OO_Unit3总结

测试相关

单元测试

单元测试是对软件中最小的可测试单元（通常是函数或方法）进行的测试，验证其在隔离环境下的正确性。目标是确保每个单元按预期工作，独立于其他组件。我们在本单元中所做的Junit测试就是简单的单元测试。

功能测试

功能测试验证软件是否满足指定的功能需求，通常从用户或系统功能的角度测试。关注系统对外提供的功能，而非内部实现细节。

集成测试

集成测试验证多个模块或组件协同工作时的正确性，关注模块间交互和数据流。目标是发现接口、通信或依赖问题。比如第一次作业中对`queryTripleSum()`方法的测试涉及到`Network`、`Person`和`DisjointSet`的交互，测试`addRelation()`方法对`tripleSum`的影响。

压力测试

压力测试评估系统在极端负载或资源限制下的性能和稳定性，验证其在高压力场景下的行为。目标是发现性能瓶颈、资源泄漏或崩溃点。由于本单元的中测Junit测试限制了测试轮次，并未进行真正的压力测试，测试10000人或者1000000条边的网络等情况就是进行了压力测试。

回归测试

回归测试验证代码修改后，现有功能是否仍然正常工作，防止新变更引入错误（回归 bug）。目标是确保系统整体稳定性。

数据构造策略

在本单元中有两个构造策略，分别针对中测Junit测试和强测程序时间复杂度，前者是后者的基础。

对于前者，我更注重于构造全面的数据，尽可能地覆盖边界情况。以第三次作业对`deleteColdEmoji()`方法的测试为例，首先进行黑盒测试，生成随机关系网络进行发送消息，对该方法进行测试，测试覆盖到全部消息类型：红包消息、emoji消息和转发消息，然后在进行白盒测试，对于emoji消息全面测试，构造emoji热度分别大于limit、小于limit、等于limit以及为0的数据进行测试，从而保证能覆盖到所有的情况。

对于后者，我更注重于重复的压力测试，对于程序中时间复杂度比较高的方法，就构造最大数量的数据然后重复调用该方法，观察CPU时间是否过长。第二次作业的strong19测试点就是如此，构造300人的关系网络然后重复调用`queryTagValueSum()`方法，目的就是检测程序中该方法的时间复杂度是否过大。

大模型的使用

在本单元中我尝试着使用了大模型检查代码实现是否和JML规格相符，并尝试使用大模型直接翻译JML，如果直接将代码丢给大模型让它去做，而这效果均不会很好，但是如果是渐进式询问，就能达到一个比较好的结果，我总结了以下步骤：

1. 首先向大模型提出自己的目的，将代码交给大模型（最好是直接上传代码文件），询问大模型有何想法。
2. 根据大模型给出的建议，选出你觉得最好的一条，然后让它给出实现步骤，并将实现步骤讲解给你，如果此时你发现了实现步骤是正确的，继续，否则，指出错误并让其修正。
3. 让大模型顺着实现步骤一步一步实现，如果是直接让大模型一次性解决大规模的任务，效果并不好。
4. 在本地实现大模型给出的结果，进行测试，如果出现问题，将错误内容反馈给大模型让其修改（最好是自己在本地调试一遍，尽可能将错误结果讲清楚），重复此步骤直到正确。关于测试，也可以由大模型实现。

架构设计

本单元架构已经由JML确定，Network类维护全局关系网络，相当于图，Person类管理人员信息，相当于节点，Person之间会产生不同的关系，相当于边，而Tag类和各Message类负责处理群组和消息传递。

图的构建

通过Network类的`addPerson()`、`addRelation()`和`modifyRelation()`来构建节点和带权边以及修改带权边，使用`DisjointSet`类并查集来维护连通性。

维护策略

动态维护

`queryTripleSum()`、`queryTagAgeVar()`、`queryTagValueSum()`方法分别增量更新`tripleSum`、`ageSum`和`valueSum`，实现动态维护，大大降低了时间复杂度，但增加了维护成本。

TreeMap和TreeSet排序

`queryBestAcquaintance()`和`queryBestContributor()`均使用这种方法实现。

对于`queryBestAcquaintance()`，在`Person`类中新建一个用于储存按关系降序排列的`acquaintances`TreeSet集合，在添加和修改关系时要对这个集合进行维护，在`queryBestAcquaintance()`方法中直接返回该集合的第一项即可，时间复杂度降为了$O(1)$，但是增加了维护成本。

```
    private final TreeSet<PersonInterface> acquaintanceSort;

    this.acquaintanceSort = new TreeSet<>((p1, p2) -> comparePersons((Person) p1,
            (Person) p2, this));

    public int getBestAcquaintanceId() {
        return acquaintanceSort.first().getId();
    }

    private int comparePersons(Person p1, Person p2, Person owner) {
        if (p1.equals(p2)) {
            return 0;
        }
        int valueCompare = owner.queryValue(p2) - owner.queryValue(p1);
        if (valueCompare != 0) {
            return valueCompare;
```

```
        }
        return Integer.compare(p1.getId(), p2.getId());
    }
```

