JSON ПАРСЕР

от Апостол Янков

1. Въведение

JSON (JavaScript Object Notation) е свободно-достъпен стандартен файлов формат за съхранение и обмен на данни, който е удобен за четене и работа, поради което е и добра алтернатива на XML формата. JSON файловете използват разширението “.json”, а тяхната употреба влиза в много съвременни езици за програмиране. Най-често JSON файловете се използват за комуникация между клиент и сървър чрез заявки.

JSON данните се записват като двойки ключ/стойност, като те са разделени, чрез двуеточие “:“ (ключът е отляво, стойността - отдясно), а стойностите могат да бъдат следните типове:

- число – short, int, long, double, etc.
- низ – поредица от Unicode символи (string)
- булева стойност – вярно/невярно
- масив – списък от стойности
- обект – колекция от двойки ключ/стойност

При работа с JSON файлове се срещат термините сериализация и десериализация. Сериализацията представлява превръщане на обект (колекция от двойки ключ/стойност) към низ (string), и се използва, когато искаме да подадем данни към някой JSON файл за дадена операция. Десериализацията е обратният процес на превръщане на текст в обект – използва се, когато искаме да прочетем даден JSON файл и да запазим данните от него в наша структура в кода.

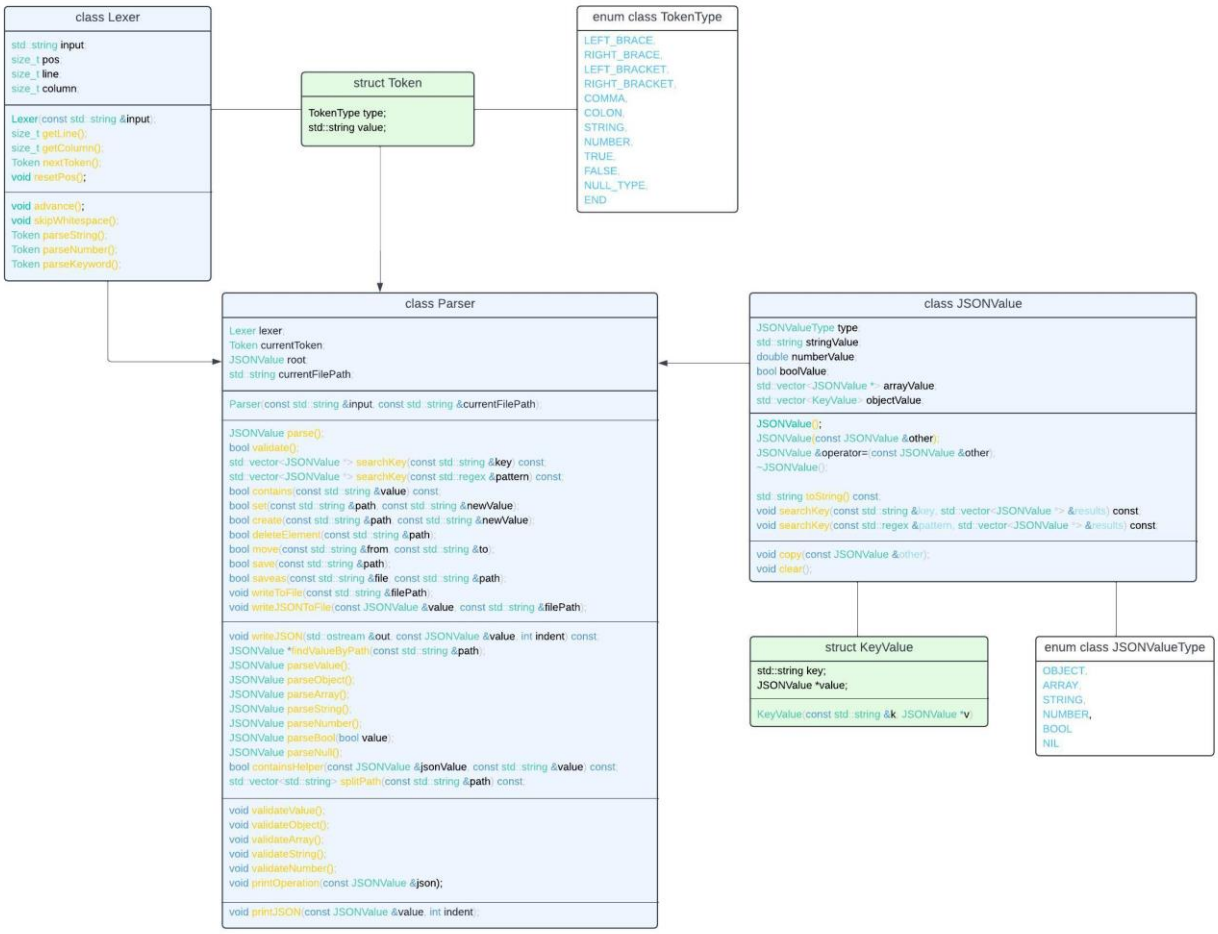
2. Преглед на предметната дейност

Този проект реализира JSON парсер на C++ без използване на външни библиотеки. Парсерът може да валидира, манипулира и изпълнява различни операции върху JSON данни. Ключовите компоненти на проекта са класовете Lexer, Token, Parser, JSONValue и Engine.

Главният проблем, който трябва да решим, е как да прочитаме и обработваме JSON файловете, така че да можем да ги променяме като обект в кода, след което да ги презапишем отново във файл. Трябва да можем рекурсивно да итерираме нивата на JSON файловете, за да четем, записваме, променяме и трием данни.

Част от този проблем е откриването на подходящ начин за пазене на тези данни в обект. Тоест какви структури от данни да използваме, че да го направим възможно. В този проект това се решава чрез основен клас JSONValue, който може да бъде всеки един тип, който може да бъде срещнат в един JSON файл, представен чрез отделно поле за всеки тип. Така всеки обект JSONValue има попълнено съответно едно от тези полета, спрямо това какъв е типът му.

