[CS209A-23Fall] Assignment 1 (100 points)

DDL: 11月12日23:55分

宽限期: 48 H。 如果你错过了DDL，你可以争取在48小时内提交你的代码，即在11月14日23:55前提交，但是你只能得到你的代码60%的分数。11月14日23:55之后，不接受任何提交。

0.概述

该应用程序实现了一个基于内存的数据库引擎。该引擎分为数据增加、查询处理两个阶段，第一阶段向第二阶段的转移由一次 `query` 的函数调用触发，不可逆。数据增加阶段可插入若干数据，并返回成功插入的数据个数以及插入失败的记录信息。数据查询处理阶段，需要根据入参的条件约束，返回符合条件的数据。其中数据增加时所插入的数据可来源于文件读入并对读入的数据反序列化生成股票数据对象。

我们已经提供了框架代码，代码结构如下：

```
.
├── CS209A Assignment1.pdf: Assignment1需求文档
├── data: TestPass测试用到的数据，真实数据，由于版权原因不得外传
│   ├── TRD_Co.csv: 上市公司基本信息数据
│   ├── TRD_Co[DES][csv].txt: 上市公司基本信息数据各字段详细信息
│   ├── TRD_Dalyr.csv: 日交易明细数据
│   └── TRD_Dalyr[DES][csv].txt: 日交易明细数据各字段详细信息
├── src
│   ├── declaration
│   │   ├── CoreUtil.java: 包含了该项目所有需要实现的接口
│   │   ├── cmp
│   │   │   └── Compare.java: 枚举类，偏序数据类型的大小比较结果
│   │   ├── condition
│   │   │   ├── AndCondition.java: 与条件约束类，Condition的实现类
│   │   │   ├── AtomicConditionFactory.java: 接口，原子条件工厂，用于创建原子条件
约束类
│   │   │   ├── CodeCondition.java: 代码型字段(如股票代码、行业代码等)约束类，
Condition的实现类
│   │   │   ├── Condition.java: 接口，查询条件约束
│   │   │   ├── DateCondition.java: 日期条件约束类，Condition的实现类
│   │   │   ├── NotCondition.java: 非条件约束类，Condition的实现类
│   │   │   ├── OrCondition.java: 或条件约束类，Condition的实现类
│   │   │   ├── PatternCondition.java: 字段模式匹配条件约束类，Condition的实现类
│   │   │   └── ValueCondition.java: 数值条件约束类，Condition的实现类
│   │   ├── database
│   │   │   └── Database.java: 接口类，数据库引擎
│   │   ├── io
│   │   │   └── CsvReader.java: 接口类，文件读取器
│   │   ├── query
│   │   │   └── extension
│   │   │       ├── Ext.java: 接口类，查询处理的拓展支持
│   │   │       ├── NormalFormatOrder.java: 决定查询结果要输出的字段及顺序，Ext的
实现类
│   │   │       ├── Range.java: 查询范围，Ext的实现类
```

```
│   │   │       └── Sort.java: 查询排序规则，Ext的实现类
│   │   ├── stock
│   │   │   ├── Field.java: 枚举类，枚举了所有可能的股票数据字段
│   │   │   ├── Stock.java: 接口类，一条有效的股票数据
│   │   │   └── factory
│   │   │       ├── StockFactory.java: 接口类，股票数据工厂，从一条股票数据文本中
创建一个Stock对象
│   │   │       └── StockFactoryProvider.java: 接口类，股票工厂提供类，通过表头获
得一个股票工厂实例
│   │   └── streamsupport
│   │       └── StreamUtil.java: 接口类，
│   └── implementation
│       └── Main.java: 唯一一个需要你修改的文件，CoreUtil的实现类
├── static: 这个文件夹包含了OJ测试用例用到的测试数据，可配合LocalTest.java进行本地测
试
└── test
    └── TestPass.java: 演示完整的数据库引擎使用流程，可用于本地测试
    └── LocalTest.java: 包含OJ上大部分测试用例，可用于本地测试
```

