GoとJSON

merpay Architect
goccy

2021/8/20 Go 1.17 Release Party

goccy (ごっしー)

merpay Architect

@goccy54

goccy

Repository	Star
goccy/go-json	1.1k
goccy/go-yaml	518
goccy/go-reflect	283
goccy/go-graphviz	249

Agenda

- encoding/json の設計思想と課題
- encoding/json v2 について
- 3rd party library の現状
- goccy/go-json

encoding/json の設計思想と課題

encoding/jsonの設計思想

- ユーザが誤解なく簡単に使えるAPI
- シンプルな実装でコードサイズも小さく
- unsafe を使わない

https://github.com/go-json-experiment/json#goals-and-objectives から読み取れる

標準ライブラリの立場として、余計な機能を削ぎ落としたり、
素朴で安全な実装を選択することは正しい

一方で痒いところに手が届くような機能や速度の面では改善の余地がある

encoding/jsonの機能的な課題

- エンコード時に map を渡すと常に sort する
  - sort させないオプションなどもないので遅い
- デコード時に JSON Object の duplicated key を許容する
  - duplicated key を見つけてもエラーにせずに上書きする
  - すべての field を見つけても Object の終端までちゃんとパースする必要があるので遅い
- エンコード・デコードをフックできる唯一の手段の **MarshalJSON** / **UnmarshalJSON** の使い勝手が良くない
  - ストリームのI/Fに対応していない
  - context.Context を受け取れない

その他、機能要望はたくさんある

https://github.com/golang/go/issues?q=is%3Aissue+is%3Aopen+encoding%2Fjson+in%3Atitle

encoding/jsonの速度的な課題

- unsafe を使わないので、次のようなことはできない

1. 型のメモリ上のレイアウトを利用した高速なキャスト (e.g. **[]byte <=> string**)
2. linkname directive による runtime や reflect で定義されている private API の利用
3. ポインタ演算 (e.g. **unsafe.Pointer(addr + offset)**)
4. **atomic.LoadPointer / atomic.StorePointer** を利用したキャッシュ

- そのため、次のように遅い代替案を使うことになる

1. コピーの伴うキャスト
2. reflect
3. atomic の代わりに sync.Map を使ったキャッシュ

Tips: reflect はなぜ遅いのか

- APIが個別の型に依存した設計になっていない
  - int も string も bool も同じ reflect.Type / reflect.Value を操作する
- 型によってできる操作が異なるので、各APIの先頭でバリデーションが走る
  - これが遅い
- 必ず reflect.TypeOf や reflect.ValueOf の引数がエスケープされる
  - 引数がヒープにエスケープされる => アロケーションが走る => 遅い

- reflect の世界にいくと、静的型付けの恩恵が消えて、
  ランタイムで通らなければいけないパスが増えてしまうので遅い

encoding/json v2 について

encoding/json v2 ができるかも…？

- github.com/go-json-experiment/json で開発中
- v2の実装や設計思想についていろいろ教えてくれた
- v1との違い（ 一部 ）

1. 基本のAPIは v1 と同じで、オプションで機能拡張する方針
2. エンコードでmapのsortを強制するのは廃止
3. デコードで duplicated key はデフォルトでエラー（ つまり後勝ちではなくなる ）
4. 真のストリーム処理ができるようになる
5. 速度の Priority はそれほど高くないので、既存の 3rd party ライブラリを上回るようなことはなさそう

真のストリーム処理がなぜ必要か

- encoding/json のストリーム処理には問題がある
- エンコード
  - sync.Pool で管理している encodeState の bytes.Buffer に対して書き込んでいき、最後に NewEncoder に与えられた io.Writer に対して一括で書き込む
    - Encode対象が巨大だとメモリにのらない
  - MarshalJSON は []byte を返り値とし、それを encodeState の buffer に書き込む
    - MarshalJSON の中で巨大な buffer を扱うとメモリにのらない
- デコード
  - UnmarshalJSON の引数が []byte のため、JSON Value として valid な単位の buffer を先にすべて読みこむ必要がある
    - UnmarshalJSON に渡す Value が巨大だとメモリにのらない

3rd party library の現状

人気の 3rd party library の比較

- json-iterator/go : Star 10k に届く勢いの一番有名なOSS
  - Pros: encoding/json と似た使用感で使える。実績十分。encoding/json より速い
  - Cons: 他のライブラリに比べてかなり遅い。完全コンパチではないので注意
- mailru/easyjson : (Star 3.3k) コード生成を採用
  - Pros: コード生成するので(結構)速い
  - Cons: コード生成するのでひと手間増える。デコード・interface{}には効果がない
- segmentio/encoding/json : (Star 750) encoding/json と似た使用感で使える
  - Pros: エンコードがかなり速い
  - Cons: デコードがかなり遅い。ストリームデコード用のAPIが足りない点も注意
- bytedance/sonic : (Star 620) JITとSIMDでカリカリにチューニングされている
  - Pros: エンコード、デコードともに最速レベル
  - Cons: まだ Alpha。 asm を多用しているのでプラットフォーム依存。開発スタイルが特殊

