

文件IO

打开文件主要使用os模块的 `open` 和 `OpenFile`。

Open

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6 )
7
8 func main() {
9     filename := "o:/test.txt"
10    // open为快捷的只读打开
11    if f, err := os.Open(filename); err == nil {
12        fmt.Printf("%T, %[1]v\n", f) // *os.File, &{0xc00011c780}
13        // 可以读取内容了
14        f.Close() // 用完一定要关闭
15    } else {
16        fmt.Println(err)
17        // open o:/test.txt: The system cannot find the file specified.
18    }
19 }
```

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。

```
1 // 读取文件内容
2 filename := "o:/test.txt"
3 // open为快捷的只读打开
4 if f, err := os.Open(filename); err == nil {
5     defer f.Close()
6     fmt.Printf("%T, %[1]v\n", f) // *os.File, &{0xc00011c780}
7     // 可以读取内容了
8     buffer := make([]byte, 2)
9     for {
10        // 为了测试每次读2个字节，直到读到文件末尾
11        // 注意每次buffer未清空，根据n来判断
12        n, err := f.Read(buffer)
13        fmt.Println(n, err)
14        if n == 0 {
15            break // 什么都没有读取到，说明读到了文件的结尾EOF
16        }
17        fmt.Println(buffer[:n], string(buffer[:n]))
18    }
19    // f.Close() // 用完一定要关闭，如何保证一定关闭？
20 }
```

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。读取操作实际上是从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

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6 )
7
8 func main() {
9     // 读取文件内容
10    filename := "o:/test.txt" // 内容为0123456789
11    // Open为快捷的只读打开
12    if f, err := os.Open(filename); err == nil {
13        defer f.Close()
14        buffer := make([]byte, 5)
15        var n int
16        // 指定位置，从头向后偏移3字节开始读取长度5
17        n, _ = f.ReadAt(buffer, 3)
18        fmt.Println(n, f.Fd(), f.Name())
19        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))
20
21        // 从头读，从索引0开始读。Read三次，请问每次打印什么？
22        n, _ = f.Read(buffer)
23        fmt.Println(n, f.Fd(), f.Name())
24        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))
25
26        n, _ = f.Read(buffer)
27        fmt.Println(n, f.Fd(), f.Name())
28        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))
29
30        n, _ = f.Read(buffer)
31        fmt.Println(n, f.Fd(), f.Name())
32        fmt.Println(buffer, len(buffer), cap(buffer), string(buffer[:n]))
33
34        // 定位，调整Seek的参数看看效果，观察e是否有错误
35        off, e := f.Seek(60, 1)
36        if e == nil {
37            fmt.Println(off, "===")
38            buffer = make([]byte, 5)
39            n, _ = f.Read(buffer)
40            fmt.Println(n, buffer)
41        } else {
```

```

42         fmt.Println(e)
43     }
44 }
45 }

```

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

```

1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "strings"
8 )
9
10 func main() {
11     // 读取文件内容
12     filename := "o:/test.txt" // 内容为0123456789\nabc\nxyz
13     // open为快捷的只读打开，f是*os.File，
14     if f, err := os.Open(filename); err == nil {
15         defer f.Close()
16         reader := bufio.NewReader(f) // File实现了Read方法
17         // reader可以按照字节或字符读取
18         b1, err := reader.ReadBytes('5')
19         fmt.Println(string(b1), err) // 012345 nil 尾部带分隔符
20         b2 := make([]byte, 3)
21         n, err := reader.Read(b2)
22         fmt.Println(n, b2, string(b2[:n]), err) // 3 [54 55 56] 678 nil
23         // 特别注意文件指针的移动
24         b3, err := reader.ReadBytes('\n')
25         fmt.Println(b3, string(b3), err) // [57 10] 9\n nil
26
27         b4, err := reader.ReadSlice('\n')

```

```

28     fmt.Println(b4, string(b4), err) // [97 98 99 10] abc\n <nil>
29
30     line, err := reader.ReadString('\n')
31     fmt.Println(line, err) // xyz EOF, 意思是读到了文件末尾EOF还没有找到\n
32     fmt.Println(
33         strings.TrimRight(line, "\n"), // 移除右边的换行符
34     )
35
36     if err == io.EOF {
37         f.Seek(0, 0)
38     }
39 }
40 }

```

注意文件test.txt保存数据的编码是utf-8，否则乱码。

注：使用bufio.Read*方法时，在开始或EOF时使用Seek，否则不起作用或导致数据可能首位相接等奇怪问题

flag

```

1  const (
2      // 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
3      O_RDONLY int = syscall.O_RDONLY // open the file read-only.
4      O_WRONLY int = syscall.O_WRONLY // open the file write-only.
5      O_RDWR  int = syscall.O_RDWR   // open the file read-write.
6      // 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
7      O_APPEND int = syscall.O_APPEND // append data to the file when writing.
      追加写入
8      O_CREATE int = syscall.O_CREAT  // create a new file if none exists.文件
      不存在则创建
9      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
10     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
11     O_TRUNC  int = syscall.O_TRUNC  // truncate regular writable file when
      opened.打开时清空可写文件
12 )

