JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 300–304
Received: 18 June 2020
Accepted: 6 July 2020

Performance comparison of chosen JSON parsers and a parser that employs a different reading method

Porównanie wydajności wybranych parserów JSON z parserem używającym innej metody odczytu

Przemysław Grzegorz Koter*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The analysis of parsing method of currently used JSON parsers, that are comprised of tokenizer and parser module, led to conclusion, that their performance could be improved. As a means to prove it, new JSON parser has been developed, whose modules are dedicated to process structure of either an object or an array. In both modules, tokenization step is combined with building step of document model, representing its structure. Performance test involved processing of sample document 1000 times, per 20 repeats, and was carried out by three compared parsers. Proposed parser was nearly 89% and 42% faster than other two, and its memory usage was mediocre. Results are promising enough to consider real-world usage, thus improving the efficiency of JSON processing.

Keywords: JSON parser; performance

Streszczenie
Analiza metody przetwarzania obecnie używanych parserów JSON, które są złożone z modułu tokenizera i parsera, doprowadziła do wniosku, że ich wydajność może być poprawiona. W celu dowiedzenia tego, opracowano parser, którego moduły są podzielonego względem przetwarzanej struktury, czyli obiektu i tablicy. W obu modułach krok wyodrębniania tokenów dokumentu jest połączony z krokiem budowy reprezentacji struktury dokumentu. Test wydajności obejmował przetworzenie przykładowego dokument 1000 razy w każdym z 20 powtórzeń, przez trzy porównywane parsery. Proponowany parser był szybszy o około 89% i 42% od pozostałych dwóch, jednocześnie zużycie pamięci było przeciętne. Wyniki są na tyle obiecujące, by rozważyć faktyczne użycie, poprawiając efektywność przetwarzania dokumentów JSON.

Słowa kluczowe: JSON; parser; wydajność

*Corresponding author
Email address: **przemyslaw.koter@pollub.edu.pl** (P. G. Koter)

1. Wstęp

Niniejszy artykuł traktuje o nowym podejściu w przetwarzaniu danych w formacie JSON [1], opartym na podziale parsera na moduły, rozpoznające i specjalizujące się w odczycie tylko obiektu lub tablicy. Według założeń taka budowa parsera i wybrany sposób odzwierciedlenia pozyskanych danych w modelu hierarchicznym ma uprościć wykonanie procesu odczytu, a co za tym idzie, poprawić jego wydajność. Realizacja założeń algorytmu jest zapewne możliwa w wielu językach programowania obiektowego, jednakże na potrzeby zbadania wydajności, w porównaniu do parserów opartych na innych założeniach, wybrano język C# [2].

2. Przegląd sposobu działania istniejących parserów

Temat budowy wydajności przetwarzania plików JSON został poruszony parokrotnie, jednakże nie w kontekście parserów ogólnego zastosowania. Analizowane publikacje skupiają się raczej na wydajności konkretnych czynności, jak poszukiwanie słów kluczowych [3] lub wyszukiwanie analityczne [4]. W tej sytuacji można mówić o proponowaniu nowej metody przetwarzania, różniącej się od istniejących implementacji.

Parsery realizujące konkurencyjną metodę przetwarzania zostały zaimplementowane w bibliotekach Newtonsoft.Json i System.Text.Json. Pierwsza z bibliotek została wybrana ze względu na jej największą popularność [5], druga zaś jako oficjalna implementacja metod przetwarzania dokumentów JSON z biblioteki standardowej platformy .NET Core [6]. Na poziomie konceptualnym rozwiązania implementowane przez oba parsery są bardzo podobne. Parser tego rodzaju wprowadza przetwarzanie w dwóch modułach. Pierwszy z nich zajmuje się wyodrębnieniem symboli strukturalnych formatu i pojedynczych wartości, takich jak ciąg tekstowy, liczba lub literał, przy każdym wywołaniu. Zapisuje też stan przetwarzania, czyli, między innymi, aktualną pozycję odczytu w dokumencie i stos zawierający informacje o typach struktur formujących obecny poziom zagnieżdżenia. Przy napotkaniu końca struktury (obiektu lub tablicy) na poziomie *n* hierarchii, moduł musi przywrócić informacje o typie struktury rodzica z poziomu *n-1*. Drugi z modułów tworzy model hierarchiczny, reprezentujący budowę odczytywanego dokumentu. Informacje o typie odczytywanej wartości i struktury pozyskuje od pierwszego modułu. Przełączenie pomiędzy różnymi typami odczytywanej struktury

