GO-INI中文文档

书栈(BookStack.CN)

目　录

致谢

当前文档 《GO-INI中文文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-08。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/go-ini-zh

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

简介

INI

本包提供了 Go 语言中读写 INI 文件的功能。

功能特性

- 支持覆盖加载多个数据源（`[]byte`、文件和 `io.ReadCloser`）
- 支持递归读取键值
- 支持读取父子分区
- 支持读取自增键名
- 支持读取多行的键值
- 支持大量辅助方法
- 支持在读取时直接转换为 Go 语言类型
- 支持读取和 写入 分区和键的注释
- 轻松操作分区、键值和注释
- 在保存文件时分区和键值会保持原有的顺序

获取帮助

- API 文档
- 创建工单

下载安装

下载安装

下载安装前必须应该正确安装 Go 语言 以及配置 `$GOPATH` 变量。

下载源码并编译

使用一个特定版本:

```
$ go get gopkg.in/ini.v1
```

使用最新版:

```
$ go get github.com/go-ini/ini
```

如需更新请添加 `-u` 选项。

测试安装

如果您想要在自己的机器上运行测试，请使用 `-t` 标记:

```
$ go get -t gopkg.in/ini.v1
```

如需更新请添加 `-u` 选项。

本地运行测试脚本

进入到源码目录后执行 `make test` 命令（根据需要替换导入路径）:

```
$ cd $GOPATH/src/gopkg.in/ini.v1
$ make test
go test -v -cover -race
=== RUN   Test_Version

   Get version ✔


1 total assertion

```

```
--- PASS: Test_Version (0.00s)
=== RUN   Test_isSlice

  Check if a string is in the slice ✔✔


3 total assertions

--- PASS: Test_isSlice (0.00s)
=== RUN   TestEmpty

...

--- PASS: Test_Duration (0.00s)
PASS
coverage: 94.3% of statements
ok      gopkg.in/ini.v1    1.121s
```

开始使用

开始使用

我们将通过一个非常简单的例子来了解如何使用。

首先，我们需要在任意目录创建两个文件（ `my.ini` 和 `main.go` ），在这里我们选择 `/tmp/ini` 目录。

```
$ mkdir -p /tmp/ini
$ cd /tmp/ini
$ touch my.ini main.go
$ tree .
.
├── main.go
└── my.ini

0 directories, 2 files
```

现在，我们编辑 `my.ini` 文件并输入以下内容（部分内容来自 *Grafana*）。

```
# possible values : production, development
app_mode = development

[paths]
# Path to where grafana can store temp files, sessions, and the sqlite3 db (if that is used)
data = /home/git/grafana

[server]
# Protocol (http or https)
protocol = http

# The http port  to use
http_port = 9999

# Redirect to correct domain if host header does not match domain
# Prevents DNS rebinding attacks
enforce_domain = true
```

很好，接下来我们需要编写 `main.go` 文件来操作刚才创建的配置文件。

```go
package main

import (
    "fmt"
    "os"

    "gopkg.in/ini.v1"
)

func main() {
    cfg, err := ini.Load("my.ini")
    if err != nil {
        fmt.Printf("Fail to read file: %v", err)
        os.Exit(1)
    }

    // 典型读取操作，默认分区可以使用空字符串表示
    fmt.Println("App Mode:", cfg.Section("").Key("app_mode").String())
    fmt.Println("Data Path:", cfg.Section("paths").Key("data").String())

    // 我们可以做一些候选值限制的操作
    fmt.Println("Server Protocol:",
        cfg.Section("server").Key("protocol").In("http", []string{"http", "https"}))
    // 如果读取的值不在候选列表内，则会回退使用提供的默认值
    fmt.Println("Email Protocol:",
        cfg.Section("server").Key("protocol").In("smtp", []string{"imap", "smtp"}))

    // 试一试自动类型转换
    fmt.Printf("Port Number: (%[1]T) %[1]d\n", cfg.Section("server").Key("http_port").MustInt(9999))
    fmt.Printf("Enforce Domain: (%[1]T) %[1]v\n", cfg.Section("server").Key("enforce_domain").MustBool(false))

    // 差不多了，修改某个值然后进行保存
    cfg.Section("").Key("app_mode").SetValue("production")
    cfg.SaveTo("my.ini.local")
}
```

