BUAA_OO_第三单元总结

前言

主题是 “规格化设计”，要求我们学会理解 JML 规格语言，并能基于规格进行代码实现。规格仅仅是一种**契约**，针对一种特定的规格可能会有很多具体技术的实现方法，尤其是，我们在编程时要特别注意**代码运行的效率**（否则很容易TLE）。

具体而言，我们的目标：**至少保证每一个方法不能超过 O(n) 级别复杂度。**如果某个方法复杂度是 O(n)，我们面前保留；如果超出，必须要想办法进行优化（并查集、按秩合并、动态维护、缓存法等）。

本单元让我在 “面向规格编程” 的过程中感受到 “契约式编程” 的魅力——**可靠性、可复用性、契约化测试。**

常见 JML 表达式

- requires子句定义该方法的前置条件
- assignable列出这个方法能够修改的类成员属性，即副作用范围限定
- ensures子句定义了后置条件
- \result表达式：表示一个非 void 类型的方法执行所获得的结果，即方法执行后的返回值。
- \old(expr)表达式：用来表示一个表达式 expr 在相应方法执行前的取值。

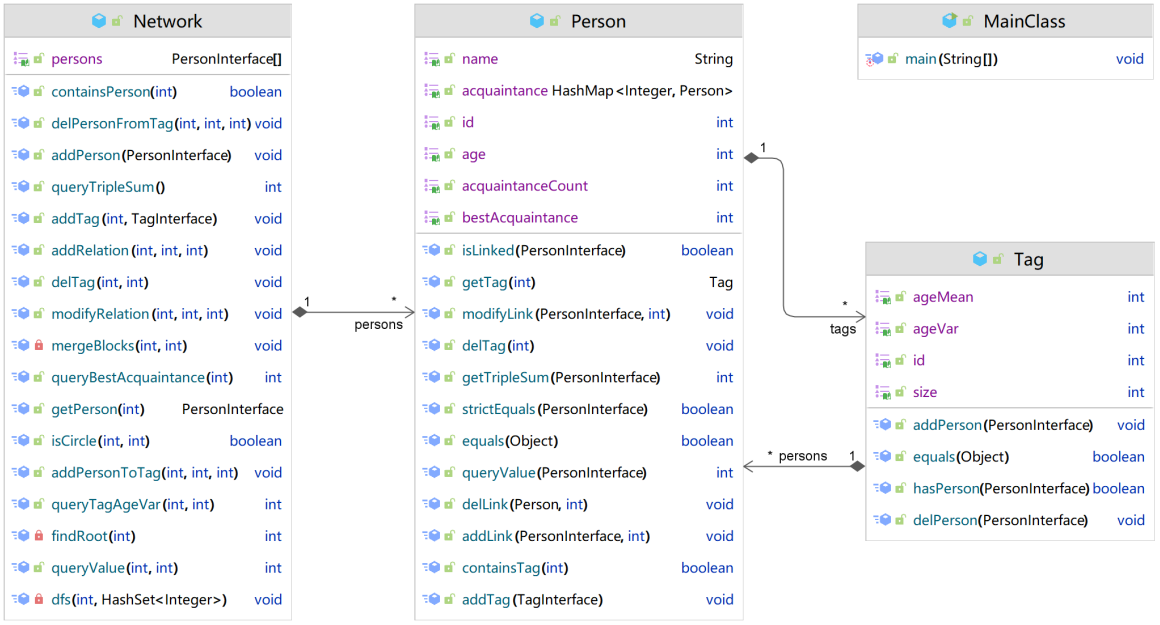
第一次作业

整体架构

要求我们根据规格，实现简单社交关系的模拟和查询。由于每个方法功能契约均完全给出，因此不必完全理解网络的具体细节。但是认真阅读 JML 规格、仔细理解网络的具体功能，对七月式编程的理解、具体代码功能维护、测试的导向性等都有好处。

