目　录

前言

本文转自：https://www.kancloud.cn/mutouzhang/gocookbook/608898

Go的标准库向来备受赞誉。

以标准库文档来说，简洁，清晰，明了。

但读标准库只是第一步，117个包涵盖面实在广泛，而且读起来太容易瞌睡。学习他人对标准库的使用方法无疑是提高技术的最佳途径。

成体系的书，目前国内对标准库解读最好的徐新华的《Go语言标准库》，但遗憾的是该书自2013年4月30日开始立项，截至2016年6月发布了继续更新的通知，到2018-05-04依然没有完成。如此巨大的时间跨度，Go已经从1.1版本升级到了1.10.1，2.0指日可待，标准库已细节性调整多次，实在令人惋惜。

除了本书，老外还出了另一本Go Cookbook，全名是《Go Standard Libray CookBook》发售于2018年2月27日，这两本书出版时间相差8个月。

本书并无对基本语法的讲解，此外原作者对于命名和代码风格并未进行仔细斟酌，章节起名也比较随意，在阅读代码时可能带来疑惑，不过影响不大。该书非常适合有其他语言开发经验，并对Go语言基本语法有所了解的开发者阅读。

coding道路既艰且险，愿与诸君共勉。
于2018-05-04。

I/O和文件系统

本章会覆盖以下内容：

- 常见的 I/O 接口
- 使用bytes和strings包
- 操作文件夹和文件
- 使用CSV格式化数据
- 操作临时文件
- 使用 text/template和HTML/templates包

介绍

Go对 I/O 操作提供了极好的支持。本章将讨论常用的 I/O 接口并展示如何使用它们。Go标准库经常使用这些接口，并且对这些接口使用将贯穿本书。通过熟悉这些接口和包，可以帮助你以抽象和灵活的方式构建数据。

常见 I/O 接口

常见的 I/O 接口

Go提供了一系列的 I/O 接口并经常在标准库源码中使用它们。尽可能使用这些接口而非传递结构或其它类型是Go推荐的方式(*不必恐惧，Go中的接口并不像Java那样浩如烟海，你喝杯咖啡的功夫就能把这几个为数不多的接口翻来覆去的看很多遍*)。其中，当属 io.Reader 和 io.Writer 这两个接口最为常用。在标准库源码中处处有他们俩的身影，熟悉并了解他们对于日常工作非常必要。

Reader 和 Writer 接口的定义是这样的：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

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

在Go中可以轻松的组合接口，看看下面的代码：

```
type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}
type ReadSeeker interface {
    Reader
    Seeker
}
```

这种使用方式你同样可以在 io 包的 Pipe() 中看到：

```
func Pipe() (*PipeReader, *PipeWriter)
```

接下来让我们看看该如何使用。

准备

根据以下步骤配置你的开发环境：

1. 从Go国内官网下载并安装Go到你的操作系统中，配置GOPATH环境变量。

2. 打开命令行，跳转到你的 GOPATH/src 文件夹内，建立项目文件夹，例如 $GOPATH/src/github.com/yourusername/customrepo。所有的代码会在该目录下进行修改。
3. 或者，你可以通过以下名直接安装本书示例代码：go get github.com/agtorre/go-cookbook/

实践

以下步骤包括编写和运行程序：

1. 在你的命令行中建立新的文件夹，名为chapter1/interfaces。
2. 跳转到该文件夹。
3. 建立interfaces.go：

```
package interfaces

import (
    "fmt"
    "io"
    "os"
)

// Copy直接将数据从in复制到out
// 它使用了buffer作为缓存
// 同时会将数据复制到Stdout
func Copy(in io.ReadSeeker, out io.Writer) error {
    // 我们将结果写入 out 和 Stdout
    w := io.MultiWriter(out, os.Stdout)

    // 标准的 io.Copy 调用,如果传入的数据 in 过大会很危险
    // 第一次向w复制数据
    if _, err := io.Copy(w, in); err != nil {
        return err
    }

    in.Seek(0, 0)

    // 使用64字节作为缓存写入 w
    // 第二次向w复制数据
    buf := make([]byte, 64)
    if _, err := io.CopyBuffer(w, in, buf); err != nil {
        return err
    }

    // 打印换行
```

```
    fmt.Println()

    return nil
}
```

4.建立pipes.go

```
package interfaces

import (
    "io"
    "os"
)

// PipeExample展现了对 io 接口的更多用法
func PipeExample() error {
    // io.Pipe()返回的r和w分别实现了io.Reader 和 io.Writer接口
    r, w := io.Pipe()

    // 在w进行写入的时候，r会发生阻塞
    go func() {
        // 这里我们简单的写入字符串，同样也可以写入json或base64编码的其他东西
        w.Write([]byte("test\n"))
        w.Close()
    }()

    if _, err := io.Copy(os.Stdout, r); err != nil {
        return err
    }
    return nil
}
```

5.建立example文件夹，在文件夹内建立main.go：

```
package main

import (
    "bytes"
    "fmt"
    "github.com/agtorre/go-cookbook/chapter1//interfaces"
)

func main() {
    in := bytes.NewReader([]byte("example"))
    out := &bytes.Buffer{}
```

```
    fmt.Print("stdout on Copy = ")
    if err := interfaces.Copy(in, out); err != nil {
        panic(err)
    }
    fmt.Println("out bytes buffer =", out.String())
    fmt.Print("stdout on PipeExample = ")
    if err := interfaces.PipeExample(); err != nil {
        panic(err)
    }
}
```

6.执行 go run main.go

7.这会输出：

```
stdout on Copy = exampleexample
out bytes buffer = exampleexample
stdout on PipeExample = test
```

8.如果复制或编写了自己的测试，请跳转至上一个目录并运行测试，并确保所有测试都通过。

说明

Copy函数在接口之间进行复制并将它们视为流。 将数据视为流的方式有很多实际用途，特别是在处理网络或文件系统时。Copy函数还创建了一个复合写入器，它将两个写入器流组合起来，并使用ReadSeeker写入两次。还有一个使用缓冲写入的例子，如果你的内存不满足数据流大小，那么使用它会很合适。

PipeReader和PipeWriter结构实现了io.Reader和io.Writer接口。他们连接在一起，从属于一个内存管道。管道的主要目的是从流中读取数据，同时从同一个数据流写入不同的数据源。实质上，它将两个流合并为一个管道。

Go接口使用了清晰的抽象来包装常见操作的数据。这使 I/O 操作变得很明显，所以使用io包学习接口组合很不错。 pip包通常使用较少，但在连接输入和输出流时提供了极高的灵活性和线程安全性。

使用bytes和strings包

bytes 和 string 包有许多有用的函数以帮助使用者在字符串和字节类型之间进行处理和转换。这些函数可用于创建多种通用 I/O 接口的缓冲区。

准备

请参阅上一节，关于使用常见I/O接口的准备步骤。

实践

1.创建buffer.go：

```
package bytestrings

import (
    "bytes"
    "io"
    "io/ioutil"
)

// Buffer 演示了初始化字节缓冲区的一些技巧
// 实现了 io.Reader 接口
func Buffer(rawString string) *bytes.Buffer {

    // 将传入的字符串转换为字节数组
    rawBytes := []byte(rawString)

    // 有很多方式使用字节数组或原始字符串建立缓冲区
    var b = new(bytes.Buffer)
    b.Write(rawBytes)

    // 或者
    b = bytes.NewBuffer(rawBytes)

    // 使用字符串建立字节数组
    b = bytes.NewBufferString(rawString)

    return b
}

// ToString接收 io.Reader 并将其转换为字符串返回
func toString(r io.Reader) (string, error) {
    b, err := ioutil.ReadAll(r)
```

```
    if err != nil {
        return "", err
    }
    return string(b), nil
}
```

2.建立bytes.go，Golang学习 - bufio 包对buffio解读很完整，值得一读：

```
package bytestrings

import (
    "bufio"
    "bytes"
    "fmt"
)

// WorkWithBuffer 会使用创建自 Buffer 函数的字节缓冲区
func WorkWithBuffer() error {
    rawString := "it's easy to encode unicode into a byte array ❤"

    b := Buffer(rawString)

    // 使用b.Bytes()可以快速从字节缓冲区获取字节切片
    // 使用b.String()可以快速从字节缓冲区获取字符串
    fmt.Println(b.String())


    // 由于*bytes.Buffer类型的b实现了io Reader 我们可以使用常见的reader函数
    s, err := toString(b)
    if err != nil {
        return err
    }
    fmt.Println(s)

    // 可以创建一个 bytes reader 它实现了
    // io.Reader, io.ReaderAt,
    // io.WriterTo, io.Seeker, io.ByteScanner, and io.RuneScanner
    // 接口
    reader := bytes.NewReader([]byte(rawString))

    // 我们可以使用其创建 scanner 以允许使用缓存读取和建立 token
    scanner := bufio.NewScanner(reader)
    scanner.Split(bufio.ScanWords)

    // 遍历所有的扫描token
```

```
    for scanner.Scan() {
        fmt.Print(scanner.Text())
    }

    return nil
}
```

3. 建立 string.go：

```
package bytestrings

import (
    "fmt"
    "io"
    "os"
    "strings"
)

// SearchString 展示了一系列在字符串中进行查询的方法
func SearchString() {
    s := "this is a test"

    // 返回 true 表明包含子串
    fmt.Println(strings.Contains(s, "this"))

    // 返回 true 表明包含子串中的任何一字符a或b或c
    fmt.Println(strings.ContainsAny(s, "abc"))

    // 返回 true 表明以该子串开头
    fmt.Println(strings.HasPrefix(s, "this"))

    // 返回 true 表明以该子串结尾
    fmt.Println(strings.HasSuffix(s, "test"))
}

// ModifyString 展示了一系列修改字符串的方法
func ModifyString() {
    s := "simple string"

    // 输出 [simple string]
    fmt.Println(strings.Split(s, " "))

    // 输出 "Simple String"
    fmt.Println(strings.Title(s))
```

```
    // 输出 "simple string" 会移除头部和尾部的空白
    s = " simple string "
    fmt.Println(strings.TrimSpace(s))
}

// StringReader 演示了如何快速创建一个字符串的io.Reader接口
func StringReader() {
    s := "simple string\n"
    r := strings.NewReader(s)

    // 在标准输出上打印 s
    io.Copy(os.Stdout, r)
}
```

4.建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter1/bytestrings"

func main() {
    err := bytestrings.WorkWithBuffer()
    if err != nil {
        panic(err)
    }

    // each of these print to stdout
    bytestrings.SearchString()
    bytestrings.ModifyString()
    bytestrings.StringReader()
}
```

5.执行 go run main.go

6.这会输出(*原文此处将心形显示为乱码，不过在译者的win7 Go1.10.1环境下可以完整显示出来*)：

```
it's easy to encode unicode into a byte array ❤
it's easy to encode unicode into a byte array ❤
it'seasytoencodeunicodeintoabytearray❤true
true
true
true
[simple string]
Simple String
```

```
simple string
simple string
```

说明

bytes 包在处理数据时提供了许多便利功能。 例如，使用流处理库或方法时，缓冲区比字节数组更灵活。一旦你创建了一个缓冲区，它可以用来满足io.Reader接口，所以你可以利用ioutil包中的各种函数来操作数据。 对于流应用，你可能需要使用buffer 和 scanner，bufio包可以在这些情况下派上用场。 有时，使用数组或切片更适合较小的数据集，或者当你的计算机上有大量内存，无需考虑过多考虑时。

Go为这些基本类型的接口之间的转换提供了很大的灵活性——在字符串和字节之间转换相对简单。使用字符串时，string包提供了许多方便的函数来搜索和处理字符串。在某些情况下，一个好的正则表达式可能是合适的，但大多数情况下，string和strconv包就足够了。string包允许你将字符串标题化，将其拆分为数组或修剪空白，它还提供了一个可以用来代替bytes包读取器类型的Reader接口。

操作文件夹和文件

Go提供了os和 ioutils包来对文件和文件夹进行操作，值得高兴的是，它们是跨平台的，无需对操作系统做过多考虑。

实践

1.建立 dirs.go ：

```
package filedirs

import (
    "errors"
    "io"
    "os"
)

// 操纵文件和目录
func Operate() error {
    // 文件权限0755类似于你在命令行中使用的chown，
    // 这将创建一个目录 /tmp/example，
    // 你也可以使用绝对路径而不是相对路径
    if err := os.Mkdir("example_dir", os.FileMode(0755)); err != nil {
        return err
    }

    // 跳转到 /tmp 目录
    if err := os.Chdir("example_dir"); err != nil {
        return err
    }

    // f是一个通用的文件对象 它还实现了多个接口，并且如果在打开时设置了正确的方式，则可以用作读取器或写入器
    f, err := os.Create("test.txt")
    if err != nil {
        return err
    }

    // 向文件写入长度已知的数据 并确认写入成功
    value := []byte("hello\n")
    count, err := f.Write(value)
    if err != nil {
        return err
    }
```

```
    if count != len(value) {
        return errors.New("incorrect length returned from write")
    }

    if err := f.Close(); err != nil {
        return err
    }

    // 读取文件
    f, err = os.Open("test.txt")
    if err != nil {
        return err
    }

    io.Copy(os.Stdout, f)

    if err := f.Close(); err != nil {
        return err
    }

    // 跳转到 /tmp 文件夹
    if err := os.Chdir(".."); err != nil {
        return err
    }

    // 删除建立的文件夹
    // os.RemoveAll如果传递了错误的文件夹路径会返回错误
    if err := os.RemoveAll("example_dir"); err != nil {
        return err
    }

    return nil
}
```

2.建立bytes.go：

```
package filedirs

import (
    "bytes"
    "io"
    "os"
    "strings"
)
```

```
// 读取并转换为大写后复制内容到目标文件
func Capitalizer(f1 *os.File, f2 *os.File) error {
    if _, err := f1.Seek(0, 0); err != nil {
        return err
    }

    var tmp = new(bytes.Buffer)

    if _, err := io.Copy(tmp, f1); err != nil {
        return err
    }

    s := strings.ToUpper(tmp.String())

    if _, err := io.Copy(f2, strings.NewReader(s)); err != nil {
        return err
    }
    return nil
}

// 建立两个文件 将其中一个的内容转换为大写复制给另一个
func CapitalizerExample() error {
    f1, err := os.Create("file1.txt")
    if err != nil {
        return err
    }

    if _, err := f1.Write([]byte(` this file contains a number of words and new
 lines`)); err != nil {
        return err
    }

    f2, err := os.Create("file2.txt")
    if err != nil {
        return err
    }

    if err := Capitalizer(f1, f2); err != nil {
        return err
    }

    if err := os.Remove("file1.txt"); err != nil {
        return err
    }

    if err := os.Remove("file2.txt"); err != nil {
```

```
        return err
    }

    return nil
}
```

3.建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter1/filedirs"

func main() {
    if err := filedirs.Operate(); err != nil {
        panic(err)
    }

    if err := filedirs.CapitalizerExample(); err != nil {
        panic(err)
    }
}
```

4.运行main.go，这会输出：

```
hello
```

示例在运行中会分别建立file1.txt和file2.txt然后删掉，file1.txt中包含：

```
this file contains a number of words and new lines
```

file2.txt中包含：

```
THIS FILE CONTAINS A NUMBER OF WORDS AND NEW LINES
```

说明

如果你熟悉Unix下的文件系统，那么Go的os包一定看起来很亲切。你可以使用该包进行基本的文件操作——例如查看文件状态，修改文件权限，操作文件及文件夹等。在示例中我们对目录和文件进行了操作，并在操作结束后对其进行删除。

操作文件对象与操作内存中的流数据非常相似。os.File还提供了许多便利的函数，例如Chown，Stat和Truncate。 想要熟悉os.File的各种函数最简单方法就是使用它们。在所有的

示例中，我们都会小心的清理程序中额外产生的临时文件，在日常工作中这也是个很好的习惯。

当你读取文件并将其存储在File结构之后，就可以轻松地将其传递给前面讨论的多个 I/O 接口。在之前示例中，我们看到的对缓冲区和内存数据的操作都可以直接替换为文件对象。因此，对于诸如将日志写入stderr以及对单个写入调用同时将其写入文件之类的操作会很有用。

使用CSV格式化数据

CSV是操作数据的常用格式。将CSV文件导入或导出到Excel中亦是很常见的操作。Go的CSV包提供了数据操作接口，因此可以轻松地将数据写入缓冲区，标准输出，文件或socket。本节将展示将数据导入和导出CSV格式的一些常用方法。

实践

1.建立read_csv.go：

```
package csvformat

import (
    "bytes"
    "encoding/csv"
    "fmt"
    "io"
    "strconv"
)

// Movie用来存储CSV解析后的内容
type Movie struct {
    Title    string
    Director string
    Year     int
}

// ReadCSV 展示了如何处理CSV
// 接收的参数通过io.Reader传入
func ReadCSV(b io.Reader) ([]Movie, error) {

    //返回的是csv.Reader
    r := csv.NewReader(b)

    // 分隔符和注释是csv.Reader结构体中的字段
    r.Comma = ';'
    r.Comment = '-'

    var movies []Movie

    // 读取并返回一个字符串切片和错误信息
    // 我们也可以将其用于字典键或其他形式的查找
    // 此处忽略了返回的切片 目的是跳过csv首行标题
    _, err := r.Read()
```

```
    if err != nil && err != io.EOF {
        return nil, err
    }

    // 循环直到全部处理完毕
    for {
        record, err := r.Read()
        if err == io.EOF {
            break
        } else if err != nil {
            return nil, err
        }

        year, err := strconv.ParseInt(record[2], 10, 64)
        if err != nil {
            return nil, err
        }

        m := Movie{record[0], record[1], int(year)}
        movies = append(movies, m)
    }
    return movies, nil
}

// AddMoviesFromText 将字符串按 CSV 格式解析
func AddMoviesFromText() error {

    in := `
- first our headers
movie title;director;year released

- then some data
Guardians of the Galaxy Vol. 2;James Gunn;2017
Star Wars: Episode VIII;Rian Johnson;2017
`

    b := bytes.NewBufferString(in)
    m, err := ReadCSV(b)
    if err != nil {
        return err
    }
    fmt.Printf("%#v\n", m)
    return nil
}
```

2.建立 write_csv.go：

```
package csvformat

import (
    "bytes"
    "encoding/csv"
    "io"
    "os"
)

// 结构体Book有Author和Title两个字段
type Book struct {
    Author string
    Title  string
}

// Books是Book的切片类型
type Books []Book

// ToCSV将Books写入传进来的 io.Writer
// 返回任何可能发生的错误
func (books *Books) ToCSV(w io.Writer) error {
    n := csv.NewWriter(w)
    err := n.Write([]string{"Author", "Title"})
    if err != nil {
        return err
    }
    for _, book := range *books {
        err := n.Write([]string{book.Author, book.Title})
        if err != nil {
            return err
        }
    }

    n.Flush()
    return n.Error()
}

// WriteCSVOutput 初始化Books并调用ToCSV
// 并写入到标准输出
func WriteCSVOutput() error {
    b := Books{
        Book{
            Author: "F Scott Fitzgerald",
            Title:  "The Great Gatsby",
        },
```

```
        Book{
            Author: "J D Salinger",
            Title:  "The Catcher in the Rye",
        },
    }

    return b.ToCSV(os.Stdout)
}

// WriteCSVBuffer 初始化Books并调用ToCSV
// 并写入到bytes.Buffers
func WriteCSVBuffer() (*bytes.Buffer, error) {
    b := Books{
        Book{
            Author: "F Scott Fitzgerald",
            Title:  "The Great Gatsby",
        },
        Book{
            Author: "J D Salinger",
            Title:  "The Catcher in the Rye",
        },
    }

    w := &bytes.Buffer{}
    err := b.ToCSV(w)
    return w, err
}
```

3.建立main.go：

```
import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter1/csvformat"
)

func main() {
    if err := csvformat.AddMoviesFromText(); err != nil {
        panic(err)
    }

    if err := csvformat.WriteCSVOutput(); err != nil {
        panic(err)
    }
```

```
    buffer, err := csvformat.WriteCSVBuffer()
    if err != nil {
        panic(err)
    }

    fmt.Println("Buffer = ", buffer.String())
}
```

4.运行main.go，这会输出：

```
[]csvformat.Movie{csvformat.Movie{Title:"Guardians of the Galaxy Vol. 2", Direct
or:"James Gunn", Year:2017}, csvformat.Movie{Title:"Star Wars: Episode VIII", Di
rector:"Rian Johnson", Year:2017}}
Author,Title
F Scott Fitzgerald,The Great Gatsby
J D Salinger,The Catcher in the Rye
Buffer =  Author,Title
F Scott Fitzgerald,The Great Gatsby
J D Salinger,The Catcher in the Rye
```

说明

为了探索CSV格式操作，我们首先将数据表示为结构。在Go中将数据格式化为结构非常有用，这会使目标封装和编码等变得相对简单。 我们的示例中使用自定义的Movie结构。它所属的函数接收io.Reader接口作为参数以使CSV数据输入时更加灵活——文件或缓冲区都可以输入进来。接下来，我们使用传入的数据来创建并填充Movie结构，我们还向CSV解析器添加了；作为分隔符 - 作为注释行。

接下来，我们以类似的方式探索CSV格式的写入操作，我们初始化一系列Books，然后将CSV格式的特定book写入io.Writer接口，同样的，目标可以是文件，标准输出或缓冲区。

CSV包是一个很棒的例子，它说明了为什么你需要考虑要将Go中的数据流视为实现通用接口。通过细微的调整，我们可以轻松更改数据的来源和目的地，同时不影响操作CSV数据，并且无需使用过多的内存或时间。例如，可以一次从一个数据流中读取一个记录，并以循环的形式将修改的格式写入单独的流。这样做不会占用大量内存或处理器。

稍后，在探索数据管道和工作池时，你将看到如何组合这些想法以及如何并行处理这些流。

操作临时文件

在前面的章节中，我们已经了解了基本的文件操作。当我们遇到需要手动清理，名称冲突等情况时，使用临时文件是更快，更简单的方法。

实践

1.建立temp_files.go：

```
package tempfiles

import (
    "fmt"
    "io/ioutil"
    "os"
)

// 这里展示了临时文件操作
func WorkWithTemp() error {
    // 如果你需要一个临时文件夹，存贮类似与template1-10.html这样的文件
    // 首个参数使用空字符串，意味着会在默认的临时目录中创建以后一个参数为开头名称的文件夹
    // 该函数实际调用了os.TempDir()
    t, err := ioutil.TempDir("", "tmp")
    if err != nil {
        return err
    }

    // 这会在整个操作完成后移除该临时文件夹及其中的所有文件
    defer os.RemoveAll(t)

    // 文件夹t必须存在否则将返回错误
    // tf是*os.File类型
    tf, err := ioutil.TempFile(t, "tmp")
    if err != nil {
        return err
    }

    fmt.Println(tf.Name())

    // 通常情况下我们在函数的最后部分删除临时文件
    // 不过通过前面的defer已经完成了这个任务
```

```
    return nil
}
```

2.建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter1/tempfiles"

func main() {
    if err := tempfiles.WorkWithTemp(); err != nil {
        panic(err)
    }
}
```

3.运行main.go，这会输出：

```
C:\Users\ADMINI~1\AppData\Local\Temp\tmp207945363\tmp063401686
```

说明

可以使用ioutil包创建临时文件和目录。 虽然你仍然需要自己删除文件，但RemoveAll会让这个操作变得非常方便。
在编写测试时，强烈建议使用临时文件。默认情况下，Go的ioutil包将尝试遵循操作系统的默认配置（或环境变量）。

使用 text/template和HTML/templates包

Go为模板操作提供了丰富的支持。嵌套模板，导入函数，表示变量，迭代数据等等都很简单。如果需要比CSV数据格式更复杂的东西，模板可能是一个不错的解决方案。

模板的另一个应用是网站的页面渲染。当我们想要将服务器端数据呈现给客户端时，模板可以很好地满足要求。起初，Go模板可能会让人感到困惑。 本章将探讨如何使用模板。

实践

1.建立templates.go：

```
package templates

import (
    "os"
    "strings"
    "text/template"
)

const sampleTemplate = `
    This template demonstrates printing a {{ .Variable | printf "%#v" }}.

