文件IO

打开文件主要使用os模块的`Open`和`OpenFile`。

Open

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    filename := "o:/test.txt"
    // Open为快捷的只读打开
    if f, err := os.Open(filename); err == nil {
        fmt.Printf("%T, %[1]v\n", f) // *os.File, &{0xc00011c780}
        // 可以读取内容了
        f.Close() // 用完一定要关闭
    } else {
        fmt.Println(err)
        // open o:/test.txt: The system cannot find the file specified.
    }
}
```

打开后可以得到一个文件操作句柄，这里是os.File类型的指针，使用它就可以操作文件了。

```go
// 读取文件内容
filename := "o:/test.txt"
// Open为快捷的只读打开
if f, err := os.Open(filename); err == nil {
    defer f.Close()
    fmt.Printf("%T, %[1]v\n", f) // *os.File, &{0xc00011c780}
    // 可以读取内容了
    buffer := make([]byte, 2)
    for {
        // 为了测试每次读2个字节，直到读到文件末尾
        // 注意每次buffer未清空，根据n来判断
        n, err := f.Read(buffer)
        fmt.Println(n, err)
        if n == 0 {
            break // 什么都没有读取到，说明读到了文件的结尾EOF
        }
        fmt.Println(buffer[:n], string(buffer[:n]))
    }
    // f.Close() // 用完一定要关闭，如何保证一定关闭?
}
```

可以看出，文件有可能很大，读取实际上是一点一点读到内存的，而且这个操作目前看来是从前向后读取。那么，能否指定位置读取呢?

实际上，任何写入存储设备的数据都是序列化后的序列，都可以看做一个大的字节数组。读取操作实际上从是获得字节数组上部分或全部内容。读取时，似乎有一个针指向某个位置，这个针随着读取向后移动。那么这个针，可以随意移动吗？

定位

Seek(offset int64, whence int) 的whence

- whence=0 相对于开头，offset 只能正，负报错
- whence=1 相对于当前，offset 可正可负，但是负指针不能超左边界
- whence=2 相对于结尾，offset 可正可负，但是负指针不能超左边界

Seek到右边界外，再读取就是长度为0，读不到内容了

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    // 读取文件内容
    filename := "o:/test.txt" // 内容为0123456789
    // Open为快捷的只读打开
    if f, err := os.Open(filename); err == nil {
        defer f.Close()
        buffer := make([]byte, 5)
        var n int
        // 指定位置，从头向后偏移3字节开始读取长度5
        n, _ = f.ReadAt(buffer, 3)
        fmt.Println(n, f.Fd(), f.Name())
        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))

        // 从头读，从索引0开始读。Read三次，请问每次打印什么？
        n, _ = f.Read(buffer)
        fmt.Println(n, f.Fd(), f.Name())
        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))

        n, _ = f.Read(buffer)
        fmt.Println(n, f.Fd(), f.Name())
        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))

        n, _ = f.Read(buffer)
        fmt.Println(n, f.Fd(), f.Name())
        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))

        // 定位，调整Seek的参数看看效果，观察e是否有错误
        off, e := f.Seek(60, 1)
        if e == nil {
            fmt.Println(off, "@@@")
            buffer = make([]byte, 5)
            n, _ = f.Read(buffer)
            fmt.Println(n, buffer)
        } else {
```

```
            fmt.Println(e)
        }
    }
}
```

带缓冲读取

文件使用Read读取，非常底层，操作起来很不方便。Go语言提供了bufio包实现了对文件的二进制或文本处理的方法。

要对文件使用带buffer的方式读取，`func NewReader(rd io.Reader) *Reader`，io.Reader接口要求只要实现Read方法就行，os.File实现了该接口的Read方法。默认是4096字节。

