OO第三单元总结

1. JML语言理论基础以及应用工具链情况梳理

1.1 JML语言理论基础

JML（Java Modeling Language）是一种用于规范Java程序行为的行为接口规范语言。JML为方法和类型的规格进行定义，为程序的形式化验证提供了基础，通过工具链可以实现静态检查和自动测试数据生成。

注释结构

- 行注释： `// @ annotation`
- 块注释：`/* @ annotation @*/`

JML表达式

- 原子表达式
  - \result，\old(expr)
- 量化表达式
  - \forall表达式：全称量词修饰的表达式
  - \exists表达式：存在量词修饰的表达式
  - \sum表达式：返回给定范围内的表达式的和
  - \max表达式：返回给定范围内的表达式的最大值
  - \min表达式：返回给定范围内的表达式的最小值
- 集合表达式
  - 可以在JML规格中构造一个局部的集合（容器），明确集合中可以包含的元素。
- 操作符
  - 子类型关系操作符： E1<:E2，
  - 等价关系操作符： b_expr1<==>b_expr2 或者 b_expr1<=!=>b_expr2
  - 推理操作符： b_expr1== >b_expr2 或者 b_expr2< ==b_expr1
  - 变量引用操作符：\nothing指示一个空集；\everything指示一个全集

方法规格

- 前置条件(pre-condition)
  - 通过requires子句来表示： requires P;
- 后置条件(post-condition)
  - 通过ensures子句来表示： ensures P;
- 副作用范围限定(side-effects)
  - 使用关键词 assignable 或者 modifiable

类型规格

- 不变式invariant
- 状态变化约束constraint

1.2 工具链简介

与规格化设计相关的工具主要有:OpenJML, JMLUnit, TestNG等等。

- OpenJML

> 对Java 编译器将带有 JML 规范注释的 Java 程序编译成 Java 字节码。 编译的字节码包括检查的运行时断言检查指令。

- JMLUnit与TestNG

> JMLUnit 与 TestNG主要是基于JML规格生成样例并自动化进行单元测试

2. OpenJML的简单使用：部署、验证及结果分析

参考伦佬的贴子。

2.1 JML规格语法check

源码

```
public class Demo {
    /*@
      @ public normal_behaviour
      @ requires lhs<0 &&
    */
    public static int compare(int lhs, int rhs) {
        return lhs - rhs;
    }

    public static void main(String[] args) {
        compare(114514, 1919810);
    }
}
```

结果分析

```
$ ./openjml -check Demo.java
Demo.java:5: 错误: The type or expression near here is invalid (or not implemented): (token <JMLEND> in JmlParser.term3())
    */
    ^
1 个错误
```

检测出JML表达式无效。

2.2 基于OpenJML的静态bug检测

源码

```
public class Demo {
    /*@
      @ public normal_behaviour
      @ requires lhs > rhs;
      @ ensures \result > 0;
      @ also
      @ requires lhs < rhs;
      @ ensures \result < 0;
      @ also
      @ requires lhs == rhs;
      @ ensures \result == 0;
      @*/
    public static int compare(int lhs, int rhs) {
        return lhs - rhs;
```

```
    }

    public static void main(String[] args) {
        compare(114514, 1919810);
    }
}
```

结果分析

```
$ ./openjml -check Demo.java
$ ./openjml -esc Demo.java
Demo.java:14: warning: The prover cannot establish an assertion
(ArithmeticOperationRange) in method compare:  underflow in int difference
        return lhs - rhs;
                   ^
demo/Demo.java:14: warning: The prover cannot establish an assertion
(ArithmeticOperationRange) in method compare:  overflow in int difference
        return lhs - rhs;
                   ^
2 warnings
```