    {{if .Condition}}
    If condition is set, we'll print this
    {{else}}
    Otherwise, we'll print this instead
    {{end}}

    Next we'll iterate over an array of strings:
    {{range $index, $item := .Items}}
        {{$index}}: {{$item}}
    {{end}}

    We can also easily import other functions like strings.Split
    then immediately used the array created as a result:
    {{ range $index, $item := split .Words ","}}
        {{$index}}: {{$item}}
    {{end}}

    Blocks are a way to embed templates into one another
    {{ block "block_example" .}}
        No Block defined!
    {{end}}
```

```
    {{/*
        This is a way
        to insert a multi-line comment
    */}}
`

const secondTemplate = `
    {{ define "block_example" }}
        {{.OtherVariable}}
    {{end}}
`

// RunTemplate初始化模板并展示了对模板的基本操作
func RunTemplate() error {
    data := struct {
        Condition     bool
        Variable      string
        Items         []string
        Words         string
        OtherVariable string
    }{
        Condition:     true,
        Variable:      "variable",
        Items:         []string{"item1", "item2", "item3"},
        Words:         "another_item1,another_item2,another_item3",
        OtherVariable: "I'm defined in a second template!",
    }

    funcmap := template.FuncMap{
        "split": strings.Split,
    }

    // 这里可以链式调用
    t := template.New("example")
    t = t.Funcs(funcmap)

    // 我们可以使用Must来替代它
    // template.Must(t.Parse(sampleTemplate))
    t, err := t.Parse(sampleTemplate)
    if err != nil {
        return err
    }

    // 为了模拟长时间操作我们通过克隆创建另一个模板 然后解析它
```

```
    t2, err := t.Clone()
    if err != nil {
        return err
    }

    t2, err = t2.Parse(secondTemplate)
    if err != nil {
        return err
    }

    // 将数据填充后的模板写入标准输出
    err = t2.Execute(os.Stdout, &data)
    if err != nil {
        return err
    }

    return nil
}
```

2. 建立template_files.go：

```
package templates

import (
    "io/ioutil"
    "os"
    "path/filepath"
    "text/template"
)

//CreateTemplate会创建一个包含数据的模板文件
func CreateTemplate(path string, data string) error {
    return ioutil.WriteFile(path, []byte(data), os.FileMode(0755))
}

// InitTemplates在文件夹中设置一系列的模板文件
func InitTemplates() error {

    tempdir, err := ioutil.TempDir("", "temp")
    if err != nil {
        return err
    }

    // 在操作完成后全部都将被删除
    defer os.RemoveAll(tempdir)
```

```
    err = CreateTemplate(filepath.Join(tempdir, "t1.tmpl"), `
        Template 1! {{ .Var1 }}
        {{ block "template2" .}} {{end}}
        {{ block "template3" .}} {{end}}
    `)
    if err != nil {
        return err
    }

    err = CreateTemplate(filepath.Join(tempdir, "t2.tmpl"), `
        {{ define "template2"}}Template 2! {{ .Var2 }}{{end}}
    `)
    if err != nil {
        return err
    }

    err = CreateTemplate(filepath.Join(tempdir, "t3.tmpl"), `
        {{ define "template3"}}Template 3! {{ .Var3 }}{{end}}
    `)
    if err != nil {
        return err
    }

    pattern := filepath.Join(tempdir, "*.tmpl")

    // 组合所有匹配glob的文件并将它们组合成一个模板
    tmpl, err := template.ParseGlob(pattern)
    if err != nil {
        return err
    }

    // Execute函数可以运用于map和struct
    tmpl.Execute(os.Stdout, map[string]string{
        "Var1": "Var1!!",
        "Var2": "Var2!!",
        "Var3": "Var3!!",
    })

    return nil
}
```

3.建立 html_templates.go：

```
package templates

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

// HTMLDifferences 展示了html/template 和 text/template的一些不同
func HTMLDifferences() error {
    t := template.New("html")
    t, err := t.Parse("<h1>Hello! {{.Name}}</h1>\n")
    if err != nil {
        return err
    }

    // html/template自动转义不安全的操作，比如javascript注入
    // 这会根据变量的位置不同而呈现不完全相同的结果
    err = t.Execute(os.Stdout, map[string]string{"Name": "<script>alert('Can you see me?')</script>"})
    if err != nil {
        return err
    }

    // 你也可以手动调用转义器
    fmt.Println(template.JSEscaper(`example <example@example.com>`))
    fmt.Println(template.HTMLEscaper(`example <example@example.com>`))
    fmt.Println(template.URLQueryEscaper(`example <example@example.com>`))

    return nil
}
```

4.以上示例打印显示较长，大家可自行运行查看。

说明

Go有两个模板包 - text/template和html/template。这两个包的部分函数看起来非常相似，实际功能也确实如此。通常，使用html/template来呈现网站。模板是纯文本，但变量和函数可以在大括号块内使用。

模板包还提供了处理文件的便捷方法。示例在临时目录中创建了许多模板，然后使用一行代码读取所有模板。

html/template包是对text/template包的包装。所有模板示例都对html/template包同样适用，除了import语句无需其他任何修改。HTML模板提供了上下文感知安全性的额外好处。这

可以防止诸如JavaScript注入之类的事情。

模板包能够满足你对页面操作的期望。在向HTML和JavaScript发布结果时，你可以轻松组合模板，添加应用程序逻辑并确保安全性。

命令行工具

本章会覆盖以下内容：

- 解析命令行flag标识
- 解析命令行参数
- 读取和设置环境变量
- 操作TOML，YAML和JSON配置文件
- 操做Unix系统下的pipe管道
- 处理信号量
- ANSI命令行着色

介绍

命令行是处理用户输入和输出的最简单的方式之一。本章将重点介绍基于命令行的交互，例如命令行参数，配置和环境变量。 本章将以一个用于在Unix和Bash for Windows中着色文本输出的库结束。

通过本章中的介绍，你将获得处理预期和意外用户输入的能力。以及学习如何处理用户可能向应用程序发送的意外信号，与标志或命令行参数相比，管道是获取用户输入的良好替代方案。

ANSI文本着色有望提供一些清理用户输出的示例。例如，在日志记录中，用户能够根据其目的为文本着色，或使大块文本更加清晰。

解析命令行flag标识

使用flag包可以轻松地将命令行标识参数添加到Go应用程序。它有一些缺点——你往往需要复制很多代码以添加简写版本的标志，并且它们是按帮助提示的字母顺序排序的。 有许多第三方库试图解决这些缺点，但本章将重点关注标准库版本而不是那些第三方库。

实践

1.创建flags.go：

```
package main

import (
    "flag"
    "fmt"
)

// Config存储接收到的标识
type Config struct {
    subject      string
    isAwesome    bool
    howAwesome   int
    countTheWays CountTheWays
}

// Setup 根据传入的标识初始化配置
func (c *Config) Setup() {
    // 你可以使用这样的方式直接初始化标识:
    // var someVar = flag.String("flag_name", "default_val", "description")
    // 但在实际操作中使用结构体来承载会更好一些

    // 完整版
    flag.StringVar(&c.subject, "subject", "", "subject is a string, it defaults to empty")
    // 简写版
    flag.StringVar(&c.subject, "s", "", "subject is a string, it defaults to empty (shorthand)")

    flag.BoolVar(&c.isAwesome, "isawesome", false, "is it awesome or what?")
    flag.IntVar(&c.howAwesome, "howawesome", 10, "how awesome out of 10?")

    // 自定义变量类型
    flag.Var(&c.countTheWays, "c", "comma separated list of integers")
}
```

```
// GetMessage 将所有的内部字段拼接成完整的句子
func (c *Config) GetMessage() string {
    msg := c.subject
    if c.isAwesome {
        msg += " is awesome"
    } else {
        msg += " is NOT awesome"
    }

    msg = fmt.Sprintf("%s with a certainty of %d out of 10. Let me count the ways %s", msg, c.howAwesome, c.countTheWays.String())
    return msg
}
```

2.建立custom.go：

```
package main

import (
    "fmt"
    "strconv"
    "strings"
)

// CountTheWays使一个自定义变量类型
// 我们会从标识中读取到它
type CountTheWays []int

// 想要实现自定义类型的flag 必须实现flag.Value接口
// 该接口包含了
// String() string
// Set(string) error
func (c *CountTheWays) String() string {
    result := ""
    for _, v := range *c {
        if len(result) > 0 {
            result += " ... "
        }
        result += fmt.Sprint(v)
    }
    return result
}

func (c *CountTheWays) Set(value string) error {
```

```
    values := strings.Split(value, ",")

    for _, v := range values {
        i, err := strconv.Atoi(v)
        if err != nil {
            return err
        }
        *c = append(*c, i)
    }

    return nil
}
```

3.建立main.go：

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    // 初始化
    c := Config{}
    c.Setup()

    // 常见的调用方式
    flag.Parse()

    // 将通过命令行输入的flag标识拼接打印
    fmt.Println(c.GetMessage())
}
```

4.将main.go打包为flags，运行会显示：

```
is NOT awesome with a certainty of 10 out of 10. Let me count the ways
```

5.运行./flags -s会显示：

```
flag needs an argument: -s
Usage of ./flags:
  -c value
        comma separated list of integers
```

```
  -howawesome int
        how awesome out of 10? (default 10)
  -isawesome
        is it awesome or what?
  -s string
        subject is a string, it defaults to empty (shorthand)
  -subject string
        subject is a string, it defaults to empty
```

6.运行./flags -c 1,2,3 -s aaa会显示：

```
aaa is NOT awesome with a certainty of 10 out of 10. Let me count the ways 1 ...
2 ... 3
```

说明

本节演示了flag包的大多数常见用法。涉及对自定义变量类型，各种内置变量，长短标识以及结构映射标识的使用。我们需要main函数以调用flag.Parse()。

在这个示例中你会发现能够自动获取-h以显示包含的标识列表。需要注意的是，可以在没有参数的情况下调用的布尔标识，并且标识的顺序无关紧要。

flag包提供了快速构建命令行应用的方法，在工作中我们可以利用其来指定设置日志级别或根据应用程序的需求引导用户输入。在命令行参数章节中，我们将探索标识集并使用参数在它们之间切换。

解析命令行参数

上一节中的flag标识是命令行参数中的一种。本章将通过构造嵌套命令来扩展命令行参数。

和上一节类似，我们同样需要一个main函数来调用并执行。有很多的第三方库支持处理复杂的嵌套参数和标识，在这里我们将研究如何仅使用标准库来实现。

实践

1.建立cmdargs.go：

```
package main

import (
    "flag"
    "fmt"
    "os"
)

const version = "1.0.0"
const usage = `Usage:

%s [command]

Commands:
    Greet
    Version
`

const greetUsage = `Usage:

%s greet name [flag]

Positional Arguments:
    name
        the name to greet

Flags:
`

// MenuConf 保存嵌套命令行的级别参数
type MenuConf struct {
    Goodbye bool
}
```

```
// SetupMenu 初始化flag标识
func (m *MenuConf) SetupMenu() *flag.FlagSet {
    menu := flag.NewFlagSet("menu", flag.ExitOnError)
    menu.Usage = func() {
        fmt.Printf(usage, os.Args[0])
        menu.PrintDefaults()
    }
    return menu
}

// GetSubMenu 返回子菜单的flag集
func (m *MenuConf) GetSubMenu() *flag.FlagSet {
    submenu := flag.NewFlagSet("submenu", flag.ExitOnError)
    submenu.BoolVar(&m.Goodbye, "goodbye", false, "Say goodbye instead of hello"
)

    submenu.Usage = func() {
        fmt.Printf(greetUsage, os.Args[0])
        submenu.PrintDefaults()
    }
    return submenu
}

// Greet 由greet命令调用
func (m *MenuConf) Greet(name string) {
    if m.Goodbye {
        fmt.Println("Goodbye " + name + "!")
    } else {
        fmt.Println("Hello " + name + "!")
    }
}

// Version 打印存储为const的当前版本值
func (m *MenuConf) Version() {
    fmt.Println("Version: " + version)
}
```

2.建立main.go：

```
package main

import (
    "fmt"
    "os"
```

```
    "strings"
)


func main() {
    c := MenuConf{}
    menu := c.SetupMenu()

    if err := menu.Parse(os.Args[1:]); err != nil {
        fmt.Printf("Error parsing params %s, error: %v", os.Args[1:], err)
        return
    }

    // 在未输出参数的情况下
    // os.Args[0]是执行文件所在的路径
    // len(os.Args) > 1说明输入了命令行参数
    if len(os.Args) > 1 {

        // 根据分支条件打印输出
        switch strings.ToLower(os.Args[1]) {
        case "version":
            c.Version()
        case "greet":
            f := c.GetSubMenu()
            if len(os.Args) < 3 {
                f.Usage()
                return
            }
            if len(os.Args) > 3 {
                if err := f.Parse(os.Args[3:]); err != nil {
                    fmt.Fprintf(os.Stderr, "Error parsing params %s, error: %v",
os.Args[3:], err)
                    return
                }

            }
            c.Greet(os.Args[2])

        default:
            fmt.Println("Invalid command")
            menu.Usage()
            return
        }
    } else {
        menu.Usage()
        return
```

```
    }
}
```

3.命令行输入不同的参数会显示：

```
$./cmdargs -h
Usage:
./cmdargs [command]
Commands:
Greet
Version
$./cmdargs version
Version: 1.0.0
$./cmdargs greet
Usage:
./cmdargs greet name [flag]
Positional Arguments:
name
the name to greet
Flags:
-goodbye
Say goodbye instead of hello
$./cmdargs greet reader
Hello reader!
$./cmdargs greet reader -goodbye
Goodbye reader!
```

说明

flag.FlagSets可用于设置预期参数。开发人员需要对各参数进行验证，在命令的正确参数子集中进行解析并定义使用字符串。这很容易出错，需要大量迭代才能保证正确性。

flag包帮助开发者解析命令行参数。本节演示了构建复杂命令行应用的基本方法，包括包级别配置，位置参数，多级命令以及如何根据需要将代码拆分为多个文件。

读取和设置环境变量

除了读取文件和命令行传递参数外，环境变量是另一个向应用传递参数的方式。本文将简单探讨如何获取和操作基本的环境变量，然后我们一起来看看如何结合第三方库 https://github.com/kelseyhightower/envconfig(Star 1741)来使用。

我们将构建一个可以通过JSON或环境变量读取配置的应用程序。下一章节将进一步探索其他替代格式，包括TOML和YAML。

实践

1.创建config.go：

```
package envvar

import (
    "encoding/json"
    "os"

    "github.com/kelseyhightower/envconfig"
    "github.com/pkg/errors"
)

// LoadConfig将从存储在路径中的json文件中选择加载文件，
// 然后根据envconfig struct标记覆盖这些值。
// envPrefix是我们为环境变量添加的前缀。
func LoadConfig(path, envPrefix string, config interface{}) error {
    if path != "" {
        err := LoadFile(path, config)
        if err != nil {
            return errors.Wrap(err, "error loading config from file")
        }
    }

    // envconfig.Process根据环境变量填充指定的结构
    err := envconfig.Process(envPrefix, config)
    return errors.Wrap(err, "error loading config from env")
}

// LoadFile 解析一个json文件并填充到config中
func LoadFile(path string, config interface{}) error {
    configFile, err := os.Open(path)
    if err != nil {
        return errors.Wrap(err, "failed to read config file")
```

```
    }
    defer configFile.Close()

    decoder := json.NewDecoder(configFile)
    if err = decoder.Decode(config); err != nil {
        return errors.Wrap(err, "failed to decode config file")
    }
    return nil
}
```

2.创建main.go：

```
package main

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "os"

    "myConcurrency/chapter2/envvar"
)

// Config 将保存我们从json文件和环境变量中捕获的配置
type Config struct {
    Version string `json:"version" required:"true"`
    IsSafe  bool   `json:"is_safe" default:"true"`
    Secret  string `json:"secret"`
}

func main() {
    var err error

    // 建立一个临时json配置文件
    tf, err := ioutil.TempFile("", "tmp")
    if err != nil {
        panic(err)
    }
    defer tf.Close()
    defer os.Remove(tf.Name())

    // json 配置文件的内容
    secrets := `{
        "secret": "so so secret"
    }`
```

```
    if _, err = tf.Write(bytes.NewBufferString(secrets).Bytes()); err != nil {
        panic(err)
    }

    // 向环境变量中添加变量及对应值
    if err = os.Setenv("EXAMPLE_VERSION", "1.0.0"); err != nil {
        panic(err)
    }
    if err = os.Setenv("EXAMPLE_ISSAFE", "false"); err != nil {
        panic(err)
    }

    c := Config{}
    // 从文件中读取配置参数
    if err = envvar.LoadConfig(tf.Name(), "EXAMPLE", &c); err != nil {
        panic(err)
    }

    fmt.Println("secrets file contains =", secrets)

    // 获取环境变量参数及对应值
    fmt.Println("EXAMPLE_VERSION =", os.Getenv("EXAMPLE_VERSION"))
    fmt.Println("EXAMPLE_ISSAFE =", os.Getenv("EXAMPLE_ISSAFE"))

    // c既保存了json配置文件的参数也保存了环境变量的参数 我们将其打印
    fmt.Printf("Final Config: %#v\n", c)

}
```

3.运行main.go，这会输出：

```
secrets file contains = {
"secret": "so so secret"
}
EXAMPLE_VERSION = 1.0.0
EXAMPLE_ISSAFE = false
Final Config: main.Config{Version:"1.0.0", IsSafe:false,
Secret:"so so secret"}
```

说明

使用os包可以很容易的设置和读取环境变量。第三方库envconfig使用结构体tag将环境变量映射给结构体的方式很巧妙，有兴趣可以阅读其源码。

LoadConfig函数可以从各种来源获取配置信息，而不会产生大量开销或太多额外的依赖关系值得借鉴。

另外，请注意错误的处理。在此章节中使用errors.Wrap来生成错误，以便我们可以在不丢失原始错误信息的情况下注释错误。有关详细信息，请参阅第4章“错误处理“

操作TOML，YAML和JSON配置文件

使用配置文件的好处有很多，例如跨语言共享配置、部署修改方便等。Go对JSON实现了开箱即用的支持。本章主要关注以配置结构体tag标签的形式映射至Go结构。

实践

1.我们需要安装以下第三方库：

```
go get github.com/BurntSushi/toml
go get github.com/go-yaml/yaml
```

2.建立toml.go：

```
package confformat

import (
    "bytes"

    "github.com/BurntSushi/toml"
)

// TOMLData是我们使用TOML结构标记的通用数据结构
type TOMLData struct {
    Name string `toml:"name"`
    Age  int    `toml:"age"`
}

// ToTOML 将TOMLData结构转储为TOML格式bytes.Buffer
func (t *TOMLData) ToTOML() (*bytes.Buffer, error) {
    b := &bytes.Buffer{}
    encoder := toml.NewEncoder(b)

    if err := encoder.Encode(t); err != nil {
        return nil, err
    }
    return b, nil
}

// Decode 将数据解码为TOMLData
func (t *TOMLData) Decode(data []byte) (toml.MetaData, error) {
    return toml.Decode(string(data), t)
}
```

3.建立yaml.go：

```
package confformat

import (
    "bytes"

    "github.com/go-yaml/yaml"
)

// YAMLData 是我们使用YAML结构标记的通用数据结构
type YAMLData struct {
    Name string `yaml:"name"`
    Age  int    `yaml:"age"`
}

// ToYAML 将YAMLData结构转储为YAML格式bytes.Buffer
func (t *YAMLData) ToYAML() (*bytes.Buffer, error) {
    d, err := yaml.Marshal(t)
    if err != nil {
        return nil, err
    }

    b := bytes.NewBuffer(d)

    return b, nil
}

// Decode 将数据解码为 YAMLData
func (t *YAMLData) Decode(data []byte) error {
    return yaml.Unmarshal(data, t)
}
```

4.建立json.go：

```
package confformat

import (
    "bytes"
    "encoding/json"
    "fmt"
)

// JSONData 是我们使用JSON结构标记的通用数据结构
type JSONData struct {
```

```
    Name string `json:"name"`
    Age  int    `json:"age"`
}

// ToJSON 将JSONData结构转储为JSON格式bytes.Buffer
func (t *JSONData) ToJSON() (*bytes.Buffer, error) {
    d, err := json.Marshal(t)
    if err != nil {
        return nil, err
    }

    b := bytes.NewBuffer(d)

    return b, nil
}

// Decode 将数据解码为JSONData
func (t *JSONData) Decode(data []byte) error {
    return json.Unmarshal(data, t)
}

// OtherJSONExamples 显示对json解析至其他类型的操作
func OtherJSONExamples() error {
    res := make(map[string]string)
    err := json.Unmarshal([]byte(`{"key": "value"}`), &res)
    if err != nil {
        return err
    }

    fmt.Println("We can unmarshal into a map instead of a struct:", res)

    b := bytes.NewReader([]byte(`{"key2": "value2"}`))
    decoder := json.NewDecoder(b)

    if err := decoder.Decode(&res); err != nil {
        return err
    }

    fmt.Println("we can also use decoders/encoders to work with streams:", res)

    return nil
}
```

5.建立marshal.go：

```
package confformat

import "fmt"

// MarshalAll 建立了不同结构类型的数据并将它们转换至对应的格式
func MarshalAll() error {
    t := TOMLData{
        Name: "Name1",
        Age:  20,
    }

    j := JSONData{
        Name: "Name2",
        Age:  30,
    }

    y := YAMLData{
        Name: "Name3",
        Age:  40,
    }

    tomlRes, err := t.ToTOML()
    if err != nil {
        return err
    }

    fmt.Println("TOML Marshal =", tomlRes.String())

    jsonRes, err := j.ToJSON()
    if err != nil {
        return err
    }

    fmt.Println("JSON Marshal=", jsonRes.String())

    yamlRes, err := y.ToYAML()
    if err != nil {
        return err
    }

    fmt.Println("YAML Marshal =", yamlRes.String())
    return nil
}
```

6.建立 unmarshal.go：

```
package confformat

import "fmt"

const (
    exampleTOML = `name="Example1"
age=99
    `

    exampleJSON = `{"name":"Example2","age":98}`

    exampleYAML = `name: Example3
age: 97
    `
)

// UnmarshalAll 将不同格式的数据转换至对应结构
func UnmarshalAll() error {
    t := TOMLData{}
    j := JSONData{}
    y := YAMLData{}

    if _, err := t.Decode([]byte(exampleTOML)); err != nil {
        return err
    }
    fmt.Println("TOML Unmarshal =", t)

    if err := j.Decode([]byte(exampleJSON)); err != nil {
        return err
    }
    fmt.Println("JSON Unmarshal =", j)

    if err := y.Decode([]byte(exampleYAML)); err != nil {
        return err
    }
    fmt.Println("Yaml Unmarshal =", y)
    return nil
}
```

