目　录

介绍
rxgo
goquery
air
quicktemplate
commonregex
gabs
jobrunner
mapstructure
cron
cli
negroni
fyne
casbin
twirp
rpcx
jsonrpc
rpc
xorm
sqlc
nutsdb
zerolog
zap
go-app
gron
plot
gentleman
gopsutil
validator
govaluate
jj

sjson
gjson
buntdb
go-cmp
jennifer
copier
mergo
wire
watermill
message-bus
gojsonq
dig
email
carbon
godotenv
logrus
log
viper
fsnotify
cast
cobra
go-ini
go-homedir
go-flags
flag

介绍

关于作者

本文档作者：大俊

大俊的博客：https://darjun.github.io

欢迎关注大俊的微信公众号【GoUpUp】，共同学习，一起进步~

rxgo

简介

ReactiveX，简称为 Rx，是一个异步编程的 API。与 **callback**（回调）、**promise**（JS 提供这种方式）和 **deferred**（Python 的 twisted 网络编程库就是使用这种方式）这些异步编程方式有所不同，Rx 是基于事件流的。这里的事件可以是系统中产生或变化的任何东西，在代码中我们一般用对象表示。在 Rx 中，事件流被称为 **Observable**（可观察的）。事件流需要被 **Observer**（观察者）处理才有意义。想象一下，我们日常作为一个 **Observer**，一个重要的工作就是观察 BUG 的事件流。每次发现一个 BUG，我们都需要去解决它。

Rx 仅仅只是一个 API 规范的定义。Rx 有多种编程语言实现，`RxJava/RxJS/Rx.NET/RxClojure/RxSwift`。RxGo 是 Rx 的 Go 语言实现。借助于 Go 语言简洁的语法和强大的并发支持（goroutine、channel），Rx 与 Go 语言的结合非常完美。

pipelines (官方博客：https://blog.golang.org/pipelines)是 Go 基础的并发编程模型。其中包含，fan-in——多个 goroutine 产生数据，一个goroutine 处理数据，fan-out——一个 goroutine 产生数据，多个 goroutine 处理数据，fan-inout——多个 goroutine 产生数据，多个 goroutine 处理数据。它们都是通过 channel 连接。RxGo 的实现就是基于 **pipelines** 的理念，并且提供了方便易用的包装和强大的扩展。

快速使用

本文代码使用 Go Modules。

创建目录并初始化：

```
$ mkdir rxgo && cd rxgo
$ go mod init github.com/darjun/go-daily-lib/rxgo
```

安装 `rxgo` 库：

```
$ go get -u github.com/reactivex/rxgo/v2
```

编码：

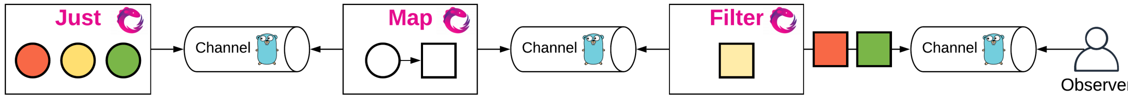
```
package main

import (
```

```
	"fmt"

	"github.com/reactivex/rxgo/v2"
)

func main() {
	observable := rxgo.Just(1, 2, 3, 4, 5)()
	ch := observable.Observe()
	for item := range ch {
		fmt.Println(item.V)
	}
}
```

使用 RxGo 的一般流程如下：

- 使用相关的 **Operator** 创建 **Observable**，**Operator** 就是用来创建 **Observable** 的。这些术语都比较难贴切地翻译，而且英文也很好懂，就不强行翻译了；
- 中间各个阶段可以使用过滤操作筛选出我们想要的数据，使用转换操作对数据进行转换；
- 调用 **Observable** 的 `Observe()` 方法，该方法返回一个 `<- chan rxgo.Item` 。然后 `for range` 遍历即可。

GitHub 上一张图很形象地描绘了这个过程：



- 首先使用 `Just` 创建一个仅有若干固定数据的 **Observable**；
- 使用 `Map()` 方法执行转换（将圆形转为方形）；
- 使用 `Filter()` 方法执行过滤（过滤掉黄色的方形）。

看懂了这张图片，就能了解 RxGo 工作的基本流程了。

上面是简单的示例，没有过滤、转换操作的使用。

运行：

```
$ go run main.go
1
2
3
4
5
```

关于上面的示例，需要注意：

`Just` 使用柯里化（currying）让它可以在第一个参数中接受多个数据，在第二个参数中接受多个选项定制行为。柯里化是函数化编程的思想，简单来说就是通过在函数中返回函数，以此来减少每个函数的参数个数。例如：

```
func add(value int) func (int) int {
  return func (a int) int {
    return value + a
  }
}

fmt.Prinlnt(add(5)(10)) // 15
```

由于 Go 不支持多个可变参数，`Just` 通过柯里化迂回地实现了这个功能：

```
// rxgo/factory.go
func Just(items ...interface{}) func(opts ...Option) Observable {
  return func(opts ...Option) Observable {
    return &ObservableImpl{
      iterable: newJustIterable(items...)(opts...),
    }
  }
}
```

实际上 `rxgo.Item` 还可以包含错误。所以在使用时，我们应该做一层判断：

```
func main() {
  observable := rxgo.Just(1, 2, errors.New("unknown"), 3, 4, 5)()
  ch := observable.Observe()
  for item := range ch {
    if item.Error() {
      fmt.Println("error:", item.E)
    } else {
      fmt.Println(item.V)
    }
  }
}
```

运行：

```
$ go run main.go
1
```

```
2
error: unknown
3
4
5
```

我们使用 `item.Error()` 检查是否出现错误。然后使用 `item.V` 访问数据，`item.E` 访问错误。

除了使用 `for range` 之外，我们还可以调用 **Observable** 的 `ForEach()` 方法来实现遍历。`ForEach()` 接受 3 个回调函数：

- `NextFunc`：类型为`func (v interface {})`，处理数据；
- `ErrFunc`：类型为`func (err error)`，处理错误；
- `CompletedFunc`：类型为`func ()`，**Observable** 完成时调用。

有点 `Promise` 那味了。使用 `ForEach()`，可以将上面的示例改写为：

```go
func main() {
  observable := rxgo.Just(1, 2, 3, 4, 5)()
  <-observable.ForEach(func(v interface{}) {
    fmt.Println("received:", v)
  }, func(err error) {
    fmt.Println("error:", err)
  }, func() {
    fmt.Println("completed")
  })
}
```

运行：

```
$ go run main.go
received: 1
received: 2
received: 3
received: 4
received: 5
completed
```

`ForEach()` 实际上是异步执行的，它返回一个接收通知的 channel。当 **Observable** 数据发送完毕时，该 channel 会关闭。所以如果要等待 `ForEach()` 执行完成，我们需要使用 `<-`。上面的示例中如果去掉 `<-`，可能就没有输出了，因为主 goroutine 结束了，整个程序就退出了。

创建 Observable

上面使用最简单的方式创建 **Observable**：直接调用 `Just()` 方法传入一系列数据。下面再介绍几种创建 **Observable** 的方式。

`Create`

传入一个 `[]rxgo.Producer` 的切片，其中 `rxgo.Producer` 的类型为 `func(ctx context.Context, next chan<- Item)` 。我们可以在代码中调用 `rxgo.Of(value)` 生成数据， `rxgo.Error(err)` 生成错误，然后发送到 `next` 通道中：

```
func main() {
  observable := rxgo.Create([]rxgo.Producer{func(ctx context.Context, next chan<- rxgo.Item) {
    next <- rxgo.Of(1)
    next <- rxgo.Of(2)
    next <- rxgo.Of(3)
    next <- rxgo.Error(errors.New("unknown"))
    next <- rxgo.Of(4)
    next <- rxgo.Of(5)
  }})

  ch := observable.Observe()
  for item := range ch {
    if item.Error() {
      fmt.Println("error:", item.E)
    } else {
      fmt.Println(item.V)
    }
  }
}
```

当然，分成两个 `rxgo.Producer` 也是一样的效果：

```
observable := rxgo.Create([]rxgo.Producer{func(ctx context.Context, next chan<- rxgo.Item) {
  next <- rxgo.Of(1)
  next <- rxgo.Of(2)
  next <- rxgo.Of(3)
  next <- rxgo.Error(errors.New("unknown"))
  }, func(ctx context.Context, next chan<- rxgo.Item) {
  next <- rxgo.Of(4)
  next <- rxgo.Of(5)
}})
```

FromChannel

`FromChannel` 可以直接从一个已存在的 `<-chan rxgo.Item` 对象中创建 **Observable**：

```
func main() {
    ch := make(chan rxgo.Item)
    go func() {
        for i := 1; i <= 5; i++ {
            ch <- rxgo.Of(i)
        }
        close(ch)
    }()

    observable := rxgo.FromChannel(ch)
    for item := range observable.Observe() {
        fmt.Println(item.V)
    }
}
```

注意：

通道需要手动调用 `close()` 关闭，上面 `Create()` 方法内部 `rxgo` 自动帮我们执行了这个步骤。

Interval

`Interval` 以传入的时间间隔生成一个无穷的数字序列，从 0 开始：

```
func main() {
  observable := rxgo.Interval(rxgo.WithDuration(5 * time.Second))
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面的程序启动后，第 5s 输出 0，第 10s 输出 1，...，而且不会停止。

我们可以用 `time.Ticker` 实现相同的功能：

```
func main() {
  t := time.NewTicker(5 * time.Second)

  var count int
  for range t.C {
```

```
    fmt.Println(count)
    count++
  }
}
```

`Range`

`Range` 可以生成一个范围内的数字：

```go
func main() {
  observable := rxgo.Range(0, 3)
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面代码依次输出 0，1，2，3。

`Repeat`

在已存在的 **Observable** 对象上调用 `Repeat` ，可以实现每隔指定时间，重复一次该序列，一共重复指定次数：

```go
func main() {
  observable := rxgo.Just(1, 2, 3)().Repeat(
    3, rxgo.WithDuration(1*time.Second),
  )
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

运行上面的代码，立即输出 1，2，3，然后等待 1s，又输出一次 1，2，3，然后又等待 1s，最后又输出一次 1，2，3。

`Start`

可以给 `Start` 方法传入 `[]rxgo.Supplier` 作为参数，它可以包含任意数量的 `rxgo.Supplier` 类型。`rxgo.Supplier` 的底层类型为：

```go
// rxgo/types.go
var Supplier func(ctx context.Context) rxgo.Item
```

Observable 内部会依次调用这些 `rxgo.Supplier` 生成 `rxgo.Item` ：

```
func Supplier1(ctx context.Context) rxgo.Item {
  return rxgo.Of(1)
}

func Supplier2(ctx context.Context) rxgo.Item {
  return rxgo.Of(2)
}

func Supplier3(ctx context.Context) rxgo.Item {
  return rxgo.Of(3)
}

func main() {
  observable := rxgo.Start([]rxgo.Supplier{Supplier1, Supplier2, Supplier3})
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

Observable 分类

根据数据在何处生成，**Observable** 被分为 **Hot** 和 **Cold** 两种类型（类比热启动和冷启动）。数据在其它地方生成的被成为 **Hot Observable**。相反，在 **Observable** 内部生成数据的就是 **Cold Observable**。

使用上面介绍的方法创建的实际上都是 **Hot Observable**。

```
func main() {
  ch := make(chan rxgo.Item)
  go func() {
    for i := 0; i < 3; i++ {
      ch <- rxgo.Of(i)
    }
    close(ch)
  }()

  observable := rxgo.FromChannel(ch)

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }

  for item := range observable.Observe() {
    fmt.Println(item.V)
```

```
    }
}
```

上面创建的是 **Hot Observable**。但是有个问题，第一次 `Observe()` 消耗了所有的数据，第二个就没有数据输出了。

而 **Cold Observable** 就不会有这个问题，因为它创建的流是独立于每个观察者的。即每次调用 `Observe()` 都创建一个新的 channel。我们使用 `Defer()` 方法创建 **Cold Observable**，它的参数与 `Create()` 方法一样。

```
func main() {
  observable := rxgo.Defer([]rxgo.Producer{func(_ context.Context, ch chan<- rxgo.Item) {
    for i := 0; i < 3; i++ {
      ch <- rxgo.Of(i)
    }
  }})

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

输出：

```
$ go run main.go
0
1
2
0
1
2
```

可连接的 Observable

可连接的（Connectable）**Observable** 对普通的 **Observable** 进行了一层组装。调用它的 `Observe()` 方法时并不会立刻产生数据。使用它，我们可以等所有的观察者都准备就绪了（即调用了 `Observe()` 方法）之后，再调用其 `Connect()` 方法开始生成数据。我们通过两个示例比较使用普通的 **Observable** 和可连接的 **Observable** 有何不同。

普通的：

```
func main() {
  ch := make(chan rxgo.Item)
  go func() {
    for i := 1; i <= 3; i++ {
      ch <- rxgo.Of(i)
    }
    close(ch)
  }()

  observable := rxgo.FromChannel(ch)

  observable.DoOnNext(func(i interface{}) {
    fmt.Printf("First observer: %d\n", i)
  })

  time.Sleep(3 * time.Second)
  fmt.Println("before subscribe second observer")

  observable.DoOnNext(func(i interface{}) {
    fmt.Printf("Second observer: %d\n", i)
  })

  time.Sleep(3 * time.Second)
}
```

上例中我们使用 `DoOnNext()` 方法来注册观察者。由于 `DoOnNext()` 方法是异步执行的，所以为了等待结果输出，在最后增加了一行 `time.Sleep` 。运行：

```
$ go run main.go
First observer: 1
First observer: 2
First observer: 3
before subscribe second observer
```

由输出可以看出，注册第一个观察者之后就开始产生数据了。

我们通过在创建 **Observable** 的方法中指定 `rxgo.WithPublishStrategy()` 选项就可以创建可连接的 **Observable**：

```
func main() {
  ch := make(chan rxgo.Item)
  go func() {
```

```
    for i := 1; i <= 3; i++ {
      ch <- rxgo.Of(i)
    }
    close(ch)
  }()

  observable := rxgo.FromChannel(ch, rxgo.WithPublishStrategy())

  observable.DoOnNext(func(i interface{}) {
    fmt.Printf("First observer: %d\n", i)
  })

  time.Sleep(3 * time.Second)
  fmt.Println("before subscribe second observer")

  observable.DoOnNext(func(i interface{}) {
    fmt.Printf("Second observer: %d\n", i)
  })

  observable.Connect(context.Background())
  time.Sleep(3 * time.Second)
}
```

运行输出：

```
$ go run main.go
before subscribe second observer
Second observer: 1
First observer: 1
First observer: 2
First observer: 3
Second observer: 2
Second observer: 3
```

上面是等两个观察者都注册之后，并且手动调用了 Observable 的 `Connect()` 方法才产生数据。而且可连接的 **Observable** 有一个特性：**它是冷启动的！！！**，即每个观察者都会收到一份相同的拷贝。

转换 Observable

rxgo 提供了很多转换函数，可以修改经过它的 `rxgo.Item` ，然后再发送给下一个阶段。

`Map`

`Map()` 方法简单修改它收到的 `rxgo.Item` 然后发送到下一个阶段（转换或过滤）。`Map()` 接受一个类型为 `func (context.Context, interface{}) (interface{}, error)` 的函数。第二个参数就是 `rxgo.Item` 中的数据，返回转换后的数据。如果出错，则返回错误。

```
func main() {
  observable := rxgo.Just(1, 2, 3)()

  observable = observable.Map(func(_ context.Context, i interface{}) (interface{}, error) {
    return i.(int)*2 + 1, nil
  }).Map(func(_ context.Context, i interface{}) (interface{}, error) {
    return i.(int)*3 + 2, nil
  })

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上例中每个数字经过两个 `Map`，第一个 `Map` 执行 `2 * i + 1`，第二个 `Map` 执行 `3 * i + 2`。即对于每个数字来说，最终进行的变换为 `3 * (2 * i + 1) + 2`。运行：

```
$ go run main.go
11
17
23
```

`Marshal`

`Marshal` 对经过它的数据进行一次 `Marshal`。这个 `Marshal` 可以是 `json.Marshal/proto.Marshal`，甚至我们自己写的 `Marshal` 函数。它接受一个类型为 `func(interface{}) ([]byte, error)` 的函数用于对数据进行处理。

```
type User struct {
  Name string `json:"name"`
  Age  int    `json:"age"`
}

func main() {
  observable := rxgo.Just(
    User{
      Name: "dj",
```

```go
			Age:  18,
		},
		User{
			Name: "jw",
			Age:  20,
		},
	)()

	observable = observable.Marshal(json.Marshal)

	for item := range observable.Observe() {
		fmt.Println(string(item.V.([]byte)))
	}
}
```

由于 `Marshal` 操作返回的是 `[]byte` 类型，我们需要进行类型转换之后再输出。

`Unmarshal`

既然有 `Marshal`，也就有它的相反操作 `Unmarshal`。`Unmarshal` 用于将一个 `[]byte` 类型转换为相应的结构体或其他类型。与 `Marshal` 不同，`Unmarshal` 需要知道转换的目标类型，所以需要提供一个函数用于生成该类型的对象。然后将 `[]byte` 数据 `Unmarshal` 到该对象中。`Unmarshal` 接受两个参数，参数一是类型为 `func([]byte, interface{}) error` 的函数，参数二是 `func () interface{}` 用于生成实际类型的对象。我们拿上面的例子中生成的 JSON 字符串作为数据，将它们重新 `Unmarshal` 为 `User` 对象：

```go
type User struct {
	Name string `json:"name"`
	Age  int    `json:"age"`
}

func main() {
	observable := rxgo.Just(
		`{"name":"dj","age":18}`,
		`{"name":"jw","age":20}`,
	)()

	observable = observable.Map(func(_ context.Context, i interface{}) (interface{}, error) {
		return []byte(i.(string)), nil
	}).Unmarshal(json.Unmarshal, func() interface{} {
		return &User{}
	})
```

```
    for item := range observable.Observe() {
        fmt.Println(item.V)
    }
}
```

由于 `Unmarshaller` 接受 `[]byte` 类型的参数，我们在 `Unmarshal` 之前加了一个 `Map` 用于将 `string` 转为 `[]byte` 。运行：

```
$ go run main.go
&{dj 18}
&{jw 20}
```

`Buffer`

`Buffer` 按照一定的规则收集接收到的数据，然后一次性发送出去（作为切片），而不是收到一个发送一个。有 3 种类型的 `Buffer` ：

- `BufferWithCount(n)` ：每收到 `n` 个数据发送一次，最后一次可能少于 `n` 个；
- `BufferWithTime(n)` ：发送在一个时间间隔 `n` 内收到的数据；
- `BufferWithTimeOrCount(d, n)` ：收到 `n` 个数据，或经过 `d` 时间间隔，发送当前收到的数据。

`BufferWithCount` ：

```
func main() {
    observable := rxgo.Just(1, 2, 3, 4)()

    observable = observable.BufferWithCount(3)

    for item := range observable.Observe() {
        fmt.Println(item.V)
    }
}
```

运行：

```
$ go run main.go
[1 2 3]
[4]
```

注意，最后一组只有一个。

`BufferWithTime` :

```
func main() {
  ch := make(chan rxgo.Item, 1)

  go func() {
    i := 0
    for range time.Tick(time.Second) {
      ch <- rxgo.Of(i)
      i++
    }
  }()

  observable := rxgo.FromChannel(ch).BufferWithTime(rxgo.WithDuration(3 * time.S
econd))

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

每 3s 发送一次：

```
$ go run main.go
[0 1 2]
[3 4 5]
[6 7 8]
...
```

`BufferWithTimeOrCount` :

```
func main() {
  ch := make(chan rxgo.Item, 1)

  go func() {
    i := 0
    for range time.Tick(time.Second) {
      ch <- rxgo.Of(i)
      i++
    }
  }()

  observable := rxgo.FromChannel(ch).BufferWithTimeOrCount(rxgo.WithDuration(3*t
ime.Second), 2)
```

```
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面 3s 可以收集 3 个数据，但是设置了收集 2 个就发送。所以，运行输出为：

```
$ go run main.go
[0 1]
[2 3]
[4 5]
...
```

`GroupBy`

`GroupBy` 根据传入一个 **Hash** 函数，为每个不同的结果分别创建新的 **Observable**。换句话说，`GroupBy` 生成一个数据类型为 **Observable** 的 **Observable**。

```
func main() {
  count := 3
  observable := rxgo.Range(0, 10).GroupBy(count, func(item rxgo.Item) int {
    return item.V.(int) % count
  }, rxgo.WithBufferedChannel(10))

  for subObservable := range observable.Observe() {
    fmt.Println("New observable:")

    for item := range subObservable.V.(rxgo.Observable).Observe() {
      fmt.Printf("item: %v\n", item.V)
    }
  }
}
```

上面根据每个数模 3 的余数将整个流分为 3 组。运行：

```
$ go run main.go
New observable:
item: 0
item: 3
item: 6
item: 9
New observable:
item: 1
```

```
item: 4
item: 7
item: 10
New observable:
item: 2
item: 5
item: 8
```

注意 `rxgo.WithBufferedChannel(10)` 的使用，由于我们的数字是连续生成的，依次为 0->1->2->...->9->10。而 **Observable** 默认是惰性的，即由 `Observe()` 驱动。内层的 `Observe()` 在返回一个 0 之后就等待下一个数，但是下一个数 1 不在此 **Observable** 中。所以会陷入死锁。使用 `rxgo.WithBufferedChannel(10)` ，设置它们之间的连接 channel 缓冲区大小为 10，这样即使我们未取出 channel 里面的数字，上游还是能发送数字进来。

并行操作

默认情况下，这些转换操作都是串行的，即只有一个 goroutine 负责执行转换函数。我们也可以使用 `rxgo.WithPool(n)` 选项设置运行 `n` 个 goroutine，或者 `rxgo.WitCPUPool()` 选项设置运行与逻辑 CPU 数量相等的 goroutine。

```
func main() {
  observable := rxgo.Range(1, 100)

  observable = observable.Map(func(_ context.Context, i interface{}) (interface{}, error) {
    time.Sleep(time.Duration(rand.Int31()))
    return i.(int)*2 + 1, nil
  }, rxgo.WithCPUPool())

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

由于是并行，所以**输出顺序就不确定了**。为了让不确定性更明显一点，我在代码中加了一行 `time.Sleep` 。

过滤 Observable

Observable 中发送过来的数据并不一定都是我们需要的，我们要把不想要的过滤掉。

`Filter`

`Filter()` 接受一个类型为 `func (i interface{}) bool` 的参数，通过的数据使用这个函数断言，返回 `true` 的将发送给下一个阶段。否则，丢弃。

```
func main() {
  observable := rxgo.Range(1, 10)

  observable = observable.Filter(func(i interface{}) bool {
    return i.(int)%2 == 0
  })

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面过滤掉奇数，最后只剩下偶数：

```
$ go run main.go
2
4
6
8
10
```

`ElementAt`

`ElementAt()` 只发送指定索引的数据，如 `ElementAt(2)` 只发送索引为 2 的数据，即第 3 个数据。

```
func main() {
  observable := rxgo.Just(0, 1, 2, 3, 4)().ElementAt(2)

  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面代码输出 2。

`Debounce`

`Debounce()` 比较有意思，它收到数据后还会等待指定的时间间隔，后续间隔内没有收到其他数据才会发送刚开始的数据。

```go
func main() {
  ch := make(chan rxgo.Item)

  go func() {
    ch <- rxgo.Of(1)
    time.Sleep(2 * time.Second)
    ch <- rxgo.Of(2)
    ch <- rxgo.Of(3)
    time.Sleep(2 * time.Second)
    close(ch)
  }()

  observable := rxgo.FromChannel(ch).Debounce(rxgo.WithDuration(1 * time.Second))
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

上面示例，先收到 1，然后 2s 内没收到数据，所以发送 1。接着收到了数据 2，由于马上又收到了 3，所以 2 不会发送。收到 3 之后 2s 内没有收到数据，发送了 3。所以最后输出为 1，3。

`Distinct`

`Distinct()` 会记录它发送的所有数据，它不会发送重复的数据。由于数据格式多样，`Distinct()` 要求我们提供一个函数，根据原数据返回一个唯一标识码（有点类似哈希值）。基于这个标识码去重。

```go
func main() {
  observable := rxgo.Just(1, 2, 2, 3, 3, 4, 4)().
    Distinct(func(_ context.Context, i interface{}) (interface{}, error) {
      return i, nil
    })
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

依次输出 1，2，3，4，没有重复。

`Skip`

`Skip` 可以跳过前若干个数据。

```go
func main() {
  observable := rxgo.Just(1, 2, 3, 4, 5)().Skip(2)
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

`Take`

`Take` 只取前若干个数据。

```go
func main() {
  observable := rxgo.Just(1, 2, 3, 4, 5)().Take(2)
  for item := range observable.Observe() {
    fmt.Println(item.V)
  }
}
```

选项

rxgo 提供的大部分方法的最后一个参数是一个可变长的选项类型。这是 Go 中特有的、经典的选项设计模式。我们前面已经使用了：

- `rxgo.WithBufferedChannel(10)`：设置 channel 的缓存大小；
- `rxgo.WithPool(n)/rxgo.WithCpuPool()`：使用多个 goroutine 执行转换操作；
- `rxgo.WithPublishStrategy()`：使用发布策略，即创建可连接的 **Observable**。

除此之外，rxgo 还提供了很多其他选项。留待大家自行探索了。

总结

rxgo 让基于 **pipelines** 的并发编程变得更容易！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. rxgo GitHub：https://github.com/jordan-wright/rxgo
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

goquery

简介

goquery是用 Go 语言编写的一个类似于 jQuery 的库。它基于 HTML 解析库net/html和 CSS 库cascadia，提供与 jQuery 相近的接口。Go 著名的爬虫框架colly就是基于 goquery 的。

快速使用

本文代码使用 Go Modules。

创建目录并初始化：

```
$ mkdir goquery && cd goquery
$ go mod init github.com/darjun/go-daily-lib/goquery
```

安装 `goquery` 库：

```
$ go get -u github.com/PuerkitoBio/goquery
```

下面我们编写一个抓取百度热榜的小程序：

```
package main

import (
  "fmt"
  "log"
  "net/http"

  "github.com/PuerkitoBio/goquery"
)

func BaiduHotSearch() {
  res, err := http.Get("http://www.baidu.com")
  if err != nil {
    log.Fatal(err)
  }
  defer res.Body.Close()
  if res.StatusCode != 200 {
```

```
        log.Fatalf("status code error: %d %s", res.StatusCode, res.Status)
    }

    doc, err := goquery.NewDocumentFromReader(res.Body)
    if err != nil {
        log.Fatal(err)
    }

    doc.Find(".s-hotsearch-content .hotsearch-item").Each(func(i int, s *goquery.Selection) {
        content := s.Find(".title-content-title").Text()
        fmt.Printf("%d: %s\n", i, content)
    })
}

func main() {
    BaiduHotSearch()
}
```

运行：

```
$ go run main.go
0: 熊孩子转走老妈5万块还赚了几十
1: 中使馆回应马来西亚扣留中国渔船
2: 科学家发现π行星
3: 医保局回应新冠疫苗医保全额报销
4: 阿塞拜疆第二大城市遭炮击
5: 李宇春周笔畅15年后再同框
```

可见，goquery 的使用非常简单：

- 创建一个 `io.Reader` ，数据来源可以多样，可以从文件读取，可以是已有的字符串，还可以通过 HTTP 请求获取；
- 调用 `NewDocumentFromReader` 传入上面的 `io.Reader` 构造一个 `*goquery.Document` 对象；
- 然后就可以调用 `Document` 相关方法查询我们感兴趣的内容了。

在获取我们感兴趣的内容之前，我们必须要知道它们在 HTML 文档的什么位置。拿上面的示例来说，我感兴趣的是百度的热榜。首先打开百度：

百度一下

百度热榜 换一换

1 熊孩子转走老妈5万块还赚了几十 热

2 科学家发现π行星

3 阿塞拜疆第二大城市遭炮击

4 中使馆回应马来西亚扣留中国渔船

5 医保局回应新冠疫苗医保全额报销 新

6 李宇春周笔畅15年后再同框

然后，打开浏览器的开发者工具。我使用的是 Chrome 浏览器，按下 F12:

```
Elements  Console  Sources  Network  Performance  »        ⊗ 1  ⚠ 3  ▪ 16
 ▶<div id="lg" class="s-p-top">…</div>
 ▶<a href="/" id="result_logo" onmousedown="return c({'fm':'tab','tab':
 'logo'})">…</a>
 ▶<form id="form" name="f" action="/s" class="fm" _lpchecked="1">…
 </form>
 ▶<div id="m" class="under-tips s_lm_hide ">…</div>
 ▼<div id="s-hotsearch-wrapper" class="s-isindex-wrap s-hotsearch-
 wrapper">
   ▶<div class="s-hotsearch-title">…</div>
   ▼<ul class="s-hotsearch-content" id="hotsearch-content-wrapper">
…    ▼<li class="hotsearch-item odd" data-index="0"> == $0
       ▼<a class="title-content c-link c-font-medium c-line-clamp1"
       href="https://www.baidu.com/s?
       cl=3&tn=baidutop10&fr=top1000&wd=%E7%86%8A%E5%A…
       %9A%E4%BA%86%E5%87%A0%E5%8D%81&rsv_idx=2&rsv_dl=fyb_n_homepage&his
       filter=1" target="_blank">
           <span class="title-content-index c-index-single c-index-single-
           hot1">1</span>
           <span class="title-content-title">熊孩子转走老妈5万块还赚了几十
           </span>
           <span class="title-content-mark c-text c-gap-left-small c-text-
           hot">热</span>
         </a>
       </li>
     ▶<li class="hotsearch-item even" data-index="3">…</li>
     ▶<li class="hotsearch-item odd" data-index="1">…</li>
     ▶<li class="hotsearch-item even" data-index="4">…</li>
     ▶<li class="hotsearch-item odd" data-index="2">…</li>
     ▶<li class="hotsearch-item even" data-index="5">…</li>
     </ul>
   </div>
 ▶<textarea id="hotsearch_data" style="display:none;">…</textarea>
```

找到想要获取的内容在文档中的位置。必要时可以使用开发者工具左上角的定位按钮来定位，点击按钮，然后再点击一下我们想要定位的内容，就会自动定位到对应的 HTML 源码位置。非常方便！

然后调用相关查找方法，传入 CSS 选择器。选择器可以有多种形式，上面我使用 `.s-hotsearch-content .hotsearch-item` 定位到热榜的每个条目。这里的语法与 jQuery 的一样。 `.s-hotsearch-content .hotsearch-item` 表示查找拥有 `class=s-hotsearch-content` 的节点下的所有拥有 `class=hotsearch-item` 的节点。

当然，这里我们使用 `.s-hotsearch-content li` 也是一样的。看你喜欢。 `Find` 方法返回一个 `*goquery.Selection` 对象。接着，我们使用 `Selection.Each` 遍历每个热榜条目，输出热榜内容，即拥有 `class=title-content-title` 的 `span` 元素的内容。

基本概念

goquery 暴露了两个结构， `Document` 和 `Selection` 和一个接口 `Matcher` 。与 jQuery 不同的是，net/html 包解析 HTML 返回的是一个个节点，而不是一个完整的 DOM 树。所以 jQuery 中的各种状态操作函数在 goquery 中是没有对应的方法的，例如 `height/css/detach` 等。

另外，jQuery 有一个显著的特点，它根据传入参数的个数和类型的不同实现不同的功能。goquery 采用静态类型的编译型语言 Go，如果也采用这种方式，静态语言的优势就发挥不出来了。为此，goquery 在方法命名上做了一些约定：

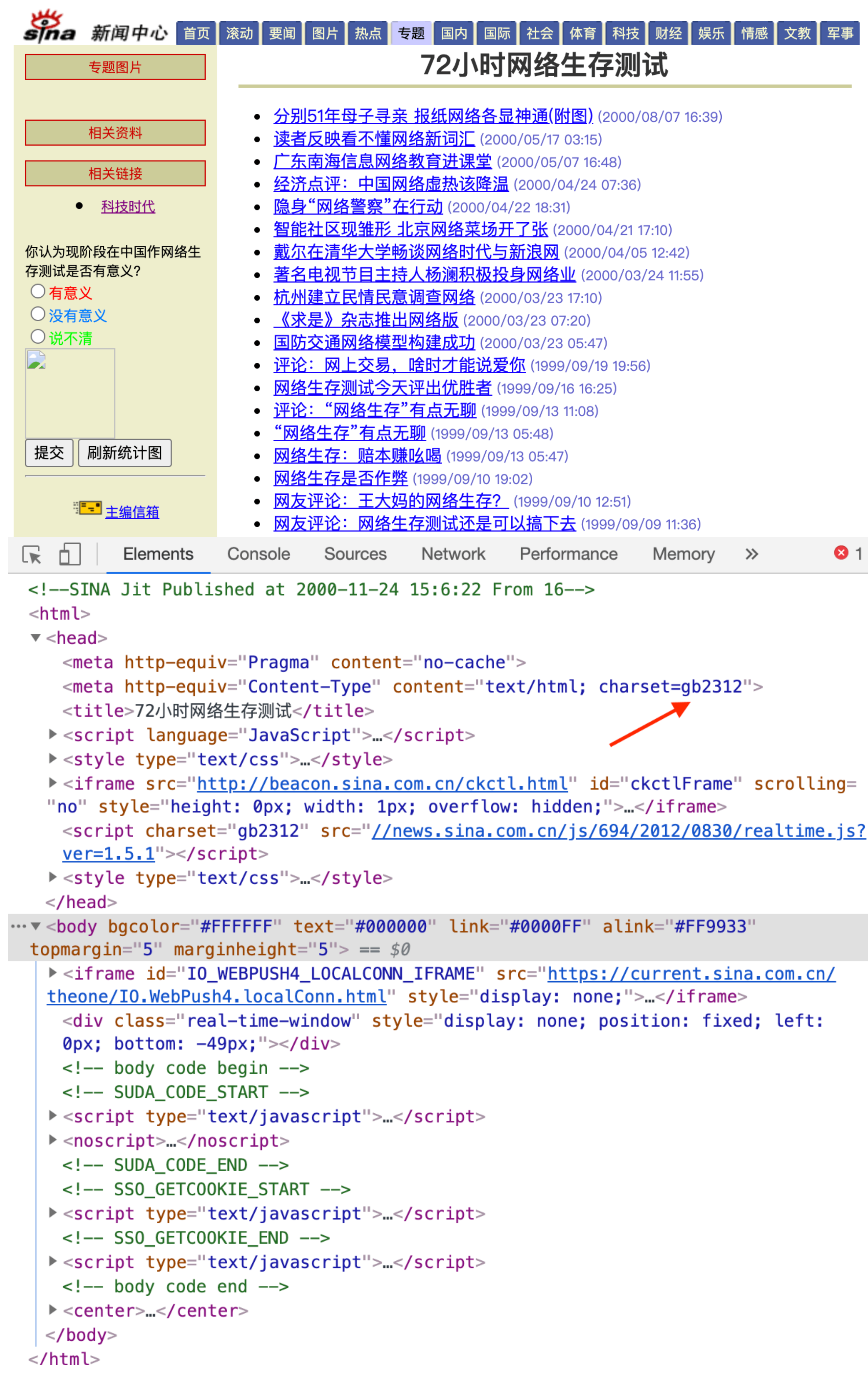
- jQuery 中可以不带参数调用的函数，在 goquery 也就是相同的名字，例如 `Prev()` 。接受一个字符串**选择器**参数的版本在 goquery 中命名为 `XxxFiltered()` ，例如 `PrevFiltered()` ；
- jQuery 只接受一个参数的函数，在 goquery 有相同的名字，例如 `Is()` ；
- jQuery 中接受一个 jQuery 对象作为参数的函数，在 goquery 中被命名为 `XxxSelection()` ，并且接受一个 `*Selection` 类型的参数，例如 `FilterSelection()` ；
- jQuery 中接受一个 DOM 元素作为参数的函数，在 goquery 中被命名为 `XxxNodes()` ，并且接受一个类型为 `*html.Node` 的可变长参数，例如 `FilterNodes()` ;
- jQuery 中接受一个函数作为参数的函数，在 goquery 中被命名为 `XxxFunction()` ，并且接受一个函数作为参数，例如 `FilterFunction()` ;
- goquery 中可以用选择器调用的函数有一个接受 `Matcher` 类型参数的版本，命名为 `XxxMatcher()` ，例如 `IsMatcher()` 。

了解 jQuery 的童鞋，熟悉了上面的约定后，使用 goquery 基本就没有什么问题了。

编码

由于 net/html 要求使用 UTF-8 编码，goquery 也是如此。我们需要保证传给 goquery 的 HTML 源字符串是 UTF-8 编码的。现在已经很少有**非**UTF-8 编码的网页了。在早些时候，国内很多网站都是使用 GB2312 或 GBK 编码。如果我们遇到了非 UTF-8 编码的网页该怎么办呢？可以使用iconv-go将字符串的编码转为 UTF-8。

我在知乎https://www.zhihu.com/question/20091439的这个回答中找到了一个 2000 年的新浪网页，72小时网络生存测试，使用 GB2312 编码：



新闻中心
首页 滚动 要闻 图片 热点 专题 国内 国际 社会 体育 科技 财经 娱乐 情感 文教 军事
专题图片
72小时网络生存测试
相关资料
相关链接
科技时代
你认为现阶段在中国作网络生存测试是否有意义?
有意义
没有意义
说不清
提交 刷新统计图
主编信箱
分别51年母子寻亲 报纸网络各显神通(附图) (2000/08/07 16:39)
读者反映看不懂网络新词汇 (2000/05/17 03:15)
广东南海信息网络教育进课堂 (2000/05/07 16:48)
经济点评：中国网络虚热该降温 (2000/04/24 07:36)
隐身“网络警察”在行动 (2000/04/22 18:31)
智能社区现雏形 北京网络菜场开了张 (2000/04/21 17:10)
戴尔在清华大学畅谈网络时代与新浪网 (2000/04/05 12:42)
著名电视节目主持人杨澜积极投身网络业 (2000/03/24 11:55)
杭州建立民情民意调查网络 (2000/03/23 17:10)
《求是》杂志推出网络版 (2000/03/23 07:20)
国防交通网络模型构建成功 (2000/03/23 05:47)
评论：网上交易，啥时才能说爱你 (1999/09/19 19:56)
网络生存测试今天评出优胜者 (1999/09/16 16:25)
评论：“网络生存”有点无聊 (1999/09/13 11:08)
“网络生存”有点无聊 (1999/09/13 05:48)
网络生存：赔本赚吆喝 (1999/09/13 05:47)
网络生存是否作弊 (1999/09/10 19:02)
网友评论：王大妈的网络生存? (1999/09/10 12:51)
网友评论：网络生存测试还是可以搞下去 (1999/09/09 11:36)
Elements Console Sources Network Performance Memory 1
<!--SINA Jit Published at 2000-11-24 15:6:22 From 16-->
<html>
<head>
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>72小时网络生存测试</title>
<script language="JavaScript">…</script>
<style type="text/css">…</style>
<iframe src="http://beacon.sina.com.cn/ckctl.html" id="ckctlFrame" scrolling="no" style="height: 0px; width: 1px; overflow: hidden;">…</iframe>
<script charset="gb2312" src="//news.sina.com.cn/js/694/2012/0830/realtime.js?ver=1.5.1"></script>
<style type="text/css">…</style>
</head>
<body bgcolor="#FFFFFF" text="#000000" link="#0000FF" alink="#FF9933" topmargin="5" marginheight="5"> == $0
<iframe id="IO_WEBPUSH4_LOCALCONN_IFRAME" src="https://current.sina.com.cn/theone/IO.WebPush4.localConn.html" style="display: none;">…</iframe>
<div class="real-time-window" style="display: none; position: fixed; left: 0px; bottom: -49px;"></div>
<!-- body code begin -->
<!-- SUDA_CODE_START -->
<script type="text/javascript">…</script>
<noscript>…</noscript>
<!-- SUDA_CODE_END -->
<!-- SSO_GETCOOKIE_START -->
<script type="text/javascript">…</script>
<!-- SSO_GETCOOKIE_END -->
<script type="text/javascript">…</script>
<!-- body code end -->
<center>…</center>
</body>
</html>

我们就来抓一下这个列表。首先安装 iconv-go：

```
$ go get -u github.com/djimenez/iconv-go
```

编码：

```
package main

import (
	"fmt"
	"log"
	"net/http"

	"github.com/PuerkitoBio/goquery"
	"github.com/djimenez/iconv-go"
)

func SinaNewSurvival() {
	res, err := http.Get("http://news.sina.com.cn/society/netsurvival")
	if err != nil {
		log.Fatal(err)
	}
	defer res.Body.Close()
	if res.StatusCode != 200 {
		log.Fatalf("status code error: %d %s", res.StatusCode, res.Status)
	}

	utf8Body, err := iconv.NewReader(res.Body, "gb2312", "utf-8")
	if err != nil {
		log.Fatal(err)
	}

	doc, err := goquery.NewDocumentFromReader(utf8Body)
	if err != nil {
		log.Fatal(err)
	}

	doc.Find(".title14 li").Each(func(i int, s *goquery.Selection) {
		content := s.Find("a").Text()
		time := s.Find("font").Text()
		fmt.Printf("%d: %s%s\n", i, content, time)
	})
}
```

```
func main() {
  SinaNewSurvival()
}
```

基本结构与第一个例子一样，注意 `iconv` 的使用：

```
utf8Body, err := iconv.NewReader(res.Body, "gb2312", "utf-8")
```

如果我们不做这一层转换的话，最后输出会是乱码。运行：

```
$ go run main.go
0: 分别51年母子寻亲 报纸网络各显神通(附图) (2000/08/07  16:39)
1: 读者反映看不懂网络新词汇 (2000/05/17  03:15)
2: 广东南海信息网络教育进课堂 (2000/05/07  16:48)
3: 经济点评：中国网络虚热该降温 (2000/04/24  07:36)
4: 隐身“网络警察”在行动 (2000/04/22  18:31)
5: 智能社区现雏形 北京网络菜场开了张 (2000/04/21  17:10)
6: 戴尔在清华大学畅谈网络时代与新浪网 (2000/04/05  12:42)
7: 著名电视节目主持人杨澜积极投身网络业 (2000/03/24  11:55)
8: 杭州建立民情民意调查网络 (2000/03/23  17:10)
9: 《求是》杂志推出网络版 (2000/03/23  07:20)
10: 国防交通网络模型构建成功 (2000/03/23  05:47)
11: 评论：网上交易，啥时才能说爱你 (1999/09/19  19:56)
12: 网络生存测试今天评出优胜者 (1999/09/16  16:25)
```

如果我们想做得通用一点，不想把网页编码写死在代码里面，可以使用x/text/encoding和x/net/html/charset这两个包去猜测网页的编码。安装 x/text 包，由于 x/net/html/charset 是 x/net/html 的子包，不需要再次安装了：

```
$ go get -u golang.org/x/text
```

工具函数：

```
func detectContentCharset(body io.Reader) string {
  r := bufio.NewReader(body)
  if data, err := r.Peek(1024); err == nil {
    if _, name, _ := charset.DetermineEncoding(data, ""); len(name) != 0 {
      return name
    }
  }

  return "utf-8"
}
```

```go
func DecodeHTMLBody(body io.Reader, charset string) (io.Reader, error) {
  if charset == "" {
    charset = detectContentCharset(body)
  }

  e, err := htmlindex.Get(charset)
  if err != nil {
    return nil, err
  }

  if name, _ := htmlindex.Name(e); name != "utf-8" {
    body = e.NewDecoder().Reader(body)
  }

  return body, nil
}
```

`charset.DetermineEncoding` 会根据 HTML 页面中的 meta 元信息猜测网页编码。

接下来只需要把使用 iconv-go 的那行代码改为下面的即可：

```go
utf8Body, err := DecodeHTMLBody(res.Body, "")
```

总结

goquery 功能强大，使用简单，是爬虫库 colly 的基石。可以用来做一些简单的爬取工作和 HTML 处理。由于过于底层，爬取大量的，复杂的网页建议还是使用 colly 来完成。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. goquery GitHub：https://github.com/PuerkitoBio/goquery
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

air

简介

`air` 是 Go 语言的热加载工具，它可以监听文件或目录的变化，自动编译，重启程序。大大提高开发期的工作效率。

快速使用

本文代码使用 Go Modules，在 Mac 上运行。

先创建目录并初始化：

```
$ mkdir air && cd air
$ go mod init github.com/darjun/go-daily-lib/air
```

执行下面的命令安装 `air` 工具：

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

上面的命令会在 `$GOPATH/bin` 目录下生成 `air` 命令。我一般会将 `$GOPATH/bin` 加入系统 `PATH` 中，所以可以方便地在任何地方执行 `air` 命令。

下面我们使用标准库 `net/http` 编写一个简单的 Web 服务器：

```
package main

import (
  "fmt"
  "log"
  "net/http"
)

func index(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "Hello, world!")
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", index)
```

```
server := &http.Server{
    Handler: mux,
    Addr:    ":8080",
}

log.Fatal(server.ListenAndServe())
}
```

运行程序：

```
$ go run main.go
```

在浏览器中访问 `localhost:8080` 即可看到 `Hello, world!` 。

如果现在要求把 `Hello, world!` 改为 `Hello, dj!` ，如果不用 `air` 只能修改代码，执行 `go build` 编译，然后重启。

使用 `air` ，我们只需要执行下面的命令就可以运行程序：

```
$ air
```

`air` 会自动编译，启动程序，并监听当前目录中的文件修改：

```
➜  air git:(master) ✗ air

  __    _   ___
 / /\  | | | |_)
/_/--\ |_| |_| \_ v1.12.3 // live reload for Go apps, with Go1.14.0

mkdir /Users/alibaba/Documents/code/golang/src/github.com/darjun/go-daily-lib/air/tmp
watching .
!exclude tmp
building...
running...
```

当我们将 `Hello, world!` 改为 `Hello, dj!` 时， `air` 监听到了这个修改，会自动编译，并且重启程序：

```
➜  air git:(master) ✗ air

  __    _   ___
 / /\  | | | |_)
/_/--\ |_| |_| \_ v1.12.3 // live reload for Go apps, with Go1.14.0

mkdir /Users/alibaba/Documents/code/golang/src/github.com/darjun/go-daily-lib/air/tmp
watching .
!exclude tmp
building...
running...
main.go has changed
building...
running...
```

这时，我们在浏览器中访问 `localhost:8080`，文本会变为 `Hello, dj!`，是不是很方便？

配置

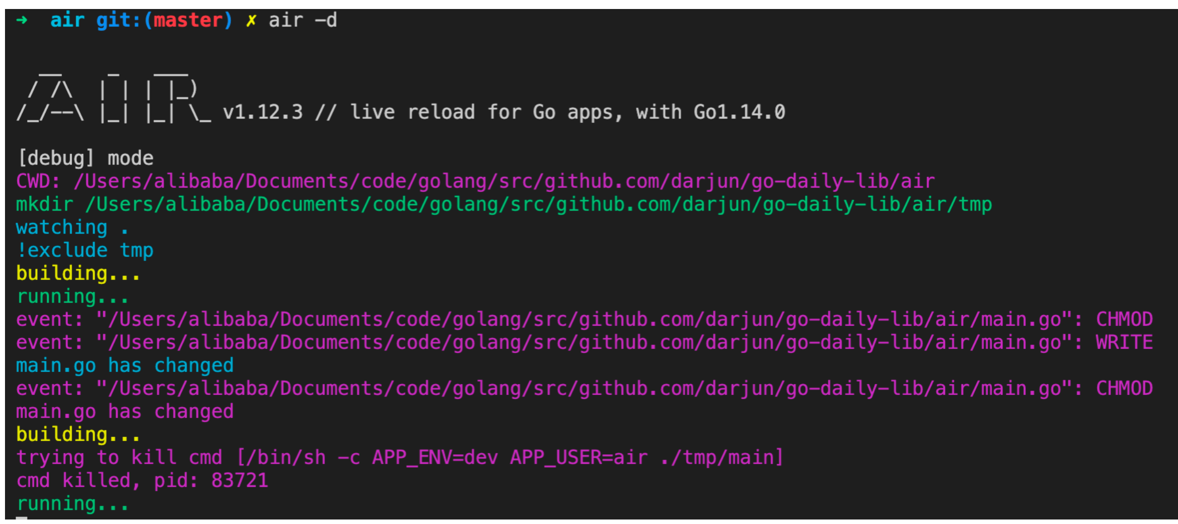
直接执行 `air` 命令，使用的就是默认的配置。一般建议将 `air` 项目中提供的 `air_example.toml` 配置文件复制一份，根据自己的需求做修改和定制：

```
root = "."
tmp_dir = "tmp"

[build]
cmd = "go build -o ./tmp/main ."
bin = "tmp/main"
full_bin = "APP_ENV=dev APP_USER=air ./tmp/main"
include_ext = ["go", "tpl", "tmpl", "html"]
exclude_dir = ["assets", "tmp", "vendor", "frontend/node_modules"]
include_dir = []
exclude_file = []
log = "air.log"
delay = 1000 # ms
stop_on_error = true
send_interrupt = false
kill_delay = 500 # ms

[log]
time = false

[color]
main = "magenta"
watcher = "cyan"
build = "yellow"
runner = "green"

[misc]
clean_on_exit = true
```

可以配置项目根目录，临时文件目录，编译和执行的命令，监听文件目录，监听后缀名，甚至控制台日志颜色都可以配置。

调试模式

如果想查看 `air` 更详细的执行流程，可以使用 `-d` 选项。



使用 `-d` 选项，`air` 会输出非常详细的信息，可以帮助排查问题。

总结

在开发期，使用 `air` 可以避免频繁地编译，重启。把这些都自动化了，大大地提升了开发效率。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. air GitHub：https://github.com/cosmtrek/air
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

quicktemplate

简介

最近在整理我们项目代码的时候，发现有很多活动的代码在结构和提供的功能上都非常相似。为了方便今后的开发，我花了一点时间编写了一个生成代码框架的工具，最大程度地降低重复劳动。代码本身并不复杂，且与项目代码关联性较大，这里就不展开介绍了。在这个过程中，我发现 Go 标准的模板库 `text/template` 和 `html/template` 使用起来比较束手束脚，很不方便。我从 GitHub 了解到 `quicktemplate` 这个第三方模板库，功能强大，语法简单，使用方便。今天我们就来介绍一下 `quicktemplate` 。

快速使用

本文代码使用 Go Modules。

先创建代码目录并初始化：

```
$ mkdir quicktemplate && cd quicktemplate
$ go mod init github.com/darjun/go-daily-lib/quicktemplate
```

`quicktemplate` 会将我们编写的模板代码转换为 Go 语言代码。因此我们需要安装 `quicktemplate` 包和一个名为 `qtc` 的编译器：

```
$ go get -u github.com/valyala/quicktemplate
$ go get -u github.com/valyala/quicktemplate/qtc
```

首先，我们需要编写 `quicktemplate` 格式的模板文件，模板文件默认以 `.qtpl` 作为扩展名。下面我编写了一个简单的模板文件 `greeting.qtpl` ：

```
All text outside function is treated as comments.

{% func Greeting(name string, count int) %}
  {% for i := 0; i < count; i++ %}
    Hello, {%s name %}
  {% endfor %}
{% endfunc %}
```

模板语法非常简单，我们只需要简单了解以下 2 点：

- 模板以函数为单位，函数可以接受任意类型和数量的参数，这些参数可以在函数中使用。所有函数外的文本都是注释， `qtc` 编译时会忽视注释；
- 函数内的内容，除了语法结构，其他都会原样输出到渲染后的文本中，**包括空格和换行**。

将 `greeting.qtpl` 保存到 `templates` 目录，然后执行 `qtc` 命令。该命令会生成对应的 Go 文件 `greeting.qtpl.go` ，包名为 `templates` 。现在，我们就可以使用这个模板了：

```
package main

import (
  "fmt"

  "github.com/darjun/go-daily-lib/quicktemplate/get-started/templates"
)

func main() {
  fmt.Println(templates.Greeting("dj", 5))
}
```

调用模板函数，传入参数，返回渲染后的文本：

```
$ go run .



    Hello, dj

    Hello, dj

    Hello, dj

    Hello, dj

    Hello, dj
```

`{%s name %}` 执行文本替换， `{% for %}` 循环生成重复文本。输出中出现多个空格和换行，这是因为**函数内除了语法结构，其他内容都会原样保留，包括空格和换行**。

需要注意的是，由于 `quicktemplate` 是将模板转换为 Go 代码使用的，所以**如果模板有修改，必须先执行 `qtc` 命令重新生成 Go 代码，否则修改不生效**。

语法结构

`quicktemplate` 支持 Go 常见的语法结构， `if/for/func/import/return` 。而且写法与直接写 Go 代码没太大的区别，几乎没有学习成本。只是在模板中使用这些语法时，需要使用 `{%` 和 `%}` 包裹起来，而且 `if` 和 `for` 等需要添加 `endif/endfor` 明确表示结束。

变量

上面我们已经看到如何渲染传入的参数 `name` ，使用 `{%s name %}` 。由于 `name` 是 string 类型，所以在 `{%` 后使用 `s` 指定类型。 `quicktemplate` 还支持其他类型的值：

- 整型： `{%d int %}` ， `{%dl int64 %}` ， `{%dul uint64 %}` ；
- 浮点数： `{%f float %}` 。还可以设置输出的精度，使用 `{%f.precision float %}` 。例如 `{%f.2 1.2345 %}` 输出 `1.23` ；
- 字节切片（ `[]byte` ）： `{%z bytes %}` ；
- 字符串： `{%q str %}` 或字节切片： `{%qz bytes %}` ，引号转义为 `"` ；
- 字符串： `{%j str %}` 或字节切片： `{%jz bytes %}` ，没有引号；
- URL 编码： `{%u str %}` ， `{%uz bytes %}` ；
- `{%v anything %}` ：输出等同于 `fmt.Sprintf("%v", anything)` 。

先编写模板：

```
{% func Types(a int, b float64, c []byte, d string) %}
  int: {%d a %}, float64: {%f.2 b %}, bytes: {%z c %}, string with quotes: {%q d %}, string without quotes: {%j d %}.
{% endfunc %}
```

然后使用：

```
func main() {
  fmt.Println(templates.Types(1, 5.75, []byte{'a', 'b', 'c'}, "hello"))
}
```

运行：

```
$ go run .

  int: 1, float64: 5.75, bytes: abc, string with quotes: &quot;hello&quot;, string without quotes: hello.
```

调用函数

`quicktemplate` 支持在模板中调用模板函数、标准库的函数。由于 `qtc` 会直接生成 Go 代码，**我们甚至还可以在同目录下编写自己的函数给模板调用**，模板 A 中也可以调用模板 B 中定义的函数。

我们先在 `templates` 目录下编写一个文件 `rank.go` ，定义一个 `Rank` 函数，传入分数，返回评级：

```
package templates

func Rank(score int) string {
  if score >= 90 {
    return "A"
  } else if score >= 80 {
    return "B"
  } else if score >= 70 {
    return "C"
  } else if score >= 60 {
    return "D"
  } else {
    return "E"
  }
}
```

然后我们可以在模板中调用这个函数：

```
{% import "fmt" %}
{% func ScoreList(name2score map[string]int) %}
  {% for name, score := range name2score %}
    {%s fmt.Sprintf("%s: score-%d rank-%s", name, score, Rank(score)) %}
  {% endfor %}
{% endfunc %}
```

编译模板：

```
$ qtc
```

编写程序：

```
func main() {
  name2score := make(map[string]int)
  name2score["dj"] = 85
```

```
  name2score["lizi"] = 96
  name2score["hjw"] = 52

  fmt.Println(templates.ScoreList(name2score))
}
```

运行程序输出：

```
$ go run .


    dj: score-85 rank-B

    lizi: score-96 rank-A

    hjw: score-52 rank-E
```

由于我们在模板中用到 `fmt` 包，需要先使用 `{% import %}` 将该包导入。

在模板中调用另一个模板的函数也是类似的，因为模板最终都会转为 Go 代码。Go 代码中有同样签名的函数。

Web

`quicktemplate` 常用来编写 HTML 页面的模板：

```
{% func Index(name string) %}
<html>
  <head>
    <title>Awesome Web</title>
  </head>
  <body>
    <h1>Hi, {%s name %}
    <p>Welcome to the awesome web!!!</p>
  </body>
</html>
{% endfunc %}
```

下面编写一个简单的 Web 服务器：

```
func index(w http.ResponseWriter, r *http.Request) {
  templates.WriteIndex(w, r.FormValue("name"))
```

```
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", index)

  server := &http.Server{
    Handler: mux,
    Addr:     ":8080",
  }

  log.Fatal(server.ListenAndServe())
}
```

`qtc` 会生成一个 `Write*` 的方法，它接受一个 `io.Writer` 的参数。将模板渲染的结果写入这个 `io.Writer` 中，我们可以直接将 `http.ResponseWriter` 作为参数传入，非常便捷。

运行：

```
$ qtc
$ go run .
```

浏览器输入 `localhost:8080?name=dj` 查看结果。

总结

`quicktemplate` 至少有下面 3 个优势：

- 语法与 Go 语言非常类似，几乎没有学习成本；
- 会先转换为 Go，渲染速度非常快，比标准库 `html/template` 快 20 倍以上；
- 为了安全考虑，会执行一些编码，避免受到攻击。

从我个人的实际使用情况来看，确实很方便，很实用。感兴趣的还可以去看看 `qtc` 生成的 Go 代码。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. quicktemplate GitHub：https://github.com/valyala/quicktemplate
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

commonregex

简介

有时，我们会遇到一些需要使用字符串的匹配和查找的任务。并且我们知道这种情况下，使用正则表达式是最简洁和优雅的。为了完成某个任务特地去系统地学习正则表达式费时费力，而且一段时间不用又很容易遗忘。下次遇到问题还要再重复这个过程。`commonregex` 库来了，它内置很多常用的正则表达式，开箱即用。当然，我并不是说没必要去学习正则表达式，熟练掌握正则表达式需要时间和练习，对于时长和文本处理打交道的开发人员，正则表达式决定是提升工作效率的一把利器。

快速使用

本文代码使用 Go Modules。

创建目录并初始化：

```
$ mkdir commonregex && cd commonregex
$ go mod init github.com/darjun/go-daily-lib/commonregex
```

安装 `commonregex` 库：

```
$ go get -u github.com/mingrammer/commonregex
```

简单使用：

```
package main

import (
  "fmt"

  cregex "github.com/mingrammer/commonregex"
)

func main() {
  text := `John, please get that article on www.linkedin.com to me by 5:00PM on Jan 9th 2012. 4:00 would be ideal, actually. If you have any questions, You can reach me at (519)-236-2723x341 or get in touch with my associate at harold.smith@gmail.com`
```

```
  dateList := cregex.Date(text)
  timeList := cregex.Time(text)
  linkList := cregex.Links(text)
  phoneList := cregex.PhonesWithExts(text)
  emailList := cregex.Emails(text)

  fmt.Println("date list:", dateList)
  fmt.Println("time list:", timeList)
  fmt.Println("link list:", linkList)
  fmt.Println("phone list:", phoneList)
  fmt.Println("email list:", emailList)
}
```

运行结果：

```
$ go run main.go
date list: [Jan 9th 2012]
time list: [5:00PM 4:00 ]
link list: [www.linkedin.com harold.smith@gmail.com]
phone list: [(519)-236-2723x341]
email list: [harold.smith@gmail.com]
```

`commonregex` 提供的 API 非常易于使用，调用相应的类别方法返回一段文本中符合这些格式的字符串列表。上面依次从 `text` 获取**日期列表**，**时间列表**，**超链接列表**，**电话号码列表**和**电子邮件列表**。

内置的正则

`commonregex` 支持很多常用的正则表达式：

- 日期；
- 时间；
- 电话号码；
- 超链接；
- 邮件地址；
- IPv4/IPv6/IP 地址；
- 价格；
- 十六进制颜色值；
- 信用卡卡号；
- 10/13 位 ISBN；

- 邮政编码；
- MD5；
- SHA1；
- SHA256；
- GUID，全局唯一标识；
- Git 仓库地址。

每种类型又支持多种格式，例如日期支持 `09.11.2020` / `Sep 11th 2020` 。

下面挑选几种类型来介绍。

日期

```
func main() {
  text := `commonregex support many date formats, like 09.11.2020, Sep 11th 2020 and so on.`
  dateList := commonregex.Date(text)

  fmt.Println(dateList)
}
```

匹配出来的日期（注意 Go 中 slice 的输出）：

```
[09.11.2020 Sep 11th 2020]
```

时间

时间相对来说格式单一一些，有 24 小时制的时间如： `08:30` / `14:35` ，有 12 小时制的时间： `08:30am` / `02:35pm` 。

看示例：

```
func main() {
  text := `I wake up at 08:30 (aka 08:30am) in the morning, take a snap at 13:00 (aka 01:00pm).`
  timeList := commonregex.Time(text)

  fmt.Println(timeList)
}
```

匹配出来的时间列表：

```
[08:30  08:30am 13:00 01:00pm]
```

IP/MAC/MD5

使用方法都是类似的，这几个放在一起举例。

IPv4 地址是 4 个以 `.` 分隔的数字，每个数字都在[0-255]范围内。

MAC 是计算机的物理地址（又叫以太网地址，局域网地址等），是 6 组以 `:` 分隔的十六进制数字，每组两个。

MD5 是一种哈希算法，将一段数据转为长度为 32 的字符串。

```
func main() {
  text := `mac address: ac:de:48:00:11:22, ip: 192.168.3.20, md5: fdbf72fdabb67ea6ef7ff5155a44def4`

  macList := commonregex.MACAddresses(text)
  ipList := commonregex.IPs(text)
  md5List := commonregex.MD5Hexes(text)

  fmt.Println("mac list:", macList)
  fmt.Println("ip list:", ipList)
  fmt.Println("md5 list:", md5List)
}
```

输出：

```
mac list: [ac:de:48:00:11:22]
ip list: [192.168.3.20]
md5 list: [fdbf72fdabb67ea6ef7ff5155a44def4]
```

总结

`commonregex` 足够我们去应付一般的使用场景了。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. commonregex GitHub：https://github.com/mingrammer/commonregex
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gabs

简介

JSON 是一种非常流行的数据交换格式。每种编程语言都有很多操作 JSON 的库，标准库、第三方库都有。Go 语言中标准库内置了 JSON 操作库 `encoding/json` 。我们之前也介绍过专门用于**查询** JSON 串的库gjson和专门用于**修改** JSON 串的库sjson，还有一个非常方便的操作 JSON 数据的命令行工具jj。今天我们再介绍一个 JSON 工具库—— `gabs` 。 `gabs` 是一个用来查询和修改 JSON 串的库。它使用 `encoding/json` 将一般的 JSON 串转为 `map[string]interface{}` ，并提供便利的方法操作 `map[string]struct{}` 。

快速使用

本文代码使用 Go Modules。

创建目录并初始化：

```
$ mkdir gabs && cd gabs
$ go mod init github.com/darjun/go-daily-lib/gabs
```

安装 `gabs` ，目前最新版本为 `v2` ，推荐使用 `v2` ：

```
$ go get -u github.com/Jeffail/gabs/v2
```

使用：

```
package main

import (
  "github.com/Jeffail/gabs/v2"
  "fmt"
)

func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{
    "info": {
      "name": {
        "first": "lee",
        "last": "darjun"
      },
```

```
        "age": 18,
        "hobbies": [
          "game",
          "programming"
        ]
      }
    }`))

  fmt.Println("first name: ", jObj.Search("info", "name", "first").Data().(string))
  fmt.Println("second name: ", jObj.Path("info.name.last").Data().(string))
  gObj, _ := jObj.JSONPointer("/info/age")
  fmt.Println("age: ", gObj.Data().(float64))
  fmt.Println("one hobby: ", jObj.Path("info.hobbies.1").Data().(string))
}
```

首先，我们调用 `gabs.ParseJSON()` 方法解析传入的 JSON 串，得到一个 `gabs.Container` 对象。后续通过该 `gabs.Container` 对象来查询和修改解析出来的数据。

`gabs` 提供 3 种查询方式：

- 以 `.` 分隔的路径调用 `Path()` 方法；
- 将路径各个部分作为可变参数传入 `Search()` 方法；
- 使用 `/` 分隔的路径调用 `JSONPointer()` 方法。

上述方法内部实现最终都是调用相同的方法，只是使用上稍微有些区别。注意：

- 3 个方法最终都返回一个 `gabs.Container` 对象，我们需要调用其 `Data()` 获取内部的数据，然后做一次类型转换得到实际的数据；
- 如果传入的路径有误或路径下无数据，则 `Search/Path` 方法返回的 `gabs.Container` 对象内部数据为 `nil`，即 `Data()` 方法返回 `nil`，而 `JSONPointer` 方法返回 `err`。实际使用时注意进行空指针和错误判断；
- 如果路径某个部分对应的数据类型为数组，则可以在后面追加索引，读取对应索引下的数据，如 `info.hobbies.1`；
- `JSONPointer()` 参数必须以 `/` 开头。

运行结果：

```
$ go run main.go
first name:  lee
second name:  darjun
```

```
age:  18
one hobby:  programming
```

查询 JSON 串

上一节中我们介绍过，在 `gabs` 中，路径有 3 种表示方式。这 3 种方式对应 3 个基础的查询方法：

- `Search(hierarchy ...string)`：也有一个简写形式 `S`；
- `Path(path string)`：`path` 以 `.` 分隔；
- `JSONPointer(path string)`：`path` 以 `/` 分隔。

它们的基本用法上面已经介绍过了，对于数组我们还能对每个数组元素做递归查询。在下面例子中，我们依次返回数组 `members` 中每个元素的 `name`、`age` 和 `relation` 字段：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{
    "user": {
      "name": "dj",
      "age": 18,
      "members": [
        {
          "name": "hjw",
          "age": 20,
          "relation": "spouse"
        },
        {
          "name": "lizi",
          "age": 3,
          "relation": "son"
        }
      ]
    }
  }`))

  fmt.Println("member names: ", jObj.S("user", "members", "*", "name").Data())
  fmt.Println("member ages: ", jObj.S("user", "members", "*", "age").Data())
  fmt.Println("member relations: ", jObj.S("user", "members", "*", "relation").D
ata())

  fmt.Println("spouse name: ", jObj.S("user", "members", "0", "name").Data().(st
```

```
ring))
}
```

运行程序，输出：

```
$ go run main.go
member names:  [hjw lizi]
member ages:  [20 3]
member relations:  [spouse son]
spouse name:  hjw
```

容易看出，在路径中遇到数组分下面两种情况处理：

- 下一个部分路径是 `*` ，则对所有的数组元素应用剩余的路径查询，结果放在一个数组中返回；
- 否则，下一个路径部分必须是数组索引，对该索引所在元素应用剩余的路径查询。

查看源码我们可以知道，实际上， `Path/JSONPointer` 内部都是先将 `path` 解析为 `hierarchy ...string` 的形式，最终都会调用 `searchStrict` 方法：

```
func (g *Container) Search(hierarchy ...string) *Container {
  c, _ := g.searchStrict(true, hierarchy...)
  return c
}

func (g *Container) Path(path string) *Container {
  return g.Search(DotPathToSlice(path)...)
}

func (g *Container) JSONPointer(path string) (*Container, error) {
  hierarchy, err := JSONPointerToSlice(path)
  if err != nil {
    return nil, err
  }
  return g.searchStrict(false, hierarchy...)
}

func (g *Container) S(hierarchy ...string) *Container {
  return g.Search(hierarchy...)
}
```

`searchStrict` 方法也不复杂，我们简单看一下：

```
func (g *Container) searchStrict(allowWildcard bool, hierarchy ...string) (*Cont
ainer, error) {
  object := g.Data()
  for target := 0; target < len(hierarchy); target++ {
    pathSeg := hierarchy[target]
    if mmap, ok := object.(map[string]interface{}); ok {
      object, ok = mmap[pathSeg]
      if !ok {
        return nil, fmt.Errorf("failed to resolve path segment '%v': key '%v' wa
s not found", target, pathSeg)
      }
    } else if marray, ok := object.([]interface{}); ok {
      if allowWildcard && pathSeg == "*" {
        tmpArray := []interface{}{}
        for _, val := range marray {
          if (target + 1) >= len(hierarchy) {
            tmpArray = append(tmpArray, val)
          } else if res := Wrap(val).Search(hierarchy[target+1:]...); res != nil
{
            tmpArray = append(tmpArray, res.Data())
          }
        }
        if len(tmpArray) == 0 {
          return nil, nil
        }
        return &Container{tmpArray}, nil
      }
      index, err := strconv.Atoi(pathSeg)
      if err != nil {
        return nil, fmt.Errorf("failed to resolve path segment '%v': found array
but segment value '%v' could not be parsed into array index: %v", target, pathSe
g, err)
      }
      if index < 0 {
        return nil, fmt.Errorf("failed to resolve path segment '%v': found array
but index '%v' is invalid", target, pathSeg)
      }
      if len(marray) <= index {
        return nil, fmt.Errorf("failed to resolve path segment '%v': found array
but index '%v' exceeded target array size of '%v'", target, pathSeg, len(marra
y))
      }
      object = marray[index]
    } else {
      return nil, fmt.Errorf("failed to resolve path segment '%v': field '%v' wa
```

```
s not found", target, pathSeg)
    }
  }
  return &Container{object}, nil
}
```

实际上就是顺着路径一层层往下走，遇到数组。如果下一个部分是通配符 `*` ，下面是处理代码：

```
tmpArray := []interface{}{}
for _, val := range marray {
  if (target + 1) >= len(hierarchy) {
    tmpArray = append(tmpArray, val)
  } else if res := Wrap(val).Search(hierarchy[target+1:]...); res != nil {
    tmpArray = append(tmpArray, res.Data())
  }
}
if len(tmpArray) == 0 {
  return nil, nil
}
return &Container{tmpArray}, nil
```

如果 `*` 是路径最后一个部分，返回所有数组元素：

```
if (target + 1) >= len(hierarchy) {
  tmpArray = append(tmpArray, val)
}
```

否则，应用剩余的路径查询每个元素，查询结果 `append` 到待返回切片中：

```
else if res := Wrap(val).Search(hierarchy[target+1:]...); res != nil {
  tmpArray = append(tmpArray, res.Data())
}
```

另一方面，如果不是通配符，那么下一个路径部分必须是索引，取这个索引的元素，继续往下查询：

```
index, err := strconv.Atoi(pathSeg)
```

遍历

`gabs` 提供了两个方法可以方便地遍历数组和对象：

- `Children()`：返回所有数组元素的切片，如果在对象上调用该方法，`Children()`将以不确定顺序返回对象所有的值的切片；
- `ChildrenMap()`：返回对象的键和值。

看示例：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{
    "user": {
      "name": "dj",
      "age": 18,
      "members": [
        {
          "name": "hjw",
          "age": 20,
          "relation": "spouse"
        },
        {
          "name": "lizi",
          "age": 3,
          "relation": "son"
        }
      ]
    }
  }`))

  for k, v := range jObj.S("user").ChildrenMap() {
    fmt.Printf("key: %v, value: %v\n", k, v)
  }

  fmt.Println()

  for i, v := range jObj.S("user", "members", "*").Children() {
    fmt.Printf("member %d: %v\n", i+1, v)
  }
}
```

运行结果：

```
$ go run main.go
key: name, value: "dj"
key: age, value: 18
key: members, value: [{"age":20,"name":"hjw","relation":"spouse"},{"age":3,"name":"lizi","relation":"son"}]
```

```
member 1: {"age":20,"name":"hjw","relation":"spouse"}
member 2: {"age":3,"name":"lizi","relation":"son"}
```

这两个方法的源码很简单，建议去看看~

存在性判断

`gabs` 提供了两个方法检查对应的路径上是否存在数据：

- `Exists(hierarchy ...string)`；
- `ExistsP(path string)`：方法名以 `P` 结尾，表示接受以 `.` 分隔的路径。

看示例：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{"user":{"name": "dj","age": 18}}`))
  fmt.Printf("has name? %t\n", jObj.Exists("user", "name"))
  fmt.Printf("has age? %t\n", jObj.ExistsP("user.age"))
  fmt.Printf("has job? %t\n", jObj.Exists("user", "job"))
}
```

运行：

```
$ go run main.go
has name? true
has age? true
has job? false
```

获取数组信息

对于类型为数组的值， `gabs` 提供了几组便捷的查询方法。

- 获取数组大小： `ArrayCount/ArrayCountP` ，不加后缀的方法接受可变参数作为路径，以 `P` 为后缀的方法需要传入 `.` 分隔的路径；
- 获取数组某个索引的元素： `ArrayElement/ArrayElementP` 。

示例：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{
    "user": {
```

```
        "name": "dj",
        "age": 18,
        "members": [
          {
            "name": "hjw",
            "age": 20,
            "relation": "spouse"
          },
          {
            "name": "lizi",
            "age": 3,
            "relation": "son"
          }
        ],
        "hobbies": ["game", "programming"]
      }
    }`))

  cnt, _ := jObj.ArrayCount("user", "members")
  fmt.Println("member count:", cnt)
  cnt, _ = jObj.ArrayCount("user", "hobbies")
  fmt.Println("hobby count:", cnt)

  ele, _ := jObj.ArrayElement(0, "user", "members")
  fmt.Println("first member:", ele)
  ele, _ = jObj.ArrayElement(1, "user", "hobbies")
  fmt.Println("second hobby:", ele)
}
```

输出：

```
member count: 2
hobby count: 2
first member: {"age":20,"name":"hjw","relation":"spouse"}
second hobby: "programming"
```

修改和删除

我们可以使用 `gabs` 构造一个 JSON 串。根据要设置的值的类型， `gabs` 将修改的方法又分为了两类：原始值、数组和对象。基本操作流程是相同的：

- 调用 `gabs.New()` 创建 `gabs.Container` 对象，或者 `ParseJSON()` 从现有 JSON 串中解析出 `gabs.Container` 对象；

- 调用方法设置或修改键值，也可以删除一些键；
- 生成最终的 JSON 串。

原始值

我们前面说过， `gabs` 使用三种方式来表达路径。在设置时也可以通过这三种方式指定在什么位置设置值。对应方法为：

- `Set(value interface{}, hierarchy ...string)` ：将路径各个部分作为可变参数传入即可；
- `SetP(value interface{}, path string)` ：路径各个部分以点 `.` 分隔；
- `SetJSONPointer(value interface{}, path string)` ：路径各个部分以 `/` 分隔，且必须以 `/` 开头。

示例：

```
func main() {
  gObj := gabs.New()

  gObj.Set("lee", "info", "name", "first")
  gObj.SetP("darjun", "info.name.last")
  gObj.SetJSONPointer(18, "/info/age")

  fmt.Println(gObj.String())
}
```

最终生成 JSON 串：

```
$ go run main.go
{"info":{"age":18,"name":{"first":"lee","last":"darjun"}}}
```

我们也可以调用 `gabs.Container` 的 `StringIndent` 方法增加前缀和缩进，让输出更美观些：

```
fmt.Println(gObj.StringIndent("", "  "))
```

观察输出变化：

```
$ go run main.go
{
  "info": {
```

```
    "age": 18,
    "name": {
      "first": "lee",
      "last": "darjun"
    }
  }
}
```

数组

相比原始值，数组的操作复杂不少。我们可以创建新的数组，也可以在原有的数组中添加、删除元素。

```go
func main() {
  gObj := gabs.New()

  arrObj1, _ := gObj.Array("user", "hobbies")
  fmt.Println(arrObj1.String())

  arrObj2, _ := gObj.ArrayP("user.bugs")
  fmt.Println(arrObj2.String())

  gObj.ArrayAppend("game", "user", "hobbies")
  gObj.ArrayAppend("programming", "user", "hobbies")

  gObj.ArrayAppendP("crash", "user.bugs")
  gObj.ArrayAppendP("panic", "user.bugs")
  fmt.Println(gObj.String())
}
```

我们先通过 `Array/ArrayP` 分别在路径 `user.hobbies` 和 `user.bugs` 下创建数组，然后调用 `ArrayAppend/ArrayAppendP` 向这两个数组中添加元素。现在我们应该可以根据方法有无后缀，后缀是什么来区分它接受什么格式的路径了！

运行结果：

```
{"user":{"bugs":["crash","panic"],"hobbies":["game","programming"]}}
```

实际上，我们甚至可以省略上面的数组创建过程，因为 `ArrayAppend/ArrayAppendP` 如果检测到中间路径上没有值，会自动创建对象。

当然我们也可以删除某个索引的数组元素，使用 `ArrayRemove/ArrayRemoveP` 方法：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{"user":{"bugs":["crash","panic"],"hobbies":
["game","programming"]}}`))

  jObj.ArrayRemove(0, "user", "bugs")
  jObj.ArrayRemoveP(1, "user.hobbies")
  fmt.Println(jObj.String())
}
```

删除完成之后还剩下：

```
{"user":{"bugs":["panic"],"hobbies":["game"]}}
```

对象

在指定路径下创建对象使用 `Object/ObjectI/ObjectP` 这组方法，其中 `ObjectI` 是指在数组的特定索引下创建。一般地我们使用 `Set` 类方法就足够了，中间路径不存在会自动创建。

对象删除使用 `Delete/DeleteP` 这组方法：

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{"info":{"age":18,"name":{"first":"lee","las
t":"darjun"}}}`))

  jObj.Delete("info", "name")
  fmt.Println(jObj.String())

  jObj.Delete("info")
  fmt.Println(jObj.String())
}
```

输出：

```
{"info":{"age":18}}
{}
```

`Flatten`

`Flatten` 操作即将嵌套很深的字段提到最外层， `gabs.Flatten` 返回一个新的 `map[string]interface{}` ， `interface{}` 为 JSON 中叶子节点的值，键为该叶子

的路径。例如： `{"foo":[{"bar":"1"},{"bar":"2"}]}` 执行 flatten 操作之后返回 `{"foo.0.bar":"1","foo.1.bar":"2"}` 。

```
func main() {
  jObj, _ := gabs.ParseJSON([]byte(`{
    "user": {
      "name": "dj",
      "age": 18,
      "members": [
        {
          "name": "hjw",
          "age": 20,
          "relation": "spouse"
        },
        {
          "name": "lizi",
          "age": 3,
          "relation": "son"
        }
      ],
      "hobbies": ["game", "programming"]
    }
  }`))

  obj, _ := jObj.Flatten()
  fmt.Println(obj)
}
```

输出：

```
map[user.age:18 user.hobbies.0:game user.hobbies.1:programming user.members.0.age:20 user.members.0.name:hjw user.members.0.relation:spouse user.members.1.age:3 user.members.1.name:lizi user.members.1.relation:son user.name:dj]
```

合并

我们可以将两个 `gabs.Container` 合并成一个。如果同一个路径下有相同的键：

- 如果两者都是对象类型，则对二者进行合并操作；
- 如果两者都是数组类型，则将后者中所有元素追加到前一个数组中；
- 其中一个为数组，合并之后另一个同名键的值将会作为元素添加到数组中。

例如：

```go
func main() {
  obj1, _ := gabs.ParseJSON([]byte(`{"user":{"name":"dj"}}`))
  obj2, _ := gabs.ParseJSON([]byte(`{"user":{"age":18}}`))
  obj1.Merge(obj2)
  fmt.Println(obj1)

  arr1, _ := gabs.ParseJSON([]byte(`{"user": {"hobbies": ["game"]}}`))
  arr2, _ := gabs.ParseJSON([]byte(`{"user": {"hobbies": ["programming"]}}`))
  arr1.Merge(arr2)
  fmt.Println(arr1)

  obj3, _ := gabs.ParseJSON([]byte(`{"user":{"name":"dj", "hobbies": "game"}}`))
  arr3, _ := gabs.ParseJSON([]byte(`{"user": {"hobbies": ["programming"]}}`))
  obj3.Merge(arr3)
  fmt.Println(obj3)

  obj4, _ := gabs.ParseJSON([]byte(`{"user":{"name":"dj", "hobbies": "game"}}`))
  arr4, _ := gabs.ParseJSON([]byte(`{"user": {"hobbies": ["programming"]}}`))
  arr4.Merge(obj4)
  fmt.Println(arr4)

  obj5, _ := gabs.ParseJSON([]byte(`{"user":{"name":"dj", "hobbies": {"first":
"game"}}}`))
  arr5, _ := gabs.ParseJSON([]byte(`{"user": {"hobbies": ["programming"]}}`))
  obj5.Merge(arr5)
  fmt.Println(obj5)
}
```

看结果：

```
{"user":{"age":18,"name":"dj"}}
{"user":{"hobbies":["game","programming"]}}
{"user":{"hobbies":["game","programming"],"name":"dj"}}
{"user":{"hobbies":["programming","game"],"name":"dj"}}
{"user":{"hobbies":[{"first":"game"},"programming"],"name":"dj"}}
```

总结

`gabs` 是一个十分方便的操作 JSON 的库，非常易于使用，而且代码实现比较简洁，值得一看。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gabs GitHub：https://github.com/Jeffail/gabs
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

jobrunner

简介

我们在 Web 开发中时常会遇到这样的需求，执行一个操作之后，需要给用户一定形式的通知。例如，用户下单之后通过邮件发送电子发票，网上购票支付后通过短信发送车次信息。但是这类需求并不需要非常及时，如果放在请求流程中处理，会影响请求的响应时间。这类任务我们一般使用异步的方式来执行。 `jobrunner` 就是其中一个用来执行异步任务的 Go 语言库。得益于强大的 `cron` 库，再搭配 `jobrunner` 的任务状态监控， `jobrunner` 非常易于使用。

快速使用

本文使用 Go Modules。

创建目录并初始化：

```
$ mkdir jobrunner && cd jobrunner
$ go mod init github.com/darjun/go-daily-lib/jobrunner
```

安装 `jobrunner` ：

```
$ go get -u github.com/bamzi/jobrunner
```

使用：

```
package main

import (
  "fmt"
  "time"

  "github.com/bamzi/jobrunner"
)

type GreetingJob struct {
  Name string
}

func (g GreetingJob) Run() {
```

```
  fmt.Println("Hello, ", g.Name)
}

func main() {
  jobrunner.Start()
  jobrunner.Schedule("@every 5s", GreetingJob{Name: "dj"})

  time.Sleep(10 * time.Second)
}
```

我们创建一个任务，每隔 5s 打印一条欢迎信息。任务的创建和执行与 `cron` 完全相同，详细使用见我前面的一篇博文。

注意， `jobrunner` 需要先 `Start()` ，然后再添加任务。因为在 `Start()` 中创建 `MainCron` 对象，先添加任务会 `panic` ！！！

注意 `main` 函数尾的 `time.Sleep(10 * time.Second)` ，因为主 goroutine 结束之后整个程序就退出了， `jobrunner` 中的任务就没有机会被执行了。加上 `time.Sleep` 是为了让大家能看到输出，实际使用中不会这样做。

与 web 框架整合

`jobrunner` 能很方便地与当前常见的 Web 框架整合，如 `Gin/Echo/Martini/Beego/Revel` 等。下面通过一个简单的例子演示如何在 Gin 中使用 `jobrunner` ：用户登录时给他的邮箱发送一封邮件。

首先需要安装相应的库：

```
$ go get -u github.com/gin-gonic/gin
$ github.com/jordan-wright/email
```

编写代码：

```
package main

import (
  "fmt"
  "net/smtp"
  "time"

  "github.com/bamzi/jobrunner"
  "github.com/gin-gonic/gin"
  "github.com/jordan-wright/email"
)
```

```
type EmailJob struct {
  Name  string
  Email string
}

type User struct {
  Name  string `form:"name"`
  Email string `form:"email"`
}

func (j EmailJob) Run() {
  e := email.NewEmail()
  e.From = "leedarjun@126.com"
  e.To = []string{j.Email}
  e.Cc = []string{"leedarjun@126.com"}
  e.Subject = "Welcome To Awesome-Web"
  e.Text = []byte(fmt.Sprintf(`
  Hello, %s
  Welcome Back
  `, j.Name))

  err := e.Send("smtp.126.com:25", smtp.PlainAuth("", "leedarjun@126.com", "yyyy
yy", "smtp.126.com"))
  if err != nil {
    fmt.Printf("failed to send email to %s, err:%v", j.Name, err)
  }
}

func login(c *gin.Context) {
  var u User
  if c.ShouldBind(&u) == nil {
    c.String(200, "login success")

    jobrunner.In(5*time.Second, EmailJob{Name: u.Name, Email: u.Email})
  } else {
    c.String(404, "login failed")
  }
}

func main() {
  r := gin.Default()
  r.GET("/login", login)
  r.Run(":8888")
}
```

这里只是为了简单演示，我们编写了一个简陋的 `login` 函数处理登录，传入 `name` 和 `email` ，然后给该 `email` 发送邮件。 `email` 库的详细使用可以查看我之前的博文了解。

只需要在浏览器中输入 `http://localhost:8888/login?name=dj&email=935653229@qq.com` ，我的 QQ 邮箱就能收到邮件：

Welcome To Awesome-Web ☆
发件人：leedarjun <leedarjun@126.com>
时 间：2020年9月2日（星期三）下午9：24
收件人：孤雨 <935653229@qq.com>
抄 送：leedarjun <leedarjun@126.com>

Hello, dj
Welcome Back

监控

`jobrunner` 内置了一个监控模块，可以很方便地通过网页或者 API 获取当前的任务状态数据：

```
package main

import (
  "fmt"
  "html/template"
  "os"
  "time"

  "github.com/bamzi/jobrunner"
  "github.com/gin-gonic/gin"
)

type GreetingJob struct {
  Name string
}

func (g GreetingJob) Run() {
  fmt.Println("Hello,", g.Name)
}

type EmailJob struct {
  Email string
```

```
}

func (e EmailJob) Run() {
  fmt.Println("Send,", e.Email)
}

func main() {
  r := gin.Default()

  jobrunner.Start()
  jobrunner.Every(5*time.Second, GreetingJob{Name: "dj"})
  jobrunner.Every(10*time.Second, EmailJob{Email: "935653229@qq.com"})

  r.GET("/jobrunner/json", JobJson)
  r.GET("/jobrunner/html", JobHtml)

  r.Run(":8888")
}

func JobJson(c *gin.Context) {
  c.JSON(200, jobrunner.StatusJson())
}

func JobHtml(c *gin.Context) {
  t, err := template.ParseFiles(os.Getenv("GOPATH") + "/src/github.com/bamzi/jobrunner/views/Status.html")
  if err != nil {
    c.JSON(400, "error")
  }
  t.Execute(c.Writer, jobrunner.StatusPage())
}
```

运行之后，在浏览器中输入 `http://localhost:8888/jobrunner/html` 查看任务状态：

JobRunner Status Report

ID	Name	Status	Last run	Next run	Latency
1	GreetingJob	IDLE	2020-09-02 21:53:26	2020-09-02 21:53:31	0s
2	EmailJob	IDLE	2020-09-02 21:53:21	2020-09-02 21:53:31	0s

这里显示任务名、任务 ID、状态、上次运行时间、下次运行时间以及处理延迟。

我们还可以通过 `http://localhost:8888/jobrunner/json` 获取原始 JSON 格式的数据自己处理：


```
Tree Chart JSON Input
{
  "jobrunner": [
    {
      "Id": 1,
      "JobRunner": {
        "Name": "GreetingJob",
        "Status": "IDLE",
        "Latency": "0s"
      },
      "Next": "2020-09-02T21:54:01+08:00",
      "Prev": "2020-09-02T21:53:56+08:00"
    },
    {
      "Id": 2,
      "JobRunner": {
        "Name": "EmailJob",
        "Status": "IDLE",
        "Latency": "0s"
      },
      "Next": "2020-09-02T21:54:01+08:00",
      "Prev": "2020-09-02T21:53:51+08:00"
    }
  ]
}
```


总结

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. jobrunner GitHub：https://github.com/bamzi/jobrunner
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

mapstructure

简介

`mapstructure` 用于将通用的 `map[string]interface{}` 解码到对应的 Go 结构体中，或者执行相反的操作。很多时候，解析来自多种源头的数据流时，我们一般事先并不知道他们对应的具体类型。只有读取到一些字段之后才能做出判断。这时，我们可以先使用标准的 `encoding/json` 库将数据解码为 `map[string]interface{}` 类型，然后根据标识字段利用 `mapstructure` 库转为相应的 Go 结构体以便使用。

快速使用

本文代码采用 Go Modules。

首先创建目录并初始化：

```
$ mkdir mapstructure && cd mapstructure

$ go mod init github.com/darjun/go-daily-lib/mapstructure
```

下载 `mapstructure` 库：

```
$ go get github.com/mitchellh/mapstructure
```

使用：

```
package main

import (
  "encoding/json"
  "fmt"
  "log"

  "github.com/mitchellh/mapstructure"
)

type Person struct {
  Name string
  Age  int
  Job  string
```

```
}

type Cat struct {
  Name  string
  Age   int
  Breed string
}

func main() {
  datas := []string{`
    {
      "type": "person",
      "name":"dj",
      "age":18,
      "job": "programmer"
    }
  `,
    `
    {
      "type": "cat",
      "name": "kitty",
      "age": 1,
      "breed": "Ragdoll"
    }
  `,
  }

  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }

    switch m["type"].(string) {
    case "person":
      var p Person
      mapstructure.Decode(m, &p)
      fmt.Println("person", p)

    case "cat":
      var cat Cat
      mapstructure.Decode(m, &cat)
      fmt.Println("cat", cat)
    }
```

```
  }
}
```

运行结果：

```
$ go run main.go
person {dj 18 programmer}
cat {kitty 1 Ragdoll}
```

我们定义了两个结构体 `Person` 和 `Cat` ，他们的字段有些许不同。现在，我们约定通信的 JSON 串中有一个 `type` 字段。当 `type` 的值为 `person` 时，该 JSON 串表示的是 `Person` 类型的数据。当 `type` 的值为 `cat` 时，该 JSON 串表示的是 `Cat` 类型的数据。

上面代码中，我们先用 `json.Unmarshal` 将字节流解码为 `map[string]interface{}` 类型。然后读取里面的 `type` 字段。根据 `type` 字段的值，再使用 `mapstructure.Decode` 将该 JSON 串分别解码为 `Person` 和 `Cat` 类型的值，并输出。

实际上，Google Protobuf 通常也使用这种方式。在协议中添加消息 ID 或**全限定消息名**。接收方收到数据后，先读取协议 ID 或**全限定消息名**。然后调用 Protobuf 的解码方法将其解码为对应的 `Message` 结构。从这个角度来看， `mapstructure` 也可以用于网络消息解码，**如果你不考虑性能的话😄**。

字段标签

默认情况下， `mapstructure` 使用结构体中字段的名称做这个映射，例如我们的结构体有一个 `Name` 字段， `mapstructure` 解码时会在 `map[string]interface{}` 中查找键名 `name` 。注意，这里的 `name` 是大小写不敏感的！

```
type Person struct {
  Name string
}
```

当然，我们也可以指定映射的字段名。为了做到这一点，我们需要为字段设置 `mapstructure` 标签。例如下面使用 `username` 代替上例中的 `name` ：

```
type Person struct {
  Name string `mapstructure:"username"`
}
```

看示例：

```
type Person struct {
  Name string `mapstructure:"username"`
  Age  int
  Job  string
}

type Cat struct {
  Name  string
  Age   int
  Breed string
}

func main() {
  datas := []string{`
    {
      "type": "person",
      "username":"dj",
      "age":18,
      "job": "programmer"
    }
  `,
    `
    {
      "type": "cat",
      "name": "kitty",
      "Age": 1,
      "breed": "Ragdoll"
    }
  `,
    `
    {
      "type": "cat",
      "Name": "rooooose",
      "age": 2,
      "breed": "shorthair"
    }
  `,
  }

  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }
```

```
    switch m["type"].(string) {
    case "person":
      var p Person
      mapstructure.Decode(m, &p)
      fmt.Println("person", p)

    case "cat":
      var cat Cat
      mapstructure.Decode(m, &cat)
      fmt.Println("cat", cat)
    }
  }
}
```

上面代码中，我们使用标签 `mapstructure:"username"` 将 `Person` 的 `Name` 字段映射为 `username` ，在 JSON 串中我们需要设置 `username` 才能正确解析。另外，注意到，我们将第二个 JSON 串中的 `Age` 和第三个 JSON 串中的 `Name` 首字母大写了，但是并没有影响解码结果。 `mapstructure` 处理字段映射是大小写不敏感的。

内嵌结构

结构体可以任意嵌套，嵌套的结构被认为是拥有该结构体名字的另一个字段。例如，下面两种 `Friend` 的定义方式对于 `mapstructure` 是一样的：

```
type Person struct {
  Name string
}

// 方式一
type Friend struct {
  Person
}

// 方式二
type Friend struct {
  Person Person
}
```

为了正确解码， `Person` 结构的数据要在 `person` 键下：

```
map[string]interface{} {
  "person": map[string]interface{}{"name": "dj"},
}
```

我们也可以设置 `mapstructure:",squash"` 将该结构体的字段提到父结构中：

```
type Friend struct {
  Person `mapstructure:",squash"`
}
```

这样只需要这样的 JSON 串，无效嵌套 `person` 键：

```
map[string]interface{} {
  "name": "dj",
}
```

看示例：

```
type Person struct {
  Name string
}

type Friend1 struct {
  Person
}

type Friend2 struct {
  Person `mapstructure:",squash"`
}

func main() {
  datas := []string{`
    {
      "type": "friend1",
      "person": {
        "name":"dj"
      }
    }
  `,
    `
    {
      "type": "friend2",
      "name": "dj2"
    }
  `,
  }
```

```
  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }

    switch m["type"].(string) {
    case "friend1":
      var f1 Friend1
      mapstructure.Decode(m, &f1)
      fmt.Println("friend1", f1)

    case "friend2":
      var f2 Friend2
      mapstructure.Decode(m, &f2)
      fmt.Println("friend2", f2)
    }
  }
}
```

注意对比 `Friend1` 和 `Friend2` 使用的 JSON 串的不同。

另外需要注意一点，如果父结构体中有同名的字段，那么 `mapstructure` 会将JSON 中对应的值**同时设置到这两个字段中**，即这两个字段有相同的值。

未映射的值

如果源数据中有未映射的值（即结构体中无对应的字段）， `mapstructure` 默认会忽略它。

我们可以在结构体中定义一个字段，为其设置 `mapstructure:",remain"` 标签。这样未映射的值就会添加到这个字段中。注意，这个字段的类型只能为 `map[string]interface{}` 或 `map[interface{}]interface{}` 。

看示例：

```
type Person struct {
  Name  string
  Age   int
  Job   string
  Other map[string]interface{} `mapstructure:",remain"`
}

func main() {
  data := `
```

```
    {
      "name": "dj",
      "age":18,
      "job":"programmer",
      "height":"1.8m",
      "handsome": true
    }
  `

  var m map[string]interface{}
  err := json.Unmarshal([]byte(data), &m)
  if err != nil {
    log.Fatal(err)
  }

  var p Person
  mapstructure.Decode(m, &p)
  fmt.Println("other", p.Other)
}
```

上面代码中，我们为结构体定义了一个 `Other` 字段，用于保存未映射的键值。输出结果：

```
other map[handsome:true height:1.8m]
```

逆向转换

前面我们都是将 `map[string]interface{}` 解码到 Go 结构体中。 `mapstructure` 当然也可以将 Go 结构体反向解码为 `map[string]interface{}` 。在反向解码时，我们可以为某些字段设置 `mapstructure:",omitempty"` 。这样当这些字段为默认值时，就不会出现在结构的 `map[string]interface{}` 中：

```
type Person struct {
  Name string
  Age  int
  Job  string `mapstructure:",omitempty"`
}

func main() {
  p := &Person{
    Name: "dj",
    Age:  18,
  }

  var m map[string]interface{}
```

```
  mapstructure.Decode(p, &m)

  data, _ := json.Marshal(m)
  fmt.Println(string(data))
}
```

上面代码中，我们为 `Job` 字段设置了 `mapstructure:",omitempty"` ，且对象 `p` 的 `Job` 字段未设置。运行结果：

```
$ go run main.go
{"Age":18,"Name":"dj"}
```

`Metadata`

解码时会产生一些有用的信息， `mapstructure` 可以使用 `Metadata` 收集这些信息。 `Metadata` 结构如下：

```
// mapstructure.go
type Metadata struct {
  Keys   []string
  Unused []string
}
```

`Metadata` 只有两个导出字段：

- `Keys` ：解码成功的键名；
- `Unused` ：在源数据中存在，但是目标结构中不存在的键名。

为了收集这些数据，我们需要使用 `DecodeMetadata` 来代替 `Decode` 方法：

```
type Person struct {
  Name string
  Age  int
}

func main() {
  m := map[string]interface{}{
    "name": "dj",
    "age":  18,
    "job":  "programmer",
  }

  var p Person
```

```
  var metadata mapstructure.Metadata
  mapstructure.DecodeMetadata(m, &p, &metadata)

  fmt.Printf("keys:%#v unused:%#v\n", metadata.Keys, metadata.Unused)
}
```

先定义一个 `Metadata` 结构，传入 `DecodeMetadata` 收集解码的信息。运行结果：

```
$ go run main.go
keys:[]string{"Name", "Age"} unused:[]string{"job"}
```

错误处理

`mapstructure` 执行转换的过程中不可避免地会产生错误，例如 JSON 中某个键的类型与对应 Go 结构体中的字段类型不一致。 `Decode/DecodeMetadata` 会返回这些错误：

```
type Person struct {
  Name   string
  Age    int
  Emails []string
}

func main() {
  m := map[string]interface{}{
    "name":   123,
    "age":    "bad value",
    "emails": []int{1, 2, 3},
  }

  var p Person
  err := mapstructure.Decode(m, &p)
  if err != nil {
    fmt.Println(err.Error())
  }
}
```

上面代码中，结构体中 `Person` 中字段 `Name` 为 `string` 类型，但输入中 `name` 为 `int` 类型；字段 `Age` 为 `int` 类型，但输入中 `age` 为 `string` 类型；字段 `Emails` 为 `[]string` 类型，但输入中 `emails` 为 `[]int` 类型。故 `Decode` 返回错误。运行结果：

```
$ go run main.go
5 error(s) decoding:
```

```
* 'Age' expected type 'int', got unconvertible type 'string'
* 'Emails[0]' expected type 'string', got unconvertible type 'int'
* 'Emails[1]' expected type 'string', got unconvertible type 'int'
* 'Emails[2]' expected type 'string', got unconvertible type 'int'
* 'Name' expected type 'string', got unconvertible type 'int'
```

从错误信息中很容易看出哪里出错了。

弱类型输入

有时候，我们并不想对结构体字段类型和 `map[string]interface{}` 的对应键值做强类型一致的校验。这时可以使用 `WeakDecode/WeakDecodeMetadata` 方法，它们会尝试做类型转换：

```go
type Person struct {
  Name   string
  Age    int
  Emails []string
}

func main() {
  m := map[string]interface{}{
    "name":   123,
    "age":    "18",
    "emails": []int{1, 2, 3},
  }

  var p Person
  err := mapstructure.WeakDecode(m, &p)
  if err == nil {
    fmt.Println("person:", p)
  } else {
    fmt.Println(err.Error())
  }
}
```

虽然键 `name` 对应的值 `123` 是 `int` 类型，但是在 `WeakDecode` 中会将其转换为 `string` 类型以匹配 `Person.Name` 字段的类型。同样的， `age` 的值 `"18"` 是 `string` 类型，在 `WeakDecode` 中会将其转换为 `int` 类型以匹配 `Person.Age` 字段的类型。
需要注意一点，如果类型转换失败了， `WeakDecode` 同样会返回错误。例如将上例中的 `age` 设置为 `"bad value"` ，它就不能转为 `int` 类型，故而返回错误。

解码器

除了上面介绍的方法外， `mapstructure` 还提供了更灵活的解码器（ `Decoder` ）。可以通过配置 `DecoderConfig` 实现上面介绍的任何功能：

```go
// mapstructure.go
type DecoderConfig struct {
    ErrorUnused       bool
    ZeroFields        bool
    WeaklyTypedInput  bool
    Metadata          *Metadata
    Result            interface{}
    TagName           string
}
```

各个字段含义如下：

- `ErrorUnused` ：为 `true` 时，如果输入中的键值没有与之对应的字段就返回错误；
- `ZeroFields` ：为 `true` 时，在 `Decode` 前清空目标 `map` 。为 `false` 时，则执行的是 `map` 的合并。用在 `struct` 到 `map` 的转换中；
- `WeaklyTypedInput` ：实现 `WeakDecode/WeakDecodeMetadata` 的功能；
- `Metadata` ：不为 `nil` 时，收集 `Metadata` 数据；
- `Result` ：为结果对象，在 `map` 到 `struct` 的转换中， `Result` 为 `struct` 类型。在 `struct` 到 `map` 的转换中， `Result` 为 `map` 类型；
- `TagName` ：默认使用 `mapstructure` 作为结构体的标签名，可以通过该字段设置。

看示例：

```go
type Person struct {
  Name string
  Age  int
}

func main() {
  m := map[string]interface{}{
    "name": 123,
    "age":  "18",
    "job":  "programmer",
  }
```

```
  var p Person
  var metadata mapstructure.Metadata

  decoder, err := mapstructure.NewDecoder(&mapstructure.DecoderConfig{
    WeaklyTypedInput: true,
    Result:           &p,
    Metadata:         &metadata,
  })

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

  err = decoder.Decode(m)
  if err == nil {
    fmt.Println("person:", p)
    fmt.Printf("keys:%#v, unused:%#v\n", metadata.Keys, metadata.Unused)
  } else {
    fmt.Println(err.Error())
  }
}
```

这里用 `Decoder` 的方式实现了前面弱类型输入小节中的示例代码。实际上 `WeakDecode` 内部就是通过这种方式实现的，下面是 `WeakDecode` 的源码：

```
// mapstructure.go
func WeakDecode(input, output interface{}) error {
  config := &DecoderConfig{
    Metadata:         nil,
    Result:           output,
    WeaklyTypedInput: true,
  }

  decoder, err := NewDecoder(config)
  if err != nil {
    return err
  }

  return decoder.Decode(input)
}
```

再实际上， `Decode/DecodeMetadata/WeakDecodeMetadata` 内部都是先设置 `DecoderConfig` 的对应字段，然后创建 `Decoder` 对象，最后调用其 `Decode` 方法实现的。

总结

`mapstructure`实现优雅，功能丰富，代码结构清晰，非常推荐一看！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. mapstructure GitHub：https://github.com/mitchellh/mapstructure
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

cron

简介

`cron` 一个用于管理定时任务的库，用 Go 实现 Linux 中 `crontab` 这个命令的效果。之前我们也介绍过一个类似的 Go 库—— `gron` 。 `gron` 代码小巧，用于学习是比较好的。但是它功能相对简单些，并且已经不维护了。如果有定时任务需求，还是建议使用 `cron` 。

