目　录

参数验证

介绍

Validator 是基于 tag（标记）实现结构体和单个字段的值验证库，它包含以下功能：

- 使用验证 tag（标记）或自定义验证器进行跨字段和跨结构体验证。
- 关于 slice、数组和 map，允许验证多维字段的任何或所有级别。
- 能够深入 map 键和值进行验证。
- 通过在验证之前确定接口的基础类型来处理类型接口。
- 处理自定义字段类型（如 sql 驱动程序 Valuer）。
- 别名验证标记，它允许将多个验证映射到单个标记，以便更轻松地定义结构体上的验证。
- 提取自定义的字段名称，例如，可以指定在验证时提取 JSON 名称，并在生成的 FieldError 中使用该名称。
- 可自定义 i18n 错误消息。
- Web 框架 gin 的默认验证器。

安装：

使用 go get：

```
go get github.com/go-playground/validator/v10
```

然后将 Validator 包导入到代码中：

```
import "github.com/go-playground/validator/v10"
```

变量验证

Var 方法使用 tag（标记）验证方式验证单个变量。

```
func (*validator.Validate).Var(field interface{}, tag string) error
```

它接收一个 interface{} 空接口类型的 field 和一个 string 类型的 tag，返回传递的非法值得无效验证错误，否则将 nil 或 ValidationErrors 作为错误。如果错误不是 nil，则需要断言错误去访问错误数组，例如：

```
validationErrors := err.(validator.ValidationErrors)
```

如果是验证数组、slice 和 map，可能会包含多个错误。

示例代码：

```
func main() {
  validate := validator.New()
  // 验证变量
  email := "admin#admin.com"
  email := ""
  err := validate.Var(email, "required,email")
  if err != nil {
    validationErrors := err.(validator.ValidationErrors)
    fmt.Println(validationErrors)
    // output: Key: '' Error:Field validation for '' failed on the 'email' tag
    // output: Key: '' Error:Field validation for '' failed on the 'required' ta
g
    return
  }
}
```

结构体验证

结构体验证结构体公开的字段，并自动验证嵌套结构体，除非另有说明。

```
func (*validator.Validate).Struct(s interface{}) error
```

它接收一个 interface{} 空接口类型的 s，返回传递的非法值得无效验证错误，否则将 nil 或 ValidationErrors 作为错误。如果错误不是 nil，则需要断言错误去访问错误数组，例如：

```
validationErrors := err.(validator.ValidationErrors)
```

实际上，Struct 方法是调用的 StructCtx 方法，因为本文不是源码讲解，所以此处不展开赘述，如有兴趣，可以查看源码。

示例代码：

```
func main() {
  validate = validator.New()
  type User struct {
    ID     int64  `json:"id" validate:"gt=0"`
```

```
    Name   string `json:"name" validate:"required"`
    Gender string `json:"gender" validate:"required,oneof=man woman"`
    Age    uint8  `json:"age" validate:"required,gte=0,lte=130"`
    Email  string `json:"email" validate:"required,email"`
  }
  user := &User{
    ID:     1,
    Name:   "frank",
    Gender: "boy",
    Age:    180,
    Email:  "gopher@88.com",
  }
  err = validate.Struct(user)
  if err != nil {
    validationErrors := err.(validator.ValidationErrors)
    // output: Key: 'User.Age' Error:Field validation for 'Age' failed on the 'l
te' tag
    // fmt.Println(validationErrors)
    fmt.Println(validationErrors.Translate(trans))
    return
  }
}
```

细心的读者可能已经发现，错误输出信息并不友好，错误输出信息中的字段不仅没有使用备用名（首字母小写的字段名），也没有翻译为中文。通过改动代码，使错误输出信息变得友好。

注册一个函数，获取结构体字段的备用名称：

```
validate.RegisterTagNameFunc(func(fld reflect.StructField) string {
    name := strings.SplitN(fld.Tag.Get("json"), ",", 2)[0]
    if name == "-" {
      return "j"
    }
    return name
  })
```

错误信息翻译为中文：

```
zh := zh.New()
uni = ut.New(zh)
trans, _ := uni.GetTranslator("zh")
_ = zh_translations.RegisterDefaultTranslations(validate, trans)
```

标签

通过以上章节的内容，读者应该已经了解到 Validator 是一个基于 tag（标签），实现结构体和单个字段的值验证库。

本章节列举一些比较常用的标签：

标签	描述
eq	等于
gt	大于
gte	大于等于
lt	小于
lte	小于等于
ne	不等于
max	最大值
min	最小值
oneof	其中一个
required	必需的
unique	唯一的
isDefault	默认值
len	长度
email	邮箱格式

转自： Golang语言开发栈

汉字转拼音

汉字转拼音

汉语拼音转换工具 Go 版。

安装

```
    go get -u github.com/mozillazg/go-pinyin
```

安装CLI工具：

```
    go get -u github.com/mozillazg/go-pinyin/cmd/pinyin
    $ pinyin 中国人
    zhōng guó rén
```

用法

```
package main

import (
    "fmt"

    "github.com/mozillazg/go-pinyin"
)

func main() {
    hans := "中国人"

    // 默认
    a := pinyin.NewArgs()
    fmt.Println(pinyin.Pinyin(hans, a))
    // [[zhong] [guo] [ren]]

    // 包含声调
    a.Style = pinyin.Tone
    fmt.Println(pinyin.Pinyin(hans, a))
    // [[zhōng] [guó] [rén]]

    // 声调用数字表示
```

```
    a.Style = pinyin.Tone2
    fmt.Println(pinyin.Pinyin(hans, a))
    // [[zho1ng] [guo2] [re2n]]

    // 开启多音字模式
    a = pinyin.NewArgs()
    a.Heteronym = true
    fmt.Println(pinyin.Pinyin(hans, a))
    // [[zhong zhong] [guo] [ren]]
    a.Style = pinyin.Tone2
    fmt.Println(pinyin.Pinyin(hans, a))
    // [[zho1ng zho4ng] [guo2] [re2n]]

    fmt.Println(pinyin.LazyPinyin(hans, pinyin.NewArgs()))
    // [zhong guo ren]

    fmt.Println(pinyin.Convert(hans, nil))
    // [[zhong] [guo] [ren]]

    fmt.Println(pinyin.LazyConvert(hans, nil))
    // [zhong guo ren]
}
```

CLI框架

因为机缘巧合，因为希望能在VPS中使用百度网盘，了解到了一个开源的项目BaiduPCS-Go，可以用来直接存取访问百度网盘，做的相当不错

而且看ISSUES，作者可能还是个学生妹子，很强的样子。稍微看了下代码，发现了一个很不错的用来写命令行程序CLI的框架，也是在Github上开源的，因为Golang主要是用来写这个的，所以感觉比较有用的样子，学习一下，并且稍微做了个笔记。

这个框架就是

github.com/urfave/cli

稍微整理了下具体使用的方式

1，最初的版本，如何引入等等

```
package main

import (
  "fmt"
  "log"
  "os"
  "github.com/urfave/cli"
)

func main() {
  app := cli.NewApp()
  app.Name = "boom"
  app.Usage = "make an explosive entrance"
  app.Action = func(c *cli.Context) error {
    fmt.Println("boom! I say!")
    return nil
  }

  err := app.Run(os.Args)
  if err != nil {
    log.Fatal(err)
  }
}
```

这里的 app.Action 中间的，就是CLI执行（回车）后的操作。乍看没啥东西，其实这个时候已经封装了 -help -version等等的方法了

执行这个GO程序，控制台输出 boom! I say! 如果执行 main -help 则输出命令行的帮助，类似下面的样子

```
NAME:
   boom - make an explosive entrance

USAGE:
   002 [global options] command [command options] [arguments...]

VERSION:
   0.0.0

COMMANDS:
     help, h  Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

2，慢慢深入，接下来有参数，也就是执行的时候 XXX.EXE Arg这个Arg部分如何处理

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/urfave/cli"
)

func main() {
  app := cli.NewApp()

  app.Action = func(c *cli.Context) error {
    //获取第一个参数
    fmt.Printf("Hello %q", c.Args().Get(0))
    return nil
  }

  err := app.Run(os.Args)
  if err != nil {
    log.Fatal(err)
  }
}
```

这时候输入 topgoer.exe 1 参数1 ， 则控制台返回 hello “1”;

3,关于FLAG

有很多命令行程序有flag 比如linux下的常用的 netstat -lnp 等等

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/urfave/cli"
)

func main() {
  app := cli.NewApp()

  app.Flags = []cli.Flag {
    cli.StringFlag{
      Name: "lang",
      Value: "english",
      Usage: "language for the greeting",
    },
  }

  app.Action = func(c *cli.Context) error {
    name := "Nefertiti"
    if c.NArg() > 0 {
      name = c.Args().Get(0)
    }
    if c.String("lang") == "spanish" {
      fmt.Println("Hola", name)
    } else {
      fmt.Println("Hello", name)
    }
    return nil
  }

  err := app.Run(os.Args)
  if err != nil {
    log.Fatal(err)
  }
}
```

这个时候我们执行 topgoer.exe -lang english 控制台会输出 Hello Nefertiti；执行topgoer -lang spanish，则会输出 Hola Nefertiti。

在程序里，通过 判断 c.String("lang") 来决定程序分叉

当然程序稍微修改一下，通过一个属性也能对参数赋值

```
app.Flags = []cli.Flag {
    cli.StringFlag{
      Name:        "lang",
      Value:       "english",
      Usage:       "language for the greeting",
      Destination: &language,          //取到的FLAG值，赋值到这个变量
    },
  }
```

使用的时候只要用 language 来判断就行了

4.关于commond和subcommand

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/urfave/cli"
)

func main() {
  app := cli.NewApp()

  app.Commands = []cli.Command{
    {
      Name:    "add",
      Aliases: []string{"a"},
      Usage:   "add a task to the list",
      Action:  func(c *cli.Context) error {
        fmt.Println("added task: ", c.Args().First())
        return nil
      },
    },
    {
```

```
      Name:    "complete",
      Aliases: []string{"c"},
      Usage:   "complete a task on the list",
      Action:  func(c *cli.Context) error {
        fmt.Println("completed task: ", c.Args().First())
        return nil
      },
    },
    {
      Name:        "template",
      Aliases:     []string{"t"},
      Usage:       "options for task templates",
      Subcommands: []cli.Command{
        {
          Name:  "add",
          Usage: "add a new template",
          Action: func(c *cli.Context) error {
            fmt.Println("new task template: ", c.Args().First())
            return nil
          },
        },
        {
          Name:  "remove",
          Usage: "remove an existing template",
          Action: func(c *cli.Context) error {
            fmt.Println("removed task template: ", c.Args().First())
            return nil
          },
        },
      },
    },
  }

  err := app.Run(os.Args)
  if err != nil {
    log.Fatal(err)
  }
}
```

上面的例子里面，罗列了好多个commond以及subcommand，通过定义这些，我们能实现类似

可执行程序 命令 子命令的操作，比如 App.go add 123 控制台输出 added task: 123

只要遵循框架，这个时候这些命令（Command）以及FLAG的帮助说明都是自动生成的。比如上面这个程序的help，控制台会输出所有使用方式

类似

```
NAME:
   002 - A new cli application

USAGE:
   002 [global options] command [command options] [arguments...]

VERSION:
   0.0.0

COMMANDS:
     add, a       add a task to the list
     complete, c  complete a task on the list
     template, t  options for task templates
     help, h      Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

最后，稍微扩张一下，类似MYSQL，FTP，TELNET等工具，很多控制台程序，都是进入类似一个自己的运行界面，这样其实CLI本身因为并没有中断，可以保存先前操作的信息。

所以比如FTP，TELNET，Mysql这种需要权限的工具，广泛使用。这时，我们往往只需要用户登陆一次，就可以继续执行上传下载查询通讯等等的后续操作。

废话不多说，直接上例子

```
package main

import (
  "fmt"
  "log"
  "os"
  "strings"
  "bufio"
  "github.com/urfave/cli"
)

func main() {
  app := cli.NewApp()


  app.Action = func(c *cli.Context) {
```

```
        if c.NArg() != 0 {
            fmt.Printf("未找到命令: %s\n运行命令 %s help 获取帮助\n", c.Args().G
et(0), app.Name)
            return
        }

        var prompt  string

        prompt = app.Name + " > "
        L:
      for {
        var input   string
            fmt.Print(prompt)
      //   fmt.Scanln(&input)

        scanner := bufio.NewScanner(os.Stdin)
        scanner.Scan() // use `for scanner.Scan()` to keep reading
        input = scanner.Text()
        //fmt.Println("captured:",input)
          switch input {
            case "close":
                fmt.Println("close.")
                break L
            default:
          }
        //fmt.Print(input)
              cmdArgs := strings.Split(input, " ")
            //fmt.Print(len(cmdArgs))
              if len(cmdArgs) == 0 {
                  continue
              }

               s := []string{app.Name}
               s = append(s,cmdArgs...)

              c.App.Run(s)

      }

  return
  }

  app.Commands = []cli.Command{
    {
      Name:    "add",
      Aliases: []string{"a"},
```