```

- 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。

```

1  package main
2

```

```

3 import (
4     "fmt"
5     "os"
6 )
7
8 func main() {
9     filename := "o:/t1.txt"
10    flag := os.O_WRONLY // 文件必须存在
11    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
12    flag = os.O_WRONLY | 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
13    flag = os.O_WRONLY | os.O_APPEND // 文件末尾追加写，但是文件得存在
14    flag = os.O_WRONLY | os.O_APPEND | os.O_CREATE // 文件不存在则创建新文件，文
    件末尾追加写
15    flag = os.O_EXCL // 不要单独使用
16    flag = os.O_WRONLY | 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
17    flag = os.O_RDWR // 既能读又能写，从头开始，要求文件存在
18    if f, err := os.OpenFile(filename, flag, 0o640); err == nil {
19        defer f.Close()
20        fmt.Println(f)
21        f.WriteString("abcd")
22
23    } else {
24        fmt.Println(err, "!!!")
25    }
26 }

```

总结一下：

- 读
 - O_RDONLY 只读，从头读，文件要存在
- 写
 - 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_RDWR 既能读又能写，从头开始
 - 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
 - 只要是写，请配合O_WRONLY或os.O_RDWR增加写能力

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

```

1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"

```

```

7 )
8
9 func main() {
10     filename := "o:/t1.txt"
11     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
12     if f, err := os.OpenFile(filename, flag, os.ModePerm); err == nil {
13         defer f.Close()
14
15         r := bufio.NewReader(f)
16         w := bufio.NewWriter(f)
17
18         w.WriteString("0123456789\n")
19         w.WriteString("abc\n")
20         w.Flush() // 写入文件
21
22         fmt.Println("~~~~~")
23         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
24         fmt.Println(r.ReadString('\n'))
25         fmt.Println(r.ReadString('\n'))
26     } else {
27         fmt.Println(err, "!!!")
28     }
29 }

```

- `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。

```

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

```

路径分解

```
1 p1 := "/a/b/c/d/f/main.ini"
2 dir, file := filepath.Split(p1)
3 fmt.Println(dir, file) // dir, basename
4 fmt.Println("~~~~~")
5 fmt.Println(filepath.Dir(p1)) // dir
6 fmt.Println(filepath.Base(p1)) // basename
7 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

```
1 p1 := "/a/b/c/d/f/main.ini"
2 info, err := os.Stat(p1)
3 fmt.Println(info, err, os.IsExist(err), os.IsNotExist(err))
4
5 // 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
6 dir := filepath.Dir(p1)
7 err = os.MkdirAll(dir, os.ModePerm) // 创建所有父目录
8 fmt.Println(err, "!!!")
9 if f, err := os.Create(p1); err == nil {
10     defer f.Close()
11     fmt.Println(f, "!!!") // 创建文件成功
12 } else {
13     fmt.Println(err, "###")
14 }
```

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。

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6     "path/filepath"
7 )
8
9 func main() {
10     p1 := "/a/b/c/d/f/main.ini"
11     dir := filepath.Dir(p1)
12     err := os.MkdirAll(dir, os.ModePerm) // 创建所有父目录
13     if err != nil {
14         fmt.Println(err)
15         return
16     }
17     f, err := os.Create(p1)
18     if err != nil {
19         fmt.Println(err)
20         return
21     }
22     defer f.Close()
23     // 文件stat
24     info, err := os.Stat(p1)
25     if err != nil {
26         fmt.Println(err)
27         return
28     }
29     fmt.Println(
30         info.Name(),    // basename
31         info.IsDir(),   // 是目录吗?
32         info.Mode(),    // mode
33         info.ModTime(), // mtime
34         info.Size(),    // size
35     )
36     fmt.Printf("%0\n", int(info.Mode()))
37     // 目录stat
38     info, err = os.Stat(dir)
39     if err != nil {
40         fmt.Println(err)
41         return
42     }
43     fmt.Println(
44         info.Name(),    // basename
45         info.IsDir(),   // 是目录吗?
46         info.Mode(),    // mode
47         info.ModTime(), // mtime
48         info.Size(),    // size
49     )
50 }
```


绝对路径

```
1 p1 := "/a/b/c/d/f/main.ini"
2 fmt.Println(filepath.IsAbs(p1))           // 是否是绝对路径
3 fmt.Println(filepath.Abs(p1))             // 取绝对路径
4 fmt.Println(filepath.Abs("a/b"))          // os.Getwd() 下的a/b
5 fmt.Println(os.Getwd())                   // 当前工作路径
6 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。

```
1 package main
2
3 import (
4     "fmt"
5     "io/fs"
6     "io/ioutil"
7     "path/filepath"
8 )
9
10 func main() {
11     p1 := "/a"
12     // walkDir() 和 walk遍历递归遍历，包含自身
13     filepath.WalkDir(p1, func(path string, d fs.DirEntry, err error) error {
14         fmt.Println(path, d.IsDir(), d.Name(), err) // 递归读出目录和文件
15         // fmt.Println(d.Info())
16         return err
17     })
18     fmt.Println("~~~~~")
19     //
20     filepath.Walk(p1, func(path string, info fs.FileInfo, err error) error {
21         fmt.Println(path, info.IsDir(), info.Name(), err) // 递归读出每一个目录
22         // fmt.Println(info)
23         return err
24     })
25
26     // 不递归遍历当前目录
27     fi, err := ioutil.ReadDir(p1)
28     if err != nil {
29         fmt.Println(err)
30     }
31     return
```

```
31 }
32 for i, v := range fi {
33     fmt.Println(i, v.IsDir(), v.Name(), v.Mode())
34     fmt.Println(filepath.Join(p1, v.Name()))
35 }
36 }
```

