Roman Scharkov

Backend Engineer

at tutti.ch

smg swiss marketplace group

Real Estate

homegate.ch
ImmoStreet.ch
Acheter-Louer.ch
CASASOFT
Immo Scout24
home•ch
IAZI CIFI
PUBLIMMO

Automotive

CAR FOR YOU
Auto Scout24
Moto Scout24

General Marketplaces

tutti.ch
anibis.ch
Ricardo

Finance & Insurance

Finance Scout24

JSCAN v3

JSON validating • JSON parsing • JSON decoding

HISTORY

 (5y ago) Aug 5th 2019: github.com/romshark/llparser

 (5y ago) some day 2019: started to love JSON

 (3y ago) Jan 8th 2022: **jscan first commit**

 (1y ago) Mar 28th 2023: **jscan v1.2 (RFC8259 Compliance)**

 (9m ago) Jun 12th 2023: **jscan v2 released (perf & feat)**

 (5m ago) Oct 3rd 2023: Zürich Gophers Meetup


I'M FINISHED

HAND-ROLLING JSON PARSERS IS A LIABILITY


1
Untapped Potential
I ONLY HAVE
UNTAPPED POTENTIAL


IT USES CODE GENERATION
NO!

REQUIREMENTS

 No code generation.

 Easy to use **Unmarshal** and **Decoder.**

 **Backward-compatibility** with **encoding/json**, ideally a drop-in replacement.

 Extensibility to the feature set of **json v2** (github.com/go-json-experiment/json)

 Best in class **performance** through memory trade-off


JUST
DO IT!
DO IT. JUST DO IT.
Nothing is impossible.
Don't let your dreams be dreams.

```
`{
    "name":"Test name",
    "number": 100553,
    "tags": ["sports", "portable"]
}`
```

```
type Struct3 struct {
    Name   string   `json:"name"`
    Number int      `json:"number"`
    Tags   []string `json:"tags"`
}
```

~**5x** faster than **encoding/json**
28% faster than fastest code generator **easyjson**
23% faster than fastest unmarshaler **goccy**
least memory allocations of all.

```
BenchmarkDecodeStruct3
hand/jscan              348 ns/op      48 B/op      3 allocs/op
unmr/jscan              403 ns/op      32 B/op      3 allocs/op
unmr/goccy              526 ns/op     192 B/op      5 allocs/op
genr/easyjson           564 ns/op      96 B/op      4 allocs/op
unmr/jsoniter           640 ns/op     128 B/op      6 allocs/op
hand/gjson              949 ns/op     352 B/op      4 allocs/op
unmr/jsonv2             962 ns/op      96 B/op      3 allocs/op
unmr/segmentio          967 ns/op     240 B/op      5 allocs/op
genr/ffjson            1175 ns/op     456 B/op     11 allocs/op
hand/fastjson          1522 ns/op    1600 B/op     14 allocs/op
unmr/encoding_json     2169 ns/op     352 B/op     11 allocs/op
```

~10x faster than **encoding/json**
44% faster than fastest code generator **easyjson**
55% faster than fastest unmarshaler **goccy.**

```
BenchmarkDecodeArrayString
hand/jscan                9107 ns/op      5720 B/op     25 allocs/op
unmr/jscan               16261 ns/op     22384 B/op    219 allocs/op
genr/easyjson            29354 ns/op     25944 B/op    201 allocs/op
unmr/goccy               36849 ns/op     24810 B/op    197 allocs/op
genr/ffjson              42601 ns/op     26821 B/op    210 allocs/op
hand/fastjson            50332 ns/op     91480 B/op    214 allocs/op
unmr/jsoniter            56687 ns/op     32259 B/op    321 allocs/op
unmr/jsonv2              74227 ns/op     21099 B/op    146 allocs/op
hand/gjson               76219 ns/op     75384 B/op     59 allocs/op
unmr/encoding_json      177211 ns/op     28888 B/op    230 allocs/op
unmr/segmentio          203192 ns/op     41472 B/op    337 allocs/op
```