其中CoreUtil.java中包含了该项目所有需要实现的接口:

- 文件读入：基于行的文本数据读入
- 对象反序列化：以 csv 格式将一行数据反序列化为股票数据对象
- 条件类型：生成、解析股票的查询条件，这是一种递归数据类型
- 流处理高级支持：实现关于流的合并处理功能
- 数据库查询引擎：根据查询条件和输出提示完成对股票数据的查询

而Main是CoreUtil的实现类，Main.java中已经完成了大部分的工作，你只需要补充完整Main.java中标注TODO部分的代码即可。或者你也可以不用我们提供的代码，只要正确实现了CoreUtil.java中定义的接口即可。

1、文件读入

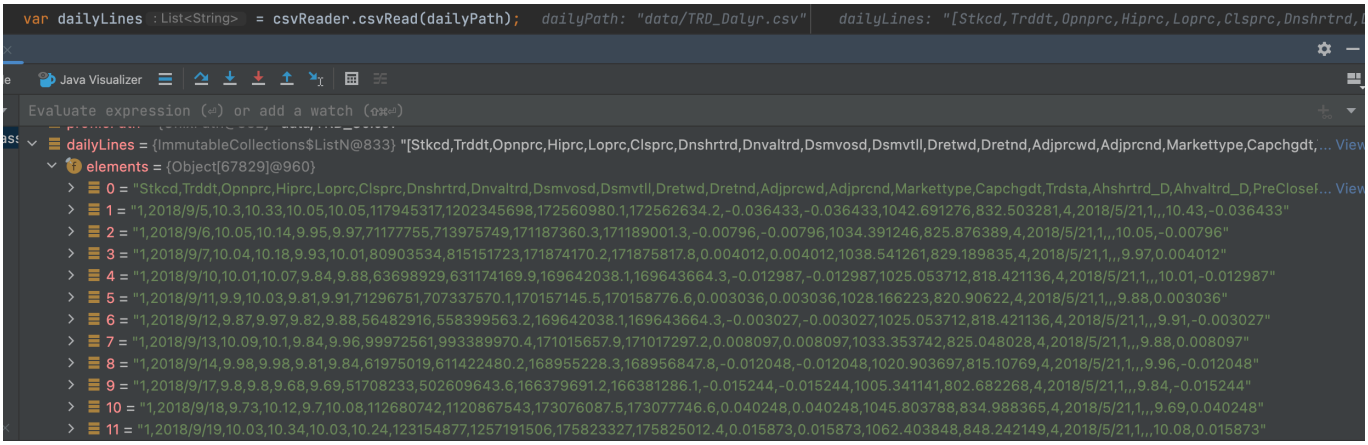
```java
    @Override
    public CsvReader getCsvReader() {
        // TODO
        throw new UnsupportedOperationException();
    }
```

你需要实现上述CoreUtil中的getCsvReader()方法，该方法需要返回一个CsvReader对象。

```java
public interface CsvReader {
    final class UnsupportEncodingException extends Exception {}
    List<String> csvRead(Path path) throws IOException,
UnsupportEncodingException;
}
```

如上所示，`CsvReader`是一个文本读取器接口，你需要实现`csvRead()`这个接口方法。可以观察到`CsvReader`只有一个抽象方法，可以通过Lambda表达式来创建该接口的对象。 `csvRead()`的入参为一个`Path`对象的csv文件路径，返回值为`List<String>`包含入参文件所有行字符串的List。 传入的csv文件路径有可能不存在，需要处理这种情况，并抛出合适的异常。csv文件有可能是UTF-8, 或带 BOM 头的以下编码：UTF-8, UTF-16, UTF-32, GB18030。你需要先识别该文件用何种方式编码，才能使用正确的编码读入该文件，否则将出现乱码。Hint:可以先读取文件开头的几个字节，根据上述几种编码头部字符的特点解析。`UnsupportEncodingException`用于表示不支持的编码处理情形，比如，如果传入的文件不是UTF-8, 或带 BOM 头的以下编码：UTF-8, UTF-16, UTF-32, GB18030文件，可以抛出此异常。由于常见的二进制文件种类繁多，该情形在测试中不会进行考察，是一个保留的异常类型。