7.建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter2/confformat"
```

```
func main() {
    if err := confformat.MarshalAll(); err != nil {
        panic(err)
    }

    if err := confformat.UnmarshalAll(); err != nil {
        panic(err)
    }

    if err := confformat.OtherJSONExamples(); err != nil {
        panic(err)
    }
}
```

8.这会输出：

```
TOML Marshal = name = "Name1"
age = 20
JSON Marshal= {"name":"Name2","age":30}
YAML Marshal = name: Name3
age: 40
TOML Unmarshal = {Example1 99}
JSON Unmarshal = {Example2 98}
Yaml Unmarshal = {Example3 97}
We can unmarshal into a map instead of a struct: map[key:value]
we can also use decoders/encoders to work with streams:
map[key:value key2:value2]
```

说明

本节给出了使用TOML，YAML和JSON解析器将原始数据写入go结构并从中读取数据并使用相应格式的示例。 与第1章 I/O和文件系统中的示例一样，我们看到在[]byte，string，bytes.Buffer和其他 I/O 接口之间快速切换是多么常见。

encoding/json包对JSON编码提供了全面的支持。我们抽象出这些函数，可以看到快速转换为对应的结构非常简单。

本节还涉及了struct标签及其用法。前一章也使用了这些，并且这是Go中的一种常见方式，用于向包和库提供有关如何处理结构中所包含数据的提示。

此外，Go对xml也支持开箱即用。而配置文件格式还有ini、cfg、properties、plist、config等，大家可以自行查询相应的第三方库。

操做Unix系统下的pipe管道

将一个程序的输出传递给另一个程序的输入时，Unix管道很有用。 例如：

```
echo "test case" | wc -l
1
```

在Go应用程序中，可以使用os.Stdin读入管道的左侧，并像文件描述符一样工作。为了证明这一点，本节将在管道的左侧进行输入并返回单词列表及其出现次数。这些单词将以空白为分隔标记。

实践

1.建立 pipes.go：

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

// WordCount 获取一个io.Reader并返回一个map，每个单词作为一个键，它的出现次数为对应的值
func WordCount(f io.Reader) map[string]int {
    result := make(map[string]int)

    // 建立scanner处理文件io.Reader接口
    scanner := bufio.NewScanner(f)
    scanner.Split(bufio.ScanWords)

    for scanner.Scan() {
        result[scanner.Text()]++
    }

    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading input:", err)
    }

    return result
}
```

```
func main() {
    fmt.Printf("string: number_of_occurrences\n\n")
    for key, value := range WordCount(os.Stdin) {
        fmt.Printf("%s: %d\n", key, value)
    }
}
```

2.运行：

```
echo "some string" | go run pipes.go
```

3.多执行一些输入：

```
$ echo "test case" | go run pipes.go
string: number_of_occurrences
test: 1
case: 1
$ echo "test case test" | go run pipes.go
string: number_of_occurrences
test: 2
case: 1
```

说明

在go中使用管道非常简单，特别是如果你熟悉使用I/O接口。例如第1章 I/O 和文件系统中的管道配方创建应用程序（https//en.wikipedia.org/wiki/Tee_(command）），其中所有管道输入都会立即写入到标准输出和文件。

本节使用scanner来标记os.Stdin对象的io.Reader接口。请注意在完成所有读取后必须检查错误。

处理信号量

信号是用户或操作系统杀死正在运行的应用程序的有效方式。有时，以更优雅的方式处理这些信号是有意义的。Go提供了一种捕获和处理信号的机制。在本节中，我们将通过使用Go例程的信号来讨论信号的处理。

实践

在通道的使用中，done的使用很常见，参见Concurrency in Go 中文笔记 or-done-channel

1.建立signals.go：

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
)

// CatchSig 为SIGINT中断设置一个监听器
func CatchSig(ch chan os.Signal, done chan bool) {
    // 在等待信号时阻塞
    sig := <-ch
    // 当接收到信号时打印
    fmt.Println("\nsig received:", sig)

    // 对信号类型进行处理
    switch sig {
    case syscall.SIGINT:
        fmt.Println("handling a SIGINT now!")
    case syscall.SIGTERM:
        fmt.Println("handling a SIGTERM in an entirely different way!")
    default:
        fmt.Println("unexpected signal received")
    }

    // 终止
    done <- true
}

func main() {
    // 初始化通道
```

```
    signals := make(chan os.Signal)
    done := make(chan bool)

    // 将它们连接到信号lib
    signal.Notify(signals, syscall.SIGINT, syscall.SIGTERM)

    // 如果一个信号被这个go例程捕获，它将写入 done
    go CatchSig(signals, done)

    fmt.Println("Press ctrl-c to terminate...")
    // 程序会持续打印日志直到done通道被写入
    <-done
    fmt.Println("Done!")

}
```

2.运行并按下ctrl+c，会显示：

```
Press ctrl-c to terminate...
^C
sig received: interrupt
handling a SIGINT now!
Done!
```

3.尝试在另一个单独的命令行窗口运行，然后杀掉其PID，会显示：

```
$./signals
Press ctrl-c to terminate...
# in a separate terminal
$ ps -ef | grep signals
501 30777 26360 0 5:00PM ttys000 0:00.00 ./signals
$ kill -SIGTERM 30777
# in the original terminal
sig received: terminated
handling a SIGTERM in an entirely different way!
Done!
```

说明

本节使用了通道，第9章“并行和并发”会对此进行了更广泛的介绍。signal.Notify函数需要通道以发送信号通知。在命令行杀掉应用是测试向应用程序传递信号的好方法。我们用信号记录程序关心的信号类型。然后，在Go例程中处理传递给该函数的通道上的活动。一旦我们收到信号，就可以随心所欲地处理它。我们可以终止应用程序，使用消息进行响应，并针对不同的信号实施不同的行为。

我们还使用done通道阻止应用程序终止，直到收到信号。对于长期运行的应用程序（如Web应用程序），这是不必要的。创建适当的信号处理例程以进行清理非常有用，尤其是在持有大量状态的应用程序中。例如，在正常关闭时，允许当前处理程序完成其HTTP请求而非中途终止它们。

ANSI命令行着色

本章将探讨一种基本的着色机制，为红色或纯文本着色。有关完整的应用程序，请查看https//github.com/agtorre/gocolorize ，它支持更多颜色和文本类型，并且还实现了fmt.Formatter接口以方便打印。

实践

1.创建color.go：

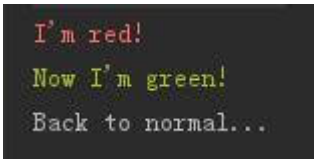
```
package ansicolor

import "fmt"

//文本的颜色
type Color int

const (
    // 默认颜色
    ColorNone = iota
    Red
    Green
    Yellow
    Blue
    Magenta
    Cyan
    White
    Black Color = -1
)

// ColorText 存储了文本及所属的颜色
type ColorText struct {
    TextColor Color
    Text      string
}

func (r *ColorText) String() string {
    if r.TextColor == ColorNone {
        return r.Text
    }

    value := 30
    if r.TextColor != Black {
        value += int(r.TextColor)
```

```
    }
    return fmt.Sprintf("\033[0;%dm%s\033[0m", value, r.Text)
}
```

2.创建main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter2/ansicolor"
)

func main() {
    r := ansicolor.ColorText{
        TextColor: ansicolor.Red,
        Text:      "I'm red!",
    }

    fmt.Println(r.String())

    r.TextColor = ansicolor.Green
    r.Text = "Now I'm green!"

    fmt.Println(r.String())

    r.TextColor = ansicolor.ColorNone
    r.Text = "Back to normal..."

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

3.这会输出：



说明

这里的演示比较简单，大家可以搜索相关颜色和第三方库学习用法。

数据类型转换和解析

本章会覆盖以下内容：

- 数据类型和接口转换
- 使用math包和math/big包处理数字类型
- 货币转换和float64注意事项
- 使用指针和SQL Null类型进行编码和解码
- 对Go数据编码和解码
- Go中的结构体标签和反射
- 通过闭包实现集合操作

介绍

熟悉Go的类型系统的是使用Go开发的关键。本章将展示如何在数据类型之间进行转换，如何使用非常大的数字，如何处理货币，处理编码和解码（包括base64和gob）以及使用闭包创建自定义集合操作。

数据类型和接口转换

Go在数据之间的转换中通常非常灵活。 类型可以"继承"另一种类型：

```
type A int
```

并且总是可以回转到我们"继承"的类型：

```
var a A = 1
fmt.Println(int(a))
```

可以使用fmt.Sprint和strconv进行转换，也可以在字符串和其他类型之间使用反射的接口转换数字。这里将探讨一些基本转换。

实践

1.创建 dataconv.go：

```
package dataconv

import "fmt"

// ShowConv演示了一些类型转换
func ShowConv() {
    // int
    var a = 24

    // float 64
    var b = 2.0

    // 将int转换为float64以进行计算
    c := float64(a) * b
    fmt.Println(c)

    // fmt.Sprintf是生成字符串的好方式
    precision := fmt.Sprintf("%.2f", b)

    // 输出值和对应类型
    fmt.Printf("%s - %T\n", precision, precision)
}
```

2.创建strconv.go：

```
package dataconv

import (
    "fmt"
    "strconv"
)

// Strconv 展示了字符串转换为基本类型
func Strconv() error {
    s := "1234"
    //指定10进制 精度64位
    res, err := strconv.ParseInt(s, 10, 64)
    if err != nil {
        return err
    }

    fmt.Println(res)

    // 让我们试一下二进制
    res, err = strconv.ParseInt("FF", 16, 64)
    if err != nil {
        return err
    }

    fmt.Println(res)

    // 转换字符串为布尔
    val, err := strconv.ParseBool("true")
    if err != nil {
        return err
    }

    fmt.Println(val)

    return nil
}
```

3.创建interfaces.go：

```
package dataconv

import "fmt"

// CheckType 演示了类型断言
func CheckType(s interface{}) {
```

```
    switch s.(type) {
    case string:
        fmt.Println("It's a string!")
    case int:
        fmt.Println("It's an int!")
    default:
        fmt.Println("not sure what it is...")
    }
}

// Interfaces 演示了如何获得断言操作结果
func Interfaces() {
    CheckType("test")
    CheckType(1)
    CheckType(false)

    var i interface{}
    i = "test"

    // manually check an interface
    if val, ok := i.(string); ok {
        fmt.Println("val is", val)
    }

    // this one should fail
    if _, ok := i.(int); !ok {
        fmt.Println("uh oh! glad we handled this")
    }
}
```

4.创建main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter3/dataconv"

func main() {
    dataconv.ShowConv()
    if err := dataconv.Strconv(); err != nil {
        panic(err)
    }
    dataconv.Interfaces()
}
```

5.这会输出：

```
48
2.00 - string
1234
255
true
It's a string!
It's an int!
not sure what it is...
val is test
uh oh! glad we handled this
```

说明

本节通过使用strconv包演示类型之间的转换。这些方法允许Go开发者在各种抽象Go类型之间快速转换。strconv包的操作会在编译期间显示错误，但类型断言不能。如果错误地使用到不受支持的类型，则会引起恐慌。在不确定类型的情况下使用switch是一种不错的方式，fmt包就是这么干的。

转换对于诸如math之类的包非常重要，这些包仅支持在float64上运行。

使用math包和math/big包处理数字类型

实践

1.创建math.go：

```
package math

import (
    "fmt"
    "math"
)

// Examples 演示了math包的基本应用
func Examples() {
    //开平方示例
    i := 25

    // i 是整型，所以需要转型
    result := math.Sqrt(float64(i))

    // 25开方结果是 5
    fmt.Println(result)

    // ceil能够获取大于或等于输入值的最小整数值
    result = math.Ceil(9.5)
    fmt.Println(result)

    // floor能够获取大于或等于输入值的最大整数值
    result = math.Floor(9.5)
    fmt.Println(result)

    // math包同样提供了常用的常数
    fmt.Println("Pi:", math.Pi, "E:", math.E)
}
```

2.创建fib.go：

```
package math

import "math/big"

// 全局变量
```

```
var memoize map[int]*big.Int

func init() {
    // 初始化map
    memoize = make(map[int]*big.Int)
}

// Fib打印斐波纳契序列的第n个数字，它将返回1以表示任何<0 ...它是递归计算并使用big.Int因为int64会快速溢出
func Fib(n int) *big.Int {
    if n < 0 {
        return nil
    }

    // 基础条件
    if n < 2 {
        memoize[n] = big.NewInt(1)
    }

    // 检查我们是否存储它之前进行了计算
    if val, ok := memoize[n]; ok {
        return val
    }

    // 使用map存储然后添加前2个fib值
    memoize[n] = big.NewInt(0)
    memoize[n].Add(memoize[n], Fib(n-1))
    memoize[n].Add(memoize[n], Fib(n-2))

    return memoize[n]
}
```

3.建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter3/math"
)

func main() {
    math.Examples()
```

```
    for i := 0; i < 10; i++ {
        fmt.Printf("%v ", math.Fib(i))
    }
    fmt.Println()
}
```

4.这会输出：

```
5
10
9
Pi: 3.141592653589793 E: 2.718281828459045
1 1 2 3 5 8 13 21 34 55
```

说明

math包使得在Go中执行复杂的数学运算成为可能。本节以执行复杂的浮点操作并根据需要在类型之间进行转换。 值得注意的是，即使使用float64，某些浮点数仍可能存在舍入错误，下一节演示了一些处理此问题的技巧。

math/big部分展示了一个递归的Fibonacci序列。如果你修改main.go循环远远超过10次，如果使用它而不是big.Int，程序将很快溢出int64。这个包还有一些辅助函数，可以将big类型转换为其他类型。

货币转换和float64注意事项

计算货币始终是一个棘手的事情。将money表示为float64可能很诱人，但这会导致计算时出现一些令人讨厌的舍入错误。出于这个原因，最好以美分来计算货币并将其存储为Int64。

从表单，命令行或其他来源收集用户输入时，钱通常以美元形式表示。 因此，最好将其视为字符串，并将该字符串直接转换为便士而不进行浮点转换。 本节将介绍将货币的字符串表示形式转换为int64（便士）并再次返回的方法(大家在进行微信支付开发的时候可以看到，结算单位为分，传递的金额为整数)。

实践

1.建立dollars.go：

```
package currency

import (
    "errors"
    "strconv"
    "strings"
)

// ConvertStringDollarsToPennies 接收美元字符串并转换为int64
func ConvertStringDollarsToPennies(amount string) (int64, error) {
    // 检查传入参数是否合法
    _, err := strconv.ParseFloat(amount, 64)
    if err != nil {
        return 0, err
    }

    // 以"."进行分割
    groups := strings.Split(amount, ".")

    // 如果没有"."则取切片中的第一个元素
    result := groups[0]

    r := ""

    // 处理"."后的数据
    if len(groups) == 2 {
        if len(groups[1]) != 2 {
            return 0, errors.New("invalid cents")
        }
        r = groups[1]
```

```
        if len(r) > 2 {
            r = r[:2]
        }
    }

    // 填充0
    for len(r) < 2 {
        r += "0"
    }

    result += r

    // 转换为int
    return strconv.ParseInt(result, 10, 64)
}
```

2.建立pennies.go：

```
package currency

import (
    "strconv"
)

// ConvertPenniesToDollarString 与上面的例子类似 这是将操作方式逆转
func ConvertPenniesToDollarString(amount int64) string {

    result := strconv.FormatInt(amount, 10)

    negative := false
    if result[0] == '-' {
        result = result[1:]
        negative = true
    }

    for len(result) < 3 {
        result = "0" + result
    }
    length := len(result)

    result = result[0:length-2] + "." + result[length-2:]

    if negative {
        result = "-" + result
    }
```

```
    return result
}
```

3.建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter3/currency"
)

func main() {
    userInput := "15.93"

    pennies, err := currency.ConvertStringDollarsToPennies(userInput)
    if err != nil {
        panic(err)
    }

    fmt.Printf("User input converted to %d pennies\n", pennies)

    pennies += 15

    dollars := currency.ConvertPenniesToDollarString(pennies)

    fmt.Printf("Added 15 cents, new values is %s dollars\n", dollars)
}
```

4.这会输出：

```
User input converted to 1593 pennies
Added 15 cents, new values is 16.08 dollars
```

说明

本节使用strconv和strings包来转换字符串格式的美元和int64中格式的便士。其中转换为float64仅作为验证之用。

strconv.ParseInt和strconv.FormatInt函数对于转换为int64和字符串非常有用。我们还利用了Go字符串可以根据需要轻松添加和切片的特性。

使用指针和SQL Null类型进行编码和解码

当您对Go中的对象进行编码或解码时，未显式设置的类型将被填充为其默认值。字符串将默认为空字符串，整数将默认为0。通常这没什么问题，但当0在业务上包含其他含义时，歧义就出现了。

此外，如果使用结构标记，例如json omitempty，即使它们有效，也会忽略0值。 另一个例子是从SQL返回的Null。 对于Int来说，什么值最能代表Null？本文将探讨Go开发人员处理此问题的一些方法。

实践

1.建立 base.go：

```
package nulls

import (
    "encoding/json"
    "fmt"
)

const (
    jsonBlob     = `{"name": "Aaron"}`
    fulljsonBlob = `{"name":"Aaron", "age":0}`
)

// Example结构体包含age和name字段
type Example struct {
    Age  int    `json:"age,omitempty"`
    Name string `json:"name"`
}

// BaseEncoding 演示了基本的编码和解码操作
func BaseEncoding() error {
    e := Example{}

    // 注意jsonBlob没有age字段
    if err := json.Unmarshal([]byte(jsonBlob), &e); err != nil {
        return err
    }
    fmt.Printf("Regular Unmarshal, no age: %+v\n", e)

    value, err := json.Marshal(&e)
    if err != nil {
```

```
        return err
    }
    //由于age被设置为omitempty(为空则不输出) 所以显示 Regular Marshal, with no age: {"name":"Aaron"}
    fmt.Println("Regular Marshal, with no age:", string(value))

    if err := json.Unmarshal([]byte(fulljsonBlob), &e); err != nil {
        return err
    }
    fmt.Printf("Regular Unmarshal, with age = 0: %+v\n", e)

    value, err = json.Marshal(&e)
    if err != nil {
        return err
    }
    //Regular Marshal, with age = 0: {"name":"Aaron"}
    fmt.Println("Regular Marshal, with age = 0:", string(value))

    return nil
}
```

2.建立pointer.go：

```
package nulls

import (
    "encoding/json"
    "fmt"
)

// 和上一个例子类似 但是*int类型会出现奇妙的nil
// uses a *Int
type ExamplePointer struct {
    Age  *int   `json:"age,omitempty"`
    Name string `json:"name"`
}

func PointerEncoding() error {

    e := ExamplePointer{}
    if err := json.Unmarshal([]byte(jsonBlob), &e); err != nil {
        return err
    }
    //Pointer Unmarshal, no age: {Age:<nil> Name:Aaron}
    fmt.Printf("Pointer Unmarshal, no age: %+v\n", e)
```

```
    value, err := json.Marshal(&e)
    if err != nil {
        return err
    }
    //Pointer Marshal, with no age: {"name":"Aaron"}
    fmt.Println("Pointer Marshal, with no age:", string(value))

    if err := json.Unmarshal([]byte(fulljsonBlob), &e); err != nil {
        return err
    }
    //Pointer Unmarshal, with age = 0: {Age:0xc04200e4b8 Name:Aaron}
    fmt.Printf("Pointer Unmarshal, with age = 0: %+v\n", e)

    value, err = json.Marshal(&e)
    if err != nil {
        return err
    }
    //Pointer Marshal, with age = 0: {"age":0,"name":"Aaron"}
    fmt.Println("Pointer Marshal, with age = 0:", string(value))

    return nil
}
```

3.建立nullencoding.go：

```
package nulls

import (
    "database/sql"
    "encoding/json"
    "fmt"
)

type nullInt64 sql.NullInt64

// 和前面的例子类似 又改变了Age的类型sql.NullInt64
type ExampleNullInt struct {
    Age  *nullInt64 `json:"age,omitempty"`
    Name string     `json:"name"`
}

func (v *nullInt64) MarshalJSON() ([]byte, error) {
    if v.Valid {
        return json.Marshal(v.Int64)
```

```
    }
    return json.Marshal(nil)
}

func (v *nullInt64) UnmarshalJSON(b []byte) error {
    v.Valid = false
    if b != nil {
        v.Valid = true
        return json.Unmarshal(b, &v.Int64)
    }
    return nil
}

func NullEncoding() error {
    e := ExampleNullInt{}

    if err := json.Unmarshal([]byte(jsonBlob), &e); err != nil {
        return err
    }
    //nullInt64 Unmarshal, no age: {Age:<nil> Name:Aaron}
    fmt.Printf("nullInt64 Unmarshal, no age: %+v\n", e)

    value, err := json.Marshal(&e)
    if err != nil {
        return err
    }
    //nullInt64 Marshal, with no age: {"name":"Aaron"}
    fmt.Println("nullInt64 Marshal, with no age:", string(value))

    if err := json.Unmarshal([]byte(fulljsonBlob), &e); err != nil {
        return err
    }
    //nullInt64 Unmarshal, with age = 0: {Age:0xc0420623f0 Name:Aaron}
    fmt.Printf("nullInt64 Unmarshal, with age = 0: %+v\n", e)

    value, err = json.Marshal(&e)
    if err != nil {
        return err
    }
    //nullInt64 Marshal, with age = 0: {"age":0,"name":"Aaron"}
    fmt.Println("nullInt64 Marshal, with age = 0:", string(value))

    return nil
}
```

4.建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter3/nulls"
)

func main() {
    if err := nulls.BaseEncoding(); err != nil {
        panic(err)
    }
    fmt.Println()

    if err := nulls.PointerEncoding(); err != nil {
        panic(err)
    }
    fmt.Println()

    if err := nulls.NullEncoding(); err != nil {
        panic(err)
    }
}
```

5.这会输出：

```
Regular Unmarshal, no age: {Age:0 Name:Aaron}
Regular Marshal, with no age: {"name":"Aaron"}
Regular Unmarshal, with age = 0: {Age:0 Name:Aaron}
Regular Marshal, with age = 0: {"name":"Aaron"}
Pointer Unmarshal, no age: {Age:<nil> Name:Aaron}
Pointer Marshal, with no age: {"name":"Aaron"}
Pointer Unmarshal, with age = 0: {Age:0xc42000a610 Name:Aaron}
Pointer Marshal, with age = 0: {"age":0,"name":"Aaron"}
nullInt64 Unmarshal, no age: {Age:<nil> Name:Aaron}
nullInt64 Marshal, with no age: {"name":"Aaron"}
nullInt64 Unmarshal, with age = 0: {Age:0xc42000a750 Name:Aaron}
nullInt64 Marshal, with age = 0: {"age":0,"name":"Aaron"}
```

说明

从值切换到指针是表示空值的快速方法。这可能会为初始化带来点麻烦，因为无法直接操作*a := 1。除此之外这是一种不错的处理方式。

本节还演示了使用sql.NullInt64类型的替代方法。这通常与SQL一起使用，如果返回Null以外的任何内容，则设置valid，否则设置为Null。我们添加了MarshalJSON和UnmarshallJSON方法以允许此类型与JSON包交互，我们选择使用指针，以便omitempty将继续按预期工作。

对Go数据编码和解码

除了JSON，TOML和YAML之外，Go还具有许多其他编码类型。这些类型主要用于在Go进程之间传输数据，例如协议和RPC，或者在某些字符格式受限制的情况下。

本节将探讨编码和解码gob格式和base64。后续章节将探讨GRPC等协议。

实践

1.创建gob.go：

