Unit3

JML(Java Modeling Language) 是用于对 Java 程序进行规格化设计的一种表示语言，它对于方法内部具体是如何实现的并无要求，只是对方法的接口以及行为进行限制，具体来说就是对方法里面允许接受的数据，可以做出的改动，必须返回的数据，异常表现等进行限制，最终保证方法的无二义性。

本单元的作业背景是一个简单的社交系统，允许对数据进行有序存储，建立数据之间的关系，以及精确查询，由于已经给出了JML规格，所以并不需要自己从零开始设计架构，只用把需要实现的方法按照JML规格实现即可，在实现的过程中必须特别注意方法的复杂度。~~卷不了性能分 那就在复杂度上区分~~

一、实现规格所采取的设计策略

为了实现规格首先肯定需要学习JML相关语法，在掌握相关语法之后，仔细研读课程组给出的JML规格，并进行实现就行了。

具体设计策略：

- 1.对代码、JML进行通读，整体了解新增指令的接口以及具体行为
    - 这样能做到心里有个谱，清楚哪些方法是难点，容易tle，需要进行算法和数据结构的设计，~~就先看看学长学姐的blog~~；不同的指令行为是如何相互影响的，除了能减少实现方法时的出错概率，由于对指令比较熟悉，之后构造数据也比较方便。
- 2.先把所有方法的签名的写好
    - 这样第一是能避免自认为完成之后，发现有几个方法忘了实现的情况；第二是避免在写的过程中，代码冗余的情况。

      像这种JML很短的方法如果夹在两个JML很长的方法之间就容易漏掉：

```
/*@ public normal_behavior
  @ requires contains(id1) && contains(id2);
  @ ensures \result == getPerson(id1).getName().compareTo(getPerson(id2).getName());
  @ also
  @ public exceptional_behavior
  @ signals (PersonIdNotFoundException e) !contains(id1);
  @ signals (PersonIdNotFoundException e) contains(id1) && !contains(id2);
  @*/
public /*@pure@*/ int compareName(int id1, int id2) throws PersonIdNotFoundException;

//@ ensures \result == people.length;
public /*@pure@*/ int queryPeopleSum();

/*@ public normal_behavior
  @ requires contains(id);
  @ ensures \result == (\sum int i; 0 <= i && i < people.length &&
  @                     compareName(id, people[i].getId()) > 0; 1) + 1;
  @ also
```

有时明明已经实现的查询方法，但是碰到查询就直接使用对应容器的查询方法了。

例如`getMessage`里对messageid进行查询时有同学会直接用对应容器的查询方法:`messages.containsKey(id)`

由于容器以及对应查询比较简单，直接应用容器的查询方法或者使用已经实现的查询方法代码长度上好像差别不大，但是这样不仅代码不美观，可维护性也不好。如果先整体把握代码，就不会出现这种情况。

```java
@Override
public boolean containsMessage(int id) { return messages.containsKey(id); }

@Override
public void addMessage(Message message) throws EqualMessageIdException, EqualPersonId
    if (messages.containsValue(message)) {
        throw new MyEqualMessageIdException(message.getId());
    } else if (message.getType() == 0 && message.getPerson1().equals(message.getPerso
        throw new MyEqualPersonIdException(message.getPerson1().getId());
    } else {
        messages.put(message.getId(),message);
    }
}

@Override
public Message getMessage(int id) {
    if (containsMessage(id)) {
        return messages.get(id);
    }
    return null;
}
```

