Roman Scharkov

Backend Engineer

at tutti.ch

smg swiss marketplace group

Real Estate

homegate.ch
ImmoStreet.ch
Acheter-Louer.ch
CASASOFT
ImmoScout24
home.ch
IAZI CIFI
PUBLIMMO

Automotive

CAR FOR YOU
AutoScout24
MotoScout24

General Marketplaces

tutti.ch
anibis.ch
Ricardo

Finance & Insurance

FinanceScout24

```
{
    "first_name": "Roman",
    "last_name":  {"de": "Scharkov", "ua": "Шарков"}

    "employer_url":         "https://tutti.ch",
    "github_username":      "romshark",
    "":                     "",
    "aprx_atoms_in_body": 70000000000000000000000000000,
    "agent_number":         007,
    "":                     "",
    "favorite_character": "\u00",
    "favorite_emoji":       "💣",
    "home_dir":             "C:\Users\roman",
}
```

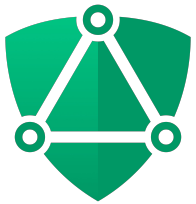
```
{
    "first_name": "Roman",
    "last_name":  {"de": "Scharkov", "ua": "Шарков"},

    "employer_url":         "https://tutti.ch",
    "github_username":      "romshark",
    "":                     "",
    "aprx_atoms_in_body": 700000000000000000000000000,
    "agent_number":         7,
    "":                     "",
    "favorite_character": "\u0000",
    "favorite_emoji":       "💣",
    "home_dir":             "C:\\Users\\roman"
}
```

▲ Parsing JSON is a Minefield (seriot.ch)

339 points by moks on April 22, 2018 | hide | past | favorite | 246 comments

http://seriot.ch/projects/parsing_json.html



GraphGuard


GraphGuard
2022 - 2023

Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get predictable results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

Internet

ggproxy
Proxy Server

GraphQL
Server

Nobody likes slow software.
Nobody likes slow servers.

🚀 More **throughput** means less servers 💸

⚡ Lower **latencies** mean better UX 😊

🔋 Better **resource usage** Eco-friendly 💸🌳

Performance Oriented Programming

1. Be aware of the hardware your code runs on.
2. Design that enable compilers to generate better code.
3. Design that enables good (re)usage of memory 🚭
   a. No dynamic memory allocation at runtime 🚫
   b. Allocate memory at init and reuse ♻️

Difficult to follow when 3rd party code is involved. 😢

BTW: check out Casey Muratori's (game engine R&D) courses
to learn more about performance oriented programming

GraphQL query with variables

```
mutation(
    $title: String!
    $description: String
    $assignees: [ID!]!
    $scope: TaskScope!
) {
    createTask(
        title: $title,
        description: $description
        assignees: $assignees
        scope: $scope
    )
}
```

Variables as JSON values

```
{
    "title": "New Task",
    "description": null,
    "assignees": ["foo", "bar"],
    "scope": {
        "teams": ["Aurora", "Bumblebee"],
        "sprint": 21
    }
}
```

{json}


```
{"title":"NewTask","description":null,"
assignees":["foo","bar"],"scope":{"team
s":["Aurora","Bumblebee"],"sprint":21}}
```


Engine
Code

```
A2 B9 90 E7 25 FF F1 59 21 2A FA 82
10 12 37 7B EE …
```

Module	GitHub ⭐	License	First Commit
github.com/tidwall/gjson	12.9k	MIT	August 2016 (oldest)
github.com/gofaster/jx	125	MIT	October 2022
github.com/valyala/fastjson	2k	MIT	May 2018
github.com/bytedance/sonic	5.5k	Apache-2.0	May 2021
github.com/ohler55/ojg	641	MIT	April 2020
github.com/json-iterator/go	12.6k	MIT	November 2016
github.com/goccy/go-json	2.5k	MIT	April 2020
github.com/romshark/jscan	58	BSD-3-Clause	January 2022 (new kid on the block 😎)

What is jscan?