运行程序，我们可以看下以下输出：

```
$ go run main.go
App Mode: development
Data Path: /home/git/grafana
Server Protocol: http
Email Protocol: smtp
Port Number: (int) 9999
Enforce Domain: (bool) true

$ cat my.ini.local
# possible values : production, development
app_mode = production

[paths]
# Path to where grafana can store temp files, sessions, and the sqlite3 db (if that is used)
data = /home/git/grafana
...
```

完美！这个例子很简单，展示的也只是极其小部分的功能，想要完全掌握还需要多读多看，毕竟学无止境嘛。

我应该如何...

- 从数据源加载
- 操作分区（Section）
- 操作键（Key）
- 操作键值（Value）
- 操作注释（Comment）

从数据源加载

从数据源加载

就像之前说的，从多个数据源加载配置是基本操作。

那么，到底什么是 数据源 呢？

一个 数据源 可以是 `[]byte` 类型的原始数据， `string` 类型的文件路径或 `io.ReadCloser` 。您可以加载 任意多个 数据源。如果您传递其它类型的数据源，则会直接返回错误。

```
cfg, err := ini.Load(
    []byte("raw data"), // 原始数据
    "filename",         // 文件路径
    ioutil.NopCloser(bytes.NewReader([]byte("some other data"))),
)
```

或者从一个空白的文件开始：

```
cfg := ini.Empty()
```

当您在一开始无法决定需要加载哪些数据源时，仍可以使用 `Append()` 在需要的时候加载它们。

```
err := cfg.Append("other file", []byte("other raw data"))
```

当您想要加载一系列文件，但是不能够确定其中哪些文件是不存在的，可以通过调用函数 `LooseLoad()` 来忽略它们。

```
cfg, err := ini.LooseLoad("filename", "filename_404")
```

更牛逼的是，当那些之前不存在的文件在重新调用 `Reload()` 方法的时候突然出现了，那么它们会被正常加载。

数据覆写

在加载多个数据源时，如果某一个键在一个或多个数据源中出现，则会出现数据覆写。该键从前一个数据源读取的值会被下一个数据源覆写。

举例来说，如果加载两个配置文件 `my.ini` 和 `my.ini.local` （ 开始使用 中的输入和输出文

件），`app_mode` 的值会是 `production` 而不是 `development`。

```
1.  cfg, err := ini.Load("my.ini", "my.ini.local")
2.  ...
3.
4.  cfg.Section("").Key("app_mode").String() // production
```

数据覆写只有在一种情况下不会触发，即使用 `ShadowLoad` 加载数据源。

保存配置

终于到了这个时刻，是时候保存一下配置了。

比较原始的做法是输出配置到某个文件：

```
1.  // ...
2.  err = cfg.SaveTo("my.ini")
3.  err = cfg.SaveToIndent("my.ini", "\t")
```

另一个比较高级的做法是写入到任何实现 `io.Writer` 接口的对象中：

```
1.  // ...
2.  cfg.WriteTo(writer)
3.  cfg.WriteToIndent(writer, "\t")
```

默认情况下，空格将被用于对齐键值之间的等号以美化输出结果，以下代码可以禁用该功能：

```
1.  ini.PrettyFormat = false
```

操作分区（Section）

操作分区（Section）

获取指定分区：

```
1.  sec, err := cfg.GetSection("section name")
```

如果您想要获取默认分区，则可以用空字符串代替分区名：

```
1.  sec, err := cfg.GetSection("")
```

相对应的，还可以使用 `ini.DEFAULT_SECTION` 来获取默认分区：

```
1.  sec, err := cfg.GetSection(ini.DEFAULT_SECTION)
```

当您非常确定某个分区是存在的，可以使用以下简便方法：

```
1.  sec := cfg.Section("section name")
```

如果不小心判断错了，要获取的分区其实是不存在的，那会发生什么呢？没事的，它会自动创建并返回一个对应的分区对象给您。

创建一个分区：

```
1.  err := cfg.NewSection("new section")
```

获取所有分区对象或名称：

```
1.  secs := cfg.Sections()
2.  names := cfg.SectionStrings()
```