not always fair to compare
like with **struct { X,Y,Z float64 }**

```
BenchmarkDecodeStructVector3D on Apple M1
genr/easyjson          107.7 ns/op      0 B/op      0 allocs/op
hand/jscan             132.4 ns/op      0 B/op      0 allocs/op
unmr/goccy             161.9 ns/op     72 B/op      2 allocs/op
unmr/segmentio         211.5 ns/op     24 B/op      1 allocs/op
hand/gjson             225.2 ns/op     48 B/op      1 allocs/op
unmr/jscan             250.2 ns/op     24 B/op      1 allocs/op
unmr/jsoniter          309.8 ns/op     56 B/op      3 allocs/op
unmr/jsonv2            311.6 ns/op     24 B/op      1 allocs/op
hand/fastjson          388.5 ns/op    776 B/op      7 allocs/op
genr/ffjson            413.8 ns/op    352 B/op      6 allocs/op
unmr/encoding_json     634.9 ns/op    240 B/op      5 allocs/op
```


Quiiz
Time!

```go
j := `[1,2,3,4,5]`
var v [3]int
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [1,2,3]
B) [3,4,5]
C) Panic

```go
j := `[1,2,3,4,5]`
var v [3]int
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [1,2,3] ✅
B) [3,4,5]
C) Panic

For arrays, encoding/json **takes first N** values and **ignores excess** values.

```go
j := `[1,null,3]`
v := [3]int{7, 8, 9} // Pre-initialized!
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [0,0,0]
B) [1,0,3]
C) [1,8,3]
D) Panic

```go
j := `[1,null,3]`
v := [3]int{7, 8, 9} // Pre-initialized!
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [0,0,0]
B) [1,0,3]
C) [1,8,3] ✅
D) Panic

null initializes **new** values to **zero value**, but **existing** values remain **untouched**.

```go
j := `[1,2,3,"42"]`
var v [3]int
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [1,2,3]
B) [0,0,0]
C) [42,2,3]
D) Panic

```go
j := `[1,2,3,"42"]`
var v [3]int
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) [1,2,3] ✅
B) [0,0,0]
C) [42,2,3]
D) Panic

encoding/json **doesn't care** about the **excess value types.**

```go
j := `{ "foo":1, "FOO":2 }`
var v struct { Foo int `json:"foo"` }
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {1}
B) {2}
C) Panic

```go
j := `{ "foo":1, "FOO":2 }`
var v struct { Foo int `json:"foo"` }
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {1}
B) {2} ✅
C) Panic

encoding/json **prefers exact** matches over **case-insensitive** matches and **overwrites** on **collision.**

```go
type User struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
}
j := `{"id": 500}`
v := User{ID: 404, Name: "Bob"}
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {500 Bob}
B) {500 }
C) Panic

```go
type User struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
}
j := `{"id": 500}`
v := User{ID: 404, Name: "Bob"}
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {500 Bob} ✅
B) {500 }
C) Panic

encoding/json **doesn't overwrite structs**, it mutates the struct instead.

```go
//      I     D
j := `{"\u0049\u0044":42}`
var v struct{ ID int }
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {42}
B) {}
C) Panic

```go
//       I     D
j := `{"\u0049\u0044":42}`
var v struct{ ID int }
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) {42} ✅
B) {}
C) Panic

encoding/json
supports escaped
field names

```go
type Stats struct {
    ID    string `json:"id"`
    Score int    `json:"score"`
}
j := `{"Bob": {"score": 4}, "Alice": {"score": 5}}`
v := map[string]Stats{"Bob": {ID: "b"}}
if err := json.Unmarshal([]byte(j), &v); err != nil {
    panic(err)
}
fmt.Println(v)
```

