点在于结构清晰明了，代码量少，虽然偏向过程，但是针对此次作业来说，针对性强。对于初学者而言，算是一个学习成本构思成本都较低的快速实现方法。（当然，我在评测中还遇见了用50多行c实现全部功能且过了全部公测的大佬。。。）



1. 第二次作业：

第二次作业UML类图如下：



以下为类的相关度量数据统计：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类名 | 属性个数 | 方法个数 | 分支个数 | 总行数 |  |  |  |
| Elevator | 3 | 10 |  | 84 |  |  |  |
| Floor | 2 | 9 |  | 79 |  |  |  |
| Request | 4 | 5 |  | 50 |  |  |  |
| Request\_queue | 1 | 8 |  | 38 |  |  |  |
| Scheduler | 2 | 2 |  | 68 |  |  |  |
| Readin | 0 | 1 |  | 78 |  |  |  |

以下为各个类的方法详情（略去getter和setter）

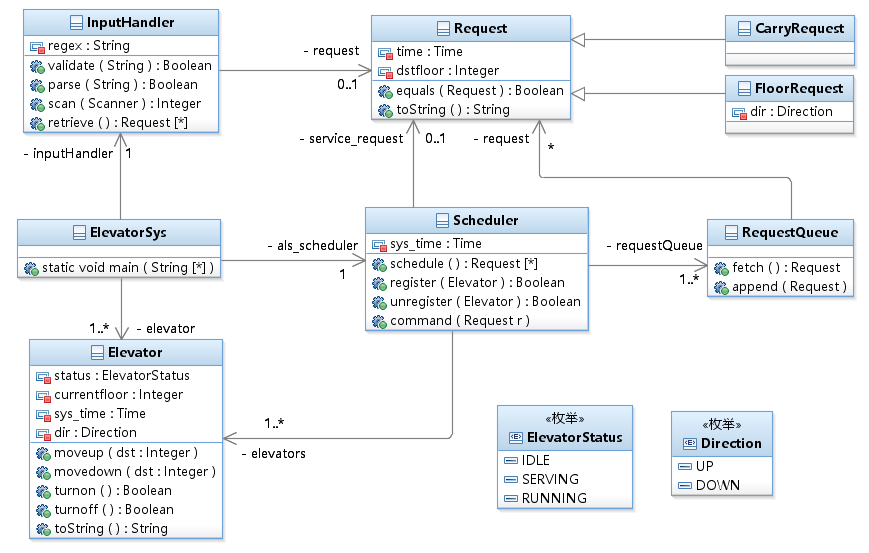
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 所属类 | 方法名 | 方法规模 |  |  |  |
| Elevator | Elevator() | 5 |  |  |  |
| Elevator | ele\_request(String):bool | 14 |  |  |  |
| Elevator | addTime(double):void | 3 |  |  |  |
| Elevator | pressButton(int k):void | 3 |  |  |  |
| Elevator | unlock(int):void | 7 |  |  |  |
| Elevator | run(double, String, int):void | 14 |  |  |  |
| Floor | unlockUp(int k):bool | 7 |  |  |  |
| Floor | unlockDown(int k):bool | 7 |  |  |  |
| Floor | flr\_request(String):bool | 12 |  |  |  |
| Request | Request(String, String) | 18 |  |  |  |
| Request\_queue | Request\_queue() | 3 |  |  |  |
| Request\_queue | add(Request):void | 3 |  |  |  |
| Request\_queue | isEmpty():bool | 3 |  |  |  |
| Request\_queue | getSize():int | 3 |  |  |  |
| Request\_queue | rmKth(int):void | 3 |  |  |  |
| Scheduler | scan\_queue(Request\_queue , Elevator, Floor):void | 58 |  |  |  |
| Readin | main | 68 |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

