**Golang io reader writer**

[合肥懒皮](https://www.jianshu.com/u/50e1d98d51ac)关注

0.8282019.03.12 15:57:36字数 2,060阅读 1,610

推荐阅读  
[Go语言小贴士1 － io包](https://links.jianshu.com/go?to=https%3A%2F%2Fzhuanlan.zhihu.com%2Fp%2F21360811) <https://zhuanlan.zhihu.com/p/21360811>  
[Go语言小贴士2 － 协议解析](https://links.jianshu.com/go?to=https%3A%2F%2Fzhuanlan.zhihu.com%2Fp%2F21367696) <https://zhuanlan.zhihu.com/p/21367696>  
[Go语言小贴士3 － bufio包](https://links.jianshu.com/go?to=https%3A%2F%2Fzhuanlan.zhihu.com%2Fp%2F21369473) <https://zhuanlan.zhihu.com/p/21369473>

**一、《GO语言实战》P194**

类 UNIX 的操作系统如此伟大的一个原因是，一个程序的输出可以是另一个程序的输入这一理念。依照这个哲学，这类操作系统创建了一系列的简单程序，每个程序只做一件事，并把这件事做得非常好。之后，将这些程序组合在一起，可以创建一些脚本做一些很惊艳的事情。这些程序使用 stdin 和 stdout 设备作为通道，在进程之间传递数据。

同样的理念扩展到了标准库的 io 包，而且提供的功能很神奇。这个包可以以流的方式高效处理数据，而不用考虑数据是什么，数据来自哪里，以及数据要发送到哪里的问题。与 stdout和 stdin 对应，这个包含有 io.Writer 和 io.Reader 两个接口。所有实现了这两个接口的类型的值，都可以使用 io 包提供的所有功能，也可以用于其他包里接受这两个接口的函数以及方法。这是用接口类型来构造函数和 API 最美妙的地方。开发人员可以基于这些现有功能进行组合，利用所有已经存在的实现，专注于解决业务问题。

io 包是围绕着实现了 io.Writer 和 io.Reader 接口类型的值而构建的。由于 io.Writer和 io.Reader 提供了足够的抽象，这些 io 包里的函数和方法并不知道数据的类型，也不知道这些数据在物理上是如何读和写的。

type Writer interface {

Write(p []byte) (n int, err error)

}

Write 从 p 里向底层的数据流写入 len(p)字节的数据。这个方法返回从 p 里写出的字节数（0 <= n <= len(p)），以及任何可能导致写入提前结束的错误。Write 在返回 n< len(p)的时候，必须返回某个非 nil 值的 error。Write 绝不能改写切片里的数据，哪怕是临时修改也不行。

type Reader interface {

Read(p []byte) (n int, err error)

}

(1) Read 最多读入 len(p)字节，保存到 p。这个方法返回读入的字节数（0 <= n<= len(p)）和任何读取时发生的错误。即便 Read 返回的 n < len(p)，方法也可能使用所有 p 的空间存储临时数据。如果数据可以读取，但是字节长度不足 len(p)，习惯上 Read 会立刻返回可用的数据，而不等待更多的数据。  
(2) 当成功读取 n > 0 字节后，如果遇到错误或者文件读取完成，Read 方法会返回读入的字节数。方法可能会在本次调用返回一个非 nil 的错误，或者在下一次调用时返回错误（同时 n == 0）。这种情况的的一个例子是，在输入的流结束时，Read 会返回非零的读取字节数，可能会返回 err == EOF，也可能会返回 err == nil。无论如何，下一次调用 Read 应该返回 0, EOF。  
(3) 调用者在返回的 n > 0 时，总应该先处理读入的数据，再处理错误 err。这样才能正确操作读取一部分字节后发生的 I/O 错误。EOF 也要这样处理。  
(4) Read 的实现不鼓励返回 0 个读取字节的同时，返回 nil 值的错误。调用者需要将这种返回状态视为没有做任何操作，而不是遇到读取结束。

**二、**[**Go 中 io 包的使用方法**](https://links.jianshu.com/go?to=https%3A%2F%2Fsegmentfault.com%2Fa%2F1190000015591319)

1.标准reader和自己实现一个reader

package main

import (

"fmt"

"io"

"strings"