Компоненти



Диаграма 1.

1) Lexer

Класът Lexer е отговорен за лексикалния анализ на JSON входа. Той токенизира входния низ в смислени токени, които парсерът Parser може да обработи.

Публични методи:

- Lexer(const std::string &input): Конструира обект Lexer с дадения вход.
- size_t getLine(): Връща текущия номер на реда.
- size_t getColumn(): Връща текущия номер на колоната.
- Token nextToken(): Взема следващия токен от входа.
- void resetPos(): Нулира позицията до началото на входа.

Частни методи:

- void advance(): Премества текущата позиция с един символ напред.
- void skipWhitespace(): Пропуска празни символи във входа.
- Token parseString(): Парсира низов токен.
- Token parseNumber(): Парсира числов токен.
- Token parseKeyword(): Парсира ключова дума (true, false, null).

2) Token

Класът Token представлява токен в JSON с тип и стойност.

Енумерация TokenType:

- LEFT_BRACE, RIGHT_BRACE, LEFT_BRACKET, RIGHT_BRACKET, COMMA, COLON, STRING, NUMBER, TRUE, FALSE, NULL_TYPE, END

Структура Token:

- TokenType type: Тип на токена.
- std::string value: Стойност на токена, ако е приложимо.

3) JSONValue

Класът JSONValue представлява стойност в JSON, която може да бъде обект, масив, низ, число, булева стойност или null.

Всеки обект от тип JSONValue съдържа член type от тип JSONValueType, който определя какъв тип стойност съхранява обектът в дадения момент. Енумерацията

JSONValueType включва всички възможни типове JSON стойности: OBJECT, ARRAY, STRING, NUMBER, BOOL, NIL. В зависимост от типа на стойността (определено от type), методът toString избира правилния член (stringValue, numberValue, boolValue и т.н.) и го конвертира в съответния JSON формат. Този подход предоставя ясно разграничение между различните типове стойности в JSON, като всеки тип има собствен член променлива. Това улеснява както четенето, така и разбирането на кода, тъй като типовете стойности и съответните им представяния са ясно дефинирани.