和推荐设计相比，我的结构区别在于：

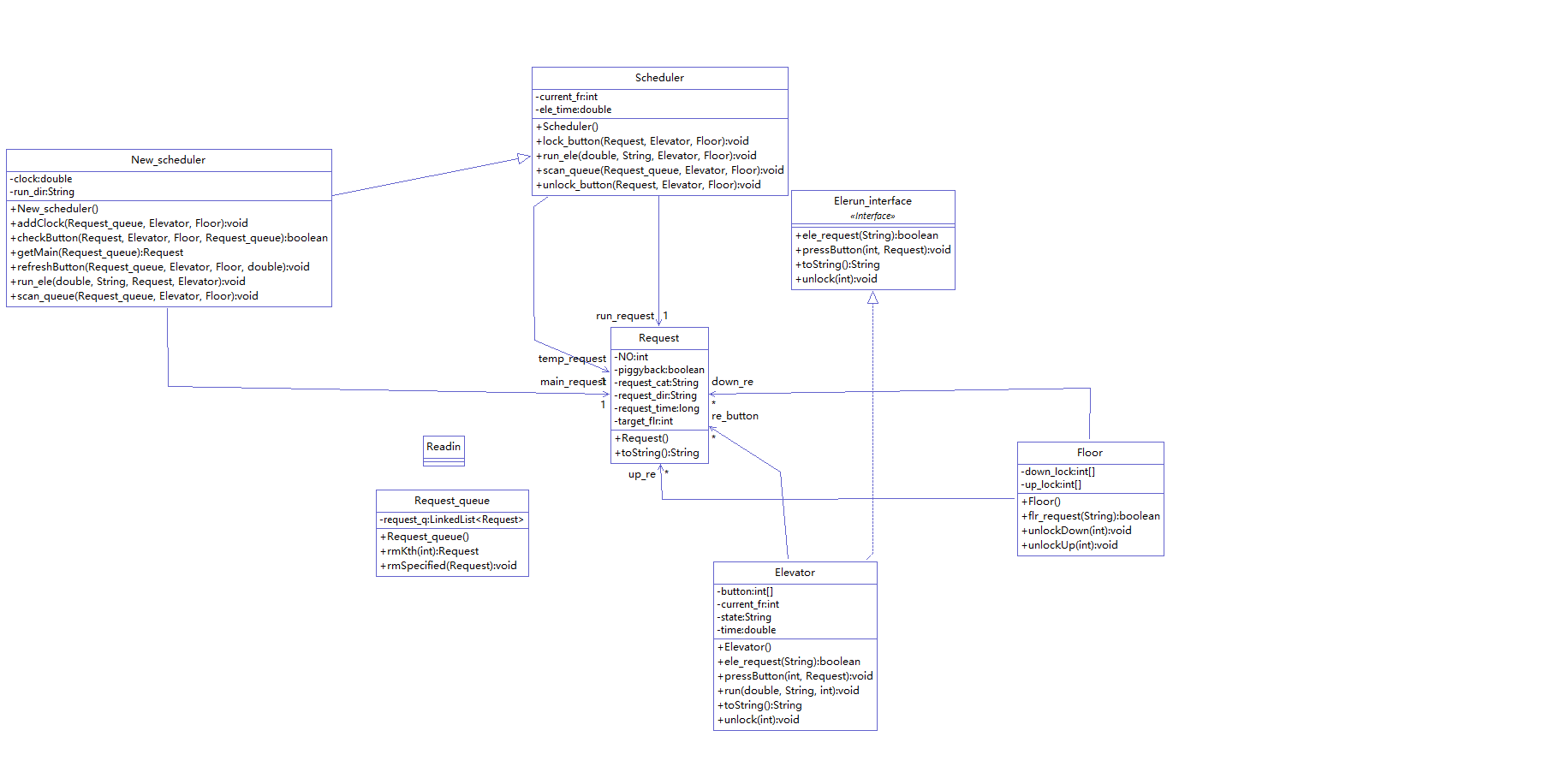
1. 没有设置ElevatorSys而是将main函数放在了Readin中
2. 电梯运行采取面向队列的刷新方式，根据下一个任务跳跃性地更新时间，而不是使用按按钮再运行并刷新时间的方法。
3. 没有设计电梯注册和解除注册的方法
4. 没有将电梯上下设置为单独的方法，而是将电梯运行放在了调度器的扫描队列方法中