调用示例及可能的output如下图所示：



2、对象反序列化

股票数据分为两大类，上市公司股票基本信息数据和日交易明细数据。 对股票对象进行反序列化的过程分为两步，一步处理表头，下一步处理实际该股票对象的持有的具体数据内容。

```
    @Override
    public StockFactoryProvider getAbstractStockFactory() {
        // TODO
        throw new UnsupportedOperationException();
    }
```

你需要实现上述`CoreUtil`中的`getAbstractStockFactory()`方法，该方法需要返回一个`StockFactoryProvider`对象。

```
public interface StockFactoryProvider {
    final class StockFactoryConstructorException extends Exception {
        public boolean containStockCode = true;
        public boolean singleLine = true;
        public boolean quotationMarks = true;
        public boolean containTradeDate = true;
    }
    StockFactory newStockFactory(String tableHeader) throws
StockFactoryConstructorException;
}
```

如上所示，StockFactoryProvider是一个产生股票工厂的接口，你需要实现newStockFactory()这个接口方法。 newStockFactory()方法的入参为一个String对象，传入的是csv文件的表头，返回值为StockFactory对象。你需要解析表头判断这是股票基本信息数据还是日交易明细数据。包含恰好一个 **Stkcd** 和 **Trddt** 的属性为日交易明细数据表；包含恰好一个 **Stkcd**, 不含 **Trddt** 的属性，则为股票基本信息数据；否则二者都不是。如果是股票基本信息数据，则返回产生股票类的工厂对象，否则返回产生日交易明细类的工厂对象。

```
public interface StockFactory {
    final class StockInstantiationException extends Exception {
        public List<Integer> unMatchColumnCount = Collections.emptyList();
        public boolean singleLine = true;
        public Set<Field> fieldParseFailed = Collections.emptySet();
        public boolean quotationMarks = true;

    }
    Stock newStock(String line) throws StockInstantiationException;
}
```

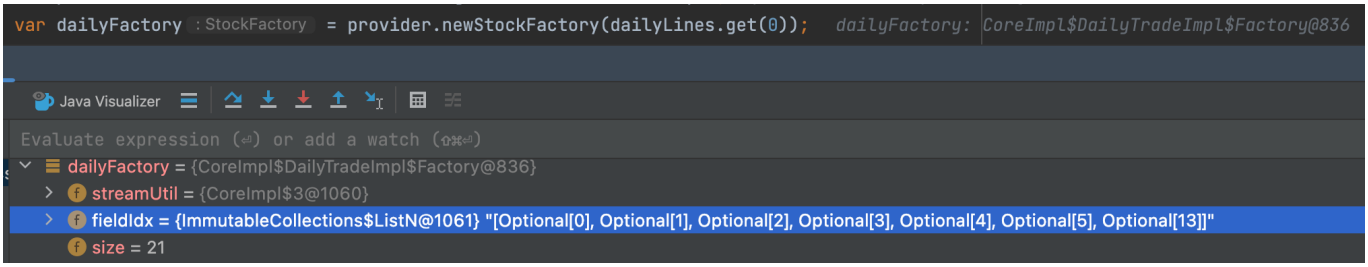
如上所示，StockFactory是一个生产股票的接口，你需要实现newStock()这个接口方法。 newStock()方法的入参为一个String对象，传入的是csv文件的一行数据，返回值为一个Stock对象。 通过观察可发现，股票类不包含数据表中的所有字段，只包含了部分字段，意味着生成股票对象时你需要分析数据与表头的对应关系，将所需要的数据值正确赋值给对应的字段。

Main.java文件中，我们给出了上市公司股票基本信息数据的股票类及其工厂类的实现，即StockImpl类及其StockImpl.Factory类，你可以参照这些代码实现日交易明细数据的股票类及其工厂类，对DailyTradeImpl类进行代码补充。