Публични методи:

- JSONValue(): Конструктор по подразбиране.
- JSONValue(const JSONValue &other): Копиращ конструктор.
- JSONValue &operator=(const JSONValue &other): Копиращ оператор за присвояване.
- ~JSONValue(): Деструктор.
- std::string toString() const: Преобразува JSON стойността в низово представяне.
- void searchKey(const std::string &key, std::vector<JSONValue *> &results) const: Търси ключ в JSON стойността и събира всички съответстващи стойности.
- void searchKey(const std::regex &pattern, std::vector<JSONValue *> &results) const: Търси ключове, съответстващи на regex шаблон в JSON стойността и събира всички съответстващи стойности.

4) Parser

Класът Parser е отговорен за парсиране, манипулиране и валидиране на JSON данни.

Публични методи:

- Parser(const std::string &input): Конструктор с даден входен низ.
- JSONValue parse(): Парсира входния JSON и връща кореновата JSON стойност.
- bool validate(): Валидира JSON структурата.
- std::vector<JSONValue *> searchKey(const std::string &key) const: Търси ключ в JSON структурата и връща вектор от съответстващи JSON стойности.
- std::vector<JSONValue *> searchKey(const std::regex &pattern) const: Търси ключове, съответстващи на regex шаблон, и връща вектор от съответстващи JSON стойности.
- bool contains(const std::string &value) const: Проверява дали стойност се съдържа в JSON структурата.
- bool set(const std::string &path, const std::string &newValue): Задава нова стойност на даден път в JSON структурата.
- bool create(const std::string &path, const std::string &newValue): Създава нов елемент на даден път в JSON структурата.

- bool deleteElement(const std::string &path): Изтрива елемент на даден път в JSON структурата.
- bool move(const std::string &from, const std::string &to): Премества елементи от един път на друг в JSON структурата.
- bool save(const std::string &path): Запазва JSON структурата във файл.
- bool saveas(const std::string &file, const std::string &path): Запазва JSON структурата в указан файл.
- void writeToFile(const std::string &filePath): Записва JSON структурата във файл.
- void writeJSONToFile(const JSONValue &value, const std::string &filePath): Записва дадена JSON стойност във файл.

Частни методи:

- void writeJSON(std::ostream &out, const JSONValue &value, int indent) const: Записва JSON стойност в изходен поток с отстъпи.
- JSONValue *findValueByPath(const std::string &path): Намира JSON стойност по даден път.

Методи за десериализация. Основният метод е parseValue(), който спрямо типа на текущия токен извиква някой от другите методи за десериализация на съответния тип данни.

- JSONValue parseValue(): Парсира JSON стойност.
- JSONValue parseObject(): Парсира JSON обект.
- JSONValue parseArray(): Парсира JSON масив.
- JSONValue parseString(): Парсира JSON низ.
- JSONValue parseNumber(): Парсира JSON число.
- JSONValue parseBool(bool value): Парсира JSON булева стойност.
- JSONValue parseNull(): Парсира JSON null стойност.

- bool containsHelper(const JSONValue &jsonValue, const std::string &value) const: Помощна функция за проверка дали стойност се съдържа в JSON стойност.
- std::vector<std::string> splitPath(const std::string &path) const: Разделя низов път на отделни ключове.

Методи за валидация на JSON файла. Главният метод е validateValue(), който спрямо типа на текущия токен извиква някой от другите методи за валидация на съответния тип данни.

- void validateValue(): Валидира JSON стойност.

- void validateObject(): Валидира JSON обект.
- void validateArray(): Валидира JSON масив.
- void validateString(): Валидира JSON низ.
- void validateNumber(): Валидира JSON число.
- void printOperation(const JSONValue &json): Отпечатва JSON стойност.

5) Engine

Класът Engine е отговорен за обработка на входа от потребителя и изпълнение на команди за манипулиране на JSON данни с помощта на парсера.

Публични методи:

- Engine(): Конструктор на Engine.
- void prompt(): Подканва потребителя за команди и ги изпълнява. Ако няма зареден файл, първо подканва потребителя да въведе пътя до JSON файла, с който ще работи.