同时，要善于对每个类创建新的属性和方法，在满足规格的前提下简化实现、增强效率。

UML类图



优化方法

- **容器的选择。**本次作业绝大多数类的实例化对象的 `id` 属性都是独一无二的。因此，选择`HashMap`作为存储容器，将`id`作为键值，这样可以保证 O(1) 级别的查找删除复杂度。
- **动态维护。**本次作业有许多计算量（例如 `Person` 中的 `ageSum`、`ageVar`、`Network`中的`tripleSum` 等）。为了简化实现，我们对 O(n) 级别及以下的属性不做动态维护，而是每次取的时候直接计算。但是，`Network` 中的 `tripleSum` 属性必须要进行维护（否则涉及到至少三层遍历）。分析发现，以下方法执行时需要更新 `tripleSum`：`addRelation`、`modifyRelation`。其实还可以做一个 O(n) 层次上的一个小小优化：**根据 `Person` 认识的对称性减少遍历量。**这样，`queryTripleSum` 只需要放回该属性即可：

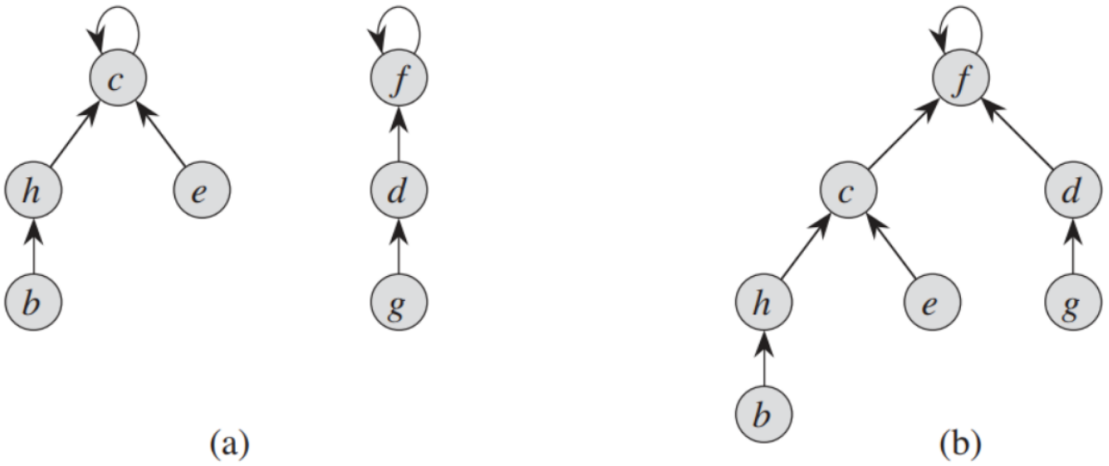
```
// 添加三元环示例实现
tripleSum += persons.get(id1).getAcquaintanceCount() <
             persons.get(id2).getAcquaintanceCount() ?
             persons.get(id1).getTripleSum(persons.get(id2)) :
             persons.get(id2).getTripleSum(persons.get(id1));

// 删除三元环示例实现
tripleSum += persons.get(id1).getAcquaintanceCount() <
             persons.get(id2).getAcquaintanceCount() ?
             persons.get(id1).getTripleSum(persons.get(id2)) :
             persons.get(id2).getTripleSum(persons.get(id1));

// queryTripleSum 实现
public int queryTripleSum() {
    return tripleSum;
}
```

- **并查集。**具体技术回顾图论中的内容。这个优化的目的是：优化 `isLinked()` 查询两个`Person`之间的连通性。并查集中也有许多高阶优化，例如**按秩合并。**

> 由于我们在找出一个元素所在集合的代表元时需要递归地找出它所在的树的根结点，所以为了减短查找路径，在合并两棵树时要尽量使合并后的树的高度降低，所以要将高度低的树指向高度更高的那棵。我们将树的高度称为秩，合并时将 **“小秩”集合的代表元**的直接上级设为 **“大秩”集合的代表元**。