)

type alphaReader struct {

// 资源

src string

// 当前读取到的位置

cur int

}

// 创建一个实例

func newAlphaReader(src string) \*alphaReader {

return &alphaReader{src: src}

}

// 过滤函数

func alpha(r byte) byte {

if (r >= 'A' && r <= 'Z') || (r >= 'a' && r <= 'z') {

return r

}

return 0

}

// Read 方法

func (a \*alphaReader) Read(p []byte) (int, error) {

// 当前位置 >= 字符串长度 说明已经读取到结尾 返回 EOF

if a.cur >= len(a.src) {

return 0, io.EOF

}

// x 是剩余未读取的长度

x := len(a.src) - a.cur

n, bound := 0, 0

if x >= len(p) {

// 剩余长度超过缓冲区大小，说明本次可完全填满缓冲区

bound = len(p)

} else if x < len(p) {

// 剩余长度小于缓冲区大小，使用剩余长度输出，缓冲区不补满

bound = x

}

buf := make([]byte, bound)

for n < bound {

// 每次读取一个字节，执行过滤函数

if char := alpha(a.src[a.cur]); char != 0 {

buf[n] = char

}

n++

a.cur++

}

// 将处理后得到的 buf 内容复制到 p 中

copy(p, buf)

return n, nil

}

func main() {

reader := strings.NewReader("Clear is better than clever")

//reader := newAlphaReader("Hello! It's 9am, where is the sun?")

p := make([]byte, 4)

for {

n, err := reader.Read(p)

if err == io.EOF {

break

}

fmt.Println(string(p[:n]))

//fmt.Print(string(p[:n]))

}

fmt.Println()

}

可以使用strings.NewReader创建一个字符串读取器，然后流式地按字节读取:

Clea

r is

bet

ter

than

cle

ver

可以看到，最后一次返回的 n 值有可能小于缓冲区大小。

也可以使用newAlphaReader自己实现一个，并且加上过滤非字母字符的功能：

Hell

o I

t s

am

whe

re i

s th

e su

n

在[Go指南 实现一个 Reader 类型，它产生一个 ASCII 字符 'A' 的无限流](https://links.jianshu.com/go?to=https%3A%2F%2Ftour.go-zh.org%2Fmethods%2F22" \t "_blank)，可以这么做：

package main

import "golang.org/x/tour/reader"

type MyReader struct{}

// TODO: 给 MyReader 添加一个 Read([]byte) (int, error) 方法

func (r MyReader) Read(b []byte) (int, error) {

// 赋值并返回

b[0] = 'A'

return 1, nil

}

func main() {

reader.Validate(MyReader{})

}

2.组合reader  
标准库已经实现了许多 Reader。使用一个 Reader 作为另一个 Reader 的实现是一种常见的用法。这样做可以让一个 Reader 重用另一个 Reader 的逻辑，下面展示通过更新 alphaReader 以接受 io.Reader 作为其来源。

type alphaReader struct {

// alphaReader 里组合了标准库的 io.Reader

reader io.Reader

}

func newAlphaReader(reader io.Reader) \*alphaReader {

return &alphaReader{reader: reader}

}

func alpha(r byte) byte {

if (r >= 'A' && r <= 'Z') || (r >= 'a' && r <= 'z') {

return r

}

return 0

}

func (a \*alphaReader) Read(p []byte) (int, error) {

// 这行代码调用的就是 io.Reader

n, err := a.reader.Read(p)

if err != nil {

return n, err

}

buf := make([]byte, n)

for i := 0; i < n; i++ {

if char := alpha(p[i]); char != 0 {

buf[i] = char

}

}

copy(p, buf)

return n, nil

}

func main() {

// 使用实现了标准库 io.Reader 接口的 strings.Reader 作为实现

reader := newAlphaReader(strings.NewReader("Hello! It's 9am, where is the sun?"))

p := make([]byte, 4)

for {

n, err := reader.Read(p)

if err == io.EOF {

break

}

fmt.Print(string(p[:n]))

}

fmt.Println()

}

这样做的另一个优点是 alphaReader 能够从任何 Reader 实现中读取。例如，以下代码展示了 alphaReader 如何与 os.File 结合以过滤掉文件中的非字母字符：

func main() {

// file 也实现了 io.Reader

file, err := os.Open("./alpha\_reader3.go")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

// 任何实现了 io.Reader 的类型都可以传入 newAlphaReader

// 至于具体如何读取文件，那是标准库已经实现了的，我们不用再做一遍，达到了重用的目的

reader := newAlphaReader(file)

p := make([]byte, 4)

for {

n, err := reader.Read(p)

if err == io.EOF {

break

}

fmt.Print(string(p[:n]))