その他のライブラリ

- francoispqt/gojay : (Star 2k)
  - Pros: エンコード・デコードともに結構速い
  - Cons: ユーザに最適化用の処理を書かせる設計なので、使うまでに多少ハードルがある
- minio/simdjson-go : (Star 1.1k)
  - Pros: SIMD最適化の勉強になる (simdjson 独自のアプローチが面白い)
  - Cons: APIが利用しづらい。デコーダのみ
- pquerna/ffjson : (Star 2.8k)
- valyala/fastjson : (Star 1.3k)
  - どちらも開発終了の様子
  - 他のライブラリに比べて大分遅い
- wI2L/jettison : (Star 111)
  - エンコードのみ。速さに注力しているが segmentio/encoding/json の方が速い

まとめると

- 安定性・実績重視
  - encoding/json
- 速度重視
  - 多少遅くても実績重視 : json-iterator/go
  - エンコードだけで良い : segmentio/encoding/json
  - デコードだけで良い : francoispqt/gojay
  - とにかく速く : bytedance/sonic
- その他、速度以外で欲しい機能に合わせて選択する

goccy/go-json

goccy/go-json

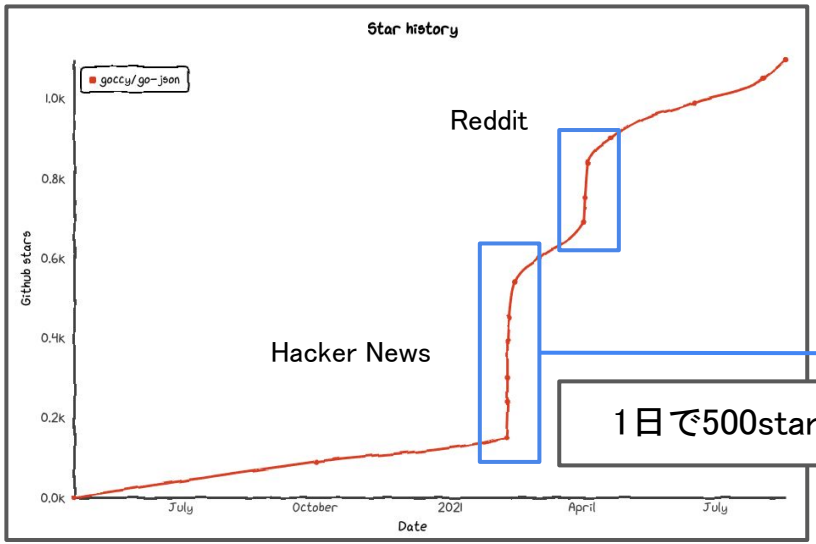
- Star 1.1k / 開発期間 1年ちょっと
- encoding/json とコンパチ
  - 現状、唯一のコンパチライブラリ
- 高速
  - import 文を差し替えるだけで速くなる
  - bytedance/sonic よりは遅いが、他のライブラリよりは速いくらい
- オプションによる拡張機能
  - context.Context を MarshalJSON / UnmarshalJSON に渡せる
  - エンコード時に map の key をソートしないこともできる
  - デコード時の挙動を先勝ちに変更できる
  - エンコード時に色付けできる
  - JSON Path