读取父子分区

您可以在分区名称中使用 `.` 来表示两个或多个分区之间的父子关系。如果某个键在子分区中不存在，则会去它的父分区中再次寻找，直到没有父分区为止。

```
1.  NAME = ini
2.  VERSION = v1
```

```
IMPORT_PATH = gopkg.in/%(NAME)s.%(VERSION)s

[package]
CLONE_URL = https://%(IMPORT_PATH)s

[package.sub]
```

```
cfg.Section("package.sub").Key("CLONE_URL").String()    // https://gopkg.in/ini.v1
```

无法解析的分区

如果遇到一些比较特殊的分区，它们不包含常见的键值对，而是没有固定格式的纯文本，则可以使用 `LoadOptions.UnparsableSections` 进行处理：

```
cfg, err := ini.LoadSources(ini.LoadOptions{
    UnparseableSections: []string{"COMMENTS"},
}, `[COMMENTS]
<1><L.Slide#2> This slide has the fuel listed in the wrong units <e.1>`)

body := cfg.Section("COMMENTS").Body()

/* --- start ---
<1><L.Slide#2> This slide has the fuel listed in the wrong units <e.1>
------  end  --- */
```

操作键（Key）

操作键（Key）

获取某个分区下的键：

```
key, err := cfg.Section("").GetKey("key name")
```

和分区一样，您也可以直接获取键而忽略错误处理：

```
key := cfg.Section("").Key("key name")
```

判断某个键是否存在：

```
yes := cfg.Section("").HasKey("key name")
```

创建一个新的键：

```
err := cfg.Section("").NewKey("name", "value")
```

获取分区下的所有键或键名：

```
keys := cfg.Section("").Keys()
names := cfg.Section("").KeyStrings()
```

获取分区下的所有键值对的克隆：

```
hash := cfg.Section("").KeysHash()
```

忽略键名的大小写

有时候分区和键的名称大小写混合非常烦人，这个时候就可以通过 `InsensitiveLoad` 将所有分区和键名在读取里强制转换为小写：

```
cfg, err := ini.InsensitiveLoad("filename")
//...

// sec1 和 sec2 指向同一个分区对象
sec1, err := cfg.GetSection("Section")
```

```
sec2, err := cfg.GetSection("SecTIOn")

// key1 和 key2 指向同一个键对象
key1, err := sec1.GetKey("Key")
key2, err := sec2.GetKey("KeY")
```

类似 MySQL 配置中的布尔值键

MySQL 的配置文件中会出现没有具体值的布尔类型的键：

```
[mysqld]
...
skip-host-cache
skip-name-resolve
```

默认情况下这被认为是缺失值而无法完成解析，但可以通过高级的加载选项对它们进行处理：

```
cfg, err := ini.LoadSources(ini.LoadOptions{
    AllowBooleanKeys: true,
}, "my.cnf")
```

这些键的值永远为 `true`，且在保存到文件时也只会输出键名。

如果您想要通过程序来生成此类键，则可以使用 `NewBooleanKey`：

```
key, err := sec.NewBooleanKey("skip-host-cache")
```

同个键名包含多个值

你是否也曾被下面的配置文件所困扰？

```
[remote "origin"]
url = https://github.com/Antergone/test1.git
url = https://github.com/Antergone/test2.git
fetch = +refs/heads/*:refs/remotes/origin/*
```

没错！默认情况下，只有最后一次出现的值会被保存到 `url` 中，可我就是想要保留所有的值怎么办啊？不要紧，用 `ShadowLoad` 轻松解决你的烦恼：

```
cfg, err := ini.ShadowLoad(".gitconfig")
// ...
```

```
3.
4.  f.Section(`remote "origin"`).Key("url").String()
5.  // Result: https://github.com/Antergone/test1.git
6.
7.  f.Section(`remote "origin"`).Key("url").ValueWithShadows()
8.  // Result:  []string{
9.  //              "https://github.com/Antergone/test1.git",
10. //              "https://github.com/Antergone/test2.git",
11. //          }
```

读取自增键名

如果数据源中的键名为 `-` ，则认为该键使用了自增键名的特殊语法。计数器从 1 开始，并且分区之间是相互独立的。

```
1.  [features]
2.  -: Support read/write comments of keys and sections
3.  -: Support auto-increment of key names
4.  -: Support load multiple files to overwrite key values
```

```
1.  cfg.Section("features").KeyStrings()    // []{"#1", "#2", "#3"}
```