}

fmt.Println()

}

在[GO指南 rot13Reader，通过应用 rot13 代换密码对数据流进行修改](https://links.jianshu.com/go?to=https%3A%2F%2Ftour.go-zh.org%2Fmethods%2F23" \t "_blank)，用一个 [io.Reader](https://links.jianshu.com/go?to=https%3A%2F%2Fgo-zh.org%2Fpkg%2Fio%2F%23Reader" \t "_blank) 包装另一个 io.Reader，然后通过某种方式修改其数据流。参考[Go指南练习之《rot13Reader》](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fqq_27818541%2Farticle%2Fdetails%2F54379030" \t "_blank)

package main

import (

"io"

"os"

"strings"

)

type rot13Reader struct {

r io.Reader

}

// 转换byte 前进13位/后退13位

func rot13(b byte) byte {

switch {

case 'A' <= b && b <= 'M':

b = b + 13

case 'M' < b && b <= 'Z':

b = b - 13

case 'a' <= b && b <= 'm':

b = b + 13

case 'm' < b && b <= 'z':

b = b - 13

}

return b

}

// 重写Read方法

func (mr rot13Reader) Read(b []byte) (int, error) {

n, e := mr.r.Read(b)

for i := 0; i < n; i++ {

b[i] = rot13(b[i])

}

return n, e

}

func main() {

s := strings.NewReader("Lbh penpxrq gur pbqr!")

r := rot13Reader{s}

io.Copy(os.Stdout, &r)

}

3.标准writer  
标准库提供了许多已经实现了 io.Writer 的类型。下面是一个简单的例子，它使用 bytes.Buffer 类型作为 io.Writer 将数据写入内存缓冲区。

func main() {

proverbs := []string{

"Channels orchestrate mutexes serialize",

"Cgo is not Go",

"Errors are values",

"Don't panic",

}

var writer bytes.Buffer

for \_, p := range proverbs {

n, err := writer.Write([]byte(p))

if err != nil {

fmt.Println(err)

os.Exit(1)

}

if n != len(p) {

fmt.Println("failed to write data")

os.Exit(1)

}

}

fmt.Println(writer.String())

}

输出打印的内容：  
Channels orchestrate mutexes serializeCgo is not GoErrors are valuesDon't panic

4.自己实现一个 Writer  
下面我们来实现一个名为 chanWriter 的自定义 io.Writer ，它将其内容作为字节序列写入 channel 。

type chanWriter struct {

// ch 实际上就是目标资源

ch chan byte

}

func newChanWriter() \*chanWriter {

return &chanWriter{make(chan byte, 1024)}

}

func (w \*chanWriter) Chan() <-chan byte {

return w.ch

}

func (w \*chanWriter) Write(p []byte) (int, error) {

n := 0

// 遍历输入数据，按字节写入目标资源

for \_, b := range p {

w.ch <- b

n++

}

return n, nil

}

func (w \*chanWriter) Close() error {

close(w.ch)

return nil

}

func main() {

writer := newChanWriter()

go func() {

defer writer.Close()

writer.Write([]byte("Stream "))

writer.Write([]byte("me!"))

}()

for c := range writer.Chan() {

fmt.Printf("%c", c)

}

fmt.Println()

}

要使用这个 Writer，只需在函数 main() 中调用 writer.Write()（在单独的goroutine中）。因为 chanWriter 还实现了接口 io.Closer ，所以调用方法 writer.Close() 来正确地关闭channel，以避免发生泄漏和死锁。

5.io 包里其他有用的类型和方法

(1)os.File  
类型 os.File 表示本地系统上的文件。它实现了 io.Reader 和 io.Writer ，因此可以在任何 io 上下文中使用。例如，下面的例子展示如何将连续的字符串切片直接写入文件：

func main() {

proverbs := []string{

"Channels orchestrate mutexes serialize\n",

"Cgo is not Go\n",

"Errors are values\n",

"Don't panic\n",

}

file, err := os.Create("./proverbs.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

for \_, p := range proverbs {

// file 类型实现了 io.Writer

n, err := file.Write([]byte(p))

if err != nil {

fmt.Println(err)

os.Exit(1)

}

if n != len(p) {

fmt.Println("failed to write data")

os.Exit(1)

}

}

fmt.Println("file write done")