对于`queryBestContributor()`方法，则是维护这样的数据结构：`TreeMap<Integer, TreeSet<Integer>>`，键是文章贡献度，值则是人员ID的集合，TreeMap按降序排序，而TreeSet按升序排序，同时在添加文章和删除文章时维护该容器，最后只需返回第一贡献度的第一位人员的ID即可。

```
    private TreeMap<Integer, TreeSet<Integer>> contributionToIds;

    this.contributionToIds = new TreeMap<>(Collections.reverseOrder());


    public int getBestContributor() {
        if (contributionToIds.isEmpty()) {
            return Integer.MAX_VALUE;
        }
        return contributionToIds.firstEntry().getValue().first();
    }
```

性能问题

在本单元中未出现因为性能问题而出现CTLE，但是在第二次作业强测的反馈结果上看，我的`queryTagValueSum()`方法时间复杂度有些偏高，对于反复调用该方法的测试点（比如strong19），CPU时间会飙升至7s，因此在第三次作业中我对该方法进行了修改。

原本我的实现方式是两层for循环，但是并非直接按照JML实现：

```
        for (PersonInterface person : persons.values()) {
            Person person1 = (Person) person;
            for (PersonInterface acquaintance :
person1.getAcquaintance().values()) {
                if (hasPerson(acquaintance)) {
                    valueSum += person1.queryValue(acquaintance);
                }
            }
        }
```

所以我并不是遍历了两边`Tag`类中的`persons`，而是在第二次循环时遍历第一次遍历的人的熟人，考虑到最多300人而且指令最多10000条，并且在询问大模型之后决定不进行修改（懒）。

第三次作业中我还是修改成了动态维护的策略，在`Tag`类中维护`valueSum`，在每次`addPerson`、`delPerson`、`addRelation`和`modifyRelation`时对该变量进行维护，在本地测试时，这个方法确实能够大大降低该方法的时间复杂度。

规格与实现分离的理解

规格定义了方法行为，但是不限制实现方式，实现则是依照规格，根据自己的需求（如让时间复杂度尽量小）提供具体方案。规格与实现分离能够提高模块化、测试独立性和实现灵活性，本单元的Junit测试就是基于JML规格验证功能。

总的来说，规格是系统设计的基石，分离允许个人特定需求不破坏功能，测试是连接规格与实现的桥梁。

Junit测试

JML规格语言的一个好处就是便于测试，在测试时首先要构造数据，然后只需要按照JML规格得出标准结果，然后调用被测试的方法，使用assert检查而这是否相等即可，同时还要检查后置条件以及pure条件等。

数据构造在于全面性，在黑盒测试之外还要补充白盒测试，保证边界条件也可以被测试到。

至于正确性验证方面，只需完全按照JML规格的表述得出标准结果，然后调用被测试方法，检查而这是否相等，同时不要忘了检查后置条件、pure条件以及不变量。

以第一次作业的测试为例：

```
    /**
     * 计算社交网络中三角关系网的数量
     */
    private int calculateTripleSum(PersonInterface[] persons) {
        int count = 0;
        for (int i = 0; i < persons.length; i++) {
            int id1 = persons[i].getId();
            HashMap<Integer, PersonInterface> neighbors1 = ((Person)
persons[i]).getAcquaintance();
            for (PersonInterface p2 : neighbors1.values()) {
                int id2 = p2.getId();
                if (id2 > id1) {
                    HashMap<Integer, PersonInterface> neighbors2 = ((Person)
p2).getAcquaintance();
                    for (PersonInterface p3 : neighbors2.values()) {
                        int id3 = p3.getId();
                        if (id3 > id2 && persons[i].isLinked(p3)) {
                            count++;
                        }
                    }
                }
            }
        }
        return count;
    }
    public void testQueryTripleSum() {
        PersonInterface[] oldPersons = ((Network) oldNetwork).getPersons();
        int stdAnswer = calculateTripleSum(oldPersons);
        int testAnswer = newNetwork.queryTripleSum();
        assertEquals("Triple sum mismatch", stdAnswer, testAnswer);

        PersonInterface[] newPersons = ((Network) newNetwork).getPersons();
        assertEquals("Person count mismatch", oldPersons.length,
newPersons.length);
        for (int i = 0; i < oldPersons.length; i++) {
            assertPersonEquals(oldPersons[i], newPersons[i]);
        }
        for (int i = 0; i < oldPersons.length; i++) {
            for (int j = 0; j < oldPersons.length; j++) {
                if (i != j) {
```

```
                    assertEquals("Relation mismatch",
                            oldPersons[i].isLinked(oldPersons[j]),
                            newPersons[i].isLinked(newPersons[j]));
                }
            }
        }
    }
```

学习体会

这一单元确实比前两单元要仁慈不少，但也是收获满满。

经过本单元的学习，我初步接触到了契约式设计和JML规格，也确实感受到了规格给代码设计带来的好处：严谨周密，但是至于便利性，我更倾向于规格语言和自然语言互补，同时给出规格语言和自然语言可能会在开发过程中更利于开发者设计代码，当然对于初学者而言只给出规格语言是必要的，毕竟要学习规格语言。

同时在本单元我也是第一次手动实现随机数据生成实现测试（感谢第一次上机实验），也算是一定程度上弥补了没有搭建评测机的遗憾。