我的实现如下（`merge` 方法）：

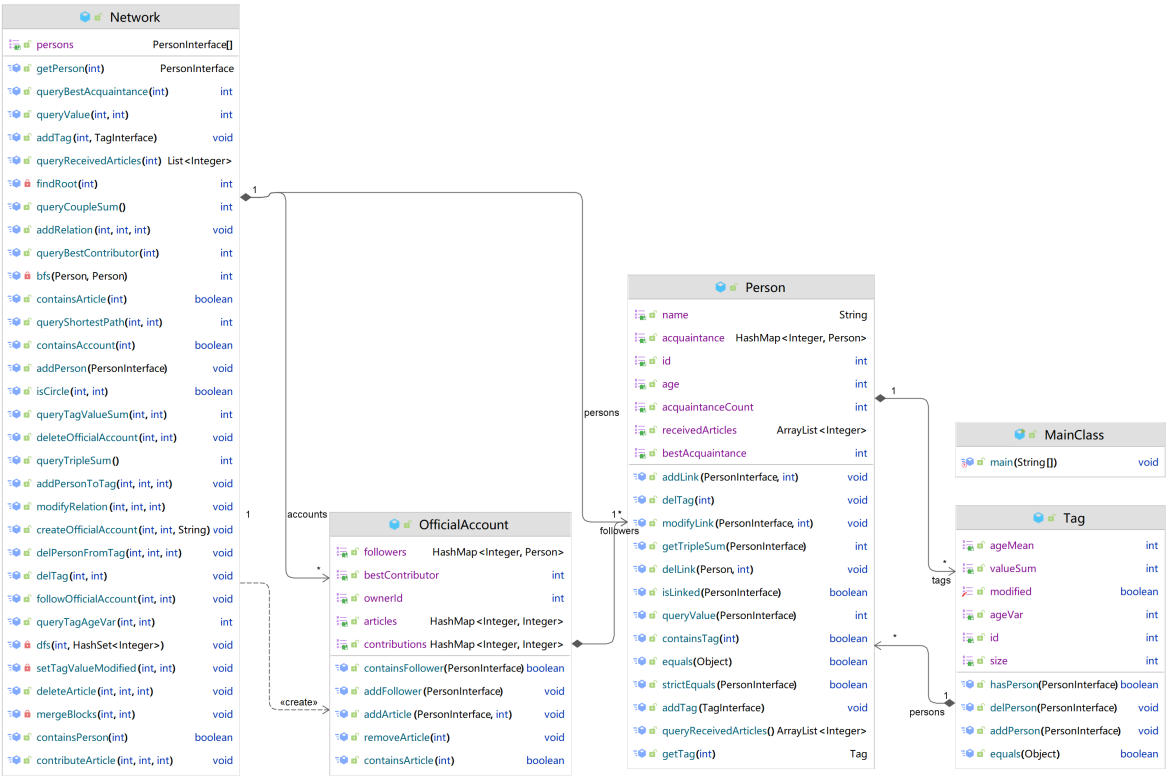
```
private void mergeBlocks(int id1, int id2) {
    int root1 = findRoot(id1);
    int root2 = findRoot(id2);
    if (root1 == root2) {
        return;
    }
    blockSum--;
    if (depthMap.get(root1) > depthMap.get(root2)) {
        rootMap.put(root2, root1);
    } else {
        rootMap.put(root1, root2);
        if (depthMap.get(root1).equals(depthMap.get(root2))) {
            depthMap.put(root2, depthMap.get(root2) + 1);
        }
    }
}
```

第二次作业

整体架构

新增了 `OfficialAccount` 类，涉及到更为复杂的、容易混淆的属性。本作业 JML 规格是三次作业中最为复杂的，其中有些充分必要条件的约束是自然成立，不需要我们添加代码满足。因此，阅读 JML 规格时要提炼重点、注重逻辑。

UML类图



优化方法

除了对于 BFS 算法的优化实现外（例如双向查询），本次作业的优化集中出现在 `qtvs` 指令中，也就是纠结于 `Tag` 中是否要动态维护一个 `ValueSum` 属性。我一开始做了动态维护，但连续好几天都有一个只有高强度数据才能出发的 bug 无法找出原因，遂放弃动态优化，采用2充遍历。但是频繁的 O(n^2) 算法必然导致 TLE，因此需要想出一些补救的措施。我采用了**缓存法**，即对于每个 `Tag` 对象维护一个标志位 `modified`，如果该标志位为脏，才重新计算，否则直接取上一次计算的结果。

- **核心注意。**该标志位必须至少在**每个** `Tag` 级别才有效。也就是说，必须**每个** `Tag` 各自有单独的、互不影响的标志位及自己的 `ValueSum` 属性。如果该标志位是全局的，那么无法起到任何优化效果。采取该方法的原因之一是我本身就在 `Network` 全局中维护了所有的 `Tag` ： `private static final HashMap<Integer, HashSet<Tag>> tagMap = new HashMap<>();` **注意** `Tag` **对象的** `id` **不是唯一的。**
- `Tag` **类中的维护。**基本操作如下：

```java
// 缓存字段
private int lastValueSum = 0;
private boolean modified = true;

public int getValueSum() {
    if (!modified) {
        return lastValueSum;
    }

    int sum = 0;
    for (Person person : persons.values()) {
        for (Person acquaintance : person.getAcquaintance().values()) {
            // sum = ...
        }
```

```
    }

    lastValueSum = sum;
    modified = false;
    return sum;
}
```

