

Golang Framework

是怎麼做出這些酷功能？

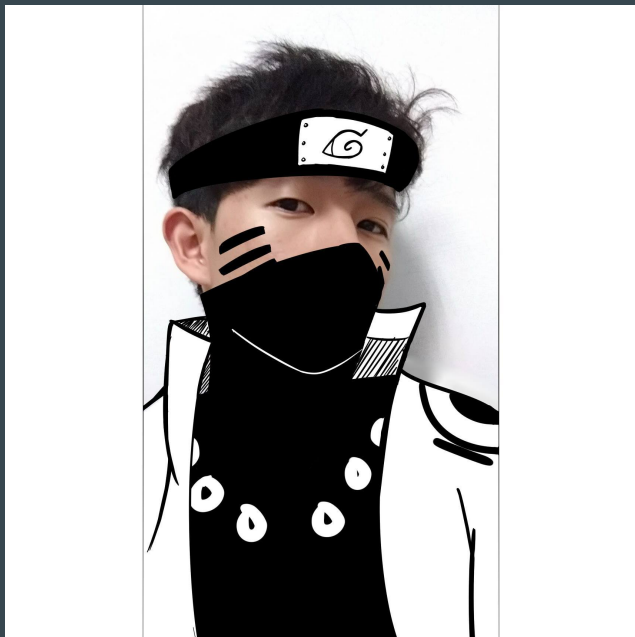
2023/12

大綱

- uber/zap 中的 json encoder
- string 和 []byte 轉換
- builder pattern, method chain

關於我 - WeiTheShinobi

- 繳了 1.5 年的勞保
- 喜歡電腦
- 喜歡電玩
- 喜歡漫畫
- 喜歡鳥類
- 特別喜歡鴨子 🦆 🦆 🦆
- Github : [WeiTheShinobi](#)



關於我 - WeiTheShinobi

我要講四十分鐘



上台前

大..大家..好...我...



上台時

uber/zap 中的 json encoder

reflection-free, zero-allocation JSON encoder

uber/zap

- log 框架
- uber 出品, 必屬精品
- 極高的測試覆蓋率
- reflection-free, zero-allocation JSON encoder
- 快


About

Blazing fast, structured, leveled logging in Go.


pkg.go.dev/go.uber.org/zap


golang zap logging

structured-logging


 Readme

 MIT license

 Code of conduct

 Activity

 20.1k stars

 246 watching

 1.4k forks

Report repository

uber/zap - quick start

```
func main() {  
    z, _ := zap.NewProduction()  
    z.Info(  
        "info",  
        zap.String("hello", "world"),  
    )  
    //  
    {"level":"info","ts":1000000000.453822,"caller":"Lab/main.go:122","msg":"info","hello":"world"}  
}
```

Performance

For applications that log in the hot path, reflection-based serialization and string formatting are prohibitively expensive — they're CPU-intensive and make many small allocations. Put differently, using `encoding/json` and `fmt.Fprintf` to log tons of `interface{}` s makes your application slow.

Zap takes a different approach. It includes a reflection-free, zero-allocation JSON encoder, and the base `Logger` strives to avoid serialization overhead and allocations wherever possible. By building the high-level `SugaredLogger` on that foundation, zap lets users *choose* when they need to count every allocation and when they'd prefer a more familiar, loosely typed API.

As measured by its own [benchmarking suite](#), not only is zap more performant than comparable structured logging packages — it's also faster than the standard library. Like all benchmarks, take these with a grain of salt.¹

Log a message and 10 fields:

Package	Time	Time % to zap	Objects Allocated
⚡ zap	656 ns/op	+0%	5 allocs/op
⚡ zap (sugared)	935 ns/op	+43%	10 allocs/op
zerolog	380 ns/op	-42%	1 allocs/op
go-kit	2249 ns/op	+243%	57 allocs/op
slog (LogAttrs)	2479 ns/op	+278%	40 allocs/op