快速使用

文本代码使用 Go Modules。

创建目录并初始化：

```
$ mkdir cron && cd cron
$ go mod init github.com/darjun/go-daily-lib/cron
```

安装 `cron` ，目前最新稳定版本为 v3：

```
$ go get -u github.com/robfig/cron/v3
```

使用：

```
package main

import (
  "fmt"
  "time"

  "github.com/robfig/cron/v3"
)

func main() {
  c := cron.New()

  c.AddFunc("@every 1s", func() {
    fmt.Println("tick every 1 second")
  })
```

```
  c.Start()
  time.Sleep(time.Second * 5)
}
```

使用非常简单，创建 `cron` 对象，这个对象用于管理定时任务。

调用 `cron` 对象的 `AddFunc()` 方法向管理器中添加定时任务。 `AddFunc()` 接受两个参数，参数 1 以字符串形式指定触发时间规则，参数 2 是一个无参的函数，每次触发时调用。 `@every 1s` 表示每秒触发一次， `@every` 后加一个时间间隔，表示每隔多长时间触发一次。例如 `@every 1h` 表示每小时触发一次， `@every 1m2s` 表示每隔 1 分 2 秒触发一次。 `time.ParseDuration()` 支持的格式都可以用在这里。

调用 `c.Start()` 启动定时循环。

注意一点，因为 `c.Start()` 启动一个新的 goroutine 做循环检测，我们在代码最后加了一行 `time.Sleep(time.Second * 5)` 防止主 goroutine 退出。

运行效果，每隔 1s 输出一行字符串：

```
$ go run main.go
tick every 1 second
tick every 1 second
tick every 1 second
tick every 1 second
tick every 1 second
```

时间格式

与Linux 中 `crontab` 命令相似， `cron` 库支持用 **5** 个空格分隔的域来表示时间。这 5 个域含义依次为：

- `Minutes` ：分钟，取值范围 `[0-59]` ，支持特殊字符 `* / , -` ；
- `Hours` ：小时，取值范围 `[0-23]` ，支持特殊字符 `* / , -` ；
- `Day of month` ：每月的第几天，取值范围 `[1-31]` ，支持特殊字符 `* / , - ?` ；
- `Month` ：月，取值范围 `[1-12]` 或者使用月份名字缩写 `[JAN-DEC]` ，支持特殊字符 `* / , -` ；
- `Day of week` ：周历，取值范围 `[0-6]` 或名字缩写 `[JUN-SAT]` ，支持特殊字符 `* / , - ?` 。

注意，月份和周历名称都是不区分大小写的，也就是说 `SUN/Sun/sun` 表示同样的含义（都是周日）。

特殊字符含义如下：

- `*`：使用`*`的域可以匹配任何值，例如将月份域（第 4 个）设置为`*`，表示每个月；
- `/`：用来指定范围的**步长**，例如将小时域（第 2 个）设置为`3-59/15`表示第 3 分钟触发，以后每隔 15 分钟触发一次，因此第 2 次触发为第 18 分钟，第 3 次为 33 分钟。。。直到分钟大于 59；
- `,`：用来列举一些离散的值和多个范围，例如将周历的域（第 5 个）设置为`MON,WED,FRI`表示周一、三和五；
- `-`：用来表示范围，例如将小时的域（第 1 个）设置为`9-17`表示上午 9 点到下午 17 点（包括 9 和 17）；
- `?`：只能用在月历和周历的域中，用来代替`*`，表示每月/周的任意一天。

了解规则之后，我们可以定义任意时间：

- `30 * * * *`：分钟域为 30，其他域都是`*`表示任意。每小时的 30 分触发；
- `30 3-6,20-23 * * *`：分钟域为 30，小时域的`3-6,20-23`表示 3 点到 6 点和 20 点到 23 点。3,4,5,6,20,21,22,23 时的 30 分触发；
- `0 0 1 1 *`：1（第 4 个） 月 1（第 3 个） 号的 0（第 2 个） 时 0（第 1 个） 分触发。

记熟了这几个域的顺序，再多练习几次很容易就能掌握格式。熟悉规则了之后，就能熟练使用 `crontab` 命令了。

```
func main() {
  c := cron.New()

  c.AddFunc("30 * * * *", func() {
    fmt.Println("Every hour on the half hour")
  })

  c.AddFunc("30 3-6,20-23 * * *", func() {
    fmt.Println("On the half hour of 3-6am, 8-11pm")
  })

  c.AddFunc("0 0 1 1 *", func() {
    fmt.Println("Jun 1 every year")
```

```
    })

    c.Start()

    for {
      time.Sleep(time.Second)
    }
}
```

预定义时间规则

为了方便使用， `cron` 预定义了一些时间规则：

- `@yearly` ：也可以写作 `@annually` ，表示每年第一天的 0 点。等价于 `0 0 1 1 *` ；
- `@monthly` ：表示每月第一天的 0 点。等价于 `0 0 1 * *` ；
- `@weekly` ：表示每周第一天的 0 点，注意第一天为周日，即周六结束，周日开始的那个 0 点。等价于 `0 0 * * 0` ；
- `@daily` ：也可以写作 `@midnight` ，表示每天 0 点。等价于 `0 0 * * *` ；
- `@hourly` ：表示每小时的开始。等价于 `0 * * * *` 。

例如：

```
func main() {
  c := cron.New()

  c.AddFunc("@hourly", func() {
    fmt.Println("Every hour")
  })

  c.AddFunc("@daily", func() {
    fmt.Println("Every day on midnight")
  })

  c.AddFunc("@weekly", func() {
    fmt.Println("Every week")
  })

  c.Start()

  for {
    time.Sleep(time.Second)
```

```
    }
}
```

上面代码只是演示用法，实际运行可能要等待非常长的时间才能有输出。

固定时间间隔

`cron` 支持固定时间间隔，格式为：

```
@every <duration>
```

含义为每隔 `duration` 触发一次。`<duration>` 会调用 `time.ParseDuration()` 函数解析，所以 `ParseDuration` 支持的格式都可以。例如 `1h30m10s` 。在快速开始部分，我们已经演示了 `@every` 的用法了，这里就不赘述了。

时区

默认情况下，所有时间都是基于当前时区的。当然我们也可以指定时区，有 2 两种方式：

- 在时间字符串前面添加一个 `CRON_TZ=` + 具体时区，具体时区的格式在之前 `carbon` 的文章中有详细介绍。东京时区为 `Asia/Tokyo` ，纽约时区为 `America/New_York` ；
- 创建 `cron` 对象时增加一个时区选项 `cron.WithLocation(location)` ，`location` 为 `time.LoadLocation(zone)` 加载的时区对象，`zone` 为具体的时区格式。或者调用已创建好的 `cron` 对象的 `SetLocation()` 方法设置时区。

示例：

```go
func main() {
  nyc, _ := time.LoadLocation("America/New_York")
  c := cron.New(cron.WithLocation(nyc))
  c.AddFunc("0 6 * * ?", func() {
    fmt.Println("Every 6 o'clock at New York")
  })

  c.AddFunc("CRON_TZ=Asia/Tokyo 0 6 * * ?", func() {
    fmt.Println("Every 6 o'clock at Tokyo")
  })

  c.Start()
```

```
  for {
    time.Sleep(time.Second)
  }
}
```

`Job` 接口

除了直接将无参函数作为回调外，`cron` 还支持 `Job` 接口：

```
// cron.go
type Job interface {
  Run()
}
```

我们定义一个实现接口 `Job` 的结构：

```
type GreetingJob struct {
  Name string
}

func (g GreetingJob) Run() {
  fmt.Println("Hello ", g.Name)
}
```

调用 `cron` 对象的 `AddJob()` 方法将 `GreetingJob` 对象添加到定时管理器中：

```
func main() {
  c := cron.New()
  c.AddJob("@every 1s", GreetingJob{"dj"})
  c.Start()

  time.Sleep(5 * time.Second)
}
```

运行效果：

```
$ go run main.go
Hello  dj
Hello  dj
Hello  dj
```

```
Hello  dj
Hello  dj
```

使用自定义的结构可以让任务携带状态（ `Name` 字段）。

实际上 `AddFunc()` 方法内部也调用了 `AddJob()` 方法。首先， `cron` 基于 `func()` 类型定义一个新的类型 `FuncJob` ：

```
// cron.go
type FuncJob func()
```

然后让 `FuncJob` 实现 `Job` 接口：

```
// cron.go
func (f FuncJob) Run() {
  f()
}
```

在 `AddFunc()` 方法中，将传入的回调转为 `FuncJob` 类型，然后调用 `AddJob()` 方法：

```
func (c *Cron) AddFunc(spec string, cmd func()) (EntryID, error) {
  return c.AddJob(spec, FuncJob(cmd))
}
```

线程安全

`cron` 会创建一个新的 goroutine 来执行触发回调。如果这些回调需要并发访问一些资源、数据，我们需要显式地做同步。

自定义时间格式

`cron` 支持灵活的时间格式，如果默认的格式不能满足要求，我们可以自己定义时间格式。时间规则字符串需要 `cron.Parser` 对象来解析。我们先来看看默认的解析器是如何工作的。

首先定义各个域：

```
// parser.go
const (
  Second         ParseOption = 1 << iota
  SecondOptional
```

```
    Minute
    Hour
    Dom
    Month
    Dow
    DowOptional
    Descriptor
)
```

除了 `Minute/Hour/Dom(Day of month)/Month/Dow(Day of week)` 外，还可以支持 `Second` 。相对顺序都是固定的：

```
// parser.go
var places = []ParseOption{
    Second,
    Minute,
    Hour,
    Dom,
    Month,
    Dow,
}

var defaults = []string{
    "0",
    "0",
    "0",
    "*",
    "*",
    "*",
}
```

默认的时间格式使用 5 个域。

我们可以调用 `cron.NewParser()` 创建自己的 `Parser` 对象，以位格式传入使用哪些域，例如下面的 `Parser` 使用 6 个域，支持 `Second` （秒）：

```
parser := cron.NewParser(
    cron.Second | cron.Minute | cron.Hour | cron.Dom | cron.Month | cron.Dow | cron.Descriptor,
)
```

调用 `cron.WithParser(parser)` 创建一个选项传入构造函数 `cron.New()` ，使用时就可以指定秒了：

```
c := cron.New(cron.WithParser(parser))
c.AddFunc("1 * * * * *", func () {
  fmt.Println("every 1 second")
})
c.Start()
```

这里时间格式必须使用 6 个域，顺序与上面的 `const` 定义一致。

因为上面的时间格式太常见了， `cron` 定义了一个便捷的函数：

```
// option.go
func WithSeconds() Option {
  return WithParser(NewParser(
    Second | Minute | Hour | Dom | Month | Dow | Descriptor,
  ))
}
```

注意 `Descriptor` 表示对 `@every/@hour` 等的支持。有了 `WithSeconds()` ，我们不用手动创建 `Parser` 对象了：

```
c := cron.New(cron.WithSeconds())
```

选项

`cron` 对象创建使用了选项模式，我们前面已经介绍了 3 个选项：

- `WithLocation` ：指定时区；
- `WithParser` ：使用自定义的解析器；
- `WithSeconds` ：让时间格式支持秒，实际上内部调用了 `WithParser` 。

`cron` 还提供了另外两种选项：

- `WithLogger` ：自定义 `Logger` ；
- `WithChain` ：Job 包装器。

`WithLogger`

`WithLogger` 可以设置 `cron` 内部使用我们自定义的 `Logger` ：

```
func main() {
  c := cron.New(
```

```
    cron.WithLogger(
      cron.VerbosePrintfLogger(log.New(os.Stdout, "cron: ", log.LstdFlags)))
  c.AddFunc("@every 1s", func() {
    fmt.Println("hello world")
  })
  c.Start()

  time.Sleep(5 * time.Second)
}
```

上面调用 `cron.VerbosPrintfLogger()` 包装 `log.Logger` ，这个 `logger` 会详细记录 `cron` 内部的调度过程：

```
$ go run main.go
cron: 2020/06/26 07:09:14 start
cron: 2020/06/26 07:09:14 schedule, now=2020-06-26T07:09:14+08:00, entry=1, next
=2020-06-26T07:09:15+08:00
cron: 2020/06/26 07:09:15 wake, now=2020-06-26T07:09:15+08:00
cron: 2020/06/26 07:09:15 run, now=2020-06-26T07:09:15+08:00, entry=1, next=2020
-06-26T07:09:16+08:00
hello world
cron: 2020/06/26 07:09:16 wake, now=2020-06-26T07:09:16+08:00
cron: 2020/06/26 07:09:16 run, now=2020-06-26T07:09:16+08:00, entry=1, next=2020
-06-26T07:09:17+08:00
hello world
cron: 2020/06/26 07:09:17 wake, now=2020-06-26T07:09:17+08:00
cron: 2020/06/26 07:09:17 run, now=2020-06-26T07:09:17+08:00, entry=1, next=2020
-06-26T07:09:18+08:00
hello world
cron: 2020/06/26 07:09:18 wake, now=2020-06-26T07:09:18+08:00
hello world
cron: 2020/06/26 07:09:18 run, now=2020-06-26T07:09:18+08:00, entry=1, next=2020
-06-26T07:09:19+08:00
cron: 2020/06/26 07:09:19 wake, now=2020-06-26T07:09:19+08:00
hello world
cron: 2020/06/26 07:09:19 run, now=2020-06-26T07:09:19+08:00, entry=1, next=2020
-06-26T07:09:20+08:0
```

我们看看默认的 `Logger` 是什么样的：

```
// logger.go
var DefaultLogger Logger = PrintfLogger(log.New(os.Stdout, "cron: ", log.LstdFla
gs))

func PrintfLogger(l interface{ Printf(string, ...interface{}) }) Logger {
```

```
  return printfLogger{l, false}
}

func VerbosePrintfLogger(l interface{ Printf(string, ...interface{}) }) Logger {
  return printfLogger{l, true}
}

type printfLogger struct {
  logger  interface{ Printf(string, ...interface{}) }
  logInfo bool
}
```

`WithChain`

Job 包装器可以在执行实际的 `Job` 前后添加一些逻辑：

- 捕获 `panic`；
- 如果 `Job` 上次运行还未结束，推迟本次执行;
- 如果 `Job` 上次运行还未介绍，跳过本次执行；
- 记录每个 `Job` 的执行情况。

我们可以将 `Chain` 类比为 Web 处理器的中间件。实际上就是在 `Job` 的执行逻辑外在封装一层逻辑。我们的封装逻辑需要写成一个函数，传入一个 `Job` 类型，返回封装后的 `Job` 。 `cron` 为这种函数定义了一个类型 `JobWrapper` ：

```
// chain.go
type JobWrapper func(Job) Job
```

然后使用一个 `Chain` 对象将这些 `JobWrapper` 组合到一起：

```
type Chain struct {
  wrappers []JobWrapper
}

func NewChain(c ...JobWrapper) Chain {
  return Chain{c}
}
```

调用 `Chain` 对象的 `Then(job)` 方法应用这些 `JobWrapper` ，返回最终的`Job：

```
func (c Chain) Then(j Job) Job {
  for i := range c.wrappers {
    j = c.wrappers[len(c.wrappers)-i-1](j)
```

```
	}
	return j
}
```

注意应用 `JobWrapper` 的顺序。

内置 `JobWrapper`

`cron` 内置了 3 个用得比较多的 `JobWrapper`：

- `Recover`：捕获内部 `Job` 产生的 panic；
- `DelayIfStillRunning`：触发时，如果上一次任务还未执行完成（耗时太长），则等待上一次任务完成之后再执行；
- `SkipIfStillRunning`：触发时，如果上一次任务还未完成，则跳过此次执行。

下面分别介绍。

`Recover`

先看看如何使用：

```
type panicJob struct {
	count int
}

func (p *panicJob) Run() {
	p.count++
	if p.count == 1 {
		panic("oooooooooooooooops!!!")
	}

	fmt.Println("hello world")
}

func main() {
	c := cron.New()
	c.AddJob("@every 1s", cron.NewChain(cron.Recover(cron.DefaultLogger)).Then(&panicJob{}))
	c.Start()

	time.Sleep(5 * time.Second)
}
```

`panicJob` 在第一次触发时，触发了 `panic` 。因为有 `cron.Recover()` 保护，后续任务还能执行：

```
go run main.go
cron: 2020/06/27 14:02:00 panic, error=oooooooooooooops!!!, stack=...
goroutine 18 [running]:
github.com/robfig/cron/v3.Recover.func1.1.1(0x514ee0, 0xc0000044a0)
        D:/code/golang/pkg/mod/github.com/robfig/cron/v3@v3.0.1/chain.go:45 +0xb
c
panic(0x4cf380, 0x513280)
        C:/Go/src/runtime/panic.go:969 +0x174
main.(*panicJob).Run(0xc0000140e8)
        D:/code/golang/src/github.com/darjun/go-daily-lib/cron/recover/main.go:1
7 +0xba
github.com/robfig/cron/v3.Recover.func1.1()
        D:/code/golang/pkg/mod/github.com/robfig/cron/v3@v3.0.1/chain.go:53 +0x6
f
github.com/robfig/cron/v3.FuncJob.Run(0xc000070390)
        D:/code/golang/pkg/mod/github.com/robfig/cron/v3@v3.0.1/cron.go:136 +0x2
c
github.com/robfig/cron/v3.(*Cron).startJob.func1(0xc00005c0a0, 0x514d20, 0xc0000
70390)
        D:/code/golang/pkg/mod/github.com/robfig/cron/v3@v3.0.1/cron.go:312 +0x6
8
created by github.com/robfig/cron/v3.(*Cron).startJob
        D:/code/golang/pkg/mod/github.com/robfig/cron/v3@v3.0.1/cron.go:310 +0x7
a
hello world
hello world
hello world
hello world
```

我们看看 `cron.Recover()` 的实现，很简单：

```go
// cron.go
func Recover(logger Logger) JobWrapper {
  return func(j Job) Job {
    return FuncJob(func() {
      defer func() {
        if r := recover(); r != nil {
          const size = 64 << 10
          buf := make([]byte, size)
          buf = buf[:runtime.Stack(buf, false)]
          err, ok := r.(error)
          if !ok {
```

```
                err = fmt.Errorf("%v", r)
            }
            logger.Error(err, "panic", "stack", "...\n"+string(buf))
        }
      }()
      j.Run()
    })
  }
}
```

就是在执行内层的 `Job` 逻辑前，添加 `recover()` 调用。如果 `Job.Run()` 执行过程中有 `panic` 。这里的 `recover()` 会捕获到，输出调用堆栈。

`DelayIfStillRunning`

还是先看如何使用：

```
type delayJob struct {
  count int
}

func (d *delayJob) Run() {
  time.Sleep(2 * time.Second)
  d.count++
  log.Printf("%d: hello world\n", d.count)
}

func main() {
  c := cron.New()
  c.AddJob("@every 1s", cron.NewChain(cron.DelayIfStillRunning(cron.DefaultLogge
r)).Then(&delayJob{}))
  c.Start()

  time.Sleep(10 * time.Second)
}
```

上面我们在 `Run()` 中增加了一个 2s 的延迟，输出中间隔变为 2s，而不是定时的 1s：

```
$ go run main.go
2020/06/27 14:11:16 1: hello world
2020/06/27 14:11:18 2: hello world
2020/06/27 14:11:20 3: hello world
2020/06/27 14:11:22 4: hello world
```

看看源码：

```go
// chain.go
func DelayIfStillRunning(logger Logger) JobWrapper {
  return func(j Job) Job {
    var mu sync.Mutex
    return FuncJob(func() {
      start := time.Now()
      mu.Lock()
      defer mu.Unlock()
      if dur := time.Since(start); dur > time.Minute {
        logger.Info("delay", "duration", dur)
      }
      j.Run()
    })
  }
}
```

首先定义一个该任务共用的互斥锁 `sync.Mutex` ，每次执行任务前获取锁，执行结束之后释放锁。所以在上一个任务结束前，下一个任务获取锁是无法成功的，从而保证的任务的串行执行。

`SkipIfStillRunning`

还是先看看如何使用：

```go
type skipJob struct {
  count int32
}

func (d *skipJob) Run() {
  atomic.AddInt32(&d.count, 1)
  log.Printf("%d: hello world\n", d.count)
  if atomic.LoadInt32(&d.count) == 1 {
    time.Sleep(2 * time.Second)
  }
}

func main() {
  c := cron.New()
  c.AddJob("@every 1s", cron.NewChain(cron.SkipIfStillRunning(cron.DefaultLogger)).Then(&skipJob{}))
  c.Start()

  time.Sleep(10 * time.Second)
}
```

输出：

```
$ go run main.go
2020/06/27 14:22:07 1: hello world
2020/06/27 14:22:10 2: hello world
2020/06/27 14:22:11 3: hello world
2020/06/27 14:22:12 4: hello world
2020/06/27 14:22:13 5: hello world
2020/06/27 14:22:14 6: hello world
2020/06/27 14:22:15 7: hello world
2020/06/27 14:22:16 8: hello world
```

注意观察时间，第一个与第二个输出之间相差 3s，因为跳过了两次执行。

注意 `DelayIfStillRunning` 与 `SkipIfStillRunning` 是有本质上的区别的，前者 `DelayIfStillRunning` 只要时间足够长，所有的任务都会按部就班地完成，只是可能前一个任务耗时过长，导致后一个任务的执行时间推迟了一点。 `SkipIfStillRunning` 会跳过一些执行。

看看源码：

```
func SkipIfStillRunning(logger Logger) JobWrapper {
  return func(j Job) Job {
    var ch = make(chan struct{}, 1)
    ch <- struct{}{}
    return FuncJob(func() {
      select {
      case v := <-ch:
        j.Run()
        ch <- v
      default:
        logger.Info("skip")
      }
    })
  }
}
```

定义一个该任务共用的缓存大小为 1 的通道 `chan struct{}` 。执行任务时，从通道中取值，如果成功，执行，否则跳过。执行完成之后再向通道中发送一个值，确保下一个任务能执行。初始发送一个值到通道中，保证第一个任务的执行。

总结

`cron` 实现比较小巧，且优雅，代码行数也不多，非常值得一看！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. cron GitHub：https://github.com/robfig/cron
2. Go 每日一库之 carbon：https://darjun.github.io/2020/02/14/godailylib/carbon/
3. Go 每日一库之 gron：https://darjun.github.io/2020/04/20/godailylib/gron/
4. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

cli

简介

`cli` 是一个用于构建命令行程序的库。我们之前也介绍过一个用于构建命令行程序的库 `cobra` 。在功能上来说两者差不多， `cobra` 的优势是提供了一个脚手架，方便开发。 `cli` 非常简洁，所有的初始化操作就是创建一个 `cli.App` 结构的对象。通过为对象的字段赋值来添加相应的功能。

`cli` 与我们上一篇文章介绍的 `negroni` 是同一个作者urfave。

快速使用

`cli` 需要搭配 Go Modules 使用。创建目录并初始化：

```
$ mkdir cli && cd cli
$ go mod init github.com/darjun/go-daily-lib/cli
```

安装 `cli` 库，有 `v1` 和 `v2` 两个版本。如果没有特殊需求，一般安装 `v2` 版本：

```
$ go get -u github.com/urfave/cli/v2
```

使用：

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/urfave/cli/v2"
)

func main() {
  app := &cli.App{
    Name:  "hello",
    Usage: "hello world example",
    Action: func(c *cli.Context) error {
```

```
        fmt.Println("hello world")
        return nil
    },
  }

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

使用非常简单，理论上创建一个 `cli.App` 结构的对象，然后调用其 `Run()` 方法，传入命令行的参数即可。一个空白的 `cli` 应用程序如下：

```
func main() {
  (&cli.App{}).Run(os.Args)
}
```

但是这个空白程序没有什么用处。我们的 `hello world` 程序，设置了 `Name/Usage/Action` 。 `Name` 和 `Usage` 都显示在帮助中， `Action` 是调用该命令行程序时实际执行的函数，需要的信息可以从参数 `cli.Context` 获取。

编译、运行（环境：Win10 + Git Bash）：

```
$ go build -o hello
$ ./hello
hello world
```

除了这些， `cli` 为我们额外生成了帮助信息：

```
$ ./hello --help
NAME:
   hello - hello world example

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

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

GLOBAL OPTIONS:
   --help, -h  show help (default: false)
```

参数

通过 `cli.Context` 的相关方法我们可以获取传给命令行的参数信息：

- `NArg()` ：返回参数个数；
- `Args()` ：返回 `cli.Args` 对象，调用其 `Get(i)` 获取位置 `i` 上的参数。

示例：

```
func main() {
  app := &cli.App{
    Name:  "arguments",
    Usage: "arguments example",
    Action: func(c *cli.Context) error {
      for i := 0; i < c.NArg(); i++ {
        fmt.Printf("%d: %s\n", i+1, c.Args().Get(i))
      }
      return nil
    },
  }

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

这里只是简单输出：

```
$ go run main.go hello world
1: hello
2: world
```

选项

一个好用的命令行程序怎么会少了选项呢？ `cli` 设置和获取选项非常简单。在 `cli.App{}` 结构初始化时，设置字段 `Flags` 即可添加选项。 `Flags` 字段是 `[]cli.Flag` 类型， `cli.Flag` 实际上是接口类型。 `cli` 为常见类型都实现了对应的 `XxxFlag` ，如 `BoolFlag/DurationFlag/StringFlag` 等。它们有一些共用的字段， `Name/Value/Usage` （名称/默认值/释义）。看示例：

```
func main() {
  app := &cli.App{
    Flags: []cli.Flag{
      &cli.StringFlag{
        Name:  "lang",
        Value: "english",
        Usage: "language for the greeting",
      },
    },
    Action: func(c *cli.Context) error {
      name := "world"
      if c.NArg() > 0 {
        name = c.Args().Get(0)
      }

      if c.String("lang") == "english" {
        fmt.Println("hello", name)
      } else {
        fmt.Println("你好", name)
      }
      return nil
    },
  }

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

上面是一个打招呼的命令行程序，可通过选项 `lang` 指定语言，默认为英语。设置选项为非 `english` 的值，使用汉语。如果有参数，使用第一个参数作为人名，否则使用 `world` 。注意选项是通过 `c.Type(name)` 来获取的， `Type` 为选项类型， `name` 为选项名。编译、运行：

```
$ go build -o flags

# 默认调用
$ ./flags
hello world

# 设置非英语
$ ./flags --lang chinese
你好 world
```

```
# 传入参数作为人名
$ ./flags --lang chinese dj
你好 dj
```

我们可以通过 `./flags --help` 来查看选项：

```
$ ./flags --help
NAME:
   flags - A new cli application

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

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

GLOBAL OPTIONS:
   --lang value  language for the greeting (default: "english")
   --help, -h    show help (default: false)
```

存入变量

除了通过 `c.Type(name)` 来获取选项的值，我们还可以将选项存到某个预先定义好的变量中。只需要设置 `Destination` 字段为变量的地址即可：

```
func main() {
  var language string

  app := &cli.App{
    Flags: []cli.Flag{
      &cli.StringFlag{
        Name:        "lang",
        Value:       "english",
        Usage:       "language for the greeting",
        Destination: &language,
      },
    },
    Action: func(c *cli.Context) error {
      name := "world"
      if c.NArg() > 0 {
        name = c.Args().Get(0)
      }

      if language == "english" {
```

```
            fmt.Println("hello", name)
        } else {
            fmt.Println("你好", name)
        }
        return nil
    },
  }

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

与上面的程序效果是一样的。

占位值

`cli` 可以在 `Usage` 字段中为选项设置占位值，占位值通过反引号 ` 包围。只有第一个生效，其他的维持不变。占位值有助于生成易于理解的帮助信息：

```
func main() {
  app := & cli.App{
    Flags : []cli.Flag {
      &cli.StringFlag{
        Name:"config",
        Usage: "Load configuration from `FILE`",
      },
    },
  }

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

设置占位值之后，帮助信息中，该占位值会显示在对应的选项后面，对短选项也是有效的：

```
$ go build -o placeholder
$ ./placeholder --help
NAME:
   placeholder - A new cli application
```

```
USAGE:
   placeholder [global options] command [command options] [arguments...]

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

GLOBAL OPTIONS:
   --config FILE  Load configuration from FILE
   --help, -h     show help (default: false)
```

别名

选项可以设置多个别名，设置对应选项的 `Aliases` 字段即可：

```
func main() {
  app := &cli.App{
    Flags: []cli.Flag{
      &cli.StringFlag{
        Name:    "lang",
        Aliases: []string{"language", "l"},
        Value:   "english",
        Usage:   "language for the greeting",
      },
    },
    Action: func(c *cli.Context) error {
      name := "world"
      if c.NArg() > 0 {
        name = c.Args().Get(0)
      }

      if c.String("lang") == "english" {
        fmt.Println("hello", name)
      } else {
        fmt.Println("你好", name)
      }
      return nil
    },
  }

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

使用 `--lang chinese` 、 `--language chinese` 和 `-l chinese` 效果是一样的。如果通过不同的名称指定同一个选项，会报错：

```
$ go build -o aliase
$ ./aliase --lang chinese
你好 world
$ ./aliase --language chinese
你好 world
$ ./aliase -l chinese
你好 world
$ ./aliase -l chinese --lang chinese
Cannot use two forms of the same flag: l lang
```

环境变量

除了通过执行程序时手动指定命令行选项，我们还可以读取指定的环境变量作为选项的值。只需要将环境变量的名字设置到选项对象的 `EnvVars` 字段即可。可以指定多个环境变量名字， `cli` 会依次查找，第一个有值的环境变量会被使用。

```
func main() {
  app := &cli.App{
    Flags: []cli.Flag{
      &cli.StringFlag{
        Name:    "lang",
        Value:   "english",
        Usage:   "language for the greeting",
        EnvVars: []string{"APP_LANG", "SYSTEM_LANG"},
      },
    },
    Action: func(c *cli.Context) error {
      if c.String("lang") == "english" {
        fmt.Println("hello")
      } else {
        fmt.Println("你好")
      }
      return nil
    },
  }

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

编译、运行：

```
$ go build -o env
$ APP_LANG=chinese ./env
你好
```

文件

`cli` 还支持从文件中读取选项的值，设置选项对象的 `FilePath` 字段为文件路径：

```
func main() {
  app := &cli.App{
    Flags: []cli.Flag{
      &cli.StringFlag{
        Name:     "lang",
        Value:    "english",
        Usage:    "language for the greeting",
        FilePath: "./lang.txt",
      },
    },
    Action: func(c *cli.Context) error {
      if c.String("lang") == "english" {
        fmt.Println("hello")
      } else {
        fmt.Println("你好")
      }
      return nil
    },
  }

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

在 `main.go` 同级目录创建一个 `lang.txt` ，输入内容 `chinese` 。然后编译运行程序：

```
$ go build -o file
$ ./file
你好
```

`cli` 还支持从 `YAML/JSON/TOML` 等配置文件中读取选项值，这里就不一一介绍了。

选项优先级

上面我们介绍了几种设置选项值的方式，如果同时有多个方式生效，按照下面的优先级从高到低设置：

- 用户指定的命令行选项值；
- 环境变量；
- 配置文件；
- 选项的默认值。

组合短选项

我们时常会遇到有多个短选项的情况。例如 linux 命令 `ls -a -l` ，可以简写为 `ls -al` 。 `cli` 也支持短选项合写，只需要设置 `cli.App` 的 `UseShortOptionHandling` 字段为 `true` 即可：

```
func main() {
  app := &cli.App{
    UseShortOptionHandling: true,
    Commands: []*cli.Command{
      {
        Name:  "short",
        Usage: "complete a task on the list",
        Flags: []cli.Flag{
          &cli.BoolFlag{Name: "serve", Aliases: []string{"s"}},
          &cli.BoolFlag{Name: "option", Aliases: []string{"o"}},
          &cli.BoolFlag{Name: "message", Aliases: []string{"m"}},
        },
        Action: func(c *cli.Context) error {
          fmt.Println("serve:", c.Bool("serve"))
          fmt.Println("option:", c.Bool("option"))
          fmt.Println("message:", c.Bool("message"))
          return nil
        },
      },
    },
  }

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

```
	}
}
```

编译运行：

```
$ go build -o short
$ ./short short -som "some message"
serve: true
option: true
message: true
```

需要特别注意一点，设置 `UseShortOptionHandling` 为 `true` 之后，我们不能再通过 `-` 指定选项了，这样会产生歧义。例如 `-lang` ， `cli` 不知道应该解释为 `l/a/n/g` 4 个选项还是 `lang` 1 个。 `--` 还是有效的。

必要选项

如果将选项的 `Required` 字段设置为 `true` ，那么该选项就是必要选项。必要选项必须指定，否则会报错：

```
func main() {
	app := &cli.App{
		Flags: []cli.Flag{
			&cli.StringFlag{
				Name:     "lang",
				Value:    "english",
				Usage:    "language for the greeting",
				Required: true,
			},
		},
		Action: func(c *cli.Context) error {
			if c.String("lang") == "english" {
				fmt.Println("hello")
			} else {
				fmt.Println("你好")
			}
			return nil
		},
	}

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

```
    }
}
```

不指定选项 `lang` 运行：

```
$ ./required
2020/06/23 22:11:32 Required flag "lang" not set
```

帮助文本中的默认值

默认情况下，帮助文本中选项的默认值显示为 `Value` 字段值。有些时候，`Value` 并不是实际的默认值。这时，我们可以通过 `DefaultText` 设置：

```
func main() {
    app := &cli.App{
        Flags: []cli.Flag{
            &cli.IntFlag{
                Name:        "port",
                Value:       0,
                Usage:       "Use a randomized port",
                DefaultText :"random",
            },
        },
    }

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

上面代码逻辑中，如果 `Value` 设置为 0 就随机一个端口，这时帮助信息中 `default: 0` 就容易产生误解了。通过 `DefaultText` 可以避免这种情况：

```
$ go build -o default-text
$ ./default-text --help
NAME:
   default-text - A new cli application

USAGE:
   default-text [global options] command [command options] [arguments...]

COMMANDS:
```

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

GLOBAL OPTIONS:
   --port value  Use a randomized port (default: random)
   --help, -h    show help (default: false)
```

子命令

子命令使命令行程序有更好的组织性。 `git` 有大量的命令，很多以某个命令下的子命令存在。例如 `git remote` 命令下有 `add/rename/remove` 等子命令， `git submodule` 下有 `add/status/init/update` 等子命令。

`cli` 通过设置 `cli.App` 的 `Commands` 字段添加命令，设置各个命令的 `SubCommands` 字段，即可添加子命令。非常方便！

```
func main() {
  app := &cli.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)
  }
}
```

上面定义了 3 个命令 `add/complete/template` ， `template` 命令定义了 2 个子命令 `add/remove` 。编译、运行：

```
$ go build -o subcommand
$ ./subcommand add dating
added task:  dating
$ ./subcommand complete dating
completed task:  dating
$ ./subcommand template add alarm
new task template:  alarm
$ ./subcommand template remove alarm
removed task template:  alarm
```

注意一点，子命令默认不显示在帮助信息中，需要显式调用子命令所属命令的帮助（ `./subcommand template --help` ）：

```
$ ./subcommand --help
NAME:
   subcommand - A new cli application

USAGE:
```

```
    subcommand [global options] command [command options] [arguments...]

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 (default: false)

$ ./subcommand template --help
NAME:
    subcommand template - options for task templates

USAGE:
    subcommand template command [command options] [arguments...]

COMMANDS:
    add      add a new template
    remove   remove an existing template
    help, h  Shows a list of commands or help for one command

OPTIONS:
    --help, -h   show help (default: false)
```

分类

在子命令数量很多的时候，可以设置 `Category` 字段为它们分类，在帮助信息中会将相同分类的命令放在一起展示：

```
func main() {
  app := &cli.App{
    Commands: []*cli.Command{
      {
        Name:  "noop",
        Usage: "Usage for noop",
      },
      {
        Name:     "add",
        Category: "template",
        Usage:    "Usage for add",
      },
      {
        Name:     "remove",
```

```
                Category: "template",
                Usage:    "Usage for remove",
            },
        },
    }

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

编译、运行：

```
$ go build -o categories
$ ./categories --help
NAME:
   categories - A new cli application

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

COMMANDS:
   noop     Usage for noop
   help, h  Shows a list of commands or help for one command
   template:
     add     Usage for add
     remove  Usage for remove

GLOBAL OPTIONS:
   --help, -h  show help (default: false)
```

看上面的 `COMMANDS` 部分。

自定义帮助信息

在 `cli` 中所有的帮助信息文本都可以自定义，整个应用的帮助信息模板通过 `AppHelpTemplate` 指定。命令的帮助信息模板通过 `CommandHelpTemplate` 设置，子命令的帮助信息模板通过 `SubcommandHelpTemplate` 设置。甚至可以通过覆盖 `cli.HelpPrinter` 这个函数自己实现帮助信息输出。下面程序在默认的帮助信息后添加个人网站和微信信息：

```
func main() {
  cli.AppHelpTemplate = fmt.Sprintf(`%s
```

```
WEBSITE: http://darjun.github.io

WECHAT: GoUpUp`, cli.AppHelpTemplate)

  (&cli.App{}).Run(os.Args)
}
```

编译运行：

```
$ go build -o help
$ ./help --help
NAME:
   help - A new cli application

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

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

GLOBAL OPTIONS:
   --help, -h  show help (default: false)



WEBSITE: http://darjun.github.io

WECHAT: GoUpUp
```

我们还可以改写整个模板：

```
func main() {
  cli.AppHelpTemplate = `NAME:
  {{.Name}} - {{.Usage}}
USAGE:
  {{.HelpName}} {{if .VisibleFlags}}[global options]{{end}}{{if .Commands}} command [command options]{{end}} {{if .ArgsUsage}}{{.ArgsUsage}}{{else}}[arguments...]{{end}}
  {{if len .Authors}}
AUTHOR:
  {{range .Authors}}{{ . }}{{end}}
  {{end}}{{if .Commands}}
COMMANDS:
 {{range .Commands}}{{if not .HideHelp}}   {{join .Names ", "}}{{ "\t"}}{{.Usage}}{{ "\n" }}{{end}}{{end}}{{end}}{{if .VisibleFlags}}
```

```
GLOBAL OPTIONS:
   {{range .VisibleFlags}}{{.}}
   {{end}}{{end}}{{if .Copyright }}
COPYRIGHT:
   {{.Copyright}}
   {{end}}{{if .Version}}
VERSION:
   {{.Version}}
{{end}}
 `

  app := &cli.App{
    Authors: []*cli.Author{
       {
          Name:  "dj",
          Email: "darjun@126.com",
       },
    },
  }
  app.Run(os.Args)
}
```

`{{.XXX}}` 其中 `XXX` 对应 `cli.App{}` 结构中设置的字段，例如上面 `Authors` ：

```
$ ./help --help
NAME:
  help - A new cli application
USAGE:
  help [global options] command [command options] [arguments...]

AUTHOR:
  dj <darjun@126.com>

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

GLOBAL OPTIONS:
  --help, -h  show help (default: false)
```

注意观察 `AUTHOR` 部分。

通过覆盖 `HelpPrinter` ，我们能自己输出帮助信息：

```
func main() {
  cli.HelpPrinter = func(w io.Writer, templ string, data interface{}) {
    fmt.Println("Simple help!")
  }

  (&cli.App{}).Run(os.Args)
}
```

编译、运行：

```
$ ./help --help
Simple help!
```

内置选项

帮助选项

默认情况下，帮助选项为 `--help/-h` 。我们可以通过 `cli.HelpFlag` 字段设置：

```
func main() {
  cli.HelpFlag = &cli.BoolFlag{
    Name:     "haaaaalp",
    Aliases: []string{"halp"},
    Usage:    "HALP",
  }

  (&cli.App{}).Run(os.Args)
}
```

查看帮助：

```
$ go run main.go --halp
NAME:
   main.exe - A new cli application

USAGE:
   main.exe [global options] command [command options] [arguments...]

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

```
GLOBAL OPTIONS:
   --haaaaalp, --halp  HALP (default: false)
```

版本选项

默认版本选项 `-v/--version` 输出应用的版本信息。我们可以通过 `cli.VersionFlag` 设置版本选项：

```
func main() {
  cli.VersionFlag = &cli.BoolFlag{
    Name:    "print-version",
    Aliases: []string{"V"},
    Usage:   "print only the version",
  }

  app := &cli.App{
    Name:    "version",
    Version: "v1.0.0",
  }
  app.Run(os.Args)
}
```

这样就可以通过指定 `--print-version/-V` 输出版本信息了。运行：

```
$ go run main.go --print-version
version version v1.0.0

$ go run main.go -V
version version v1.0.0
```

我们还可以通过设置 `cli.VersionPrinter` 字段控制版本信息的输出内容：

```
const (
  Revision = "0cebd6e32a4e7094bbdbf150a1c2ffa56c34e91b"
)

func main() {
  cli.VersionPrinter = func(c *cli.Context) {
    fmt.Printf("version=%s revision=%s\n", c.App.Version, Revision)
  }

  app := &cli.App{
    Name:    "version",
```

```
        Version: "v1.0.0",
    }
    app.Run(os.Args)
}
```

上面程序同时输出版本号和 `git` 提交的 SHA 值：

```
$ go run main.go -v
version=v1.0.0 revision=0cebd6e32a4e7094bbdbf150a1c2ffa56c34e91b
```

总结

`cli` 非常灵活，只需要设置 `cli.App` 的字段值即可实现相应的功能，不需要额外记忆函数、方法。另外 `cli` 还支持 Bash 自动补全的功能，对 zsh 的支持也比较好，感兴趣可自行探索。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. cli GitHub：https://github.com/urfave/cli
2. Go 每日一库之 cobra：https://darjun.github.io/2020/01/17/godailylib/cobra/
3. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

negroni

简介

`negroni` 是一个专注于 HTTP 中间件的库。它小巧，无侵入，鼓励使用标准库 `net/http` 的处理器（ `Handler` ）。本文就来介绍一下这个库。

为什么要使用中间件？有一些逻辑代码，如统计、日志、调试等，每一个处理器中都需要，如果一个个去添加太繁琐了、容易出错、容易遗漏。如果我们要统计处理器耗时，可以在每个处理器中添加代码统计耗时：

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

func index(w http.ResponseWriter, r *http.Request) {
  start := time.Now()
  fmt.Fprintf(w, "home page")
  fmt.Printf("index elasped:%fs", time.Since(start).Seconds())
}

func greeting(w http.ResponseWriter, r *http.Request) {
  start := time.Now()
  name := r.FormValue("name")
  if name == "" {
    name = "world"
  }

  fmt.Fprintf(w, "hello %s", name)
  fmt.Printf("greeting elasped:%fs\n", time.Since(start).Seconds())
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", index)
  mux.HandleFunc("/greeting", greeting)

  http.ListenAndServe(":8000", mux)
}
```

但是这个做法非常不灵活：

- 每增加一个处理器，都需要添加这部分代码。而这些代码与实际的处理器逻辑并没有什么关系。编写处理器时比较容易遗忘，特别是要考虑所有的返回路径。增加了编码负担；
- 不利于修改：如果统计代码有错误或者需要调整，必须要改动所有的处理器；
- 添加麻烦：要添加其他的统计逻辑也需要改动所有的处理器代码。

利用 Go 语言的闭包，我们可以将实际的处理器代码封装到一个函数中，在这个函数中执行额外的逻辑：

```
func elasped(h func(w http.ResponseWriter, r *http.Request)) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Path
    start := time.Now()
    h(w, r)
    fmt.Printf("path:%s elasped:%fs\n", path, time.Since(start).Seconds())
  }
}

func index(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "home page")
}

func greeting(w http.ResponseWriter, r *http.Request) {
  name := r.FormValue("name")
  if name == "" {
    name = "world"
  }

  fmt.Fprintf(w, "hello %s", name)
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", elasped(index))
  mux.HandleFunc("/greeting", elasped(greeting))

  http.ListenAndServe(":8000", mux)
}
```

我们将额外的与处理器无关的代码放在另外的函数中。注册处理器函数时，我们不直接使用原始的处理器函数，而是用 `elasped` 函数封装一层。实际上 `elasped` 这样的函数就是中

间件。它封装原始的处理器函数，返回一个新的处理器函数。从而能很方便在实际的处理逻辑前后插入代码，便于添加、修改和维护。

快速使用

先安装：

```
$ go get github.com/urfave/negroni
```

后使用：

```
package main

import (
  "fmt"
  "net/http"

  "github.com/urfave/negroni"
)

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
  })

  n := negroni.Classic()
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

`negroni` 的使用非常简单，它可以很方便的与 `http.Handler` 一起使用。`negroni.Classic()` 提供了几个常用的中间件：

- `negroni.Recovery` ：恢复 `panic` ，处理器代码中有 `panic` 会被这个中间件捕获，程序不会退出；
- `negroni.Logger` ：日志，记录请求和响应的基本信息；
- `negroni.Static` ：在 `public` 目录提供静态文件服务。

调用 `n.UseHandler(mux)` ，将这些中间件应用到多路复用器上。运行，在浏览器中输入 `localhost:3000` ，查看控制台输出：

```
$ go run main.go
[negroni] 2020-06-22T06:48:53+08:00 | 200 |      20.9966ms | localhost:3000 | GE
T /
[negroni] 2020-06-22T06:48:54+08:00 | 200 |      0s | localhost:3000 | GET /favi
con.ico
```

`negroni.Handler`

接口 `negroni.Handler` 让我们对中间件的执行流程有更灵活的控制：

```
type Handler interface {
  ServeHTTP(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)
}
```

我们编写的中间件签名必须是 `func(http.ResponseWriter,*http.Request,http.HandlerFunc)` ，或者实现 `negroni.Handler` 接口：

```
func RandomMiddleware(w http.ResponseWriter, r *http.Request, next http.HandlerF
unc) {
  if rand.Int31n(100) <= 50 {
    fmt.Fprintf(w, "hello from RandomMiddleware")
  } else {
    next(w, r)
  }
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
  })

  n := negroni.New()
  n.Use(negroni.HandlerFunc(RandomMiddleware))
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

上面代码中实现了一个随机的中间件，有一半的概率直接从 `RandomMiddleware` 这个中间件返回，一半的概率执行实际的处理器函数。运行程序，在浏览器中不停地刷新页面 `localhost:3000` 看看效果。

注意，实际上 `func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)` 只是一个方便的写法。在调用 `n.Use` 时使用了 `negroni.HandlerFunc` 做了一层封装，而 `negroni.HandlerFunc` 实现了 `negroni.Handler` 接口：

```
// src/github.com/urfave/negroni/negroni.go
type HandlerFunc func(rw http.ResponseWriter, r *http.Request, next http.HandlerFunc)

func (h HandlerFunc) ServeHTTP(rw http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
  h(rw, r, next)
}
```

`net/http` 中也有类似的代码，通过 `http.HandlerFunc` 封装 `func(http.ResponseWriter,*http.Request)` 从而实现接口 `http.Handler` 。

`negroni.With`

如果有多个中间件，每个都需要 `n.Use()` 有些繁琐。 `negroni` 提供了一个 `With()` 方法，它接受一个或多个 `negroni.Handler` 参数，返回一个新的对象：

```
func Middleware1(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)
 {
  fmt.Println("Middleware1 begin")
  next(w, r)
  fmt.Println("Middleware1 end")
}

func Middleware2(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)
 {
  fmt.Println("Middleware2 begin")
  next(w, r)
  fmt.Println("Middleware2 end")
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
  })

  n := negroni.New()
  n = n.With(
```

```
    negroni.HandlerFunc(Middleware1),
    negroni.HandlerFunc(Middleware2),
  )
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

Run

`Negroni` 对象提供了一个方便的 `Run()` 方法来运行服务器程序。它接受与 `http.ListenAndServe()` 一样的地址（ `Addr` ）参数：

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
  })

  n := negroni.New()
  n.UseHandler(mux)
  n.Run(":3000")
}
```

如果未指定端口，那么尝试使用 `PORT` 环境变量。如果 `PORT` 环境变量也未设置，那么使用默认的端口 `:8080` 。

作为 `http.Handler` 使用

`negroni` 很容易在 `net/http` 程序中使用， `negroni.Negroni` 对象可直接作为 `http.Handler` 传给相应的方法：

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
  })

  n := negroni.Classic()
  n.UseHandler(mux)

  s := &http.Server{
    Addr:           ":8080",
```

```
    Handler:        n,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
  }
  s.ListenAndServe()
}
```

内置中间件

`negroni` 内置了一些常用的中间件，可直接使用。

`Static`

`negroni.Static` 可在指定目录中提供文件服务：

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "hello world")
  })

  n := negroni.New()
  n.Use(negroni.NewStatic(http.Dir("./public")))
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

在程序运行目录下创建 `public` 目录，然后放入一些文件 `1.txt` ， `2.jpg` 。程序运行之后，就能通过浏览器 `localhost:3000/1.txt` 和 `localhost:3000/2.jpg` 请求这些文件了。

另外需要特别注意一点，如果找不到对应的文件， `Static` 会将请求传给下一个中间件或处理器函数。在上面的例子中就是 `hello world` 。在浏览器中输入 `localhost:3000/none-exist.txt` 看看效果。

`Logger`

在快速开始中，我们通过 `negroni.Classic()` 已经使用过这个中间件了。我们也可以单独使用，它可以记录请求的信息。我们还可以调用 `SetFormat()` 方法设置日志的格式：

```
func main() {
  mux := http.NewServeMux()
```

```
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "hello world")
    })

    n := negroni.New()
    logger := negroni.NewLogger()
    logger.SetFormat("[{{.Status}} {{.Duration}}] - {{.Request.UserAgent}}")
    n.Use(logger)
    n.UseHandler(mux)

    http.ListenAndServe(":3000", n)
}
```

上面代码中将日志格式设置为 `[{{.Status}} {{.Duration}}] - {{.Request.UserAgent}}` ，即响应状态、耗时和 `UserAgent` 。

使用 Chrome 浏览器请求：

```
[negroni] [200 26.0029ms] - Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.106 Safari/537.36
```

`Recovery`

`negroni.Recovery` 可以捕获后续的中间件或处理器函数中出现的 `panic` ，返回一个 `500` 的响应码：

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        panic("internal server error")
    })

    n := negroni.New()
    n.Use(negroni.NewRecovery())
    n.UseHandler(mux)

    http.ListenAndServe(":3000", n)
}
```

请求时 `panic` 的堆栈会显示在浏览器中：

```
PANIC: internal server error
goroutine 7 [running]:
github.com/urfave/negroni.(*Recovery).ServeHTTP.func1(0x26a85018, 0xc000006048, 0xc0001a0000, 0xc00004c190)
        D:/code/golang/src/github.com/urfave/negroni/recovery.go:168 +0x162
panic(0x729820, 0x8164b0)
        C:/Go/src/runtime/panic.go:969 +0x174
main.main.func1(0x26a85018, 0xc000006048, 0xc0001a0000)
        D:/code/golang/src/github.com/darjun/go-daily-lib/negroni/middlewares/recovery/main.go:12 +0x40
net/http.HandlerFunc.ServeHTTP(0x7bc800, 0x26a85018, 0xc000006048, 0xc0001a0000)
        C:/Go/src/net/http/server.go:2012 +0x4b
net/http.(*ServeMux).ServeHTTP(0xc00001e380, 0x26a85018, 0xc000006048, 0xc0001a0000)
        C:/Go/src/net/http/server.go:2387 +0x1ac
github.com/urfave/negroni.Wrap.func1(0x26a85018, 0xc000006048, 0xc0001a0000, 0xc000004740)
        D:/code/golang/src/github.com/urfave/negroni/negroni.go:55 +0x54
github.com/urfave/negroni.HandlerFunc.ServeHTTP(0xc0000046e0, 0x26a85018, 0xc000006048, 0xc0001a0000, 0xc000004740)
        D:/code/golang/src/github.com/urfave/negroni/negroni.go:29 +0x55
github.com/urfave/negroni.middleware.ServeHTTP(...)
        D:/code/golang/src/github.com/urfave/negroni/negroni.go:47
github.com/urfave/negroni.(*Recovery).ServeHTTP(0xc00004c190, 0x26a85018, 0xc000006048, 0xc0001a0000, 0xc000004760)
        D:/code/golang/src/github.com/urfave/negroni/recovery.go:210 +0x93
github.com/urfave/negroni.middleware.ServeHTTP(...)
        D:/code/golang/src/github.com/urfave/negroni/negroni.go:47
github.com/urfave/negroni.(*Negroni).ServeHTTP(0xc000070de0, 0x823840, 0xc0001380e0, 0xc0001a0000)
        D:/code/golang/src/github.com/urfave/negroni/negroni.go:107 +0x102
net/http.serverHandler.ServeHTTP(0xc000138000, 0x823840, 0xc0001380e0, 0xc0001a0000)
        C:/Go/src/net/http/server.go:2807 +0xaa
net/http.(*conn).serve(0xc000050aa0, 0x823e00, 0xc0001880c0)
        C:/Go/src/net/http/server.go:1895 +0x873
created by net/http.(*Server).Serve
        C:/Go/src/net/http/server.go:2933 +0x363
```

这在开发环境比较有用，但是生成环境中不能泄露这个信息。这时可以设置 `PrintStack` 字段为 `false`：

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    panic("internal server error")
  })

  n := negroni.New()
  r := negroni.NewRecovery()
  r.PrintStack = false
  n.Use(r)
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

除了在控制台和浏览器中输出 `panic` 信息，`Recovery` 还提供了钩子函数，可以向其他服务上报 `panic`，如 `Sentry/Airbrake`。当然上报的代码要自己写😄。

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    panic("internal server error")
  })

  n := negroni.New()
```

```go
  r := negroni.NewRecovery()
  r.PanicHandlerFunc = reportToSentry
  n.Use(r)
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}

func reportToSentry(info *negroni.PanicInformation) {
  fmt.Println("sent to sentry")
}
```

设置 `PanicHandlerFunc` 之后，发生 `panic` 就会调用此函数。

我们还可以对输出的格式进行设置，设置 `Formatter` 字段为 `negroni.HTMLPanicFormatter` 能让输出更好地在浏览器中呈现：

```go
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    panic("internal server error")
  })

  n := negroni.New()
  r := negroni.NewRecovery()
  r.Formatter = &negroni.HTMLPanicFormatter{}
  n.Use(r)
  n.UseHandler(mux)

  http.ListenAndServe(":3000", n)
}
```

效果：

Negroni - PANIC

GET /

Runtime error: internal server error

Runtime Stack

```
goroutine 18 [running]:
github.com/urfave/negroni.(*Recovery).ServeHTTP.func1(0x26a84058, 0xc000006050, 0xc000132100, 0xc000192050)
	D:/code/golang/src/github.com/urfave/negroni/recovery.go:168 +0x162
panic(0x729820, 0x8164b0)
	C:/Go/src/runtime/panic.go:969 +0x174
main.main.func1(0x26a84058, 0xc000006050, 0xc000132100)
	D:/code/golang/src/github.com/darjun/go-daily-lib/negroni/middlewares/recovery/html-panic-formatter/main.go:13 +0x40
net/http.HandlerFunc.ServeHTTP(0x7bc800, 0x26a84058, 0xc000006050, 0xc000132100)
	C:/Go/src/net/http/server.go:2012 +0x4b
net/http.(*ServeMux).ServeHTTP(0xc000184080, 0x26a84058, 0xc000006050, 0xc000132100)
	C:/Go/src/net/http/server.go:2387 +0x1ac
github.com/urfave/negroni.Wrap.func1(0x26a84058, 0xc000006050, 0xc000132100, 0xc0001900e0)
	D:/code/golang/src/github.com/urfave/negroni/negroni.go:55 +0x54
github.com/urfave/negroni.HandlerFunc.ServeHTTP(0xc000190080, 0x26a84058, 0xc000006050, 0xc000132100, 0xc0001900e0)
	D:/code/golang/src/github.com/urfave/negroni/negroni.go:29 +0x55
github.com/urfave/negroni.middleware.ServeHTTP(...)
	D:/code/golang/src/github.com/urfave/negroni/negroni.go:47
github.com/urfave/negroni.(*Recovery).ServeHTTP(0xc000192050, 0x26a84058, 0xc000006050, 0xc000132100, 0xc000190100)
	D:/code/golang/src/github.com/urfave/negroni/recovery.go:210 +0x93
github.com/urfave/negroni.middleware.ServeHTTP(...)
	D:/code/golang/src/github.com/urfave/negroni/negroni.go:47
github.com/urfave/negroni.(*Negroni).ServeHTTP(0xc0001820f0, 0x823820, 0xc000142000, 0xc000132100)
	D:/code/golang/src/github.com/urfave/negroni/negroni.go:107 +0x102
net/http.serverHandler.ServeHTTP(0xc000196000, 0x823820, 0xc000142000, 0xc000132100)
	C:/Go/src/net/http/server.go:2807 +0xaa
net/http.(*conn).serve(0xc0001a6000, 0x823de0, 0xc00001e3c0)
	C:/Go/src/net/http/server.go:1895 +0x873
created by net/http.(*Server).Serve
	C:/Go/src/net/http/server.go:2933 +0x363
```

第三方中间件

除了内置中间件外，`negroni` 还有很多第三方的中间件。完整列表看这里：https://github.com/urfave/negroni#third-party-middleware。

我们只介绍一个 `xrequestid` ，它在每个请求中增加一个随机的 `Header` ： `X-Request-Id` 。

安装 `xrequestid` ：

```
$ go get github.com/pilu/xrequestid
```

使用：

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "X-Request-Id is `%s`", r.Header.Get("X-Request-Id"))
  })

  n := negroni.New()
  n.Use(xrequestid.New(16))
```

```
    n.UseHandler(mux)
    n.Run(":3000")
}
```

给每个请求增加一个 16 字节的 `X-Request-Id` ，处理器函数中将这个 `X-Request-Id` 写入响应中，最后呈现在浏览器中。运行程序，在浏览器中输入 `localhost:3000` 查看效果。

总结

`negroni` 专注于中间件，没有很多花哨的功能。无侵入性使得它很容易与标准库 `net/http` 和其他的 Web 库（如 `gorilla/mux` ）一起使用。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. negroni GitHub：https://github.com/urfave/negroni
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

fyne

简介

Go 语言生态中，GUI 一直是短板，更别说跨平台的 GUI 了。`fyne` 向前迈了一大步。`fyne` 是 Go 语言编写的**跨平台的** UI 库，它可以很方便地移植到手机设备上。`fyne` 使用上非常简单，同时它还提供 `fyne` 命令打包静态资源和应用程序。我们先简单介绍基本控件和布局，然后介绍如何发布一个 `fyne` 应用程序。

快速使用

本文代码使用 Go Modules。

先初始化：

```
$ mkdir fyne && cd fyne
$ go mod init github.com/darjun/go-daily-lib/fyne
```

由于 `fyne` 包含一些 C/C++ 的代码，所以需要 `gcc` 编译工具。在 Linux/Mac OSX 上，`gcc` 基本是标配，在 windows 上我们有 3 种方式安装 `gcc` 工具链：

- MSYS2 + MingW-w64：https://www.msys2.org/；
- TDM-GCC：https://jmeubank.github.io/tdm-gcc/download/；
- Cygwin：https://www.cygwin.com/。

本文选择 `TDM-GCC` 的方式安装。到https://jmeubank.github.io/tdm-gcc/download/下载安装程序并安装。正常情况下安装程序会自动设置 `PATH` 路径。打开命令行，键入 `gcc -v`。如果正常输出版本信息，说明安装成功且环境变量设置正确。

安装 `fyne`：

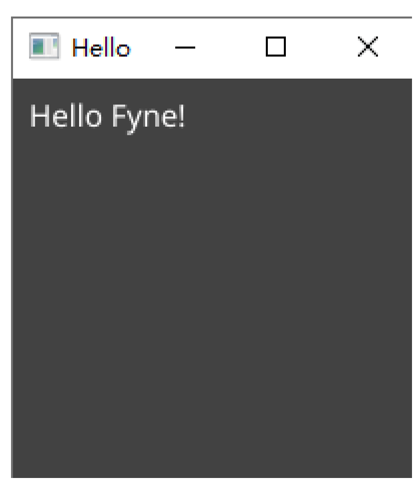
```
$ go get -u fyne.io/fyne
```

到此准备工作已经完成，我们开始编码。按照惯例，先以**Hello, World**程序开始：

```
package main

import (
  "fyne.io/fyne"
```

```go
    "fyne.io/fyne/app"
    "fyne.io/fyne/widget"
)

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

    myWin := myApp.NewWindow("Hello")
    myWin.SetContent(widget.NewLabel("Hello Fyne!"))
    myWin.Resize(fyne.NewSize(200, 200))
    myWin.ShowAndRun()
}
```

运行结果如下：



`fyne` 的使用很简单。每个 `fyne` 程序都包括两个部分，一个是应用程序对象 `myApp` ，通过 `app.New()` 创建。另一个是窗口对象，通过应用程序对象 `myApp` 来创建 `myApp.NewWindow("Hello")` 。 `myApp.NewWindow()` 方法中传入的字符串就是窗口标题。

`fyne` 提供了很多常用的组件，通过 `widget.NewXXX()` 创建（ `XXX` 为组件名）。上面示例中，我们创建了一个 `Label` 控件，然后设置到窗口中。最后，调用 `myWin.ShowAndRun()` 开始运行程序。实际上 `myWin.ShowAndRun()` 等价于

```go
myWin.Show()
myApp.Run()
```

`myWin.Show()` 显示窗口， `myApp.Run()` 开启事件循环。

注意一点， `fyne` 默认窗口大小是根据内容的宽高来设置的。上面我们调用 `myWin.Resize()` 手动设置了大小。否则窗口只能放下字符串 `Hello Fyne!` 。

`fyne` 包结构划分

`fyne` 将功能划分到多个子包中：

- `fyne.io/fyne` ：提供所有 `fyne` 应用程序代码共用的基础定义，包括数据类型和接口；
- `fyne.io/fyne/app` ：提供创建应用程序的 API；
- `fyne.io/fyne/canvas` ：提供 `Fyne` 使用的绘制 API；
- `fyne.io/fyne/dialog` ：提供对话框组件；
- `fyne.io/fyne/layout` ：提供多种界面布局；
- `fyne.io/fyne/widget` ：提供多种组件， `fyne` 所有的窗体控件和交互元素都在这个子包中。

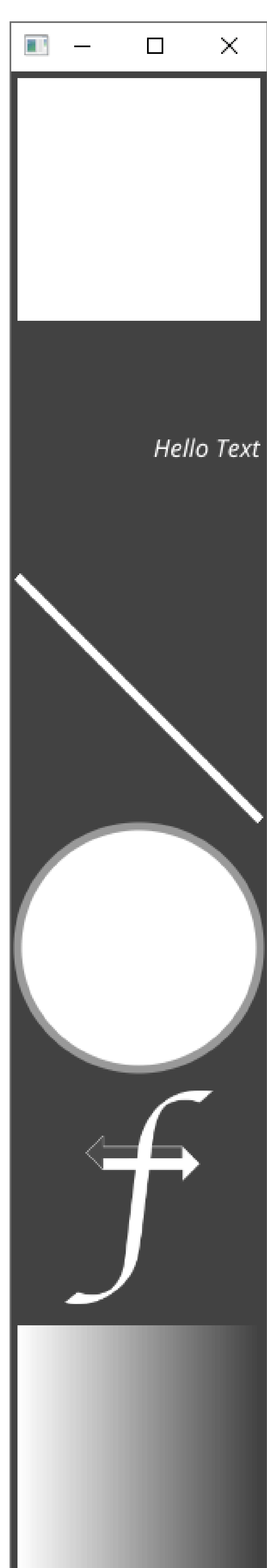
Canvas

在 `fyne` 应用程序中，所有显示元素都是绘制在画布（ `Canvas` ）上的。这些元素都是画布对象（ `CanvasObject` ）。调用 `Canvas.SetContent()` 方法可设置画布内容。 `Canvas` 一般和布局（ `Layout` ）容器（ `Container` ）一起使用。 `canvas` 子包中提供了一些基础的画布对象：

```
package main

import (
  "image/color"
  "math/rand"

  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/canvas"
  "fyne.io/fyne/layout"
  "fyne.io/fyne/theme"
)

func main() {
  a := app.New()
  w := a.NewWindow("Canvas")

  rect := canvas.NewRectangle(color.White)

  text := canvas.NewText("Hello Text", color.White)
  text.Alignment = fyne.TextAlignTrailing
```

```
    text.TextStyle = fyne.TextStyle{Italic: true}

    line := canvas.NewLine(color.White)
    line.StrokeWidth = 5

    circle := canvas.NewCircle(color.White)
    circle.StrokeColor = color.Gray{0x99}
    circle.StrokeWidth = 5

    image := canvas.NewImageFromResource(theme.FyneLogo())
    image.FillMode = canvas.ImageFillOriginal

    raster := canvas.NewRasterWithPixels(
        func(_, _, w, h int) color.Color {
            return color.RGBA{uint8(rand.Intn(255)),
                uint8(rand.Intn(255)),
                uint8(rand.Intn(255)), 0xff}
        },
    )

    gradient := canvas.NewHorizontalGradient(color.White, color.Transparent)

    container := fyne.NewContainerWithLayout(
        layout.NewGridWrapLayout(fyne.NewSize(150, 150)),
        rect, text, line, circle, image, raster, gradient))
    w.SetContent(container)
    w.ShowAndRun()
}
```

程序运行结果如下：


Hello Text

`canvas.Rectangle` 是最简单的画布对象了，通过 `canvas.NewRectangle()` 创建，传入填充颜色。

`canvas.Text` 是显示文本的画布对象，通过 `canvas.NewText()` 创建，传入文本字符串和颜色。该对象可设置对齐方式和字体样式。对齐方式通过设置 `Text` 对象的 `Alignment` 字段值，取值有：

- `TextAlignLeading`：左对齐；
- `TextAlignCenter`：中间对齐；
- `TextAlignTrailing`：右对齐。

字体样式通过设置 `Text` 对象的 `TextStyle` 字段值，`TextStyle` 是一个结构体：

```
type TextStyle struct {
  Bold      bool
  Italic    bool
  Monospace bool
}
```

对应字段设置为 `true` 将显示对应的样式：

- `Bold`：粗体；
- `Italic`：斜体；
- `Monospace`：系统等宽字体。

我们还可以通过设置环境变量 `FYNE_FONT` 为一个 `.ttf` 文件从而使用外部字体。

`canvas.Line` 是线段，通过 `canvas.NewLine()` 创建，传入颜色。可以通过 `line.StrokeWidth` 设置线段宽度。默认情况下，线段是从父控件或画布的**左上角**到**右下角**的。可通过 `line.Move()` 和 `line.Resize()` 修改位置。

`canvas.Circle` 是圆形，通过 `canvas.NewCircle()` 创建，传入颜色。另外通过 `StrokeColor` 和 `StrokeWidth` 设置圆形边框的颜色和宽度。

`canvas.Image` 是图像，可以通过已加载的程序资源创建（`canvas.NewImageFromResource()`），传入资源对象。或通过文件路径创建（`canvas.NewImageFromFile()`），传入文件路径。或通过已构造的 `image.Image` 对象创建（`canvas.NewImageFromImage()`）。可以通过 `FillMode` 设置图像的填充模式：

- `ImageFillStretch`：拉伸，填满空间；
- `ImageFillContain`：保持宽高比；
- `ImageFillOriginal`：保持原始大小，不缩放。

下面程序演示了这 3 种创建图像的方式：

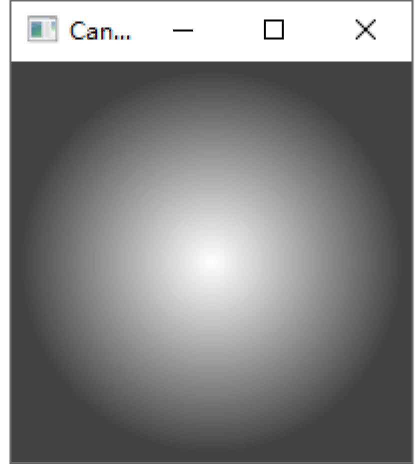
```
package main

import (
  "image"
  "image/color"

  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/canvas"
  "fyne.io/fyne/layout"
  "fyne.io/fyne/theme"
)

func main() {
  a := app.New()
  w := a.NewWindow("Hello")

  img1 := canvas.NewImageFromResource(theme.FyneLogo())
  img1.FillMode = canvas.ImageFillOriginal

  img2 := canvas.NewImageFromFile("./luffy.jpg")
  img2.FillMode = canvas.ImageFillOriginal

  image := image.NewAlpha(image.Rectangle{image.Point{0, 0}, image.Point{100, 10
0}})
  for i := 0; i < 100; i++ {
    for j := 0; j < 100; j++ {
      image.Set(i, j, color.Alpha{uint8(i % 256)})
    }
  }
  img3 := canvas.NewImageFromImage(image)
  img3.FillMode = canvas.ImageFillOriginal

  container := fyne.NewContainerWithLayout(
    layout.NewGridWrapLayout(fyne.NewSize(150, 150)),
    img1, img2, img3)
  w.SetContent(container)
  w.ShowAndRun()
}
```

`theme.FyneLogo()` 是 Fyne 图标资源，`luffy.jpg` 是磁盘中的文件，最后创建一个 `image.Image` ，从中生成 `canvas.Image` 。

最后一种是梯度渐变效果，有两种类型 `canvas.LinearGradient` （线性渐变）和 `canvas.RadialGradient` （放射渐变），指从一种颜色渐变到另一种颜色。线性渐变又分为两种**水平线性渐变**和**垂直线性渐变**，分别通过 `canvas.NewHorizontalGradient()` 和 `canvas.NewVerticalGradient()` 创建。放射渐变通过 `canvas.NewRadialGradient()` 创建。我们在上面的示例中已经看到了水平线性渐变的效果，接下来一起看看放射渐变的效果：

```
func main() {
  a := app.New()
  w := a.NewWindow("Canvas")

  gradient := canvas.NewRadialGradient(color.White, color.Transparent)
  w.SetContent(gradient)
  w.Resize(fyne.NewSize(200, 200))
  w.ShowAndRun()
}
```

运行效果如下：



放射效果就是从中心向周围渐变。

Widget

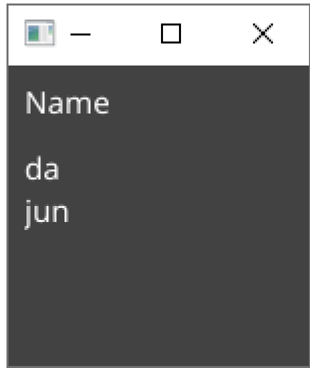
窗体控件是一个 `Fyne` 应用程序的主要组成部分。它们能适配当前的主题，并且处理与用户的交互。

Label

标签（ `Label` ）是最简单的一个控件了，用于显示字符串。它有点类似于 `canvas.Text` ，不同之处在于 `Label` 可以处理简单的格式化，例如 `\n` ：

```
func main() {
  myApp := app.New()
  myWin := myApp.NewWindow("Label")

  l1 := widget.NewLabel("Name")
  l2 := widget.NewLabel("da\njun")

  container := fyne.NewContainerWithLayout(layout.NewVBoxLayout(), l1, l2)
  myWin.SetContent(container)
  myWin.Resize(fyne.NewSize(150, 150))
  myWin.ShowAndRun()
}
```

第二个 `widget.Label` 中 `\n` 后面的内容会在下一行渲染：

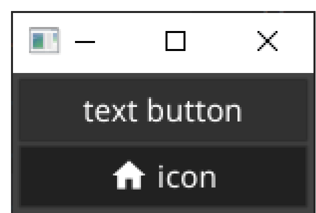


Button

按钮（ `Button` ）控件让用户点击，给用户反馈。 `Button` 可以包含文本，图标或两者皆有。调用 `widget.NewButton()` 创建一个默认的文本按钮，传入文本和一个无参的回调函数。带图标的按钮需要调用 `widget.NewButtonWithIcon()` ，传入文本和回调参数，还需要一个 `fyne.Resource` 类型的图标资源：

```
func main() {
  myApp := app.New()
  myWin := myApp.NewWindow("Button")

  btn1 := widget.NewButton("text button", func() {
    fmt.Println("text button clicked")
  })

  btn2 := widget.NewButtonWithIcon("icon", theme.HomeIcon(), func() {
    fmt.Println("icon button clicked")
  })

  container := fyne.NewContainerWithLayout(layout.NewVBoxLayout(), btn1, btn2)
  myWin.SetContent(container)
  myWin.Resize(fyne.NewSize(150, 50))
  myWin.ShowAndRun()
}
```

上面创建了一个文本按钮和一个图标按钮， `theme` 子包中包含一些默认的图标资源，也可以加载外部的图标。运行：



点击按钮，对应的回调就会被调用，试试看！

Box

盒子控件（`Box`）就是一个简单的水平或垂直的容器。在内部，`Box`对子控件采用盒状布局（`Box Layout`），详见后文布局。我们可以通过传入控件对象给`widget.NewHBox()`或`widget.NewVBox()`创建盒子。或者调用已经创建好的`widget.Box`对象的`Append()`或`Prepend()`向盒子中添加控件。前者在尾部追加，后者在头部添加。

```
func main() {
  myApp := app.New()
  myWin := myApp.NewWindow("Box")

  content := widget.NewVBox(
    widget.NewLabel("The top row of VBox"),
    widget.NewHBox(
      widget.NewLabel("Label 1"),
      widget.NewLabel("Label 2"),
    ),
  )
  content.Append(widget.NewButton("Append", func() {
    content.Append(widget.NewLabel("Appended"))
  }))
  content.Append(widget.NewButton("Prepend", func() {
    content.Prepend(widget.NewLabel("Prepended"))
  }))

  myWin.SetContent(content)
  myWin.Resize(fyne.NewSize(150, 150))
  myWin.ShowAndRun()
}
```

我们甚至可以嵌套`widget.Box`控件，这样就可以实现比较灵活的布局。上面的代码中添加了两个按钮，点击时分别在尾部和头部添加一个`Label`：

Entry

输入框（`Entry`）控件用于给用户输入简单的文本内容。调用`widget.NewEntry()`即可创建一个输入框控件。我们一般保存输入框控件的引用，以便访问其`Text`字段来获取内容。注册`OnChanged`回调函数。每当内容有修改时，`OnChanged`就会被调用。我们可以调用`SetReadOnly(true)`设置输入框的只读属性。方法`SetPlaceHolder()`用来设置占位字符串，设置字段`Multiline`让输入框接受多行文本。另外，我们可以使用`NewPasswordEntry()`创建一个密码输入框，输入的文本不会以明文显示。

```
func main() {
  myApp := app.New()
  myWin := myApp.NewWindow("Entry")

  nameEntry := widget.NewEntry()
  nameEntry.SetPlaceHolder("input name")
  nameEntry.OnChanged = func(content string) {
    fmt.Println("name:", nameEntry.Text, "entered")
  }

  passEntry := widget.NewPasswordEntry()
  passEntry.SetPlaceHolder("input password")

  nameBox := widget.NewHBox(widget.NewLabel("Name"), layout.NewSpacer(), nameEnt
ry)
  passwordBox := widget.NewHBox(widget.NewLabel("Password"), layout.NewSpacer(),
passEntry)

  loginBtn := widget.NewButton("Login", func() {
    fmt.Println("name:", nameEntry.Text, "password:", passEntry.Text, "login in"
)
  })

  multiEntry := widget.NewEntry()
  multiEntry.SetPlaceHolder("please enter\nyour description")
  multiEntry.MultiLine = true

  content := widget.NewVBox(nameBox, passwordBox, loginBtn, multiEntry)
  myWin.SetContent(content)
  myWin.ShowAndRun()
}
```

这里我们实现了一个简单的登录界面：

`Checkbox/Radio/Select`

`CheckBox` 是简单的选择框，每个选择是独立的，例如爱好可以是足球、篮球，也可以都是。创建方法 `widget.NewCheck()` ，传入选项字符串（足球，篮球）和回调函数。回调函数接受一个 `bool` 类型的参数，表示该选项是否选中。

`Radio` 是单选框，每个组内只能选择一个，例如性别，只能是男或女（？）。创建方法 `widget.NewRadio()` ，传入字符串切片和回调函数作为参数。回调函数接受一个字符串参数，表示选中的选项。也可以使用 `Selected` 字段读取选中的选项。

`Select` 是下拉选择框，点击时显示一个下拉菜单，点击选择。选项非常多的时候，比较适合用 `Select` 。创建方法 `widget.NewSelect()` ，参数与 `NewRadio()` 完全相同。

```
func main() {
  myApp := app.New()
  myWin := myApp.NewWindow("Choices")

  nameEntry := widget.NewEntry()
  nameEntry.SetPlaceHolder("input name")

  passEntry := widget.NewPasswordEntry()
  passEntry.SetPlaceHolder("input password")

  repeatPassEntry := widget.NewPasswordEntry()
  repeatPassEntry.SetPlaceHolder("repeat password")

  nameBox := widget.NewHBox(widget.NewLabel("Name"), layout.NewSpacer(), nameEntry)
  passwordBox := widget.NewHBox(widget.NewLabel("Password"), layout.NewSpacer(), passEntry)
  repeatPasswordBox := widget.NewHBox(widget.NewLabel("Repeat Password"), layout.NewSpacer(), repeatPassEntry)

  sexRadio := widget.NewRadio([]string{"male", "female", "unknown"}, func(value string) {
    fmt.Println("sex:", value)
  })
  sexBox := widget.NewHBox(widget.NewLabel("Sex"), sexRadio)

  football := widget.NewCheck("football", func(value bool) {
    fmt.Println("football:", value)
  })
  basketball := widget.NewCheck("basketball", func(value bool) {
    fmt.Println("basketball:", value)
  })
  pingpong := widget.NewCheck("pingpong", func(value bool) {
    fmt.Println("pingpong:", value)
  })
  hobbyBox := widget.NewHBox(widget.NewLabel("Hobby"), football, basketball, pingpong)

  provinceSelect := widget.NewSelect([]string{"anhui", "zhejiang", "shanghai"}, func(value string) {
    fmt.Println("province:", value)
  })
  provinceBox := widget.NewHBox(widget.NewLabel("Province"), layout.NewSpacer(),
```

```
provinceSelect)

  registerBtn := widget.NewButton("Register", func() {
    fmt.Println("name:", nameEntry.Text, "password:", passEntry.Text, "register"
)
  })

  content := widget.NewVBox(nameBox, passwordBox, repeatPasswordBox,
    sexBox, hobbyBox, provinceBox, registerBtn)
  myWin.SetContent(content)
  myWin.ShowAndRun()
}
```

这里我们实现了一个简单的注册界面：

Form

表单控件（ `Form` ）用于对很多 `Label` 和输入控件进行布局。如果指定了 `OnSubmit` 或 `OnCancel` 函数，表单控件会自动添加对应的 `Button` 按钮。我们调用 `widget.NewForm()` 传入一个 `widget.FormItem` 切片创建 `Form` 控件。每一项中一个字符串作为 `Label` 的文本，一个控件对象。创建好 `Form` 对象之后还能调用其 `Append(label, widget)` 方法添加控件。

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Form")

  nameEntry := widget.NewEntry()
  passEntry := widget.NewPasswordEntry()

  form := widget.NewForm(
    &widget.FormItem{"Name", nameEntry},
    &widget.FormItem{"Pass", passEntry},
  )
  form.OnSubmit = func() {
    fmt.Println("name:", nameEntry.Text, "pass:", passEntry.Text, "login in")
  }
  form.OnCancel = func() {
    fmt.Println("login canceled")
  }

  myWindow.SetContent(form)
  myWindow.Resize(fyne.NewSize(150, 150))
```

```
    myWindow.ShowAndRun()
}
```

使用 `Form` 能大大简化表单的构建，我们使用 `Form` 重新编写了上面的登录界面：

注意 `Submit` 和 `Cancel` 按钮是自动生成的！

ProgressBar

进度条控件（ `ProgressBar` ）用来表示任务的进度，例如文件下载的进度。创建方法 `widget.NewProgressBar()` ，默认最小值为 `0.0` ，最大值为 `1.1` ，可通过 `Min/Max` 字段设置。调用 `SetValue()` 方法来控制进度。还有一种进度条是循环动画，它表示有任务在进行中，并不能表示具体的完成情况。

```
func main() {
    myApp := app.New()
    myWindow := myApp.NewWindow("ProgressBar")

    bar1 := widget.NewProgressBar()
    bar1.Min = 0
    bar1.Max = 100
    bar2 := widget.NewProgressBarInfinite()

    go func() {
        for i := 0; i <= 100; i ++ {
            time.Sleep(time.Millisecond * 500)
            bar1.SetValue(float64(i))
        }
    }()

    content := widget.NewVBox(bar1, bar2)
    myWindow.SetContent(content)
    myWindow.Resize(fyne.NewSize(150, 150))
    myWindow.ShowAndRun()
}
```

在另一个 goroutine 中更新进度。效果如下：

TabContainer

标签容器（ `TabContainer` ）允许用户在不同的内容面板之间切换。标签可以是文本或图标。创建方法 `widget.NewTabContainer()` ，传入 `widget.TabItem` 作为参

数。`widget.TabItem` 可通过 `widget.NewTabItem(label, widget)` 创建。标签还可以设置位置：

- `TabLocationBottom`：显示在底部；
- `TabLocationLeading`：显示在顶部左边；
- `TabLocationTrailing`：显示在顶部右边。

看示例：

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("TabContainer")

  nameLabel := widget.NewLabel("Name: dajun")
  sexLabel := widget.NewLabel("Sex: male")
  ageLabel := widget.NewLabel("Age: 18")
  addressLabel := widget.NewLabel("Province: shanghai")
  addressLabel.Hide()
  profile := widget.NewVBox(nameLabel, sexLabel, ageLabel, addressLabel)

  musicRadio := widget.NewRadio([]string{"on", "off"}, func(string) {})
  showAddressCheck := widget.NewCheck("show address?", func(value bool) {
    if !value {
      addressLabel.Hide()
    } else {
      addressLabel.Show()
    }
  })
  memberTypeSelect := widget.NewSelect([]string{"junior", "senior", "admin"}, fu
nc(string) {})

  setting := widget.NewForm(
    &widget.FormItem{"music", musicRadio},
    &widget.FormItem{"check", showAddressCheck},
    &widget.FormItem{"member type", memberTypeSelect},
  )

  tabs := widget.NewTabContainer(
    widget.NewTabItem("Profile", profile),
    widget.NewTabItem("Setting", setting),
  )

  myWindow.SetContent(tabs)
  myWindow.Resize(fyne.NewSize(200, 200))
```

```
  myWindow.ShowAndRun()
}
```

上面代码编写了一个简单的个人信息面板和设置面板，点击 `show address?` 可切换地址信息是否显示：

Toolbar

工具栏（ `Toolbar` ）是很多 GUI 应用程序必备的部分。工具栏将常用命令用图标的方式很形象地展示出来，方便使用。创建方法 `widget.NewToolbar()` ，传入多个 `widget.ToolbarItem` 作为参数。最常使用的 `ToolbarItem` 有命令（ `Action` ）、分隔符（ `Separator` ）和空白（ `Spacer` ），分别通过 `widget.NewToolbarItemAction(resource, callback)` / `widget.NewToolbarSeparator()` / `widget.NewToolbarSpacer()` 创建。命令需要指定回调，点击时触发。

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Toolbar")

  toolbar := widget.NewToolbar(
    widget.NewToolbarAction(theme.DocumentCreateIcon(), func() {
      fmt.Println("New document")
    }),
    widget.NewToolbarSeparator(),
    widget.NewToolbarAction(theme.ContentCutIcon(), func() {
      fmt.Println("Cut")
    }),
    widget.NewToolbarAction(theme.ContentCopyIcon(), func() {
      fmt.Println("Copy")
    }),
    widget.NewToolbarAction(theme.ContentPasteIcon(), func() {
      fmt.Println("Paste")
    }),
    widget.NewToolbarSpacer(),
    widget.NewToolbarAction(theme.HelpIcon(), func() {
      log.Println("Display help")
    }),
  )

  content := fyne.NewContainerWithLayout(
    layout.NewBorderLayout(toolbar, nil, nil, nil),
    toolbar, widget.NewLabel(`Lorem ipsum dolor,
    sit amet consectetur adipisicing elit.
```

```
    Quidem consectetur ipsam nesciunt,
    quasi sint expedita minus aut,
    porro iusto magnam ducimus voluptates cum vitae.
    Vero adipisci earum iure consequatur quidem.`),
  )
  myWindow.SetContent(content)
  myWindow.ShowAndRun()
}
```

工具栏一般使用 `BorderLayout` ，将工具栏放在其他任何控件上面，布局后文会详述。运行：

扩展控件

标准的 Fyne 控件提供了最小的功能集和定制化以适应大部分的应用场景。有些时候，我们需要更高级的功能。除了自己编写控件外，我们还可以扩展现有的控件。例如，我们希望图标控件 `widget.Icon` 能响应鼠标左键、右键和双击。首先编写一个**构造函数**，调用 `ExtendBaseWidget()` 方法获得基础的控件功能：

```go
type tappableIcon struct {
  widget.Icon
}

func newTappableIcon(res fyne.Resource) *tappableIcon {
  icon := &tappableIcon{}
  icon.ExtendBaseWidget(icon)
  icon.SetResource(res)

  return icon
}
```

然后实现相关的接口：

```go
// src/fyne.io/fyne/canvasobject.go
// 鼠标左键
type Tappable interface {
  Tapped(*PointEvent)
}

// 鼠标右键或长按
type SecondaryTappable interface {
  TappedSecondary(*PointEvent)
}
```

```
// 双击
type DoubleTappable interface {
  DoubleTapped(*PointEvent)
}
```

接口实现：

```
func (t *tappableIcon) Tapped(e *fyne.PointEvent) {
  log.Println("I have been left tapped at", e)
}

func (t *tappableIcon) TappedSecondary(e *fyne.PointEvent) {
  log.Println("I have been right tapped at", e)
}

func (t *tappableIcon) DoubleTapped(e *fyne.PointEvent) {
  log.Println("I have been double tapped at", e)
}
```

最后使用：

```
func main() {
  a := app.New()
  w := a.NewWindow("Tappable")
  w.SetContent(newTappableIcon(theme.FyneLogo()))
  w.Resize(fyne.NewSize(200, 200))
  w.ShowAndRun()
}
```

运行，点击图标控制台有相应输出：

```
2020/06/18 06:44:02 I have been left tapped at &{{110 97} {106 93}}
2020/06/18 06:44:03 I have been left tapped at &{{110 97} {106 93}}
2020/06/18 06:44:05 I have been right tapped at &{{88 102} {84 98}}
2020/06/18 06:44:06 I have been right tapped at &{{88 102} {84 98}}
2020/06/18 06:44:06 I have been left tapped at &{{88 101} {84 97}}
2020/06/18 06:44:07 I have been double tapped at &{{88 101} {84 97}}
```

输出的 `fyne.PointEvent` 中有绝对位置（对于窗口左上角）和相对位置（对于容器左上角）。

Layout

布局（ `Layout` ）就是控件如何在界面上显示，如何排列的。要想界面好看，布局是必须要掌握的。几乎所有的 GUI 框架都提供了布局或类似的接口。实际上，在前面的示例中我们已经在 `fyne.NewContainerWithLayout()` 函数中使用了布局。

BoxLayout

盒状布局（ `BoxLayout` ）是最常使用的一个布局。它将控件都排在一行或一列。在 `fyne` 中，我们可以通过 `layout.NewHBoxLayout()` 创建一个水平盒装布局，通过 `layout.NewVBoxLayout()` 创建一个垂直盒装布局。水平布局中的控件都排列在一行中，每个控件的宽度等于其内容的最小宽度（ `MinSize().Width` ），它们都拥有相同的高度，即所有控件的最大高度（ `MinSize().Height` ）。

垂直布局中的控件都排列在一列中，每个控件的高度等于其内容的最小高度，它们都拥有相同的宽度，即所有控件的最大宽度。

一般地，在 `BoxLayout` 中使用 `layout.NewSpacer()` 辅助布局，它会占满剩余的空间。对于水平盒状布局来说，第一个控件前添加一个 `layout.NewSpacer()` ，所有控件右对齐。最后一个控件后添加一个 `layout.NewSpacer()` ，所有控件左对齐。前后都有，那么控件中间对齐。如果在中间有添加一个 `layout.NewSpacer()` ，那么其它控件两边对齐。

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Box Layout")

  hcontainer1 := fyne.NewContainerWithLayout(layout.NewHBoxLayout(),
    canvas.NewText("left", color.White),
    canvas.NewText("right", color.White))

  // 左对齐
  hcontainer2 := fyne.NewContainerWithLayout(layout.NewHBoxLayout(),
    layout.NewSpacer(),
    canvas.NewText("left", color.White),
    canvas.NewText("right", color.White))

  // 右对齐
  hcontainer3 := fyne.NewContainerWithLayout(layout.NewHBoxLayout(),
    canvas.NewText("left", color.White),
    canvas.NewText("right", color.White),
    layout.NewSpacer())

  // 中间对齐
  hcontainer4 := fyne.NewContainerWithLayout(layout.NewHBoxLayout(),
    layout.NewSpacer(),
```

```
        canvas.NewText("left", color.White),
        canvas.NewText("right", color.White),
        layout.NewSpacer())

    // 两边对齐
    hcontainer5 := fyne.NewContainerWithLayout(layout.NewHBoxLayout(),
        canvas.NewText("left", color.White),
        layout.NewSpacer(),
        canvas.NewText("right", color.White))

    myWindow.SetContent(fyne.NewContainerWithLayout(layout.NewVBoxLayout(),
        hcontainer1, hcontainer2, hcontainer3, hcontainer4, hcontainer5))
    myWindow.Resize(fyne.NewSize(200, 200))
    myWindow.ShowAndRun()
}
```

运行效果：

GridLayout

格子布局（ `GridLayout` ）每一行有固定的列，添加的控件数量超过这个值时，后面的控件将会在新的行显示。创建方法 `layout.NewGridLayout(cols)` ，传入每行的列数。

```
func main() {
    myApp := app.New()
    myWindow := myApp.NewWindow("Grid Layout")

    img1 := canvas.NewImageFromResource(theme.FyneLogo())
    img2 := canvas.NewImageFromResource(theme.FyneLogo())
    img3 := canvas.NewImageFromResource(theme.FyneLogo())
    myWindow.SetContent(fyne.NewContainerWithLayout(layout.NewGridLayout(2),
        img1, img2, img3))
    myWindow.Resize(fyne.NewSize(300, 300))
    myWindow.ShowAndRun()
}
```

运行效果：

该布局有个优势，我们缩放界面时，控件会自动调整大小。试试看~

GridWrapLayout

`GridWrapLayout` 是 `GridLayout` 的扩展。`GridWrapLayout` 创建时会指定一个初始 `size`，这个 `size` 会应用到所有的子控件上，每个子控件都保持这个 `size`。初始，每行一个控件。如果界面大小变化了，这些子控件会重新排列。例如宽度翻倍了，那么一行就可以排两个控件了。有点像流动布局：

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Grid Wrap Layout")

  img1 := canvas.NewImageFromResource(theme.FyneLogo())
  img2 := canvas.NewImageFromResource(theme.FyneLogo())
  img3 := canvas.NewImageFromResource(theme.FyneLogo())
  myWindow.SetContent(
    fyne.NewContainerWithLayout(
      layout.NewGridWrapLayout(fyne.NewSize(150, 150)),
      img1, img2, img3))
  myWindow.ShowAndRun()
}
```

初始：

加大宽度：

再加大宽度：

BorderLayout

边框布局（`BorderLayout`）比较常用于构建用户界面，上面例子中的 `Toolbar` 一般都和 `BorderLayout` 搭配使用。创建方法 `layout.NewBorderLayout(top, bottom, left, right)`，分别传入顶部、底部、左侧、右侧的控件对象。添加到容器中的控件如果是这些**边界**对象，则显示在对应位置，其他都显示在中心：

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Border Layout")

  left := canvas.NewText("left", color.White)
  right := canvas.NewText("right", color.White)
  top := canvas.NewText("top", color.White)
  bottom := canvas.NewText("bottom", color.White)
  content := widget.NewLabel(`Lorem ipsum dolor,
  sit amet consectetur adipisicing elit.
```

```
    Quidem consectetur ipsam nesciunt,
    quasi sint expedita minus aut,
    porro iusto magnam ducimus voluptates cum vitae.
    Vero adipisci earum iure consequatur quidem.`)

    container := fyne.NewContainerWithLayout(
        layout.NewBorderLayout(top, bottom, left, right),
        top, bottom, left, right, content,
    )
    myWindow.SetContent(container)
    myWindow.ShowAndRun()
}
```

效果：

FormLayout

表单布局（ `FormLayout` ）其实就是一个 2 列的 `GridLayout` ，但是针对表单做了一些微调。

```
func main() {
    myApp := app.New()
    myWindow := myApp.NewWindow("Border Layout")

    nameLabel := canvas.NewText("Name", color.Black)
    nameValue := canvas.NewText("dajun", color.White)
    ageLabel := canvas.NewText("Age", color.Black)
    ageValue := canvas.NewText("18", color.White)

    container := fyne.NewContainerWithLayout(
        layout.NewFormLayout(),
        nameLabel, nameValue, ageLabel, ageValue,
    )
    myWindow.SetContent(container)
    myWindow.Resize(fyne.NewSize(150, 150))
    myWindow.ShowAndRun()
}
```

运行效果：

CenterLayout

`CenterLayout` 将容器内的所有控件显示在中心位置，按传入的顺序显示。最后传入的控件显示最上层。`CenterLayout` 中所有控件将保持它们的最小尺寸（大小能容纳其内容）。

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Center Layout")

  image := canvas.NewImageFromResource(theme.FyneLogo())
  image.FillMode = canvas.ImageFillOriginal
  text := canvas.NewText("Fyne Logo", color.Black)

  container := fyne.NewContainerWithLayout(
    layout.NewCenterLayout(),
    image, text,
  )
  myWindow.SetContent(container)
  myWindow.ShowAndRun()
}
```

运行结果：

字符串 `Fyne Logo` 显示在图片上层。如果我们把 `text` 和 `image` 顺序对调，字符串将会被图片挡住，无法看到。动手试一下～

MaxLayout

`MaxLayout` 与 `CenterLayout` 类似，不同之处在于 `MaxLayout` 会让容器内的元素都显示为最大尺寸（等于容器的大小）。细心的朋友可能发现了，在 `CenterLayout` 的示例中。我们设置了图片的填充模式为 `ImageFillOriginal` 。如果不设置填充模式，图片的默认 `MinSize` 为 `(1, 1)` 。可以 `fmt.Println(image.MinSize())` 验证一下。这样图片就不会显示在界面中。

在 `MaxLayout` 的容器中，我们不需要这样处理：

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Max Layout")

  image := canvas.NewImageFromResource(theme.FyneLogo())
  text := canvas.NewText("Fyne Logo", color.Black)

  container := fyne.NewContainerWithLayout(
    layout.NewMaxLayout(),
    image, text,
```

```
  )
  myWindow.SetContent(container)
  myWindow.Resize(fyne.Size(200, 200))
  myWindow.ShowAndRun()
}
```

运行结果：

注意，`canvas.Text`显示为左对齐了。如果要居中对齐，设置其`Alignment`属性为`fyne.TextAlignCenter`。

自定义 Layout

内置布局在子包`layout`中。它们都实现了`fyne.Layout`接口：

```
// src/fyne.io/fyne/layout.go
type Layout interface {
  Layout([]CanvasObject, Size)
  MinSize(objects []CanvasObject) Size
}
```

要实现自定义的布局，只需要实现这个接口。下面我们实现一个台阶（对角）的布局，好似一个矩阵的对角线，从左上到右下。首先定义一个新的类型。然后实现接口`fyne.Layout`的两个方法：

```
type diagonal struct {
}

func (d *diagonal) MinSize(objects []fyne.CanvasObject) fyne.Size {
  w, h := 0, 0
  for _, o := range objects {
    childSize := o.MinSize()

    w += childSize.Width
    h += childSize.Height
  }

  return fyne.NewSize(w, h)
}

func (d *diagonal) Layout(objects []fyne.CanvasObject, containerSize fyne.Size)
 {
  pos := fyne.NewPos(0, 0)
  for _, o := range objects {
```

```
        size := o.MinSize()
        o.Resize(size)
        o.Move(pos)

        pos = pos.Add(fyne.NewPos(size.Width, size.Height))
    }
}
```

`MinSize()` 返回所有子控件的 `MinSize` 之和。 `Layout()` 从左上到右下排列控件。然后是使用：

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Diagonal Layout")

  img1 := canvas.NewImageFromResource(theme.FyneLogo())
  img1.FillMode = canvas.ImageFillOriginal
  img2 := canvas.NewImageFromResource(theme.FyneLogo())
  img2.FillMode = canvas.ImageFillOriginal
  img3 := canvas.NewImageFromResource(theme.FyneLogo())
  img3.FillMode = canvas.ImageFillOriginal

  container := fyne.NewContainerWithLayout(
    &diagonal{},
    img1, img2, img3,
  )
  myWindow.SetContent(container)
  myWindow.ShowAndRun()
}
```

运行结果：

fyne demo

`fyne` 提供了一个 Demo，演示了大部分控件和布局的使用。可使用下面命令安装，执行：

```
$ go get fyne.io/fyne/cmd/fyne_demo
$ fyne_demo
```

效果图：

`fyne` 命令

`fyne` 库为了方便开发者提供了 `fyne` 命令。 `fyne` 可以用来将静态资源打包进可执行程序，还能将整个应用程序打包成可发布的形式。 `fyne` 命令通过下面命令安装：

```
$ go get fyne.io/fyne/cmd/fyne
```

安装完成之后 `fyne` 就在 `$GOPATH/bin` 目录中，将 `$GOPATH/bin` 添加到系统 `$PATH` 中就可以直接运行 `fyne` 命令了。

静态资源

其实在前面的示例中我们已经多次使用了 `fyne` 内置的静态资源，使用最多的要属 `fyne.FyneLogo()` 了。下面我们有两个图片 `image1.png/image2.jpg` 。我们使用 `fyne bundle` 命令将这两个图片打包进代码：

```
$ fyne bundle image1.png >> bundled.go
$ fyne bundle -append image2.jpg >> bundled.go
```

第二个命令指定 `-append` 选项表示添加到现有文件中，生成的文件如下：

```
// bundled.go
package main

import "fyne.io/fyne"

var resourceImage1Png = &fyne.StaticResource{
    StaticName: "image1.png",
    StaticContent: []byte{...}}

var resourceImage2Jpg = &fyne.StaticResource{
    StaticName: "image2.jpg",
    StaticContent: []byte{...}}
```

实际上就是将图片内容存入一个字节切片中，我们在代码中就可以调用 `canvas.NewImageFromResource()` ，传入 `resourceImage1Png` 或 `resourceImage2Jpg` 来创建 `canvas.Image` 对象了。

```
func main() {
  myApp := app.New()
  myWindow := myApp.NewWindow("Bundle Resource")
```

```go
	img1 := canvas.NewImageFromResource(resourceImage1Png)
	img1.FillMode = canvas.ImageFillOriginal
	img2 := canvas.NewImageFromResource(resourceImage2Jpg)
	img2.FillMode = canvas.ImageFillOriginal
	img3 := canvas.NewImageFromResource(theme.FyneLogo())
	img3.FillMode = canvas.ImageFillOriginal

	container := fyne.NewContainerWithLayout(
		layout.NewGridLayout(1),
		img1, img2, img3,
	)
	myWindow.SetContent(container)
	myWindow.ShowAndRun()
}
```

运行结果：

注意，由于现在是两个文件，不能使用 `go run main.go` ，应该用 `go run .` 。

`theme.FyneLogo()` 实际上是也是提前打包进代码的，代码文件是 `bundled-icons.go` ：

```go
// src/fyne.io/fyne/theme/icons.go
func FyneLogo() fyne.Resource {
	return fynelogo
}

// src/fyne.io/fyne/theme/bundled-icons.go
var fynelogo = &fyne.StaticResource{
	StaticName: "fyne.png",
	StaticContent: []byte{}}
```

发布应用程序

发布图像应用程序到多个操作系统是非常复杂的任务。图形界面应用程序通常有图标和一些元数据。 `fyne` 命令提供了将应用程序发布到多个平台的支持。使用 `fyne package` 命令将创建一个可在其它计算机上安装/运行的应用程序。在 Windows 上， `fyne package` 会创建一个 `.exe` 文件。在 macOS 上，会创建一个 `.app` 文件。在 Linux 上，会生成一个 `.tar.xz` 文件，可手动安装。

我们将上面的应用程序打包成一个 `exe` 文件：

```
$ fyne package -os windows -icon icon.jpg
```

上面命令会在同目录下生成两个文件 `bundle.exe` 和 `fyne.syso` ，将这两个文件拷贝到任何目录或其他 Windows 计算机都可以通过直接双击 `bundle.exe` 运行了。没有其他的依赖。

bundle.exe

bundled.go

fyne.syso

icon.jpg

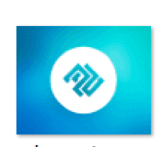



image1.png

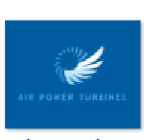



image2.jpg

main.go

`fyne` 还支持交叉编译，能在 windows 上编译 mac 的应用程序，不过需要安装额外的工具，感兴趣可自行探索。

总结

`fyne` 提供了丰富的组件和功能，我们介绍的只是很基础的一部分，还有剪切板、快捷键、滚动条、菜单等等等等内容。 `fyne` 命令实现打包静态资源和应用程序，非常方便。 `fyne` 还有其他高级功能留待大家探索、挖掘~

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. fyne GitHub：https://github.com/fyne-io/fyne
2. fyne 官网：https://fyne.io/
3. fyne 官方入门教程：https://developer.fyne.io/tour/introduction/hello.html
4. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

casbin

简介

权限管理在几乎每个系统中都是必备的模块。如果项目开发每次都要实现一次权限管理，无疑会浪费开发时间，增加开发成本。因此， `casbin` 库出现了。 `casbin` 是一个强大、高效的访问控制库。支持常用的多种访问控制模型，如 `ACL/RBAC/ABAC` 等。可以实现灵活的访问权限控制。同时， `casbin` 支持多种编程语言， `Go/Java/Node/PHP/Python/.NET/Rust` 。我们只需要**一次学习，多处运用**。

快速使用

我们依然使用 Go Module 编写代码，先初始化：