检测结果显示JML表达式合法，但存在减法溢出的漏洞。

2.3 运行时检查

源码

```
public class Demo {
    /*@
      @ public normal_behaviour
      @ requires lhs > rhs;
      @ ensures \result > 0;
      @ also
      @ requires lhs < rhs;
      @ ensures \result < 0;
      @ also
      @ requires lhs == rhs;
      @ ensures \result == 0;
      @*/
    public static int compare(int lhs, int rhs) {
        return lhs - rhs;
    }

    public static void main(String[] args) {
        compare(2147483647, -2147483647);
    }
}
```

结果分析

```
94831@LAPTOP-35U6BFJF MINGW64
/d/MyCollege/CourseCenter/Semester4/CC/OO/JML/testOpenJml
$ ./openjml -rac Demo.java

94831@LAPTOP-35U6BFJF MINGW64
/d/MyCollege/CourseCenter/Semester4/CC/OO/JML/testOpenJml
$ java -Djava.ext.dirs=lib Demo
Demo.java:14: 警告: JML result of numeric operation is out of range of the target type
        return lhs - rhs;
                   ^
Demo.java:13: 警告: JML postcondition is false
    public static int compare(int lhs, int rhs) {
                      ^
Demo.java:5: 警告: Associated declaration: Demo.java:13: 注:
      @ ensures \result > 0;
        ^
```

openjml 的使用和 javac 大致相同，先编译后运行即可。结果显示出现了减法溢出的情况。

3. JMLUnit的使用：部署、验证及结果分析

参考伦佬的贴子——第二篇。

以下是针对MyPath中的部分方法进行测试

3.1 生成测试文件

名称	修改日期	类型	大小
com	2019/5/22 17:49	文件夹	
MyPath.class	2019/5/22 19:26	CLASS 文件	14 KB
MyPath.java	2019/5/22 19:18	Java 源文件	2 KB
MyPath_ClassStrategy_int.class	2019/5/22 19:25	CLASS 文件	1 KB
MyPath_ClassStrategy_int.java	2019/5/22 19:22	Java 源文件	1 KB
MyPath_ClassStrategy_int1DArray.cla...	2019/5/22 19:25	CLASS 文件	1 KB
MyPath_ClassStrategy_int1DArray.java	2019/5/22 19:22	Java 源文件	2 KB
MyPath_containsNode__int_node__0...	2019/5/22 19:25	CLASS 文件	1 KB
MyPath_containsNode__int_node__0...	2019/5/22 19:22	Java 源文件	1 KB
MyPath_getNode__int_index__0__ind...	2019/5/22 19:25	CLASS 文件	1 KB
MyPath_getNode__int_index__0__ind...	2019/5/22 19:22	Java 源文件	1 KB
MyPath_InstanceStrategy.class	2019/5/22 19:25	CLASS 文件	2 KB
MyPath_InstanceStrategy.java	2019/5/22 19:22	Java 源文件	3 KB
MyPath_JML_Test.class	2019/5/22 19:25	CLASS 文件	5 KB
MyPath_JML_Test.java	2019/5/22 19:22	Java 源文件	11 KB
MyPath_MyPath__int1DArray_nodeLi...	2019/5/22 19:25	CLASS 文件	1 KB
MyPath_MyPath__int1DArray_nodeLi...	2019/5/22 19:22	Java 源文件	2 KB
PackageStrategy_int.class	2019/5/22 19:25	CLASS 文件	1 KB
PackageStrategy_int.java	2019/5/22 19:22	Java 源文件	1 KB
PackageStrategy_int1DArray.class	2019/5/22 19:25	CLASS 文件	1 KB
PackageStrategy_int1DArray.java	2019/5/22 19:22	Java 源文件	2 KB

3.2 编译

> 编译部分共分为两步：
>
> - 用 javac 编译 JMLUnitNG 的生成文件

> - 用 jmlc 编译自己的文件，生成带有运行时检查的 class 文件

```
94831@LAPTOP-35U6BFJF MINGW64 /d/MyCollege/CourseCenter/Semester4/CC/OO/JML/testOpenJml
$ javac -cp jmlunitng.jar src/*.java

94831@LAPTOP-35U6BFJF MINGW64 /d/MyCollege/CourseCenter/Semester4/CC/OO/JML/testOpenJml
$ ./openjml -rac src/MyPath.java