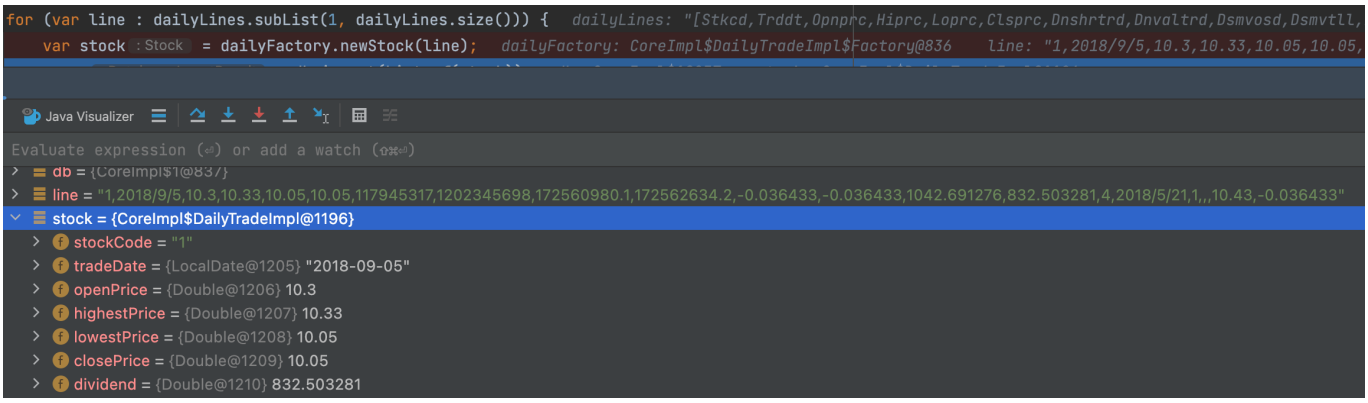
以日交易明细数据表的处理为例，当newStockFactory()方法传入的表头（第0行）为日交易明细数据表的表头时，应返回一个日交易明细类的工厂对象，该对象的fieldIdx解析出所需要的如下字段在表头中的index。

```
            public static final List<String> fieldNames = List.of(
                    "Stkcd",
                    "Trddt",
                    "Opnprc",
                    "Hiprc",
                    "Loprc",
                    "Clsprc",
                    "Adjprcnd"
            );
```

调用示例及可能的output如下图所示：



`newStock()`方法的入参传入的是csv文件的一行数据（第一行及后续行），返回值为一个`Stock`对象，将所需要的数据值正确赋值给Stock对象各个对应的字段。应检查主键相关的数据不允许为 `null`。调用示例及可能的output如下图所示：



3、条件类型

条件接口的实现类用于约束查询条件的类型，以便于在查询时进行类型检查。 条件类型总共有七种，分为复合条件类型和原子条件类型。 复合条件类型包括：

- 条件与类型 `AndCondition`
- 条件或类型 `OrCondition`
- 条件否类型 `NotCondition`

原子条件类型包括：

- 股票代码条件判断类 `CodeCondition`,
- 模式条件判断类 `PatternCondition`,
- 日期条件判断类 `DateCondition`,
- 值条件判断类 `ValueCondition`。

原子条件类型通过`AtomicConditionFactory` 构造得到。

```
    @Override
    public AtomicConditionFactory getAtomicConditionFactory(){}
```

`Main`需要实现上述`CoreUtil`中的`getAtomicConditionFactory()`方法，该方法需要返回一个原子条件工厂`AtomicConditionFactory`对象。

在`Main.java`文件中，如何new一个股票代码条件判断类 `CodeCondition`对象以及如何new一个值条件判断类对象`ValueCondition`的代码我们已给出，你需要参照已有代码完成模式条件判断类

PatternCondition对象和日期条件判断类 DateCondition对象的生成代码。其中LIST_DATE和TRADE_DATE属于可以应用DateCondition条件判断的字段，STOCK_CODE, STOCK_NAME, COMPANY_NAME, COMPANY_EN_NAME, INDUSTRY_CODE_A, INDUSTRY_NAME_C, PROVINCE, CITY, IPO_CURRENCY和MARKET_TYPE属于可以应用PatternCondition条件判断的字段。