由此可见，我的设计缺点在于，调度器（并且调度器内部方法不平衡）和Readin掌管功能太多，导致严重的局部不平衡，尤其是调度器一个扫描队列的方法不仅包含了处理任务队列，调度运行，还包括了输出运行结果的功能；另外一个很大的缺点在于，很多方法重复调用了同一个电梯和楼层对象但是没有很好的复用，而是直接创造了新的局部对象，这导致很多方法传递的参数过于臃肿，也导致需要更加细致的getter方法，类间耦合度过高。

但由于将时间的更新都放在了扫描队列的过程中，也相对简化了方法之间的调用关系，调度器可以同时获得时间和电梯运行情况，这样使得电梯类内部逻辑得到简化，电梯类相比较推荐设计中还带有上下方法的写法，就变成了只记录信息的一个载体。



第三次作业  
第三次作业UML类图如下:



以下为类的相关度量数据统计：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类名 | 属性个数 | 方法个数 | 分支个数 | 总行数 |  |  |  |
| Elevator | 3 | 10 |  | 84 |  |  |  |
| Floor | 2 | 9 |  | 79 |  |  |  |
| Request | 4 | 5 |  | 50 |  |  |  |
| Request\_queue | 1 | 8 |  | 38 |  |  |  |
| Scheduler | 2 | 2 |  | 68 |  |  |  |
| Readin | 0 | 1 |  | 78 |  |  |  |
| New\_scheduler | 3 | 7 |  | 158 |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

第三次作业由于是在第二次作业的基础上完成，因此主要重写了调度器，并继承了部分上一次作业的调度器方法。以下表格主要为相比较第二次作业新添加的方法。

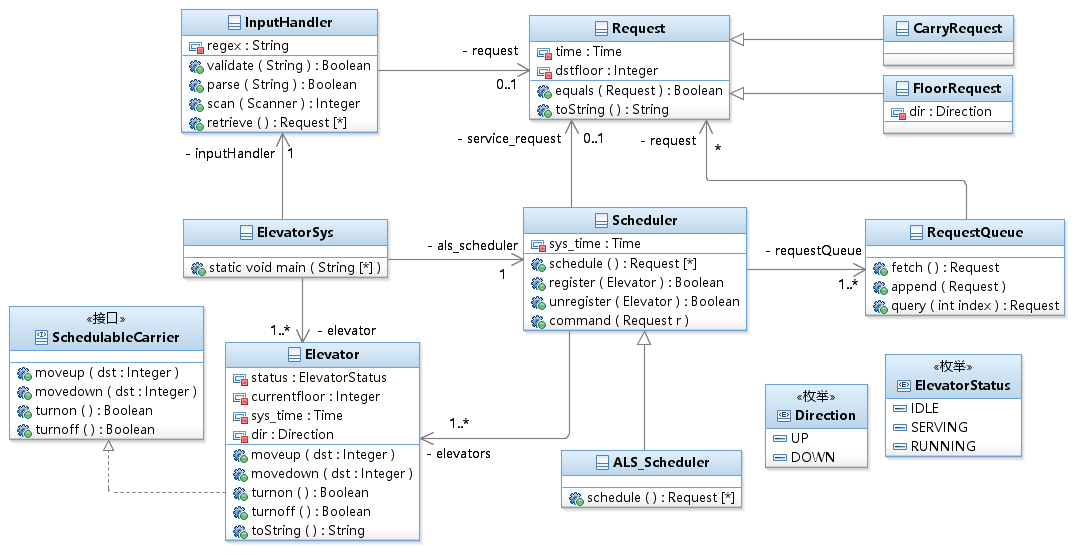
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 所属类 | 方法名 | 方法规模 |  |  |  |
| New\_scheduler | getMain(Request\_queue):Request | 10 |  |  |  |
| New\_scheduler | scan\_queue(Request\_queue, Elevator, Floor):void | 67 |  |  |  |
| New\_scheduler | addClock(Request\_queue, Elevator, Floor):void | 4 |  |  |  |
| New\_scheduler | refreshButton(Request\_queue ,Elevator, Floor, double):void | 20 |  |  |  |
| New\_scheduler | checkButton(Request, Elevator, Floor, Request\_queue):bool | 26 |  |  |  |
| New\_scheduler | run\_ele(double, String, Request, Elevator):void | 5 |  |  |  |
| New\_scheduler() | New\_scheduler():void | 5 |  |  |  |
| Elevator | toString():String | 3 |  |  |  |
| Request | toString():String | 5 |  |  |  |

