Go语言 标准库 os包

os 包提供了平台无关的操作系统功能接口，主要是文件相关的`I/O`，目录，环境，进程等。

权限

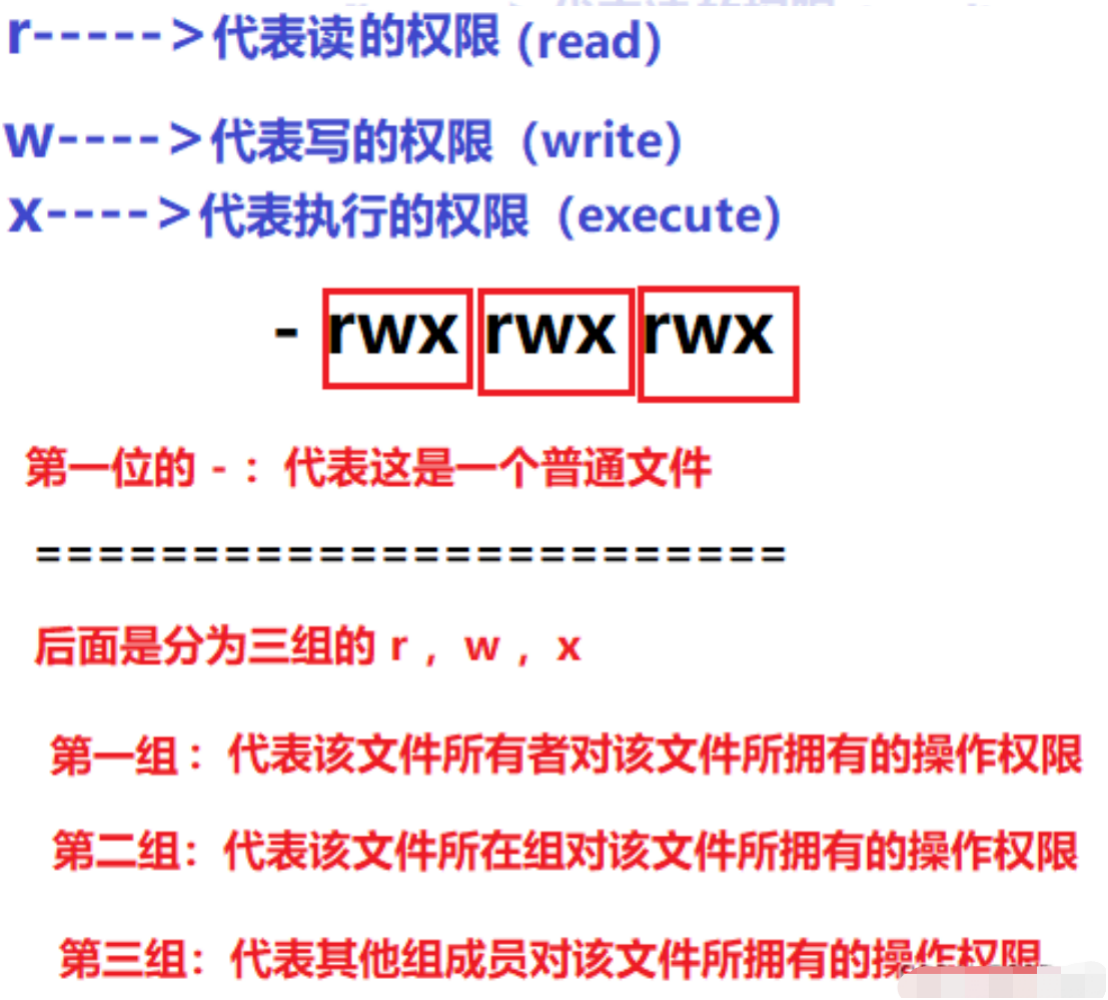
权限**perm**，在创建文件时才需要指定，不需要创建新文件时可以将其设定为 0 。虽然go语言给perm权限设定了很多的常量，但是习惯上也可以直接使用数字，如0666(具体含义和Unix系统的一致)。