- reader.ReadByte() (byte, error)，成功返回一个字节，失败返回错误
- reader.Read(p []byte) (n int, err error)
  - 成功则将读取到的数据写入p，并返回数据长度n
  - 读取到文件结尾时，n为0且err为io.EOF
- reader.ReadBytes(delim byte) ([]byte, error)
  - 成功则将读取到的数据写入字节切片中，切片中最后一个元素就是分隔符
  - 读取到文件结尾时，err为io.EOF
- reader.ReadString(delim byte) (string, error)
  - 成功则将返回字符串中，字符串中最后一个字符就是分隔符
  - 读取到文件结尾时，err为io.EOF

读文件，如下

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {
    // 读取文件内容
    filename := "o:/test.txt" // 内容为0123456789\nabc\nxyz
    // Open为快捷的只读打开，f是*os.File,
    if f, err := os.Open(filename); err == nil {
        defer f.Close()
        reader := bufio.NewReader(f) // File实现了Read方法
        // reader可以按照字节或字符读取
        b1, err := reader.ReadBytes('5')
        fmt.Println(string(b1), err) // 012345 nil 尾部带分隔符
        b2 := make([]byte, 3)
        n, err := reader.Read(b2)
        fmt.Println(n, b2, string(b2[:n]), err) // 3 [54 55 56] 678 nil
        // 特别注意文件指针的移动
        b3, err := reader.ReadBytes('\n')
        fmt.Println(b3, string(b3), err) // [57 10] 9\n nil

        b4, err := reader.ReadSlice('\n')
```

```
        fmt.Println(b4, string(b4), err) // [97 98 99 10] abc\n <nil>

        line, err := reader.ReadString('\n')
        fmt.Println(line, err) // xyz EOF，意思是读到了文件末尾EOF还没有找到\n
        fmt.Println(
            strings.TrimRight(line, "\n"), // 移除右边的换行符
        )
    }
}
```

注意文件test.txt保存数据的编码是utf-8，否则乱码。

flag

```
const (
    // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.下面3个中的1个必须被指定
    O_RDONLY int = syscall.O_RDONLY // open the file read-only.
    O_WRONLY int = syscall.O_WRONLY // open the file write-only.
    O_RDWR   int = syscall.O_RDWR   // open the file read-write.
    // The remaining values may be or'ed in to control behavior.剩余的值用或符号加入来控制行为
    O_APPEND int = syscall.O_APPEND // append data to the file when writing.追加写入
    O_CREATE int = syscall.O_CREAT  // create a new file if none exists.文件不存在则创建
    O_EXCL   int = syscall.O_EXCL   // used with O_CREATE, file must not exist.和O_CREATE一起使用，要求文件必须不存在。也就是文件已经存在报错
    O_SYNC   int = syscall.O_SYNC   // open for synchronous I/O.同步IO，等待上一次IO完成再进行
    O_TRUNC  int = syscall.O_TRUNC  // truncate regular writable file when opened.打开时清空可写文件
)
```

- O_RDONLY、O_WRONLY、O_APPEND、O_RDWR 单独使用，如果文件不存在都报错。也就是要求读写的前提是文件必须存在
- O_RDONLY 只读打开，用的较少，因为使用Open方法只读打开用的就是它

注意：os.O_RDONLY|os.O_APPEND 等价于 os.O_RDWR | os.O_APPEND，但是前者这种写法容易歧义，不要使用，推荐使用后者。

常用文件操作

使用flag就可以操作文件的读写模式。

