# Go语言的IO操作

在前一章节中。

本章重点为大家介绍如下的内容：

* 文件信息FileInfo
* 文件的常规操作（os包）
* 读取文件和写入文件（io及os包）
* ioutil包
* bufio包

## 文件信息

### FileInfo接口

FileInfo 描述一个文件，并由 Stat 和 Lstat 返回。

type FileInfo interface {

Name() string // 文件名，无扩展名

Size() int64 // 文件大小

Mode() FileMode // 文件模式位

ModTime() time.Time // 修改时间

IsDir() bool // 是否是目录

Sys() interface{} // 底层数据来源(可以return nil)

}

例:

package main

import (

"fmt"

"os"

)

func main() {

/\*

文件路径：

绝对路径：absolute

/Users/steven/Documents/go\_project/files/dsa.png

相对路径：relative 都是相当于当前的工程

.当前目录

..上一层

\*/

//绝对路径形式

fileInfo, err := os.Stat("/Users/steven/Documents/go\_project/files")

fileInfo, err = os.Stat("/Users/steven/Documents/go\_project/files/dsa.png")

//相对路径

fileInfo, err = os.Stat("./files/yesterday.mp3")

fileInfo, err = os.Stat("../node\_test/open.js")

if err != nil {

fmt.Println("err:", err.Error())

} else {

fmt.Printf("%v, %T \n", fileInfo, fileInfo)

//文件名

fmt.Println(fileInfo.Name())

//是否是目录

fmt.Println(fileInfo.IsDir())

//尺寸大小

fmt.Println(fileInfo.Size())

//权限

fmt.Println(fileInfo.Mode())

//修改时间

fmt.Println(fileInfo.ModTime())

}

}

【备注：】

一共10个字符。第一符号表示类型。如果是-表示文件，如果是d表示目录。

文件的权限一共有9个字符表示，分成三组，分别表示文件所属用户owner的权限，文件所属用户组group的权限，其他人others的权限；

r 表示读权限, w表示写权限， x表示执行权限。

并且此文件所属用户拥有读、写、执行三项权限，其余的用户组，其他用户不拥有任何权限（全部都是-）

例如：-rwxrwxrwx 表示这是一个文件，用户权限、用户组权限、其他人权限都是可读、可写、可操作

drwxr-xr-x 表示这是一个目录，用户权限是可读、可写、可操作，用户组权限是可读可操作，其他人权限是可读可操作。

还可以用8进制表示法：

r 4

w 2

x 1

- 0

例如：-rwxrwxrwx 权限用8进制表示为：0777

### 文件路径

路径操作相关的函数

filepath.IsAbs():判断是否是绝对路径

例：

fileName1 := "/Users/steven/Documents/go\_project/files/dsa.png"

ileName2 := "files/blockchain.txt"

fmt.Println(filepath.IsAbs(fileName1)) //true

fmt.Println(filepath.IsAbs(fileName2)) //false

filepath.Rel():获取相对路径

例：

fmt.Println(filepath.Rel("/Users/steven/Documents", fileName1))

filepath.Abs():获取绝对路径

例：

fmt.Println(filepath.Abs(fileName2))

path.Join():将任意数量的路径元素加入到单个路径中

例：

fmt.Println("获取父目录：", path.Join(fileName1, "."))

## 文件常规操作

### 创建目录

os.MKdir()：使用指定的名称和权限位创建一个新目录。如果有错误，它将是 \* PathError 类型。

os.MKdirAll()：创建一个名为 path 的目录以及任何必要的父项，并返回 nil ，否则返回错误。许可位 perm 用于 MkdirAll 创建的所有目录。如果 path 已经是一个目录，MkdirAll 什么也不做，并返回 nil 。

### 创建文件

os.Create() :使用模式0666（在 umask 之前）创建命名文件，如果它已经存在，则截断它。如果成功，返回文件上的方法可用于 I/O ； 关联的文件描述符具有模式 O\_RDWR 。如果有错误，它将是 \* PathError 类型。

### 打开文件

打开文件，让当前的程序和指定的文件建立了一个链接

os.Open(filename) 打开指定文件。如果成功，以 O\_RDONLY 模式打开文件并返回文件对象。如果有错误，返回 \* PathError 类型。

func OpenFile(name string, flag int, perm FileMode) (\*File, error)

FileMode代表文件的模式和权限位。这些字位在所有的操作系统都有相同的含义，因此文件的信息可以在不同的操作系统之间安全的移植。不是所有的位都能用于所有的系统，唯一共有的是用于表示目录的ModeDir位。

文件的打开方式:

O\_RDONLY：只读模式(read-only)

O\_WRONLY：只写模式(write-only)

O\_RDWR：读写模式(read-write)

O\_APPEND：追加模式(append)

O\_CREATE：文件不存在就创建(create a new file if none exists.)