realizuje w postaci umówionych kodów stanu. Funkcja odczytująca pierwszego modułu jest wywoływana w pętli modułu drugiego, dopóki nie zakończono odczytu dokumentu. Moduł drugi jest z reguły nieświadomy, jakiego rodzaju wartość zostanie zwrócona przy następnym wywołaniu.

Różnica między parserami przejawia się w typie struktur służących przechowaniu odczytanych danych. Pierwszy z nich zapisuje wartości z obiektu i tablicy JSON odpowiednio jako obiekty słownika i listy, które rozszerzają się w miarę dodawania wartości. Drugi przechowuje wszystkie wartości w tablicy bajtów, utrzymując specyficzną strukturę i zasady zapisu. Tablica jest alokowana adekwatnie do rozmiaru całego dokumentu.

3. Omówienie proponowanej metody przetwarzania

Proponowana zmiana opiera się na wprowadzeniu dwóch modułów, przeznaczonych do budowania reprezentacji obiektu i tablicy. Dzięki tej zmianie liczba potencjalnych kombinacji symboli jest ograniczona i przewidywalna, a co za tym idzie, moduł taki może bezpośrednio odczytywać znaki z dokumentu. Funkcja przetwarzania może być wielokrotnie wywoływana przez samą siebie w przypadku odczytu zagnieżdżonych struktur, zaś kończy wykonanie przy napotkaniu końca struktury, zwracając aktualną pozycję w dokumencie. W efekcie stan odczytu istnieje tylko na stosie, powielony *n* razy, w miejscach wywołań procedury, gdzie *n* to poziom zagnieżdżenia aktualnej struktury. Stan jest efektywnie reprezentowany przez pozycję w dokumencie, gdyż typ struktury jest zaszyty w kodzie wykonywanej procedury.

W kwestii reprezentacji danych dokumentu zdecydowano się na wybór listy jednokierunkowej, jako struktury stosowanej do reprezentacji zarówno obiektu i tablicy JSON. Zaletą takiego podejścia jest zredukowana liczba realokacji. W celu jednoczesnego przyśpieszenia późniejszych wyszukiwań wartości z obiektu JSON zastosowano grupowanie po typie. Wszystkie wartości nadal są dodawane do jednej listy, jednakże wstawienie wartości może zajść w jednej z trzech referencji, odpowiednio dla obiektów, tablic i wartości prymitywnych (literałów null, true i false, wartości liczbowych i ciągów tekstowych). Obiekt reprezentujący tablicę tworzy dwie listy – listę zawierającą obiekty i tablice, i drugą, zawierającą wartości prymitywne.

W samych metodach budowy modelu pola będące końcami listy jednokierunkowej nie są weryfikowane pod kątem przechowywania wartości null (brak obiektu). To uproszczenie wymaga jednak, by w konstruktorze klasy reprezentującej obiektu lub tablicę JSON umieścić kod tworzący obiekty wartości i zapisujący je pod te pola. Obiekty te nie otrzymają nigdy faktycznej wartości pochodzącej z dokumentu, dlatego na koniec przetwarzania muszą zostać usunięte z list.

Wartości były początkowo zawsze pozyskiwane jako łańcuch tekstowy, jednak takie podejście pokazało poważne braki wydajnościowe i zwiększone użycie pamięci. W ramach poprawy zmieniono sposób odczytu wartości liczbowych, by były od razu konwertowane na odpowiadające im typy long (liczba całkowita) lub double (liczba zmiennoprzecinkowa).

Podczas implementacji metod przetwarzania przyjęto strategię optymalizacji odczytu dla dokumentów zminimalizowanych, czyli pozbawionych wszystkich znaków białych. Jeśli odczytany znak *c* jest znakiem niedrukowalnym (białym), wówczas trzeba wywołać metodę która odszuka pierwszy drukowalny znak. Jeśli jednak znak *c* jest znakiem drukowalnym, to oszczędność polega na pominięciu wywołania metody i wykonania minimum 1 sprawdzenia wartości znaku. Sprawdzenie, czy znak jest drukowalny, może być przeprowadzone poprzez porównanie wartości liczbowej znaku *c* do spacji, co wynika z ułożenia tablicy znaków ASCII [7] (Listing 1).

