Ikko-Database

https://github.com/nikifaets/Ikko-Database

Николай Пашов

26 април 2020 г.

Съдържание

1 Въведение

1.1 Описание на задачата

Настоящият проект представлява имплементация на база данни, описана по-подробно на следния линк. Базата от данни е изградена от таблици. Поставената задача изисква таблиците да поддържат три типа записи - цяло число (*Int*), дробно число (*Double*), и низов масив (*String*), както и имплементацията на потребителски интерфейс, позволяващ базови операции върху таблиците - четене, променяне, запазване и др. (описани са подробно на горепосочения линк).

1.2 Акценти

1.2.1 Дефиниция на типовете данни

За да се работи удобно и структурирано със заданените типове данни, тяхното представяне трябва да бъде стриктно дфенирано. Това улеснява имплементацията на методи за боравене с тези данни, както и валидирането им. Правилата за записване на трите типа записи в таблиците са следните:

- ***Int*** - съдържа само символи в интервала 0-9. Пример: 123
- ***Double*** - съдържа символи в интервала 0-9 и един символ за дробна запетая - ".". Пример: 123.123
- ***String*** - може да съдържа всякакви символи, обградени в кавички. Кавички и наклонена черта се представят с наклонена черта преди тях. Пример:

```
"This\\That is a \"string\""
```

Ще бъде прочетено от програмата като низ от символи със стойност:

```
This\That is a "string".
```

1.2.2 Архитектура

Тъй като целта на проекта е да позволява разнообразни операции върху таблици с данни, създаването на подходящи модули, които да съдържат данните по време на изпълнение на програмата, е от ключова важност. Ако записите от таблиците се конвертират в удобни за работа програмни единици, то операциите за манипулация на информация стават тривиални. По тази причина, на модулите за зареждане на таблица по време на изпълнение е поставен най-голям акцент.

2 Реализация

2.1 Планиране

2.1.1 Файлов формат

Файл, съдържащ таблица се състои от следните задължителни компоненти в показания ред:

- Ред с имената на колоните;
- Ред, индициращ типа на всяка колона. Типовете *{Int, Double, String}* се обозначават съответно с цифрите {0, 1, 2};
- Редове със записи. Поле с празна стойност се бележи със символния низ *"Null"*.

Примерна таблица може да се види на Таб. 1

NameInt	NameDouble
0	1
123	Null
455	34.14

Таблица 1: Примерна таблица, съдържаща записи от тип *Int* и *Double*, както и един празен запис.

2.1.2 Главни модули

Поради причините, описани в Секция 1.2.2, акцентът на реализацията до този момент бе създаване на надеждни и скалируеми модули за зареждане на данни от таблица по време на изпълнение. С цел ускорение имплементацията на функционалността за зареждани на данни, поддържаните типове записи за момента са *Int* и *Double*. Имплементирани са следните ключови класове:

- ***Record*** - съдържа базова информация за запис от определен тип. Наследява се от ***RecordInt***, ***RecordDouble***, ***RecordString*** и ***RecordInvalid***
- ***Row*** - съдържа контейнер с обекти от тип ***Record***. Предоставя функционалност за конвертиране в тип *String*, за да се запише във файл, добавяне на колона, както и други методи за манипулация на данни.
- ***Table*** - съдържа контейнер с обекти от тип ***Row***, както и базова информация, касаеща таблица - имена и типове на колони, име

на таблица и др. Предоставя функционалност за четене и записване на таблица, манипулация на записите й, както и съответните валидации

- ***Parser*** - предоставя интерфейс, чрез който обекти от клас ***Table*** четат таблица от файл.
- ***Message*** - позволява изписването на съобщения в конзолния екран. Създаден е с цел разделяне на бизнес и вход/изход логиката.

2.2 Имплементация

2.2.1 Прочитане и съхраняване на таблица

Във Фрагмент 1 е изобразен методът за четене на таблица от файл. Проследявайки този метод, може да се покаже целият процес за конвертиране на таблицата в програмни единици, удобни за манипулация. Следният анализ е с цел демонстрация на тези програмни единици.

```
1  void Table::read_table(std::string filename){
2
3      std::ifstream table;
4      table.open(filename);
5
6      if(!table.is_open()){
7
8          Message::FileNotFound(filename);
9          return;
10     }
11
12     std::string line;
13
14     //read column names - first line
15     std::getline(table, line);
16     col_names = Parser::parse_line_str(line);
17
```

```
18      //read type data - second line
19      std::getline(table, line);
20      row_types = Parser::parse_type_data(line);
21
22      if(!row_types.size()){
23
24          Message::CorruptedTypeInformation(filename);
25          return;
26      }
27
28      if(row_types.size() != col_names.size()){
29
30          Message::CorruptedTypeInformation(filename);
31          return;
32      }
33
34      //read rows
35      while(std::getline(table, line)){
36
37          Row row(Parser::parse_line(line, row_types));
38
39          if(!validate_row(row)){
40
41              Message::CorruptedRow(rows.size());
42              rows.clear();
43              return;
44          }
45
46          rows.push_back(row);
47      }
48
49      table.close();
50  }
```

Фрагмент 1: Метод на клас ***Table*** за прочитане на таблица от файл.