OpenFile 是广义的公开称呼；大多数用户将使用“打开”或“创建”。它打开具有指定标志（O\_RDONLY等）和指定（0666等）的指定文件（如果适用）。如果成功，返回文件上的方法可用于 I/O 。如果有错误，它将是 \* PathError 类型

### 关闭文件

file.Close():关闭文件f，使文件不能用于读写。它返回可能出现的错误。关闭文件,程序和文件之间的链接断开。

### 删除

os.Remove() 删除已命名的文件或目录，该目录必须是个空目录。

os.RemoveAll() 移除所有的路径和它包含的任何子节点。

## 读写文件及复制文件

### 读取文件

读取文件的步骤：

打开文件->读取文件->关闭文件

使用file.Read([]byte)从文件中开始读取数据，返回值n是实际读取的字节数。如果读取到文件末尾，n为0，err为EOF（end of file）

例：

//打开文件

fileName = "./files/blockchain.txt"

file, err := os.Open(fileName)

if err != nil {

fmt.Println("打开文件有误：", err.Error())

return

}

//读取文件内容

//从file对应的文件中读取最多len(bs)个数据，存入到bs切片中，n是实际读入的数量

sli := make([]byte, 1024, 1024)

n := -1

for {

n, err = file.Read(sli)

if n == 0 || err == io.EOF {

fmt.Println("读取到文件末尾了，结束读取操作。。")

break

}

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

}

//关闭文件

file.Close()

### 写入文件

写入文件的步骤：

打开或创建文件->写入文件->关闭文件

例：

//打开文件

file, err := os.OpenFile("./test1/abc2.txt", os.O\_CREATE|os.O\_WRONLY, os.ModePerm)

if err != nil {

fmt.Println("打开文件有误：", err.Error())

}

fmt.Println(file)

//延迟执行关闭文件

defer file.Close()

//写入文件

n, err := file.Write([]byte("abcde123456"))

fmt.Println(err)

fmt.Println(n)

n, err = file.WriteString("中国人")

fmt.Println(err)

fmt.Println(n)

### 复制文件

/\*

该函数的功能：实现文件的拷贝，返回值是拷贝的总数量(字节),错误

\*/

func copyFile1(srcFile, destFile string) (int, error) {

file1, err := os.Open(srcFile)

if err != nil {

return 0, err

}

file2, err := os.OpenFile(destFile, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

if err != nil {

return 0, err

}

defer file1.Close()

defer file2.Close()

//拷贝数据

bs := make([]byte, 1024, 1024)

n := -1 //读取的数据量

total := 0

for {

n, err = file1.Read(bs)

if err == io.EOF || n == 0 {

fmt.Println("拷贝完毕。。")

break

}

if err != nil {

fmt.Println("err:" , err.Error())

return total, err

} else {

total += n

file2.Write(bs[:n])

}

}

return total, nil

}

func copyFile2(srcFile, destFile string) (int64, error) {

file1, err := os.Open(srcFile)

if err != nil {

return 0, err

}

file2, err := os.OpenFile(destFile, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

if err != nil {

return 0, err

}

defer file1.Close()

defer file2.Close()

return io.Copy(file2, file1)

}

## ioutil包

### ioutil包核心函数

ReadFile()

读取文件中的所有的数据，返回读取的字节数组

WriteFile()

向指定文件写入数据，如果文件不存在，则创建文件，写入数据之前清空文件

ReadDir()

读取一个目录下的子内容：子文件和子目录，但是仅有一层

TempDir(）

在当前目录下，创建一个以指定字符串为前缀的临时文件夹，并返回文件夹路径

TempFile()