Listing 1: Kod metody przetwarzającej obiekt JSON – pierwsza i ostatnia instrukcja if odpowiada za uproszczenie znajdywania początku klucza i wartości w obiekcie, jeśli nie ma białych znaków. Metoda FindValue znajduje pozycję w dokumencie za znakiem dwukropka (separatora między kluczem i wartością).

```
internal override int ParseAll(string json, int start)
{
    int c;
    JsonToken token;
    next_token:
    if ((c = json[++start]) == '"')
        goto parse_name;

    start = FindProperty(json, start, ref c);
    if (c == '}')
        goto ret;

    parse_name:
    int index = start + 1;
    start     = FindValue(json, index, out int nameLength);
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);
```

4. Metoda przeprowadzenia testów wydajności

Poprawnie przyprowadzone testy wydajności, pozwalające uzyskać rzetelne wyniki, są niezbędne do oceny przydatności zastosowanego rozwiązania. Proces odczytu przykładowego dokumentu może trwać bardzo krótko (poniżej 1 milisekundy), co czyni trudnym dokładne zbadanie różnic w czasach wykonania. Rozwiązaniem tego problemu jest powtórzenie kroku przetwarzania wielokrotnie. W przeprowadzonym teście liczba powtórzeń *n* wynosi 1000. Zmierzony czas jest łącznym czasem potrzebnym do wykonania odczytu *n* razy.

Wyniki nie zostały przeliczone na czas jednostkowy, czyli czas wykonania pojedynczego odczytu, ponieważ bardziej interesująca jest relacja wyników pomiędzy różnymi parserami. Drugim zagadnieniem jest liczba przeprowadzonych prób. Nie może być zbyt mała, gdyż nie ma wtedy możliwości stwierdzenia, czy otrzymane wyniki nie są skrajne.

Listing 2: Kod metody przetwarzającej obiekt JSON – instrukcje w blokach case służą utworzeniu obiektów modelu i ustawieniu odpowiednich referencji listy. Metoda LoadBools wczytuje literały true, false i null. Metoda LoadNumber tworzy obiekt klasy dziedziczący po JsonToken, który posiada dodatkowe pole z wartością typu long lub double.

```
switch (c)
{
    case '{':
        token = new JsonObject((JsonToken)this);
        token._next = _lastObject._next;
        _lastObject._next = token;
        _lastObject = token;
        parse_composite:
        start = token.ParseAll(json, start);
        break;
    case '[':
        token  = new JsonArray((JsonToken)this);
        _lastArray._next = token;
        _lastArray = token;
        goto parse_composite;
    default:
        if (c == '"')
        {
            token = new JsonToken();
            start = token.LoadString(json, start);
        }
        else if (c > '9')
        {
            token = new JsonToken();
            start = token.LoadBools(json, start, c);
        }
        else
        {
            start = LoadNumber(json, start, c);
            token = _last;
            break;
        }

        AddValue(token);
        break;
}
```

Listing 3: Kod metody przetwarzającej obiekt JSON – wszystkim wartościom ustawia się klucz w polu name. Pierwsza instrukcja if odpowiada za pominięcie wywołania metody poszukiwawczej, jeśli po wartości nie ma znaku białego. Kolejny if przenosi kontrolę na początek metody, w celu przetworzenia kolejnej wartości. Na końcu następuje usunięcie pustych obiektów wartości.

```
token._name = json.Substring(index, nameLength);
_count++;
if ((c = json[++start]) <= ' ')
    start = FindFirstAnything(json, start, out c);

if (c == ',')
    goto next_token;

if (c != '}')
    throw new InvalidJsonException();

ret:
TrimEmpty();
return start;
}
```