注意，`Tag` 类中的 `addPerson`、`delPerson` 方法都要使 `modified` 标志位为脏，同时还要维护一个公共方法 `setModified` 作为外界置位 。

- `Network` **中的维护。** `Network` 类中需要实现该方法用于修改置位：

```
private void setTagValueModified(int id1, int id2) {
    if (tagMap.containsKey(id1)) {
        for (Tag tag : tagMap.get(id1)) {
            tag.setModified(true);
        }
    }
    if (tagMap.containsKey(id2)) {
        for (Tag tag : tagMap.get(id2)) {
            tag.setModified(true);
        }
    }
}
```

同时在以下方法中调用该方法：`addRelation`、`modifyRelation`。

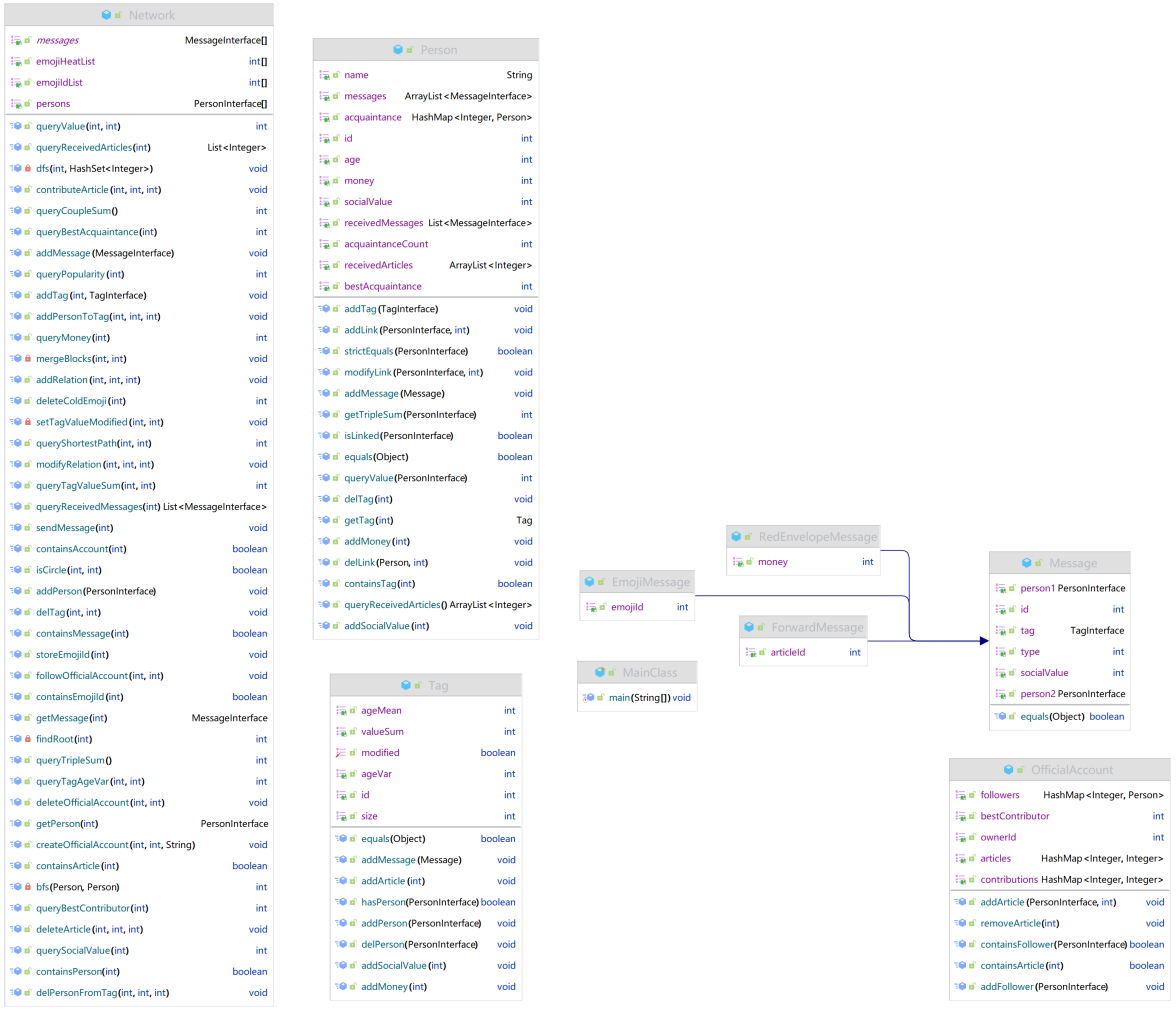
我一开始的错误在于对于`modifyRelation`方法，我只对 `queryValue(id1, id2) + value <= 0` 的情况使用该方法，对另一种情况置之不顾。**但其实另一种情况也必须维护（原因很细节）。**