```
$ mkdir casbin && cd casbin
$ go mod init github.com/darjun/go-daily-lib/casbin
```

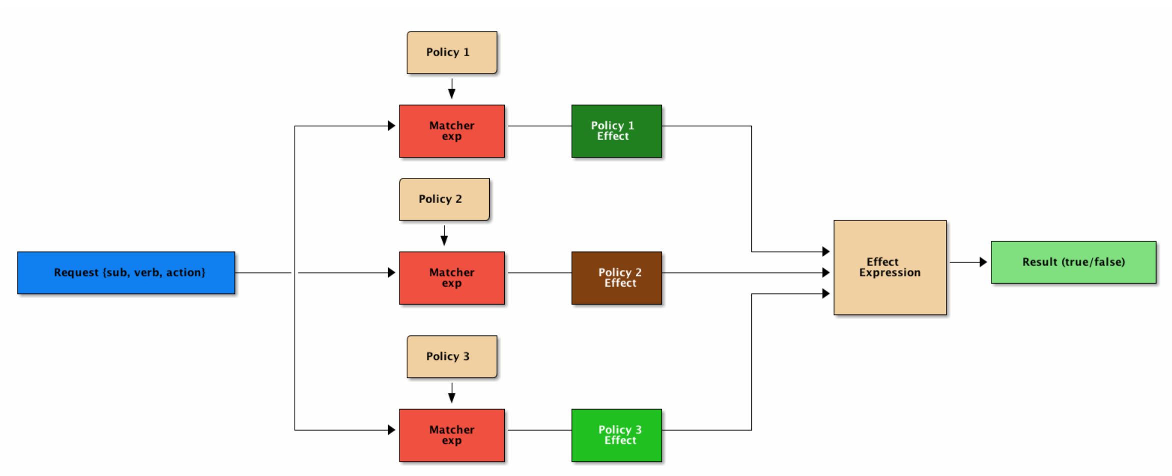
然后安装 `casbin` ，目前是 `v2` 版本：

```
$ go get github.com/casbin/casbin/v2
```

权限实际上就是控制**谁**能对**什么资源**进行什么操作。 `casbin` 将访问控制模型抽象到一个基于 PERM（Policy，Effect，Request，Matchers） 元模型的配置文件（模型文件）中。因此切换或更新授权机制只需要简单地修改配置文件。

- `policy` 是策略或者说是规则的定义。它定义了具体的规则。
- `request` 是对访问请求的抽象，它与 `e.Enforce()` 函数的参数是一一对应的
- `matcher` 匹配器会将请求与定义的每个 `policy` 一一匹配，生成多个匹配结果。
- `effect` 根据对请求运用匹配器得出的所有结果进行汇总，来决定该请求是**允许**还是**拒绝**。

下面这张图很好地描绘了这个过程：



我们首先编写模型文件：

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[matchers]
m = r.sub == p.sub && r.obj == p.obj && r.act == p.act

[policy_effect]
e = some(where (p.eft == allow))
```

上面模型文件规定了权限由 `sub,obj,act` 三要素组成，只有在策略列表中有和它完全相同的策略时，该请求才能通过。匹配器的结果可以通过 `p.eft` 获取， `some(where (p.eft == allow))` 表示只要有一条策略允许即可。

然后我们策略文件（即谁能对什么资源进行什么操作）：

```
p, dajun, data1, read
p, lizi, data2, write
```

上面 `policy.csv` 文件的两行内容表示 `dajun` 对数据 `data1` 有 `read` 权限， `lizi` 对数据 `data2` 有 `write` 权限。

接下来就是使用的代码：

```
package main

import (
  "fmt"
```

```go
	"log"

	"github.com/casbin/casbin/v2"
)

func check(e *casbin.Enforcer, sub, obj, act string) {
	ok, _ := e.Enforce(sub, obj, act)
	if ok {
		fmt.Printf("%s CAN %s %s\n", sub, act, obj)
	} else {
		fmt.Printf("%s CANNOT %s %s\n", sub, act, obj)
	}
}

func main() {
	e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
	if err != nil {
		log.Fatalf("NewEnforecer failed:%v\n", err)
	}

	check(e, "dajun", "data1", "read")
	check(e, "lizi", "data2", "write")
	check(e, "dajun", "data1", "write")
	check(e, "dajun", "data2", "read")
}
```

代码其实不复杂。首先创建一个 `casbin.Enforcer` 对象，加载模型文件 `model.conf` 和策略文件 `policy.csv` ，调用其 `Enforce` 方法来检查权限。运行程序：

```
$ go run main.go
dajun CAN read data1
lizi CAN write data2
dajun CANNOT write data1
dajun CANNOT read data2
```

请求必须完全匹配某条策略才能通过。 `("dajun", "data1", "read")` 匹配 `p, dajun, data1, read` ， `("lizi", "data2", "write")` 匹配 `p, lizi, data2, write` ，所以前两个检查通过。第 3 个因为 `"dajun"` 没有对 `data1` 的 `write` 权限，第 4 个因为 `dajun` 对 `data2` 没有 `read` 权限，所以检查都不能通过。输出结果符合预期。

`sub/obj/act` 依次对应传给 `Enforce` 方法的三个参数。实际上这里的 `sub/obj/act` 和 `read/write/data1/data2` 是我自己随便取的，你完全可以使用其它的名字，只要能前后一致即可。

上面例子中实现的就是 `ACL` （access-control-list，访问控制列表）。 `ACL` 显示定义了每个主体对每个资源的权限情况，未定义的就没有权限。我们还可以加上超级管理员，超级管理员可以进行任何操作。假设超级管理员为 `root` ，我们只需要修改匹配器：

```
[matchers]
e = r.sub == p.sub && r.obj == p.obj && r.act == p.act || r.sub == "root"
```

只要访问主体是 `root` 一律放行。

验证：

```
func main() {
  e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
  if err != nil {
    log.Fatalf("NewEnforecer failed:%v\n", err)
  }

  check(e, "root", "data1", "read")
  check(e, "root", "data2", "write")
  check(e, "root", "data1", "execute")
  check(e, "root", "data3", "rwx")
}
```

因为 `sub = "root"` 时，匹配器一定能通过，运行结果：

```
$ go run main.go
root CAN read data1
root CAN write data2
root CAN execute data1
root CAN rwx data3
```

RBAC 模型

`ACL` 模型在用户和资源都比较少的情况下没什么问题，但是用户和资源量一大， `ACL` 就会变得异常繁琐。想象一下，每次新增一个用户，都要把他需要的权限重新设置一遍是多么地痛苦。 `RBAC` （role-based-access-control）模型通过引入角色（ `role` ）这个中间层来解决这个问题。每个用户都属于一个角色，例如开发者、管理员、运维等，每个角色都有其特定的权限，权限的增加和删除都通过角色来进行。这样新增一个用户时，我们只需要给他指派一个角色，他就能拥有该角色的所有权限。修改角色的权限时，属于这个角色的用户权限就会相应的修改。

在 `casbin` 中使用 `RBAC` 模型需要在模型文件中添加 `role_definition` 模块：

```
[role_definition]
g = _, _

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

`g = _,_` 定义了用户——角色，角色——角色的映射关系，前者是后者的成员，拥有后者的权限。然后在匹配器中，我们不需要判断 `r.sub` 与 `p.sub` 完全相等，只需要使用 `g(r.sub, p.sub)` 来判断请求主体 `r.sub` 是否属于 `p.sub` 这个角色即可。最后我们修改策略文件添加用户——角色定义：

```
p, admin, data, read
p, admin, data, write
p, developer, data, read
g, dajun, admin
g, lizi, developer
```

上面的 `policy.csv` 文件规定了，`dajun` 属于 `admin` 管理员，`lizi` 属于 `developer` 开发者，使用 `g` 来定义这层关系。另外 `admin` 对数据 `data` 用 `read` 和 `write` 权限，而 `developer` 对数据 `data` 只有 `read` 权限。

```
package main

import (
  "fmt"
  "log"

  "github.com/casbin/casbin/v2"
)

func check(e *casbin.Enforcer, sub, obj, act string) {
  ok, _ := e.Enforce(sub, obj, act)
  if ok {
    fmt.Printf("%s CAN %s %s\n", sub, act, obj)
  } else {
    fmt.Printf("%s CANNOT %s %s\n", sub, act, obj)
  }
}

func main() {
  e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
  if err != nil {
    log.Fatalf("NewEnforecer failed:%v\n", err)
```

```
  }

  check(e, "dajun", "data", "read")
  check(e, "dajun", "data", "write")
  check(e, "lizi", "data", "read")
  check(e, "lizi", "data", "write")
}
```

很显然 `lizi` 所属角色没有 `write` 权限：

```
dajun CAN read data
dajun CAN write data
lizi CAN read data
lizi CANNOT write data
```

多个 `RBAC`

`casbin` 支持同时存在多个 `RBAC` 系统，即用户和资源都有角色：

```
[role_definition]
g=_,_
g2=_,_

[matchers]
m = g(r.sub, p.sub) && g2(r.obj, p.obj) && r.act == p.act
```

上面的模型文件定义了两个 `RBAC` 系统 `g` 和 `g2` ，我们在匹配器中使用 `g(r.sub, p.sub)` 判断请求主体属于特定组， `g2(r.obj, p.obj)` 判断请求资源属于特定组，且操作一致即可放行。

策略文件:

```
p, admin, prod, read
p, admin, prod, write
p, admin, dev, read
p, admin, dev, write
p, developer, dev, read
p, developer, dev, write
p, developer, prod, read
g, dajun, admin
g, lizi, developer
g2, prod.data, prod
g2, dev.data, dev
```

先看角色关系，即最后 4 行， `dajun` 属于 `admin` 角色， `lizi` 属于 `developer` 角色， `prod.data` 属于生产资源 `prod` 角色， `dev.data` 属于开发资源 `dev` 角色。 `admin` 角色拥有对 `prod` 和 `dev` 类资源的读写权限， `developer` 只能拥有对 `dev` 的读写权限和 `prod` 的读权限。

```
check(e, "dajun", "prod.data", "read")
check(e, "dajun", "prod.data", "write")
check(e, "lizi", "dev.data", "read")
check(e, "lizi", "dev.data", "write")
check(e, "lizi", "prod.data", "write")
```

第一个函数中 `e.Enforce()` 方法在实际执行的时候先获取 `dajun` 所属角色 `admin` ，再获取 `prod.data` 所属角色 `prod` ，根据文件中第一行 `p, admin, prod, read` 允许请求。最后一个函数中 `lizi` 属于角色 `developer` ，而 `prod.data` 属于角色 `prod` ，所有策略都不允许，故该请求被拒绝：

```
dajun CAN read prod.data
dajun CAN write prod.data
lizi CAN read dev.data
lizi CAN write dev.data
lizi CANNOT write prod.data
```

多层角色

`casbin` 还能为角色定义所属角色，从而实现多层角色关系，这种权限关系是可以传递的。例如 `dajun` 属于高级开发者 `senior` ， `seinor` 属于开发者，那么 `dajun` 也属于开发者，拥有开发者的所有权限。我们可以定义开发者共有的权限，然后额外为 `senior` 定义一些特殊的权限。

模型文件不用修改，策略文件改动如下：

```
p, senior, data, write
p, developer, data, read
g, dajun, senior
g, senior, developer
g, lizi, developer
```

上面 `policy.csv` 文件定义了高级开发者 `senior` 对数据 `data` 有 `write` 权限，普通开发者 `developer` 对数据只有 `read` 权限。同时 `senior` 也是 `developer` ，所以 `senior` 也继承其 `read` 权限。 `dajun` 属于 `senior` ，所以 `dajun` 对 `data` 有 `read` 和 `write` 权限，而 `lizi` 只属于 `developer` ，对数据 `data` 只有 `read` 权限。

```
check(e, "dajun", "data", "read")
check(e, "dajun", "data", "write")
check(e, "lizi", "data", "read")
check(e, "lizi", "data", "write")
```

`RBAC` domain

在 `casbin` 中，角色可以是全局的，也可以是特定 `domain` （领域）或 `tenant` （租户），可以简单理解为**组**。例如 `dajun` 在组 `tenant1` 中是管理员，拥有比较高的权限，在 `tenant2` 可能只是个弟弟。

使用 `RBAC domain` 需要对模型文件做以下修改：

```
[request_definition]
r = sub, dom, obj, act

[policy_definition]
p = sub, dom, obj, act

[role_definition]
g = _,_,_

[matchers]
m = g(r.sub, p.sub, r.dom) && r.dom == p.dom && r.obj == p.obj && r.act == p.obj
```

`g=_,_,_` 表示前者在后者中拥有中间定义的角色，在匹配器中使用 `g` 要带上 `dom` 。

```
p, admin, tenant1, data1, read
p, admin, tenant2, data2, read
g, dajun, admin, tenant1
g, dajun, developer, tenant2
```

在 `tenant1` 中，只有 `admin` 可以读取数据 `data1` 。在 `tenant2` 中，只有 `admin` 可以读取数据 `data2` 。 `dajun` 在 `tenant1` 中是 `admin` ，但是在 `tenant2` 中不是。

```
func check(e *casbin.Enforcer, sub, domain, obj, act string) {
  ok, _ := e.Enforce(sub, domain, obj, act)
  if ok {
    fmt.Printf("%s CAN %s %s in %s\n", sub, act, obj, domain)
  } else {
    fmt.Printf("%s CANNOT %s %s in %s\n", sub, act, obj, domain)
```

```
    }
}

func main() {
    e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
    if err != nil {
        log.Fatalf("NewEnforecer failed:%v\n", err)
    }

    check(e, "dajun", "tenant1", "data1", "read")
    check(e, "dajun", "tenant2", "data2", "read")
}
```

结果不出意料：

```
dajun CAN read data1 in tenant1
dajun CANNOT read data2 in tenant2
```

ABAC

`RBAC` 模型对于实现比较规则的、相对静态的权限管理非常有用。但是对于特殊的、动态的需求，`RBAC` 就显得有点力不从心了。例如，我们在不同的时间段对数据 `data` 实现不同的权限控制。正常工作时间 `9:00-18:00` 所有人都可以读写 `data` ，其他时间只有数据所有者能读写。这种需求我们可以很方便地使用 `ABAC` （attribute base access list）模型完成：

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[matchers]
m = r.sub.Hour >= 9 && r.sub.Hour < 18 || r.sub.Name == r.obj.Owner

[policy_effect]
e = some(where (p.eft == allow))
```

该规则不需要策略文件：

```
type Object struct {
    Name  string
    Owner string
```

```
}

type Subject struct {
  Name string
  Hour int
}

func check(e *casbin.Enforcer, sub Subject, obj Object, act string) {
  ok, _ := e.Enforce(sub, obj, act)
  if ok {
    fmt.Printf("%s CAN %s %s at %d:00\n", sub.Name, act, obj.Name, sub.Hour)
  } else {
    fmt.Printf("%s CANNOT %s %s at %d:00\n", sub.Name, act, obj.Name, sub.Hour)
  }
}

func main() {
  e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
  if err != nil {
    log.Fatalf("NewEnforecer failed:%v\n", err)
  }

  o := Object{"data", "dajun"}
  s1 := Subject{"dajun", 10}
  check(e, s1, o, "read")

  s2 := Subject{"lizi", 10}
  check(e, s2, o, "read")

  s3 := Subject{"dajun", 20}
  check(e, s3, o, "read")

  s4 := Subject{"lizi", 20}
  check(e, s4, o, "read")
}
```

显然 `lizi` 在 `20:00` 不能 `read` 数据 `data` ：

```
dajun CAN read data at 10:00
lizi CAN read data at 10:00
dajun CAN read data at 20:00
lizi CANNOT read data at 20:00
```

我们知道，在 `model.conf` 文件中可以通过 `r.sub` 和 `r.obj` ， `r.act` 来访问传给 `Enforce` 方法的参数。实际上 `sub/obj` 可以是结构体对象，得益

于 `govaluate` 库的强大功能，我们可以在 `model.conf` 文件中获取这些结构体的字段值。如上面的 `r.sub.Name` 、 `r.Obj.Owner` 等。 `govaluate` 库的内容可以参见我之前的一篇文章《Go 每日一库之 govaluate》。

使用 `ABAC` 模型可以非常灵活的权限控制，但是一般情况下 `RBAC` 就已经够用了。

模型存储

上面代码中，我们一直将模型存储在文件中。 `casbin` 也可以实现在代码中动态初始化模型，例如 `get-started` 的例子可以改写为：

```
func main() {
  m := model.NewModel()
  m.AddDef("r", "r", "sub, obj, act")
  m.AddDef("p", "p", "sub, obj, act")
  m.AddDef("e", "e", "some(where (p.eft == allow))")
  m.AddDef("m", "m", "r.sub == g.sub && r.obj == p.obj && r.act == p.act")

  a := fileadapter.NewAdapter("./policy.csv")
  e, err := casbin.NewEnforcer(m, a)
  if err != nil {
    log.Fatalf("NewEnforecer failed:%v\n", err)
  }

  check(e, "dajun", "data1", "read")
  check(e, "lizi", "data2", "write")
  check(e, "dajun", "data1", "write")
  check(e, "dajun", "data2", "read")
}
```

同样地，我们也可以从字符串中加载模型：

```
func main() {
  text := `
  [request_definition]
  r = sub, obj, act

  [policy_definition]
  p = sub, obj, act

  [policy_effect]
  e = some(where (p.eft == allow))

  [matchers]
```

```
  m = r.sub == p.sub && r.obj == p.obj && r.act == p.act
  `

  m, _ := model.NewModelFromString(text)
  a := fileadapter.NewAdapter("./policy.csv")
  e, _ := casbin.NewEnforcer(m, a)

  check(e, "dajun", "data1", "read")
  check(e, "lizi", "data2", "write")
  check(e, "dajun", "data1", "write")
  check(e, "dajun", "data2", "read")
}
```

但是这两种方式并不推荐。

策略存储

在前面的例子中，我们都是将策略存储在 `policy.csv` 文件中。一般在实际应用中，很少使用文件存储。 `casbin` 以第三方适配器的方式支持多种存储方式包括 `MySQL/MongoDB/Redis/Etcd` 等，还可以实现自己的存储。完整列表看这里 https://casbin.org/docs/en/adapters。下面我们介绍使用 `Gorm Adapter` 。先连接到数据库，执行下面的 `SQL` ：

```
CREATE DATABASE IF NOT EXISTS casbin;

USE casbin;

CREATE TABLE IF NOT EXISTS casbin_rule (
  p_type VARCHAR(100) NOT NULL,
  v0 VARCHAR(100),
  v1 VARCHAR(100),
  v2 VARCHAR(100),
  v3 VARCHAR(100),
  v4 VARCHAR(100),
  v5 VARCHAR(100)
);

INSERT INTO casbin_rule VALUES
('p', 'dajun', 'data1', 'read', '', '', ''),
('p', 'lizi', 'data2', 'write', '', '', '');
```

然后使用 `Gorm Adapter` 加载 `policy` ， `Gorm Adapter` 默认使用 `casbin` 库中的 `casbin_rule` 表：

```
package main

import (
  "fmt"

  "github.com/casbin/casbin/v2"
  gormadapter "github.com/casbin/gorm-adapter/v2"
  _ "github.com/go-sql-driver/mysql"
)

func check(e *casbin.Enforcer, sub, obj, act string) {
  ok, _ := e.Enforce(sub, obj, act)
  if ok {
    fmt.Printf("%s CAN %s %s\n", sub, act, obj)
  } else {
    fmt.Printf("%s CANNOT %s %s\n", sub, act, obj)
  }
}

func main() {
  a, _ := gormadapter.NewAdapter("mysql", "root:12345@tcp(127.0.0.1:3306)/")
  e, _ := casbin.NewEnforcer("./model.conf", a)

  check(e, "dajun", "data1", "read")
  check(e, "lizi", "data2", "write")
  check(e, "dajun", "data1", "write")
  check(e, "dajun", "data2", "read")
}
```

运行：

```
dajun CAN read data1
lizi CAN write data2
dajun CANNOT write data1
dajun CANNOT read data2
```

使用函数

我们可以在匹配器中使用函数。`casbin` 内置了一些函数 `keyMatch/keyMatch2/keyMatch3/keyMatch4` 都是匹配 URL 路径的，`regexMatch` 使用正则匹配，`ipMatch` 匹配 IP 地址。参见 https://casbin.org/docs/en/function。使用内置函数我们能很容易对路由进行权限划分：

```
[matchers]
m = r.sub == p.sub && keyMatch(r.obj, p.obj) && r.act == p.act
```

```
p, dajun, user/dajun/*, read
p, lizi, user/lizi/*, read
```

不同用户只能访问其对应路由下的 URL：

```
func main() {
  e, err := casbin.NewEnforcer("./model.conf", "./policy.csv")
  if err != nil {
    log.Fatalf("NewEnforecer failed:%v\n", err)
  }

  check(e, "dajun", "user/dajun/1", "read")
  check(e, "lizi", "user/lizi/2", "read")
  check(e, "dajun", "user/lizi/1", "read")
}
```

输出：

```
dajun CAN read user/dajun/1
lizi CAN read user/lizi/2
dajun CANNOT read user/lizi/1
```

我们当然也可以定义自己的函数。先定义一个函数，返回 bool：

```
func KeyMatch(key1, key2 string) bool {
  i := strings.Index(key2, "*")
  if i == -1 {
    return key1 == key2
  }

  if len(key1) > i {
    return key1[:i] == key2[:i]
  }

  return key1 == key2[:i]
}
```

这里实现了一个简单的正则匹配，只处理 `*` 。

然后将这个函数用 `interface{}` 类型包装一层：

```
func KeyMatchFunc(args ...interface{}) (interface{}, error) {
  name1 := args[0].(string)
  name2 := args[1].(string)

  return (bool)(KeyMatch(name1, name2)), nil
}
```

然后添加到权限认证器中：

```
e.AddFunction("my_func", KeyMatchFunc)
```

这样我们就可以在匹配器中使用该函数实现正则匹配了：

```
[matchers]
m = r.sub == p.sub && my_func(r.obj, p.obj) && r.act == p.act
```

接下来我们在策略文件中为 `dajun` 赋予权限：

```
p, dajun, data/*, read
```

`dajun` 对匹配模式 `data/*` 的文件都有 `read` 权限。

验证一下：

```
check(e, "dajun", "data/1", "read")
check(e, "dajun", "data/2", "read")
check(e, "dajun", "data/1", "write")
check(e, "dajun", "mydata", "read")
```

`dajun` 对 `data/1` 没有 `write` 权限，`mydata` 不符合 `data/*` 模式，也没有 `read` 权限：

```
dajun CAN read data/1
dajun CAN read data/2
dajun CANNOT write data/1
dajun CANNOT read mydata
```

总结

`casbin` 功能强大，简单高效，且多语言通用。值得学习。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. casbin GitHub：https://github.com/casbin/casbin
2. casbin 官网：https://casbin.org/
3. 一种基于元模型的访问控制策略描述语言：http://www.jos.org.cn/html/2020/2/5624.htm
4. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

twirp

简介

twirp是一个基于 Google Protobuf 的 RPC 框架。`twirp` 通过在 `.proto` 文件中定义服务，然后自动生产服务器和客户端的代码。让我们可以将更多的精力放在业务逻辑上。咦？这不就是 gRPC 吗？不同的是，gRPC 自己实现了一套 HTTP 服务器和网络传输层，twirp 使用标准库 `net/http`。另外 gRPC 只支持 HTTP/2 协议，twirp 还可以运行在 HTTP 1.1 之上。同时 twirp 还可以使用 JSON 格式交互。当然并不是说 twirp 比 gRPC 好，只是多了解一种框架也就多了一个选择😊

快速使用

首先需要安装 twirp 的代码生成插件：

```
$ go get github.com/twitchtv/twirp/protoc-gen-twirp
```

上面命令会在 `$GOPATH/bin` 目录下生成可执行程序 `protoc-gen-twirp`。我的习惯是将 `$GOPATH/bin` 放到 PATH 中，所以可在任何地方执行该命令。

接下来安装 protobuf 编译器，直接到 GitHub 上 https://github.com/protocolbuffers/protobuf/releases下载编译好的二进制程序放到 PATH 目录即可。

最后是 Go 语言的 protobuf 生成插件：

```
$ go get github.com/golang/protobuf/protoc-gen-go
```

同样地，命令 `protoc-gen-go` 会安装到 `$GOPATH/bin` 目录中。

本文代码采用**Go Modules**。先创建目录，然后初始化：

```
$ mkdir twirp && cd twirp
$ go mod init github.com/darjun/go-daily-lib/twirp
```

接下来，我们开始代码编写。先编写 `.proto` 文件：

```
syntax = "proto3";
option go_package = "proto";
```

```
service Echo {
  rpc Say(Request) returns (Response);
}

message Request {
  string text = 1;
}

message Response {
  string text = 2;
}
```

我们定义一个 `service` 实现**echo**功能，即发送什么就返回什么。切换到 `echo.proto` 所在目录，使用 `protoc` 命令生成代码：

```
$ protoc --twirp_out=. --go_out=. ./echo.proto
```

上面命令会生成 `echo.pb.go` 和 `echo.twirp.go` 两个文件。前一个是 Go Protobuf 文件，后一个文件中包含了 `twirp` 的服务器和客户端代码。

然后我们就可以编写服务器和客户端程序了。服务器：

```
package main

import (
  "context"
  "net/http"

  "github.com/darjun/go-daily-lib/twirp/get-started/proto"
)

type Server struct{}

func (s *Server) Say(ctx context.Context, request *proto.Request) (*proto.Response, error) {
  return &proto.Response{Text: request.GetText()}, nil
}

func main() {
  server := &Server{}
  twirpHandler := proto.NewEchoServer(server, nil)

  http.ListenAndServe(":8080", twirpHandler)
}
```

使用自动生成的代码，我们只需要 3 步即可完成一个 RPC 服务器：

1. 定义一个结构，可以存储一些状态。让它实现我们定义的`service`接口；
2. 创建一个该结构的对象，调用生成的`New{{ServiceName}}Server`方法创建`net/http`需要的处理器，这里的`ServiceName`为我们的服务名；
3. 监听端口。

客户端：

```
package main

import (
  "context"
  "fmt"
  "log"
  "net/http"

  "github.com/darjun/go-daily-lib/twirp/get-started/proto"
)

func main() {
  client := proto.NewEchoProtobufClient("http://localhost:8080", &http.Client{})

  response, err := client.Say(context.Background(), &proto.Request{Text: "Hello World"})
  if err != nil {
    log.Fatal(err)
  }
  fmt.Printf("response:%s\n", response.GetText())
}
```

`twirp`也生成了客户端相关代码，直接调用`NewEchoProtobufClient`连接到对应的服务器，然后调用`rpc`请求。

开启两个控制台，分别运行服务器和客户端程序。服务器：

```
$ cd server && go run main.go
```

客户端：

```
$ cd client && go run main.go
```

正确返回结果：

```
response:Hello World
```

为了便于对照，下面列出该程序的目录结构。也可以去我的 GitHub 上查看示例代码：

```
get-started
├── client
│   └── main.go
├── proto
│   ├── echo.pb.go
│   ├── echo.proto
│   └── echo.twirp.go
└── server
    └── main.go
```

JSON 客户端

除了使用 Protobuf，`twirp` 还支持 JSON 格式的请求。使用也非常简单，只需要在创建 `Client` 时将 `NewEchoProtobufClient` 改为 `NewEchoJSONClient` 即可：

```
func main() {
  client := proto.NewEchoJSONClient("http://localhost:8080", &http.Client{})

  response, err := client.Say(context.Background(), &proto.Request{Text: "Hello
 World"})
  if err != nil {
    log.Fatal(err)
  }
  fmt.Printf("response:%s\n", response.GetText())
}
```

Protobuf Client 发送的请求带有 `Content-Type: application/protobuf` 的 `Header` ，JSON Client 则设置 `Content-Type` 为 `application/json` 。服务器收到请求时根据 `Content-Type` 来区分请求类型：

```
// proto/echo.twirp.go
unc (s *echoServer) serveSay(ctx context.Context, resp http.ResponseWriter, req
 *http.Request) {
  header := req.Header.Get("Content-Type")
  i := strings.Index(header, ";")
  if i == -1 {
    i = len(header)
```

```
  }
  switch strings.TrimSpace(strings.ToLower(header[:i])) {
  case "application/json":
    s.serveSayJSON(ctx, resp, req)
  case "application/protobuf":
    s.serveSayProtobuf(ctx, resp, req)
  default:
    msg := fmt.Sprintf("unexpected Content-Type: %q", req.Header.Get("Content-Ty
pe"))
    twerr := badRouteError(msg, req.Method, req.URL.Path)
    s.writeError(ctx, resp, twerr)
  }
}
```

提供其他 HTTP 服务

实际上，`twirpHandler`只是一个`http`的处理器，正如其他千千万万的处理器一样，没什么特殊的。我们当然可以挂载我们自己的处理器或处理器函数（概念有不清楚的可以参见我的《Go Web 编程》系列文章：

```
type Server struct{}

func (s *Server) Say(ctx context.Context, request *proto.Request) (*proto.Respon
se, error) {
  return &proto.Response{Text: request.GetText()}, nil
}

func greeting(w http.ResponseWriter, r *http.Request) {
  name := r.FormValue("name")
  if name == "" {
    name = "world"
  }

  w.Write([]byte("hi," + name))
}

func main() {
  server := &Server{}
  twirpHandler := proto.NewEchoServer(server, nil)

  mux := http.NewServeMux()
  mux.Handle(twirpHandler.PathPrefix(), twirpHandler)
  mux.HandleFunc("/greeting", greeting)
```

```
    err := http.ListenAndServe(":8080", mux)
    if err != nil {
        log.Fatal(err)
    }
}
```

上面程序挂载了一个简单的 `/greeting` 请求，可以通过浏览器来请求地址 `http://localhost:8080/greeting` 。 `twirp` 的请求会挂载到路径 `twirp/{{ServiceName}}` 这个路径下，其中 `ServiceName` 为服务名。上面程序中的 `PathPrefix()` 会返回 `/twirp/Echo` 。

客户端：

```
func main() {
    client := proto.NewEchoProtobufClient("http://localhost:8080", &http.Client{})

    response, _ := client.Say(context.Background(), &proto.Request{Text: "Hello World"})
    fmt.Println("echo:", response.GetText())

    httpResp, _ := http.Get("http://localhost:8080/greeting")
    data, _ := ioutil.ReadAll(httpResp.Body)
    httpResp.Body.Close()
    fmt.Println("greeting:", string(data))

    httpResp, _ = http.Get("http://localhost:8080/greeting?name=dj")
    data, _ = ioutil.ReadAll(httpResp.Body)
    httpResp.Body.Close()
    fmt.Println("greeting:", string(data))
}
```

先运行服务器，然后执行客户端程序：

```
$ go run main.go
echo: Hello World
greeting: hi,world
greeting: hi,dj
```

发送自定义的 Header

默认情况下， `twirp` 实现会发送一些 Header。例如上面介绍的，使用 `Content-Type` 辨别客户端使用的协议格式。有时候我们可能需要发送一些自定义的 Header，例如 `token` 。 `twirp` 提供了 `WithHTTPRequestHeaders` 方法实现这个功能，该方法

返回一个 `context.Context` 。发送时会将保存在该对象中的 Header 一并发送。类似地，服务器使用 `WithHTTPResponseHeaders` 发送自定义 Header。

由于 `twirp` 封装了 `net/http` ，导致外层拿不到原始的 `http.Request` 和 `http.Response` 对象，所以 Header 的读取有点麻烦。在服务器端， `NewEchoServer` 返回的是一个 `http.Handler` ，我们加一层中间件读取 `http.Request` 。看下面代码：

```
type Server struct{}

func (s *Server) Say(ctx context.Context, request *proto.Request) (*proto.Response, error) {
  token := ctx.Value("token").(string)
  fmt.Println("token:", token)

  err := twirp.SetHTTPResponseHeader(ctx, "Token-Lifecycle", "60")
  if err != nil {
    return nil, twirp.InternalErrorWith(err)
  }
  return &proto.Response{Text: request.GetText()}, nil
}

func WithTwirpToken(h http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    token := r.Header.Get("Twirp-Token")
    ctx = context.WithValue(ctx, "token", token)
    r = r.WithContext(ctx)

    h.ServeHTTP(w, r)
  })
}

func main() {
  server := &Server{}
  twirpHandler := proto.NewEchoServer(server, nil)
  wrapped := WithTwirpToken(twirpHandler)

  http.ListenAndServe(":8080", wrapped)
}
```

上面程序给客户端返回了一个名为 `Token-Lifecycle` 的 Header。客户端代码：

```
func main() {
  client := proto.NewEchoProtobufClient("http://localhost:8080", &http.Client{})
```

```
  header := make(http.Header)
  header.Set("Twirp-Token", "test-twirp-token")

  ctx := context.Background()
  ctx, err := twirp.WithHTTPRequestHeaders(ctx, header)
  if err != nil {
    log.Fatalf("twirp error setting headers: %v", err)
  }

  response, err := client.Say(ctx, &proto.Request{Text: "Hello World"})
  if err != nil {
    log.Fatalf("call say failed: %v", err)
  }
  fmt.Printf("response:%s\n", response.GetText())
}
```

运行程序，服务器正确获取客户端传过来的 token。

请求路由

我们前面已经介绍过了， `twirp` 的 `Server` 实际上也就是一个 `http.Handler` ，如果我们知道了它的挂载路径，完全可以通过浏览器或者 `curl` 之类的工具去请求。我们启动 `get-started` 的服务器，然后用 `curl` 命令行工具去请求：

```
$ curl --request "POST" \
  --location "http://localhost:8080/twirp/Echo/Say" \
  --header "Content-Type:application/json" \
  --data '{"text":"hello world"}'\
  --verbose
{"text":"hello world"}
```

这在调试的时候非常有用。

总结

本文介绍了 Go 的一个基于 Protobuf 生成代码的 RPC 框架，非常简单，小巧，实用。 `twirp` 对许多常用的编程语言都提供了支持。可以作为 gRPC 等的备选方案考虑。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. twirp GitHub：https://github.com/twitchtv/twirp
2. twirp 官方文档：https://twitchtv.github.io/twirp/docs/intro.html
3. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

rpcx

简介

在之前的两篇文章 `rpc` 和 `json-rpc` 中，我们介绍了 Go 标准库提供的 `rpc` 实现。在实际开发中， `rpc` 库的功能还是有所欠缺。今天我们介绍一个非常优秀的 Go RPC 库—— `rpcx` 。 `rpcx` 是一位国人大牛开发的，详细开发历程可以在 `rpcx` 官方博客了解。 `rpcx` 拥有媲美，甚至某种程度上超越 `gRPC` 的性能，有完善的中文文档，提供服务发现和治理的插件。

快速使用

本文示例使用 `go modules` 。

首先是安装：

```
$ go get -v -tags "reuseport quic kcp zookeeper etcd consul ping" github.com/smallnest/rpcx/...
```

可以看出 `rpcx` 的安装有点特殊。使用 `go get -v github.com/smallnest/rpcx/...` 命令只会安装 `rpcx` 的基础功能。扩展功能都是通过 `build tags` 指定。为了使用方便，一般安装所有的 `tags` ，如上面命令所示。这也是官方推荐的安装方式。

我们先编写服务端程序，实际上这个程序与用 `rpc` 标准库编写的程序几乎一模一样：

```
package main

import (
  "context"
  "errors"

  "github.com/smallnest/rpcx/server"
)

type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
```

```
}

type Arith int

func (t *Arith) Mul(cxt context.Context, args *Args, reply *int) error {
  *reply = args.A * args.B
  return nil
}

func (t *Arith) Div(cxt context.Context, args *Args, quo *Quotient) error {
  if args.B == 0 {
    return errors.New("divide by 0")
  }

  quo.Quo = args.A / args.B
  quo.Rem = args.A % args.B
  return nil
}

func main() {
  s := server.NewServer()
  s.RegisterName("Arith", new(Arith), "")
  s.Serve("tcp", ":8972")
}
```

首先创建一个 `Server` 对象，调用它的 `RegisterName()` 方法在服务路径 `Arith` 下注册 `Mul` 和 `Div` 方法。与标准库相比， `rpcx` 要求注册方法的第一个参数必须为 `context.Context` 类型。最后调用 `s.Serve("tcp", ":8972")` 监听 TCP 端口 8972。是不是很简单？启动服务器：

```
$ go run main.go
```

然后是客户端程序：

```
package main

import (
  "context"
  "flag"
  "log"

  "github.com/smallnest/rpcx/client"
)
```

```
var (
  addr = flag.String("addr", ":8972", "service address")
)

func main() {
  flag.Parse()

  d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
  xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d,
 client.DefaultOption)
  defer xclient.Close()

  args := &Args{A:10, B:20}
  var reply int

  err :=xclient.Call(context.Background(), "Mul", args, &reply)
  if err != nil {
    log.Fatalf("failed to call: %v", err)
  }

  fmt.Printf("%d * %d = %d\n", args.A, args.B, reply)

  args = &Args{50, 20}
  var quo Quotient
  err = xclient.Call(context.Background(), "Div", args, &quo)
  if err != nil {
    log.Fatalf("failed to call: %v", err)
  }

  fmt.Printf("%d * %d = %d...%d\n", args.A, args.B, quo.Quo, quo.Rem)
}
```

`rpcx` 支持多种服务发现的方式让客户端找到服务器。上面代码中我们使用的是最简单的点到点的方式，也就是**直连**。要调用服务端的方法，必须先创建一个 `Client` 对象。使用 `Client` 对象来调用远程方法。运行客户端：

```
$ go run main.go
10 * 20 = 200
50 * 20 = 2...10
```

注意到，创建 `Client` 对象的参数有 `client.Failtry` 和 `client.RandomSelect` 。这两个参数分别为**失败模式**和**如何选择服务器**。

传输

`rpcx` 支持多种传输协议：

- `TCP`：TCP 协议，网络名称为`tcp`；
- `HTTP`：HTTP 协议，网络名称为`http`；
- `UnixDomain`：unix 域协议，网络名称为`unix`；
- `QUIC`：是 Quick UDP Internet Connections 的缩写，意为**快速UDP网络连接**。HTTP/3 底层就是 QUIC 协议，Google 出品。网络名称为`quic`；
- `KCP`：快速并且可靠的 ARQ 协议，网络名称为`kcp`。

`rpcx` 对这些协议做了非常好的封装。除了在创建服务器和客户端连接时需要指定协议名称，其它时候的使用基本是透明的。我们将上面的例子改装成使用 `http` 协议的：

服务端改动：

```
s.Serve("http", ":8972")
```

客户端改动：

```
d := client.NewPeer2PeerDiscovery("http@"+*addr, "")
```

`QUIC` 和 `KCP` 的使用有点特殊， `QUIC` 必须与 TLS 一起使用， `KCP` 也需要做传输加密。使用 Go 语言我们能很方便地生成一个证书和私钥：

```
package main

import (
  "crypto/rand"
  "crypto/rsa"
  "crypto/x509"
  "crypto/x509/pkix"
  "encoding/pem"
  "math/big"
  "net"
  "os"
  "time"
)

func main() {
  max := new(big.Int).Lsh(big.NewInt(1), 128)
  serialNumber, _ := rand.Int(rand.Reader, max)
  subject := pkix.Name{
    Organization:       []string{"Go Daily Lib"},
```

```
		OrganizationalUnit: []string{"TechBlog"},
		CommonName:         "go daily lib",
	}

	template := x509.Certificate{
		SerialNumber: serialNumber,
		Subject:      subject,
		NotBefore:    time.Now(),
		NotAfter:     time.Now().Add(365 * 24 * time.Hour),
		KeyUsage:     x509.KeyUsageKeyEncipherment | x509.KeyUsageDigitalSignature,
		ExtKeyUsage:  []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
		IPAddresses:  []net.IP{net.ParseIP("127.0.0.1")},
	}

	pk, _ := rsa.GenerateKey(rand.Reader, 2048)

	derBytes, _ := x509.CreateCertificate(rand.Reader, &template, &template, &pk.PublicKey, pk)
	certOut, _ := os.Create("server.pem")
	pem.Encode(certOut, &pem.Block{Type: "CERTIFICATE", Bytes: derBytes})
	certOut.Close()

	keyOut, _ := os.Create("server.key")
	pem.Encode(keyOut, &pem.Block{Type: "RSA PRIVATE KEY", Bytes: x509.MarshalPKCS1PrivateKey(pk)})
	keyOut.Close()
}
```

上面代码生成了一个证书和私钥，有效期为 1 年。运行程序，得到两个文件 `server.pem` 和 `server.key` 。然后我们就可以编写使用 `QUIC` 协议的程序了。服务端：

```
func main() {
	cert, _ := tls.LoadX509KeyPair("server.pem", "server.key")
	config := &tls.Config{Certificates: []tls.Certificate{cert}}

	s := server.NewServer(server.WithTLSConfig(config))
	s.RegisterName("Arith", new(Arith), "")
	s.Serve("quic", "localhost:8972")
}
```

实际上就是加载证书和密钥，然后在创建 `Server` 对象时作为选项传入。客户端改动：

```
conf := &tls.Config{
	InsecureSkipVerify: true,
```

```
}

option := client.DefaultOption
option.TLSConfig = conf
d := client.NewPeer2PeerDiscovery("quic@"+*addr, "")
xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, op
tion)
defer xclient.Close()
```

客户端也需要配置 TLS。

有一点需要注意， `rpcx` 对 `quic/kcp` 这些协议的支持是通过 `build tags` 实现的。默认不会编译 `quic/kcp` 相关文件。如果要使用，必须自己手动指定 `tags` 。先启动服务端程序：

```
$ go run -tags quic main.go
```

然后切换到客户端程序目录，执行下面命令：

```
$ go run -tags quic main.go
```

还有一点需要注意，在使用 `tcp` 和 `http` （底层也是 `tcp` ）协议的时候，我们可以将地址简写为 `:8972` ，因为默认就是本地地址。但是 `quic` 不行，必须把地址写完整：

```
// 服务端
s.Serve("quic", "localhost:8972")
// 客户端
addr = flag.String("addr", "localhost:8972", "service address")
```

注册函数

上面的例子都是调用对象的方法，我们也可以调用函数。函数的类型与对象方法相比只是没有**接收者**。注册函数需要指定一个服务路径。服务端：

```
type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
}
```

```go
func Mul(cxt context.Context, args *Args, reply *int) error {
	*reply = args.A * args.B
	return nil
}

func Div(cxt context.Context, args *Args, quo *Quotient) error {
	if args.B == 0 {
		return errors.New("divide by 0")
	}

	quo.Quo = args.A / args.B
	quo.Rem = args.A % args.B
	return nil
}

func main() {
	s := server.NewServer()
	s.RegisterFunction("function", Mul, "")
	s.RegisterFunction("function", Div, "")
	s.Serve("tcp", ":8972")
}
```

只是注册方法由 `RegisterName` 变为了 `RegisterFunction` ，参数由一个对象变为一个函数。我们需要为注册的函数指定一个服务路径，客户端调用时会根据这个路径查找对应方法。客户端：

```go
func main() {
	flag.Parse()

	d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
	xclient := client.NewXClient("function", client.Failtry, client.RandomSelect, d, client.DefaultOption)
	defer xclient.Close()

	args := &Args{A: 10, B: 20}
	var reply int

	err := xclient.Call(context.Background(), "Mul", args, &reply)
	if err != nil {
		log.Fatalf("failed to call: %v", err)
	}

	fmt.Printf("%d * %d = %d\n", args.A, args.B, reply)
```

```
  args = &Args{50, 20}
  var quo Quotient
  err = xclient.Call(context.Background(), "Div", args, &quo)
  if err != nil {
    log.Fatalf("failed to call: %v", err)
  }

  fmt.Printf("%d * %d = %d...%d\n", args.A, args.B, quo.Quo, quo.Rem)
}
```

注册中心

`rpcx` 支持多种注册中心：

- 点对点：其实就是直连，没有注册中心；
- 点对多：可以配置多个服务器；
- `zookeeper` ：常用的注册中心；
- `Etcd` ：Go 语言编写的注册中心；
- 进程内调用：方便调试功能，在同一个进程内查找服务；
- `Consul/mDNS` 等。

我们之前演示的都是**点对点**的连接，接下来我们介绍如何使用 `zookeeper` 作为注册中心。在 `rpcx` 中，注册中心是通过插件的方式集成的。使用 `ZooKeeperRegisterPlugin` 这个插件来集成 `Zookeeper` 。服务端代码：

```
type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
}

var (
  addr     = flag.String("addr", ":8972", "service address")
  zkAddr   = flag.String("zkAddr", "127.0.0.1:2181", "zookeeper address")
  basePath = flag.String("basePath", "/services/math", "service base path")
)

type Arith int
```

```
func (t *Arith) Mul(cxt context.Context, args *Args, reply *int) error {
	fmt.Println("Mul on", *addr)
	*reply = args.A * args.B
	return nil
}

func (t *Arith) Div(cxt context.Context, args *Args, quo *Quotient) error {
	fmt.Println("Div on", *addr)
	if args.B == 0 {
		return errors.New("divide by 0")
	}

	quo.Quo = args.A / args.B
	quo.Rem = args.A % args.B
	return nil
}

func main() {
	flag.Parse()

	p := &serverplugin.ZooKeeperRegisterPlugin{
		ServiceAddress:   "tcp@" + *addr,
		ZooKeeperServers: []string{*zkAddr},
		BasePath:         *basePath,
		Metrics:          metrics.NewRegistry(),
		UpdateInterval:   time.Minute,
	}
	if err := p.Start(); err != nil {
		log.Fatal(err)
	}

	s := server.NewServer()
	s.Plugins.Add(p)

	s.RegisterName("Arith", new(Arith), "")
	s.Serve("tcp", *addr)
}
```

在 `ZooKeeperRegisterPlugin` 中，我们指定了本服务地址，zookeeper 集群地址（可以是多个），起始路径等。服务器启动时自动向 zookeeper 注册本服务的信息，客户端可直接从 zookeeper 拉取可用的服务列表。

首先启动 zookeeper 服务器，zookeeper 的安装与启动可以参考我的上一篇文章。分别在 3 个控制台中启动 3 个服务器，指定不同的端口（注意需要指定 `-tags zookeeper` ）：

```
// 控制台1
$ go run -tags zookeeper main.go -addr 127.0.0.1:8971
// 控制台2
$ go run -tags zookeeper main.go -addr 127.0.0.1:8972
// 控制台3
$ go run -tags zookeeper main.go -addr 127.0.0.1:8973
```

启动之后，我们观察 zookeeper 路径 `/services/math` 中的内容：

```
[zk: localhost:2181(CONNECTED) 0] ls -w -R /services/math
/services/math
/services/math/Arith
/services/math/Arith/tcp@127.0.0.1:8971
/services/math/Arith/tcp@127.0.0.1:8972
/services/math/Arith/tcp@127.0.0.1:8973
[zk: localhost:2181(CONNECTED) 1]
```

非常棒，可用的服务地址不用我们手动维护了！

接下来是客户端：

```
var (
    zkAddr   = flag.String("zkAddr", "127.0.0.1:2181", "zookeeper address")
    basePath = flag.String("basePath", "/services/math", "service base path")
)

func main() {
    flag.Parse()

    d := client.NewZookeeperDiscovery(*basePath, "Arith", []string{*zkAddr}, nil)
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d,
client.DefaultOption)
    defer xclient.Close()

    args := &Args{A: 10, B: 20}
    var reply int

    err := xclient.Call(context.Background(), "Mul", args, &reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    fmt.Printf("%d * %d = %d\n", args.A, args.B, reply)

    args = &Args{50, 20}
    var quo Quotient
    err = xclient.Call(context.Background(), "Div", args, &quo)
```

```
  if err != nil {
    log.Fatalf("failed to call: %v", err)
  }

  fmt.Printf("%d * %d = %d...%d\n", args.A, args.B, quo.Quo, quo.Rem)
}
```

我们通过 zookeeper 读取可用的 `Arith` 服务列表，然后随机选择一个服务发送请求：

```
$ go run -tags zookeeper main.go
2020/05/26 23:03:40 Connected to 127.0.0.1:2181
2020/05/26 23:03:40 authenticated: id=72057658440744975, timeout=10000
2020/05/26 23:03:40 re-submitting `0` credentials after reconnect
10 * 20 = 200
50 * 20 = 2...10
```

我们的客户端发送了两条请求。由于使用了 `client.RandomSelect` 策略，所以这两个请求随机发送到某个服务端。我在 `Mul` 和 `Div` 方法中增加了一个打印，可以观察一下各个控制台的输出！

如果我们关闭了某个服务器，对应的服务地址会从 zookeeper 中移除。我关闭了服务器 1，zookeeper 服务列表变为：

```
[zk: localhost:2181(CONNECTED) 1] ls -w -R /services/math

WATCHER::

WatchedEvent state:SyncConnected type:NodeDeleted path:/services/math/Arith/tcp@127.0.0.1:8971
/services/math

WATCHER::

WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/services/math/Arith
/services/math/Arith
/services/math/Arith/tcp@127.0.0.1:8972
/services/math/Arith/tcp@127.0.0.1:8973
[zk: localhost:2181(CONNECTED) 2]
```

相比上一篇文章中需要手动维护 zookeeper 的内容，`rpcx` 的自动注册和维护明显要方便太多了！

总结

`rpcx` 是 Go 语言中首屈一指的 rpc 库，功能丰富，性能出众，文档丰富，已经被不少公司和个人采用。本文介绍的只是最基础的功能，`rpcx` 支持各种路由选择策略、分组、限流、身份认证等高级功能，推荐深入学习！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. rpcx GitHub：https://github.com/smallnest/rpcx
2. rpcx 博客：https://blog.rpcx.io/
3. rpcx 官网：https://rpcx.io/
4. rpcx 文档：https://doc.rpcx.io/
5. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

jsonrpc

简介

在上一篇文章中我们介绍了 Go 标准库 `net/rpc` 的用法。在默认情况下，`rpc` 库内部使用 `gob` 格式传输数据。我们仿造 `gob` 的编解码器实现了一个 `json` 格式的。实际上标准库 `net/rpc/jsonrcp` 中已有实现。本文是对上一篇文章的补充。

快速使用

标准库无需安装。

首先是服务端，使用 `net/rpc/jsonrpc` 之后，我们就不用自己去编写 `json` 的编解码器了：

```
package main

import (
  "log"
  "net"
  "net/rpc"
  "net/rpc/jsonrpc"
)

type Args struct {
  A, B int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
  *reply = args.A * args.B
  return nil
}

func main() {
  l, err := net.Listen("tcp", ":1234")
  if err != nil {
    log.Fatal("listen error:", err)
  }

  arith := new(Arith)
```

```
  rpc.Register(arith)

  for {
    conn, err := l.Accept()
    if err != nil {
      log.Fatal("accept error:", err)
    }

    // 注意这一行
    go rpc.ServeCodec(jsonrpc.NewServerCodec(conn))
  }
}
```

直接调用 `jsonrpc.NewServerCodec(conn)` 创建一个服务端的 `codec` 。客户端也是类似的：

```
func main() {
  conn, err := net.Dial("tcp", ":1234")
  if err != nil {
    log.Fatal("dial error:", err)
  }

  // 这里，这里😁
  client := rpc.NewClientWithCodec(jsonrpc.NewClientCodec(conn))

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

先运行服务端程序：

```
$ go run main.go
```

然后在一个新的控制台中运行客户端程序：

```
$ go run client.go
Multiply: 7*8=56
```

下面这段代码基本上每个使用 `jsonrpc` 的程序都要编写：

```
conn, err := net.Dial("tcp", ":1234")
if err != nil {
  log.Fatal("dial error:", err)
}

client := rpc.NewClientWithCodec(jsonrpc.NewClientCodec(conn))
```

因此 `jsonrpc` 为了方便直接提供了一个 `Dial` 方法。使用 `Dial` 简化上面的客户端程序：

```
func main() {
  client, err := jsonrpc.Dial("tcp", ":1234")
  if err != nil {
    log.Fatal("dial error:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

效果是一样的。

JSON-RPC 标准

JSON-RPC 1.0 标准在 2005 年发布，经过数年演化，于 2010 年发布了 2.0 版本。JSON-RPC 标准的内容可在https://www.jsonrpc.org/specification查看。Go 标准库 `net/rpc/jsonrpc` 实现了 1.0 版本。关于 2.0 版本的实现可以在 `pkg.go.dev` 上搜索 `json-rpc+2.0` 。本文以 1.0 版本为基础进行介绍。

JSON-RPC 传输的是单一的对象，序列化为 JSON 格式。请求对象包含以下 3 个属性：

- `method` ：请求调用的方法；
- `params` ：一个数组表示传给方法的各个参数；
- `id` ：请求 ID。ID 可以是任何类型，在收到响应时根据这个属性判断对应哪个请求。

响应对象包含以下 3 个属性：

- `result` ：方法返回的对象，如果 `error` 非空时，该属性必须为 `null` ；
- `error` ：表示调用是否出错；
- `id` ：对应请求的 ID。

另外标准还定义了一种通知类型，除了 `id` 属性为 `null` 之外，通知对象的属性与请求对象完全一样。

调用 `client.Call("echo", "Hello JSON-RPC", &reply)` 时：

```
请求：{ "method": "echo", "params": ["Hello JSON-RPC"], "id": 1}
响应：{ "result": "Hello JSON-RPC", "error": null, "id": 1}
```

使用 zookeeper 实现简单的负载均衡

下面我们使用 `zookeeper` 实现一个简单的**客户端**侧的负载均衡。 `zookeeper` 中记录所有的可提供服务的服务器，客户端每次请求时都随机挑选一个。**我们的示例中，请求必须是无状态的**。首先，我们改造一下服务端程序，将监听地址提取出来，通过 `flag` 指定：

```
package main

import (
  "flag"
  "log"
  "net"
  "net/rpc"
  "net/rpc/jsonrpc"
)

var (
  addr *string
)

type Args struct {
  A, B int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
  *reply = args.A * args.B
  return nil
}
```

```
func init() {
  addr = flag.String("addr", ":1111", "addr to listen")
}

func main() {
  flag.Parse()

  l, err := net.Listen("tcp", *addr)
  if err != nil {
    log.Fatal("listen error:", err)
  }

  arith := new(Arith)
  rpc.Register(arith)

  for {
    conn, err := l.Accept()
    if err != nil {
      log.Fatal("accept error:", err)
    }

    go rpc.ServeCodec(jsonrpc.NewServerCodec(conn))
  }
}
```

关于有哪些服务器可用，我们存储在 `zookeeper` 中。

首先要启动一个 `zookeeper` 的程序。在 Apache Zookeeper 官网可以下载能直接运行的 Windows 程序。下载之后解压，将 `conf` 文件夹中的样板配置 `zoo_sample.cfg` 复制一份，文件名改为 `zoo.cfg` 。在编辑器中打开 `zoo.cfg` ，将 `dataDir` 改为一个已存在的目录，或创建一个新目录。我在 `bin` 同级目录中创建了一个 `data` 目录，然后设置 `dataDir=../data` 。切换到 `bin` 目录下执行 `zkServer.bat` ， `zookeeper` 程序就运行起来了。使用 `zkClient.bat` 连接上这个 `zookeeper` ，增加一个节点，设置数据：

```
$ create /rpcserver
$ set /rpcserver 127.0.0.1:1111,127.0.0.1:1112,127.0.0.1:1113
```

我们用 `,` 分隔多个服务器地址。

准备工作完成后，接下来就开始编写客户端代码了。我们实现一个代理类，负责监听 `zookeeper` 的数据变化，根据 `zookeeper` 中新的地址创建到服务器的连接，删除老的连接，将调用请求随机转发到一个服务器处理：

```
type Proxy struct {
  zookeeper     string
  clients       map[string]*rpc.Client
  events        <-chan zk.Event
  zookeeperConn *zk.Conn
  mutex         sync.Mutex
}

func NewProxy(addr string) *Proxy {
  return &Proxy{
    zookeeper: addr,
    clients:   make(map[string]*rpc.Client),
  }
}
```

这里我们使用了 `go-zookeeper` 这个库，需要额外安装：

```
$ go get github.com/samuel/go-zookeeper/zk
```

程序启动时，代理对象从 `zookeeper` 中获取服务端地址，创建连接：

```
func (p *Proxy) Connect() {
  c, _, err := zk.Connect([]string{p.zookeeper}, time.Second) //*10)
  if err != nil {
    panic(err)
  }

  data, _, event, err := c.GetW("/rpcserver")
  if err != nil {
    panic(err)
  }

  p.events = event
  p.zookeeperConn = c

  p.CreateClients(string(data))
}

func (p *Proxy) CreateClients(server string) {
  p.mutex.Lock()
  defer p.mutex.Unlock()

  addrs := strings.Split(server, ",")
  allAddr := make(map[string]struct{})
```

```
  for _, addr := range addrs {
    allAddr[addr] = struct{}{}
    if _, exist := p.clients[addr]; exist {
      continue
    }

    client, err := jsonrpc.Dial("tcp", addr)
    if err != nil {
      log.Println("jsonrpc Dial error:", err)
      continue
    }

    p.clients[addr] = client
    log.Println("new addr:", addr)
  }

  for addr := range p.clients {
    if _, exist := allAddr[addr]; !exist {
      // 不在 zookeeper 中的地址，删除对应连接
      oldClient.Close()
      delete(p.clients, addr)

      log.Println("delete addr", addr)
    }
  }
}
```

同时，需要监听 `zookeeper` 中的数据变化，当新增或删除某个服务端地址时， `Proxy` 要及时更新连接：

```
func (p *Proxy) Run() {
  for {
    select {
    case event := <-p.events:
      if event.Type == zk.EventNodeDataChanged {
        data, _, err := p.zookeeperConn.Get("/rpcserver")
        if err != nil {
          log.Println("get zookeeper data failed:", err)
          continue
        }

        p.CreateClients(string(data))
      }
    }
```

```
    }
}
```

客户端主体程序使用 `Proxy` 结构非常方便：

```
package main

import (
    "flag"
    "fmt"
    "math/rand"
)

var (
    zookeeperAddr *string
)

func init() {
    zookeeperAddr = flag.String("addr", ":2181", "zookeeper address")
}

type Args struct {
    A, B int
}

func main() {
    flag.Parse()

    fmt.Println(*zookeeperAddr)
    p := NewProxy(*zookeeperAddr)
    p.Connect()

    go p.Run()

    for i := 0; i < 10; i++ {
        var reply int
        args := &Args{rand.Intn(1000), rand.Intn(1000)}
        p.Call("Arith.Multiply", args, &reply)
        fmt.Printf("%d*%d=%d\n", args.A, args.B, reply)
    }

    // sleep 过程中可以修改 zookeeper 中的数据
    time.Sleep(1 * time.Minute)

    // 使用新的地址做随机
```

```
    for i := 0; i < 100; i++ {
        var reply int
        args := &Args{rand.Intn(1000), rand.Intn(1000)}
        p.Call("Arith.Multiply", args, &reply)
        fmt.Printf("%d*%d=%d\n", args.A, args.B, reply)
    }
}
```

创建一个代理对象，在一个新的 goroutine 中监听 `zookeeper` 事件。然后通过 `Proxy` 的 `Call` 调用远程服务端的方法：

```
func (p *Proxy) Call(method string, args interface{}, reply interface{}) error {
    var client *rpc.Client
    var addr string
    idx := rand.Int31n(int32(len(p.clients)))
    var i int32
    p.mutex.Lock()
    for a, c := range p.clients {
        if i == idx {
            client = c
            addr = a
            break
        }
        i++
    }
    p.mutex.Unlock()

    fmt.Println("use", addr)
    return client.Call(method, args, reply)
}
```

首先我们要启动 3 个服务端程序，分别监听端口 1111、1112、1113，需要 3 个控制台：

控制台 1：

```
$ go run main.go -addr :1111
```

控制台 2：

```
$ go run main.go -addr :1112
```

控制台 3：

```
$ go run main.go -addr :1113
```

客户端在一个新的控制台启动，指定 `zookeeper` 地址：

```
$ go run . -addr=127.0.0.1:2181
```

在输出中，我们可以看到是怎么随机挑选服务器的。

我们可以尝试在客户端程序运行的过程中，将某个服务器地址从 `zookeeper` 中删除。我特意在程序中加了一个 1 分钟的延迟。在 `sleep` 过程中，通过 `zkClient.cmd` 将 `127.0.0.1:1113` 这个地址从 `zookeeper` 中删除：

```
$ set /rpcserver 127.0.0.1:1111,127.0.0.1:1112
```

控制台输出:

```
$ 2020/05/10 23:47:47 delete addr 127.0.0.1:1113
```

并且后续的请求不会再发到 `127.0.0.1:1113` 这个服务器了。

其实，在实际的项目中， `Proxy` 一般是一个独立的服务器，而不是放在客户端侧。上面示例这样处理只是为了方便。

总结

RPC 底层可以使用各种协议传输数据，JSON/XML/Protobuf 都可以。对 rpc 感兴趣的建议看看 `rpcx` 这个库，https://github.com/smallnest/rpcx。非常强大！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. jsonrpc GitHub：https://golang.org/pkg/net/rpc/jsonrpc/
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

rpc

简介

RPC（Remote Procedure Call）是远程方法调用的缩写，它可以通过网络调用远程对象的方法。Go 标准库 `net/rpc` 提供了一个**简单、强大且高性能**的 RPC 实现。仅需编写很少的代码就能实现 RPC 服务。本文就来介绍一下这个库。

快速使用

标准库无需安装。

由于是网络程序，我们需要编写服务端和客户端两个程序。首先是服务端程序：

```
package main

import (
  "errors"
  "log"
  "net"
  "net/http"
  "net/rpc"
)

type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
  *reply = args.A * args.B
  return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
  if args.B == 0 {
    return errors.New("divide by 0")
```

```
    }

    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {
    arith := new(Arith)
    rpc.Register(arith)
    rpc.HandleHTTP()
    if err := http.ListenAndServe(":1234", nil); err != nil {
        log.Fatal("serve error:", err)
    }
}
```

我们定义了一个 `Arith` 类型，为它编写了两个方法 `Multiply` 和 `Divide` 。创建 `Arith` 类型的对象 `arith` ，调用 `rpc.Register(arith)` 会注册这两个方法。 `rpc` 库对注册的方法有一定的限制，方法必须满足签名 `func (t *T) MethodName(argType T1, replyType *T2) error` ：

- 首先，方法必须是导出的（名字首字母大写）；
- 其次，方法接受两个参数，必须是导出的或内置类型。第一个参数表示客户端传递过来的请求参数，第二个是需要返回给客户端的响应。第二个参数必须为指针类型（需要修改）；
- 最后，方法必须返回一个 `error` 类型的值。返回非 `nil` 的值，表示调用出错。

`rpc.HandleHTTP()` 注册 HTTP 路由。 `http.ListenAndServe(":1234", nil)` 在端口 `1234` 上启动一个 HTTP 服务，请求 rpc 方法会交给 `rpc` 内部路由处理。这样我们就可以通过客户端调用这两个方法了：

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
)

type Args struct {
    A, B int
}
```

```
type Quotient struct {
  Quo, Rem int
}

func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)

  args = &Args{15, 6}
  var quo Quotient
  err = client.Call("Arith.Divide", args, &quo)
  if err != nil {
    log.Fatal("Divide error:", err)
  }
  fmt.Printf("Divide: %d/%d=%d...%d\n", args.A, args.B, quo.Quo, quo.Rem)
}
```

客户端比服务端稍微简单一点，我们使用 `rpc.DialHTTP("tcp", ":1234")` 连接到服务端的监听地址，返回一个 rpc 的客户端对象。后续就可以调用该对象的 `Call()` 方法调用服务端对象的对应方法，依次传入方法名（需要加上类型限定）、参数、一个指针（用于接收返回值）。首先运行服务端程序：

```
$ go run main.go
```

然后在一个新的控制台中运行客户端程序，输出：

```
$ go run client.go
Multiply: 7*8=56
Divide: 15/6=2...3
```

对 `net/http` 包不熟悉的童鞋可能会觉得奇怪，`rpc.HandleHTTP()` 与 `http.ListenAndServer(":1234", nil)` 是怎么联系起来的？我们简单看一下源码：

```go
// src/net/rpc/server.go
const (
  // Defaults used by HandleHTTP
  DefaultRPCPath   = "/_goRPC_"
  DefaultDebugPath = "/debug/rpc"
)

func (server *Server) HandleHTTP(rpcPath, debugPath string) {
  http.Handle(rpcPath, server)
  http.Handle(debugPath, debugHTTP{server})
}

func HandleHTTP() {
  DefaultServer.HandleHTTP(DefaultRPCPath, DefaultDebugPath)
}
```

实际上， `rpc.HandleHTTP()` 会调用 `http.Handle()` 在预定义的路径上（ `/_goRPC_` ）注册处理器。这个处理器最终被添加到 `net/http` 包中的默认多路复用器上：

```go
// src/net/http/server.go
func Handle(pattern string, handler Handler) {
  DefaultServeMux.Handle(pattern, handler)
}
```

而 `http.ListenAndServer()` 第二个参数传入 `nil` 时也是使用默认的多路复用器。具体可以看看我之前的文章Go Web 编程之 程序结构。

细心的朋友可能发现了，除了默认的路径 `/_goRPC_` 用来处理 RPC 请求， `rpc.HandleHTTP()` 方法还注册了一个调试路径 `/debug/rpc` 。我们可以直接在浏览器中访问这个网址（需要服务端程序开启。如果服务端在远程，需要相应地修改地址）localhost:1234，直观的查看各个方法的调用情况：

Services

localhost:1234/debug/rpc

Service Arith

Method	Calls
Divide(*main.Args, *main.Quotient) error	1
Multiply(*main.Args, *int) error	2

异步调用

上面的例子中，我们在客户端使用了同步的调用方式，即一直等待服务端的响应或出错。在等待的过程中，客户端就不能处理其它的任务了。当然，我们也可以采用异步的调用方式：

```
func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args1 := &Args{7, 8}
  var reply int
  multiplyReply := client.Go("Arith.Multiply", args1, &reply, nil)

  args2 := &Args{15, 6}
  var quo Quotient
  divideReply := client.Go("Arith.Divide", args2, &quo, nil)

  ticker := time.NewTicker(time.Millisecond)
  defer ticker.Stop()

  var multiplyReplied, divideReplied bool
  for !multiplyReplied || !divideReplied {
    select {
    case replyCall := <-multiplyReply.Done:
      if err := replyCall.Error; err != nil {
        fmt.Println("Multiply error:", err)
      } else {
        fmt.Printf("Multiply: %d*%d=%d\n", args1.A, args1.B, reply)
      }
      multiplyReplied = true
    case replyCall := <-divideReply.Done:
      if err := replyCall.Error; err != nil {
        fmt.Println("Divide error:", err)
      } else {
        fmt.Printf("Divide: %d/%d=%d...%d\n", args2.A, args2.B, quo.Quo, quo.Re
m)
      }
      divideReplied = true
    case <-ticker.C:
      fmt.Println("tick")
    }
  }
}
```

异步调用使用 `client.Go()` 方法，参数与同步调用基本一样。它返回一个 `rpc.Call` 对象：

```
// src/net/rpc/client.go
type Call struct {
  ServiceMethod string
  Args          interface{}
  Reply         interface{}
  Error         error
  Done          chan *Call
}
```

我们可以通过该对象获取此次调用的信息，如方法名、参数、返回值和错误。我们通过监听通道 `Done` 是否有值判断调用是否完成。上面代码中使用一个 `select` 语句轮询两次调用的状态。注意一点，**如果多个通道都有值， `select` 执行哪个 `case` 是随机的**。所以可能先输出 `divide` 的信息：

```
$ go run client.go
Divide: 15/6=2...3
Multiply: 7*8=56
```

服务端可以继续使用一开始的。

定制方法名

默认情况下， `rpc.Register()` 将方法接收者（ `receiver` ）的类型名作为方法名前缀。我们也可以自己设置。这时需要调用 `RegisterName(name string, rcvr interface{}) error` 方法：

```
func main() {
  arith := new(Arith)
  rpc.RegisterName("math", arith)
  rpc.HandleHTTP()
  if err := http.ListenAndServe(":1234", nil); err != nil {
    log.Fatal("serve error:", err)
  }
}
```

上面我们将注册的方法名前缀改为 `math` 了，客户端调用时传入的方法名也需要相应的修改：

```
func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("math.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

TCP

上面我们都是使用 HTTP 协议来实现 rpc 服务的， `rpc` 库也支持直接使用 TCP 协议。首先，服务端先调用 `net.Listen("tcp", ":1234")` 创建一个监听某个 TCP 端口的监听器（Accepter），然后使用 `rpc.Accept(l)` 在此监听器上接受连接并处理：

```
func main() {
  l, err := net.Listen("tcp", ":1234")
  if err != nil {
    log.Fatal("listen error:", err)
  }

  arith := new(Arith)
  rpc.Register(arith)
  rpc.Accept(l)
}
```

然后，客户端调用 `rpc.Dial()` 以 TCP 协议连接到服务端：

```
func main() {
  client, err := rpc.Dial("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
```

```
    if err != nil {
        log.Fatal("Multiply error:", err)
    }
    fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

自己接收连接

我们可以自己接受连接，然后在此连接上应用 rpc 协议：

```
func main() {
    l, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("listen error:", err)
    }

    arith := new(Arith)
    rpc.Register(arith)

    for {
        conn, err := l.Accept()
        if err != nil {
            log.Fatal("accept error:", err)
        }

        go rpc.ServeConn(conn)
    }
}
```

这个客户端与上面 TCP 的客户端一样，不用修改。

自定义编码格式

默认客户端与服务端之间的数据使用 `gob` 编码，我们可以使用其它的格式来编码。在服务端，我们要实现 `rpc.ServerCodec` 接口：

```
// src/net/rpc/server.go
type ServerCodec interface {
    ReadRequestHeader(*Request) error
    ReadRequestBody(interface{}) error
    WriteResponse(*Response, interface{}) error
```

```
  Close() error
}
```

实际上不用这么麻烦，我们查看源码看看 `gobServerCodec` 是怎么实现的，然后仿造实现一个就行了。下面我实现了一个 JSON 格式的编解码器：

```
type JsonServerCodec struct {
  rwc    io.ReadWriteCloser
  dec    *json.Decoder
  enc    *json.Encoder
  encBuf *bufio.Writer
  closed bool
}

func NewJsonServerCodec(conn io.ReadWriteCloser) *JsonServerCodec {
  buf := bufio.NewWriter(conn)
  return &JsonServerCodec{conn, json.NewDecoder(conn), json.NewEncoder(buf), bu
f, false}
}

func (c *JsonServerCodec) ReadRequestHeader(r *rpc.Request) error {
  return c.dec.Decode(r)
}

func (c *JsonServerCodec) ReadRequestBody(body interface{}) error {
  return c.dec.Decode(body)
}

func (c *JsonServerCodec) WriteResponse(r *rpc.Response, body interface{}) (err
 error) {
  if err = c.enc.Encode(r); err != nil {
    if c.encBuf.Flush() == nil {
      log.Println("rpc: json error encoding response:", err)
      c.Close()
    }
    return
  }
  if err = c.enc.Encode(body); err != nil {
    if c.encBuf.Flush() == nil {
      log.Println("rpc: json error encoding body:", err)
      c.Close()
    }
    return
  }
  return c.encBuf.Flush()
```

```
}

func (c *JsonServerCodec) Close() error {
  if c.closed {
    return nil
  }
  c.closed = true
  return c.rwc.Close()
}

func main() {
  l, err := net.Listen("tcp", ":1234")
  if err != nil {
    log.Fatal("listen error:", err)
  }

  arith := new(Arith)
  rpc.Register(arith)

  for {
    conn, err := l.Accept()
    if err != nil {
      log.Fatal("accept error:", err)
    }

    go rpc.ServeCodec(NewJsonServerCodec(conn))
  }
}
```

在 `for` 循环中需要创建编解码器 `JsonServerCodec` 传给 `ServeCodec` 方法。同样的，客户端要实现 `rpc.ClientCodec` 接口，也是仿造 `gobClientCodec` 的实现：

```
type JsonClientCodec struct {
  rwc    io.ReadWriteCloser
  dec    *json.Decoder
  enc    *json.Encoder
  encBuf *bufio.Writer
}

func NewJsonClientCodec(conn io.ReadWriteCloser) *JsonClientCodec {
  encBuf := bufio.NewWriter(conn)
  return &JsonClientCodec{conn, json.NewDecoder(conn), json.NewEncoder(encBuf),
 encBuf}
}
```

```
func (c *JsonClientCodec) WriteRequest(r *rpc.Request, body interface{}) (err error) {
	if err = c.enc.Encode(r); err != nil {
		return
	}
	if err = c.enc.Encode(body); err != nil {
		return
	}
	return c.encBuf.Flush()
}

func (c *JsonClientCodec) ReadResponseHeader(r *rpc.Response) error {
	return c.dec.Decode(r)
}

func (c *JsonClientCodec) ReadResponseBody(body interface{}) error {
	return c.dec.Decode(body)
}

func (c *JsonClientCodec) Close() error {
	return c.rwc.Close()
}

func main() {
	conn, err := net.Dial("tcp", ":1234")
	if err != nil {
		log.Fatal("dial error:", err)
	}

	client := rpc.NewClientWithCodec(NewJsonClientCodec(conn))

	args := &Args{7, 8}
	var reply int
	err = client.Call("Arith.Multiply", args, &reply)
	if err != nil {
		log.Fatal("Multiply error:", err)
	}
	fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

要使用 `NewClientWithCodec` 以指定的编解码器创建客户端。

自定义服务器

实际上，上面我们调用的方法 `rpc.Register`，`rpc.RegisterName`，`rpc.ServeConn`，`rpc.ServeCodec` 都是转而去调用默认 `DefaultServer` 的相关方法：

```go
// src/net/rpc/server.go
var DefaultServer = NewServer()

func Register(rcvr interface{}) error { return DefaultServer.Register(rcvr) }

func RegisterName(name string, rcvr interface{}) error {
  return DefaultServer.RegisterName(name, rcvr)
}

func ServeConn(conn io.ReadWriteCloser) {
  DefaultServer.ServeConn(conn)
}

func ServeCodec(codec ServerCodec) {
  DefaultServer.ServeCodec(codec)
}
```

但是因为 `DefaultServer` 是全局共享的，如果有第三方库使用了相关方法，并且注册了一些对象的方法，我们引用这个第三方库之后，就出现两个问题。第一，可能与我们注册的方法冲突；第二，带来额外的安全隐患（库中方法直接 `panic` ？）。故而推荐做法是自己 `NewServer` ：

```go
func main() {
  arith := new(Arith)
  server := rpc.NewServer()
  server.RegisterName("math", arith)
  server.HandleHTTP(rpc.DefaultRPCPath, rpc.DefaultDebugPath)

  if err := http.ListenAndServe(":1234", nil); err != nil {
    log.Fatal("serve error:", err)
  }
}
```

这其实是一个套路，很多库会提供一个默认的实现直接使用，如 `log` 、 `net/http` 这些库。但是也提供了创建和自定义的方法。一般测试时为了方便可以使用默认实现，实践中最好自己创建相应的对象，避免干扰和安全问题。

总结

本文介绍了 Go 标准库中的 `rpc`，它使用非常简单，性能异常强大。很多 `rpc` 的第三方库都是对 `rpc` 的封装，早期版本的 `rpcx` 就是基于 `rpc` 做的封装。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. rpc 官方文档https://golang.org/pkg/net/rpc/
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

xorm

简介

Go 标准库提供的数据库接口 `database/sql` 比较底层，使用它来操作数据库非常繁琐，而且容易出错。因而社区开源了不少第三方库，如上一篇文章中的 `sqlc` 工具，还有各式各样的 ORM （Object Relational Mapping，对象关系映射库），如 `gorm` 和 `xorm` 。本文介绍 `xorm` 。 `xorm` 是一个简单但强大的 Go 语言 ORM 库，使用它可以大大简化我们的数据库操作。

快速使用

先安装：

```
$ go get xorm.io/xorm
```

由于需要操作具体的数据库（本文中我们使用 MySQL），需要安装对应的驱动：

```
$ go get github.com/go-sql-driver/mysql
```

使用：

```
package main

import (
  "log"
  "time"

  _ "github.com/go-sql-driver/mysql"
  "xorm.io/xorm"
)

type User struct {
  Id      int64
  Name    string
  Salt    string
  Age     int
  Passwd  string    `xorm:"varchar(200)"`
  Created time.Time `xorm:"created"`
  Updated time.Time `xorm:"updated"`
```

```
}

func main() {
  engine, err := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

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

  err = engine.Sync2(new(User))
  if err != nil {
    log.Fatal(err)
  }
}
```

使用 `xorm` 来操作数据库，首先需要使用 `xorm.NewEngine()` 创建一个引擎。该方法的参数与 `sql.Open()` 参数相同。

上面代码中，我们演示了 `xorm` 的一个非常实用的功能，将数据库中的表与对应 Go 代码中的结构体做同步。初始状态下，数据库 `test` 中没有表 `user`，调用 `Sync2()` 方法会根据 `User` 的结构自动创建一个 `user` 表。执行后，通过 `describe user` 查看表结构：

```
mysql> describe user;
+---------+--------------+------+-----+---------+----------------+
| Field   | Type         | Null | Key | Default | Extra          |
+---------+--------------+------+-----+---------+----------------+
| id      | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| name    | varchar(255) | YES  |     | NULL    |                |
| salt    | varchar(255) | YES  |     | NULL    |                |
| age     | int(11)      | YES  |     | NULL    |                |
| passwd  | varchar(200) | YES  |     | NULL    |                |
| created | datetime     | YES  |     | NULL    |                |
| updated | datetime     | YES  |     | NULL    |                |
+---------+--------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

如果表 `user` 已经存在，`Sync()` 方法会对比 `User` 结构与表结构的不同，对表做相应的修改。我们给 `User` 结构添加一个 `Level` 字段：

```
type User struct {
  Id      int64
  Name    string
  Salt    string
  Age     int
  Level   int
  Passwd  string    `xorm:"varchar(200)"`
  Created time.Time `xorm:"created"`
```

```
  Updated time.Time `xorm:"updated"`
}
```

再次执行这个程序后，用 `describe user` 命令查看表结构：

```
mysql> describe user;
+---------+--------------+------+-----+---------+----------------+
| Field   | Type         | Null | Key | Default | Extra          |
+---------+--------------+------+-----+---------+----------------+
| id      | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| name    | varchar(255) | YES  |     | NULL    |                |
| salt    | varchar(255) | YES  |     | NULL    |                |
| age     | int(11)      | YES  |     | NULL    |                |
| passwd  | varchar(200) | YES  |     | NULL    |                |
| created | datetime     | YES  |     | NULL    |                |
| updated | datetime     | YES  |     | NULL    |                |
| level   | int(11)      | YES  |     | NULL    |                |
+---------+--------------+------+-----+---------+----------------+
8 rows in set (0.00 sec)
```

发现表中多了一个 `level` 字段。

此修改只限于添加字段。删除表中已有的字段会带来比较大的风险。如果我们 `User` 结构的 `Salt` 字段删除，然后执行程序。出现下面错误：

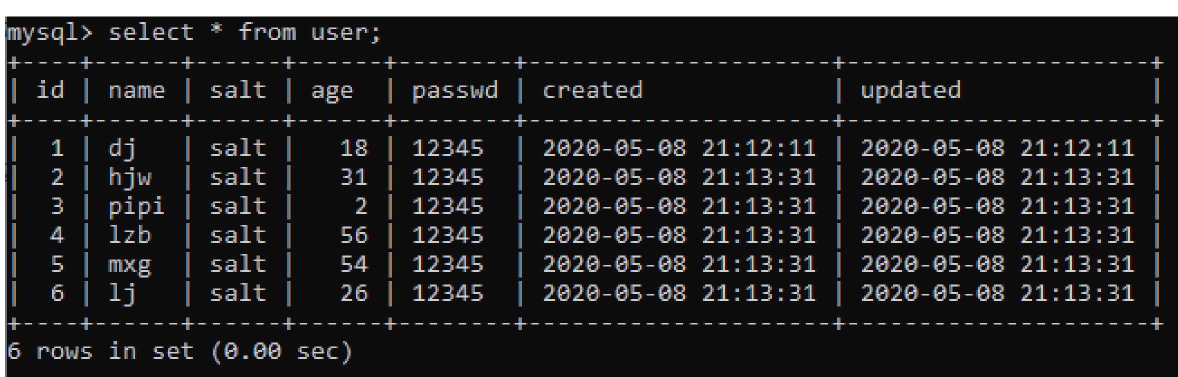
```
[xorm] [warn]  2020/05/07 22:44:38.528784 Table user has column salt but struct has not related field
```

数据库操作

查询&统计

`xorm` 提供了几个查询和统计方法， `Get/Exist/Find/Iterate/Count/Rows/Sum` 。下面逐一介绍。

为了代码演示方便，我在 `user` 表中插入了一些数据：



```
mysql> select * from user;
+----+------+------+-----+--------+---------------------+---------------------+
| id | name | salt | age | passwd | created             | updated             |
+----+------+------+-----+--------+---------------------+---------------------+
|  1 | dj   | salt |  18 | 12345  | 2020-05-08 21:12:11 | 2020-05-08 21:12:11 |
|  2 | hjw  | salt |  31 | 12345  | 2020-05-08 21:13:31 | 2020-05-08 21:13:31 |
|  3 | pipi | salt |   2 | 12345  | 2020-05-08 21:13:31 | 2020-05-08 21:13:31 |
|  4 | lzb  | salt |  56 | 12345  | 2020-05-08 21:13:31 | 2020-05-08 21:13:31 |
|  5 | mxg  | salt |  54 | 12345  | 2020-05-08 21:13:31 | 2020-05-08 21:13:31 |
|  6 | lj   | salt |  26 | 12345  | 2020-05-08 21:13:31 | 2020-05-08 21:13:31 |
+----+------+------+-----+--------+---------------------+---------------------+
6 rows in set (0.00 sec)
```

后面的代码为了简单起见，忽略了错误处理，实际使用中不要漏掉！

Get

`Get()` 方法用于查询单条数据，并使用返回的字段为传入的对象赋值：

```
type User struct {
  Id      int64
  Name    string
  Salt    string
  Age     int
  Passwd  string    `xorm:"varchar(200)"`
  Created time.Time `xorm:"created"`
  Updated time.Time `xorm:"updated"`
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  user1 := &User{}
  has, _ := engine.ID(1).Get(user1)
  if has {
    fmt.Printf("user1:%v\n", user1)
  }

  user2 := &User{}
  has, _ = engine.Where("name=?", "dj").Get(user2)
  if has {
    fmt.Printf("user2:%v\n", user2)
  }

  user3 := &User{Id: 5}
  has, _ = engine.Get(user3)
  if has {
    fmt.Printf("user3:%v\n", user3)
  }

  user4 := &User{Name: "pipi"}
  has, _ = engine.Get(user4)
  if has {
    fmt.Printf("user4:%v\n", user4)
  }
}
```

上面演示了 3 种使用 `Get()` 的方式：

- 使用主键：`engine.ID(1)` 查询主键（即 `id` ）为 1 的用户；

- 使用条件语句：`engine.Where("name=?", "dj")` 查询 `name = "dj"` 的用户；
- 使用对象中的非空字段：`user3` 设置了 `Id` 字段为 5，`engine.Get(user3)` 查询 `id = 5` 的用户；`user4` 设置了字段 `Name` 为 `"pipi"`，`engine.Get(user4)` 查询 `name = "pipi"` 的用户。

运行程序：

```
user1:&{1 dj salt 18 12345 2020-05-08 21:12:11 +0800 CST 2020-05-08 21:12:11 +0800 CST}
user2:&{1 dj salt 18 12345 2020-05-08 21:12:11 +0800 CST 2020-05-08 21:12:11 +0800 CST}
user3:&{5 mxg salt 54 12345 2020-05-08 21:13:31 +0800 CST 2020-05-08 21:13:31 +0800 CST}
user4:&{3 pipi salt 2 12345 2020-05-08 21:13:31 +0800 CST 2020-05-08 21:13:31 +0800 CST}
```

查询条件的使用不区分调用顺序，但是必须在 `Get()` 方法之前调用。实际上后面介绍的查询&统计方法也是如此，可以在调用实际的方法前添加一些过滤条件。除此之外 `xorm` 支持只返回指定的列（`xorm.Cols()`）或忽略特定的列（`xorm.Omit()`）：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  user1 := &User{}
  engine.ID(1).Cols("id", "name", "age").Get(user1)
  fmt.Printf("user1:%v\n", user1)

  user2 := &User{Name: "pipi"}
  engine.Omit("created", "updated").Get(user2)
  fmt.Printf("user2:%v\n", user2)
}
```

上面第一个查询使用 `Cols()` 方法指定只返回 `id`、`name`、`age` 这 3 列，第二个查询使用 `Omit()` 方法忽略列 `created` 和 `updated`。

另外，为了便于排查可能出现的问题，`xorm` 提供了 `ShowSQL()` 方法设置将执行的 SQL 同时在控制台中输出：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  engine.ShowSQL(true)

  user := &User{}
```

```
    engine.ID(1).Omit("created", "updated").Get(user)
    fmt.Printf("user:%v\n", user)
}
```

运行程序：

```
[xorm] [info]  2020/05/08 21:38:29.349976 [SQL] SELECT `id`, `name`, `salt`, `ag
e`, `passwd` FROM `user` WHERE `id`=? LIMIT 1 [1] - 4.0033ms
user:&{1 dj salt 18 12345 0001-01-01 00:00:00 +0000 UTC 0001-01-01 00:00:00 +000
0 UTC}
```

由输出可以看出，执行的 SQL 语句为：

```
SELECT `id`, `name`, `salt`, `age`, `passwd` FROM `user` WHERE `id`=? LIMIT 1
```

该语句耗时 4.003 ms。在开发中这个方法非常好用！

有时候，调试信息都输出到控制台并不利于我们查询， `xorm` 可以设置日志选项，将日志输出到文件中：

```
func main() {
    engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
    f, err := os.Create("sql.log")
    if err != nil {
        panic(err)
    }

    engine.SetLogger(log.NewSimpleLogger(f))
    engine.Logger().SetLevel(log.LOG_DEBUG)
    engine.ShowSQL(true)

    user := &User{}
    engine.ID(1).Omit("created", "updated").Get(user)
    fmt.Printf("user:%v\n", user)
}
```

这样 `xorm` 就会将调试日志输出到 `sql.log` 文件中。注意 `log.NewSimpleLogger(f)` 是 `xorm` 的子包 `xorm.io/xorm/log` 提供的简单日志封装，而非标准库 `log` 。

`Exist`

`Exist()` 方法查询符合条件的记录是否存在，它的返回与 `Get()` 方法一致，都是 `(bool, error)` 。不同之处在于 `Get()` 会将查询得到的字段赋值给传入的对象。相

比之下 `Exist()` 方法效率要高一些。如果不需要获取数据，只要判断是否存在建议使用 `Exist()` 方法。

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  user1 := &User{}
  has, _ := engine.ID(1).Exist(user1)
  if has {
    fmt.Println("user with id=1 exist")
  } else {
    fmt.Println("user with id=1 not exist")
  }

  user2 := &User{}
  has, _ = engine.Where("name=?", "dj2").Get(user2)
  if has {
    fmt.Println("user with name=dj2 exist")
  } else {
    fmt.Println("user with name=dj2 not exist")
  }
}
```

`Find`

`Get()` 方法只能返回单条记录，其生成的 SQL 语句总是有 `LIMIT 1`。`Find()` 方法返回所有符合条件的记录。`Find()` 需要传入对象切片的指针或 map 的指针：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  slcUsers:= make([]User, 1)
  engine.Where("age > ? and age < ?", 12, 30).Find(&slcUsers)
  fmt.Println("users whose age between [12,30]:", slcUsers)

  mapUsers := make(map[int64]User)
  engine.Where("length(name) = ?", 3).Find(&mapUsers)
  fmt.Println("users whose has name of length 3:", mapUsers)
}
```

`map` 的键为主键，所以如果表为复合主键就不能使用这种方式了。

`Iterate`

与 `Find()` 一样，`Iterate()` 也是找到满足条件的所有记录，只不过传入了一个回调去处理每条记录：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  engine.Where("age > ? and age < ?", 12, 30).Iterate(&User{}, func(i int, bean interface{}) error {
    fmt.Printf("user%d:%v\n", i, bean.(*User))
    return nil
  })
}
```

如果回调返回一个非 `nil` 的错误，后面的记录就不会再处理了。

`Count`

`Count()` 方法统计满足条件的记录数量：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  num, _ := engine.Where("age >= ?", 50).Count(&User{})
  fmt.Printf("there are %d users whose age >= 50", num)
}
```

`Rows`

`Rows()` 方法与 `Iterate()` 类似，不过返回一个 `Rows` 对象由我们自己迭代，更加灵活：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  rows, _ := engine.Where("age > ? and age < ?", 12, 30).Rows(&User{})
  defer rows.Close()

  u := &User{}
  for rows.Next() {
    rows.Scan(u)

    fmt.Println(u)
  }
}
```

`Rows()` 的使用与 `database/sql` 有些类似，但是 `rows.Scan()` 方法可以传入一个对象，比 `database/sql` 更方便。

`Sum`

`xorm` 提供了两组求和的方法：

- `Sum/SumInt`：求某个字段的和，`Sum` 返回 `float64`，`SumInt` 返回 `int64`；
- `Sums/SumsInt`：分别求某些字段的和，`Sums` 返回 `[]float64`，`SumsInt` 返回 `[]int64`。

例如：

```
type Sum struct {
  Id    int64
  Money int32
  Rate  float32
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  engine.Sync2(&Sum{})

  var slice []*Sum
  for i := 0; i < 100; i++ {
    slice = append(slice, &Sum{
      Money: rand.Int31n(10000),
      Rate:  rand.Float32(),
    })
  }
  engine.Insert(&slice)

  totalMoney, _ := engine.SumInt(&Sum{}, "money")
  fmt.Println("total money:", totalMoney)

  totalRate, _ := engine.Sum(&Sum{}, "rate")
  fmt.Println("total rate:", totalRate)

  totals, _ := engine.Sums(&Sum{}, "money", "rate")
  fmt.Printf("total money:%f & total rate:%f", totals[0], totals[1])
}
```

插入

使用 `engine.Insert()` 方法，可以插入单条数据，也可以批量插入多条数据：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  user := &User{Name: "lzy", Age: 50}

  affected, _ := engine.Insert(user)
  fmt.Printf("%d records inserted, user.id:%d\n", affected, user.Id)

  users := make([]*User, 2)
  users[0] = &User{Name: "xhq", Age: 41}
  users[1] = &User{Name: "lhy", Age: 12}

  affected, _ = engine.Insert(&users)
  fmt.Printf("%d records inserted, id1:%d, id2:%d", affected, users[0].Id, users
[1].Id)
}
```

插入单条记录传入一个对象指针，批量插入传入一个切片。需要注意的是，批量插入时，每个对象的 `Id` 字段不会被自动赋值，所以上面最后一行输出 `id1` 和 `id2` 均为 0。另外，一次 `Insert()` 调用可以传入多个参数，可以对应不同的表。

更新

更新通过 `engine.Update()` 实现，可以传入结构指针或 `map[string]interface{}` 。对于传入结构体指针的情况， **`xorm` 只会更新非空的字段**。如果一定要更新空字段，需要使用 `Cols()` 方法显示指定更新的列。使用 `Cols()` 方法指定列后，即使字段为空也会更新：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")
  engine.ID(1).Update(&User{Name: "ldj"})
  engine.ID(1).Cols("name", "age").Update(&User{Name: "dj"})

  engine.Table(&User{}).ID(1).Update(map[string]interface{}{"age": 18})
}
```

由于使用 `map[string]interface{}` 类型的参数， `xorm` 无法推断表名，必须使用 `Table()` 方法指定。第一个 `Update()` 方法只会更新 `name` 字段，其他空字段不更新。第二个 `Update()` 方法会更新 `name` 和 `age` 两个字段， `age` 被更新为 0。

删除

直接调用 `engine.Delete()` 删除符合条件的记录，返回删除的条目数量：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  affected, _ := engine.Where("name = ?", "lzy").Delete(&User{})
  fmt.Printf("%d records deleted", affected)
}
```

创建时间、更新时间、软删除

如果我们为 `time.Time/int/int64` 这些类型的字段设置 `xorm:"created"` 标签，**插入数据时**，该字段会自动更新为当前时间；

如果我们为 `tiem.Time/int/int64` 这些类型的字段设置 `xorm:"updated"` 标签，**插入和更新数据时**，该字段会自动更新为当前时间；

如果我们为 `time.Time` 类型的字段设置了 `xorm:"deleted"` 标签，**删除数据时**，只是设置删除时间，并不真正删除记录。

```
type Player struct {
  Id int64
  Name string
  Age int
  CreatedAt time.Time `xorm:"created"`
  UpdatedAt time.Time `xorm:"updated"`
  DeletedAt time.Time `xorm:"deleted"`
}

func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  engine.Sync2(&Player{})
  engine.Insert(&Player{Name:"dj", Age:18})

  p := &Player{}
  engine.Where("name = ?", "dj").Get(p)
  fmt.Println("after insert:", p)
  time.Sleep(5 * time.Second)

  engine.Table(&Player{}).ID(p.Id).Update(map[string]interface{}{"age":30})

  engine.Where("name = ?", "dj").Get(p)
  fmt.Println("after update:", p)
  time.Sleep(5 * time.Second)
```

```
  engine.ID(p.Id).Delete(&Player{})

  engine.Where("name = ?", "dj").Unscoped().Get(p)
  fmt.Println("after delete:", p)
}
```

输出：

```
after insert: &{1 dj 18 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:19 +0800
 CST 0001-01-01 00:00:00 +0000 UTC}
after update: &{1 dj 30 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:24 +0800
 CST 0001-01-01 00:00:00 +0000 UTC}
after delete: &{1 dj 30 2020-05-08 23:09:19 +0800 CST 2020-05-08 23:09:24 +0800
 CST 2020-05-08 23:09:29 +0800 CST}
```

创建时间一旦创建成功就不会再改变了，更新时间每次更新都会变化。已删除的记录必须使用 `Unscoped()` 方法查询，如果要真正 删除某条记录，也可以使用 `Unscoped()` 。

执行原始的 SQL

除了上面提供的方法外， `xorm` 还可以执行原始的 SQL 语句：

```
func main() {
  engine, _ := xorm.NewEngine("mysql", "root:12345@/test?charset=utf8")

  querySql := "select * from user limit 1"
  reuslts, _ := engine.Query(querySql)
  for _, record := range reuslts {
    for key, val := range record {
      fmt.Println(key, string(val))
    }
  }

  updateSql := "update `user` set name=? where id=?"
  res, _ := engine.Exec(updateSql, "ldj", 1)
  fmt.Println(res.RowsAffected())
}
```

`Query()` 方法返回 `[]map[string][]byte` ，切片中的每个元素都代表一条记录， `map` 的键对应列名， `[]byte` 为值。还有 `QueryInterface()` 方法返回 `[]map[string]interface{}` ， `QueryString()` 方法返回 `[]map[string]interface{}` 。

运行程序：

```
salt salt
age 18
passwd 12345
created 2020-05-08 21:12:11
updated 2020-05-08 22:44:58
id 1
name ldj
1 <nil>
```

总结

本文对 `xorm` 做了一个简单的介绍，`xorm` 的特性远不止于此。`xorm` 可以定义结构体字段与表列名映射规则、创建索引、执行事务、导入导出 SQL 脚本等。感兴趣可自行探索。好在 `xorm` 有比较详尽的中文文档。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. xorm GitHub：https://github.com/go-xorm/xorm
2. xorm 手册：http://gobook.io/read/gitea.com/xorm/manual-zh-CN/
3. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

sqlc

简介

在 Go 语言中编写数据库操作代码真的非常痛苦！ `database/sql` 标准库提供的都是比较底层的接口。我们需要编写大量重复的代码。大量的模板代码不仅写起来烦，而且还容易出错。有时候字段类型修改了一下，可能就需要改动很多地方；添加了一个新字段，之前使用 `select *` 查询语句的地方都要修改。如果有些地方有遗漏，可能就会造成运行时 `panic` 。即使使用 ORM 库，这些问题也不能完全解决！这时候， `sqlc` 来了！ `sqlc` 可以根据我们编写的 SQL 语句生成类型安全的、地道的 Go 接口代码，我们要做的只是调用这些方法。

快速使用

先安装：

```
$ go get github.com/kyleconroy/sqlc/cmd/sqlc
```

当然还有对应的数据库驱动：

```
$ go get github.com/lib/pq
$ go get github.com/go-sql-driver/mysql
```

`sqlc` 是一个命令行工具，上面代码会将可执行程序 `sqlc` 放到 `$GOPATH/bin` 目录下。我习惯把 `$GOPATH/bin` 目录加入到系统 `PATH` 中。所以可以执行使用这个命令。

因为 `sqlc` 用到了一个 linux 下的库，在 windows 上无法正常编译。在 windows 上我们可以使用 docker 镜像 `kjconroy/sqlc` 。docker 的安装就不介绍了，网上有很多教程。拉取 `kjconroy/sqlc` 镜像：

```
$ docker pull kjconroy/sqlc
```

然后，编写 SQL 语句。在 `schema.sql` 文件中编写建表语句：

```
CREATE TABLE authors (
  id   BIGSERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  bio  TEXT
);
```

在 `query.sql` 文件中编写查询语句：

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;

-- name: CreateAuthor :exec
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING *;

-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

`sqlc` 支持 PostgreSQL 和 MySQL，**不过对 MySQL 的支持是实验性的**。期待后续完善对 MySQL 的支持，增加对其它数据库的支持。本文我们使用的是 PostgreSQL。编写数据库程序时，上面两个 sql 文件是少不了的。 `sqlc` 额外只需要一个小小的配置文件 `sqlc.yaml` ：

```
version: "1"
packages:
  - name: "db"
    path: "./db"
    queries: "./query.sql"
    schema: "./schema.sql"
```

- `version` ：版本；
- `packages` ：
    - `name` ：生成的包名；
    - `path` ：生成文件的路径；
    - `queries` ：查询 SQL 文件；
    - `schema` ：建表 SQL 文件。

在 windows 上执行下面的命令生成对应的 Go 代码：

```
docker run --rm -v CONFIG_PATH:/src -w /src kjconroy/sqlc generate
```

上面的 `CONFIG_PATH` 替换成配置所在目录，我的是 `D:\code\golang\src\github.com\darjun\go-daily-lib\sqlc\get-started` 。 `sqlc` 为我们在同级目录下生成了数据库操作代码，目录结构如下：

```
db
├── db.go
├── models.go
└── query.sql.go
```

`sqlc` 根据我们 `schema.sql` 和 `query.sql` 生成了模型对象结构：

```
// models.go
type Author struct {
  ID   int64
  Name string
  Bio  sql.NullString
}
```

和操作接口：

```
// query.sql.go
func (q *Queries) CreateAuthor(ctx context.Context, arg CreateAuthorParams) (Author, error)
func (q *Queries) DeleteAuthor(ctx context.Context, id int64) error
func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error)
func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error)
```

其中 `Queries` 是 `sqlc` 封装的一个结构。

说了这么多，来看看如何使用：

```
package main

import (
  "database/sql"
  "fmt"
  "log"

  _ "github.com/lib/pq"
  "golang.org/x/net/context"
```

```
	"github.com/darjun/go-daily-lib/sqlc/get-started/db"
)

func main() {
	pq, err := sql.Open("postgres", "dbname=sqlc sslmode=disable")
	if err != nil {
		log.Fatal(err)
	}

	queries := db.New(pq)

	authors, err := queries.ListAuthors(context.Background())
	if err != nil {
		log.Fatal("ListAuthors error:", err)
	}
	fmt.Println(authors)

	insertedAuthor, err := queries.CreateAuthor(context.Background(), db.CreateAuthorParams{
		Name: "Brian Kernighan",
		Bio:  sql.NullString{String: "Co-author of The C Programming Language and The Go Programming Language", Valid: true},
	})
	if err != nil {
		log.Fatal("CreateAuthor error:", err)
	}
	fmt.Println(insertedAuthor)

	fetchedAuthor, err := queries.GetAuthor(context.Background(), insertedAuthor.ID)
	if err != nil {
		log.Fatal("GetAuthor error:", err)
	}
	fmt.Println(fetchedAuthor)

	err = queries.DeleteAuthor(context.Background(), insertedAuthor.ID)
	if err != nil {
		log.Fatal("DeleteAuthor error:", err)
	}
}
```

生成的代码在包 `db` 下（由 `packages.name` 选项指定），首先调用 `db.New()` 将 `sql.Open()` 的返回值 `sql.DB` 作为参数传入，得到 `Queries` 对象。我们对 `authors` 表的操作都需要通过该对象的方法。

上面程序要运行，还需要启动 PostgreSQL，创建数据库和表：

```
$ createdb sqlc
$ psql -f schema.sql -d sqlc
```

上面第一条命令创建一个名为 `sqlc` 的数据库，第二条命令在数据库 `sqlc` 中执行 `schema.sql` 文件中的语句，即创建表。

最后运行程序（多文件程序不能用 `go run main.go` ）：

```
$ go run .
[]
{1 Brian Kernighan {Co-author of The C Programming Language and The Go Programming Language true}}
{1 Brian Kernighan {Co-author of The C Programming Language and The Go Programming Language true}}
```

代码生成

除了 SQL 语句本身， `sqlc` 需要我们在编写 SQL 语句的时候通过注释的方式为生成的程序提供一些基本信息。语法为：

```
-- name: <name> <cmd>
```

`name` 为生成的方法名，如上面的 `CreateAuthor/ListAuthors/GetAuthor/DeleteAuthor` 等， `cmd` 可以有以下取值：

- `:one` ：表示 SQL 语句返回一个对象，生成的方法的返回值为 `(对象类型, error)` ，对象类型可以从表名得出；
- `:many` ：表示 SQL 语句会返回多个对象，生成的方法的返回值为 `([]对象类型, error)` ；
- `:exec` ：表示 SQL 语句不返回对象，只返回一个 `error` ；
- `:execrows` ：表示 SQL 语句需要返回受影响的行数。

`:one`

```
-- name: GetAuthor :one
SELECT id, name, bio FROM authors
WHERE id = $1 LIMIT 1
```

注释中 `--name` 指示生成方法 `GetAuthor` ，从表名得出返回的基础类型为 `Author` 。 `:one` 又表示只返回一个对象。故最终的返回值为 `(Author,`

`error)` ：

```
// db/query.sql.go
const getAuthor = `-- name: GetAuthor :one
SELECT id, name, bio FROM authors
WHERE id = $1 LIMIT 1
`

func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error) {
  row := q.db.QueryRowContext(ctx, getAuthor, id)
  var i Author
  err := row.Scan(&i.ID, &i.Name, &i.Bio)
  return i, err
}
```

`:many`

```
-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;
```

注释中 `--name` 指示生成方法 `ListAuthors` ，从表名 `authors` 得到返回的基础类型为 `Author` 。 `:many` 表示返回一个对象的切片。故最终的返回值为 `([]Author, error)` ：

