os包用法

os包中实现了平台无关的接口，设计向Unix风格，但是错误处理是go风格，当os包使用时，如果失败之后返回错误类型而不是错误数量．

os包中函数设计方式和Unix类似，下面来看一下．

func Chdir(dir string) error   //chdir将当前工作目录更改为dir目录．

func Getwd() (dir string, err error)    //获取当前目录，类似linux中的pwd

func Chmod(name string, mode FileMode) error     //更改文件的权限（读写执行，分为三类：all-group-owner）  
func Chown(name string, uid, gid int) error  //更改文件拥有者owner  
func Chtimes(name string, atime time.Time, mtime time.Time) error    //更改文件的访问时间和修改时间，atime表示访问时间，mtime表示更改时间  
func Clearenv()    //清除所有环境变量（慎用）  
func Environ() []string  //返回所有环境变量  
func Exit(code int)     //系统退出，并返回code，其中０表示执行成功并退出，非０表示错误并退出，其中执行Exit后程序会直接退出，defer函数不会执行．

func Expand(s string, mapping func(string) string) string   //Expand用mapping 函数指定的规则替换字符串中的${var}或者$var（注：变量之前必须有$符号）。比如，os.ExpandEnv(s)等效于os.Expand(s, os.Getenv)。

package main

import (

"fmt"

"os"

)

func main() {

mapping := func(key string) string {

m := make(map[string]string)

m = map[string]string{

"world": "kitty",

"hello": "hi",

}

if m[key] != "" {

return m[key]

}

return key

}

s := "hello,world" // hello,world，由于hello world之前没有$符号，则无法利用map规则进行转换

s1 := "$hello,$world $finish" // hi,kitty finish，finish没有在map规则中，所以还是返回原来的值

fmt.Println(os.Expand(s, mapping))

fmt.Println(os.Expand(s1, mapping))

}

func ExpandEnv(s string) string  //ExpandEnv根据当前环境变量的值来替换字符串中的${var}或者$var。如果引用变量没有定义，则用空字符串替换。

func main() {

s := "hello $GOROOT"

fmt.Println(os.ExpandEnv(s)) // hello /home/work/software/go，$GOROOT替换为环境变量的值，而h没有环境变量可以替换，返回空字符串

}

func Getenv(key string) string  //获取系统key的环境变量，如果没有环境变量就返回空

  fmt.Println(os.Getenv("GOROOT")) // /home/software/go

func Geteuid() int  //获取调用者用户id

func Getgid() int   //获取调用者的组id

fmt.Println(os.Getgid()) // 1000

func Getgroups() ([]int, error)  //返回调用者属于的group，其和chown配合使用，改变文件属于的group．

func main() {

fmt.Println(os.Getgroups())　　　　　//获取调用者属于的组 [4 24 27 30 46 108 124 1000]

fmt.Println(os.Getgid())　　　　　　　//获取调用者当前所在的组　1000

fmt.Println(os.Chown("tmp.txt", 1000, 46)) //更改文件所在的组

}

func Getpagesize() int　　　//获取底层系统内存页的数量  
func Getpid() int　　　　//获取进程id  
func Getppid() int             //获取调用者进程父id  
  
func Hostname() (name string, err error)    //获取主机名  
func IsExist(err error) bool    　　　　　 //返回一个布尔值，它指明err错误是否报告了一个文件或者目录已经存在。它被ErrExist和其它系统调用满足。  
  
func IsNotExist(err error) bool　　　　　//返回一个布尔值，它指明err错误是否报告了一个文件或者目录不存在。它被ErrNotExist 和其它系统调用满足。  
func IsPathSeparator(c uint8) bool         //判断c是否是一个路径分割符号，是的话返回true,否则返回false

fmt.Println(os.IsPathSeparator('/')) //true

fmt.Println(os.IsPathSeparator('|')) //false

func IsPermission(err error) bool   //判定err错误是否是权限错误。它被ErrPermission 和其它系统调用满足。  
func Lchown(name string, uid, gid int) error　　　//改变了文件的gid和uid。如果文件是一个符号链接，它改变的链接自己。如果出错，则会是\*PathError类型。  
func Link(oldname, newname string) error       //创建一个从oldname指向newname的硬连接，对一个进行操作，则另外一个也会被修改．  
func Mkdir(name string, perm FileMode) error　//创建一个新目录，该目录具有FileMode权限，当创建一个已经存在的目录时会报错

func main() {

var path string

if os.IsPathSeparator('\\') {

path = "\\"