4、流处理高级支持

流处理高级支持包括四个特殊函数。

- zip() 函数用于将两个不同的 stream 按照指定的方式合并.
- flatZip() 用于增强地合并两个不同的 stream, 完成映射后并展开扁平化 stream. （zip()与flatZip()的关系类似于 map() 和 flatMap() 之间的关系，后面有例子介绍）
- onlyOne() 用于断言该 stream 只有唯一一个元素并返回它。
- split() 用于分叉一个 stream, 取其中部分元素产生一个新的 stream, 其余剩余元素又产生另一新的 stream, 因此 split 可以完成对 stream 的分叉。

```
    @Override
    StreamUtil getStreamUtil(){}
```

Main需要实现上述CoreUtil中的getStreamUtil()方法，该方法需要返回一个StreamUtil对象。 在Main.java文件中，我们已经实现了StreamUtil接口中的onlyOne()方法和split()方法，你需要实现zip()和flatZip()方法。

zip() 函数

```
    /**
     * 流合并组合
     * @param stream1 待合并流 1
     * @param stream2 待合并流 2
     * @param map 合并组合函数
     * @return 合并组合后的流
     * @param <R> 合并组合后的流元素类型
     * @param <T> 待合并流 1 元素类型
     * @param <U> 待合并流 2 元素类型
     */
    <R, T, U> Stream<R> zip(Stream<T> stream1, Stream<U> stream2,
BiFunction<? super T, ? super U, ? extends R> map);
```

zip()函数简单调用示例如下：

```
    var sUtil = core.getStreamUtil();
    Stream<Integer> leftList = Stream.of(1,2,3,4,5);
    Stream<String> rightList = Stream.of("B","A","C","D","E");
    var zipStream = sUtil.zip(leftList, rightList, (l, r)-
```

```
>r+l.toString());
    zipStream.forEach((s)->System.out.print(s+" "));
```

输出如下：

```
B1 A2 C3 D4 E5
```

`flatzip()` 函数

```
    /**
     * 流合并组合扁平化
     * @param stream1 待合并流 1
     * @param stream2 待合并流 2
     * @param flatMap 合并组合扁平化函数
     * @return 合并组合后的流
     * @param <R> 合并组合后的流元素类型
     * @param <T> 待合并流 1 元素类型
     * @param <U> 待合并流 2 元素类型
     */
    <R, T, U> Stream<R> flatZip(Stream<T> stream1, Stream<U> stream2,
BiFunction<? super T, ? super U, Stream<? extends R>> flatMap);
```

`flatzip()`函数简单调用示例如下：

```
    var sUtil = core.getStreamUtil();
    Stream<Integer> leftList = Stream.of(1,2,3,4,5);
    Stream<String> rightList = Stream.of("B","A","C","D","E");
    var flatZipStream = sUtil.flatZip(leftList, rightList, (l, r)-> {
        List<String> segment = new ArrayList<>();
        for (int k = 0; k < l; ++k) {
            segment.add(r + k);
        }
        return segment.stream();
    }
    );
    flatZipStream.forEach((s)->System.out.print(s+" "));
```