```
package encoding

import (
    "bytes"
    "encoding/gob"
    "fmt"
)

type pos struct {
    X      int
    Y      int
    Object string
}

// GobExample展示了如何使用gob包
func GobExample() error {
    buffer := bytes.Buffer{}

    p := pos{
        X:      10,
        Y:      15,
        Object: "wrench",
    }

    // 注意如果p是个接口我们需要先调用gob.Register
    e := gob.NewEncoder(&buffer)
    if err := e.Encode(&p); err != nil {
        return err
    }

    // 这里是二进制的 所以打印出来的长度可能并不准确
    fmt.Println("Gob Encoded valued length: ", len(buffer.Bytes()))
```

```
    p2 := pos{}
    d := gob.NewDecoder(&buffer)
    if err := d.Decode(&p2); err != nil {
        return err
    }

    fmt.Println("Gob Decode value: ", p2)

    return nil
}
```

2.创建base64.go：

```
package encoding

import (
    "bytes"
    "encoding/base64"
    "fmt"
    "io/ioutil"
)

// Base64Example 演示了如何使用 base64 包
func Base64Example() error {
    // base64对于不支持以字节/字符串操作的二进制格式的情况很有用

    // 使用辅助函数和URL编码
    value := base64.URLEncoding.EncodeToString([]byte("encoding some data!"))
    fmt.Println("With EncodeToString and URLEncoding: ", value)

    // 解码
    decoded, err := base64.URLEncoding.DecodeString(value)
    if err != nil {
        return err
    }
    fmt.Println("With DecodeToString and URLEncoding: ", string(decoded))

    return nil
}

// Base64ExampleEncoder 与上面的函数类似 但并没有使用URL编码
func Base64ExampleEncoder() error {

    buffer := bytes.Buffer{}
```

```
    // 建立编码器
    encoder := base64.NewEncoder(base64.StdEncoding, &buffer)

    // 确认关闭
    if err := encoder.Close(); err != nil {
        return err
    }
    if _, err := encoder.Write([]byte("encoding some other data")); err != nil {
        return err
    }

    fmt.Println("Using encoder and StdEncoding: ", buffer.String())

    decoder := base64.NewDecoder(base64.StdEncoding, &buffer)
    results, err := ioutil.ReadAll(decoder)
    if err != nil {
        return err
    }

    fmt.Println("Using decoder and StdEncoding: ", string(results))

    return nil
}
```

3.建立main.go：

```
package main

import (
    "github.com/agtorre/go-cookbook/chapter3/encoding"
)

func main() {
    if err := encoding.Base64Example(); err != nil {
        panic(err)
    }

    if err := encoding.Base64ExampleEncoder(); err != nil {
        panic(err)
    }

    if err := encoding.GobExample(); err != nil {
        panic(err)
    }
}
```

4.这会输出：

```
With EncodeToString and URLEncoding:
ZW5jb2Rpbmcgc29tZSBkYXRhIQ==
With DecodeToString and URLEncoding: encoding some data!
Using encoder and StdEncoding: ZW5jb2Rpbmcgc29tZSBvdGhlciBkYXRh
Using decoder and StdEncoding: encoding some other data
Gob Encoded valued length: 57
Gob Decode value: {10 15 wrench}
```

说明

Gob编码是一种使用Go数据类型构建的流格式。在连续发送和编码时效率最高。对于单个项目，其他编码格式（如JSON）可能更高效且易于移植。使用gob编码使得编组大型复杂结构并在单独的过程中重建它们变得简单。虽然这里没有显示，但gob也可以使用自定义MarshalBinary和UnmarshalBinary方法对自定义类型或未导出类型进行操作。

Base64编码对于通过GET请求中的URL进行通信或生成二进制数据的字符串表示编码非常有用。大多数语言都支持这种格式，并在另一端解码数据。因此，在不支持JSON格式的情况下，对JSON等内容进行Base64编码是很常见的。

Go中的结构体标签和反射

反射是一个复杂的话题。在Go中反射最常用在处理结构体标签，其核心是处理键值字符串。即查找键，然后处理对应值。可以想象，在使用JSON marshal和unmarshal进行操作的时候，处理这些值存在很多复杂性。

反射包用于解析接口对象。它能够帮助我们查看结构类型，值，结构标签等。如果你需要处理的不仅仅是基本类型转换，那么这你应该关注reflect包的使用。

实践

1.创建serialize.go：

```
func SerializeStructStrings(s interface{}) (string, error) {
    result := ""

    // reflect.TypeOf使用传入的接口生成type类型
    r := reflect.TypeOf(s)
    // reflect.ValueOf返回结构体每个字段对应的值
    value := reflect.ValueOf(s)

    // 如果传入的是个结构体的指针 那么可以针对性的对其进行单独处理
    if r.Kind() == reflect.Ptr {
        r = r.Elem()
        value = value.Elem()
    }

    // 循环所有的内部字段
    for i := 0; i < r.NumField(); i++ {
        field := r.Field(i)
        // 字段的名称
        key := field.Name

        // Lookup返回与标记字符串中的key关联的值。 如果密钥存在于标记中，则返回值（可以为空）。
        // 否则返回的值将是空字符串。ok返回值报告是否在标记字符串中显式设置了值。
        // 如果标记没有传统格式，则Lookup返回的值不做指定。
        if serialize, ok := field.Tag.Lookup("serialize"); ok {
            // 忽略“ - ”否则整个值成为序列化'键'
            if serialize == "-" {
                continue
            }
            key = serialize
```

```
        }

        // 判断每个字段的类型
        switch value.Field(i).Kind() {
        // 当前示例我们仅简单判断字符串
        case reflect.String:
            result += key + ":" + value.Field(i).String() + ";"
        default:
            continue
        }
    }
    return result, nil
}
```

2.建立deserialize.go：

```
package tags

import (
    "errors"
    "reflect"
    "strings"
)

// DeSerializeStructStrings 反序列化字符串为对应的结构体
func DeSerializeStructStrings(s string, res interface{}) error {
    r := reflect.TypeOf(res)

    // 我们要求传入的必须是指针
    if r.Kind() != reflect.Ptr {
        return errors.New("res must be a pointer")
    }

    // 解指针
    // Elem返回r(Type类型)元素的type
    // 如果该type.Kind不是Array, Chan, Map, Ptr, 或 Slice会产生panic
    r = r.Elem()
    value := reflect.ValueOf(res).Elem()

    // 将传入的序列化字符串分割为map
    vals := strings.Split(s, ";")
    valMap := make(map[string]string)
    for _, v := range vals {
        keyval := strings.Split(v, ":")
        if len(keyval) != 2 {
```

```
            continue
        }
        valMap[keyval[0]] = keyval[1]
    }

    // 循环所有的内部字段
    for i := 0; i < r.NumField(); i++ {
        field := r.Field(i)

        // 检查是否符合预置的tag
        if serialize, ok := field.Tag.Lookup("serialize"); ok {
            // 忽略'-'否则整个值成为序列化'键'
            if serialize == "-" {
                continue
            }
            // 判断是否处于map内
            if val, ok := valMap[serialize]; ok {
                value.Field(i).SetString(val)
            }
        } else if val, ok := valMap[field.Name]; ok {
            // 是否是在map中的字段名称
            value.Field(i).SetString(val)
        }
    }
    return nil
}
```

3.建立 tags.go：

```
package tags

import "fmt"

// 注意Person内个字段的tag标签
type Person struct {
    Name  string `serialize:"name"`
    City  string `serialize:"city"`
    State string
    Misc  string `serialize:"-"`
    Year  int    `serialize:"year"`
}

// EmptyStruct 演示了根据 tag 序列化和反序列化一个空结构体
func EmptyStruct() error {
    p := Person{}
```

```
    res, err := SerializeStructStrings(&p)
    if err != nil {
        return err
    }
    fmt.Printf("Empty struct: %#v\n", p)
    fmt.Println("Serialize Results:", res)

    newP := Person{}
    if err := DeSerializeStructStrings(res, &newP); err != nil {
        return err
    }
    fmt.Printf("Deserialize results: %#v\n", newP)
    return nil
}

// FullStruct 演示了根据 tag 序列化和反序列化一个非空结构体
func FullStruct() error {
    p := Person{
        Name:  "Aaron",
        City:  "Seattle",
        State: "WA",
        Misc:  "some fact",
        Year:  2017,
    }
    res, err := SerializeStructStrings(&p)
    if err != nil {
        return err
    }
    fmt.Printf("Full struct: %#v\n", p)
    fmt.Println("Serialize Results:", res)

    newP := Person{}
    if err := DeSerializeStructStrings(res, &newP); err != nil {
        return err
    }
    fmt.Printf("Deserialize results: %#v\n", newP)
    return nil
}
```

4.建立main.go：

```
package main

import (
```

```
    "fmt"

    "github.com/agtorre/go-cookbook/chapter3/tags"
)

func main() {

    if err := tags.EmptyStruct(); err != nil {
        panic(err)
    }

    fmt.Println()

    if err := tags.FullStruct(); err != nil {
        panic(err)
    }
}
```

5.这会输出：

```
Empty struct: tags.Person{Name:"", City:"", State:"", Misc:"", Year:0}
Serialize Results: name:;city:;State:;
Deserialize results: tags.Person{Name:"", City:"", State:"", Misc:"", Year:0}

Full struct: tags.Person{Name:"Aaron", City:"Seattle", State:"WA", Misc:"some fa
ct", Year:2017}
Serialize Results: name:Aaron;city:Seattle;State:WA;
Deserialize results: tags.Person{Name:"Aaron", City:"Seattle", State:"WA", Misc:
"", Year:0}
```

说明

本节简单的展示了使用反射根据结构体的tag标签来进行字符串序列化和序列化，我们并未处理一些特殊情况，例如字符串包含 ':' 或 ';'。针对于本示例，需要注意：

1. 如果字段是字符串，则将对其进行序列化/反序列化。
2. 如果字段不是字符串，则将忽略该字段。
3. 如果字段的struct标记不包含"serialize"，则需要进行额外操作以保证序列化/反序列化正确完成。
4. 没有考虑处理重复项。
5. 如果未指定struct标记，则简单使用字段名称。
6. 如果指定了标签为'-'，则即使该字段是字符串，也会忽略。
7. 还需要注意的是，反射不能与非导出值一起使用。

一个完善的反射操作需要考虑的细节很多，因此，在面对一个不是那么完美的第三方反射库时，尽量保持仁慈之心是值得赞美的。

通过闭包实现集合操作

如果你一直在使用函数或动态编程语言，可能会觉得for循环和if语句会产生冗长的代码。用于处理列表的函数构造（例如map和filter）通常很有用，并使代码看起来更具可读性。但是，在Go标准库中很少支持类似的操作，并且在没有泛型或非常复杂的映射和使用空接口的情况下很难使用。本节将提供使用Go闭包实现集合的一些基本示例。

实践

创建collections.go：

```
package collections

// WorkWith会实现集合接口
type WorkWith struct {
    Data    string
    Version int
}

// Filter是一个过滤函数。
func Filter(ws []WorkWith, f func(w WorkWith) bool) []WorkWith {
    // 初始化返回值
    result := make([]WorkWith, 0)
    for _, w := range ws {
        if f(w) {
            result = append(result, w)
        }
    }
    return result
}

// Map是一个映射函数。
func Map(ws []WorkWith, f func(w WorkWith) WorkWith) []WorkWith {
    // 返回值的长度应该与传入切片长度一致
    result := make([]WorkWith, len(ws))

    for pos, w := range ws {
        newW := f(w)
        result[pos] = newW
    }
    return result
}
```

创建functions.go：

```
package collections

import "strings"

// LowerCaseData 将传入WorkWith的data字段变为小写
func LowerCaseData(w WorkWith) WorkWith {
    w.Data = strings.ToLower(w.Data)
    return w
}

// IncrementVersion 将传入WorkWith的Version加1
func IncrementVersion(w WorkWith) WorkWith {
    w.Version++
    return w
}

// OldVersion 返回一个闭包，用于验证版本是否大于指定的值
func OldVersion(v int) func(w WorkWith) bool {
    return func(w WorkWith) bool {
        return w.Version >= v
    }
}
```

创建main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter3/collections"
)

func main() {
    ws := []collections.WorkWith{
        {"Example", 1},
        {"Example 2", 2},
    }

    fmt.Printf("Initial list: %#v\n", ws)

    ws = collections.Map(ws, collections.LowerCaseData)
    fmt.Printf("After LowerCaseData Map: %#v\n", ws)

    ws = collections.Map(ws, collections.IncrementVersion)
```

```
    fmt.Printf("After IncrementVersion Map: %#v\n", ws)

    ws = collections.Filter(ws, collections.OldVersion(3))
    fmt.Printf("After OldVersion Filter: %#v\n", ws)
}
```

这会输出：

```
Initial list: []collections.WorkWith{collections.WorkWith{Data:"Example", Version:1}, collections.WorkWith{Data:"Example 2", Version:2}}
After LowerCaseData Map: []collections.WorkWith{collections.WorkWith{Data:"example", Version:1}, collections.WorkWith{Data:"example 2", Version:2}}
After IncrementVersion Map: []collections.WorkWith{collections.WorkWith{Data:"example", Version:2}, collections.WorkWith{Data:"example 2", Version:3}}
After OldVersion Filter: []collections.WorkWith{collections.WorkWith{Data:"example 2", Version:3}}
```

说明

Go中的闭包非常强大。虽然我们的集合函数不是通用的，但它们相对较小，并且很容易应用于WorkWith结构的各种函数。你可能会注意到我们在任何地方都没有返回错误。而且我们返回了一个全新的切片。

如果需要将修改层应用于列表或列表结构，则此模式可以为节省大量的操作并使测试变得非常简单。还可以将map和filter链接在一起，以获得非常富有表现力的编码风格。

错误处理

本章会覆盖以下内容：

- 错误接口
- 使用第三方errors包
- 使用log包记录错误
- 结构化日志记录
- 使用context包进行日志记录
- 使用包级全局变量
- 处理恐慌

介绍

错误处理非常重要。在Go中错误处理更自由，也更严格——这取决于开发者如何看待。Go错误不像异常一样工作，未处理的错误可能导致巨大的问题。你应该努力在发生错误时处理并提前考虑错误。

错误接口

Error接口看起来非常简洁：

```
type Error interface{
    Error() string
}
```

这个接口很容易满足，不幸的是，这也很容易导致需要根据收到的错误与采取操作的包的混淆。

在Go中有很多方法可以创建错误，本节将探讨基本错误的创建，已分配值或类型的错误，以及使用结构的自定义错误。

实践

创建basicerrors.go：

```
package basicerrors

import (
    "errors"
    "fmt"
)

// ErrorValue创建了包级错误
// 可以采用这样的方式判断： if err == ErrorValue
var ErrorValue = errors.New("this is a typed error")

// TypedError创建了包含错误类型的匿名字段
// 可以采用断言的方式判断：err.(type) == ErrorValue
type TypedError struct {
    error
}

//BasicErrors 演示了错误的创建
func BasicErrors() {
    err := errors.New("this is a quick and easy way to create an error")
    fmt.Println("errors.New: ", err)

    err = fmt.Errorf("an error occurred: %s", "something")
    fmt.Println("fmt.Errorf: ", err)

    err = ErrorValue
```

```
    fmt.Println("value error: ", err)

    err = TypedError{errors.New("typed error")}
    fmt.Println("typed error: ", err)

}
```

创建custom.go：

```
package basicerrors

import (
    "fmt"
)

// CustomError 实现了Error接口
type CustomError struct {
    Result string
}

func (c CustomError) Error() string {
    return fmt.Sprintf("there was an error; %s was the result", c.Result)
}

// SomeFunc 返回一个 error
func SomeFunc() error {
    c := CustomError{Result: "this"}
    return c
}
```

创建 main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter4/basicerrors"
)

func main() {
    basicerrors.BasicErrors()

    err := basicerrors.SomeFunc()
```

```
    fmt.Println("custom error: ", err)
}
```

这会输出：

```
errors.New: this is a quick and easy way to create an error
fmt.Errorf: an error occurred: something
typed error: this is a typed error
custom error: there was an error; this was the result
```

说明

无论你是使用errors.New，fmt.Errorf还是自定义错误，最重要的是你应该永远不要在代码中忽略错误。这些不同的错误定义方法提供了很大的灵活性。例如，你可以在结构中添加额外的函数，以进一步查看可能发生的错误并将接口转换为调用函数中的错误类型，以获得一些额外的功能。

错误接口本身非常简单，唯一的要求是返回一个有效的字符串。将它连接到一个结构可能对某些高级应用程序很有用，这些应用程序在整个过程中都有统一的错误处理，同时也希望能够与其他应用程序进行良好地协作。

使用第三方errors包

位于github.com/pkg/errors 的errors包是Go标准库的替代品。它提供了一些非常有用的操作用于封装和处理错误。在之前的章节我们已经看到了如何对错误进行封装，值得注意的是，使用标准库的方式对错误进行封装，会改变其类型并使类型断言失败。

本节我们会演示对github.com/pkg/errors 的使用。

实践

获取第三方库：

```
go get github.com/pkg/errors/
```

建立errwrap.go ：

```
package errwrap

import (
    "fmt"

    "github.com/pkg/errors"
)

// WrappedError 演示了如何对错误进行封装
func WrappedError(e error) error {
    return errors.Wrap(e, "An error occurred in WrappedError")
}

type ErrorTyped struct {
    error
}

func Wrap() {
    e := errors.New("standard error")

    fmt.Println("Regular Error - ", WrappedError(e))

    fmt.Println("Typed Error - ", WrappedError(ErrorTyped{errors.New("typed error")}))

    fmt.Println("Nil -", WrappedError(nil))

}
```

建立unwrap.go：

```
package errwrap

import (
    "fmt"

    "github.com/pkg/errors"
)

// Unwrap 解除封装并进行断言处理
func Unwrap() {

    err := error(ErrorTyped{errors.New("an error occurred")})
    err = errors.Wrap(err, "wrapped")

    fmt.Println("wrapped error: ", err)

    // 处理错误类型
    switch errors.Cause(err).(type) {
    case ErrorTyped:
        fmt.Println("a typed error occurred: ", err)
    default:
        fmt.Println("an unknown error occurred")
    }
}

// StackTrace 打印错误栈
func StackTrace() {
    err := error(ErrorTyped{errors.New("an error occurred")})
    err = errors.Wrap(err, "wrapped")

    fmt.Printf("%+v\n", err)
}
```

建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter4/errwrap"
)
```

```
func main() {
    errwrap.Wrap()
    fmt.Println()
    errwrap.Unwrap()
    fmt.Println()
    errwrap.StackTrace()
}
```

这会输出：

```
Regular Error - An error occurred in WrappedError: standard
error
Typed Error - An error occurred in WrappedError: typed error
Nil - <nil>
wrapped error: wrapped: an error occurred
a typed error occurred: wrapped: an error occurred
an error occurred
github.com/agtorre/go-cookbook/chapter4/errwrap.StackTrace
/Users/lothamer/go/src/github.com/agtorre/gocookbook/chapter4/errwrap/unwrap.go:
30
main.main
/tmp/go/src/github.com/agtorre/gocookbook/chapter4/errwrap/example/main.go:14
```

说明

封装错误在开发中很常见，尤其对日志及问题查找作用巨大。该示例为大家提供了基本的思路。此外，该库对错误为nil的情况进行了处理：

```
func RetError() error{
    err := ThisReturnsAnError()
    return errors.Wrap(err, "This only does something if err != nil")
}
```

该库的作者Dave Cheney提供了一系列的博客来讨论Go语言的错误处理，可以查看 https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully 以了解更多。

使用log包记录错误

在应用出现异常或意外情况时，日志会显得无比重要。如果提供系统级别的日志，在代码的关键位置记录相关信息，也有助于在开发期间快速调试问题。太多的日志记录会导致很难找到有用的东西，但没有足够的日志记录可能在系统崩溃时无法追寻其根本原因。本节将演示如何使用Go标准库中的log包。

实践

获取第三方库：

```
go get github.com/pkg/errors/
```

建立log.go：

```
package log

import (
    "bytes"
    "fmt"
    "log"
)

// 对日志进行设置
func Log() {
    // logger会写入bytes.Buffer类型的数据
    buf := bytes.Buffer{}

    // 第二个参数是前缀最后一个参数是关于选项
    // 配置选项可以用逻辑或符号组合起来
    logger := log.New(&buf, "logger: ", log.Lshortfile|log.Ldate)

    logger.Println("test")

    logger.SetPrefix("new logger: ")

    logger.Printf("you can also add args(%v) and use Fataln to log and crash", t
rue)

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

建立error.go：

```
package log

import "github.com/pkg/errors"
import "log"

// OriginalError返回错误的原始信息
func OriginalError() error {
    return errors.New("error occurred")
}

// PassThroughError 调用OriginalError并将其封装
func PassThroughError() error {
    err := OriginalError()
    // 无需检查错误，因为使用该库时此操作适用于nil
    return errors.Wrap(err, "in passthrougherror")
}

// FinalDestination处理错误而不传递它
func FinalDestination() {
    err := PassThroughError()
    if err != nil {
        // 将任何产生的意外记录到日志中
        log.Printf("an error occurred: %s\n", err.Error())
        return
    }
}
```

建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter4/log"
)

func main() {
    fmt.Println("basic logging and modification of logger:")
    log.Log()
    fmt.Println("logging 'handled' errors:")
    log.FinalDestination()
}
```

这会输出：

```
basic logging and modification of logger:
logger: 2018/07/23 log.go:18: test
new logger: 2018/07/23 log.go:22: you can also add args(true) and use Fataln to log and crash

logging 'handled' errors:
2018/07/23 11:37:07 an error occurred: in passthrougherror: error occurred
```

说明

你可以使用log.NewLogger()初始化日志并传递它，或直接调用log包记录消息。

这种操作方式并不优雅，因为你需要传递过去一堆变量，这是令人困惑并难以理解的。下一节，我们探讨如何使用更加灵活的方式设置日志。

结构化日志记录

记录日志的主要目的是便于在事件发生或过后检查系统的状态。当你有大量正在记录日志的微服务时，日志消息会很难梳理。

本节使用第三方库来对日志进行记录。

实践

获取第三方库：

```
go get github.com/sirupsen/logrus
go get github.com/apex/log
```

建立logrus.go：

```
package structured

import "github.com/sirupsen/logrus"

// Hook 实现了logrus中的hook 接口
type Hook struct {
    id string
}

// Fire 在每次记录日志时都会触发
func (hook *Hook) Fire(entry *logrus.Entry) error {
    entry.Data["id"] = hook.id
    return nil
}

// Levels 日志等级
func (hook *Hook) Levels() []logrus.Level {
    return logrus.AllLevels
}

// Logrus 演示了一些基本的logrus库操作
func Logrus() {

    logrus.SetFormatter(&logrus.TextFormatter{})
    logrus.SetLevel(logrus.InfoLevel)
    logrus.AddHook(&Hook{"123"})

    fields := logrus.Fields{}
```

```
    fields["success"] = true
    fields["complex_struct"] = struct {
        Event string
        When  string
    }{"Something happened", "Just now"}

    x := logrus.WithFields(fields)
    x.Warn("warning!")
    x.Error("error!")
}
```

