# 面对对象 单元总结

## 一、架构设计、图模型构建

### 第1次作业

**实现自己的 `Person`、`Network`和`Group`类和异常类。**

图模型构建：需要实现社交网络`Network`内的连通块，只要能通过一条关系链链接上的`Person`都在同一个连通块内。

具体策略：初始情况下，每个人都是独立的个体，此时尚未产生关系，为每个人先各自分配一个连通块Block。当为两人添加关系addRelation后，查看这两个人的连通块号是否相同，如果不同则将这两个连通块合并，因为这两个连通块内的任意两个人都可以通过Linked的这两个人连通。

存储方面，为了加快存取速度和避免增删改过程中出现遗漏，使用`hashmap`容器存储同一个事物的一对属性，如：`Person`中的`acquaintance <personId,value>`，`Group`中的`people <personId, Person>`，`Network`中的`people <personId, Person>`、`groups <groupId, Group>`。

### 第2次作业

本次作业增加了`Message`类，我的做法是先用`gitlab`比较`JML`规格与上次作业的差异，再对方法进行修改；按照规格增加新的类；最后完成`Network`类即可。

图模型构建：查询最小关系（查询一个连通块的最小生成树）

具体策略：采用并查集实现`kruskal`算法

```
public class Dsu {
    private int[] size;
    private int[] root;

    public Dsu(int n) {
        size = new int[n + 1];
        root = new int[n + 1];
        Arrays.fill(size, 1);
        for (int i = 0; i < n + 1; i++) {
            root[i] = i;
        }
    }

    public int find(int x) {    // 递归寻找根节点
        if (root[x] != x) {
            root[x] = find(root[x]);
        }
        return root[x];
    }

    public boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) {
            return false;
        }
```

```
        // 路径压缩合并，如果合并成功证明这条边的两个节点原来不属于同一个联通分支，即这条边可以加入，不会形成环
        if (size[rootX] < size[rootY]) {
            root[rootX] = rootY;
            size[rootY] += size[rootX];
        } else {
            root[rootY] = rootX;
            size[rootX] += size[rootY];
        }
        return true;
    }
}
```

```
// 将整个block传入
public int krus(int n, ArrayList<Integer> nodes) {
        Dsu dsu = new Dsu(n);
        // a[0]:节点1     a[1]:节点2     a[2]:边权
        PriorityQueue<int[]> heap = new PriorityQueue<>((a,b) -> a[2] - b[2]);
        // 遍历block中所有有关系的成对节点，并把每条边加入到PriorityQueue中
        for (int i = 0; i < n; i++) {
            MyPerson p1 = (MyPerson) getPerson(nodes.get(i));
            for (int j = i + 1; j < n; j++) {
                MyPerson p2 = (MyPerson) getPerson(nodes.get(j));
                if (p1.isLinked(p2)) {
                    heap.offer(new int[]{i, j, p1.queryValue(p2)});
                }
            }
        }
        // 看是否可以合并
        int res = 0;
        while (!heap.isEmpty()) {
            int[] cur = heap.poll();
            if (dsu.union(cur[0], cur[1])) {
                res += cur[2];
            }
        }
        return res;
    }
```

**第3次作业**

本次作业新增了`EmojiMessage`、`NoticeMessage`和`RedEnvelopeMessage`类，均是继承了上一次作业完成的`Message`类，所以只需要完成各自特殊的属性和方法即可。其余做法同上一次作业。