输出如下：

```
B0 A0 A1 C0 C1 C2 D0 D1 D2 D3 E0 E1 E2 E3 E4
```

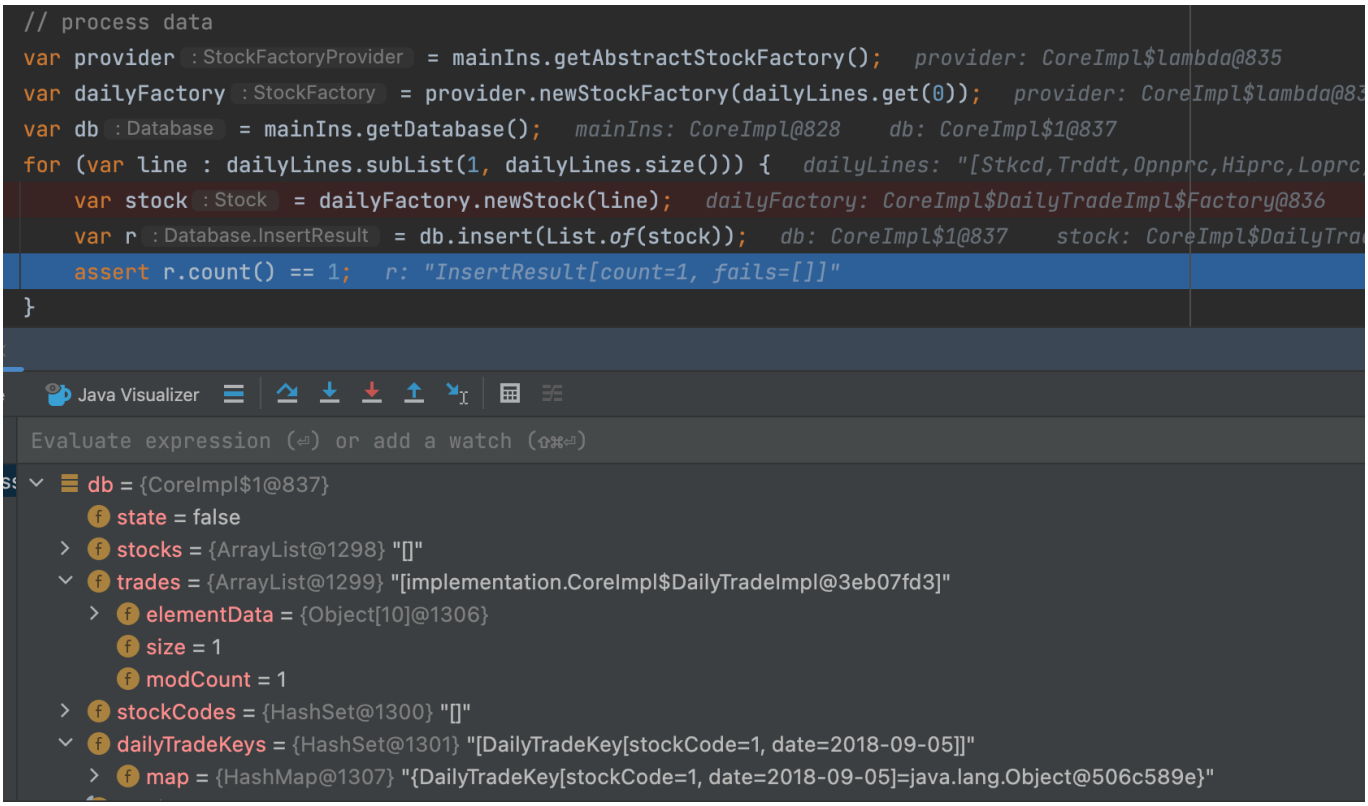
5、数据库引擎

该引擎分为数据增加、查询处理两个阶段，第一阶段向第二阶段的转移由一次 `query` 的函数调用触发，不可逆。 我们已提供数据库引擎相关接口的所有实现代码，不需要你实现。

数据增加

数据增加阶段插入数据的过程需要检查数据的合法性，其中股票数据合法性的检查应在反序列化阶段完成，此阶段只检查不同的股票数据在主键上是否出现重复，对于公司的信息数据，键值为**Stkcd**. 而对于日交易数据， 键值为**Stkcd**和**Tradt**。

如下是插入一条日交易明细股票对象后数据库的状态：



查询处理

查询处理的过程会分为两步，第一步进行查询合法性的检查，第二步进行查询操作。

合法查询的检查保证如下：

- 查询、格式化及排序所涉及到的字段不应该同时具有两个表格的独特字段，这会产生表格查询歧义。
- 在两个表格查询操作都可行的情形下，公司信息表格具有更高的优先级被选择。
- 格式化操作指示对象至多只有一个，如果没有，则查询会直接得到 `Stock` 对象。
- 区间过滤 (`Range`) 对象至多只有一个，不必实现对多个区间的合并。
- 排序指示符 (`Sort`) 可以有多个，从前到后表示它们的优先级。
- 条件判断时，如果一个股票数据对象对应字段缺失，则得到 `false`.
- 区间过滤后进行格式化（不直接输出股票对象）时，格式化涉及的对象字段缺失，则该对象被忽略。