- Most efficient (maybe?) **JSON iterator** and validator for Go.
- **Pure Go,** no cgo, no platform-specific ASM.
- **RFC8259 compliant** (tested against the JSON Parsing Test Suite).

What it's not:

- No Marshal/Unmarshal*

* but with jscan you can build your own task-specific parser
  when you need the extra efficiency!

```go
type JSONValueType int8

const (
    _ JSONValueType = iota
    JSONValueTypeObject
    JSONValueTypeArray
    JSONValueTypeString
    JSONValueTypeNumber
    JSONValueTypeBoolean
    JSONValueTypeNull
)

type GraphQLVariablesTraverser interface {
    // TraverseJSON calls onVariable for every GraphQL variable encountered in input.
    // Returns an error if input is invalid JSON or doesn't contain an object.
    TraverseJSON(
        input []byte,
        onVariable func(name []byte, t JSONValueType),
    ) error
}
```

github.com/romshark/pres-jscan

```go
import "encoding/json"

type EncodingJSON struct{}

func (EncodingJSON) TraverseJSON(
    input []byte, onVar func(name []byte, t JSONValueType),
) error {
    var m map[string]any
    if err := json.Unmarshal(input, &m); err ≠ nil {
        return err
    }
    for k, v := range m {
        tp := JSONValueTypeNull
        switch v.(type) {
        case map[string]any: tp = JSONValueTypeObject
        case []any:          tp = JSONValueTypeArray
        case string:         tp = JSONValueTypeString
        case int, float64:   tp = JSONValueTypeNumber
        case bool:           tp = JSONValueTypeBoolean
        }
        onVar([]byte(k), tp)
    }
    return nil
}
```

In any situation,
Use the std library first 🙂

1. Unmarshal to map[string]any
2. Loop over key-value pairs
3. Use type switch to determine the value type

```json
{
  "object": {
    "key": "value",
    "subobject": {
      "subarray_of_subobject": [
        null, 🪨, 3.7735735335,
        12355.122334, null
      ]
    }
  },
  "array2D": [
    [1,2,3], null, [], null, [1]
  ],
  "string": 🪨,
  "number": 3.1415,
  "true": true,
  "false": false,
  "null": null
}
```

🪨 = Long Strings

goos: linux; goarch: amd64
cpu: AMD Ryzen 7 5700X

encoding/json is rather wasteful

44828 ns/op 😕

14896 B/op

77 allocs/op 👈 😐

```go
type EncodingJSONOptimized struct{}

var encodingJSONOptCharMap = [256]JSONValueType{
    '{': JSONValueTypeObject, '[': JSONValueTypeArray,
    '"': JSONValueTypeString,

    '0': JSONValueTypeNumber, '1': JSONValueTypeNumber,
    '2': JSONValueTypeNumber, '3': JSONValueTypeNumber,
    '4': JSONValueTypeNumber, '5': JSONValueTypeNumber,
    '6': JSONValueTypeNumber, '7': JSONValueTypeNumber,
    '8': JSONValueTypeNumber, '9': JSONValueTypeNumber,

    '-': JSONValueTypeNumber, 't': JSONValueTypeBoolean,
    'f': JSONValueTypeBoolean, 'n': JSONValueTypeNull,
}

func (EncodingJSONOptimized) TraverseJSON(
    input []byte, onVar func(name []byte, t JSONValueType),
) error {
    var m map[string]json.RawMessage
    if err := json.Unmarshal(input, &m); err ≠ nil {
        return err
    }
    for k, v := range m {
        onVar([]byte(k), encodingJSONOptCharMap[v[0]])
    }
    return nil
}
```

goos: linux; goarch: amd64
cpu: AMD Ryzen 7 5700X

42611 ns/op

7400 B/op

36 allocs/op

better, but still meh □

https://github.com/romshark/pres-jscan/tree/main/parsejson

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/bytedance/sonic is very fast 💪 due to SIMD
but only on AMD64. On ARM it's one of the worst 🤦

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/tidwall/gjson is pretty good on both AMD64 and ARM 💪
But still allocates memory and it's still rather slow in comparison