A) map[Alice:{ 5} Bob:{"b" 4}]
B) Panic

```go
type Stats struct {
    ID    string `json:"id"`
    Score int    `json:"score"`
}
j := `{"Bob": {"score": 4}, "Alice": {"score": 5}}`
v := map[string]Stats{"Bob": {ID: "b"}}
if err := json.Unmarshal([]byte(j), &v); err ≠ nil {
    panic(err)
}
fmt.Println(v)
```

A) map[Alice:{ 5} Bob:{"b" 4}]
B) Panic
C) map[Alice:{ 5} Bob: { 4}] ✅

encoding/json always **overwrites map keys**

Not always fair to compare performance because **easyjson** and many other are **not backward compatible** with encoding/json

- no case-insensitive matching
- no unescaping
- no support for non-struct types
- etc.

Best to compare with **goccy** and **jsoniter**.

Reusing memory is key

```go
// T is the type we want unmarshal to.
type T struct { Foo string; Bar int }

tokenizer := jscan.NewTokenizer[string](
    16,   // Stack size, maximum document depth.
    1024, // Preallocated tokens buffer size.
)

// Preallocate decoder parsing string into `T`.
decoderT, err := jscandec.NewDecoder[string, T](
    tokenizer, jscandec.DefaultInitOptions,
)
if err ≠ nil {
    panic(err)
}

// Now we can reuse decoderT many times.
var v T
if _, err := decoderT.Decode(in, &v, jscandec.DefaultOptions); err ≠ nil {
    panic(err)
}
```

Init-Time
(ones per type)

create decoding table
using reflect

Runtime
(per input)

Tokenize input

init decoding table

decode to variable

```
type T struct { IDs []int; Name string; Map map[string]float64 }
```

type ExpectType	**offset,size** uintptr	**dest** unsafe.Pointer	**parent** uint32	**fields** []fieldFrame
Struct	0,48	nil	4294967295	[{1 "IDs"}, {3 "Name"}, {4 "Map"}]
Slice	0,24	nil	0	nil
Int	0,8	nil	1	nil
String	24,16	nil	0	nil
Map	40,8	nil	0	nil
String	16	nil	4	nil
Float64	8	nil	4	nil

*simplified representation

Let's parse "[1,2]" into &v of type []int

stackIndex=0; tokenIndex=0

type ExpectType	**offset,size** uintptr	**dest** unsafe.Pointer	**parent** uint32
Slice	0,16	**addr(v)** 👉	4294967295
Int	0,8	nil	0

Set destination pointer and transition
to next stack frame

Let's parse "[1,2]" into &v of type []int

stackIndex=1; tokenIndex=1

type ExpectType	**offset,size** uintptr	**dest** unsafe.Pointer	**parent** uint32
Slice	0,16	addr(v)	4294967295
Int	**8** 👈,8	**addr(v)+offset(0)**👈	0

Set destination pointer, write value, update offset and transition to next token

Let's parse "[1,2]" into &v of type []int

stackIndex=1; tokenIndex=2

type ExpectType	**offset,size** uintptr	**dest** unsafe.Pointer	**parent** uint32
Slice	0,16	addr(v)	4294967295
Int	**16** 👈,8	**addr(v)+offset(8)**👈	0

Set destination pointer, write value, update offset and transition to next token

Let's parse "[1,2]" into &v of type []int

stackIndex=0; tokenIndex=3

type ExpectType	**offset,size** uintptr	**dest** unsafe.Pointer	**parent** uint32
Slice	0,16	addr(v)	4294967295
Int	8,8	addr(v)+offset(16)	**0** 👈

Transition back to parent frame, last token at root frame, return no error.