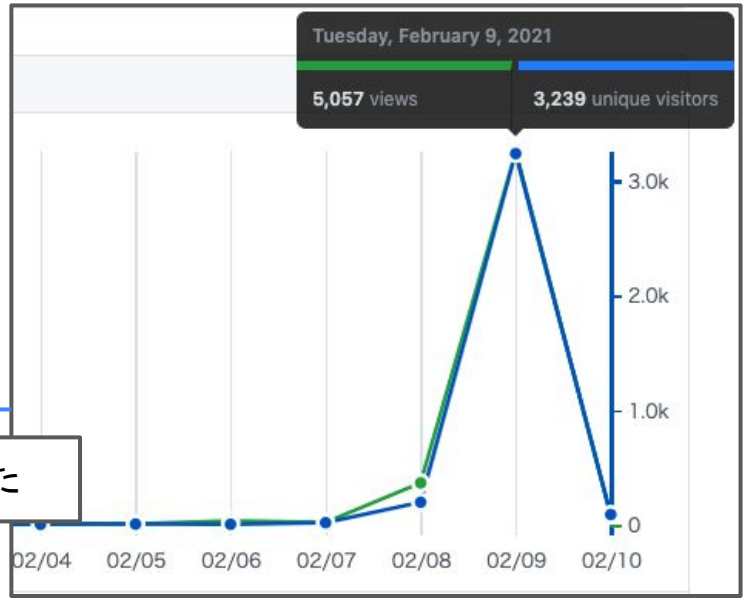
Tips: 開発中に気づいた encoding/json の仕様

- string tag の便利な使い方
  - (u)int64 な ID を JSON に含めるときは **“id,string”** とするだけ
  - 数値を文字列にする場合は quote を足すだけなので高速
- MarshalJSONは遅い
  - []byte で返すため、ライブラリ側で valid な JSONか確かめる必要がある
  - インデントの有無によって返された文字列を整形する必要がある
- tag が conflict したときの仕様
  - https://pkg.go.dev/encoding/json#Marshal の comment に詳しく書かれている
- デコード時にポインタやスライスの値がすでに存在した場合、再利用する

Tips: Starの伸ばし方

- 日本向けの宣伝では限界がある
- README に目を引く情報を書く (Benchmark のグラフや How it works)
- Hacker News、Reddit に投稿するのが良い





1日で500star近く伸びた

最適化テクニック (type id x cache)

type id

同一バイナリにおいて、reflect.Typeのアドレスは常に固定で型によってユニーク になる性質を利用して
アドレスを型のIDとして扱うテクニック

type id を使って型に依存する処理をキャッシュできる

encoding/jsonでは右の サンプルコード のように
reflect.Type と sync.Map を組み合わせている

```go
package main

import (
    "fmt"
    "reflect"
    "sync"
)

var cache sync.Map // map[reflect.Type]func()

func storeProcess(typ reflect.Type, fn func()) {
    cache.Store(typ, fn)
}

func loadProcess(typ reflect.Type) func() {
    p, ok := cache.Load(typ)
    if ok {
        return p.(func())
    }
    return nil
}

func main() {
    v := "Go"
    typ := reflect.TypeOf(v)
    storeProcess(typ, func() { fmt.Println(v) })
    if proc := loadProcess(typ); proc != nil {
        proc()
    }
}
```

最適化テクニック（ type id x　slice　cache ）

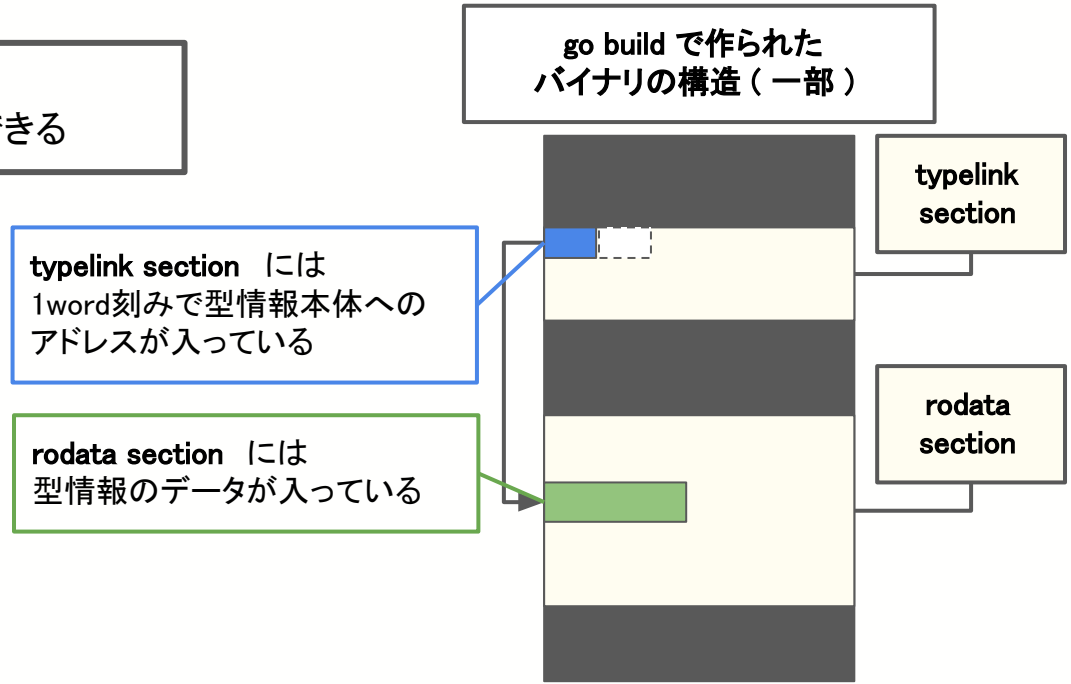
- バイナリに存在するすべての型情報のアドレスがわかれば、データ構造をmapからsliceにできて高速にアクセスできる

reflect.typelinks　を使うことで、
typelink section にある型情報を全て探索できる

↓

すべてのアドレスがわかれば
その範囲からスライスを作れる

[実際のコードはこちら]