```
      Usage:   "add a task to the list",
      Action:  func(c *cli.Context) error {
        fmt.Println("added task: ", c.Args().First())
        return nil
      },
    },
    {
      Name:    "complete",
      Aliases: []string{"c"},
      Usage:   "complete a task on the list",
      Action:  func(c *cli.Context) error {
        fmt.Println("completed task: ", c.Args().First())
        return nil
      },
    },
    {
      Name:        "template",
      Aliases:     []string{"t"},
      Usage:       "options for task templates",
      Subcommands: []cli.Command{
        {
          Name:  "add",
          Usage: "add a new template",
          Action: func(c *cli.Context) error {
            fmt.Println("new task template: ", c.Args().First())
            return nil
          },
        },
        {
          Name:  "remove",
          Usage: "remove an existing template",
          Action: func(c *cli.Context) error {
            fmt.Println("removed task template: ", c.Args().First())
            return nil
          },
        },
      },
    },
  }

  err := app.Run(os.Args)

  if err != nil {
    log.Fatal(err)
  }
}
```

这里在Action里面，其实加入了一个死循环，一直在听取程序的输入，直接执行（回车）的话，其实进入一个类似MySQL或者FTP类似的命令行界面。等待用户的进一步输入。
然后读取这个输入，通过调用原来的框架的app.Run(os.Args)来处理需求逻辑。

上述所有基本涵盖了命令行的所有可能的形式，以后就可以按照这个框架写出起码帮助感觉很正式的程序了。

转自：http://www.sz-ming.com/2018/06/20/golang%E7%9A%84%E4%B8%80%E4%B8%AAcli%E6%A1%86%E6%9E%B6%E4%BB%8B%E7%BB%8D%EF%BC%8C%E4%B8%AA%E4%BA%BA%E5%AD%A6%E4%B9%A0%E5%A4%87%E5%BF%98/

Google 翻译API

无须翻墙

Demo 翻译 url `https://translate.googleapis.com/translate_a/single?client=gtx&sl=en&tl=zh-cn&dt=t&q=Worldwide observations confirm nearby ‘lensing’ exoplanet`

参数	类型	说明
url	GET	https://translate.googleapis.com/translate_a/single
client	url-query	默认值(不要修改) gtx
sl	url-query	来源语言 en zh-cn 语言代码如下
tl	url-query	目标语言 en zh-cn 语言代码如下
dt	url-query	默认值(不要修改) t
q	url-query	翻译的文本 建议先url-encode

Google 翻译API参数语言代码

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
    "strings"
)

func TranslateEn2Ch(text string) (string, error) {
    url := fmt.Sprintf("https://translate.googleapis.com/translate_a/single?client=gtx&sl=zh-cn&tl=en&dt=t&q=%s", url.QueryEscape(text))
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
    defer resp.Body.Close()
    if err != nil {
        return "", err
```

```
    }
    bs, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
    //返回的json反序列化比较麻烦, 直接字符串拆解
    ss := string(bs)
    ss = strings.ReplaceAll(ss, "[", "")
    ss = strings.ReplaceAll(ss, "]", "")
    ss = strings.ReplaceAll(ss, "null,", "")
    ss = strings.Trim(ss, `"`)
    ps := strings.Split(ss, `","`)
    return ps[0], nil
}
func main() {
    str, err := TranslateEn2Ch("www.topgoer.com是个不错的go语言中文文档")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(str)
}
```

Google 翻译API参数语言代码

语言	ISO-639-1 代码
南非荷兰语	af
阿尔巴尼亚语	sq
阿姆哈拉语	am
阿拉伯语	ar
亚美尼亚语	hy
阿塞拜疆语	az
巴斯克语	eu
白俄罗斯语	be
孟加拉语	bn
波斯尼亚语	bs

语言	ISO-639-1 代码
保加利亚语	bg
加泰罗尼亚语	ca
宿务语	ceb (ISO-639-2)
中文（简体）	zh-CN 或 zh (BCP-47)
中文（繁体）	zh-TW (BCP-47)
科西嘉语	co
克罗地亚语	hr
捷克语	cs
丹麦语	da
荷兰语	nl
英语	en
世界语	eo
爱沙尼亚语	et
芬兰语	fi
法语	fr
弗里斯兰语	fy
加利西亚语	gl
格鲁吉亚语	ka
德语	de
希腊语	el
古吉拉特语	gu
海地克里奥尔语	ht
豪萨语	ha

语言	ISO-639-1 代码
夏威夷语	haw (ISO-639-2)
希伯来语	he 或 iw
印地语	hi
苗语	hmn (ISO-639-2)
匈牙利语	hu
冰岛语	is
伊博语	ig
印度尼西亚语	id
爱尔兰语	ga
意大利语	it
日语	ja
爪哇语	jw
卡纳达语	kn
哈萨克语	kk
高棉文	km
韩语	ko
库尔德语	ku
吉尔吉斯语	ky
老挝语	lo
拉丁文	la
拉脱维亚语	lv
立陶宛语	lt
卢森堡语	lb

语言	ISO-639-1 代码
马其顿语	mk
马尔加什语	mg
马来语	ms
马拉雅拉姆文	ml
马耳他语	mt
毛利语	mi
马拉地语	mr
蒙古文	mn
缅甸语	my
尼泊尔语	ne
挪威语	no
尼杨扎语（齐切瓦语）	ny
普什图语	ps
波斯语	fa
波兰语	pl
葡萄牙语（葡萄牙、巴西）	pt
旁遮普语	pa
罗马尼亚语	ro
俄语	ru
萨摩亚语	sm
苏格兰盖尔语	gd
塞尔维亚语	sr
塞索托语	st

语言	ISO-639-1 代码
修纳语	sn
信德语	sd
僧伽罗语	si
斯洛伐克语	sk
斯洛文尼亚语	sl
索马里语	o
西班牙语	es
巽他语	su
斯瓦希里语	sw
瑞典语	sv
塔加路语（菲律宾语）	tl
塔吉克语	tg
泰米尔语	ta
泰卢固语	te
泰文	th
土耳其语	tr
乌克兰语	uk
乌尔都语	ur
乌兹别克语	uz
越南语	vi
威尔士语	cy
班图语	xh
意第绪语	yi

语言	ISO-639-1 代码
约鲁巴语	yo
祖鲁语	zu

限流器

限流器是后台服务中的非常重要的组件，可以用来限制请求速率，保护服务，以免服务过载。
限流器的实现方法有很多种，例如滑动窗口法、Token Bucket、Leaky Bucket等。

其实golang标准库中就自带了限流算法的实现，即golang.org/x/time/rate。
该限流器是基于Token Bucket(令牌桶)实现的。

简单来说，令牌桶就是想象有一个固定大小的桶，系统会以恒定速率向桶中放Token，桶满则暂时不放。
而用户则从桶中取Token，如果有剩余Token就可以一直取。如果没有剩余Token，则需要等到系统中被放置了Token才行。

本文则主要集中介绍下该组件的具体使用方法：

我们可以使用以下方法构造一个限流器对象：

```
limiter := NewLimiter(10, 1);
```

这里有两个参数：

- 第一个参数是r Limit。代表每秒可以向Token桶中产生多少token。Limit实际上是float64的别名。
- 第二个参数是b int。b代表Token桶的容量大小。

  那么，对于以上例子来说，其构造出的限流器含义为，其令牌桶大小为1, 以每秒10个Token的速率向桶中放置Token。

除了直接指定每秒产生的Token个数外，还可以用Every方法来指定向Token桶中放置Token的间隔，例如：

```
limit := Every(100 * time.Millisecond);
limiter := NewLimiter(limit, 1);
```

以上就表示每100ms往桶中放一个Token。本质上也就是一秒钟产生10个。

Limiter提供了三类方法供用户消费Token，用户可以每次消费一个Token，也可以一次性消费多个Token。
而每种方法代表了当Token不足时，各自不同的对应手段。

Wait/WaitN

```
func (lim *Limiter) Wait(ctx context.Context) (err error)
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)
```

Wait实际上就是WaitN(ctx,1)。

当使用Wait方法消费Token时，如果此时桶内Token数组不足(小于N)，那么Wait方法将会阻塞一段时间，直至Token满足条件。如果充足则直接返回。

这里可以看到，Wait方法有一个context参数。
我们可以设置context的Deadline或者Timeout，来决定此次Wait的最长时间。

Allow/AllowN

```
func (lim *Limiter) Allow() bool
func (lim *Limiter) AllowN(now time.Time, n int) bool
```

Allow实际上就是AllowN(time.Now(),1)。

AllowN方法表示，截止到某一时刻，目前桶中数目是否至少为n个，满足则返回true，同时从桶中消费n个token。
反之返回不消费Token，false。

通常对应这样的线上场景，如果请求速率过快，就直接丢到某些请求。

Reserve/ReserveN

```
func (lim *Limiter) Reserve() *Reservation
func (lim *Limiter) ReserveN(now time.Time, n int) *Reservation
```

Reserve相当于ReserveN(time.Now(), 1)。

ReserveN的用法就相对来说复杂一些，当调用完成后，无论Token是否充足，都会返回一个Reservation*对象。

你可以调用该对象的Delay()方法，该方法返回了需要等待的时间。如果等待时间为0，则说明不用等待。
必须等到等待时间之后，才能进行接下来的工作。

或者，如果不想等待，可以调用Cancel()方法，该方法会将Token归还。

举一个简单的例子，我们可以这么使用Reserve方法。

```
r := lim.Reserve()
f !r.OK() {
    // Not allowed to act! Did you remember to set lim.burst to be > 0 ?
    return
}
time.Sleep(r.Delay())
Act() // 执行相关逻辑
```

动态调整速率

Limiter支持可以调整速率和桶大小：

```
SetLimit(Limit) 改变放入Token的速率
SetBurst(int) 改变Token桶大小
```

有了这两个方法，可以根据现有环境和条件，根据我们的需求，动态的改变Token桶大小和速率

实例代码

```
package main

import (
    "context"
    "log"
    "time"

    "golang.org/x/time/rate"
)

//limit表示每秒产生token数，buret最多存token数
//Allow判断当前是否可以取到token
//Wait阻塞等待知道取到token
//Reserve返回等待时间，再去取token

func main() {
    l := rate.NewLimiter(1, 5)
    log.Println(l.Limit(), l.Burst())
    for i := 0; i < 100; i++ {
        //阻塞等待直到，取到一个token
        log.Println("before Wait")
        c, _ := context.WithTimeout(context.Background(), time.Second*2)
        if err := l.Wait(c); err != nil {
            log.Println("limiter wait err:" + err.Error())
        }
```

```
        log.Println("after Wait")

        //返回需要等待多久才有新的token,这样就可以等待指定时间执行任务
        r := l.Reserve()
        log.Println("reserve Delay:", r.Delay())

        //判断当前是否可以取到token
        a := l.Allow()
        log.Println("Allow:", a)
    }
}
```

缓存

go-cache是内存中的键：类似于memcached的值存储/缓存，适用于在一台计算机上运行的应用程序。它的主要优点是，由于本质上是map[string]interface{}具有到期时间的线程安全的，因此不需要通过网络序列化或传输其内容。

可以在给定的持续时间内或永久存储任何对象，并且可以由多个goroutine安全使用缓存。

尽管不打算将go-cache用作持久性数据存储，但可以将整个缓存保存到文件中并从文件中加载（c.Items()用于检索要映射的项目映射并NewFrom()从反序列化的缓存中创建缓存）以进行恢复从停机时间很快。（有关NewFrom()警告，请参阅文档。）

安装

```
import (
    "fmt"
    "github.com/patrickmn/go-cache"
    "time"
)

func main() {
    // Create a cache with a default expiration time of 5 minutes, and which
    // purges expired items every 10 minutes
    c := cache.New(5*time.Minute, 10*time.Minute)

    // Set the value of the key "foo" to "bar", with the default expiration time
    c.Set("foo", "bar", cache.DefaultExpiration)

    // Set the value of the key "baz" to 42, with no expiration time
    // (the item won't be removed until it is re-set, or removed using
    // c.Delete("baz")
    c.Set("baz", 42, cache.NoExpiration)

    // Get the string associated with the key "foo" from the cache
    foo, found := c.Get("foo")
    if found {
        fmt.Println(foo)
    }

    // Since Go is statically typed, and cache values can be anything, type
    // assertion is needed when values are being passed to functions that don't
    // take arbitrary types, (i.e. interface{}). The simplest way to do this for
    // values which will only be used once--e.g. for passing to another
    // function--is:
```

```
    foo, found := c.Get("foo")
    if found {
        MyFunction(foo.(string))
    }

    // This gets tedious if the value is used several times in the same functio
n.
    // You might do either of the following instead:
    if x, found := c.Get("foo"); found {
        foo := x.(string)
        // ...
    }
    // or
    var foo string
    if x, found := c.Get("foo"); found {
        foo = x.(string)
    }
    // ...
    // foo can then be passed around freely as a string

    // Want performance? Store pointers!
    c.Set("foo", &MyStruct, cache.DefaultExpiration)
    if x, found := c.Get("foo"); found {
        foo := x.(*MyStruct)
            // ...
    }
}
```

支付插件

一、安装

```
$ go get github.com/iGoogle-ink/gopay
```

查看 GoPay 版本

版本更新记录

```
import (
    "fmt"

    "github.com/iGoogle-ink/gopay"
)

func main() {
    fmt.Println("GoPay Version: ", gopay.Version)
}
```

微信支付API

- 统一下单：client.UnifiedOrder()
    - JSAPI - JSAPI支付（或小程序支付）
    - NATIVE - Native支付
    - APP - app支付
    - MWEB - H5支付
- 提交付款码支付：client.Micropay()
- 查询订单：client.QueryOrder()
- 关闭订单：client.CloseOrder()
- 撤销订单：client.Reverse()
- 申请退款：client.Refund()
- 查询退款：client.QueryRefund()
- 下载对账单：client.DownloadBill()
- 下载资金账单（正式）：client.DownloadFundFlow()

- 交易保障：client.Report()
- 拉取订单评价数据（正式）：client.BatchQueryComment()
- 企业付款（正式）：client.Transfer()
- 查询企业付款（正式）：client.GetTransferInfo()
- 授权码查询OpenId（正式）：client.AuthCodeToOpenId()
- 公众号纯签约（正式）：client.EntrustPublic()
- APP纯签约-预签约接口-获取预签约ID（正式）：client.EntrustAppPre()
- H5纯签约（正式）：client.EntrustH5()
- 支付中签约（正式）：client.EntrustPaying()
- 请求单次分账（正式）：client.ProfitSharing()
- 请求多次分账（正式）：client.MultiProfitSharing()
- 查询分账结果（正式）：client.ProfitSharingQuery()
- 添加分账接收方（正式）：client.ProfitSharingAddReceiver()
- 删除分账接收方（正式）：client.ProfitSharingRemoveReceiver()
- 完结分账（正式）：client.ProfitSharingFinish()
- 分账回退（正式）：client.ProfitSharingReturn()
- 分账回退结果查询（正式）：client.ProfitSharingReturnQuery()
- 企业付款到银行卡API（正式）：client.PayBank()
- 查询企业付款到银行卡API（正式）：client.QueryBank()
- 获取RSA加密公钥API（正式）：client.GetRSAPublicKey()
- 自定义方法请求微信API接口：client.PostWeChatAPISelf()