图模型构建：在分发间接消息`sendIndirectMessage`方法中，需要查询两点间最短路

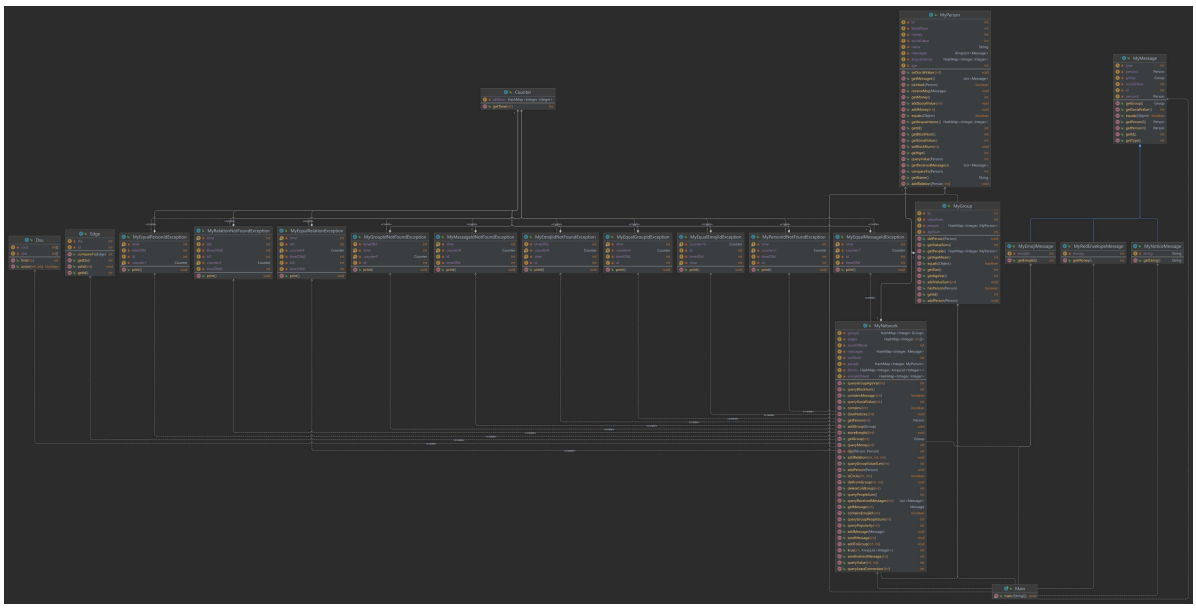
具体策略：使用堆优化的`dijkstra`算法，构造实现了`Comparable`接口的`Edge`类，使用优先队列实现本方法。

```
public class Edge implements Comparable<Edge> {
    private int id; //邻接结点:person id
    private int dis;//queryValue
```

```
    public Edge(int id, int dis) {
        this.id = id;
        this.dis = dis;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getDis() {
        return dis;
    }

    @Override
    public int compareTo(Edge o) {
        return this.dis - o.dis;
    }
}

private int dijs(Person p1, Person p2) {
        PriorityQueue<Edge> queue = new PriorityQueue<>();
        // 记录每个点到p1的dis
        HashMap<Integer, Integer> id2dis = new HashMap<>();
        HashSet<Integer> visitedId = new HashSet<>();
        queue.add(new Edge(p1.getId(), 0));
        while (!queue.isEmpty()) {
            Edge edge = queue.poll();
            if (edge.getId() == p2.getId()) {
                // dis: 从p1->p2的最短边权和
                return edge.getDis();
            }
            if (visitedId.contains(edge.getId())) {
                continue;
            }
            visitedId.add(edge.getId());
            int base = edge.getDis();
            MyPerson p = ((MyPerson) getPerson(edge.getId()));
            p.getAcquaintance().forEach((linkedId, weight) -> {
                // 以base(最小值) 更新 新加入结点的邻接结点的dis
                int key = linkedId;
                int value = weight + base;
                if (!visitedId.contains(key)
                        && (!id2dis.containsKey(key) || id2dis.get(key) >
value)) {
                    id2dis.put(key, value);
                    queue.add(new Edge(key, value));
                }
            });
        }
        return -1;
    }
```

1、每次从未标记节点中选择距离出发点最近的节点，标记为visited，收录到最优路径集合中。
2、计算刚加入节点A的邻近节点B的距离（不包含标记过的节点），若（节点A的距离+节点A到节点B的边长）< 节点B的距离，就更新节点B的距离。