W teście przyjęto liczbę prób *p* równą 20. Podczas pojedynczej próby parsery wykonują po kolei *n* odczytów. Przetwarzany dokument jest zachowany w pamięci operacyjnej jako łańcuch znaków, co eliminuje opóźnienia związane z odczytem pliku z dysku. Cały test został przeprowadzony trzy razy, co pozwoliło upewnić się, że wyniki są powtarzalne. W przypadku pomiarów użycia pamięci operacyjnej można mówić o wartościach przybliżonych. Wynika to z faktu, że wykorzystane API zwraca tylko „najlepsze możliwe przybliżenie" zaalokowanych bajtów w zarządzanej pamięci [8]. Dzięki zastosowaniu odpowiednio dużego dokumentu całkowite użycie pamięci jest wielokrotnie większe niż błąd pomiaru. Nadto wyniki te są traktowane jako wartości referencyjne, gdyż główne kryterium użyteczności to czas wykonania.

Listing 4: Kod metody przetwarzającej tablicę JSON

```
internal override int ParseAll(string json, int start)
{
    JsonComposite token;
    int           c;
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    if (c == ']')
        goto ret;

    goto value_parse;

    next_value:
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    value_parse:
    switch (c)
    {
        case '{':
            token = new JsonObject((JsonToken)this);
            _flags |= (int)JSymbol.OBJECT;
            parse_composite:
            _lastObject._next = token;
            _lastObject = token;
            start = token.ParseAll(json, start);
            token._index = _count;
            break;
        case '[':
            token = new JsonArray((JsonToken)this);
            _flags |= (int)JSymbol.ARRAY;
            goto parse_composite;
        default:
            _valueCount++;
            JsonValue value;
            if (c == '"')
            {
                value = new JsonValue();
                start = value.LoadString(json, start);
            }
            else if (c > '9')
            {
                value = new JsonValue();
                start = value.LoadBools(json, start, c);
            }
            else
            {
                start = LoadNumber(json, start, c);
                break;
            }

            AddValue(value);
            break;
    }

    _count++;
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    if (c == ',')
        goto next_value;

    if (c != ']')
        throw new InvalidJsonException();

    ret:
    TrimEmpty();
    return start;
}
```

4.1. Charakterystyka testu

Na potrzeby zbadania i porównania wydajności parserów opracowano test, w którym każda z bibliotek wykonuje odczyt i buduje reprezentację dokumentu, zawartego w obiekcie łańcucha znaków (klasy string). Dokument składa się z tablicy zawierającej 100 identycznych obiektów. Struktura pojedynczego obiektu została przedstawiona na listingu 5. Dane zawarte w dokumencie nie mają znaczenia dla przeprowadzanego testu.

Listing 5: Struktura obiektu JSON, składającego się na dokument testowy.

```
{"nullvalue":null,"falseValue":false,"truevalue":true,
"stringvalue1":"30","stringvalue2":"40",
"stringvalue3":"50","stringvalue4":"60",
"stringvalue5":"70","stringvalue6":"80",
"stringvalue7":"90","stringvalue8":"100",
"stringvalue9":"110","stringvalue10":"120",
"stringvalue11":"130","stringvalue12":"140",
"stringvalue13":"150","stringvalue14":"160",
"stringvalue15":"170","stringvalue16":"180",
"stringvalue17":"190","stringvalue18":"200",
"stringvalue19":"210","stringvalue20":"220",
"stringvalue21":"230","stringvalue22":"240",
"stringvalue23":"250","stringvalue24":"260",
"stringvalue25":"270","stringvalue26":"280",
"stringvalue27":"290","stringvalue28":"300",
"stringvalue29":"310","stringvalue30":"320","iuwwbhh1.kex":
[489,879,1269,1659,2049,2439,2829,3219,3609,3999,4389,4779,
5169,5559,5949,6339,6729,7119,7509,7899,8289,8679,9069,
9459,9849,10239,10629,11019,11409,11799,12189,12579,12969,
13359,13749,14139,14529,14919,15309,15699,16089,16479,
16869,17259,17649,18039,18429,18819,19209,19599]}
```

Tabela 1: Wartości pomiarów czasu realizacji zadania testowego.