```

3.3 测试及结果分析

```
94831@LAPTOP-35U6BFJF MINGW64
/d/MyCollege/CourseCenter/Semester4/CC/OO/JML/testOpenJml/src
$ java -Djava.ext.dirs=../lib MyPath_JML_Test
[TestNG] Running:
  Command line suite

Passed: racEnabled()
Failed: constructor MyPath(null)
Skipped: <<null>>.containsNode(-2147483648)
Skipped: <<null>>.containsNode(0)
Skipped: <<null>>.containsNode(2147483647)
Skipped: <<null>>.getDistinctNodeCount()
Skipped: <<null>>.getNode(-2147483648)
Skipped: <<null>>.getNode(0)
Skipped: <<null>>.getNode(2147483647)
Skipped: <<null>>.isValid()
Skipped: <<null>>.size()

===============================================
Command line suite
Total tests run: 11, Failures: 1, Skips: 9
===============================================

```

由结果可知，MyPath的构造未通过测试，经查，发现，当传进来的nodeList为空时，在 `nodeList.length` 时可能会发生访问空指针的错误。

```
    public MyPath(int... nodeList) {
        for (int i = 0; i < nodeList.length; i++) {
            myNodes[i] = nodeList[i];
        }
    }
```

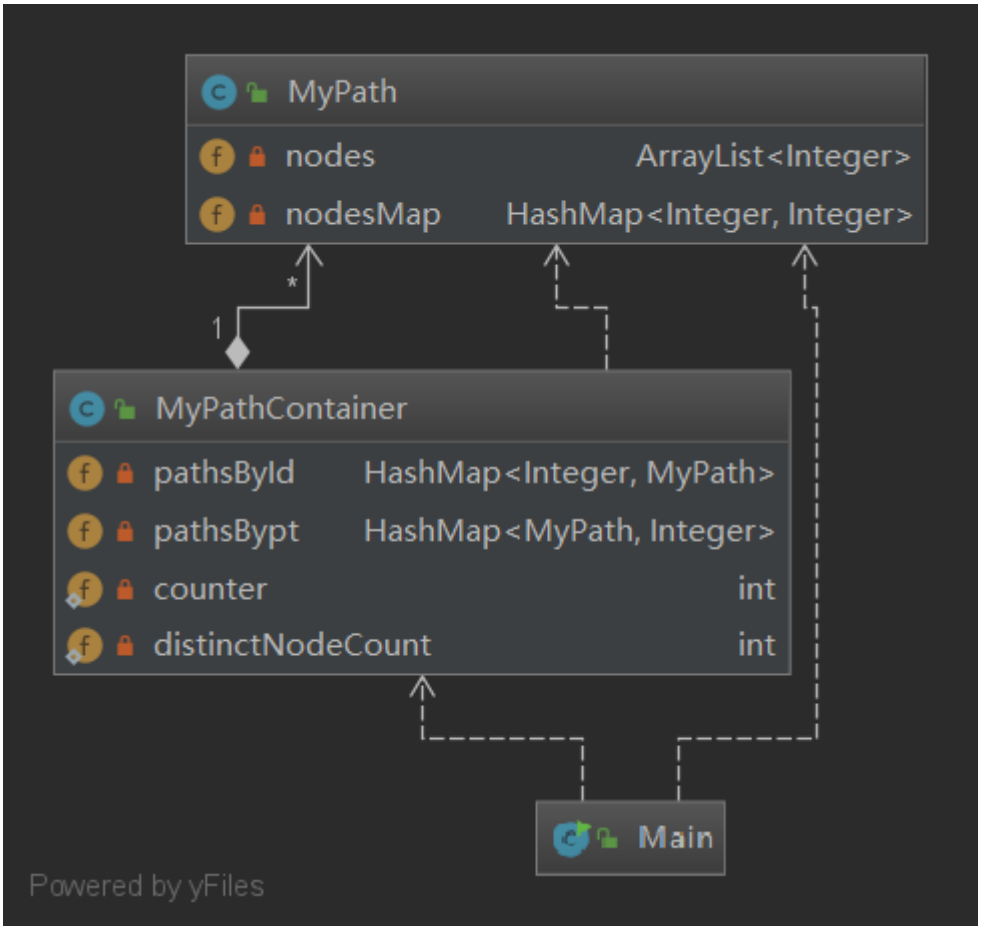
4. 作业架构设计梳理

4.1 第一次作业

分析

第一次作业较为简单，利用好HashMap实现 O(1) 的查询即可。

类图如下：



- MyPath
  - 使用 `ArrayList<Integer> nodes` 结构保存节点序列
  - 使用 `HashMap<Integer, Integer> nodesMap` 来保存结点及结点在path中的出现次数，以便由 `nodesMap.size()` 直接 `getDistinctNodeCount()`
- MyPathContainer
  - 使用 `HashMap<Integer, MyPath> id2Path` 和 `HashMap<MyPath, Integer> path2Id` 两个HashMap来实现 `Path` 与 `PathId` 的双向映射，从而快速索引。