建立apex.go：

```
package structured

import (
    "errors"
    "os"

    "github.com/apex/log"
    "github.com/apex/log/handlers/text"
)

// ThrowError抛出我们将追踪的错误
func ThrowError() error {
    err := errors.New("a crazy failure")
    log.WithField("id", "123").Trace("ThrowError").Stop(&err)
    return err
}

type CustomHandler struct {
    id      string
    handler log.Handler
}

// HandleLog 会对日志进行处理
func (h *CustomHandler) HandleLog(e *log.Entry) error {
    e.WithField("id", h.id)
    return h.handler.HandleLog(e)
}

func Apex() {
    log.SetHandler(&CustomHandler{"123", text.New(os.Stdout)})
    err := ThrowError()
```

```
    //WithError可以便利的记录错误
    log.WithError(err).Error("an error occurred")
}
```

建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter4/structured"
)

func main() {
    fmt.Println("Logrus:")
    structured.Logrus()

    fmt.Println()
    fmt.Println("Apex:")
    structured.Apex()
}
```

这会输出：

```
Logrus:
WARN[0000] warning! complex_struct={Something happened Just now}
id=123 success=true
ERRO[0000] error! complex_struct={Something happened Just now}
id=123 success=true
Apex:
INFO[0000] ThrowError id=123
ERROR[0000] ThrowError duration=133ns error=a crazy failure
id=123
ERROR[0000] an error occurred error=a crazy failure
```

说明

sirupsen/logrus和apex/log包都是优秀的结构化日志库。二者都提供了用于发送到多个事件或向日志条目添加额外字段的处理方案。其中使用logrus为日志和服务名称添加行号将相对简单。这方便了跨不同服务跟踪日志。

使用context包进行日志记录

本节将展示如何在各种函数间传递log操作。使用Go标准库中的context包是在函数之间传递和取消变量的绝佳方式。本节将探讨如何使用该方案在函数之间分配变量以进行日志记录。

本节使用到了上节中提到的apex库。

实践

获取第三方库：

```
go get github.com/apex/log
```

建立log.go：

```
package context

import (
    "context"

    "github.com/apex/log"
)

type key int

const logFields key = 0

func getFields(ctx context.Context) *log.Fields {
    fields, ok := ctx.Value(logFields).(*log.Fields)
    if !ok {
        f := make(log.Fields)
        fields = &f
    }
    return fields
}

// FromContext 接收一个log.Interface和context
// 然后返回由context对象填充的log.Entry指针
func FromContext(ctx context.Context, l log.Interface) (context.Context, *log.En
try) {
    fields := getFields(ctx)
    e := l.WithFields(fields)
    ctx = context.WithValue(ctx, logFields, fields)
    return ctx, e
```

```
}

// WithField 将log.Fielder添加到context
func WithField(ctx context.Context, key string, value interface{}) context.Conte
xt {
    return WithFields(ctx, log.Fields{key: value})
}

// WithFields 将log.Fielder添加到context
func WithFields(ctx context.Context, fields log.Fielder) context.Context {
    f := getFields(ctx)
    for key, val := range fields.Fields() {
        (*f)[key] = val
    }
    ctx = context.WithValue(ctx, logFields, f)
    return ctx
}
```

建立collect.go：

```
package context

import (
    "context"
    "os"

    "github.com/apex/log"
    "github.com/apex/log/handlers/text"
)

// Initialize调用3个函数来设置，然后在操作完成之前记录日志
func Initialize() {

    log.SetHandler(text.New(os.Stdout))
    //初始化context
    ctx := context.Background()
    // 将context与log.Log建立联系
    ctx, e := FromContext(ctx, log.Log)

    ctx = WithField(ctx, "id", "123")
    e.Info("starting")
    gatherName(ctx)
    e.Info("after gatherName")
    gatherLocation(ctx)
    e.Info("after gatherLocation")
```

```
}

func gatherName(ctx context.Context) {
    ctx = WithField(ctx, "name", "Go Cookbook")
}

func gatherLocation(ctx context.Context) {
    ctx = WithFields(ctx, log.Fields{"city": "Seattle", "state": "WA"})
}
```

建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter4/context"

func main() {
    context.Initialize()
}
```

这会输出：

```
INFO[0000] starting id=123
INFO[0000] after gatherName id=123 name=Go Cookbook
INFO[0000] after gatherLocation city=Seattle id=123 name=Go
Cookbook state=WA
```

说明

context包的使用在databases包和HTTP包中都可以看到。本节将log附加到context并将其用于记录日志以达到并发安全的目的。我们通过在方法中传递context来同时传递附加于其上的属性，以达到在最终调用位置记录日志的目的。

这些修改了存储在上下文中的单个值，并提供了使用context的另一个好处：可以执行取消和超时操作，并且线程安全。

使用包级全局变量

在前面的章节中，我们使用的log都是包级别的变量。有时，构建库以支持具有各种方法和顶级函数的结构是有用的，这样就可以直接使用它们而无需传递。

本节演示了如何使用sync.Once确保全局日志logger仅初始化一次。它也可以通过Set方法绕过。示例中只导出WithField和Debug，但对于你可以想象到的其他附加到日志对象的每个方法也适用。

实践

获取第三方库：

```
go get github.com/apex/log
```

建立global.go：

```
package global

import (
    "errors"
    "os"
    "sync"

    "github.com/sirupsen/logrus"
)

var (
    log     *logrus.Logger
    initLog sync.Once
)

// Init设置logger 如果多次初始化 会返回错误
func Init() error {
    err := errors.New("already initialized")
    initLog.Do(func() {
        err = nil
        log = logrus.New()
        log.Formatter = &logrus.JSONFormatter{}
        log.Out = os.Stdout
        log.Level = logrus.DebugLevel
    })
    return err
}
```

```
// SetLog 设置log
func SetLog(l *logrus.Logger) {
    log = l
}

// WithField 使用全局log返回logrus.Entry指针
func WithField(key string, value interface{}) *logrus.Entry {
    return log.WithField(key, value)
}

// Debug 使用全局log记录信息
func Debug(args ...interface{}) {
    log.Debug(args...)
}
```

建立log.go：

```
package global

// UseLog 演示了使用全局 log
func UseLog() error {
    if err := Init(); err != nil {
        return err
    }

    // 如果在其他包 应该调用global.WithField 和 global.Debug
    WithField("key", "value").Debug("hello")
    Debug("test")

    return nil
}
```

建立main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter4/global"

func main() {
    if err := global.UseLog(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
{"key":"value","level":"debug","msg":"hello","time":"2017-02-12T19:22:50-08:00"}
{"level":"debug","msg":"test","time":"2017-02-12T19:22:50-08:00"}
```

说明

这些全局包级对象的常见模式是保持全局未导出，并仅通过方法公开所需的功能。 通常，你还可以包含一个方法，以便为需要调用对象的包返回该对象的副本。

sync.Once类型是新引入的结构。这个结构与Do方法一起只在代码中执行一次。我们在初始化代码中使用它，如果多次调用Init，Init函数将抛出一个错误。

虽然此示例使用的是日志操作，但同样也可以想象为对数据库连接，数据流和许多常见用例的使用。

处理恐慌

实现长时间运行的进程时，某些代码路径可能会导致混乱。这通常见于未初始化的map和指针，以及在验证用户输入不良的情况下，例如除零问题。

在这些情况下，系统的部分崩溃比整体崩溃更糟糕，因此就需要捕获和处理发生的恐慌。

实践

获取第三方库：

```
go get github.com/a
```

建立panic.go：

```
package panic

import (
    "fmt"
    "strconv"
)

// Panic 演示除零恐慌
func Panic() {
    zero, err := strconv.ParseInt("0", 10, 64)
    if err != nil {
        panic(err)
    }

    a := 1 / zero
    fmt.Println("we'll never get here", a)
}

// Catcher 处理恐慌
func Catcher() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("panic occurred:", r)
        }
    }()
    Panic()
}
```

建立main.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter4/panic"
)

func main() {
    fmt.Println("before panic")
    panic.Catcher()
    fmt.Println("after panic")
}
```

这会输出：

```
before panic
panic occurred: runtime error: integer divide by zero
after panic
```

说明

这是捕捉恐慌的一个非常基本的例子。你可以想象使用更复杂的中间件，如何在运行许多嵌套函数后推迟恢复并捕获它。在恢复过程中，可以做任何你想要的事情，例如发送日志。

在大多数Web应用程序中，常见的是捕获恐慌并在发生混乱时发出http.InternalServerError消息。

数据存储

本章会覆盖以下内容：

- 使用database/sql包操作MySQL
- 执行数据库事务接口
- SQL的连接池速率限制和超时
- 操作Redis
- 操作MongoDB
- 创建存储接口以实现数据可移植性

介绍

应用程序经常需要使用数据持久化。这通常采用关系数据库和非关系数据库的形式，以及键值存储等。

本章将介绍各种存储接口，如何使用连接池等内容进行并行访问，并提出连接数据库时通常会遇到的一些注意事项。

使用database/sql包操作MySQL

关系数据库较为常见且易于理解。MySQL和Postgres是两个最流行的开源关系数据库。本节将演示如何使用database/sql包，该包为许多关系数据库提供钩子并自动处理连接池、连接持续时间等。

该包的未来版本将提供包括对上下文和超时的支持。

实践

获取第三方库：

```
go get github.com/go-sql-driver/mysql
```

建立config.go：

```
package database

import (
    "database/sql"
    "fmt"
    "os"
    "time"

    _ "github.com/go-sql-driver/mysql" //注意这里的导入方式
)

// Example 保存了查询的结果
type Example struct {
    Name    string
    Created *time.Time
}

// Setup 配置数据库连接
func Setup() (*sql.DB, error) {
    db, err := sql.Open("mysql",
        fmt.Sprintf("%s:%s@/gocookbook?parseTime=true", os.Getenv("MYSQLUSERNAM
E"), os.Getenv("MYSQLPASSWORD")))
    if err != nil {
        return nil, err
    }
    return db, nil
}
```

建立create.go：

```
package database

import (
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

// Create 建立一个表并填充数据
func Create(db *sql.DB) error {

    if _, err := db.Exec("CREATE TABLE example (name VARCHAR(20), created DATETI
ME)"); err != nil {
        return err
    }

    if _, err := db.Exec(`INSERT INTO example (name, created) values ("Aaron", N
OW())`); err != nil {
        return err
    }

    return nil
}
```

建立query.go：

```
package database

import (
    "database/sql"
    "fmt"

    _ "github.com/go-sql-driver/mysql"
)

// Query 获取一个新的连接到数据库 并进行查询
func Query(db *sql.DB) error {
    name := "Aaron"
    rows, err := db.Query("SELECT name, created FROM example where name=?", nam
e)
    if err != nil {
        return err
    }
```

```
    defer rows.Close()
    for rows.Next() {
        var e Example
        if err := rows.Scan(&e.Name, &e.Created); err != nil {
            return err
        }
        fmt.Printf("Results:\n\tName: %s\n\tCreated: %v\n", e.Name, e.Created)
    }
    return rows.Err()
}
```

建立 exec.go：

```
package database

import (
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)

// Exec 删除该表
func Exec(db *sql.DB) error {
    // 在删除该表时存在未处理的错误 这样写并不推荐
    defer db.Exec("DROP TABLE example")

    if err := Create(db); err != nil {
        return err
    }

    if err := Query(db); err != nil {
        return err
    }
    return nil
}
```

建立 main.go：

```
package main

import (
    "github.com/agtorre/go-cookbook/chapter5/database"
    _ "github.com/go-sql-driver/mysql"
)
```

```
func main() {
    db, err := database.Setup()
    if err != nil {
        panic(err)
    }

    if err := database.Exec(db); err != nil {
        panic(err)
    }
}
```

这会输出：

```
Results:
    Name: Aaron
    Created: 2017-02-16 19:02:36 +0000 UTC
```

说明

```
_ "github.com/go-sql-driver/mysql"
```

该行引入了数据库驱动。如果要连接到Postgres，SQLite或任何其他实现了database/sql接口的驱动，命令将类似。

一旦建立连接成功，该包会建立连接池。你可以直接在连接上执行SQL，也可以创建用commit和rollback命令执行的事务对象。在随后的章节会进一步探讨连接池的相关问题。

在与数据库通信时，mysql包为Go的时间对象提供了一些便利支持。 注意本节是从从MYSQLUSERNAME和MYSQLPASSWORD环境变量中检索数据库用户名和密码的。

执行数据库事务接口

在处理数据库连接服务时，测试相对困难。原因在于Go的鸭子模型导致在运行时模拟很不方便。我建议在使用数据库时使用存储接口，使用这样的方式模拟数据库事务接口同样可行，在后续章节会讨论这一点。本节将着眼于对数据库事务的操作。

我们将改写上一节的create和query文件，最终的输出很类似，但是create和query操作将包含在事务中。

实践

建立 transaction.go：

```
package dbinterface

import "database/sql"

// DB是sql.DB或sql.Transaction满足的接口
type DB interface {
    Exec(query string, args ...interface{}) (sql.Result, error)
    Prepare(query string) (*sql.Stmt, error)
    Query(query string, args ...interface{}) (*sql.Rows, error)
    QueryRow(query string, args ...interface{}) *sql.Row
}

// Transaction可以执行任何 Query, Commit, Rollback, 和 Stmt 操作
type Transaction interface {
    DB
    Commit() error
    Rollback() error
    Stmt(stmt *sql.Stmt) *sql.Stmt
}
```

建立 create.go：

```
package dbinterface

import _ "github.com/go-sql-driver/mysql"

// Create建立example表并填充数据
func Create(db DB) error {
    // create the database
    if _, err := db.Exec("CREATE TABLE example (name VARCHAR(20), created DATETI
ME)"); err != nil {
```

```
        return err
    }

    if _, err := db.Exec(`INSERT INTO example (name, created) values ("Aaron", N
OW())`); err != nil {
        return err
    }

    return nil
}
```

建立 exec.go：

```
package dbinterface

func Exec(db DB) error {

    // 依然不推荐这么写 因为没有处理所返回的可能存在的错误
    defer db.Exec("DROP TABLE example")

    if err := Create(db); err != nil {
        return err
    }

    if err := Query(db); err != nil {
        return err
    }
    return nil
}
```

建立 main.go：

```
package main

import (
    "github.com/agtorre/go-cookbook/chapter5/database"
    "github.com/agtorre/go-cookbook/chapter5/dbinterface"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := database.Setup()
    if err != nil {
        panic(err)
    }
```

```
    tx, err := db.Begin()
    if err != nil {
        panic(err)
    }

    defer tx.Rollback()

    if err := dbinterface.Exec(db); err != nil {
        panic(err)
    }
    if err := tx.Commit(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
Results:
    Name: Aaron
    Created: 2017-02-16 20:00:00 +0000 UTC
```

说明

本节与上一节以非常相似的方式工作，演示了使用事务并生成适用于sql.DB连接和sql.Transaction对象的通用数据库函数。我们会在第八章看到对这些模拟接口的测试使用。

SQL的连接池速率限制和超时

尽管database/sql包提供了连接池速率限制和超时，调整默认值以更好地适应数据库配置通常很重要。当你在微服务上进行水平扩展并且不希望保持与数据库的过多活动连接时，这可能很重要。

实践

建立 pools.go：

```
package pools

import (
    "database/sql"
    "fmt"
    "os"

    _ "github.com/go-sql-driver/mysql"
)

func Setup() (*sql.DB, error) {
    db, err := sql.Open("mysql",
        fmt.Sprintf("%s:%s@/gocookbook?parseTime=true", os.Getenv("MYSQLUSERNAM
E"), os.Getenv("MYSQLPASSWORD")))
    if err != nil {
        return nil, err
    }

    // 仅开放24个连接
    db.SetMaxOpenConns(24)

    // MaxIdleConns不可以比SetMaxOpenConns的值小 否则会将SetMaxOpenConns的值作为
默认值
    db.SetMaxIdleConns(24)

    return db, nil
}
```

建立 timeout.go：

```
package pools

import (
```

```
    "context"
    "time"
)

// ExecWithTimeout 使用context来实现超时
func ExecWithTimeout() error {
    db, err := Setup()
    if err != nil {
        return err
    }

    ctx := context.Background()

    // 我们希望立刻超时
    ctx, can := context.WithDeadline(ctx, time.Now())

    // 在函数完成后调用cancel
    defer can()

    // 以当前context为参数
    _, err = db.BeginTx(ctx, nil)
    return err
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter5/pools"

func main() {
    if err := pools.ExecWithTimeout(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
panic: context deadline exceeded
goroutine 1 [running]:
main.main()
/go/src/github.com/agtorre/gocookbook/chapter5/pools/example/main.go:7 +0x4e
exit status 2
```

说明

限制数据库连接池深度非常有用，这将保护数据库不会超载，不过更重要的是考虑context在这里的使用。如果你强制限制一定数量的连接并严格的基于上下文的超时，就像我们在示例中所做的那样，在某些情况下，会出现有一些请求经常超时并试图建立过多连接导致程序重载。

这个新加入sql包的函数使得包括查询在内的请求共享超时更加简单。

使用全局配置对象传递给Setup()函数是有意义的，当然我们在这里使用的是环境变量。

操作Redis

本文将探讨Redis作为非关系数据存储的一种形式，并展示Go等语言如何与这些服务进行交互。

由于Redis通过简单的接口支持键值存储，因此它是会话存储或具有持续时间的临时数据的理想选择。为Redis中存储的数据指定超时的能力非常有价值。本节将探讨从配置，查询到使用自定义排序的基本Redis用法。

实践

获取第三方库：

```
go get gopkg.in/redis.v5
```

建立 config.go：

```
package redis

import (
    "os"

    redis "gopkg.in/redis.v5"
)

// Setup 初始化 redis 连接
func Setup() (*redis.Client, error) {
    client := redis.NewClient(&redis.Options{
        Addr:     "localhost:6379",
        Password: os.Getenv("REDISPASSWORD"),
        DB:       0, // 使用默认 DB
    })

    // 命令返回“PONG”，测试连接是否存活
    _, err := client.Ping().Result()
    return client, err
}
```

建立 exec.go：

```
package redis

import (
```

```
    "fmt"
    "time"

    redis "gopkg.in/redis.v5"
)

// Exec 执行一些redis操作
func Exec() error {
    conn, err := Setup()
    if err != nil {
        return err
    }

    c1 := "value"
    // 我们可以把任何想要的类型存入key所对应的值
    // 当前存入的是key到期的时间
    conn.Set("key", c1, 5*time.Second)

    var result string
    if err := conn.Get("key").Scan(&result); err != nil {
        switch err {
        // 这意味着key找不到对应的值
        case redis.Nil:
            return nil
        default:
            return err
        }
    }

    fmt.Println("result =", result)

    return nil
}
```

建立 sort.go：

```
package redis

import (
    "fmt"

    redis "gopkg.in/redis.v5"
)

// Sort 执行排序redis操作
```

```
func Sort() error {
    conn, err := Setup()
    if err != nil {
        return err
    }

    if err := conn.LPush("list", 1).Err(); err != nil {
        return err
    }
    if err := conn.LPush("list", 3).Err(); err != nil {
        return err
    }
    if err := conn.LPush("list", 2).Err(); err != nil {
        return err
    }

    res, err := conn.Sort("list", redis.Sort{Order: "ASC"}).Result()
    if err != nil {
        return err
    }
    fmt.Println(res)
    conn.Del("list")
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter5/redis"

func main() {
    if err := redis.Exec(); err != nil {
        panic(err)
    }

    if err := redis.Sort(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
result = value
[1 2 3]
```

说明

可以看到Redis的操作与MySQL很相似，Scan()等函数都遵循相同的约定从Redis读取数据映射Go类型。选择合适的第三方库可能具有挑战性，我建议定期调查可用的内容。

本节使用redis包进行基本设置和查询，执行更复杂的排序功能。与database/sql一样，你可以通过控制超时，设置连接池大小等形式设置其他配置。Redis本身还提供了许多其他功能，包括Redis群集支持，Zscore和计数器对象，分布式锁定等。

与前面的章节一样，我建议使用配置对象，它存储Redis设置和配置详细信息，以便于设置操作和提高安全性。

操作MongoDB

mgo包可以很好地满足我们对MongoDB的日常操作需要。本节将以与Redis和MySQL类似的方式创建连接，存储和检索对象。

实践

获取第三方库：

```
go get gopkg.in/mgo.v2
```

建立 config.go：

```
package mongodb

import mgo "gopkg.in/mgo.v2"

func Setup() (*mgo.Session, error) {
    session, err := mgo.Dial("localhost")
    if err != nil {
        return nil, err
    }
    return session, nil
}
```

建立 exec.go：

```
package mongodb

import (
    "fmt"

    "gopkg.in/mgo.v2/bson"
)

type State struct {
    Name       string `bson:"name"`
    Population int    `bson:"pop"`
}

// Exec 演示创建和查询
func Exec() error {
    db, err := Setup()
```

```
    if err != nil {
        return err
    }

    conn := db.DB("gocookbook").C("example")

    // 我们可以一次性插入多条
    if err := conn.Insert(&State{"Washington", 7062000}, &State{"Oregon", 397000
0}); err != nil {
        return err
    }

    var s State
    if err := conn.Find(bson.M{"name": "Washington"}).One(&s); err != nil {
        return err
    }

    if err := conn.DropCollection(); err != nil {
        return err
    }

    fmt.Printf("State: %#v\n", s)
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter5/redis"

func main() {
    if err := redis.Exec(); err != nil {
        panic(err)
    }

    if err := redis.Sort(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
State: mongodb.State{Name:"Washington", Population:7062000}
```

说明

mgo包还提供连接池，以及调整和配置与mongodb数据库连接的许多方法。本节的例子非常基础，但说明了操作基于文档的数据库是多么容易。该软件包实现了BSON数据类型，与常见的JSON非常相似。

创建存储接口以实现数据可移植性

为了便于模拟,使用外部存储接口时对操作进行抽象会很有帮助。这可以在更改存储后端时提高可移植性。但如果需要在事务内执行多个操作，这样做就比较尴尬，此时进行复合操作或允许通过上下文对象或其他函数参数传递会比较好。

本节将实现一个非常简单的接口来处理MongoDB中的操作。我们会使用一个接口来持久化和检索对象。

实践

建立 storage.go：

```
package storage

import "context"

type Item struct {
    Name  string
    Price int64
}

// Storage是我们的存储接口 将使用Mongo存储实现它
type Storage interface {
    GetByName(context.Context, string) (*Item, error)
    Put(context.Context, *Item) error
}
```

建立 mongoconfig.go：

```
package storage

import mgo "gopkg.in/mgo.v2"

// MongoStorage实现了storage 接口
type MongoStorage struct {
    *mgo.Session
    DB         string
    Collection string
}

// NewMongoStorage 初始化MongoStorage
func NewMongoStorage(connection, db, collection string) (*MongoStorage, error) {
    session, err := mgo.Dial("localhost")
```

```
    if err != nil {
        return nil, err
    }
    ms := MongoStorage{
        Session:    session,
        DB:         db,
        Collection: collection,
    }
    return &ms, nil
}
```