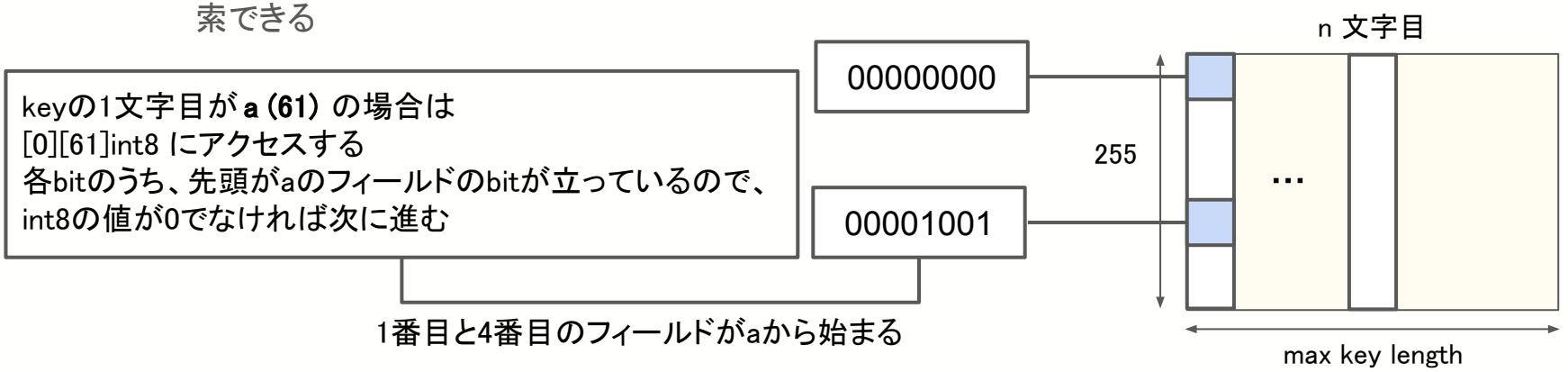
最適化テクニック (構造体フィールドのデコード 1/2)

- Object key の値に対応するデコード処理を呼び出す際、通常は map を用いるが、これが遅い (デコード処理の中で無視できない割合を占める)
- 各 3rd party library はこれを次のような方法で最適化している

1. **json-iterator/go** ：構造体のフィールド数が10以下に関してのみ、用意した switch-case で処理するが、case の値に FNV アルゴリズムで作ったハッシュ値を利用しているので衝突の危険性がある
2. **francoispqt/gojay** ：key による dispatch 処理 (switch-case)自体をユーザに書かせる

最適化テクニック (構造体フィールドのデコード 2/2)

- goccy/go-json は独自の Bitmap を利用した方法で解決
  - 構造体のフィールド数が8以下の場合は int8 で、16以下の場合は int16 でフィールドの状態を表現できる (bit とフィールドがそれぞれ対応するイメージ)
  - 1byteの文字が取りうる値の範囲は 0 - 255 までなので、構造体のフィールドの中で一番文字数が長いものの長さを **maxKeyLength** とすると、フィールド数が8以下の場合は **[maxKeyLength][256]int8** の領域を確保すれば bit 演算で Object key に対応するフィールドを検索できる



その他の最適化について

- 今回載せていない最適化については以前発表した資料に掲載
  - 最速のJSONライブラリを求めて
  - Goで高速JSONライブラリを作るためにしたこと
- その他の細かな最適化に関する質問や今後の高速化アイディアについては

  gophers.slack.com の #go-json チャンネルでしましょう (日本語 or 英語)

おわりに

- encoding/json v2 に期待
  - ただし依然として 3rd party library の需要はありそう
- 3rd party library を使うなら
  - goccy/go-json がオススメ！
  - 実績以外の面で他のライブラリを採用するとき、
    理由を Issue で教えてくれると(改善に役立つので) 嬉しい
- **明日は ISUCON11 予選当日ですね！**
  - JSON まわりの処理が遅かったらぜひ goccy/go-json を！
  - (注意) 標準ライブラリは **go mod edit -replace** できないので、framework で使われている encoding/json を置き換える必要がある場合は、直接 import の置き換えをする必要あり
    (**go mod vendor** と組み合わせると良い)