It includes a reflection-free, zero-allocation JSON encoder,

It includes a reflection-free, zero-allocation JSON encoder,



reflection-free

json encoder - 標準函式庫

- 使用 reflect 取得物件資料
- tag 能做一些設定
- tag 非必要
- 簡單易用, 幾乎不需要額外的工作

```
type Duck struct {  
    Age int `json:"score"`  
}  
  
func main() {  
    d, _ := json.Marshal(Duck{Age: 3})  
    fmt.Println(string(d)) // {"score":3}  
}
```

json encoder - uber/zap: reflection-free

- 標準函式庫使用 reflect
- zap 提供 zap.Field 物件, 使用者選好型別、填入 key, value
- 既然你都手動把型別寫好了, 自然就不需要 reflect 了

```
z.Info(  
    "info",  
    zap.String("string", "s"),  
    zap.Int("int", 0),  
)
```

```
// String constructs a field with the given key and value.  
func String(key string, val string) Field {  
    return Field{Key: key, Type: zapcore.StringType, String: val}  
}  
  
// Int constructs a field with the given key and value.  
func Int(key string, val int) Field {  
    return Int64(key, int64(val))  
}  
  
// Int64 constructs a field with the given key and value.  
func Int64(key string, val int64) Field {  
    return Field{Key: key, Type: zapcore.Int64Type, Integer: val}  
}
```

json encoder - uber/zap: reflection-free

- 知道型別就可以繼續拼裝
- encoder 會拼接上 json 會用到的符號
- buf 是一個包裝過的 []byte, 你可以想成 []byte
- 下方為示意圖

```
func (enc *jsonEncoder) AppendArray(arr ArrayMarshaler) error {
    enc.addElementSeparator()
    enc.buf.AppendByte('[')
    err := arr.MarshalLogArray(enc)
    enc.buf.AppendByte(']')
    return err
}
```

```
func (enc *jsonEncoder) AppendString(val string) {
    enc.addElementSeparator()
    enc.buf.AppendByte('"')
    enc.safeAddString(val)
    enc.buf.AppendByte('"')
}
```

```
var b []byte
b = append(b, '{')
b = append(b, "\"key\" : 33...")
b = append(b, '}')
fmt.Println(string(b)) // {"key" : 33}
```

json encoder - uber/zap: reflection-free

```
func (enc *jsonEncoder) EncodeEntry(ent Entry, fields []Field) (*buffer.Buffer, error) {
    final := enc.clone()
    final.buf.AppendByte('{')

    // ...

    final.buf.AppendByte('}')
    final.buf.AppendString(final.LineEnding)

    ret := final.buf
    putJSONEncoder(final)
    return ret, nil
}

func (enc *jsonEncoder) addKey(key string) {
    enc.addElementSeparator()
    enc.buf.AppendByte('"')
    enc.safeAddString(key)
    enc.buf.AppendByte('"')
    enc.buf.AppendByte(':')
    if enc.spaced {
        enc.buf.AppendByte(' ')
    }
}
```

物件呢？zap 總不會通靈吧？

json encoder - uber/zap: reflection-free

- 如果要印出物件, 只要實作 `zapcore.ObjectMarshaler` 即可
- 把物件的樣子手動填進去
- 有點麻煩
- 補充: `zap` 有提供語法糖模式

```
type ObjectMarshaler interface {  
    MarshalLogObject(ObjectEncoder) error  
}
```

```
type Duck struct {  
    Age int  
}  
  
func (d Duck) MarshalLogObject(encoder zapcore.ObjectEncoder) error {  
    encoder.AddInt("age", d.Age)  
    return nil  
}
```

```
z.Info(  
    "info",  
    zap.Object("object", Duck{}),  
)
```

zero-allocation

json encoder - uber/zap: zero-allocation

- 還是需要有個空間去記下資料，不是魔法
- 並不是重頭到尾完全零 allocate
- 第一次用還是要 allocate 的
- 之後重複使用達到 zero-allocation

Log a message and 10 fields:

Package	Time	Time % to zap	Objects Allocated
⚡ zap	656 ns/op	+0%	5 allocs/op
⚡ zap (sugared)	935 ns/op	+43%	10 allocs/op
zerolog	380 ns/op	-42%	1 allocs/op
go-kit	2249 ns/op	+243%	57 allocs/op
slog (LogAttrs)	2479 ns/op	+278%	40 allocs/op
slog	2481 ns/op	+278%	42 allocs/op
apex/log	9591 ns/op	+1362%	63 allocs/op
log15	11393 ns/op	+1637%	75 allocs/op
logrus	11654 ns/op	+1677%	79 allocs/op