```
package main

import (
    "fmt"
    "os"
)

```

```go
func main() {
    filename := "o:/t1.txt"
    flag := os.O_WRONLY                // 文件必须存在
    flag = os.O_WRONLY | os.O_CREATE // 文件不存在就创建后写入，文件存在就写入，从头写覆盖
    flag = os.O_CREATE                 // 相当于os.O_WRONLY | os.O_CREATE
    flag = os.O_CREATE | os.O_TRUNC  // 文件不存在创建新文件从，从头写；文件存在清空文件，从头写
    flag = os.O_WRONLY | os.O_APPEND // 文件末尾追加写，但是文件得存在
    flag = os.O_APPEND                 // 相当于os.O_WRONLY | os.O_APPEND
    flag = os.O_APPEND | os.O_CREATE // 文件不存在则创建新文件，文件末尾追加写
    flag = os.O_EXCL                   // 不要单独使用
    flag = os.O_EXCL | os.O_CREATE   // 文件存在报错，不存在创建新文件，从头开始写
    flag = os.O_RDWR                   // 既能读又能写，从头开始，要求文件存在
    if f, err := os.OpenFile(filename, flag, 0o640); err == nil {
        defer f.Close()
        fmt.Println(f)
        f.WriteString("abcd")

    } else {
        fmt.Println(err, "!!!")
    }
}
```

总结一下：

- O_WRONLY 只读，从头读，文件要存在
- O_WRONLY 只写，从头写，文件要存在。如果文件已存在有内容，从头覆盖
- O_CREATE | O_TRUNC 没有文件创建新文件，从头写；有文件清空内容从头写
- O_APPEND 追加写，文件要存在
- O_CREATE 文件存在，从头写；文件不存在创建新文件，从头写
- O_EXCL | O_CREATE 文件不存在创建新文件，从头写；文件存在报错
- O_RDWR 既能读又能写，从头开始

大家根据自己的需求，使用各种常量组合使用模式。

带缓冲读写

```go
package main

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

func main() {
    filename := "o:/t1.txt"
    flag := os.O_RDWR | os.O_CREATE | os.O_TRUNC // 文件不存在创建，文件存在则清空，可读可写，从头开始
    if f, err := os.OpenFile(filename, flag, os.ModePerm); err == nil {
        defer f.Close()

```

```
        r := bufio.NewReader(f)
        w := bufio.NewWriter(f)

        w.WriteString("0123456789\n")
        w.WriteString("abc\n")
        w.Flush() // 写入文件

        fmt.Println("~~~~~~~~~~~~~~")
        f.Seek(0, 0) // 底层共用同一个f，指针已经指到了EOF，拉回到开始
        fmt.Println(r.ReadString('\n'))
        fmt.Println(r.ReadString('\n'))
    } else {
        fmt.Println(err, "!!!")
    }
}
```

- os.Create(name string) (*File, error) 创建文件，本质上就是`OpenFile(name, O_RDWR|O_CREATE|O_TRUNC, 0666)`，相当于touch

路径和目录

路径

存储设备保存着数据，但是得有一种方便的模式让用户可以定位资源位置，操作系统采用一种路径字符串的表达方式，这是一棵倒置的层级目录树，从根开始。

- 相对路径：不是以根目录开始的路径，例如`a/b`、`a/b/c`、`.`、`../a/b`、`./a/b/c`
- 绝对路径：以斜杆开始的路径，例如`/a/b`、`/a/b/c`、`/t/../a/b`，window需要盘符`e:\a\b\c`
- 路径分隔符：windows为`\`，但是也支持`/`；Linux系统等为`/`

路径处理包

为了方便处理，Go语言标准库提供path包和更加方便的path/filepath包，使用filepath即可。

路径拼接

由于路径是字符串，直接使用字符串拼接就可以了，也可以使用join方法。

```
p1 := "/a/b" + "/" + "c/d" + "/" + "f"
p2 := filepath.Join("a/b", "c/d", "f")
```

路径分解

```
p1 := "/a/b/c/d/f/main.ini"
dir, file := filepath.Split(p1)
fmt.Println(dir, file) // dir, basename
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
fmt.Println(filepath.Dir(p1))  // dir
fmt.Println(filepath.Base(p1)) // basename
fmt.Println(filepath.Ext(p1))  // 扩展名.ini
```

目录

- os.UserHomeDir()，家目录
- os.Getwd()，当前工作目录
- os.Mkdir(name string, perm FileMode) error，要求父路径都已经存在，才能创建目录成功，否则报错
- os.MkdirAll(path string, perm FileMode) error，相当于mkdir -p

存在性

如果os.Stat(path)返回错误不为nil，通过os.IsExist(err)或os.IsNotExist(err)来判断文件存在否