```
// db/query.sql.go
const listAuthors = `-- name: ListAuthors :many
SELECT id, name, bio FROM authors
ORDER BY name
`

func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
  rows, err := q.db.QueryContext(ctx, listAuthors)
  if err != nil {
    return nil, err
  }
  defer rows.Close()
  var items []Author
  for rows.Next() {
    var i Author
    if err := rows.Scan(&i.ID, &i.Name, &i.Bio); err != nil {
      return nil, err
    }
    items = append(items, i)
```

```
  }
  if err := rows.Close(); err != nil {
    return nil, err
  }
  if err := rows.Err(); err != nil {
    return nil, err
  }
  return items, nil
}
```

这里注意一个细节，即使我们使用了 `select *` ，生成的代码中 SQL 语句被也改写成了具体的字段：

```
SELECT id, name, bio FROM authors
ORDER BY name
```

这样后续如果我们需要添加或删除字段，只要执行了 `sqlc` 命令，这个 SQL 语句和 `ListAuthors()` 方法就能保持一致！是不是很方便？

`:exec`

```
-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1
```

注释中 `--name` 指示生成方法 `DeleteAuthor` ，从表名 `authors` 得到返回的基础类型为 `Author` 。 `:exec` 表示不返回对象。故最终的返回值为 `error` ：

```
// db/query.sql.go
const deleteAuthor = `-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1
`

func (q *Queries) DeleteAuthor(ctx context.Context, id int64) error {
  _, err := q.db.ExecContext(ctx, deleteAuthor, id)
  return err
}
```

`:execrows`

```
-- name: DeleteAuthorN :execrows
DELETE FROM authors
```

```
WHERE id = $1
```

注释中 `--name` 指示生成方法 `DeleteAuthorN` ，从表名 `authors` 得到返回的基础类型为 `Author` 。 `:exec` 表示返回受影响的行数（即删除了多少行）。故最终的返回值为 `(int64, error)` ：

```
// db/query.sql.go
const deleteAuthorN = `-- name: DeleteAuthorN :execrows
DELETE FROM authors
WHERE id = $1
`

func (q *Queries) DeleteAuthorN(ctx context.Context, id int64) (int64, error) {
  result, err := q.db.ExecContext(ctx, deleteAuthorN, id)
  if err != nil {
    return 0, err
  }
  return result.RowsAffected()
}
```

不管编写的 SQL 多复杂，总是逃不过上面的规则。我们只需要在编写 SQL 语句时额外添加一行注释， `sqlc` 就能为我们生成地道的 SQL 操作方法。生成的代码与我们自己手写的没什么不同，错误处理都很完善，而且了避免手写的麻烦与错误。

模型对象

`sqlc` 为所有的建表语句生成对应的模型结构。结构名为表名的单数形式，且首字母大写。例如：

```
CREATE TABLE authors (
  id   SERIAL PRIMARY KEY,
  name text   NOT NULL
);
```

生成对应的结构：

```
type Author struct {
  ID   int
  Name string
}
```

而且 `sqlc` 可以解析 `ALTER TABLE` 语句，它会根据最终的表结构来生成模型对象的结构。例如：

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  birth_year  int    NOT NULL
);

ALTER TABLE authors ADD COLUMN bio text NOT NULL;
ALTER TABLE authors DROP COLUMN birth_year;
ALTER TABLE authors RENAME TO writers;
```

上面的 SQL 语句中，建表时有两列 `id` 和 `birth_year` 。第一条 `ALTER TABLE` 语句添加了一列 `bio` ，第二条删除了 `birth_year` 列，第三条将表名 `authors` 改为 `writers` 。 `sqlc` 依据最终的表名 `writers` 和表中的列 `id` 、 `bio` 生成代码：

```
package db

type Writer struct {
  ID  int
  Bio string
}
```

配置字段

`sqlc.yaml` 文件中还可以设置其他的配置字段。

`emit_json_tags`

默认为 `false` ，设置该字段为 `true` 可以为生成的模型对象结构添加 JSON 标签。例如：

```
CREATE TABLE authors (
  id          SERIAL    PRIMARY KEY,
  created_at timestamp NOT NULL
);
```

生成：

```
package db

import (
  "time"
)

```

```
type Author struct {
  ID        int       `json:"id"`
  CreatedAt time.Time `json:"created_at"`
}
```

`emit_prepared_queries`

默认为 `false` ，设置该字段为 `true` ，会为 SQL 生成对应的 `prepared statement` 。例如，在**快速开始**的示例中设置这个选项，最终生成的结构 `Queries` 中会添加所有 SQL 对应的 `prepared statement` 对象：

```
type Queries struct {
  db                DBTX
  tx                *sql.Tx
  createAuthorStmt  *sql.Stmt
  deleteAuthorStmt  *sql.Stmt
  getAuthorStmt     *sql.Stmt
  listAuthorsStmt   *sql.Stmt
}
```

和一个 `Prepare()` 方法：

```
func Prepare(ctx context.Context, db DBTX) (*Queries, error) {
  q := Queries{db: db}
  var err error
  if q.createAuthorStmt, err = db.PrepareContext(ctx, createAuthor); err != nil
 {
    return nil, fmt.Errorf("error preparing query CreateAuthor: %w", err)
  }
  if q.deleteAuthorStmt, err = db.PrepareContext(ctx, deleteAuthor); err != nil
 {
    return nil, fmt.Errorf("error preparing query DeleteAuthor: %w", err)
  }
  if q.getAuthorStmt, err = db.PrepareContext(ctx, getAuthor); err != nil {
    return nil, fmt.Errorf("error preparing query GetAuthor: %w", err)
  }
  if q.listAuthorsStmt, err = db.PrepareContext(ctx, listAuthors); err != nil {
    return nil, fmt.Errorf("error preparing query ListAuthors: %w", err)
  }
  return &q, nil
}
```

生成的其它方法都使用了这些对象，而非直接使用 SQL 语句：

```
func (q *Queries) CreateAuthor(ctx context.Context, arg CreateAuthorParams) (Author, error) {
  row := q.queryRow(ctx, q.createAuthorStmt, createAuthor, arg.Name, arg.Bio)
  var i Author
  err := row.Scan(&i.ID, &i.Name, &i.Bio)
  return i, err
}
```

我们需要在程序初始化时调用这个 `Prepare()` 方法。

`emit_interface`

默认为 `false` ，设置该字段为 `true` ，会为查询结构生成一个接口。例如，在**快速开始**的示例中设置这个选项，最终生成的代码会多出一个文件 `querier.go` ：

```
// db/querier.go
type Querier interface {
  CreateAuthor(ctx context.Context, arg CreateAuthorParams) (Author, error)
  DeleteAuthor(ctx context.Context, id int64) error
  DeleteAuthorN(ctx context.Context, id int64) (int64, error)
  GetAuthor(ctx context.Context, id int64) (Author, error)
  ListAuthors(ctx context.Context) ([]Author, error)
}
```

覆写类型

`sqlc` 在生成模型对象结构时会根据数据库表的字段类型推算出一个 Go 语言类型，例如 `text` 对应 `string` 。我们也可以在配置文件中指定这种类型映射。

```
version: "1"
packages:
  - name: "db"
    path: "./db"
    queries: "./query.sql"
    schema: "./schema.sql"
overrides:
  - go_type: "github.com/uniplaces/carbon.Time"
    db_type: "pg_catalog.timestamp"
```

在 `overrides` 下 `go_type` 表示使用的 Go 类型。如果是非标准类型，必须指定**全限定类型**（即包路径 + 类型名）。 `db_type` 设置为要映射的数据库类型。 `sqlc` 会自动导入对应的标准包或第三方包。生成代码如下：

```
package db

import (
  "github.com/uniplaces/carbon"
)

type Author struct {
  ID       int32
  Name     string
  CreateAt carbon.Time
}
```

需要注意的是 `db_type` 的表示，文档这里一笔带过，使用上还是有些晦涩。我也是看源码才找到如何覆写 `timestamp` 类型的，需要将 `db_type` 设置为 `pg_catalog.timestamp` 。同理 `timestamptz` 、 `timetz` 等类型也需要加上这个前缀。**一般复杂类型都需要加上前缀，一般的基础类型可以加也可以不加。**遇到不确定的情况，可以去看看源码gen.go#L634。

也可以设定某个字段的类型，例如我们要将创建时间字段 `created_at` 设置为使用 `carbon.Time` ：

```
version: "1"
packages:
  - name: "db"
    path: "./db"
    queries: "./query.sql"
    schema: "./schema.sql"
overrides:
  - column: "authors.create_at"
    go_type: "github.com/uniplaces/carbon.Time"
```

生成代码如下：

```
// db/models.go
package db

import (
  "github.com/uniplaces/carbon"
)

type Author struct {
  ID       int32
  Name     string
```

```
  CreateAt carbon.Time
}
```

最后我们还可以给生成的结构字段命名：

```
version: "1"
packages:
  - name: "db"
    path: "./db"
    queries: "./query.sql"
    schema: "./schema.sql"
rename:
    id: "Id"
    name: "UserName"
    create_at: "CreateTime"
```

上面配置为生成的结构设置字段名，生成代码：

```
package db

import (
  "time"
)

type Author struct {
  Id         int32
  UserName   string
  CreateTime time.Time
}
```

安装 PostgreSQL

我之前使用 MySQL 较多。由于 `sqlc` 对 MySQL 的支持不太好，在体验这个库的时候还是选择支持较好的 PostgreSQL。不得不说，在 win10 上，PostgreSQL 的**安装门槛**实在是太高了！我摸索了很久最后只能在https://www.enterprisedb.com/download-postgresql-binaries下载可执行文件。我选择了 10.12 版本，下载、解压、将文件夹中的 `bin` 加入系统 `PATH` 。创建一个 `data` 目录，然后执行下面的命令初始化数据：

```
$ initdb data
```

注册 PostgreSQL 服务，这样每次系统重启后会自动启动：

```
$ pg_ctl register -N "pgsql" -D D:\data
```

这里的 `data` 目录就是上面创建的，并且一定要使用绝对路径！

启动服务：

```
$ sc start pgsql
```

最后使用我们前面介绍的命令创建数据库和表即可。

如果有使用 `installer` 成功安装的小伙伴，还请不吝赐教！

总结

虽然目前还有一些不完善的地方，例如对 MySQL 的支持是实验性的， `sqlc` 工具的确能大大简化我们使用 Go 编写数据库代码的复杂度，提升我们的编码效率，减少我们出错的概率。使用 PostgreSQL 的小伙伴非常建议尝试一番！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. sqlc GitHub：https://github.com/kyleconroy/sqlc
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

nutsdb

简介

`nutsdb` 是一个完全由 Go 编写的简单、快速、可嵌入的持久化存储。 `nutsdb` 与我们之前介绍过的 `buntdb` 有些类似，但是支持 `List` 、 `Set` 、 `Sorted Set` 这些数据结构。

快速使用

先安装：

```
$ go get github.com/xujiajun/nutsdb
```

后使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/xujiajun/nutsdb"
)

func main() {
  opt := nutsdb.DefaultOptions
  opt.Dir = "./nutsdb"
  db, err := nutsdb.Open(opt)
  if err != nil {
    log.Fatal(err)
  }
  defer db.Close()

  err = db.Update(func(tx *nutsdb.Tx) error {
    key := []byte("name")
    val := []byte("dj")
    if err := tx.Put("", key, val, 0); err != nil {
      return err
    }
    return nil
```

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

  err = db.View(func(tx *nutsdb.Tx) error {
    key := []byte("name")
    if e, err := tx.Get("", key); err != nil {
      return err
    } else {
      fmt.Println(string(e.Value))
    }
    return nil
  })
  if err != nil {
    log.Fatal(err)
  }
}
```

看过前面介绍 `buntdb` 文章的小伙伴会发现， `nutsdb` 的简单使用与 `buntdb` 非常相似。首先打开数据库 `nutsdb.Open()` ，通过选项指定数据库文件存放目录。数据的插入、修改和查找都是包装在一个事务方法中执行的。 `nutsdb` 允许同时存在多个读事务。但是有写事务存在时，其他事务不能并发执行。需要修改数据的操作在 `db.Update()` 的回调中执行，无副作用的操作在 `db.View()` 的回调中执行。上面代码先插入一个键值对，然后读取这个键。

从代码我们可以看出，由于涉及数据库操作，需要大量的错误处理。为了简洁起见，本文后面的代码省略了错误处理，在实际使用中必须加上！

特性

桶

桶（ `bucket` ）有点像命名空间的概念。在同一个桶中的键不能重复，不同的桶可以包含相同的键。 `nutsdb` 提供的更新和查询接口都需要传入桶名，只是我们在最开始的例子中将桶名设置为空字符串了。

```
func main() {
  opt := nutsdb.DefaultOptions
  opt.Dir = "./nutsdb"
  db, _ := nutsdb.Open(opt)
  defer db.Close()

  key := []byte("name")
```

```
    val := []byte("dj")

    db.Update(func(tx *nutsdb.Tx) error {
        tx.Put("bucket1", key, val, 0)
        return nil
    })

    db.Update(func(tx *nutsdb.Tx) error {
        tx.Put("bucket2", key, val, 0)
        return nil
    })

    db.View(func(tx *nutsdb.Tx) error {
        e, _ := tx.Get("bucket1", key)
        fmt.Println("val1:", string(e.Value))

        e, _ = tx.Get("bucket2", key)
        fmt.Println("val2:", string(e.Value))
        return nil
    })
}
```

运行：

```
val1: dj
val2: dj
```

我们可以将桶类比于 redis 中的 db 的概念，redis 可以在不同的 db 中存储相同的键，但是同一个 db 的键是唯一的。通过 redis 客户端连接服务器后，使用命令 `select db` 切换不同的 db。

更新和删除

上面我们看到使用 `tx.Put()` 插入字段，其实 `tx.Put()` 也用来更新（如果键已存在）。 `tx.Delete()` 用来删除一个字段。

```
func main() {
    opt := nutsdb.DefaultOptions
    opt.Dir = "./nutsdb"
    db, _ := nutsdb.Open(opt)
    defer db.Close()

    key := []byte("name")
    val := []byte("dj")
```

```
  db.Update(func(tx *nutsdb.Tx) error {
    tx.Put("", key, val, 0)
    return nil
  })

  db.View(func(tx *nutsdb.Tx) error {
    e, _ := tx.Get("", key)
    fmt.Println(string(e.Value))
    return nil
  })

  db.Update(func(tx *nutsdb.Tx) error {
    tx.Delete("", key)
    return nil
  })

  db.View(func(tx *nutsdb.Tx) error {
    e, err := tx.Get("", key)
    if err != nil {
      log.Fatal(err)
    } else {
      fmt.Println(string(e.Value))
    }
    return nil
  })
}
```

删除后再次 `Get()` ，返回 `err` ：

```
dj
2020/04/27 22:28:19 key not found in the bucket
exit status 1
```

过期

`nutsdb` 支持在插入或更新键值对时设置一个过期时间。 `Put()` 的第四个参数即为过期时间，单位 s。传 0 表示不过期：

```
func main() {
  opt := nutsdb.DefaultOptions
  opt.Dir = "./nutsdb"
  db, _ := nutsdb.Open(opt)
  defer db.Close()
```

```
    key := []byte("name")
    val := []byte("dj")
    db.Update(func(tx *nutsdb.Tx) error {
        tx.Put("", key, val, 10)
        return nil
    })

    db.View(func(tx *nutsdb.Tx) error {
        e, _ := tx.Get("", key)
        fmt.Println(string(e.Value))
        return nil
    })

    time.Sleep(10 * time.Second)

    db.View(func(tx *nutsdb.Tx) error {
        e, err := tx.Get("", key)
        if err != nil {
            log.Fatal(err)
        } else {
            fmt.Println(string(e.Value))
        }
        return nil
    })
}
```

插入一个数据，设置过期时间为 10s。等待 10s 之后返回 `err` ：

```
dj
2020/04/27 22:31:16 key not found in the bucket
exit status 1
```

遍历

在 `nutsdb` 的每个桶中，键是以**字节**顺序保存的。这使得顺序遍历异常迅速。

前缀遍历

我们可以使用 `PrefixScan()` 遍历具有特定前缀的键值对。它可以指定从第几个数据开始，返回多少条满足条件的数据。例如，每个玩家在 `nutsdb` 中保存在 `user_` **+ 玩家id**的键中：

```
func main() {
    opt := nutsdb.DefaultOptions
```

```
    opt.Dir = "./nutsdb"
    db, _ := nutsdb.Open(opt)
    defer db.Close()

    bucket := "user_list"
    prefix := "user_"
    db.Update(func(tx *nutsdb.Tx) error {
        for i := 1; i <= 300; i++ {
            key := []byte(prefix + strconv.FormatInt(int64(i), 10))
            val := []byte("dj" + strconv.FormatInt(int64(i), 10))
            tx.Put(bucket, key, val, 0)
        }
        return nil
    })

    db.View(func(tx *nutsdb.Tx) error {
        entries, _, _ := tx.PrefixScan(bucket, []byte(prefix), 25, 100)
        for _, entry := range entries {
            fmt.Println(string(entry.Key), string(entry.Value))
        }
        return nil
    })
}
```

先插入 300 条数据，然后使用 `PrefixScan()` 从第 25 条数据开始，一共返回 100 条数据。需要注意的是，键是以**字节**顺序排列，所以 `user_21` 在 `user_209` 之后。观察输出：

```
...
user_208 dj208
user_209 dj209
user_21 dj21
user_210 dj210
```

范围遍历

可以使用 `tx.RangeScan()` 只返回键在指定范围内的数据：

```
func main() {
    opt := nutsdb.DefaultOptions
    opt.Dir = "./nutsdb"
    db, _ := nutsdb.Open(opt)
    defer db.Close()
```

```
    bucket := "user_list"
    prefix := "user_"
    db.Update(func(tx *nutsdb.Tx) error {
        for i := 1; i <= 300; i++ {
            key := []byte(prefix + strconv.FormatInt(int64(i), 10))
            val := []byte("dj" + strconv.FormatInt(int64(i), 10))
            tx.Put(bucket, key, val, 0)
        }
        return nil
    })

    db.View(func(tx *nutsdb.Tx) error {
        lbound := []byte("user_100")
        ubound := []byte("user_199")
        entries, _ := tx.RangeScan(bucket, lbound, ubound)
        for _, entry := range entries {
            fmt.Println(string(entry.Key), string(entry.Value))
        }
        return nil
    })
}
```

获取全部

调用 `tx.GetAll()` 返回某个桶中所有的数据：

```
func main() {
    opt := nutsdb.DefaultOptions
    opt.Dir = "./nutsdb"
    db, _ := nutsdb.Open(opt)
    defer db.Close()

    bucket := "user_list"
    prefix := "user_"
    db.Update(func(tx *nutsdb.Tx) error {
        for i := 1; i <= 300; i++ {
            key := []byte(prefix + strconv.FormatInt(int64(i), 10))
            val := []byte("dj" + strconv.FormatInt(int64(i), 10))
            tx.Put(bucket, key, val, 0)
        }
        return nil
    })

    db.View(func(tx *nutsdb.Tx) error {
        entries, _ := tx.GetAll(bucket)
```

```
    for _, entry := range entries {
      fmt.Println(string(entry.Key), string(entry.Value))
    }
    return nil
  })
}
```

数据结构

相比其他数据库， `nutsdb` 比较强大的地方在于它支持多种数据结构： `list/sorted set` 。命令主要仿造 `redis` 命令编写。这三种结构的操作与 `redis` 中对应的命令非常相似，本文简单介绍一下 `list` 相关方法， `set/sorted set` 可自行探索。

`nutsdb` 中支持的 `list` 方法如下：

- `LPush` ：从头部插入一个元素；
- `RPush` ：从尾部插入一个元素；
- `LPop` ：从头部删除一个元素；
- `RPop` ：从尾部删除一个元素；
- `LPeek` ：返回头部第一个元素；
- `RPeek` ：返回尾部第一个元素；
- `LRange` ：返回指定**索引**范围内的元素；
- `LRem` ：删除指定数量的值等于特定值的项；
- `LSet` ：设置某个索引的值；
- `Ltrim` ：只保留指定索引范围内的元素，其它都移除；
- `LSize` ：返回 `list` 长度。

下面简单演示一下如何使用这些方法，每一步的操作结果都以注释写在了命令下方：

```
func main() {
  opt := nutsdb.DefaultOptions
  opt.Dir = "./nutsdb"
  db, _ := nutsdb.Open(opt)
  defer db.Close()

  bucket := "list"
  key := []byte("userList")

  db.Update(func(tx *nutsdb.Tx) error {
    // 从头部依次插入多个值，注意顺序
```

```
    tx.LPush(bucket, key, []byte("user1"), []byte("user3"), []byte("user5"))
    // 当前list: user5, user3, user1

    // 从尾部依次插入多个值
    tx.RPush(bucket, key, []byte("user7"), []byte("user9"), []byte("user11"))
    // 当前list: user5, user3, user1, user7, user9, user11
    return nil
  })

  db.Update(func(tx *nutsdb.Tx) error {
    // 从头部删除一个值
    tx.LPop(bucket, key)
    // 当前list: user3, user1, user7, user9, user11

    // 从尾部删除一个值
    tx.RPop(bucket, key)
    // 当前list: user3, user1, user7, user9

    // 从头部删除两个值
    tx.LRem(bucket, key, 2)
    // 当前list: user7, user9
    return nil
  })

  db.View(func(tx *nutsdb.Tx) error {
    // 头部第一个值, user7
    b, _ := tx.LPeek(bucket, key)
    fmt.Println(string(b))

    // 长度
    l, _ := tx.LSize(bucket, key)
    fmt.Println(l)
    return nil
  })
}
```

注意不要在同一个 `Update` 中执行插入和删除。

数据库备份

`nutsdb` 可以很方便地进行数据库备份，只需要调用 `db.Backup()` ，传入备份存放目录即可：

```
func main() {
  opt := nutsdb.DefaultOptions
  opt.Dir = "./nutsdb"
  db, _ := nutsdb.Open(opt)

  key := []byte("name")
  val := []byte("dj")
  db.Update(func(tx *nutsdb.Tx) error {
    tx.Put("", key, val, 0)
    return nil
  })

  db.Backup("./backup")
  db.Close()

  opt.Dir = "./backup"
  backupDB, _ := nutsdb.Open(opt)
  backupDB.View(func(tx *nutsdb.Tx) error {
    e, _ := tx.Get("", key)
    fmt.Println(string(e.Value))
    return nil
  })
}
```

上面先备份，再从备份中加载数据库，读取键。

总结

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. nutsdb GitHub：https://github.com/xujiajun/nutsdb
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

zerolog

简介

每个编程语言都有很多日志库，因为记录日志在每个项目中都是必须的。前面我们介绍了标准日志库 `log` 、好用的 `logrus` 和上一篇文章中介绍的由 uber 开源的高性能日志库 `zap` 。 `zerolog` 相比 `zap` 更进了一步，它的 API 设计非常注重开发体验和性能。 `zerolog` 只专注于记录 JSON 格式的日志，号称 0 内存分配！

快速使用

先安装：

```
$ go get github.com/rs/zerolog/log
```

后使用：

```
package main

import "github.com/rs/zerolog/log"

func main() {
  log.Print("hello world")
}
```

常规使用与标准库 `log` 非常相似，只不过输出的是 JSON 格式的日志：

```
{"level":"debug","time":"2020-04-25T13:43:08+08:00","message":"hello world"}
```

字段

我们可以在日志中添加额外的字段信息，有助于调试和问题追踪。与 `zap` 一样， `zerolog` 也区分字段类型，不同的是 `zerolog` 采用**链式调用**的方式：

```
func main() {
  log.Debug().
    Str("Scale", "833 cents").
    Float64("Interval", 833.09).
    Msg("Fibonacci is everywhere")
```

```
  log.Debug().
    Str("Name", "Tom").
    Send()
}
```

调用 `Msg()` 或 `Send()` 之后，日志会被输出：

```
{"level":"debug","Scale":"833 cents","Interval":833.09,"time":"2020-04-25T13:55:44+08:00","message":"Fibonacci is everywhere"}
{"level":"debug","Name":"Tom","time":"2020-04-25T13:55:44+08:00"}
```

嵌套

记录的字段可以任意嵌套，这通过 `Dict()` 来实现：

```
func main() {
  log.Info().
    Dict("dict", zerolog.Dict().
      Str("bar", "baz").
      Int("n", 1),
    ).Msg("hello world")
}
```

输出中 `dict` 字段为嵌套结构：

```
{"level":"info","dict":{"bar":"baz","n":1},"time":"2020-04-25T14:34:51+08:00","message":"hello world"}
```

全局 `Logger`

上面我们使用 `log.Debug()` 、 `log.Info()` 调用的是全局的 `Logger` 。全局的 `Logger` 使用比较简单，不需要额外创建。

设置日志级别

每个日志库都有日志级别的概念，而且划分基本上都差不多。 `zerolog` 有 `panic/fatal/error/warn/info/debug/trace` 这几种级别。我们可以调用 `SetGlobalLevel()` 设置全局 `Logger` 的日志级别。

```
func main() {
  debug := flag.Bool("debug", false, "sets log level to debug")
```

```go
  flag.Parse()

  if *debug {
    zerolog.SetGlobalLevel(zerolog.DebugLevel)
  } else {
    zerolog.SetGlobalLevel(zerolog.InfoLevel)
  }

  log.Debug().Msg("This message appears only when log level set to debug")
  log.Info().Msg("This message appears when log level set to debug or info")

  if e := log.Debug(); e.Enabled() {
    e.Str("foo", "bar").Msg("some debug message")
  }
}
```

在上面代码中，我们根据传入的命令行选项设置日志级别是 `Debug` 还是 `Info` 。如果日志级别为 `Info` ， `Debug` 的日志是不会输出的。也可以调用 `Enabled()` 方法来判断日志是否需要输出，需要时再调用相应方法输出，节省了添加字段和日志信息的开销：

```go
if e := log.Debug(); e.Enabled() {
  e.Str("foo", "bar").Msg("some debug message")
}
```

先不加命令行参数运行，默认为 `Info` 级别， `Debug` 日志不会输出：

```
$ go run main.go
{"level":"info","time":"2020-04-25T14:13:34+08:00","message":"This message appears when log level set to debug or info"}
```

加上 `-debug` 选项， `Debug` 和 `Info` 日志都输出了：

```
$ go run main.go -debug
{"level":"debug","time":"2020-04-25T14:18:19+08:00","message":"This message appears only when log level set to debug"}
{"level":"info","time":"2020-04-25T14:18:19+08:00","message":"This message appears when log level set to debug or info"}
{"level":"debug","foo":"bar","time":"2020-04-25T14:18:19+08:00","message":"some
 debug
message"}
```

不输出级别和信息

有时候我们不想输出日志级别（即 `level` 字段），这时可以使用 `log.Log()` 方法。有时，我们没有日志信息可输出，这时传一个空字符串给 `Msg()` 方法：

```
func main() {
  log.Log().
    Str("foo", "bar").
    Msg("")
}
```

运行：

```
{"foo":"bar","time":"2020-04-25T14:19:48+08:00"}
```

创建 `Logger`

上面我们使用的都是全局的 `Logger`，这种方式有一个明显的缺点：如果在某个地方修改了设置，将影响全局的日志记录。为了消除这种影响，我们需要创建新的 `Logger`：

```
func main() {
  logger := zerolog.New(os.Stderr)
  logger.Info().Str("foo", "bar").Msg("hello world")
}
```

调用 `zerlog.New()` 传入一个 `io.Writer` 作为日志写入器即可。

子 `Logger`

基于当前的 `Logger` 可以创建一个子 `Logger`，子 `Logger` 可以在父 `Logger` 上附加一些额外的字段。调用 `logger.With()` 创建一个上下文，然后为它添加字段，最后调用 `Logger()` 返回一个新的 `Logger`：

```
func main() {
  logger := zerolog.New(os.Stderr)
  sublogger := logger.With().
    Str("foo", "bar").
    Logger()
  sublogger.Info().Msg("hello world")
}
```

`sublogger` 会额外输出 `"foo": "bar"` 这个字段。

设置

`zerolog` 提供了多种选项定制输入日志的行为。

美化输出

`zerolog` 提供了一个 `ConsoleWriter` 可输出便于我们阅读的，带颜色的日志。调用 `zerolog.Output()` 来启用 `ConsoleWriter` ：

```
func main() {
  logger := log.Output(zerolog.ConsoleWriter{Out: os.Stderr})
  logger.Info().Str("foo", "bar").Msg("hello world")
}
```

输出：

```
$ go run main.go
2:40PM INF hello world foo=bar
```

我们还能进一步对 `ConsoleWriter` 进行配置，定制输出的级别、信息、字段名、字段值的格式：

```
func main() {
  output := zerolog.ConsoleWriter{Out: os.Stderr, TimeFormat: time.RFC3339}
  output.FormatLevel = func(i interface{}) string {
    return strings.ToUpper(fmt.Sprintf("| %-6s|", i))
  }
  output.FormatMessage = func(i interface{}) string {
    return fmt.Sprintf("***%s****", i)
  }
  output.FormatFieldName = func(i interface{}) string {
    return fmt.Sprintf("%s:", i)
  }
  output.FormatFieldValue = func(i interface{}) string {
    return strings.ToUpper(fmt.Sprintf("%s", i))
  }

  logger := log.Output(output).With().Timestamp().Logger()
  logger.Info().Str("foo", "bar").Msg("hello world")
}
```

实际上就是对级别、信息、字段名和字段值设置钩子，输出前经过钩子函数转换一次：

```
$ go run main.go
2020-04-25T14:46:55+08:00 | INFO  | ***hello world**** foo:BAR
```

`ConsoleWriter` 的性能不够理想，建议只在开发环境中使用！

设置自动添加的字段名

输出的日志中级别默认的字段名为 `level`，信息默认为 `message`，时间默认为 `time`。可以通过 `zerolog` 中 `LevelFieldName/MessageFieldName/TimestampFieldName` 来设置：

```
func main() {
  zerolog.TimestampFieldName = "t"
  zerolog.LevelFieldName = "l"
  zerolog.MessageFieldName = "m"

  logger := zerolog.New(os.Stderr).With().Timestamp().Logger()
  logger.Info().Msg("hello world")
}
```

输出：

```
{"l":"info","t":"2020-04-25T14:53:08+08:00","m":"hello world"}
```

注意，这个设置是全局的！！！

输出文件名和行号

有时我们需要输出文件名和行号，以便能很快定位代码位置，方便找出问题。这可以通过在创建子 `Logger` 时带入 `Caller()` 选项完成：

```
func main() {
  logger := zerolog.New(os.Stderr).With().Caller().Logger()
  logger.Info().Msg("hello world")
}
```

输出：

```
{"level":"info","caller":"d:/code/golang/src/github.com/darjun/go-daily-lib/zerolog/setting/file-line/main.go:11","message":"hello world"}
```

日志采样

有时候日志太多了反而对我们排查问题造成干扰， `zerolog` 支持日志采样的功能，可以每隔多少条日志输出一次，其他日志丢弃：

```
func main() {
  sampled := log.Sample(&zerolog.BasicSampler{N: 10})

  for i := 0; i < 20; i++ {
    sampled.Info().Msg("will be logged every 10 message")
  }
}
```

结果只输出两条：

```
{"level":"info","time":"2020-04-25T15:01:02+08:00","message":"will be logged every 10 message"}
{"level":"info","time":"2020-04-25T15:01:02+08:00","message":"will be logged every 10 message"}
```

还有更高级的设置：

```
func main() {
  sampled := log.Sample(&zerolog.LevelSampler{
    DebugSampler: &zerolog.BurstSampler{
      Burst:       5,
      Period:      time.Second,
      NextSampler: &zerolog.BasicSampler{N: 100},
    },
  })

  sampled.Debug().Msg("hello world")
}
```

上面代码只采样 `Debug` 日志，在 1s 内最多输出 5 条日志，超过 5条 时，每隔 100 条输出一条。

钩子

`zerolog` 支持钩子，我们可以针对不同的日志级别添加一些额外的字段或进行其他的操作：

```
type AddFieldHook struct {
}

func (AddFieldHook) Run(e *zerolog.Event, level zerolog.Level, msg string) {
  if level == zerolog.DebugLevel {
    e.Str("name", "dj")
  }
}

func main() {
  hooked := log.Hook(AddFieldHook{})
  hooked.Debug().Msg("")
}
```

如果是 `Debug` 级别，额外输出 `"name":"dj"` 字段：

```
{"level":"debug","time":"2020-04-25T15:09:04+08:00","name":"dj"}
```

性能

关于性能，GitHub 上有一份详细的性能测试，与 `logrus/zap` 等日志库的比较。感兴趣可以去看看：https://github.com/rs/zerolog#benchmarks。 `zerolog` 的性能比 `zap` 还要优秀！

总结

正是因为有很多人不满足于现状，才带来了技术的进步和丰富多彩的开源世界！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. zerolog GitHub：https://github.com/rs/zerolog
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

zap

简介

在很早之前的文章中，我们介绍过 Go 标准日志库 `log` 和结构化的日志库 `logrus` 。在热点函数中记录日志对日志库的执行性能有较高的要求，不能影响正常逻辑的执行时间。 `uber` 开源的日志库 `zap` ，对性能和内存分配做了极致的优化。

快速使用

先安装：

```
$ go get go.uber.org/zap
```

后使用：

```
package main

import (
  "time"

  "go.uber.org/zap"
)

func main() {
  logger := zap.NewExample()
  defer logger.Sync()

  url := "http://example.org/api"
  logger.Info("failed to fetch URL",
    zap.String("url", url),
    zap.Int("attempt", 3),
    zap.Duration("backoff", time.Second),
  )

  sugar := logger.Sugar()
  sugar.Infow("failed to fetch URL",
    "url", url,
    "attempt", 3,
    "backoff", time.Second,
  )
```

```
    sugar.Infof("Failed to fetch URL: %s", url)
}
```

`zap` 库的使用与其他的日志库非常相似。先创建一个 `logger` ，然后调用各个级别的方法记录日志（ `Debug/Info/Error/Warn` ）。 `zap` 提供了几个快速创建 `logger` 的方法， `zap.NewExample()` 、 `zap.NewDevelopment()` 、 `zap.NewProduction()` ，还有高度定制化的创建方法 `zap.New()` 。创建前 3 个 `logger` 时， `zap` 会使用一些预定义的设置，它们的使用场景也有所不同。 `Example` 适合用在测试代码中， `Development` 在开发环境中使用， `Production` 用在生成环境。

`zap` 底层 API 可以设置缓存，所以一般使用 `defer logger.Sync()` 将缓存同步到文件中。

由于 `fmt.Printf` 之类的方法大量使用 `interface{}` 和反射，会有不少性能损失，并且增加了内存分配的频次。 `zap` 为了提高性能、减少内存分配次数，没有使用反射，而且默认的 `Logger` 只支持强类型的、结构化的日志。必须使用 `zap` 提供的方法记录字段。 `zap` 为 Go 语言中所有的基本类型和其他常见类型都提供了方法。这些方法的名称也比较好记忆， `zap.Type` （ `Type` 为 `bool/int/uint/float64/complex64/time.Time/time.Duration/error` 等）就表示该类型的字段， `zap.Typep` 以 `p` 结尾表示该类型指针的字段， `zap.Types` 以 `s` 结尾表示该类型切片的字段。如：

- `zap.Bool(key string, val bool) Field` ： `bool` 字段
- `zap.Boolp(key string, val *bool) Field` ： `bool` 指针字段；
- `zap.Bools(key string, val []bool) Field` ： `bool` 切片字段。

当然也有一些特殊类型的字段：

- `zap.Any(key string, value interface{}) Field` ：任意类型的字段；
- `zap.Binary(key string, val []byte) Field` ：二进制串的字段。

当然，每个字段都用方法包一层用起来比较繁琐。 `zap` 也提供了便捷的方法 `SugarLogger` ，可以使用 `printf` 格式符的方式。调用 `logger.Sugar()` 即可创建 `SugaredLogger` 。 `SugaredLogger` 的使用比 `Logger` 简单，只是性能比 `Logger` 低 50% 左右，可以用在非热点函数中。调用 `SugarLogger` 以 `f` 结尾的方法与 `fmt.Printf` 没什么区别，如例子中的 `Infof` 。同时 `SugarLogger` 还支持以 `w` 结尾的方法，这种方式不需要先创建字段对象，直接将字段名和值依次放在参数中即可，如例子中的 `Infow` 。

默认情况下， `Example` 输出的日志为 JSON 格式：

```
{"level":"info","msg":"failed to fetch URL","url":"http://example.org/api","attempt":3,"backoff":"1s"}
{"level":"info","msg":"failed to fetch URL","url":"http://example.org/api","attempt":3,"backoff":"1s"}
{"level":"info","msg":"Failed to fetch URL: http://example.org/api"}
```

记录层级关系

前面我们记录的日志都是一层结构，没有嵌套的层级。我们可以使用 `zap.Namespace(key string) Field` 构建一个**命名空间**，后续的 `Field` 都记录在此命名空间中：

```go
func main() {
  logger := zap.NewExample()
  defer logger.Sync()

  logger.Info("tracked some metrics",
    zap.Namespace("metrics"),
    zap.Int("counter", 1),
  )

  logger2 := logger.With(
    zap.Namespace("metrics"),
    zap.Int("counter", 1),
  )
  logger2.Info("tracked some metrics")
}
```

输出：

```
{"level":"info","msg":"tracked some metrics","metrics":{"counter":1}}
{"level":"info","msg":"tracked some metrices","metrics":{"counter":1}}
```

上面我们演示了两种 `Namespace` 的用法，一种是直接作为字段传入 `Debug/Info` 等方法，一种是调用 `With()` 创建一个新的 `Logger`，新的 `Logger` 记录日志时总是带上预设的字段。`With()` 方法实际上是创建了一个新的 `Logger`：

```go
// src/go.uber.org/zap/logger.go
func (log *Logger) With(fields ...Field) *Logger {
  if len(fields) == 0 {
    return log
  }
  l := log.clone()
```

```
  l.core = l.core.With(fields)
  return l
}
```

定制 `Logger`

调用 `NexExample()/NewDevelopment()/NewProduction()` 这 3 个方法，`zap` 使用默认的配置。我们也可以手动调整，配置结构如下：

```
// src/go.uber.org/zap/config.go
type Config struct {
  Level AtomicLevel `json:"level" yaml:"level"`
  Encoding string `json:"encoding" yaml:"encoding"`
  EncoderConfig zapcore.EncoderConfig `json:"encoderConfig" yaml:"encoderConfig"`
  OutputPaths []string `json:"outputPaths" yaml:"outputPaths"`
  ErrorOutputPaths []string `json:"errorOutputPaths" yaml:"errorOutputPaths"`
  InitialFields map[string]interface{} `json:"initialFields" yaml:"initialFields"`
}
```

- `Level`：日志级别；
- `Encoding`：输出的日志格式，默认为 JSON；
- `OutputPaths`：可以配置多个输出路径，路径可以是文件路径和 `stdout`（标准输出）；
- `ErrorOutputPaths`：错误输出路径，也可以是多个；
- `InitialFields`：每条日志中都会输出这些值。

其中 `EncoderConfig` 为编码配置：

```
// src/go.uber.org/zap/zapcore/encoder.go
type EncoderConfig struct {
  MessageKey    string `json:"messageKey" yaml:"messageKey"`
  LevelKey      string `json:"levelKey" yaml:"levelKey"`
  TimeKey       string `json:"timeKey" yaml:"timeKey"`
  NameKey       string `json:"nameKey" yaml:"nameKey"`
  CallerKey     string `json:"callerKey" yaml:"callerKey"`
  StacktraceKey string `json:"stacktraceKey" yaml:"stacktraceKey"`
  LineEnding    string `json:"lineEnding" yaml:"lineEnding"`
  EncodeLevel    LevelEncoder    `json:"levelEncoder" yaml:"levelEncoder"`
  EncodeTime     TimeEncoder     `json:"timeEncoder" yaml:"timeEncoder"`
```

```
    EncodeDuration DurationEncoder `json:"durationEncoder" yaml:"durationEncoder"`
    EncodeCaller   CallerEncoder   `json:"callerEncoder" yaml:"callerEncoder"`
    EncodeName NameEncoder `json:"nameEncoder" yaml:"nameEncoder"`
}
```

- `MessageKey`：日志中信息的键名，默认为`msg`；
- `LevelKey`：日志中级别的键名，默认为`level`；
- `EncodeLevel`：日志中级别的格式，默认为小写，如`debug/info`。

调用`zap.Config`的`Build()`方法即可使用该配置对象创建一个`Logger`：

```go
func main() {
    rawJSON := []byte(`{
      "level":"debug",
      "encoding":"json",
      "outputPaths": ["stdout", "server.log"],
      "errorOutputPaths": ["stderr"],
      "initialFields":{"name":"dj"},
      "encoderConfig": {
        "messageKey": "message",
        "levelKey": "level",
        "levelEncoder": "lowercase"
      }
    }`)

    var cfg zap.Config
    if err := json.Unmarshal(rawJSON, &cfg); err != nil {
      panic(err)
    }
    logger, err := cfg.Build()
    if err != nil {
      panic(err)
    }
    defer logger.Sync()

    logger.Info("server start work successfully!")
}
```

上面创建一个输出到标准输出`stdout`和文件`server.log`的`Logger`。观察输出：

```
{"level":"info","message":"server start work successfully!","name":"dj"}
```

使用 `NewDevelopment()` 创建的 `Logger` 使用的是如下的配置：

```
// src/go.uber.org/zap/config.go
func NewDevelopmentConfig() Config {
  return Config{
    Level:            NewAtomicLevelAt(DebugLevel),
    Development:      true,
    Encoding:         "console",
    EncoderConfig:    NewDevelopmentEncoderConfig(),
    OutputPaths:      []string{"stderr"},
    ErrorOutputPaths: []string{"stderr"},
  }
}

func NewDevelopmentEncoderConfig() zapcore.EncoderConfig {
  return zapcore.EncoderConfig{
    // Keys can be anything except the empty string.
    TimeKey:        "T",
    LevelKey:       "L",
    NameKey:        "N",
    CallerKey:      "C",
    MessageKey:     "M",
    StacktraceKey:  "S",
    LineEnding:     zapcore.DefaultLineEnding,
    EncodeLevel:    zapcore.CapitalLevelEncoder,
    EncodeTime:     zapcore.ISO8601TimeEncoder,
    EncodeDuration: zapcore.StringDurationEncoder,
    EncodeCaller:   zapcore.ShortCallerEncoder,
  }
}
```

`NewProduction()` 的配置可自行查看。

选项

`NewExample()/NewDevelopment()/NewProduction()` 这 3 个函数可以传入若干类型为 `zap.Option` 的选项，从而定制 `Logger` 的行为。又一次见到了**选项模式**！！

`zap` 提供了丰富的选项供我们选择。

输出文件名和行号

调用 `zap.AddCaller()` 返回的选项设置输出文件名和行号。但是有一个前提，必须设置配置对象 `Config` 中的 `CallerKey` 字段。也因此 `NewExample()` 不能输出这个信息

（它的 `Config` 没有设置 `CallerKey` ）。

```
func main() {
  logger, _ := zap.NewProduction(zap.AddCaller())
  defer logger.Sync()

  logger.Info("hello world")
}
```

输出：

```
{"level":"info","ts":1587740198.9508286,"caller":"caller/main.go:9","msg":"hello world"}
```

`Info()` 方法在 `main.go` 的第 9 行被调用。 `AddCaller()` 与 `zap.WithCaller(true)` 等价。

有时我们稍微封装了一下记录日志的方法，但是我们希望输出的文件名和行号是调用封装函数的位置。这时可以使用 `zap.AddCallerSkip(skip int)` 向上跳 1 层：

```
func Output(msg string, fields ...zap.Field) {
  zap.L().Info(msg, fields...)
}

func main() {
  logger, _ := zap.NewProduction(zap.AddCaller(), zap.AddCallerSkip(1))
  defer logger.Sync()

  zap.ReplaceGlobals(logger)

  Output("hello world")
}
```

输出：

```
{"level":"info","ts":1587740501.5592482,"caller":"skip/main.go:15","msg":"hello world"}
```

输出在 `main` 函数中调用 `Output()` 的位置。如果不指定 `zap.AddCallerSkip(1)` ，将输出 `"caller":"skip/main.go:6"` ，这是在 `Output()` 函数中调用 `zap.Info()` 的位置。因为这个 `Output()` 函数可能在很多地方被调用，所以这个位置参考意义并不大。试试看！

输出调用堆栈

有时候在某个函数处理中遇到了异常情况，因为这个函数可能在很多地方被调用。如果我们能输出此次调用的堆栈，那么分析起来就会很方便。我们可以使用 `zap.AddStackTrace(lvl zapcore.LevelEnabler)` 达成这个目的。该函数指定 `lvl` 和之上的级别都需要输出调用堆栈：

```
func f1() {
  f2("hello world")
}

func f2(msg string, fields ...zap.Field) {
  zap.L().Warn(msg, fields...)
}

func main() {
  logger, _ := zap.NewProduction(zap.AddStacktrace(zapcore.WarnLevel))
  defer logger.Sync()

  zap.ReplaceGlobals(logger)

  f1()
}
```

将 `zapcore.WarnLevel` 传入 `AddStacktrace()` ，之后 `Warn()/Error()` 等级别的日志会输出堆栈， `Debug()/Info()` 这些级别不会。运行结果：

```
{"level":"warn","ts":1587740883.4965692,"caller":"stacktrace/main.go:13","msg":"hello world","stacktrace":"main.f2\n\td:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:13\nmain.f1\n\td:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:9\nmain.main\n\td:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:22\nruntime.main\n\tC:/Go/src/runtime/proc.go:203"}
```

把 `stacktrace` 单独拉出来：

```
main.f2
d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:13
  main.f1
  d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/main.go:9
    main.main
    d:/code/golang/src/github.com/darjun/go-daily-lib/zap/option/stacktrace/mai
```

```
n.go:22
        runtime.main
        C:/Go/src/runtime/proc.go:203
```

很清楚地看到调用路径。

全局 `Logger`

为了方便使用， `zap` 提供了两个全局的 `Logger` ，一个是 `*zap.Logger` ，可调用 `zap.L()` 获得；另一个是 `*zap.SugaredLogger` ，可调用 `zap.S()` 获得。需要注意的是，全局的 `Logger` 默认并不会记录日志！它是一个无实际效果的 `Logger` 。看源码：

```
// go.uber.org/zap/global.go
var (
  _globalMu sync.RWMutex
  _globalL  = NewNop()
  _globalS  = _globalL.Sugar()
)
```

我们可以使用 `ReplaceGlobals(logger *Logger) func()` 将 `logger` 设置为全局的 `Logger` ，该函数返回一个无参函数，用于恢复全局 `Logger` 设置：

```
func main() {
  zap.L().Info("global Logger before")
  zap.S().Info("global SugaredLogger before")

  logger := zap.NewExample()
  defer logger.Sync()

  zap.ReplaceGlobals(logger)
  zap.L().Info("global Logger after")
  zap.S().Info("global SugaredLogger after")
}
```

输出：

```
{"level":"info","msg":"global Logger after"}
{"level":"info","msg":"global SugaredLogger after"}
```

可以看到在调用 `ReplaceGlobals` 之前记录的日志并没有输出。

预设日志字段

如果每条日志都要记录一些共用的字段，那么使用 `zap.Fields(fs ...Field)` 创建的选项。例如在服务器日志中记录可能都需要记录 `serverId` 和 `serverName` ：

```
func main() {
  logger := zap.NewExample(zap.Fields(
    zap.Int("serverId", 90),
    zap.String("serverName", "awesome web"),
  ))

  logger.Info("hello world")
}
```

输出：

```
{"level":"info","msg":"hello world","serverId":90,"serverName":"awesome web"}
```

与标准日志库搭配使用

如果项目一开始使用的是标准日志库 `log` ，后面想转为 `zap` 。这时不必修改每一个文件。我们可以调用 `zap.NewStdLog(l *Logger) *log.Logger` 返回一个标准的 `log.Logger` ，内部实际上写入的还是我们之前创建的 `zap.Logger` ：

```
func main() {
  logger := zap.NewExample()
  defer logger.Sync()

  std := zap.NewStdLog(logger)
  std.Print("standard logger wrapper")
}
```

输出：

```
{"level":"info","msg":"standard logger wrapper"}
```

很方便不是吗？我们还可以使用 `NewStdLogAt(l *logger, level zapcore.Level) (*log.Logger, error)` 让标准接口以 `level` 级别写入内部的 `*zap.Logger` 。

如果我们只是想在一段代码内使用标准日志库 `log` ，其它地方还是使用 `zap.Logger` 。可以调用 `RedirectStdLog(l *Logger) func()` 。它会返回一个无参函数恢复设置：

```
func main() {
  logger := zap.NewExample()
  defer logger.Sync()

  undo := zap.RedirectStdLog(logger)
  log.Print("redirected standard library")
  undo()

  log.Print("restored standard library")
}
```

看前后输出变化：

```
{"level":"info","msg":"redirected standard library"}
2020/04/24 22:13:58 restored standard library
```

当然 `RedirectStdLog` 也有一个对应的 `RedirectStdLogAt` 以特定的级别调用内部的 `*zap.Logger` 方法。

总结

`zap` 用在日志性能和内存分配比较关键的地方。本文仅介绍了 `zap` 库的基本使用，子包 `zapcore` 中有更底层的接口，可以定制丰富多样的 `Logger` 。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. zap GitHub：https://github.com/jordan-wright/zap
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

go-app

简介

`go-app` 是一个使用 Go + WebAssembly 技术编写**渐进式 Web 应用**的库。WebAssembly 是一种可以运行在现代浏览器中的新式代码。近两年来，WebAssembly 技术取得了较大的发展。我们现在已经可以使用 C/C++/Rust/Go 等高级语言编写 WebAssembly 代码。本来就来介绍 `go-app` 这个可以方便地使用 Go 语言来编写 WebAssembly 代码的库。

快速使用

`go-app` 对 Go 语言版本有较高的要求（**Go 1.14+**），而且必须使用 `Go module` 。先创建一个目录并初始化 `Go Module` （Win10 + Git Bash）：

```
$ mkdir go-app && cd go-app
$ go mod init
```

然后下载安装 `go-app` 包：

```
$ go get -u -v github.com/maxence-charriere/go-app/v6
```

至于 `Go module` 的详细使用，去看煎鱼大佬的Go Modules 终极入门。

首先，我们要编写 WebAssembly 程序：

```
package main

import "github.com/maxence-charriere/go-app/v6/pkg/app"

type Greeting struct {
  app.Compo
  name string
}

func (g *Greeting) Render() app.UI {
  return app.Div().Body(
    app.Main().Body(
      app.H1().Body(
        app.Text("Hello, "),
```

```
            app.If(g.name != "",
                app.Text(g.name),
            ).Else(
                app.Text("World"),
            ),
        ),
    ),
    app.Input().
        Value(g.name).
        Placeholder("What is your name?").
        AutoFocus(true).
        OnChange(g.OnInputChange),
  )
}

func (g *Greeting) OnInputChange(src app.Value, e app.Event) {
  g.name = src.Get("value").String()
  g.Update()
}

func main() {
  app.Route("/", &Greeting{})
  app.Run()
}
```

在 `go-app` 中使用**组件**来划分功能模块，每个组件结构中必须内嵌 `app.Compo` 。组件要实现 `Render()` 方法，在需要显示该组件时会调用此方法返回显示的页面。 `go-app` 使用**声明式语法**，完全使用 Go 就可以编写 HTML 页面，上面绘制 HTML 的部分比较好理解。上面代码中还实现了一个输入框的功能，并为它添加了一个监听器。每当输入框内容有修改， `OnInputChange` 方法就会调用， `g.Update()` 会使该组件重新渲染显示。

最后将该组件挂载到路径 `/` 上。

编写 WebAssembly 程序之后，需要使用交叉编译的方式将它编译为 `.wasm` 文件：

```
$ GOARCH=wasm GOOS=js go build -o app.wasm
```

如果编译出现错误，使用 `go version` 命令检查 Go 是否是 1.14 或更新的版本。

接下来，我们需要编写一个 Go Web 程序使用这个 `app.wasm` ：

```
package main

import (
  "log"
```

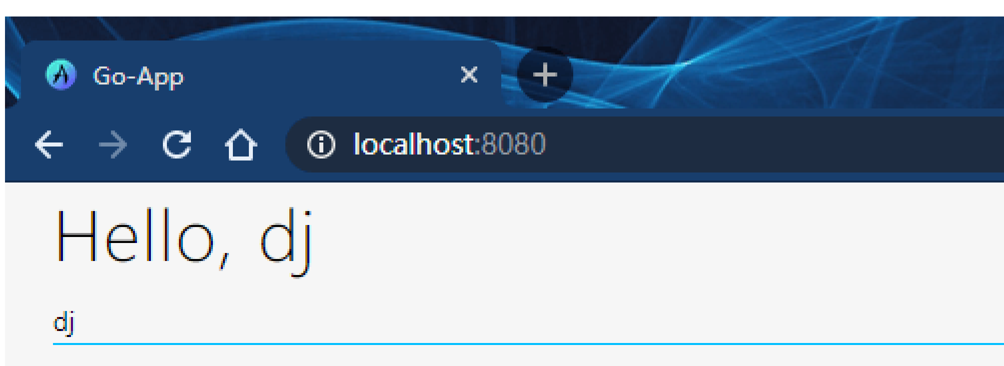
```
	"net/http"

	"github.com/maxence-charriere/go-app/v6/pkg/app"
)

func main() {
	h := &app.Handler{
		Title:  "Go-App",
		Author: "dj",
	}

	if err := http.ListenAndServe(":8080", h); err != nil {
		log.Fatal(err)
	}
}
```

`go-app` 提供了一个 `app.Handler` 结构，它会自动查找同目录下的 `app.wasm`（这也是为什么将目标文件设置为 `app.wasm` 的原因）。然后我们将前面编译生成的 `app.wasm` 放到同一目录下，执行该程序：

```
$ go run main.go
```

默认显示 `"Hello World"`：

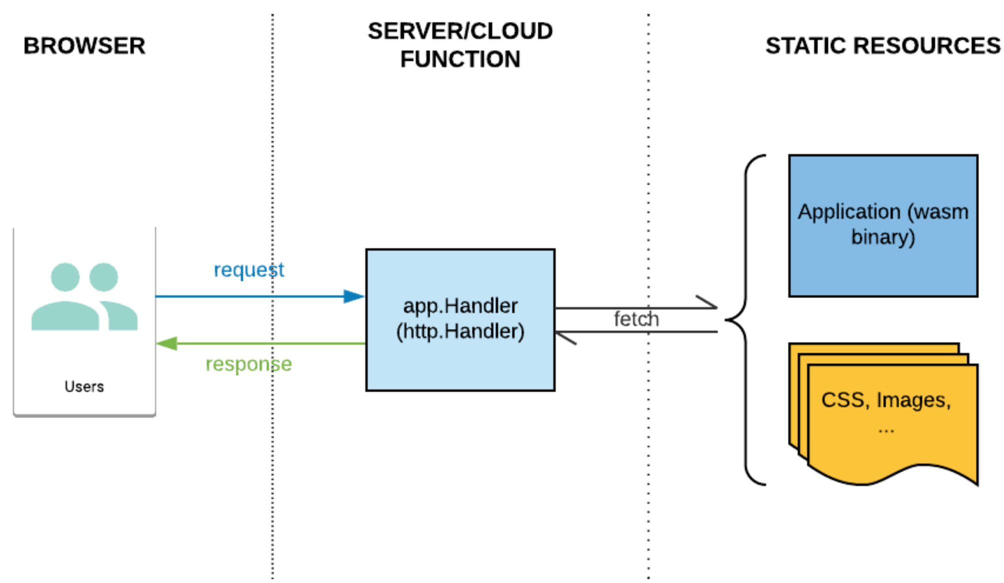
在输入框中输入内容之后，显示会随之变化：



可以看到， `go-app` 为我们设置了一些基本的样式，网页图标等。

简单原理

GitHub 上这张图很好地说明了 HTTP 请求的执行流程：



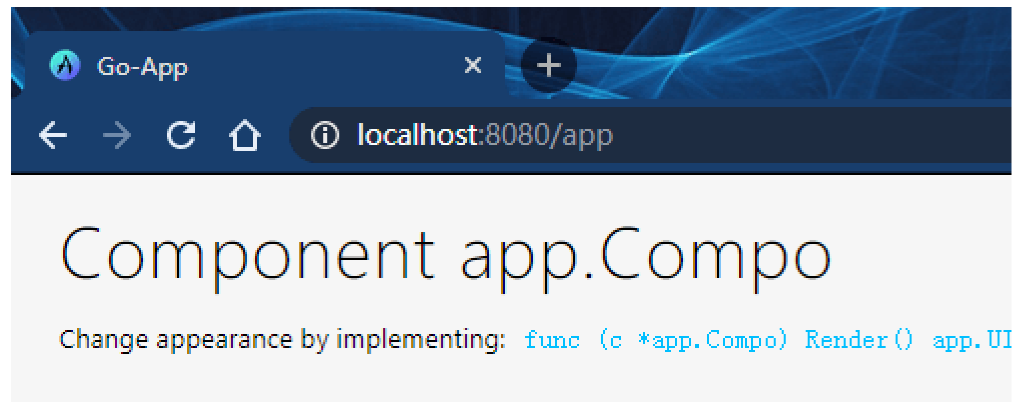
用户请求先到 `app.Handler` 层，它会去 `app.wasm` 中执行相关的路由逻辑、去磁盘上查找静态文件。响应经由 `app.Handler` 中转返回给用户。用户就看到了 `app.wasm` 渲染的页面。实际上，在本文中我们只需要编写一个 Go Web 程序，每次编写新的 WebAssembly 之后，将新编译生成的 app.wasm 文件拷贝到 Go Web 目录下重新运行程序即可。注意，**如果页面未能及时刷新，可能是缓存导致的，可尝试清理浏览器缓存**。

组件

自定义一个组件很简单，只需要将 `app.Compo` 内嵌到结构中即可。实现 `Render()` 方法可定义组件的外观，实际上 `app.Compo` 有一个默认的外观，我们可以这样来查看：

```
func main() {
  app.Route("/app", &app.Compo{})
  app.Run()
}
```

编译生成 `app.wasm` 之后，一开始的 Go Web 程序不需要修改，直接运行，打开浏览器查看：



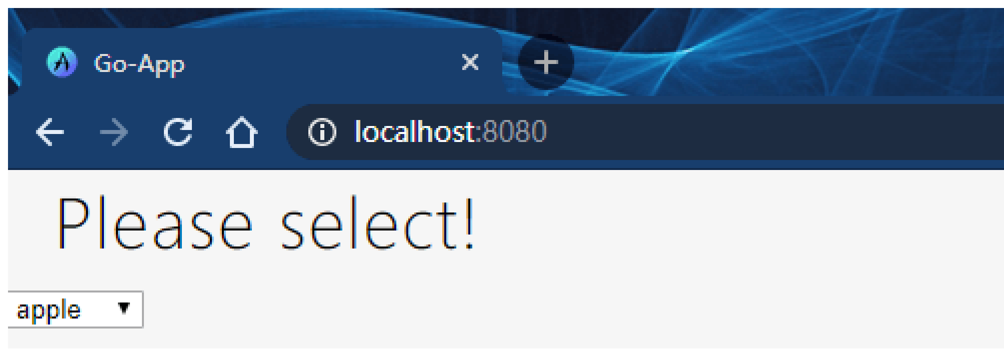
事件处理

在**快速开始**中，我们还介绍了如何使用事件。使用声明式语法 `app.Input().OnChange(handler)` 即可监听内容变化。事件处理函数必须为 `func (src app.Value, e app.Event)` 类型， `app.Value` 是触发对象， `app.Event` 是事件的内容。通过 `app.Value` 我们可以得到输入框内容、选择框的选项等信息，通过 `app.Event` 可以得到事件的信息，是鼠标事件、键盘事件还是其它事件：

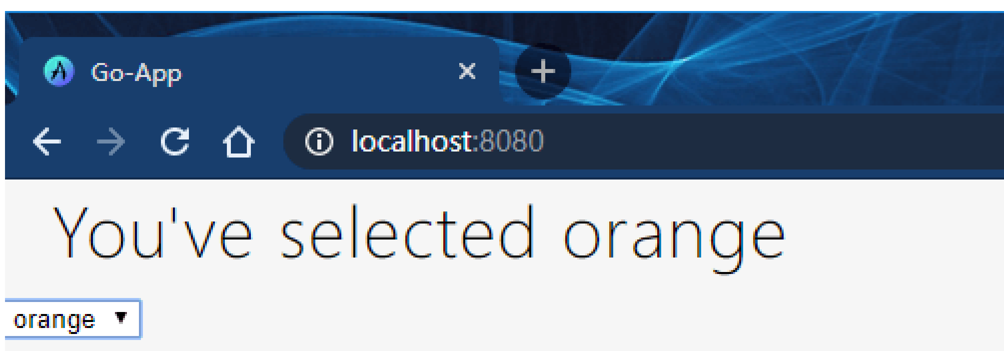
```
type ShowSelect struct {
  app.Compo
  option string
}

func (s *ShowSelect) Render() app.UI {
  return app.Div().Body(
    app.Main().Body(
      app.H1().Body(
        app.If(s.option == "",
          app.Text("Please select!"),
        ).Else(
          app.Text("You've selected "+s.option),
        ),
      ),
    ),
    app.Select().Body(
      app.Option().Body(
        app.Text("apple"),
      ),
      app.Option().Body(
        app.Text("orange"),
      ),
      app.Option().Body(
        app.Text("banana"),
      ),
```

```
    ).
      OnChange(s.OnSelectChange),
  )
}

func (s *ShowSelect) OnSelectChange(src app.Value, e app.Event) {
  s.option = src.Get("value").String()
  s.Update()
}

func main() {
  app.Route("/", &ShowSelect{})
  app.Run()
}
```

上面代码显示一个选择框，当选项改变时上面显示的文字会做相应的改变。初始时：



选择后：



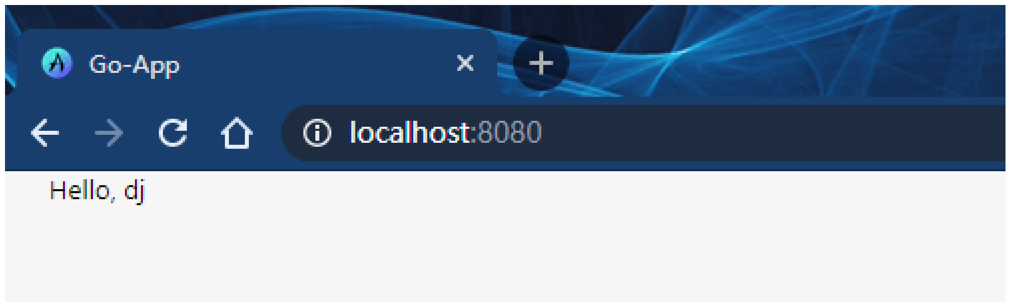
嵌套组件

组件可以嵌套使用，即在一个组件中使用另一个组件。渲染时将内部的组件表现为外部组件的一部分：

```
type Greeting struct {
  app.Compo
}
```

```
func (g *Greeting) Render() app.UI {
  return app.P().Body(
    app.Text("Hello, "),
    &Name{name: "dj"},
  )
}

type Name struct {
  app.Compo
  name string
}

func (n *Name) Render() app.UI {
  return app.Text(n.name)
}

func main() {
  app.Route("/", &Greeting{})
  app.Run()
}
```

上面代码在组件 `Greeting` 中内嵌了一个 `Name` 组件，运行显示：



生命周期

`go-app` 提供了组件的 3 个生命周期的钩子函数：

- `OnMount`：当组件插入到 DOM 时调用；
- `OnNav`：当一个组件所在页面被加载、刷新时调用；
- `OnDismount`：当一个组件从页面中移除时调用。

例如：

```
type Foo struct {
  app.Compo
```

```
}

func (*Foo) Render() app.UI {
  return app.P().Body(
    app.Text("Hello World"),
  )
}

func (*Foo) OnMount() {
  fmt.Println("component mounted")
}

func (*Foo) OnNav(u *url.URL) {
  fmt.Println("component navigated:", u)
}

func (*Foo) OnDismount() {
  fmt.Println("component dismounted")
}

func main() {
  app.Route("/", &Foo{})
  app.Run()
}
```

编译运行，在浏览器中打开页面，打开**浏览器控制台**观察输出：

```
component mounted
component navigated: http://localhost:8080/
```

编写 HTML

在前面的例子中我们已经看到了如何使用**声明式语法**编写 HTML 页面。 `go-app` 为所有标准的 HTML 元素都提供了相关的类型。创建这些对象的方法名也比较好记，就是元素名的首字母大写。如 `app.Div()` 创建一个 `div` 元素， `app.P()` 创建一个 `p` 元素， `app.H1()` 创建一个 `h1` 元素等等。在 `go-app` 中，这些结构都是暴露出对应的接口供开发者使用的，如 `div` 对应 `HTMLDiv` 接口：

```
type HTMLDiv interface {
  Body(nodes ...Node) HTMLDiv
  Class(v string) HTMLDiv
  ID(v string) HTMLDiv
  Style(k, v string) HTMLDiv
```

```
  OnClick(h EventHandler) HTMLDiv
  OnKeyPress(h EventHandler) HTMLDiv
  OnMouseOver(h EventHandler) HTMLDiv
}
```

可以看到每个方法都返回该 `HTMLDiv` 自身，所以支持**链式调用**。调用这些方法可以设置元素的各方面属性：

- `Class`：添加 CSS Class；
- `ID`：设置 ID 属性；
- `Style`：设置内置样式；
- `Body`：设置元素内容，可以随意嵌套。`div` 中包含 `h1` 和 `p`，`p` 中包含 `img` 等；

和设置事件监听：

- `OnClick`：点击事件；
- `OnKeyPress`：按键事件；
- `OnMouseOver`：鼠标移过事件。

例如下面代码：

```
app.Div().Body(
  app.H1().Body(
    app.Text("Title"),
  ),
  app.P().ID("id").
    Class("content").Body(
      app.Text("something interesting"),
    ),
)
```

相当于 HTML 代码：

```
<div>
  <h1>title</h1>
  <p id="id" class="content">
    something interesting
  </p>
</div>
```

原生元素

我们可以在 `app.Raw()` 中直接写 HTML 代码，`app.Raw()` 会生成对应的 `app.UI` 返回：

```
svg := app.Raw(`
<svg width="100" height="100">
    <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow"
/>
</svg>
`)
```

但是这种写法是不安全的，因为没有检查 HTML 的结构。

条件

我们在最开始的例子中就已经用到了条件语句，条件语句对应 3 个方法：`If()/ElseIf()/Else()`。

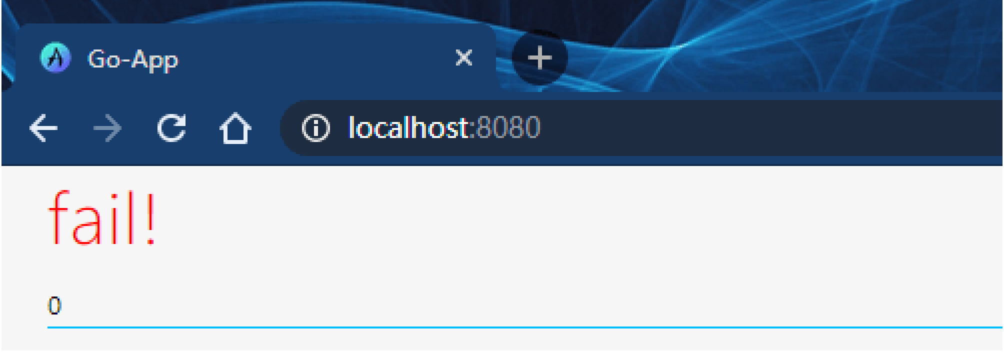
`If` 和 `ElseIf` 接收两个参数，第一个参数为 `bool` 值。如果为 `true`，则显示第二个参数（类型为 `app.UI`），否则不显示。

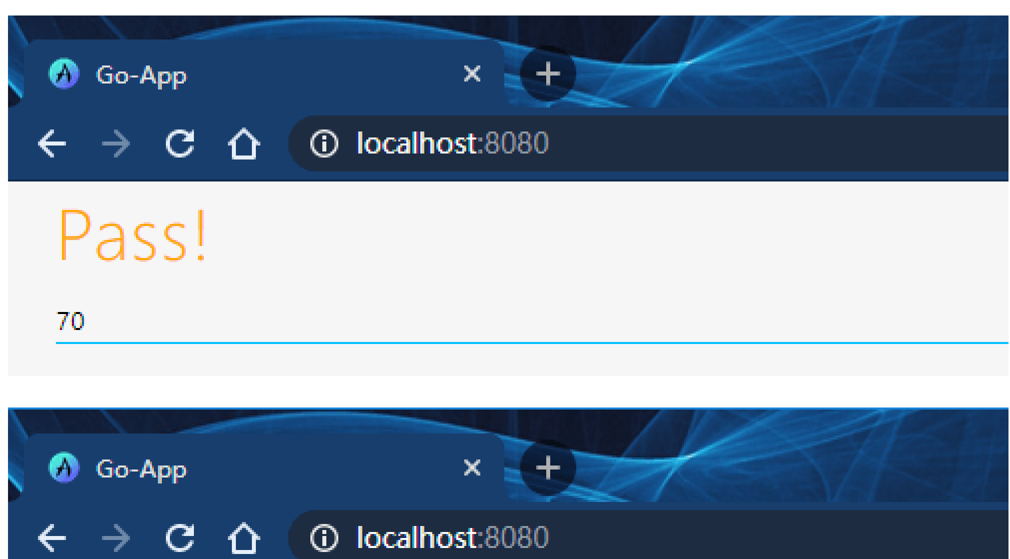
`Else` 必须在 `If` 或 `ElseIf` 后使用，如果前面的条件都不满足，则显示传入 `Else` 方法的 `app.UI`：

```
type ScoreUI struct {
  app.Compo
  score int
}

func (c *ScoreUI) Render() app.UI {
  return app.Div().Body(
    app.If(c.score >= 90,
      app.H1().
        Style("color", "green").
        Body(
          app.Text("Good!"),
        ),
    ).ElseIf(c.score >= 60,
      app.H1().
        Style("color", "orange").
        Body(
          app.Text("Pass!"),
        ),
    ).Else(
```

```go
        app.H1().
            Style("color", "red").
            Body(
                app.Text("fail!"),
            ),
        ),
    app.Input().
        Value(c.score).
        Placeholder("Input your score?").
        AutoFocus(true).
        OnChange(c.OnInputChange),
  )
}

func (c *ScoreUI) OnInputChange(src app.Value, e app.Event) {
  score, _ := strconv.ParseUint(src.Get("value").String(), 10, 32)
  c.score = int(score)
  c.Update()
}

func main() {
  app.Route("/", &ScoreUI{})
  app.Run()
}
```

上面我们根据输入的分数显示对应的文字，`90`及以上显示绿色的`Good!`，`60-90`之间显示橙色的`Pass!`，小于`60`显示红色的`Fail!`。下面是运行结果：





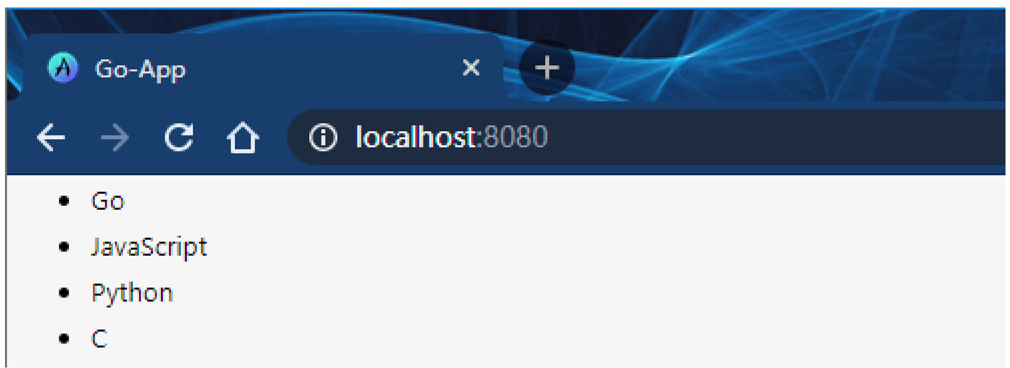
`Range`

假设我们要编写一个 HTML 列表，当前有一个字符串的切片。如果一个个写就太繁琐了，而且不够灵活，且容易出错。这时就可以使用 `Range()` 方法了：

```
type RangeUI struct {
  app.Compo
  name string
}

func (*RangeUI) Render() app.UI {
  langs := []string{"Go", "JavaScript", "Python", "C"}
  return app.Ul().Body(
    app.Range(langs).Slice(func(i int) app.UI {
      return app.Li().Body(
        app.Text(langs[i]),
      )
    }),
  )
}

func main() {
  app.Route("/", &RangeUI{})
  app.Run()
}
```

`Range()` 可以对切片或 `map` 中每一项生成一个 `app.UI` ，然后平铺在某个元素的 `Body()` 方法中。

运行结果：



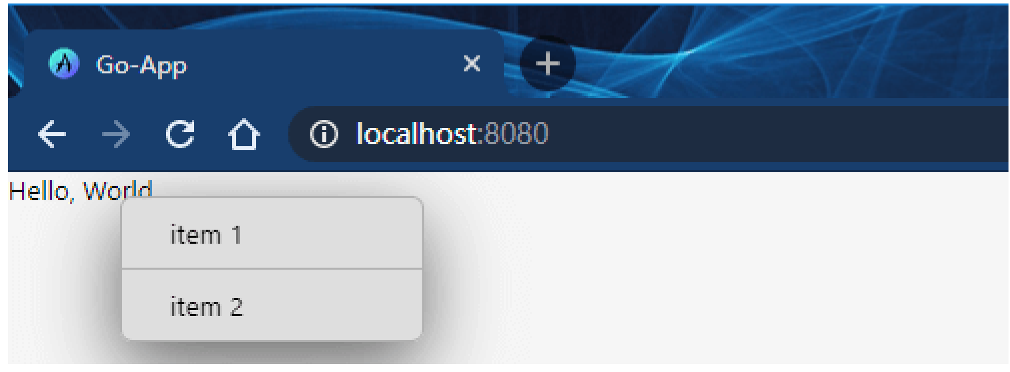
上下文菜单

在 `go-app` 中，我们可以很方便的自定义右键弹出的菜单，并且为菜单项编写响应：

```
type ContextMenuUI struct {
  app.Compo
  name string
}

func (c *ContextMenuUI) Render() app.UI {
  return app.Div().Body(
    app.Text("Hello, World"),
  ).OnContextMenu(c.OnContextMenu)
}

func (*ContextMenuUI) OnContextMenu(src app.Value, event app.Event) {
  event.PreventDefault()

  app.NewContextMenu(
    app.MenuItem().
      Label("item 1").
      OnClick(func(src app.Value, e app.Event) {
        fmt.Println("item 1 clicked")
      }),
    app.MenuItem().Separator(),
    app.MenuItem().
      Label("item 2").
      OnClick(func(src app.Value, e app.Event) {
        fmt.Println("item 2 clicked")
      }),
  )
```

```
}

func main() {
  app.Route("/", &ContextMenuUI{})
  app.Run()
}
```

我们在 `OnContextMenu` 中调用了 `event.PreventDefault()` 阻止默认菜单的弹出。看运行结果：



点击菜单项，观察控制台输出~

`app.Handler`

上面我们都是使用 `go-app` 内置的 `app.Handler` 处理客户端的请求。我们只设置了简单的两个属性 `Author` 和 `Title` 。 `app.Handler` 还有其它很多字段可以定制：

```
type Handler struct {
  Author string
  BackgroundColor string
  CacheableResources []string
  Description string
  Env Environment
  Icon Icon
  Keywords []string
  LoadingLabel string
  Name string
  RawHeaders []string
  RootDir string
  Scripts []string
  ShortName string
  Styles []string
  ThemeColor string
  Title string
```

```
  UseMinimalDefaultStyles bool
  Version string
}
```

- `Icon` ：设置应用图标；
- `Styles` ：CSS 样式文件；
- `Scripts` ：JS 脚本文件。

CSS 和 JS 文件必须在 `app.Handler` 中声明。下面是一个示例 `app.Handler` ：

```
h := &app.Handler{
  Name:        "Luck",
  Author:      "Maxence Charriere",
  Description: "Lottery numbers generator.",
  Icon: app.Icon{
    Default: "/web/icon.png",
  },
  Keywords: []string{
    "EuroMillions",
    "MEGA Millions",
    "Powerball",
  },
  ThemeColor:      "#000000",
  BackgroundColor: "#000000",
  Styles: []string{
    "/web/luck.css",
  },
  Version: "wIKiverSiON",
}
```

本文代码

本文中 WebAssembly 代码都在各自的目录中。Go Web 演示代码在 web 目录中。先进入某个目录，使用下面的命令编译：

```
$ GOARCH=wasm GOOS=js go build -o app.wasm
```

然后将生成的 `app.wasm` 拷贝到 `web` 目录：

```
$ cp app.wasm ../web/
```

切换到 web 目录，启动服务器：

```
$ cd ../web/
$ go run main.go
```

总结

本文介绍如何使用 `go-app` 编写基于 WebAssembly 的 Web 应用程序。可能有人会觉得，`go-app` 编写 HTML 的方式有点繁琐。但是我们可以写一个转换程序将普通的 HTML 代码转为 `go-app` 代码，感兴趣可以自己实现一下。WebAssembly 技术非常值得关注一波~

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. go-app GitHub：https://github.com/maxence-charriere/go-app
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gron

简介

`gron` 是一个比较小巧、灵活的定时任务库，可以执行定时的、周期性的任务。`gron` 提供简洁的、并发安全的接口。我们先介绍 `gron` 库的使用，然后简单分析一下源码。

快速使用

先安装：

```
$ go get github.com/roylee0704/gron
```

后使用：

```
package main

import (
  "fmt"
  "sync"
  "time"

  "github.com/roylee0704/gron"
)

func main() {
  var wg sync.WaitGroup
  wg.Add(1)

  c := gron.New()
  c.AddFunc(gron.Every(5*time.Second), func() {
    fmt.Println("runs every 5 seconds.")
  })
  c.Start()

  wg.Wait()
}
```

`gron` 的使用比较简单：

- 首先调用 `gron.New()` 创建一个**管理器**，这是一个定时任务的管理器；
- 然后调用管理器的 `AddFunc()` 或 `Add()` 方法向它添加任务，在启动时添加也是可以的，见下文分析；
- 最后调用管理器的 `Start()` 方法启动它。

`gron` 支持两种添加任务的方式，一种是使用**无参数**的函数，另一种是实现任务接口。上面例子中使用的是前一种方式，实现接口的方式我们后面会介绍。添加任务时通过 `gron.Every()` 指定周期任务的间隔，上面添加了一个 5s 的周期任务，每隔 5s 输出一行文字。

需要注意的是，我们使用 `sync.WaitGroup` 保证主 goroutine 不退出。因为 `c.Start()` 中只是启动了一个 goroutine，如果主 goroutine 退出了，整个程序就停止了。

运行程序，每隔 5s 输出：

```
runs every 5 seconds.
runs every 5 seconds.
runs every 5 seconds.
```

该程序需要按下 `ctrl + c` 停止！

时间格式

`gron` 接受 `time.Duration` 类型的时间间隔，除了 `time` 包中定义的基础 `Second/Minute/Hour` ， `gron` 中的 `xtime` 子包还提供了 `Day/Week` 单位的时间。有一点需要注意， `gron` 支持的时间精度为 1s，小于 1s 的间隔是不支持的。除了单位时间间隔，我们还可以使用 `4m10s` 这样的时间：

```
func main() {
  var wg sync.WaitGroup
  wg.Add(1)

  c := gron.New()
  c.AddFunc(gron.Every(1*time.Second), func() {
    fmt.Println("runs every second.")
  })
  c.AddFunc(gron.Every(1*time.Minute), func() {
    fmt.Println("runs every minute.")
  })
  c.AddFunc(gron.Every(1*time.Hour), func() {
    fmt.Println("runs every hour.")
  })
```

```
  c.AddFunc(gron.Every(1*xtime.Day), func() {
    fmt.Println("runs every day.")
  })
  c.AddFunc(gron.Every(1*xtime.Week), func() {
    fmt.Println("runs every week.")
  })
  t, _ := time.ParseDuration("4m10s")
  c.AddFunc(gron.Every(t), func() {
    fmt.Println("runs every 4 minutes 10 seconds.")
  })
  c.Start()

  wg.Wait()
}
```

通过 `gron.Every()` 设置每隔多长时间执行一次任务。对于大于 1 天的时间间隔，我们还可以使用 `gron.Every().At()` 指定其在某个时间点执行。例如下面的程序，从第二天的 `22:00` 开始，每隔一天触发一次，即每天的 `22:00` 触发：

```go
func main() {
  var wg sync.WaitGroup
  wg.Add(1)

  c := gron.New()
  c.AddFunc(gron.Every(1*xtime.Day).At("22:00"), func() {
    fmt.Println("runs every second.")
  })
  c.Start()

  wg.Wait()
}
```

自定义任务

实现自定义任务也很简单，只需要实现 `gron.Job` 接口即可：

```go
// src/github.com/roylee0704/gron/cron.go
type Job interface {
  Run()
}
```

我们需要调用调度器的 `Add()` 方法向管理器添加自定义任务：

```go
type GreetingJob struct {
  Name string
}

func (g GreetingJob) Run() {
  fmt.Println("Hello ", g.Name)
}

func main() {
  var wg sync.WaitGroup
  wg.Add(1)

  g1 := GreetingJob{Name: "dj"}
  g2 := GreetingJob{Name: "dajun"}

  c := gron.New()
  c.Add(gron.Every(5*time.Second), g1)
  c.Add(gron.Every(10*time.Second), g2)
  c.Start()

  wg.Wait()
}
```

上面我们编写了一个 `GreetingJob` 结构，实现 `gron.Job` 接口，然后创建两个对象 `g1/g2` ，一个 5s 触发一次，一个 10s 触发一次。使用自定义任务的方式可以比较好地处理携带状态的任务，如上面的 `Name` 字段。

实际上， `AddFunc()` 方法内部也是通过 `Add()` 实现的：

```go
// src/github.com/roylee0704/gron/cron.go
func (c *Cron) AddFunc(s Schedule, j func()) {
  c.Add(s, JobFunc(j))
}

type JobFunc func()

func (j JobFunc) Run() {
  j()
}
```

在 `AddFunc()` 内部，将传入的函数转为 `JobFunc` 类型，而 `gron` 为 `JobFunc` 实现了 `gron.Job` 接口。是不是与 `net/http` 包中的 `HandleFunc` 和 `Handle` 很像。如果注意观察的话，在很多 Go 语言的代码中都有此类模式。

一点源码

`gron` 的源码只有两个文件 `cron.go` 和 `schedule.go` ， `cron.go` 中实现添加任务和调度的方法， `schedule.go` 中是时间策略相关的代码。两个文件算上注释一共才 260 行！我们添加的任务在 `gron` 内部都是以 `Entry` 结构表示的：

```
type Entry struct {
  Schedule Schedule
  Job      Job
  Next time.Time
  Prev time.Time
}
```

`Next` 为下次执行时间， `Prev` 为上次执行时间， `Job` 是要执行的任务， `Schedule` 为 `gron.Schedule` 接口类型，调用其 `Next()` 可计算出下次执行的时间点。

管理器使用 `gron.Cron` 结构表示：

```
type Cron struct {
  entries []*Entry
  running bool
  add     chan *Entry
  stop    chan struct{}
}
```

任务的调度在另外一个 goroutine 中。如果调度未开始，添加任务可直接 `append` 到 `entries` 切片中；如果调度已开始（ `Start()` 方法已调用），需要向通道 `add` 发送待添加的任务。任务调度的核心逻辑在 `Run()` 方法中：

```
func (c *Cron) run() {
  var effective time.Time
  now := time.Now().Local()

  // to figure next trig time for entries, referenced from now
  for _, e := range c.entries {
    e.Next = e.Schedule.Next(now)
  }

  for {
    sort.Sort(byTime(c.entries))
    if len(c.entries) > 0 {
      effective = c.entries[0].Next
    } else {
```

```
        effective = now.AddDate(15, 0, 0) // to prevent phantom jobs.
    }

    select {
    case now = <-after(effective.Sub(now)):
        // entries with same time gets run.
        for _, entry := range c.entries {
            if entry.Next != effective {
                break
            }
            entry.Prev = now
            entry.Next = entry.Schedule.Next(now)
            go entry.Job.Run()
        }
    case e := <-c.add:
        e.Next = e.Schedule.Next(time.Now())
        c.entries = append(c.entries, e)
    case <-c.stop:
        return // terminate go-routine.
    }
  }
}
```

执行流程如下：

1. 调度器刚启动时，先计算所有任务的下次执行时间；
2. 然后在一个 `for` 循环中，按照执行时间从早到晚排序，取出最近需要执行任务的时间点；
3. 在 `select` 语句中等待到这个时间点，启动新的 goroutine 执行到期的任务，每个任务一个新的 goroutine；
4. 如果在等待的过程中，又添加了新的任务（通过通道 `c.add` ），计算这个新任务的首次执行时间。跳到步骤 2，因为新添加的任务可能最早执行。

有几个细节需要注意一下：

1. 任务到期判断使用的是本地时间： `time.Now().Local()` ；
2. 如果没有任务，等待时间设置为 `now.AddDate(15, 0, 0)` ，即 15 年，防止 CPU 空转；
3. 任务都是在独立的 goroutine 中执行的；
4. 通过实现 `sort.Interface` 接口可以实现自定义排序（代码中的 `byTime` ）。

最后，我们来看一下时间策略的代码。我们知道在 `Entry` 结构中存储了一个 `gron.Schedule` 类型的对象，调用该对象的 `Next()` 方法返回下次执行的时间点：

```go
// src/github.com/roylee0704/gron/schedule.go
type Schedule interface {
  Next(t time.Time) time.Time
}
```

`gron` 内置实现了两种 `Schedule`，一种是 `periodicSchedule`，即**周期触发**，`gron.Every()` 函数返回的就是这个对象：

```go
// src/github.com/roylee0704/gron/schedule.go
type periodicSchedule struct {
  period time.Duration
}
```

一种是**固定时刻的周期触发**，它实际上也是周期触发，只是固定了时间点：

```go
type atSchedule struct {
  period time.Duration
  hh     int
  mm     int
}
```

他们的核心逻辑在 `Next()` 方法中，`periodicSchedule` 只需要用当前时间加上周期即可得到下次触发时间。这里 `Truncate()` 方法截掉了当前时间中小于 1s 的部分：

```go
func (ps periodicSchedule) Next(t time.Time) time.Time {
  return t.Truncate(time.Second).Add(ps.period)
}
```

`atSchedule` 的 `Next()` 方法先计算当天该时间点，再加上周期就是下次触发的时间：

```go
func (as atSchedule) reset(t time.Time) time.Time {
  return time.Date(t.Year(), t.Month(), t.Day(), as.hh, as.mm, 0, 0, time.UTC)
}

func (as atSchedule) Next(t time.Time) time.Time {
  next := as.reset(t)
  if t.After(next) {
    return next.Add(as.period)
  }
  return next
}
```

`periodicSchedule` 提供了 `At()` 方法可以转为 `atSchedule` ：

```
func (ps periodicSchedule) At(t string) Schedule {
  if ps.period < xtime.Day {
    panic("period must be at least in days")
  }

  // parse t naively
  h, m, err := parse(t)

  if err != nil {
    panic(err.Error())
  }

  return &atSchedule{
    period: ps.period,
    hh:     h,
    mm:     m,
  }
}
```

自定义时间策略

我们可以很轻松的实现一个自定义的时间策略。例如，我们要实现一个“指数退避”的时间序列，先等待 1s，然后 2s、4s...

```
type ExponentialBackOffSchedule struct {
	last int
}

func (e *ExponentialBackOffSchedule) Next(t time.Time) time.Time {
	interval := time.Duration(math.Pow(2.0, float64(e.last))) * time.Second
	e.last += 1
	return t.Truncate(time.Second).Add(interval)
}

func main() {
	var wg sync.WaitGroup
	wg.Add(1)

	c := gron.New()
	c.AddFunc(&ExponentialBackOffSchedule{}, func() {
		fmt.Println(time.Now().Local().Format("2006-01-02 15:04:05"), "hello")
	})
```

```
    c.Start()

    wg.Wait()
}
```

运行结果如下：

```
2020-04-20 23:47:11 hello
2020-04-20 23:47:13 hello
2020-04-20 23:47:17 hello
2020-04-20 23:47:25 hello
```

第二次输出与第一次相差 2s，第三次与第二次相差 4s，第4次与第三次相差 8s，完美！

总结

本文介绍了 `gron` 这个小巧的定时任务库，如何使用，如何自定义任务和时间策略，顺带分析了一下源码。`gron` 源码实现非常简洁，非常推荐阅读！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gron GitHub：https://github.com/roylee0704/gron
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

plot

简介

本文介绍 Go 语言的一个非常强大、好用的绘图库—— `plot` 。 `plot` 内置了很多常用的组件，基本满足日常需求。同时，它也提供了定制化的接口，可以实现我们的个性化需求。 `plot` 主要用于将数据可视化，便于我们观察、比较。

快速使用

先安装：

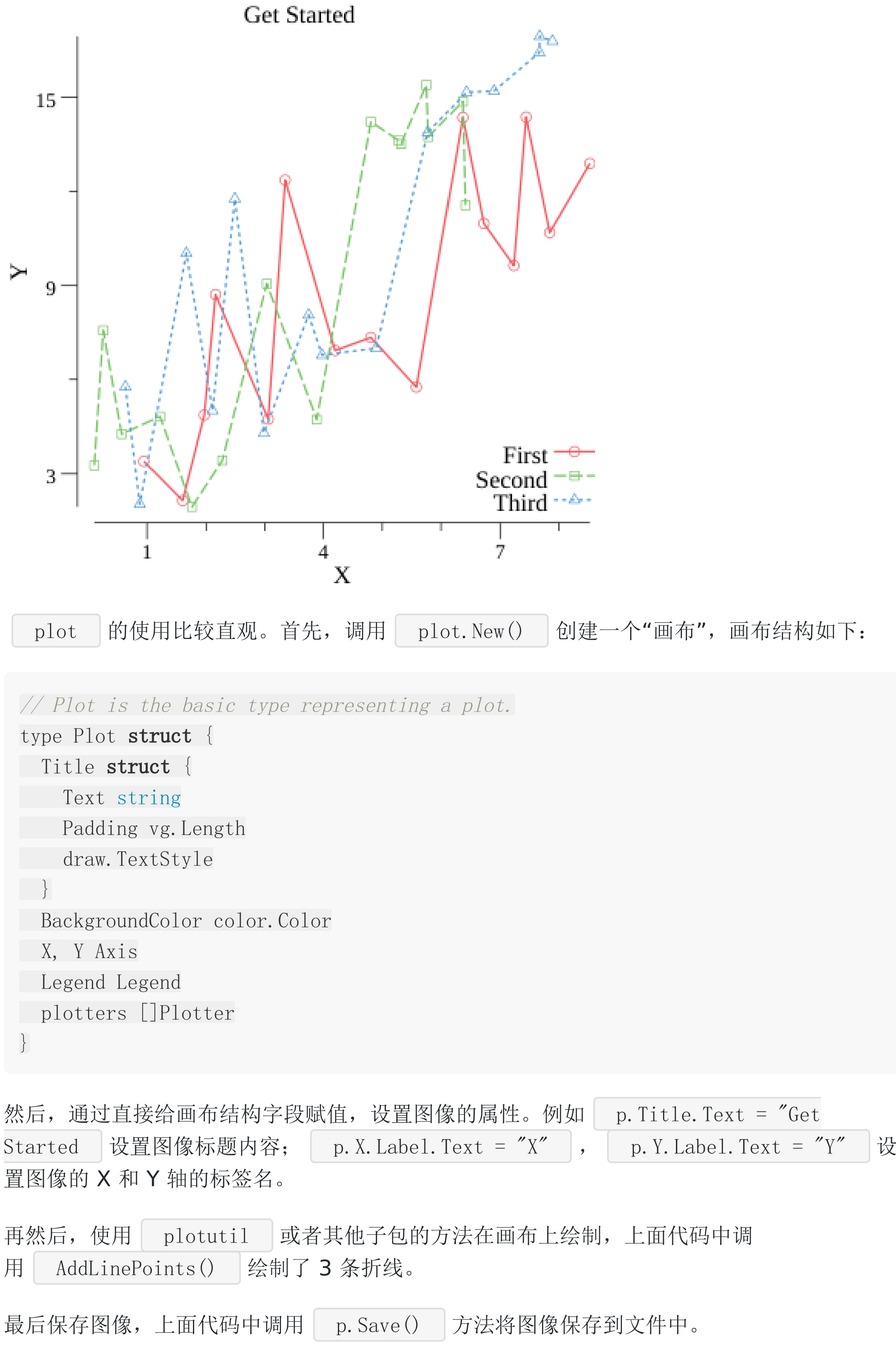
```
$ go get gonum.org/v1/plot/...
```

后使用：

```
package main

import (
  "log"
  "math/rand"

  "gonum.org/v1/plot"
  "gonum.org/v1/plot/plotter"
  "gonum.org/v1/plot/plotutil"
  "gonum.org/v1/plot/vg"
)

func main() {
  rand.Seed(int64(0))

  p, err := plot.New()
  if err != nil {
    log.Fatal(err)
  }

  p.Title.Text = "Get Started"
  p.X.Label.Text = "X"
  p.Y.Label.Text = "Y"

  err = plotutil.AddLinePoints(p,
```

```
    "First", randomPoints(15),
    "Second", randomPoints(15),
    "Third", randomPoints(15))
  if err != nil {
    log.Fatal(err)
  }

  if err = p.Save(4*vg.Inch, 4*vg.Inch, "points.png"); err != nil {
    log.Fatal(err)
  }
}

func randomPoints(n int) plotter.XYs {
  points := make(plotter.XYs, n)
  for i := range points {
    if i == 0 {
      points[i].X = rand.Float64()
    } else {
      points[i].X = points[i-1].X + rand.Float64()
    }
    points[i].Y = points[i].X + 10 * rand.Float64()
  }

  return points
}
```

程序运行输出 `points.png` 图片文件：



`plot` 的使用比较直观。首先，调用 `plot.New()` 创建一个“画布”，画布结构如下：

```
// Plot is the basic type representing a plot.
type Plot struct {
  Title struct {
    Text string
    Padding vg.Length
    draw.TextStyle
  }
  BackgroundColor color.Color
  X, Y Axis
  Legend Legend
  plotters []Plotter
}
```

然后，通过直接给画布结构字段赋值，设置图像的属性。例如 `p.Title.Text = "Get Started` 设置图像标题内容；`p.X.Label.Text = "X"`，`p.Y.Label.Text = "Y"` 设置图像的 X 和 Y 轴的标签名。

再然后，使用 `plotutil` 或者其他子包的方法在画布上绘制，上面代码中调用 `AddLinePoints()` 绘制了 3 条折线。

最后保存图像，上面代码中调用 `p.Save()` 方法将图像保存到文件中。

更多图形

`gonum/plot` 将不同层次的接口封装到特定的子包中：

- `plot`：提供了布局和绘图的简单接口；
- `plotter`：使用 `plot` 提供的接口实现了一组标准的绘图器，例如散点图、条形图、箱状图等。可以使用 `plotter` 提供的接口实现自己的绘图器；
- `plotutil`：为绘制常见图形提供简便的方法；
- `vg`：封装各种后端，并提供了一个通用矢量图形 API。

条形图

条形图通过相同宽度**条形**的高度或长短来表示数据的大小关系。将相同类型的数据放在一起比较能非常直观地看出不同，我们经常在比较几个库的性能时使用条形图。下面我们采用 `json-iter/go` 的 GitHub 仓库中用来比较 `jsoniter`、`easyjson`、`std` 三个 JSON 库性能的数据来绘制条形图：

```go
package main

import (
  "log"

  "gonum.org/v1/plot"
  "gonum.org/v1/plot/plotter"
  "gonum.org/v1/plot/plotutil"
  "gonum.org/v1/plot/vg"
)

func main() {
  std := plotter.Values{35510, 1960, 99}
  easyjson := plotter.Values{8499, 160, 4}
  jsoniter := plotter.Values{5623, 160, 3}

  p, err := plot.New()
  if err != nil {
    log.Fatal(err)
  }

  p.Title.Text = "jsoniter vs easyjson vs std"
  p.Y.Label.Text = ""

  w := vg.Points(20)
  stdBar, err := plotter.NewBarChart(std, w)
  if err != nil {
    log.Fatal(err)
```

```
    }
    stdBar.LineStyle.Width = vg.Length(0)
    stdBar.Color = plotutil.Color(0)
    stdBar.Offset = -w

    easyjsonBar, err := plotter.NewBarChart(easyjson, w)
    if err != nil {
        log.Fatal(err)
    }
    easyjsonBar.LineStyle.Width = vg.Length(0)
    easyjsonBar.Color = plotutil.Color(1)

    jsoniterBar, err := plotter.NewBarChart(jsoniter, w)
    if err != nil {
        log.Fatal(err)
    }
    jsoniterBar.LineStyle.Width = vg.Length(0)
    jsoniterBar.Color = plotutil.Color(2)
    jsoniterBar.Offset = w

    p.Add(stdBar, easyjsonBar, jsoniterBar)
    p.Legend.Add("std", stdBar)
    p.Legend.Add("easyjson", easyjsonBar)
    p.Legend.Add("jsoniter", jsoniterBar)
    p.Legend.Top = true
    p.NominalX("ns/op", "allocation bytes", "allocation times")

    if err = p.Save(5*vg.Inch, 5*vg.Inch, "barchart.png"); err != nil {
        log.Fatal(err)
    }
}
```

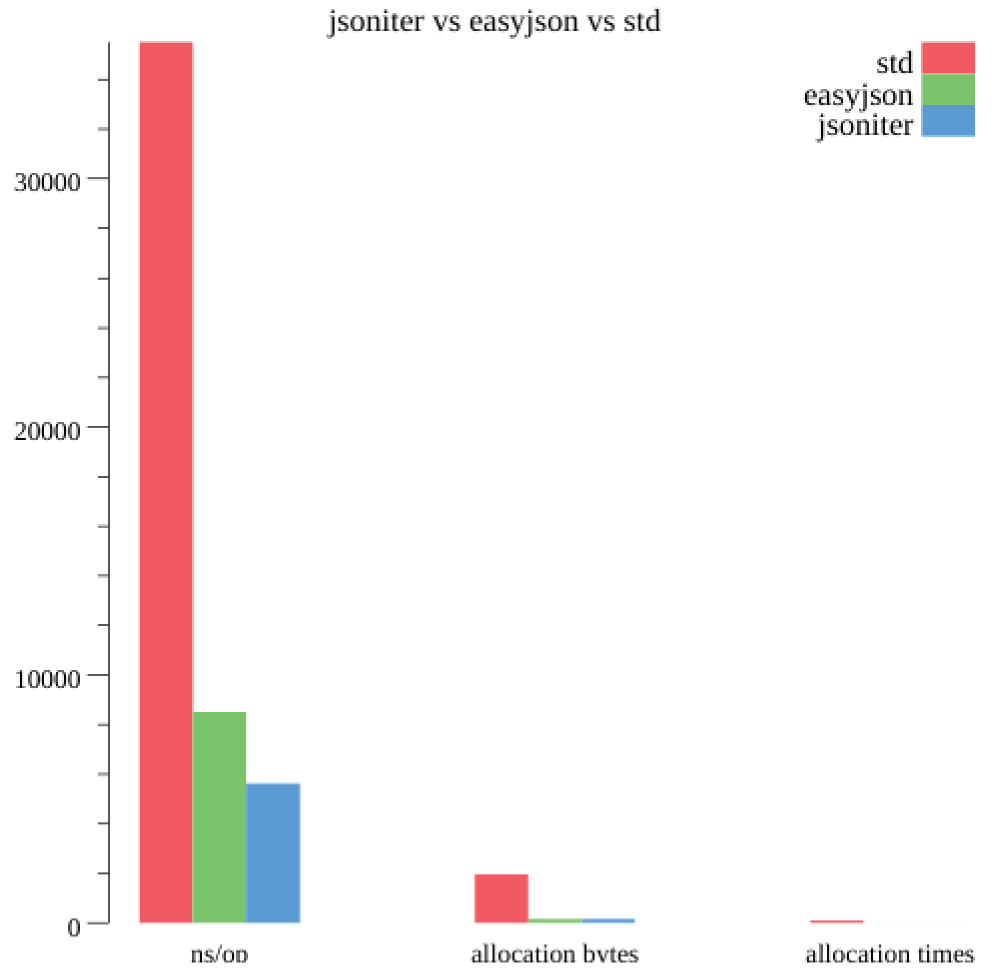
首先生成值列表，我们在最开始的例子中生成了二维坐标列表 `plotter.XYs` ，实际上还有三维坐标列表 `plotter.XYZs` 。

然后，调用 `plotter.NewBarChart()` 分别为三组数据生成条形图。 `w = vg.Points(20)` 用来设置条形的宽度。 `LineStyle.Width` 设置线宽，这个实际上是边框的宽度。 `Color` 设置颜色。 `Offset` 设置偏移，因为每组对应位置的条形放在一起显示更好比较，将 `stdBar.Offset` 设置为 `-w` 会让其向左偏移一个条形的宽度； `easyjson` 偏移不设置，默认为 0，不偏移； `jsoniter` 偏移设置为 `w` ，向右偏移一个条形的宽度。最终它们紧挨着显示。

然后，将 3 个条形图添加到画布上。紧接着，设置它们的**图例**，并将其显示在顶部。

最后调用 `p.Save()` 保存图片。

程序运行生成下面的图片：



可以很直观地看到 `jsoniter` 的性能、内存占用、内存分配次数各方面都是顶尖的。**可能用同一种维度的数据，数量级相差不大，图像会好看点(┬＿┬)。**

注意 `plotter.Color(2)` 这类用法。 `plot` 预定义了一组颜色值，如果我们想要使用它们，可以直接传入索引获取对应的颜色，更多的是为了区分不同的图形（例如上面的 3 个条形图用了 3 个不同的索引）：

```
// src/gonum.org/v1/plot/plotutil/plotutil.go
var DefaultColors = SoftColors
var SoftColors = []color.Color{
  rgb(241, 90, 96),
  rgb(122, 195, 106),
  rgb(90, 155, 212),
  rgb(250, 167, 91),
  rgb(158, 103, 171),
  rgb(206, 112, 88),
  rgb(215, 127, 180),
}
```

```
func Color(i int) color.Color {
  n := len(DefaultColors)
  if i < 0 {
    return DefaultColors[i%n+n]
  }
  return DefaultColors[i%n]
}
```