建立 mongointerface.go：

```
package storage

import (
    "context"

    "gopkg.in/mgo.v2/bson"
)

// GetByName 查询mongodb以获取具有正确名称的item
func (m *MongoStorage) GetByName(ctx context.Context, name string) (*Item, error
) {
    c := m.Session.DB(m.DB).C(m.Collection)
    var i Item
    if err := c.Find(bson.M{"name": name}).One(&i); err != nil {
        return nil, err
    }

    return &i, nil
}

// Put 添加一个item到mongo 中
func (m *MongoStorage) Put(ctx context.Context, i *Item) error {
    c := m.Session.DB(m.DB).C(m.Collection)
    return c.Insert(i)
}
```

建立 exec.go：

```
package storage

import (
    "context"
```

```
    "fmt"
)

// Exec 初始化存储 storage然后使用存储接口执行操作
func Exec() error {
    m, err := NewMongoStorage("localhost", "gocookbook", "items")
    if err != nil {
        return err
    }
    if err := PerformOperations(m); err != nil {
        return err
    }

    if err := m.Session.DB(m.DB).C(m.Collection).DropCollection(); err != nil {
        return err
    }

    return nil
}

// PerformOperations 演示创建并存入一个item然后对其进行查询查询
func PerformOperations(s Storage) error {
    ctx := context.Background()
    i := Item{Name: "candles", Price: 100}
    if err := s.Put(ctx, &i); err != nil {
        return err
    }

    candles, err := s.GetByName(ctx, "candles")
    if err != nil {
        return err
    }
    fmt.Printf("Result: %#v\n", candles)
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter5/storage"

func main() {
    if err := storage.Exec(); err != nil {
        panic(err)
```

```
    }
}
```

这会输出：

```
Result: &storage.Item{Name:"candles", Price:100}
```

说明

在PerformOperation中我们可以看到，此函数将存储接口作为参数。这意味着我们可以动态替换底层存储，而无需修改此函数。如果想要将存储连接到单独的API以便使用和修改它将很简单。

我们为这些接口添加了一个context，以增加额外的灵活性，并允许接口处理超时。将应用程序逻辑与底层存储分离可提供多种好处，但是很难选择正确的位置来处理边界，这在不同的应用程序中会有很大差异。

Web客户端和APIs

本章会覆盖以下内容：

- 使用http.Client
- 调用REST API
- 并发操作客户端请求
- 使用OAuth2
- 实现OAuth2令牌存储接口
- 封装http请求客户端
- 理解GRPC的使用

介绍

使用API和编写Web客户端并不是一个轻松的话题。不同的API具有不同类型的授权，身份验证和协议。我们将探索http.Client结构对象，使用OAuth2客户端和令牌存储，并使用REST接口完成GRPC。

到本章结束时，你会了解如何与第三方或内部API进行交互，并了解常见操作能够使用哪些模式。

使用http.Client

标准库的net/http包的http.Client结构十分灵活，可用户处理HTTP API。此结构具有独立的传输函数，并且相对简单，可以短路请求，修改每个客户端操作的请求头以及处理REST操作。 创建http请求是常见的操作，本节将从操作http.Client对象的基础开始。

实践

建立 client.go：

```
package client

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

// Setup 设置http.Client并重新定义全局DefaultClient
func Setup(isSecure, nop bool) *http.Client {
    c := http.DefaultClient

    // 有时为了方便测试，我们需要关闭SSL验证
    if !isSecure {
        c.Transport = &http.Transport{
            TLSClientConfig: &tls.Config{
                InsecureSkipVerify: false,
            },
        }
    }
    if nop {
        c.Transport = &NopTransport{}
    }
    http.DefaultClient = c
    return c
}

// NopTransport 没有任何操作的传输
type NopTransport struct {
}

// RoundTrip 实现了 RoundTripper 接口
func (n *NopTransport) RoundTrip(*http.Request) (*http.Response, error) {
    // 注意这里只为StatusCode赋值
```

```
    return &http.Response{StatusCode: http.StatusTeapot}, nil
}
```

建立 exec.go：

```
package client

import (
    "fmt"
    "net/http"
)

// DoOps 接收client参数 然后请求 google.com
func DoOps(c *http.Client) error {
    resp, err := c.Get("http://www.google.com")
    if err != nil {
        return err
    }
    fmt.Println("results of DoOps:", resp.StatusCode)

    return nil
}

// DefaultGetGolang 使用默认的client请求 golang.org
func DefaultGetGolang() error {
    resp, err := http.Get("https://www.golang.org")
    if err != nil {
        return err
    }
    fmt.Println("results of DefaultGetGolang:", resp.StatusCode)
    return nil
}
```

建立 storage.go：

```
package client

import (
    "fmt"
    "net/http"
)

// Controller 包含有匿名字段*http.Client
type Controller struct {
    *http.Client
```

```
}

func (c *Controller) DoOps() error {
    resp, err := c.Client.Get("http://www.google.com")
    if err != nil {
        return err
    }
    fmt.Println("results of client.DoOps", resp.StatusCode)
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter6/client"

func main() {

    cli := client.Setup(true, false)

    if err := client.DefaultGetGolang(); err != nil {
        panic(err)
    }

    if err := client.DoOps(cli); err != nil {
        panic(err)
    }

    c := client.Controller{Client: cli}
    if err := c.DoOps(); err != nil {
        panic(err)
    }

    client.Setup(true, true)

    if err := client.DefaultGetGolang(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
results of DefaultGetGolang: 200
results of DoOps: 200
```

```
results of client.DoOps 200
results of DefaultGetGolang: 418
```

说明

net/http包公开了一个DefaultClient包变量，该变量使用默认配置操作Do，GET，POST等。我们的Setup函数返回一个客户端，该客户端在设置时，灵活的修改可以通过调整实现的RoundTripper接口实现。

通过调整Setup函数的nop参数，可以方便的对client的配置进行调整，这对测试来说很重要。

调用REST API

为REST API编写客户端不仅可以帮助你更好地理解相关API，还可以为使用该API的所有应用程序提供有用的工具。本节将探索构建客户端并展示一些在日常开发中可以使用的策略。

本节我们假设身份验证由基本auth处理，同时也支持token令牌。为简单起见，假设我们的API公开了一个端点GetGoogle()，该端点将执行GET获得的状态码返回给https//www.google.com。

实践

建立 client.go：

```
package rest

import "net/http"

// APIClient是我们自定义的client
type APIClient struct {
    *http.Client
}

// NewAPIClient 使用自定义的Transport初始化client
func NewAPIClient(username, password string) *APIClient {
    t := http.Transport{}
    return &APIClient{
        Client: &http.Client{
            Transport: &APITransport{
                Transport: &t,
                username:  username,
                password:  password,
            },
        },
    }
}

// GetGoogle 是一个API调用 我们抽象出API操作
func (c *APIClient) GetGoogle() (int, error) {
    resp, err := c.Get("http://www.google.com")
    if err != nil {
        return 0, err
    }
    return resp.StatusCode, nil
}
```

建立 transport.go：

```
package rest

import "net/http"

// APITransport 为每个请求执行SetBasicAuth函数
// 实现了RoundTripper接口
type APITransport struct {
    *http.Transport
    username, password string
}

// RoundTrip 在默认传输之前添加基础的auth
func (t *APITransport) RoundTrip(req *http.Request) (*http.Response, error) {
    req.SetBasicAuth(t.username, t.password)
    return t.Transport.RoundTrip(req)
}
```

建立 exec.go：

```
package rest

import "fmt"

// Exec 创建API客户端并使用其GetGoogle方法，然后打印结果
func Exec() error {
    c := NewAPIClient("username", "password")

    StatusCode, err := c.GetGoogle()
    if err != nil {
        return err
    }
    fmt.Println("Result of GetGoogle:", StatusCode)
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter6/rest"

func main() {
```

```
    if err := rest.Exec(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
Result of GetGoogle: 200
```

说明

本节演示了如何使用Transport接口隐藏诸如身份验证，令牌刷新等逻辑。它还演示了通过方法公开API调用。 如果我们需要实现用户API这样的东西，可以考虑：

```
type API interface{
    GetUsers() (Users, error)
    CreateUser(User) error
    UpdateUser(User) error
    DeleteUser(User)
}
```

如果你已经读过第5章的”创建存储接口以实现数据可移植性”，会发现可以考虑使用同样的方式进行思考。通过接口，特别是常见的接口，如RoundTripper接口，这种组合为编写API提供了很大的灵活性。此外，像我们之前所做的那样编写顶级接口并直接传递接口而不是client可能很有用。

并发操作客户端请求

在Go中，并行执行客户端请求相对简单。 在下面示例中，我们将使用客户端使用Go缓冲通道执行多个URL请求。 响应和错误都将转到一个单独的通道，任何有权访问客户端的人都可以轻松访问。

在本节中，创建客户端，读取通道以及处理响应和错误都将在main.go文件中完成。

实践

建立 config.go：

```
package async

import "net/http"

// NewClient 建立一个新的client并为其分配通道
func NewClient(client *http.Client, bufferSize int) *Client {
    respch := make(chan *http.Response, bufferSize)
    errch := make(chan error, bufferSize)
    return &Client{
        Client: client,
        Resp:   respch,
        Err:    errch,
    }
}

type Client struct {
    *http.Client
    Resp chan *http.Response
    Err  chan error
}

// AsyncGet 执行Get然后将resp/error返回到适当的通道
func (c *Client) AsyncGet(url string) {
    resp, err := c.Get(url)
    if err != nil {
        c.Err <- err
        return
    }
    c.Resp <- resp
}
```

建立 exec.go：

```
package async

// FetchAll 遍历请求所有的url
func FetchAll(urls []string, c *Client) {
    for _, url := range urls {
        go c.AsyncGet(url)
    }
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter6/async"
)

func main() {
    urls := []string{
        "https://www.google.com",
        "https://golang.org",
        "https://www.github.com",
    }
    c := async.NewClient(http.DefaultClient, len(urls))
    async.FetchAll(urls, c)

    for i := 0; i < len(urls); i++ {
        select {
        case resp := <-c.Resp:
            fmt.Printf("Status received for %s: %d\n", resp.Request.URL, resp.St
atusCode)
        case err := <-c.Err:
            fmt.Printf("Error received: %s\n", err)
        }
    }
}
```

这会输出：

```
Status received for https://www.google.com: 200
Status received for https://golang.org: 200
```

```
Status received for https://github.com/: 200
```

说明

该示例创建了一个框架，用于使用单个客户端以扇出异步方式处理请求。它尝试尽可能快地检索尽可能多的URL。 在许多情况下，你可能希望通过类似工作池的方式进一步限制此操作。在特定存储或检索接口时这些异步Go例程也是有借鉴意义的。

这里还使用case语句处理多个通道。我们无需担心处理锁定问题，因为我们非常清楚将收到多少响应。如果放弃某些响应处理，那么另一种选择就是超时。

使用OAuth2

OAuth2是一种用于API通信的相对常见的协议。golang.org/x/oauth2 包提供了非常灵活的OAuth2操作，它的子包能为Facebook，Google和GitHub等各种提供商提供支持。

本节将演示如何为GitHub的搭建OAuth2和一些基本操作。

实践

获取支持库：

```
go get golang.org/x/oauth2
```

配置OAuth Client

设置 Client ID and Secret：

```
1. export GITHUB_CLIENT="your_client"
2. export GITHUB_SECRET="your_secret"
```

可以在https://developer.github.com/v3/ 查看相关操作文档。

建立 config.go：

```
package oauthcli

import (
    "context"
    "fmt"
    "os"

    "golang.org/x/oauth2"
    "golang.org/x/oauth2/github"
)

// 设置oauth2.Config
// 需要在环境变量中设置GITHUB_CLIENT GITHUB_SECRET
func Setup() *oauth2.Config {
    return &oauth2.Config{
        ClientID:     os.Getenv("GITHUB_CLIENT"),
        ClientSecret: os.Getenv("GITHUB_SECRET"),
        Scopes:       []string{"repo", "user"},
        Endpoint:     github.Endpoint,
```

```
    }
}

// GetToken 检索github oauth2令牌
func GetToken(ctx context.Context, conf *oauth2.Config) (*oauth2.Token, error) {
    url := conf.AuthCodeURL("state")
    fmt.Printf("Type the following url into your browser and follow the directio
ns on screen: %v\n", url)
    fmt.Println("Paste the code returned in the redirect URL and hit Enter:")

    var code string
    if _, err := fmt.Scan(&code); err != nil {
        return nil, err
    }
    return conf.Exchange(ctx, code)
}
```

建立 exec.go：

```
package oauthcli

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

// GetUsers 使用oauth2获取用户信息
func GetUsers(client *http.Client) error {
    url := fmt.Sprintf("https://api.github.com/user")

    resp, err := client.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    fmt.Println("Status Code from", url, ":", resp.StatusCode)
    io.Copy(os.Stdout, resp.Body)
    return nil
}
```

建立 main.go：

```
package main

import (
    "context"

    "github.com/agtorre/go-cookbook/chapter6/oauthcli"
)

func main() {
    ctx := context.Background()
    conf := oauthcli.Setup()

    tok, err := oauthcli.GetToken(ctx, conf)
    if err != nil {
        panic(err)
    }
    client := conf.Client(ctx, tok)

    if err := oauthcli.GetUsers(client); err != nil {
        panic(err)
    }

}
```

这会输出：

```
Visit the URL for the auth dialog:
https://github.com/login/oauth/authorize?
access_type=offline&client_id=
<your_id>&response_type=code&scope=repo+user&state=state
Paste the code returned in the redirect URL and hit Enter:
<your_code>
Status Code from https://api.github.com/user: 200
{<json_payload>}
```

说明

标准OAuth2流是基于重定向的，会以服务器重定向到你指定的端点作为结束。然后你所在的端负责获取返回值并将其交换为令牌。示例通过允许我们使用诸如https://localhost 或https://a-domain-you-own 之类的URL并手动复制/粘贴代码然后按Enter键来绕过该要求。交换令牌后，客户端将根据需要自行刷新令牌。

重要的是要注意我们不以任何方式存储令牌。 如果程序崩溃，则必须重新交换令牌。同样要注意，除非刷新令牌过期，丢失或损坏，否则我们只需要显式检索一次令牌。配置客户端后，它

应该能够为其授权的API执行所有典型的HTTP操作，并且具有适当的范围。

实现OAuth2令牌存储接口

在上一个小节中，我们为客户端检索了一个令牌并执行了API请求。这种方法的缺点是我们的令牌没有长期存储空间。例如，在HTTP服务器中，我们希望在请求之间保持一致的令牌存储。

本节将探讨修改OAuth2客户端以在请求之间存储令牌并使用密钥检索它们。为简单起见，此密钥将是一个文件，但它也可以是数据库，Redis等。

实践

建立 config.go：

```
package oauthstore

import (
    "context"
    "net/http"

    "golang.org/x/oauth2"
)

// Config 包含了 oauth2.Config和 Storage接口
type Config struct {
    *oauth2.Config
    Storage
}

// Exchange 在接收到令牌后将其存储
func (c *Config) Exchange(ctx context.Context, code string) (*oauth2.Token, erro
r) {
    token, err := c.Config.Exchange(ctx, code)
    if err != nil {
        return nil, err
    }
    if err := c.Storage.SetToken(token); err != nil {
        return nil, err
    }
    return token, nil
}

// TokenSource 可以传递已被存储的令牌
// 或当新令牌被接收时将其转换为oauth2.TokenSource
func (c *Config) TokenSource(ctx context.Context, t *oauth2.Token) oauth2.TokenS
ource {
```

```
    return StorageTokenSource(ctx, c, t)
}

// Client 附加到TokenSource
func (c *Config) Client(ctx context.Context, t *oauth2.Token) *http.Client {
    return oauth2.NewClient(ctx, c.TokenSource(ctx, t))
}
```

建立 tokensource.go：

```
package oauthstore

import (
    "context"

    "golang.org/x/oauth2"
)

type storageTokenSource struct {
    *Config
    oauth2.TokenSource
}

// Token满足TokenSource接口
func (s *storageTokenSource) Token() (*oauth2.Token, error) {
    if token, err := s.Config.Storage.GetToken(); err == nil && token.Valid() {
        return token, err
    }
    token, err := s.TokenSource.Token()
    if err != nil {
        return token, err
    }
    if err := s.Config.Storage.SetToken(token); err != nil {
        return nil, err
    }
    return token, nil
}

// StorageTokenSource 将由config.TokenSource方法调用
func StorageTokenSource(ctx context.Context, c *Config, t *oauth2.Token) oauth2.
TokenSource {
    if t == nil || !t.Valid() {
        if tok, err := c.Storage.GetToken(); err == nil {
            t = tok
        }
```

```
    }
    ts := c.Config.TokenSource(ctx, t)
    return &storageTokenSource{c, ts}
}
```

建立 storage.go：

```
package oauthstore

import (
    "context"
    "fmt"

    "golang.org/x/oauth2"
)

// Storage 是我们的通用存储接口
type Storage interface {
    GetToken() (*oauth2.Token, error)
    SetToken(*oauth2.Token) error
}

// GetToken 检索github oauth2令牌
func GetToken(ctx context.Context, conf Config) (*oauth2.Token, error) {
    token, err := conf.Storage.GetToken()
    if err == nil && token.Valid() {
        return token, err
    }
    url := conf.AuthCodeURL("state")
    fmt.Printf("Type the following url into your browser and follow the directio
ns on screen: %v\n", url)
    fmt.Println("Paste the code returned in the redirect URL and hit Enter:")

    var code string
    if _, err := fmt.Scan(&code); err != nil {
        return nil, err
    }
    return conf.Exchange(ctx, code)
}
```

建立 filestorage.go：

```
package oauthstore

import (
```

```
    "encoding/json"
    "errors"
    "os"
    "sync"

    "golang.org/x/oauth2"
)

// FileStorage 满足storage 接口
type FileStorage struct {
    Path string
    mu   sync.RWMutex
}

// GetToken 从文件中检索令牌
func (f *FileStorage) GetToken() (*oauth2.Token, error) {
    f.mu.RLock()
    defer f.mu.RUnlock()
    in, err := os.Open(f.Path)
    if err != nil {
        return nil, err
    }
    defer in.Close()
    var t *oauth2.Token
    data := json.NewDecoder(in)
    return t, data.Decode(&t)
}

// SetToken 将令牌存储在文件中
func (f *FileStorage) SetToken(t *oauth2.Token) error {
    if t == nil || !t.Valid() {
        return errors.New("bad token")
    }

    f.mu.Lock()
    defer f.mu.Unlock()
    out, err := os.OpenFile(f.Path, os.O_RDWR|os.O_CREATE|os.O_TRUNC, 0755)
    if err != nil {
        return err
    }
    defer out.Close()
    data, err := json.Marshal(&t)
    if err != nil {
        return err
    }
```

```
    _, err = out.Write(data)
    return err
}
```

建立 main.go：

```
package main

import (
    "context"
    "io"
    "os"

    "github.com/agtorre/go-cookbook/chapter6/oauthstore"

    "golang.org/x/oauth2"
    "golang.org/x/oauth2/github"
)

func main() {
    conf := oauthstore.Config{
        Config: &oauth2.Config{
            ClientID:     os.Getenv("GITHUB_CLIENT"),
            ClientSecret: os.Getenv("GITHUB_SECRET"),
            Scopes:       []string{"repo", "user"},
            Endpoint:     github.Endpoint,
        },
        Storage: &oauthstore.FileStorage{Path: "token.txt"},
    }
    ctx := context.Background()
    token, err := oauthstore.GetToken(ctx, conf)
    if err != nil {
        panic(err)
    }

    cli := conf.Client(ctx, token)
    resp, err := cli.Get("https://api.github.com/user")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

这会输出：

```
$ go run main.go
Visit the URL for the auth dialog:
https://github.com/login/oauth/authorize?
access_type=offline&client_id=
<your_id>&response_type=code&scope=repo+user&state=state
Paste the code returned in the redirect URL and hit Enter:
<your_code>
{<json_payload>}

$ go run main.go
{<json_payload>}
```

说明

本节将令牌的内容存储到文件中并从文件中检索令牌的内容。 如果是第一次运行，它必须执行整个代码交换，但后续运行将重用访问令牌，如果有可用，它将使用刷新令牌刷新。

此代码中目前无法区分用户/令牌，但这可以通过使用cookie作为文件名或数据库中的行的键来实现。让我们来看看这段代码的作用：

- config.go文件封装了标准的OAuth2配置。对于涉及检索令牌的每种方法，首先检查本地存储中是否有有效令牌。 如果没有，使用标准配置检索一个，然后存储它。
- tokensource.go文件实现了与Config配对的自定义TokenSource接口。 与Config类似，总是首先尝试从文件中检索令牌，否则使用新令牌设置它。
- storage.go文件是Config和TokenSource使用的存储接口。 它只定义了两个方法和辅助函数来引导OAuth2基于代码的流程，类似于我们在上一个方法中所做的那样，但是如果已经存在具有有效令牌的文件，则将使用它。
- filestorage.go文件实现存储接口。 当我们存储新令牌时，首先截断该文件并编写令牌结构的JSON表示。 否则，我们解析文件并返回令牌。

封装http请求客户端

在2015年，TomásSenart就使用接口包装http.Client结构进行了精彩的探讨，你可以在https//github.com/gophercon/2015-talks上找到更多相关信息。 本节从他的想法中获取并演示了一个与http.Client结构的Transport接口相同的示例，类似于我们之前的示例，为REST API编写客户端。

本节将为标准的http.Client结构实现日志记录和基本身份验证中间件。 它还包括一个decorate函数，可以在需要使用各种中间件时使用。

实践

建立 config.go：

```
package decorator

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

// Setup 初始化客户端接口
func Setup() *http.Client {
    c := http.Client{}

    t := Decorate(&http.Transport{},
        Logger(log.New(os.Stdout, "", 0)),
        BasicAuth("username", "password"),
    )
    c.Transport = t
    return &c
}
```

建立 decorator.go：

```
package decorator

import "net/http"

// TransportFunc 实现了 RountTripper 接口
type TransportFunc func(*http.Request) (*http.Response, error)
```

```
// RoundTrip 仅调用原始的函数
func (tf TransportFunc) RoundTrip(r *http.Request) (*http.Response, error) {
    return tf(r)
}

// Decorator 是一个方便的函数来表示我们的中间件内部功能
type Decorator func(http.RoundTripper) http.RoundTripper

// Decorate 用于包装所有中间件
func Decorate(t http.RoundTripper, rts ...Decorator) http.RoundTripper {
    decorated := t
    for _, rt := range rts {
        decorated = rt(decorated)
    }
    return decorated
}
```

建立 middleware.go：

```
package decorator

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

// Logger 日志中间件
func Logger(l *log.Logger) Decorator {
    return func(c http.RoundTripper) http.RoundTripper {
        return TransportFunc(func(r *http.Request) (*http.Response, error) {
            start := time.Now()
            l.Printf("started request to %s at %s", r.URL, start.Format("2006-01
-02 15:04:05"))
            resp, err := c.RoundTrip(r)
            l.Printf("completed request to %s in %s", r.URL, time.Since(start))
            return resp, err
        })
    }
}

// BasicAuth 基础身份认证中间件
func BasicAuth(username, password string) Decorator {
    return func(c http.RoundTripper) http.RoundTripper {
        return TransportFunc(func(r *http.Request) (*http.Response, error) {
```

```
            r.SetBasicAuth(username, password)
            resp, err := c.RoundTrip(r)
            return resp, err
        })
    }
}
```

建立 exec.go：

```
package decorator

import "fmt"

// Exec 创建一个客户端，调用google.com然后打印响应
func Exec() error {
    c := Setup()

    resp, err := c.Get("https://www.google.com")
    if err != nil {
        return err
    }
    fmt.Println("Response code:", resp.StatusCode)
    return nil
}
```

建立 main.go：

```
package main

import "github.com/agtorre/go-cookbook/chapter6/decorator"

func main() {
    if err := decorator.Exec(); err != nil {
        panic(err)
    }
}
```

这会输出：

```
started request to https://www.google.com at 2017-01-01 13:38:42
completed request to https://www.google.com in 194.013054ms
Response code: 200
```

本节将闭包作为一流的公民和接口。主要技巧是具有功能实现接口。这允许封装具有由函数实现的接口的结构实现的接口。

middleware.go文件包含两个示例客户端中间件函数。 这些可以扩展为包含其他中间件，例如更复杂的身份验证。 本示例也可以与之前的示例结合使用，以生成可以通过其他中间件进行扩展的OAuth2客户端。

Decorator函数是一种便利的封装函数，可以这样比较下：

```
Decorate(RoundTripper, Middleware1, Middleware2, etc)
vs
var t RoundTripper
t = Middleware1(t)
t = Middleware2(t)
etc
```

与封装客户端相比，这种方法的优点是我们可以保持接口解耦。如果你想要一个功能齐全的客户端，你还需要实现GET，POST和PostForm等方法。

理解GRPC的使用

GRPC是一个使用缓冲协议和HTTP/2构建的高性能RPC框架。在Go中创建GRPC客户端与使用Go HTTP客户端有很多相同的复杂性。

为了演示基本的客户端使用情况，最简单的方法是实现一个服务器。这个配方将创建一个欢迎服务，它接受问候和名称并返回欢迎语句。

本文不会探讨有关GRPC的一些细节，例如流媒体。

实践

安装第三方库：

安装GRPC

https://github.com/grpc/grpc/blob/master/INSTALL.md.