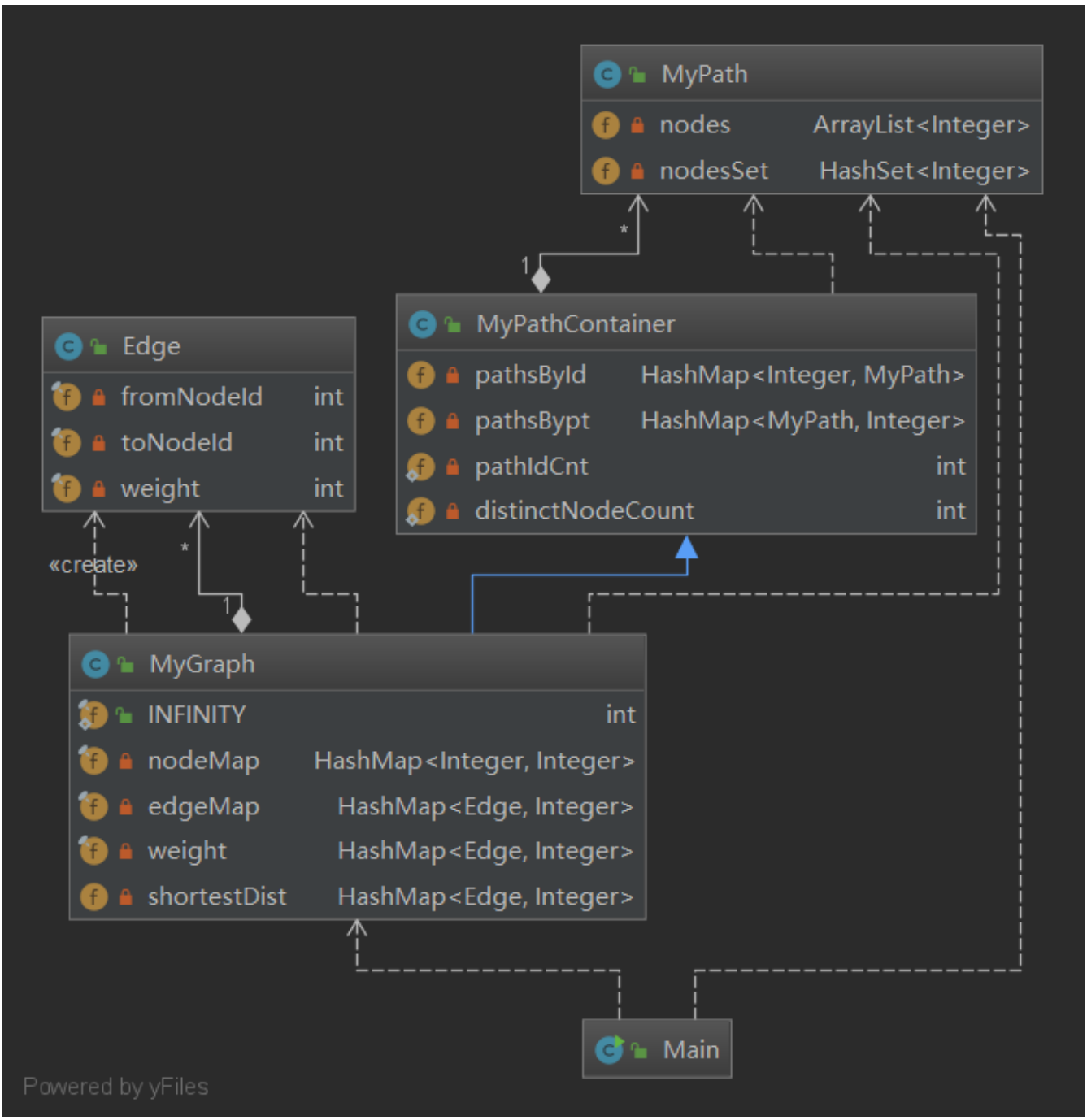
4.2 第二次作业

分析

- 第二次作业在 `PathContainer` 的基础上新增了 `Graph` 接口， `MyPathContainer`无需重构，直接继承即可。
- 需要注意的是由于增删指令不超过80条，查询指令将接近7000条，而评测又严格限制CPU时间，因此最短路的计算就需要一定的策略了。考虑到查询指令数，如果采用Floyd每次都把所有结点的最短路算出来，将会产生大量的性能浪费，同时由于其O(n^3)的复杂度，因而弃用。

- 考虑到第二次作业为**无边无权图**，本人采用 **BFS + Lazy Search** 算法。即类似缓存机制，将答案存储在“TLB”中，每次进行增删操作时由于图结构改变，需要清空TLB。每次查询先查TLB，若命中，直接返回；若MISS，就BFS一下同时更新TLB。

类图如下：



- Edge
  - 抽象出来的无向边，存储两个结点以及边权（本次作业恒为1）
  - 重写equals及hashcode方法
- MyGraph
  - 使用 `HashMap<Integer, Integer> nodeMap` 和 `HashMap<Edge, Integer> edgeMap` 来存储结点和边及其度数

- `HashMap<Edge, Integer> weight` 和 `HashMap<Edge, Integer> shortestDist` 分别存储权重即相应的最短路

4.3 第三次作业

本次作业难度有了较大的提升，且由于当周冯如杯答辩等原因导致时间紧迫，加上周一晚上盲从讨论区大佬的观点。在没有完全参悟拆点法的情况下，临时更换策略极限重构，结果在建图时遇到了很大的困难，最后因为时间关系草草完成了图的构建，由于结构混乱，且复杂度较高，导致强测大半CTLE。真应了助教那句：