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
tidwall_gjson	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/json-iterator/go comes close to 0 allocations,
but it's still slow and still allocates a little memory

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
tidwall_gjson	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
jsoniter	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
jsoniter_unsafe	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/gofaster/jx doesn't allocate memory 🎉
and it's very efficient on both AMD64 and ARM! 💪🏻
but can we do even better? 🤓

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
tidwall_gjson	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
jsoniter	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
jsoniter_unsafe	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
gofaster_jx	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/romshark/jscan can still squeeze out a little more 🙂
But how on earth is **github.com/valyala/fastjson** so fast?! 🤯
2x jscan; ~60x encoding/json

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
tidwall_gjson	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
jsoniter	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
jsoniter_unsafe	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
gofaster_jx	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
romshark_jscan	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
valyala_fastjson	712 ns/op	0 allocs/op	795 ns/op	0 allocs/op

AMD Ryzen 7 5700X & Apple M1 Max

github.com/valyala/fastjson takes advantage of package **bytealg** from std library providing hand-optimized ASM for different platforms.

https://go.googlesource.com/go/+/refs/heads/master/src/internal/bytealg/

```
1  package bytealg
2
3  //go:noescape
4  func IndexByte(b []byte, c byte) int
5
6  //go:noescape
7  func IndexByteString(s string, c byte) int
8
```

- indexbyte_386.s
- indexbyte_amd64.s
- indexbyte_arm.s
- indexbyte_arm64.s
- indexbyte_generic.go
- indexbyte_loong64.s
- indexbyte_mips64x.s
- indexbyte_mipsx.s
- indexbyte_native.go
- indexbyte_ppc64x.s
- indexbyte_riscv64.s
- indexbyte_s390x.s
- indexbyte_wasm.s

```
func main() {
        // This string contains an illegal control character \u0000
        j := "\"a\u0000a\""

        // Validation works correctly, but the parser doesn't care!
        fmt.Println("Validate:")
        fmt.Println(fastjson.Validate(j))

        fmt.Println("Now let's just iterate:")
        p := new(fastjson.Parser)
        v, err := p.Parse(j)
        if err != nil {
                panic(err)
        }
        fmt.Println(v.Type())
        fmt.Println(string(v.GetStringBytes()))
}
```

```
Validate:
cannot parse JSON: string cannot contain control char 0x00; unparsed tail: ""
Now let's just iterate:
string
aa

Program exited.
```

7. Strings

The representation of strings is similar to conventions used in the C family of programming languages. A string begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks, except for the characters that MUST be escaped: quotation mark, reverse solidus, and the control characters (U+0000 through U+001F).

fastjson is violating ☝ RFC8259

Parser does not fully validate the JSON #88

Open mna opened this issue on Jan 20 · 1 comment

github.com/valyala/fastjson is disqualified for violating RFC8259.
jscan wins 🍾🥳🎉 … but does it really or is the difference just noise? 🧐

Implementation	5700X time	5700X allocs	M1 time	M1 allocs
encoding_json	57526 ns/op	77 allocs/op	29808 ns/op	77 allocs/op
encoding_json_opt	42500 ns/op	36 allocs/op	22315 ns/op	36 allocs/op
bytedance_sonic	4653 ns/op	12 allocs/op	24005 ns/op	46 allocs/op
tidwall_gjson	32290 ns/op	3 allocs/op	17950 ns/op	32 allocs/op
jsoniter	32205 ns/op	32 allocs/op	18040 ns/op	25 allocs/op
jsoniter_unsafe	30673 ns/op	25 allocs/op	17400 ns/op	3 allocs/op
gofaster_jx	1629 ns/op	0 allocs/op	1773 ns/op	0 allocs/op
romshark_jscan	1590 ns/op	0 allocs/op	1616 ns/op	0 allocs/op
~~valyala_fastjson~~	~~712 ns/op~~	~~0 allocs/op~~	~~795 ns/op~~	~~0 allocs/op~~

github.com/romshark/jscan-benchmark
5700X = Ryzen 7 5700X | M1 = Apple M1

jscan is **38.1%** faster on **AMD64** and **46.1%** faster on **ARM**

validate large_26m.json (26 mb); sorted by (5700X %)