**类图：**



## 二、代码实现出现的性能问题和修复情况

### 第一次作业

```
query_circle id(int) id(int)
query_block_sum
```

本次作业的性能问题主要是`query_circle`和`query_block_sum`这两个方法出现问题。

降低复杂度的方法是将这两个方法的查询时间平摊到每次加人和加关系上，维护一个**HashMap<Integer, ArrayList> blocks**， 记录的是连通块号和这个块内所有人的id。

每次`addPerson`时：创建一个新的`block`，此时`block`内只有他自己。

每次`addRelation`时：这两个人原本不在一个`block`内，需要将这两个`block`合并，即将一个`block`内所有`id`拷贝到另一个`block`，并将`numOfBlock`--。

经过以上处理，查询时只有O(1)复杂度：`query_circle`返回两个人所在`block`连通块号是否相同；`query_block_sum`返回连通块的个数。实现见第一部分。

### 第二次作业

```
query_group_value_sum id(int)
query_least_connection id(int)
```

本次作业的性能问题主要是`query_group_value_sum`和`query_least_connection`这两个方法出现问题。

`query_group_value_sum`：同样是通过增删时维护`group_value_sum`和`ageSum`来降低查询时的时间复杂度，同样可以做到`O(1)`查询。

```
public void addPerson(Person person) {
    people.put(person.getId(), (MyPerson) person);
    ageSum += person.getAge();
    for (MyPerson p : people.values()) {
        // renew valueSum
        if (p.isLinked(person)) {
            valueSum += (p.queryValue(person)) * 2;
        }
    }
}

public void delPerson(Person person) {
    people.remove(person.getId());
    ageSum -= person.getAge();
    for (MyPerson p : people.values()) {
        if (p.isLinked(person)) {
            valueSum -= (p.queryValue(person)) * 2;
        }
    }
}

//不要忘记在Network中添加关系后，若两人同属于一个Group也要修改valueSum的值！
for (Group group : groups.values()) {
    if (group.hasPerson(person1) && group.hasPerson(person2)) {
        ((MyGroup)group).addValueSum(value * 2);
    }
}
```

`query_least_connection`： `JML`语言叙述比较复杂，给出的`result`数组中相邻两个点（2*k*与2k+1）表示一条边权，数组大小为偶数，本质上是寻找一个`block`中的最小生成树。实现见第一部分。

**第三次作业**

```
send_indirect_message id(int)
```

`send_message`要求收发信息的两人必须`isLinked`，而`send_indirect_message`只需要两人`isCircle`，即寻找两人之间的最短路径。实现见第一部分。

## 三、基于JML规格的测试和bug分析

**第一次作业**

鉴于此前测试经验几乎为0，所以在尝试使用单元测试未果后就没有进行测试...只是在实现了全部功能后，重新读规格检查自己的实现和规格有无出入，以及复杂度比较高的查询是否优化。本想着第一次作业方法不多，肉眼比对下来也没觉得有什么问题，结果强测结果给我狠狠上了一课。异常处理中有一个需要排序的地方忘记修改导致强测挂了一半，果然，不要靠近未经检测的代码...

**第二次作业**

有了第一次作业的前车之鉴，第二次作业测试自己终于构造了极其简陋的数据生成器，大概思路是：第一次作业中的指令已经经过强测的检验就不用专门进行测试了，将新增指令分为互有关联的几组进行伪单元测试，借一份大佬的代码用bat脚本进行对拍。但也并没有在自测过程中发现bug。

**第三次作业**

仍然采用对拍，发现了一个bug，方法返回容器的clone而非容器本身导致的无效修改问题。互测被发现一个bug，在群发红包指令时由于小组成员(p)和发红包的人(p1)的命名不当导致p1误写成了p。（发红包发了个寂寞...）但侥幸如此车祸的bug没有被强测测出来...这同时也昭示了本人的测试确实是漏洞百出，只是机械的把指令堆积在了一起，没有按照逻辑把每种情况分类并且测试。实际上`sendMessage`、`sendIndirectMessage`，群发和私聊组合起来总共只有四种情况，分别测试也不复杂...