在当前目录下，创建一个以指定字符串为前缀的文件，并以读写模式打开文件，并返回os.File指针对象

### 示例代码

//读取文件中的所有的数据

fileName1 := "./files/blockchain.txt"

data, err := ioutil.ReadFile(fileName1)

if err != nil {

fmt.Println("读取文件异常：", err.Error())

} else {

fmt.Println(string(data))

}

//写出数据

fileName2 := "./files/xyz.txt"

s1 := "helloworld面朝大海春暖花开"

err = ioutil.WriteFile(fileName2, []byte(s1), 0777)

if err != nil {

fmt.Println("写入文件异常：", err.Error())

} else {

fmt.Println("写入文件ok")

}

//ReadDir(),读取一个目录下的子内容：子文件和子目录，但是仅有一层

dirName := "./src/"

fileInfos, \_ := ioutil.ReadDir(dirName)

fmt.Println(len(fileInfos))

for i := 0; i < len(fileInfos); i++ {

//fmt.Printf("%T\n",fileInfos[i])

fmt.Println(i, fileInfos[i].Name(), fileInfos[i].IsDir())

}

## bufio包

### bufio的原理

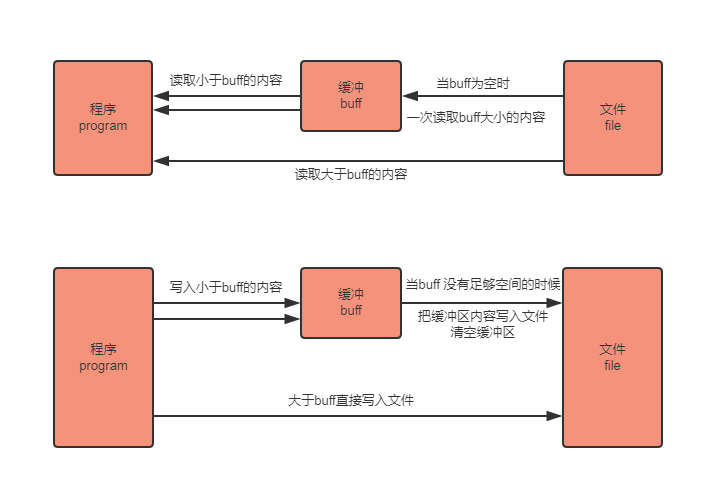
bufio实现了带缓冲的 I/O 操作，达到高效io读写。

bufio 包封装一个 Reader 及 Writer结构体对象；bufio 包中的Reader 及 Writer结构体分别实现了io.Reader和io.Writer接口；bufio包对io包下的Reader、Write对象进行包装，通过对io模块的封装，提供了数据缓冲功能，能够一定程度减少大块数据读写带来的开销，所以bufio 要比io的读写更快速。

bufio 是通过缓冲来提高效率；把文件读取进缓冲区之后，再读取的时候就可以避免文件系统的io ，从而提高速度；在进行写操作时，先把文件写入缓冲区，然后由缓冲写入文件系统。

有人可能会表示困惑，直接把 内容->文件 和 内容->缓冲->文件相比， 缓冲区好像没有起到作用嘛。

其实缓冲区的设计是为了存储多次的写入，最后一口气把缓冲区内容写入文件。当发起一次读写操作时，会首先尝试从缓冲区获取数据；只有当缓冲区没有数据时，才会从数据源获取数据更新缓冲。