Implementation	5700X tm	M1 tm	5700X mem	M1 mem	5700X %	M1 %
romshark_jscan	12 ms	11.1 ms	0	0	100 %	100 %
gofaster_jx	19.4 ms	20.6 ms	0	0	61.9 %	53.9 %
ohler55_ojg_oj	21.8 ms	22.9 ms	0	0	55.0 %	48.5 %
tidwall_gjson	22.4 ms	27.4 ms	0	0	53.6 %	40.5 %
valyala_fastjson	22.5 ms	25.8 ms	0	0	53.3 %	43 %
minio_simdjson	42 ms	N/A	14	N/A	28.6 %	N/A
jsoniter	45.3 ms	43.1 ms	644,360	644,360	26.5 %	25.8 %
bytedance_sonic	74.5 ms	68.9 ms	0	0	16.2 %	16.1 %
encoding_json	74.6 ms	68.6 ms	0	0	16.1 %	16.2 %
goccy_go_json	18.6 s	7.2 s	2,335,782	2,335,820	0.06 %	0.2 %

github.com/romshark/jscan-benchmark

5700X = Ryzen 7 5700X | M1 = Apple M1 | % = efficiency relative to jscan

jscan is **21.4%** faster on **AMD64** and **33.3%** faster on **ARM**

validate nasa_SxSW_2016_125k.json (125kb); sorted by (5700X %)

Implementation	5700X tm	M1 tm	5700X mem	M1 mem	5700X %	M1 %
romshark_jscan	80.6 µs	85.9 µs	0	0	100 %	100 %
tidwall_gjson	102.5 µs	128.8 µs	0	0	78.6 %	66.7 %
gofaster_jx	129.1 µs	140.8 µs	0	0	62.5 %	61 %
ohler55_ojg_oj	145 µs	154.2 µs	0	0	55.6 %	55.8 %
valyala_fastjson	183.2 µs	286.5 µs	0	0	44 %	30 %
bytedance_sonic	322 µs	352.4 µs	0	0	25 %	24.4 %
encoding_json	337 µs	351.7 µs	0	0	23.9 %	24.4 %
jsoniter	404.1 µs	237 µs	2,121	2,121	20 %	36.3 %
minio_simdjson	440.1 µs	N/A	13	N/A	18.3 %	N/A
goccy_go_json	4.5 ms	2.9 ms	20,801	20,801	1.8 %	3 %

github.com/romshark/jscan-benchmark

5700X = Ryzen 7 5700X | M1 = Apple M1 | % = efficiency relative to jscan

jscan is **14.3%** faster on **AMD64** and **29.8%** faster on **ARM**

validate small_336b.json (336 bytes); sorted by (5700X %)

Implementation	5700X tm	M1 tm	5700X mem	M1 mem	5700X %	M1 %
romshark_jscan	212.1 ns	237.6 ns	0	0	100 %	100 %
tidwall_gjson	247.8 ns	338.7 ns	0	0	85.7 %	70.2 %
valyala_fastjson	299.3 ns	379.8 ns	0	0	70.9 %	62.6 %
ohler55_ojg_oj	339.2 ns	376.3 ns	0	0	62.5 %	63.2 %
gofaster_jx	355.6 ns	385 ns	0	0	59.5 %	61.8 %
jsoniter	807.8 ns	690.2 ns	7	7	26.3 %	34.4 %
encoding_json	926.2 ns	893.7 ns	0	0	23 %	26.6 %
bytedance_sonic	924.9 ns	898.1 ns	0	0	22.9 %	26.5 %
goccy_go_json	6.1 µs	2.5 µs	61	61	3.5 %	9.5 %
minio_simdjson	18.8 µs	N/A	9	N/A	1.1 %	N/A

github.com/romshark/jscan-benchmark

5700X = Ryzen 7 5700X | M1 = Apple M1 | % = efficiency relative to jscan

jscan is **75.3% slower** on **AMD64** and **7.5% slower** on **ARM**

validate tiny_8b.json (`{"x":0}`); sorted by (5700X %)