微信公共API

- wechat.GetParamSign() => 获取微信支付所需参数里的Sign值（通过支付参数计算Sign值）
- wechat.GetSanBoxParamSign() => 获取微信支付沙箱环境所需参数里的Sign值（通过支付参数计算Sign值）
- wechat.GetMiniPaySign() => 获取微信小程序支付所需要的paySign
- wechat.GetH5PaySign() => 获取微信内H5支付所需要的paySign
- wechat.GetAppPaySign() => 获取APP支付所需要的paySign
- wechat.ParseNotifyToBodyMap() => 解析微信支付异步通知的参数到BodyMap
- wechat.ParseNotify() => 解析微信支付异步通知的参数

- wechat.ParseRefundNotify() => 解析微信退款异步通知的参数
- wechat.VerifySign() => 微信同步返回参数验签或异步通知参数验签
- wechat.Code2Session() => 登录凭证校验：获取微信用户OpenId、UnionId、SessionKey
- wechat.GetAppletAccessToken() => 获取微信小程序全局唯一后台接口调用凭据
- wechat.GetAppletPaidUnionId() => 微信小程序用户支付完成后，获取该用户的UnionId，无需用户授权
- wechat.GetUserInfo() => 微信公众号：获取用户基本信息(UnionID机制)
- wechat.DecryptOpenDataToStruct() => 加密数据，解密到指定结构体
- wechat.DecryptOpenDataToBodyMap() => 加密数据，解密到 BodyMap
- wechat.GetOpenIdByAuthCode() => 授权码查询openid
- wechat.GetAppLoginAccessToken() => App应用微信第三方登录，code换取access_token
- wechat.RefreshAppLoginAccessToken() => 刷新App应用微信第三方登录后，获取的access_token
- wechat.DecryptRefundNotifyReqInfo() => 解密微信退款异步通知的加密数据

QQ支付API

- 提交付款码支付：client.MicroPay()
- 撤销订单：client.Reverse()
- 统一下单：client.UnifiedOrder()
- 订单查询：client.OrderQuery()
- 关闭订单：client.CloseOrder()
- 申请退款：client.Refund()
- 退款查询：client.RefundQuery()
- 交易账单：client.StatementDown()
- 资金账单：client.AccRoll()
- 自定义方法请求微信API接口：client.PostQQAPISelf()

QQ公共API

- qq.ParseNotifyToBodyMap() => 解析QQ支付异步通知的结果到BodyMap
- qq.ParseNotify() => 解析QQ支付异步通知的参数

- qq.VerifySign() => QQ同步返回参数验签或异步通知参数验签

支付宝支付API

> **因支付宝接口太多，如没实现的接口，还请开发者自行调用client.PostAliPayAPISelf()方法实现！**
>
> - 支付宝接口自行实现方法：client.PostAliPayAPISelf()
> - 手机网站支付接口2.0（手机网站支付）：client.TradeWapPay()
> - 统一收单下单并支付页面接口（电脑网站支付）：client.TradePagePay()
> - APP支付接口2.0（APP支付）：client.TradeAppPay()
> - 统一收单交易支付接口（商家扫用户付款码）：client.TradePay()
> - 统一收单交易创建接口（小程序支付）：client.TradeCreate()
> - 统一收单线下交易查询：client.TradeQuery()
> - 统一收单交易关闭接口：client.TradeClose()
> - 统一收单交易撤销接口：client.TradeCancel()
> - 统一收单交易退款接口：client.TradeRefund()
> - 统一收单退款页面接口：client.TradePageRefund()
> - 统一收单交易退款查询：client.TradeFastPayRefundQuery()
> - 统一收单交易结算接口：client.TradeOrderSettle()
> - 统一收单线下交易预创建（用户扫商品收款码）：client.TradePrecreate()
> - 单笔转账接口：client.FundTransUniTransfer()
> - 转账业务单据查询接口：client.FundTransCommonQuery()
> - 支付宝资金账户资产查询接口：client.FundAccountQuery()
> - 换取授权访问令牌（获取access_token，user_id等信息）：client.SystemOauthToken()
> - 支付宝会员授权信息查询接口（App支付宝登录）：client.UserInfoShare()
> - 换取应用授权令牌（获取app_auth_token，auth_app_id，user_id等信息）：client.OpenAuthTokenApp()
> - 获取芝麻信用分：client.ZhimaCreditScoreGet()
> - 身份认证初始化服务：client.UserCertifyOpenInit()
> - 身份认证开始认证（获取认证链接）：client.UserCertifyOpenCertify()
> - 身份认证记录查询：client.UserCertifyOpenQuery()

> - 用户登陆授权：client.UserInfoAuth()
> - 支付宝商家账户当前余额查询：client.DataBillBalanceQuery()
> - 查询对账单下载地址：client.DataBillDownloadUrlQuery()

支付宝公共API

- alipay.GetCertSN() => 获取证书SN号（app_cert_sn、alipay_cert_sn）
- alipay.GetRootCertSN() => 获取证书SN号（alipay_root_cert_sn）
- alipay.GetRsaSign() => 获取支付宝参数签名（参数sign值）
- alipay.SystemOauthToken() => 换取授权访问令牌（得到access_token，user_id等信息）
- alipay.FormatPrivateKey() => 格式化应用私钥
- alipay.FormatPublicKey() => 格式化支付宝公钥
- alipay.FormatURLParam() => 格式化支付宝请求URL参数
- alipay.ParseNotifyToBodyMap() => 解析支付宝支付异步通知的参数到BodyMap
- alipay.ParseNotifyByURLValues() => 通过 url.Values 解析支付宝支付异步通知的参数到BodyMap
- alipay.VerifySign() => 支付宝异步通知参数验签
- alipay.VerifySignWithCert() => 支付宝异步通知参数验签（证书方式）
- alipay.VerifySyncSign() => 支付宝同步返回参数验签
- alipay.DecryptOpenDataToStruct() => 解密支付宝开放数据到 结构体
- alipay.DecryptOpenDataToBodyMap() => 解密支付宝开放数据到 BodyMap
- alipay.MonitorHeartbeatSyn() => 验签接口

二、文档说明

1、初始化GoPay客户端并做配置（HTTP请求均默认设置tls.Config{InsecureSkipVerify: true}）

• 微信

微信官方文档：官方文档

```
import (
    "github.com/iGoogle-ink/gopay/wechat"
```

```
)

// 初始化微信客户端
//    appId: 应用ID
//    mchId: 商户ID
//    apiKey: API秘钥值
//    isProd: 是否是正式环境
client := wechat.NewClient("wxdaa2ab9ef87b5497", mchId, apiKey, false)

// 设置国家: 不设置默认 中国国内
//    gopay.China: 中国国内
//    gopay.China2: 中国国内备用
//    gopay.SoutheastAsia: 东南亚
//    gopay.Other: 其他国家
client.SetCountry(gopay.China)

// 添加微信证书 Path 路径
//    certFilePath: apiclient_cert.pem 路径
//    keyFilePath: apiclient_key.pem 路径
//    pkcs12FilePath: apiclient_cert.p12 路径
//    返回err
client.AddCertFilePath()
```

- **支付宝**

支付宝官方文档：官方文档

支付宝RSA秘钥生成文档：生成RSA密钥 （推荐使用 RSA2）

沙箱环境使用说明：文档地址

```
import (
    "github.com/iGoogle-ink/gopay/alipay"
)

// 初始化支付宝客户端
//    appId: 应用ID
//    privateKey: 应用私钥，支持PKCS1和PKCS8
//    isProd: 是否是正式环境
client := alipay.NewClient("2016091200494382", privateKey, false)

// 设置支付宝请求 公共参数
//    注意: 具体设置哪些参数，根据不同的方法而不同，此处列举出所有设置参数
client.SetLocation().                          // 设置时区，不设置或出错均为默认服务器时间
```

```
    SetPrivateKeyType().                    // 设置 支付宝 私钥类型，alipay.PKCS1 或 alipay.PKCS8，默认 PKCS1
    SetAliPayRootCertSN().                  // 设置支付宝根证书SN，通过 alipay.GetRootCertSN() 获取
    SetAppCertSN().                         // 设置应用公钥证书SN，通过 alipay.GetCertSN() 获取
    SetAliPayPublicCertSN().                // 设置支付宝公钥证书SN，通过 alipay.GetCertSN() 获取
    SetCharset("utf-8").                    // 设置字符编码，不设置默认 utf-8
    SetSignType(alipay.RSA2).               // 设置签名类型，不设置默认 RSA2
    SetReturnUrl("https://www.gopay.ink").  // 设置返回URL
    SetNotifyUrl("https://www.gopay.ink").  // 设置异步通知URL
    SetAppAuthToken().                      // 设置第三方应用授权
    SetAuthToken()                          // 设置个人信息授权

err := client.SetCertSnByPath("appCertPublicKey.crt", "alipayRootCert.crt", "alipayCertPublicKey_RSA2.crt")
```

2、初始化并赋值BodyMap（client的方法所需的入参）

• 微信请求参数

具体参数请根据不同接口查看：微信支付接口文档

```
import (
    "github.com/iGoogle-ink/gopay/wechat"
)

// 初始化 BodyMap
bm := make(gopay.BodyMap)
bm.Set("nonce_str", gotil.GetRandomString(32))
bm.Set("body", "小程序测试支付")
bm.Set("out_trade_no", number)
bm.Set("total_fee", 1)
bm.Set("spbill_create_ip", "127.0.0.1")
bm.Set("notify_url", "http://www.gopay.ink")
bm.Set("trade_type", gopay.TradeType_Mini)
bm.Set("device_info", "WEB")
bm.Set("sign_type", gopay.SignType_MD5)
bm.Set("openid", "o0Df70H2QOfY8JXh1aFPIRyOBgu8")

// 嵌套json格式数据（例如：H5支付的 scene_info 参数）
h5Info := make(map[string]string)
h5Info["type"] = "Wap"
h5Info["wap_url"] = "http://www.gopay.ink"
```

```
h5Info["wap_name"] = "H5测试支付"

sceneInfo := make(map[string]map[string]string)
sceneInfo["h5_info"] = h5Info

bm.Set("scene_info", sceneInfo)

// 参数 sign ，可单独生成赋值到BodyMap中；也可不传sign参数，client内部会自动获取
// 如需单独赋值 sign 参数，需通过下面方法，最后获取sign值并在最后赋值此参数
sign := wechat.GetParamSign("wxdaa2ab9ef87b5497", mchId, apiKey, body)
// sign, _ := wechat.GetSanBoxParamSign("wxdaa2ab9ef87b5497", mchId, apiKey, bod
y)
bm.Set("sign", sign)
```

- **支付宝请求参数**

具体参数请根据不同接口查看：支付宝支付API接口文档

```
// 初始化 BodyMap
bm := make(gopay.BodyMap)
bm.Set("subject", "手机网站测试支付")
bm.Set("out_trade_no", "GZ201901301040355703")
bm.Set("quit_url", "https://www.gopay.ink")
bm.Set("total_amount", "100.00")
bm.Set("product_code", "QUICK_WAP_WAY")
```

3、client 方法调用

- **微信 client**

```
wxRsp, err := client.UnifiedOrder(bm)
wxRsp, err := client.Micropay(bm)
wxRsp, err := client.QueryOrder(bm)
wxRsp, err := client.CloseOrder(bm)
wxRsp, err := client.Reverse(bm, "apiclient_cert.pem", "apiclient_key.pem",
"apiclient_cert.p12")
wxRsp, err := client.Refund(bm, "apiclient_cert.pem", "apiclient_key.pem",
"apiclient_cert.p12")
wxRsp, err := client.QueryRefund(bm)
wxRsp, err := client.DownloadBill(bm)
wxRsp, err := client.DownloadFundFlow(bm, "apiclient_cert.pem", "apiclient_k
ey.pem", "apiclient_cert.p12")
wxRsp, err := client.BatchQueryComment(bm, "apiclient_cert.pem", "apiclient_
```

```
key.pem", "apiclient_cert.p12")
wxRsp, err := client.Transfer(bm, "apiclient_cert.pem", "apiclient_key.pem",
"apiclient_cert.p12")
```

- **支付宝 client**

```
// 手机网站支付是通过服务端获取支付URL后，然后返回给客户端，请求URL地址即可打开支付页面
payUrl, err := client.TradeWapPay(bm)

// 电脑网站支付是通过服务端获取支付URL后，然后返回给客户端，请求URL地址即可打开支付页面
payUrl, err := client.TradePagePay(bm)

// APP支付是通过服务端获取支付参数后，然后通过Android/iOS客户端的SDK调用支付功能
payParam, err := client.TradeAppPay(bm)

// 商家使用扫码枪等条码识别设备扫描用户支付宝钱包上的条码/二维码，完成收款
aliRsp, err := client.TradePay(bm)

// 支付宝小程序支付时 buyer_id 为必传参数，需要提前获取，获取方法如下两种
//    1、gopay.SystemOauthToken()     返回取值：rsp.SystemOauthTokenResponse.UserId
//    2、client.SystemOauthToken()    返回取值：aliRsp.SystemOauthTokenResponse.UserId
aliRsp, err := client.TradeCreate(bm)

aliRsp, err := client.TradeQuery(bm)
aliRsp, err := client.TradeClose(bm)
aliRsp, err := client.TradeCancel(bm)
aliRsp, err := client.TradeRefund(bm)
aliRsp, err := client.TradePageRefund(bm)
aliRsp, err := client.TradeFastPayRefundQuery(bm)
aliRsp, err := client.TradeOrderSettle(bm)
aliRsp, err := client.TradePrecreate(bm)
aliRsp, err := client.FundTransUniTransfer(bm)
aliRsp, err := client.FundTransCommonQuery(bm)
aliRsp, err := client.FundAccountQuery(bm)
aliRsp, err := client.SystemOauthToken(bm)
aliRsp, err := client.OpenAuthTokenApp(bm)
aliRsp, err := client.ZhimaCreditScoreGet(bm)
aliRsp, err := client.UserCertifyOpenInit(bm)
aliRsp, err := client.UserCertifyOpenCertify(bm)
aliRsp, err := client.UserCertifyOpenQuery(bm)
```