除了颜色，还有形状 `plotter.Shape(i)` 和划线模式 `plotter.Dashes(i)` 。

`vg.Length(0)` **有所不同，这个只是将 0 转换为** `vg.Length` **类型！**

函数图像

`plot` 可以绘制函数图像！

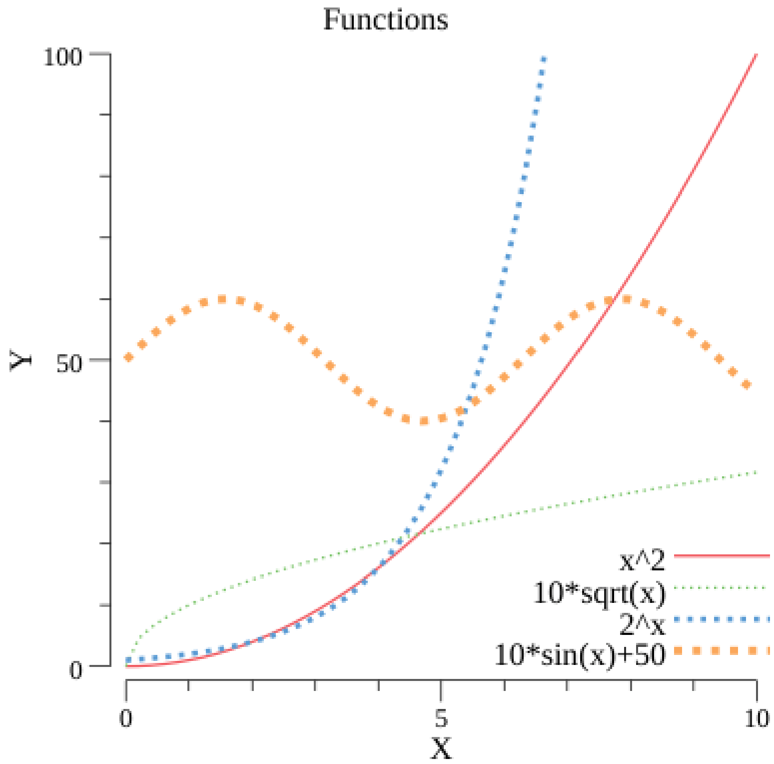
```
func main() {
  p, err := plot.New()
  if err != nil {
    log.Fatal(err)
  }
  p.Title.Text = "Functions"
  p.X.Label.Text = "X"
  p.Y.Label.Text = "Y"

  square := plotter.NewFunction(func(x float64) float64 { return x * x })
  square.Color = plotutil.Color(0)

  sqrt := plotter.NewFunction(func(x float64) float64 { return 10 * math.Sqrt(x)
})
  sqrt.Dashes = []vg.Length{vg.Points(1), vg.Points(2)}
  sqrt.Width = vg.Points(1)
  sqrt.Color = plotutil.Color(1)

  exp := plotter.NewFunction(func(x float64) float64 { return math.Pow(2, x) })
  exp.Dashes = []vg.Length{vg.Points(2), vg.Points(3)}
  exp.Width = vg.Points(2)
  exp.Color = plotutil.Color(2)

  sin := plotter.NewFunction(func(x float64) float64 { return 10*math.Sin(x) + 5
0 })
  sin.Dashes = []vg.Length{vg.Points(3), vg.Points(4)}
  sin.Width = vg.Points(3)
  sin.Color = plotutil.Color(3)
```

```
    p.Add(square, sqrt, exp, sin)
    p.Legend.Add("x^2", square)
    p.Legend.Add("10*sqrt(x)", sqrt)
    p.Legend.Add("2^x", exp)
    p.Legend.Add("10*sin(x)+50", sin)
    p.Legend.ThumbnailWidth = 0.5 * vg.Inch

    p.X.Min = 0
    p.X.Max = 10
    p.Y.Min = 0
    p.Y.Max = 100

    if err = p.Save(4*vg.Inch, 4*vg.Inch, "functions.png"); err != nil {
        log.Fatal(err)
    }
}
```

首先调用 `plotter.NewFunction()` 创建一个函数图像。它接受一个函数，单输入参数 `float64` ，单输出参数 `float64` ，故只能画出单自变量的函数图像。接着为函数图像设置了三个属性 `Dashes` （划线）、 `Width` （线宽）和 `Color` （颜色）。默认使用连续的线条来绘制函数，如图中的平方函数。可以通过设置 `Dashes` 让 `plot` 绘制不连续的线条， `Dashes` 接受两个长度值，第一个长度表示间隔距离，第二个长度表示连续线的长度。这里也使用到了 `plotutil.Color(i)` 依次使用前 4 个预定义的颜色。

创建画布、设置图例这些都与前面的相同。这里还通过 `p.X` 和 `p.Y` 的 `Min/Max` 属性限制了图像绘制的坐标范围。

运行程序生成图像：



气泡图

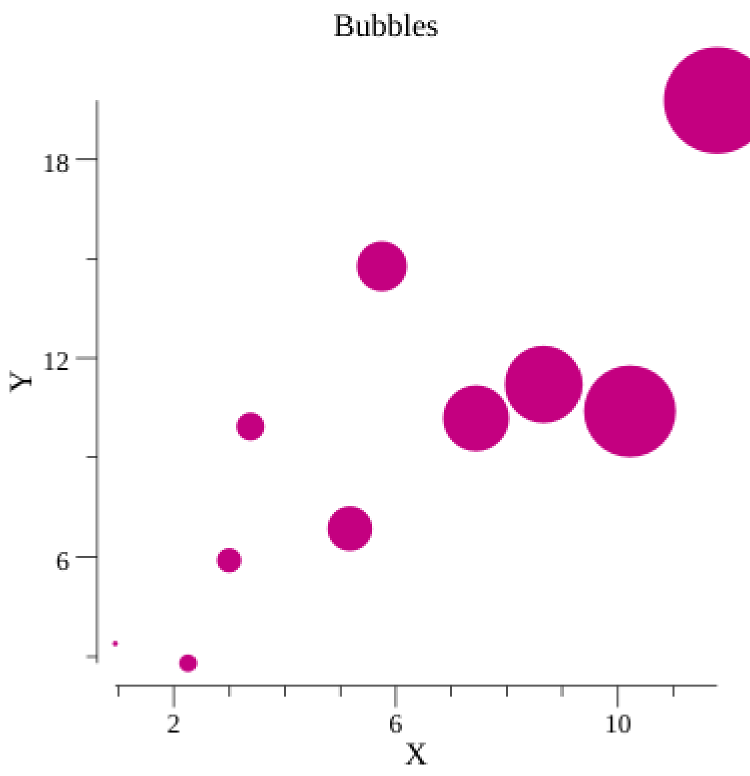
使用 `plot` 可以画出非常好看的气泡图：

```
func main() {
  n := 10
  bubbleData := randomTriples(n)

  minZ, maxZ := math.Inf(1), math.Inf(-1)
  for _, xyz := range bubbleData {
    if xyz.Z > maxZ {
      maxZ = xyz.Z
    }
    if xyz.Z < minZ {
      minZ = xyz.Z
    }
  }

  p, err := plot.New()
  if err != nil {
    log.Fatal(err)
  }
  p.Title.Text = "Bubbles"
  p.X.Label.Text = "X"
  p.Y.Label.Text = "Y"
```

```
	bs, err := plotter.NewScatter(bubbleData)
	if err != nil {
		log.Fatal(err)
	}
	bs.GlyphStyleFunc = func(i int) draw.GlyphStyle {
		c := color.RGBA{R: 196, B: 128, A: 255}
		var minRadius, maxRadius = vg.Points(1), vg.Points(20)
		rng := maxRadius - minRadius
		_, _, z := bubbleData.XYZ(i)
		d := (z - minZ) / (maxZ - minZ)
		r := vg.Length(d)*rng + minRadius
		return draw.GlyphStyle{Color: c, Radius: r, Shape: draw.CircleGlyph{}}
	}
	p.Add(bs)

	if err = p.Save(4*vg.Inch, 4*vg.Inch, "bubble.png"); err != nil {
		log.Fatal(err)
	}
}

func randomTriples(n int) plotter.XYZs {
	data := make(plotter.XYZs, n)
	for i := range data {
		if i == 0 {
			data[i].X = rand.Float64()
		} else {
			data[i].X = data[i-1].X + 2*rand.Float64()
		}
		data[i].Y = data[i].X + 10*rand.Float64()
		data[i].Z = data[i].X
	}

	return data
}
```

我们生成一组三维坐标点，调用 `plotter.NewScatter()` 生成散点图。我们设置了 `GlyphStyleFunc` 钩子函数，在绘制每个点之前都会调用它，它返回一个 `draw.GlyphStyle` 类型， `plot` 会根据返回的这个对象来绘制。我们的例子中，每次我们都返回一个表示圆形的 `draw.GlyphStyle` 对象，通过 `Z` 坐标与最大、最小坐标的比例映射到[`vg.Points(1)` ， `vg.Points(20)`]区间中得到半径。

生成的图像：



同样地，我们可以返回正方形的 `draw.GlyphStyle` 的对象来绘制“方形图”，只需要把钩子函数 `GlyphStyleFunc` 的返回语句做些修改：

```
return draw.GlyphStyle{Color: c, Radius: r, Shape: draw.SquareGlyph{}}
```

即可绘制“方形图”😄：

实际应用

下面我们应用之前文章中介绍的 `gopsutil` 和本文中的 `plot` 搭建一个网页，可以实时观察机器的 CPU 和内存占用：

```
func index(w http.ResponseWriter, r *http.Request) {
  t, err := template.ParseFiles("index.html")
  if err != nil {
    log.Fatal(err)
  }

  t.Execute(w, nil)
}

func image(w http.ResponseWriter, r *http.Request) {
  monitor.WriteTo(w)
```

```
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", index)
  mux.HandleFunc("/image", image)

  go monitor.Run()

  s := &http.Server{
    Addr:    ":8080",
    Handler: mux,
  }
  if err := s.ListenAndServe(); err != nil {
    log.Fatal(err)
  }
}
```

首先，我们编写了一个 HTTP 服务器，监听在 8080 端口。设置两个路由， `/` 显示主页， `/image` 调用 `Monitor` 的方法生成 CPU 和内存占用图返回。 `Monitor` 结构稍后会介绍。 `index.html` 的内容如下：

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Monitor</title>
</head>
<body>
  <img src="/image" alt="" id="img">
  <script>
    let img = document.querySelector("#img")
    setInterval(function () {
      img.src = "/image?s=" + Math.random()
    }, 500)
  </script>
</body>
</html>
```

页面比较简单，就显示了一张图片。然后在 JS 中启动一个 500ms 的定时器，每隔 500ms 就重新请求一次图片替换现有的图片。我在设置 `img.src` 属性时在后面添加了一个随机数，这是为了防止缓存导致得到的可能不是最新的图片。

下面看看 `Monitor` 的结构：

```
type Monitor struct {
  Mem       []float64
  CPU       []float64
  MaxRecord int
  Lock      sync.Mutex
}

func NewMonitor(max int) *Monitor {
  return &Monitor{
    MaxRecord: max,
  }
}

var monitor = NewMonitor(50)
```

这个结构中记录了最近的 50 条记录。每隔 500ms 会收集一次 CPU 和内存的占用情况，记录到 `CPU` 和 `Mem` 字段中：

```
func (m *Monitor) Collect() {
  mem, err := mem.VirtualMemory()
  if err != nil {
    log.Fatal(err)
  }

  cpu, err := cpu.Percent(500*time.Millisecond, false)
  if err != nil {
    log.Fatal(err)
  }

  m.Lock.Lock()
  defer m.Lock.Unlock()

  m.Mem = append(m.Mem, mem.UsedPercent)
  m.CPU = append(m.CPU, cpu[0])
}

func (m *Monitor) Run() {
  for {
    m.Collect()
    time.Sleep(500 * time.Millisecond)
  }
}
```

当 HTTP 请求 `/image` 路由时，根据目前已经收集到的 `CPU` 和 `Mem` 数据生成图片返回：

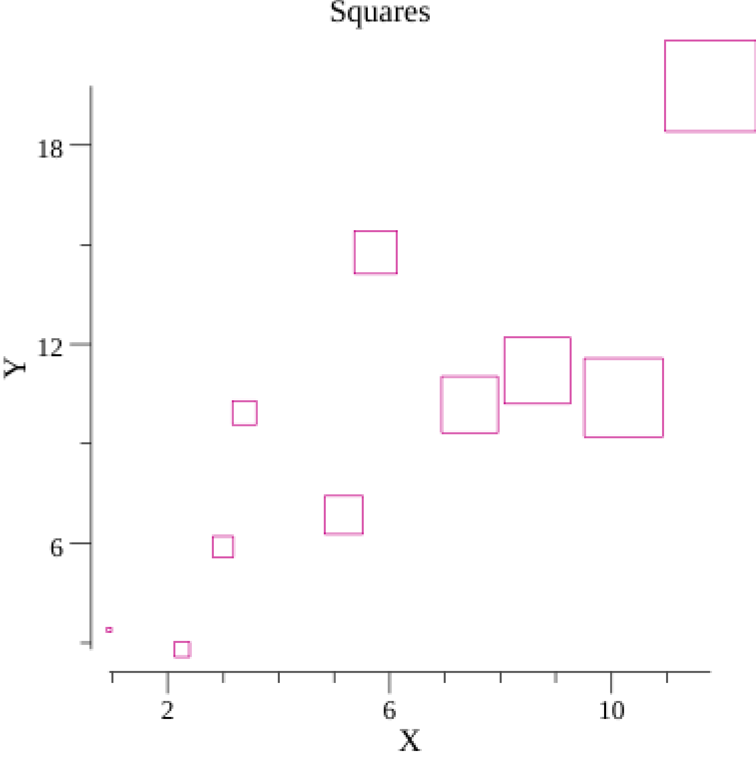
```
func (m *Monitor) WriteTo(w io.Writer) {
  m.Lock.Lock()
  defer m.Lock.Unlock()

  cpuData := make(plotter.XYs, len(m.CPU))
  for i, p := range m.CPU {
    cpuData[i].X = float64(i + 1)
    cpuData[i].Y = p
  }

  memData := make(plotter.XYs, len(m.Mem))
  for i, p := range m.Mem {
    memData[i].X = float64(i + 1)
    memData[i].Y = p
  }

  p, err := plot.New()
  if err != nil {
    log.Fatal(err)
  }

  cpuLine, err := plotter.NewLine(cpuData)
  if err != nil {
    log.Fatal(err)
  }
  cpuLine.Color = plotutil.Color(1)

  memLine, err := plotter.NewLine(memData)
  if err != nil {
    log.Fatal(err)
  }
  memLine.Color = plotutil.Color(2)

  p.Add(cpuLine, memLine)

  p.Legend.Add("cpu", cpuLine)
  p.Legend.Add("mem", memLine)

  p.X.Min = 0
  p.X.Max = float64(m.MaxRecord)
  p.Y.Min = 0
  p.Y.Max = 100
```

```
  wc, err := p.WriterTo(4*vg.Inch, 4*vg.Inch, "png")
  if err != nil {
    log.Fatal(err)
  }
  wc.WriteTo(w)
}
```

运行服务器：

```
$ go run main.go
```

打开浏览器，输入 `localhost:8080` ，观察图片变化：



总结

本文介绍了强大的绘图库 `plot` ，最后通过一个监控程序结尾。限于篇幅， `plot` 提供的多种绘图类型未能一一介绍。 `plot` 还支持 `svg/pdf` 等多种格式的保存。感兴趣的童鞋可自行研究。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. plot GitHub：https://github.com/gonum/plot
2. Example Plots: https://github.com/gonum/plot/wiki/Example-plots
3. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gentleman

简介

`gentleman` 是一个功能齐全、**插件**驱动的 HTTP 客户端。 `gentleman` 以扩展性为原则，可以基于内置的或第三方插件创建具有丰富特性的、可复用的 HTTP 客户端。相比标准库 `net/http` ， `gentleman` 更灵活、易用。

快速使用

先安装：

```
$ go get gopkg.in/h2non/gentleman.v2
```

后使用：

```
package main

import (
  "fmt"

  "gopkg.in/h2non/gentleman.v2"
)

func main() {
  cli := gentleman.New()

  cli.URL("https://dog.ceo")

  req := cli.Request()

  req.Path("/api/breeds/image/random")

  req.SetHeader("Client", "gentleman")

  res, err := req.Send()

  if err != nil {
    fmt.Printf("Request error: %vn", err)
    return
  }
```

```
  if !res.Ok {
    fmt.Printf("Invalid server response: %dn", res.StatusCode)
    return
  }

  fmt.Printf("Body: %s", res.String())
}
```

`gentleman` 目前有两个版本 `v1` 和 `v2` ， `v2` 已经稳定，推荐使用，示例中使用的就是 `v2` 。 `gentleman` 的使用遵循下面的流程：

- 调用 `gentleman.New()` 创建一个 HTTP 客户端 `cli` ，**此 `cli` 对象可复用**；
- 调用 `cli.URL()` 设置要请求的 URL 基础地址；
- 调用 `cli.Request()` 创建一个请求对象 `req` ；
- 调用 `req.Path()` 设置请求的路径，基于前面设置的 URL；
- 调用 `req.Header()` 设置请求首部（ `Header` ），上面代码设置首部 `Client` 为 `gentleman` ；
- 调用 `req.Send()` 发送请求，获取响应对象 `res` ；
- 对响应对象 `res` 进行处理。

上面的测试 API 是我从public-apis找的。 `public-apis` 是 GitHub 上一个收集各种开放 API 的仓库。本文后面部分的 API 也来自于这个仓库。从 `https://dog.ceo` 我们可以获取各种和**狗**相关的信息，上面请求的路径 `/api/breeds/image/random` 将返回一个随机品种的狗的图片。运行结果：

```
Body: {"message":"https://images.dog.ceo/breeds/malamute/n02110063_10567.jpg","status":"success"}
```

由于是随机的，每次运行结果可能都不相同， `status` 为 `success` 表示运行成功， `message` 对应的值为图片的 URL。感兴趣自己在浏览器中打开返回的 URL，我获取的图片如下：

插件

`gentleman` 中的特性很多都是通过**插件**来实现的。 `gentleman` 内置了很多常用的插件。如果要实现的特性无法通过内置插件来完成，还有第三方插件可供选择，当然还可以自定义插件！ `gentleman` 的插件都是存放在 `plugins` 子目录中的，下面介绍几个常用的插件。

`body`

客户端有时需要发送 JSON、XML 等格式的数据， `body` 插件可以很好地完成这个任务：

```
package main

import (
  "fmt"
  "gopkg.in/h2non/gentleman.v2"
  "gopkg.in/h2non/gentleman.v2/plugins/body"
)
```

```
func main() {
  cli := gentleman.New()
  cli.URL("http://httpbin.org/post")

  data := map[string]string{"foo": "bar"}
  cli.Use(body.JSON(data))

  req := cli.Request()
  req.Method("POST")

  res, err := req.Send()
  if err != nil {
    fmt.Printf("Request error: %s\n", err)
    return
  }

  if !res.Ok {
    fmt.Printf("Invalid server response: %d\n", res.StatusCode)
    return
  }

  fmt.Printf("Status: %d\n", res.StatusCode)
  fmt.Printf("Body: %s", res.String())
}
```

注意插件的导入方式： `import "gopkg.in/h2non/gentleman.v2/plugins/body"` 。

调用客户端对象 `cli` 或请求对象 `req` 的 `Use()` 方法使用插件。区别在于 `cli.Use()` 调用之后，所有通过该 `cli` 创建的请求对象都使用该插件， `req.Use()` 只对该请求生效，在本例中使用 `req.Use(body.JSON(data))` 也是可以的。上面使用 `body.JSON()` 插件，每次发送请求时，都将 `data` 转为 JSON 设置到请求体中，并设置相应的首部（ `Content-Type/Content-Length` ）。 `req.Method("POST")` 设置使用 POST 方法。本次请求使用的 URL `http://httpbin.org/post` 会回显请求的信息，看运行结果：

```
Status: 200
Body: {
  "args": {},
  "data": "{\"foo\":\"bar\"}\n",
  "files": {},
  "form": {},
  "headers": {
    "Accept-Encoding": "gzip",
    "Content-Length": "14",
    "Content-Type": "application/json",
```

```
    "Host": "httpbin.org",
    "User-Agent": "gentleman/2.0.4",
    "X-Amzn-Trace-Id": "Root=1-5e8dd0c7-ab423c10fb530deade846500"
  },
  "json": {
    "foo": "bar"
  },
  "origin": "124.77.254.163",
  "url": "http://httpbin.org/post"
}
```

发送 XML 格式与上面的非常类似：

```
type User struct {
  Name string `xml:"name"`
  Age  int    `xml:"age"`
}

func main() {
  cli := gentleman.New()
  cli.URL("http://httpbin.org/post")

  req := cli.Request()
  req.Method("POST")

  u := User{Name: "dj", Age: 18}
  req.Use(body.XML(u))
  // ...
}
```

后半部分一样的代码我就省略了，运行结果：

```
Status: 200
Body: {
  "args": {},
  "data": "<User><name>dj</name><age>18</age></User>",
  "files": {},
  "form": {},
  "headers": {
    "Accept-Encoding": "gzip",
    "Content-Length": "41",
    "Content-Type": "application/xml",
    "Host": "httpbin.org",
    "User-Agent": "gentleman/2.0.4",
    "X-Amzn-Trace-Id": "Root=1-5e8dd339-830dba04536ceef247156746"
```

```
  },
  "json": null,
  "origin": "222.64.16.70",
  "url": "http://httpbin.org/post"
}
```

`header`

`header` 插件用于在发送请求前添加一些通用的首部，如 APIKey；或者删除一些自动加上的首部，如 `User-Agent` 。一般 `header` 插件应用在 `cli` 对象上：

```go
package main

import (
  "fmt"
  "gopkg.in/h2non/gentleman.v2"
  "gopkg.in/h2non/gentleman.v2/plugins/headers"
)

func main() {
  cli := gentleman.New()
  cli.URL("https://api.thecatapi.com")

  cli.Use(headers.Set("x-api-key", "479ce48d-db30-46a4-b1a0-91ac4c1477b8"))
  cli.Use(headers.Del("User-Agent"))

  req := cli.Request()
  req.Path("/v1/breeds")
  res, err := req.Send()
  if err != nil {
    fmt.Printf("Request error: %s\n", err)
    return
  }
  if !res.Ok {
    fmt.Printf("Invalid server response: %d\n", res.StatusCode)
    return
  }

  fmt.Printf("Status: %d\n", res.StatusCode)
  fmt.Printf("Body: %s", res.String())
}
```

上面我们使用了 `https://api.thecatapi.com` ，这个 API 可以获取**猫**的品种信息，支持返回全部品种，搜索，分页等操作。API 使用需要申请 APIKey，我自己申请了一个 `479ce48d-

`db30-46a4-b1a0-91ac4c1477b8` 。 `thecatapi` 要求在请求首部中设置 `x-api-key` 为我们申请到的 APIKey。

`headers` 可以很方便的实现这个功能，只需要在 `cli` 对象上设置一次即可。另外， `gentleman` 会自动在请求中添加一个 `User-Agent` 首部，内容是 `gentleman` 的版本信息。细心的童鞋可能已经发现了，在上一节的输出中有 `User-Agent: gentleman/2.0.4` 这个首部。在本例中，我们使用 `header.Del()` 删除这个首部。

输出内容太多，我这里就不贴了。

`query`

HTTP 请求通常会在 URL 的 `?` 后带上查询字符串（ `query string` ）， `gentleman` 的内置插件 `query` 可以很好的管理这个信息。我们可以基于上面代码，给请求带上参数 `page` 和 `limit` 使之分页返回：

```go
package main

import (
  "fmt"

  "gopkg.in/h2non/gentleman.v2"
  "gopkg.in/h2non/gentleman.v2/plugins/headers"
  "gopkg.in/h2non/gentleman.v2/plugins/query"
)

func main() {
  cli := gentleman.New()
  cli.URL("https://api.thecatapi.com")

  cli.Use(headers.Set("x-api-key", "479ce48d-db30-46a4-b1a0-91ac4c1477b8"))
  cli.Use(query.Set("attach_breed", "beng"))
  cli.Use(query.Set("limit", "2"))
  cli.Use(headers.Del("User-Agent"))

  req := cli.Request()
  req.Path("/v1/breeds")
  req.Use(query.Set("page", "1"))
  res, err := req.Send()
  if err != nil {
    fmt.Printf("Request error: %s\n", err)
    return
  }
  if !res.Ok {
    fmt.Printf("Invalid server response: %d\n", res.StatusCode)
```

```
    return
  }

  fmt.Printf("Status: %d\n", res.StatusCode)
  fmt.Printf("Body: %s", res.String())
}
```

品种和每页显示数量最好还是在 `cli` 对象中设置，每个请求对象共用：

```
cli.Use(query.Set("attach_breed", "beng"))
cli.Use(query.Set("limit", "2"))
```

当前请求的页数在 `req` 对象上设置：

```
req.Use(query.Set("page", "1"))
```

其他的代码与上一个示例完全一样。除了设置 `query string` ，还可以通过 `query.Del()` 删除某个键值对。

`url`

路径参数有些时候很有用，因为我们在开发中时常会碰到相似的路径，只是中间某个部分不一样，例如 `/info/user/1` ， `/info/book/1` 等。重复写这些路径不仅很枯燥，而且容易出错。于是，偷懒的程序员发明了路径参数，形如 `/info/:class/1` ，我们可以传入参数 `user` 或 `book` 组成完整的路径。 `gentleman` 内置了插件 `url` 用来处理路径参数问题：

```
package main

import (
  "fmt"
  "os"

  "gopkg.in/h2non/gentleman.v2"
  "gopkg.in/h2non/gentleman.v2/plugins/headers"
  "gopkg.in/h2non/gentleman.v2/plugins/url"
)

func main() {
  cli := gentleman.New()
  cli.URL("https://api.thecatapi.com/")

  cli.Use(headers.Set("x-api-key", "479ce48d-db30-46a4-b1a0-91ac4c1477b8"))
  cli.Use(url.Path("/v1/:type"))
```

```
  for _, arg := range os.Args[1:] {
    req := cli.Request()
    req.Use(url.Param("type", arg))
    res, err := req.Send()
    if err != nil {
      fmt.Printf("Request error: %s\n", err)
      return
    }
    if !res.Ok {
      fmt.Printf("Invalid server response: %d\n", res.StatusCode)
      return
    }

    fmt.Printf("Status: %d\n", res.StatusCode)
    fmt.Printf("Body: %s\n", res.String())
  }
}
```

`thecatapi` 除了可以获取猫的品种，还有用户投票、各种分类信息。它们的请求路径都差不多，`/v1/breeds`、`/v1/votes`、`/v1/categories`。我们使用 `url` 简化程序编写。上面程序在客户端对象 `cli` 上使用插件 `url.Path("/v1/:type")`，调用 `url.Param("type", arg)` 用命令行中的参数分别替换 `type` 进行 HTTP 请求。运行程序：

```
$ go run main.go breeds votes categories
```

其他

`gentleman` 内置了将近 20 个插件，有身份认证相关的 `auth`、有 `cookies`、有压缩相关的 `compression`、有代理相关的 `proxy`、有重定向相关的 `redirect`、有超时相关的 `timeout`、有重试的 `retry`、有服务发现的 `consul` 等等等等。感兴趣可自行去探索。

自定义

如果内置的和第三方的插件都不能满足我们的需求，我们还可以自定义插件。自定义的插件需要实现下面的接口：

```
// src/gopkg.in/h2non/gentleman.v2/plugin/plugin.go
type Plugin interface {
  Enable()
  Disable()
  Disabled() bool
```

```
  Remove()
  Removed() bool
  Exec(string, *context.Context, context.Handler)
}
```

`Exec()` 方法在 HTTP 请求的各个生命周期都会调用，可以在请求前添加一些首部、删除查询字符串，响应返回后进行一些处理等。

通过实现 `Plugin` 接口的方式实现插件比较繁琐，且很多插件往往只关注生命周期的某个点，不用处理所有的生命周期事件。 `gentleman` 提供了一个 `Layer` 结构，可以注册某个生命周期的方法，同时提供 `NewRequestPlugin/NewResponsePlugin/NewErrorPlugin` 等便捷函数。

我们现在来实现一个插件，在请求之前输出一行信息，收到响应之后输出一行信息：

```
package main

import (
  "fmt"

  "gopkg.in/h2non/gentleman.v2"
  c "gopkg.in/h2non/gentleman.v2/context"
  "gopkg.in/h2non/gentleman.v2/plugin"
)

func main() {
  cli := gentleman.New()
  cli.URL("https://httpbin.org")

  cli.Use(plugin.NewRequestPlugin(func(ctx *c.Context, h c.Handler) {
    fmt.Println("request")

    h.Next(ctx)
  }))

  cli.Use(plugin.NewResponsePlugin(func(ctx *c.Context, h c.Handler) {
    fmt.Println("response")

    h.Next(ctx)
  }))

  req := cli.Request()
  req.Path("/headers")
  res, err := req.Send()
  if err != nil {
```

```go
    fmt.Printf("Request error: %s\n", err)
    return
  }
  if !res.Ok {
    fmt.Printf("Invalid server response: %d\n", res.StatusCode)
    return
  }

  fmt.Printf("Status: %d\n", res.StatusCode)
  fmt.Printf("Body: %s", res.String())
}
```

由于 `NewRequestPlugin/NewResponsePlugin` 这些便利函数，我们只需要实现一个类型为 `func(ctx *c.Context, h c.Handler)` 的函数即可，在 `ctx` 中有 `Request` 和 `Response` 等信息，可以在发起请求前对请求进行一些操作以及获得响应时对响应进行一些操作。上面只是简单地输出信息。

总结

使用 `gentleman` 可以实现灵活、便捷的 HTTP 客户端，它提供了丰富的插件，用起来吧~

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gentleman GitHub：https://github.com/h2non/gentleman
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gopsutil

简介

`gopsutil` 是 Python 工具库 `psutil` 的 Golang 移植版，可以帮助我们方便地获取各种系统和硬件信息。`gopsutil` 为我们屏蔽了各个系统之间的差异，具有非常强悍的可移植性。有了 `gopsutil` ，我们不再需要针对不同的系统使用 `syscall` 调用对应的系统方法。更棒的是 `gopsutil` 的实现中没有任何 `cgo` 的代码，使得交叉编译成为可能。

快速使用

先安装：

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

由于 `gopsutil` 库用到了 `golang.org/x/sys` ，后者在墙外，如果有类似下面的报错：

```
cannot find package "golang.org/x/sys/windows"
```

可使用下面的命令下载 `golang.org/x/sys` 在 GitHub 上的镜像：

```
$ git clone git@github.com:golang/sys.git $GOPATH/src/golang.org/x/sys
```

使用：

```
package main

import (
  "fmt"

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

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

  fmt.Printf("Total: %v, Available: %v, UsedPercent:%f%%\n", v.Total, v.Availabl
e, v.UsedPercent)
```

```
    fmt.Println(v)
}
```

`gopsutil` 将不同的功能划分到不同的子包中：

- `cpu` ：CPU 相关；
- `disk` ：磁盘相关；
- `docker` ：docker 相关；
- `host` ：主机相关；
- `mem` ：内存相关；
- `net` ：网络相关；
- `process` ：进程相关；
- `winservices` ：Windows 服务相关。

想要使用对应的功能，要导入对应的子包。例如，上面代码中，我们要获取内存信息，导入的是 `mem` 子包。 `mem.VirtualMemory()` 方法返回内存信息结构 `mem.VirtualMemoryStat` ，该结构有丰富的字段，我们最常使用的无外乎 `Total` （总内存）、 `Available` （可用内存）、 `Used` （已使用内存）和 `UsedPercent` （内存使用百分比）。 `mem.VirtualMemoryStat` 还实现了 `fmt.Stringer` 接口，以 JSON 格式返回内存信息。语句 `fmt.Println(v)` 会自动调用 `v.String()` ，将返回信息输出。程序输出：

```
Total: 8526921728, Available: 3768975360, UsedPercent:55.000000%
{"total":8526921728,"available":3768975360,"used":4757946368,"usedPercent":55,"free":0,"active":0,"inactive":0,"wired":0,"laundry":0,"buffers":0,"cached":0,"writeback":0,"dirty":0,"writebacktmp":0,"shared":0,"slab":0,"sreclaimable":0,"sunreclaim":0,"pagetables":0,"swapcached":0,"commitlimit":0,"committedas":0,"hightotal":0,"highfree":0,"lowtotal":0,"lowfree":0,"swaptotal":0,"swapfree":0,"mapped":0,"vmalloctotal":0,"vmallocused":0,"vmallocchunk":0,"hugepagestotal":0,"hugepagesfree":0,"hugepagesize":0}
```

单位为字节，我的电脑内存 8GB，当前使用百分比为 55%，可用内存 3768975360B（即 3.51GB）。

CPU

我们知道 CPU 的核数有两种，一种是物理核数，一种是逻辑核数。物理核数就是主板上实际有多少个 CPU，一个物理 CPU 上可以有多个核心，这些核心被称为逻辑核。 `gopsutil` 中 CPU 相关功能在 `cpu` 子包中， `cpu` 子包提供了获取物理和逻辑核数、CPU 使用率的接口：

- `Counts(logical bool)`：传入 `false`，返回物理核数，传入 `true`，返回逻辑核数；
- `Percent(interval time.Duration, percpu bool)`：表示获取 `interval` 时间间隔内的 CPU 使用率，`percpu` 为 `false` 时，获取总的 CPU 使用率，`percpu` 为 `true` 时，分别获取每个 CPU 的使用率，返回一个 `[]float64` 类型的值。

例如：

```
func main() {
  physicalCnt, _ := cpu.Counts(false)
  logicalCnt, _ := cpu.Counts(true)
  fmt.Printf("physical count:%d logical count:%d\n", physicalCnt, logicalCnt)

  totalPercent, _ := cpu.Percent(3*time.Second, false)
  perPercents, _ := cpu.Percent(3*time.Second, true)
  fmt.Printf("total percent:%v per percents:%v", totalPercent, perPercents)
}
```

上面代码获取物理核数和逻辑核数，并获取 3s 内的总 CPU 使用率和每个 CPU 各自的使用率，程序输出（注意每次运行输出可能都不相同）：

```
physical count:4 logical count:8
total percent:[30.729166666666668] per percents:[32.64248704663213 26.94300518134715 44.559585492227974 23.958333333333336 36.787564766839374 20.3125 38.54166666666667 28.125]
```

详细信息

调用 `cpu.Info()` 可获取 CPU 的详细信息，返回 `[]cpu.InfoStat`：

```
func main() {
  infos, _ := cpu.Info()
  for _, info := range infos {
    data, _ := json.MarshalIndent(info, "", " ")
    fmt.Print(string(data))
  }
}
```

为了方便查看，我使用 JSON 输出结果：

```
{
"cpu": 0,
"vendorId": "GenuineIntel",
"family": "198",
"model": "",
"stepping": 0,
"physicalId": "BFEBFBFF000906E9",
"coreId": "",
"cores": 8,
"modelName": "Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz",
"mhz": 3601,
"cacheSize": 0,
"flags": [],
"microcode": ""
}
```

由结果可以看出，CPU 是 Intel 的 i7-7700 系列，频率 3.60GHz。上面是我在 Windows 上运行的返回结果，内部使用了 `github.com/StackExchange/wmi` 库。在 Linux 下每个逻辑 CPU 都会返回一个 `InfoStat` 结构。

时间占用

调用 `cpu.Times(percpu bool)` 可以获取从开机算起，总 CPU 和 每个单独的 CPU 时间占用情况。传入 `percpu=false` 返回总的，传入 `percpu=true` 返回单个的。每个 CPU 时间占用情况是一个 `TimeStat` 结构：

```
// src/github.com/shirou/gopsutil/cpu/cpu.go
type TimesStat struct {
  CPU       string  `json:"cpu"`
  User      float64 `json:"user"`
  System    float64 `json:"system"`
  Idle      float64 `json:"idle"`
  Nice      float64 `json:"nice"`
  Iowait    float64 `json:"iowait"`
  Irq       float64 `json:"irq"`
  Softirq   float64 `json:"softirq"`
  Steal     float64 `json:"steal"`
  Guest     float64 `json:"guest"`
  GuestNice float64 `json:"guestNice"`
}
```

- `CPU`：CPU 标识，如果是总的，该字段为 `cpu-total`，否则为 `cpu0`、`cpu1`...；

- `User` ：用户时间占用（用户态）；
- `System` ：系统时间占用（内核态）；
- `Idle` ：空闲时间；
- …

例如：

```
func main() {
  infos, _ := cpu.Times(true)
  for _, info := range infos {
    data, _ := json.MarshalIndent(info, "", " ")
    fmt.Print(string(data))
  }
}
```

为了方便查看，我用 JSON 输出结果，下面是其中一个输出：

```
{
 "cpu": "cpu0",
 "user": 674.46875,
 "system": 1184.984375,
 "idle": 7497.1875,
 "nice": 0,
 "iowait": 0,
 "irq": 75.578125,
 "softirq": 0,
 "steal": 0,
 "guest": 0,
 "guestNice": 0
}
```

磁盘

子包 `disk` 用于获取磁盘信息。 `disk` 可获取 IO 统计、分区和使用率信息。下面依次介绍。

IO 统计

调用 `disk.IOCounters()` 函数，返回的 IO 统计信息用 `map[string]IOCountersStat` 类型表示。每个分区一个结构，键为分区名，值为统计信息。这里摘取统计结构的部分字段，主要有读写的次数、字节数和时间：

```
// src/github.com/shirou/gopsutil/disk/disk.go
type IOCountersStat struct {
  ReadCount        uint64 `json:"readCount"`
  MergedReadCount  uint64 `json:"mergedReadCount"`
  WriteCount       uint64 `json:"writeCount"`
  MergedWriteCount uint64 `json:"mergedWriteCount"`
  ReadBytes        uint64 `json:"readBytes"`
  WriteBytes       uint64 `json:"writeBytes"`
  ReadTime         uint64 `json:"readTime"`
  WriteTime        uint64 `json:"writeTime"`
  // ...
}
```

例如：

```
func main() {
  mapStat, _ := disk.IOCounters()
  for name, stat := range mapStat {
    fmt.Println(name)
    data, _ := json.MarshalIndent(stat, "", "  ")
    fmt.Println(string(data))
  }
}
```

输出包括所有分区，我这里只展示一个：

```
C:
{
  "readCount": 184372,
  "mergedReadCount": 0,
  "writeCount": 42252,
  "mergedWriteCount": 0,
  "readBytes": 5205152768,
  "writeBytes": 701583872,
  "readTime": 333,
  "writeTime": 27,
  "iopsInProgress": 0,
  "ioTime": 0,
  "weightedIO": 0,
  "name": "C:",
  "serialNumber": "",
  "label": ""
}
```

注意，`disk.IOCounters()` 可传入可变数量的字符串参数用于标识分区，**此参数在 Windows 上无效**。

分区

调用 `disk.PartitionStat(all bool)` 函数，返回分区信息。如果 `all = false`，只返回实际的物理分区（包括硬盘、CD-ROM、USB），忽略其它的虚拟分区。如果 `all = true` 则返回所有的分区。返回类型为 `[]PartitionStat`，每个分区对应一个 `PartitionStat` 结构：

```
// src/github.com/shirou/gopsutil/disk/
type PartitionStat struct {
  Device     string `json:"device"`
  Mountpoint string `json:"mountpoint"`
  Fstype     string `json:"fstype"`
  Opts       string `json:"opts"`
}
```

- `Device`：分区标识，在 Windows 上即为 `C:` 这类格式；
- `Mountpoint`：挂载点，即该分区的文件路径起始位置；
- `Fstype`：文件系统类型，Windows 常用的有 FAT、NTFS 等，Linux 有 ext、ext2、ext3等；
- `Opts`：选项，与系统相关。

例如：

```
func main() {
  infos, _ := disk.Partitions(false)
  for _, info := range infos {
    data, _ := json.MarshalIndent(info, "", "  ")
    fmt.Println(string(data))
  }
}
```

我的 Windows 机器输出（只展示第一个分区）：

```
{
  "device": "C:",
  "mountpoint": "C:",
  "fstype": "NTFS",
  "opts": "rw.compress"
}
```

由上面的输出可知，我的第一个分区为 `C:` ，文件系统类型为 `NTFS` 。

使用率

调用 `disk.Usage(path string)` 即可获得路径 `path` 所在磁盘的使用情况，返回一个 `UsageStat` 结构：

```
// src/github.com/shirou/gopsutil/disk.go
type UsageStat struct {
  Path              string  `json:"path"`
  Fstype            string  `json:"fstype"`
  Total             uint64  `json:"total"`
  Free              uint64  `json:"free"`
  Used              uint64  `json:"used"`
  UsedPercent       float64 `json:"usedPercent"`
  InodesTotal       uint64  `json:"inodesTotal"`
  InodesUsed        uint64  `json:"inodesUsed"`
  InodesFree        uint64  `json:"inodesFree"`
  InodesUsedPercent float64 `json:"inodesUsedPercent"`
}
```

- `Path` ：路径，传入的参数；
- `Fstype` ：文件系统类型；
- `Total` ：该分区总容量；
- `Free` ：空闲容量；
- `Used` ：已使用的容量；
- `UsedPercent` ：使用百分比。

例如：

```
func main() {
  info, _ := disk.Usage("D:/code/golang")
  data, _ := json.MarshalIndent(info, "", "  ")
  fmt.Println(string(data))
}
```

由于返回的是磁盘的使用情况，所以路径 `D:/code/golang` 和 `D:` 返回同样的结果，只是结构中的 `Path` 字段不同而已。程序输出：

```
{
  "path": "D:/code/golang",
```

```
  "fstype": "",
  "total": 475779821568,
  "free": 385225650176,
  "used": 90554171392,
  "usedPercent": 19.032789388496106,
  "inodesTotal": 0,
  "inodesUsed": 0,
  "inodesFree": 0,
  "inodesUsedPercent": 0
}
```

主机

子包 `host` 可以获取主机相关信息，如开机时间、内核版本号、平台信息等等。

开机时间

`host.BootTime()` 返回主机开机时间的时间戳：

```
func main() {
  timestamp, _ := host.BootTime()
  t := time.Unix(int64(timestamp), 0)
  fmt.Println(t.Local().Format("2006-01-02 15:04:05"))
}
```

上面先获取开机时间，然后通过 `time.Unix()` 将其转为 `time.Time` 类型，最后输出 `2006-01-02 15:04:05` 格式的时间：

```
2020-04-06 20:25:32
```

内核版本和平台信息

```
func main() {
  version, _ := host.KernelVersion()
  fmt.Println(version)

  platform, family, version, _ := host.PlatformInformation()
  fmt.Println("platform:", platform)
  fmt.Println("family:", family,
  fmt.Println("version:", version)
}
```

在我的 Win10 上运行输出：

```
10.0.18362 Build 18362
platform: Microsoft Windows 10 Pro
family: Standalone Workstation
version: 10.0.18362 Build 18362
```

终端用户

`host.Users()` 返回终端连接上来的用户信息，每个用户一个 `UserStat` 结构：

```
// src/github.com/shirou/gopsutil/host/host.go
type UserStat struct {
  User     string `json:"user"`
  Terminal string `json:"terminal"`
  Host     string `json:"host"`
  Started  int    `json:"started"`
}
```

字段一目了然，看示例：

```
func main() {
  users, _ := host.Users()
  for _, user := range users {
    data, _ := json.MarshalIndent(user, "", " ")
    fmt.Println(string(data))
  }
}
```

内存

在快速开始中，我们演示了如何使用 `mem.VirtualMemory()` 来获取内存信息。该函数返回的只是物理内存信息。我们还可以使用 `mem.SwapMemory()` 获取交换内存的信息，信息存储在结构 `SwapMemoryStat` 中：

```
// src/github.com/shirou/gopsutil/mem/
type SwapMemoryStat struct {
  Total       uint64  `json:"total"`
  Used        uint64  `json:"used"`
  Free        uint64  `json:"free"`
  UsedPercent float64 `json:"usedPercent"`
  Sin         uint64  `json:"sin"`
```

```
    Sout        uint64  `json:"sout"`
    PgIn        uint64  `json:"pgin"`
    PgOut       uint64  `json:"pgout"`
    PgFault     uint64  `json:"pgfault"`
}
```

字段含义很容易理解， `PgIn/PgOut/PgFault` 这三个字段我们重点介绍一下。交换内存是以**页**为单位的，如果出现缺页错误(`page fault`)，操作系统会将磁盘中的某些页载入内存，同时会根据特定的机制淘汰一些内存中的页。 `PgIn` 表征载入页数， `PgOut` 淘汰页数， `PgFault` 缺页错误数。

例如：

```go
func main() {
    swapMemory, _ := mem.SwapMemory()
    data, _ := json.MarshalIndent(swapMemory, "", " ")
    fmt.Println(string(data))
}
```

进程

`process` 可用于获取系统当前运行的进程信息，创建新进程，对进程进行一些操作等。

```go
func main() {
    var rootProcess *process.Process
    processes, _ := process.Processes()
    for _, p := range processes {
        if p.Pid == 0 {
            rootProcess = p
            break
        }
    }

    fmt.Println(rootProcess)

    fmt.Println("children:")
    children, _ := rootProcess.Children()
    for _, p := range children {
        fmt.Println(p)
    }
}
```

先调用 `process.Processes()` 获取当前系统中运行的所有进程，然后找到 `Pid` 为 0 的进程，即操作系统的第一个进程，最后调用 `Children()` 返回其子进程。还有很多方法可获取进程信息，感兴趣可查看文档了解~

Windows 服务

`winservices` 子包可以获取 Windows 系统中的服务信息，内部使用了 `golang.org/x/sys` 包。在 `winservices` 中，一个服务对应一个 `Service` 结构：

```
// src/github.com/shirou/gopsutil/winservices/winservices.go
type Service struct {
  Name   string
  Config mgr.Config
  Status ServiceStatus
  // contains filtered or unexported fields
}
```

`mgr.Config` 为包 `golang.org/x/sys` 中的结构，该结构详细记录了服务类型、启动类型（自动/手动）、二进制文件路径等信息：

```
// src/golang.org/x/sys/windows/svc/mgr/config.go
type Config struct {
  ServiceType      uint32
  StartType        uint32
  ErrorControl     uint32
  BinaryPathName   string
  LoadOrderGroup   string
  TagId            uint32
  Dependencies     []string
  ServiceStartName string
  DisplayName      string
  Password         string
  Description      string
  SidType          uint32
  DelayedAutoStart bool
}
```

`ServiceStatus` 结构记录了服务的状态：

```
// src/github.com/shirou/gopsutil/winservices/winservices.go
type ServiceStatus struct {
  State         svc.State
```

```
  Accepts       svc.Accepted
  Pid           uint32
  Win32ExitCode uint32
}
```

- `State` ：为服务状态，有已停止、运行、暂停等；
- `Accepts` ：表示服务接收哪些操作，有暂停、继续、会话切换等；
- `Pid` ：进程 ID；
- `Win32ExitCode` ：应用程序退出状态码。

下面程序中，我将系统中所有服务的名称、二进制文件路径和状态输出到控制台：

```
func main() {
  services, _ := winservices.ListServices()

  for _, service := range services {
    newservice, _ := winservices.NewService(service.Name)
    newservice.GetServiceDetail()
    fmt.Println("Name:", newservice.Name, "Binary Path:", newservice.Config.BinaryPathName, "State: ", newservice.Status.State)
  }
}
```

注意，调用 `winservices.ListServices()` 返回的 `Service` 对象信息是不全的，我们通过 `NewService()` 以该服务名称创建一个服务，然后调用 `GetServiceDetail()` 方法获取该服务的详细信息。不能直接通过 `service.GetServiceDetail()` 来调用，因为 `ListService()` 返回的对象缺少必要的系统资源句柄（为了节约资源），调用 `GetServiceDetail()` 方法会 `panic` ！！！

错误和超时

由于大部分函数都涉及到底层的系统调用，所以发生错误和超时是在所难免的。几乎所有的接口都有两个返回值，第二个作为错误。在前面的例子中，我们为了简化代码都忽略了错误，在实际使用中，建议对错误进行处理。

另外，大部分接口都是一对，一个不带 `context.Context` 类型的参数，另一个带有该类型参数，用于做上下文控制。在内部调用发生错误或超时后能及时处理，避免长时间等待返回。实际上，不带 `context.Context` 参数的函数内部都是以 `context.Background()` 为参数调用带有 `context.Context` 的函数的：

```
// src/github.com/shirou/gopsutil/cpu_windows.go
func Times(percpu bool) ([]TimesStat, error) {
```

```
  return TimesWithContext(context.Background(), percpu)
}

func TimesWithContext(ctx context.Context, percpu bool) ([]TimesStat, error) {
  // ...
}
```

总结

`gopsutil` 库方便了我们获取本机的信息，且很好地处理了各个系统间的兼容问题，提供了一致的接口。还有几个子包例如 `net/docker` 限于篇幅没有介绍，感兴趣的童鞋可自行探索。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gopsutil GitHub：https://github.com/shirou/gopsutil
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

validator

简介

今天我们来介绍一个非常实用的库—— `validator` 。 `validator` 用于对数据进行校验。在 Web 开发中，对用户传过来的数据我们都需要进行严格校验，防止用户的恶意请求。例如日期格式，用户年龄，性别等必须是正常的值，不能随意设置。

快速使用

先安装：

```
$ go get gopkg.in/go-playground/validator.v10
```

后使用：

```
package main

import (
  "fmt"

  "gopkg.in/go-playground/validator.v10"
)

type User struct {
  Name string `validate:"min=6,max=10"`
  Age  int    `validate:"min=1,max=100"`
}

func main() {
  validate := validator.New()

  u1 := User{Name: "lidajun", Age: 18}
  err := validate.Struct(u1)
  fmt.Println(err)

  u2 := User{Name: "dj", Age: 101}
  err = validate.Struct(u2)
  fmt.Println(err)
}
```

`validator` 在结构体标签（ `struct tag` ）中定义字段的**约束**。使用 `validator` 验证数据之前，我们需要调用 `validator.New()` 创建一个**验证器**，这个验证器可以指定选项、添加自定义约束，然后通过调用它的 `Struct()` 方法来验证各种结构对象的字段是否符合定义的约束。

在上面代码中，我们定义了一个结构体 `User` ， `User` 有名称 `Name` 字段和年龄 `Age` 字段。通过 `min` 和 `max` 约束，我们设置 `Name` 的字符串长度为 `[6,10]` 之间， `Age` 的范围为 `[1,100]` 。

第一个对象 `Name` 和 `Age` 字段都满足约束，故 `Struct()` 方法返回 `nil` 错误。第二个对象的 `Name` 字段值为 `dj` ，长度 2，小于最小值 `min` ， `Age` 字段值为 101，大于最大值 `max` ，故返回错误：

```
<nil>
Key: 'User.Name' Error:Field validation for 'Name' failed on the 'min' tag
Key: 'User.Age' Error:Field validation for 'Age' failed on the 'max' tag
```

错误信息比较好理解， `User.Name` 违反了 `min` 约束， `User.Age` 违反了 `max` 约束，一眼就能看出问题所在。

注意：

- `validator` 已经更新迭代了很多版本，当前最新的版本是 `v10` ，各个版本之间有一些差异，大家平时在使用和阅读代码时要注意区分。我这里使用最新的版本 `v10` 作为演示版本；
- 字符串长度和数值的范围都可以通过 `min` 和 `max` 来约束。

约束

`validator` 提供了非常丰富的约束可供使用，下面依次来介绍。

范围约束

我们上面已经看到了使用 `min` 和 `max` 来约束字符串的长度或数值的范围，下面再介绍其它的范围约束。范围约束的字段类型有以下几种：

- 对于数值，则约束其值；
- 对于字符串，则约束其长度；
- 对于切片、数组和 `map` ，则约束其长度。

下面如未特殊说明，则是根据上面各个类型对应的值与参数值比较。

- `len`：等于参数值，例如`len=10`；
- `max`：小于等于参数值，例如`max=10`；
- `min`：大于等于参数值，例如`min=10`；
- `eq`：等于参数值，注意与`len`不同。对于字符串，`eq`约束字符串本身的值，而`len`约束字符串长度。例如`eq=10`；
- `ne`：不等于参数值，例如`ne=10`；
- `gt`：大于参数值，例如`gt=10`；
- `gte`：大于等于参数值，例如`gte=10`；
- `lt`：小于参数值，例如`lt=10`；
- `lte`：小于等于参数值，例如`lte=10`；
- `oneof`：只能是列举出的值其中一个，这些值必须是数值或字符串，以空格分隔，如果字符串中有空格，将字符串用单引号包围，例如`oneof=red green`。

大部分还是比较直观的，我们通过一个例子看看其中几个约束如何使用：

```
type User struct {
  Name    string    `validate:"ne=admin"`
  Age     int       `validate:"gte=18"`
  Sex     string    `validate:"oneof=male female"`
  RegTime time.Time `validate:"lte"`
}

func main() {
  validate := validator.New()

  u1 := User{Name: "dj", Age: 18, Sex: "male", RegTime: time.Now().UTC()}
  err := validate.Struct(u1)
  if err != nil {
    fmt.Println(err)
  }

  u2 := User{Name: "admin", Age: 15, Sex: "none", RegTime: time.Now().UTC().Add(
1 * time.Hour)}
  err = validate.Struct(u2)
  if err != nil {
    fmt.Println(err)
  }
}
```

上面例子中，我们定义了`User`对象，为它的 4 个字段分别设置了约束：

- `Name`：字符串不能是`admin`；
- `Age`：必须大于等于 18，**未成年人禁止入内**；
- `Sex`：性别必须是`male`和`female`其中一个；
- `RegTime`：注册时间必须小于当前的 UTC 时间，注意如果字段类型是`time.Time`，使用`gt/gte/lt/lte`等约束时不用指定参数值，默认与当前的 UTC 时间比较。

同样地，第一个对象的字段都是合法的，校验通过。第二个对象的 4 个字段都非法，通过输出信息很好定错误位置：

```
Key: 'User.Name' Error:Field validation for 'Name' failed on the 'ne' tag
Key: 'User.Age' Error:Field validation for 'Age' failed on the 'gte' tag
Key: 'User.Sex' Error:Field validation for 'Sex' failed on the 'oneof' tag
Key: 'User.RegTime' Error:Field validation for 'RegTime' failed on the 'lte' tag
```

跨字段约束

`validator`允许定义跨字段的约束，即该字段与其他字段之间的关系。这种约束实际上分为两种，一种是参数字段就是同一个结构中的平级字段，另一种是参数字段为结构中其他字段的字段。约束语法很简单，要想使用上面的约束语义，只需要稍微修改一下。例如**相等约束**（`eq`），如果是约束同一个结构中的字段，则在后面添加一个`field`，使用`eqfield`定义字段间的相等约束。如果是更深层次的字段，在`field`之前还需要加上`cs`（可以理解为`cross-struct`），`eq`就变为`eqcsfield`。它们的参数值都是需要比较的字段名，内层的还需要加上字段的类型：

```
eqfield=ConfirmPassword
eqcsfield=InnerStructField.Field
```

看示例：

```
type RegisterForm struct {
  Name      string `validate:"min=2"`
  Age       int    `validate:"min=18"`
  Password  string `validate:"min=10"`
  Password2 string `validate:"eqfield=Password"`
}

func main() {
  validate := validator.New()

  f1 := RegisterForm{
    Name:     "dj",
    Age:      18,
```

```
    Password:  "1234567890",
    Password2: "1234567890",
  }
  err := validate.Struct(f1)
  if err != nil {
    fmt.Println(err)
  }

  f2 := RegisterForm{
    Name:      "dj",
    Age:       18,
    Password:  "1234567890",
    Password2: "123",
  }
  err = validate.Struct(f2)
  if err != nil {
    fmt.Println(err)
  }
}
```

我们定义了一个简单的注册表单结构，使用 `eqfield` 约束其两次输入的密码必须相等。第一个对象满足约束，第二个对象两次密码明显不等。程序输出：

```
Key: 'RegisterForm.Password2' Error:Field validation for 'Password2' failed on the 'eqfield' tag
```

字符串

`validator` 中关于字符串的约束有很多，这里介绍几个：

- `contains=`：包含参数子串，例如 `contains=email`；
- `containsany`：包含参数中任意的 UNICODE 字符，例如 `containsany=abcd`；
- `containsrune`：包含参数表示的 rune 字符，例如 `containsrune=☻`；
- `excludes`：不包含参数子串，例如 `excludes=email`；
- `excludesall`：不包含参数中任意的 UNICODE 字符，例如 `excludesall=abcd`；
- `excludesrune`：不包含参数表示的 rune 字符，`excludesrune=☻`；
- `startswith`：以参数子串为前缀，例如 `startswith=hello`；
- `endswith`：以参数子串为后缀，例如 `endswith=bye`。

看示例：

```
type User struct {
  Name string `validate:"containsrune=☻"`
  Age  int    `validate:"min=18"`
}

func main() {
  validate := validator.New()

  u1 := User{"d☻j", 18}
  err := validate.Struct(u1)
  if err != nil {
    fmt.Println(err)
  }

  u2 := User{"dj", 18}
  err = validate.Struct(u2)
  if err != nil {
    fmt.Println(err)
  }
}
```

限制 `Name` 字段必须包含 UNICODE 字符 `☻` 。

唯一性

使用 `unqiue` 来指定唯一性约束，对不同类型的处理如下：

- 对于数组和切片， `unique` 约束没有重复的元素；
- 对于 `map` ， `unique` 约束没有重复的**值**；
- 对于元素类型为结构体的切片， `unique` 约束结构体对象的某个字段不重复，通过 `unqiue=field` 指定这个字段名。

例如：

```
type User struct {
  Name    string   `validate:"min=2"`
  Age     int      `validate:"min=18"`
  Hobbies []string `validate:"unique"`
  Friends []User   `validate:"unique=Name"`
}

func main() {
  validate := validator.New()
```

```
  f1 := User{
    Name: "dj2",
    Age:  18,
  }
  f2 := User{
    Name: "dj3",
    Age:  18,
  }

  u1 := User{
    Name:    "dj",
    Age:     18,
    Hobbies: []string{"pingpong", "chess", "programming"},
    Friends: []User{f1, f2},
  }
  err := validate.Struct(u1)
  if err != nil {
    fmt.Println(err)
  }

  u2 := User{
    Name:    "dj",
    Age:     18,
    Hobbies: []string{"programming", "programming"},
    Friends: []User{f1, f1},
  }
  err = validate.Struct(u2)
  if err != nil {
    fmt.Println(err)
  }
}
```

我们限制爱好 `Hobbies` 中不能有重复元素，好友 `Friends` 的各个元素不能有同样的名字 `Name` 。第一个对象满足约束，第二个对象的 `Hobbies` 字段包含了重复的 `"programming"` ， `Friends` 字段中两个元素的 `Name` 字段都是 `dj2` 。程序输出：

```
Key: 'User.Hobbies' Error:Field validation for 'Hobbies' failed on the 'unique' tag
Key: 'User.Friends' Error:Field validation for 'Friends' failed on the 'unique' tag
```

邮件

通过 `email` 限制字段必须是邮件格式：

```
type User struct {
  Name  string `validate:"min=2"`
  Age   int    `validate:"min=18"`
  Email string `validate:"email"`
}

func main() {
  validate := validator.New()

  u1 := User{
    Name:  "dj",
    Age:   18,
    Email: "dj@example.com",
  }
  err := validate.Struct(u1)
  if err != nil {
    fmt.Println(err)
  }

  u2 := User{
    Name:  "dj",
    Age:   18,
    Email: "djexample.com",
  }
  err = validate.Struct(u2)
  if err != nil {
    fmt.Println(err)
  }
}
```

上面我们约束 `Email` 字段必须是邮件的格式，第一个对象满足约束，第二个对象不满足，程序输出：

```
Key: 'User.Email' Error:Field validation for 'Email' failed on the 'email' tag
```

特殊

有一些比较特殊的约束：

- `-`：跳过该字段，不检验；
- `|`：使用多个约束，只需要满足其中一个，例如 `rgb|rgba`；

- `required` ：字段必须设置，不能为默认值；
- `omitempty` ：如果字段未设置，则忽略它。

其他

`validator` 提供了大量的、各个方面的、丰富的约束，如 `ASCII/UNICODE` 字母、数字、十六进制、十六进制颜色值、大小写、RBG 颜色值，HSL 颜色值、HSLA 颜色值、JSON 格式、文件路径、URL、base64 编码串、ip 地址、ipv4、ipv6、UUID、经纬度等等等等等等等等等等等。限于篇幅这里就不一一介绍了。感兴趣自行去文档中挖掘。

`VarWithValue`方法

在一些很简单的情况下，我们仅仅想对两个变量进行比较，如果每次都要先定义结构和 `tag` 就太繁琐了。 `validator` 提供了 `VarWithValue()` 方法，我们只需要传入要验证的两个变量和约束即可

```
func main() {
  name1 := "dj"
  name2 := "dj2"

  validate := validator.New()
  fmt.Println(validate.VarWithValue(name1, name2, "eqfield"))

  fmt.Println(validate.VarWithValue(name1, name2, "nefield"))
}
```

自定义约束

除了使用 `validator` 提供的约束外，还可以定义自己的约束。例如现在有个奇葩的需求，产品同学要求用户必须使用回文串作为用户名，我们可以自定义这个约束：

```
type RegisterForm struct {
  Name string `validate:"palindrome"`
  Age  int    `validate:"min=18"`
}

func reverseString(s string) string {
  runes := []rune(s)
  for from, to := 0, len(runes)-1; from < to; from, to = from+1, to-1 {
    runes[from], runes[to] = runes[to], runes[from]
  }
```

```
    return string(runes)
}

func CheckPalindrome(fl validator.FieldLevel) bool {
    value := fl.Field().String()
    return value == reverseString(value)
}

func main() {
    validate := validator.New()
    validate.RegisterValidation("palindrome", CheckPalindrome)

    f1 := RegisterForm{
        Name: "djd",
        Age:  18,
    }
    err := validate.Struct(f1)
    if err != nil {
        fmt.Println(err)
    }

    f2 := RegisterForm{
        Name: "dj",
        Age:  18,
    }
    err = validate.Struct(f2)
    if err != nil {
        fmt.Println(err)
    }
}
```

首先定义一个类型为 `func (validator.FieldLevel) bool` 的函数检查约束是否满足，可以通过 `FieldLevel` 取出要检查的字段的信息。然后，调用验证器的 `RegisterValidation()` 方法将该约束注册到指定的名字上。最后我们就可以在结构体中使用该约束。上面程序中，第二个对象不满足约束 `palindrome` ，输出：

```
Key: 'RegisterForm.Name' Error:Field validation for 'Name' failed on the 'palindrome' tag
```

错误处理

在上面的例子中，校验失败时我们仅仅只是输出返回的错误。其实，我们可以进行更精准的处理。 `validator` 返回的错误实际上只有两种，一种是参数错误，一种是校验错误。参数错误时，返回 `InvalidValidationError` 类型；校验错误时返回 `ValidationErrors` ，它

们都实现了 `error` 接口。而且 `ValidationErrors` 是一个错误切片，它保存了每个字段违反的每个约束信息：

```
// src/gopkg.in/validator.v10/errors.go
type InvalidValidationError struct {
  Type reflect.Type
}

// Error returns InvalidValidationError message
func (e *InvalidValidationError) Error() string {
  if e.Type == nil {
    return "validator: (nil)"
  }

  return "validator: (nil " + e.Type.String() + ")"
}

type ValidationErrors []FieldError

func (ve ValidationErrors) Error() string {
  buff := bytes.NewBufferString("")
  var fe *fieldError

  for i := 0; i < len(ve); i++ {
    fe = ve[i].(*fieldError)
    buff.WriteString(fe.Error())
    buff.WriteString("\n")
  }
  return strings.TrimSpace(buff.String())
}
```

所以 `validator` 校验返回的结果只有 3 种情况：

- `nil`：没有错误；
- `InvalidValidationError`：输入参数错误；
- `ValidationErrors`：字段违反约束。

我们可以在程序中判断 `err != nil` 时，依次将 `err` 转换为 `InvalidValidationError` 和 `ValidationErrors` 以获取更详细的信息：

```
func processErr(err error) {
  if err == nil {
    return
  }
```

```
    invalid, ok := err.(*validator.InvalidValidationError)
    if ok {
      fmt.Println("param error:", invalid)
      return
    }

    validationErrs := err.(validator.ValidationErrors)
    for _, validationErr := range validationErrs {
      fmt.Println(validationErr)
    }
}

func main() {
    validate := validator.New()

    err := validate.Struct(1)
    processErr(err)

    err = validate.VarWithValue(1, 2, "eqfield")
    processErr(err)
}
```

总结

`validator` 功能非常丰富，使用较为简单方便。本文介绍的约束只是其中的冰山一角。它的应用非常广泛，建议了解一下。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. validator GitHub：https://github.com/go-playground/validator
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

govaluate

简介

今天我们介绍一个比较好玩的库 `govaluate` 。 `govaluate` 与 JavaScript 中的 `eval` 功能类似，用于计算任意表达式的值。此类功能函数在 JavaScript/Python 等动态语言中比较常见。 `govaluate` 让 Go 这个编译型语言也有了这个能力！

快速使用

先安装：

```
$ go get github.com/Knetic/govaluate
```

后使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/Knetic/govaluate"
)

func main() {
  expr, err := govaluate.NewEvaluableExpression("10 > 0")
  if err != nil {
    log.Fatal("syntax error:", err)
  }

  result, err := expr.Evaluate(nil)
  if err != nil {
    log.Fatal("evaluate error:", err)
  }

  fmt.Println(result)
}
```

使用 `govaluate` 计算表达式只需要两步：

- 调用 `NewEvaluableExpression()` 将表达式转为一个**表达式对象**；
- 调用表达式对象的 `Evaluate` 方法，传入参数，返回表达式的值。

上面演示了一个很简单的例子，我们使用 `govaluate` 计算 `10 > 0` 的值，该表达式不需要参数，故传给 `Evaluate()` 方法 `nil` 值。当然，这个例子并不实用，显然我们直接在代码中计算 `10 > 0` 更简单。但问题是，有些时候我们并不知道需要计算的表达式的所有信息，甚至我们都不知道表达式的结构。这时 `govaluate` 的作用就体现出来了。

参数

`govaluate` 支持在表达式中使用参数，调用表达式对象的 `Evaluate()` 方法时通过 `map[string]interface{}` 类型将参数传入计算。其中 `map` 的键为参数名，值为参数值。例如：

```
func main() {
  expr, _ := govaluate.NewEvaluableExpression("foo > 0")
  parameters := make(map[string]interface{})
  parameters["foo"] = -1
  result, _ := expr.Evaluate(parameters)
  fmt.Println(result)

  expr, _ = govaluate.NewEvaluableExpression("(requests_made * requests_succeeded / 100) >= 90")
  parameters = make(map[string]interface{})
  parameters["requests_made"] = 100
  parameters["requests_succeeded"] = 80
  result, _ = expr.Evaluate(parameters)
  fmt.Println(result)

  expr, _ = govaluate.NewEvaluableExpression("(mem_used / total_mem) * 100")
  parameters = make(map[string]interface{})
  parameters["total_mem"] = 1024
  parameters["mem_used"] = 512
  result, _ = expr.Evaluate(parameters)
  fmt.Println(result)
}
```

第一个表达式中，我们想要计算 `foo > 0` 的结果，在传入参数中将 `foo` 设置为 -1，最终输出 `false` 。

第二个表达式中，我们想要计算 `(requests_made * requests_succeeded / 100) >= 90` 的值，在参数中设置 `requests_made` 为 100， `requests_succeeded` 为 80，结果为 `true` 。

上面两个表达式都返回 `bool` 结果，第三个表达式返回一个浮点数。 `(mem_used / total_mem) * 100` 根据传入的总内存 `total_mem` 和当前使用内存 `mem_used` ，返回内存占用百分比，结果为 50。

命名

使用 `govaluate` 与直接编写 Go 代码不同，在 Go 代码中标识符中不能出现 `-` 、 `+` 、 `$` 等符号。 `govaluate` 可以通过转义使用这些符号。有两种转义方式：

- 将名称用 `[` 和 `]` 包裹起来，例如 `[response-time]` ；
- 使用 `\` 将紧接着**下一个**的字符转义。

例如：

```
func main() {
  expr, _ := govaluate.NewEvaluableExpression("[response-time] < 100")
  parameters := make(map[string]interface{})
  parameters["response-time"] = 80
  result, _ := expr.Evaluate(parameters)
  fmt.Println(result)

  expr, _ = govaluate.NewEvaluableExpression("response\\-time < 100")
  parameters = make(map[string]interface{})
  parameters["response-time"] = 80
  result, _ = expr.Evaluate(parameters)
  fmt.Println(result)
}
```

注意一点，因为在字符串中 `\` 本身就是需要转义的，所以在第二个表达式中要使用 `\\` 。或者可以使用

```
`response\-time` < 100
```

一次“编译”多次运行

使用带参数的表达式，我们可以实现一个表达式的一次“编译”，多次运行。只需要使用编译返回的表达式对象即可，可多次调用其 `Evaluate()` 方法：

```
func main() {
  expr, _ := govaluate.NewEvaluableExpression("a + b")
  parameters := make(map[string]interface{})
```

```
  parameters["a"] = 1
  parameters["b"] = 2
  result, _ := expr.Evaluate(parameters)
  fmt.Println(result)

  parameters = make(map[string]interface{})
  parameters["a"] = 10
  parameters["b"] = 20
  result, _ = expr.Evaluate(parameters)
  fmt.Println(result)
}
```

第一次运行，传入参数 `a = 1, b = 2` 得到结果 3；第二次运行，传入参数 `a = 10, b = 20` 得到结果 30。

函数

如果仅仅能进行常规的算数和逻辑运算， `govaluate` 的功能会大打折扣。 `govaluate` 提供了自定义函数的功能。所有自定义函数需要先定义好，存入一个 `map[string]govaluate.ExpressionFunction` 变量中，然后调用 `govaluate.NewEvaluableExpressionWithFunctions()` 生成表达式，此表达式中就可以使用这些函数了。自定义函数类型为 `func (args ...interface{}) (interface{}, error)` ，如果函数返回错误，则这个表达式求值返回错误。

```
func main() {
  functions := map[string]govaluate.ExpressionFunction{
    "strlen": func(args ...interface{}) (interface{}, error) {
      length := len(args[0].(string))
      return length, nil
    },
  }

  exprString := "strlen('teststring')"
  expr, _ := govaluate.NewEvaluableExpressionWithFunctions(exprString, function
s)
  result, _ := expr.Evaluate(nil)
  fmt.Println(result)
}
```

上面例子中，我们定义一个函数 `strlen` 计算第一个参数的字符串长度。表达式 `strlen('teststring')` 调用 `strlen` 函数返回字符串 `teststring` 的长度。

函数可以接受任意数量的参数，而且可以处理嵌套函数调用的问题。所以可以写出类似下面这种复杂的表达式：

```
sqrt(x1 ** y1, x2 ** y2)

max(someValue, abs(anotherValue), 10 * lastValue)
```

访问器

在 Go 语言中，访问器（ `Accessors` ）就是通过 `.` 操作访问结构中的字段。如果传入的参数中有结构体类型， `govaluate` 也支持使用 `.` 访问其内部字段或调用它们的方法：

```
type User struct {
  FirstName string
  LastName  string
  Age       int
}

func (u User) Fullname() string {
  return u.FirstName + " " + u.LastName
}

func main() {
  u := User{FirstName: "li", LastName: "dajun", Age: 18}
  parameters := make(map[string]interface{})
  parameters["u"] = u

  expr, _ := govaluate.NewEvaluableExpression("u.Fullname()")
  result, _ := expr.Evaluate(parameters)
  fmt.Println("user", result)

  expr, _ = govaluate.NewEvaluableExpression("u.Age > 18")
  result, _ = expr.Evaluate(parameters)
  fmt.Println("age > 18?", result)
}
```

在上面代码中，我们定义了一个 `User` 结构，并为它编写了一个 `Fullname()` 方法。第一个表达式中，我们调用 `u.Fullname()` 返回全名，第二个表达式比较年龄是否大于 18。

需要注意的一点是，我们不能使用 `foo.SomeMap['key']` 的方式访问 `map` 的值。由于访问器涉及到很多反射，所以它一般比直接使用参数慢 4 倍左右。如果能使用参数的形式，尽量使用参数。在上面的例子中，我们可以直接调用 `u.Fullname()` ，将结果作为参数传给表达式求值。涉及到复杂的计算可以通过自定义函数来解决。我们还可以实现 `govaluate.Parameter` 接口，对于表达式中使用的未知参数， `govaluate` 会自动调用其 `Get()` 方法获取：

```go
// src/github.com/Knetic/govaluate/parameters.go
type Parameters interface {
  Get(name string) (interface{}, error)
}
```

例如，我们可以让 `User` 实现 `Parameter` 接口：

```go
type User struct {
  FirstName string
  LastName  string
  Age       int
}

func (u User) Get(name string) (interface{}, error) {
  if name == "FullName" {
    return u.FirstName + " " + u.LastName, nil
  }

  return nil, errors.New("unsupported field " + name)
}

func main() {
  u := User{FirstName: "li", LastName: "dajun", Age: 18}
  expr, _ := govaluate.NewEvaluableExpression("FullName")
  result, _ := expr.Eval(u)
  fmt.Println("user", result)
}
```

表达式对象实际上有两个方法，一个是我们前面用的 `Evaluate()` ，这个方法接受一个 `map[string]interface{}` 参数。另一个就是我们在这个例子中使用的 `Eval()` 方法，该方法接受一个 `Parameter` 接口。实际上，在 `Evaluate()` 实现内部也是调用的 `Eval()` 方法：

```go
// src/github.com/Knetic/govaluate/EvaluableExpression.go
func (this EvaluableExpression) Evaluate(parameters map[string]interface{}) (interface{}, error) {
  if parameters == nil {
    return this.Eval(nil)
  }
  return this.Eval(MapParameters(parameters))
}
```

在表达式计算时，未知的参数都需要调用 `Parameter` 的 `Get()` 方法获取。上面的例子中我们直接使用 `FullName` 就可以调用 `u.Get()` 方法返回全名。

支持的操作和类型

`govaluate` 支持的操作和类型与 Go 语言有些不同。一方面 `govaluate` 中的类型和操作不如 Go 丰富，另一方面 `govaluate` 也对一些操作进行了扩展。

算数、比较和逻辑运算：

- `+` `-` `/` `*` `&` `|` `^` `**` `%` `>>` `<<`：加减乘除，按位与，按位或，异或，乘方，取模，左移和右移；
- `>` `>=` `<` `<=` `==` `!=` `=~` `!~`：`=~`为正则匹配，`!~`为正则不匹配；
- `||` `&&`：逻辑或和逻辑与。

常量：

- 数字常量，`govaluate`中将数字都作为 64 位浮点数处理；
- 字符串常量，注意在`govaluate`中，字符串用单引号`'`；
- 日期时间常量，格式与字符串相同，**`govaluate`会尝试自动解析字符串是否是日期**，只支持 RFC3339、ISO8601等有限的格式；
- 布尔常量：`true`、`false`。

其他：

- 圆括号可以改变计算优先级；
- 数组定义在`()`中，每个元素之间用`,`分隔，可以支持任意的元素类型，如`(1, 2, 'foo')`。实际上在`govaluate`中数组是用`[]interface{}`来表示的；
- 三目运算符：`? :`。

在下面代码中，`govaluate` 会先将 `2014-01-02` 和 `2014-01-01 23:59:59` 转为 `time.Time` 类型，然后再比较大小：

```
func main() {
  expr, _ := govaluate.NewEvaluableExpression("'2014-01-02' > '2014-01-01 23:59:59'")
  result, _ := expr.Evaluate(nil)
  fmt.Println(result)
}
```

错误处理

在上面的例子中，我们刻意忽略了错误处理。实际上， `govaluate` 在创建表达式对象和表达式求值这两个操作中都可能产生错误。在生成表达式对象时，如果表达式有语法错误，则返回错误。表达式求值，如果传入的参数不合法，或者某些参数缺失，或者访问结构体中不存在的字段都会报错。

```
func main() {
  exprString := `>>>`
  expr, err := govaluate.NewEvaluableExpression(exprString)
  if err != nil {
    log.Fatal("syntax error:", err)
  }
  result, err := expr.Evaluate(nil)
  if err != nil {
    log.Fatal("evaluate error:", err)
  }
  fmt.Println(result)
}
```

我们可以依次修改表达式字符串，验证各种错误，首先是 `>>>` ：

```
2020/04/01 22:31:59 syntax error:Invalid token: '>>>'
```

然后我们将其修改为 `foo > 0` ，但是我们没有传入参数 `foo` ，执行失败：

```
2020/04/01 22:33:07 evaluate error:No parameter 'foo' found.
```

其他错误可以自行验证。

总结

`govaluate` 虽然支持的操作和类型有限，也能实现比较有意思的功能。例如，可以写一个 Web 服务，由用户自己编写表达式，设置参数，服务器算出结果。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. govaluate GitHub：https://github.com/Knetic/govaluate
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

jj

简介

在前面两篇文章中，我们分别介绍了快速读取 JSON 值的库 `gjson` 和快速设置 JSON 值的库 `sjson` 。今天我们介绍它们的作者tidwall的一个基于 `gjson` 和 `sjson` 的非常实用的命令行工具 `jj` 。它是使用 Go 编写的快速读取和设置 JSON 值的命令行程序。

快速使用

Mac 上可以直接使用 `brew install tidwall/jj/jj` 安装。其他系统可以通过下载编译好的可执行程序，下载地址为https://github.com/tidwall/jj/releases。

我选择使用 `go get` 安装：

```
$ go get github.com/tidwall/jj/cmd/jj
```

上面命令执行完成之后，编译生成的 `jj` 程序会放在 `$GOPATH/bin` 目录中，我习惯把 `$GOPATH/bin` 加入系统可执行目录 `$PATH` 中，故可以直接使用。

简单的读取和设置（我的环境为 Win10 + Git Bash）：

```
$ echo '{"name":{"first":"li","last":"dj"}}' | jj name.last
dj

$ echo '{"name":{"first":"li","last":"dj"}}' | jj -v dajun name.last
{"name":{"first":"li","last":"dajun"}}
```

通过**键路径**来指定读取/设置的位置，上面第一个命令读取字段 `name.last` ，返回 `dj` 。

`-v` 选项指定设置的值。第二个命令将字段 `name.last` 设置为 `dajun` ，输出设置之后的 JSON 串。键路径在前两篇文章中有详细的介绍，不熟悉的可以回去看一下。

读取和设置

实际上读取和设置的语法和形式与我们前面介绍 `gjson` 和 `sjson` 提到的基本一样，只不过是在命令行上完成的而已。

读取不存在的字段，返回 `null` ：

```
$ echo '{"name":{"first":"li","last":"dj"}}' | jj name.middle
null
```

读取一个对象类型的字段，返回该对象的 JSON 表示：

```
$ echo '{"name":{"first":"li","last":"dj"}}' | jj name
{"first":"li","last":"dj"}
```

使用索引（从 0 开始）读取数组的元素，非法的索引将返回空：

```
$ echo '{"fruits":["apple","orange","banana"]}' | jj fruits.1
orange

$ echo '{"fruits":["apple","orange","banana"]}' | jj fruits.3
```

使用索引设置数组的元素，下面命令将数组 `fruits` 的第二个元素设置为 `pear` ：

```
$ echo '{"fruits":["apple","orange","banana"]}' | jj -v pear fruits.1
{"fruits":["apple","pear","banana"]}
```

使用 `-1` 或数组长度作为索引，可以在数组后添加一个元素。如果索引超过了数组长度，则会多一定数量的 `null` ：

```
$ echo '{"fruits":["apple","orange","banana"]}' | jj -v strawberry fruits.-1
{"fruits":["apple","orange","banana","strawberry"]}

$ echo '{"fruits":["apple","orange","banana"]}' | jj -v grape fruits.3
{"fruits":["apple","orange","banana","grape"]}

$ echo '{"fruits":["apple","orange","banana"]}' | jj -v watermelon fruits.5
{"fruits":["apple","orange","banana",null,null,"watermelon"]}
```

使用选项 `-D` 删除指定键路径上的元素，如果对应元素不存在，则无效果：

```
$ echo '{"name":"dj","age":18}' | jj -D age
{"name":"dj"}

$ echo '{"fruits":["apple","orange","banana"]}' | jj -D fruits.2
{"fruits":["apple","orange"]}

$ echo '{"fruits":["apple","orange","banana"]}' | jj -D fruits.5
{"fruits":["apple","orange","banana"]}
```

第 1 个命令删除字段 `age` ；第 2 个命令删除数组 `fruits` 的第 2 个元素；第 3 个命令删除数组 `fruits` 的第 5 个元素，由于数组长度只有 3，故无效果。

文件

`jj` 支持从文件中读取 JSON 串和将结果写到文件中。使用选项 `-i` 指定输入文件，选项 `-o` 指定输出文件。下面将从文件 `fruits.txt` 中读取 JSON 串，取数组的第 2 个元素，写到 `out.txt` 中：

```
$ jj -i fruits.txt -o out.txt fruits.1
```

`fruits.txt` 的文件内容如下：

```
{"fruits":["apple","orange","banana"]}
```

执行命令，输出文件的内容为：

```
orange
```

格式化

`jj` 支持将输出的 JSON 串进行一定的格式化。选项 `-u` 移除所有的空白符，节省存储空间。选项 `-p` 美化格式，便于阅读。

```
$ echo '{"name":{"first": "li", "last":"dj"}, "age":18}' | jj -u name
{"first":"li","last":"dj"}

$ echo '{"name":{"first": "li", "last":"dj"}, "age":18}' | jj -p name
{
  "first": "li",
  "last": "dj"
}
```

性能

与另一个 JSON 的命令行工具 `jq` 相比， `jj` 是其性能的 10 倍以上。因为 `jj` 不会验证 JSON 串的有效性，并且它只关心键路径指定的值，一旦该值处理完成就停止。这里有性能对比：https://github.com/tidwall/jj#performance

用途

`jj` 一个很方便的用途在于日志处理，当前很多日志库都支持 JSON 的格式，例如前面我们介绍的 `logrus` 。我们可以使用 `jj` 在这些日志中找到相应的信息。我们先用 `logrus` 生成 20 条玩家登陆和下线的日志：

```
package main

import "github.com/sirupsen/logrus"

func main() {
  logrus.SetFormatter(&logrus.JSONFormatter{})

  for i := 1; i <= 10; i++ {
    logrus.WithFields(logrus.Fields{
      "userid": i,
    }).Info("login")
    logrus.WithFields(logrus.Fields{
      "userid": i,
    }).Info("logoff")
  }
}
```

生成日志存储在 `log.txt` 文件中：

```
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":1}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":1}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":2}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":2}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":3}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":3}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":4}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":4}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":5}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":5}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":6}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":6}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":7}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":7}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":8}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":8}
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":9}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":9}
```

```
{"level":"info","msg":"login","time":"2020-03-26T23:36:04+08:00","userid":10}
{"level":"info","msg":"logoff","time":"2020-03-26T23:36:04+08:00","userid":10}
```

由于每一行都是一个单独的 JSON 串，我们可以使用 `jj` 支持的 JSON 行特性，使用 `..` 路径标识这些行。`..` 使得 `jj` 将这些行看成数组的元素。我们可以读取这些数组元素。

获取数组长度，返回 20：

```
$ jj -i log.txt ..#
20
```

只读取每一行中的 `userid` 信息：

```
$ jj -i log.txt ..#.userid
[1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
```

只读取每一行中的 `msg` 信息：

```
$ jj -i log.txt ..#.msg
["login","logoff","login","logoff","login","logoff","login","logoff","login","logoff","login","logoff","login","logoff","login","logoff","login","logoff","login","logoff"]
```

更复杂一点的，如果我们要查看所有 `userid=1` 的日志：

```
$ jj -i log.txt ..#\(userid=1\)# -p
[
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 1
  },
  {
    "level": "info",
    "msg": "logoff",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 1
  }
]
```

上面的命令注意两点， `(` 和 `)` 是 shell 中的特殊字符，需要 `\` 转义。命令中我们使用 `-p` 选项使结果更易读。

如果我们只需要查找第一条符合条件的日志，则可以去掉最右侧的 `#` ：

```
$ jj -i log.txt ..#\(userid=1\) -p
{
  "level": "info",
  "msg": "login",
  "time": "2020-03-26T23:36:04+08:00",
  "userid": 1
}
```

如果要查看所有的登录信息：

```
$ jj -i log.txt ..#\(msg="login"\)# -p
[
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 1
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 2
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 3
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 4
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
```

```
    "userid": 5
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 6
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 7
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 8
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 9
  },
  {
    "level": "info",
    "msg": "login",
    "time": "2020-03-26T23:36:04+08:00",
    "userid": 10
  }
]
```

总结

`jj` 是一个非常使用的 JSON 命令行工具，性能超赞。执行 `jj -h` 去看看其他选项吧。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. jj GitHub：https://github.com/tidwall/jj

2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

sjson

简介

在上一篇文章中我们介绍了如何使用 `gjson` 快速读取 JSON 串中的值。为了内容的完整性，今天我们介绍一下如何使用 `sjson` 快速设置 JSON 串中的值。

快速使用

先安装：

```
$ go get github.com/tidwall/sjson
```

后使用：

```
package main

import (
  "fmt"

  "github.com/tidwall/sjson"
)

const json = `{"name":{"first":"li","last":"dj"},"age":18}`

func main() {
  value, _ := sjson.Set(json, "name.last", "dajun")
  fmt.Println(value)
}
```

上面代码通过 `sjson.Set()` 将 JSON 串中 `name.last` 对应的值设置为 `dajun` 。与 `gjson` 一样， `sjson` 也通过**键路径**指定具体的位置，键路径即为一系列以 `.` 分隔的键。 `sjson` 支持的键路径语法是 `gjson` 的一个子集，具体键路径的语法可以参见上一篇文章。 `sjson.Set()` 返回设置之后的 JSON 串。最终程序输出：

```
{"name":{"first":"li","last":"dajun"},"age":18}
```

支持的类型

`sjson` 支持的类型包括 `nil/bool/int/float/string` 等。如果传入 `sjson` 不支持的类型， `sjson` 会调用 `json.Marshal` ，然后将生成的字符串设置到对应的键路径上：

```
type User struct {
  Name string `json:"name"`
  Age  int    `json:"age"`
}

func main() {
  nilJSON, _ := sjson.Set("", "key", nil)
  fmt.Println(nilJSON)

  boolJSON, _ := sjson.Set("", "key", false)
  fmt.Println(boolJSON)

  intJSON, _ := sjson.Set("", "key", 1)
  fmt.Println(intJSON)

  floatJSON, _ := sjson.Set("", "key", 10.5)
  fmt.Println(floatJSON)

  strJSON, _ := sjson.Set("", "key", "hello")
  fmt.Println(strJSON)

  mapJSON, _ := sjson.Set("", "key", map[string]interface{}{"hello": "world"})
  fmt.Println(mapJSON)

  u := User{Name: "dj", Age: 18}
  structJSON, _ := sjson.Set("", "key", u)
  fmt.Println(structJSON)
}
```

注意，我们传入一个空字符串， `sjson.Set()` 会生成一个空对象，然后按照键路径依次设置值。下面分析上述程序输出：

- `nil` ：在 JSON 中用 `null` 表示，输出 `{"key":null}` ；
- `false` ：在 JSON 中布尔值用 `true/false` 表示，输出 `{"key":false}` ；
- `1` 和 `10.5` ：整数和浮点数在 JSON 中都用 `number` 表示，分别输出 `{"key":1}` 和 `{"key":10.5}` ；
- `hello` ：输出 `{"key":"hello"}` ；
- `map[string]interface{}` ： `sjson` 并不原生支持 `map` 类型，故通过 `json.Marshal` 将其序列化为 `{"hello":"world"}` 再设置到键 `key` 上，输

出 `{"key":{"hello":"world"}}`；
- `User` 对象：先通过 `json.Marshal` 序列化为 `{"name":"dj","age":18}` 再设置；

修改数组

修改数组可以通过在键路径后添加索引，有两种特殊情况：

- 使用 `-1` 或数组长度为索引表示在数组后添加一个新元素；
- 使用的索引超出数组的长度，会在数组中添加很多 `null` 值。

看下面示例：

```
func main() {
  fruits := `{"fruits":["apple", "orange", "banana"]}`

  var newValue string
  newValue, _ = sjson.Set(fruits, "fruits.1", "grape")
  fmt.Println(newValue)

  newValue, _ = sjson.Set(fruits, "fruits.3", "pear")
  fmt.Println(newValue)

  newValue, _ = sjson.Set(fruits, "fruits.-1", "strawberry")
  fmt.Println(newValue)

  newValue, _ = sjson.Set(fruits, "fruits.5", "watermelon")
  fmt.Println(newValue)
}
```

- `fruits.1`：设置第二个水果为 `grape`（**索引从 0 开始**），输出 `{"fruits":["apple", "grape", "banana"]}`；
- `fruits.3`：由于数组长度为 3，使用 3 表示在数组后添加一个元素，输出 `{"fruits":["apple","orange","banana","pear"]}`；
- `fruits.-1`：使用 `-1` 同样表示在数组后添加一个元素，输出 `{"fruits":["apple", "orange", "banana","strawberry"]}`；
- `fruits.5`：索引 5 已经大于数组长度 3 了，所以会多出两个 `null`，输出 `{"fruits":["apple","orange","banana",null,null,"watermelon"]}`。

删除

删除数组元素需要调用 `sjson.Delete()` 方法，键路径语法相同。如果键路径对应的值不存在，则 `Delete()` 无效果：

```
func main() {
  var newValue string
  user := `{"name":{"first":"li","last":"dj"},"age":18}`

  newValue, _ = sjson.Delete(user, "name.first")
  fmt.Println(newValue)

  newValue, _ = sjson.Delete(user, "name.full")
  fmt.Println(newValue)

  fruits := `{"fruits":["apple", "orange", "banana"]}`

  newValue, _ = sjson.Delete(fruits, "fruits.1")
  fmt.Println(newValue)

  newValue, _ = sjson.Delete(fruits, "fruits.-1")
  fmt.Println(newValue)

  newValue, _ = sjson.Delete(fruits, "fruits.5")
  fmt.Println(newValue)
}
```

- `name.first`：删除字段 `name.first`，输出 `{"name":{"last":"dj"},"age":18}`；
- `name.full`：由于字段 `name.full` 不存在，无效果，输出 `{"name":{"first":"li","last":"dj"},"age":18}`；
- `fruits.1`：删除数组 `fruits` 的第二个元素，输出 `{"fruits":["apple", "banana"]}`；
- `fruits.-1`：删除数组最后一个元素，输出 `{"fruits":["apple", "orange"]}`；
- `fruits.5`：索引 5 超出数组长度 3，无效果，输出 `{"fruits":["apple", "orange", "banana"]}`。

错误处理

使用 `sjson` 出现的错误分为两种，一种是传入的 JSON 串不是合法的串，另一种是键路径语法错误。 `Set()` 和 `Delete()` 方法返回的第二个参数为错误，只有非法的键路径会返回错误，非法 JSON 串不会。

非法 JSON 串

同 `gjson` 一样， `sjson` 同样不会检查传入的 JSON 串的合法性，它假设传入的是合法的串。如果传入一个非法的 JSON 串，程序输出不确定的结果：

```
func main() {
  user := `{"name":dj,age:18}`
  newValue, err := sjson.Set(user, "name", "dajun")
  fmt.Println(err, newValue)
}
```

上面程序中，我故意传入一个非法的 JSON 串（ `dj` 和 `age` 漏掉了双引号）。最终程序输出：

```
<nil> {"name":dj,age:"dajun"}
```

将 `age` 变为了 `dajun` ，显然不正确。然而此时返回的 `err = nil` 。

非法键路径

与 `gjson` 相比， `sjson` 能使用的键路径语法比较有限，不能使用通配符和一些条件语法。如果传入的键路径非法，将返回非空的错误值：

```
func main() {
  user := `{"name":"dj","age":18}`
  newValue, err := sjson.Set(user, "na?e", "dajun")
  fmt.Println(err, newValue)
}
```

上次使用通配符 `?` ，输出：

```
wildcard characters not allowed in path
```

总结

`sjson` 比较简单易用，性能不俗。我们在确定 JSON 串合法的情况下，可使用它快速设置值。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. sjson GitHub：https://github.com/tidwall/sjson
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gjson

简介

之前我们介绍过 `gojsonq` ，可以方便地从一个 JSON 串中读取值。同时它也支持各种查询、汇总统计等功能。今天我们再介绍一个类似的库 `gjson` 。在上一篇文章Go 每日一库之 buntdb中我们介绍过 JSON 索引，内部实现其实就是使用 `gjson` 这个库。 `gjson` 实际上是 `get + json` 的缩写，用于读取 JSON 串，同样的还有一个 `sjson` （ `set + json` ）库用来设置 JSON 串。

快速使用

先安装：

```
$ go get github.com/tidwall/gjson
```

后使用：

```
package main

import (
  "fmt"

  "github.com/tidwall/gjson"
)

func main() {
  json := `{"name":{"first":"li","last":"dj"},"age":18}`
  lastName := gjson.Get(json, "name.last")
  fmt.Println("last name:", lastName.String())

  age := gjson.Get(json, "age")
  fmt.Println("age:", age.Int())
}
```

使用很简单，只需要传入 JSON 串和要读取的键路径即可。注意一点细节，因为 `gjson.Get()` 函数实际上返回的是 `gjson.Result` 类型，我们要调用其相应的方法进行转换对应的类型。如上面的 `String()` 和 `Int()` 方法。

如果是直接打印输出，其实可以省略 `String()` ， `fmt` 包的大部分函数都可以对实现 `fmt.Stringer` 接口的类型调用 `String()` 方法。

键路径

键路径实际上是以 `.` 分隔的一系列键。 `gjson` 支持在键中包含通配符 `*` 和 `?` ， `*` 匹配任意多个字符， `?` 匹配单个字符，例如 `ca*` 可以匹配 `cat/cate/cake` 等以 `ca` 开头的键， `ca?` 只能匹配 `cat/cap` 等以 `ca` 开头且后面只有一个字符的键。

数组使用**键名** + `.` + **索引**（索引从 0 开始）的方式读取元素，如果键 `pets` 对应的值是一个数组，那么 `pets.0` 读取数组的第一个元素， `pets.1` 读取第二个元素。

数组长度使用**键名** + `.` + `#` 获取，例如 `pets.#` 返回数组 `pets` 的长度。

如果键名中出现 `.` ，那么需要使用 `\` 进行转义。

```
package main

const json = `
{
  "name":{"first":"Tom", "last": "Anderson"},
  "age": 37,
  "children": ["Sara", "Alex", "Jack"],
  "fav.movie": "Dear Hunter",
  "friends": [
    {"first": "Dale", "last":"Murphy", "age": 44, "nets": ["ig", "fb", "tw"]},
    {"first": "Roger", "last": "Craig", "age": 68, "nets": ["fb", "tw"]},
    {"first": "Jane", "last": "Murphy", "age": 47, "nets": ["ig", "tw"]}
  ]
}
`

func main() {
  fmt.Println("last name:", gjson.Get(json, "name.last"))
  fmt.Println("age:", gjson.Get(json, "age"))
  fmt.Println("children:", gjson.Get(json, "children"))
  fmt.Println("children count:", gjson.Get(json, "children.#"))
  fmt.Println("second child:", gjson.Get(json, "children.1"))
  fmt.Println("third child*:", gjson.Get(json, "child*.2"))
  fmt.Println("first c?ild:", gjson.Get(json, "c?ildren.0"))
  fmt.Println("fav.moive", gjson.Get(json, `fav.\moive`))
  fmt.Println("first name of friends:", gjson.Get(json, "friends.#.first"))
  fmt.Println("last name of second friend:", gjson.Get(json, "friends.1.last"))
}
```

前 3 个比较简单，就不赘述了。看后面几个：

- `children.#`：返回数组`children`的长度；
- `children.1`：读取数组`children`的第 2 个元素（注意索引从 0 开始）；
- `child*.2`：首先`child*`匹配`children`，`.2`读取第 3 个元素；
- `c?ildren.0`：`c?ildren`匹配到`children`，`.0`读取第一个元素；
- `fav.\moive`：因为键名中含有`.`，故需要`\`转义；
- `friends.#.first`：如果数组后`#`后还有内容，则以后面的路径读取数组中的每个元素，返回一个新的数组。所以该查询返回的数组所有`friends`的`first`字段组成；
- `friends.1.last`：读取`friends`第 2 个元素的`last`字段。

运行结果：

```
last name: Anderson
age: 37
children: ["Sara", "Alex", "Jack"]
children count: 3
second child: Alex
third child*: Jack
first c?ild: Sara
fave.moive
first name of friends: ["Dale","Roger","Jane"]
last name of second friend: Craig
```

对于数组，`gjson`还支持按条件查询元素，`#(条件)`返回第一个满足条件的元素，`#(条件)#`返回所有满足条件的元素。括号内的条件可以有`==`、`!=`、`<`、`<=`、`>`、`>=`，还有简单的模式匹配`%`（符合某个模式），`!%`（不符合某个模式）：

```
fmt.Println(gjson.Get(json, `friends.#(last="Murphy").first`))
fmt.Println(gjson.Get(json, `friends.#(last="Murphy")#.first`))
fmt.Println(gjson.Get(json, "friends.#(age>45)#.last"))
fmt.Println(gjson.Get(json, `friends.#(first%"D*").last`))
fmt.Println(gjson.Get(json, `friends.#(first!%"D*").last`))
fmt.Println(gjson.Get(json, `friends.#(nets.#(=="fb"))#.first`))
```

还是使用上面的 JSON 串。

- `friends.#(last="Murphy").first`：`friends.#(last="Murphy")`返回数组`friends`中第一个`last`为`Murphy`的元素，`.first`表示取出该元素

的 `first` 字段返回；
- `friends.#(last="Murphy")#.first`：`friends.#(last="Murphy")#` 返回数组 `friends` 中所有的 `last` 为 `Murphy` 的元素，然后读取它们的 `first` 字段放在一个数组中返回。注意与上面一个的区别；
- `friends.#(age>45)#.last`：`friends.#(age>45)#` 返回数组 `friends` 中所有年龄大于 45 的元素，然后读取它们的 `last` 字段返回；
- `friends.#(first%"D*").last`：`friends.#(first%"D*")` 返回数组 `friends` 中第一个 `first` 字段满足模式 `D*` 的元素，取出其 `last` 字段返回；
- `friends.#(first!%"D*").last`：` `friends.#(first!%"D*")` 返回数组 `friends` 中第一个 `first` 字段**不**满足模式 `D*` 的元素，读取其 `last`字段返回；
- `friends.#(nets.#(=="fb"))#.first`：这是个嵌套条件，`friends.#(nets.#(=="fb"))#` 返回数组 `friends` 的元素的 `nets` 字段中有 `fb` 的所有元素，然后取出 `first` 字段返回。

运行结果：

```
Dale
["Dale","Jane"]
["Craig","Murphy"]
Murphy
Craig
["Dale","Roger"]
```

修饰符

修饰符是 `gjson` 提供的非常强大的功能，和键路径搭配使用。`gjson` 提供了一些内置的修饰符：

- `@reverse`：翻转一个数组；
- `@ugly`：移除 JSON 中的所有空白符；
- `@pretty`：使 JSON 更易用阅读；
- `@this`：返回当前的元素，可以用来返回根元素；
- `@valid`：校验 JSON 的合法性；
- `@flatten`：数组平坦化，即将 `["a", ["b", "c"]]` 转为 `["a","b","c"]`；
- `@join`：将多个对象合并到一个对象中。

修饰符的语法和管道类似，以 `|` 分隔键路径和分隔符。

```
const json = `{
  "name":{"first":"Tom", "last": "Anderson"},
  "age": 37,
  "children": ["Sara", "Alex", "Jack"],
  "fav.movie": "Dear Hunter",
  "friends": [
    {"first": "Dale", "last":"Murphy", "age": 44, "nets": ["ig", "fb", "tw"]},
    {"first": "Roger", "last": "Craig", "age": 68, "nets": ["fb", "tw"]},
    {"first": "Jane", "last": "Murphy", "age": 47, "nets": ["ig", "tw"]}
  ]
}`

func main() {
  fmt.Println(gjson.Get(json, "children|@reverse"))
  fmt.Println(gjson.Get(json, "children|@reverse|0"))
  fmt.Println(gjson.Get(json, "friends|@ugly"))
  fmt.Println(gjson.Get(json, "friends|@pretty"))
  fmt.Println(gjson.Get(json, "@this"))

  nestedJSON := `{"nested": ["one", "two", ["three", "four"]]}`
  fmt.Println(gjson.Get(nestedJSON, "nested|@flatten"))

  userJSON := `{"info":[{"name":"dj", "age":18},{"phone":"123456789","email":"dj
@example.com"}]}`
  fmt.Println(gjson.Get(userJSON, "info|@join"))
}
```

`children|@reverse` 先读取数组 `children`，然后使用修饰符 `@reverse` 翻转之后返回，输出：

```
["Jack","Alex","Sara"]
```

`children|@reverse|0` 在上面翻转的基础上读取第一个元素，即原数组的最后一个元素，输出：

```
Jack
```

`friends|@ugly` 移除 `friends` 数组中的所有空白字符，返回一行长长的字符串：

```
[{"first":"Dale","last":"Murphy","age":44,"nets":["ig","fb","tw"]},{"first":"Rog
er","last":"Craig","age":68,"nets":["fb","tw"]},{"first":"Jane","last":"Murphy",
```

```
"age":47,"nets":["ig","tw"]}]
```

`friends|@pretty` 格式化 `friends` 数组，使之更易读：

```
[
  {
    "first": "Dale",
    "last": "Murphy",
    "age": 44,
    "nets": ["ig", "fb", "tw"]
  },
  {
    "first": "Roger",
    "last": "Craig",
    "age": 68,
    "nets": ["fb", "tw"]
  },
  {
    "first": "Jane",
    "last": "Murphy",
    "age": 47,
    "nets": ["ig", "tw"]
  }
]
```

`@this` 返回原始的 JSON 串。

`@flatten` 将数组 `nested` 的内层数组平坦到外层后返回，即将所有内层数组的元素依次添加到外层数组后面并移除内层数组，输出：

```
["one","two","three", "four"]
```