权限项	文件类型	读	写	执行	读	写	执行	读	写	执行
字符表示	(d|l|c|s|p)	r	w	x	r	w	x	r	w	x
数字表示		4	2	1	4	2	1	4	2	1
权限分配		文件所有者	文件所有者	文件所有者	文件所属组用户	文件所属组用户	文件所属组用户	其他用户	其他用户	其他用户

关于`- rwx rwx rwx`:

`-`代表这是一个普通文件(regular), 其中其他文件类型还包括了:
d--------->目录文件(directory);
l --------->链接文件(link);
b--------->块设备文件(block);
c--------->字符设备文件(character);
s--------->套接字文件(socket);
p--------->管道文件(pipe);

一共有三组rwx，分别是以下几种用户的rwx权限:
第一组:该文件拥有者的权限。
第二组:该文件拥有者所在组的其他成员对该文件的操作权限。
第三组:其他用户组的成员对该文件的操作权限。



注意在这里计算权限的值是使用**八进制**进行计算的；例如上图中的`-rwxrwxrwx`(前面的-代表这是一个普通文件)文件权限，我们计算过程如下：

rwx	rwx	rwx
111	111	111
7	7	7

此时该文件的权限值为0777，当然还有0666、0755等都是这么得来的。

目录相关

Create函数

创建文件。

```
func Create(name string) (file *File, err error)
```

示例：

```
// 创建文件
func createFile() {
    f, err := os.Create("test.txt")
    if err != nil {
        fmt.Printf("err: %v\n", err)
    } else {
        fmt.Printf("f: %v\n", f)
    }
}
```

Mkdir函数

创建单个目录。只能创建单个目录，不能创建多级目录。

```
func Mkdir(name string, perm FileMode) error {}
```

示例：

```
func createDir1() {
    err := os.Mkdir("test", os.ModePerm)
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
}
```

MkdirAll函数

创建多级目录。

```
func MkdirAll(path string, perm FileMode) error {}
```

示例：

```
func createDir2() {
    err := os.MkdirAll("test/a/b", os.ModePerm)
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
}
```

Remove函数

只能删除一个空的目录或一个文件。

```
func Remove(name string) error {}
```

示例：

```
func removeDir1() {
    err := os.Remove("test.txt")
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
}
```

RemoveAll函数

可以强制删除目录以及目录汇中的文件。

```
func RemoveAll(path string) error {
    return removeAll(path)
}
```

示例：

```
func removeDir2() {
    err := os.RemoveAll("test")
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
}
```

Getwd函数

获得工作目录。

```
func Getwd() (dir string, err error) {}
```

示例：

```
func getWd() {
    dir, err := os.Getwd()
    if err != nil {
        fmt.Printf("err: %v\n", err)
    } else {
        fmt.Printf("dir: %v\n", dir)
    }
}
```

Chdir函数

修改工作目录。改变工作目录到f，其中f必须为一个目录，否则便会报错。

```
func (f *File) Chdir() error
```

示例：

```
func chWd() {
    err := os.Chdir("d:/")
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
    fmt.Println(os.Getwd())
}
```

TempDir

获得临时目录。

```
func TempDir() string {
    return tempDir()
}
```

示例：

```
func getTemp() {
    s := os.TempDir()
    fmt.Printf("s: %v\n", s)
}
```

Rename

重命名文件。

```
func Rename(oldpath, newpath string) error {
    return rename(oldpath, newpath)
}
```

示例：

```
func renameFile() {
    err := os.Rename("test.txt", "test2.txt")
    if err != nil {
        fmt.Printf("err: %v\n", err)
    }
}
```

IsExist

检查文件是否存在，返回一个布尔值说明该错误是否表示一个文件或目录已经存在。ErrExist 和一些系统调用错误会使它返回真。

```
func IsExist(err error) bool
```

示例：

```
func IsExist() {
    // 文件不存在则返回 error
    _, err := os.Open("test1.txt")
    if os.IsExist(err) {
        fmt.Println("File is exist.")
    }
}
```