那又是怎麼重複使用的？

json encoder - uber/zap: zero-allocation

- 繼續追蹤 jsonEncoder
- jsonEncoder 有一個 buf 用來記錄 encode 後的資料
- 發現 jsonEncoder 和 buf 都會從 pool 中取得
- 前面提到 buf 即是 []byte 包裝
- 這 pool 是什麼？

```
var _jsonPool = pool.New(func() *jsonEncoder {  
    return &jsonEncoder{}  
})
```

```
func newJSONEncoder(cfg EncoderConfig, spaced bool) *jsonEncoder {  
    // ...  
  
    return &jsonEncoder{  
        EncoderConfig: &cfg,  
        buf:            bufferpool.Get(),  
        spaced:         spaced,  
    }  
}
```

json encoder - sync.Pool

- document 說明了用途
- 使用了標準函式庫的 sync.Pool
- zap 用泛型包了一層
- p.pool.Get() 回傳 any, 轉型成 T
- 為了強型別, 讓機器檢查

```
// A Pool is a generic wrapper around [sync.Pool] to provide strongly-typed
// object pooling.
type Pool[T any] struct {
    pool sync.Pool
}
```

```
// New returns a new [Pool] for T, and will use fn to construct new Ts when
// the pool is empty.
func New[T any](fn func() T) *Pool[T] {
    return &Pool[T]{
        pool: sync.Pool{
            New: func() any {
                return fn()
            },
        },
    }
}

// Get gets a T from the pool, or creates a new one if the pool is empty.
func (p *Pool[T]) Get() T {
    return p.pool.Get().(T)
}

// Put returns x into the pool.
func (p *Pool[T]) Put(x T) {
    p.pool.Put(x)
}
```

json encoder - uber/zap: zero-allocation

- 標準函式庫的 `sync.Pool`
 - `Pool` 的目的是 `cache` 已分配但未使用的 `item`, 以減輕垃圾回收的壓力
 - `Pool` 中的 `item` 會被垃圾回收
 - 沒有使用泛型, 手動轉型
-
- `Get()`: 拿物件, 若池子無物件則 `New()`
 - `Put()`: 放物件回去

```
pool := sync.Pool{
    New: func() any {
        return &Duck{}
    },
}
d := pool.Get().(*Duck)
pool.Put(d)
```

json encoder - gin

- 擷取自 gin-gonic/gin
- 伺服器框架
- 從 pool 中取得 Context
- handle request
- 用完了再放回去

```
func (engine *Engine) ServeHTTP(w http.ResponseWriter, req *http.Request) {  
    c := engine.pool.Get().(*Context)  
    c.writemem.reset(w)  
    c.Request = req  
    c.reset()  
  
    engine.handleHTTPRequest(c)  
  
    engine.pool.Put(c)  
}
```


小結

- reflection-free
 - 手動設定就不用自動 (reflect)
- zero-allocation
 - 不是真的完全沒有 allocation
 - 使用了 sync.Pool, 減少 allocation
 - pool.Get() 回傳 any, pool.Put() 傳入 any, 配合泛型可以更安全

string 和 []byte 轉換

unsafe - string 是 immutable

- 在 go 中, string 是 immutable
- 修改會做一個新的給你
- string 沒辦法像 []byte 那樣, 修改數值
- 舉例來說: 右圖中 s 會新分配一個 string

```
func add(s string) string {  
    s += "a"  
    return s  
}
```

unsafe - string 和 []byte 轉換

- 某些函式需要使用 []byte, 還蠻重要的
- 可以像下圖那樣轉換, 寫法簡單
- 但會 allocate, 因為 string 是 immutable 的

```
// io.Writer
type Writer interface {
    Write(p []byte) (n int, err error)
}

// json.Unmarshal
func Unmarshal(data []byte, v any) error {
    // ...
}
```

```
str := "example"
```

```
b := []byte(str)
```

```
s := string(b)
```