`@join` 将一个数组中的各个对象合并到一个中，例子中将数组中存放的部分个人信息合并成一个对象返回：

```
{"name":"dj","age":18,"phone":"123456789","email":"dj@example.com"}
```

修饰符参数

修饰符还可以有参数，通过在修饰符后加 `:` 后跟参数。如果我们在格式化 JSON 串时，想要对键进行排序，那么可以使用 `@pretty` 修饰符的 `sortKeys` 参数。我们还是拿上面的 JSON 数据举例：

```
fmt.Println(gjson.Get(json, `friends|@pretty:{"sortKeys":true}`))
```

最终按键名顺序输出 JSON 串：

```
[
  {
    "age": 44,
    "first": "Dale",
    "last": "Murphy",
    "nets": ["ig", "fb", "tw"]
  },
  {
    "age": 68,
    "first": "Roger",
    "last": "Craig",
    "nets": ["fb", "tw"]
  },
  {
    "age": 47,
    "first": "Jane",
    "last": "Murphy",
    "nets": ["ig", "tw"]
  }
]
```

当然还可以指定每行缩进 `indent` （默认两个空格），每行开头字符串 `prefix` （默认为空串）和一行最多显示字符数 `width` （默认 80 字符）。下面在每行前增加两个空格：

```
fmt.Println(gjson.Get(json, `friends|@pretty:{"sortKeys":true,"prefix":"  "}`))
```

自定义修饰符

如此强大的功当然要支持自定义！ `gjson` 使用 `AddModifier()` 添加一个修饰符，传入一个名字和类型为 `func(json arg string) string` 的处理函数。处理函数接受待处理的 JSON 值和修饰符参数，返回处理后的结果。下面编写一个转换大小写的修饰符：

```
func main() {
  gjson.AddModifier("case", func(json, arg string) string {
    if arg == "upper" {
      return strings.ToUpper(json)
    }

    if arg == "lower" {
```

```
        return strings.ToLower(json)
      }

    return json
  })

  const json = `{"children": ["Sara", "Alex", "Jack"]}`
  fmt.Println(gjson.Get(json, "children|@case:upper"))
  fmt.Println(gjson.Get(json, "children|@case:lower"))
}
```

输出：

```
["SARA", "ALEX", "JACK"]
["sara", "alex", "jack"]
```

JSON 行

`gjson` 提供 `..` 语法可以将多行数据看成一个数组，每行数据是一个元素：

```
const json = `
{"name": "Gilbert", "age": 61}
{"name": "Alexa", "age": 34}
{"name": "May", "age": 57}
{"name": "Deloise", "age": 44}`

func main() {
  fmt.Println(gjson.Get(json, "..#"))
  fmt.Println(gjson.Get(json, "..1"))
  fmt.Println(gjson.Get(json, "..#.name"))
  fmt.Println(gjson.Get(json, `..#(name="May").age`))
}
```

- `..#`：返回有多少行 JSON 数据；
- `..1`：返回第一行，即 `{"name": "Gilbert", "age": 61}`；
- `..#.name`：`#` 后再接路径，表示对数组中每个元素读取后面的路径，将读取到的值组成一个新数组返回；`..#.name` 表示读取每一行中的 `name` 字段，最终返回 `["Gilbert","Alexa","May","Deloise"]`；
- `..#(name="May").age`：括号中的内容 `(name="May")` 表示条件，所以该条含义为取 `name` 为 `"May"` 的行中的 `age` 字段。

`gjson` 还提供了遍历 JSON 行的方法：`gjson.ForEachLine()`，参数为 JSON 串和类型为 `func(line gjson.Result) bool` 的回调函数。回调返回 `false` 时遍历停止。下面代码读取输出每一行的 `name` 字段：

```
gjson.ForEachLine(json, func(line gjson.Result) bool {
  fmt.Println("name:", gjson.Get(line.String(), "name"))
  return true
})
```

遍历

上面我们介绍了遍历 JSON 行的方式，实际上 `gjson` 还提供了通用的遍历数组和对象的方式。`gjson.Get()` 方法返回一个 `gjson.Result` 类型的对象，`json.Result` 提供了 `ForEach()` 方法用于遍历。该方法接受一个类型为 `func (key, value gjson.Result) bool` 的回调函数。遍历对象时 `key` 和 `value` 分别为对象的键和值；遍历数组时，`value` 为数组元素，`key` 为空（**不是索引**）。回调返回 `false` 时，遍历停止。

```
const json = `
{
  "name":"dj",
  "age":18,
  "pets": ["cat", "dog"],
  "contact": {
    "phone": "123456789",
    "email": "dj@example.com"
  }
}`

func main() {
  pets := gjson.Get(json, "pets")
  pets.ForEach(func(_, pet gjson.Result) bool {
    fmt.Println(pet)
    return true
  })

  contact := gjson.Get(json, "contact")
  contact.ForEach(func(key, value gjson.Result) bool {
    fmt.Println(key, value)
    return true
  })
}
```

校验 JSON

调用 `gjson.Get()` 时， `gjson` 假设我们传入的 JSON 串是合法的。如果 JSON 非法也不会 `panic` ，这时会返回不确定的结果：

```
func main() {
  const json = `{"name":dj,age:18}`
  fmt.Println(gjson.Get(json, "name"))
}
```

上面 JSON 串是非法的， `dj` 和 `age` 都没有加上双引号（实际上习惯了 Go 语言 `map` 的写法，很容易把 JSON 写成这样😭）。上面代码输出**18**，显然是错误的。我们可以使用 `gjson.Valid()` 检测 JSON 串是否合法：

```
if !gjson.Valid(json) {
  fmt.Println("error")
} else {
  fmt.Println("ok")
}
```

一次获取多个值

调用 `gjson.Get()` 一次只能读取一个值，多次调用又比较麻烦， `gjson` 提供了 `GetMany()` 可以一次读取多个值，返回一个数组 `[]gjson.Result` 。

```
const json = `
{
  "name":"dj",
  "age":18,
  "pets": ["cat", "dog"],
  "contact": {
    "phone": "123456789",
    "email": "dj@example.com"
  }
}`

func main() {
  results := gjson.GetMany(json, "name", "age", "pets.#", "contact.phone")
  for _, result := range results {
    fmt.Println(result)
  }
}
```

上面代码返回字段 `name` 、 `age` 、数组 `pets` 的长度和 `contact.phone` 字段。

总结

`gjson` 使用比较方便，功能强大，性能可观，值得一学。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gjson GitHub：https://github.com/tidwall/gjson
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

buntdb

简介

`buntdb` 是一个完全用 Go 语言编写的内存键值数据库。它支持 ACID、并发读、自定义索引和空间信息数据。 `buntdb` 只用一个源码文件就实现了这些功能，对于想要学习数据库底层知识的童鞋更是不容错过。

感谢@kiyonlin推荐！

快速使用

先安装：

```
$ go get github.com/tidwall/buntdb
```

后使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/tidwall/buntdb"
)

func main() {
  db, err := buntdb.Open(":memory:")
  if err != nil {
    log.Fatal(err)
  }
  defer db.Close()

  db.Update(func(tx *buntdb.Tx) error {
    oldValue, replaced, err := tx.Set("testkey", "testvalue", nil)
    if err != nil {
      return err
    }

    fmt.Printf("old value:%q replaced:%t\n", oldValue, replaced)
```

```
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    value, err := tx.Get("testkey")
    if err != nil {
      return err
    }

    fmt.Println("value is:", value)
    return nil
  })
}
```

`buntdb` 在使用方式上与我们熟知的 `sqlite` 有些类似，只是前者支持的是键值对，后者支持的关系型数据。首先，我们要打开一个数据库， `buntdb` 支持将数据存储到文件和内存，将数据保存在磁盘上的文件中，断电不会丢失。直接存放在内存中，程序退出后数据就丢失了。调用 `buntdb.Open()` 方法需要传入一个文件名的参数，指定数据保存的文件路径。如果传入特殊字符串 `:memory:` ，则 `buntdb` 不会将数据保存到磁盘。

在 `buntdb` 中，所有的读写操作都必须在一个事务中执行。同一时间只能存在一个写事务，但是可以同时存在多个并发的读事务。如果只需要读取数据，那么调用 `db.View()` 方法。方法接收一个类型为 `func (tx *buntdb.Tx) error` 的函数作为参数， `db.View()` 方法内部会生成一个事务对象 `tx` ，然后将这个 `tx` 作为参数传给该函数。在此函数中使用事务对象 `tx` 的 `Get()` 方法执行读取的逻辑：

```
db.View(func(tx *buntdb.Tx) error {
  value, err := tx.Get("testkey")
  if err != nil {
    return err
  }

  fmt.Println("value is:", value)
  return nil
})
```

如果需要读写数据，那么使用 `db.Update()` 方法。同样地，也需要传入一个类型为 `func (tx *buntdb.Tx) error` 的函数，在此函数中使用事务对象 `tx` 的 `Set` 方法执行写入逻辑。 `tx.Set()` 方法返回 3 个值。如果 `Set()` 替换了当前值，则返回替换之前的值和 `true` 。如果此函数返回非空错误， `db.Update()` 会回退此前所做的修改，反之会提交此次修改。

如果运行两次上面的程序，我们会看到下面的输出：

```
// 第一次运行
$ go run main.go
old value:"" replaced:false
value is: testvalue

// 第二次运行
$ go run main.go
old value:"testvalue" replaced:true
value is: testvalue
```

注意：

- 数据库操作很容易出错，所以基本上所有的方法都会返回错误，在实际中需要处理每个可能的错误。示例中为了代码简洁，有点地方忽略了；
- 在传入 `db.View()` 和 `db.Update()` 的函数中不要直接使用 `db` 对象，否则可能会导致程序死锁；
- 默认情况下，若键对应的值不存在，则返回 `ErrNotFound` 错误。

遍历

`buntdb` 中存储的数据是根据键排序的，我们可以按顺序依次遍历这些数据。由于遍历是读取操作，我们用 `db.View()` 方法。 `buntdb` 提供了很多遍历的方法，基本形式都差不多，这里只介绍一个基本的 `Ascend()` 方法：

```
func (tx *Tx) Ascend(index string, iterator func(key, value string) bool) error
```

`Ascend()` 方法接收一个索引名，然后以该索引定义的顺序遍历所有键值对，将遍历到的键值对传给 `iterator` 函数处理，如果 `iterator` 返回 `false` ，终止遍历。另外，如果未指定索引名，则根据键升序遍历：

```
func main() {
  db, err := buntdb.Open(":memory:")
  if err != nil {
    log.Fatal(err)
  }
  defer db.Close()

  db.Update(func(tx *buntdb.Tx) error {
    data := map[string]string{
      "a": "apple",
      "b": "banana",
```

```
        "p": "pear",
        "o": "orange",
    }
    for key, value := range data {
        tx.Set(key, value, nil)
    }
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    var count int
    tx.Ascend("", func(key, value string) bool {
      fmt.Printf("key:%s value:%s\n", key, value)
      count++
      if count >= 3 {
        return false
      }
      return true
    })
    return nil
  })
}
```

上面代码中，我们按键升序遍历（因为传入索引名为 `""` ），在处理完第三个键值对后， `iterator` 函数返回 `false` ，停止遍历。最终输出：

```
key:a value:apple
key:b value:banana
key:o value:orange
```

索引

`buntdb` 将所有数据都存储在一个**B-tree**中，每组数据都有一个键和值。所有数据是根据键来排序的。我们也可以创建自定义索引，这样就可以对值进行排序了。创建索引需要调用 `db.CreateIndex()` 方法，该方法签名如下：

```
func (db *DB) CreateIndex(name, pattern string, less ...func(a, b string) bool)
 error
```

`name` 为索引名，在上一节介绍遍历的时候，我们说过遍历时需要传入索引名，以便按照该索引所定义的顺序遍历。 `pattern` 为模式，指定索引对哪些键生效，可以只对某些特定模式的键创建索引。 `*` 表示所有键， `user:*:name` 表示键名是 `user:` 和 `:name` 之间有任意字符的键。通过 `less` 函数，我们可以自定义排序

规则。 `buntdb` 内置了一些排序规则，如 `IndexString` 对值进行大小写不敏感的排序， `IndexInt/IndexUint/IndexFloat` 执行数值类型的排序。

```
func main() {
  db, err := buntdb.Open(":memory:")
  if err != nil {
    log.Fatal(err)
  }
  defer db.Close()

  db.CreateIndex("names", "user:*:name", buntdb.IndexString)
  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1:name", "tom", nil)
    tx.Set("user:2:name", "Randi", nil)
    tx.Set("user:3:name", "jane", nil)
    tx.Set("user:4:name", "Janet", nil)
    tx.Set("user:5:name", "Paula", nil)
    tx.Set("user:6:name", "peter", nil)
    tx.Set("user:7:name", "Terri", nil)
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    tx.Ascend("names", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })
    return nil
  })
}
```

我们先为键名满足模式 `user:*:name` 的数据创建一个名为 `names` 的索引，执行大小写不敏感的排序（ `buntdb.IndexString` ）。然后向 `buntdb` 中写入几组数据。最后，我们使用 `Ascend()` 方法，传入索引名 `names` 按该索引指定次序遍历键值对（这里只是遍历满足模式 `user:*:name` 的键值对）。

如果我们的键只有 `user:*:name` 这种模式的，也可以直接使用模式 `*` 或 `user:*`

对于整数等非字符串类型的排序，我们需要注意一点：因为 `buntdb` 存储的键值都是字符串，所以自定义的排序函数需要执行相应的类型转换。一般需求的数值排序，内置函数就可以满足要求了：

```
func main() {
  db, err := buntdb.Open(":memory:")
  if err != nil {
```

```
    log.Fatal(err)
  }
  defer db.Close()

  db.CreateIndex("ages", "user:*:age", buntdb.IndexInt)
  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1:age", "16", nil)
    tx.Set("user:2:age", "35", nil)
    tx.Set("user:3:age", "24", nil)
    tx.Set("user:4:age", "32", nil)
    tx.Set("user:5:age", "25", nil)
    tx.Set("user:6:age", "28", nil)
    tx.Set("user:7:age", "31", nil)
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    tx.Ascend("ages", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })
    return nil
  })
}
```

首先，为键名满足 `user:*:age` 的键创建索引 `ages` ，因为在这些键对应的值中，我们存储的都是年龄（整数），故使用排序规则 `IndexInt` 。

JSON 索引

`buntdb` 提供了强大的 JSON 索引功能。如果存储的值是一个 JSON 字符串， `buntdb` 可以对 JSON 串内部的键创建索引。 `buntdb.IndexJSON()` 实现了 JSON 索引的排序规则，我们需要传入键在 JSON 内部的路径，如 `name.first` ， `contact.email` 等：

```
func main() {
  db, _ := buntdb.Open(":memory:")
  defer db.Close()

  db.CreateIndex("first_name", "user:*", buntdb.IndexJSON("name.first"))
  db.CreateIndex("age", "user:*", buntdb.IndexJSON("age"))
  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1", `{"name":{"first":"zhang","last":"san"},"age":18}`, nil)
    tx.Set("user:2", `{"name":{"first":"li","last":"si"},"age":27`, nil)
    tx.Set("user:3", `{"name":{"first":"wang","last":"wu"},"age":32}`, nil)
```

```
    tx.Set("user:4", `{"name":{"first":"sun","last":"qi"},"age":8}`, nil)
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    fmt.Println("Order by first name")
    tx.Ascend("first_name", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })

    fmt.Println("Order by age")
    tx.Ascend("age", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })

    fmt.Println("Order by age range 18-30")
    tx.AscendRange("age", `{"age":18}`, `{"age":30}`, func(key, value string) bo
ol {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })
    return nil
  })
}
```

JSON 给我们提供了一种很好的存储用户数据的格式。以 `user:` 后加上用户 ID 作为键名，用户数据以 JSON 格式存储在值中，如上所示。

我们分别为 JSON 内部的键 `name.first` 和 `age` 创建索引。然后分别以 `name.first` 和 `age` 定义的顺序遍历输出。值得一提的是最后一个遍历使用了 `AscendRange` ，可以只遍历指定范围内的数据，例子中为年龄在 18~30 之间。范围遍历并非 JSON 索引独有的，与普通的 `Ascend` 相比， `AscendRange` 需要传入区间上下限 `min` 和 `max` ，所有处于 `[min, max)` 之间的数据都会被遍历到（注意不包含 `max` ）。

多重索引

细节的盆友应该发现了，创建索引的方法 `CreateIndex()` 接受可变数量的排序规则函数，如果第一个函数无法判断两个值的大小，则继续使用后一个函数，直到可以判断或没有其他函数了。这个就是多重索引。在上面的示例中，我们可以将 `first_name` 和 `age` 两个索引放在一起，先对 `name.first` 比较，如果相等，再比较 `age` ：

```
func main() {
  db, _ := buntdb.Open(":memory:")
  defer db.Close()

  db.CreateIndex("first_name_age", "user:*", buntdb.IndexJSON("name.first"), buntdb.IndexJSON("age"))
  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1", `{"name":{"first":"zhang","last":"san"},"age":18}`, nil)
    tx.Set("user:2", `{"name":{"first":"li","last":"si"},"age":27`, nil)
    tx.Set("user:3", `{"name":{"first":"wang","last":"wu"},"age":30}`, nil)
    tx.Set("user:4", `{"name":{"first":"sun","last":"qi"},"age":8}`, nil)
    tx.Set("user:5", `{"name":{"first":"li", "name":"dajun"},"age":20}`, nil)
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    tx.Ascend("first_name_age", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })
    return nil
  })
}
```

由于 `user:2` 和 `user:5` 的 `name.first` 都是 `li` ，相等。故使用 `age` 的值排序，所以输出中 `user:5` 在 `user:2` 前面。

降序

我们使用的内置函数都是升序规则。可以使用 `buntdb.Desc()` 将升序规则变为降序，拿前面整数排序的例子来说，只需要将 `buntdb.IndexInt` 变为 `buntdb.Desc(buntdb.IndexInt)` 即可：

```
func main() {
  db, _ := buntdb.Open(":memory:")
  defer db.Close()

  db.CreateIndex("ages", "user:*:age", buntdb.Desc(buntdb.IndexInt))
  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1:age", "16", nil)
    tx.Set("user:2:age", "35", nil)
    tx.Set("user:3:age", "24", nil)
    tx.Set("user:4:age", "32", nil)
    tx.Set("user:5:age", "25", nil)
```

```
    tx.Set("user:6:age", "28", nil)
    tx.Set("user:7:age", "31", nil)
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    tx.Ascend("ages", func(key, value string) bool {
      fmt.Printf("%s: %s\n", key, value)
      return true
    })
    return nil
  })
}
```

过期

在向 `buntdb` 中设置键值时，我们可以通过选项 `buntdb.SetOptions` 指定过期时间，超过这个时间数据会自动从 `buntdb` 中移除。如果想要移除过期时间，重新使用 `nil` 选项设置该键值即可：

```
func main() {
  db, _ := buntdb.Open(":memory:")
  defer db.Close()

  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("testkey", "testvalue", &buntdb.SetOptions{Expires: true, TTL: time.S
econd})
    return nil
  })

  db.View(func(tx *buntdb.Tx) error {
    value, _ := tx.Get("testkey")
    fmt.Println("value is:", value)
    return nil
  })

  time.Sleep(time.Second)

  db.View(func(tx *buntdb.Tx) error {
    value, _ := tx.Get("testkey")
    fmt.Println("value is:", value)
    return nil
  })
}
```

上面例子中，我们先写入数据，并设置过期时间为 `1s` 。然后立刻读取，这时可以读到刚刚设置的值。然后 `Sleep` 1s 之后再次读取，读到空值，说明已被删除：

```
value is: testvalue
value is:
```

杂项

遍历时删除

`buntdb` 不支持遍历时删除数据，一般迂回的做法是先记录需要删除的键，遍历结束后统一删除。下面将年龄 >= 30 的用户删掉（嗯，程序员年龄大了，干不动了）：

```
func main() {
  db, _ := buntdb.Open(":memory:")
  defer db.Close()

  db.Update(func(tx *buntdb.Tx) error {
    tx.Set("user:1:age", "16", nil)
    tx.Set("user:2:age", "35", nil)
    tx.Set("user:3:age", "24", nil)
    tx.Set("user:4:age", "32", nil)
    tx.Set("user:5:age", "25", nil)
    tx.Set("user:6:age", "28", nil)
    tx.Set("user:7:age", "31", nil)
    return nil
  })

  db.Update(func(tx *buntdb.Tx) error {
    // 先汇总
    deleteKeys := make([]string, 0)
    tx.Ascend("", func(key, value string) bool {
      age, _ := strconv.ParseUint(value, 10, 64)
      if age >= 30 {
        deleteKeys = append(deleteKeys, key)
      }
      return true
    })

    // 再删除
    for _, key := range deleteKeys {
      tx.Delete(key)
    }
    return nil
```

```
    })

    db.View(func(tx *buntdb.Tx) error {
        tx.Ascend("", func(key, value string) bool {
            fmt.Printf("%s: %s\n", key, value)
            return true
        })
        return nil
    })
}
```

Web 服务

`buntdb` 只能在本地程序中操作，我们简单为它编写一个 Web 服务，可以通过 HTTP 请求操作远程的 `buntdb` 。代码如下：

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "strconv"
    "time"

    "github.com/tidwall/buntdb"
)

var db *buntdb.DB

func init() {
    var err error
    db, err = buntdb.Open("data.db")
    if err != nil {
        log.Fatal(err)
    }
}

func response(w http.ResponseWriter, err error, data interface{}) {
    bytes, _ := json.Marshal(map[string]interface{}{
        "error": err,
        "data":  data,
    })
```

```
	w.Write(bytes)
}

func set(w http.ResponseWriter, r *http.Request) {
	key := r.FormValue("key")
	value := r.FormValue("value")
	expire, _ := strconv.ParseBool(r.FormValue("expire"))
	ttl, _ := time.ParseDuration(r.FormValue("ttl"))

	var setOption *buntdb.SetOptions
	if expire && ttl > 0 {
		setOption = &buntdb.SetOptions{Expires: true, TTL: ttl}
	}

	err := db.Update(func(tx *buntdb.Tx) error {
		_, _, err := tx.Set(key, value, setOption)
		return err
	})

	response(w, err, nil)
}

func get(w http.ResponseWriter, r *http.Request) {
	key := r.FormValue("key")

	var value string
	err := db.View(func(tx *buntdb.Tx) error {
		var err error
		value, err = tx.Get(key)
		return err
	})

	response(w, err, value)
}

type Pair struct {
	Key   string
	Value string
}

func iterate(w http.ResponseWriter, r *http.Request) {
	index := r.FormValue("index")
	fmt.Println(index)

	var items []Pair
	err := db.View(func(tx *buntdb.Tx) error {
```

```
    err := tx.Ascend(index, func(key, value string) bool {
      fmt.Println(key, value)
      items = append(items, Pair{key, value})
      return true
    })
    return err
  })

  response(w, err, items)
}

func createIndex(w http.ResponseWriter, r *http.Request) {
  name := r.FormValue("name")
  pattern := r.FormValue("pattern")
  less := buntdb.IndexString

  err := db.CreateIndex(name, pattern, less)
  response(w, err, nil)
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/get", get)
  mux.HandleFunc("/set", set)
  mux.HandleFunc("/iterate", iterate)
  mux.HandleFunc("/create_index", createIndex)

  server := &http.Server{
    Addr:    ":8000",
    Handler: mux,
  }

  if err := server.ListenAndServe(); err != nil {
    log.Fatal(err)
  }
}
```

我只编写了基本读取、设置、创建索引和遍历的功能，代码并不难理解。下面我们先运行程序，然后用浏览器请求：

请求 `localhost:8000/set?key=name&value=dj` ，返回：

```
{"error":null, "data":null}
```

`error` 为 `null` 表示无错误。

请求 `localhost:8000/set?key=dj&value=18` ，返回：

```
{"error":null, "data":null}
```

请求 `localhost:8000/iterate` ，返回：

```
{
  "data": [
    {
      "Key": "age",
      "Value": "18"
    },
    {
      "Key": "name",
      "Value": "dj"
    }
  ],
  "error": null
}
```

感兴趣可以试着添加更多的功能。如果对 Go Web 编程不太了解，可以去看看我的Go Web 编程系列文章。

总结

本文介绍 `buntdb` 的读取、写入、创建索引等基本操作，最后编写一个简单的 web 服务可以在远程运行，其他程序通过 HTTP 与之交互。 `buntdb` 还支持空间索引等高级特性，感兴趣可自行研究。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. buntdb GitHub：https://github.com/tidwall/buntdb
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

go-cmp

简介

我们时常有比较两个值是否相等的需求，最直接的方式就是使用 `==` 操作符，其实 `==` 的细节远比你想象的多，我在深入理解 Go 之 `==` 中有详细介绍，有兴趣去看看。但是直接用 `==` ，一个最明显的弊端就是对于指针，只有两个指针指向同一个对象时，它们才相等，不能进行递归比较。为此， `reflect` 包提供了一个 `DeepEqual` ，它可以进行递归比较。但是相对的， `reflect.DeepEqual` 不够灵活，无法提供选项实现我们想要的行为，例如允许浮点数误差。所以今天的主角 `go-cmp` 登场了。 `go-cmp` 是 Google 开源的比较库，它提供了丰富的选项。**最初定位是用在测试中。**

感谢thinkgos的推荐！

快速使用

先安装：

```
$ go get github.com/com/google/go-cmp/cmp
```

后使用：

```
package main

import (
  "fmt"

  "github.com/google/go-cmp/cmp"
)

type Contact struct {
  Phone string
  Email string
}

type User struct {
  Name    string
  Age     int
  Contact *Contact
}
```

```go
func main() {
  u1 := User{Name: "dj", Age: 18}
  u2 := User{Name: "dj", Age: 18}

  fmt.Println("u1 == u2?", u1 == u2)
  fmt.Println("u1 equals u2?", cmp.Equal(u1, u2))

  c1 := &Contact{Phone: "123456789", Email: "dj@example.com"}
  c2 := &Contact{Phone: "123456789", Email: "dj@example.com"}

  u1.Contact = c1
  u2.Contact = c1
  fmt.Println("u1 == u2 with same pointer?", u1 == u2)
  fmt.Println("u1 equals u2 with same pointer?", cmp.Equal(u1, u2))

  u2.Contact = c2
  fmt.Println("u1 == u2 with different pointer?", u1 == u2)
  fmt.Println("u1 equals u2 with different pointer?", cmp.Equal(u1, u2))
}
```

上面的例子中，我们将 `==` 与 `cmp.Equal` 放在一起做个比较：

- 在指针类型的字段 `Contact` 未设置时， `u1 == u2` 和 `cmp.Equal(u1, u2)` 都返回 `true` ；
- 两个结构的 `Contact` 字段都指向同一个对象时， `u1 == u2` 和 `cmp.Equal(u1, u2)` 都返回 `true` ；
- 两个结构的 `Contact` 字段指向不同的对象时，尽管这两个对象包含相同的内容， `u1 == u2` 也返回了 `false` 。而 `cmp.Equal(u1, u2)` 可以比较指针指向的内容，从而返回 `true` 。

以下是运行结果：

```
u1 == u2? true
u1 equals u2? true
u1 == u2 with same pointer? true
u1 equals u2 with same pointer? true
u1 == u2 with different pointer? false
u1 equals u2 with different pointer? true
```

高级选项

未导出字段

默认情况下， `cmp.Equal()` 函数不会比较未导出字段（即字段名首字母小写的字段）。遇到未导出字段， `cmp.Equal()` 直接 `panic` 。这一点与 `reflect.DeepEqual()` 有所不同，后者也会比较未导出的字段。

我们可以使用 `cmdopts.IgnoreUnexported` 选项忽略未导出字段，也可以使用 `cmdopts.AllowUnexported` 选项指定某些类型的未导出字段需要比较。

```go
package main

import (
  "fmt"

  "github.com/google/go-cmp/cmp"
)

type Contact struct {
  Phone string
  Email string
}

type User struct {
  Name    string
  Age     int
  contact *Contact
}

func main() {
  c1 := &Contact{Phone: "123456789", Email: "dj@example.com"}
  c2 := &Contact{Phone: "123456789", Email: "dj@example.com"}

  u1 := User{"dj", 18, c1}
  u2 := User{"dj", 18, c2}

  fmt.Println("u1 equals u2?", cmp.Equal(u1, u2))
}
```

运行上面的代码会 `panic` ，因为 `cmd.Equal()` 比较的类型中有未导出字段 `contact` 。我们先使用 `cmdopts.IngoreUnexported` 忽略未导出字段：

```go
fmt.Println("u1 equals u2?", cmp.Equal(u1, u2, cmpopts.IgnoreUnexported(User{})))
```

我们在 `cmp.Equal()` 的调用中添加了选项 `cmpopts.IgnoreUnexported` ，选项参数传入 `User{}` ，**表示忽略 `User` 的直接未导出字段。导出字段中的未导出字段是不会被忽略的，除非显示指定该类型。**如果我们将 `User` 稍作修改：

```
type Address struct {
  Province string
  city     string
}

type User struct {
  Name    string
  Age     int
  Address Address
}

func main() {
  u1 := User{"dj", 18, Address{}}
  u2 := User{"dj", 18, Address{}}

  fmt.Println("u1 equals u2?", cmp.Equal(u1, u2, cmpopts.IgnoreUnexported(User
{})))
}
```

注意， `city` 字段未导出，这种情况下，使用 `cmpopts.IngoreUnexported(User{})` 还是会 `panic` ，因为 `city` 是 `Address` 中的未导出字段，而非 `User` 的直接字段。

我们也可以使用 `cmdopts.AllowUnexported(User{})` 表示需要比较 `User` 的未导出字段：

```
fmt.Println("u1 equals u2?", cmp.Equal(u1, u2, cmp.AllowUnexported(User{})))
```

浮点数比较

我们知道，计算机中浮点数的表示是不精确的，如果涉及到运算，可能会产生误差累计。此外，还有一个特殊的浮点数 `NaN` （Not a Number），**它与任何浮点数都不等，包括它自己**。这样，有时候会出现一些反直觉的结果：

```
package main

import (
  "fmt"
  "math"
```

```go
	"github.com/google/go-cmp/cmp"
)

type FloatPair struct {
	X float64
	Y float64
}

func main() {
	p1 := FloatPair{X: math.NaN()}
	p2 := FloatPair{X: math.NaN()}
	fmt.Println("p1 equals p2?", cmp.Equal(p1, p2))

	f1 := 0.1
	f2 := 0.2
	f3 := 0.3
	p3 := FloatPair{X: f1 + f2}
	p4 := FloatPair{X: f3}
	fmt.Println("p3 equals p4?", cmp.Equal(p3, p4))

	p5 := FloatPair{X: 0.1 + 0.2}
	p6 := FloatPair{X: 0.3}
	fmt.Println("p5 equals p6?", cmp.Equal(p5, p6))
}
```

运行程序，输出：

```
p1 equals p2? false
p3 equals p4? false
p5 equals p6? true
```

是不是很反直觉？ `NaN` 不等于 `NaN` ， `0.1 + 0.2` 竟然不等于 `0.3` ！前者是由于标准的规定，后者是浮点数的表示不精确导致的计算误差。

奇怪的是第三组表示，为什么直接用字面量运算就不会导致误差呢？实际上，在 Go 语言中这些字面量的运算直接是在编译器完成的，可以做到精确。如果先赋值给浮点类型的变量，就像第 2 组所示，受限于变量的存储空间，就会存在误差。

关于这一点，我这里再顺带介绍一个知识点。我们都知道使用 `const` 定义常量时可以不指定类型，这种常量被称为无类型的常量，它的值可以超出正常数值的表示范围，可以相互进行的运算。**只是不能赋值给超过其类型表示范围的普通变量**：

```go
package main

import "fmt"
```

```go
const (
	_  = 1 << (10 * iota)
	KB // 1024
	MB // 1048576
	GB // 1073741824
	TB // 1099511627776
	PB // 1125899906842624
	EB // 1152921504606846976
	ZB // 1180591620717411303424
	YB // 1208925819614629174706176
)

func main() {
	// constant 1180591620717411303424 overflows int
	// fmt.Println(ZB)

	// constant 1208925819614629174706176 overflows uint64
	// var mem uint64 = YB

	fmt.Println(YB / ZB)
}
```

后面 `ZB` 和 `YB` 都已经超出了 `uint64` 的表示范围。直接使用时，如 `fmt.Println(ZB)` 编译器会自动将其转为 `int` 类型，但是它的值超出了 `int` 的表示范围，所以编译报错。赋值时也是如此。

`go-cmp` 提供比较浮点数的选项，我们希望两个 `NaN` 的比较返回 `true` ，两个浮点数相差不超过一定范围就认为它们相等：

- `cmpopts.EquateNaNs()` ：两个 `NaN` 比较，返回 `true` ；
- `cmpopts.EquateApprox(fraction, margin)` ：这个选项有两个参数，第二个参数比较好理解，如果两个浮点数的差的绝对值小于 `margin` 则认为它们相等。第一个参数的含义是取两个数绝对值的较小者，乘以 `fraction` ，如果两个数的差的绝对值小于这个数即 `|x-y| ≤ max(fraction*min(|x|, |y|), margin)` ，则认为它们相等。如果 `fraction` 和 `margin` 同时设置，**只需要满足一个就行了**。

例如：

```go
type FloatPair struct {
	X float64
	Y float64
}
```

```
func main() {
  p1 := FloatPair{X: math.NaN()}
  p2 := FloatPair{X: math.NaN()}
  fmt.Println("p1 equals p2?", cmp.Equal(p1, p2, cmpopts.EquateNaNs()))

  f1 := 0.1
  f2 := 0.2
  f3 := 0.3
  p3 := FloatPair{X: f1 + f2}
  p4 := FloatPair{X: f3}
  fmt.Println("p3 equals p4?", cmp.Equal(p3, p4, cmpopts.EquateApprox(0.1, 0.00
1)))
}
```

运行输出：

```
p1 equals p2? true
p3 equals p4? true
```

Nil

默认情况下，如果一个切片变量值为 `nil` ，另一个是使用 `make` 创建的长度为 0 的切片，那么 `go-cmp` 认为它们是不等的。同样的，一个 `map` 变量值为 `nil` ，另一个是使用 `make` 创建的长度为 0 的 `map` ，那么 `go-cmp` 也认为它们不等。我们可以指定 `cmpopts.EquateEmpty` 选项，让 `go-cmp` 认为它们相等：

```
func main() {
  var s1 []int
  var s2 = make([]int, 0)

  var m1 map[int]int
  var m2 = make(map[int]int)

  fmt.Println("s1 equals s2?", cmp.Equal(s1, s2))
  fmt.Println("m1 equals m2?", cmp.Equal(m1, m2))

  fmt.Println("s1 equals s2 with option?", cmp.Equal(s1, s2, cmpopts.EquateEmpty
()))
  fmt.Println("m1 equals m2 with option?", cmp.Equal(m1, m2, cmpopts.EquateEmpty
()))
}
```

切片

默认情况下，两个切片只有当长度相同，且对应位置上的元素都相等时， `go-cmp` 才认为它们相等。如果，我们想要实现无序切片的比较（即只要两个切片包含相同的值就认为它们相等），可以使用 `cmpopts.SortedSlice` 选项先对切片进行排序，然后再进行比较：

```
func main() {
  s1 := []int{1, 2, 3, 4}
  s2 := []int{4, 3, 2, 1}
  fmt.Println("s1 equals s2?", cmp.Equal(s1, s2))
  fmt.Println("s1 equals s2 with option?", cmp.Equal(s1, s2, cmpopts.SortSlices
(func(i, j int) bool { return i < j })))

  m1 := map[int]int{1: 10, 2: 20, 3: 30}
  m2 := map[int]int{1: 10, 2: 20, 3: 30}
  fmt.Println("m1 equals m2?", cmp.Equal(m1, m2))
  fmt.Println("m1 equals m2 with option?", cmp.Equal(m1, m2, cmpopts.SortMaps(fu
nc(i, j int) bool { return i < j })))
}
```

对于 `map` 来说，由于本身就是无序的，所以 `map` 比较差不多是下面这种形式。没有上面的顺序问题：

```
func compareMap(m1, m2 map[int]int) bool {
  if len(m1) != len(m2) {
    return false
  }

  for k, v := range m1 {
    if v != m2[k] {
      return false
    }
  }

  return true
}
```

`cmpopts.SortMaps` 会将 `map[K]V` 类型按照键排序，生成一个 `[]struct{K, V}` 的切片，然后逐个比较。

`SortSlices` 和 `SortMaps` 都需要提供一个比较函数 `less` ，函数必须是 `func(T, T) bool` 这种形式，切片的元素类型必须可以赋值给 `T` 类型， `map` 的键也必须可以赋值给 `T` 类型。

自定义 `Equal` 方法

对于有些类型来说， `go-cmp` 内置的比较结果不符合我们的要求，这时我们可以自定义 `Equal` 方法来比较该类型。例如我们想要表示 `IP` 地址的字符串比较时 `127.0.0.1` 与 `localhost` 相等：

```
package main

type NetAddr struct {
  IP   string
  Port int
}

func (a NetAddr) Equal(b NetAddr) bool {
  if a.Port != b.Port {
    return false
  }

  if a.IP != b.IP {
    if a.IP == "127.0.0.1" && b.IP == "localhost" {
      return true
    }

    if a.IP == "localhost" && b.IP == "127.0.0.1" {
      return true
    }

    return false
  }

  return true
}

func main() {
    a1 := NetAddr{"127.0.0.1", 5000}
    a2 := NetAddr{"localhost", 5000}
    a3 := NetAddr{"192.168.1.1", 5000}

    fmt.Println("a1 equals a2?", cmp.Equal(a1, a2))
    fmt.Println("a1 equals a3?", cmp.Equal(a1, a3))
}
```

很简单，只需要给想要自定义比较操作的类型提供一个 `Equal()` 方法即可，方法接受该类型的参数，返回一个 `bool` 表示是否相等。如果我们将上面的 `Equal()` 方法注释掉，那么比较输出都是 `false` 。

自定义比较器

如果 `go-cmp` 默认的行为无法满足我们的需求，我们可以针对某些类型自定义比较器。我们使用 `cmp.Comparer()` 传入比较函数，比较函数必须是 `func (T, T) bool` 这种形式。所有能转为 `T` 类型的值，都会调用该函数进行比较。所以如果 `T` 是接口类型，那么可能传给比较函数的参数的实际类型并不相同，只是它们都实现了 `T` 接口。我们使用 `Comparer()` 重构一下上面的程序：

```
type NetAddr struct {
  IP   string
  Port int
}

func compareNetAddr(a, b NetAddr) bool {
  if a.Port != b.Port {
    return false
  }

  if a.IP != b.IP {
    if a.IP == "127.0.0.1" && b.IP == "localhost" {
      return true
    }

    if a.IP == "localhost" && b.IP == "127.0.0.1" {
      return true
    }

    return false
  }

  return true
}

func main() {
  a1 := NetAddr{"127.0.0.1", 5000}
  a2 := NetAddr{"localhost", 5000}

  fmt.Println("a1 equals a2?", cmp.Equal(a1, a2))
  fmt.Println("a1 equals a2 with comparer?", cmp.Equal(a1, a2, cmp.Comparer(compareNetAddr)))
}
```

这种方式与上面介绍的自定义 `Equal()` 方法有些类似，但更灵活。有时，我们要自定义比较操作的类型定义在第三方包中，这样就无法给它定义 `Equal` 方法。这时，我们就可以采

用自定义 `Comparer` 的方式。

`Exporter`

从前面的介绍我们知道默认情况下，未导出字段会导致 `cmp.Equal()` 直接 `panic` 。前面也介绍过两种方式处理未导出字段，这里再介绍一种方式—— `cmp.Exporter` 。通过传入一个函数 `func (t reflec.Type) bool` ，返回传入的类型是否比较其未导出字段。例如，下面代码中，我们指定需要比较类型 `User` 的未导出字段：

```
type Contact struct {
  Phone string
  Email string
}

type User struct {
  Name    string
  Age     int
  contact Contact
}

func allowUnExportedInType(t reflect.Type) bool {
  if t.Name() == "User" {
    return true
  }

  return false
}

func main() {
  c1 := Contact{Phone: "123456789", Email: "dj@example.com"}
  c2 := Contact{Phone: "123456789", Email: "dj@example.com"}

  u1 := User{"dj", 18, c1}
  u2 := User{"dj", 18, c2}

  fmt.Println("u1 equals u2?", cmp.Equal(u1, u2, cmp.Exporter(allowType)))
}
```

`cmp.Exporter` 的使用不多，且可以通过 `AllowUnexported` 选项来实现。

转换器

转换器可以将特定类型的值转为另一种类型的值。转换器有很多用法，下面介绍两种。

忽略字段

如果我们想忽略结构中的某些字段，我们可以定义转换，返回一个不设置这些字段的对象：

```
type User struct {
  Name string
  Age  int
}

func omitAge(u User) string {
  return u.Name
}

type User2 struct {
  Name    string
  Age     int
  Email   string
  Address string
}

func omitAge2(u User2) User2 {
  return User2{u.Name, 0, u.Email, u.Address}
}

func main() {
  u1 := User{Name: "dj", Age: 18}
  u2 := User{Name: "dj", Age: 28}

  fmt.Println("u1 equals u2?", cmp.Equal(u1, u2, cmp.Transformer("omitAge", omit
Age)))

  u3 := User2{Name: "dj", Age: 18, Email: "dj@example.com"}
  u4 := User2{Name: "dj", Age: 28, Email: "dj@example.com"}

  fmt.Println("u3 equals u4?", cmp.Equal(u3, u4, cmp.Transformer("omitAge", omit
Age2)))
}
```

如果一个类型，我们只关心一个字段，忽略其它字段，那么直接返回这个字段就行了，如上面的 `omitAge` 。如果该类型有多个字段，我们只忽略很少的字段，我们要返回一个同样的类型，不设置忽略的字段即可，如上面的 `omitAge2` 。

转换值

上面我们介绍了如何使用自定义 `Equal()` 方法和 `Comparer` 比较器的方式来实现 IP 地址的比较。实际上转换器也可以实现同样的效果，我们可以将 `localhost` 转换为 `127.0.0.1` ：

```
type NetAddr struct {
  IP   string
  Port int
}

func transformLocalhost(a NetAddr) NetAddr {
  if a.IP == "localhost" {
    return NetAddr{IP: "127.0.0.1", Port: a.Port}
  }

  return a
}

func main() {
  a1 := NetAddr{"127.0.0.1", 5000}
  a2 := NetAddr{"localhost", 5000}

  fmt.Println("a1 equals a2?", cmp.Equal(a1, a2, cmp.Transformer("localhost", transformLocalhost)))
}
```

遇到 `IP` 为 `localhost` 的对象，将其转换为 `IP` 为 `127.0.0.1` 的对象。

`Diff`

除了能比较两个值是否相等， `go-cmp` 还能汇总两个值的不同之处，方便我们查看。上面介绍的选项都可以用在 `Diff` 中：

```
type Contact struct {
  Phone string
  Email string
}

type User struct {
  Name    string
  Age     int
  Contact *Contact
}

func main() {
  c1 := &Contact{Phone: "123456789", Email: "dj@example.com"}
```

```
  c2 := &Contact{Phone: "123456879", Email: "dj2@example.com"}
  u1 := User{Name: "dj", Age: 18, Contact: c1}
  u2 := User{Name: "dj2", Age: 18, Contact: c2}

  fmt.Println(cmp.Diff(u1, u2))
}
```

我们着重介绍一下输出的格式：

```
  main.User{
-   Name: "dj",
+   Name: "dj2",
    Age:  18,
    Contact: &main.Contact{
-     Phone: "123456789",
+     Phone: "123456879",
-     Email: "dj@example.com",
+     Email: "dj2@example.com",
    },
  }
```

相信使用过 SVN 或对 Linux 的 `diff` 命令熟悉的童鞋对上面的格式应该不会陌生。我们可以这样认为，第一个对象为原来的版本，第二个对象为新的版本。这样上面的输出我们可以想象成如何将对象从原来的版本变为新版本。没有前缀的行不需要改变，前缀为 `-` 的行表示新版本删除了这一行，前缀 `+` 表示新版本增加了这一行。

总结

`go-cmp` 库大大地方便两个值的比较操作。源码中大量使用我们之前介绍过的**选项模式**，提供给使用者简洁、一致的接口。这种设计思想也值得我们学习、借鉴。本文介绍了这是 `go-cmp` 的一部分内容，还有一些特性如**过滤器**感兴趣可自行探索。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. go-cmp GitHub：https://github.com/google/go-cmp
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

jennifer

简介

今天我们介绍一个 Go 代码生成库 `jennifer`。`jennifer` 支持所有的 Go 语法和特性，可以用它来生成任何 Go 语言代码。

感谢kiyonlin的推荐！

快速使用

先安装：

```
$ go get github.com/dave/jennifer
```

今天我们换个思路来介绍 `jennifer` 这个库。既然，它是用来生成 Go 语言代码的。我们就先写出想要生成的程序，然后看看如何使用 `jennifer` 来生成。先从第一个程序 `Hello World` 开始：

```
package main

import "fmt"

func main() {
  fmt.Println("Hello World")
}
```

我们如何用 `jennifer` 来生成上面的程序代码呢：

```
package main

import (
  "fmt"

  . "github.com/dave/jennifer/jen"
)

func main() {
  f := NewFile("main")
  f.Func().Id("main").Params().Block(
```

```
    Qual("fmt", "Println").Call(Lit("Hello, world")),
  )
  fmt.Printf("%#v", f)
}
```

Go 程序的基本组织单位是文件，每个文件属于一个包，可执行程序必须有一个 `main` 包， `main` 包中又必须有一个 `main` 函数。所以，我们使用 `jennifer` 生成代码的大体步骤也是差不多的：

- 先使用 `NewFile` 定义一个包文件对象，参数即为包名；
- 然后调用这个包文件对象的 `Func()` 定义函数；
- 函数可以使用 `Params()` 定义参数，通过 `Block()` 定义函数体；
- 函数体是由一条一条语句组成的。语句的内容比较多样，后面会详细介绍。

上面代码中，我们首先定义一个 `main` 包文件对象。然后调用其 `Func()` 定义一个函数， `Id()` 为函数命名为 `main` ， `Params()` 传入空参数， `Block()` 中传入函数体。函数体中，我们使用 `Qual("fmt", "Println")` 来表示引用 `fmt.Println` 这个函数。使用 `Call()` 表示函数调用，然后 `Lit()` 将字符串**“Hello World”**字面量作为参数传给 `fmt.Println()` 。

`Qual` 函数这里需要特意讲一下，我们不需要显示导入包， `Qual` 函数的第一个参数就是包路径。如果是标准库，直接就是包名，例如这里的 `fmt` 。如果是第三方库，需要加上包路径限定，例如 `github.com/spf13/viper` 。这也是 `Qual` 名字的由来（ `Qualified` ，想到**full qualified class name**了吗😄）。 `jennifer` 在生成程序时会汇总所有用到的包，统一导入。

运行程序，我们最终输出了一开始想要生成的程序代码！

实际上，大多数编程语言的语法都有相通之处。Go 语言的语法比较简单：

- 基本的概念：变量、函数、结构等，它们都有一个名字，又称为标识符。直接写在程序中的数字、字符串等被称为字面量，如上面的**“Hello World”**；
- 流程控制：条件（ `if` ）、循环（ `for` ）；
- 函数和方法；
- 并发相关：goroutine 和 channel。

有几点注意：

- 我们在导入 `jennifer` 包的时候在包路径前面加了一个 `.` ，使用这种方式导入，在后面使用该库的相关函数和变量时不需要添加 `jen.` 限定。一般是不建议这样做的。但是 `jennifer` 的函数比较多，如果不这样的话，每次都需要加上 `jen.` 比较繁琐。

- `jennifer` 的大部分方法都是可以**链式调用**的，每个方法处理完成之后都会返回当前对象，便于代码的编写。

下面我们从上面几个部分依次来介绍。

变量定义与运算

其实从语法层面来讲，变量就是标识符 + 类型。上面我们直接使用了字面量，这次我们想先定义一个变量存储欢迎信息：

```
func main() {
  var greeting = "Hello World"
  fmt.Println(greeting)
}
```

变量定义的方式有好几种， `jennifer` 可以比较直观的表达我们的意图。例如，上面的 `var greeting = "Hello World"` ，我们基本上可以逐字翻译：

- `var` 是变量定义， `jennifer` 中有对应的函数 `Var()` ；
- `greeting` 实际上是一个标识符，我们可以用 `Id()` 来定义；
- `=` 是赋值操作符，我们使用 `Op("=")` 来表示；
- **“Hello World”**是一个字符串字面量，最开始的例子中已经介绍过了，可以使用 `Lit()` 定义。

所以，这条语句翻译过来就是：

```
Var().Id("greeting").Op("=").Lit("Hello World")
```

同样的，我们可以试试另一种变量定义方式 `greeting := "Hello World"` ：

```
Id("greeting").Op(":=").Lit("Hello World")
```

是不是很简单。整个程序如下（省略包名和导入，下同）：

```
func main() {
  f := NewFile("main")
  f.Func().Id("main").Params().Block(
    // Var().Id("greeting").Op("=").Lit("Hello World"),
    Id("greeting").Op(":=").Lit("Hello World"),
    Qual("fmt", "Println").Call(Id("greeting")),
```

```
    )
    fmt.Printf("%#v\n", f)
}
```

接下来，我们用变量做一些运算。假设，我们要生成下面这段程序：

```
package main

import "fmt"

func main() {
    var a = 10
    var b = 2
    fmt.Printf("%d + %d = %d\n", a, b, a+b)
    fmt.Printf("%d + %d = %d\n", a, b, a-b)
    fmt.Printf("%d + %d = %d\n", a, b, a*b)
    fmt.Printf("%d + %d = %d\n", a, b, a/b)
}
```

变量定义这里不再赘述了，方法和函数调用实际上快速开始部分也介绍过。首先用 `Qual("fmt", "Printf")` 表示取包 `fmt` 中的 `Printf` 函数这一概念。然后使用 `Call()` 表示函数调用，参数第一个是字符串字面量，用 `Lit()` 表示。第二个和第三个都是一个标识符，用 `Id("a")` 和 `Id("b")` 即可表示。最后一个参数是两个标识符之间的运算，运算用 `Op()` 表示，所以最终就是生成程序：

```
func main() {
    f := NewFile("main")
    f.Func().Id("main").Params().Block(
        Var().Id("a").Op("=").Lit(10),
        Var().Id("b").Op("=").Lit(2),
        Qual("fmt", "Printf").Call(Lit("%d + %d = %d\n"), Id("a"), Id("b"), Id("a").
Op("+").Id("b")),
        Qual("fmt", "Printf").Call(Lit("%d + %d = %d\n"), Id("a"), Id("b"), Id("a").
Op("-").Id("b")),
        Qual("fmt", "Printf").Call(Lit("%d + %d = %d\n"), Id("a"), Id("b"), Id("a").
Op("*").Id("b")),
        Qual("fmt", "Printf").Call(Lit("%d + %d = %d\n"), Id("a"), Id("b"), Id("a").
Op("/").Id("b")),
    )
    fmt.Printf("%#v\n", f)
}
```

逻辑运算是类似的。

条件和循环

假设我们要生成下面的程序：

```
func main() {
  score := 70

  if score >= 90 {
    fmt.Println("优秀")
  } else if score >= 80 {
    fmt.Println("良好")
  } else if score >= 60 {
    fmt.Println("及格")
  } else {
    fmt.Println("不及格")
  }
}
```

依然采取我们的**逐字翻译**大法：

- `if` 关键字用 `If()` 来表示，条件语句是基本的标识符和常量操作。条件语句块与函数体一样，都使用 `Block()` ；
- `else` 关键字用 `Else()` 来表示， `else if` 就是 `Else()` 后面再调用 `If()` 即可。

完整的代码如下：

```
func main() {
  f := NewFile("main")

  f.Func().Id("main").Params().Block(
    Id("score").Op(":=").Lit(70),

    If(Id("score").Op(">=").Lit(90)).Block(
      Qual("fmt", "Println").Call(Lit("优秀")),
    ).Else().If(Id("score").Op(">=").Lit(80)).Block(
      Qual("fmt", "Println").Call(Lit("良好")),
    ).Else().If(Id("score").Op(">=").Lit(60)).Block(
      Qual("fmt", "Println").Call(Lit("及格")),
    ).Else().Block(
      Qual("fmt", "Println").Call(Lit("不及格")),
    ),
  )
```

```
  fmt.Printf("%#v\n", f)
}
```

对于 `for` 循环也是类似的，如果我们要生成下面的程序：

```
package main

import "fmt"

func main() {
  var sum int
  for i := 1; i <= 100; i++ {
    sum += i
  }

  fmt.Println(sum)
}
```

我们需要编写下面的程序：

```
func main() {
  f := NewFile("main")

  f.Func().Id("main").Params().Block(
    Var().Id("sum").Int(),

    For(
      Id("i").Op(":=").Lit(0),
      Id("i").Op("<=").Lit(100),
      Id("i").Op("++"),
    ).Block(
      Id("sum").Op("+=").Id("i"),
    ),

    Qual("fmt", "Println").Call(Id("sum")),
  )

  fmt.Printf("%#v\n", f)
}
```

`For()` 里面的 3 条语句对应实际 `for` 语句中的 3 个部分。

函数

函数是每个编程语言的必要元素。函数的核心要素是名字（标识符）、参数列表、返回值，最关键的就是函数体。我们之前编写 `main` 函数的时候大概介绍过。假设我们要编写一个计算两个数的和的函数：

```
func add(a, b int) int {
  return a + b
}
```

- 函数我们使用 `Func()` 表示，参数用 `Params()` 表示，返回值使用 `Int()` 表示；
- 函数体用 `Block()` ；
- `return` 语句使用 `Return()` 函数表示，其他都是一样的。

看下面的完整代码：

```
func main() {
  f := NewFile("main")

  f.Func().Id("add").Params(Id("a"), Id("b").Int()).Int().Block(
    Return(Id("a").Op("+").Id("b")),
  )

  f.Func().Id("main").Params().Block(
    Id("a").Op(":=").Lit(1),
    Id("b").Op(":=").Lit(2),
    Qual("fmt", "Println").Call(Id("add").Call(Id("a"), Id("b"))),
  )

  fmt.Printf("%#v\n", f)
}
```

一定要注意，即使没有参数， `Params()` 也一定要调用，否则生成的代码有语法错误。

结构和方法

下面我们看看结构和方法如何生成，假设我们想生成下面的程序：

```
package main

import "fmt"

type User struct {
```

```
    Name string
    Age  int
}

func (u *User) Greeting() {
    fmt.Printf("Hello %s", u.Name)
}
func main() {
    u := User{Name: "dj", Age: 18}
    u.Greeting()
}
```

需要用到的新函数：

- 结构体是一个类型，所以需要用到类型定义函数 `Type()` ，然后结构体的字段在 `Struct()` 内通过 `Id()` +类型定义；
- 方法其实也是一个函数，只不过多了一个接收器，我们还是使用 `Func()` 定义，接收者也可以用定义参数的 `Params()` 函数来指定，其它与函数没什么不同；
- 然后结构体初始化，在 `Values()` 中给字段赋值；
- 方法先用 `Dot("方法名")` 找到方法，然后 `Call()` 调用。

最后的程序：

```
func main() {
    f := NewFile("main")

    f.Type().Id("User").Struct(
        Id("Name").String(),
        Id("Age").Int(),
    )

    f.Func().Params(Id("u").Id("*User")).Id("Greeting").Params().Block(
        Qual("fmt", "Printf").Call(Lit("Hello %s"), Id("u").Dot("Name")),
    )

    f.Func().Id("main").Params().Block(
        Id("u").Op(":=").Id("User").Values(
            Id("Name").Op(":").Lit("dj"),
            Id("Age").Op(":").Lit(18),
        ),
        Id("u").Dot("Greeting").Call(),
    )
```

```
    fmt.Printf("%#v\n", f)
}
```

并发支持

还是一样，假设我想生成下面的程序：

```
package main

import "fmt"

func generate() chan int {
    out := make(chan int)
    go func() {
        for i := 1; i <= 100; i++ {
            out <- i
        }
        close(out)
    }()

    return out
}

func double(in <-chan int) chan int {
    out := make(chan int)

    go func() {
        for i := range in {
            out <- i * 2
        }
        close(out)
    }()

    return out
}

func main() {
    for i := range double(generate()) {
        fmt.Println(i)
    }
}
```

需要用到的新函数：

- 首先是 `make` 一个 `chan` ，用 `Make(Chan().Int())` ；
- 然后启动一个 goroutine，用 `Go()` ；
- 关闭 `chan` ，用 `Close()` ；
- `for ... range` 对应使用 `Range()` 。

拼在一起就是这样：

```
func main() {
	f := NewFile("main")
	f.Func().Id("generate").Params().Chan().Int().Block(
		Id("out").Op(":=").Make(Chan().Int()),
		Go().Func().Params().Block(
			For(
				Id("i").Op(":=").Lit(1),
				Id("i").Op("<=").Lit(100),
				Id("i").Op("++"),
			).Block(Id("out").Op("<-").Id("i")),
			Close(Id("out")),
		).Call(),
		Return().Id("out"),
	)

	f.Func().Id("double").Params(Id("in").Op("<-").Chan().Int()).Chan().Int().Bloc
k(
		Id("out").Op(":=").Make(Chan().Int()),
		Go().Func().Params().Block(
			For().Id("i").Op(":=").Range().Id("in").Block(Id("out").Op("<-").Id("i").O
p("*").Lit(2)),
			Close(Id("out")),
		).Call(),
		Return().Id("out"),
	)

	f.Func().Id("main").Params().Block(
		For(
			Id("i").Op(":=").Range().Id("double").Call(Id("generate").Call()),
		).Block(
			Qual("fmt", "Println").Call(Id("i")),
		),
	)

	fmt.Printf("%#v\n", f)
}
```

保存代码

上面的程序中，我们生成代码后直接输出了。在实际应用中，肯定是需要保存到文件中，然后编译运行的。 `jennifer` 也提供了保存到文件的方法 `File.Save()` ，直接传入文件名即可，这个 `File` 就是我们上面调用 `NewFile()` 生成的对象：

```
func main() {
  f := NewFile("main")
  f.Func().Id("main").Params().Block(
    Qual("fmt", "Println").Call(Lit("Hello, world")),
  )

  _, err := os.Stat("./generated")
  if os.IsNotExist(err) {
    os.Mkdir("./generated", 0666)
  }

  err = f.Save("./generated/main.go")
  if err != nil {
    log.Fatal(err)
  }
}
```

这种方式必须要保证 `generated` 目录存在。所以，我们使用 `os` 库在目录不存在时创建一个。

常见问题

`jennifer` 在生成代码后会调用 `go fmt` 对代码进行格式化，如果代码存在语法错误，这时候会输出错误信息。我遇到最多的问题就是最后生成的程序代码以 `})` 结尾，这明显不符合语法。查了半天发现 `Func()` 后忘记加 `Params()` 。即使是空参数，这个 `Params()` 也不能省略！

总结

`jennifer` 支持的远不止我上面介绍的那些，实际上 Go 的语法和特性它都支持，如 `select/goto/panic/recover/continue/break` ，还有类型断言 `b, ok := i.(bool)` 、注释、cgo 等等等等。感兴趣可以自己去探索。虽然 `jennifer` 函数众多，但是按照我们的**逐字翻译**大法，实际上用起来很简单。说实话，我在写上次的测试程序时，基本上没看文档，先按照自己的理解去写，结果大部分都是对！只有一些有问题的地方再去看文档。

说了这么多， `jennifer` 的用途是什么呢？答曰，根据配置生成代码，编写生成代码的工具。另外 `jennifer` 的 GitHub 中有一个 `genjen` 目录，实际上我们用到的很多函数都是通过它来生成的，是不是很棒😄。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. jennifer GitHub：https://github.com/dave/jennifer
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

copier

简介

上一篇文章介绍了 `mergo` 库的使用， `mergo` 是用来给结构体或 `map` 赋值的。 `mergo` 有一个明显的不足——它只能处理相同类型的结构！如果类型不同，即使字段名和类型完全相同， `mergo` 也无能为力。今天我们要介绍的 `copier` 库就能处理不同类型之间的赋值。除此之外， `copier` 还能：

- 调用同名方法为字段赋值；
- 以源对象字段为参数调用目标对象的方法，从而为目标对象赋值（当然也可以做其它的任何事情）；
- 将切片赋值给切片（可以是不同类型哦）；
- 将结构体追加到切片中。

感谢@thinkgos推荐。

顺带一提，作者是国人jinzhu大佬，如果你想找一个 Go 语言的 ORM 库，gorm你值得拥有！

快速使用

先安装：

```
$ go get github.com/jinzhu/copier
```

后使用：

```
package main

import (
  "fmt"

  "github.com/jinzhu/copier"
)

type User struct {
  Name string
  Age  int
}
```

```
type Employee struct {
  Name string
  Age  int
  Role string
}

func main() {
  user := User{Name: "dj", Age: 18}
  employee := Employee{}

  copier.Copy(&employee, &user)
  fmt.Printf("%#v\n", employee)
}
```

很好理解，就是将 `user` 对象中的字段赋值到 `employee` 的同名字段中。如果目标对象中没有同名的字段，则该字段被忽略。

高级特性

方法赋值

目标对象中的一些字段，源对象中没有，但是源对象有同名的方法。这时 `Copy` 会调用这个方法，将返回值赋值给目标对象中的字段：

```
type User struct {
  Name string
  Age  int
}

func (u *User) DoubleAge() int {
  return 2 * u.Age
}

type Employee struct {
  Name      string
  DoubleAge int
  Role      string
}

func main() {
  user := User{Name: "dj", Age: 18}
  employee := Employee{}
```

```
  copier.Copy(&employee, &user)
  fmt.Printf("%#v\n", employee)
}
```

我们给 `User` 添加一个 `DoubleAge` 方法。`Employee` 结构有字段 `DoubleAge`，`User` 中没有，但是 `User` 有一个同名的方法，这时 `Copy` 调用 `user` 的 `DoubleAge` 方法为 `employee` 的 `DoubleAge` 赋值，得到 36。

调用目标方法

有时候源对象中的某个字段没有出现在目标对象中，但是目标对象有一个同名的方法，方法接受一个同类型的参数，这时 `Copy` 会以源对象的这个字段作为参数调用目标对象的该方法：

```
type User struct {
  Name string
  Age  int
  Role string
}

type Employee struct {
  Name      string
  Age       int
  SuperRole string
}

func (e *Employee) Role(role string) {
  e.SuperRole = "Super" + role
}

func main() {
  user := User{Name: "dj", Age: 18, Role: "Admin"}
  employee := Employee{}

  copier.Copy(&employee, &user)
  fmt.Printf("%#v\n", employee)
}
```

我们给 `Employee` 添加了一个 `Role` 方法，`User` 的字段 `Role` 没有出现在 `Employee` 中，但是 `Employee` 有一个同名方法。`Copy` 函数内部会以 `user` 对象的 `Role` 字段为参数调用 `employee` 的 `Role` 方法。最终，我们的 `employee` 对象的 `SuperRole` 值变为 `SuperAdmin` 。实际上，这个方法中可以执行任何操作，不一定是赋值。

切片赋值

使用一个切片来为另一个切片赋值。如果类型相同，那好办，直接 `append` 就行。如果类型不同呢？ `copier` 就派上大用场了：

```
type User struct {
  Name string
  Age  int
}

type Employee struct {
  Name string
  Age  int
  Role string
}

func main() {
  users := []User{
    {Name: "dj", Age: 18},
    {Name: "dj2", Age: 18},
  }
  employees := []Employee{}

  copier.Copy(&employees, &users)
  fmt.Printf("%#v\n", employees)
}
```

这个实际上就是将源切片中每个元素分别赋值到目标切片中。

将结构赋值到切片

这个不难，实际上就是根据源对象生成一个和目标切片类型相符合的对象，然后 `append` 到目标切片中：

```
type User struct {
  Name string
  Age  int
}

type Employee struct {
  Name string
  Age  int
  Role string
}
```

```go
func main() {
  user := User{Name: "dj", Age: 18}
  employees := []Employee{}

  copier.Copy(&employees, &user)
  fmt.Printf("%#v\n", employees)
}
```

上面代码中，`Copy`先通过`user`生成一个`Employee`对象，然后`append`到切片`employees`中。

汇总

最后将所有的特性汇总在一个例子中，其实就是`Copier`的 GitHub 仓库首页的例子：

```go
type User struct {
  Name string
  Age  int
  Role string
}

func (u *User) DoubleAge() int {
  return u.Age * 2
}

type Employee struct {
  Name      string
  Age       int
  SuperRole string
}

func (e *Employee) Role(role string) {
  e.SuperRole = "Super" + role
}

func main() {
  var (
    user  = User{Name: "dj", Age: 18}
    users = []User{
      {Name: "dj", Age: 18, Role: "Admin"},
      {Name: "dj2", Age: 18, Role: "Dev"},
    }
    employee  = Employee{}
    employees = []Employee{}
```

```
    )

    copier.Copy(&employee, &user)
    fmt.Printf("%#v\n", employee)

    copier.Copy(&employees, &user)
    fmt.Printf("%#v\n", employees)

    // employees = []Employee{}

    copier.Copy(&employees, &users)
    fmt.Printf("%#v\n", employees)
}
```

上面例子中，我故意把 `employees = []Employee{}` 这一行注释掉，最后输出的 `employees` 是 3 个元素，能更清楚的看出切片到切片是 `append` 的，目标切片原来的元素还是保留的。

总结

`copier` 库的代码量很小，用了不到 200 行的代码就实现了如此实用的一个功能，非常值得一看！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. copier GitHub：https://github.com/jinzhu/copier
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

mergo

简介

今天我们介绍一个合并结构体字段的库 `mergo` 。 `mergo` 可以在相同的结构体或 `map` 之间赋值，可以将结构体的字段赋值到 `map` 中，可以将 `map` 的值赋值给结构体的字段。感谢@thinkgos推荐。

快速使用

先安装：

```
$ go get github.com/imdario/mergo
```

后使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/imdario/mergo"
)

type redisConfig struct {
  Address string
  Port    int
  DB      int
}

var defaultConfig = redisConfig{
  Address: "127.0.0.1",
  Port:    6381,
  DB:      1,
}

func main() {
  var config redisConfig

  if err := mergo.Merge(&config, defaultConfig); err != nil {
```

```go
    log.Fatal(err)
  }

  fmt.Println("redis address: ", config.Address)
  fmt.Println("redis port: ", config.Port)
  fmt.Println("redis db: ", config.DB)

  var m = make(map[string]interface{})
  if err := mergo.Map(&m, defaultConfig); err != nil {
    log.Fatal(err)
  }

  fmt.Println(m)
}
```

使用非常简单。`mergo` 提供了两组接口（其实就是两个，`*WithOverwrite` 已经废弃了，可使用 `WithOverride` 选项代替）：

- `Merge`：合并两个相同类型的结构或`map`；
- `Map`：在结构和`map`之间赋值。

参数 1 是目标对象，参数 2 是源对象，这两个函数的功能就是将源对象中的字段复制到目标对象的对应字段上。

高级选项

如果仅仅只是复制结构体，为啥不直接写 `redisConfig = defaultConfig` 呢？`mergo` 提供了很多选项。

覆盖

默认情况下，如果目标对象的字段已经设置了，那么 `Merge/Map` 不会用源对象中的字段替换它。我们在上面程序的 `var config redisConfig` 定义下添加一行：

```go
config.DB = 2
```

再看看运行结果，发现输出的 `db` 是 2，而非 1。

可以通过选项来改变这个行为，调用 `Merge/Map` 时，传入 `WithOverride` 参数，那么目标对象中已经设置的字段也会被覆盖：

```go
if err := mergo.Merge(&config, defaultConfig, mergo.WithOverride); err != nil {
  log.Fatal(err)
```

```
}
```

只需要修改这一行调用。结果输出 `db` 是 1，覆盖了！

这里用到了 Go 中的**选项模式**。在参数比较多，且大部分有默认值的情况下，我们可以在函数最后添加一个可变的选项参数，通过传入选项来改变函数的行为，不传入的选项就使用默认值。选项模式在 Go 语言中使用非常广泛，能大大提高代码的可扩展性，使用可变参数也能使函数更易用。 `mergo` 中的选项都是这种形式。想要深入了解一下？看这里 https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis。

`mergo` 老的接口 `MergeWithOverride` 和 `MapWithOverride` 都使用选项模式重构了。

切片

如果某个字段是一个切片，不覆盖就保留目标对象的值，或者用源对象的值覆盖都不合适。我们可能想将源对象中切片的值对添加到目标对象的字段中，这时可以使用 `WithAppendSlice` 选项。

```
package main

import (
  "fmt"
  "log"

  "github.com/imdario/mergo"
)

type redisConfig struct {
  Address string
  Port    int
  DBs     []int
}

var defaultConfig = redisConfig{
  Address: "127.0.0.1",
  Port:    6381,
  DBs:     []int{1},
}

func main() {
  var config redisConfig
  config.DBs = []int{2, 3}

  if err := mergo.Merge(&config, defaultConfig, mergo.WithAppendSlice); err != n
```

```
il {
    log.Fatal(err)
  }

  fmt.Println("redis address: ", config.Address)
  fmt.Println("redis port: ", config.Port)
  fmt.Println("redis dbs: ", config.DBs)
}
```

我们将 `DB` 字段改为 `[]int` 类型的 `DBs` ，使用 `WithAppendSliec` 选项，最后输出的 `DBs` 为 `[2 3 1]` 。

空值覆盖

默认情况下，如果源对象中的字段为空值（数组、切片长度为 0 ，指针为 `nil` ，数字为 0，字符串为""等），即使我们使用了 `WithOverride` 选项也是不会覆盖的。下面两个选项就是强制这种情况下也覆盖：

- `WithOverrideEmptySlice` ：源对象的空切片覆盖目标对象的对应字段；
- `WithOverwriteWithEmptyValue` ：源对象中的空值覆盖目标对象的对应字段，其实这个对切片也有效。

文档中这两个选项的介绍比较混乱，我通过看源码和自己试验下来发现：

- 这两个选项都必须和`WithOverride`一起使用；
- `WithOverwriteWithEmptyValue`这个选项也可以处理切片类型的值。

看下面代码：

```
type redisConfig struct {
  Address string
  Port    int
  DBs     []int
}

var defaultConfig = redisConfig{
  Address: "127.0.0.1",
  Port:    6381,
}

func main() {
  var config redisConfig
  config.DBs = []int{2, 3}
```

```
  if err := mergo.Merge(&config, defaultConfig, mergo.WithOverride, mergo.WithOverrideEmptySlice); err != nil {
    log.Fatal(err)
  }

  fmt.Println("redis address: ", config.Address)
  fmt.Println("redis port: ", config.Port)
  fmt.Println("redis dbs: ", config.DBs)
}
```

最终会输出空的 `DBs` 。

类型检查

这个主要用在 `map` 之间的切片字段的赋值，因为使用 `mergo` 在两个结构体之间赋值必须保证两个结构体类型相同，没有类型检查的必要。因为 `map` 类型为 `map[string]interface{}` ，所以默认情况下， `map` 切片类型不一致也是可以赋值的：

```
func main() {
  m1 := make(map[string]interface{})
  m1["dbs"] = []uint32{2, 3}

  m2 := make(map[string]interface{})
  m2["dbs"] = []int{1}

  if err := mergo.Map(&m1, &m2, mergo.WithOverride); err != nil {
    log.Fatal(err)
  }

  fmt.Println(m1)
}
```

如果添加 `mergo.WithTypeCheck` 选项，则切片类型不一致会抛出错误：

```
if err := mergo.Map(&m1, &m2, mergo.WithOverride, mergo.WithTypeCheck); err != nil {
  log.Fatal(err)
}
```

输出：

```
cannot override two slices with different type ([]int, []uint32)
exit status 1
```

注意事项

1. `mergo`不会赋值非导出字段；
2. `map`中对应的键名首字母会转为小写；
3. `mergo`可嵌套赋值，我们演示的只有一层结构。

总结

`mergo`其实在很多知名项目中都有应用，如`moby/kubernetes`等。本文介绍了`mergo`的基本用法，感兴趣可以去 GitHub 上深入学习。关于选项模式，这里多说一句，我在实际项目中多次应用，能极大地提高可扩展性，方便今后添加新的功能。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. mergo GitHub：https://github.com/imdario/mergo
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

wire

简介

之前的一篇文章Go 每日一库之 dig介绍了 uber 开源的依赖注入框架 `dig` 。读了这篇文章后，@overtalk推荐了 Google 开源的 `wire` 工具。所以就有了今天这篇文章，感谢推荐

`wire` 是 Google 开源的一个依赖注入工具。它是一个代码生成器，并不是一个框架。我们只需要在一个特殊的 `go` 文件中告诉 `wire` 类型之间的依赖关系，它会自动帮我们生成代码，帮助我们创建指定类型的对象，并组装它的依赖。

快速使用

先安装工具：

```
$ go get github.com/google/wire/cmd/wire
```

上面的命令会在 `$GOPATH/bin` 中生成一个可执行程序 `wire` ，这就是代码生成器。我个人习惯把 `$GOPATH/bin` 加入系统环境变量 `$PATH` 中，所以可直接在命令行中执行 `wire` 命令。

下面我们在一个例子中看看如何使用 `wire` 。

现在，我们来到一个黑暗的世界，这个世界中有一个邪恶的怪兽。我们用下面的结构表示，同时编写一个创建方法：

```
type Monster struct {
  Name string
}

func NewMonster() Monster {
  return Monster{Name: "kitty"}
}
```

有怪兽肯定就有勇士，结构如下，同样地它也有创建方法：

```
type Player struct {
  Name string
}
```

```go
func NewPlayer(name string) Player {
  return Player{Name: name}
}
```

终于有一天，勇士完成了他的使命，战胜了怪兽：

```go
type Mission struct {
  Player  Player
  Monster Monster
}

func NewMission(p Player, m Monster) Mission {
  return Mission{p, m}
}

func (m Mission) Start() {
  fmt.Printf("%s defeats %s, world peace!\n", m.Player.Name, m.Monster.Name)
}
```

这可能是某个游戏里面的场景哈，我们看如何将上面的结构组装起来放在一个应用程序中：

```go
func main() {
  monster := NewMonster()
  player := NewPlayer("dj")
  mission := NewMission(player, monster)

  mission.Start()
}
```

代码量少，结构不复杂的情况下，上面的实现方式确实没什么问题。但是项目庞大到一定程度，结构之间的关系变得非常复杂的时候，这种手动创建每个依赖，然后将它们组装起来的方式就会变得异常繁琐，并且容易出错。这个时候勇士 `wire` 出现了！

`wire` 的要求很简单，新建一个 `wire.go` 文件（文件名可以随意），创建我们的初始化函数。比如，我们要创建并初始化一个 `Mission` 对象，我们就可以这样：

```go
//+build wireinject

package main

import "github.com/google/wire"

func InitMission(name string) Mission {
  wire.Build(NewMonster, NewPlayer, NewMission)
```

```
    return Mission{}
}
```

首先这个函数的返回值就是我们需要创建的对象类型， `wire` 只需要知道类型， `return` 后返回什么不重要。然后在函数中，我们调用 `wire.Build()` 将创建 `Mission` 所依赖的类型的构造器传进去。例如，需要调用 `NewMission()` 创建 `Mission` 类型， `NewMission()` 接受两个参数一个 `Monster` 类型，一个 `Player` 类型。 `Monster` 类型对象需要调用 `NewMonster()` 创建， `Player` 类型对象需要调用 `NewPlayer()` 创建。所以 `NewMonster()` 和 `NewPlayer()` 我们也需要传给 `wire` 。

文件编写完成之后，执行 `wire` 命令：

```
$ wire
wire: github.com/darjun/go-daily-lib/wire/get-started/after: \
wrote D:\code\golang\src\github.com\darjun\go-daily-lib\wire\get-started\after\w
ire_gen.go
```

我们看看生成的 `wire_gen.go` 文件：

```
// Code generated by Wire. DO NOT EDIT.

//go:generate wire
//+build !wireinject

package main

// Injectors from wire.go:

func InitMission(name string) Mission {
    player := NewPlayer(name)
    monster := NewMonster()
    mission := NewMission(player, monster)
    return mission
}
```

这个 `InitMission()` 函数是不是和我们在 `main.go` 中编写的代码一毛一样！接下来，我们可以直接在 `main.go` 调用 `InitMission()` ：

```
func main() {
    mission := InitMission("dj")

    mission.Start()
}
```

细心的童鞋可能发现了， `wire.go` 和 `wire_gen.go` 文件头部位置都有一个 `+build` ，不过一个后面是 `wireinject` ，另一个是 `!wireinject` 。 `+build` 其实是 Go 语言的一个特性。类似 C/C++ 的条件编译，在执行 `go build` 时可传入一些选项，根据这个选项决定某些文件是否编译。 `wire` 工具只会处理有 `wireinject` 的文件，所以我们的 `wire.go` 文件要加上这个。生成的 `wire_gen.go` 是给我们来使用的， `wire` 不需要处理，故有 `!wireinject` 。

由于现在是两个文件，我们不能用 `go run main.go` 运行程序，可以用 `go run .` 运行。运行结果与之前的例子一模一样！

注意，如果你运行时，出现了 `InitMission` 重定义，那么检查一下你的 `//+build wireinject` 与 `package main` 这两行之间是否有空行，这个空行必须要有！见 https://github.com/google/wire/issues/117。中招的默默在心里打个 1 好嘛😂

基础概念

`wire` 有两个基础概念， `Provider` （构造器）和 `Injector` （注入器）。 `Provider` 实际上就是创建函数，大家意会一下。我们上面 `InitMission` 就是 `Injector` 。每个注入器实际上就是一个对象的创建和初始化函数。在这个函数中，我们只需要告诉 `wire` 要创建什么类型的对象，这个类型的依赖， `wire` 工具会为我们生成一个函数完成对象的创建和初始化工作。

参数

同样细心的你应该发现了，我们上面编写的 `InitMission()` 函数带有一个 `string` 类型的参数。并且在生成的 `InitMission()` 函数中，这个参数传给了 `NewPlayer()` 。 `NewPlayer()` 需要 `string` 类型的参数，而参数类型就是 `string` 。所以生成的 `InitMission()` 函数中，这个参数就被传给了 `NewPlayer()` 。如果我们让 `NewMonster()` 也接受一个 `string` 参数呢？

```
func NewMonster(name string) Monster {
  return Monster{Name: name}
}
```

那么生成的 `InitMission()` 函数中 `NewPlayer()` 和 `NewMonster()` 都会得到这个参数：

```
func InitMission(name string) Mission {
  player := NewPlayer(name)
  monster := NewMonster(name)
  mission := NewMission(player, monster)
```

```
  return mission
}
```

实际上， `wire` 在生成代码时，构造器需要的参数（或者叫依赖）会从参数中查找或通过其它构造器生成。决定选择哪个参数或构造器完全根据类型。如果参数或构造器生成的对象有类型相同的情况，运行 `wire` 工具时会报错。如果我们想要定制创建行为，就需要为不同类型创建不同的参数结构：

```
type PlayerParam string
type MonsterParam string

func NewPlayer(name PlayerParam) Player {
  return Player{Name: string(name)}
}

func NewMonster(name MonsterParam) Monster {
  return Monster{Name: string(name)}
}

func main() {
  mission := InitMission("dj", "kitty")
  mission.Start()
}

// wire.go
func InitMission(p PlayerParam, m MonsterParam) Mission {
  wire.Build(NewPlayer, NewMonster, NewMission)
  return Mission{}
}
```

生成的代码如下：

```
func InitMission(m MonsterParam, p PlayerParam) Mission {
  player := NewPlayer(p)
  monster := NewMonster(m)
  mission := NewMission(player, monster)
  return mission
}
```

在参数比较复杂的时候，建议将参数放在一个结构中。

错误

不是所有的构造操作都能成功，没准勇士出山前就死于小人之手：

```
func NewPlayer(name string) (Player, error) {
  if time.Now().Unix()%2 == 0 {
    return Player{}, errors.New("player dead")
  }
  return Player{Name: name}, nil
}
```

我们使创建**随机失败**，修改注入器 `InitMission()` 的签名，增加 `error` 返回值：

```
func InitMission(name string) (Mission, error) {
  wire.Build(NewMonster, NewPlayer, NewMission)
  return Mission{}, nil
}
```

生成的代码，会将 `NewPlayer()` 返回的错误，作为 `InitMission()` 的返回值：

```
func InitMission(name string) (Mission, error) {
  player, err := NewPlayer(name)
  if err != nil {
    return Mission{}, err
  }
  monster := NewMonster()
  mission := NewMission(player, monster)
  return mission, nil
}
```

`wire` 遵循**fail-fast**的原则，错误必须被处理。如果我们的注入器不返回错误，但构造器返回错误， `wire` 工具会报错！

高级特性

下面简单介绍一下 `wire` 的高级特性。

ProviderSet

有时候可能多个类型有相同的依赖，我们每次都将相同的构造器传给 `wire.Build()` 不仅繁琐，而且不易维护，一个依赖修改了，所有传入 `wire.Build()` 的地方都要修改。为此， `wire` 提供了一个 `ProviderSet` （构造器集合），可以将多个构造器打包成一个集合，后续只需要使用这个集合即可。假设，我们有关勇士和怪兽的故事有两个结局：

```go
type EndingA struct {
  Player  Player
  Monster Monster
}

func NewEndingA(p Player, m Monster) EndingA {
  return EndingA{p, m}
}

func (p EndingA) Appear() {
  fmt.Printf("%s defeats %s, world peace!\n", p.Player.Name, p.Monster.Name)
}

type EndingB struct {
  Player  Player
  Monster Monster
}

func NewEndingB(p Player, m Monster) EndingB {
  return EndingB{p, m}
}

func (p EndingB) Appear() {
  fmt.Printf("%s defeats %s, but become monster, world darker!\n", p.Player.Name, p.Monster.Name)
}
```

编写两个注入器：

```go
func InitEndingA(name string) EndingA {
  wire.Build(NewMonster, NewPlayer, NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(NewMonster, NewPlayer, NewEndingB)
  return EndingB{}
}
```

我们观察到两次调用 `wire.Build()` 都需要传入 `NewMonster` 和 `NewPlayer` 。两个还好，如果很多的话写起来就麻烦了，而且修改也不容易。这种情况下，我们可以先定义一个 `ProviderSet` ：

```go
var monsterPlayerSet = wire.NewSet(NewMonster, NewPlayer)
```

后续直接使用这个 `set` ：

```
func InitEndingA(name string) EndingA {
  wire.Build(monsterPlayerSet, NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(monsterPlayerSet, NewEndingB)
  return EndingB{}
}
```

而后如果要添加或删除某个构造器，直接修改 `set` 的定义处即可。

结构构造器

因为我们的 `EndingA` 和 `EndingB` 的字段只有 `Player` 和 `Monster` ，我们就不需要显式为它们提供构造器，可以直接使用 `wire` 提供的**结构构造器**（Struct Provider）。结构构造器创建某个类型的结构，然后用参数或调用其它构造器填充它的字段。例如上面的例子，我们去掉 `NewEndingA()` 和 `NewEndingB()` ，然后为它们提供结构构造器：

```
var monsterPlayerSet = wire.NewSet(NewMonster, NewPlayer)

var endingASet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingA), "Player", "Monster"))
var endingBSet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingB), "Player", "Monster"))

func InitEndingA(name string) EndingA {
  wire.Build(endingASet)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(endingBSet)
  return EndingB{}
}
```

结构构造器使用 `wire.Struct` 注入，第一个参数固定为 `new(结构名)` ，后面可接任意多个参数，表示需要为该结构的哪些字段注入值。上面我们需要注入 `Player` 和 `Monster` 两个字段。或者我们也可以使用通配符 `*` 表示注入所有字段：

```go
var endingASet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingA), "*"))
var endingBSet = wire.NewSet(monsterPlayerSet, wire.Struct(new(EndingB), "*"))
```

`wire` 为我们生成正确的代码，非常棒：

```go
func InitEndingA(name string) EndingA {
  player := NewPlayer(name)
  monster := NewMonster()
  endingA := EndingA{
    Player:  player,
    Monster: monster,
  }
  return endingA
}
```

绑定值

有时候，我们需要为某个类型绑定一个值，而不想依赖构造器每次都创建一个新的值。有些类型天生就是单例，例如配置，数据库对象（ `sql.DB` ）。这时我们可以使用 `wire.Value` 绑定值，使用 `wire.InterfaceValue` 绑定接口。例如，我们的怪兽一直是一个 `Kitty` ，我们就不用每次都去创建它了，直接绑定这个值就 ok 了：

```go
var kitty = Monster{Name: "kitty"}

func InitEndingA(name string) EndingA {
  wire.Build(NewPlayer, wire.Value(kitty), NewEndingA)
  return EndingA{}
}

func InitEndingB(name string) EndingB {
  wire.Build(NewPlayer, wire.Value(kitty), NewEndingB)
  return EndingB{}
}
```

注意一点，这个值每次使用时都会拷贝，需要确保拷贝无副作用：

```go
// wire_gen.go
func InitEndingA(name string) EndingA {
  player := NewPlayer(name)
  monster := _wireMonsterValue
  endingA := NewEndingA(player, monster)
  return endingA
}
```

```
var (
  _wireMonsterValue = kitty
)
```

结构字段作为构造器

有时候我们编写一个构造器，只是简单的返回某个结构的一个字段，这时可以使用 `wire.FieldsOf` 简化操作。现在我们直接创建了 `Mission` 结构，如果想获得 `Monster` 和 `Player` 类型的对象，就可以对 `Mission` 使用 `wire.FieldsOf` ：

```
func NewMission() Mission {
  p := Player{Name: "dj"}
  m := Monster{Name: "kitty"}

  return Mission{p, m}
}

// wire.go
func InitPlayer() Player {
  wire.Build(NewMission, wire.FieldsOf(new(Mission), "Player"))
}

func InitMonster() Monster {
  wire.Build(NewMission, wire.FieldsOf(new(Mission), "Monster"))
}

// main.go
func main() {
  p := InitPlayer()
  fmt.Println(p.Name)
}
```

同样的，第一个参数为 `new(结构名)` ，后面跟多个参数表示将哪些字段作为构造器， `*` 表示全部。

清理函数

构造器可以提供一个清理函数，如果后续的构造器返回失败，前面构造器返回的清理函数都会调用：

```
func NewPlayer(name string) (Player, func(), error) {
  cleanup := func() {
```

```go
        fmt.Println("cleanup!")
    }
    if time.Now().Unix()%2 == 0 {
        return Player{}, cleanup, errors.New("player dead")
    }
    return Player{Name: name}, cleanup, nil
}

func main() {
    mission, cleanup, err := InitMission("dj")
    if err != nil {
        log.Fatal(err)
    }

    mission.Start()
    cleanup()
}

// wire.go
func InitMission(name string) (Mission, func(), error) {
    wire.Build(NewMonster, NewPlayer, NewMission)
    return Mission{}, nil, nil
}
```

一些细节

首先，我们调用 `wire` 生成 `wire_gen.go` 之后，如果 `wire.go` 文件有修改，只需要执行 `go generate` 即可。 `go generate` 很方便，我之前一篇文章写过 `generate` ，感兴趣可以看看深入理解Go之generate。

总结

`wire` 是随着 `go-cloud` 的示例 `guestbook` 一起发布的，可以阅读 `guestbook` 看看它是怎么使用 `wire` 的。与 `dig` 不同， `wire` 只是生成代码，不使用 `reflect` 库，性能方面是不用担心的。因为它生成的代码与你自己写的基本是一样的。如果生成的代码有性能问题，自己写大概率也会有😂。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. wire GitHub：https://github.com/google/wire
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

watermill

简介

在上一篇文章Go 每日一库之 message-bus中，我们介绍了一款小巧、实现简单的异步通信库。作为学习， `message-bus` 确实不错。但是在实际使用上， `message-bus` 的功能就有点捉襟见肘了。例如， `message-bus` 将消息发送到订阅者管道之后就不管了，这样如果订阅者处理压力较大，会在管道中堆积太多消息，一旦订阅者异常退出，这些消息将会全部丢失！另外， `message-bus` 不负责保存消息，如果订阅者后启动，之前发布的消息，这个订阅者是无法收到的。这些问题，我们将要介绍的watermill都能解决！

watermill是 Go 语言的一个异步消息解决方案，它支持消息重传、保存消息，后启动的订阅者也能收到前面发布的消息。 `watermill` 内置了多种**订阅-发布**实现，包括 `Kafka/RabbitMQ` ，甚至还支持 `HTTP/MySQL binlog` 。当然也可以编写自己的订阅-发布实现。此外，它还提供了监控、限流等中间件。

快速使用

`watermill` 内置了很多**订阅-发布**实现，最简单、直接的要属 `GoChannel` 。我们就以这个实现为例介绍 `watermill` 的特性。

安装：

```
$ go get github.com/ThreeDotsLabs/watermill
```

使用：

```
package main

import (
  "context"
  "log"
  "time"

  "github.com/ThreeDotsLabs/watermill"
  "github.com/ThreeDotsLabs/watermill/message"
  "github.com/ThreeDotsLabs/watermill/pubsub/gochannel"
)

func main() {
  pubSub := gochannel.NewGoChannel(
```

```
        gochannel.Config{},
        watermill.NewStdLogger(false, false),
    )

    messages, err := pubSub.Subscribe(context.Background(), "example.topic")
    if err != nil {
        panic(err)
    }

    go process(messages)

    publishMessages(pubSub)
}

func publishMessages(publisher message.Publisher) {
    for {
        msg := message.NewMessage(watermill.NewUUID(), []byte("Hello, world!"))

        if err := publisher.Publish("example.topic", msg); err != nil {
            panic(err)
        }

        time.Sleep(time.Second)
    }
}

func process(messages <-chan *message.Message) {
    for msg := range messages {
        log.Printf("received message: %s, payload: %s", msg.UUID, string(msg.Payloa
d))
        msg.Ack()
    }
}
```

首先，我们创建一个 `GoChannel` 对象，它是一个消息管理器。可以调用其 `Subscribe` 订阅某个主题（**topic**）的消息，调用其 `Publish()` 以某个主题发布消息。 `Subscribe()` 方法会返回一个 `<-chan *message.Message` ，一旦该主题有消息发布， `GoChannel` 就会将消息发送到该管道中。订阅者只需监听此管道，接收消息进行处理。在上面的例子中，我们启动了一个消息处理的 `goroutine` ，持续从管道中读取消息，然后打印输出。主 `goroutine` 在一个死循环中每隔 1s 发布一次消息。

`message.Message` 这个结构是 `watermill` 库的核心，每个消息都会封装到该结构中发送。 `Message` 保存的是原始的字节流（ `[]byte` ），所以可以将 JSON/protobuf/XML 等等格式的序列化结果保存到 `Message` 中。

有两点注意：

- 收到的每个消息都需要调用 `Message` 的 `Ack()` 方法确认，否则 `GoChannel` 会重发当前消息；
- `Message` 有一个 `UUID` 字段，建议设置为唯一的，方便定位问题。`watermill` 提供方法 `NewUUID()` 生成唯一 id。

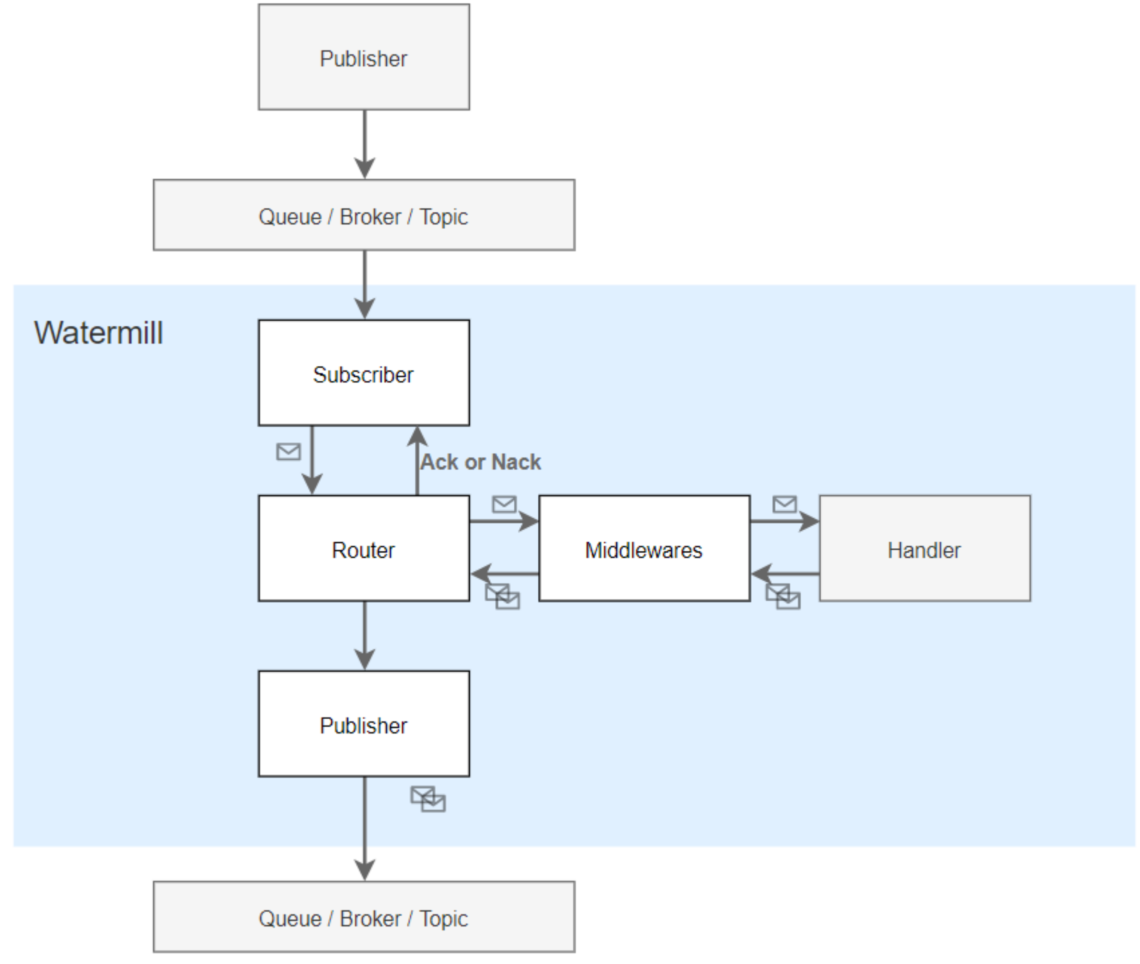
下面看示例运行：

```
dajun.ldj MINGW64 /d/code/golang/src/github.com/darjun/go-daily-lib/watermill/get-started (master)
$ go run main.go
```

路由

上面的发布和订阅实现是非常底层的模式。在实际应用中，我们通常想要监控、重试、统计等一些功能。而且上面的例子中，每个消息处理结束需要手动调用 `Ack()` 方法，消息管理器才会下发后面一条信息，很容易遗忘。还有些时候，我们有这样的需求，处理完某个消息后，重新发布另外一些消息。

这些功能都是比较通用的，为此 `watermill` 提供了路由（**Router**）功能。直接拿来官网的图：



路由其实管理多个订阅者，每个订阅者在一个独立的 `goroutine` 中运行，彼此互不干扰。订阅者收到消息后，交由注册时指定的处理函数（**HandlerFunc**）。路由还可以设置插件（**plugin**）和中间件（**middleware**），插件是定制路由的行为，而中间件是定制处理器的行为。处理器处理消息后会返回若干消息，这些消息会被路由重新发布到（另一个）管理器中。

```
var (
	logger = watermill.NewStdLogger(false, false)
)

func main() {
	router, err := message.NewRouter(message.RouterConfig{}, logger)
	if err != nil {
		panic(err)
	}

	pubSub := gochannel.NewGoChannel(gochannel.Config{}, logger)
	go publishMessages(pubSub)

	router.AddHandler("myhandler", "in_topic", pubSub, "out_topic", pubSub, myHand
```

```
ler{}.Handler)

	router.AddNoPublisherHandler("print_in_messages", "in_topic", pubSub, printMessages)
	router.AddNoPublisherHandler("print_out_messages", "out_topic", pubSub, printMessages)

	ctx := context.Background()
	if err := router.Run(ctx); err != nil {
		panic(err)
	}
}

func publishMessages(publisher message.Publisher) {
	for {
		msg := message.NewMessage(watermill.NewUUID(), []byte("Hello, world!"))
		if err := publisher.Publish("in_topic", msg); err != nil {
			panic(err)
		}

		time.Sleep(time.Second)
	}
}

func printMessages(msg *message.Message) error {
	fmt.Printf("\n> Received message: %s\n> %s\n>\n", msg.UUID, string(msg.Payload))
	return nil
}

type myHandler struct {
}

func (m myHandler) Handler(msg *message.Message) ([]*message.Message, error) {
	log.Println("myHandler received message", msg.UUID)

	msg = message.NewMessage(watermill.NewUUID(), []byte("message produced by myHandler"))
	return message.Messages{msg}, nil
}
```

首先，我们创建一个路由：

```
router, err := message.NewRouter(message.RouterConfig{}, logger)
```

然后为路由注册处理器。注册的处理器有两种类型，一种是：

```
router.AddHandler("myhandler", "in_topic", pubSub, "out_topic", pubSub, myHandler{}.Handler)
```

这个方法原型为：

```
func (r *Router) AddHandler(
  handlerName string,
  subscribeTopic string,
  subscriber Subscriber,
  publishTopic string,
  publisher Publisher,
  handlerFunc HandlerFunc,
) *Handler
```

该方法的作用是创建一个名为 `handlerName` 的处理器，监听 `subscriber` 中主题为 `subscribeTopic` 的消息，收到消息后调用 `handlerFunc` 处理，将返回的消息以主题 `publishTopic` 发布到 `publisher` 中。

另外一种处理器是下面这种形式：

```
router.AddNoPublisherHandler("print_in_messages", "in_topic", pubSub, printMessages)
router.AddNoPublisherHandler("print_out_messages", "out_topic", pubSub, printMessages)
```

从名字我们也可以看出，这种形式的处理器只处理接收到的消息，不发布新消息。

最后，我们调用 `router.Run()` 运行这个路由。

其中，创建 `GoChannel` 发布消息和上面的没什么不同。

使用路由还有个好处，处理器返回时，若无错误，路由会自动调用消息的 `Ack()` 方法；若发生错误，路由会调用消息的 `Nack()` 方法通知管理器重发这条消息。

上面只是路由的最基本用法，路由的强大之处在于中间件。

中间件

`watermill` 中内置了几个比较常用的中间件：

- `IgnoreErrors`：可以忽略指定的错误；
- `Throttle`：限流，限制单位时间内处理的消息数量；

- `Poison` ：将处理失败的消息以另一个主题发布；
- `Retry` ：重试，处理失败可以重试；
- `Timeout` ：超时，如果消息处理时间超过给定的时间，直接失败。
- `InstantAck` ：直接调用消息的 `Ack()` 方法，不管后续成功还是失败；
- `RandomFail` ：随机抛出错误，测试时使用；
- `Duplicator` ：调用两次处理函数，两次返回的消息都重新发布出去，double~
- `Correlation` ：处理函数生成的消息都统一设置成原始消息中的 `correlation id` ，方便追踪消息来源；
- `Recoverer` ：捕获处理函数中的 `panic` ，包装成错误返回。

中间件的使用也是比较简单和直接的：调用 `router.AddMiddleware()` 。例如，我们想要把处理返回的消息 double 一下：

```
router.AddMiddleware(middleware.Duplicator)
```

想重试？可以：

```
router.AddMiddleware(middleware.Retry{
  MaxRetries:      3,
  InitialInterval: time.Millisecond * 100,
  Logger:          logger,
}.Middleware)
```

上面设置最大重试次数为 3，重试初始时间间隔为 100ms。

一般情况下，生产环境需要保证稳定性，某个处理异常不能影响后续的消息处理。故设置 `Recoverer` 是比较好的选择：

```
router.AddMiddleware(middleware.Recoverer)
```

也可以实现自己的中间件：

```
func MyMiddleware(h message.HandlerFunc) message.HandlerFunc {
  return func(message *message.Message) ([]*message.Message, error) {
    fields := watermill.LogFields{"name": m.Name}
    logger.Info("myMiddleware before", fields)
    ms, err := h(message)
    logger.Info("myMiddleware after", fields)
    return ms, err
```

```
    }
}
```

中间件有两种实现方式，如果不需要参数或依赖，那么直接实现为函数即可，像上面这样。如果需要有参数，那么可以实现为一个结构：

```go
type myMiddleware struct {
  Name string
}

func (m myMiddleware) Middleware(h message.HandlerFunc) message.HandlerFunc {
  return func(message *message.Message) ([]*message.Message, error) {
    fields := watermill.LogFields{"name": m.Name}
    logger.Info("myMiddleware before", fields)
    ms, err := h(message)
    logger.Info("myMiddleware after", fields)
    return ms, err
  }
}
```

这两种中间件的添加方式有所不同，第一种直接添加：

```go
router.AddMiddleware(MyMiddleware)
```

第二种要构造一个对象，然后将其 `Middleware` 方法传入，在该方法中可以访问 `MyMiddleware` 对象的字段：

```go
router.AddMiddleware(MyMiddleware{Name:"dj"}.Middleware)
```

设置

如果运行上面程序，你很可能会看到这样一条日志：

```
No subscribers to send message
```

因为发布消息是在另一个 `goroutine` ，我们没有控制何时发布，可能发布消息时，我们还未订阅。我们观察后面的处理日志，对比 uuid 发现这条消息直接被丢弃了。 `watermill` 提供了一个选项，可以将消息都保存下来，订阅某个主题时将该主题之前的消息也发送给它：

```
pubSub := gochannel.NewGoChannel(
  gochannel.Config{
    Persistent: true,
  }, logger)
```

创建 `GoChannel` 时将 `Config` 中 `Persistent` 字段设置为 `true` 即可。此时运行，我们仔细观察一下，出现 `No subscribers to send message` 信息的消息后续确实被处理了。

RabbitMQ

除了 `GoChannel` ， `watermill` 还内置了其他的发布-订阅实现。这些实现除了发布-订阅器创建的方式不同，其他与我们之前介绍的基本一样。这里我们简单介绍一下 `RabbitMQ` ，其他的可自行研究。

使用 `RabbitMQ` 需要先运行 `RabbitMQ` 程序， `RabbitMQ` 采用 `Erlang` 开发。我们之前很多文章也介绍过 windows 上的软件安装神器 `choco` 。使用 `choco` 安装 `RabbitMQ` ：