IsNotExist

检查文件是否不存在，返回一个布尔值说明该错误是否表示一个文件或目录不存在。ErrNotExist 和一些系统调用错误会使它返回真。

```
func IsNotExist(err error) bool
```

示例：

```
func IsNotExist() {
    // 文件不存在则返回 error
    _, err := os.Open("test1.txt")
    if os.IsNotExist(err) {
        fmt.Println("File does not exist.")
    }
}
```

Link/Symlink

创建硬链接和软链接。

硬链接Link：文件是通过索引节点（Inode）来识别文件，硬链接可以认为是一个指针，指向文件索引节点的指针，每添加一个一个硬链接，文件的链接数就加 1，只有所有的硬链接被删除后文件才会被删除。只有在同一文件系统中的文件之间才能创建硬链接，不能对目录进行创建，但是这个硬链接又可以建立多个，也就是可以有多个文件指向同一个索引节点，或者说一个文件可以拥有多个路径名，因此一个文件可以对应多个文件名。

软链接Symlink：和硬链接有点不一样，它不直接指向索引节点，而是通过名字引用其它文件，类似 Windows 的快捷方式。软链接可以指向不同的文件系统中的不同文件，但是并不是所有的操作系统都支持软链接

创建一个名为 newname 指向 oldname 的硬链接。如果出错，会返回 *LinkError 底层类型的错误。

```
func Link(oldname, newname string) error
```

创建一个名为 newname 指向 oldname 的软链接。如果出错，会返回 *LinkError 底层类型的错误。

```
func Symlink(oldname, newname string) error
```

示例：

```
package main

import (
```

```
    "fmt"
    "os"
)

func main() {
    // 创建一个硬链接
    // 创建后同一个文件内容会有两个文件名，改变一个文件的内容会影响另一个
    // 删除和重命名不会影响另一个
    err := os.Link("./test1.txt", "./test1_also.txt")
    if err != nil {
        panic(err)
    }

    // 创建软链接（Windows不支持软链接）
    err = os.Symlink("./test1.txt", "./test1_sym.txt")
    if err != nil {
        panic(err)
    }

    // Lstat 返回一个文件的信息，但是当文件是一个软链接时，它返回软链接的信息，而不是引用的文件的信息
    fileInfo, err := os.Lstat("./test1_sym.txt")
    if err != nil {
        panic(err)
    }
    fmt.Printf("Link info: %+v", fileInfo)

    // 改变软链接的拥有者不会影响原始文件
    err = os.Lchown("./test1_sym.txt", os.Getuid(), os.Getgid())
    if err != nil {
        panic(err)
    }
}
```

文件相关

文件打开模式

```
const (
    O_RDONLY int = syscall.O_RDONLY // 只读模式打开文件
    O_WRONLY int = syscall.O_WRONLY // 只写模式打开文件
    O_RDWR   int = syscall.O_RDWR   // 读写模式打开文件
    O_APPEND int = syscall.O_APPEND // 写操作时将数据附加到文件尾部
    O_CREATE int = syscall.O_CREAT  // 如果不存在将创建一个新文件
    O_EXCL   int = syscall.O_EXCL   // 和O_CREATE配合使用，文件必须不存在
    O_SYNC   int = syscall.O_SYNC   // 打开文件用于同步I/O
    O_TRUNC  int = syscall.O_TRUNC  // 如果可能，打开时清空文件
)
```

多种访问模式可以使用`|`操作符来连接，例如：`O_RDWR|O_CREATE|O_TRUNC`。其中，`O_RDONLY`、`O_WRONLY`、`O_RDWR`三种模式只能指定一个。

文件的打开与关闭操作

Create函数

创建一个空文件，注意当文件已经存在时，会直接覆盖掉原文件，不会报错。`Create()`函数本质上是调用`OpenFile()`函数 等价于：`OpenFile(name, O_RDWF|O_CREATE|O_TRUNK, 0666)`。