bufio.Read(p []byte) 相当于读取大小len(p)的内容，思路如下：

当缓冲区有内容时，将缓冲区内容全部填入p并清空缓冲区

当缓冲区没有内容的时候且len(p)>len(buf)，即要读取的内容比缓冲区还要大，直接去文件读取即可

当缓冲区没有内容的时候且len(p)<len(buf)，即要读取的内容比缓冲区小，缓冲区从文件读取内容充满缓冲区，并将p填满（此时缓冲区有剩余内容）

以后再次读取时缓冲区有内容，将缓冲区内容全部填入p并清空缓存冲（此时和情况1一样）

bufio.Write(p []byte) 的思路如下：

判断buf中可用容量是否可以放下 p

如果能放下，直接把p拼接到buf后面，即把内容放到缓冲区

如果缓冲区的可用容量不足以放下，且此时缓冲区是空的，直接把p写入文件即可

如果缓冲区的可用容量不足以放下，且此时缓冲区有内容，则用p把缓冲区填满，把缓冲区所有内容写入文件，并清空缓冲区

判断p的剩余内容大小能否放到缓冲区，如果能放下（此时和步骤1情况一样）则把内容放到缓冲区

如果p的剩余内容依旧大于缓冲区，（注意此时缓冲区是空的，情况和步骤2一样）则把p的剩余内容直接写入文件

### bufio.Reader结构体

1.bufio.Reader的所有方法

func NewReader(rd io.Reader) \*Reader

func NewReaderSize(rd io.Reader, size int) \*Reader

func (b \*Reader) Buffered() int

func (b \*Reader) Discard(n int) (discarded int, err error) //丢弃接下来n个byte数据

func (b \*Reader) Peek(n int) ([]byte, error) //获取当前缓冲区内接下来的n个byte，但是不移动指针

func (b \*Reader) Read(p []byte) (n int, err error) //读取n个byte数据

func (b \*Reader) ReadByte() (byte, error) //读取一个byte

func (b \*Reader) ReadBytes(delim byte) ([]byte, error) //读取byte列表

func (b \*Reader) ReadLine() (line []byte, isPrefix bool, err error) //读取一行数据，由’\n’分隔

func (b \*Reader) ReadRune() (r rune, size int, err error) //读取一个utf-8字符

func (b \*Reader) ReadSlice(delim byte) (line []byte, err error)

func (b \*Reader) ReadString(delim byte) (string, error) //读取一个字符串

func (b \*Reader) Reset(r io.Reader) //清空整个缓冲区

func (b \*Reader) UnreadByte() error

func (b \*Reader) UnreadRune() error

func (b \*Reader) WriteTo(w io.Writer) (n int64, err error)

2.NewReader()与NewReaderSize()

将 rd 封装成一个拥有 size 大小缓存的 bufio.Reader 对象

NewReader 相当于 NewReaderSize(rd, 4096)

3.ReadLine()

ReadLine 是一个低级的原始的行读取操作

大多数情况下，应该使用 ReadBytes('\n') 或 ReadString('\n')，或者使用一个 Scanner

ReadLine 通过调用 ReadSlice 方法实现，返回的也是缓存的切片

ReadLine 尝试返回一个单行数据，不包括行尾标记（\n 或 \r\n）

如果在缓存中找不到行尾标记，则设置 isPrefix 为 true，表示查找未完成

同时读出缓存中的数据并作为切片返回

只有在当前缓存中找到行尾标记，才将 isPrefix 设置为 false，表示查找完成

可以多次调用 ReadLine 来读出一行

返回的数据在下一次读取操作之前是有效的

如果 ReadLine 无法获取任何数据，则返回一个错误信息（通常是 io.EOF）

4. ReadBytes()

ReadBytes 在 b 中查找 delim 并读出 delim 及其之前的所有数据

如果ReadBytes 在找到 delim 之前遇到错误，则返回遇到错误之前的所有数据，同时返回遇到的错误（通常是 io.EOF）

只有当 ReadBytes 找不到 delim 时，err 才不为 nil

对于简单的用途，使用 Scanner 可能更方便

5.ReadString()

ReadString 功能同 ReadBytes，只不过返回的是一个字符串

例：

//测试Reader的ReadString()

func testReader() {

fileName := "./files/blockchain.txt"

file1, \_ := os.Open(fileName) //看作是io包下的Reader，Write的实现

reader1 := bufio.NewReader(file1) //构建带缓存的Reader对象：bufio.Reader

fmt.Printf("%T\n", reader1)

for {

s1, err := reader1.ReadString('\n')

//ReadBytes reads until the first occurrence of delim in the input,returning a slice containing the data up to and including the delimiter.

fmt.Print(s1)

if err == io.EOF {

fmt.Println("\n读取完毕！")

break

}

}

file1.Close()