```go
p1 := "/a/b/c/d/f/main.ini"
info, err := os.Stat(p1)
fmt.Println(info, err, os.IsExist(err), os.IsNotExist(err))

// 如果文件不存在，创建父目录，创建文件；文件存在，就不创建父目录和文件
dir := filepath.Dir(p1)
err = os.MkdirAll(dir, os.ModePerm) // 创建所有父目录
fmt.Println(err, "@@@")
if f, err := os.Create(p1); err == nil {
    defer f.Close()
    fmt.Println(f, "!!!") // 创建文件成功
} else {
    fmt.Println(err, "###")
}
```

总结表如下

文件	err	os.IsExist(err)	os.IsNotExist(err)
不存在	错误	false	true
存在	nil	false	false

因此使用os.IsExist(err)容易让人疑惑，建议使用os.IsNotExist(err)。

stat

stat返回目录或文件的信息。如果是符号链接，stat会跟踪。如果不想跟踪，请使用Lstat。

```go
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    p1 := "/a/b/c/d/f/main.ini"
    dir := filepath.Dir(p1)
    err := os.MkdirAll(dir, os.ModePerm) // 创建所有父目录
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
    f, err := os.Create(p1)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
    // 文件stat
    info, err := os.Stat(p1)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(
        info.Name(),    // basename
        info.IsDir(),   // 是目录吗?
        info.Mode(),    // mode
        info.ModTime(), // mtime
        info.Size(),    // size
    )
    fmt.Printf("%O\n", int(info.Mode()))
    // 目录stat
    info, err = os.Stat(dir)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(
        info.Name(),    // basename
        info.IsDir(),   // 是目录吗?
        info.Mode(),    // mode
        info.ModTime(), // mtime
        info.Size(),    // size
    )
}
```

绝对路径

```
p1 := "/a/b/c/d/f/main.ini"
fmt.Println(filepath.IsAbs(p1))                 // 是否是绝对路径
fmt.Println(filepath.Abs(p1))                   // 取绝对路径
fmt.Println(filepath.Abs("a/b"))                // os.Getwd()下的a/b
fmt.Println(os.Getwd())                         // 当前工作路径
fmt.Println(filepath.Rel("/a/b", "/a/b/c/d")) // 计算相对路径
```

遍历

filepath.WalkDir和filepath.Walk

- **递归**遍历目录树
- 每遍历到一个节点，都会执行回调函数，只不过返回的参数略有不同
- 都不跟踪软链接

- 内部都是按照字典序输出
- 深度优先

Go 1.16加入的WalkDir效率更高一些。

ioutil.ReadDir(path) 不递归遍历当前目录。

```go
package main

import (
    "fmt"
    "io/fs"
    "io/ioutil"
    "path/filepath"
)

func main() {
    p1 := "/a"
    // WalkDir() 和 Walk遍历递归遍历，包含自身
    filepath.WalkDir(p1, func(path string, d fs.DirEntry, err error) error {
        fmt.Println(path, d.IsDir(), d.Name(), err) // 递归读出目录和文件
        // fmt.Println(d.Info())
        return err
    })
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    //
    filepath.Walk(p1, func(path string, info fs.FileInfo, err error) error {
        fmt.Println(path, info.IsDir(), info.Name(), err) // 递归读出每一个目录和文件
        // fmt.Println(info)
        return err
    })

    // 不递归遍历当前目录
    fi, err := ioutil.ReadDir(p1)
    if err != nil {
        fmt.Println(err)
        return
    }
    for i, v := range fi {
        fmt.Println(i, v.IsDir(), v.Name(), v.Mode())
        fmt.Println(filepath.Join(p1, v.Name()))
    }
}
```