Частни методи:

- void executeCommand(const std::string &command): Изпълнява дадена команда.
- void openFile(const std::string &filePath): Отваря указания файл и зарежда съдържанието му в парсера. Ако файлът не съществува, създава нов файл с празно съдържание.

Основни части от кода

Класът JSONValue и структурата KeyValue – един обект представлява вектор от двойки ключ/стойност (KeyValue). Масивът е представен като вектор от JSONValue указатели.

```
struct KeyValue
{
    std::string key;
    JSONValue *value;
    KeyValue(const std::string &k, JSONValue *v) : key(k), value(v) {}
};
class JSONValue
{
public:
    JSONValueType type;
    std::string stringValue;
    double numberValue;
    bool boolValue;
    std::vector<JSONValue *> arrayValue;
    std::vector<KeyValue> objectValue;
```

Извадка 1.

Класът Parser съдържа основната логика за сериализация/десериализация, както и за командите необходими за удовлетворяването на условието на проекта.

writeJSON – попълва даден поток с данните в подходящ формат от JSONValue обект

```
void Parser::writeJSON(std::ostream &out, const JSONValue &value, int indent = 0)
const
{
    std::string indentStr(indent, ' ');
    switch (value.type)
    {
    case JSONValueType::OBJECT:
        out << "{\n";
        for (size_t i = 0; i < value.objectValue.size(); ++i)
        {
            out << indentStr << "  \"" << value.objectValue[i].key << "\": ";
            writeJSON(out, *value.objectValue[i].value, indent + 2);
            if (i < value.objectValue.size() - 1)
                out << ",";
            out << "\n";
        }
        out << indentStr << "}";
        break;
    case JSONValueType::ARRAY:
        out << "[\n";
        for (size_t i = 0; i < value.arrayValue.size(); ++i)
        {
            out << indentStr << "  ";
            writeJSON(out, *value.arrayValue[i], indent + 2);
            if (i < value.arrayValue.size() - 1)
                out << ",";
            out << "\n";
        }
        out << indentStr << "]";
        break;
    case JSONValueType::STRING:
        out << "\"" << value.stringValue << "\"";
        break;
    case JSONValueType::NUMBER:
        out << value.numberValue;
        break;
    case JSONValueType::BOOL:
        out << (value.boolValue ? "true" : "false");
        break;
    case JSONValueType::NIL:
        out << "null";
        break;
    default:
        throw std::runtime_error("Unknown JSONType encountered in writeJSON.");
    }
}
```

Извадка 2.

Основният метод parseValue() се грижи да преобразува стрингов вход в JSONValue обект (десериализация). Извадки са показани от кода на parseValue() и parseObject(). Аналогично за останалите типове данни съществува десериализиращ метод, ползван в parseValue(). Забелязва се, че за обхождане на входния стринг се използва обект Lexer, от който също така се извлича информация за текущ ред и колона – полезно за дебъг от потребителя.

```
JSONValue Parser::parseValue()
{
    switch (currentToken.type)
    {
    case TokenType::LEFT_BRACE:
        return parseObject();
    case TokenType::LEFT_BRACKET:
        return parseArray();
    case TokenType::STRING:
        return parseString();
    case TokenType::NUMBER:
        return parseNumber();
    case TokenType::TRUE:
        currentToken = lexer.nextToken();
        return parseBool(true);
    case TokenType::FALSE:
        currentToken = lexer.nextToken();
        return parseBool(false);
    case TokenType::NULL_TYPE:
        currentToken = lexer.nextToken();
        return parseNull();
    default:
        throw std::runtime_error("Unexpected token at line " +
std::to_string(lexer.getLine()) + ", column " +
std::to_string(lexer.getColumn()));
    }
}
```

