Go语言文件操作

目录：

一、文件信息

（一）、FileInfo接口

1、接口属性

```
type FileInfo interface {
    Name() string       // base name of the file
    Size() int64        // length in bytes for regular files; system-dependent for others
    Mode() FileMode     // file mode bits
    ModTime() time.Time // modification time
    IsDir() bool        // abbreviation for Mode().IsDir()
    Sys() interface{}   // underlying data source (can return nil)
}
```

2、fileStat结构体

- A fileStat is the implementation of FileInfo returned by Stat and Lstat.

```
type fileStat struct {
    name    string
    size    int64
    mode    FileMode
```

```
    modTime time.Time
    sys     syscall.Stat_t
}
```

3、fileStat结构体的常用方法

- func (fs *fileStat) Name() string { return fs.name }
- func (fs *fileStat) IsDir() bool { return fs.Mode().IsDir() }
- func (fs *fileStat) Size() int64   { return fs.size }
- func (fs *fileStat) Mode() FileMode   { return fs.mode }
- func (fs *fileStat) ModTime() time.Time { return fs.modTime }
- func (fs *fileStat) Sys() interface{}   { return &fs.sys }

4、示例代码

```go
package main
import (
  "fmt"
  "os"
)

func main() {
  /*
  文件路径：
  1、绝对路径：absolute
    /Users/steven/Documents/go_project/files/dsa.png
  2、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 表示是个文件，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 表示是个目录，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

- 参考网站：http://permissions-calculator.org/

（二）、文件路径

1、绝对路径：absolute

- /Users/steven/Documents/go_project/files/dsa.png

2、相对路径：relative 都是相当于当前的工程

- .当前目录
- ..上一层

3、跟路径相关的函数

（1）、判断是否是绝对路径filepath.IsAbs()

- fileName1 := "/Users/steven/Documents/go_project/files/dsa.png"
- fileName2 := "files/blockchain.txt"
- fmt.Println(filepath.IsAbs(fileName1)) //true
- fmt.Println(filepath.IsAbs(fileName2)) //false

（2）、获取相对路径filepath.Rel()

- Rel returns a relative path that is lexically equivalent to targpath when joined to basepath with an intervening separator.
- fmt.Println(filepath.Rel("/Users/steven/Documents", fileName1))

（3）、获取绝对路径filepath.Abs()

- Abs returns an absolute representation of path.If the path is not absolute it will be joined with the current working directory to turn it into an absolute path. The absolute path name for a given file is not guaranteed to be unique.
- fmt.Println(filepath.Abs(fileName2))

（4）、拼接路径path.Join()

- Join joins any number of path elements into a single path, adding a separating slash if necessary. The result is Cleaned; in particular,all empty strings are ignored.
- 将任意数量的路径元素加入到单个路径中
- fmt.Println("获取父目录：", path.Join(fileName1, "."))

二、文件常规操作

（一）、创建目录， 如果目录存在，创建失败

1、os.MKdir()

- Mkdir creates a new directory with the specified name and permission bits. If there is an error, it will be of type *PathError.
- os.MKdir()，仅创建一层

2、os.MKdirAll()

- os.MKdirAll()，创建多层目录

（二）、创建文件：如果文件存在，会覆盖

- Create creates the named file with mode 0666 (before umask), truncating it if it already exists.
- os.Create() -->*File
- 该函数本质上是在调用os.OpenFile()函数

（三）、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
- Open opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O_RDONLY.
- os.Open(filename) -->*File
- os.Open()函数本质上是在调用os.OpenFile()函数
- os.OpenFile(filename, mode, perm) -->*File
  - 第一个参数：文件名称
  - 第二个参数：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.)
  - 第三个参数：文件的权限：文件不存在创建文件，需要指定权限

（四）、关闭文件：

- 关闭文件,程序和文件之间的链接断开。
- *File指针的方法
- file.Close()

（五）、删除：

- Remove removes the named file or directory.If there is an error, it will be of type *PathError.
- os.Remove() 删除已命名的文件或目录，该目录必须是个空目录。
- RemoveAll removes path and any children it contains.
- os.RemoveAll() 移除所有的路径和它包含的任何子节点。

三、读写文件及复制文件

（一）、读取文件

1、读取文件的步骤：

- 打开文件
  - os.Open(fileName)
- 读取文件
  - file.Read([]byte)-->n,err
  - Read reads up to len(b) bytes from the File.It returns the number of bytes read and any error encountered.At end of file, Read returns 0, io.EOF.
  - 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）
- 关闭文件

2、示例代码

```
//step1：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
```

```
}
//step2：读/写
//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]))
}

//step3：关闭文件
file.Close()
```

（二）、写入文件

1、写入文件的步骤：

- 打开或创建文件
  - os.OpenFile()
- 写入文件
  - file.Write([]byte)-->n,err
  - file.WriteString(string)-->n,err
- 关闭文件

2、示例代码

```
/*
写出数据到文件：*/
//1、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)

//2、关闭文件
defer file.Close()

//3、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)
```

（三）、复制文件

1、示例代码

```
/*
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)
}
```

（四）、seek

四、ioutil包

(一)、ioutil包核心函数

1、ReadFile()

- 读取文件中的所有的数据，返回读取的字节数组

2、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

3、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

4、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

5、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

（二）、示例代码

```
//1.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))
}

//2.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")
}

//3、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())
}
```

五、bufio包

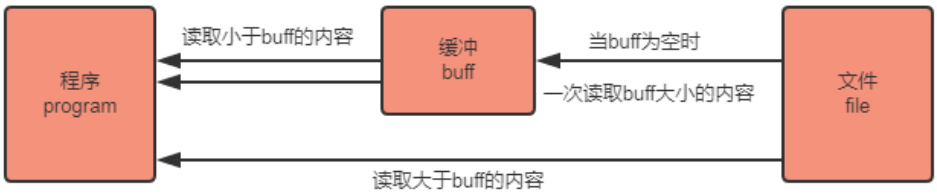
（一）、bufio的原理

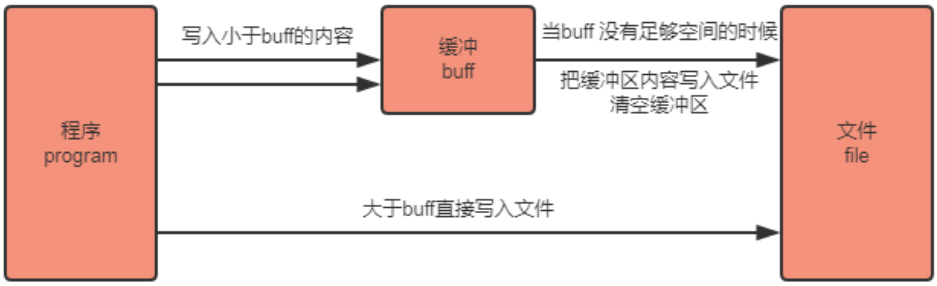
1、概述

- bufio实现了带缓冲的 I/O 操作，达到高效io读写。
  - buffer缓冲
  - io：input/output
- 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。

2、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。





3、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）

4、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

（二）、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

6、示例代码

```
//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()
    }
```

（三）、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()

4、示例代码

```
//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()
}
```

（四）、Scanner

1、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
```

2、常用方法

- func (s *Scanner) Scan() bool
- func (s *Scanner) Text() string
- func (s *Scanner) Text() []byte

3、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)

4、示例代码：

```go
//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
    }
  }
}
```