- 3.对方法进行具体实现
    - 通常一个比较复杂的方法会涉及到异常行为的处理，这时候if-else分支比较多，逻辑有一些复杂，但个人感觉从异常开始处理，最后实现方法的正常行为，按照这个流程来做逻辑上会很简单。

```java
/*@ public normal_behavior
  @ requires (\exists int i; 0 <= i && i < groups.length; groups[i].getId() == id2) &&
  @           (\exists int i; 0 <= i && i < people.length; people[i].getId() == id1) &&
  @            getGroup(id2).hasPerson(getPerson(id1)) == false &&
  @             getGroup(id2).people.length < 1111;
  @ assignable groups;
  @ ensures (\forall int i; 0 <= i < groups.length; \not_assigned(groups[i]));
  @ ensures (\forall Person i; \old(getGroup(id2).hasPerson(i));
  @          getGroup(id2).hasPerson(i));
  @ ensures \old(getGroup(id2).people.length) == getGroup(id2).people.length - 1;
  @ ensures getGroup(id2).hasPerson(getPerson(id1));
  @ also
  @ public normal_behavior
  @ requires (\exists int i; 0 <= i && i < groups.length; groups[i].getId() == id2) &&
  @           (\exists int i; 0 <= i && i < people.length; people[i].getId() == id1) &&
  @            getGroup(id2).hasPerson(getPerson(id1)) == false &&
  @             getGroup(id2).people.length >= 1111;
  @ assignable \nothing
  @ also
  @ public exceptional_behavior
  @ signals (GroupIdNotFoundException e) !(\exists int i; 0 <= i && i < groups.length;
  @          groups[i].getId() == id2);
  @ signals (PersonIdNotFoundException e) (\exists int i; 0 <= i && i < groups.length;
  @          groups[i].getId() == id2) && !(\exists int i; 0 <= i && i < people.length;
  @           people[i].getId() == id1);
  @ signals (EqualPersonIdException e) (\exists int i; 0 <= i && i < groups.length;
  @          groups[i].getId() == id2) && (\exists int i; 0 <= i && i < people.length;
  @           people[i].getId() == id1) && getGroup(id2).hasPerson(getPerson(id1));
  @*/
public void addToGroup(int id1, int id2) throws GroupIdNotFoundException,
        PersonIdNotFoundException, EqualPersonIdException;
```

比如addToGroup方法，首先处理`exceptional_behavior`，容易发现三种抛异常的情况时层层递进的，也就是说如果没在第一个地方抛异常，而在第二个地方抛了异常，那么异常原因就一定不满足一个if里的条件，同理最后正常行为的`requires`也可以此类推。这样就不会出现恶心的if-else三四次嵌套的情况了，个人感觉这样实现起来不但简洁美观，而且正确性也能得到保证。

```
@Override
public void addToGroup(int id1, int id2)
        throws GroupIdNotFoundException, PersonIdNotFoundException, EqualPerso
    if (!groups.containsKey(id2)) {
        throw new MyGroupIdNotFoundException(id2);
    } else if (!people.containsKey(id1)) {
        throw new MyPersonIdNotFoundException(id1);
    } else if (getGroup(id2).hasPerson(getPerson(id1))) {
        throw new MyEqualPersonIdException(id1);
    } else {
        MyGroup group = (MyGroup) getGroup(id2);
        if (group.getSize() < 1111) {
            group.addPerson(getPerson(id1));
        }
    }
}
```