明明都是一樣的東西
怎麼只是轉換還要複製一次
有沒有什麼辦法？



unsafe - gin-gonic/gin

- 引用自 gin/internal/bytesconv/bytesconv_1.20.go
- 使用標準函式庫 unsafe
- 轉換且不會有 allocation

```
// StringToBytes converts string to byte slice without a memory allocation.  
// For more details, see https://github.com/golang/go/issues/53003#issuecomment-1140276077.  
func StringToBytes(s string) []byte {  
    return unsafe.Slice(unsafe.StringData(s), len(s))  
}  
  
// BytesToString converts byte slice to string without a memory allocation.  
// For more details, see https://github.com/golang/go/issues/53003#issuecomment-1140276077.  
func BytesToString(b []byte) string {  
    return unsafe.String(unsafe.SliceData(b), len(b))  
}
```

unsafe - Benchmark

- 引用自 `gin/internal/bytesconv/bytesconv_test.go`

```
func rawBytesToStr(b []byte) string {  
    return string(b)  
}  
  
func BytesToString(b []byte) string {  
    return unsafe.String(unsafe.SliceData(b), len(b))  
}
```

BenchmarkBytesConvStrToBytesRaw				
BenchmarkBytesConvStrToBytesRaw-10	57382176	20.66 ns/op	96 B/op	1 allocs/op
BenchmarkBytesConvStrToBytes				
BenchmarkBytesConvStrToBytes-10	1000000000	0.6259 ns/op	0 B/op	0 allocs/op

等等！既然 string 可以轉換成 []byte





也許我能修改 string ?



unsafe - 修改 string 🤔

- 我先把 string 轉成 []byte, 再修改 []byte, 這樣 string 就會被我修改了吧? 嘿 嘿嘿嘿
- 預計最後 s 會變成 “axample”

```
func StringToBytes(s string) []byte {  
    return unsafe.Slice(unsafe.StringData(s), len(s))  
}  
  
func main() {  
    s := "example"  
    b := StringToBytes(s)  
    fmt.Println(string(b))  
    b[0] = 'a'  
    fmt.Println(string(b))  
}
```





```
$ go run main.go  
example  
unexpected fault address // ...
```



unsafe - 修改 string 🤔

- 不能這樣玩

```
func StringToBytes(s string) []byte {  
    return unsafe.Slice(unsafe.StringData(s), len(s))  
}  
  
func main() {  
    s := "example"  
    b := StringToBytes(s)  
    fmt.Println(string(b)) // example  
    b[0] = 'a'             // 程式結束 unexpected fault address ...  
    fmt.Println(string(b))  
}
```

unsafe - 修改 string 🤔

- 右圖取自 package unsafe
- 輸入空字串可能得到 nil
- 用 `unsafe.StringData()` 將 string 轉換成 `[]byte`, 不能修改 `[]byte`
- 剛剛做的就是修改了 `[]byte`

```
// StringData returns a pointer to the underlying bytes of str.  
// For an empty string the return value is unspecified, and may be nil.  
//  
// Since Go strings are immutable, the bytes returned by StringData  
// must not be modified.  
func StringData(str string) *byte
```

小結 奇犽

- unsafe 轉換 string, []byte, 減少 allocation
- 依然不能修改 string
- 使用 unsafe 讓你有更多機會把程式搞壞(或接手的人)
- 難怪叫 unsafe

Builder Pattern, Method Chain

為什麼要寫測試？

人類是有極限的

從短暫的程式生涯中我學到一件事
越是攻於心計
越容易栽在自己意想不到的地方

讓機器檢查

測試 API - 開始寫測試

- 設置、執行、檢查
- 使用官方的東西
- 一大團還蠻醜的
- 要記下很多東西, 再拼裝起來, 使用者體驗不太好

```
body, _ := json.Marshal(Body{})
req, _ := http.NewRequest(
    "GET",
    "/quack",
    bytes.NewReader(body),
)
resp := httptest.NewRecorder()

r.ServeHTTP(resp, req)

assert.Equal(t, "something", resp.Body.String())
// ...
```

應該有更好的方法



測試 API - Rust 中的測試

- 從一個 client 做所有事

```
let response = request::Client::new()  
    .get("/quack")  
    .send()  
    .await  
    .expect("Failed to execute request.");  
  
assert!(response.status().is_success());
```


也許我可以弄一個類似的框架

上 github 看看

📄 websocket_message_test.go	Cleanup tests for Raw() and getters	2 months ago
📄 websocket_test.go	Refactor websocket printer test	9 months ago

☰ README.md



httpexpect

httpexpect

go.dev [reference](#)

build [passing](#)

coverage [95%](#)

tag [v2.16.0](#)

discord [3 online](#)

Concise, declarative, and easy to use end-to-end HTTP and REST API testing for Go (golang).