采用 0666 权限（任何人都可读写，不可执行）创建一个名为 name 的文件，如果文件已存在会截断它（为空文件）。

如果成功，返回的文件对象可用于 I/O；对应的文件描述符具有 O_RDWR 模式。如果出错，错误底层类型是 *PathError

```
func Create(name string) (file *File, err error)
```

示例：

```
// 创建文件
func createFile() {
     // 以 O_RDWR|O_CREATE|O_TRUNC 模式，0666 权限打开文件
    newFile, err := os.Create("test.txt")
    if err != nil {
        panic(err)
    }
    defer newFile.Close()
}
```

Open 函数

打开一个文件,注意打开的文件只能读，不能写。

```
func Open(name string) (file *File, err error)
```

示例：

```
func openFile1() {
     // 以只读的方式打开文件
    file1, err := os.Open("test1.txt")
    if err != nil {
        panic(err)
    }
    defer file1.Close()
}
```

OpenFile函数

指定的权限打开文件。

`OpenFile()`是一个一般性的文件打开函数，大多数调用都应该使用`Open()`或`Create()`代替本函数。它会使用指定的选项（如`O_RDONLY`等）、指定的模式（如 0666 等）打开指定名称的文件。如果操作成功，返回的文件对象可用于 I/O。如果出错，错误底层类型是`*PathError`。

```
func OpenFile(name string, flag int, perm FileMode) (file *File, err error)
```

此处需要特别介绍`openFile()`函数参数：

- `name`：要打开的文件名，可以是一个绝对路径或相对路径，也可以是一个符号链接。
- `flag`：指定文件的访问模式，可用值已在 os 包中定义为常量。及上文文件打开模式。多种访问模式可以使用 | 操作符来连接，例如：`O_RDWR|O_CREATE|O_TRUNC`。其中，`O_RDONLY`、`O_WRONLY`、`O_RDWR`三种模式只能指定一个。
- `perm`：指定了文件的模式和权限位，类型是`os.FileMode`。

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。

```
type FileMode uint32

const (
    // 单字符是被 String 方法用于格式化的属性缩写。
    ModeDir        FileMode = 1 << (32 - 1 - iota) // d: 目录
    ModeAppend                                     // a: 只能写入，且只能写入到末尾
    ModeExclusive                                  // l: 用于执行
    ModeTemporary                                  // T: 临时文件（非备份文件）
    ModeSymlink                                    // L: 符号链接（不是快捷方式文件）
    ModeDevice                                     // D: 设备
    ModeNamedPipe                                  // p: 命名管道（FIFO）
    ModeSocket                                     // S: Unix 域 socket
    ModeSetuid                                     // u: 表示文件具有其创建者用户 id
权限
    ModeSetgid                                     // g: 表示文件具有其创建者组 id 的
权限
    ModeCharDevice                                 // c: 字符设备，需已设置
ModeDevice
    ModeSticky                                     // t: 只有 root/ 创建者能删除 /
移动文件

    // 覆盖所有类型位（用于通过 & 获取类型位），对普通文件，所有这些位都不应被设置
    ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket | ModeDevice
    ModePerm FileMode = 0777 // 覆盖所有 Unix 权限位（用于通过 & 获取类型位）
)
```

文件的权限打印出来一共十个字符。文件有三种基本权限：r（read，读权限）、w（write，写权限）、x（execute，执行权限），具体如下：

```
 - rwx rw- r--
```

- 第 1 位：文件类型（d为目录，-为普通文件）
- 第 2-4 位：所属用户权限，用 u（user）表示。
- 第 5-7 位：所属组权限，用 g（group）表示。
- 第 8-10 位：其他用户权限，用 o（other）表示。

文件的权限还可以用八进制表示：r 表示为 4，w 表示为 2，x 表示为 1，- 表示为 0。例如：

- `0777`：权限`- rwx rwx rwx`的八进制表示，任何人都可读写，可执行。
- `0666`：权限`- rw- rw- rw-`的八进制表示，任何人都可读写，但不可执行。