如果需要查询股票代码为"49"，且`HIGH_PRICE`小于25.0的数据，且数据的显示格式为依次显示每条记录的`STOCK_CODE`，`HIGH_PRICE`，`LOW_PRICE`，记录按照`HIGH_PRICE`升序排列，可用如下代码进行查询：

```
    var code49 = new AndCondition(List.of(new
CodeCondition(Field.STOCK_CODE, "49"),new ValueCondition(Field.HIGH_PRICE,
```

```
25.0, Compare.Lt)));
    var showTag1 = new NormalFormatOrder(List.of(Field.STOCK_CODE,
Field.HIGH_PRICE, Field.LOW_PRICE));
    var sort1 = new Sort(Field.HIGH_PRICE, true);
    var res = db.query(code49, showTag1, sort1);
    var r1 = res.get();
    System.out.println(r1);
```

输出如下：

```
[[49, 22.00, 20.00], [49, 22.80, 21.91], [49, 23.10, 21.90], [49, 23.38,
22.61], [49, 23.65, 22.74], [49, 23.80, 22.62], [49, 23.83, 23.43], [49,
24.20, 22.10], [49, 24.25, 23.63], [49, 24.35, 23.62], [49, 24.42, 24.12],
[49, 24.65, 23.30], [49, 24.67, 23.48], [49, 24.69, 23.91], [49, 24.70,
22.81], [49, 24.77, 24.40], [49, 24.79, 24.36], [49, 24.80, 24.05], [49,
24.81, 24.49], [49, 24.83, 23.96], [49, 24.85, 23.52], [49, 24.86, 24.40],
[49, 24.87, 24.15], [49, 24.88, 24.42], [49, 24.90, 24.43], [49, 24.96,
24.20]]
```

6、代码规范

异常安全

本次 assignment 代码大范围使用 checked exception, 请依据函数签名上的异常抛出列表及相应描述正确实现异常安全的代码。

unchecked exception, 即继承自 `RuntimeException`, 只会用于逻辑错误和断言失败的代码场景，不会在正常的业务逻辑中发生。

特别地，以下列出几种不需要处理的情形，即你可以主动断言这些情形不会发生。

- 如**空值安全**所言，接口函数所描述的模块边界也不会传入 `null` 值。
- `Condition` 类型尽管是递归的，对其中的递归数据使用可以基于更严格的 `UnmodifiedList` 假设，（尽管它可能事实上不是）不会影响到处理问题的正确性；这意味着不需要担心该结构会构成一个无穷循环的环路。
- 你得到的所有关于原子条件的 `Condition` 都由你定义的 `AtomicConditionFactory` 产生，关于对它在查询过程中的条件判断的安全检查可以基于此假设进行放松。
- 数据库中插入的数据类型 `Stock` 一定来自 `StockFactory.newStock` 产生的实例对象，可以大胆假设它的实际类型必然属于你所认识到的几种股票类型之一。

空值安全

- 接口定义的函数不允许传出 `null` 值。
- 任何用于模块边界、接口函数边界的对象的内部数据也不应该有字段为 `null`, 这个要求显然是递归的。

更多内容请参照 **assignment1 代码中的注释**。

7、Q&A文档

23Fall Assignment1 Q&A

点击上述链接可访问Assignment1 Q&A文档，该文档将不定期更新。

8、作业评估

我们在OJ(https://oj.cse.sustech.edu.cn) 上部署了自动评测样例去测试你代码的正确性。请提交 `Main.java` 到 OJ。在你提交代码到OJ之前，你可以使用我们提供的测试用例和数据先在本地测试通过。