Basically, httpexpect is a set of chainable *builders* for HTTP requests and *assertions* for HTTP responses and payload, on top of net/http and several utility packages.

Workflow:

- Incrementally build HTTP requests.
- Inspect HTTP responses.
- Inspect response payload recursively.

Features

Request builder

- URL path construction, with simple string interpolation provided by [go-interpol](#) package.
- URL query parameters (encoding using [go-querystring](#) package).
- Headers, cookies, payload: JSON, urlencoded or multipart forms (encoding using [form](#) package), plain text.
- Custom reusable [request builders](#) and [request transformers](#).

Response assertions

好吧果然有人做過了

測試 API - 使用 gavv/httpexpect 後

- 看起來蠻讚的
- Default() 得到物件
- Expect() 前:設定
- Expect():執行
- Expect() 後:assert
- IDE 會提示, 體驗很好(下圖)

```
httpexpect.
```

```
Default(t, server.URL).
```

```
GET(path string, pathargs .  
String(value string) → *Exp  
POST(path string, pathargs  
PATCH(path string, pathargs  
DELETE(path string, pathargs
```

```
httpexpect.
```

```
Default(t, server.URL).
```

```
GET("/quack").
```

```
WithJSON(body).
```

```
Expect().
```

```
Status(http.StatusOK).
```

```
Body().
```

```
IsEqual("something")
```

測試 API - 使用 gavv/httpexpect 後

- WithJSON(), 修改了物件的狀態
- 函式最後回傳了自己

```
func (r *Request) WithJSON(object interface{}) *Request {  
    // ...  
  
    b, err := json.Marshal(object)  
    if err != nil {  
        // ...  
        return r  
    }  
  
    r.setType(opChain, "WithJSON()",  
              "application/json; charset=utf-8", false)  
    r.setBody(opChain, "WithJSON()",  
              bytes.NewReader(b), len(b), false)  
  
    return r  
}
```

```
httpexpect.  
    Default(t, server.URL).  
    GET("/quack").  
    WithJSON(body).  
    Expect().  
    Status(http.StatusOK).  
    Body().  
    IsEqual("something")
```

Builder Pattern - 簡單示範

- 邏輯大概是這樣:設定、做事
- 怎麼做到類似的效果?
- 讓我們再次用 Duck 做示範

```
type Duck struct {}  
  
func (d *Duck) Prepare() {}  
  
func (d *Duck) Quack() {}
```

```
httpexpect.  
    Default(t, server.URL).  
    GET("/quack").  
    WithJSON(body).  
    Expect().  
    Status(http.StatusOK).  
    Body().  
    IsEqual("something")
```

Builder Pattern - 簡單示範

- Prepare() 改變物件狀態
- Quack() 執行
- 每準備一次多叫一聲 (蹲得越低跳得越高)
- 還不夠, 還可以再稍微改變一下 Prepare() 函式簽名

```
type Duck struct {  
    prepare int  
}  
  
func (d *Duck) Prepare() {  
    d.prepare++  
}  
  
func (d *Duck) Quack() {  
    for i := 0; i < d.prepare; i++ {  
        fmt.Print("quack ")  
    }  
}  
  
func main() {  
    duck := Duck{}  
    duck.Prepare()  
    duck.Prepare()  
    duck.Quack() // quack quack  
}
```


Builder Pattern - 簡單示範

- Prepare() 加上 return *Duck
- 回傳了自己, 就可以再次使用相同的函式
- main() 中改成了 method chain
- 做法簡單


```
type Duck struct {  
    prepare int  
}  
  
func (d *Duck) Prepare() *Duck {  
    d.prepare++  
    return d  
}  
  
func (d *Duck) Quack() {  
    for i := 0; i < d.prepare; i++ {  
        fmt.Print("quack ")  
    }  
}  
  
func main() {  
    (&Duck{}).  
        Prepare().  
        Prepare().  
        Quack() // quack quack  
}
```

小結

- 一個使用者體驗佳的手法
- 漂亮(個人喜好, 勿戰)
- 複製時注意物件內的指標


Fun Fact


- 因為做這個簡報，反而推了兩個 PR

 **aceld/zinx Refactor verifyLogIsolation() for a more concise expression** ✓

#287 by WeiTheShinobi was merged 2 days ago

 1

 **gin-gonic/gin Remove redundant comments** document

#3765 by WeiTheShinobi was merged last month • Approved  v1.10

END