获取上级父分区下的所有键名

```
1.  cfg.Section("package.sub").ParentKeys() // ["CLONE_URL"]
```

操作键值（**Value**）

操作键值（Value）

获取一个类型为字符串（string）的值：

```
val := cfg.Section("").Key("key name").String()
```

获取值的同时通过自定义函数进行处理验证：

```
val := cfg.Section("").Key("key name").Validate(func(in string) string {
    if len(in) == 0 {
        return "default"
    }
    return in
})
```

如果您不需要任何对值的自动转变功能（例如递归读取），可以直接获取原值（这种方式性能最佳）：

```
val := cfg.Section("").Key("key name").Value()
```

判断某个原值是否存在：

```
yes := cfg.Section("").HasValue("test value")
```

获取其它类型的值：

```
// 布尔值的规则：
// true 当值为：1, t, T, TRUE, true, True, YES, yes, Yes, y, ON, on, On
// false 当值为：0, f, F, FALSE, false, False, NO, no, No, n, OFF, off, Off
v, err = cfg.Section("").Key("BOOL").Bool()
v, err = cfg.Section("").Key("FLOAT64").Float64()
v, err = cfg.Section("").Key("INT").Int()
v, err = cfg.Section("").Key("INT64").Int64()
v, err = cfg.Section("").Key("UINT").Uint()
v, err = cfg.Section("").Key("UINT64").Uint64()
v, err = cfg.Section("").Key("TIME").TimeFormat(time.RFC3339)
v, err = cfg.Section("").Key("TIME").Time() // RFC3339

```

```
v = cfg.Section("").Key("BOOL").MustBool()
v = cfg.Section("").Key("FLOAT64").MustFloat64()
v = cfg.Section("").Key("INT").MustInt()
v = cfg.Section("").Key("INT64").MustInt64()
v = cfg.Section("").Key("UINT").MustUint()
v = cfg.Section("").Key("UINT64").MustUint64()
v = cfg.Section("").Key("TIME").MustTimeFormat(time.RFC3339)
v = cfg.Section("").Key("TIME").MustTime() // RFC3339

// 由 Must 开头的方法名允许接收一个相同类型的参数来作为默认值，
// 当键不存在或者转换失败时，则会直接返回该默认值。
// 但是，MustString 方法必须传递一个默认值。

v = cfg.Section("").Key("String").MustString("default")
v = cfg.Section("").Key("BOOL").MustBool(true)
v = cfg.Section("").Key("FLOAT64").MustFloat64(1.25)
v = cfg.Section("").Key("INT").MustInt(10)
v = cfg.Section("").Key("INT64").MustInt64(99)
v = cfg.Section("").Key("UINT").MustUint(3)
v = cfg.Section("").Key("UINT64").MustUint64(6)
v = cfg.Section("").Key("TIME").MustTimeFormat(time.RFC3339, time.Now())
v = cfg.Section("").Key("TIME").MustTime(time.Now()) // RFC3339
```

如果我的值有好多行怎么办？

```
[advance]
ADDRESS = """404 road,
NotFound, State, 5000
Earth"""
```

嗯哼？小 case！

```
cfg.Section("advance").Key("ADDRESS").String()

/* --- start ---
404 road,
NotFound, State, 5000
Earth
------  end  --- */
```

赞爆了！那要是我属于一行的内容写不下想要写到第二行怎么办？

```
[advance]
two_lines = how about \
    continuation lines?
lots_of_lines = 1 \
    2 \
    3 \
    4
```

简直是小菜一碟！

```
cfg.Section("advance").Key("two_lines").String() // how about continuation lines?
cfg.Section("advance").Key("lots_of_lines").String() // 1 2 3 4
```

可是我有时候觉得两行连在一起特别没劲，怎么才能不自动连接两行呢？

```
cfg, err := ini.LoadSources(ini.LoadOptions{
    IgnoreContinuation: true,
}, "filename")
```

哇靠给力啊！

需要注意的是，值两侧的单引号会被自动剔除：

```
foo = "some value" // foo: some value
bar = 'some value' // bar: some value
```