4、微信统一下单后，获取微信小程序支付、APP支付、微信内H5支付所需要的 paySign

- 微信（只有微信需要此操作）

微信小程序支付官方文档：微信小程序支付API

APP支付官方文档：APP端调起支付的参数列表文档

微信内H5支付官方文档：微信内H5支付文档

```
import (
    "github.com/iGoogle-ink/gopay/wechat"
)

// ====微信小程序 paySign====
timeStamp := strconv.FormatInt(time.Now().Unix(), 10)
prepayId := "prepay_id=" + wxRsp.PrepayId    // 此处的 wxRsp.PrepayId ，统一下单成功后得到
// 获取微信小程序支付的 paySign
//    appId: AppID
//    nonceStr: 随机字符串
//    prepayId: 统一下单成功后得到的值
//    signType: 签名方式，务必与统一下单时用的签名方式一致
//    timeStamp: 时间
//    apiKey: API秘钥值
paySign := wechat.GetMiniPaySign(AppID, wxRsp.NonceStr, prepayId, wechat.SignType_MD5, timeStamp, apiKey)

// ====APP支付 paySign====
timeStamp := strconv.FormatInt(time.Now().Unix(), 10)
// 获取APP支付的 paySign
// 注意：package 参数因为是固定值，无需开发者再传入
//    appId: AppID
//    partnerid: partnerid
//    nonceStr: 随机字符串
//    prepayId: 统一下单成功后得到的值
//    signType: 签名方式，务必与统一下单时用的签名方式一致
//    timeStamp: 时间
//    apiKey: API秘钥值
paySign := wechat.GetAppPaySign(appid, partnerid, wxRsp.NonceStr, wxRsp.PrepayId, wechat.SignType_MD5, timeStamp, apiKey)

// ====微信内H5支付 paySign====
timeStamp := strconv.FormatInt(time.Now().Unix(), 10)
```

```
packages := "prepay_id=" + wxRsp.PrepayId   // 此处的 wxRsp.PrepayId ，统一下单成功后得到
// 获取微信内H5支付 paySign
//    appId：AppID
//    nonceStr：随机字符串
//    packages：统一下单成功后拼接得到的值
//    signType：签名方式，务必与统一下单时用的签名方式一致
//    timeStamp：时间
//    apiKey：API秘钥值
paySign := wechat.GetH5PaySign(AppID, wxRsp.NonceStr, packages, wechat.SignType_MD5, timeStamp, apiKey)
```

5、同步返回参数验签Sign、异步通知参数解析和验签Sign、异步通知返回

异步参数需要先解析，解析出来的结构体或BodyMap再验签

Echo Web框架，有兴趣的可以尝试一下

异步通知处理完后，需回复平台固定数据

- 微信

```
import (
    "github.com/iGoogle-ink/gopay"
    "github.com/iGoogle-ink/gopay/wechat"
)

// ====同步返回参数验签Sign====
wxRsp, err := client.UnifiedOrder(bm)
// 微信同步返回参数验签或异步通知参数验签
//    apiKey：API秘钥值
//    signType：签名类型（调用API方法时填写的类型）
//    bean：微信同步返回的结构体 wxRsp 或 异步通知解析的结构体 notifyReq
//    返回参数 ok：是否验签通过
//    返回参数 err：错误信息
ok, err := wechat.VerifySign(apiKey, wechat.SignType_MD5, wxRsp)

// ====支付异步通知参数解析和验签Sign====
// 解析支付异步通知的参数
//    req：*http.Request
//    返回参数 notifyReq：通知的参数
//    返回参数 err：错误信息
notifyReq, err := wechat.ParseNotify(c.Request())    // c.Request()是 echo 框架
```

```
的获取 *http.Request 的写法
// 验签操作
ok, err := wechat.VerifySign(apiKey, wechat.SignType_MD5, notifyReq)

// ====退款异步通知参数解析，退款通知无sign，不用验签====
//
// 解析退款异步通知的参数，解析出来的 req_info 是加密数据，需解密
//    req：*http.Request
//    返回参数 notifyReq：通知的参数
//    返回参数 err：错误信息
notifyReq, err := wechat.ParseRefundNotify(c.Request())

// ==解密退款异步通知的加密参数 req_info ==
refundNotify, err := wechat.DecryptRefundNotifyReqInfo(notifyReq.ReqInfo, apiKe
y)

// ==异步通知，返回给微信平台的信息==
rsp := new(wechat.NotifyResponse) // 回复微信的数据
rsp.ReturnCode = gopay.SUCCESS
rsp.ReturnMsg = gopay.OK
return c.String(http.StatusOK, rsp.ToXmlString())   // 此写法是 echo 框架返回客
户端数据的写法
```

- **支付宝**

注意：APP支付、手机网站支付、电脑网站支付 暂不支持同步返回验签

支付宝支付后的同步/异步通知验签文档：支付结果通知

```
import (
    "github.com/iGoogle-ink/gopay/alipay"
)

// ====同步返回参数验签Sign====
aliRsp, err := client.TradePay(bm)
// 支付宝同步返回验签
//    注意：APP支付，手机网站支付，电脑网站支付 暂不支持同步返回验签
//    aliPayPublicKey：支付宝公钥
//    signData：待验签参数，aliRsp.SignData
//    sign：待验签sign，aliRsp.Sign
//    返回参数ok：是否验签通过
//    返回参数err：错误信息
ok, err := alipay.VerifySyncSign(aliPayPublicKey, aliRsp.SignData, aliRsp.Sign)

// ====异步通知参数解析和验签Sign====
```

```
// 解析异步通知的参数
//     req：*http.Request
//     返回参数 notifyReq：通知的参数
//     返回参数 err：错误信息
notifyReq, err = alipay.ParseNotify(c.Request())     // c.Request()是 echo 框架的获取
// 验签操作
ok, err = alipay.VerifySign(aliPayPublicKey, notifyReq)

// ==异步通知，返回支付宝平台的信息==
//     文档：https://opendocs.alipay.com/open/203/105286
//     程序执行完后必须打印输出“success”（不包含引号）。如果商户反馈给支付宝的字符不是success这7个字符，支付宝服务器会不断重发通知，直到超过24小时22分钟。一般情况下，25小时以内完成8次通知（通知的间隔频率一般是：4m,10m,10m,1h,2h,6h,15h）
return c.String(http.StatusOK, "success")    // 此写法是 echo 框架返回客户端数据的写法
```

6、微信、支付宝 公共API（仅部分说明）

• 微信 公共API

官方文档：code2Session

button按钮获取手机号码：button组件文档

微信解密算法文档：解密算法文档

```
import (
    "github.com/iGoogle-ink/gopay/wechat"
)

// 获取微信小程序用户的OpenId、SessionKey、UnionId
//     appId：微信小程序的APPID
//     appSecret：微信小程序的AppSecret
//     wxCode：小程序调用wx.login 获取的code
sessionRsp, err := wechat.Code2Session(appId, appSecret, wxCode)

// ====解密微信加密数据到指定结构体====

//小程序获取手机号
data := "Kf3TdPbzEmhWMuPKtlKxIWDkijhn402w1bxoHL4kLdcKr6jT1jNcIhvDJfjXmJcgDWLjmBiIGJ5acUuSvxLws3WgAkERmtTuiCG1OCKLsJiR+AXVk7B2TUQzsq88YVilDz/YAN3647REE7glGmeBPfvUmdbfDzhL9BzvEiuRhABuCYyTMz4iaM8hFjbLB1caaeoOlykYAFMWC5pZi9P8uw=="
iv := "Cds8j3VYoGvnTp1BrjXdJg=="
session := "lyY4HPQbaOYzZdG+JcYK9w=="
```

```
phone := new(wechat.UserPhone)
// 解密开放数据
//     encryptedData: 包括敏感数据在内的完整用户信息的加密数据，小程序获取到
//     iv: 加密算法的初始向量，小程序获取到
//     sessionKey: 会话密钥，通过 gopay.Code2Session() 方法获取到
//     beanPtr: 需要解析到的结构体指针，操作完后，声明的结构体会被赋值
err := wechat.DecryptOpenDataToStruct(data, iv, session, phone)
fmt.Println(*phone)
// 获取微信小程序用户信息
sessionKey := "tiihtNczf5v6AKRyjwEUhQ=="
encryptedData := "CiyLU1Aw2KjvrjMdj8YKliAjtP4gsMZMQmRzooG2xrDcvSnxIMXFufNstNGTyaGS9uT5geRaOW4oTOb1WT7fJlAC+oNPdbB+3hVbJSRgv+4lGOETKUQz6OYStslQ142dNCuabNPGBzlooOmB231qMM85d2/fV6ChevvXvQP8Hkue1poOFtnEtpyxVLW1zAo6/1Xx1COxFvrc2d7UL/lmHInNlxuacJXwuOfjpXfz/YqYzBIBzD6WUfTIF9GRHpOn/Hz7saL8xz+W//FRAUid1OksQaQx4CMs8LOddcQhULW4ucetDf96JcR3gOgfRK4PC7E/r7Z6xNrXd2UIeorGj5Ef7b1pJAYB6Y5anaHqZ9J6nKEBvB4DnNLIVWSgARns/8wR2SiRS7MNACwTyrGvt9ts8p12PKFdlqYTopNHR1Vf7XjfhQlVsAJdNiKdYmYVoKlaRv85IfVunYzOOIKXsyl7JCUjCpoG20f0a04COwfneQAGGwd5oa+T8yO5hzuyDb/XcxxmKO1EpqOyuxINew=="
iv2 := "r7BXXKkLb8qrSNn05n0qiA=="

// 微信小程序 用户信息
userInfo := new(wechat.AppletUserInfo)
err = wechat.DecryptOpenDataToStruct(encryptedData, iv2, sessionKey, userInfo)
fmt.Println(*userInfo)

data := "Kf3TdPbzEmhWMuPKtlKxIWDkijhn402w1bxoHL4kLdcKr6jT1jNcIhvDJfjXmJcgDWLjmBiIGJ5acUuSvxLws3WgAkERmtTuiCG1OCKLsJiR+AXVk7B2TUQzsq88YVilDz/YAN3647REE7glGmeBPfvUmdbfDzhL9BzvEiuRhABuCYyTMz4iaM8hFjbLB1caaeoOlykYAFMWC5pZi9P8uw=="
iv := "Cds8j3VYoGvnTp1BrjXdJg=="
session := "lyY4HPQbaOYzZdG+JcYK9w=="

// 解密开放数据到 BodyMap
//     encryptedData:包括敏感数据在内的完整用户信息的加密数据
//     iv:加密算法的初始向量
//     sessionKey:会话密钥
bm, err := wechat.DecryptOpenDataToBodyMap(data, iv, session)
if err != nil {
    fmt.Println("err:", err)
    return
}
fmt.Println("WeChatUserPhone:", bm)
```

• 支付宝 公共API

支付宝换取授权访问令牌文档：换取授权访问令牌

获取用户手机号文档：获取用户手机号

支付宝加解密文档：AES配置文档 ， AES加解密文档

```
import (
    "github.com/iGoogle-ink/gopay/alipay"
)

// 换取授权访问令牌（默认使用utf-8，RSA2）
//    appId：应用ID
//    privateKey：应用私钥，支持PKCS1和PKCS8
//    grantType：值为 authorization_code 时，代表用code换取；值为 refresh_token 时，代表用refresh_token换取，传空默认code换取
//    codeOrToken：支付宝授权码或refresh_token
rsp, err := alipay.SystemOauthToken(appId, privateKey, grantType, codeOrToken)

// 解密支付宝开放数据带到指定结构体
//    以小程序获取手机号为例
phone := new(alipay.UserPhone)
// 解密支付宝开放数据
//    encryptedData:包括敏感数据在内的完整用户信息的加密数据
//    secretKey:AES密钥，支付宝管理平台配置
//    beanPtr:需要解析到的结构体指针
err := alipay.DecryptOpenDataToStruct(encryptedData, secretKey, phone)
fmt.Println(*phone)
```

License

```
Copyright 2019 Jerry

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

获取https过期时间

HTTPS介绍

HTTPS （全称：Hyper Text Transfer Protocol over SecureSocket Layer），是以安全为目标的 HTTP 通道，在HTTP的基础上通过传输加密和身份认证保证了传输过程的安全性 。HTTPS 在HTTP 的基础下加入SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP与 TCP 之间）。这个系统提供了身份验证与加密通讯方法。它被广泛用于万维网上安全敏感的通讯，例如交易支付等方面

获取HTTPS证书过期时间

```
package main

import (
    "crypto/tls"
    "fmt"
    "net/http"
)

func main() {
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    }
    client := &http.Client{Transport: tr}

    seedUrl := "https://www.baidu.cn"
    resp, err := client.Get(seedUrl)
    defer resp.Body.Close()

    if err != nil {
        fmt.Errorf(seedUrl, " 请求失败")
        panic(err)
    }

    //fmt.Println(resp.TLS.PeerCertificates[0])
    certInfo := resp.TLS.PeerCertificates[0]
    fmt.Println("过期时间:", certInfo.NotAfter)
    fmt.Println("组织信息:", certInfo.Subject)
}
```

获取服务器配置

获取服务器相关的信息

```
package main

import (
    "fmt"
    "runtime"
    "time"