}

同时，io.File 也可以用作读取器来从本地文件系统读取文件的内容。例如，下面的例子展示了如何读取文件并打印其内容：

func main() {

file, err := os.Open("./proverbs.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

p := make([]byte, 4)

for {

n, err := file.Read(p)

if err == io.EOF {

break

}

fmt.Print(string(p[:n]))

}

}

(2)标准输入、输出和错误  
os 包有三个可用变量 os.Stdout ，os.Stdin 和 os.Stderr ，它们的类型为 \*os.File，分别代表 系统标准输入，系统标准输出 和 系统标准错误 的文件句柄。例如，下面的代码直接打印到标准输出：

func main() {

proverbs := []string{

"Channels orchestrate mutexes serialize\n",

"Cgo is not Go\n",

"Errors are values\n",

"Don't panic\n",

}

for \_, p := range proverbs {

// 因为 os.Stdout 也实现了 io.Writer

n, err := os.Stdout.Write([]byte(p))

if err != nil {

fmt.Println(err)

os.Exit(1)

}

if n != len(p) {

fmt.Println("failed to write data")

os.Exit(1)

}

}

}

(3)io.Copy()  
io.Copy() 可以轻松地将数据从一个 Reader 拷贝到另一个 Writer。它抽象出 for 循环模式（我们上面已经实现了）并正确处理 io.EOF 和 字节计数。 下面是我们之前实现的简化版本：

func main() {

proverbs := new(bytes.Buffer)

proverbs.WriteString("Channels orchestrate mutexes serialize\n")

proverbs.WriteString("Cgo is not Go\n")

proverbs.WriteString("Errors are values\n")

proverbs.WriteString("Don't panic\n")

file, err := os.Create("./proverbs.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

// io.Copy 完成了从 proverbs 读取数据并写入 file 的流程

if \_, err := io.Copy(file, proverbs); err != nil {

fmt.Println(err)

os.Exit(1)

}

fmt.Println("file created")

}

那么，我们也可以使用 io.Copy() 函数重写从文件读取并打印到标准输出的先前程序，如下所示：

func main() {

file, err := os.Open("./proverbs.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

if \_, err := io.Copy(os.Stdout, file); err != nil {

fmt.Println(err)

os.Exit(1)

}

}

(4)io.WriteString()  
此函数让我们方便地将字符串类型写入一个 Writer：

func main() {

file, err := os.Create("./magic\_msg.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

if \_, err := io.WriteString(file, "Go is fun!"); err != nil {

fmt.Println(err)

os.Exit(1)

}

}

(5)使用管道的 Writer 和 Reader  
类型 io.PipeWriter 和 io.PipeReader 在内存管道中模拟 io 操作。  
数据被写入管道的一端，并使用单独的 goroutine 在管道的另一端读取。  
下面使用 io.Pipe() 创建管道的 reader 和 writer，然后将数据从 proverbs 缓冲区复制到io.Stdout ：

func main() {

proverbs := new(bytes.Buffer)

proverbs.WriteString("Channels orchestrate mutexes serialize\n")

proverbs.WriteString("Cgo is not Go\n")

proverbs.WriteString("Errors are values\n")

proverbs.WriteString("Don't panic\n")

piper, pipew := io.Pipe()

// 将 proverbs 写入 pipew 这一端

go func() {

defer pipew.Close()

io.Copy(pipew, proverbs)

}()

// 从另一端 piper 中读取数据并拷贝到标准输出

io.Copy(os.Stdout, piper)

piper.Close()

}

(6)缓冲区 io  
标准库中 bufio 包支持 缓冲区 io 操作，可以轻松处理文本内容。  
例如，以下程序逐行读取文件的内容，并以值 '\n' 分隔：

func main() {

file, err := os.Open("./planets.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

defer file.Close()

reader := bufio.NewReader(file)

for {

line, err := reader.ReadString('\n')

if err != nil {

if err == io.EOF {

break

} else {

fmt.Println(err)

os.Exit(1)

}

}

fmt.Print(line)

}

}

(7)ioutil  
io 包下面的一个子包 utilio 封装了一些非常方便的功能  
例如，下面使用函数 ReadFile 将文件内容加载到 []byte 中。

package main

import (

"io/ioutil"

...

)

func main() {

bytes, err := ioutil.ReadFile("./planets.txt")

if err != nil {

fmt.Println(err)

os.Exit(1)

}

fmt.Printf("%s", bytes)

}