二、基于JML规格设计测试的方法和策略

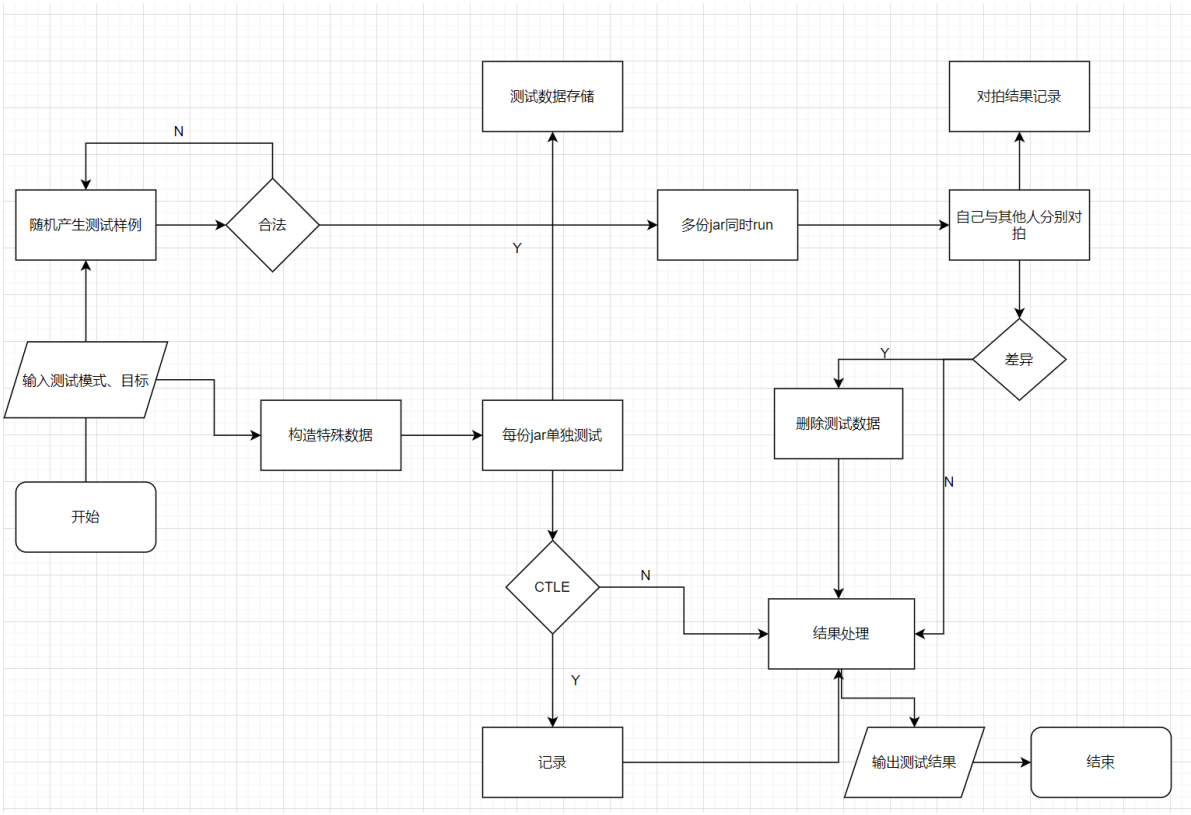
1.Junit

按照教程费了很大力气终于配好了环境，结果发现测试样例还是得自己构造，找同学了解后发现，貌似可以通过JML语法自动生成测试样例，但是这些样例大都是边界样例，强度还是不高，因此在三次作业中我基本都没有使用Junit进行单元测试。