示例：

```
func openFile2() {
    // 自定义方式打开文件
    file2, err := os.OpenFile("test2.txt", os.O_RDWR|os.O_CREATE|os.O_TRUNC,
0666)
    if err != nil {
        panic(err)
    }
    defer file2.Close()
}
```

Close函数

关闭文件，关闭后不可读写。用于关闭一个打开的文件描述符，并将其释放回调用进程，供该进程继续使用。当进程终止时，也会自动关闭其已打开的所有文件描述符。通常情况下，我们应该主动关闭文件描述符，如果不关闭，长期运行的服务可能会把文件描述符耗尽。关于返回值 error，以下两种情况会导致`Close()`返回错误：

- 关闭一个未打开的文件
- 两次关闭同一个文件

因此，通常我们不会去检查`Close()`返回的错误。

文件读操作

Read

从 f 中读取最多 len(b) 字节数据并写入 b。它返回读取的字节数和可能遇到的任何错误,文件终止标志是读取 0 个字节，且返回值 err 为 io.EOF。

```
func (f *File) Read(b []byte) (n int, err error)
```

示例：

```
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    f, _ := os.Open("a.txt")
    for {
        buf := make([]byte, 4) // 设置一个缓冲区，一次读4个字节
        n, err := f.Read(buf)  // 将读到的内容放入缓冲区内
        fmt.Printf("string(buf): %v\n", string(buf))
        fmt.Printf("n: %v\n", n)
        if err == io.EOF { // EOF表示文件读取完毕
            break
        }
    }
    f.Close()
}
```

ReadAt

从指定的位置（相对于文件开始位置）读取 len(b) 字节数据并写入 b。它返回读取的字节数和可能遇到的任何错误。当 n < len(b) 时，本方法总是会返回错误；如果是因为到达文件结尾，返回值 err 会是 io.EOF。

`Read()` 和 `ReadAt()` 的区别：前者从文件当前偏移量处读，且会改变文件当前的偏移量；而后者从 off 指定的位置开始读，且不会改变文件当前偏移量。

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    buf := make([]byte, 10)
    f2, _ := os.Open("a.txt")
    // 从5开始读10个字节
    n, _ := f2.ReadAt(buf, 5)
    fmt.Printf("n: %v\n", n)
    fmt.Printf("string(buf): %v\n", string(buf))
    f2.Close()
}
```

ReadDir

读取目录并返回排好序的文件以及子目录名切片。

```
func ReadDir(name string) ([]DirEntry, error)
```

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 测试 a目录下有b和c目录
    f, _ := os.Open("a")
    de, _ := f.ReadDir(-1) // 返回一个切片de
    for _, v := range de {
        fmt.Printf("v.IsDir(): %v\n", v.IsDir())
```

```
        fmt.Printf("v.Name(): %v\n", v.Name())
    }
}
```

Seek

Seek设置下一次读/写的位置。offset为相对偏移量，而whence决定相对位置：0为相对文件开头，1为相对当前位置，2为相对文件结尾。它返回新的偏移量（相对开头）和可能的错误。

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

whence 的值，在 io 包中定义了相应的常量，推荐使用这些常量

```
const (
  SeekStart   = 0 // seek relative to the origin of the file
  SeekCurrent = 1 // seek relative to the current offset
  SeekEnd     = 2 // seek relative to the end
)
```

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, _ := os.Open("a.txt") // 打开文件后，光标默认在文件开头
    f.Seek(3, 0) // 从索引值为3处开始读
    buf := make([]byte, 10) // 设置缓冲区
    n, _ := f.Read(buf) // 将内容读到缓冲区内
    fmt.Printf("n: %v\n", n)
    fmt.Printf("string(buf): %v\n", string(buf))
    f.Close()
}
```

文件写操作

Write

Write向文件中写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值n!=len(b)，本方法会返回一个非nil的错误。

```
func (f *File) Write(b []byte) (n int, err error)
```

示例：

```
package main

import (
    "os"
)

func main() {
    f, _ := os.OpenFile("a.txt", os.O_RDWR|os.O_APPEND, 0775) // 以读写模式打开文件，并且在写操作时将数据附加到文件尾部
    f.Write([]byte(" hello golang"))
    f.Close()
}
```

WriteString

WriteString类似Write，但接受一个字符串参数。