    "github.com/shirou/gopsutil/cpu"
    "github.com/shirou/gopsutil/disk"
    "github.com/shirou/gopsutil/load"
    "github.com/shirou/gopsutil/mem"
)

const (
    B  = 1
    KB = 1024 * B
    MB = 1024 * KB
    GB = 1024 * MB
)

func main() {
    DiskCheck()
    OSCheck()
    CPUCheck()
    RAMCheck()
}

//服务器硬盘使用量
func DiskCheck() {
    u, _ := disk.Usage("/")
    usedMB := int(u.Used) / MB
    usedGB := int(u.Used) / GB
    totalMB := int(u.Total) / MB
    totalGB := int(u.Total) / GB
    usedPercent := int(u.UsedPercent)
    fmt.Printf("Free space: %dMB (%dGB) / %dMB (%dGB) | Used: %d%%\n", usedMB, u
sedGB, totalMB, totalGB, usedPercent)
}
```

```
//OS
func OSCheck() {
    fmt.Printf("goOs:%s,compiler:%s,numCpu:%d,version:%s,numGoroutine:%d\n", runtime.GOOS, runtime.Compiler, runtime.NumCPU(), runtime.Version(), runtime.NumGoroutine())
}

//CPU 使用量
func CPUCheck() {
    cores, _ := cpu.Counts(false)

    cpus, err := cpu.Percent(time.Duration(200)*time.Millisecond, true)
    if err == nil {
        for i, c := range cpus {
            fmt.Printf("cpu%d : %f%%\n", i, c)
        }
    }
    a, _ := load.Avg()
    l1 := a.Load1
    l5 := a.Load5
    l15 := a.Load15
    fmt.Println(l1)
    fmt.Println(l5)
    fmt.Println(l15)
    fmt.Println(cores)
}

//内存使用量
func RAMCheck() {
    u, _ := mem.VirtualMemory()
    usedMB := int(u.Used) / MB
    totalMB := int(u.Total) / MB
    usedPercent := int(u.UsedPercent)
    fmt.Printf("usedMB:%d,totalMB:%d,usedPercent:%d", usedMB, totalMB, usedPercent)
}
```

热重启

需求来源

作为一个新萌的假后端，开发接口时候经常需要修改调试重启素质三连。但是频繁修改和频繁的重启会导致非常麻烦。作为一个老前端，自然是厌恶手动重启这种劳心费力的操作的。于是就像找一个在开发环境快速热重启的方案。

工具

经过一番探索，找到以下开源项目：

```
https://github.com/cosmtrek/air
```

项目中使用

按照该项目文档说法直接一波操作。

安装

```
go get -u github.com/cosmtrek/air
```

创建配置文件

在项目根目录创建一个名为 .air.conf 的配置文件。创建完毕之后，在文件中写入你应用运行的命令如：

```
go build main.go
```

运行项目

运行项目只需要在项目根目录执行如下命令：

```
air -c .air.conf
```

如果你的配置文件是 .air.conf 那么你只需要运行 air 就可以了。

总结

项目运行后，我们再次编辑项目中的文件，项目的服务会自动重启了。话说，如果是接口测试，如果能够在有个回调，在项目重启后，自动帮我们在浏览器刷新一下测试请求那就更完美了。

转自：https://segmentfault.com/a/1190000021808960

原生中间件

web开发的背景下，“中间件”通常意思是“包装原始应用并添加一些额外的功能的应用的一部分”。这个概念似乎总是不被人理解，但是我认为中间件非常棒。

首先，一个好的中间件有一个责任就是可插拔并且自足。这就意味着你可以在接口级别嵌入你的中间件他就能直接运行。它不会影响你编码方式，不是框架，仅仅是你请求处理里面的一层而已。完全没必要重写你的代码，如果你想使用中间件的一个功能，你就帮他插入到那里，如果不想使用了，就可以直接移除。

纵观Go语言，中间件是非常普遍的，即使在标准库中。虽然开始的时候不会那么明显，在标准库net/http中的函数StripText或者TimeoutHandler就是我们要定义和的中间件的样子，处理请求和相应的时候他们包装你的handler，并处理一些额外的步骤。

一开始，我们认为编写中间件似乎很容易，但是我们实际编写的时候也会遇到各种各样的坑。让我们来看看一些例子。

代码实现

```
//middleware/slice_router.go
package middleware

import (
    "context"
    "math"
    "net/http"
    "strings"
)

//目标定位是 tcp、http通用的中间件
//知其然也知其所以然

const abortIndex int8 = math.MaxInt8 / 2 //最多 63 个中间件

type HandlerFunc func(*SliceRouterContext)

// router 结构体
type SliceRouter struct {
    groups []*SliceGroup
}

// group 结构体
type SliceGroup struct {
    *SliceRouter
```

```
    path     string
    handlers []HandlerFunc
}

// router上下文
type SliceRouterContext struct {
    Rw  http.ResponseWriter
    Req *http.Request
    Ctx context.Context
    *SliceGroup
    index int8
}

func newSliceRouterContext(rw http.ResponseWriter, req *http.Request, r *SliceRo
uter) *SliceRouterContext {
    newSliceGroup := &SliceGroup{}

    //最长url前缀匹配
    matchUrlLen := 0
    for _, group := range r.groups {
        //fmt.Println("req.RequestURI")
        //fmt.Println(req.RequestURI)
        if strings.HasPrefix(req.RequestURI, group.path) {
            pathLen := len(group.path)
            if pathLen > matchUrlLen {
                matchUrlLen = pathLen
                *newSliceGroup = *group //浅拷贝数组指针
            }
        }
    }

    c := &SliceRouterContext{Rw: rw, Req: req, SliceGroup: newSliceGroup, Ctx: r
eq.Context()}
    c.Reset()
    return c
}

func (c *SliceRouterContext) Get(key interface{}) interface{} {
    return c.Ctx.Value(key)
}

func (c *SliceRouterContext) Set(key, val interface{}) {
    c.Ctx = context.WithValue(c.Ctx, key, val)
}

type SliceRouterHandler struct {
```

```
    coreFunc func(*SliceRouterContext) http.Handler
    router   *SliceRouter
}

func (w *SliceRouterHandler) ServeHTTP(rw http.ResponseWriter, req *http.Request
) {
    c := newSliceRouterContext(rw, req, w.router)
    if w.coreFunc != nil {
        c.handlers = append(c.handlers, func(c *SliceRouterContext) {
            w.coreFunc(c).ServeHTTP(rw, req)
        })
    }
    c.Reset()
    c.Next()
}

func NewSliceRouterHandler(coreFunc func(*SliceRouterContext) http.Handler, rout
er *SliceRouter) *SliceRouterHandler {
    return &SliceRouterHandler{
        coreFunc: coreFunc,
        router:   router,
    }
}

// 构造 router
func NewSliceRouter() *SliceRouter {
    return &SliceRouter{}
}

// 创建 Group
func (g *SliceRouter) Group(path string) *SliceGroup {
    return &SliceGroup{
        SliceRouter: g,
        path:        path,
    }
}

// 构造回调方法
func (g *SliceGroup) Use(middlewares ...HandlerFunc) *SliceGroup {
    g.handlers = append(g.handlers, middlewares...)
    existsFlag := false
    for _, oldGroup := range g.SliceRouter.groups {
        if oldGroup == g {
            existsFlag = true
        }
    }
```

```
    if !existsFlag {
        g.SliceRouter.groups = append(g.SliceRouter.groups, g)
    }
    return g
}

// 从最先加入中间件开始回调
func (c *SliceRouterContext) Next() {
    c.index++
    for c.index < int8(len(c.handlers)) {
        //fmt.Println("c.index")
        //fmt.Println(c.index)
        c.handlers[c.index](c)
        c.index++
    }
}

// 跳出中间件方法
func (c *SliceRouterContext) Abort() {
    c.index = abortIndex
}

// 是否跳过了回调
func (c *SliceRouterContext) IsAborted() bool {
    return c.index >= abortIndex
}

// 重置回调
func (c *SliceRouterContext) Reset() {
    c.index = -1
}
```

代码测试

实现两个中间件

```
//middleware/tracelog_slicemw.go
package middleware

import (
    "log"
)

func TraceLogSliceMW() func(c *SliceRouterContext) {
    return func(c *SliceRouterContext) {
```

```
        log.Println("trace_in")
        c.Next()
        log.Println("trace_out")
    }
}
```

```
//middleware/url.go
package middleware

func Url() func(c *SliceRouterContext) {
    return func(c *SliceRouterContext) {
        if c.Req.RequestURI == "/favicon.ico" {
            c.Abort()
        }
    }
}
```

代码测试

```
//main.go
package main

import (
    "fmt"
    "net/http"

    "github.com/student/middleware/middleware"
)

func main() {
    //初始化方法数组路由器
    sliceRouter := middleware.NewSliceRouter()
    //中间件可充当业务逻辑代码
    sliceRouter.Group("/").Use(middleware.Url(), middleware.TraceLogSliceMW(), f
unc(c *middleware.SliceRouterContext) {
        fmt.Println("中间件")
    })
    sliceRouter.Group("/topgoer").Use(middleware.Url(), middleware.TraceLogSlice
MW(), topgoer)
    routerHandler := middleware.NewSliceRouterHandler(nil, sliceRouter)
    err := http.ListenAndServe(":9090", routerHandler)
    if err != nil {
        fmt.Println("HTTP server failed,err:", err)
        return
```

```
    }
}
func topgoer(c *middleware.SliceRouterContext) {
    fmt.Println("www.topgoer.com是个不错的go语言中文文档")
}
```

设置https

Secure

Secure是用于Go的HTTP中间件，可促进快速获得安全性。这是一个标准的net / http Handler，可以与许多框架一起使用，也可以直接与Go的net / http包一起使用。

用法

```
// main.go
package main

import (
    "net/http"

    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

var myHandler = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("hello world"))
})

func main() {
    secureMiddleware := secure.New(secure.Options{
        AllowedHosts:          []string{"example\\.com", ".*\\.example\\.com"},
        AllowedHostsAreRegex:  true,
        HostsProxyHeaders:     []string{"X-Forwarded-Host"},
        SSLRedirect:           true,
        SSLHost:               "ssl.example.com",
        SSLProxyHeaders:       map[string]string{"X-Forwarded-Proto": "https"},
        STSSeconds:            31536000,
        STSIncludeSubdomains:  true,
        STSPreload:            true,
        FrameDeny:             true,
        ContentTypeNosniff:    true,
        BrowserXssFilter:      true,
        ContentSecurityPolicy: "script-src $NONCE",
        PublicKey:             `pin-sha256="base64+primary=="; pin-sha256="base64+backup=="; max-age=5184000; includeSubdomains; report-uri="https://www.example.com/hpkp-report"`,
    })

    app := secureMiddleware.Handler(myHandler)
```

```
    http.ListenAndServe("127.0.0.1:3000", app)
}
```

确保将安全中间件尽可能地靠近顶部（开始）（但在记录和恢复之后）。最好先进行允许的主机和SSL检查。

上面的示例仅允许使用主机名“ example.com”或“ ssl.example.com”的请求。同样，如果请求不是HTTPS，则将使用主机名“ ssl.example.com”将其重定向到HTTPS。满足这些要求后，它将添加以下标头：

```
Strict-Transport-Security: 31536000; includeSubdomains; preload
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Content-Security-Policy: script-src 'nonce-a2ZobGFoZg=='
PublicKey: pin-sha256="base64+primary=="; pin-sha256="base64+backup=="; max-age=5184000; includeSubdomains; report-uri="https://www.example.com/hpkp-report"
```

在开发时将IsDevelopment选项设置为true！

如果IsDevelopment为true，则AllowedHosts，SSLRedirect，STS头和HPKP头将无效。这使您可以在开发/测试模式下工作，而不必进行任何恼人的重定向到HTTPS（即，开发可以在HTTP上进行），或者阻止localhost主机出现问题。

可用选项

Secure带有多种配置选项（注意：这些不是默认选项值。请参见下面的默认值。）：

```
// ...
s := secure.New(secure.Options{
    AllowedHosts: []string{"ssl.example.com"}, // AllowedHosts is a list of fully qualified domain names that are allowed. Default is empty list, which allows any and all host names.
    AllowedHostsAreRegex: false,  // AllowedHostsAreRegex determines, if the provided AllowedHosts slice contains valid regular expressions. Default is false.
    HostsProxyHeaders: []string{"X-Forwarded-Hosts"}, // HostsProxyHeaders is a set of header keys that may hold a proxied hostname value for the request.
    SSLRedirect: true, // If SSLRedirect is set to true, then only allow HTTPS requests. Default is false.
    SSLTemporaryRedirect: false, // If SSLTemporaryRedirect is true, the a 302 will be used while redirecting. Default is false (301).
    SSLHost: "ssl.example.com", // SSLHost is the host name that is used to redirect HTTP requests to HTTPS. Default is "", which indicates to use the same host.
```