和推荐设计相比，我的结构区别在于：

1. 将main函数仍旧放在readin里面，只是更换了调用的调度器（此次用了New\_scheduler对象）
2. 模拟真实情况下的电梯运行，按照0.5秒的频率进行刷新，每次刷新，都会根据当前主请求得到运动方向并更新电梯状态和按钮状态，输出相应的反馈信息。
3. 在按钮中记录了按下按钮的请求信息，方便输出或者判断重复或者在选取下一个主请求时进行判断。

经过分析，和推荐设计相比，我的结构主要缺点在于调度器类内仍然不够平衡，这一点从行数上可以看出；而且从类图来看，我声明了过多的request中间变量，浪费空间而且类间调用有重复的倾向，逻辑上存在冗余；按钮上还存储了请求信息，浪费空间；

相对应地，本架构的优点在于调度器模拟了电梯运行，统一管理当前时间和电梯状态，消除了时间不同步的状态；同时按钮上记录请求信息，可以方便地判断请求重复，或者输出响应信息，或者选择最近的主请求。



1. 分析个人程序中出现过的bug
2. 第一次作业：第一次作业出现的bug可以归纳为正则表达式运用不熟练，导致对find()和match()方法的混淆，从而引发了部分匹配问题。

经过总结发现，第一次公测集的错误全部都是因为出现部分正确输入的时候，不能提示输入不正确，而是给出了正确部分输入的计算结果。这是由于在匹配输入字符串的时候，我直接使用了



来寻找匹配的项，但是find方法只负责找到匹配的部分，对于剩下的字符串并不做要求，因此当输入中混杂了正确和错误的输入时，我的程序会在第一个错误的输入处停止。

解决：如果使用了match()方法先对整体输入进行检查的话，就可以避免部分正确的输入这种情况了。当然，为了防止直接使用match导致的正则表达式匹配过多，容易爆栈的问题，我们也可以继续使用match但是记录一下整体match的字符串长度，如果和去掉空格之后的输入字符串长度不一样，就说明出现了部分匹配问题，而单独匹配一个项的正则表达式是很简练的，不可能出现爆栈，这也就在解决问题的同时大大提高了鲁棒性。

1. 第二次作业：第二次作业暂时未发现bug
2. 第三次作业：第三次作业出现了两个bug，一个是输出请求时出现null，另一个是运行三个及以上的重复连续请求时可能会出现“button not lit”的按钮提前被熄灭问题。经过分析，这两个bug其实来自于同一个逻辑上的错误，我的运行模式是根据时钟走动，每0.5秒刷新一次电梯和当前按钮的状态，如果可以响应就输出响应信息，并且将对应的按钮熄灭（相当于按钮数组对应位置0且删除对应位置记录的请求）扫描任务队列，如果此时刻还有按钮被按下，就将对应楼层或者电梯按钮按下（也即记录相关信息），但是在一个请求被选为主请求的时候，由于人为的疏忽，忘记了在选为主请求时，记录对应按钮请求，这就导致了出现重复的请求的时候，只有非主请求才能被记录并输出结果，而主请求一旦执行完毕，由于按钮并没有加上锁导致解锁步骤出错并输出按钮并未点亮的报错信息，同理，由于按钮信息没有记录完全，导致输出结果时出现null字样，因为当前执行的按钮并不能获取到信息。

解决：在控制分支中加入对主请求的处理，记录对应信息，在判断重复的时候加入对主请求的判断和处理。

1. 关于寻找bug的策略

首先，我觉得bug按照程序功能可以分为三类，输入检查和数据清洗，运算逻辑，输出格式；而按照来源可以分为两类：考虑不周和笔误。因此，接下来我将按照程序功能的实现谈一谈我对于找bug的理解。