```
go get github.com/golang/protobuf/proto
go get github.com/golang/protobuf/protoc-gen-go
```

建立 greeter.proto：

```
syntax = "proto3";

package greeter;

service GreeterService{
    rpc Greet(GreetRequest) returns (GreetResponse) {}
}

message GreetRequest {
    string greeting = 1;
    string name = 2;
}

message GreetResponse{
    string response = 1;
}
```

运行

```
protoc --go_out=plugins=grpc:. greeter.proto
```

建立 server.go：

```
package main

import (
    "fmt"
    "net"

    "github.com/agtorre/go-cookbook/chapter6/grpc/greeter"
    "google.golang.org/grpc"
)

func main() {
    grpcServer := grpc.NewServer()
    greeter.RegisterGreeterServiceServer(grpcServer, &Greeter{Exclaim: true})
    lis, err := net.Listen("tcp", ":4444")
    if err != nil {
        panic(err)
    }
    fmt.Println("Listening on port :4444")
    grpcServer.Serve(lis)
}
```

建立 greeter.go：

```
package main

import (
    "fmt"

    "github.com/agtorre/go-cookbook/chapter6/grpc/greeter"
    "golang.org/x/net/context"
)

// Greeter 实现了protoc生成的接口
type Greeter struct {
    Exclaim bool
}

// Greet 实现grpc Greet
func (g *Greeter) Greet(ctx context.Context, r *greeter.GreetRequest) (*greeter.
GreetResponse, error) {
    msg := fmt.Sprintf("%s %s", r.GetGreeting(), r.GetName())
    if g.Exclaim {
        msg += "!"
```

```
    } else {
        msg += "."
    }
    return &greeter.GreetResponse{Response: msg}, nil
}
```

建立 client.go：

```
package main

import (
    "context"
    "fmt"

    "github.com/agtorre/go-cookbook/chapter6/grpc/greeter"
    "google.golang.org/grpc"
)

func main() {
    conn, err := grpc.Dial(":4444", grpc.WithInsecure())
    if err != nil {
        panic(err)
    }
    defer conn.Close()

    client := greeter.NewGreeterServiceClient(conn)

    ctx := context.Background()
    req := greeter.GreetRequest{Greeting: "Hello", Name: "Reader"}
    resp, err := client.Greet(ctx, &req)
    if err != nil {
        panic(err)
    }
    fmt.Println(resp)

    req.Greeting = "Goodbye"
    resp, err = client.Greet(ctx, &req)
    if err != nil {
        panic(err)
    }
    fmt.Println(resp)
}
```

分别运行server.go和greeter.go，然后在另一个命令行运行client.go，这会输出：

```
response:"Hello Reader!"
response:"Goodbye Reader!"
```

说明

GRPC服务器设置为侦听端口4444.一旦客户端连接，它就可以发送请求并从服务器接收响应。请求，响应和支持的方法的结构由protoc生成的。实际上，当针对GRPC服务器进行集成时，应该提供.proto文件，该文件可用于自动生成客户端 。

除了客户端之外，protoc命令还会为服务器生成存根，所需的只是填写实现细节。 生成的Go代码也具有JSON标记，并且可以为JSON REST服务重用相同的结构。 我们的代码设置了一个不安全的客户端。 要安全地处理GRPC，你需要使用SSL证书。

网络服务

本章会覆盖以下内容：

- 处理Web请求
- 使用闭包进行状态处理
- 请求参数验证
- 内容渲染
- 使用中间件
- 构建反向代理
- 将GRPC导出为JSON API

介绍

开箱即用的特性使得Go是编写Web应用程序的绝佳选择。标准库中的net/http和html/template包对全功能现代Web提供了极大的便利。虽然标准库功能齐全，但仍然有各种各样的第三方Web包可用于从路由到全栈框架的所有内容，包括：

- https://github.com/urfave/negroni
- https://github.com/gin-gonic/gin
- https://github.com/labstack/echo
- http://www.gorillatoolkit.org/
- https://github.com/julienschmidt/httprouter

本章将重点介绍在处理请求，路由和请求对象以及处理中间件等概念时可能遇到的问题。

处理Web请求

Go定义了HandlerFuncs和一个Handler接口：

```
// HandlerFunc 实现了Handler接口
type HandlerFunc func(http.ResponseWriter, *http.Request)

type Handler interface {
    ServeHTTP(http.ResponseWriter, *http.Request)
}
```

net/http包对这种操作方式广泛使用。例如路由可以附加到Handler或HandlerFunc接口。本节将探讨如何在处理http.Request之后创建Handler接口，侦听本地端口以及在http.ResponseWriter接口上执行某些操作。 这是建立Go Web应用程序和RESTFul API的基础。

实践

建立 get.go：

```
package handlers

import (
    "fmt"
    "net/http"
)

// HelloHandler 接收GET请求中的参数"name"
// 在responds中返回 Hello <name>! 文本数据
func HelloHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/plain")
    if r.Method != http.MethodGet {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    name := r.URL.Query().Get("name")

    w.WriteHeader(http.StatusOK)
    w.Write([]byte(fmt.Sprintf("Hello %s!", name)))
}
```

建立 post.go：

```
package handlers

import (
    "encoding/json"
    "net/http"
)

// GreetingResponse 用于序列化GreetingHandler返回的JSON数据
type GreetingResponse struct {
    Payload struct {
        Greeting string `json:"greeting,omitempty"`
        Name     string `json:"name,omitempty"`
        Error    string `json:"error,omitempty"`
    } `json:"payload"`
    Successful bool `json:"successful"`
}

// GreetingHandler 返回GreetingResponse格式的数据
func GreetingHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    if r.Method != http.MethodPost {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    var gr GreetingResponse
    if err := r.ParseForm(); err != nil {
        gr.Payload.Error = "bad request"
        if payload, err := json.Marshal(gr); err == nil {
            w.Write(payload)
        }
    }
    name := r.FormValue("name")
    greeting := r.FormValue("greeting")

    w.WriteHeader(http.StatusOK)
    gr.Successful = true
    gr.Payload.Name = name
    gr.Payload.Greeting = greeting
    if payload, err := json.Marshal(gr); err == nil {
        w.Write(payload)
    }
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/handlers"
)

func main() {
    http.HandleFunc("/name", handlers.HelloHandler)
    http.HandleFunc("/greeting", handlers.GreetingHandler)
    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

执行go run main.go会显示

```
Listening on port :3333
```

在命令行中测试：

```
curl "http://localhost:3333/name?name=Reader" -X GET
Hello Reader!

curl "http://localhost:3333/greeting" -X POST -d
'name=Reader;greeting=Goodbye'
{"payload":
{"greeting":"Goodbye","name":"Reader"},"successful":true}
```

说明

示例中我们对GET请求和POST请求分别进行了处理。注意POST的响应是如何返回JOSN格式数据的。

这里只是简单的演示，更丰富的路由解析、限制、处理关闭等复杂操作，可以挑一些第三方库来看看他们是如何思考的。

使用闭包进行状态处理

将状态传递给处理程序通常很棘手。有两种方法：通过闭包传递状态，这有助于提高单个处理程序的灵活性，或使用结构体进行传递。

我们将使用结构控制器来存储接口，并使用由外部函数修改的单个处理程序创建两个路由。

实践

建立 controller.go：

```
package controllers

// Controller 传递状态给处理函数
type Controller struct {
    storage Storage
}

func New(storage Storage) *Controller {
    return &Controller{
        storage: storage,
    }
}

type Payload struct {
    Value string `json:"value"`
}
```

建立 storage.go：

```
package controllers

// Storage 接口支持存取单个值
type Storage interface {
    Get() string
    Put(string)
}

// MemStorage 实现了 Storage接口
type MemStorage struct {
    value string
}

func (m *MemStorage) Get() string {
```

```
    return m.value
}

func (m *MemStorage) Put(s string) {
    m.value = s
}
```

建立 post.go：

```
package controllers

import (
    "encoding/json"
    "net/http"
)

// SetValue 修改Controller的存储内容
func (c *Controller) SetValue(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    if err := r.ParseForm(); err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    value := r.FormValue("value")
    c.storage.Put(value)
    w.WriteHeader(http.StatusOK)
    p := Payload{Value: value}
    if payload, err := json.Marshal(p); err == nil {
        w.Write(payload)
    }

}
```

建立 get.go：

```
package controllers

import (
    "encoding/json"
    "net/http"
)
```

```
// GetValue是一个封装HandlerFunc的闭包，如果UseDefault为true，则值始终为“default”，否则它将是存储在storage中的任何内容
func (c *Controller) GetValue(UseDefault bool) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        if r.Method != "GET" {
            w.WriteHeader(http.StatusMethodNotAllowed)
            return
        }
        value := "default"
        if !UseDefault {
            value = c.storage.Get()
        }
        p := Payload{Value: value}
        w.WriteHeader(http.StatusOK)
        if payload, err := json.Marshal(p); err == nil {
            w.Write(payload)
        }
    }
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/controllers"
)

func main() {
    storage := controllers.MemStorage{}
    c := controllers.New(&storage)
    http.HandleFunc("/get", c.GetValue(false))
    http.HandleFunc("/get/default", c.GetValue(true))
    http.HandleFunc("/set", c.SetValue)

    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

运行：

```
go run main.go
```

这会输出

```
Listening on port :3333
```

进行请求测试：

```
$curl "http://localhost:3333/set -X POST -d "value=value"
{"value":"value"}
$curl "http://localhost:3333/get -X GET
{"value":"value"}
$curl "http://localhost:3333/get/default -X GET
{"value":"default"}
```

说明

这种策略有效，因为Go允许函数传递。我们可以用类似的方法传入数据库连接、日志记录等。在示例中，我们插入了一个storage接口，所有请求的处理方法都可以使用其方法和属性。

GetValue方法没有传递http.HandlerFunc签名，而是直接返回它，我们通过这种方式来注入状态。在main.go中，我们定义了两个路由，其中UseDefault设置为false，另一个路由设置为true。这可以在定义跨越多个路由的函数时使用，也可以在使用处理程序感觉过于繁琐的结构时使用。

请求参数验证

对Web进行验证是一个难题。本节将探讨使用闭包来支持验证函数，并允许在初始化控制器结构时执行验证类型以增强灵活性。

我们将在结构上执行验证，但不会讨论如何填充结构。我们可以假设通过解析JSON，从表单输入或其他方法显式填充数据。

实践

建立 controller.go：

```
package validation

// Controller 保存了验证方法
type Controller struct {
    ValidatePayload func(p *Payload) error
}

// New 使用我们的本地验证初始化controller 它可以被覆盖
func New() *Controller {
    return &Controller{
        ValidatePayload: ValidatePayload,
    }
}
```

建立 validate.go：

```
package validation

import "errors"

// Verror是在验证期间发生的错误，我们可以将其返回给用户
type Verror struct {
    error
}

// Payload 是我们处理的内容
type Payload struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

// ValidatePayload是我们控制器中闭包的1个实现
```

```
func ValidatePayload(p *Payload) error {
    if p.Name == "" {
        return Verror{errors.New("name is required")}
    }

    if p.Age <= 0 || p.Age >= 120 {
        return Verror{errors.New("age is required and must be a value greater th
an 0 and less than 120")}
    }
    return nil
}
```

建立 process.go：

```
package validation

import (
    "encoding/json"
    "fmt"
    "net/http"
)

// Process是一个验证post传入的数据
func (c *Controller) Process(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodPost {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }

    decoder := json.NewDecoder(r.Body)
    defer r.Body.Close()
    var p Payload

    if err := decoder.Decode(&p); err != nil {
        fmt.Println(err)
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    if err := c.ValidatePayload(&p); err != nil {
        switch err.(type) {
        case Verror:
            w.WriteHeader(http.StatusBadRequest)
            // pass the Verror along
            w.Write([]byte(err.Error()))
```

```
            return
        default:
            w.WriteHeader(http.StatusInternalServerError)
            return
        }
    }
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/validation"
)

func main() {
    c := validation.New()
    http.HandleFunc("/", c.Process)
    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

运行：

```
go run main.go
```

这会输出

```
Listening on port :3333
```

进行请求测试：

```
$curl "http://localhost:3333/-X POST -d '{}'
name is required
$curl "http://localhost:3333/-X POST -d '{"name":"test"}'
age is required and must be a value greater than 0 and
less than 120
$curl "http://localhost:3333/-X POST -d '{"name":"test",
```

```
"age": 5}' -v
<lots of output, should contain a 200 OK status code>
```

说明

我们通过将闭包传递给控制器结构来处理验证。这种方法的优点是我们可以在运行时模拟和替换验证函数，因此测试变得更加简单。 另外，我们没有绑定到单个函数签名，因此可以将诸如数据库连接之类的东西传递给验证函数。

另外需要关注的是返回了一个名为Verror的类型错误。此类型包含用于向用户显示的验证错误消息。这种方法的一个缺点是它不能同时处理多个验证消息。这可以通过修改Verror类型以允许更多状态（例如，通过包含映射）来实现，以便在从ValidatePayload函数返回之前容纳更多验证错误。说明
我们通过将闭包传递给控制器结构来处理验证。这种方法的优点是我们可以在运行时模拟和替换验证函数，因此测试变得更加简单。 另外，我们没有绑定到单个函数签名，因此可以将诸如数据库连接之类的东西传递给验证函数。

内容渲染

Web处理程序可以返回各种内容类型，例如，JSON，纯文本，图像等。 通常在与API通信时，可以指定并接收内容类型，以阐明将传入数据的格式以及要接收的数据。

本节将通过第三方库对数据格式进行切换。

实践

获取第三方库：

```
go get github.com/unrolled/render
```

建立 negotiate.go：

```
package negotiate

import (
    "net/http"

    "github.com/unrolled/render"
)

// Negotiator 封装render并对ContentType进行一些切换
type Negotiator struct {
    ContentType string
    *render.Render
}

// GetNegotiator 接收http请求 并从Content-Type标头中找出ContentType
func GetNegotiator(r *http.Request) *Negotiator {
    contentType := r.Header.Get("Content-Type")

    return &Negotiator{
        ContentType: contentType,
        Render:      render.New(),
    }
}
```

建立 respond.go：

```
package negotiate
```

```
import "io"
import "github.com/unrolled/render"

// Respond 根据 Content Type 判断应该返回什么样类型的数据
func (n *Negotiator) Respond(w io.Writer, status int, v interface{}) {
    switch n.ContentType {
    case render.ContentJSON:
        n.Render.JSON(w, status, v)
    case render.ContentXML:
        n.Render.XML(w, status, v)
    default:
        n.Render.JSON(w, status, v)
    }
}
```

建立 handler.go：

```
package negotiate

import (
    "encoding/xml"
    "net/http"
)

// Payload 甚至数据模板
type Payload struct {
    XMLName xml.Name `xml:"payload" json:"-"`
    Status  string   `xml:"status" json:"status"`
}

// Handler 调用GetNegotiator处理返回格式
func Handler(w http.ResponseWriter, r *http.Request) {
    n := GetNegotiator(r)

    n.Respond(w, http.StatusOK, &Payload{Status: "Successful!"})
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"
```

```
    "github.com/agtorre/go-cookbook/chapter7/negotiate"
)

func main() {
    http.HandleFunc("/", negotiate.Handler)
    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

运行：

```
$ go run main.go
Listening on port :3333

$curl "http://localhost:3333 -H "Content-Type: text/xml"
<payload><status>Successful!</status></payload>
$curl "http://localhost:3333 -H "Content-Type: application/json"
{"status":"Successful!"}
```

说明

github.com/unrolled/render 包可以帮助你处理各种类型的请求头。请求头通常包含多个值，您的代码必须考虑到这一点。

使用中间件

Go程序的中间件是一个被广泛探索的领域。有许多用于处理中间件的包。 本节将从头开始创建中间件，并实现一个ApplyMiddleware函数将一堆中间件链接在一起。此外还会在请求上下文对象中设置值，并使用中间件检索它们。以演示如何将中间件逻辑与处理程序分离。

实践

建立 middleware.go：

```
package middleware

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

// Middleware是所有的中间件函数都会返回的
type Middleware func(http.HandlerFunc) http.HandlerFunc

// ApplyMiddleware 将应用所有中间件，最后一个参数将是用于上下文传递目的的外部包装
func ApplyMiddleware(h http.HandlerFunc, middleware ...Middleware) http.HandlerFunc {
    applied := h
    for _, m := range middleware {
        applied = m(applied)
    }
    return applied
}

// Logger 记录请求日志 这会通过SetID()传递id
func Logger(l *log.Logger) Middleware {
    return func(next http.HandlerFunc) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
            start := time.Now()
            l.Printf("started request to %s with id %s", r.URL, GetID(r.Context()))
            next(w, r)
            l.Printf("completed request to %s with id %s in %s", r.URL, GetID(r.Context()), time.Since(start))
        }
```

```
    }
}
```

建立 context.go：

```
package middleware

import (
    "context"
    "net/http"
    "strconv"
)

// ContextID 是自定义类型 用于检索context
type ContextID int

// ID是我们定义的唯一ID
const ID ContextID = 0

// SetID 使用自增唯一id更新context
func SetID(start int64) Middleware {
    return func(next http.HandlerFunc) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
            ctx := context.WithValue(r.Context(), ID, strconv.FormatInt(start, 10))
            start++
            r = r.WithContext(ctx)
            next(w, r)
        }
    }
}

// GetID 如果设置，则从上下文中获取ID，否则返回空字符串
func GetID(ctx context.Context) string {
    if val, ok := ctx.Value(ID).(string); ok {
        return val
    }
    return ""
}
```

建立 handler.go：

```
package middleware

import (
```

```
    "net/http"
)

func Handler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("success"))
}
```

建立 main.go：

```
package main

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

    "github.com/agtorre/go-cookbook/chapter7/middleware"
)

func main() {
    // We apply from bottom up
    h := middleware.ApplyMiddleware(
        middleware.Handler,
        middleware.Logger(log.New(os.Stdout, "", 0)),
        middleware.SetID(100),
    )
    http.HandleFunc("/", h)
    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

运行：

```
$ go run main.go
Listening on port :3333

$curl "http://localhost:3333
success
$curl "http://localhost:3333
success
$curl "http://localhost:3333
success
```

此外在运行main.go的命令行你还会看到：

```
Listening on port :3333
started request to / with id 100
completed request to / with id 100 in 52.284µs
started request to / with id 101
completed request to / with id 101 in 40.273µs
started request to / with id 102
```

说明

中间件可用于执行简单操作，例如日志记录，度量标准收集和分析。它还可用于在每个请求上动态填充变量。例如，可以用于从请求中收集X-Header以设置ID或生成ID，就像我们在示例中所做的那样。另一个ID策略可能是为每个请求生成一个UUID，这样我们可以轻松地将日志消息关联在一起，并在构建响应时跟踪请求。

使用上下文值时，考虑中间件的顺序很重要。通常，最好不要让中间件相互依赖。 例如，最好在日志记录中间件本身中生成UUID。

构建反向代理

本节我们将建立一个反向代理应用。我们的想法是，通过请求http：// localhost：3333，所有流量都将转发到可配置的主机，响应将转发到你的浏览器。

这可以与端口转发和ssh隧道结合使用，以便通过中间服务器安全地访问网站。本节将从头开始构建反向代理，但标准库的net/http/httputil包也提供此功能。使用此包，可以通过Director func(*http.Request)修改传入请求，并且可以通过 ModifyResponse func(*http.Response) error错误修改传出响应。 此外，还支持缓冲响应。

实践

建立 proxy.go：

```
package proxy

import (
    "log"
    "net/http"
)

// Proxy 保存了客户端配置和需要代理的BaseURL地址
type Proxy struct {
    Client  *http.Client
    BaseURL string
}

// ServeHTTP 表示代理部署了Handler接口它操纵请求，将其转发给BaseURL，然后返回响应
func (p *Proxy) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if err := p.ProcessRequest(r); err != nil {
        log.Printf("error occurred during process request: %s", err.Error())
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    resp, err := p.Client.Do(r)
    if err != nil {
        log.Printf("error occurred during client operation: %s", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    defer resp.Body.Close()
```

```
    CopyResponse(w, resp)
}
```

建立 process.go：

```
package proxy

import (
    "bytes"
    "net/http"
    "net/url"
)

// ProcessRequest 根据Proxy设置修改请求
func (p *Proxy) ProcessRequest(r *http.Request) error {
    proxyURLRaw := p.BaseURL + r.URL.String()

    proxyURL, err := url.Parse(proxyURLRaw)
    if err != nil {
        return err
    }
    r.URL = proxyURL
    r.Host = proxyURL.Host
    r.RequestURI = ""
    return nil
}

// CopyResponse获取客户端响应并将所有内容写入原始处理程序中的ResponseWriter
func CopyResponse(w http.ResponseWriter, resp *http.Response) {
    var out bytes.Buffer
    out.ReadFrom(resp.Body)

    for key, values := range resp.Header {
        for _, value := range values {
            w.Header().Add(key, value)
        }
    }

    w.WriteHeader(resp.StatusCode)
    w.Write(out.Bytes())
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/proxy"
)

func main() {
    p := &proxy.Proxy{
        Client:  http.DefaultClient,
        BaseURL: "https://www.golang.org",
    }
    http.Handle("/", p)
    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

这会输出：

```
$ go run main.go
Listening on port :3333
```

在浏览器地址栏输入localhost:3333/， 你会看到跳转到了https://golang.org/。

说明

Go请求和响应对象在很大程度上可以在客户端和处理程序之间共享。示例代码接受由满足Handler接口的Proxy结构获取的请求。一旦请求可用，它就被修改为在请求之前添加Proxy.BaseURL。最后，响应被复制回ResponseWriter接口。

我们还可以添加一些其他功能，例如请求的基本身份验证，令牌管理等。这对于代理管理JavaScript或其他客户端应用程序的会话令牌管理非常有用。

将GRPC导出为JSON API

在第六章的“理解GRPC的使用”一节中，我们实现了一个基础的GRPC服务器和客户端。本节将通过将常见的RPC函数放在一个包中并将它们包装在GRPC服务器和标准Web处理程序中来进行扩展。当你的API希望支持两种类型的客户端，但不希望复制代码以实现常见功能时，这非常有用。

实践

安装GRPC：

https://github.com/grpc/grpc/blob/master/INSTALL.md.