```
$ choco install rabbitmq
```

启动 `RabbitMQ` 服务器：

```
$ rabbitmq-server.bat
```

`watermill` 对 `RabbitMQ` 的支持使用独立库的形式，需要另行安装：

```
$ go get -u github.com/ThreeDotsLabs/watermill-amqp/pkg/amqp
```

发布订阅：

```
var amqpURI = "amqp://localhost:5672/"

func main() {
  amqpConfig := amqp.NewDurableQueueConfig(amqpURI)

  subscriber, err := amqp.NewSubscriber(
    amqpConfig,
    watermill.NewStdLogger(false, false),
  )
  if err != nil {
    panic(err)
```

```
	}

	messages, err := subscriber.Subscribe(context.Background(), "example.topic")
	if err != nil {
		panic(err)
	}

	go process(messages)

	publisher, err := amqp.NewPublisher(amqpConfig, watermill.NewStdLogger(false, false))
	if err != nil {
		panic(err)
	}

	publishMessages(publisher)
}

func publishMessages(publisher message.Publisher) {
	for {
		msg := message.NewMessage(watermill.NewUUID(), []byte("Hello, world!"))

		if err := publisher.Publish("example.topic", msg); err != nil {
			panic(err)
		}

		time.Sleep(time.Second)
	}
}

func process(messages <-chan *message.Message) {
	for msg := range messages {
		log.Printf("received message: %s, payload: %s", msg.UUID, string(msg.Payload))
		msg.Ack()
	}
}
```

如果有自定义发布-订阅实现的需求，可以参考 `RabbitMQ` 的实现：github.com/ThreeDotsLabs/watermill-amqp/pkg/amqp。

总结

`watermill` 提供丰富的功能，且预留了扩展点，可自行扩展。另外，源码中处理 `goroutine` 创建和通信、多种并发模式的应用都是值得一看的。官方 GitHub 上还有一

个事件驱动示例：https://github.com/ThreeDotsLabs/event-driven-example。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. watermill 官方文档：https://watermill.io/
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

message-bus

简介

在一个涉及多模块交互的系统中，如果模块的交互需要手动去调用对方的方法，那么代码的耦合度就太高了。所以产生了异步消息通信。实际上，各种各样的消息队列都是基于异步消息的。不过它们大部分都有着非常复杂的设计，很多被设计成一个独立的软件来使用。今天我们介绍一个非常小巧的异步消息通信库 `[message-bus]` (https://github.com/vardius/message-bus)，它只能在一个进程中使用。源代码只有一个文件，我们也简单看一下实现。

快速使用

安装：

```
$ go get github.com/vardius/message-bus
```

使用：

```
package main

import (
  "fmt"
  "sync"

  messagebus "github.com/vardius/message-bus"
)

func main() {
  queueSize := 100
  bus := messagebus.New(queueSize)

  var wg sync.WaitGroup
  wg.Add(2)

  _ = bus.Subscribe("topic", func(v bool) {
    defer wg.Done()
    fmt.Println(v)
  })

  _ = bus.Subscribe("topic", func(v bool) {
```

```
        defer wg.Done()
        fmt.Println(v)
    })

    bus.Publish("topic", true)
    wg.Wait()
}
```

这是官网提供的例子， `message-bus` 承担了模块间消息分发的角色。模块 A 和 模块 B 先向 `message-bus` 订阅主题（**topic**），即告诉 `message-bus` 对什么样的消息感兴趣。其他模块 C 产生某个主题的消息，通知 `message-bus` ，由 `message-bus` 分发到对此感兴趣的模块。这样就实现了模块之间的解耦，模块 A、B 和 C 之间不需要知道彼此。

上面的例子中：

- 首先，调用 `messagebuss.New()` 创建一个消息管理器；
- 其次调用 `Subscribe()` 方法向管理器订阅主题；
- 调用 `Publish()` 向管理器发布主题消息，这样订阅该主题的模块就会收到通知。

更复杂的例子

其实很多人会对何时使用这个库产生疑问， `message-bus` GitHub 仓库中 issue 中至今还躺着这个问题，https://github.com/vardius/message-bus/issues/4。我是做游戏后端开发的，在一个游戏中会涉及各种各样的功能模块，它们需要了解其他模块产生的事件。例如每日任务有玩家升多少级的任务、成就系统有等级的成就、其他系统还可能根据玩家等级执行其他操作...如果硬写的话，最后可能是这样：

```
func (p *Player) LevelUp() {
  // ...
  p.DailyMission.OnPlayerLevelUp(oldLevel, newLevel)
  p.Achievement.OnPlayerLevelUp(oldLevel, newLevel)
  p.OtherSystem.OnPlayerLevelUp(oldLevel, newLevel)
}
```

需求一直在新增和迭代，如果新增一个模块，也需要在玩家升级时进行一些处理，除了实现模块自身的 `OnPlayerLevelUp` 方法，还必须在玩家的 `LevelUp()` 方法调用。这样玩家模块必须清楚地知道其他模块的情况。如果功能模块再多一点，而且由不同的人开发的，那么情况会更复杂。使用异步消息可有效解决这个问题：在升级时我们只需要向消息管理器发布这个升级"消息"，由消息管理器通知订阅该消息的模块。

我们设计的目录结构如下：

```
game
├── achievement.go
├── daily_mission.go
├── main.go
├── manager.go
└── player.go
```

其中 `manager.go` 负责 `message-bus` 的创建：

```
package main

import (
  messagebus "github.com/vardius/message-bus"
)

var bus = messagebus.New(10)
```

`player.go` 对应玩家结构（为了简便起见，很多字段省略了）：

```
package main

type Player struct {
  level uint32
}

func NewPlayer() *Player {
  return &Player{}
}

func (p *Player) LevelUp() {
  oldLevel := p.level
  newLevel := p.level+1
  p.level++

  bus.Publish("UserLevelUp", oldLevel, newLevel)
}
```

`achievement.go` 和 `daily_mission.go` 分别是成就和每日任务（也是省略了很多无关细节）：

```
// achievement.go
package main
```

```
import "fmt"

type Achievement struct {
  // ...
}

func NewAchievement() *Achievement {
  a := &Achievement{}
  bus.Subscribe("UserLevelUp", a.OnUserLevelUp)
  return a
}

func (a *Achievement) OnUserLevelUp(oldLevel, newLevel uint32) {
  fmt.Printf("daily mission old level:%d new level:%d\n", oldLevel, newLevel)
}
```

```
// daily_mission.go
package main

import "fmt"

type DailyMission struct {
  // ...
}

func NewDailyMission() *DailyMission {
  d := &DailyMission{}
  bus.Subscribe("UserLevelUp", d.OnUserLevelUp)
  return d
}

func (d *DailyMission) OnUserLevelUp(oldLevel, newLevel uint32) {
  fmt.Printf("daily mission old level:%d new level:%d\n", oldLevel, newLevel)
}
```

在创建这两个功能的对象时，我们订阅了 `UserLevelUp` 主题。玩家在升级时会发布这个主题。

最后 `main.go` 驱动整个程序：

```
package main

import "time"

func main() {
```

```
  p := NewPlayer()
  NewDailyMission()
  NewAchievement()

  p.LevelUp()
  p.LevelUp()
  p.LevelUp()

  time.Sleep(1000)
}
```

注意，由于 `message-bus` 是异步通信，为了能看到结果我特意加了 `time.Sleep` ，实际开发中不太可能使用 `Sleep` 。

最后我们运行整个程序：

```
$ go run .
```

因为要运行的是一个多文件程序，不能使用 `go run main.go` ！

实际上，当年我因为苦于模块之间调来调去太麻烦了，自己用 C++ 撸了一个 `event-manager` ，https://github.com/darjun/event-manager。思路是一样的。

缺点

`message-bus` 订阅主题时传入一个函数，函数的参数可任意设置，发布时必须使用相同数量的参数，这个限制感觉有点勉强。如果我们传入的参数个数不一致，程序就 `panic` 了。我认为可以只用一个参数 `interface{}` ，传入对象即可。例如，上面的升级事件可以使用 `EventUserLevelUp` 的对象：

```
type EventUserLevelUp struct {
  oldLevel uint32
  newLevel uint32
}
```

对应地修改一下 `Player` 的 `LevelUp` 方法：

```
func (p *Player) LevelUp() {
  event := &EventUserLevelUp {
    oldLevel: p.level,
    newLevel: p.level+1,
  }
  p.level++
```

```
	bus.Publish("UserLevelUp", event)
}
```

和处理方法：

```
func (d *DailyMission) OnUserLevelUp(arg interface{}) {
	event := arg.(*EventUserLevelUp)
	fmt.Printf("daily mission old level:%d new level:%d\n", event.oldLevel, event.newLevel)
}
```

这样一来，我们似乎用不上反射了，订阅者都是 `func (interface{})` 类型的函数或方法。感兴趣的可自己实现一下，我 fork 了 `message-bus` ，做了这个修改。改动在这里：https://github.com/darjun/message-bus，`message-bus` 有测试和性能用例，改完跑一下😄。

源码分析

`message-bus` 的源码只有一个文件，加上注释还不到 130 行，我们简单来看一下。

`MessageBus` 就是一个简单的接口：

```
type MessageBus interface {
	Publish(topic string, args ...interface{})
	Close(topic string)
	Subscribe(topic string, fn interface{}) error
	Unsubscribe(topic string, fn interface{}) error
}
```

`Publish` 和 `Subscribe` 都讲过了，`Unsubscribe` 表示对某个主题不感兴趣了，取消订阅，`Close` 直接关闭某个主题的队列，删除所有订阅者。

在 `message-bus` 内部，每个主题对应一组订阅者。每个订阅者使用 `handler` 结构存储回调和参数通道：

```
type handler struct {
	callback reflect.Value
	queue    chan []reflect.Value
}
```

所有订阅者都存储在一个 map 中：

```
type handlersMap map[string][]*handler

type messageBus struct {
  handlerQueueSize int
  mtx              sync.RWMutex
  handlers         handlersMap
}
```

`messageBus` 是 `MessageBus` 接口的实现。我们来看看各个方法是如何实现的。

```
func (b *messageBus) Subscribe(topic string, fn interface{}) error {
  h := &handler{
    callback: reflect.ValueOf(fn),
    queue:    make(chan []reflect.Value, b.handlerQueueSize),
  }

  go func() {
    for args := range h.queue {
      h.callback.Call(args)
    }
  }()

  b.handlers[topic] = append(b.handlers[topic], h)
  return nil
}
```

调用 `Subscribe` 时传入一个函数， `message-bus` 为每个订阅者创建一个 `handler` 对象，在该对象中创建一个带缓冲的参数通道，缓冲大小由 `message-bus` 创建时的参数指定。
同时启动一个 `goroutine` ，监听通道，每当有参数到来时就执行注册的回调。

```
func (b *messageBus) Publish(topic string, args ...interface{}) {
  rArgs := buildHandlerArgs(args)
  if hs, ok := b.handlers[topic]; ok {
    for _, h := range hs {
      h.queue <- rArgs
    }
  }
}
```

`Publish` 发布主题， `buildHandlerArgs` 将传入的参数转为 `[]reflect.Value` ，以便反射调用回调时传入。发送参数到该主题下所有 `handler` 的通道中。
由 `Subscribe` 时创建的 `goroutine` 读取并触发回调。

```
func (b *messageBus) Unsubscribe(topic string, fn interface{}) error {
	rv := reflect.ValueOf(fn)

	if _, ok := b.handlers[topic]; ok {
		for i, h := range b.handlers[topic] {
			if h.callback == rv {
				close(h.queue)

				b.handlers[topic] = append(b.handlers[topic][:i], b.handlers[topic][i+1
:]...)
			}
		}

		return nil
	}

	return fmt.Errorf("topic %s doesn't exist", topic)
}
```

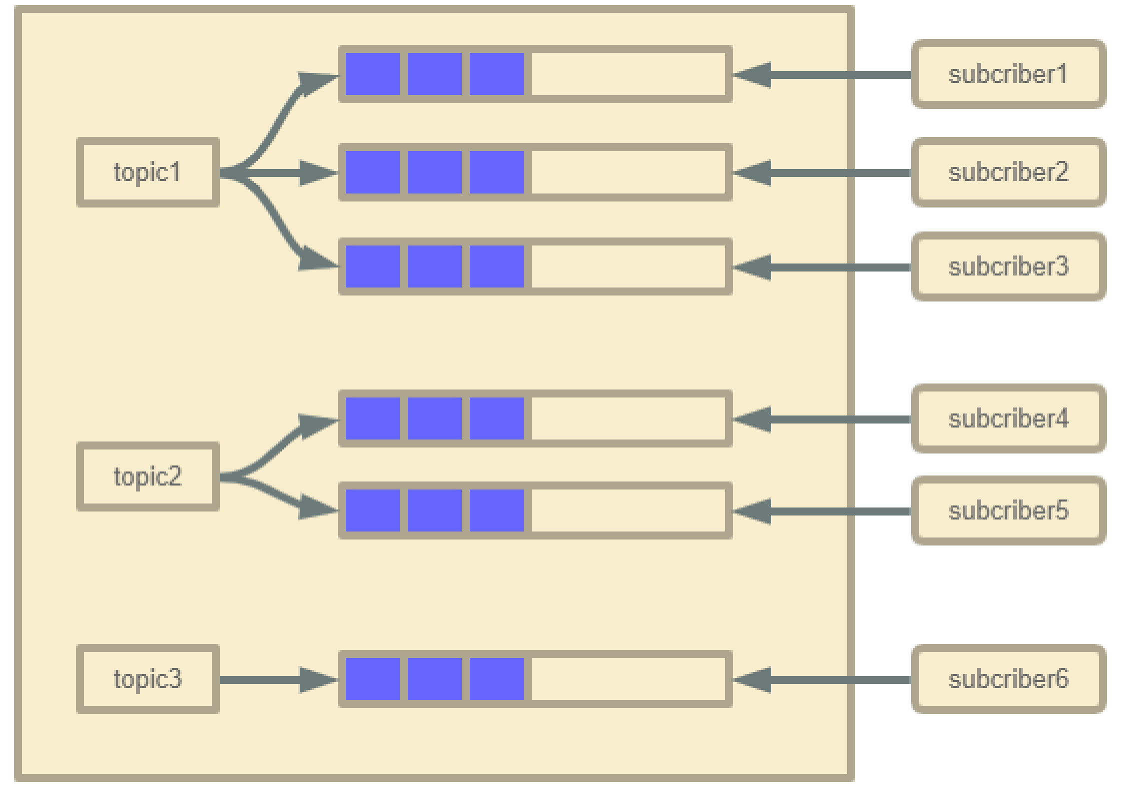
`Unsubscribe` 将某个订阅者从 `message-bus` 中移除，移除时需要关闭通道，否则会造成订阅者的 goroutine 泄露。

```
func (b *messageBus) Close(topic string) {
	if _, ok := b.handlers[topic]; ok {
		for _, h := range b.handlers[topic] {
			close(h.queue)
		}

		delete(b.handlers, topic)

		return
	}
}
```

`Close` 关闭某主题下所有的订阅者参数通道，并删除该主题。

注意，为了保证并发安全，每个方法都加了锁，分析实现时先忽略锁和错误处理。

为了更直观的理解，我画了一个 `message-bus` 内部结构图：



总结

`message-bus` 是一个小巧的异步通信库，实际使用可能不多，但却是学习源码的好资源。

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. message-bus GitHub：https://github.com/vardius/message-bus
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

gojsonq

简介

在日常工作中，每一名开发者，不管是前端还是后端，都经常使用 JSON。JSON 是一个很简单的数据交换格式。相比于 XML，它灵活、轻巧、使用方便。JSON 也是RESTful API推荐的格式。有时，我们只想读取 JSON 中的某一些字段。如果自己手动解析、一层一层读取，这就变得异常繁琐了。特别是在嵌套层次很深的情况下。今天我们介绍 `gojsonq` 。它可以帮助我们很方便的操作 JSON。

快速使用

先安装：

```
$ go get github.com/thedevsaddam/gojsonq
```

后使用：

```
package main

import (
  "fmt"

  "github.com/thedevsaddam/gojsonq"
)

func main() {
  content := `{
  "user": {
    "name": "dj",
    "age": 18,
    "address": {
      "provice": "shanghai",
      "district": "xuhui"
    },
    "hobbies":["chess", "programming", "game"]
  }
}`

  gq := gojsonq.New().FromString(content)
  district := gq.Find("user.address.district")
```

```
    fmt.Println(district)

    gq.Reset()

    hobby := gq.Find("user.hobbies.[0]")
    fmt.Println(hobby)
}
```

操作非常简单：

- 首先调用`gojsonq.New()`创建一个`JSONQ`的对象；
- 然后就可以使用该类型的方法来查询属性了。

上面代码我们直接读取位于最内层的 `district` 值和 `hobbies` 数组的第一个元素！层与层之间用 `.` 隔开，如果是数组，则在属性字段后通过 `.[index]` 读取下标为 `index` 的元素。这种方式可以实现很灵活的读取。

注意到一个细节：在查询之后，我们手动调用了一次 `Reset()` 方法。因为 `JSONQ` 对象在调用 `Find` 方法时，内部会记录当前的节点，下一个查询会从上次查找的节点开始。也就是说如果我们注释掉 `jq.Reset()` ，第二个 `Find()` 方法实际上查找的是 `user.address.district.user.hobbies.[0]` ，自然就返回 `nil` 了。除此之外， `gojsonq` 也提供了另外一种方式。如果你想要保存当前查询的一些状态信息，可以调用 `JSONQ` 的 `Copy` 方法返回一个初始状态下的对象，它们会共用底层的 JSON 字符串和解析后的对象。上面的 `gq.Reset()` 可以由下面这行代码代替：

```
gpCopy := gp.Copy()
```

后面就可以使用 `gpCopy` 查询 `hobbies` 了。

这个算是 `gojsonq` 库的一个特点，但也是初学者带来了很多困扰，需要特别注意。实际上， `JSONQ` 提供的很多方法会改变当前节点，稍后部分我们会更清楚的看到。

数据源

除了从字符串中加载， `jsonq` 还允许从文件和 `io.Reader` 中读取内容。分别使用 `JSONQ` 对象的 `File` 和 `Reader` 方法：

```
func main() {
    gq := gojsonq.New().File("./data.json")

    fmt.Println(gq.Find("items.[1].price"))
}
```

和下面程序的效果是一样的：

```
func main() {
  file, err := os.OpenFile("./data.json", os.O_RDONLY, 0666)
  if err != nil {
    log.Fatal(err)
  }

  gq := gojsonq.New().Reader(file)

  fmt.Println(gq.Find("items.[1].price"))
}
```

为了后面演示方便，我构造了一个 `data.json` 文件：

```
{
  "name": "shopping cart",
  "description": "List of items in your cart",
  "prices": ["2400", "2100", "1200", "400.87", "89.90", "150.10"],
  "items": [
    {
      "id": 1,
      "name": "Apple",
      "count": 2,
      "price": 12
    },
    {
      "id": 2,
      "name": "Notebook",
      "count": 10,
      "price": 3
    },
    {
      "id": 3,
      "name": "Pencil",
      "count": 5,
      "price": 1
    },
    {
      "id": 4,
      "name": "Camera",
      "count": 1,
      "price": 1750
    },
    {
```

```
      "id": null,
      "name": "Invalid Item",
      "count": 1,
      "price": 12000
    }
  ]
}
```

高级查询

`gojsonq` 的独特之处在于，它可以像 SQL 一样进行条件查询，可以选择返回哪些字段，可以做一些聚合统计。

字段映射

有时候，我们只关心对象中的几个字段，这时候就可以使用 `Select` 指定返回哪些字段，其余字段不返回：

```
func main() {
  r := gojsonq.New().File("./data.json").From("items").Select("id", "name").Get
()
  data, _ := json.MarshalIndent(r, "", "  ")
  fmt.Println(string(data))
}
```

只会输出 `id` 和 `name` 字段：

```
$ go run main.go
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Notebook"
  },
  {
    "id": 3,
    "name": "Pencil"
  },
  {
    "id": 4,
```

```
    "name": "Camera"
  },
  {
    "id": null,
    "name": "Invalid Item"
  }
]
```

为了显示更直观一点，我这里用 `json.MarshalIndent()` 对输出做了一些美化。

是不是和 SQL 有点像 `Select id,name From items` ...

这里介绍一下 `From` 方法，这个方法的作用是将当前节点移动到指定位置。上面也说过当前节点的位置是记下来的。例如，上面的代码中我们先将当前节点移动到 `items` ，后面的查询和聚合操作都是针对这个数组。实际上 `Find` 方法内部就调用了 `From` ：

```go
// src/github.com/thedevsaddam/gojsonq/jsonq.go
func (j *JSONQ) Find(path string) interface{} {
  return j.From(path).Get()
}

func (j *JSONQ) From(node string) *JSONQ {
  j.node = node
  v, err := getNestedValue(j.jsonContent, node, j.option.separator)
  if err != nil {
    j.addError(err)
  }
  // ============= 注意这一行，记住当前节点位置
  j.jsonContent = v
  return j
}
```

最后必须要调用 `Get()` ，它组合所有条件后执行这个查询，返回结果。

条件查询

有了 `Select` 和 `From` ，怎么能没有 `Where` 呢？ `gojsonq` 提供的 `Where` 方法非常多，我们大概看几个就行了。

首先是， `Where(key, op, val)` ，这个是通用的 `Where` 条件，表示 `key` 和 `val` 是否满足 `op` 关系。 `op` 内置的就有将近 20 种，还支持自定义。例如 `=` 表示相等， `!=` 表示不等， `startsWith` 表示 `val` 是否是 `key` 字段的前缀等等等等；

其他很多条件都是 `Where` 的特例，例如 `WhereIn(key, val)` 就等价于 `Where(key, "in", val)`，`WhereStartsWith(key, val)` 就等价于 `Where(key, "startsWith", val)`。

默认情况下，`Where` 的条件都是 `And` 连接的，我们可以通过 `OrWhere` 让其以 `Or` 连接：

```
func main() {
  gq := gojsonq.New().File("./data.json")

  r := gq.From("items").Select("id", "name").
    Where("id", "=", 1).OrWhere("id", "=", 2).Get()
  fmt.Println(r)

  gq.Reset()

  r = gq.From("items").Select("id", "name", "count").
    Where("count", ">", 1).Where("price", "<", 100).Get()
  fmt.Println(r)
}
```

上面第一个查询，查找 `id` 为 1 **或** 2 的记录。第二个查询，查找 `count` 大于 1 **且** `price` 小于 100 的记录。

指定偏移和返回条目数

有时我们想要分页显示，第一次查询时返回前 3 条内容，第二次查询时返回接下来的 3 条记录。我们可以使用 `JSONQ` 对象的 `Offset` 和 `Limit` 方法来指定偏移和返回的条目数：

```
func main() {
  gq := gojsonq.New().File("./data.json")

  r1 := gq.From("items").Select("id", "name").Offset(0).Limit(3).Get()
  fmt.Println("First Page:", r1)

  gq.Reset()

  r2 := gq.From("items").Select("id", "name").Offset(3).Limit(3).Get()
  fmt.Println("Second Page:", r2)
}
```

来看看运行结果：

```
$ go run main.go
First Page: [map[id:1 name:Apple] map[id:2 name:Notebook] map[id:3 name:Pencil]]
Second Page: [map[id:4 name:Camera] map[id:<nil> name:Invalid Item]]
```

聚合统计

我们还能可以对一些字段做简单的统计，计算和、平均数、最大、最小值等：

```
func main() {
  gq := gojsonq.New().File("./data.json").From("items")

  fmt.Println("Total Count:", gq.Sum("count"))
  fmt.Println("Min Price:", gq.Min("price"))
  fmt.Println("Max Price:", gq.Max("price"))
  fmt.Println("Avg Price:", gq.Avg("price"))
}
```

上面统计商品的总数量、最低价格、最高价格和平均价格。

聚合统计类的方法都不会修改当前节点的指向，所以 `JSONQ` 对象可以重复使用！

还可以对数据进行分组和排序：

```
func main() {
  gq := gojsonq.New().File("./data.json")

  fmt.Println(gq.From("items").GroupBy("price").Get())
  gq.Reset()
  fmt.Println(gq.From("items").SortBy("price", "desc").Get())
}
```

其他格式

默认情况下， `gojsonq` 使用 JSON 格式解析数据。我们也可以设置其他格式解析器让 `gojsonq` 可以处理其他格式的数据：

```
func main() {
  jq := gojsonq.New(gojsonq.SetDecoder(&yamlDecoder{})).File("./data.yaml")
  jq.From("items").Where("price", "<=", 500)
  fmt.Printf("%v\n", jq.First())
}
```

```go
type yamlDecoder struct {
}

func (i *yamlDecoder) Decode(data []byte, v interface{}) error {
  bb, err := yaml.YAMLToJSON(data)
  if err != nil {
    return err
  }
  return json.Unmarshal(bb, &v)
}
```

上面代码用到了 `yaml` 库，需要额外安装：

```
$ go get github.com/ghodss/yaml
```

解析器只要实现 `gojsonq.Decoder` 接口，都可以作为设置到 `gojsonq` 中，这样就可以实现任何格式的处理：

```go
// src/github.com/thedevsaddam/gojsonq/decoder.go
type Decoder interface {
  Decode(data []byte, v interface{}) error
}
```

总结

`gojsonq` 还有一些高级特性，例如自定义 `Where` 的操作类型，取第一个、最后一个、第 N 个值等。感兴趣可自行研究~

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. gojsonq GitHub：https://github.com/thedevsaddam/gojsonq
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

dig

简介

今天我们来介绍 Go 语言的一个依赖注入（DI）库——dig。dig 是 uber 开源的库。Java 依赖注入的库有很多，相信即使不是做 Java 开发的童鞋也听过大名鼎鼎的 Spring。相比庞大的 Spring，dig 很小巧，实现和使用都比较简洁。

快速使用

第三方库需要先安装，由于我们的示例中使用了前面介绍的go-ini和go-flags，这两个库也需要安装：

```
$ go get go.uber.org/dig
$ go get gopkg.in/ini.v1
$ go get github.com/jessevdk/go-flags
```

下面看看如何使用：

```
package main

import (
  "fmt"

  "github.com/jessevdk/go-flags"
  "go.uber.org/dig"
  "gopkg.in/ini.v1"
)

type Option struct {
  ConfigFile string `short:"c" long:"config" description:"Name of config file."`
}

func InitOption() (*Option, error) {
  var opt Option
  _, err := flags.Parse(&opt)

  return &opt, err
}

func InitConf(opt *Option) (*ini.File, error) {
```

```
	cfg, err := ini.Load(opt.ConfigFile)
	return cfg, err
}

func PrintInfo(cfg *ini.File) {
	fmt.Println("App Name:", cfg.Section("").Key("app_name").String())
	fmt.Println("Log Level:", cfg.Section("").Key("log_level").String())
}

func main() {
	container := dig.New()

	container.Provide(InitOption)
	container.Provide(InitConf)

	container.Invoke(PrintInfo)
}
```

在同一目录下创建配置文件 `my.ini` ：

```
app_name = awesome web
log_level = DEBUG

[mysql]
ip = 127.0.0.1
port = 3306
user = dj
password = 123456
database = awesome

[redis]
ip = 127.0.0.1
port = 6381
```

运行程序，输出：

```
$ go run main.go -c=my.ini
App Name: awesome web
Log Level: DEBUG
```

`dig` 库帮助开发者管理这些对象的创建和维护，每种类型的对象会**创建且只创建一次**。 `dig` 库使用的一般流程：

- 创建一个容器： `dig.New` ；

- 为想要让 `dig` 容器管理的类型创建构造函数，构造函数可以返回多个值，这些值都会被容器管理；
- 使用这些类型的时候直接编写一个函数，将这些类型作为参数，然后使用 `container.Invoke` 执行我们编写的函数。

参数对象

有时候，创建对象有很多依赖，或者编写函数时有多个参数依赖。如果将这些依赖都作为参数传入，那么代码将变得非常难以阅读：

```
container.Provide(func (arg1 *Arg1, arg2 *Arg2, arg3 *Arg3, ....) {
  // ...
})
```

`dig` 支持将所有参数打包进一个对象中，唯一需要的就是将 `dig.In` 内嵌到该类型中：

```
type Params {
  dig.In

  Arg1 *Arg1
  Arg2 *Arg2
  Arg3 *Arg3
  Arg4 *Arg4
}

container.Provide(func (params Params) *Object {
  // ...
})
```

内嵌了 `dig.In` 之后， `dig` 会将该类型中的其它字段看成 `Object` 的依赖，创建 `Object` 类型的对象时，会先将依赖的 `Arg1/Arg2/Arg3/Arg4` 创建好。

```
package main

import (
  "fmt"
  "log"

  "github.com/jessevdk/go-flags"
  "go.uber.org/dig"
  "gopkg.in/ini.v1"
```

```
)

type Option struct {
  ConfigFile string `short:"c" long:"config" description:"Name of config file."`
}

type RedisConfig struct {
  IP   string
  Port int
  DB   int
}

type MySQLConfig struct {
  IP       string
  Port     int
  User     string
  Password string
  Database string
}

type Config struct {
  dig.In

  Redis *RedisConfig
  MySQL *MySQLConfig
}

func InitOption() (*Option, error) {
  var opt Option
  _, err := flags.Parse(&opt)

  return &opt, err
}

func InitConfig(opt *Option) (*ini.File, error) {
  cfg, err := ini.Load(opt.ConfigFile)
  return cfg, err
}

func InitRedisConfig(cfg *ini.File) (*RedisConfig, error) {
  port, err := cfg.Section("redis").Key("port").Int()
  if err != nil {
    log.Fatal(err)
    return nil, err
  }
```

```
    db, err := cfg.Section("redis").Key("db").Int()
    if err != nil {
        log.Fatal(err)
        return nil, err
    }

    return &RedisConfig{
        IP:   cfg.Section("redis").Key("ip").String(),
        Port: port,
        DB:   db,
    }, nil
}

func InitMySQLConfig(cfg *ini.File) (*MySQLConfig, error) {
    port, err := cfg.Section("mysql").Key("port").Int()
    if err != nil {
        return nil, err
    }

    return &MySQLConfig{
        IP:       cfg.Section("mysql").Key("ip").String(),
        Port:     port,
        User:     cfg.Section("mysql").Key("user").String(),
        Password: cfg.Section("mysql").Key("password").String(),
        Database: cfg.Section("mysql").Key("database").String(),
    }, nil
}

func PrintInfo(config Config) {
    fmt.Println("=========== redis section ===========")
    fmt.Println("redis ip:", config.Redis.IP)
    fmt.Println("redis port:", config.Redis.Port)
    fmt.Println("redis db:", config.Redis.DB)

    fmt.Println("=========== mysql section ===========")
    fmt.Println("mysql ip:", config.MySQL.IP)
    fmt.Println("mysql port:", config.MySQL.Port)
    fmt.Println("mysql user:", config.MySQL.User)
    fmt.Println("mysql password:", config.MySQL.Password)
    fmt.Println("mysql db:", config.MySQL.Database)
}

func main() {
    container := dig.New()

    container.Provide(InitOption)
```

```
    container.Provide(InitConfig)
    container.Provide(InitRedisConfig)
    container.Provide(InitMySQLConfig)

    err := container.Invoke(PrintInfo)
    if err != nil {
        log.Fatal(err)
    }
}
```

上面代码中，类型 `Config` 内嵌了 `dig.In` ， `PrintInfo` 接受一个 `Config` 类型的参数。调用 `Invoke` 时， `dig` 自动调用 `InitRedisConfig` 和 `InitMySQLConfig` ，并将生成的 `*RedisConfig` 和 `*MySQLConfig` “打包”成一个 `Config` 对象传给 `PrintInfo` 。

运行结果：

```
$ go run main.go -c=my.ini
=========== redis section ===========
redis ip: 127.0.0.1
redis port: 6381
redis db: 1
=========== mysql section ===========
mysql ip: 127.0.0.1
mysql port: 3306
mysql user: dj
mysql password: 123456
mysql db: awesome
```

结果对象

前面说过，如果构造函数返回多个值，这些不同类型的值都会存储到 `dig` 容器中。参数过多会影响代码的可读性和可维护性，返回值过多同样也是如此。为此， `dig` 提供了返回值对象，返回一个包含多个类型对象的对象。返回的类型，必须内嵌 `dig.Out` ：

```
type Results struct {
    dig.Out

    Result1 *Result1
    Result2 *Result2
    Result3 *Result3
    Result4 *Result4
}
```

```
dig.Provide(func () (Results, error) {
  // ...
})
```

我们把上面的例子稍作修改。将 `Config` 内嵌的 `dig.In` 变为 `dig.Out` ：

```
type Config struct {
  dig.Out

  Redis *RedisConfig
  MySQL *MySQLConfig
}
```

提供构造函数 `InitRedisAndMySQLConfig` 同时创建 `RedisConfig` 和 `MySQLConfig` ，通过 `Config` 返回。这样就不需要将 `InitRedisConfig` 和 `InitMySQLConfig` 加入 `dig` 容器了：

```
func InitRedisAndMySQLConfig(cfg *ini.File) (Config, error) {
  var config Config

  redis, err := InitRedisConfig(cfg)
  if err != nil {
    return config, err
  }

  mysql, err := InitMySQLConfig(cfg)
  if err != nil {
    return config, err
  }

  config.Redis = redis
  config.MySQL = mysql
  return config, nil
}

func main() {
  container := dig.New()

  container.Provide(InitOption)
  container.Provide(InitConfig)
  container.Provide(InitRedisAndMySQLConfig)

  err := container.Invoke(PrintInfo)
```

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

`PrintInfo` 直接依赖 `RedisConfig` 和 `MySQLConfig` ：

```
func PrintInfo(redis *RedisConfig, mysql *MySQLConfig) {
    fmt.Println("=========== redis section ===========")
    fmt.Println("redis ip:", redis.IP)
    fmt.Println("redis port:", redis.Port)
    fmt.Println("redis db:", redis.DB)

    fmt.Println("=========== mysql section ===========")
    fmt.Println("mysql ip:", mysql.IP)
    fmt.Println("mysql port:", mysql.Port)
    fmt.Println("mysql user:", mysql.User)
    fmt.Println("mysql password:", mysql.Password)
    fmt.Println("mysql db:", mysql.Database)
}
```

可以看到 `InitRedisAndMySQLConfig` 返回 `Config` 类型的对象，该类型中的 `RedisConfig` 和 `MySQLConfig` 都被添加到了容器中， `PrintInfo` 函数可直接使用。

运行结果与之前的例子完全一样。

可选依赖

默认情况下，容器如果找不到对应的依赖，那么相应的对象无法创建成功，调用 `Invoke` 时也会返回错误。有些依赖不是必须的， `dig` 也提供了一种方式将依赖设置为可选的：

```
type Config struct {
    dig.In

    Redis *RedisConfig `optional:"true"`
    MySQL *MySQLConfig
}
```

通过在字段后添加结构标签 `optional:"true"` ，我们将 `RedisConfig` 这个依赖设置为可选的，容器中 `RedisConfig` 对象也不要紧，这时传入的 `Config` 中 `redis` 为 nil，方法可以正常调用。显然可选依赖只能在参数对象中使用。

我们直接注释掉 `InitRedisConfig` ，然后运行程序：

```go
// 省略部分代码
func PrintInfo(config Config) {
  if config.Redis == nil {
    fmt.Println("no redis config")
  }
}

func main() {
  container := dig.New()

  container.Provide(InitOption)
  container.Provide(InitConfig)
  container.Provide(InitMySQLConfig)

  container.Invoke(PrintInfo)
}
```

输出：

```
$ go run main.go -c=my.ini
no redis config
```

注意，创建失败和没有提供构造函数是两个概念。如果 `InitRedisConfig` 调用失败了，使用 `Invoke` 执行 `PrintInfo` 还是会报错的。

命名

前面我们说过，`dig` 默认只会为每种类型创建一个对象。如果要创建某个类型的多个对象怎么办呢？可以为对象命名！

调用容器的 `Provide` 方法时，可以为构造函数的返回对象命名，这样同一个类型就可以有多个对象了。

```go
type User struct {
  Name string
  Age  int
}

func NewUser(name string, age int) func() *User{} {
  return func() *User {
    return &User{name, age}
  }
}
```

```
container.Provide(NewUser("dj", 18), dig.Name("dj"))
container.Provide(NewUser("dj2", 18), dig.Name("dj2"))
```

也可以在结果对象中通过结构标签指定：

```
type UserResults struct {
  dig.Out

  User1 *User `name:"dj"`
  User2 *User `name:"dj2"`
}
```

然后在参数对象中通过名字指定使用哪个对象：

```
type UserParams struct {
  dig.In

  User1 *User `name:"dj"`
  User2 *User `name:"dj2"`
}
```

完整代码：

```
package main

import (
  "fmt"

  "go.uber.org/dig"
)

type User struct {
  Name string
  Age  int
}

func NewUser(name string, age int) func() *User {
  return func() *User {
    return &User{name, age}
  }
}

type UserParams struct {
  dig.In
```

```
  User1 *User `name:"dj"`
  User2 *User `name:"dj2"`
}

func PrintInfo(params UserParams) error {
  fmt.Println("User 1 ===========")
  fmt.Println("Name:", params.User1.Name)
  fmt.Println("Age:", params.User1.Age)

  fmt.Println("User 2 ===========")
  fmt.Println("Name:", params.User2.Name)
  fmt.Println("Age:", params.User2.Age)
  return nil
}

func main() {
  container := dig.New()

  container.Provide(NewUser("dj", 18), dig.Name("dj"))
  container.Provide(NewUser("dj2", 18), dig.Name("dj2"))

  container.Invoke(PrintInfo)
}
```

程序运行结果：

```
$ go run main.go
User 1 ===========
Name: dj
Age: 18
User 2 ===========
Name: dj2
Age: 18
```

需要注意的时候，`NewUser` 返回的是一个函数，由 `dig` 在需要的时候调用。

组

组可以将相同类型的对象放到一个切片中，可以直接使用这个切片。组的定义与上面名字定义类似。可以通过为 `Provide` 提供额外的参数：

```
container.Provide(NewUser("dj", 18), dig.Group("user"))
container.Provide(NewUser("dj2", 18), dig.Group("user"))
```

也可以在结果对象中添加结构标签 `group:"user"` 。

然后我们定义一个参数对象，通过指定同样的结构标签来使用这个切片：

```
type UserParams struct {
  dig.In

  Users []User `group:"user"`
}

func Info(params UserParams) error {
  for _, u := range params.Users {
    fmt.Println(u.Name, u.Age)
  }

  return nil
}

container.Invoke(Info)
```

最后我们通过一个完整的例子演示组的使用，我们将创建一个 HTTP 服务器：

```
package main

import (
  "fmt"
  "net/http"

  "go.uber.org/dig"
)

type Handler struct {
  Greeting string
  Path     string
}

func (h Handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "%s from %s", h.Greeting, h.Path)
}

func NewHello1Handler() HandlerResult {
  return HandlerResult{
    Handler: Handler{
      Path:     "/hello1",
```

```
			Greeting: "welcome",
		},
	}
}

func NewHello2Handler() HandlerResult {
	return HandlerResult{
		Handler: Handler{
			Path:     "/hello2",
			Greeting: "😄",
		},
	}
}

type HandlerResult struct {
	dig.Out

	Handler Handler `group:"server"`
}

type HandlerParams struct {
	dig.In

	Handlers []Handler `group:"server"`
}

func RunServer(params HandlerParams) error {
	mux := http.NewServeMux()
	for _, h := range params.Handlers {
		mux.Handle(h.Path, h)
	}

	server := &http.Server{
		Addr:    ":8080",
		Handler: mux,
	}
	if err := server.ListenAndServe(); err != nil {
		return err
	}

	return nil
}

func main() {
	container := dig.New()
```

```
    container.Provide(NewHello1Handler)
    container.Provide(NewHello2Handler)

    container.Invoke(RunServer)
}
```

我们创建了两个处理器，添加到 `server` 组中，在 `RunServer` 函数中创建 HTTP 服务器，将这些处理器注册到服务器中。

运行程序，在浏览器中输入 `localhost:8080/hello1` 和 `localhost:8080/hello2` 看看。关于 Go Web 编程相关的知识，可以看看我写的 Go Web 编程系列文章：

- Go Web 编程之 Hello World
- Go Web 编程之 程序结构
- Go Web 编程之 请求
- Go Web 编程之 响应
- Go Web 编程之 模板（一）
- Go Web 编程之 模板（二）
- Go Web 编程之 静态文件
- Go Web 编程之 数据库

常见错误

使用 `dig` 过程中会遇到一些错误，我们来看看常见的错误。

`Invoke` 方法在以下几种情况下会返回一个 `error` ：

- 无法找到依赖，或依赖创建失败；
- `Invoke` 执行的函数返回 `error` ，该错误也会被传给调用者。

这两种情况，我们都可以判断 `Invoke` 的返回值来查找原因。

总结

本文介绍了 `dig` 库，它适用于解决循环依赖的对象创建问题。同时也有利于将关注点分离，我们不需要将各种对象传来传去，只需要将构造函数交给 `dig` 容器，然后通过 `Invoke` 直接使用依赖即可，连判空逻辑都可以省略了！

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. dig GitHub：https://github.com/uber-go/dig
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

email

简介

程序中时常有发送邮件的需求。有异常情况了需要通知管理员和负责人，用户下单后可能需要通知订单信息，电商平台、中国移动和联通都有每月账单，这些都可以通过邮件来推送。还有我们平时收到的垃圾邮件大都也是通过这种方式发送的😭。那么如何在 Go 语言发送邮件？本文我们介绍一下 `email` 库的使用。

快速使用

这个库的使用快不了，为什么呢？

先安装库，这个自不必说：

```
$ go get github.com/jordan-wright/email
```

我们需要额外一些工作。我们知道邮箱使用 `SMTP/POP3/IMAP` 等协议从邮件服务器上拉取邮件。邮件并不是直接发送到邮箱的，而是邮箱请求拉取的。
所以，我们需要配置 `SMTP/POP3/IMAP` 服务器。从头搭建固然可行，而且也有现成的开源库，但是比较麻烦。现在一般的邮箱服务商都开放了 `SMTP/POP3/IMAP` 服务器。
我这里拿 126 邮箱来举例，使用 `SMTP` 服务器。当然，用 QQ 邮箱也可以。

- 首先，登录邮箱；
- 点开顶部的设置，选择 `POP3/SMTP/IMAP`；
- 点击开启 `IMAP/SMTP` 服务，按照步骤开启即可，有个密码设置，记住这个密码，后面有用。

然后就可以编码了：

```
package main

import (
  "log"
  "net/smtp"

  "github.com/jordan-wright/email"
)

func main() {
```

```
  e := email.NewEmail()
  e.From = "dj <xxx@126.com>"
  e.To = []string{"935653229@qq.com"}
  e.Subject = "Awesome web"
  e.Text = []byte("Text Body is, of course, supported!")
  err := e.Send("smtp.126.com:25", smtp.PlainAuth("", "xxx@126.com", "yyy", "smtp.126.com"))
  if err != nil {
    log.Fatal(err)
  }
}
```

这里为了我的信息安全，我把真实信息都隐藏了。代码中 `xxx` 替换成你的邮箱账号， `yyy` 替换成上面设置的密码。

代码步骤比较简单清晰：

- 先调用 `NewEmail` 创建一封邮件；
- 设置 `From` 发送方， `To` 接收者， `Subject` 邮件主题（标题）， `Text` 设置邮件内容；
- 然后调用 `Send` 发送，参数1是 SMTP 服务器的地址，参数2为验证信息。

运行程序将会向我的 QQ 邮箱发送一封邮件：

Awesome web ☆
发件人：dj 126.com>
时　间：2020年2月16日(星期天) 下午3:25
收件人：孤雨 <935653229@qq.com>
这是一封垃圾箱中的邮件。请勿轻信中奖、汇款等虚假信息，勿轻易拨打陌生电话。 举报垃圾邮件 移回收件箱

Text Body is, of course, supported!

有的邮箱会把这种邮件放在垃圾箱中，例如 QQ😭。如果收件箱找不到，记得到垃圾箱瞅瞅。

抄送

平常我们发邮件的时候可能会抄送给一些人，还有一些人要**秘密抄送**😄，即 CC（Carbon Copy）和 BCC （Blind Carbon Copy）。 `email` 我们也可以设置这两个参数：

```
package main

import (
```

```
  "log"
  "net/smtp"

  "github.com/jordan-wright/email"
)

func main() {
  e := email.NewEmail()
  e.From = "dj <xxx@126.com>"
  e.To = []string{"935653229@qq.com"}
  e.Cc = []string{"test1@126.com", "test2@126.com"}
  e.Bcc = []string{"secret@126.com"}
  e.Subject = "Awesome web"
  e.Text = []byte("Text Body is, of course, supported!")
  err := e.Send("smtp.126.com:25", smtp.PlainAuth("", "xxx@126.com", "yyy", "smtp.126.com"))
  if err != nil {
    log.Fatal(err)
  }
}
```

还是一样的，抄送的邮箱自己替换 `test1/test2/secret` 用自己的。

运行程序将会向我的 QQ 邮件发送一封邮件，同时抄送一封到我另一个 126 邮箱：

Awesome web
发件人: dj <126.com>
时 间: 2020年2月16日(星期天) 下午3:37
收件人: 孤雨 <935653229@qq.com>
抄 送: lidajun9426 <lidajun9426@126.com>
这是一封垃圾箱中的邮件。请勿轻信中奖、汇款等虚假信息，勿轻易拨打陌生电话。 举报垃圾邮件 移回收件箱

Text Body is, of course, supported!

HTML 格式

发送纯文本，邮件不太美观。 `email` 支持发送 HTML 格式的内容。与发送纯文本类似，直接设置对象的 `HTML` 字段：

```
package main

import (
  "log"
  "net/smtp"

  "github.com/jordan-wright/email"
)

func main() {
  e := email.NewEmail()
  e.From = "dj <xxx@126.com>"
  e.To = []string{"935653229@qq.com"}
  e.Cc = []string{"xxx@126.com"}
  e.Subject = "Go 每日一库"
  e.HTML = []byte(`
  <ul>
<li><a "https://darjun.github.io/2020/01/10/godailylib/flag/">Go 每日一库之 flag
</a></li>
<li><a "https://darjun.github.io/2020/01/10/godailylib/go-flags/">Go 每日一库之
 go-flags</a></li>
<li><a "https://darjun.github.io/2020/01/14/godailylib/go-homedir/">Go 每日一库
之 go-homedir</a></li>
<li><a "https://darjun.github.io/2020/01/15/godailylib/go-ini/">Go 每日一库之 go
-ini</a></li>
<li><a "https://darjun.github.io/2020/01/17/godailylib/cobra/">Go 每日一库之 cob
ra</a></li>
<li><a "https://darjun.github.io/2020/01/18/godailylib/viper/">Go 每日一库之 vip
er</a></li>
<li><a "https://darjun.github.io/2020/01/19/godailylib/fsnotify/">Go 每日一库之
 fsnotify</a></li>
<li><a "https://darjun.github.io/2020/01/20/godailylib/cast/">Go 每日一库之 cast
</a></li>
</ul>
  `)
  err := e.Send("smtp.126.com:25", smtp.PlainAuth("", "xxx@126.com", "yyy", "smt
p.126.com"))
  if err != nil {
    log.Fatal("failed to send email:", err)
```

```
    }
}
```

发送结果：

Go 每日一库 ☆
发件人：dj <leedarjun@126.com>
时　间：2020年2月16日(星期天) 下午4:01
收件人：孤雨 <935653229@qq.com>
抄　送：leedarjun <leedarjun@126.com>

- Go 每日一库之 flag
- Go 每日一库之 go-flags
- Go 每日一库之 go-homedir
- Go 每日一库之 go-ini
- Go 每日一库之 cobra
- Go 每日一库之 viper
- Go 每日一库之 fsnotify
- Go 每日一库之 cast

注意，126 的 SMTP 服务器检测比较严格，加上 HTML 之后，很容易被识别为垃圾邮件不让发送，这时 CC 自己就 OK 了。

附件

添加附件也很容易，直接调用 `AttachFile` 即可：

```go
package main

import (
    "log"
    "net/smtp"

    "github.com/jordan-wright/email"
)

func main() {
    e := email.NewEmail()
    e.From = "dj <xxx@126.com>"
    e.To = []string{"935653229@qq.com"}
    e.Subject = "Go 每日一库"
    e.Text = []byte("请看附件")
```

```
  e.AttachFile("test.txt")
  err := e.Send("smtp.126.com:25", smtp.PlainAuth("", "xxx@126.com", "yyy", "smt
p.126.com"))
  if err != nil {
    log.Fatal("failed to send email:", err)
  }
}
```

收到的邮件：

Go 每日一库 ☆

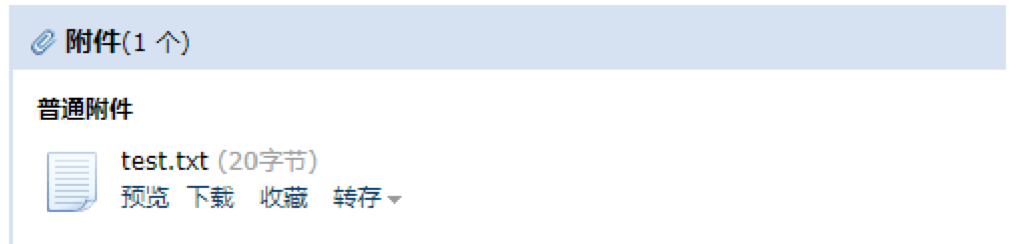
发件人：**dj** <leedarjun@126.com>

时　间：2020年2月16日(星期天) 下午4:15

收件人：孤雨 <935653229@qq.com>

附　件：1 个 (test.txt)

请看附件



连接池

实际上每次调用 `Send` 时都会和 SMTP 服务器建立一次连接，如果发送邮件很多很频繁的话可能会有性能问题。 `email` 提供了连接池，可以复用网络连接：

```go
package main

import (
  "fmt"
  "log"
  "net/smtp"
  "os"
  "sync"
  "time"
```

```go
	"github.com/jordan-wright/email"
)

func main() {
	ch := make(chan *email.Email, 10)
	p, err := email.NewPool(
		"smtp.126.com:25",
		4,
		smtp.PlainAuth("", "leedarjun@126.com", "358942617ldj", "smtp.126.com"),
	)

	if err != nil {
		log.Fatal("failed to create pool:", err)
	}

	var wg sync.WaitGroup
	wg.Add(4)
	for i := 0; i < 4; i++ {
		go func() {
			defer wg.Done()
			for e := range ch {
				err := p.Send(e, 10*time.Second)
				if err != nil {
					fmt.Fprintf(os.Stderr, "email:%v sent error:%v\n", e, err)
				}
			}
		}()
	}

	for i := 0; i < 10; i++ {
		e := email.NewEmail()
		e.From = "dj <leedarjun@126.com>"
		e.To = []string{"935653229@qq.com"}
		e.Subject = "Awesome web"
		e.Text = []byte(fmt.Sprintf("Awesome Web %d", i+1))
		ch <- e
	}

	close(ch)
	wg.Wait()
}
```

上面程序中，我们创建 4 goroutine 共用一个连接池发送邮件，发送 10 封邮件后程序退出。为了等邮件都发送完成或失败，程序才退出，我们使用了 `sync.WaitGroup` 。

邮箱被轰炸了：

今天 (13 封)

dj	Awesome web - Awesome Web 7
dj	Awesome web - Awesome Web 10
dj	Awesome web - Awesome Web 2
dj	Awesome web - Awesome Web 5
dj	Awesome web - Awesome Web 8
dj	Awesome web - Awesome Web 4
dj	Awesome web - Awesome Web 1
dj	Awesome web - Awesome Web 3
dj	Awesome web - Awesome Web 6
dj	Awesome web - Awesome Web 9

由于使用了 goroutine，邮件顺序不能保证。

总结

本文介绍了如何使用 Go 程序发送邮件，程序代码都已经放在 GitHub 上 https://github.com/darjun/go-daily-lib/tree/master/email。所有代码都通过测试，大家请放心食用~

大家如果发现好玩、好用的 Go 语言库，欢迎到 Go 每日一库 GitHub 上提交 issue😄

参考

1. email GitHub：https://github.com/jordan-wright/email
2. Go 每日一库 GitHub：https://github.com/darjun/go-daily-lib

carbon

简介

一线开发人员每天都要使用日期和时间相关的功能，各种定时器，活动时间处理等。标准库time使用起来不太灵活，特别是日期时间的创建和运算。carbon库是一个时间扩展库，基于 PHP 的carbon库编写。提供易于使用的接口。本文就来介绍一下这个库。

快速使用

第三方库需要先安装：

```
$ go get github.com/uniplaces/carbon
```

后使用：

```
package main

import (
  "fmt"
  "time"

  "github.com/uniplaces/carbon"
)

func main() {
  fmt.Printf("Right now is %s\n", carbon.Now().DateTimeString())

  today, _ := carbon.NowInLocation("Japan")
  fmt.Printf("Right now in Japan is %s\n", today)

  fmt.Printf("Tomorrow is %s\n", carbon.Now().AddDay())
  fmt.Printf("Last week is %s\n", carbon.Now().SubWeek())

  nextOlympics, _ := carbon.CreateFromDate(2016, time.August, 5, "Europe/London"
)
  nextOlympics = nextOlympics.AddYears(4)
  fmt.Printf("Next olympics are in %d\n", nextOlympics.Year())

  if carbon.Now().IsWeekend() {
    fmt.Printf("Happy time!")
```

```
  }
}
```

`carbon` 库的使用很便捷，首先它完全兼容标准库的 `time.Time` 类型，实际上该库的日期时间类型 `Carbon` 直接将 `time.Time` 内嵌到结构中，所以 `time.Time` 的方法可直接调用：

```go
// src/github.com/uniplaces/carbon/carbon.go
type Carbon struct {
  time.Time
  weekStartsAt time.Weekday
  weekEndsAt   time.Weekday
  weekendDays  []time.Weekday
  stringFormat string
  Translator   *Translator
}
```

其次，简化了创建操作。标准库 `time` 创建一个 `Time` 对象，如果不是本地或 UTC 时区，需要自己先调用 `LoadLocation` 加载对应时区。然后将该时区对象传给 `time.Date` 方法创建。`carbon` 可以直接传时区名字。

`carbon` 还提供了很多方法做日期运算，如例子中的 `AddDay` ，`SubWeek` 等，都是见名知义的。

时区

在介绍其它内容之前，我们先说一说这个时区的问题。以下引用维基百科的描述：

> 时区是地球上的区域使用同一个时间定义。以前，人们通过观察太阳的位置（时角）决定时间，这就使得不同经度的地方的时间有所不同（地方时）。1863年，首次使用时区的概念。时区通过设立一个区域的标准时间部分地解决了这个问题。
> 世界各国位于地球不同位置上，因此不同国家，特别是东西跨度大的国家日出、日落时间必定有所偏差。这些偏差就是所谓的时差。

例如，日本东京位于东九区，北京位于东八区，所以日本比中国快一个小时，日本14:00的时候中国13:00。

在 Linux 中，时区一般存放在类似 `/usr/share/zoneinfo` 这样的目录。这个目录中有很多文件，每个时区一个文件。时区文件是二进制文件，可以执行 `info tzfile` 查看具体格式。

时区名称的一般格式为 `city` ，或 `country/city` ，或 `continent/city` 。即要么就是一个城市名，要么是国家名+/+城市名，要么是洲名+/+城市名。例如上海时区

为 `Asia/Shanghai` ，香港时区为 `Asia/Hong_Kong` 。也有一些特殊的，如 UTC，Local等。

Go 语言为了可移植性，在安装包中提供了时区文件，在安装目录下（我的为 `C:\Go` ）的 `lib/time/zoneinfo.zip` 文件，大家可以执行解压看看。

使用 Go 标准库 `time` 创建某个时区的时间，需要先加载时区：

```
package main

import (
  "fmt"
  "log"
  "time"
)

func main() {
  loc, err := time.LoadLocation("Japan")
  if err != nil {
    log.Fatal("failed to load location: ", err)
  }

  d := time.Date(2020, time.July, 24, 20, 0, 0, 0, loc)
  fmt.Printf("The opening ceremony of next olympics will start at %s in Japan\n"
, d)
}
```

使用 `carbon` 就不用这么麻烦：

```
package main

import (
  "fmt"
  "log"
  "time"

  "github.com/uniplaces/carbon"
)

func main() {
  c, err := carbon.Create(2020, time.July, 24, 20, 0, 0, 0, "Japan")
  if err != nil {
    log.Fatal(err)
  }
```

```
    fmt.Printf("The opening ceremony of next olympics will start at %s in Japan\n"
, c)
}
```

时间运算

使用标准库 `time` 的时间运算需要先定义一个 `time.Duration` 对象， `time` 库预定义的只有纳秒到小时的精度：

```
const (
    Nanosecond  Duration = 1
    Microsecond = 1000 * Nanosecond
    Millisecond = 1000 * Microsecond
    Second      = 1000 * Millisecond
    Minute      = 60 * Second
    Hour        = 60 * Minute
)
```

其它的时长就需要自己使用 `time.ParseDuration` 构造了，而且 `time.ParseDuration` 不能构造其它精度的时间。
如果想要增加/减少年月日，就需要使用 `time.Time` 的 `AddDate` 方法：

```
package main

import (
    "fmt"
    "log"
    "time"
)

func main() {
    now := time.Now()

    fmt.Println("now is:", now)

    fmt.Println("one second later is:", now.Add(time.Second))
    fmt.Println("one minute later is:", now.Add(time.Minute))
    fmt.Println("one hour later is:", now.Add(time.Hour))

    d, err := time.ParseDuration("3m20s")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("3 minutes and 20 seconds later is:", now.Add(d))
```

```
  d, err = time.ParseDuration("2h30m")
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println("2 hours and 30 minutes later is:", now.Add(d))

  fmt.Println("3 days and 2 hours later is:", now.AddDate(0, 0, 3).Add(time.Hour
*2))
}
```

需要注意的是，时间操作都是返回一个新的对象，原对象不会修改。 `carbon` 库也是如此。**Go 的标准库也建议我们不要使用 `time.Time` 的指针**。
当然 `carbon` 库也能使用上面的方法，它还提供了多种粒度的方法：

```
package main

import (
  "fmt"

  "github.com/uniplaces/carbon"
)

func main() {
  now := carbon.Now()

  fmt.Println("now is:", now)

  fmt.Println("one second later is:", now.AddSecond())
  fmt.Println("one minute later is:", now.AddMinute())
  fmt.Println("one hour later is:", now.AddHour())
  fmt.Println("3 minutes and 20 seconds later is:", now.AddMinutes(3).AddSeconds
(20))
  fmt.Println("2 hours and 30 minutes later is:", now.AddHours(2).AddMinutes(30
))
  fmt.Println("3 days and 2 hours later is:", now.AddDays(3).AddHours(2))
}
```

`carbon` 还提供了：

- 增加**季度**的方法： `AddQuarters/AddQuarter` ，复数形式介绍一个表示倍数的参数，单数形式倍数为1；
- 增加**世纪**的方法： `AddCenturies/AddCentury` ；

- 增加**工作日**的方法：`AddWeekdays/AddWeekday`，这个方法会跳过非工作日；
- 增加**周**的方法：`AddWeeks/AddWeek`。

其实给上面方法传入负数就表示减少，另外 `carbon` 也提供了对应的 `Sub*` 方法。

时间比较

标准库 `time` 可以使用 `time.Time` 对象的 `Before/After/Equal` 判断是否在另一个时间对象前，后，或相等。`carbon` 库也可以使用上面的方法比较时间。除此之外，它还提供了多组方法，每个方法提供一个简短名，一个详细名：

- `Eq/EqualTo`：是否相等；
- `Ne/NotEqualTo`：是否不等；
- `Gt/GreaterThan`：是否在之后；
- `Lt/LessThan`：是否在之前；
- `Lte/LessThanOrEqual`：是否相同或在之前；
- `Between`：是否在两个时间之间。

另外 `carbon` 提供了：

- 判断当前时间是周几的方法：`IsMonday/IsTuesday/.../IsSunday`；
- 是否是工作日，周末，闰年，过去时间还是未来时间：`IsWeekday/IsWeekend/IsLeapYear/IsPast/IsFuture`。

```
package main

import (
  "fmt"

  "github.com/uniplaces/carbon"
)

func main() {
  t1, _ := carbon.CreateFromDate(2010, 10, 1, "Asia/Shanghai")
  t2, _ := carbon.CreateFromDate(2011, 10, 20, "Asia/Shanghai")

  fmt.Printf("t1 equal to t2: %t\n", t1.Eq(t2))
  fmt.Printf("t1 not equal to t2: %t\n", t1.Ne(t2))

  fmt.Printf("t1 greater than t2: %t\n", t1.Gt(t2))
  fmt.Printf("t1 less than t2: %t\n", t1.Lt(t2))
```

```
    t3, _ := carbon.CreateFromDate(2011, 1, 20, "Asia/Shanghai")
    fmt.Printf("t3 between t1 and t2: %t\n", t3.Between(t1, t2, true))

    now := carbon.Now()
    fmt.Printf("Weekday? %t\n", now.IsWeekday())
    fmt.Printf("Weekend? %t\n", now.IsWeekend())
    fmt.Printf("LeapYear? %t\n", now.IsLeapYear())
    fmt.Printf("Past? %t\n", now.IsPast())
    fmt.Printf("Future? %t\n", now.IsFuture())
}
```

我们还可以使用 `carbon` 计算两个日期之间相差多少秒、分、小时、天：

```
package main

import (
    "fmt"

    "github.com/uniplaces/carbon"
)

func main() {
    vancouver, _ := carbon.Today("Asia/Shanghai")
    london, _ := carbon.Today("Asia/Hong_Kong")
    fmt.Println(vancouver.DiffInSeconds(london, true)) // 0

    ottawa, _ := carbon.CreateFromDate(2000, 1, 1, "America/Toronto")
    vancouver, _ = carbon.CreateFromDate(2000, 1, 1, "America/Vancouver")
    fmt.Println(ottawa.DiffInHours(vancouver, true)) // 3

    fmt.Println(ottawa.DiffInHours(vancouver, false)) // 3
    fmt.Println(vancouver.DiffInHours(ottawa, false)) // -3

    t, _ := carbon.CreateFromDate(2012, 1, 31, "UTC")
    fmt.Println(t.DiffInDays(t.AddMonth(), true))  // 31
    fmt.Println(t.DiffInDays(t.SubMonth(), false)) // -31

    t, _ = carbon.CreateFromDate(2012, 4, 30, "UTC")
    fmt.Println(t.DiffInDays(t.AddMonth(), true)) // 30
    fmt.Println(t.DiffInDays(t.AddWeek(), true))  // 7

    t, _ = carbon.CreateFromTime(10, 1, 1, 0, "UTC")
    fmt.Println(t.DiffInMinutes(t.AddSeconds(59), true))  // 0
    fmt.Println(t.DiffInMinutes(t.AddSeconds(60), true))  // 1
```

```
    fmt.Println(t.DiffInMinutes(t.AddSeconds(119), true)) // 1
    fmt.Println(t.DiffInMinutes(t.AddSeconds(120), true)) // 2
}
```

格式化

我们知道 `time.Time` 提供了一个 `Format` 方法，相比于其他编程语言使用格式化符来描述格式（需要记忆 `%d/%m/%h` 等的含义），Go 提供了一个一种更简单、直观的方式——使用 layout。即我们传入一个日期字符串，表示我们想要格式化成什么样子。Go 会用当前的时间替换字符串中的对应部分：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    fmt.Println(t.Format("2006-01-02 15:04:05"))
}
```

上面我们只需要传入一个 `2006-01-02 15:04:05` 表示我们想要的格式为 `yyyy-mm-dd hh:mm:ss`，省去了我们需要记忆的麻烦。

为了使用方便，Go 内置了一些标准的时间格式：

```
// src/time/format.go
const (
    ANSIC       = "Mon Jan _2 15:04:05 2006"
    UnixDate    = "Mon Jan _2 15:04:05 MST 2006"
    RubyDate    = "Mon Jan 02 15:04:05 -0700 2006"
    RFC822      = "02 Jan 06 15:04 MST"
    RFC822Z     = "02 Jan 06 15:04 -0700" // RFC822 with numeric zone
    RFC850      = "Monday, 02-Jan-06 15:04:05 MST"
    RFC1123     = "Mon, 02 Jan 2006 15:04:05 MST"
    RFC1123Z    = "Mon, 02 Jan 2006 15:04:05 -0700" // RFC1123 with numeric zone
    RFC3339     = "2006-01-02T15:04:05Z07:00"
    RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
    Kitchen     = "3:04PM"
    // Handy time stamps.
    Stamp      = "Jan _2 15:04:05"
    StampMilli = "Jan _2 15:04:05.000"
```

```
	StampMicro = "Jan _2 15:04:05.000000"
	StampNano  = "Jan _2 15:04:05.000000000"
)
```

除了上面这些格式， `carbon` 还提供了其他一些格式：

```
// src/github.com/uniplaces/carbon
const (
	DefaultFormat       = "2006-01-02 15:04:05"
	DateFormat          = "2006-01-02"
	FormattedDateFormat = "Jan 2, 2006"
	TimeFormat          = "15:04:05"
	HourMinuteFormat    = "15:04"
	HourFormat          = "15"
	DayDateTimeFormat   = "Mon, Aug 2, 2006 3:04 PM"
	CookieFormat        = "Monday, 02-Jan-2006 15:04:05 MST"
	RFC822Format        = "Mon, 02 Jan 06 15:04:05 -0700"
	RFC1036Format       = "Mon, 02 Jan 06 15:04:05 -0700"
	RFC2822Format       = "Mon, 02 Jan 2006 15:04:05 -0700"
	RSSFormat           = "Mon, 02 Jan 2006 15:04:05 -0700"
)
```

注意一点， `time` 库默认使用 `2006-01-02 15:04:05.999999999 -0700 MST` 格式，有点复杂了， `carbon` 库默认使用更简洁的 `2006-01-02 15:04:05` 。

高级特性

修饰器

所谓修饰器（modifier）就是对一些特定的时间操作，获取开始和结束时间。如当天、月、季度、年、十年、世纪、周的开始和结束时间，还能获得上一个周二、下一个周一、下一个工作日的时间等等：

```
package main

import (
	"fmt"
	"time"

	"github.com/uniplaces/carbon"
)

func main() {
```

```
    t := carbon.Now()
    fmt.Printf("Start of day:%s\n", t.StartOfDay())
    fmt.Printf("End of day:%s\n", t.EndOfDay())
    fmt.Printf("Start of month:%s\n", t.StartOfMonth())
    fmt.Printf("End of month:%s\n", t.EndOfMonth())
    fmt.Printf("Start of year:%s\n", t.StartOfYear())
    fmt.Printf("End of year:%s\n", t.EndOfYear())
    fmt.Printf("Start of decade:%s\n", t.StartOfDecade())
    fmt.Printf("End of decade:%s\n", t.EndOfDecade())
    fmt.Printf("Start of century:%s\n", t.StartOfCentury())
    fmt.Printf("End of century:%s\n", t.EndOfCentury())
    fmt.Printf("Start of week:%s\n", t.StartOfWeek())
    fmt.Printf("End of week:%s\n", t.EndOfWeek())
    fmt.Printf("Next:%s\n", t.Next(time.Wednesday))
    fmt.Printf("Previous:%s\n", t.Previous(time.Wednesday))
}
```

自定义工作日和周末

有些地区每周的开始、周末和我们的不一样。例如，在美国周日是新的一周开始。没关系，`carbon` 可以自定义每周的开始和周末：

```
package main

import (
    "fmt"
    "log"
    "time"

    "github.com/uniplaces/carbon"
)

func main() {
    t, err := carbon.Create(2020, 02, 11, 0, 0, 0, 0, "Asia/Shanghai")
    if err != nil {
        log.Fatal(err)
    }

    t.SetWeekStartsAt(time.Sunday)
    t.SetWeekEndsAt(time.Saturday)
    t.SetWeekendDays([]time.Weekday{time.Monday, time.Tuesday, time.Wednesday})

    fmt.Printf("Today is %s, weekend? %t\n", t.Weekday(), t.IsWeekend())
}
```

总结

`carbon` 提供了很多的实用方法，另外 `time` 的方法它也能使用，使得它的功能非常强大。时间其实是一个非常复杂的问题，考虑到时区、闰秒、各地的夏令时等，自己处理起来简直是火葬场。幸好有这些库(┬＿┬)

参考

1. carbon GitHub 仓库：https://github.com/uniplaces/carbon

godotenv

简介

twelve-factor应用提倡将配置存储在环境变量中。任何从开发环境切换到生产环境时需要修改的东西都从代码抽取到环境变量里。
但是在实际开发中，如果同一台机器运行多个项目，设置环境变量容易冲突，不实用。
godotenv库从 `.env` 文件中读取配置，
然后存储到程序的环境变量中。在代码中可以使用读取非常方便。 `godotenv` 源于一个 Ruby 的开源项目dotenv。

快速使用

第三方库需要先安装：

```
$ go get github.com/joho/godotenv
```

后使用：

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/joho/godotenv"
)

func main() {
  err := godotenv.Load()
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", os.Getenv("name"))
  fmt.Println("age: ", os.Getenv("age"))
}
```

然后在可执行程序相同目录下，添加一个 `.env` 文件：

```
name = dj
age = 18
```

运行程序，输出：

```
name:  dj
age:  18
```

可见，使用非常方便。默认情况下， `godotenv` 读取项目根目录下的 `.env` 文件，文件中使用 `key = value` 的格式，每行一个键值对。
调用 `godotenv.Load()` 即可加载，可直接调用 `os.Getenv("key")` 读取。 `os.Getenv` 是用来读取环境变量的：

```
package main

import (
  "fmt"
  "os"
)

func main() {
  fmt.Println(os.Getenv("GOPATH"))
}
```

高级特性

自动加载

如果你有程序员的优良传统——懒，你可能连 `Load` 方法都不想自己调用。没关系， `godotenv` 给你懒的权力！

导入 `github.com/joho/godotenv/autoload` ，配置会自动读取：

```
package main

import (
  "fmt"
  "os"

  _ "github.com/joho/godotenv/autoload"
)
```

```go
func main() {
  fmt.Println("name: ", os.Getenv("name"))
  fmt.Println("age: ", os.Getenv("age"))
}
```

注意，由于代码中没有显式用到 `godotenv` 库，需要使用空导入，即导入时包名前添加一个 `_` 。

看 `autoload` 包的源码，其实就是库帮你调用了 `Load` 方法：

```go
// src/github.com/joho/godotenv/autoload/autoload.go
package autoload

/*
    You can just read the .env file on import just by doing

        import _ "github.com/joho/godotenv/autoload"

    And bob's your mother's brother
*/

import "github.com/joho/godotenv"

func init() {
  godotenv.Load()
}
```

仔细看注释，程序员的恶趣味😂！

加载自定义文件

默认情况下，加载的是项目根目录下的 `.env` 文件。当然我们可以加载任意名称的文件，文件也不必以 `.env` 为后缀：

```go
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/joho/godotenv"
)
```

```go
func main() {
  err := godotenv.Load("common", "dev.env")
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", os.Getenv("name"))
  fmt.Println("version: ", os.Getenv("version"))
  fmt.Println("database: ", os.Getenv("database"))
}
```

`common` 文件内容：

```
name = awesome web
version = 0.0.1
```

`dev.env`：

```
database = sqlite
```

`production.env`：

```
database = mysql
```

自己运行看看结果吧！

注意：`Load` 接收多个文件名作为参数，如果不传入文件名，默认读取 `.env` 文件的内容。
如果多个文件中存在同一个键，那么先出现的优先，后出现的不生效。所以，上面输出的 `database` 是什么？

注释

`.env` 文件中可以添加注释，注释以 `#` 开始，直到该行结束。

```
# app name
name = awesome web
# current version
version = 0.0.1
```

YAML

`.env` 文件还可以使用 YAML 格式：

```
name: awesome web
version: 0.0.1
```

```
package main

import (
  "fmt"
  "os"

  _ "github.com/joho/godotenv/autoload"
)

func main() {
  fmt.Println("name: ", os.Getenv("name"))
  fmt.Println("version: ", os.Getenv("version"))
}
```

不存入环境变量

`godotenv` 允许不将 `.env` 文件内容存入环境变量，使用 `godotenv.Read()` 返回一个 `map[string]string` ，可直接使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/joho/godotenv"
)

func main() {
  var myEnv map[string]string
  myEnv, err := godotenv.Read()
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", myEnv["name"])
  fmt.Println("version: ", myEnv["version"])
}
```

直接操作 `map` ，简单直接！

数据源

除了读取文件，还可以从 `io.Reader` ，从 `string` 中读取配置：

```
package main

import (
  "fmt"
  "log"

  "github.com/joho/godotenv"
)

func main() {
  content := `
name: awesome web
version: 0.0.1
  `
  myEnv, err := godotenv.Unmarshal(content)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", myEnv["name"])
  fmt.Println("version: ", myEnv["version"])
}
```

只要实现了 `io.Reader` 接口，就能作为数据源。可以从文件（ `os.File` ），网络（ `net.Conn` ）， `bytes.Buffer` 等多种来源读取：

```
package main

import (
    "bytes"
    "fmt"
    "log"
    "os"

    "github.com/joho/godotenv"
)

func main() {
  file, _ := os.OpenFile(".env", os.O_RDONLY, 0666)
```

```go
  myEnv, err := godotenv.Parse(file)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", myEnv["name"])
  fmt.Println("version: ", myEnv["version"])

  buf := &bytes.Buffer{}
  buf.WriteString("name: awesome web @buffer")
  buf.Write([]byte{'\n'})
  buf.WriteString("version: 0.0.1")
  myEnv, err = godotenv.Parse(buf)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("name: ", myEnv["name"])
  fmt.Println("version: ", myEnv["version"])
}
```

注意，从字符串读取和从 `io.Reader` 读取使用的方法是不同的。前者为 `Unmarshal` ，后者是 `Parse` 。

生成 `.env` 文件

可以通过程序生成一个 `.env` 文件的内容，可以直接写入到文件中：

```go
package main

import (
  "bytes"
  "log"

  "github.com/joho/godotenv"
)

func main() {
  buf := &bytes.Buffer{}
  buf.WriteString("name = awesome web")
  buf.WriteByte('\n')
  buf.WriteString("version = 0.0.1")

  env, err := godotenv.Parse(buf)
  if err != nil {
```

```
    log.Fatal(err)
  }

  err = godotenv.Write(env, "./.env")
  if err != nil {
    log.Fatal(err)
  }
}
```

查看生成的 `.env` 文件：

```
name="awesome web"
version="0.0.1"
```

还可以返回一个字符串，怎么揉捏随你：

```
package main

import (
  "bytes"
  "fmt"
  "log"

  "github.com/joho/godotenv"
)

func main() {
  buf := &bytes.Buffer{}
  buf.WriteString("name = awesome web")
  buf.WriteByte('\n')
  buf.WriteString("version = 0.0.1")

  env, err := godotenv.Parse(buf)
  if err != nil {
    log.Fatal(err)
  }

  content, err := godotenv.Marshal(env)
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(content)
}
```

命令行模式

`godotenv` 还提供了一个命令行的模式：

```
$ godotenv -f ./.env command args
```

前面在 `go get` 安装 `godotenv` 时， `godotenv` 就已经安装在 `$GOPATH/bin` 目录下了，我习惯把 `$GOPATH/bin` 加入系统 `PATH` ，所以 `godotenv` 命令可以直接使用。

命令行模式就是读取指定文件（如果不通过 `-f` 指定，则使用 `.env` 文件），设置环境变量，然后运行后面的程序。

我们简单写一个程序验证一下：

```
package main

import (
  "fmt"
  "os"
)

func main() {
  fmt.Println(os.Getenv("name"))
  fmt.Println(os.Getenv("version"))
}
```

使用 `godotenv` 运行一下：

```
$ godotenv -f ./.env go run main.go
```

输出：

```
awesome web
0.0.1
```

多个环境

实践中，一般会根据 `APP_ENV` 环境变量的值加载不同的文件：

```
package main
```

```go
import (
	"fmt"
	"log"
	"os"

	"github.com/joho/godotenv"
)

func main() {
	env := os.Getenv("GODAILYLIB_ENV")
	if env == "" {
		env = "development"
	}

	err := godotenv.Load(".env." + env)
	if err != nil {
		log.Fatal(err)
	}

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

	fmt.Println("name: ", os.Getenv("name"))
	fmt.Println("version: ", os.Getenv("version"))
	fmt.Println("database: ", os.Getenv("database"))
}
```

我们先读取环境变量 `GODAILYLIB_ENV` ，然后读取对应的 `.env.` + env，最后读取默认的 `.env` 文件。

前面也提到过，先读取到的优先。我们可以在默认的 `.env` 文件中配置基础信息和一些默认的值，
如果在开发/测试/生产环境需要修改，那么在对应的 `.env.development/.env.test/.env.production` 文件中再配置一次即可。

`.env` 文件内容：

```
name = awesome web
version = 0.0.1
database = file
```

`.env.development` ：

```
database = sqlite3
```

`.env.production`：

```
database = mysql
```

运行程序：

```
# 默认是开发环境
$ go run main.go
name:  awesome web
version:  0.0.1
database:  sqlite3

# 设置为生成环境
$ GODAILYLIB_ENV=production go run main.go
name:  awesome web
version:  0.0.1
database:  mysql
```

一点源码

`godotenv` 读取文件内容，为什么可以使用 `os.Getenv` 访问：

```
// src/github.com/joho/godotenv/godotenv.go
func loadFile(filename string, overload bool) error {
    envMap, err := readFile(filename)
    if err != nil {
        return err
    }

    currentEnv := map[string]bool{}
    rawEnv := os.Environ()
    for _, rawEnvLine := range rawEnv {
        key := strings.Split(rawEnvLine, "=")[0]
        currentEnv[key] = true
    }

    for key, value := range envMap {
        if !currentEnv[key] || overload {
            os.Setenv(key, value)
        }
```

```
    }

    return nil
}
```

因为 `godotenv` 调用 `os.Setenv` 将键值对设置到环境变量中了。

总结

本文介绍了 `godotenv` 库的基础和高级用法。`godotenv` 的源码也比较好读，有时间，有兴趣的童鞋建议一看~

参考

1. godotenv GitHub 仓库：https://github.com/joho/godotenv

logrus

简介

前一篇文章介绍了 Go 标准库中的日志库 `log` 。最后我们也提到， `log` 库只提供了三组接口，功能过于简单了。
今天，我们来介绍一个日志库中的“明星库”—— `logrus` 。本文编写之时（2020.02.07），logrus 在 GitHub 上 star 数已达到 13.8k。
`logrus` 完全兼容标准的 `log` 库，还支持文本、JSON 两种日志输出格式。很多知名的开源项目都使用了这个库，如大名鼎鼎的 docker。

快速使用

第三方库需要先安装：

```
$ go get github.com/sirupsen/logrus
```

后使用：

```
package main

import (
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.SetLevel(logrus.TraceLevel)

  logrus.Trace("trace msg")
  logrus.Debug("debug msg")
  logrus.Info("info msg")
  logrus.Warn("warn msg")
  logrus.Error("error msg")
  logrus.Fatal("fatal msg")
  logrus.Panic("panic msg")
}
```

`logrus` 的使用非常简单，与标准库 `log` 类似。 `logrus` 支持更多的日志级别：

- `Panic` ：记录日志，然后 `panic` 。

- `Fatal` ：致命错误，出现错误时程序无法正常运转。输出日志后，程序退出；
- `Error` ：错误日志，需要查看原因；
- `Warn` ：警告信息，提醒程序员注意；
- `Info` ：关键操作，核心流程的日志；
- `Debug` ：一般程序中输出的调试信息；
- `Trace` ：很细粒度的信息，一般用不到；

日志级别从上向下依次增加， `Trace` 最大， `Panic` 最小。 `logrus` 有一个日志级别，高于这个级别的日志不会输出。
默认的级别为 `InfoLevel` 。所以为了能看到 `Trace` 和 `Debug` 日志，我们在 `main` 函数第一行设置日志级别为 `TraceLevel` 。

运行程序，输出：

```
$ go run main.go
time="2020-02-07T21:22:42+08:00" level=trace msg="trace msg"
time="2020-02-07T21:22:42+08:00" level=debug msg="debug msg"
time="2020-02-07T21:22:42+08:00" level=info msg="info msg"
time="2020-02-07T21:22:42+08:00" level=info msg="warn msg"
time="2020-02-07T21:22:42+08:00" level=error msg="error msg"
time="2020-02-07T21:22:42+08:00" level=fatal msg="fatal msg"
exit status 1
```

由于 `logrus.Fatal` 会导致程序退出，下面的 `logrus.Panic` 不会执行到。

另外，我们观察到输出中有三个关键信息， `time` 、 `level` 和 `msg` ：

- `time` ：输出日志的时间；
- `level` ：日志级别；
- `msg` ：日志信息。

定制

输出文件名

调用 `logrus.SetReportCaller(true)` 设置在输出日志中添加文件名和方法信息：

```go
package main

import (
  "github.com/sirupsen/logrus"
```

```
)

func main() {
  logrus.SetReportCaller(true)

  logrus.Info("info msg")
}
```

输出多了两个字段 `file` 为调用 `logrus` 相关方法的文件名， `method` 为方法名：

```
$ go run main.go
time="2020-02-07T21:46:03+08:00" level=info msg="info msg" func=main.main file=
"D:/code/golang/src/github.com/darjun/go-daily-lib/logrus/caller/main.go:10"
```

添加字段

有时候需要在输出中添加一些字段，可以通过调用 `logrus.WithField` 和 `logrus.WithFields` 实现。 `logrus.WithFields` 接受一个 `logrus.Fields` 类型的参数，其底层实际上为 `map[string]interface{}` ：

```
// github.com/sirupsen/logrus/logrus.go
type Fields map[string]interface{}
```

下面程序在输出中添加两个字段 `name` 和 `age` ：

```
package main

import (
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.WithFields(logrus.Fields{
    "name": "dj",
    "age": 18,
  }).Info("info msg")
}
```

如果在一个函数中的所有日志都需要添加某些字段，可以使用 `WithFields` 的返回值。例如在 Web 请求的处理器中，日志都要加上 `user_id` 和 `ip` 字段：

```
package main

import (
  "github.com/sirupsen/logrus"
)

func main() {
  requestLogger := logrus.WithFields(logrus.Fields{
    "user_id": 10010,
    "ip":       "192.168.32.15",
  })

  requestLogger.Info("info msg")
  requestLogger.Error("error msg")
}
```

实际上， `WithFields` 返回一个 `logrus.Entry` 类型的值，它将 `logrus.Logger` 和设置的 `logrus.Fields` 保存下来。
调用 `Entry` 相关方法输出日志时，保存下来的 `logrus.Fields` 也会随之输出。

重定向输出

默认情况下，日志输出到 `io.Stderr` 。可以调用 `logrus.SetOutput` 传入一个 `io.Writer` 参数。后续调用相关方法日志将写到 `io.Writer` 中。
现在，我们就能像上篇文章介绍log时一样，可以搞点事情了。传入一个 `io.MultiWriter` ，
同时将日志写到 `bytes.Buffer` 、标准输出和文件中：

```
package main

import (
  "bytes"
  "io"
  "log"
  "os"

  "github.com/sirupsen/logrus"
)

func main() {
  writer1 := &bytes.Buffer{}
  writer2 := os.Stdout
  writer3, err := os.OpenFile("log.txt", os.O_WRONLY|os.O_CREATE, 0755)
  if err != nil {
```

```
    log.Fatalf("create file log.txt failed: %v", err)
  }

  logrus.SetOutput(io.MultiWriter(writer1, writer2, writer3))
  logrus.Info("info msg")
}
```

自定义

实际上，考虑到易用性，库一般会使用默认值创建一个对象，包最外层的方法一般都是操作这个默认对象。

我们之前好几篇文章都提到过这点：

- Go 每日一库之 flag：`flag`标准库中的`CommandLine`对象；
- Go 每日一库之 log：`log`标准库中的`std`对象。

这个技巧应用在很多库的开发中，`logrus`也是如此：

```
// github.com/sirupsen/logrus/exported.go
var (
  std = New()
)

func StandardLogger() *Logger {
  return std
}

func SetOutput(out io.Writer) {
  std.SetOutput(out)
}

func SetFormatter(formatter Formatter) {
  std.SetFormatter(formatter)
}

func SetReportCaller(include bool) {
  std.SetReportCaller(include)
}

func SetLevel(level Level) {
  std.SetLevel(level)
}
```

首先，使用默认配置定义一个 `Logger` 对象 `std` ， `SetOutput/SetFormatter/SetReportCaller/SetLevel` 这些方法都是调用 `std` 对象的对应方法！

我们当然也可以创建自己的 `Logger` 对象，使用方式与直接调用 `logrus` 的方法类似：

```
package main

import "github.com/sirupsen/logrus"

func main() {
  log := logrus.New()

  log.SetLevel(logrus.InfoLevel)
  log.SetFormatter(&logrus.JSONFormatter{})

  log.Info("info msg")
}
```

日志格式

`logrus` 支持两种日志格式，文本和 JSON，默认为文本格式。可以通过 `logrus.SetFormatter` 设置日志格式：

```
package main

import (
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.SetLevel(logrus.TraceLevel)
  logrus.SetFormatter(&logrus.JSONFormatter{})

  logrus.Trace("trace msg")
  logrus.Debug("debug msg")
  logrus.Info("info msg")
  logrus.Warn("warn msg")
  logrus.Error("error msg")
  logrus.Fatal("fatal msg")
  logrus.Panic("panic msg")
}
```

程序输出 JSON 格式的日志：

```
$ go run main.go
{"level":"trace","msg":"trace msg","time":"2020-02-07T21:40:04+08:00"}
{"level":"debug","msg":"debug msg","time":"2020-02-07T21:40:04+08:00"}
{"level":"info","msg":"info msg","time":"2020-02-07T21:40:04+08:00"}
{"level":"info","msg":"warn msg","time":"2020-02-07T21:40:04+08:00"}
{"level":"error","msg":"error msg","time":"2020-02-07T21:40:04+08:00"}
{"level":"fatal","msg":"fatal msg","time":"2020-02-07T21:40:04+08:00"}
exit status 1
```

第三方格式

除了内置的 `TextFormatter` 和 `JSONFormatter` ，还有不少第三方格式支持。我们这里介绍一个nested-logrus-formatter。

先安装：

```
$ go get github.com/antonfisher/nested-logrus-formatter
```

后使用：

```
package main

import (
  nested "github.com/antonfisher/nested-logrus-formatter"
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.SetFormatter(&nested.Formatter{
    HideKeys:    true,
    FieldsOrder: []string{"component", "category"},
  })

  logrus.Info("info msg")
}
```

程序输出：

```
Feb  8 15:22:59.077 [INFO] info msg
```

`nested` 格式提供了多个字段用来定制行为：

```
// github.com/antonfisher/nested-logrus-formatter/formatter.go
type Formatter struct {
  FieldsOrder     []string
  TimestampFormat string
  HideKeys        bool
  NoColors        bool
  NoFieldsColors  bool
  ShowFullLevel   bool
  TrimMessages    bool
}
```

- 默认，`logrus`输出日志中字段是`key=value`这样的形式。使用`nested`格式，我们可以通过设置`HideKeys`为`true`隐藏键，只输出值；
- 默认，`logrus`是按键的字母序输出字段，可以设置`FieldsOrder`定义输出字段顺序；
- 通过设置`TimestampFormat`设置日期格式。

```
package main

import (
  "time"

  nested "github.com/antonfisher/nested-logrus-formatter"
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.SetFormatter(&nested.Formatter{
    // HideKeys:        true,
    TimestampFormat: time.RFC3339,
    FieldsOrder:     []string{"name", "age"},
  })

  logrus.WithFields(logrus.Fields{
    "name": "dj",
    "age":  18,
  }).Info("info msg")
}
```

如果不隐藏键，程序输出：

```
$ 2020-02-08T15:40:07+08:00 [INFO] [name:dj] [age:18] info msg
```

隐藏键，程序输出：

```
$ 2020-02-08T15:41:58+08:00 [INFO] [dj] [18] info msg
```

注意到，我们将时间格式设置成 `time.RFC3339` ，即 `2006-01-02T15:04:05Z07:00` 这种形式。

通过实现接口 `logrus.Formatter` 可以实现自己的格式。

```
// github.com/sirupsen/logrus/formatter.go
type Formatter interface {
  Format(*Entry) ([]byte, error)
}
```

设置钩子

还可以为 `logrus` 设置钩子，每条日志输出前都会执行钩子的特定方法。所以，我们可以添加输出字段、根据级别将日志输出到不同的目的地。 `logrus` 也内置了一个 `syslog` 的钩子，将日志输出到 `syslog` 中。这里我们实现一个钩子，在输出的日志中增加一个 `app=awesome-web` 字段。

钩子需要实现 `logrus.Hook` 接口：

```
// github.com/sirupsen/logrus/hooks.go
type Hook interface {
  Levels() []Level
  Fire(*Entry) error
}
```

`Levels()` 方法返回感兴趣的日志级别，输出其他日志时不会触发钩子。 `Fire` 是日志输出前调用的钩子方法。

```
package main

import (
  "github.com/sirupsen/logrus"
)

type AppHook struct {
  AppName string
}

func (h *AppHook) Levels() []logrus.Level {
```

```
  return logrus.AllLevels
}

func (h *AppHook) Fire(entry *logrus.Entry) error {
  entry.Data["app"] = h.AppName
  return nil
}

func main() {
  h := &AppHook{AppName: "awesome-web"}
  logrus.AddHook(h)

  logrus.Info("info msg")
}
```

只需要在 `Fire` 方法实现中，为 `entry.Data` 添加字段就会输出到日志中。

程序输出：

```
$ time="2020-02-08T15:51:52+08:00" level=info msg="info msg" app=awesome-web
```

`logrus` 的第三方 Hook 很多，我们可以使用一些 Hook 将日志发送到 redis/mongodb 等存储中：

- mgorus：将日志发送到 mongodb；
- logrus-redis-hook：将日志发送到 redis；
- logrus-amqp：将日志发送到 ActiveMQ。

这里我们演示一个 redis，感兴趣自行验证其他的。先安装 `logrus-redis-hook`：

```
$ go get github.com/rogierlommers/logrus-redis-hook
```

然后编写程序：

```
package main

import (
  "io/ioutil"

  logredis "github.com/rogierlommers/logrus-redis-hook"
  "github.com/sirupsen/logrus"
)
```

```
func init() {
  hookConfig := logredis.HookConfig{
    Host:     "localhost",
    Key:      "mykey",
    Format:   "v0",
    App:      "aweosome",
    Hostname: "localhost",
    TTL:      3600,
  }

  hook, err := logredis.NewHook(hookConfig)
  if err == nil {
    logrus.AddHook(hook)
  } else {
    logrus.Errorf("logredis error: %q", err)
  }
}

func main() {
  logrus.Info("just some info logging...")

  logrus.WithFields(logrus.Fields{
    "animal": "walrus",
    "foo":    "bar",
    "this":   "that",
  }).Info("additional fields are being logged as well")

  logrus.SetOutput(ioutil.Discard)
  logrus.Info("This will only be sent to Redis")
}
```

为了程序能正常工作，我们还需要安装 `redis` 。

windows 上直接使用choco安装 redis：

```
PS C:\Users\Administrator> choco install redis-64
Chocolatey v0.10.15
Installing the following packages:
redis-64
By installing you accept licenses for the packages.
Progress: Downloading redis-64 3.0.503... 100%

redis-64 v3.0.503 [Approved]
redis-64 package files install completed. Performing other installation steps.
 ShimGen has successfully created a shim for redis-benchmark.exe
```

```
 ShimGen has successfully created a shim for redis-check-aof.exe
 ShimGen has successfully created a shim for redis-check-dump.exe
 ShimGen has successfully created a shim for redis-cli.exe
 ShimGen has successfully created a shim for redis-server.exe
 The install of redis-64 was successful.
  Software install location not explicitly set, could be in package or
  default install location if installer.

Chocolatey installed 1/1 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

直接输入 `redis-server` ，启动服务器：

运行程序后，我们使用 `redis-cli` 查看：

```
PS C:\Users\Administrator> redis-cli
127.0.0.1:6379> keys *
1) "mykey"
127.0.0.1:6379> type mykey
list
127.0.0.1:6379> lrange mykey 0 -1
1) "{\"@fields\":{\"application\":\"aweosome\",\"level\":\"info\"},\"@message\":\"just some info logging...\",\"@source_
host\":\"localhost\",\"@timestamp\":\"2020-02-08T08:15:19.508072Z\"}"
2) "{\"@fields\":{\"animal\":\"walrus\",\"application\":\"aweosome\",\"foo\":\"bar\",\"level\":\"info\",\"this\":\"that\
"},\"@message\":\"additional fields are being logged as well\",\"@source_host\":\"localhost\",\"@timestamp\":\"2020-02-0
8T08:15:19.5449722Z\"}"
3) "{\"@fields\":{\"application\":\"aweosome\",\"level\":\"info\"},\"@message\":\"This will only be sent to Redis\",\"@s
ource_host\":\"localhost\",\"@timestamp\":\"2020-02-08T08:15:19.5459705Z\"}"
127.0.0.1:6379>
```

我们看到 `mykey` 是一个 `list` ，每过来一条日志，就在 `list` 后新增一项。

总结

本文介绍了 `logrus` 的基本用法。 `logrus` 的可扩展性非常棒，可以引入第三方格式和 Hook 增强功能。在社区也比较受欢迎。

参考

1. logrus GitHub 仓库
2. Hooks

log

简介

在日常开发中，日志是必不可少的功能。虽然有时可以用 `fmt` 库输出一些信息，但是灵活性不够。Go 标准库提供了一个日志库 `log` 。本文介绍 `log` 库的使用。

快速使用

`log` 是 Go 标准库提供的，不需要另外安装。可直接使用：

```
package main

import (
  "log"
)

type User struct {
  Name string
  Age  int
}

func main() {
  u := User{
    Name: "dj",
    Age:  18,
  }

  log.Printf("%s login, age:%d", u.Name, u.Age)
  log.Panicf("Oh, system error when %s login", u.Name)
  log.Fatalf("Danger! hacker %s login", u.Name)
}
```

`log` 默认输出到标准错误（ `stderr` ），每条日志前会自动加上日期和时间。如果日志不是以换行符结尾的，那么 `log` 会自动加上换行符。即每条日志会在新行中输出。

`log` 提供了三组函数：

- `Print/Printf/Println` ：正常输出日志；
- `Panic/Panicf/Panicln` ：输出日志后，以拼装好的字符串为参数调用 `panic` ；

- `Fatal/Fatalf/Fatalln`：输出日志后，调用`os.Exit(1)`退出程序。

命名比较容易辨别，带 `f` 后缀的有格式化功能，带 `ln` 后缀的会在日志后增加一个换行符。

注意，上面的程序中由于调用 `log.Panicf` 会 `panic` ，所以 `log.Fatalf` 并不会调用。

定制

前缀

调用 `log.SetPrefix` 为每条日志文本前增加一个前缀。例如，在上面的程序中设置 `Login:` 前缀：

```
package main

import (
  "log"
)

type User struct {
  Name string
  Age  int
}

func main() {
  u := User{
    Name: "dj",
    Age:  18,
  }

  log.SetPrefix("Login: ")
  log.Printf("%s login, age:%d", u.Name, u.Age)
}
```

调用 `log.Prefix` 可以获取当前设置的前缀。

选项

设置选项可在每条输出的文本前增加一些额外信息，如日期时间、文件名等。

`log` 库提供了 6 个选项：

```go
// src/log/log.go
const (
  Ldate         = 1 << iota
  Ltime
  Lmicroseconds
  Llongfile
  Lshortfile
  LUTC
)
```

- `Ldate`：输出当地时区的日期，如`2020/02/07`；
- `Ltime`：输出当地时区的时间，如`11:45:45`；
- `Lmicroseconds`：输出的时间精确到微秒，设置了该选项就不用设置`Ltime`了。如`11:45:45.123123`；
- `Llongfile`：输出长文件名+行号，含包名，如`github.com/darjun/go-daily-lib/log/flag/main.go:50`；
- `Lshortfile`：输出短文件名+行号，不含包名，如`main.go:50`；
- `LUTC`：如果设置了`Ldate`或`Ltime`，将输出 UTC 时间，而非当地时区。

调用`log.SetFlag`设置选项，可以一次设置多个：

```go
package main

import (
  "log"
)

type User struct {
  Name string
  Age  int
}

func main() {
  u := User{
    Name: "dj",
    Age:  18,
  }

  log.SetFlags(log.Lshortfile | log.Ldate | log.Lmicroseconds)

  log.Printf("%s login, age:%d", u.Name, u.Age)
}
```

调用 `log.Flags()` 可以获取当前设置的选项。

运行代码，输出：

```
2020/02/07 11:56:59.061615 main.go:20: dj login, age:18
```

注意，调用 `log.SetFlag` 之后，原有的选项会被覆盖掉！

`log` 库还定义了一个 `Lstdflag` ，为 `Ldate | Ltime` ，这就是我们默认的选项。

```
// src/log/log.go
const (
  LstdFlags = Ldate | Ltime
)
```

这就是为什么默认情况下，每条日志前会自动加上日期和时间。

自定义

实际上， `log` 库为我们定义了一个默认的 `Logger` ，名为 `std` ，意为标准日志。我们直接调用的 `log` 库的方法，其内部是调用 `std` 的对应方法：

```
// src/log/log.go
var std = New(os.Stderr, "", LstdFlags)

func Printf(format string, v ...interface{}) {
  std.Output(2, fmt.Sprintf(format, v...))
}

func Fatalf(format string, v ...interface{}) {
  std.Output(2, fmt.Sprintf(format, v...))
  os.Exit(1)
}

func Panicf(format string, v ...interface{}) {
  s := fmt.Sprintf(format, v...)
  std.Output(2, s)
  panic(s)
}
```

当然，我们也可以定义自己的 `Logger` ：

```go
package main

import (
  "bytes"
  "fmt"
  "log"
)

type User struct {
  Name string
  Age  int
}

func main() {
  u := User{
    Name: "dj",
    Age:  18,
  }

  buf := &bytes.Buffer{}
  logger := log.New(buf, "", log.Lshortfile|log.LstdFlags)

  logger.Printf("%s login, age:%d", u.Name, u.Age)

  fmt.Print(buf.String())
}
```

`log.New` 接受三个参数：

- `io.Writer`：日志都会写到这个`Writer`中；
- `prefix`：前缀，也可以后面调用`logger.SetPrefix`设置；
- `flag`：选项，也可以后面调用`logger.SetFlag`设置。

上面代码将日志输出到一个 `bytes.Buffer` ，然后将这个 `buf` 打印到标准输出。

运行代码：

```
$ go run main.go
2020/02/07 13:48:54 main.go:23: dj login, age:18
```

注意到，第一个参数为 `io.Writer` ，我们可以使用 `io.MultiWriter` 实现多目的地输出。下面我们将日志同时输出到标准输出、 `bytes.Buffer` 和文件中：

```go
package main

import (
  "bytes"
  "io"
  "log"
  "os"
)

type User struct {
  Name string
  Age  int
}

func main() {
  u := User{
    Name: "dj",
    Age:  18,
  }

  writer1 := &bytes.Buffer{}
  writer2 := os.Stdout
  writer3, err := os.OpenFile("log.txt", os.O_WRONLY|os.O_CREATE, 0755)
  if err != nil {
    log.Fatalf("create file log.txt failed: %v", err)
  }

  logger := log.New(io.MultiWriter(writer1, writer2, writer3), "", log.Lshortfile|log.LstdFlags)
  logger.Printf("%s login, age:%d", u.Name, u.Age)
}
```

如果你愿意，还可以发送到到网络。

实现

`log` 库的核心是 `Output` 方法，我们简单看一下：

```go
// src/log/log.go
func (l *Logger) Output(calldepth int, s string) error {
  now := time.Now() // get this early.
  var file string
  var line int
  l.mu.Lock()
```

```
	defer l.mu.Unlock()
	if l.flag&(Lshortfile|Llongfile) != 0 {
		// Release lock while getting caller info - it's expensive.
		l.mu.Unlock()
		var ok bool
		_, file, line, ok = runtime.Caller(calldepth)
		if !ok {
			file = "???"
			line = 0
		}
		l.mu.Lock()
	}
	l.buf = l.buf[:0]
	l.formatHeader(&l.buf, now, file, line)
	l.buf = append(l.buf, s...)
	if len(s) == 0 || s[len(s)-1] != '\n' {
		l.buf = append(l.buf, '\n')
	}
	_, err := l.out.Write(l.buf)
	return err
}
```

如果设置了 `Lshortfile` 或 `Llongfile` ， `Ouput` 方法中会调用 `runtime.Caller` 获取文件名和行号。 `runtime.Caller` 的参数 `calldepth` 表示获取调用栈向上多少层的信息，当前层为 0。

一般的调用路径是：

- 程序中使用 `log.Printf` 之类的函数；
- 在 `log.Printf` 内调用 `std.Output` 。

我们在 `Output` 方法中需要获取调用 `log.Printf` 的文件和行号。 `calldepth` 传入 0 表示 `Output` 方法内调用 `runtime.Caller` 的那一行信息，传入 1 表示 `log.Printf` 内调用 `std.Output` 那一行的信息，传入 2 表示程序中调用 `log.Printf` 的那一行信息。显然这里要用 2。

然后调用 `formatHeader` 处理前缀和选项。

最后将生成的字节流写入到 `Writer` 中。

这里有两个优化技巧：

- 由于 `runtime.Caller` 调用比较耗时，先释放锁，避免等待时间过长；
- 为了避免频繁的内存分配， `logger` 中保存了一个类型为 `[]byte` 的 `buf` ，可重复使用。前缀和日志内容先写到这个 `buf` 中，然后统一写入 `Writer` ，减少 io 操

作。

总结

`log` 实现了一个小巧的日志库，可供简单使用。本文介绍了它的基本使用，简单地分析了一下源码。
如果 `log` 库的功能不能满足需求，我们可以在它之上做二次封装。看煎鱼大佬的这篇文章。

除此之外，社区也涌现了很多优秀的、功能丰富的日志库，可以选用。

参考

1. log官方文档

viper

简介

上一篇文章介绍 cobra 的时候提到了 viper，今天我们就来介绍一下这个库。
viper 是一个配置解决方案，拥有丰富的特性：

- 支持 JSON/TOML/YAML/HCL/envfile/Java properties 等多种格式的配置文件；
- 可以设置监听配置文件的修改，修改时自动加载新的配置；
- 从环境变量、命令行选项和 `io.Reader` 中读取配置；
- 从远程配置系统中读取和监听修改，如 etcd/Consul；
- 代码逻辑中显示设置键值。

快速使用

安装：

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

使用：

```
package main

import (
  "fmt"
  "log"