2.评测机

我和hxd一起构造了评测机，进行了**所有指令的正确性测试**和**压力测试**

评测机大概思路如图：

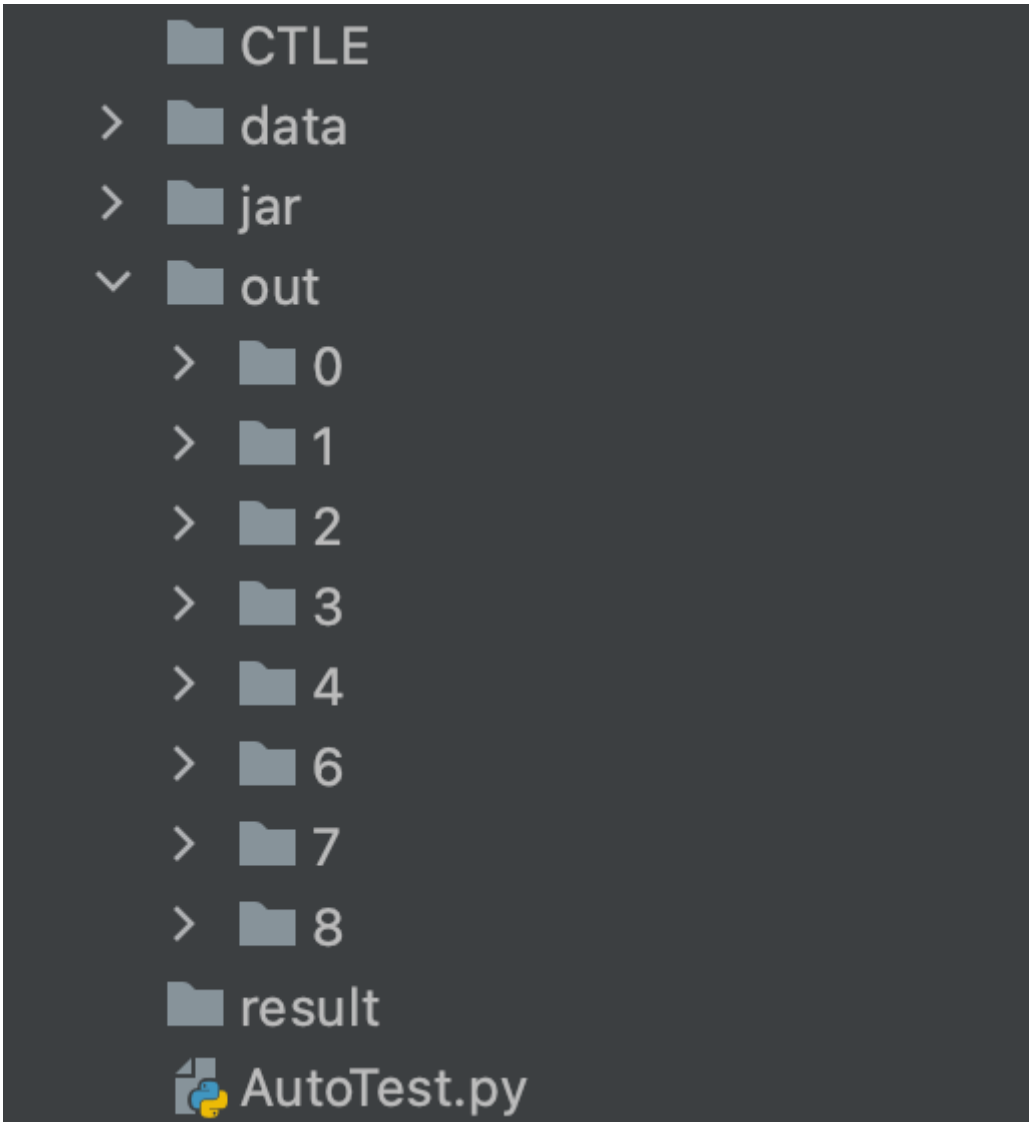


优点:

- 通过运行大量随机数据，并与同学对拍，基本能保证指令正确性

```
cmdList = ['ap', 'ar', 'qv', 'cn', 'qnr', 'qps', 'qci', 'qbs']
def getCmd():
    global numOfMaxHeat
    cmd = cmdList[random.randint(0, len(cmdList) - 1)]
    if cmd == 'ap':
        id = getRandomId()
        name = getRandomName()
        age = getRandomAge()
        cmd = cmd + " " + str(id) + " " + name + " " + str(age)
    elif cmd == 'ar':
        id1 = getRandomId()
        id2 = getRandomId()
        value = getRandomValue()
        cmd = cmd + " " + str(id1) + " " + str(id2) + " " + str(value)
```

- 对评测机的使用进行了优化，每次使用都只需输入特定的参数，而不用去对代码进行一系列更改
- 随机数据压力测试效果尚可，能检测出比较明显的ＣＴＬＥ
- 黑箱测试，在ｈａｃｋ的时候非常方便
- 涉及到大量数据、运行结果、对拍结果数据的生成，分别建立文件夹进行存放，方便之后快速查找