```
    SSLHostFunc: nil, // SSLHostFunc is a function pointer, the return value of
 the function is the host name that has same functionality as `SSHost`. Default
is nil. If SSLHostFunc is nil, the `SSLHost` option will be used.
    SSLProxyHeaders: map[string]string{"X-Forwarded-Proto": "https"}, // SSLProx
yHeaders is set of header keys with associated values that would indicate a vali
d HTTPS request. Useful when using Nginx: `map[string]string{"X-Forwarded-Proto"
: "https"}`. Default is blank map.
    STSSeconds: 31536000, // STSSeconds is the max-age of the Strict-Transport-S
ecurity header. Default is 0, which would NOT include the header.
    STSIncludeSubdomains: true, // If STSIncludeSubdomains is set to true, the `
includeSubdomains` will be appended to the Strict-Transport-Security header. Def
ault is false.
    STSPreload: true, // If STSPreload is set to true, the `preload` flag will b
e appended to the Strict-Transport-Security header. Default is false.
    ForceSTSHeader: false, // STS header is only included when the connection is
HTTPS. If you want to force it to always be added, set to true. `IsDevelopment`
 still overrides this. Default is false.
    FrameDeny: true, // If FrameDeny is set to true, adds the X-Frame-Options he
ader with the value of `DENY`. Default is false.
    CustomFrameOptionsValue: "SAMEORIGIN", // CustomFrameOptionsValue allows the
X-Frame-Options header value to be set with a custom value. This overrides the F
rameDeny option. Default is "".
    ContentTypeNosniff: true, // If ContentTypeNosniff is true, adds the X-Conte
nt-Type-Options header with the value `nosniff`. Default is false.
    BrowserXssFilter: true, // If BrowserXssFilter is true, adds the X-XSS-Prote
ction header with the value `1; mode=block`. Default is false.
    CustomBrowserXssValue: "1; report=https://example.com/xss-report", // Custom
BrowserXssValue allows the X-XSS-Protection header value to be set with a custom
value. This overrides the BrowserXssFilter option. Default is "".
    ContentSecurityPolicy: "default-src 'self'", // ContentSecurityPolicy allows
the Content-Security-Policy header value to be set with a custom value. Default
is "". Passing a template string will replace `$NONCE` with a dynamic nonce valu
e of 16 bytes for each request which can be later retrieved using the Nonce func
tion.
    PublicKey: `pin-sha256="base64+primary=="; pin-sha256="base64+backup=="; max
-age=5184000; includeSubdomains; report-uri="https://www.example.com/hpkp-repor
t"`, // PublicKey implements HPKP to prevent MITM attacks with forged certificat
es. Default is "".
    ReferrerPolicy: "same-origin", // ReferrerPolicy allows the Referrer-Policy
 header with the value to be set with a custom value. Default is "".
    FeaturePolicy: "vibrate 'none';", // FeaturePolicy allows the Feature-Policy
header with the value to be set with a custom value. Default is "".
    ExpectCTHeader: `enforce, max-age=30, report-uri="https://www.example.com/ct
-report"`,

    IsDevelopment: true, // This will cause the AllowedHosts, SSLRedirect, and S
```

```
TSSeconds/STSIncludeSubdomains options to be ignored during development. When de
ploying to production, be sure to set this to false.
})
// ...
```

默认选项

```
s := secure.New()

// Is the same as the default configuration options:

l := secure.New(secure.Options{
    AllowedHosts: []string,
    AllowedHostsAreRegex: false,
    HostsProxyHeaders: []string,
    SSLRedirect: false,
    SSLTemporaryRedirect: false,
    SSLHost: "",
    SSLProxyHeaders: map[string]string{},
    STSSeconds: 0,
    STSIncludeSubdomains: false,
    STSPreload: false,
    ForceSTSHeader: false,
    FrameDeny: false,
    CustomFrameOptionsValue: "",
    ContentTypeNosniff: false,
    BrowserXssFilter: false,
    ContentSecurityPolicy: "",
    PublicKey: "",
    ReferrerPolicy: "",
    FeaturePolicy: "",
    ExpectCTHeader: "",
    IsDevelopment: false,
})
```

另请注意，默认的错误主机处理程序将返回错误：

```
http.Error(w, "Bad Host", http.StatusInternalServerError)
```

将HTTP重定向到HTTPS

如果要将所有HTTP请求重定向到HTTPS，则可以使用以下示例。

```
// main.go
package main

import (
    "log"
    "net/http"

    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

var myHandler = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("hello world"))
})

func main() {
    secureMiddleware := secure.New(secure.Options{
        SSLRedirect: true,
        SSLHost:     "localhost:8443", // This is optional in production. The default behavior is to just redirect the request to the HTTPS protocol. Example: http://github.com/some_page would be redirected to https://github.com/some_page.
    })

    app := secureMiddleware.Handler(myHandler)

    // HTTP
    go func() {
        log.Fatal(http.ListenAndServe(":8080", app))
    }()

    // HTTPS
    // To generate a development cert and key, run the following from your *nix terminal:
    // go run $GOROOT/src/crypto/tls/generate_cert.go --host="localhost"
    log.Fatal(http.ListenAndServeTLS(":8443", "cert.pem", "key.pem", app))
}
```

STS标头仅在经过验证的HTTPS连接上发送（如果IsDevelopment为false，则发送）。SSLProxyHeaders如果您的应用程序位于代理之后，请确保设置该选项，以确保行为正确。如果所有HTTP和HTTPS请求都需要STS标头（不应该），则可以使用该ForceSTSHeader选项。请注意，如果IsDevelopment为true，即使ForceSTSHeader设置为true，它仍将禁用此标头。

要将该preload域包含在Chrome的预加载列表中，必须使用该标志。

整合范例

chi

```
// main.go
package main

import (
    "net/http"

    "github.com/pressly/chi"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })

    r := chi.NewRouter()

    r.Get("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("X-Frame-Options header is now `DENY`."))
    })
    r.Use(secureMiddleware.Handler)

    http.ListenAndServe("127.0.0.1:3000", r)
}
```

Echo

```
// main.go
package main

import (
    "net/http"

    "github.com/labstack/echo"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })

    e := echo.New()
```

```
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "X-Frame-Options header is now `DENY`.")
    })

    e.Use(echo.WrapMiddleware(secureMiddleware.Handler))
    e.Logger.Fatal(e.Start("127.0.0.1:3000"))
}
```

Gin

```
// main.go
package main

import (
    "github.com/gin-gonic/gin"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })
    secureFunc := func() gin.HandlerFunc {
        return func(c *gin.Context) {
            err := secureMiddleware.Process(c.Writer, c.Request)

            // If there was an error, do not continue.
            if err != nil {
                c.Abort()
                return
            }

            // Avoid header rewrite if response is a redirection.
            if status := c.Writer.Status(); status > 300 && status < 399 {
                c.Abort()
            }
        }
    }()

    router := gin.Default()
    router.Use(secureFunc)

    router.GET("/", func(c *gin.Context) {
        c.String(200, "X-Frame-Options header is now `DENY`.")
    })
```

```
    router.Run("127.0.0.1:3000")
}
```

Goji

```
// main.go
package main

import (
    "net/http"

    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
    "github.com/zenazn/goji"
    "github.com/zenazn/goji/web"
)

func main() {
    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })

    goji.Get("/", func(c web.C, w http.ResponseWriter, req *http.Request) {
        w.Write([]byte("X-Frame-Options header is now `DENY`."))
    })
    goji.Use(secureMiddleware.Handler)
    goji.Serve() // Defaults to ":8000".
}
```

Iris

```
//main.go
package main

import (
    "github.com/kataras/iris/v12"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    app := iris.New()

    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })
```

```
    app.Use(iris.FromStd(secureMiddleware.HandlerFuncWithNext))
    // Identical to:
    // app.Use(func(ctx iris.Context) {
    //     err := secureMiddleware.Process(ctx.ResponseWriter(), ctx.Request())
    //
    //     // If there was an error, do not continue.
    //     if err != nil {
    //         return
    //     }
    //
    //     ctx.Next()
    // })

    app.Get("/home", func(ctx iris.Context) {
        ctx.Writef("X-Frame-Options header is now `%s`.", "DENY")
    })

    app.Listen(":8080")
}
```

Mux

```
//main.go
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })

    r := mux.NewRouter()
    r.Use(secureMiddleware.Handler)
    http.Handle("/", r)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d", 8080), nil))
}
```

Negroni

请注意，此实现具有一个名为的特殊帮助程序功能HandlerFuncWithNext。

```
// main.go
package main

import (
    "net/http"

    "github.com/urfave/negroni"
    "github.com/unrolled/secure" // or "gopkg.in/unrolled/secure.v1"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
        w.Write([]byte("X-Frame-Options header is now `DENY`."))
    })

    secureMiddleware := secure.New(secure.Options{
        FrameDeny: true,
    })

    n := negroni.Classic()
    n.Use(negroni.HandlerFunc(secureMiddleware.HandlerFuncWithNext))
    n.UseHandler(mux)

    n.Run("127.0.0.1:3000")
}
```

判断切片数组是否存在值

```
func Contain(obj interface{}, target interface{}) (bool, error) {
    targetValue := reflect.ValueOf(target)
    switch reflect.TypeOf(target).Kind() {
    case reflect.Slice, reflect.Array:
        for i := 0; i < targetValue.Len(); i++ {
            if targetValue.Index(i).Interface() == obj {
                return true, nil
            }
        }
    case reflect.Map:
        if targetValue.MapIndex(reflect.ValueOf(obj)).IsValid() {
            return true, nil
        }
    }

    return false, errors.New("not in array")
}
```

敏感词过滤

一个简单的基于Golang的敏感词过滤算法

```
package main
import (
   "fmt"
 "unicode/utf8")
// 敏感词过滤
type Trie struct {
   child map[rune]*Trie
 word  string
}
// 插入
func (trie *Trie) insert(word string) *Trie {
   cur := trie
   for _, v := range []rune(word) {
      if _, ok := cur.child[v]; !ok {
         newTrie := NewTrie()
         cur.child[v] = newTrie
      }
      cur = cur.child[v]
   }
   cur.word = word
   return trie
}
// 过滤
func (trie *Trie) filerKeyWords(word string) string {
   cur := trie
   for i, v := range []rune(word) {
      if _, ok := cur.child[v]; ok {
         cur = cur.child[v]
         if cur.word != "" {
            word = replaceStr(word, "*", i-utf8.RuneCountInString(cur.word)+1,
 i)
            cur = trie
         }
      } else {
         cur = trie
      }
   }
   return word
}
func replaceStr(word string, replace string, left, right int) string {
```

```
    str := ""
 for i, v := range []rune(word) {
      if i >= left && i <= right {
          str += replace
      } else {
          str += string(v)
      }
    }
    return str
}
func NewTrie() *Trie {
    return &Trie{
       word:  "",
 child: make(map[rune]*Trie, 0),
 }
}
func main() {
    trie := NewTrie()
    trie.insert("sb").insert("狗日").insert("cnm").insert("狗日的")
    fmt.Println(trie.filerKeyWords("狗日，你就是个狗日的，我要cnm，你个sb，嘿嘿"
))
}
```

转自：https://segmentfault.com/a/1190000037477984

文件流下载

简单的使用

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
    "os"
)

func main() {
    http.HandleFunc("/", downloadHandler) //   设置访问路由
    http.ListenAndServe(":8080", nil)
}
func downloadHandler(w http.ResponseWriter, r *http.Request) {
    fileName := "ceshi.png"  //filename  文件名
    path := "./data/images/" //文件存放目录防止用户下载其他目录文件
    file, err := os.Open(path + fileName)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    content, err := ioutil.ReadAll(file)
    fileNames := url.QueryEscape(fileName) // 防止中文乱码
    w.Header().Add("Content-Type", "application/octet-stream")
    w.Header().Add("Content-Disposition", "attachment; filename=\""+fileNames+
"\"")
    if err != nil {
        fmt.Println("Read File Err:", err.Error())
    } else {
        w.Write(content)
    }
}
```

简单封装

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path"
    "strconv"
    "strings"
)

func DownLoad(w http.ResponseWriter, r *http.Request) {
    DownLoadFileHandler(w, r, "./topgoer.rar", "./")
}
func main() {
    http.HandleFunc("/", DownLoad)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        fmt.Println("HTTP server failed,err:", err)
        return
    }
}
func DownLoadFileHandler(w http.ResponseWriter, req *http.Request, fpath string,
resPrefix string) {
    defer req.Body.Close()
    err := req.ParseForm()
    //request header
    ranStr := req.Header.Get("Range")
    fmt.Println("Range: " + ranStr)
    //    w.Header().Set("Access-Control-Allow-Origin", "*")
    if err != nil {
        w.Header().Set("Content-Type", " text/plain; charset=UTF-8")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte(err.Error()))
        return
    }
    if fpath == "" || !strings.HasPrefix(fpath, resPrefix) {
        w.Header().Set("Content-Type", " text/plain; charset=UTF-8")
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("No file can be downloaded."))
        return
    }
    file, err := os.Open(fpath)
    if err != nil {
```

```
        w.Header().Set("Content-Type", " text/plain; charset=UTF-8")
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(err.Error()))
        return
    }
    defer file.Close()
    //设置下载完成以后的名称
    aliasName := path.Base(fpath)
    w.Header().Set("Content-Disposition", "attachment; filename="+aliasName)
    //设置下载的偏移量
    fstat, _ := file.Stat()
    fsize := fstat.Size()
    var spos int64
    if ranStr != "" {
        rs := strings.Split(strings.TrimPrefix(ranStr, "bytes="), ",")[0]
        sePos := strings.Split(rs, "-")
        spStr := sePos[0]
        spos, _ = strconv.ParseInt(spStr, 0, 64)
        file.Seek(spos, 0)
    }
    fmt.Println(spos)
    w.Header().Set("Content-Range", fmt.Sprintf("bytes %d-%d/%d", spos, fsize-1,
fsize))
    w.Header().Set("Content-Length", fmt.Sprintf("%d", fsize-spos))
    //    w.WriteHeader(http.StatusPartialContent)
    if spos == 0 {
        w.WriteHeader(http.StatusOK)
    } else {
        w.WriteHeader(http.StatusPartialContent)
    }
    wcount, err := io.Copy(w, file)
    if err != nil {
        fmt.Println("io.Copy: ", err.Error())
    }
    if spos+wcount == fsize {
        fmt.Println("remove from cache info.")
        fmt.Println("delete finished file." + fpath)
        file.Close()
        err := os.Remove(fpath)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Viper使用

什么是 Viper?

Viper 是适用于 Go 应用程序（包括 Twelve-Factor App）的完整配置解决方案。它被设计为在应用程序中工作，并且可以处理所有类型的配置需求和格式。它支持

- 设置默认值
- 可以读取 JSON，TOML，YAML，HCL，envfile 和 Java properties 格式的配置文件
- 实时监控和重新读取配置文件（可选）
- 读取环境变量中的配置信息
- 读取远程配置系统（etcd 或 Consul）中的配置信息，并监控配置信息发生改变
- 读取命令行参数中的配置信息
- 读取 buffer 中的配置信息
- 显式设置配置项

可以将 Viper 视为满足您所有应用程序配置需求的注册表。

为什么使用 Viper?