Implementation	5700X tm	M1 tm	5700X mem	M1 mem	5700X %	M1 %
tidwall_gjson	14.6 ns	16.2 ns	0	0	175.3 %	107.5 %
ohler55_ojg_oj	17.9 ns	21.2 ns	0	0	143 %	82.1 %
valyala_fastjson	21.3 ns	19.9 ns	0	0	120 %	87.4 %
romshark_jscan	25.6 ns	17.4 ns	0	0	100 %	100 %
gofaster_jx	29.9 ns	29.3 ns	0	0	85.6 %	59.5 %
encoding_json	42.3 ns	45.7 ns	0	0	60.5 %	38.1 %
bytedance_sonic	43.3 ns	46.3 ns	0	0	59.1 %	37.6 %
jsoniter	45 ns	46.2 ns	0	0	56.9 %	37.7 %
goccy_go_json	806.8 ns	351.5 ns	9	9	3.2 %	5 %
minio_simdjson	20 µs	N/A	11	N/A	0.1 %	N/A

github.com/romshark/jscan-benchmark

5700X = Ryzen 7 5700X | M1 = Apple M1 | % = efficiency relative to jscan

```
[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[EOF
```

```
parseArray() error
  parseArray() error
    parseArray() error
      parseArray() error
        parseArray() error
          parseArray() error
            return err
          return err
        return err
      return err
    return err
  return err
return err
```

1. Grow a deep call stack
2. Encounter an error
3. Unwind the call stack

```
[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[EOF

parseArray() error
  parseArray() error
    parseArray() error
      parseArray() error
        parseArray() error
          parseArray() error
            return fmt.Errorf("parsing array: %w", err)
          return fmt.Errorf("parsing array: %w", err)
        return fmt.Errorf("parsing array: %w", err)
      return fmt.Errorf("parsing array: %w", err)
    return fmt.Errorf("parsing array: %w", err)
  return fmt.Errorf("parsing array: %w", err)
return fmt.Errorf("parsing array: %w", err)
```

Benchmark 4: stack unwinding attack (1kb)

validate unwind_stack (1kb; "[" repeated 1024 times); sorted by (5700X %)

Implementation	5700X tm	M1 tm	5700X mem	M1 mem	5700X %	M1 %
romshark_jscan	1564 ns	1663 ns	0	0	100 %	100 %
ohler55_ojg_oj	1608 ns	1682 ns	0	0	97.2 %	98.9 %
tidwall_gjson	4 µs	14 µs	0	0	39.1 %	11.9 %
encoding_json	5.1 µs	5.1 µs	1	1	30.7 %	32.6 %
bytedance_sonic	5.2 µs	5.1 µs	1	1	30 %	32.6 %
minio_simdjson	19.6 µs	N/A	9	N/A	8 %	N/A
gofaster_jx	767 µs	482.7 µs	1,026	1,026	2 %	3.4 %
jsoniter	108.9 µs	66.9 µs	1,033	1,033	1.4 %	2.5 %
goccy_go_json	321 µs	155.6 µs	4,105	4,105	0.5 %	1 %
valyala_fastjson	10.2 ms	5.7 ms	4,145	4,141	0.02 %	0.03 %