首先，输入检查和数据清洗，目前大部分同学都已经学会了正则表达式基础的运用，而很多后续的bug都从输入开始，我们需要明确的是，除了自身逻辑问题之外，输入检查是保证程序稳定的第一道关卡，也是唯一一副铠甲。有的同学采取逐字符检测的形式进行检查，但是这存在着巨大的弊病，错误的输入千千万万，在情况变得复杂的时候，我们很难对错误输入进行穷举，同时正确的情况也可能使得逻辑变得复杂，不堪调试，因此我们需要反向思维，从另一个方向解决问题，那就是使用正则表达式直接限定唯一能够被接受的输入格式，并采用多级正则表达式防止爆栈（只是用一个很长的复杂的正则表达式，兼具输入检查和提取信息功能的话容易被爆）。

运算逻辑方面，我会通过类之间的关系和跟踪代码运行情况来大致掌握代码逻辑，这一点需要我们在写代码之前***认真***阅读指导书（阅读理解大家从小学就开始练了。。。也不差大学这半年复习一下）有规定的部分严格执行，未规定的部分记得加入Readme定义，经过细致的考虑，同时在写代码过程中还需要不断自检，最后测试的时候跟踪运行轨迹以确保代码符合自己的预期运行情况。最后要注意的一点是涉及到循环，计算相关的东西，都需要检查极限值和边界值，防止正负问题和越界问题。

输出格式上我们在学习了重写Tostring方法之后就有了得力的工具，只要在包含需要输出的信息的类中加入该方法，并重写即可。调用的时候可以直接在print中写对象名调用，如下：



而谈到如何找到别人的bug，对此我主要有三点策略：

1. 记录自己在逻辑构思和编码阶段碰到的设计问题和相关的测试样例，记录自己出现过的bug
2. 测试边界性的bug，比如超长输入，越界输入，部分正确的输入等
3. 阅读对方代码，结合跟踪调试和类图理解对方逻辑并根据发现的逻辑漏洞针对性构造样例。

以上执行顺序是123，虽然现在只在公测阶段对输入格式等进行检查，但我们依然需要关注对方的输入检查部分，任何非法输入在进入了运算阶段，都会产生意想不到的蝴蝶效应。

1. 心得体会

三次作业带给我最大的收获有两个，一个是快速自学新知识的能力（相信大部分同学都已经被逼出来这个技能点了）另一个则是高效的工程时间管理。首先，学知识方面已经是老生常谈，无非就是上课认真听，回来了自己看书，书可以跳着看，挑选本次作业需要的部分着重看，比如第三次作业，仔细看看继承和重写，同时结合网络搜索，基本技术上的问题都能够被很好地解决；而导致作业质量低下或者甚至无效的主要原因，并不是学不会，更多可能是因为时间管理失误。还记得开学初老师就推荐过一本书叫人月神话，书中内容可以算是软件工程领域时间管理进度管理的肺腑之言，我印象最深的就是作者认为编码时间才是整个工程中最不起眼的部分，构思和调试才是重头，而理想状态下，构思应该花最长的时间（或许是因为商业因素，需要快速上马新产品才导致的压缩构思时间）。因此大家最好在指导书发布后留一天甚至两天来认真思考，最好能画出你预期的思路和处理方式，俗称“脑补阶段”。之后才是编码，编码过程中碰到了问题才是检验上课听讲和自学能力的时候。一般来说，只要初始构思考虑到了扩展性和大部分逻辑问题，测试阶段需要大批返工的概率微乎其微，基本上都是在修复编码阶段的人为失误。而调试阶段，我们需要考虑以下几个方面（如有遗漏还请大家指出）：

1. 要实现的功能（单个测试）
2. 功能组合的测试
3. 输入检查
4. 输出规范

在测试过程中，保持一个良好的随时记录总结样例和自己思考内容的好习惯，这样方便在测试他人的时候使用或者发现自己的新问题。