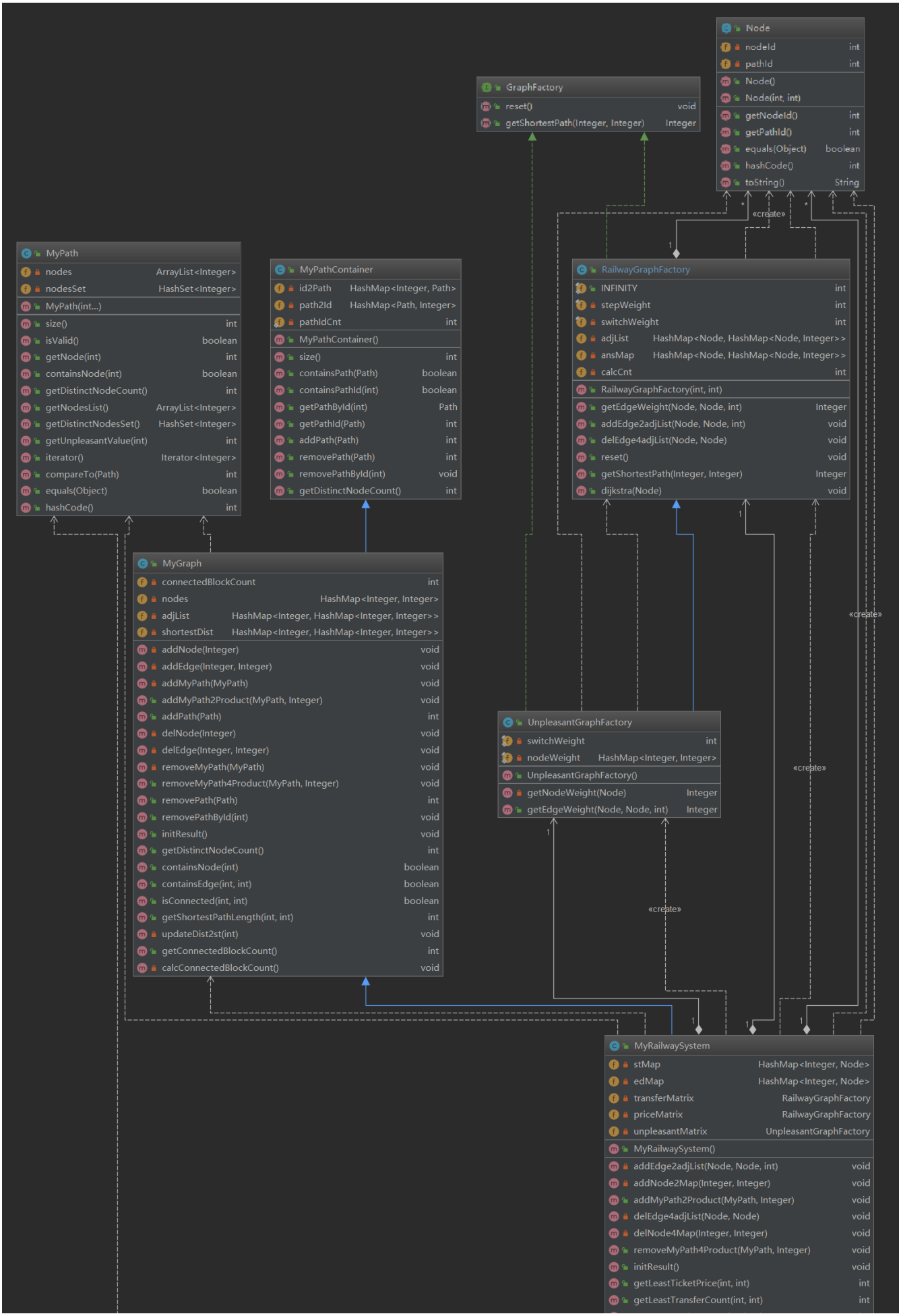
> **“重构一时爽，一次重构火葬场”**

由于没有将图结构的存储与答案的存储分离开，导致二者的耦合度较高，维护成本非常大； 因而在认真研究了标程的架构设计后，重构了图的存储逻辑，借鉴了标程的工厂模式，如下。