有时您会获得像从 Crowdin 网站下载的文件那样具有特殊格式的值（值使用双引号括起来，内部的双引号被转义）：

```
create_repo="created repository <a href=\"%s\">%s</a>"
```

那么，怎么自动地将这类值进行处理呢？

```
cfg, err := ini.LoadSources(ini.LoadOptions{UnescapeValueDoubleQuotes: true}, "en-US.ini"))
cfg.Section("<name of your section>").Key("create_repo").String()
// You got: created repository <a href="%s">%s</a>
```

这就是全部了？哈哈，当然不是。

操作键值的辅助方法

获取键值时设定候选值：

```
v = cfg.Section("").Key("STRING").In("default", []string{"str", "arr", "types"})
v = cfg.Section("").Key("FLOAT64").InFloat64(1.1, []float64{1.25, 2.5, 3.75})
v = cfg.Section("").Key("INT").InInt(5, []int{10, 20, 30})
v = cfg.Section("").Key("INT64").InInt64(10, []int64{10, 20, 30})
v = cfg.Section("").Key("UINT").InUint(4, []int{3, 6, 9})
v = cfg.Section("").Key("UINT64").InUint64(8, []int64{3, 6, 9})
v = cfg.Section("").Key("TIME").InTimeFormat(time.RFC3339, time.Now(), []time.Time{time1, time2, time3})
v = cfg.Section("").Key("TIME").InTime(time.Now(), []time.Time{time1, time2, time3}) // RFC3339
```

如果获取到的值不是候选值的任意一个，则会返回默认值，而默认值不需要是候选值中的一员。

验证获取的值是否在指定范围内：

```
vals = cfg.Section("").Key("FLOAT64").RangeFloat64(0.0, 1.1, 2.2)
vals = cfg.Section("").Key("INT").RangeInt(0, 10, 20)
vals = cfg.Section("").Key("INT64").RangeInt64(0, 10, 20)
vals = cfg.Section("").Key("UINT").RangeUint(0, 3, 9)
vals = cfg.Section("").Key("UINT64").RangeUint64(0, 3, 9)
vals = cfg.Section("").Key("TIME").RangeTimeFormat(time.RFC3339, time.Now(), minTime, maxTime)
vals = cfg.Section("").Key("TIME").RangeTime(time.Now(), minTime, maxTime) // RFC3339
```

自动分割键值到切片（slice）

当存在无效输入时，使用零值代替：

```
// Input: 1.1, 2.2, 3.3, 4.4 -> [1.1 2.2 3.3 4.4]
// Input: how, 2.2, are, you -> [0.0 2.2 0.0 0.0]
vals = cfg.Section("").Key("STRINGS").Strings(",")
vals = cfg.Section("").Key("FLOAT64S").Float64s(",")
vals = cfg.Section("").Key("INTS").Ints(",")
vals = cfg.Section("").Key("INT64S").Int64s(",")
vals = cfg.Section("").Key("UINTS").Uints(",")
vals = cfg.Section("").Key("UINT64S").Uint64s(",")
vals = cfg.Section("").Key("TIMES").Times(",")
```

从结果切片中剔除无效输入：

```
// Input: 1.1, 2.2, 3.3, 4.4 -> [1.1 2.2 3.3 4.4]
// Input: how, 2.2, are, you -> [2.2]
vals = cfg.Section("").Key("FLOAT64S").ValidFloat64s(",")
vals = cfg.Section("").Key("INTS").ValidInts(",")
vals = cfg.Section("").Key("INT64S").ValidInt64s(",")
vals = cfg.Section("").Key("UINTS").ValidUints(",")
vals = cfg.Section("").Key("UINT64S").ValidUint64s(",")
vals = cfg.Section("").Key("TIMES").ValidTimes(",")
```

当存在无效输入时，直接返回错误：

```
// Input: 1.1, 2.2, 3.3, 4.4 -> [1.1 2.2 3.3 4.4]
// Input: how, 2.2, are, you -> error
vals = cfg.Section("").Key("FLOAT64S").StrictFloat64s(",")
vals = cfg.Section("").Key("INTS").StrictInts(",")
vals = cfg.Section("").Key("INT64S").StrictInt64s(",")
vals = cfg.Section("").Key("UINTS").StrictUints(",")
vals = cfg.Section("").Key("UINT64S").StrictUint64s(",")
vals = cfg.Section("").Key("TIMES").StrictTimes(",")
```