- 对拍完之后将没有意义的测试数据以及输出删除，避免磁盘空间的浪费

```
def delete(WrongList):
    for i in range(numOfData):
        for jarNum in jarList:
            outName = "out\\" + jarNum + "\\" + jarNum + "_out_" + str(i)
+ ".txt"
            if i not in WrongList:
                os.system("del " + outName)
```

- 在正确性检查时采用了ｐｙｔｈｏｎ多进程，提高测试效率；而在测时间时为了避免干扰不采用多进程

```
def run():
    if not os.path.exists(outPath):
        os.makedirs(outPath)
    pool = multiprocessing.Pool(processes=8)
    results = []
    if not os.path.exists(outPath):
        print("jarPath WA")
    for jarNum in jarList:
        path = outPath + jarNum + "/"
        if not os.path.exists(path):
            os.makedirs(path)
```

```
        for j in range(numOfData):
            results.append(pool.apply_async(test, (jarNum, j, )))
    for i in results:
        i.wait()
```

缺点:

- 正确性检查时不能做到自查，必须要有同学的jar包才能进行对拍
- 数据是随机的，不能保证覆盖所有的情况，不能保证边界情况
- 压力测试中，python的time库测的是程序实际运行时间，和ＣＰＵ使用时间有一定的差别，再加上ＣＰＵ运行时间本身有一定波动，导致最终测得的时间不一定准确
- 正确性检查的速度理论上受到本地ＣＰＵ核心数的限制，由于本地只有8核，所以ｐｒｏｃｅｓｓｅｓ只开到了8，在数据量较大时，测试时间还是比较长的
- 随机出来的数据没有测出过ＣＴＬＥ，只有看过代码后，手动构造出的针对性数据才能测出ＣＴＬＥ
- 缺少ＵＩ

效果:

- 三次作业的结果表明，在本单元中评测机能百分之百测出正确性上的问题~~主要还是看对拍的hxd~~，不仅是写完作业自查，还是ｈａｃｋ都十分好用
- 如果发现代码中存在复杂度很高的方法，通过构造特殊数据基本能保证ｈａｃｋ到

三、容器选择和使用

`HashMap`：用于存储键值对，本单元来说就是id与对应的实例，具体来说有`Person`,`Message`,`Group`类等等。

由于对键可以实现O(1)查找，而作业中对于内存使用几乎可以说没有限制，所以凡是能用HashMap的坚决不用其他的容器，因此它也是我本单元作业中主要使用的容器类型。

需要注意的是，在进行集合遍历得时候，如果存在对集合内部的元素进行改动的操作，最好使用迭代器进行遍历，否则会出错。例如dce这条指令需要对heat过小的emoji进行过滤：

```
Iterator iterator = emojiIdToemojiHeat.entrySet().iterator();
while (iterator.hasNext()) {
    Map.Entry entry = (Map.Entry) iterator.next();
    if ((Integer)entry.getValue() < limit) {
        iterator.remove();
    }
}
```

`ArrayList`:`qrm`这条指令需要查询某个`person`最新收到的**四条**`message`,可以利用`ArrayList`进行存储，每次加到末尾，查询的时候从最后开始反向遍历即可。

`Priority Queue`:内部用堆进行实现，可以用于dijikstra算法中进行堆优化。

在构造队列的时候可以传入比较器，也可以将要传入的实例对应的类实现`Comparable`接口，否则会出错:

```
Comparator comparator = new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
       assert o1 instanceof Node;
       assert o2 instanceof Node;
       return ((Node) o1).getDis() - ((Node) o2).getDis();
       }
   };
   PriorityQueue<Node> queue = new PriorityQueue<>(comparator);
```