}

### bufio.Writer结构体

1.bufio.Writer的所有方法

func NewWriter(w io.Writer) \*Writer

func NewWriterSize(w io.Writer, size int) \*Writer

func (b \*Writer) Write(p []byte) (nn int, err error) // 写入n个 byte数据

func (b \*Writer) Reset(w io.Writer) // 重置当前缓冲区

func (b \*Writer) Flush() error // 清空当前缓冲区，将数据写入输出

func (b \*Writer) WriteByte(c byte) error // 写入一个字节

func (b \*Writer) WriteRune(r rune) (size int, err error） // 写入一个字符

func (b \*Writer) WriteString(s string) (int, error) // 写入字符串

2.NewWriter()与NewWriterSize()

func NewWriter(wr io.Writer) \*Writer

NewWriter 相当于 NewWriterSize(wr, 4096)

3.Write()和WriteString(）

例：

//通过bufio拷贝文件

func testWriter() {

fileName2 := "./files/music.mp3"

file2, \_ := os.Open(fileName2) //看作是io包下的Reader，Write的实现

reader2 := bufio.NewReader(file2)

fileName3 := "./files/abc.mp3"

file3, \_ := os.OpenFile(fileName3, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

writer1 := bufio.NewWriter(file3)

for {

bs, err := reader2.ReadBytes(' ')

writer1.Write(bs)

writer1.Flush()

if err == io.EOF {

fmt.Println("读取完毕。。")

break

}

}

file2.Close()

file3.Close()

}

### Scanner

实际使用中，更推荐使用Scanner对数据进行读取，而非直接使用Reader类。Scanner可以通过splitFunc将输入数据拆分为多个token，然后依次进行读取。和Reader类似，Scanner需要绑定到某个io.Reader上，通过NewScannner进行创建，函数声明如下：

func NewScanner(r io.Reader) \*Scanner

常用方法

func (s \*Scanner) Scan() bool

func (s \*Scanner) Text() string

func (s \*Scanner) Text() []byte

bufio模块提供了几个默认splitFunc，能够满足大部分场景的需求，包括：

ScanBytes，按照byte进行拆分

ScanLines，按照行(“\n”)进行拆分

ScanRunes，按照utf-8字符进行拆分

ScanWords，按照单词(” “)进行拆分

通过Scanner的Split方法，可以为Scanner指定splitFunc。使用方法如下：

scanner.split(bufio.ScanWords）

例：

//测试scanner

func testScanner() {

//s := ""

//fmt.Scanln(&s)

//fmt.Print(s)

reader1 := bufio.NewReader(os.Stdin)

scanner := bufio.NewScanner(reader1)

//ScanBytes，按照byte进行拆分

//ScanLines，按照行(“\n”)进行拆分

//ScanRunes，按照utf-8字符进行拆分

//ScanWords，按照单词(” “)进行拆分

scanner.Split(bufio.ScanWords)

//等待输入

for scanner.Scan() {

fmt.Println(scanner.Text())

if scanner.Text() == "q!" {

break

}

}

}

[第9章 Go语言的IO操作 1](#_Toc761790533)

[9.1 文件信息 1](#_Toc115984717)

[9.1.1 FileInfo接口 1](#_Toc1587470790)

[9.1.2 文件路径 3](#_Toc284737202)

[9.2 文件常规操作 4](#_Toc984588498)

[9.2.1 创建目录 4](#_Toc1617385751)

[9.2.2 创建文件 4](#_Toc554313331)

[9.2.3 打开文件 5](#_Toc560093431)

[9.2.4 关闭文件 5](#_Toc1069470016)

[9.2.5 删除 5](#_Toc144433522)

[9.3 读写文件及复制文件 6](#_Toc837683144)

[9.3.1 读取文件 6](#_Toc37811476)

[9.3.2 写入文件 7](#_Toc1989801267)

[9.3.3 复制文件 7](#_Toc1974543385)

[9.4 ioutil包 9](#_Toc1085874604)

[9.4.1 ioutil包核心函数 9](#_Toc978437222)

[9.4.2 示例代码 10](#_Toc1312105075)

[9.5 bufio包 11](#_Toc40424482)

[9.5.1 bufio的原理 11](#_Toc809436522)

[9.5.2 bufio.Reader结构体 13](#_Toc2038205156)

[9.5.3 bufio.Writer结构体 15](#_Toc1602403595)

[9.5.4 Scanner 16](#_Toc4804138)