递归读取键值

在获取所有键值的过程中，特殊语法 `%(<name>)s` 会被应用，其中 `<name>` 可以是相同分区或者默认分区下的键名。字符串 `%(<name>)s` 会被相应的键值所替代，如果指定的键不存在，则会用空字符串替代。您可以最多使用 99 层的递归嵌套。

```
NAME = ini

[author]
NAME = Unknwon
GITHUB = https://github.com/%(NAME)s

[package]
FULL_NAME = github.com/go-ini/%(NAME)s
```

```
cfg.Section("author").Key("GITHUB").String()        //
https://github.com/Unknwon
cfg.Section("package").Key("FULL_NAME").String()    // github.com/go-ini/ini
```

Python 多行值

如果您刚将服务从 Python 迁移过来，可能会遇到一些使用旧语法的配置文件，别慌！

```
cfg, err := ini.LoadSources(ini.LoadOptions{
    AllowPythonMultilineValues: true,
}, []byte(`
[long]
long_rsa_private_key = -----BEGIN RSA PRIVATE KEY-----
   foo
   bar
   foobar
   barfoo
   -----END RSA PRIVATE KEY-----
`)

/*
-----BEGIN RSA PRIVATE KEY-----
foo
bar
foobar
barfoo
-----END RSA PRIVATE KEY-----
*/
```

操作注释（Comment）

操作注释（Comment）

下述几种情况的内容将被视为注释：

1. 所有以 `#` 或 `;` 开头的行
2. 所有在 `#` 或 `;` 之后的内容
3. 分区标签后的文字（即 `[分区名]` 之后的内容）

如果你希望使用包含 `#` 或 `;` 的值，请使用 `` ` `` 或 `"""` 进行包覆。

除此之外，您还可以通过 `LoadOptions` 完全忽略行内注释：

```
1.  cfg, err := ini.LoadSources(ini.LoadOptions{
2.      IgnoreInlineComment: true,
3.  }, "app.ini")
```

高级用法

结构体与分区双向映射

映射到结构

想要使用更加面向对象的方式玩转 INI 吗？好主意。

```
Name = Unknwon
age = 21
Male = true
Born = 1993-01-01T20:17:05Z

[Note]
Content = Hi is a good man!
Cities = HangZhou, Boston
```

```
type Note struct {
    Content string
    Cities  []string
}

type Person struct {
    Name string
    Age  int `ini:"age"`
    Male bool
    Born time.Time
    Note
    Created time.Time `ini:"-"`
}

func main() {
    cfg, err := ini.Load("path/to/ini")
    // ...
    p := new(Person)
    err = cfg.MapTo(p)
    // ...

    // 一切竟可以如此的简单。
    err = ini.MapTo(p, "path/to/ini")
    // ...

```

```
    // 嗯哼？只需要映射一个分区吗？
    n := new(Note)
    err = cfg.Section("Note").MapTo(n)
    // ...
}
```

结构的字段怎么设置默认值呢？很简单，只要在映射之前对指定字段进行赋值就可以了。如果键未找到或者类型错误，该值不会发生改变。

```
// ...
p := &Person{
    Name: "Joe",
}
// ...
```

这样玩 INI 真的好酷啊！然而，如果不能还给我原来的配置文件，有什么卵用？

从结构反射

可是，我有说不能吗？

```
type Embeded struct {
    Dates  []time.Time `delim:"|" comment:"Time data"`
    Places []string    `ini:"places,omitempty"`
    None   []int       `ini:",omitempty"`
}

type Author struct {
    Name      string `ini:"NAME"`
    Male      bool
    Age       int `comment:"Author's age"`
    GPA       float64
    NeverMind string `ini:"-"`
    *Embeded `comment:"Embeded section"`
}

func main() {
    a := &Author{"Unknwon", true, 21, 2.8, "",
        &Embeded{
            []time.Time{time.Now(), time.Now()},
            []string{"HangZhou", "Boston"},
            []int{},
```

```
        }}
    cfg := ini.Empty()
    err = ini.ReflectFrom(cfg, a)
    // ...
}
```