在构建现代应用程序时，您无需担心配置文件格式；您想专注于构建出色的软件。Viper 的出现就是为了在这方面给您提供帮助。

Viper 为您执行以下操作：

1. 查找，加载和反序列化 JSON，TOML，YAML，HCL，INI，envfile 或 Java properties 格式的配置文件。
2. 提供一种机制来为您的不同配置选项设置默认值。
3. 提供一种机制来通过命令行参数覆盖指定的选项的值。
4. 提供别名系统，以在不会破坏现有代码的情况下轻松重命名参数。
5. 用户提供了与默认值相同的命令行或配置文件时，可以容易地于区分它们的区别。

Viper 使用以下优先顺序。每个项目优先于其下面的项目：

- 显式调用 Set 方法设置值
- flag（命令行参数）
- env（环境变量）
- config（配置文件）
- key/value 存储

- 默认值

重要：Viper 配置项的 Key 不区分大小写。正在讨论是否设置为可选项。

怎么将配置项写入 Viper？

安装

```
go get github.com/spf13/viper
```

建立默认值

一个好的配置系统应该支持默认值。默认值对于 Key 不是必须的，但是如果未通过配置文件，环境变量，远程配置或标志（flag）设置 Key 的值，那么 Key 的默认值很有用。

示例：

```
viper.SetDefault("ContentDir", "content")
viper.SetDefault("LayoutDir", "layouts")
viper.SetDefault("Taxonomies", map[string]string{"tag": "tags", "category": "cat
egor
```

读取配置文件

Viper 需要最少的配置，以便它知道在哪里寻找配置文件。Viper 支持JSON，TOML，YAML，HCL，INI，envfile 和 Java Properties 格式的文件。Viper 可以搜索多个路径，但是当前单个 Viper 实例仅支持单个配置文件。Viper 不会默认使用任何配置搜索路径，而会将默认决定留给应用程序。

下面是如何使用 Viper 搜索和读取配置文件的示例。不需要任何特定路径，但至少需要提供一个配置文件的预期路径（见代码 3-5 行）。

```
viper.SetConfigName("config") // name of config file (without extension)
viper.SetConfigType("yaml") // REQUIRED if the config file does not have the ext
ension in the name
viper.AddConfigPath("/etc/appname/")   // path to look for the config file in
viper.AddConfigPath("$HOME/.appname")  // call multiple times to add many search
paths
viper.AddConfigPath(".")               // optionally look for config in the work
ing directory
err := viper.ReadInConfig() // Find and read the config file
if err != nil { // Handle errors reading the config file
```

```
  panic(fmt.Errorf("Fatal error config file: %s \n", err))
}
```

您可以处理未找到配置文件的特定情况，如下所示：

```
if err := viper.ReadInConfig(); err != nil {
    if _, ok := err.(viper.ConfigFileNotFoundError); ok {
        // Config file not found; ignore error if desired
    } else {
        // Config file was found but another error was produced
    }
}

// Config file found and successfully parsed
```

注意 [自 1.6]：您也可以有一个没有扩展名的文件， 并以编程方式指定格式。对于位于用户 $HOME 目录中的配置文件，没有任何扩展名，如 .bashrc

写入配置文件

从配置文件中读取文件很有用，但有时您希望存储运行时所做的所有修改。为此，有一堆命令可用，每个命令都有自己的用途：

- WriteConfig - 将当前 viper 配置写入预定义路径并覆盖（如果存在）。如果没有预定义的路径，则返回错误。
- SafeWriteConfig - 将当前 viper 配置写入预定义路径。如果没有预定义的路径，则返回错误。如果存在，不会覆盖当前配置文件。
- WriteConfigAs - 将当前 viper 配置写入给定的文件路径。将覆盖给定的文件（如果存在）。
- SafeWriteConfigAs - 将当前 viper 配置写入给定的文件路径。如果存在，不会覆盖给定文件。

根据经验，所有标有 safe 标记的方法都不会覆盖任何文件，而是直接创建（如果不存在），而默认行为是创建或截断。

一个小示例：

```
viper.WriteConfig() // writes current config to predefined path set by 'viper.Ad
dConfigPath()' and 'viper.SetConfigName'
viper.SafeWriteConfig()
viper.WriteConfigAs("/path/to/my/.config")
viper.SafeWriteConfigAs("/path/to/my/.config") // will error since it has alread
```

```
y been written
viper.SafeWriteConfigAs("/path/to/my/.other_config")
```

监控和重新读取配置文件

Viper 支持在运行时让应用程序实时读取配置文件的能力。

需要重新启动服务器才能使配置生效的日子已经一去不复返了，viper 支持的应用程序可以在运行时读取对配置文件的更新，并且不会错过任何更新。

只需告诉 viper 实例 watchConfig。您可以为 Viper 提供一个回调函数，在每次发生更改时运行的该函数。

请确保在调用 WatchConfig() 之前添加了所有配置路径

```
viper.WatchConfig()
viper.OnConfigChange(func(e fsnotify.Event) {
  fmt.Println("Config file changed:", e.Name)
})
```

从 io.Reader 读取配置

Viper 预定义许多配置源（如文件、环境变量、命令行参数和远程 K/V 存储，但您不受他们的约束。您还可以实现自己所需的配置源，并提供给 viper。

```
viper.SetConfigType("yaml") // or viper.SetConfigType("YAML")

// any approach to require this configuration into your program.
var yamlExample = []byte(`
Hacker: true
name: steve
hobbies:
- skateboarding
- snowboarding
- go
clothing:
  jacket: leather
  trousers: denim
age: 35
eyes : brown
beard: true
`)

viper.ReadConfig(bytes.NewBuffer(yamlExample))
```

```
viper.Get("name") // this would be "steve"
```

覆盖设置

这些可能是来自命令行参数，也可以来自您自己的应用程序逻辑。

```
viper.Set("Verbose", true)
viper.Set("LogFile", LogFile)
```

注册和使用别名

别名允许由多个键引用单个值

```
viper.RegisterAlias("loud", "Verbose")

viper.Set("verbose", true) // same result as next line
viper.Set("loud", true)   // same result as prior line

viper.GetBool("loud") // true
viper.GetBool("verbose") // true
```

使用环境变量

Viper 完全支持环境变量。这使 Twelve-Factor App 开箱即用。有五种方法可以帮助使用 ENV：

- AutomaticEnv()
- BindEnv(string...) : error
- SetEnvPrefix(string)
- SetEnvKeyReplacer(string...) *strings.Replacer
- AllowEmptyEnv(bool)

使用 ENV 变量时，必须认识到 Viper 将 ENV 变量视为对大小敏感。

Viper 提供了一种机制，用于尝试确保 ENV 变量是唯一的。通过使用 SetEnvPrefix，您可以告诉 Viper 在从环境变量读取时使用前缀。BindEnv 和AutomaticEnv 都将使用前缀。

BindEnv 采用一个或多个参数。第一个参数是键名称，其余参数是要绑定到此键的环境变量的名称。如果提供了多个，它们将按指定顺序优先。环境变量的名称是大小写敏感。如果未提供 ENV 变量名称，则 Viper 将自动假定 ENV 变量与以下格式匹配：前缀 + “_” + 所有 CAPS

中的键名称。当您显式提供 ENV 变量名称（第二个参数）时，它不会自动添加前缀。例如，如果第二个参数为”id”，Viper 将查找 ENV 变量”ID”。

使用 ENV 变量时，需要注意的一个重要问题是每次访问该值时都会重新读取该值。调用 BindEnv 时，viper 不会固定该值。

AutomaticEnv 是一个强大的帮助器，尤其是当与SerenvPrefix 结合。调用时，viper 将会在发出 viper.Get 请求时，随时检查环境变量。它将应用以下规则。如果使用 EnvPrefix 设置了前缀，它将检查一个环境变量的名称是否与键匹配。

SetEnvKeyReplacer 允许您使用 strings.Replacer 对象将 Env 键在一定程度上重写。如果您想要使用 - 或者其它符号在 Get() 调用中，但希望环境变量使用 _ 分隔符，这非常有用。使用它的示例可以在 viper_test.go 中找到。

或者，您也可以将 EnvKeyReplacer 与 NewWithOptions 工厂函数一起使用。与 SetEnvKeyReplacer 不同，它接受 StringReplacer 接口，允许您编写自定义字符串替换逻辑。

默认情况下，空环境变量被视为未设置，并将回退到下一个配置源。若要将空环境变量视为已设置，请使用 AllowEmptyEnv 方法。

环境变量-示例代码：

```
SetEnvPrefix("spf") // will be uppercased automatically
BindEnv("id")

os.Setenv("SPF_ID", "13") // typically done outside of the app

id := Get("id") // 13
```

使用 Flags

Viper 能够绑定到 flags。具体来说，viper 支持 Cobra 库中使用的 Pflags。

与 BindEnv 一样，在调用绑定方法时，不会设置该值，而是在访问绑定方法时设置该值。这意味着您可以尽早绑定，即使在 init() 函数中。

对于单个 Flag，BindPFlag() 方法提供此功能。

例如：

```
serverCmd.Flags().Int("port", 1138, "Port to run Application server on")
viper.BindPFlag("port", serverCmd.Flags().Lookup("port"))
```

您还可以绑定一组现有的 pflags（pflag.FlagSet）：

例如：

```
pflag.Int("flagname", 1234, "help message for flagname")

pflag.Parse()
viper.BindPFlags(pflag.CommandLine)

i := viper.GetInt("flagname") // retrieve values from viper instead of pflag
```

在 Viper 中使用 pflag 并不阻碍其他包中使用标准库中的 flag 包。pflag 包可以通过导入这些 flags 来处理为 flag 包定义的 flags。这是通过调用一个pflag 包提供的便利函数 AddGoFlagSet() 实现的。

例如：

```
package main

import (
  "flag"
  "github.com/spf13/pflag"
)

func main() {

  // using standard library "flag" package
  flag.Int("flagname", 1234, "help message for flagname")

  pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
  pflag.Parse()
  viper.BindPFlags(pflag.CommandLine)

  i := viper.GetInt("flagname") // retrieve value from viper

  ...
}
```

flag 接口

如果您不使用 Pflags，Viper 提供两个 Go 接口来绑定其他 flag 系统。

FlagValue 表示单个 flag。这是一个说明如何实现此接口的非常简单的示例：

```
type myFlag struct {}
func (f myFlag) HasChanged() bool { return false }
```

```
func (f myFlag) Name() string { return "my-flag-name" }
func (f myFlag) ValueString() string { return "my-flag-value" }
func (f myFlag) ValueType() string { return "string" }
```

一旦您的 flag 实现此接口，您只需告诉 Viper 将其绑定：

```
viper.BindFlagValue("my-flag-name", myFlag{})
```

FlagValueSet 表示一组 flags。这是一个说明如何实现此接口的非常简单的示例：

```
type myFlagSet struct {
  flags []myFlag
}

func (f myFlagSet) VisitAll(fn func(FlagValue)) {
  for _, flag := range flags {
    fn(flag)
  }
}
```

一旦您的 flag 集合实现此接口，您只需告诉 Viper 绑定它：

```
fSet := myFlagSet{
  flags: []myFlag{myFlag{}, myFlag{}},
}
viper.BindFlagValues("my-flags", fSet)
```

远程 Key/Value 存储支持

若要在 Viper 中启用远程支持，请对 viper/remote 包进行空白导入：

```
import _ "github.com/spf13/viper/remote"
```

Viper 将读取从 Key/Value 存储（例如 etcd 或 Consul ）中的路径检索到的配置字符串（如 JSON，TOML，YAML，HCL 或 envfile）。这些值优先级高于默认值，但会被从磁盘，命令行参数（flag）或环境变量检索的配置值覆盖。

Viper 使用 crypt 从 K / V 存储中检索配置，这意味着如果您具有正确的 gpg 密钥，您可以将配置值加密后存储，并可以自动将其解密。加密是可选的。

您可以将远程配置与本地配置结合使用，也可以独立使用。

crypt 有一个命令行帮助程序，您可以用来将配置放入 K / V 存储中。crypt 默认使用在 http://127.0.0.1:4001 上的 etcd。

```
$ go get github.com/bketelsen/crypt/bin/crypt
$ crypt set -plaintext /config/hugo.json /Users/hugo/settings/config.json
```

确认已设置值：

```
$ crypt get -plaintext /config/hugo.json
```

有关如何设置加密值或如何使用 Consul 的示例，请参见 crypt 文档。
https://github.com/bketelsen/crypt

远程 Key/Value 存储示例 - 未加密

etcd

```
viper.AddRemoteProvider("etcd", "http://127.0.0.1:4001","/config/hugo.json")
viper.SetConfigType("json") // because there is no file extension in a stream of bytes, supported extensions are "json", "toml", "yaml", "yml", "properties", "props", "prop", "env", "dotenv"
err := viper.ReadRemoteConfig()
```

Consul
您需要使用具有所需配置的 JSON 值设置 Consul 存储中的 key。例如，创建一个具有 JSON 值得 Consul key/value 存储的 key MY_CONSUL_KEY。

```
{
    "port": 8080,
    "hostname": "myhostname.com"
}
```

```
viper.AddRemoteProvider("consul", "localhost:8500", "MY_CONSUL_KEY")
viper.SetConfigType("json") // Need to explicitly set this to json
err := viper.ReadRemoteConfig()

fmt.Println(viper.Get("port")) // 8080
fmt.Println(viper.Get("hostname")) // myhostname.com
```

Firestore

```
viper.AddRemoteProvider("firestore", "google-cloud-project-id", "collection/document")
viper.SetConfigType("json") // Config's format: "json", "toml", "yaml", "yml"
err := viper.ReadRemoteConfig()
```

当然，您也可以使用 SecureRemoteProvider

远程 Key/Value 存储示例 - 加密

```
viper.AddSecureRemoteProvider("etcd","http://127.0.0.1:4001","/config/hugo.json","/etc/secrets/mykeyring.gpg")
viper.SetConfigType("json") // because there is no file extension in a stream of bytes,  supported extensions are "json", "toml", "yaml", "yml", "properties", "props", "prop", "env", "dotenv"
err := viper.ReadRemoteConfig()
```