第三次作业

整体架构

本次作业将 `Message`进行了细分，增加了继承自 `Message` 的许多新消息类型——`RedEnvelopeMessage`、`ForwardMessage`和 `EmojiMessage`。这些子类分别各自有 2 种不同的构造方法，只需要对照 JML 对代码稍作修改即可。

UML 类图



测试方法概览

本单元作业进行测试的唯一方法是评测机。具体而言，我对于每次作业实现了数据生成器和多人制对拍机。对于数据生成，有2种类别：**普通数据**和**死亡三角**。所谓死亡三角，就是通过 `1n` 构造完全图刷TLE。

本单元中我对测试方法有了全新的认识：

- **单元测试**

例如 junit 单元测试。单元测试是针对程序中的**最小可测试单元**进行的测试，一般来说，是对方法进行测试。它实际上也是白箱测试，因为我们默认是知道源代码的，通过测试我们检查代码的逻辑和输出是否正确。

- **功能测试**

功能测试只保证我们**测试的单元能够正常实现功能**，而不关心功能与功能之间的影响。只要正确地实现了功能，我们就认为功能测试是成功的。

- **集成测试**

用于**检查功能与功能之间的联系和影响**，必须要正确实现功能与功能之间的关系才能通过测试。

- **压力测试**

根据数据范围**构造极端数据**进行测试，比如测试 `TLE` 的情况，在本单元作业中的压力测试为`TLE`，通过反复使用规格中复杂度高的方法，检查是否`TLE`，从而将数据构造成最后一直使用某条指令，具有十足的特殊性，这就是压力测试。它也是我在hack时候经常采用的方法。

- **回归测试**

因为工程上一般是迭代开发的，所以有些时候难免保证实现了新的功能后原功能是否受到影响。所以迭代开发后要重新进行测试，保证新写的代码没有改变原有的代码功能。

- **数据构造有何策略**

通过随机数据生成器，大范围的生成一般性数据，充分覆盖测试范围，通过大规模的**随机数据**测试正确性和普适性。然后根据`JML`规格找出复杂度最高的方法，反复使用该方法，执行**压力测试**，测试程序在极端数据下的表现情况和正确程度。

大模型辅助进行规格化设计

三次作业课程组均建议我们合理使用LLM，我在和大模型交互过程中得出以下经验：

1. **尽量使用高阶模型。**高阶模型的推理能力和应用能力和普通模型不是一个级别，在 prompt 优化和反复迭代上做文章效果往往不如替换高阶模型。
2. prompt 设计和交互中，尽量采用“**原始功能需求+例子+反复迭代询问确认**”三步走模式，对于复杂方法，请大模型一定要对着逐条语句说出是否实现，否则大模型可能会遗漏条件或功能。
3. 对于极其复杂 JML（例如第三次作业中的`sendMessage`）他们往往涉及到多个类之间的交互作用，此时必须要进行**人工验证或直接人工书写。**

学习体会

本单元突出程序的规格化标准化设计，对工程应用实践具有重要指导和训练意义。

本单元我的最大收获是：对于方法的**功能和实现相分离**有了更深入的理解。具体而言，“功能”在本单元体现就是 JML 规格，而“实现”就是具体的算法。功能是永远固定的，但是实现可以千变万化。对于确定的功能，我们可以通过自行设计辅助类、引用高阶算法、利用动态维护或缓存方法提升性能等方式优化具体技术实现，达到更好的性能和应用效果。

本单元在面向对象课程学习中给我不一样的体验，极大转变和加深了我对高质量规格化编程的理解和应用。