```
func (f *File) WriteString(s string) (ret int, err error)
```

示例：

```
package main

import (
    "os"
)

func main() {
    f, _ := os.OpenFile("a.txt", os.O_RDWR|os.O_TRUNC, 0775) // 以读写模式打开文件，并且打开时清空文件
    f.WriteString("hello world...")
    f.Close()
}
```

WriteAt

WriteAt在指定的位置（相对于文件开始位置）写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值n!=len(b)，本方法会返回一个非nil的错误。

`Write()`与`WriteAt()`的区别：前者从文件当前偏移量处写，且会改变文件当前的偏移量；而后者从 off 指定的位置开始写，且不会改变文件当前偏移量。

写操作调用成功并不能保证数据已经写入磁盘，因为内核会缓存磁盘的`I/O`操作。如果希望立刻将数据写入磁盘（一般场景不建议这么做，因为会影响性能），有两种办法：

- 打开文件时指定`os.O_SYNC`模式；
- 调用`File.Sync()`方法，该方法底层是 fsync 系统调用，这会将数据和元数据都刷到磁盘。

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

示例：

```
package main

import (
    "os"
)

func main() {
    f, _ := os.OpenFile("a.txt", os.O_RDWR, 0775) // 以读写模式打开文件
    f.WriteAt([]byte("aaa"), 3) // 从索引值为3的地方开始写入aaa并覆盖原来当前位置的数据
    f.Close()
}
```

文件属性

FileInfo

文件属性，也即文件元数据。在 Go 中，文件属性具体信息通过`os.FileInfo`接口获取，文件的信息包括文件名、文件大小、修改权限、修改时间等。

```
type FileInfo interface {
    Name() string       // 文件的名字（不含扩展名）
    Size() int64        // 普通文件返回值表示其大小；其他文件的返回值含义各系统不同
    Mode() FileMode     // 文件的权限
    ModTime() time.Time // 文件的修改时间
    IsDir() bool        // 等价于 Mode().IsDir()
    Sys() interface{}   // 底层数据来源（可以返回 nil）
}
```

`os.FileMode`类型表示文件的模式和权限位，在介绍`OpenFile()`函数的时候提及过，其方法如下：

```
type FileMode uint32

// 报告 m 是否是一个目录
func (m FileMode) IsDir() bool

// 报告 m 是否是一个普通文件
func (m FileMode) IsRegular() bool

// 返回 m 的 Unix 权限位
func (m FileMode) Perm() FileMode

// 打印出文件权限
func (m FileMode) String() string

// 返回文件类型
func (m FileMode) Type() FileMode {
```

Stat

获取文件的信息，里面有文件的名称，大小，修改时间等。

返回一个描述 name 指定的文件对象的 FileInfo。

如果指定的文件对象是一个符号链接，返回的 FileInfo 描述该符号链接指向的文件的信息，本函数会尝试跳转该链接。如果出错，返回的错误值为 *PathError 类型

```
func (f *File) Stat() (fi FileInfo, err error)
```

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, _ := os.Open("a.txt") // 打开文件后，光标默认在文件开头
    fInfo, _ := f.Stat()
    fmt.Printf("f是否是一个文件: %v\n", fInfo.IsDir())
    fmt.Printf("f文件的修改时间: %v\n", fInfo.ModTime().String())
    fmt.Printf("f文件的名称: %v\n", fInfo.Name())
    fmt.Printf("f文件的大小: %v\n", fInfo.Size())
    fmt.Printf("f文件的权限: %v\n", fInfo.Mode().String())
}
```

Lstat

返回一个描述 name 指定的文件对象的 FileInfo 。

如果指定的文件对象是一个符号链接，返回的 FileInfo 描述该符号链接的信息，本函数不会试图跳转该链接。如果出错，返回的错误值为 *PathError 类型。