Методът започва с опит за четене на файл и валидиране дали той е прочетен успешно. Следва прочитане на реда, съдържащ имената на

колоните в съответния файл. За тази цел се използва статичният метод *parse_ line_ str* на клас ***Parser***. Той конвентира ред от таблицата в обект от тип *std::vector<std::string>*, съдържащ записа от всяка една колона като тип *String*. Имплементацията му се вижда във Фрагмент 2. В него се използват други помощни методи на клас ***Parser***.

```
1  std::vector<std::string> Parser::parse_line_str(std::string line){
2
3      std::vector<std::string> line_recs;
4
5      for(int i=0; i<line.size(); i++){
6
7          if(is_separator(line[i])) continue;
8
9
10         std::string val = read_until_whitespace(line.substr(i));
11         i += val.size();
12
13         line_recs.push_back(val);
14
15     }
16
17     return line_recs;
18 }
```

Фрагмент 2: Метод на клас ***Parser*** за конвертиране на ред от таблица във вектор с елементи от тип *String*.

Следва прочитането на реда, съдържащ типовете в таблицата (ред 18-32). Методът *parse_ type_ data* работи по подобен начин като *parse_ line_ str*, само че конвертира реда не във вектор от тип *String*, ами от тип *Type*, който е *enum*, дефиниран в клас ***Record***. Той съдържа типовете, описани в 2.1.1.

Веднага след това се извършва валидация. Методът *parse_ type_ data* ще върне празен вектор ако е имало грешка във валидацията на типовете, която се извършва в метода. Тази валидация е наложителна, тъй

като има ясно ограничение за типа на записите в този ред от таблицата, както и стойностите, които те могат да заемат.

Втората валидация е дали броят на типовете в таблицата съответства на броя имена на колони. Ако не съответстват, значи има грешка във файла и той не може да бъде обработен.

Следва същинското прочитане на записите. То се осъществява като всеки следващ ред от таблицата се конвертира в тип ***Row***, използвайки метода *parse_line* на класа ***Parser***. Методът е изобразен във Фрагмент 3.

```
1  std::vector<Record*> Parser::parse_line(std::string line, std::
       vector<Type> types){
2
3      std::vector<Record*> recs;
4
5      std::vector<std::string> line_str = parse_line_str(line);
6
7      for(int i=0; i<line_str.size(); i++){
8
9          Record* rec;
10
11         if(!line_str[i].compare(NULL_REC)){
12
13             rec = Caster::type_to_rec(types[i]);
14             rec->set_empty(true);
15
16         }
17
18         else{
19
20             rec = Caster::string_to_rec(line_str[i]);
21
22         }
23
24         if(rec->get_type() == Invalid){
25
```

```
26              Message::InvalidRecord(i);
27              return std::vector<Record*>();
28
29          }
30
31          recs.push_back(rec);
32      }
33
34      return recs;
35  }
```

Фрагмент 3: Метод на клас ***Parser*** за конвертиране на ред от таблица във вектор с тип ***Record****.

Първото, което се забелязва, е че той използва метода *parse_line_str*, за да конвертира реда във вектор от тип *String*. След това всеки елемент от този вектор се конвертира в обект от клас ***Record***.

Първата валидация, която се извършва е на ред 11. Тя проверява дали записът е празен (със стойност *Null*). В този случай, връща обект като запис от типа, който съответства на типа, който беше прочетен преди това от *parse_type_data*. Използва се клас ***Caster*** за създаване на клас, наследяващ ***Record*** (вж. Секция 2.1.2). Този клас е имплементиран за подобни преобразувания

Ако записът не е празен, то тогава той се конвертира от тип *String* в обект на някой от класовете, наследяващи ***Record***, описани в 2.1.2. Това се прави също от клас ***Caster***.

Методът *string_to_rec* връща обект на клас ***RecordInvalid*** ако записът е невалиден, тоест не съответства на нито една от дефинициите от Секция 1.2.1.

След като редът от таблицата е конвертиран в обект на клас ***Row***, методът *read_table* прави валидация на целия ред. Това включва проверки като дали броят на записи в реда съответства на броя типове, описани в таблицата преди това; дали редът е празен и др. Ако валидацията е

успешна, редът се добавя към вектора, който съхранява всички редове в таблицата.