分析

- 本次作业采用了 **拆点 + Dijkstra + Lazy Search**，大体思路同上次作业，拆点遵从zyy大佬的`2+x`模式，具体见这里。
- 难点之一在于四个最短问题的求解，而它们的主要区别在于邻接边与换乘边的权值不同，设邻接边权为x，换乘边权为y，则有：
  - 最短路径：x = 1, y = 0 // 实际上，最短路径的求解我仍然放在了 MyGraph 中
  - 最少换乘：x = 0, y = 1
  - 最低票价：x = 1, y = 2
  - 最小不满意度：x = `MAX( F(fromNode), F(toNode) )`, y = 32;
- 难点之二则在于图的存储，本人采用了嵌套HashMap以邻接表的形式来存储。设计模式上，重构版采用了工厂模式，将三种最短问题抽象为同一种产品，不同的仅仅是 `stepWeight` 与 `switchWeight`，同时由于**最少不满意度**的邻接边权非固定值，需要单独对其进行拓展，处理 `stepWeight` 的求解。这里仅仅需要继承 `RailwayGraphFactory` 即可，这样在 `MyRailwaySystem` 中只需要维护好这三个产品即可。
- 跨越上面两座大山后，问题基本就已经解决了，唯一仍需注意的是Dijkstra的写法，本人原始写法在更新dist[]时是无脑遍历所有结点，而拆点法会导致节点数最大会达到4000+，因而时间巨慢。实际上，每次遍历只需要遍历离源点st最近结点vertex的**可达集**即可大大降低时间复杂度。

类图如下：


Node
nodeId int
pathId int
Node()
Node(int, int)
getNodeId() int
getPathId() int
equals(Object) boolean
hashCode() int
toString() String
GraphFactory
reset() void
getShortestPath(Integer, Integer) Integer
«create»
MyPath
nodes ArrayList<Integer>
nodesSet HashSet<Integer>
MyPath(int...)
size() int
isValid() boolean
getNode(int) int
containsNode(int) boolean
getDistinctNodeCount() int
getNodesList() ArrayList<Integer>
getDistinctNodesSet() HashSet<Integer>
getUnpleasantValue(int) int
iterator() Iterator<Integer>
compareTo(Path) int
equals(Object) boolean
hashCode() int
MyPathContainer
id2Path HashMap<Integer, Path>
path2Id HashMap<Path, Integer>
pathIdCnt int
MyPathContainer()
size() int
containsPath(Path) boolean
containsPathId(int) boolean
getPathById(int) Path
getPathId(Path) int
addPath(Path) int
removePath(Path) int
removePathById(int) void
getDistinctNodeCount() int
RailwayGraphFactory
INFINITY int
stepWeight int
switchWeight int
adjList HashMap<Node, HashMap<Node, Integer>>
ansMap HashMap<Node, HashMap<Node, Integer>>
calcCnt int
RailwayGraphFactory(int, int)
getEdgeWeight(Node, Node, int) Integer
addEdge2adjList(Node, Node, int) void
delEdge4adjList(Node, Node) void
reset() void
getShortestPath(Integer, Integer) Integer
dijkstra(Node) void
MyGraph
connectedBlockCount int
nodes HashMap<Integer, Integer>
adjList HashMap<Integer, HashMap<Integer, Integer>>
shortestDist HashMap<Integer, HashMap<Integer, Integer>>
addNode(Integer) void
addEdge(Integer, Integer) void
addMyPath(MyPath) void
addMyPath2Product(MyPath, Integer) void
addPath(Path) int
delNode(Integer) void
delEdge(Integer, Integer) void
removeMyPath(MyPath) void
removeMyPath4Product(MyPath, Integer) void
removePath(Path) int
removePathById(int) void
initResult() void
getDistinctNodeCount() int
containsNode(int) boolean
containsEdge(int, int) boolean
isConnected(int, int) boolean
getShortestPathLength(int, int) int
updateDist2st(int) void
getConnectedBlockCount() int
calcConnectedBlockCount() void
UnpleasantGraphFactory
switchWeight int
nodeWeight HashMap<Integer, Integer>
UnpleasantGraphFactory()
getNodeWeight(Node) Integer
getEdgeWeight(Node, Node, int) Integer
«create»
«create»
«create»
MyRailwaySystem
stMap HashMap<Integer, Node>
edMap HashMap<Integer, Node>
transferMatrix RailwayGraphFactory
priceMatrix RailwayGraphFactory
unpleasantMatrix UnpleasantGraphFactory
MyRailwaySystem()
addEdge2adjList(Node, Node, int) void
addNode2Map(Integer, Integer) void
addMyPath2Product(MyPath, Integer) void
delEdge4adjList(Node, Node) void
delNode4Map(Integer, Integer) void
removeMyPath4Product(MyPath, Integer) void
initResult() void
getLeastTicketPrice(int, int) int
getLeastTransferCount(int, int) int