`Stat()` 和 `Lstat()` 无需对其所操作的文件本身拥有任何权限，但针对指定 name 的父目录要有执行（搜索）权限。而只要 File 对象 ok，`File.Stat()` 总是成功。

```
func Lstat(name string) (fi FileInfo, err error)
```

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fileInfo2, err := os.Lstat("test.txt")
    if err != nil {
        panic(err)
    }
    fmt.Println("文件名:", fileInfo2.Name())         // test.txt
    fmt.Println("文件大小:", fileInfo2.Size())        // 11
    fmt.Println("文件权限:", fileInfo2.Mode())        // -rw-rw-rw-
```

```
    fmt.Println("文件最后修改时间:", fileInfo2.ModTime()) // 2022-12-21
19:11:15.686021 +0800 CST
    fmt.Println("是否为目录:", fileInfo2.IsDir())      // false
}
```

函数`Stat()`、`Lstat()`可以得到os.FileInfo`接口的实例，分别对应三个系统调用：stat、lstat 。它们区别在于：

- `Stat()`会返回所命名文件的相关信息；
- `Lstat()`与`Stat()`类似，区别在于如果文件是符号链接，那么所返回的信息针对的是符号链接自身（而非符号链接所指向的文件）；

Chmod函数

修改文件权限。修改 name 指定的文件对象的 mode。如果 name 指定的文件是一个符号链接，它会修改该链接的目的地文件的 mode。如果出错，会返回 *PathError 底层类型的错误。

```
func Chmod(name string, mode FileMode) error {
    return chmod(name, mode)
}
```

示例：

```
func ChmodFile() {
    err := os.Chmod("test.txt", 0777)
    if err != nil {
        panic(err)
    }
}
```

Chown函数

修改文件所有者。修改 name 指定的文件对象的用户 id 和组 id，如果 name 指定的文件是一个符号链接，它会修改该链接的目的地文件的用户 id 和组 id。如果出错，会返回 *PathError 底层类型的错误。

```
func Chown(name string, uid, gid int) error {}
```

示例：

```
func ChownFile() {
    err := os.Chown("test.txt", os.Getuid(), os.Getgid())
    if err != nil {
        panic(err)
    }
}
```

Chtimes

改变时间戳。修改 name 指定的文件对象的访问时间和修改时间，类似 Unix 的 utime() 或 utimes() 函数，底层的文件系统可能会截断/舍入时间单位到更低的精确度。如果出错，会返回 *PathError 底层类型的错误。

```
func Chtimes(name string, atime time.Time, mtime time.Time) error
```

示例：

```
func ChtimesFile() {
    fileInfo, err := os.Stat("test.txt")
    if err != nil {
        panic(err)
    }
    fmt.Println("最后修改时间：", fileInfo.ModTime())

    // 改变文件时间戳为两天前
    twoDaysFromNow := time.Now().Add(48 * time.Hour)
    lastAccessTime := twoDaysFromNow
    lastModifyTime := twoDaysFromNow
    err = os.Chtimes("./test.txt", lastAccessTime, lastModifyTime)
    if err != nil {
        panic(err)
    }
    fileInfo, err = os.Stat("./test.txt")
    if err != nil {
        panic(err)
    }
    fmt.Println("最后修改时间：", fileInfo.ModTime())
}
```

IsPermission

检查读写权限。返回一个布尔值说明该错误是否表示因权限不足要求被拒绝。ErrPermission 和一些系统调用错误会使它返回真。

```
func IsPermission(err error) bool
```

示例：

```
func IsPermissionFile() {
    // 这个例子测试写权限，如果没有写权限则返回 error
    // 注意文件不存在也会返回 error，需要检查 error 的信息来获取到底是哪个错误导致
    file1, err := os.OpenFile("test.txt", os.O_RDONLY, 0666)
    if err != nil {
        panic(err)
    }
    defer file1.Close()
    _, err = file1.WriteString("hello world")
    if os.IsPermission(err) {
        fmt.Println(err) // write ./test.txt: Access is denied.
    }