White House urges developers to dump C and C++

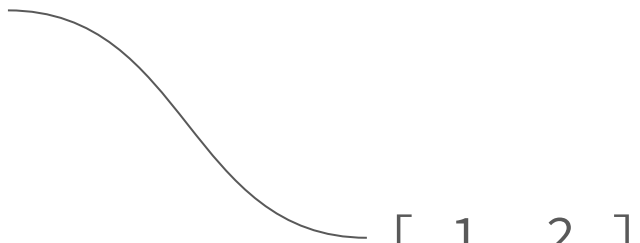
Biden administration calls for developers to embrace memory-safe programing languages and move away from those that cause buffer overflows and other memory access vulnerabilities.

By Grant Gross

InfoWorld | FEB 27, 2024 10:35 AM PST

Rust as an example of a programming language it considers safe. In addition, an NSA cybersecurity information sheet from November 2022 listed C#, Go, Java, Ruby, and Swift, in addition to Rust, as programming languages it considers to be memory-safe.

```
type GoSlice struct {
    Data     unsafe.Pointer
    Length   int
    Capacity int
}
```



Maps are complex!
No way around reflection
…right?

```
1  mapType := reflect.TypeOf(map[string]int)
2  vm := reflect.MakeMap(mapType)
3  key, value := "Meaning of life", 42
4  vm.SetMapIndex(reflect.ValueOf(key), reflect.ValueOf(value))
```

```go

//go:linkname makemap reflect.makemap
func makemap(*typ, int) unsafe.Pointer

//nolint:golint
//go:linkname mapassign_faststr runtime.mapassign_faststr
//go:noescape
func mapassign_faststr(t *typ, m unsafe.Pointer, s string) unsafe.Pointer

//go:linkname mapassign runtime.mapassign
//go:noescape
func mapassign(t *typ, m unsafe.Pointer, k unsafe.Pointer) unsafe.Pointer

// typ represents reflect.rtype for noescape trick
type typ struct{}

type emptyInterface struct { _ *typ; ptr unsafe.Pointer }

func getTyp(t reflect.Type) *typ {
    return (*typ)(((*emptyInterface)(unsafe.Pointer(&t))).ptr)
}

func canUseAssignFaststr(mapType reflect.Type) bool {
    return mapType.Elem().Size() ⩽ 128 && mapType.Key().Kind() == reflect.String
}

```

```
MakeMapInlined        49.8 ns/op    144 B/op    1 allocs/op
MakeMap              121.2 ns/op    400 B/op    3 allocs/op
MakeMapReflect       192.9 ns/op    560 B/op    5 allocs/op
MakeMapFastUnsafe    127.4 ns/op    400 B/op    3 allocs/op
```

```go
// MakeMapInlined
m := make(map[string]int)
m["Meaning of life"] = 42
```

```
m := make(map[string]string, tokens[ti].Elements)
tiEnd := tokens[ti].End

for ti++; ti < tiEnd; ti += 2 {
    tokVal := tokens[ti+1]
    if tokVal.Type ≠ jscan.TokenTypeString {
        if tokVal.Type == jscan.TokenTypeNull {
            key := s[tokens[ti].Index+1 : tokens[ti].End-1]
            keyUnescaped := unescape.Valid[S, string](key)
            m[keyUnescaped] = ""
            continue
        }
        err = ErrorDecode{
            Err:   ErrUnexpectedValue,
            Index: tokVal.Index,
        }
        return true
    }
    key := s[tokens[ti].Index+1 : tokens[ti].End-1]
    keyUnescaped := unescape.Valid[S, string](key)
    value := s[tokVal.Index+1 : tokVal.End-1]
    m[keyUnescaped] = unescape.Valid[S, string](value)
}
ti++
*(*map[string]string)(p) = m
```

Type **map[string]string** is very common, inline it as **ExpectTypeMapStringString**

ROADMAP

 Testing, testing, testing, benchmarking…

 jscan v3 release (Tokenizer, Decoder, Unmarshal)

 json/v2 options

Duplicates, InvalidUTF8, CaseInsensitiveFieldMatch

 Further performance improvements

 io.Reader support?

github.com/romshark/jscan

github.com/romshark/jscan-experimental-decoder