监控 etcd 中的更改 - 未加密

```
// alternatively, you can create a new viper instance.
var runtime_viper = viper.New()

runtime_viper.AddRemoteProvider("etcd", "http://127.0.0.1:4001", "/config/hugo.yml")
runtime_viper.SetConfigType("yaml") // because there is no file extension in a stream of bytes, supported extensions are "json", "toml", "yaml", "yml", "properties", "props", "prop", "env", "dotenv"

// read from remote config the first time.
err := runtime_viper.ReadRemoteConfig()

// unmarshal config
runtime_viper.Unmarshal(&runtime_conf)

// open a goroutine to watch remote changes forever
go func(){
  for {
      time.Sleep(time.Second * 5) // delay after each request

      // currently, only tested with etcd support
      err := runtime_viper.WatchRemoteConfig()
      if err != nil {
          log.Errorf("unable to read remote config: %v", err)
          continue
      }
```

```
        // unmarshal new config into our runtime config struct. you can also use c
hannel
        // to implement a signal to notify the system of the changes
        runtime_viper.Unmarshal(&runtime_conf)
    }
}()
```

怎么在 Viper 中获取配置项？

在 Viper 中，有几种根据值的类型获取值的方法。存在以下功能和方法：

- Get(key string) : interface{}
- GetBool(key string) : bool
- GetFloat64(key string) : float64
- GetInt(key string) : int
- GetIntSlice(key string) : []int
- GetString(key string) : string
- GetStringMap(key string) : map[string]interface{}
- GetStringMapString(key string) : map[string]string
- GetStringSlice(key string) : []string
- GetTime(key string) : time.Time
- GetDuration(key string) : time.Duration
- IsSet(key string) : bool
- AllSettings() : map[string]interface{}

认识到的一件重要事情是，每个 Get 函数如果找不到值，它将返回零值。为了检查给定键是否存在，提供了 IsSet() 方法。

例如：

```
viper.GetString("logfile") // case-insensitive Setting & Getting
if viper.GetBool("verbose") {
    fmt.Println("verbose enabled")
}
```

访问嵌套键

访问器方法还接受深度嵌套键的格式化路径。例如，如果加载了以下JSON文件：

```
{
    "host": {
        "address": "localhost",
        "port": 5799
    },
    "datastore": {
        "metric": {
            "host": "127.0.0.1",
            "port": 3099
        },
        "warehouse": {
            "host": "198.0.0.1",
            "port": 2112
        }
    }
}
```

Viper 可以通过传递「.」分隔键的路径来访问嵌套字段：

```
GetString("datastore.metric.host") // (returns "127.0.0.1")
```

遵守上面建立的优先级规则；搜索路径将遍历其余配置注册表，直到找到为止。

例如，在给定此配置文件的情况下，datastore.metric.host 和 datastore.metric.port 均已定义（并且可以被覆盖）。如果另外在默认设置中定义了 datastore.metric.protocol，Viper 也会找到它。

但是，如果 datastore.metric 被直接赋值覆盖（通过 flag，环境变量，Set() 方法等），则 datastore.metric 的所有子键也都变为未定义状态，它们被较高的优先级配置遮蔽（shadowed）了。

Viper 可以使用路径中的数字访问数组索引。例如：

```
{
    "host": {
        "address": "localhost",
        "ports": [
            5799,
            6029
        ]
    },
    "datastore": {
        "metric": {
            "host": "127.0.0.1",
```

```
            "port": 3099
        },
        "warehouse": {
            "host": "198.0.0.1",
            "port": 2112
        }
    }
}

GetInt("host.ports.1") // returns 6029
```

最后，如果存在与分隔的键路径匹配的键，则将返回其值。例如

```
{
    "datastore.metric.host": "0.0.0.0",
    "host": {
        "address": "localhost",
        "port": 5799
    },
    "datastore": {
        "metric": {
            "host": "127.0.0.1",
            "port": 3099
        },
        "warehouse": {
            "host": "198.0.0.1",
            "port": 2112
        }
    }
}

GetString("datastore.metric.host") // returns "0.0.0.0"
```

提取子树

在开发可重用模块时，提取配置的子集并传递给模块通常很有用。这样，模块可以实例化一次，就获取到不同的配置。

例如，应用程序可能出于不同的目的使用多个不同的缓存存储：

```
cache:
  cache1:
    max-items: 100
    item-size: 64
  cache2:
```

```
    max-items: 200
    item-size: 80
```

我们可以将缓存名称传递给模块（例如 NewCache("缓存1")，但访问配置键需要奇怪的串联，并且与全局配置的分离更少。

因此，与其这样做，我们不要将 Viper 实例传递给表示配置子集的构造函数：

```
cache1Config := viper.Sub("cache.cache1")
if cache1Config == nil { // Sub returns nil if the key cannot be found
    panic("cache configuration not found")
}

cache1 := NewCache(cache1Config)
```

注意：始终检查 Sub 的返回值。如果找不到 Key，则返回 nil。

在内部，NewCache 函数可以直接处理 max-items 和 item-size 的键：

```
func NewCache(v *Viper) *Cache {
    return &Cache{
        MaxItems: v.GetInt("max-items"),
        ItemSize: v.GetInt("item-size"),
    }
}
```

生成的代码易于测试，因为它与主配置结构分离，并且更易于重用（出于同样的原因）。

反序列化

您还可以选择将所有值或特定值解析到 struct、map 和 etc。

有两种方法可以做到这一点：

- Unmarshal(rawVal interface{}) : error
- UnmarshalKey(key string, rawVal interface{}) : error

例如：

```
type config struct {
  Port int
  Name string
  PathMap string `mapstructure:"path_map"`
}
```

```
var C config

err := viper.Unmarshal(&C)
if err != nil {
  t.Fatalf("unable to decode into struct, %v", err)
}
```

如果要解析 Key 本身包含「.」（默认键分隔符）的配置，必须更改分隔符：

```
v := viper.NewWithOptions(viper.KeyDelimiter("::"))

v.SetDefault("chart::values", map[string]interface{}{
    "ingress": map[string]interface{}{
        "annotations": map[string]interface{}{
            "traefik.frontend.rule.type":                 "PathPrefix",
            "traefik.ingress.kubernetes.io/ssl-redirect": "true",
        },
    },
})

type config struct {
  Chart struct{
        Values map[string]interface{}
    }
}

var C config

v.Unmarshal(&C)
```

Viper 还支持解析到嵌入结构体：

```
/*
Example config:

module:
    enabled: true
    token: 89h3f98hbwf987h3f98wenf89ehf
*/
type config struct {
  Module struct {
    Enabled bool

    moduleConfig `mapstructure:",squash"`
```

```
  }
}

// moduleConfig could be in a module specific package
type moduleConfig struct {
  Token string
}

var C config

err := viper.Unmarshal(&C)
if err != nil {
  t.Fatalf("unable to decode into struct, %v", err)
}
```

Viper 在内部使用 github.com/mitchellh/mapstructure 解析值，默认情况下使用 mapstructure tag。

序列化为字符串

您可能需要将 viper 中保存的所有设置序列化到字符串中，而不是将它们写入文件。您可以将您最喜爱的格式的序列化程序与 AllSettings() 返回的配置一起使用。

```
import (
    yaml "gopkg.in/yaml.v2"
    // ...
)

func yamlStringSettings() string {
    c := viper.AllSettings()
    bs, err := yaml.Marshal(c)
    if err != nil {
        log.Fatalf("unable to marshal config to YAML: %v", err)
    }
    return string(bs)
}
```

使用单个 Viper 实例，还是使用多个 Viper 实例？

Viper 可以开箱即用。无需配置或初始化，就可以使用 Viper。由于大多数应用程序都希望使用单个中央存储库进行配置，因此 viper 包提供了此功能。它类似于单例模式。

在上面的所有示例中，他们都以单例模式风格演示了使用 Viper 的使用方法。

使用多个 Viper 实例

您还可以创建许多不同的 Viper 实例，供应用程序使用。每个都有其独特的配置和值集。每个都可以从不同的配置文件、Key/Value 存储等读取。Viper 包支持的所有函数都镜像为 Viper 上的方法。

例如：

```
x := viper.New()
y := viper.New()

x.SetDefault("ContentDir", "content")
y.SetDefault("ContentDir", "foobar")

//...
```

当使用多个 Viper 时，由用户管理不同的 Viper。

使用 Viper 读取配置文件的模拟示例

使用 Viper 读取配置文件的模拟示例

```
.
├── configs
│   └── config.yaml
├── go.mod
├── go.sum
└── main.go
```

配置文件：
configs/config.yaml

```
Server:
  RunMode: debug
  HttpPort: 8080
  ReadTimeout: 60
  WriteTimeout: 60
```

使用 Viper 读取配置文件中的内容，并解码到 struct 中：
main.go

```
type ServerSetting struct {
  RunMode      string
```

```
  HttpPort     string
  ReadTimeout  time.Duration
  WriteTimeout time.Duration
}

var server ServerSetting

func main() {
  // 实例化 viper 实例
  vp := viper.New()
  // 设置配置文件名称
  vp.SetConfigName("config")
  // 设置配置文件类型
  vp.SetConfigType("yaml")
  // 添加配置文件路径
  vp.AddConfigPath("configs/")
  // 读取配置文件内容
  err := vp.ReadInConfig()
  if err != nil {
    panic(fmt.Errorf("Fatal error config file: %s\n", err))
  }
  //  解码 key 到 struct
  err = vp.UnmarshalKey("Server", &server)
  if err != nil {
    panic(fmt.Errorf("unable to decode into struct, %v", err))
  }
  // 打印
  fmt.Printf("Server: %+v\n", server)
}
```

总结

本文是 Viper 开源库的 README 的中文翻译，文章内容介绍了什么是 Viper，Viper 包含哪些功能和 Viper 管理配置信息的不同方式的使用方法，以及不同方式之间的优先级顺序。翻译内容难免有不准确的地方，建议对照英文原稿阅读。文章结尾，还给出了一个使用 Viper 读取配置文件的模拟示例。截止发稿，Viper 的最新版本为 v1.7.1，并且作者目前正在收集使用反馈，为开发 v2.0 做准备。

转自:微信公众号：Golang语言开发栈

Gopsutil

相信不少 Python 开发者都知道一个开源库 —— psutil，它是一个跨平台的库，提供了便利的获取系统信息的方法，包括 CPU、内存、硬盘等，以及与进程相关的操作，包括进程列表和运行信息等，另外，psutil 还提供了函数形式的命令行工具，包括 ps, top 等。其强大的功能和优秀的跨平台特性，使得开发者可以很轻松地获取系统运行信息，实现系统监控，以及进行进程管理。而 Gopsutil，则是 psutil 的 Go 语言版本，为 Golang 带来了跨平台的系统和进程工具箱。

简介

Gopsutil，是 shirou 在 Github 上开源的 Golang 系统和进程工具库，项目位于 https://github.com/shirou/gopsutil， 目前版本为 v2.20.9。参照 psutil，gopsutil 实现了绝大部分的功能，并针对不同平台，实现了比较好的跨平台特性，同时，考虑到了 Go 语言自身的特性，实现了大量相关的概念和实体的数据结构，并对不同的功能分模块实现整合。Gopsutil 并未使用 CGo 进行 C 语言接口的直接转换，而是使用原生 Go 语言实现的。 另外，gopsutil 还提供了部分 psutil 没有的功能，提供了更大程度的便利，和更强大的功能。

v3迁移

从v3.20.10开始，gopsutil变为v3，从而破坏了后退兼容性。请参阅 https://github.com/shirou/gopsutil。

安装

Gopsutil 要求 Go 1.7 或以上，使用 go get 安装：

```
go get github.com/shirou/gopsutil
```

Gopsutil 实现了对多个操作系统平台的兼容支持，包括：

- FreeBSD i386/amd64/arm
- Linux i386/amd64/arm (raspberry pi)
- Windows/amd64
- Darwin i386/amd64
- OpenBSD amd64
- Solaris amd64

并部分支持

- CPU on DragonFly BSD
- host on Linux RISC-V

示例

Gopsutil 按照功能，区分了子模块进行实现，模块包括：

- cpu：系统 CPU 信息
- disk：系统硬盘信息
- docker：Docker 容器相关的系统信息
- host：主机操作系统运行信息
- internal/common：共同的工具类接口
- load：负载统计信息
- mem：系统内存信息
- net：网络相关的系统信息
- process：进程工具箱
- winservices：Windows 系统服务相关信息

我们来看一个简单的例子：

```
package main

import (
    "fmt"

    "github.com/shirou/gopsutil/mem"
)

func main() {
    v, _ := mem.VirtualMemory()

    // almost every return value is a struct
    fmt.Printf("Total: %v, Free:%v, UsedPercent:%f%%\n", v.Total, v.Free, v.Used
Percent)

    // convert to JSON. String() is also implemented
    fmt.Println(v)
}
```

代码中使用了 gopsutil 的 mem 模块，通过 mem.VirtualMemory 接口，获取了当前系统的内存信息，并进行打印输出。输出结果如下：

```
Total: 3179569152, Free:284233728, UsedPercent:84.508194%
{"total":3179569152,"available":492572672,"used":2895335424,"usedPercent":84.508
19439828305, (snip...)}
```

Gopsutil 提供了描述进程的类型 Process，定义如下：

```
type Process struct {
    Pid int32 `json:"pid"`
    // contains filtered or unexported fields
}
```

可以通过 NewProcess 进行新进程的创建：

```
func NewProcess(pid int32) (*Process, error)
```

通过提供 pid 创建，返回 Process 指针和错误信息。

```
package main

import (
    "fmt"
    "github.com/shirou/gopsutil/process"
)

func main() {
    p, _ := process.NewProcess(100)
    fmt.Println(p)
}
```

转自：https://m.toutiao.com/is/e8BPgmf/