    // 测试读权限
    file2, err := os.OpenFile("test.txt", os.O_WRONLY, 0666)
    if err != nil {
        panic(err)
```

```
    }
    defer file2.Close()
    _, err = file2.Read(make([]byte, 10))
    if os.IsPermission(err) {
        fmt.Println(err) // read ./test.txt: Access is denied.
    }
}
```

进程相关操作

Exit

让当前程序以给出的状态码（code）退出。一般来说，状态码0表示成功，非0表示出错。程序会立刻终止，defer的函数不会被执行。

```
func Exit(code int)
```

Getuid

获取调用者的用户id。

```
func Getuid() int
```

Geteuid

获取调用者的有效用户id。

```
func Geteuid() int
```

Getgid

获取调用者的组id。

```
func Getgid() int
```

Getegid

获取调用者的有效组id。

```
func Getegid() int
```

Getgroups

获取调用者所在的所有组的组id。

```
func Getgroups() ([]int, error)
```

Getpid

获取调用者所在进程的进程id。

```
func Getpid() int
```

Getppid

获取调用者所在进程的父进程的进程id。

```
func Getppid() int
```

示例

```
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {
    // 获得当前正在运行的进程id
    fmt.Printf("os.Getpid(): %v\n", os.Getpid())
    // 父id
    fmt.Printf("os.Getppid(): %v\n", os.Getppid())

    // 设置新进程的属性
    attr := &os.ProcAttr{
        // files指定新进程继承的活动文件对象
        // 前三个分别为，标准输入、标准输出、标准错误输出
        Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
        // 新进程的环境变量
        Env: os.Environ(),
    }

    // 开始一个新进程
    p, err := os.StartProcess("c:\\windows\\system32\\notepad.exe",
[]string{"c:\\windows\\system32\\notepad.exe", "d:\\a.txt"}, attr)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(p)
    fmt.Println("进程ID：", p.Pid)

    // 通过进程ID查找进程
    p2, _ := os.FindProcess(p.Pid)
    fmt.Println(p2)

    // 等待10秒，执行函数
    time.AfterFunc(time.Second*10, func() {
        // 向p进程发出退出信号
```

```
        p.Signal(os.Kill)
    })

    // 等待进程p的退出，返回进程状态
    ps, _ := p.Wait()
    fmt.Println(ps.String())
}
```

环境相关操作

Hostname

获取主机名。

```
func Hostname() (name string, err error)
```

Getenv

获取某个环境变量。

```
func Getenv(key string) string
```

Setenv

设置一个环境变量,失败返回错误，经测试当前设置的环境变量只在 当前进程有效（当前进程衍生的所以的go程都可以拿到，子go程与父go程的环境变量可以互相获取）；进程退出消失。

```
func Setenv(key, value string) error
```

Clearenv

删除当前程序已有的所有环境变量。不会影响当前电脑系统的环境变量，这些环境变量都是对当前go程序而言的。

```
func Clearenv()
```

Chdir

改变当前工作目录。

```
func Chdir(dir string) error
```

Environ

Environ返回表示环境变量的格式为”key=value”的字符串的切片拷贝。

```
func Environ() []string
```

Args

Args保管了命令行参数，第一个是程序名。

```
func Args()
```

示例

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 获得所有环境变量
    s := os.Environ()
    fmt.Printf("s: %v\n", s)
    // 获得某个环境变量
    s2 := os.Getenv("GOPATH")
    fmt.Printf("s2: %v\n", s2)
    // 设置环境变量
    os.Setenv("env1", "env1")
    s2 = os.Getenv("aaa")
    fmt.Printf("s2: %v\n", s2)
    fmt.Println("--------------")

    // 查找
    s3, b := os.LookupEnv("env")
    fmt.Printf("b: %v\n", b)
    fmt.Printf("s3: %v\n", s3)

    // 替换
    os.Setenv("NAME", "gopher")
    os.Setenv("BURROW", "/usr/gopher")

    os.ExpandEnv("$NAME lives in ${BURROW}.")

    // 清空环境变量
    // os.Clearenv()
}
```