```
go get github.com/golang/protobuf/proto
go get github.com/golang/protobuf/protoc-gen-go
```

建立greeter.proto：

```
syntax = "proto3";

package keyvalue;

service KeyValue{
    rpc Set(SetKeyValueRequest) returns (KeyValueResponse){}
    rpc Get(GetKeyValueRequest) returns (KeyValueResponse){}
}

message SetKeyValueRequest {
    string key = 1;
    string value = 2;
}

message GetKeyValueRequest{
    string key = 1;
}

message KeyValueResponse{
    string success = 1;
    string value = 2;
}
```

运行

```
protoc --go_out=plugins=grpc:. greeter.proto
```

建立 keyvalue.go：

```
package internal

import (
    "golang.org/x/net/context"
    "sync"

    "github.com/agtorre/go-cookbook/chapter7/grpcjson/keyvalue"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
)

type KeyValue struct {
    mutex sync.RWMutex
    m     map[string]string
}

// NewKeyValue 初始化了KeyValue中的map
func NewKeyValue() *KeyValue {
    return &KeyValue{
        m: make(map[string]string),
    }
}

// Set 为键设置一个值，然后返回该值
func (k *KeyValue) Set(ctx context.Context, r *keyvalue.SetKeyValueRequest) (*ke
yvalue.KeyValueResponse, error) {
    k.mutex.Lock()
    k.m[r.GetKey()] = r.GetValue()
    k.mutex.Unlock()
    return &keyvalue.KeyValueResponse{Value: r.GetValue()}, nil
}

// Get 得到一个给定键的值，或者如果它不存在报告查找失败
func (k *KeyValue) Get(ctx context.Context, r *keyvalue.GetKeyValueRequest) (*ke
yvalue.KeyValueResponse, error) {
    k.mutex.RLock()
    defer k.mutex.RUnlock()
    val, ok := k.m[r.GetKey()]
    if !ok {
        return nil, grpc.Errorf(codes.NotFound, "key not set")
    }
```

```
    return &keyvalue.KeyValueResponse{Value: val}, nil
}
```

建立 grpc/main.go：

```
package main

import (
    "fmt"
    "net"

    "github.com/agtorre/go-cookbook/chapter7/grpcjson/internal"
    "github.com/agtorre/go-cookbook/chapter7/grpcjson/keyvalue"
    "google.golang.org/grpc"
)

func main() {
    grpcServer := grpc.NewServer()
    keyvalue.RegisterKeyValueServer(grpcServer, internal.NewKeyValue())
    lis, err := net.Listen("tcp", ":4444")
    if err != nil {
        panic(err)
    }
    fmt.Println("Listening on port :4444")
    grpcServer.Serve(lis)
}
```

建立 set.go：

```
package main

import (
    "encoding/json"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/grpcjson/internal"
    "github.com/agtorre/go-cookbook/chapter7/grpcjson/keyvalue"
    "github.com/apex/log"
)

// Controller 保存一个内部的KeyValueObject
type Controller struct {
    *internal.KeyValue
}
```

```
// SetHandler 封装了RPC的Set调用
func (c *Controller) SetHandler(w http.ResponseWriter, r *http.Request) {
    var kv keyvalue.SetKeyValueRequest

    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&kv); err != nil {
        log.Errorf("failed to decode: %s", err.Error())
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    gresp, err := c.Set(r.Context(), &kv)
    if err != nil {
        log.Errorf("failed to set: %s", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
    }

    resp, err := json.Marshal(gresp)
    if err != nil {
        log.Errorf("failed to marshal: %s", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    w.WriteHeader(http.StatusOK)
    w.Write(resp)
}
```

建立 get.go：

```
package main

import (
    "encoding/json"
    "net/http"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"

    "github.com/agtorre/go-cookbook/chapter7/grpcjson/keyvalue"
    "github.com/apex/log"
)

// GetHandler 封装了RPC的Get调用
func (c *Controller) GetHandler(w http.ResponseWriter, r *http.Request) {
```

```
    key := r.URL.Query().Get("key")
    kv := keyvalue.GetKeyValueRequest{Key: key}

    gresp, err := c.Get(r.Context(), &kv)
    if err != nil {
        if grpc.Code(err) == codes.NotFound {
            w.WriteHeader(http.StatusNotFound)
            return
        }
        log.Errorf("failed to get: %s", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusOK)
    resp, err := json.Marshal(gresp)
    if err != nil {
        log.Errorf("failed to marshal: %s", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    w.Write(resp)
}
```

建立 main.go：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/agtorre/go-cookbook/chapter7/grpcjson/internal"
)

func main() {
    c := Controller{KeyValue: internal.NewKeyValue()}
    http.HandleFunc("/set", c.SetHandler)
    http.HandleFunc("/get", c.GetHandler)

    fmt.Println("Listening on port :3333")
    err := http.ListenAndServe(":3333", nil)
    panic(err)
}
```

运行：

```
$ go run http/*.go
Listening on port :3333

$curl "http://localhost:3333/set" -d '{"key":"test",
"value":"123"}' -v
{"value":"123"}
$curl "http://localhost:3333/get?key=badtest" -v
'name=Reader;greeting=Goodbye'
<should return a 404>
$curl "http://localhost:3333/get?key=test" -v
'name=Reader;greeting=Goodbye'
{"value":"123"}
```

说明

虽然示例中省略了客户端，但你可以复制第6章GRPC章节中的步骤，这样应该看到与示例中相同的结果。http和grpc使用了相同的内部包。我们必须返回适当的GRPC错误代码，并将这些错误代码映射到HTTP响应。在这种情况下，我们使用codes.NotFound，它映射到http.StatusNotFound。如果必须处理多种错误，则switch语句可能比if ... else语句更有意义。

你可能注意到GRPC签名通常非常一致。他们接受请求并返回可选响应和错误。如果你的GRPC调用重复性很强并且看起来很适合代码生成，那么可以创建一个通用的处理程序来填充它，像github.com/goadesign/goa 这样的库就是这么干的。

测试

本章会覆盖以下内容：

- 使用标准库进行模拟
- 使用Mockgen包
- 使用表驱动测试
- 使用第三方测试工具
- 模糊测试
- 行为驱动测试

介绍

本章与前面的章节有所不同，我们的目光将转向编写测试。Go语言提供了开箱即用的测试支持，使得monkey patching和模拟相对简单。

Go测试鼓励代码的结构化，特别是测试和模拟接口非常简单且对此给予了很好的支持。某些类型的代码可能更难以测试，例如，测试使用包级别全局变量的代码，未被抽象到接口的函数以及具有非导出变量或方法的结构可能很困难。本章将分享一些测试Go代码的思路。

使用标准库进行模拟

在Go中，模拟通常意味着实现具有测试版本的接口，该测试版本允许从测试中控制运行时行为。它也可以指模拟函数和方法，我们将探索如何实现它。示例中使用的Patch和Restore函数可以在https://play.golang.org/p/oLF1XnRX3C 找到。

包含大量分支条件或深度嵌套逻辑的代码可能很难测试，最后测试往往更加效果很差。这是因为开发人员需要在其测试中跟踪很多模拟对象，返回值和状态。

实践

建立 mock.go：

```
package mocking

// DoStuffer 是一个简单的接口
type DoStuffer interface {
    DoStuff(input string) error
}
```

建立 patch.go：

```
package mocking

import "reflect"

// Restorer是一个可用于恢复先前状态的函数
type Restorer func()

// Restore存储了之前的状态
func (r Restorer) Restore() {
    r()
}

// Patch将给定目标指向的值设置为给定值，并返回一个函数以将其恢复为原始值。 该值必须可分配给目标的元素类型。
func Patch(dest, value interface{}) Restorer {
    destv := reflect.ValueOf(dest).Elem()
    oldv := reflect.New(destv.Type()).Elem()
    oldv.Set(destv)
    valuev := reflect.ValueOf(value)
    if !valuev.IsValid() {
        // 对于目标类型不可用的情况，这种解决方式并不优雅
        valuev = reflect.Zero(destv.Type())
```

```
    }
    destv.Set(valuev)
    return func() {
        destv.Set(oldv)
    }
}
```

建立 exec.go：

```
package mocking

import "errors"

var ThrowError = func() error {
    return errors.New("always fails")
}

func DoSomeStuff(d DoStuffer) error {

    if err := d.DoStuff("test"); err != nil {
        return err
    }

    if err := ThrowError(); err != nil {
        return err
    }

    return nil
}
```

建立 mock_test.go：

```
package mocking

type MockDoStuffer struct {
    // 使用闭包模拟
    MockDoStuff func(input string) error
}

func (m *MockDoStuffer) DoStuff(input string) error {
    if m.MockDoStuff != nil {
        return m.MockDoStuff(input)
    }
    // 如果我们不模拟输入，就返回一个常见的情况
```

```
    return nil
}
```

建立 exec_test.go：

```
package mocking

import (
    "errors"
    "testing"
)

func TestThrowError(t *testing.T) {
    tests := []struct {
        name    string
        wantErr bool
    }{
        {"base-case", true},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if err := ThrowError(); (err != nil) != tt.wantErr {
                t.Errorf("DoSomeStuff() error = %v, wantErr %v", err, tt.wantEr
r)
            }
        })
    }
}

func TestDoSomeStuff(t *testing.T) {
    tests := []struct {
        name       string
        DoStuff    error
        ThrowError error
        wantErr    bool
    }{
        {"base-case", nil, nil, false},
        {"DoStuff error", errors.New("failed"), nil, true},
        {"ThrowError error", nil, errors.New("failed"), true},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // 使用模拟结构来模拟接口
            d := MockDoStuffer{}
            d.MockDoStuff = func(string) error { return tt.DoStuff }
```

```
            // 模拟声明为变量的函数对func A()不起作用，必须是var A = func()
            defer Patch(&ThrowError, func() error { return tt.ThrowError }).Rest
ore()

            if err := DoSomeStuff(&d); (err != nil) != tt.wantErr {
                t.Errorf("DoSomeStuff() error = %v, wantErr %v", err, tt.wantEr
r)
            }
        })
    }
}
```

运行go test：

```
PASS
ok github.com/agtorre/go-cookbook/chapter8/mocking 0.006s
```

说明

无论是使用errors.New，fmt.Errorf还是自定义错误，最重要的是不应该在代码中不处理错误。这些定义错误的不同方法提供了很大的灵活性。例如，你可以在结构中添加额外的函数，以进一步检查错误并将接口转换为调用函数中的错误类型，以获得一些额外的功能。

接口本身非常简单，唯一的要求是返回一个有效的字符串。(在测试中明显将其复杂化了)这样的测试保证对某些要求严格的应用程序同样可用。

使用Mockgen包

前面的小节我们使用了自行模拟的方式。当你需要面对很多的接口时，这么干会变得极为麻烦且极易发生错误。这是自动化测试的意义所在。本节我们使用 github.com/golang/mock/gomock ，该库提供了一组模拟对象，可以与接口测试结合使用。

实践

获取第三方库：

```
go get github.com/golang/mock/
```

建立 interface.go：

```
package mockgen

type GetSetter interface {
    Set(key, val string) error
    Get(key string) (string, error)
}
```

运行命令行建立 mocks.go：

```
mockgen -destination internal/mocks.go -package internal
github.com/agtorre/go-cookbook/chapter8/mockgen GetSetter
```

```
// Automatically generated by MockGen. DO NOT EDIT!
// Source: github.com/agtorre/go-cookbook/chapter8/mockgen (interfaces: GetSette
r)

package internal

import (
    gomock "github.com/golang/mock/gomock"
)

// Mock of GetSetter interface
type MockGetSetter struct {
    ctrl     *gomock.Controller
    recorder *_MockGetSetterRecorder
}
```

```
// Recorder for MockGetSetter (not exported)
type _MockGetSetterRecorder struct {
    mock *MockGetSetter
}

func NewMockGetSetter(ctrl *gomock.Controller) *MockGetSetter {
    mock := &MockGetSetter{ctrl: ctrl}
    mock.recorder = &_MockGetSetterRecorder{mock}
    return mock
}

func (_m *MockGetSetter) EXPECT() *_MockGetSetterRecorder {
    return _m.recorder
}

func (_m *MockGetSetter) Get(_param0 string) (string, error) {
    ret := _m.ctrl.Call(_m, "Get", _param0)
    ret0, _ := ret[0].(string)
    ret1, _ := ret[1].(error)
    return ret0, ret1
}

func (_mr *_MockGetSetterRecorder) Get(arg0 interface{}) *gomock.Call {
    return _mr.mock.ctrl.RecordCall(_mr.mock, "Get", arg0)
}

func (_m *MockGetSetter) Set(_param0 string, _param1 string) error {
    ret := _m.ctrl.Call(_m, "Set", _param0, _param1)
    ret0, _ := ret[0].(error)
    return ret0
}

func (_mr *_MockGetSetterRecorder) Set(arg0, arg1 interface{}) *gomock.Call {
    return _mr.mock.ctrl.RecordCall(_mr.mock, "Set", arg0, arg1)
}
```

建立 exec.go：

```
package mockgen

// Controller 这个结构体演示了一种初始化接口的方式
type Controller struct {
    GetSetter
}
```

```
// GetThenSet 检查值是否已设置。如果没有设置就将其设置
func (c *Controller) GetThenSet(key, value string) error {
    val, err := c.Get(key)
    if err != nil {
        return err
    }

    if val != value {
        return c.Set(key, value)
    }
    return nil
}
```

建立 interface_test.go：

```
package mockgen

import (
    "errors"
    "testing"

    "github.com/agtorre/go-cookbook/chapter8/mockgen/internal"
    "github.com/golang/mock/gomock"
)

func TestExample(t *testing.T) {
    ctrl := gomock.NewController(t)
    defer ctrl.Finish()

    mockGetSetter := internal.NewMockGetSetter(ctrl)

    var k string
    mockGetSetter.EXPECT().Get("we can put anything here!").Do(func(key string)
 {
        k = key
    }).Return("", nil)

    customError := errors.New("failed this time")

    mockGetSetter.EXPECT().Get(gomock.Any()).Return("", customError)

    if _, err := mockGetSetter.Get("we can put anything here!"); err != nil {
        t.Errorf("got %#v; want %#v", err, nil)
    }
```

```
    if k != "we can put anything here!" {
        t.Errorf("bad key")
    }

    if _, err := mockGetSetter.Get("key"); err == nil {
        t.Errorf("got %#v; want %#v", err, customError)
    }
}
```

建立 exec_test.go：

```
package mockgen

import (
    "errors"
    "testing"

    "github.com/agtorre/go-cookbook/chapter8/mockgen/internal"
    "github.com/golang/mock/gomock"
)

func TestController_Set(t *testing.T) {
    tests := []struct {
        name         string
        getReturnVal string
        getReturnErr error
        setReturnErr error
        wantErr      bool
    }{
        {"get error", "value", errors.New("failed"), nil, true},
        {"value match", "value", nil, nil, false},
        {"no errors", "not set", nil, nil, false},
        {"set error", "not set", nil, errors.New("failed"), true},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            ctrl := gomock.NewController(t)
            defer ctrl.Finish()

            mockGetSetter := internal.NewMockGetSetter(ctrl)
            mockGetSetter.EXPECT().Get("key").AnyTimes().Return(tt.getReturnVal,
tt.getReturnErr)
            mockGetSetter.EXPECT().Set("key", gomock.Any()).AnyTimes().Return(t
t.setReturnErr)
```

```
            c := &Controller{
                GetSetter: mockGetSetter,
            }
            if err := c.GetThenSet("key", "value"); (err != nil) != tt.wantErr {
                t.Errorf("Controller.Set() error = %v, wantErr %v", err, tt.want
Err)
            }
        })
    }
}
```

说明

生成的模拟对象允许测试预定的参数，调用函数的次数以及返回的内容，并且允许我们设置其他工作流程。interface_test.go文件展示了在线调用它们时使用模拟对象的一些示例。 通常，测试看起来更像exec_test.go，我们希望拦截由实际代码执行的接口函数调用，并在测试时更改它们的行为。

exec_test.go文件还展示了如何在表驱动的测试环境中使用模拟对象。Any()函数意味着模拟函数可以被调用零次或多次，这对于代码提前终止的情况非常有用。

示例演示的最后一个技巧是将模拟对象粘贴到内部包中。当需要模拟在自己之外的包中声明的函数时，这非常有用。 这允许在非*test.go文件中定义这些方法，但不允许将它们导出到库的情况。通常，使用与当前编写的测试相同的包名称将模拟对象粘贴到*test.go文件中更容易。

使用表驱动测试

本节将演示如何编写表驱动测试，收集测试覆盖率并对其进行改进。还将使用 github.com/cweill/gotests 包来生成测试。 如果你一直在下载其他章节的测试代码，这些代码应该看起来非常熟悉。使用本节与前几节的测试组合，应该能够实现100%的测试覆盖率。

实践

获取第三方库：

```
go get github.com/cweill/gotests/
```

建立 coverage.go：

```
package main

import "errors"

// Coverage 是一个具有一些分支条件的简单函数
func Coverage(condition bool) error {
    if condition {
        return errors.New("condition was set")
    }
    return nil
}
```

建立 coverage_test.go：运行

```
gotests -all -w
```

这会生成：

```
package main

import "testing"

func TestCoverage(t *testing.T) {
    type args struct {
        condition bool
    }
    tests := []struct {
        name    string
```

```
        args    args
        wantErr bool
    }{
        //TODO
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if err := Coverage(tt.args.condition); (err != nil) != tt.wantErr {
                t.Errorf("Coverage() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}
```

填充TODO部分：

```
{"no condition", args{true}, true},
```

运行测试：

```
go test -cover
PASS
coverage: 66.7% of statements
ok github.com/agtorre/go-cookbook/chapter8/coverage 0.007s
```

```
go test -coverprofile=cover.out
go tool cover -html=cover.out -o coverage.html
```

打开coverage.html可以看到覆盖率报告。

说明

go test -cover命令附带一个基本的Go安装。它可用于收集Go应用程序的测试覆盖率报告。此外，它还能够输出覆盖率指标和HTML覆盖率报告。此工具通常由其他工具包装，将在下一节中介绍。 https//github.com/golang/go/wiki/TableDrivenTests 涵盖了这些表驱动的测试样例，可以在不编写大量额外代码的情况下完成可以处理许多情况的干净测试。

首先自动生成测试代码，然后根据需要填写测试用例以帮助创建更多的覆盖范围。在调用非变量函数或方法时，或测试输入和输出的许多变化，可能很难达到100％的测试覆盖率，在这样的情况下，模糊测试会变得很有用。

使用第三方测试工具

Go测试有许多有用的工具。 这些工具可以让你更容易地了解每个功能级别的代码覆盖率，还可以使用断言工具来减少测试代码行。 本文将介绍 github.com/axw/gocov和 github.com/smartystreets/goconvey 软件包，以展示其中的一些功能。此外 github.com/smartystreets/goconvey 包支持断言和运行时测试。

实践

获取第三方库：

```
go get github.com/axw/gocov
go get github.com/smartystreets/goconvey/
```

建立 funcs.go：

```
package tools

import (
    "fmt"
)

func example() error {
    fmt.Println("in example")
    return nil
}

var example2 = func() int {
    fmt.Println("in example2")
    return 10
}
```

建立 structs.go：

```
package tools

import (
    "errors"
    "fmt"
)

type c struct {
    Branch bool
```

```
}

func (c *c) example3() error {
    fmt.Println("in example3")
    if c.Branch {
        fmt.Println("branching code!")
        return errors.New("bad branch")
    }
    return nil
}
```

建立 funcs_test.go：

```
package tools

import (
    "testing"

    . "github.com/smartystreets/goconvey/convey"
)

func Test_example(t *testing.T) {
    tests := []struct {
        name string
    }{
        {"base-case"},
    }
    for _, tt := range tests {
        Convey(tt.name, t, func() {
            res := example()
            So(res, ShouldBeNil)
        })
    }
}

func Test_example2(t *testing.T) {
    tests := []struct {
        name string
    }{
        {"base-case"},
    }
    for _, tt := range tests {
        Convey(tt.name, t, func() {
            res := example2()
            So(res, ShouldBeGreaterThanOrEqualTo, 1)
```

```
        })
    }
}
```

建立 structs_test.go：

```
package tools

import (
    "testing"

    . "github.com/smartystreets/goconvey/convey"
)

func Test_c_example3(t *testing.T) {
    type fields struct {
        Branch bool
    }
    tests := []struct {
        name    string
        fields  fields
        wantErr bool
    }{
        {"no branch", fields{false}, false},
        {"branch", fields{true}, true},
    }
    for _, tt := range tests {
        Convey(tt.name, t, func() {
            c := &c{
                Branch: tt.fields.Branch,
            }
            So((c.example3() != nil), ShouldEqual, tt.wantErr)
        })
    }
}
```

运行：

```
$ gocov test | gocov report
ok github.com/agtorre/go-cookbook/chapter8/tools 0.006s
coverage: 100.0% of statements
github.com/agtorre/go-cookbook/chapter8/tools/struct.go
c.example3 100.00% (5/5)
github.com/agtorre/go-cookbook/chapter8/tools/funcs.go example
100.00% (2/2)
```

```
github.com/agtorre/go-cookbook/chapter8/tools/funcs.go @12:16
100.00% (2/2)
github.com/agtorre/go-cookbook/chapter8/tools ----------
100.00% (9/9)
Total Coverage: 100.00% (9/9)
```

执行goconvey命令，会在浏览器中显示：

说明

本节演示了如何使用goconvry。与前面的小节相比，Convey关键字基本替代了t.Run，并且可以生成在goconvey生成的UI中显示的标签，但与t,Run的表现有所不同。如果你有嵌套的这样测试块：

```
Convey("Outer loop", t, func(){
    a := 1
    Convey("Inner loop", t, func() {
        a = 2
    })
    Convey ("Inner loop2", t, func(){
        fmt.Println(a)
    })
})
```

使用goconvey命令，将打印1。如果我们使用t.Run，将打印2。换句话说，t.Run按顺序运行测试，永远不会重复。此行为对于将变量设置在外部代码块中非常有用。如果混合使用，就必须记住这种区别。

在使用convey断言时，在UI中有成功的复选标记和其他统计信息。使用它还可以减少if检查单行的大小，甚至可以创建自定义断言。

如果保留goconvey Web界面并打开通知，则在保存代码时，将自动运行测试，并且将收到有关任何增加或减少覆盖范围以及构建失败时的通知。

以上功能都可以单独或一起使用。

在努力提高测试覆盖率时，gocov工具非常有用。它可以快速识别尚未测试的函数，并帮助了解覆盖率报告。此外，gocov可用于生成使用 github.com/matm/gocov-html 包随Go代码一起提供的备用HTML报告。