四、设计容易出现的性能问题

1.第一次作业

容易出现性能问题的就是`qbs`这个操作了，由于涉及到连通块的查询，如果无脑按照ML的暴力做法来做，是一定会CTLE的。

由于本单元三次作业都没有对`relation`的删除操作，因此可以使用并查集：

```
public class MyPerson implements Person {
  private int fatherId;
  ……
    }

public int find(int id) {
        MyPerson person = (MyPerson) getPerson(id);
        if (person.getFatherId() == id) {
            return id;
        }
        int fatherId = find(person.getFatherId());
        person.setFatherId(fatherId);
        return fatherId;
    }

public void merge(int id1,int id2) {
        int fatherId1 = find(id1);
        int fatherId2 = find(id2);
        if (fatherId1 != fatherId2) {
            MyPerson person = (MyPerson) getPerson(fatherId1);
            person.setFatherId(fatherId2);
        }
    }
```

这里我留下了一个**bug**，在merge方法中fatherId 写成了id，导致虽然不会出错（之后再进行find，会自动更新为祖先节点）~~会出错 就好了~~但是多做了很多没无意义的工作，增大了时间成本，而第一次作业中强测指令数较小，bug没有发现

第二次作业发现CTLE了，一查发现是第一次的残留问题 (ノ°Д°)ノ

然后就修了bug，顺带将连通块的统计做成了动态更新的,每次加人、合并都要考虑连通块数量的更新，这样就能实现qbs的动态查询了，修复之后发现运行时间大概快了一倍。

```
@Override
public void addPerson(Person person) throws EqualPersonIdException {
        int personId = person.getId();
        if (people.containsKey(personId)) {
```

```
            throw new MyEqualPersonIdException(personId);
        } else {
            people.put(personId,person);
            block++;
        }
    }

public void merge(int id1,int id2) {
        int fatherId1 = find(id1);
        int fatherId2 = find(id2);
        if (fatherId1 != fatherId2) {
            MyPerson person = (MyPerson) getPerson(fatherId1);
            person.setFatherId(fatherId2);
            block--;
        }
    }
```

2.第二次作业

`qgvs`:如果完全按照JML，采用二重循环实现，同样也是必CTLE的

解决方法是采用动态维护的方法：

- 每向`group`中加入或删除一个人，都会对`groupValue`产生影响，只需要遍历目前组中存在的人，统计目前正在操作的人和组中存在的人的`relation`更新`groupValue`即可，当然还要考虑对已经在`group`内的人之间增加`relation`的情况，在`ar`操作时稍微注意就好，这样单次方法复杂度是O(n)

`qgav` `qgam`:一个是求方差，一个是求平均值，考虑到~~内存无限~~，在实现较简单的情况下，能做记忆化就尽量做记忆化

在更新方差的时候，需要将公式展开：从而可动态维护，⚠这里不能化简，以免造成精度与JML规格不符导致出错

$$Var（x）=\frac{\sum_{i=1}^{n}(x_i-\overline{x})^2}{n}=\frac{\sum_{i=1}^{n}x_i^2-2\times\overline{x}\times\sum_{i=1}^{n}x_i+n\times\overline{x}^2}{n}$$

每次只有向组内**加人**、**减人**才会影响方差和均值，这时才需要进行相应的更新，包括**平方和**，**总和**，**均值**等。考虑到需要查询两个属性，因此设置两个标记位，如果标志是`false`，就直接查询，如果是`true`，则进行相应更新。每次**加人**，**减人**后需要将两标志位都设成`true`，更新完之后相应标志位设为`false`

所以我的group类里面增设了很多属性。

```
    private int valueSum;
    private int ageSum; //组内所有人的年龄和
    private int ageMean;//组内所有人的年龄平均值
    private int ageVar; //组内所有人的年龄方差
    private int ageSquareSum; //所有人的年龄平方和
    private boolean needUpdateMean; //需要更新年龄平均值
    private boolean needUpdateVar; //需要更新年龄方差
```

3.第三次作业

`sim`:非直接相邻节点发送信息，单源最短路径，采用**dijikstra**算法实现，为了防T采用堆优化(~~甚么堆优化，java里就是优先队列，不要重复造轮子~~)

由于加人，加边等会影响原来的图结构，因此记忆化不太好做，选择放弃。

4.效果

总的来说效果很好，第三次作业最长CPU使用时间是1.944s。~~6s似乎太长了~~(⊙_⊙)?