github.com/romshark/jscan-benchmark

5700X = Ryzen 7 5700X | M1 = Apple M1 | % = efficiency relative to jscan

ROMAN SCHARKOV COLLECTION

PARSING JSON, THE HARD PART

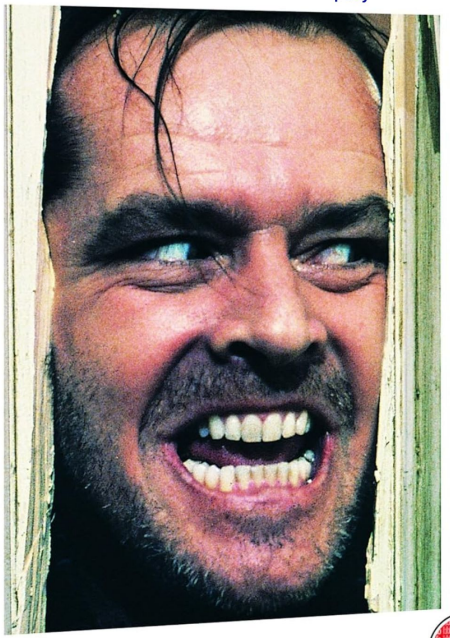
Z1
21156

PARSING JSON, THE HARD PART

State Machine

Streeng Parser

Skipper Spacy



State Macheene

Mad, insane. Nobody likes him. But he still manages to get casted to roles nobody else can play.

```
goto VALUE
goto OBJ_KEY
goto FRACTION
goto CHECK_INT
goto VALUE_TRUE
goto VALUE_NULL
goto CHECK_FRAC
goto VALUE_ARRAY
goto VALUE_FALSE
goto AFTER_VALUE
goto VALUE_NUMBER
goto VALUE_OBJECT
goto VALUE_STRING
goto EXPONENT_SIGN
goto AFTER_OBJ_KEY_STRING
goto CHECK_STRING_CHARACTER
goto CHECK_FIELDNAME_STRING_CHARACTER
```

1. 17 labels & 95 goto 👻
2. Only 1 function call
3. No nested calls
4. Small inlined functions

~644 LoC State-Machine

```
VALUE:
    switch s[0] {
    case '{':
        goto VALUE_OBJECT
    case '[':
        goto VALUE_ARRAY
    case '-', '0', '1', '2', '3', '4',
         '5', '6', '7', '8', '9':
        goto VALUE_NUMBER
    case '"':
        goto VALUE_STRING
    case 'n':
        goto VALUE_NULL
    case 'f':
        goto VALUE_FALSE
    case 't':
        goto VALUE_TRUE
    }
    if s[0] < 0x20 {
        return s, getError(ErrorCodeIllegalControlChar, src, s)
    }
    return s, getError(ErrorCodeUnexpectedToken, src, s)
```

```
VALUE_OBJECT:
    s = s[1:]
    if len(s) < 1 {
        return s,
getError(ErrorCodeUnexpectedEOF, src, s)
    }
    if lutSX[s[0]] == 1 {
        s = skipSpace(s)
        if len(s) < 1 {
            return s,
getError(ErrorCodeUnexpectedEOF, src, s)
        }
    }
    if s[0] == '}' {
        s = s[1:]
        goto AFTER_VALUE
    }
    stPush(stackNodeTypeObject)
    goto OBJ_KEY
```

Skipper Spacy

Skipper is a small but very important unique character.
He's the only function in the state machine.
He skips over the irrelevant bits of JSON like **whitespaces**, **tabs** and **line-breaks, carriage-returns**.

```go
func SpIndex(s []byte) []byte {
    for i := 0; i < len(s); i++ {
        switch s[i] {
        case ' ', '\r', '\n', '\t':
            continue
        }
        // Not a space character
        return s[i:]
    }
    return nil
}
```

```
no space:       1.7 ns/op
1 space:        2.8 ns/op
40 spaces:     15.5 ns/op
130 spaces:    49.9 ns/op
```

```go
// SpUnrol8LUT played by Skipper Spacy
func SpUnrol8LUT(s []byte) []byte {
    for ; len(s) > 7; s = s[8:] {
        if lookupTable[s[0]] != 1  { return s }
        if lookupTable[s[1]] != 1  { return s[1:] }
        // ...this goes on for 2-6...
        if lookupTable[s[7]] != 1 { return s[7:] }
    }
    for ; len(s) > 0; s = s[1:] {
        if lookupTable[s[0]] != 1 { return s }
    }
    return s
}

// lookupTable maps space characters such as whitespace, tab, line-break and
// carriage-return to 1 and all other ASCII characters to 0.
var lookupTable = [256]byte{' ': 1, '\n': 1, '\t': 1, '\r': 1}
```

```
no space:      1.7 ns/op ->  1.7 ns/op
1 space:       2.8 ns/op ->  2.2 ns/op
40 spaces:    15.5 ns/op -> 10.1 ns/op
130 spaces:   49.9 ns/op -> 30.1 ns/op
```