- RailwayGraphFactory

```
    public static final int INFINITY = 0x3fffffff;

    private final int stepWeight;   // 邻接边权
    private final int switchWeight; // 换乘边权

    // <stNode, edNode> -> edgeWeight
    private HashMap<Node, HashMap<Node, Integer>> adjList = new HashMap<>();
    // <fromNode, toNode> -> shortest distance
    private HashMap<Node, HashMap<Node, Integer>> ansMap = new HashMap<>();
```

- MyRailwaySystem

```
/*   Abstract Graph  */
// <NodeId, stNode>     : pathId = 0
private HashMap<Integer, Node> stMap = new HashMap<>(); // 抽象起点
// <NodeId, edNode>     : pathId = -1
private HashMap<Integer, Node> edMap = new HashMap<>(); // 抽象终点

/*   weight matrix  */
private RailwayGraphFactory transferMatrix;     // 最少换乘
private RailwayGraphFactory priceMatrix;        // 最低票价
private UnpleasantGraphFactory unpleasantMatrix;// 最小不满意度
```

5. bug分析及修复情况

- 第一次作业中的compareTo出现了减法溢出错误。原因是采用了一种比较“取巧”的做法 `return a-b;`，由于第一次作业比较掉以轻心，没有进行有效的测试，导致强测WA了快一半点。修复时只需改用 `if-else` 即可。
- 第二次作业在强测与互测中均未被发现bug。
- 第三次作业即在3.3陈述的CTLE错误，莫得办法，只能重构(捂脸)，但在重构的过程中也确实有所收获。

6. 心得体会

三周的JML之旅已然结束，这期间当然又踩了很多坑，也吃了不少经验教训。

首先，给我留下印象最深刻的还是那句话，**“重构一时爽，但也可能一次重构火葬场”**。不过虽然如此，在重构的过程中也有了很多新的感悟与体会，一个优美的架构好处不仅仅在于其可扩展性，由于模块与模块之间耦合度低，逻辑重合度低，其bug调试及定位难度也大幅降低，同时也能更好的借助JUnit进行覆盖性测试。

下面再来谈一谈对JML规格撰写的体会。

通过查阅JML手册以及参考已有规格，为一个方法写一份简单的规格描述难度并不大，在几次上机中也并没有太大的问题，而想要针对某个特定项目写一份完整、严谨的规格却是十分困难的。但 JML 的好处在于，测试人员能够从设计层面直接判断方法的正确性，而不用花费大量时间阅读代码。同时由于其语法的严谨性、规范性，给开发人员也带来极大地便利，不需要再去啃模棱两可的指导书中的需求，专注于代码的编写，正如指导书所言，**”只要你确保满足了JML的需求，那么你的程序一定是正确的“**。

此外，通过对 JML 规格的阅读与撰写，我对每个方法的 `Pre-Condition, Post-Condition, Side-Effect` 有了更加深刻的理解，这是一个方法，或是一段程序正确性的根本保证，能更好地帮助我们判断一个架构设计的正确性与合理性。

最后再谈一谈Openjml。本单元一开始对基于JML规格的形式化验证与自动化测试是充满期待的，但不得不说，JML的工具链使用体验极差，对语法的要求过于苛刻，导致写一份能够完美通过编译的JML规格的难度要远远大于完成程序本身，似乎有些本末倒置？但 JML 教给我们**规格设计的思想以及这种规范编程的思维方式**，对我们的程序员进化之路是大有裨益的。

国际惯例，祝OO课程越来越好！