五、框架设计（图模型的构建和维护策略）

本单元涉及到的需要图论知识解决的指令较少，只有：

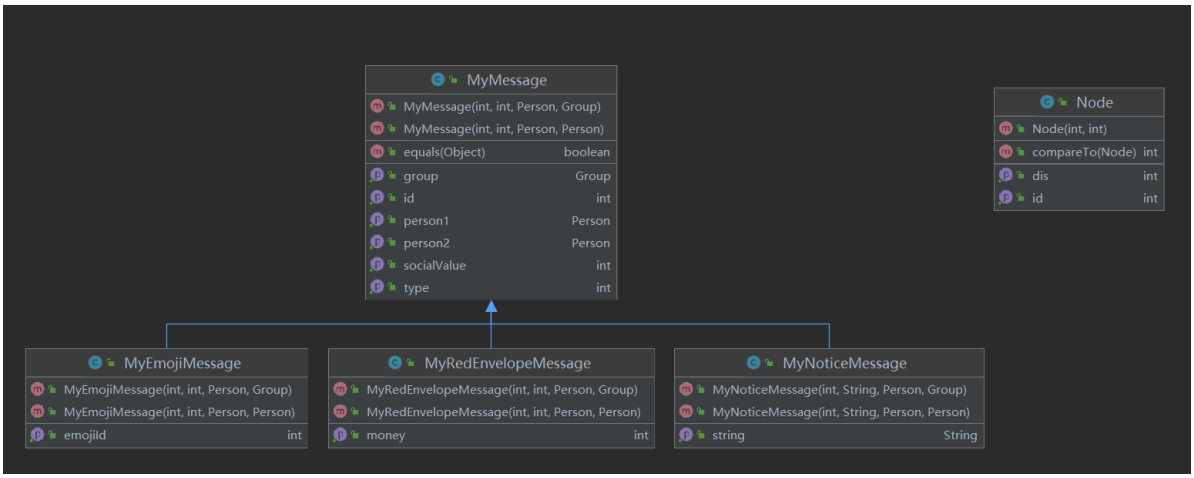
`qbs`：**并查集**

`sim`：**dijikstra**

两种算法不再赘述

在搜索单源最端路径的时候，我新建了`Node`类，里面存了`Person`的**id**和**距离起点的distance**，每次进行搜索时都要重新new一轮`Node`实例，并按照算法流程走一遍

其他框架设计感觉课程组都提前弄好了，没有什么可说的，唯一需要自己做的就是不同`XXmessage`对`message`的继承和`Node`类的构建



六、感想

- 本单元的重点是JML规格的学习，经过三次作业的训练，个人感觉已经能很快并且准确地读懂JML规格了，但是真要自己写可能还是比较困难。除此之外，个人感觉收获最大的是对于规格与契约式编程等概念在脑中的建立。
- 由于框架不需要自己从头设计，只需要按照JML规格填写代码，作业难度相比前两单元下降了很多，作业最大的难点我想应该就是如果保证自己实现的方法有较高的性能，而这训练的往往是数据结构和算法这类技能，除了最后一次作业不同message的继承能看到面向对象的影子，整个单元和面向对象的关系似乎不大?
- 此外，相比于用Junit等本应该进行学习并使用的工具，用对拍机和hxd对拍似乎来的更加方便简洁，但这样似乎就背离了整个单元的初衷。
- 总的来说收获还是挺大的，期待第四单元的课程内容~ ┗| ｀O'|┛