Streeng Parser

She's the actual MVP. Since JSON is mostly strings, her role shouldn't be underestimated. Her performance determines the performance of the movie.

```
func StrNaive(s []byte) ([]byte, error) {
   for len(s) > 0 {
       switch s[0] {
       case '\\': // ...Handle escape sequence...
       case '"': // ...Handle end of string...
       default:
           // Skip or err on control characters
           if s[0] < 0x20 {
               return s, ErrIllegalControlChar
           }
           s = s[1:]
       }
   }
   return s, ErrUnexpectedEOF
}
```

```go
package main

func StrUnrol8LUT(s []byte) ([]byte, error) {
    for {
        for ; len(s) > 7; s = s[8:] {
            if lutStr[s[0]] != 0 {
                goto CHECK_STRING_CHARACTER
            }
            if lutStr[s[1]] != 0 {
                s = s[1:]
                goto CHECK_STRING_CHARACTER
            }
            // ...this goes on for 2-6...
            if lutStr[s[7]] != 0 {
                s = s[7:]
                goto CHECK_STRING_CHARACTER
            }
            continue
        }

    CHECK_STRING_CHARACTER:
        if len(s) < 1 { return s, ErrUnexpectedEOF }
        switch s[0] {
        case '\\': // Handle escape sequence
        case '"': // Handle end of string
        default: // Handle control characters
        }
    }
}
```

AMD Ryzen 7 5700X:

wikipedia_regex:	94,143	ns/op	->	**30,067 ns/op**
tiny_string:	9.9	ns/op	->	**5 ns/op**

Apple M1:

wikipedia_regex:	83,177	ns/op	->	**24,729 ns/op**
tiny_string:	8.4	ns/op	->	**5.6 ns/op**

```
// lutSX maps space characters such as whitespace, tab, line-break and
// carriage-return to 1, remaining control characters to 3,
// valid hex digits to 2, and everything else to 0.
var lutSX = [256]byte{
   0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1,
   8: 1, 11: 1, 12: 1, 14: 1, 15: 1, 16: 1, 17: 1, 18: 1,
   19: 1, 20: 1, 21: 1, 22: 1, 23: 1, 24: 1, 25: 1, 26: 1,
   27: 1, 28: 1, 29: 1, 30: 1, 31: 1,

   ' ': 2, '\n': 2, '\t': 2, '\r': 2,

   '0': 3, '1': 3, '2': 3, '3': 3, '4': 3, '5': 3, '6': 3, '7': 3, '8': 3, '9': 3,
   'a': 3, 'b': 3, 'c': 3, 'd': 3, 'e': 3, 'f': 3,
   'A': 3, 'B': 3, 'C': 3, 'D': 3, 'E': 3, 'F': 3,
}

// lutStr maps 0 to all bytes that don't require checking during string traversal.
// 1 is mapped to control, quotation mark (") and reverse solidus ("\").
var lutStr = [256]byte{
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   '"': 1, '\\': 1,
}

// lutEscape maps escapable characters to 1,
// all other ASCII characters are mapped to 0.
var lutEscape = [256]byte{
   '"': 1, '\\': 1, '/': 1, 'b': 1, 'f': 1, 'n': 1, 'r': 1, 't': 1,
}
```

Roadmap?

- **["Correctness", "Stability", "Performance"] > "Features"**
- `io.Reader` support for reading files bigger than memory.
- Tokenization API besides iterator.

github.com/romshark/jscan
github.com/romshark/jscan-benchmark
github.com/romshark/pres-jscan