2.2.2 Съхраняване и манипулация на записи

Съхранявани по този начин, записите от таблиците са удобни за манипулация. Разбиването на данните на по-малки структури като ***Row*** и ***Record*** позволява гъвкавост при работата с данни. Например, добавяне на нови записи може да стане лесно, използвайки класа ***Row***. Такива методи са изобразени във Фрагмент 4.

```
void Row::add_empty_record(Type type){

    Record* new_rec = Caster::type_to_rec(type);
    new_rec->set_empty(true);
    records.push_back(new_rec);
}

void Row::add_record(Record* rec){

    records.push_back(rec);
}

void Row::add_record(std::string val){

    Record* new_rec = Caster::string_to_rec(val);
    records.push_back(new_rec);
}
```

Фрагмент 4: Методи на клас ***Row*** за добавяне на нов запис.

2.2.3 Запазване на таблица

Конвертирането от таблица, представена като обект на клас ***Table*** във файл с формат, описан в Секция 2.1.1 също става лесно. Може да се види във Фрагмент 5.

Забележка: Не се извършва валидация дали файлът не е съществуващ, тъй като все още не е имплементирана работа с бази от данни, съдържащи множество таблици.

```
1
2  void Table::save_table(std::string filename){
3
4      std::ofstream table(filename);
5
6      //write names
7      std::string names = col_names_to_str();
8      table << names << std::endl;
9
10     //write types
11     std::string types = types_to_str();
12     table << types << std::endl;
13
14     //write rows
15     for(int i=0; i<rows.size(); i++){
16
17         std::string row_string = rows[i].to_string();
18         table << row_string;
19
20         if(i < rows.size()-1){
21
22             table << std::endl;
23         }
24     }
25 }
```

Фрагмент 5: Метод на клас ***Table*** за запазване на таблица във файл.

Всеки компонент се конвертира в тип *String*, след което се записва във файла. Методите за конвертиране поддържат всички възможни типове записи, както и празни такива.

2.3 Съобщения за грешки

В тази секция са показани две възможни съобщения за грешки при неуспешно валидиране на таблица при прочитане.

2.3.1 Невалиден запис

С демонстративна цел е създадена невалидна таблица (Таблица 2).

NameInt	NameDouble
0	1
123	321
455	34.14.57

Таблица 2: Примерна таблица, съдържаща записи от тип *Int* и *Double*, с един неправилен запис.

При опит за прочитане, функцията за четене на таблица *read_table* приключва и в конзолата се изписва следното съобщение:

```
$ Invalid record at column 1
$ Corrupted row with index 1
```

2.3.2 Неправилен брой колони

Друг пример за невалидна таблица е случай с невалиден брой колони. Такава е Таблица 3.

При опит за прочитане, функцията за четене на таблица *read_table* приключва и в конзолата се изписва следното съобщение:

NameInt	NameDouble	n/a
0	1	n/a
123	12.1	n/a
455	34.14	12

Таблица 3: Примерна таблица, съдържаща записи от тип *Int* и *Double* с неправилен брой колони в един от редовете.

```
$ Row has 3 columns, but expected are 2
$ Corrupted row with index 1
```

3 Заключение

До настоящия момент, за реализацията на проекта са имплементираните базовите функционалности, които позволяват прочитане и запазване на таблица. Информацията от таблицата се съхранява в програмни единици, които позволяват много лесно функционалността на проекта да бъде разширена с повече модули за манипулация на данните, като тези описани в заданието.

4 Бъдещо развитие

Бъдещото развитие на проекта включва:

- Добавяне на функционалност за манипулация на данни в таблица
- Добавяне на възможността за работа с база от данни, съхраняваща множество таблици.

- Потенциално пренаписване на базовите класове, деривати на ***Record***. Генерализацията на записите към момента не е достатъчно добра и изисква помощни класове като ***Caster***, за да се справя с факта, че различните записи представляват различни типове. По същата причина е затруднен и полиморфизмът между тези класове.
- Методите на класовете, които съществуват само в композиции с други класове, потенциално могат да станат *private*, а класовете им да станат приятелски с тези, с които са в композиция.

5 Използвани технологии

Целият проект е реализиран на езика C++. Използваните библиотеки са:

- iostream
- vector
- string
- fstream
- algorithm
- unordered_set

Проектът е тестван на Arch Linux 5.5.10. Компилиран е с g++ за C++11.

6 Литература

Много справки, свързани с реализацията на проекта и документацията му са правени в:

- `https://stackoverflow.com/`
- `http://www.cplusplus.com/doc/`
- `https://www.overleaf.com/learn`
- `https://tex.stackexchange.com/`