## 四、Network扩展及相应的JML规格

**选择3个核心业务功能的接口方法撰写JML规格：**

- `sendAd: Advertiser 持续向外发送产品广告`

  `Advertisement`类似`message`，需要广告商先编辑好存储在`Network`里，再投放给指定用户。

```
/*@ public normal_behavior
  @ requires containsAdvertisement(id);
  @ assignable advertisements, people;
  @ ensures !containsAdvertisement(id) && advertisements.length ==
\old(advertisements.length) - 1 &&
  @         (\forall int i; 0 <= i && i < \old(advertisements.length) &&
\old(advertisements[i].getId()) != id;
  @         (\exists int j; 0 <= j && j < advertisements.length;
advertisements[j].equals(\old(advertisements[i]))));
  @ ensures (\forall int i; 0 <= i && i < people.length;
people[i].isFavorable(id) &&
people[i].isFollowed(advertisements.get(id).getAdvertiser()) ==>
  @             (\exists int j; 0 <= j && j <
people[i].getAdvertisementsLength(); people[i].advertisements[j] == id) &&
  @              people[i].getAdvertisementsLength() ==
\old(people[i].getAdvertisementsLength()) + 1);
  @*/
public void sendAd(int id);
```

- 关注广告商

```
/*@ public normal_behavior
  @ requires contains(customerId);
  @ requires contains(advertiserId);
  @ assignable getPerson(customerId);
  @ ensures getPerson(customerId).getAdvertisersLength() ==
\old(getPerson(customerId).getAdvertisersLength()) + 1 &&
  @             (\exists int i; 0 <= i && i <
getPerson(customerId).getAdvertisersLength();
getPerson(customerId).getAdvertiser().get(i).getId() == advertiserId);
  @*/
public void follow(int customerId, int advertiserId);
```

- 购买产品，`Customer`直接通过`Advertiser`给相应`Producer`发一个购买消息

```
/*@ public normal_behavior
  @ requires contains(customerId);
  @ requires contains(advertiserId);
  @ requires contains(producerId);
  @ assignable people;
  @ ensures getPerson(advertiserId).getNoticesLength() ==
\old(getPerson(advertiserId).getNoticesLength()) + 1;
  @ ensures getPerson(producerId).getNoticesLength() ==
\old(getPerson(producerId).getNoticesLength()) + 1;
  @ ensures (\exists int i; 0 <= i && i <
getPerson(advertiserId).getNoticesLength(); \advertisements[i].getId ==
orderId);
  @ ensures (\exists int i; 0 <= i && i <
getPerson(producerId).getNoticesLength(); \advertisements[i].getId == orderId);
  @*/
public void buyProduct(int customerId, int advertiserId, int producerId, int
orderId);
```

## 五、本单元学习体会

总体回顾，本单元接触了JML规格，在几次作业和实验中逐步体会了按照规格实现方法和自己编写规格。从一开始看着规格描述丈二和尚摸不着头脑，到傻傻地原封不动地模仿JML给的数组挨个实现，再到阅读了学长学姐的博客知道JML只是选择了一种规范的过程表述体现用户需求，追求的是最终需要达到的“结果”，中间过程如何具体实现和优化没有做严格的约束。对于一些普通方法，只要JML规格书写的严谨规范，实现起来容易且不容易漏掉细节；然而对于一些复杂方法，规格读起来就很令人头大了，比如作业中对于最小生成树的建模，如果不加以自然语言的辅助说明其实挺难理解，所以二者结合可能能更好地平衡严谨性和可阅读性吧。 本单元最大的体会还是在于测试，由于本人第一单元全部寄的很彻底，第二单元重点在于多线程bug，导致对于测试完全0经验。而本单元数量较多的方法和规格中的许多细节可能导致稍有不慎就漏掉或写错，可以说只要能做到完备的测试本单元就完全没有问题。