Извадка 3.

```
JSONValue Parser::parseObject()
{
    JSONValue objectValue;
    objectValue.type = JSONValueType::OBJECT;

    currentToken = lexer.nextToken();
    if (currentToken.type != TokenType::RIGHT_BRACE)
    {
        while (true)
        {
            if (currentToken.type != TokenType::STRING)
            {
                throw std::runtime_error("Expected string key");
            }
            std::string key = currentToken.value;
            currentToken = lexer.nextToken();

            if (currentToken.type != TokenType::COLON)
            {
                throw std::runtime_error("Expected ':'");
            }
            currentToken = lexer.nextToken();
            objectValue.objectValue.push_back(KeyValue(key, new
JSONValue(parseValue())));
            if (currentToken.type == TokenType::COMMA)
            {
                currentToken = lexer.nextToken();
            }
            else
            {
                break;
            }
        }
        if (currentToken.type != TokenType::RIGHT_BRACE)
            throw std::runtime_error("Expected '}' at line " +
std::to_string(lexer.getLine()) + ", column: " + std::to_string(lexer.getColumn()));
    }

    currentToken = lexer.nextToken();
    return objectValue;
}
```

Извадка 4.

Ключова част от Lexer класа е именно методът nextToken(), чрез който се обхожда даден стринг.

```
Token Lexer::nextToken()
{
    skipWhitespace();

    if (pos >= input.length())
    {
        return {TokenType::END, ""};
    }

    char curr = input[pos];
    switch (curr)
    {
    case '{':
        pos++;
        return {TokenType::LEFT_BRACE, "{"};
    case '}':
        pos++;
        return {TokenType::RIGHT_BRACE, "}"};
    case '[':
        pos++;
        return {TokenType::LEFT_BRACKET, "["};
    case ']':
        pos++;
        return {TokenType::RIGHT_BRACKET, "]"};
    case ',':
        pos++;
        return {TokenType::COMMA, ","};
    case ':':
        pos++;
        return {TokenType::COLON, ":"};
    case '"':
        return parseString();
    case 't':
    case 'f':
    case 'n':
        return parseKeyword();
    default:
        if (isdigit(curr) || curr == '-')
        {
            return parseNumber();
        }
        throw std::runtime_error("Unexpected character at line " +
std::to_string(line) + ", column " + std::to_string(column));
    }
}
```

Извадка 5.

Следните методи се грижат за обработването на низове, числа и други ключови думи като true, false и null в *nextToken()* от Извадка 5.

```
Token Lexer::parseString()
{
    size_t start = pos + 1;
    advance();
    while (pos < input.size() && input[pos] != '"')
    {
        advance();
    }
    if (pos >= input.size())
    {
        throw std::runtime_error("Unterminated string at line " +
std::to_string(line) + ", column " + std::to_string(column));
    }
    size_t end = pos;
    advance();
    return {TokenType::STRING, input.substr(start, end - start)};
}

Token Lexer::parseNumber()
{
    size_t start = pos;
    while (pos < input.size() && (isdigit(input[pos]) || input[pos] == '.' ||
input[pos] == '-' || input[pos] == '+'))
    {
        advance();
    }
    return {TokenType::NUMBER, input.substr(start, pos - start)};
}

Token Lexer::parseKeyword()
{
    size_t start = pos;
    while (pos < input.size() && isalpha(input[pos]))
    {
        advance();
    }
    std::string keyword = input.substr(start, pos - start);
    if (keyword == "true")
        return {TokenType::TRUE, "true"};
    if (keyword == "false")
        return {TokenType::FALSE, "false"};
    if (keyword == "null")
        return {TokenType::NULL_TYPE, "null"};
    throw std::runtime_error("Invalid keyword '" + keyword + "' at line " +
std::to_string(line) + ", column " + std::to_string(column));
}
```

Извадка 6.

3. Заключение

Проектът JSON парсер в C++ демонстрира как може да се реализира мощен инструмент за манипулиране на JSON данни без използването на външни библиотеки. Ключовите компоненти, включващи класовете Lexer, Token, Parser, JSONValue и Engine, работят съвместно, за да осигурят ефективно парсване, валидиране и модификация на JSON документи.

В бъдеще могат да бъдат направени подобрения по потребителския интерфейс, както и да бъдат добавени методи по-подходящи за разработка на код.

Използвана литература:

https://www.json.org/json-bg.html

https://cplusplus.com/reference

https://stackoverflow.com/