瞧瞧，奇迹发生了。

```
NAME = Unknwon
Male = true
; Author's age
Age = 21
GPA = 2.8

; Embeded section
[Embeded]
; Time data
Dates = 2015-08-07T22:14:22+08:00|2015-08-07T22:14:22+08:00
places = HangZhou,Boston
```

配合 ShadowLoad 进行映射

如果您希望配合 ShadowLoad 将某个分区映射到结构体，则需要指定 `allowshadow` 标签。

假设您有以下配置文件：

```
[IP]
value = 192.168.31.201
value = 192.168.31.211
value = 192.168.31.221
```

您应当通过如下方式定义对应的结构体：

```
type IP struct {
   Value    []string `ini:"value,omitempty,allowshadow"`
}
```

如果您不需要前两个标签规则，可以使用 `ini:",,allowshadow"` 进行简写。

映射/反射的其它说明

任何嵌入的结构都会被默认认作一个不同的分区，并且不会自动产生所谓的父子分区关联：

```
type Child struct {
    Age string
}

type Parent struct {
    Name string
    Child
}

type Config struct {
    City string
    Parent
}
```

示例配置文件：

```
City = Boston

[Parent]
Name = Unknwon

[Child]
Age = 21
```

很好，但是，我就是要嵌入结构也在同一个分区。好吧，你爹是李刚！

```
type Child struct {
    Age string
}

type Parent struct {
    Name string
    Child `ini:"Parent"`
}

type Config struct {
    City string
    Parent
}
```

示例配置文件：

```
City = Boston

[Parent]
Name = Unknwon
Age = 21
```

另请查看 自定义键名和键值映射器 的有关用法。

自定义键名和键值映射器

键名映射器（Name Mapper）

为了节省您的时间并简化代码，本库支持类型为 `NameMapper` 的名称映射器，该映射器负责结构字段名与分区名和键名之间的映射。

目前有 2 款内置的映射器：

- `AllCapsUnderscore`：该映射器将字段名转换至格式 `ALL_CAPS_UNDERSCORE` 后再去匹配分区名和键名。
- `TitleUnderscore`：该映射器将字段名转换至格式 `title_underscore` 后再去匹配分区名和键名。

使用方法：

```
type Info struct {
    PackageName string
}

func main() {
    err = ini.MapToWithMapper(&Info{}, ini.TitleUnderscore,
[]byte("package_name=ini"))
    // ...

    cfg, err := ini.Load([]byte("PACKAGE_NAME=ini"))
    // ...
    info := new(Info)
    cfg.NameMapper = ini.AllCapsUnderscore
    err = cfg.MapTo(info)
    // ...
}
```

使用函数 `ini.ReflectFromWithMapper` 时也可应用相同的规则。

键值映射器（Value Mapper）

值映射器允许使用一个自定义函数自动展开值的具体内容，例如在运行时获取环境变量：

```
type Env struct {
    Foo string `ini:"foo"`
```

```
}

func main() {
    cfg, err := ini.Load([]byte("[env]\nfoo = ${MY_VAR}\n")
    cfg.ValueMapper = os.ExpandEnv
    // ...
    env := &Env{}
    err = cfg.Section("env").MapTo(env)
}
```

本例中，`env.Foo` 将会是运行时所获取到环境变量 `MY_VAR` 的值。

常见问题

常见问题

字段 `BlockMode` 是什么？

默认情况下，本库会在您进行读写操作时采用锁机制来确保数据时间。但在某些情况下，您非常确定只进行读操作。此时，您可以通过设置 `cfg.BlockMode = false` 来将读操作提升大约 **50-70%** 的性能。

为什么要写另一个 INI 解析库？

许多人都在使用我的 goconfig 来完成对 INI 文件的操作，但我希望使用更加 Go 风格的代码。并且当您设置 `cfg.BlockMode = false` 时，会有大约 **10-30%** 的性能提升。

为了做出这些改变，我必须对 API 进行破坏，所以新开一个仓库是最安全的做法。除此之外，本库直接使用 `gopkg.in` 来进行版本化发布。（其实真相是导入路径更短了）