  "github.com/spf13/viper"
)

func main() {
  viper.SetConfigName("config")
  viper.SetConfigType("toml")
  viper.AddConfigPath(".")
  viper.SetDefault("redis.port", 6381)
  err := viper.ReadInConfig()
  if err != nil {
    log.Fatal("read config failed: %v", err)
  }
```

```
    fmt.Println(viper.Get("app_name"))
    fmt.Println(viper.Get("log_level"))

    fmt.Println("mysql ip: ", viper.Get("mysql.ip"))
    fmt.Println("mysql port: ", viper.Get("mysql.port"))
    fmt.Println("mysql user: ", viper.Get("mysql.user"))
    fmt.Println("mysql password: ", viper.Get("mysql.password"))
    fmt.Println("mysql database: ", viper.Get("mysql.database"))

    fmt.Println("redis ip: ", viper.Get("redis.ip"))
    fmt.Println("redis port: ", viper.Get("redis.port"))
}
```

我们使用之前Go 每日一库之 go-ini一文中使用的配置，不过改为 toml 格式。
toml 的语法很简单，快速入门请看learn X in Y minutes。

```
app_name = "awesome web"

# possible values: DEBUG, INFO, WARNING, ERROR, FATAL
log_level = "DEBUG"

[mysql]
ip = "127.0.0.1"
port = 3306
user = "dj"
password = 123456
database = "awesome"

[redis]
ip = "127.0.0.1"
port = 7381
```

viper 的使用非常简单，它需要很少的设置。设置文件名（ `SetConfigName` ）、配置类型（ `SetConfigType` ）和搜索路径（ `AddConfigPath` ），然后调用 `ReadInConfig` 。
viper会自动根据类型来读取配置。使用时调用 `viper.Get` 方法获取键值。

编译、运行程序：

```
awesome web
DEBUG
mysql ip:  127.0.0.1
mysql port:  3306
mysql user:  dj
```

```
mysql password:  123456
mysql database:  awesome
redis ip:  127.0.0.1
redis port:  7381
```

有几点需要注意：

- 设置文件名时不要带后缀；
- 搜索路径可以设置多个，viper 会根据设置顺序依次查找；
- viper 获取值时使用 `section.key` 的形式，即传入嵌套的键名；
- 默认值可以调用 `viper.SetDefault` 设置。

读取键

viper 提供了多种形式的读取方法。在上面的例子中，我们看到了 `Get` 方法的用法。`Get` 方法返回一个 `interface{}` 的值，使用有所不便。

`GetType` 系列方法可以返回指定类型的值。其中，Type 可以为 `Bool/Float64/Int/String/Time/Duration/IntSlice/StringSlice` 。但是请注意，**如果指定的键不存在或类型不正确，`GetType` 方法返回对应类型的零值**。

如果要判断某个键是否存在，使用 `IsSet` 方法。另外，`GetStringMap` 和 `GetStringMapString` 直接以 map 返回某个键下面所有的键值对，前者返回 `map[string]interface{}` ，后者返回 `map[string]string` 。`AllSettings` 以 `map[string]interface{}` 返回所有设置。

```go
// 省略包名和 import 部分

func main() {
  viper.SetConfigName("config")
  viper.SetConfigType("toml")
  viper.AddConfigPath(".")
  err := viper.ReadInConfig()
  if err != nil {
    log.Fatal("read config failed: %v", err)
  }

  fmt.Println("protocols: ", viper.GetStringSlice("server.protocols"))
  fmt.Println("ports: ", viper.GetIntSlice("server.ports"))
  fmt.Println("timeout: ", viper.GetDuration("server.timeout"))

  fmt.Println("mysql ip: ", viper.GetString("mysql.ip"))
  fmt.Println("mysql port: ", viper.GetInt("mysql.port"))
```

```
	if viper.IsSet("redis.port") {
		fmt.Println("redis.port is set")
	} else {
		fmt.Println("redis.port is not set")
	}

	fmt.Println("mysql settings: ", viper.GetStringMap("mysql"))
	fmt.Println("redis settings: ", viper.GetStringMap("redis"))
	fmt.Println("all settings: ", viper.AllSettings())
}
```

我们在配置文件 config.toml 中添加 `protocols` 和 `ports` 配置：

```
[server]
protocols = ["http", "https", "port"]
ports = [10000, 10001, 10002]
timeout = 3s
```

编译、运行程序，输出：

```
protocols:  [http https port]
ports:  [10000 10001 10002]
timeout:  3s
mysql ip:  127.0.0.1
mysql port:  3306
redis.port is set
mysql settings:  map[database:awesome ip:127.0.0.1 password:123456 port:3306 user:dj]
redis settings:  map[ip:127.0.0.1 port:7381]
all settings:  map[app_name:awesome web log_level:DEBUG mysql:map[database:awesome ip:127.0.0.1 password:123456 port:3306 user:dj] redis:map[ip:127.0.0.1 port:7381] server:map[ports:[10000 10001 10002] protocols:[http https port]]]
```

如果将配置中的 `redis.port` 注释掉，将输出 `redis.port is not set` 。

上面的示例中还演示了如何使用 `time.Duration` 类型，只要是 `time.ParseDuration` 接受的格式都可以，例如 `3s` 、 `2min` 、 `1min30s` 等。

设置键值

viper 支持在多个地方设置，使用下面的顺序依次读取：

- 调用 `Set` 显示设置的；
- 命令行选项；
- 环境变量；
- 配置文件；
- 默认值。

`viper.Set`

如果某个键通过 `viper.Set` 设置了值，那么这个值的优先级最高。

```
viper.Set("redis.port", 5381)
```

如果将上面这行代码放到程序中，运行程序，输出的 `redis.port` 将是 5381。

命令行选项

如果一个键没有通过 `viper.Set` 显示设置值，那么获取时将尝试从命令行选项中读取。
如果有，优先使用。viper 使用 pflag 库来解析选项。
我们首先在 `init` 方法中定义选项，并且调用 `viper.BindPFlags` 绑定选项到配置中：

```
func init() {
  pflag.Int("redis.port", 8381, "Redis port to connect")

  // 绑定命令行
  viper.BindPFlags(pflag.CommandLine)
}
```

然后，在 `main` 方法开头处调用 `pflag.Parse` 解析选项。

编译、运行程序：

```
$ ./main.exe --redis.port 9381
awesome web
DEBUG
mysql ip:  127.0.0.1
mysql port:  3306
mysql user:  dj
mysql password:  123456
mysql database:  awesome
redis ip:  127.0.0.1
redis port:  9381
```

如何不传入选项：

```
$ ./main.exe
awesome web
DEBUG
mysql ip:  127.0.0.1
mysql port:  3306
mysql user:  dj
mysql password:  123456
mysql database:  awesome
redis ip:  127.0.0.1
redis port:  7381
```

注意，这里并不会使用选项 `redis.port` 的默认值。

但是，如果通过下面的方法都无法获得键值，那么返回选项默认值（如果有）。试试注释掉配置文件中 `redis.port` 看看效果。

环境变量

如果前面都没有获取到键值，将尝试从环境变量中读取。我们既可以一个个绑定，也可以自动全部绑定。

在 `init` 方法中调用 `AutomaticEnv` 方法绑定全部环境变量：

```
func init() {
  // 绑定环境变量
  viper.AutomaticEnv()
}
```

为了验证是否绑定成功，我们在 `main` 方法中将环境变量 GOPATH 打印出来：

```
func main() {
  // 省略部分代码

  fmt.Println("GOPATH: ", viper.Get("GOPATH"))
}
```

通过 系统 -> 高级设置 -> 新建 创建一个名为 `redis.port` 的环境变量，值为 10381。运行程序，输出的 `redis.port` 值为 10381，并且输出中有 GOPATH 信息。

也可以单独绑定环境变量：

```
func init() {
  // 绑定环境变量
  viper.BindEnv("redis.port")
  viper.BindEnv("go.path", "GOPATH")
}

func main() {
  // 省略部分代码
  fmt.Println("go path: ", viper.Get("go.path"))
}
```

调用 `BindEnv` 方法，如果只传入一个参数，则这个参数既表示键名，又表示环境变量名。如果传入两个参数，则第一个参数表示键名，第二个参数表示环境变量名。

还可以通过 `viper.SetEnvPrefix` 方法设置环境变量前缀，这样一来，通过 `AutomaticEnv` 和一个参数的 `BindEnv` 绑定的环境变量，在使用 `Get` 的时候，viper 会自动加上这个前缀再从环境变量中查找。

如果对应的环境变量不存在，viper 会自动将键名全部转为大写再查找一次。所以，使用键名 `gopath` 也能读取环境变量 `GOPATH` 的值。

配置文件

如果经过前面的途径都没能找到该键，viper 接下来会尝试从配置文件中查找。为了避免环境变量的影响，需要删除 `redis.port` 这个环境变量。

看快速使用中的示例。

默认值

在上面的快速使用一节，我们已经看到了如何设置默认值，这里就不赘述了。

读取配置

从 `io.Reader` 中读取

viper 支持从 `io.Reader` 中读取配置。这种形式很灵活，来源可以是文件，也可以是程序中生成的字符串，甚至可以从网络连接中读取的字节流。

```
package main

import (
  "bytes"
```

```
    "fmt"
    "log"

    "github.com/spf13/viper"
)

func main() {
    viper.SetConfigType("toml")
    tomlConfig := []byte(`
app_name = "awesome web"

# possible values: DEBUG, INFO, WARNING, ERROR, FATAL
log_level = "DEBUG"

[mysql]
ip = "127.0.0.1"
port = 3306
user = "dj"
password = 123456
database = "awesome"

[redis]
ip = "127.0.0.1"
port = 7381
`)
    err := viper.ReadConfig(bytes.NewBuffer(tomlConfig))
    if err != nil {
        log.Fatal("read config failed: %v", err)
    }

    fmt.Println("redis port: ", viper.GetInt("redis.port"))
}
```

`Unmarshal`

viper 支持将配置 `Unmarshal` 到一个结构体中，为结构体中的对应字段赋值。

```
package main

import (
    "fmt"
    "log"

    "github.com/spf13/viper"
)
```

```go
type Config struct {
	AppName  string
	LogLevel string

	MySQL    MySQLConfig
	Redis    RedisConfig
}

type MySQLConfig struct {
	IP       string
	Port     int
	User     string
	Password string
	Database string
}

type RedisConfig struct {
	IP   string
	Port int
}

func main() {
	viper.SetConfigName("config")
	viper.SetConfigType("toml")
	viper.AddConfigPath(".")
	err := viper.ReadInConfig()
	if err != nil {
		log.Fatal("read config failed: %v", err)
	}

	var c Config
	viper.Unmarshal(&c)

	fmt.Println(c.MySQL)
}
```

编译，运行程序，输出：

```
{127.0.0.1 3306 dj 123456 awesome}
```

保存配置

有时候，我们想要将程序中生成的配置，或者所做的修改保存下来。viper 提供了接口！

- `WriteConfig` ：将当前的 viper 配置写到预定义路径，如果没有预定义路径，返回错误。将会覆盖当前配置；
- `SafeWriteConfig` ：与上面功能一样，但是如果配置文件存在，则不覆盖；
- `WriteConfigAs` ：保存配置到指定路径，如果文件存在，则覆盖；
- `SafeWriteConfig` ：与上面功能一样，但是入股配置文件存在，则不覆盖。

下面我们通过程序生成一个 `config.toml` 配置：

```
package main

import (
  "log"

  "github.com/spf13/viper"
)

func main() {
  viper.SetConfigName("config")
  viper.SetConfigType("toml")
  viper.AddConfigPath(".")

  viper.Set("app_name", "awesome web")
  viper.Set("log_level", "DEBUG")
  viper.Set("mysql.ip", "127.0.0.1")
  viper.Set("mysql.port", 3306)
  viper.Set("mysql.user", "root")
  viper.Set("mysql.password", "123456")
  viper.Set("mysql.database", "awesome")

  viper.Set("redis.ip", "127.0.0.1")
  viper.Set("redis.port", 6381)

  err := viper.SafeWriteConfig()
  if err != nil {
    log.Fatal("write config failed: ", err)
  }
}
```

编译、运行程序，生成的文件如下：

```
app_name = "awesome web"
log_level = "DEBUG"
```

```
[mysql]
  database = "awesome"
  ip = "127.0.0.1"
  password = "123456"
  port = 3306
  user = "root"

[redis]
  ip = "127.0.0.1"
  port = 6381
```

监听文件修改

viper 可以监听文件修改，热加载配置。因此不需要重启服务器，就能让配置生效。

```
package main

import (
  "fmt"
  "log"
  "time"

  "github.com/spf13/viper"
)

func main() {
  viper.SetConfigName("config")
  viper.SetConfigType("toml")
  viper.AddConfigPath(".")
  err := viper.ReadInConfig()
  if err != nil {
    log.Fatal("read config failed: %v", err)
  }

  viper.WatchConfig()

  fmt.Println("redis port before sleep: ", viper.Get("redis.port"))
  time.Sleep(time.Second * 10)
  fmt.Println("redis port after sleep: ", viper.Get("redis.port"))
}
```

只需要调用 `viper.WatchConfig` ，viper 会自动监听配置修改。如果有修改，重新加载的配置。

上面程序中，我们先打印 `redis.port` 的值，然后 `Sleep` 10s。在这期间修改配置中 `redis.port` 的值，`Sleep` 结束后再次打印。
发现打印出修改后的值：

```
redis port before sleep: 7381
redis port after sleep: 73810
```

另外，还可以为配置修改增加一个回调：

```
viper.OnConfigChange(func(e fsnotify.Event) {
  fmt.Printf("Config file:%s Op:%s\n", e.Name, e.Op)
})
```

这样文件修改时会执行这个回调。

viper 使用fsnotify这个库来实现监听文件修改的功能。

完整示例代码见 GitHub。

参考

1. viper GitHub 仓库

fsnotify

简介

上一篇文章Go 每日一库之 viper中，我们介绍了 viper 可以监听文件修改进而自动重新加载。其内部使用的就是 `fsnotify` 这个库，它是跨平台的。今天我们就来介绍一下它。

快速使用

先安装：

```
$ go get github.com/fsnotify/fsnotify
```

后使用：

```
package main

import (
  "log"

  "github.com/fsnotify/fsnotify"
)

func main() {
  watcher, err := fsnotify.NewWatcher()
  if err != nil {
    log.Fatal("NewWatcher failed: ", err)
  }
  defer watcher.Close()

  done := make(chan bool)
  go func() {
    defer close(done)

    for {
      select {
      case event, ok := <-watcher.Events:
        if !ok {
          return
        }
        log.Printf("%s %s\n", event.Name, event.Op)
```

```
      case err, ok := <-watcher.Errors:
        if !ok {
          return
        }
        log.Println("error:", err)
      }
    }
  }()

  err = watcher.Add("./")
  if err != nil {
    log.Fatal("Add failed:", err)
  }
  <-done
}
```

`fsnotify` 的使用比较简单：

- 先调用 `NewWatcher` 创建一个监听器；
- 然后调用监听器的 `Add` 增加监听的文件或目录；
- 如果目录或文件有事件产生，监听器中的通道 `Events` 可以取出事件。如果出现错误，监听器中的通道 `Errors` 可以取出错误信息。

上面示例中，我们在另一个 goroutine 中循环读取发生的事件及错误，然后输出它们。

编译、运行程序。在当前目录创建一个 `新建文本文档.txt` ，然后重命名为 `file1.txt` 文件，输入内容 `some test text` ，然后删除它。观察控制台输出：

```
2020/01/20 08:41:17 新建文本文档.txt CREATE
2020/01/20 08:41:25 新建文本文档.txt RENAME
2020/01/20 08:41:25 file1.txt CREATE
2020/01/20 08:42:28 file1.txt REMOVE
```

其实，重命名时会产生两个事件，一个是原文件的 `RENAME` 事件，一个是新文件的 `CREATE` 事件。

注意， `fsnotify` 使用了操作系统接口，监听器中保存了系统资源的句柄，所以使用后需要关闭。

事件

上面示例中的事件是 `fsnotify.Event` 类型：

```go
// fsnotify/fsnotify.go
type Event struct {
  Name string
  Op   Op
}
```

事件只有两个字段，`Name` 表示发生变化的文件或目录名，`Op` 表示具体的变化。`Op` 有 5 种取值：

```go
// fsnotify/fsnotify.go
type Op uint32

const (
  Create Op = 1 << iota
  Write
  Remove
  Rename
  Chmod
)
```

在快速使用中，我们已经演示了前 4 种事件。`Chmod` 事件在文件或目录的属性发生变化时触发，在 Linux 系统中可以通过 `chmod` 命令改变文件或目录属性。

事件中的 `Op` 是按照位来存储的，可以存储多个，可以通过 `&` 操作判断对应事件是不是发生了。

```go
if event.Op & fsnotify.Write != 0 {
  fmt.Println("Op has Write")
}
```

我们在代码中不需要这样判断，因为 `Op` 的 `String()` 方法已经帮我们处理了这种情况了：

```go
// fsnotify.go
func (op Op) String() string {
  // Use a buffer for efficient string concatenation
  var buffer bytes.Buffer

  if op&Create == Create {
    buffer.WriteString("|CREATE")
  }
  if op&Remove == Remove {
    buffer.WriteString("|REMOVE")
```

```
    }
    if op&Write == Write {
        buffer.WriteString("|WRITE")
    }
    if op&Rename == Rename {
        buffer.WriteString("|RENAME")
    }
    if op&Chmod == Chmod {
        buffer.WriteString("|CHMOD")
    }
    if buffer.Len() == 0 {
        return ""
    }
    return buffer.String()[1:] // Strip leading pipe
}
```

应用

`fsnotify` 的应用非常广泛，在 godoc 上，我们可以看到哪些库导入了 `fsnotify` 。只需要在 `fsnotify` 文档的 URL 后加上 `?imports` 即可：

https://godoc.org/github.com/fsnotify/fsnotify?importers。有兴趣打开看看，要 fq。

上一篇文章中，我们介绍了调用 `viper.WatchConfig` 就可以监听配置修改，自动重新加载。下面我们就来看看 `WatchConfig` 是怎么实现的：

```
// viper/viper.go
func WatchConfig() { v.WatchConfig() }

func (v *Viper) WatchConfig() {
    initWG := sync.WaitGroup{}
    initWG.Add(1)
    go func() {
        watcher, err := fsnotify.NewWatcher()
        if err != nil {
            log.Fatal(err)
        }
        defer watcher.Close()
        // we have to watch the entire directory to pick up renames/atomic saves in a cross-platform way
        filename, err := v.getConfigFile()
        if err != nil {
            log.Printf("error: %v\n", err)
            initWG.Done()
            return
```

```
    }

    configFile := filepath.Clean(filename)
    configDir, _ := filepath.Split(configFile)
    realConfigFile, _ := filepath.EvalSymlinks(filename)

    eventsWG := sync.WaitGroup{}
    eventsWG.Add(1)
    go func() {
      for {
        select {
        case event, ok := <-watcher.Events:
          if !ok { // 'Events' channel is closed
            eventsWG.Done()
            return
          }
          currentConfigFile, _ := filepath.EvalSymlinks(filename)
          // we only care about the config file with the following cases:
          // 1 - if the config file was modified or created
          // 2 - if the real path to the config file changed (eg: k8s ConfigMap
replacement)
          const writeOrCreateMask = fsnotify.Write | fsnotify.Create
          if (filepath.Clean(event.Name) == configFile &&
            event.Op&writeOrCreateMask != 0) ||
            (currentConfigFile != "" && currentConfigFile != realConfigFile) {
            realConfigFile = currentConfigFile
            err := v.ReadInConfig()
            if err != nil {
              log.Printf("error reading config file: %v\n", err)
            }
            if v.onConfigChange != nil {
              v.onConfigChange(event)
            }
          } else if filepath.Clean(event.Name) == configFile &&
            event.Op&fsnotify.Remove&fsnotify.Remove != 0 {
            eventsWG.Done()
            return
          }

        case err, ok := <-watcher.Errors:
          if ok { // 'Errors' channel is not closed
            log.Printf("watcher error: %v\n", err)
          }
          eventsWG.Done()
          return
        }
```

```
        }
      }()
      watcher.Add(configDir)
      initWG.Done()   // done initializing the watch in this go routine, so the parent routine can move on...
      eventsWG.Wait() // now, wait for event loop to end in this go-routine...
    }()
    initWG.Wait() // make sure that the go routine above fully ended before returning
}
```

其实流程是相似的：

- 首先，调用 `NewWatcher` 创建一个监听器；
- 调用 `v.getConfigFile()` 获取配置文件路径，抽出文件名、目录，配置文件如果是一个符号链接，获得链接指向的路径；
- 调用 `watcher.Add(configDir)` 监听配置文件所在目录，另起一个 goroutine 处理事件。

`WatchConfig` 不能阻塞主 goroutine，所以创建监听器也是新起 goroutine 进行的。代码中有两个 `sync.WaitGroup` 变量， `initWG` 是为了保证监听器初始化， `eventsWG` 是在事件通道关闭，或配置被删除了，或遇到错误时退出事件处理循环。

然后就是核心事件循环：

- 有事件发生时，判断变化的文件是否是在 viper 中设置的配置文件，发生的是否是创建或修改事件（只处理这两个事件）；
- 如果配置文件为符号链接，若符合链接的指向修改了，也需要重新加载配置；
- 如果需要重新加载配置，调用 `v.ReadInConfig()` 读取新的配置；
- 如果注册了事件回调，以发生的事件为参数执行回调。

总结

`fsnotify` 的接口非常简单直接，所有系统相关的复杂性都被封装起来了。这也是我们平时设计模块和接口时可以参考的案例。

参考

1. fsnotify API 设计
2. fsnotify GitHub 仓库

cast

简介

今天我们再来介绍 spf13 大神的另一个库cast。 `cast` 是一个小巧、实用的类型转换库，用于将一个类型转为另一个类型。
最初开发 `cast` 是用在hugo中的。

快速使用

先安装：

```
$ go get github.com/spf13/cast
```

后使用：

```
package main

import (
  "fmt"

  "github.com/spf13/cast"
)

func main() {
  // ToString
  fmt.Println(cast.ToString("leedarjun"))         // leedarjun
  fmt.Println(cast.ToString(8))                   // 8
  fmt.Println(cast.ToString(8.31))                // 8.31
  fmt.Println(cast.ToString([]byte("one time")))  // one time
  fmt.Println(cast.ToString(nil))                 // ""

  var foo interface{} = "one more time"
  fmt.Println(cast.ToString(foo))                 // one more time

  // ToInt
  fmt.Println(cast.ToInt(8))      // 8
  fmt.Println(cast.ToInt(8.31))   // 8
  fmt.Println(cast.ToInt("8"))    // 8
  fmt.Println(cast.ToInt(true))   // 1
  fmt.Println(cast.ToInt(false))  // 0
```

```go
    var eight interface{} = 8
    fmt.Println(cast.ToInt(eight)) // 8
    fmt.Println(cast.ToInt(nil))   // 0
}
```

实际上， `cast` 实现了多种常见类型之间的相互转换，返回最符合直觉的结果。例如：

- `nil` 转为 `string` 的结果为 `""` ，而不是 `"nil"` ；
- `true` 转为 `string` 的结果为 `"true"` ，而 `true` 转为 `int` 的结果为 `1` ；
- `interface{}` 转为其他类型，要看它里面存储的值类型。

这些类型包括所有的基本类型（整形、浮点型、布尔值和字符串）、空接口、 `nil` ，时间（ `time.Time` ）、时长（ `time.Duration` ）以及它们的切片类型，还有 `map[string]Type` （其中 `Type` 为前面提到的类型）：

```
byte     bool      float32    float64    string
int8     int16     int32      int64      int
uint8    uint16    uint32     uint64     uint
interface{}   time.Time  time.Duration   nil
```

高级转换

`cast` 提供了两组函数：

- `ToType` （其中 `Type` 可以为任何支持的类型），将参数转换为 `Type` 类型。如果无法转换，返回 `Type` 类型的零值或 `nil` ；
- `ToTypeE` 以 E 结尾，返回转换后的值和一个 `error` 。这组函数可以区分参数中实际存储了零值，还是转换失败了。

实现上大部分代码都类似， `ToType` 在内部调用 `ToTypeE` 函数，返回结果并忽略错误。 `ToType` 函数的实现在文件 `cast.go` 中，而 `ToTypeE` 函数的实现在文件 `caste.go` 中。

```go
// cast/cast.go
func ToBool(i interface{}) bool {
    v, _ := ToBoolE(i)
    return v
}
```

```
// ToDuration casts an interface to a time.Duration type.
func ToDuration(i interface{}) time.Duration {
  v, _ := ToDurationE(i)
  return v
}
```

`ToTypeE` 函数都接受任意类型的参数（ `interface{}` ），然后使用类型断言根据具体的类型来执行不同的转换。如果无法转换，返回错误。

```
// cast/caste.go
func ToBoolE(i interface{}) (bool, error) {
  i = indirect(i)

  switch b := i.(type) {
  case bool:
    return b, nil
  case nil:
    return false, nil
  case int:
    if i.(int) != 0 {
      return true, nil
    }
    return false, nil
  case string:
    return strconv.ParseBool(i.(string))
  default:
    return false, fmt.Errorf("unable to cast %#v of type %T to bool", i, i)
  }
}
```

首先调用 `indirect` 函数将参数中可能的指针去掉。如果类型本身不是指针，那么直接返回。否则返回指针指向的值。
循环直到返回一个非指针的值：

```
// cast/caste.go
func indirect(a interface{}) interface{} {
  if a == nil {
    return nil
  }
  if t := reflect.TypeOf(a); t.Kind() != reflect.Ptr {
    // Avoid creating a reflect.Value if it's not a pointer.
    return a
  }
  v := reflect.ValueOf(a)
```

```
    for v.Kind() == reflect.Ptr && !v.IsNil() {
        v = v.Elem()
    }
    return v.Interface()
}
```

所以，下面代码输出都是 8：

```
package main

import (
    "fmt"

    "github.com/spf13/cast"
)

func main() {
    p := new(int)
    *p = 8
    fmt.Println(cast.ToInt(p))    // 8

    pp := &p
    fmt.Println(cast.ToInt(pp))  // 8
}
```

时间和时长转换

时间类型的转换代码如下：

```
func ToTimeE(i interface{}) (tim time.Time, err error) {
    i = indirect(i)

    switch v := i.(type) {
    case time.Time:
        return v, nil
    case string:
        return StringToDate(v)
    case int:
        return time.Unix(int64(v), 0), nil
    case int64:
        return time.Unix(v, 0), nil
    case int32:
        return time.Unix(int64(v), 0), nil
    case uint:
```

```
        return time.Unix(int64(v), 0), nil
    case uint64:
        return time.Unix(int64(v), 0), nil
    case uint32:
        return time.Unix(int64(v), 0), nil
    default:
        return time.Time{}, fmt.Errorf("unable to cast %#v of type %T to Time", i,
 i)
    }
}
```

根据传入的类型执行不同的处理：

- 如果是 `time.Time`，直接返回；
- 如果是整型，将参数作为时间戳（自 UTC 时间 `1970.01.01 00:00:00` 到现在的秒数）调用 `time.Unix` 生成时间。`Unix` 接受两个参数，第一个参数指定秒，第二个参数指定纳秒；
- 如果是字符串，调用 `StringToDate` 函数依次尝试以下面这些时间格式调用 `time.Parse` 解析该字符串。如果某个格式解析成功，则返回获得的 `time.Time`。否则解析失败，返回错误；
- 其他任何类型都无法转换为 `time.Time`。

字符串转换为时间：

```
// cast/caste.go
func StringToDate(s string) (time.Time, error) {
    return parseDateWith(s, []string{
        time.RFC3339,
        "2006-01-02T15:04:05", // iso8601 without timezone
        time.RFC1123Z,
        time.RFC1123,
        time.RFC822Z,
        time.RFC822,
        time.RFC850,
        time.ANSIC,
        time.UnixDate,
        time.RubyDate,
        "2006-01-02 15:04:05.999999999 -0700 MST", // Time.String()
        "2006-01-02",
        "02 Jan 2006",
        "2006-01-02T15:04:05-0700", // RFC3339 without timezone hh:mm colon
        "2006-01-02 15:04:05 -07:00",
```

```
    "2006-01-02 15:04:05 -0700",
    "2006-01-02 15:04:05Z07:00", // RFC3339 without T
    "2006-01-02 15:04:05Z0700",  // RFC3339 without T or timezone hh:mm colon
    "2006-01-02 15:04:05",
    time.Kitchen,
    time.Stamp,
    time.StampMilli,
    time.StampMicro,
    time.StampNano,
  })
}

func parseDateWith(s string, dates []string) (d time.Time, e error) {
  for _, dateType := range dates {
    if d, e = time.Parse(dateType, s); e == nil {
      return
    }
  }
  return d, fmt.Errorf("unable to parse date: %s", s)
}
```

时长类型的转换代码如下：

```
// cast/caste.go
func ToDurationE(i interface{}) (d time.Duration, err error) {
  i = indirect(i)

  switch s := i.(type) {
  case time.Duration:
    return s, nil
  case int, int64, int32, int16, int8, uint, uint64, uint32, uint16, uint8:
    d = time.Duration(ToInt64(s))
    return
  case float32, float64:
    d = time.Duration(ToFloat64(s))
    return
  case string:
    if strings.ContainsAny(s, "nsuµmh") {
      d, err = time.ParseDuration(s)
    } else {
      d, err = time.ParseDuration(s + "ns")
    }
    return
  default:
    err = fmt.Errorf("unable to cast %#v of type %T to Duration", i, i)
```

```
        return
    }
}
```

根据传入的类型进行不同的处理：

- 如果是 `time.Duration` 类型，直接返回；
- 如果是整型或浮点型，将其数值强制转换为 `time.Duration` 类型，单位默认为 `ns` ；
- 如果是字符串，分为两种情况：如果字符串中有时间单位符号 `nsuµmh` ，直接调用 `time.ParseDuration` 解析；否则在字符串后拼接 `ns` 再调用 `time.ParseDuration` 解析；
- 其他类型解析失败。

示例：

```
package main

import (
    "fmt"
    "time"

    "github.com/spf13/cast"
)

func main() {
    now := time.Now()
    timestamp := 1579615973
    timeStr := "2020-01-21 22:13:48"

    fmt.Println(cast.ToTime(now))         // 2020-01-22 06:31:50.5068465 +0800 CST m
=+0.000997701
    fmt.Println(cast.ToTime(timestamp)) // 2020-01-21 22:12:53 +0800 CST
    fmt.Println(cast.ToTime(timeStr))   // 2020-01-21 22:13:48 +0000 UTC

    d, _ := time.ParseDuration("1m30s")
    ns := 30000
    strWithUnit := "130s"
    strWithoutUnit := "130"

    fmt.Println(cast.ToDuration(d))                   // 1m30s
    fmt.Println(cast.ToDuration(ns))                  // 30µs
    fmt.Println(cast.ToDuration(strWithUnit))         // 2m10s
```

```
	fmt.Println(cast.ToDuration(strWithoutUnit))  // 130ns
}
```

转换为切片

实际上，这些函数的实现基本类似。使用类型断言判断类型。如果就是要返回的类型，直接返回。否则根据类型进行相应的转换。

我们主要分析两个实现：`ToIntSliceE` 和 `ToStringSliceE`。`ToBoolSliceE/ToDurationSliceE` 与 `ToIntSliceE` 基本相同。

首先是 `ToIntSliceE`：

```
func ToIntSliceE(i interface{}) ([]int, error) {
	if i == nil {
		return []int{}, fmt.Errorf("unable to cast %#v of type %T to []int", i, i)
	}

	switch v := i.(type) {
	case []int:
		return v, nil
	}

	kind := reflect.TypeOf(i).Kind()
	switch kind {
	case reflect.Slice, reflect.Array:
		s := reflect.ValueOf(i)
		a := make([]int, s.Len())
		for j := 0; j < s.Len(); j++ {
			val, err := ToIntE(s.Index(j).Interface())
			if err != nil {
				return []int{}, fmt.Errorf("unable to cast %#v of type %T to []int", i,
 i)
			}
			a[j] = val
		}
		return a, nil
	default:
		return []int{}, fmt.Errorf("unable to cast %#v of type %T to []int", i, i)
	}
}
```

根据传入参数的类型：

- 如果是 `nil` ，直接返回错误；
- 如果是 `[]int` ，不用转换，直接返回；
- 如果传入类型为**切片**或**数组**，新建一个 `[]int` ，将切片或数组中的每个元素转为 `int` 放到该 `[]int` 中。最后返回这个 `[]int` ；
- 其他情况，不能转换。

`ToStringSliceE` ：

```
func ToStringSliceE(i interface{}) ([]string, error) {
  var a []string

  switch v := i.(type) {
  case []interface{}:
    for _, u := range v {
      a = append(a, ToString(u))
    }
    return a, nil
  case []string:
    return v, nil
  case string:
    return strings.Fields(v), nil
  case interface{}:
    str, err := ToStringE(v)
    if err != nil {
      return a, fmt.Errorf("unable to cast %#v of type %T to []string", i, i)
    }
    return []string{str}, nil
  default:
    return a, fmt.Errorf("unable to cast %#v of type %T to []string", i, i)
  }
}
```

根据传入的参数类型：

- 如果是 `[]interface{}` ，将该参数中每个元素转为 `string` ，返回结果切片；
- 如果是 `[]string` ，不需要转换，直接返回；
- 如果是 `interface{}` ，将参数转为 `string` ，返回只包含这个值的切片；
- 如果是 `string` ，调用 `strings.Fields` 函数按空白符将参数拆分，返回拆分后的字符串切片；
- 其他情况，不能转换。

示例：

```go
package main

import (
  "fmt"

  "github.com/spf13/cast"
)

func main() {
  sliceOfInt := []int{1, 3, 7}
  arrayOfInt := [3]int{8, 12}
  // ToIntSlice
  fmt.Println(cast.ToIntSlice(sliceOfInt))  // [1 3 7]
  fmt.Println(cast.ToIntSlice(arrayOfInt))  // [8 12 0]

  sliceOfInterface := []interface{}{1, 2.0, "darjun"}
  sliceOfString := []string{"abc", "dj", "pipi"}
  stringFields := " abc  def hij   "
  any := interface{}(37)
  // ToStringSliceE
  fmt.Println(cast.ToStringSlice(sliceOfInterface))  // [1 2 darjun]
  fmt.Println(cast.ToStringSlice(sliceOfString))     // [abc dj pipi]
  fmt.Println(cast.ToStringSlice(stringFields))      // [abc def hij]
  fmt.Println(cast.ToStringSlice(any))               // [37]
}
```

转为 `map[string]Type` 类型

`cast` 库能将传入的参数转为 `map[string]Type` 类型， `Type` 为上面支持的类型。

其实只需要分析一个 `ToStringMapStringE` 函数就可以了，其他的实现基本一样。 `ToStringMapStringE` 返回 `map[string]string` 类型的值。

```go
func ToStringMapStringE(i interface{}) (map[string]string, error) {
  var m = map[string]string{}

  switch v := i.(type) {
  case map[string]string:
    return v, nil
  case map[string]interface{}:
    for k, val := range v {
      m[ToString(k)] = ToString(val)
    }
```

```
        return m, nil
    case map[interface{}]string:
        for k, val := range v {
            m[ToString(k)] = ToString(val)
        }
        return m, nil
    case map[interface{}]interface{}:
        for k, val := range v {
            m[ToString(k)] = ToString(val)
        }
        return m, nil
    case string:
        err := jsonStringToObject(v, &m)
        return m, err
    default:
        return m, fmt.Errorf("unable to cast %#v of type %T to map[string]string",
 i, i)
    }
}
```

根据传入的参数类型：

- 如果是 `map[string]string` ，不用转换，直接返回；
- 如果是 `map[string]interface{}` ，将每个值转为 `string` 存入新的 map，最后返回新的 map；
- 如果是 `map[interface{}]string` ，将每个键转为 `string` 存入新的 map，最后返回新的 map；
- 如果是 `map[interface{}]interface{}` ，将每个键和值都转为 `string` 存入新的 map，最后返回新的 map；
- **如果是 `string` 类型， `cast` 将它看成一个 JSON 串，解析这个 JSON 到 `map[string]string` ，然后返回结果；**
- 其他情况，返回错误。

示例：

```
package main

import (
    "fmt"

    "github.com/spf13/cast"
)
```

```go
func main() {
  m1 := map[string]string {
    "name": "darjun",
    "job": "developer",
  }

  m2 := map[string]interface{} {
    "name": "jingwen",
    "age": 18,
  }

  m3 := map[interface{}]string {
    "name": "pipi",
    "job": "designer",
  }

  m4 := map[interface{}]interface{} {
    "name": "did",
    "age": 29,
  }

  jsonStr := `{"name":"bibi", "job":"manager"}`

  fmt.Println(cast.ToStringMapString(m1))      // map[job:developer name:darjun]
  fmt.Println(cast.ToStringMapString(m2))      // map[age:18 name:jingwen]
  fmt.Println(cast.ToStringMapString(m3))      // map[job:designer name:pipi]
  fmt.Println(cast.ToStringMapString(m4))      // map[job:designer name:pipi]
  fmt.Println(cast.ToStringMapString(jsonStr)) // map[job:manager name:bibi]
}
```

总结

`cast` 库能在几乎所有常见类型之间转换，使用非常方便。代码量也很小，有时间建议读读源码。

完整示例代码在GitHub上。

ps: 本以为春节就这么几天，偷个懒，没想到。。。

参考

1. cast GitHub 仓库

cobra

简介

cobra是一个命令行程序库，可以用来编写命令行程序。同时，它也提供了一个脚手架，
用于生成基于 cobra 的应用程序框架。非常多知名的开源项目使用了 cobra 库构建命令行，如
Kubernetes、Hugo、etcd等等等等。
本文介绍 cobra 库的基本使用和一些有趣的特性。

关于作者spf13，这里多说两句。spf13 开源不少项目，而且他的开源项目质量都比较高。
相信使用过 vim 的都知道spf13-vim，号称 vim 终极配置。
可以一键配置，对于我这样的懒人来说绝对是福音。他的viper是一个完整的配置解决方案。
完美支持 JSON/TOML/YAML/HCL/envfile/Java properties 配置文件等格式，还有一些比较
实用的特性，如配置热更新、多查找目录、配置保存等。
还有非常火的静态网站生成器hugo也是他的作品。

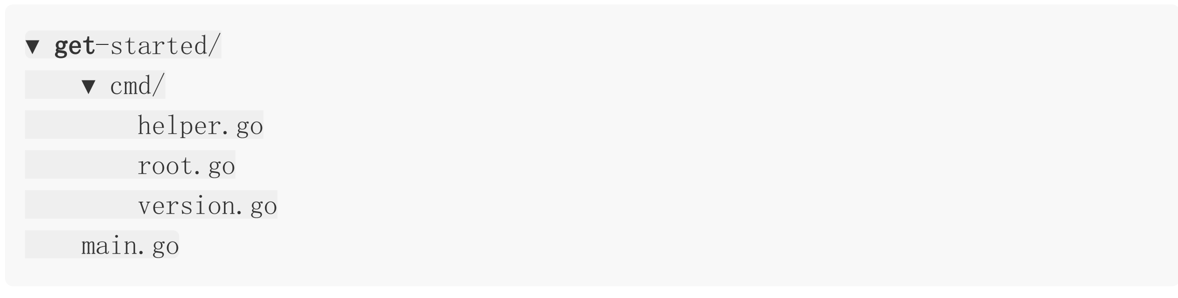
快速使用

第三方库都需要先安装，后使用。下面命令安装了 `cobra` 生成器程序和 cobra 库：

```
$ go get github.com/spf13/cobra/cobra
```

如果出现了 `golang.org/x/text` 库找不到之类的错误，需要手动从 GitHub 上下载该库，
再执行上面的安装命令。我以前写过一篇博客搭建 Go 开发环境提到了这个方法。

我们实现一个简单的命令行程序 git，当然这不是真的 git，只是模拟其命令行。最终还是通
过 `os/exec` 库调用外部程序执行真实的 git 命令，返回结果。
所以我们的系统上要安装 git，且 git 在可执行路径中。目前我们只添加一个子命
令 `version` 。目录结构如下：



```
▼ get-started/
    ▼ cmd/
        helper.go
        root.go
        version.go
    main.go
```

`root.go` ：

```
package cmd

import (
  "errors"

  "github.com/spf13/cobra"
)

var rootCmd = &cobra.Command {
  Use: "git",
  Short: "Git is a distributed version control system.",
  Long: `Git is a free and open source distributed version control system
designed to handle everything from small to very large projects
with speed and efficiency.`,
  Run: func(cmd *cobra.Command, args []string) {
    Error(cmd, args, errors.New("unrecognized command"))
  },
}

func Execute() {
  rootCmd.Execute()
}
```

`version.go` :

```
package cmd

import (
  "fmt"
  "os"

  "github.com/spf13/cobra"
)

var versionCmd = &cobra.Command {
  Use: "version",
  Short: "version subcommand show git version info.",

  Run: func(cmd *cobra.Command, args []string) {
    output, err := ExecuteCommand("git", "version", args...)
    if err != nil {
      Error(cmd, args, err)
    }

    fmt.Fprint(os.Stdout, output)
```

```
	},
}

func init() {
	rootCmd.AddCommand(versionCmd)
}
```

`main.go` 文件中只是调用命令入口：

```
package main

import (
	"github.com/darjun/go-daily-lib/cobra/get-started/cmd"
)

func main() {
	cmd.Execute()
}
```

为了编码方便，在 `helpers.go` 中封装了调用外部程序和错误处理函数：

```
package cmd

import (
	"fmt"
	"os"
	"os/exec"

	"github.com/spf13/cobra"
)

func ExecuteCommand(name string, subname string, args ...string) (string, error)
{
	args = append([]string{subname}, args...)

	cmd := exec.Command(name, args...)
	bytes, err := cmd.CombinedOutput()

	return string(bytes), err
}

func Error(cmd *cobra.Command, args []string, err error) {
	fmt.Fprintf(os.Stderr, "execute %s args:%v error:%v\n", cmd.Name(), args, err)
	os.Exit(1)
}
```

每个 cobra 程序都有一个根命令，可以给它添加任意多个子命令。我们在 `version.go` 的 `init` 函数中将子命令添加到根命令中。

编译程序。注意，不能直接 `go run main.go` ，这已经不是单文件程序了。如果强行要用，请使用 `go run .` ：

```
$ go build -o main.exe
```

cobra 自动生成的帮助信息，very cool：

```
$ ./main.exe -h
Git is a free and open source distributed version control system
designed to handle everything from small to very large projects
with speed and efficiency.

Usage:
  git [flags]
  git [command]

Available Commands:
  help        Help about any command
  version     version subcommand show git version info.

Flags:
  -h, --help   help for git

Use "git [command] --help" for more information about a command.
```

单个子命令的帮助信息：

```
$ ./main.exe version -h
version subcommand show git version info.

Usage:
  git version [flags]

Flags:
  -h, --help   help for version
```

调用子命令：

```
$ ./main.exe version
git version 2.19.1.windows.1
```

未识别的子命令：

```
$ ./main.exe clone
Error: unknown command "clone" for "git"
Run 'git --help' for usage.
```

编译时可以将 `main.exe` 改成 `git` ，用起来会更有感觉□。

```
$ go build -o git
$ ./git version
git version 2.19.1.windows.1
```

使用 cobra 构建命令行时，程序的目录结构一般比较简单，推荐使用下面这种结构：

```
▼ appName/
    ▼ cmd/
        cmd1.go
        cmd2.go
        cmd3.go
        root.go
    main.go
```

每个命令实现一个文件，所有命令文件存放在 `cmd` 目录下。外层的 `main.go` 仅初始化 cobra。

特性

cobra 提供非常丰富的功能：

- 轻松支持子命令，如 `app server` ， `app fetch` 等；
- 完全兼容 POSIX 选项（包括短、长选项）；
- 嵌套子命令；
- 全局、本地层级选项。可以在多处设置选项，按照一定的顺序取用；
- 使用脚手架轻松生成程序框架和命令。

首先需要明确 3 个基本概念：

- 命令（Command）：就是需要执行的操作；
- 参数（Arg）：命令的参数，即要操作的对象；
- 选项（Flag）：命令选项可以调整命令的行为。

下面示例中，`server` 是一个（子）命令，`--port` 是选项：

`hugo server --port=1313`

下面示例中，`clone` 是一个（子）命令，`URL` 是参数，`--bare` 是选项：

`git clone URL --bare`

命令

在 cobra 中，命令和子命令都是用 `Command` 结构表示的。`Command` 有非常多的字段，用来定制命令的行为。
在实际中，最常用的就那么几个。我们在前面示例中已经看到了 `Use/Short/Long/Run` 。

`Use` 指定使用信息，即命令怎么被调用，格式为 `name arg1 [arg2]` 。`name` 为命令名，后面的 `arg1` 为必填参数，`arg3` 为可选参数，参数可以多个。

`Short/Long` 都是指定命令的帮助信息，只是前者简短，后者详尽而已。

`Run` 是实际执行操作的函数。

定义新的子命令很简单，就是创建一个 `cobra.Command` 变量，设置一些字段，然后添加到根命令中。例如我们要添加一个 `clone` 子命令：

```
package cmd

import (
  "fmt"
  "os"

  "github.com/spf13/cobra"
)

var cloneCmd = &cobra.Command {
  Use: "clone url [destination]",
  Short: "Clone a repository into a new directory",
  Run: func(cmd *cobra.Command, args []string) {
    output, err := ExecuteCommand("git", "clone", args...)
    if err != nil {
      Error(cmd, args, err)
    }
```

```
    fmt.Fprintf(os.Stdout, output)
  },
}

func init() {
  rootCmd.AddCommand(cloneCmd)
}
```

其中 `Use` 字段 `clone url [destination]` 表示子命令名为 `clone` ，参数 `url` 是必须的，目标路径 `destination` 可选。

我们将程序编译为 `mygit` 可执行文件，然后将它放到 `$GOPATH/bin` 中。我喜欢将 `$GOPATH/bin` 放到 `$PATH` 中，所以可以直接调用 `mygit` 命令了：

```
$ go build -o mygit
$ mv mygit $GOPATH/bin
$ mygit clone https://github.com/darjun/leetcode
Cloning into 'leetcode'...
```

大家可以继续添加命令。但是我这边只是偷了个懒，将操作都转发到实际的 git 去执行了。这确实没什么实际的用处。
有这个思路，试想一下，我们可以结合多个命令实现很多有用的工具，例如打包工具😉。

选项

cobra 中选项分为两种，一种是**永久选项**，定义它的命令和其子命令都可以使用。通过给根命令添加一个选项定义全局选项。
另一种是**本地选项**，只能在定义它的命令中使用。

cobra 使用pflag解析命令行选项。 `pflag` 使用上基本与 `flag` 相同，该系列文章有一篇介绍 `flag` 库的，Go 每日一库之 flag。

与 `flag` 一样，存储选项的变量也需要提前定义好：

```
var Verbose bool
var Source string
```

设置永久选项：

```
rootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")
```

设置本地选项：

```
localCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read from")
```

两种参数都是相同的，长选项/短选项名、默认值和帮助信息。

下面，我们通过一个案例来演示选项的使用。

假设我们要做一个简单的计算器，支持加、减、乘、除操作。并且可以通过选项设置是否忽略非数字参数，设置除 0 是否报错。
显然，前一个选项应该放在全局选项中，后一个应该放在除法命令中。程序结构如下：

```
▼ math/
    ▼ cmd/
        add.go
        divide.go
        minus.go
        multiply.go
        root.go
    main.go
```

这里展示 `divide.go` 和 `root.go` ，其它命令文件都类似。完整代码我放在GitHub上了。

`divide.go` ：

```
var (
  dividedByZeroHanding int // 除 0 如何处理
)
var divideCmd = &cobra.Command {
  Use: "divide",
  Short: "Divide subcommand divide all passed args.",
  Run: func(cmd *cobra.Command, args []string) {
    values := ConvertArgsToFloat64Slice(args, ErrorHandling(parseHandling))
    result := calc(values, DIVIDE)
    fmt.Printf("%s = %.2f\n", strings.Join(args, "/"), result)
  },
}

func init() {
  divideCmd.Flags().IntVarP(&dividedByZeroHanding, "divide_by_zero", "d", int(PanicOnDividedByZero), "do what when divided by zero")
```

```
    rootCmd.AddCommand(divideCmd)
}
```

`root.go` ：

```
var (
    parseHandling int
)

var rootCmd = &cobra.Command {
    Use: "math",
    Short: "Math calc the accumulative result.",
    Run: func(cmd *cobra.Command, args []string) {
        Error(cmd, args, errors.New("unrecognized subcommand"))
    },
}

func init() {
    rootCmd.PersistentFlags().IntVarP(&parseHandling, "parse_error", "p", int(ContinueOnParseError), "do what when parse arg error")
}

func Execute() {
    rootCmd.Execute()
}
```

在 `divide.go` 中定义了如何处理除 0 错误的选项，在 `root.go` 中定义了如何处理解析错误的选项。选项枚举如下：

```
const (
    ContinueOnParseError  ErrorHandling = 1 // 解析错误尝试继续处理
    ExitOnParseError      ErrorHandling = 2 // 解析错误程序停止
    PanicOnParseError     ErrorHandling = 3 // 解析错误 panic
    ReturnOnDividedByZero ErrorHandling = 4 // 除0返回
    PanicOnDividedByZero  ErrorHandling = 5 // 除0 painc
)
```

其实命令的执行逻辑并不复杂，就是将参数转为 `float64` 。然后执行相应的运算，输出结果。

测试程序：

```
$ go build -o math
$ ./math add 1 2 3 4
```

```
1+2+3+4 = 10.00

$ ./math minus 1 2 3 4
1-2-3-4 = -8.00

$ ./math multiply 1 2 3 4
1*2*3*4 = 24.00

$ ./math divide 1 2 3 4
1/2/3/4 = 0.04
```

默认情况，解析错误被忽略，只计算格式正确的参数的结果：

```
$ ./math add 1 2a 3b 4
1+2a+3b+4 = 5.00

$ ./math divide 1 2a 3b 4
1/2a/3b/4 = 0.25
```

设置解析失败的处理，2 表示退出程序，3 表示 panic（看上面的枚举）：

```
$ ./math add 1 2a 3b 4 -p 2
invalid number: 2a

$ ./math add 1 2a 3b 4 -p 3
panic: strconv.ParseFloat: parsing "2a": invalid syntax

goroutine 1 [running]:
github.com/darjun/go-daily-lib/cobra/math/cmd.ConvertArgsToFloat64Slice(0xc00004
e300, 0x4, 0x6, 0x3, 0xc00008bd70, 0x504f6b, 0xc000098600)
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/helper.go:5
8 +0x2c3
github.com/darjun/go-daily-lib/cobra/math/cmd.glob..func1(0x74c620, 0xc00004e300
, 0x4, 0x6)
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/add.go:14 +
0x6d
github.com/spf13/cobra.(*Command).execute(0x74c620, 0xc00004e1e0, 0x6, 0x6, 0x74
c620, 0xc00004e1e0)
    D:/code/golang/src/github.com/spf13/cobra/command.go:835 +0x2b1
github.com/spf13/cobra.(*Command).ExecuteC(0x74d020, 0x0, 0x599ee0, 0xc000056058
)
    D:/code/golang/src/github.com/spf13/cobra/command.go:919 +0x302
github.com/spf13/cobra.(*Command).Execute(...)
    D:/code/golang/src/github.com/spf13/cobra/command.go:869
github.com/darjun/go-daily-lib/cobra/math/cmd.Execute(...)
```

```
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/root.go:45
main.main()
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/main.go:8 +0x35
```

至于除 0 选项大家自己试试。

细心的朋友应该都注意到了，该程序还有一些不完善的地方。例如这里如果输入非数字参数，该参数也会显示在结果中：

```
$ ./math add 1 2 3d cc
1+2+3d+cc = 3.00
```

感兴趣可以自己完善一下~

脚手架

通过前面的介绍，我们也看到了其实 cobra 命令的框架还是比较固定的。这就有了工具的用武之地了，可极大地提高我们的开发效率。

前面安装 cobra 库的时候也将脚手架程序安装好了。下面我们介绍如何使用这个生成器。

使用 `cobra init` 命令创建一个 cobra 应用程序：

```
$ cobra init scaffold --pkg-name github.com/darjun/go-daily-lib/cobra/scaffold
```

其中 `scaffold` 为应用程序名，后面通过 `pkg-name` 选项指定包路径。生成的程序目录结构如下：

```
▼ scaffold/
    ▼ cmd/
        root.go
    LICENSE
    main.go
```

这个项目结构与之前介绍的完全相同，也是 cobra 推荐使用的结构。同样地， `main.go` 也仅仅是入口。

在 `root.go` 中，工具额外帮我们生成了一些代码。

在根命令中添加了配置文件选项，大部分应用程序都需要配置文件：

```
func init() {
  cobra.OnInitialize(initConfig)
```

```
  rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file (default is $HOME/.scaffold.yaml)")
  rootCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

在初始化完成的回调中，如果发现该选项为空，则默认使用主目录下的 `.scaffold.yaml` 文件：

```
func initConfig() {
  if cfgFile != "" {
    viper.SetConfigFile(cfgFile)
  } else {
    home, err := homedir.Dir()
    if err != nil {
      fmt.Println(err)
      os.Exit(1)
    }

    viper.AddConfigPath(home)
    viper.SetConfigName(".scaffold")
  }

  viper.AutomaticEnv()

  if err := viper.ReadInConfig(); err == nil {
    fmt.Println("Using config file:", viper.ConfigFileUsed())
  }
}
```

这里用到了我前几天介绍的go-homedir库。配置文件的读取使用了 spf13 自己的开源项目 viper（毒龙？真是起名天才）。

除了代码文件，cobra 还生成了一个 LICENSE 文件。

现在这个程序还不能做任何事情，我们需要给它添加子命令，使用 `cobra add` 命令：

```
$ cobra add date
```

该命令在 `cmd` 目录下新增了 `date.go` 文件。基本结构已经搭好了，剩下的就是修改一些描述，添加一些选项了。

我们现在实现这样一个功能，根据传入的年、月，打印这个月的日历。如果没有传入选项，使用当前的年、月。

选项定义：

```
func init() {
  rootCmd.AddCommand(dateCmd)

  dateCmd.PersistentFlags().IntVarP(&year, "year", "y", 0, "year to show (should in [1000, 9999]")
  dateCmd.PersistentFlags().IntVarP(&month, "month", "m", 0, "month to show (should in [1, 12]")
}
```

修改 `dateCmd` 的 `Run` 函数：

```
Run: func(cmd *cobra.Command, args []string) {
  if year < 1000 && year > 9999 {
    fmt.Fprintln(os.Stderr, "invalid year should in [1000, 9999], actual:%d", year)
    os.Exit(1)
  }

  if month < 1 && year > 12 {
    fmt.Fprintln(os.Stderr, "invalid month should in [1, 12], actual:%d", month)
    os.Exit(1)
  }

  showCalendar()
}
```

`showCalendar` 函数就是利用 `time` 提供的方法实现的，这里就不赘述了。感兴趣可以去我的 GitHub 上查看实现。

看看程序运行效果：

```
$ go build -o main.exe
$ ./main.exe date
  Sun  Mon  Tue  Wed  Thu  Fri  Sat
                   1    2    3    4
    5    6    7    8    9   10   11
   12   13   14   15   16   17   18
   19   20   21   22   23   24   25
   26   27   28   29   30   31

$ ./main.exe date --year 2019 --month 12
  Sun  Mon  Tue  Wed  Thu  Fri  Sat
```

```
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

可以再为这个程序添加其他功能，试一试吧~

其他

cobra 提供了非常丰富的特性和定制化接口，例如：

- 设置钩子函数，在命令执行前、后执行某些操作；
- 生成 Markdown/ReStructed Text/Man Page 格式的文档；
- 等等等等。

由于篇幅限制，就不一一介绍了。有兴趣可自行研究。cobra 库的使用非常广泛，很多知名项目都有用到，前面也提到过这些项目。
学习这些项目是如何使用 cobra 的，可以从中学习 cobra 的特性和最佳实践。这也是学习开源项目的一个很好的途径。

文中所有示例代码都已上传至我的 GitHub，Go 每日一库，https://github.com/darjun/go-daily-lib/tree/master/cobra。

参考

1. cobra GitHub 仓库

go-ini

简介

ini 是 Windows 上常用的配置文件格式。MySQL 的 Windows 版就是使用 ini 格式存储配置的。
go-ini是 Go 语言中用于操作 ini 文件的第三方库。

本文介绍 `go-ini` 库的使用。

快速使用

go-ini 是第三方库，使用前需要安装：

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

也可以使用 GitHub 上的仓库：

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

首先，创建一个 `my.ini` 配置文件：

```
app_name = awesome web

# possible values: DEBUG, INFO, WARNING, ERROR, FATAL
log_level = DEBUG

[mysql]
ip = 127.0.0.1
port = 3306
user = dj
password = 123456
database = awesome

[redis]
ip = 127.0.0.1
port = 6381
```

使用 go-ini 库读取：

```go
package main

import (
	"fmt"
	"log"

	"gopkg.in/ini.v1"
)

func main() {
	cfg, err := ini.Load("my.ini")
	if err != nil {
		log.Fatal("Fail to read file: ", err)
	}

	fmt.Println("App Name:", cfg.Section("").Key("app_name").String())
	fmt.Println("Log Level:", cfg.Section("").Key("log_level").String())

	fmt.Println("MySQL IP:", cfg.Section("mysql").Key("ip").String())
	mysqlPort, err := cfg.Section("mysql").Key("port").Int()
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println("MySQL Port:", mysqlPort)
	fmt.Println("MySQL User:", cfg.Section("mysql").Key("user").String())
	fmt.Println("MySQL Password:", cfg.Section("mysql").Key("password").String())
	fmt.Println("MySQL Database:", cfg.Section("mysql").Key("database").String())

	fmt.Println("Redis IP:", cfg.Section("redis").Key("ip").String())
	redisPort, err := cfg.Section("redis").Key("port").Int()
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println("Redis Port:", redisPort)
}
```

在 ini 文件中，每个键值对占用一行，中间使用 `=` 隔开。以 `#` 开头的内容为注释。ini 文件是以分区（section）组织的。
分区以 `[name]` 开始，在下一个分区前结束。所有分区前的内容属于默认分区，如 `my.ini` 文件中的 `app_name` 和 `log_level` 。

使用 `go-ini` 读取配置文件的步骤如下：

- 首先调用 `ini.Load` 加载文件，得到配置对象 `cfg` ；

- 然后以分区名调用配置对象的`Section`方法得到对应的分区对象`section`，默认分区的名字为`""`，也可以使用`ini.DefaultSection`；
- 以键名调用分区对象的`Key`方法得到对应的配置项`key`对象；
- 由于文件中读取出来的都是字符串，`key`对象需根据类型调用对应的方法返回具体类型的值使用，如上面的`String`、`MustInt`方法。

运行以下程序，得到输出：

```
App Name: awesome web
Log Level: DEBUG
MySQL IP: 127.0.0.1
MySQL Port: 3306
MySQL User: dj
MySQL Password: 123456
MySQL Database: awesome
Redis IP: 127.0.0.1
Redis Port: 6381
```

配置文件中存储的都是字符串，所以类型为字符串的配置项不会出现类型转换失败的，故`String()`方法只返回一个值。
但如果类型为`Int/Uint/Float64`这些时，转换可能失败。所以`Int()/Uint()/Float64()`返回一个值和一个错误。

要留意这种不一致！如果我们将配置中 redis 端口改成非法的数字 x6381，那么运行程序将报错：

```
2020/01/14 22:43:13 strconv.ParseInt: parsing "x6381": invalid syntax
```

`Must*`便捷方法

如果每次取值都需要进行错误判断，那么代码写起来会非常繁琐。为此，`go-ini`也提供对应的`MustType`（Type 为`Init/Uint/Float64`等）方法，这个方法只返回一个值。同时它接受可变参数，如果类型无法转换，取参数中第一个值返回，并且该参数设置为这个配置的值，下次调用返回这个值：

```
package main

import (
  "fmt"
  "log"
```

```
	"gopkg.in/ini.v1"
)

func main() {
	cfg, err := ini.Load("my.ini")
	if err != nil {
		log.Fatal("Fail to read file: ", err)
	}

	redisPort, err := cfg.Section("redis").Key("port").Int()
	if err != nil {
		fmt.Println("before must, get redis port error:", err)
	} else {
		fmt.Println("before must, get redis port:", redisPort)
	}

	fmt.Println("redis Port:", cfg.Section("redis").Key("port").MustInt(6381))

	redisPort, err = cfg.Section("redis").Key("port").Int()
	if err != nil {
		fmt.Println("after must, get redis port error:", err)
	} else {
		fmt.Println("after must, get redis port:", redisPort)
	}
}
```

配置文件还是 redis 端口为非数字 x6381 时的状态，运行程序：

```
before must, get redis port error: strconv.ParseInt: parsing "x6381": invalid syntax
redis Port: 6381
after must, get redis port: 6381
```

我们看到第一次调用 `Int` 返回错误，以 6381 为参数调用 `MustInt` 之后，再次调用 `Int` ，成功返回 6381。 `MustInt` 源码也比较简单：

```
// gopkg.in/ini.v1/key.go
func (k *Key) MustInt(defaultVal ...int) int {
	val, err := k.Int()
	if len(defaultVal) > 0 && err != nil {
		k.value = strconv.FormatInt(int64(defaultVal[0]), 10)
		return defaultVal[0]
	}
	return val
}
```

分区操作

获取信息

在加载配置之后，可以通过 `Sections` 方法获取所有分区， `SectionStrings()` 方法获取所有分区名。

```
sections := cfg.Sections()
names := cfg.SectionStrings()

fmt.Println("sections: ", sections)
fmt.Println("names: ", names)
```

运行输出 3 个分区：

```
[DEFAULT mysql redis]
```

调用 `Section(name)` 获取名为 `name` 的分区，如果该分区不存在，则自动创建一个分区返回：

```
newSection := cfg.Section("new")

fmt.Println("new section: ", newSection)
fmt.Println("names: ", cfg.SectionStrings())
```

创建之后调用 `SectionStrings` 方法，新分区也会返回：

```
names: [DEFAULT mysql redis new]
```

也可以手动创建一个新分区，如果分区已存在，则返回错误：

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

父子分区

在配置文件中，可以使用占位符 `%(name)s` 表示用之前已定义的键 `name` 的值来替换，这里的 `s` 表示值为字符串类型：

```
NAME = ini
VERSION = v1
IMPORT_PATH = gopkg.in/%(NAME)s.%(VERSION)s

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

[package.sub]
```

上面在默认分区中设置 `IMPORT_PATH` 的值时，使用了前面定义的 `NAME` 和 `VERSION` 。
在 `package` 分区中设置 `CLONE_URL` 的值时，使用了默认分区中定义的 `IMPORT_PATH` 。

我们还可以在分区名中使用 `.` 表示两个或多个分区之间的父子关系，例如 `package.sub` 的父分区为 `package` ， `package` 的父分区为默认分区。如果某个键在子分区中不存在，则会在它的父分区中再次查找，直到没有父分区为止：

```
cfg, err := ini.Load("parent_child.ini")
if err != nil {
  fmt.Println("Fail to read file: ", err)
  return
}

fmt.Println("Clone url from package.sub:", cfg.Section("package.sub").Key("CLONE_URL").String())
```

运行程序输出：

```
Clone url from package.sub: https://gopkg.in/ini.v1
```

子分区中 `package.sub` 中没有键 `CLONE_URL` ，返回了父分区 `package` 中的值。

保存配置

有时候，我们需要将生成的配置写到文件中。例如在写工具的时候。保存有两种类型的接口，一种直接保存到文件，另一种写入到 `io.Writer` 中：

```
err = cfg.SaveTo("my.ini")
err = cfg.SaveToIndent("my.ini", "\t")
```

```
cfg.WriteTo(writer)
cfg.WriteToIndent(writer, "\t")
```

下面我们通过程序生成前面使用的配置文件 `my.ini` 并保存：

```
package main

import (
  "fmt"
  "os"

  "gopkg.in/ini.v1"
)

func main() {
  cfg := ini.Empty()

  defaultSection := cfg.Section("")
  defaultSection.NewKey("app_name", "awesome web")
  defaultSection.NewKey("log_level", "DEBUG")

  mysqlSection, err := cfg.NewSection("mysql")
  if err != nil {
    fmt.Println("new mysql section failed:", err)
    return
  }
  mysqlSection.NewKey("ip", "127.0.0.1")
  mysqlSection.NewKey("port", "3306")
  mysqlSection.NewKey("user", "root")
  mysqlSection.NewKey("password", "123456")
  mysqlSection.NewKey("database", "awesome")

  redisSection, err := cfg.NewSection("redis")
  if err != nil {
    fmt.Println("new redis section failed:", err)
    return
  }
  redisSection.NewKey("ip", "127.0.0.1")
  redisSection.NewKey("port", "6381")

  err = cfg.SaveTo("my.ini")
  if err != nil {
    fmt.Println("SaveTo failed: ", err)
  }
```

```
    err = cfg.SaveToIndent("my-pretty.ini", "\t")
    if err != nil {
      fmt.Println("SaveToIndent failed: ", err)
    }

    cfg.WriteTo(os.Stdout)
    fmt.Println()
    cfg.WriteToIndent(os.Stdout, "\t")
}
```

运行程序，生成两个文件 `my.ini` 和 `my-pretty.ini` ，同时控制台输出文件内容。

`my.ini` ：

```
app_name  = awesome web
log_level = DEBUG

[mysql]
ip       = 127.0.0.1
port     = 3306
user     = root
password = 123456
database = awesome

[redis]
ip   = 127.0.0.1
port = 6381
```

`my-pretty.ini` ：

```
app_name  = awesome web
log_level = DEBUG

[mysql]
    ip       = 127.0.0.1
    port     = 3306
    user     = root
    password = 123456
    database = awesome

[redis]
    ip   = 127.0.0.1
    port = 6381
```

`*Indent` 方法会对子分区下的键增加缩进，看起来美观一点。

分区与结构体字段映射

定义结构变量，加载完配置文件后，调用 `MapTo` 将配置项赋值到结构变量的对应字段中。

```
package main

import (
    "fmt"

    "gopkg.in/ini.v1"
)

type Config struct {
    AppName    string `ini:"app_name"`
    LogLevel   string `ini:"log_level"`

    MySQL      MySQLConfig `ini:"mysql"`
    Redis      RedisConfig `ini:"redis"`
}

type MySQLConfig struct {
    IP         string `ini:"ip"`
    Port       int `ini:"port"`
    User       string `ini:"user"`
    Password   string `ini:"password"`
    Database   string `ini:"database"`
}

type RedisConfig struct {
    IP       string `ini:"ip"`
    Port     int `ini:"port"`
}

func main() {
    cfg, err := ini.Load("my.ini")
    if err != nil {
        fmt.Println("load my.ini failed: ", err)
    }

    c := Config{}
    cfg.MapTo(&c)
```

```
    fmt.Println(c)
}
```

`MapTo` 内部使用了反射，**所以结构体字段必须都是导出的**。如果键名与字段名不相同，那么需要在结构标签中指定对应的键名。
这一点与 Go 标准库 `encoding/json` 和 `encoding/xml` 不同。标准库 `json/xml` 解析时可以将键名 `app_name` 对应到字段名 `AppName` 。
或许这是 `go-ini` 库可以优化的点？

先加载，再映射有点繁琐，直接使用 `ini.MapTo` 将两步合并：

```
err = ini.MapTo(&c, "my.ini")
```

也可以只映射一个分区：

```
mysqlCfg := MySQLConfig{}
err = cfg.Section("mysql").MapTo(&mysqlCfg)
```

还可以通过结构体生成配置：

```
cfg := ini.Empty()

c := Config {
    AppName:     "awesome web",
    LogLevel:     "DEBUG",
    MySQL: MySQLConfig {
        IP:     "127.0.0.1",
        Port:     3306,
        User:     "root",
        Password:"123456",
        Database:"awesome",
    },
    Redis: RedisConfig {
        IP:         "127.0.0.1",
        Port:     6381,
    },
}

err := ini.ReflectFrom(cfg, &c)
if err != nil {
    fmt.Println("ReflectFrom failed: ", err)
    return
}
```

```
err = cfg.SaveTo("my-copy.ini")
if err != nil {
  fmt.Println("SaveTo failed: ", err)
  return
}
```

总结

本文介绍了 `go-ini` 库的基本用法和一些有趣的特性。示例代码已上传GitHub。
其实 `go-ini` 还有很多高级特性。官方文档非常详细，推荐去看，而且有中文哟~
作者无闻，相信做 Go 开发的都不陌生。

参考

1. go-ini GitHub 仓库
2. go-ini 官方文档

go-homedir

简介

今天我们来看一个很小，很实用的库go-homedir。顾名思义，`go-homedir` 用来获取用户的主目录。
实际上，使用标准库 `os/user` 我们也可以得到这个信息：

```
package main

import (
  "fmt"
  "log"
  "os/user"
)

func main() {
  u, err := user.Current()
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("Home dir:", u.HomeDir)
}
```

那么为什么还要 `go-homedir` 库？

在 Darwin 系统上，标准库 `os/user` 的使用需要 cgo。所以，任何使用 `os/user` 的代码都不能交叉编译。
但是，大多数人使用 `os/user` 的目的仅仅只是想获取主目录。因此，`go-homedir` 库出现了。

快速使用

`go-homedir` 是第三方包，使用前需要先安装：

```
$ go get github.com/mitchellh/go-homedir
```

使用非常简单：

```go
package main

import (
  "fmt"
  "log"

  "github.com/mitchellh/go-homedir"
)

func main() {
  dir, err := homedir.Dir()
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println("Home dir:", dir)

  dir = "~/golang/src"
  expandedDir, err := homedir.Expand(dir)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Printf("Expand of %s is: %s\n", dir, expandedDir)
}
```

`go-homedir` 有两个功能：

- `Dir` ：获取用户主目录；
- `Expand` ：将路径中的第一个 `~` 扩展成用户主目录。

高级用法

由于 `Dir` 的调用可能涉及一些系统调用和外部执行命令，多次调用费性能。所以 `go-homedir` 提供了缓存的功能。默认情况下，缓存是开启的。
我们也可以将 `DisableCache` 设置为 `false` 来关闭它。

```go
package main

import (
  "fmt"
  "log"
```

```
    "github.com/mitchellh/go-homedir"
)

func main() {
    homedir.DisableCache = false

    dir, err := homedir.Dir()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("Home dir:", dir)
}
```

使用缓存时，如果程序运行中修改了主目录，再次调用 `Dir` 还是返回之前的目录。如果需要获取最新的主目录，可以先调用 `Reset` 清除缓存。

实现

`go-homedir` 源码只有一个文件homedir.go，今天我们大概看一下 `Dir` 的实现，去掉缓存相关代码：

```
func Dir() (string, error) {
    var result string
    var err error
    if runtime.GOOS == "windows" {
        result, err = dirWindows()
    } else {
        // Unix-like system, so just assume Unix
        result, err = dirUnix()
    }

    if err != nil {
        return "", err
    }
    return result, nil
}
```

判断当前的系统是 `windows` 还是类 Unix，分别调用不同的方法。先看 windows 的，比较简单：

```
func dirWindows() (string, error) {
    // First prefer the HOME environmental variable
    if home := os.Getenv("HOME"); home != "" {
```

```go
		return home, nil
	}

	// Prefer standard environment variable USERPROFILE
	if home := os.Getenv("USERPROFILE"); home != "" {
		return home, nil
	}

	drive := os.Getenv("HOMEDRIVE")
	path := os.Getenv("HOMEPATH")
	home := drive + path
	if drive == "" || path == "" {
		return "", errors.New("HOMEDRIVE, HOMEPATH, or USERPROFILE are blank")
	}

	return home, nil
}
```

流程如下：

- 读取环境变量 `HOME`，如果不为空，返回这个值；
- 读取环境变量 `USERPROFILE`，如果不为空，返回这个值；
- 读取环境变量 `HOMEDRIVE` 和 `HOMEPATH`，如果两者都不为空，拼接这两个值返回。

类 Unix 系统的实现稍微复杂一点：

```go
func dirUnix() (string, error) {
	homeEnv := "HOME"
	if runtime.GOOS == "plan9" {
		// On plan9, env vars are lowercase.
		homeEnv = "home"
	}

	// First prefer the HOME environmental variable
	if home := os.Getenv(homeEnv); home != "" {
		return home, nil
	}

	var stdout bytes.Buffer

	// If that fails, try OS specific commands
	if runtime.GOOS == "darwin" {
		cmd := exec.Command("sh", "-c", `dscl -q . -read /Users/"$(whoami)" NFSHomeDirectory | sed 's/^[^ ]*: //'`)
```

```
    cmd.Stdout = &stdout
    if err := cmd.Run(); err == nil {
      result := strings.TrimSpace(stdout.String())
      if result != "" {
        return result, nil
      }
    }
  } else {
    cmd := exec.Command("getent", "passwd", strconv.Itoa(os.Getuid()))
    cmd.Stdout = &stdout
    if err := cmd.Run(); err != nil {
      // If the error is ErrNotFound, we ignore it. Otherwise, return it.
      if err != exec.ErrNotFound {
        return "", err
      }
    } else {
      if passwd := strings.TrimSpace(stdout.String()); passwd != "" {
        // username:password:uid:gid:gecos:home:shell
        passwdParts := strings.SplitN(passwd, ":", 7)
        if len(passwdParts) > 5 {
          return passwdParts[5], nil
        }
      }
    }
  }

  // If all else fails, try the shell
  stdout.Reset()
  cmd := exec.Command("sh", "-c", "cd && pwd")
  cmd.Stdout = &stdout
  if err := cmd.Run(); err != nil {
    return "", err
  }

  result := strings.TrimSpace(stdout.String())
  if result == "" {
    return "", errors.New("blank output when reading home directory")
  }

  return result, nil
}
```

流程如下：

- 先读取环境变量`HOME`（注意 plan9 系统上为`home`），如果不为空，返回这个值；

- 使用 `getnet` 命令查看系统的数据库中的相关记录，我们知道 `passwd` 文件中存储了用户信息，包括用户的主目录。使用 `getent` 命令查看 `passwd` 中当前用户的那条记录，然后从中找到主目录部分返回；
- 如果上一个步骤失败了，我们知道 `cd` 后不加参数是直接切换到用户主目录的，而 `pwd` 可以显示当前目录。那么就可以结合这两个命令返回主目录。

这里分析源码并不是表示使用任何库都要熟悉它的源码，毕竟使用库就是为了方便开发。
但是源码是我们学习和提高的一个非常重要的途径。我们在使用库遇到问题的时候也要有能力从文档或甚至源码中查找原因。

参考

1. home-dir GitHub 仓库

go-flags

简介

在上一篇文章中，我们介绍了 `flag` 库。 `flag` 库是用于解析命令行选项的。但是 `flag` 有几个缺点：

- 不显示支持短选项。当然上一篇文章中也提到过可以通过将两个选项共享同一个变量迂回实现，但写起来比较繁琐；
- 选项变量的定义比较繁琐，每个选项都需要根据类型调用对应的 `Type` 或 `TypeVar` 函数；
- 默认只支持有限的数据类型，当前只有基本类型 `bool/int/uint/string` 和 `time.Duration` ；

为了解决这些问题，出现了不少第三方解析命令行选项的库，今天的主角 `go-flags` 就是其中一个。第一次看到 `go-flags` 库是在阅读pgweb源码的时候。

`go-flags` 提供了比标准库 `flag` 更多的选项。它利用结构标签（struct tag）和反射提供了一个方便、简洁的接口。它除了基本的功能，还提供了丰富的特性：

- 支持短选项（-v）和长选项（–verbose）；
- 支持短选项合写，如 `-aux` ；
- 同一个选项可以设置多个值；
- 支持所有的基础类型和 map 类型，甚至是函数；
- 支持命名空间和选项组；
- 等等。

上面只是粗略介绍了 `go-flags` 的特性，下面我们依次来介绍。

快速开始

学习从使用开始！我们先来看看 `go-flags` 的基本使用。

由于是第三方库，使用前需要安装，执行下面的命令安装：

```
$ go get github.com/jessevdk/go-flags
```

代码中使用 `import` 导入该库：

```
import "github.com/jessevdk/go-flags"
```

完整示例代码如下：

```
package main

import (
  "fmt"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Verbose []bool `short:"v" long:"verbose" description:"Show verbose debug messa
ge"`
}

func main() {
  var opt Option
  flags.Parse(&opt)

  fmt.Println(opt.Verbose)
}
```

使用 `go-flags` 的一般步骤：

- 定义选项结构，在结构标签中设置选项信息。通过 `short` 和 `long` 设置短、长选项名字，`description` 设置帮助信息。命令行传参时，短选项前加 `-`，长选项前加 `--`；
- 声明选项变量；
- 调用 `go-flags` 的解析方法解析。

编译、运行代码（我的环境是 Win10 + Git Bash）：

```
$ go build -o main.exe main.go
```

短选项：

```
$ ./main.exe -v
[true]
```

长选项：

```
$ ./main.exe --verbose
[true]
```

由于 `Verbose` 字段是切片类型，每次遇到 `-v` 或 `--verbose` 都会追加一个 `true` 到切片中。

多个短选项：

```
$ ./main.exe -v -v
[true true]
```

多个长选项：

```
$ ./main.exe --verbose --verbose
[true true]
```

短选项 + 长选项：

```
$ ./main.exe -v --verbose -v
[true true true]
```

短选项合写：

```
$ ./main.exe -vvv
[true true true]
```

基本特性

支持丰富的数据类型

`go-flags` 相比标准库 `flag` 支持更丰富的数据类型：

- 所有的基本类型（包括有符号整数 `int/int8/int16/int32/int64`，无符号整数 `uint/uint8/uint16/uint32/uint64`，浮点数 `float32/float64`，布尔类型 `bool` 和字符串 `string`）和**它们的切片**；
- map 类型。只支持键为 `string`，值为基础类型的 map；

- 函数类型。

如果字段是基本类型的切片，基本解析流程与对应的基本类型是一样的。切片类型选项的不同之处在于，遇到相同的选项时，值会被追加到切片中。而非切片类型的选项，后出现的值会覆盖先出现的值。

下面来看一个示例：

```
package main

import (
  "fmt"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  IntFlag         int             `short:"i" long:"int" description:"int flag value"`
  IntSlice        []int           `long:"intslice" description:"int slice flag value"`
  BoolFlag        bool            `long:"bool" description:"bool flag value"`
  BoolSlice       []bool          `long:"boolslice" description:"bool slice flag value"`
  FloatFlag       float64         `long:"float", description:"float64 flag value"`
  FloatSlice      []float64       `long:"floatslice" description:"float64 slice flag value"`
  StringFlag      string          `short:"s" long:"string" description:"string flag value"`
  StringSlice     []string        `long:"strslice" description:"string slice flag value"`
  PtrStringSlice  []*string       `long:"pstrslice" description:"slice of pointer of string flag value"`
  Call            func(string)    `long:"call" description:"callback"`
  IntMap          map[string]int  `long:"intmap" description:"A map from string to int"`
}

func main() {
  var opt Option
  opt.Call = func (value string) {
    fmt.Println("in callback: ", value)
  }
```

```
    _, err := flags.Parse(&opt)
    if err != nil {
      fmt.Println("Parse error:", err)
      return
    }

    fmt.Printf("int flag: %v\n", opt.IntFlag)
    fmt.Printf("int slice flag: %v\n", opt.IntSlice)
    fmt.Printf("bool flag: %v\n", opt.BoolFlag)
    fmt.Printf("bool slice flag: %v\n", opt.BoolSlice)
    fmt.Printf("float flag: %v\n", opt.FloatFlag)
    fmt.Printf("float slice flag: %v\n", opt.FloatSlice)
    fmt.Printf("string flag: %v\n", opt.StringFlag)
    fmt.Printf("string slice flag: %v\n", opt.StringSlice)
    fmt.Println("slice of pointer of string flag: ")
    for i := 0; i < len(opt.PtrStringSlice); i++ {
      fmt.Printf("\t%d: %v\n", i, *opt.PtrStringSlice[i])
    }
    fmt.Printf("int map: %v\n", opt.IntMap)
}
```

基本类型和其切片比较简单，就不过多介绍了。值得留意的是基本类型指针的切片，即上面的 `PtrStringSlice` 字段，类型为 `[]*string` 。
由于结构中存储的是字符串指针， `go-flags` 在解析过程中遇到该选项会自动创建字符串，将指针追加到切片中。

运行程序，传入 `--pstrslice` 选项：

```
$ ./main.exe --pstrslice test1 --pstrslice test2
slice of pointer of string flag:
    0: test1
    1: test2
```

另外，我们可以在选项中定义函数类型。该函数的唯一要求是有一个字符串类型的参数。解析中每次遇到该选项就会以选项值为参数调用这个函数。
上面代码中， `Call` 函数只是简单的打印传入的选项值。运行代码，传入 `--call` 选项：

```
$ ./main.exe --call test1 --call test2
in callback:  test1
in callback:  test2
```

最后， `go-flags` 还支持 map 类型。虽然限制键必须是 `string` 类型，值必须是基本类型，也能实现比较灵活的配置。

`map` 类型的选项值中键-值通过 `:` 分隔，如 `key:value` ，可设置多个。运行代码，传入 `--intmap` 选项：

```
$ ./main.exe --intmap key1:12 --intmap key2:58
int map: map[key1:12 key2:58]
```

常用设置

`go-flags` 提供了非常多的设置选项，具体可参见文档。这里重点介绍两个 `required` 和 `default` 。

`required` 非空时，表示对应的选项必须设置值，否则解析时返回 `ErrRequired` 错误。

`default` 用于设置选项的默认值。如果已经设置了默认值，那么 `required` 是否设置并不影响，也就是说命令行参数中该选项可以没有。

看下面示例：

```go
package main

import (
  "fmt"
  "log"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Required    string  `short:"r" long:"required" required:"true"`
  Default     string  `short:"d" long:"default" default:"default"`
}

func main() {
  var opt Option
  _, err := flags.Parse(&opt)
  if err != nil {
    log.Fatal("Parse error:", err)
  }

  fmt.Println("required: ", opt.Required)
  fmt.Println("default: ", opt.Default)
}
```

运行程序，不传入 `default` 选项， `Default` 字段取默认值，不传入 `required` 选项，执行报错：

```
$ ./main.exe -r required-data
required:  required-data
default:  default

$ ./main.exe -d default-data -r required-data
required:  required-data
default:  default-data

$ ./main.exe
the required flag `/r, /required' was not specified
2020/01/09 18:07:39 Parse error:the required flag `/r, /required' was not specif
ied
```

高级特性

选项分组

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Basic GroupBasicOption `description:"basic type" group:"basic"`
  Slice GroupSliceOption `description:"slice of basic type" group:"slice"`
}

type GroupBasicOption struct {
  IntFlag    int     `short:"i" long:"intflag" description:"int flag"`
  BoolFlag   bool    `short:"b" long:"boolflag" description:"bool flag"`
  FloatFlag  float64 `short:"f" long:"floatflag" description:"float flag"`
  StringFlag string  `short:"s" long:"stringflag" description:"string flag"`
}

type GroupSliceOption struct {
```

```
  IntSlice        int           `long:"intslice" description:"int slice"`
  BoolSlice        bool         `long:"boolslice" description:"bool slice"`
  FloatSlice     float64     `long:"floatslice" description:"float slice"`
  StringSlice     string     `long:"stringslice" description:"string slice"`
}

func main() {
  var opt Option
  p := flags.NewParser(&opt, flags.Default)
  _, err := p.ParseArgs(os.Args[1:])
  if err != nil {
    log.Fatal("Parse error:", err)
  }

  basicGroup := p.Command.Group.Find("basic")
  for _, option := range basicGroup.Options() {
    fmt.Printf("name:%s value:%v\n", option.LongNameWithNamespace(), option.Valu
e())
  }

  sliceGroup := p.Command.Group.Find("slice")
  for _, option := range sliceGroup.Options() {
    fmt.Printf("name:%s value:%v\n", option.LongNameWithNamespace(), option.Valu
e())
  }
}
```

上面代码中我们将基本类型和它们的切片类型选项拆分到两个结构体中，这样可以使代码看起来更清晰自然，特别是在代码量很大的情况下。
这样做还有一个好处，我们试试用 `--help` 运行该程序：

```
$ ./main.exe --help
Usage:
  D:\code\golang\src\github.com\darjun\go-daily-lib\go-flags\group\main.exe [OPT
IONS]

basic:
  /i, /intflag:       int flag
  /b, /boolflag       bool flag
  /f, /floatflag:     float flag
  /s, /stringflag:    string flag

slice:
  /intslice:      int slice
  /boolslice      bool slice
```

```
  /floatslice:   float slice
  /stringslice:  string slice

Help Options:
  /?                   Show this help message
  /h, /help            Show this help message
```

输出的帮助信息中，也是按照我们设定的分组显示了，便于查看。

子命令

`go-flags` 支持子命令。我们经常使用的 Go 和 Git 命令行程序就有大量的子命令。例如 `go version`、`go build`、`go run`、`git status`、`git commit` 这些命令中 `version/build/run/status/commit` 就是子命令。
使用 `go-flags` 定义子命令比较简单：

```go
package main

import (
  "errors"
  "fmt"
  "log"
  "strconv"
  "strings"

  "github.com/jessevdk/go-flags"
)

type MathCommand struct {
  Op string `long:"op" description:"operation to execute"`
  Args []string
  Result int64
}

func (this *MathCommand) Execute(args []string) error {
  if this.Op != "+" && this.Op != "-" && this.Op != "x" && this.Op != "/" {
    return errors.New("invalid op")
  }

  for _, arg := range args {
    num, err := strconv.ParseInt(arg, 10, 64)
    if err != nil {
      return err
    }
```

```
    this.Result += num
  }

  this.Args = args
  return nil
}

type Option struct {
    Math MathCommand `command:"math"`
}

func main() {
    var opt Option
    _, err := flags.Parse(&opt)

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

    fmt.Printf("The result of %s is %d", strings.Join(opt.Math.Args, opt.Math.O
p), opt.Math.Result)
}
```

子命令必须实现 `go-flags` 定义的 `Commander` 接口：

```
type Commander interface {
    Execute(args []string) error
}
```

解析命令行时，如果遇到不是以 `-` 或 `--` 开头的参数， `go-flags` 会尝试将其解释为子命令名。子命令的名字通过在结构标签中使用 `command` 指定。
子命令后面的参数都将作为子命令的参数，子命令也可以有选项。

上面代码中，我们实现了一个可以计算任意个整数的加、减、乘、除子命令 `math` 。

接下来看看如何使用：

```
$ ./main.exe math --op + 1 2 3 4 5
The result of 1+2+3+4+5 is 15

$ ./main.exe math --op - 1 2 3 4 5
The result of 1-2-3-4-5 is -13

$ ./main.exe math --op x 1 2 3 4 5
The result of 1x2x3x4x5 is 120
```

```
$ ./main.exe math --op ÷ 120 2 3 4 5
The result of 120÷2÷3÷4÷5 is 1
```

注意，不能使用乘法符号 `*` 和除法符号 `/` ，它们都不可识别。

其他

`go-flags` 库还有很多有意思的特性，例如支持 Windows 选项格式（ `/v` 和 `/verbose` ）、从环境变量中读取默认值、从 ini 文件中读取默认设置等等。大家有兴趣可以自行去研究~

参考

1. go-flagsGithub 仓库
2. go-flagsGoDoc 文档

flag

缘起

我一直在想，有什么方式可以让人比较轻易地保持每日学习，持续输出的状态。写博客是一种方式，但不是每天都有想写的，值得写的东西。
有时候一个技术比较复杂，写博客的时候经常会写着写着发现自己的理解有偏差，或者细节还没有完全掌握，要去查资料，了解了之后又继续写，如此反复。
这样会导致一篇博客的耗时过长。

我在每天浏览思否、掘金和Github的过程中，发现一些比较好的想法，有**JS 每日一题**，**NodeJS 每日一库**，**每天一道面试题**等等等等。
https://github.com/parro-it/awesome-micro-npm-packages这个仓库收集 NodeJS 小型库，一天看一个不是梦！这也是我这个系列的灵感。
我计划每天学习一个 Go 语言的库，输出一篇介绍型的博文。每天一库当然是理想状态，我心中的预期是一周 3-5 个。

今天是第一天，我们从一个基础库聊起————Go 标准库中的 `flag`。

简介

`flag` 用于解析命令行选项。有过类 Unix 系统使用经验的童鞋对命令行选项应该不陌生。例如命令 `ls -al` 列出当前目录下所有文件和目录的详细信息，其中 `-al` 就是命令行选项。

命令行选项在实际开发中很常用，特别是在写工具的时候。

- 指定配置文件的路径，如 `redis-server ./redis.conf` 以当前目录下的配置文件 `redis.conf` 启动 Redis 服务器；
- 自定义某些参数，如 `python -m SimpleHTTPServer 8080` 启动一个 HTTP 服务器，监听 8080 端口。如果不指定，则默认监听 8000 端口。

快速使用

学习一个库的第一步当然是使用它。我们先看看 `flag` 库的基本使用：

```
package main

import (
```

```
    "fmt"
    "flag"
)

var (
    intflag int
    boolflag bool
    stringflag string
)

func init() {
    flag.IntVar(&intflag, "intflag", 0, "int flag value")
    flag.BoolVar(&boolflag, "boolflag", false, "bool flag value")
    flag.StringVar(&stringflag, "stringflag", "default", "string flag value")
}

func main() {
    flag.Parse()

    fmt.Println("int flag:", intflag)
    fmt.Println("bool flag:", boolflag)
    fmt.Println("string flag:", stringflag)
}
```

可以先编译程序，然后运行（我使用的是 Win10 + Git Bash）：

```
$ go build -o main.exe main.go
$ ./main.exe -intflag 12 -boolflag 1 -stringflag test
```

输出：

```
int flag: 12
bool flag: true
string flag: test
```

如果不设置某个选项，相应变量会取默认值：

```
$ ./main.exe -intflag 12 -boolflag 1
```

输出：

```
int flag: 12
bool flag: true
```

```
string flag: default
```

可以看到没有设置的选项 `stringflag` 为默认值 `default` 。

还可以直接使用 `go run` ，这个命令会先编译程序生成可执行文件，然后执行该文件，将命令行中的其它选项传给这个程序。

```
$ go run main.go -intflag 12 -boolflag 1
```

可以使用 `-h` 显示选项帮助信息：

```
$ ./main.exe -h
Usage of D:\code\golang\src\github.com\darjun\cmd\flag\main.exe:
  -boolflag
        bool flag value
  -intflag int
        int flag value
  -stringflag string
        string flag value (default "default")
```

总结一下，使用 `flag` 库的一般步骤：

- 定义一些全局变量存储选项的值，如这里的 `intflag/boolflag/stringflag` ；
- 在 `init` 方法中使用 `flag.TypeVar` 方法定义选项，这里的 `Type` 可以为基本类型 `Int/Uint/Float64/Bool` ，还可以是时间间隔 `time.Duration` 。定义时传入变量的地址、选项名、默认值和帮助信息；
- 在 `main` 方法中调用 `flag.Parse` 从 `os.Args[1:]` 中解析选项。因为 `os.Args[0]` 为可执行程序路径，会被剔除。

注意点：

`flag.Parse` 方法必须在所有选项都定义之后调用，且 `flag.Parse` 调用之后不能再定义选项。如果按照前面的步骤，基本不会出现问题。
因为 `init` 在所有代码之前执行，将选项定义都放在 `init` 中， `main` 函数中执行 `flag.Parse` 时所有选项都已经定义了。

选项格式

`flag` 库支持三种命令行选项格式。

```
-flag
-flag=x
-flag x
```

`-` 和 `--` 都可以使用，它们的作用是一样的。有些库使用 `-` 表示短选项，`--` 表示长选项。相对而言，`flag` 使用起来更简单。

第一种形式只支持布尔类型的选项，出现即为 `true` ，不出现为默认值。
第三种形式不支持布尔类型的选项。因为这种形式的布尔选项在类 Unix 系统中可能会出现意想不到的行为。看下面的命令：

```
cmd -x *
```

其中， `*` 是 shell 通配符。如果有名字为 0、false的文件，布尔选项 `-x` 将会取 `false` 。反之，布尔选项 `-x` 将会取 `true` 。而且这个选项消耗了一个参数。如果要显示设置一个布尔选项为 `false` ，只能使用 `-flag=false` 这种形式。

遇到第一个非选项参数（即不是以 `-` 和 `--` 开头的）或终止符 `--` ，解析停止。运行下面程序：

```
$ ./main.exe noflag -intflag 12
```

将会输出：

```
int flag: 0
bool flag: false
string flag: default
```

因为解析遇到 `noflag` 就停止了，后面的选项 `-intflag` 没有被解析到。所以所有选项都取的默认值。

运行下面的程序：

```
$ ./main.exe -intflag 12 -- -boolflag=true
```

将会输出：

```
int flag: 12
bool flag: false
string flag: default
```

首先解析了选项 `intflag` ，设置其值为 12。遇到 `--` 后解析终止了，后面的 `--boolflag=true` 没有被解析到，所以 `boolflag` 选项取默认值 `false` 。

解析终止之后如果还有命令行参数， `flag` 库会存储下来，通过 `flag.Args` 方法返回这些参数的切片。
可以通过 `flag.NArg` 方法获取未解析的参数数量， `flag.Arg(i)` 访问位置 `i` （从 0 开始）上的参数。
选项个数也可以通过调用 `flag.NFlag` 方法获取。

稍稍修改一下上面的程序：

```
func main() {
  flag.Parse()

  fmt.Println(flag.Args())
  fmt.Println("Non-Flag Argument Count:", flag.NArg())
  for i := 0; i < flag.NArg(); i++ {
    fmt.Printf("Argument %d: %s\n", i, flag.Arg(i))
  }

  fmt.Println("Flag Count:", flag.NFlag())
}
```

编译运行该程序：

```
$ go build -o main.exe main.go
$ ./main.exe -intflag 12 -- -stringflag test
```

输出：

```
[-stringflag test]
Non-Flag Argument Count: 2
Argument 0: -stringflag
Argument 1: test
```

解析遇到 `--` 终止后，剩余参数 `-stringflag test` 保存在 `flag` 中，可以通过 `Args/NArg/Arg` 等方法访问。

整数选项值可以接受 1234（十进制）、0664（八进制）和 0x1234（十六进制）的形式，并且可以是负数。实际上 `flag` 在内部使用 `strconv.ParseInt` 方法将字符串解析成 `int` 。
所以理论上， `ParseInt` 接受的格式都可以。

布尔类型的选项值可以为：

- 取值为 `true` 的：1、t、T、true、TRUE、True；
- 取值为 `false` 的：0、f、F、false、FALSE、False。

另一种定义选项的方式

上面我们介绍了使用 `flag.TypeVar` 定义选项，这种方式需要我们先定义变量，然后变量的地址。
还有一种方式，调用 `flag.Type` （其中 `Type` 可以为 `Int/Uint/Bool/Float64/String/Duration` 等）会自动为我们分配变量，返回该变量的地址。用法与前一种方式类似：

```
package main

import (
  "fmt"
  "flag"
)

var (
  intflag *int
  boolflag *bool
  stringflag *string
)

func init() {
  intflag = flag.Int("intflag", 0, "int flag value")
  boolflag = flag.Bool("boolflag", false, "bool flag value")
  stringflag = flag.String("stringflag", "default", "string flag value")
}

func main() {
  flag.Parse()

  fmt.Println("int flag:", *intflag)
  fmt.Println("bool flag:", *boolflag)
  fmt.Println("string flag:", *stringflag)
}
```

编译并运行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -intflag 12
```

将输出：

```
int flag: 12
bool flag: false
string flag: default
```

除了使用时需要解引用，其它与前一种方式基本相同。

高级用法

定义短选项

`flag` 库并没有显示支持短选项，但是可以通过给某个相同的变量设置不同的选项来实现。即两个选项共享同一个变量。
由于初始化顺序不确定，必须保证它们拥有相同的默认值。否则不传该选项时，行为是不确定的。

```
package main

import (
  "fmt"
  "flag"
)

var logLevel string

func init() {
  const (
    defaultLogLevel = "DEBUG"
    usage = "set log level value"
  )

  flag.StringVar(&logLevel, "log_type", defaultLogLevel, usage)
  flag.StringVar(&logLevel, "l", defaultLogLevel, usage + "(shorthand)")
}

func main() {
  flag.Parse()

  fmt.Println("log level:", logLevel)
}
```

编译、运行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -log_type WARNING
$ ./main.exe -l WARNING
```

使用长、短选项均输出：

```
log level: WARNING
```

不传入该选项，输出默认值：

```
$ ./main.exe
log level: DEBUG
```

解析时间间隔

除了能使用基本类型作为选项， `flag` 库还支持 `time.Duration` 类型，即时间间隔。时间间隔支持的格式非常之多，例如"300ms"、"-1.5h"、"2h45m"等等等等。时间单位可以是 ns/us/ms/s/m/h/day 等。实际上 `flag` 内部会调用 `time.ParseDuration` 。具体支持的格式可以参见time（需fq）库的文档。

```
package main

import (
  "flag"
  "fmt"
  "time"
)

var (
  period time.Duration
)

func init() {
  flag.DurationVar(&period, "period", 1*time.Second, "sleep period")
}

func main() {
  flag.Parse()
  fmt.Printf("Sleeping for %v...", period)
  time.Sleep(period)
  fmt.Println()
}
```

根据传入的命令行选项 `period` ，程序睡眠相应的时间，默认 1 秒。编译、运行程序：

```
$ go build -o main.exe main.go
$ ./main.exe
Sleeping for 1s...

$ ./main.exe -period 1m30s
Sleeping for 1m30s...
```

自定义选项

除了使用 `flag` 库提供的选项类型，我们还可以自定义选项类型。我们分析一下标准库中提供的案例：

```
package main

import (
  "errors"
  "flag"
  "fmt"
  "strings"
  "time"
)

type interval []time.Duration

func (i *interval) String() string {
  return fmt.Sprint(*i)
}

func (i *interval) Set(value string) error {
  if len(*i) > 0 {
    return errors.New("interval flag already set")
  }
  for _, dt := range strings.Split(value, ",") {
    duration, err := time.ParseDuration(dt)
    if err != nil {
      return err
    }
    *i = append(*i, duration)
  }
  return nil
}
```

```
var (
  intervalFlag interval
)

func init() {
  flag.Var(&intervalFlag, "deltaT", "comma-seperated list of intervals to use be
tween events")
}

func main() {
  flag.Parse()

  fmt.Println(intervalFlag)
}
```

首先定义一个新类型，这里定义类型 `interval` 。

新类型必须实现 `flag.Value` 接口：

```
// src/flag/flag.go
type Value interface {
  String() string
  Set(string) error
}
```

其中 `String` 方法格式化该类型的值， `flag.Parse` 方法在执行时遇到自定义类型的选项会将选项值作为参数调用该类型变量的 `Set` 方法。
这里将以 `,` 分隔的时间间隔解析出来存入一个切片中。

自定义类型选项的定义必须使用 `flag.Var` 方法。

编译、执行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -deltaT 30s
[30s]
$ ./main.exe -deltaT 30s,1m,1m30s
[30s 1m0s 1m30s]
```

如果指定的选项值非法， `Set` 方法返回一个 `error` 类型的值， `Parse` 执行终止，打印错误和使用帮助。

```
$ ./main.exe -deltaT 30x
invalid value "30x" for flag -deltaT: time: unknown unit x in duration 30x
```

```
Usage of D:\code\golang\src\github.com\darjun\go-daily-lib\flag\self-defined\mai
n.exe:
  -deltaT value
        comma-seperated list of intervals to use between events
```

解析程序中的字符串

有时候选项并不是通过命令行传递的。例如，从配置表中读取或程序生成的。这时候可以使用 `flag.FlagSet` 结构的相关方法来解析这些选项。

实际上，我们前面调用的 `flag` 库的方法，都会间接调用 `FlagSet` 结构的方法。 `flag` 库中定义了一个 `FlagSet` 类型的全局变量 `CommandLine` 专门用于解析命令行选项。
前面调用的 `flag` 库的方法只是为了提供便利，它们内部都是调用的 `CommandLine` 的相应方法：

```
// src/flag/flag.go
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)

func Parse() {
  CommandLine.Parse(os.Args[1:])
}

func IntVar(p *int, name string, value int, usage string) {
  CommandLine.Var(newIntValue(value, p), name, usage)
}

func Int(name string, value int, usage string) *int {
  return CommandLine.Int(name, value, usage)
}

func NFlag() int { return len(CommandLine.actual) }

func Arg(i int) string {
  return CommandLine.Arg(i)
}

func NArg() int { return len(CommandLine.args) }
```

同样的，我们也可以自己创建 `FlagSet` 类型变量来解析选项。

```
package main

import (
```

```
    "flag"
    "fmt"
)

func main() {
    args := []string{"-intflag", "12", "-stringflag", "test"}

    var intflag int
    var boolflag bool
    var stringflag string

    fs := flag.NewFlagSet("MyFlagSet", flag.ContinueOnError)
    fs.IntVar(&intflag, "intflag", 0, "int flag value")
    fs.BoolVar(&boolflag, "boolflag", false, "bool flag value")
    fs.StringVar(&stringflag, "stringflag", "default", "string flag value")

    fs.Parse(args)

    fmt.Println("int flag:", intflag)
    fmt.Println("bool flag:", boolflag)
    fmt.Println("string flag:", stringflag)
}
```

`NewFlagSet` 方法有两个参数，第一个参数是程序名称，输出帮助或出错时会显示该信息。第二个参数是解析出错时如何处理，有几个选项：

- `ContinueOnError`：发生错误后继续解析，`CommandLine` 就是使用这个选项；
- `ExitOnError`：出错时调用 `os.Exit(2)` 退出程序；
- `PanicOnError`：出错时产生 panic。

随便看一眼 `flag` 库中的相关代码：

```
// src/flag/flag.go
func (f *FlagSet) Parse(arguments []string) error {
    f.parsed = true
    f.args = arguments
    for {
        seen, err := f.parseOne()
        if seen {
            continue
        }
        if err == nil {
            break
        }
```

```
    switch f.errorHandling {
    case ContinueOnError:
      return err
    case ExitOnError:
      os.Exit(2)
    case PanicOnError:
      panic(err)
    }
  }
  return nil
}
```

与直接使用 `flag` 库的方法有一点不同， `FlagSet` 调用 `Parse` 方法时需要显示传入字符串切片作为参数。因为 `flag.Parse` 在内部调用了 `CommandLine.Parse(os.Args[1:])` 。
示例代码都放在GitHub上了。

参考

1. flag库文档