Numer próby	Newtonsoft.Json (ms)	System.Text.Json (ms)	Opisany parser (ms)
1.	3723	723	411
2.	3574	659	374
3.	3583	657	387
4.	3543	665	385
5.	3567	658	379
6.	3593	660	383
7.	3558	664	393
8.	3559	659	382
9.	3570	657	384
10.	3564	665	382
11.	3558	657	378
12.	3578	657	378
13.	3537	657	379
14.	3544	656	383
15.	3541	656	380
16.	3542	657	380
17.	3540	662	376
18.	3577	659	385
19.	3548	657	380
20.	3552	659	380

Tabela 2: Średnia pomiarów czasu dla 20 prób.

	Newtonsoft.Json (ms)	System.Text.Json (ms)	Opisany parser (ms)
Średnia	3567,55	662,2	382,95

Tabela 3: Wartości pomiarów użycia pamięci podczas realizacji zadania testowego. We wszystkich próbach uzyskano jednakowe wartości.

Próby	Newtonsoft.Json (B)	System.Text.Json (B)	Opisany parser (B)
1.-20.	2105344	401456	737280

5. Wnioski

Opisany parser w przeprowadzonym zadaniu testowym osiągnął średni wynik około 42% lepszy niż System.Text.Json, a także 89% szybszy niż Newtonsoft.Json. Niska wydajność biblioteki Newtonsoft.Json była przyczyną prac nad nową biblioteką standardową do obsługi dokumentów JSON, a także nad omawianym parserem. Użycie pamięci przez proponowany parser uplasowało się pomiędzy wynikami obu bibliotek, a uzyskana wartość jest bliższa niższej. Na podstawie pozyskanych rezultatów można wysunąć wniosek, że parser o zaproponowanej budowie lepiej radzi sobie z zadaniami, w których wynikiem jest reprezentacja danych dokumentu. Niekoniecznie musi mieć to przełożenie przy np. wyszukiwaniu informacji w dokumencie. Do przeprowadzenia mapowania danych z dokumentu na podany typ (w procesie zwanym deserializacją) parser musiałby zostać zmodyfikowany, poprzez zamianę kodu odpowiedzialnego za budowę obiektów hierarchii, na kod tworzący obiekty danych typów i konwertujący wartości tekstowe na wartości typów prymitywnych. By w pełni ocenić charakterystykę wydajności takiego sposobu przetwarzania, trzeba wykonać wiele testów, z różnymi danymi testowymi. Istnieje bowiem szansa, że wykonywanie wielu małych alokacji będzie sprawdzało się gorzej, gdy rozmiar dokumentu znacznie wzrośnie (np. stukrotnie).

Na ten moment pozostaje jedynie możliwość spekulacji wpływu niektórych decyzji. Prawdopodobnie usunięcie dodatkowych referencji do fragmentów listy wartości, w klasie reprezentującej obiekt JSON, pozwoliłoby poprawić ten wynik dzięki uproszczeniu procesu budowania listy. Mając tylko jedną referencję można też zrezygnować z wstawiania obiektów pustych wartości, co przełoży się na zmniejszenie użycia pamięci.

Literatura

[1] T. Bray i Ed., „The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 8259, 2017.

[2] Standard ECMA-334, C# Language Specification, https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf [06.06.2020].

[3] E. Wahyudi, S. Sfenrianto, M. J. Hakim, R. O. Subandi, R. Sulaeman i R. Setiyawan, „Information Retrieval System for Searching JSON Files with Vector Space Model Method," w 2019 International Conference of Artificial Intelligence and Information Technology (ICAIIT), Yogyakarta, Indonesia, Indonesia, 2019.

[4] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein i D. Kossmann, „Mison: a fast JSON parser for data analytics," Proceedings of the VLDB Endowment, 2017.

[5] Strona repozytorium bibliotek Nuget.org, lista popularnych bibliotek, Microsoft, https://www.nuget.org/packages [06.06.2020].

[6] Strona blogu z nowościami technicznymi, „Try new System.Text.Json APIs", Microsoft, https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-apis/ [06.06.2020].

[7] ANSI, ISO-IR-006: ASCII Graphic character set, https://www.itscj.ipsj.or.jp/iso-ir/006.pdf (1975-12-01), [06.06.2020].

[8] Strona dokumentacji metody użytej przy pomiarach użycia pamięci, https://docs.microsoft.com/en-us/dotnet/api/system.gc.gettotalmemory [06.06.2020].