} else {

path = "/"

}

pwd, \_ := os.Getwd()

err := os.Mkdir(pwd+path+"tmp", os.ModePerm)

if err != nil {

fmt.Println(err)

}

}

func MkdirAll(path string, perm FileMode) error　//创建一个新目录，该目录是利用路径（包括绝对路径和相对路径）进行创建的，如果需要创建对应的父目录，也一起进行创建，如果已经有了该目录，则不进行新的创建，当创建一个已经存在的目录时，不会报错.  
func NewSyscallError(syscall string, err error) error    //NewSyscallError返回一个SyscallError 错误，带有给出的系统调用名字和详细的错误信息。也就是说，如果err为空，则返回空

func main() {

a := os.NewSyscallError("mkdir", nil)

fmt.Println(a) // nil

a = os.NewSyscallError("mkdir", errors.New("bad error"))

fmt.Println(a) //mkdir: bad error

}

func Readlink(name string) (string, error)         //返回符号链接的目标。如果出错，将会是 \*PathError类型。  
func Remove(name string) error           //删除文件或者目录  
func RemoveAll(path string) error　　//删除目录以及其子目录和文件，如果path不存在的话，返回nil

func main() {

err := os.MkdirAll("./a", os.ModePerm)

os.Chdir("./a")

os.Create("file.txt")

fmt.Println(err)//成功创建文件file.txt，返回nil

os.Chdir("../")

err = os.RemoveAll("./a")

fmt.Println(err)//成功删除目录a，返回nil

}

func Rename(oldpath, newpath string) error　　//重命名文件，如果oldpath不存在，则报错no such file or directory  
func SameFile(fi1, fi2 FileInfo) bool　　　　　　//查看f1和f2这两个是否是同一个文件，如果再Unix系统，这意味着底层结构的device和inode完全一致，在其他系统上可能是基于文件绝对路径的．SameFile只适用于本文件包stat返回的状态，其他情况下都返回false  
func Setenv(key, value string) error           //设定环境变量，经常与Getenv连用，用来设定环境变量的值

func main() {

err := os.Setenv("goenv", "go environment")

a := os.Getenv("goenv")

fmt.Println(err, a) // <nil> go environment

}

func Symlink(oldname, newname string) error　　　//创建一个newname作为oldname的符号连接，这是一个符号连接（Link是硬连接），与link的硬连接不同，利用Link创建的硬连接，则newname和oldname的file互不影响，一个文件删除，另外一个文件不受影响；但是利用SymLink创建的符号连接，其newname只是一个指向oldname文件的符号连接，当oldname　file删除之后，则newname的文件也就不能够继续使用．  
func TempDir() string　　　　　　　　//创建临时目录用来存放临时文件，这个临时目录一般为/tmp  
func Truncate(name string, size int64) error     //按照指定长度size将文件截断，如果这个文件是一个符号链接，则同时也改变其目标连接的长度，如果有错误，则返回一个错误．

文件操作：

type File

type File struct {

// contains filtered or unexported fields

}

File表示打开的文件描述符  
func Create(name string) (file \*File, err error)　　//创建一个文件，文件mode是0666(读写权限)，如果文件已经存在，则重新创建一个，原文件被覆盖，创建的新文件具有读写权限，能够备用与i/o操作．其相当于OpenFile的快捷操作，等同于OpenFile(name string, O\_CREATE,0666)  
func NewFile(fd uintptr, name string) \*File　　　//根据文件描述符和名字创建一个新的文件

Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")

Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")

Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")

func Open(name string) (file \*File, err error)　　　//打开一个文件，返回文件描述符，该文件描述符只有只读权限．他相当于OpenFile(name string,O\_RDWR,0)

func OpenFile(name string, flag int, perm FileMode) (file \*File, err error)　//指定文件权限和打开方式打开name文件或者create文件，其中flag标志如下:

**打开标记：**

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.)  
O\_EXCL：与 O\_CREATE 一起用，构成一个新建文件的功能，它要求文件必须不存在(used with O\_CREATE, file must not exist)  
O\_SYNC：同步方式打开，即不使用缓存，直接写入硬盘  
O\_TRUNC：打开并清空文件

至于操作权限perm，除非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).

func Pipe() (r \*File, w \*File, err error)        //返回一对连接的文件，从r中读取写入w中的数据，即首先向w中写入数据，此时从r中变能够读取到写入w中的数据，Pipe()函数返回文件和该过程中产生的错误．

func main() {

r, w, \_ := os.Pipe()

w.WriteString("hello,world!")

var s = make([]byte, 20)

n, \_ := r.Read(s)

fmt.Println(string(s[:n])) // hello,world!

}

func (f \*File) Chdir() error　　　　　　　//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 main() {

dir, \_ := os.Getwd()

fmt.Println(dir)

f, \_ := os.Open("tmp.txt")

err := f.Chdir()

if err != nil {

fmt.Println(err)　　　//chdir tmp.txt: not a directory，因为tmp.txt不是目录，所以报错

}

f, \_ = os.Open("develop")

err = f.Chdir()

if err != nil {

fmt.Println(err)

}

dir1, \_ := os.Getwd()

fmt.Println(dir1)　　　 //home/work/develop，因为develop是工作目录，所以变切换到该目录

}

func (f \*File) Chmod(mode FileMode) error　　　//更改文件权限，其等价与os.Chmod(name string,mode FileMode)error  
func (f \*File) Chown(uid, gid int) error                     //更改文件所有者，与os.Chown等价  
func (f \*File) Close() error　　　　　　　　　　//关闭文件，使其不能够再进行i/o操作，其经常和defer一起使用，用在创建或者打开某个文件之后，这样在程序退出前变能够自己关闭响应的已经打开的文件．  
func (f \*File) Fd() uintptr　　　//返回系统文件描述符，也叫做文件句柄．  
func (f \*File) Name() string　　//返回文件名字，与file.Stat().Name()等价  
func (f \*File) Read(b []byte) (n int, err error)　//将len(b)的数据从f中读取到b中，如果无错，则返回n和nil,否则返回读取的字节数n以及响应的错误  
func (f \*File) ReadAt(b []byte, off int64) (n int, err error)　//和Read类似，不过ReadAt指定开始读取的位置offset  
func (f \*File) Readdir(n int) (fi []FileInfo, err error)           

Readdir读取file指定的目录的内容，然后返回一个切片，它最多包含n个FileInfo值，这些值可能是按照目录顺序的Lstat返回的。接下来调用相同的文件会产生更多的FileInfos。

如果n>0，Readdir返回最多n个FileInfo结构。在这种情况下，如果Readdir返回一个空的切片，它将会返回一个非空的错误来解释原因。在目录的结尾，错误将会是io.EOF。

如果n<=0，Readdir返回目录的所有的FileInfo，用一个切片表示。在这种情况下，如果Readdir成功（读取直到目录的结尾），它会返回切片和一个空的错误。如果它在目录的结尾前遇到了一个错误，Readdir返回直到当前所读到的FIleInfo和一个非空的错误。

func (f \*File) Readdirnames(n int) (names []string, err error)

Readdirnames读取并返回目录f里面的文件的名字切片。

如果n>0，Readdirnames返回最多n个名字。在这种情况下，如果Readdirnames返回一个空的切片，它会返回一个非空的错误来解释原因。在目录的结尾，错误为EOF。

如果n<0，Readdirnames返回目录下所有的文件的名字，用一个切片表示。在这种情况下，如果用一个切片表示成功（读取直到目录结尾），它返回切片和一个空的错误。如果在目录结尾之前遇到了一个错误，Readdirnames返回直到当前所读到的names和一个非空的错误。

package main

import (

"fmt"

"os"

)

func main() {

file, err := os.Open("/home/chenbaoke")

if err != nil {

fmt.Println(err)

}

fileslice, err := file.Readdir(5)

if err != nil {

fmt.Println(err)

}

for \_, f := range fileslice {

fmt.Println(f.Name()) //输出５个文件的名字

}

filename, err := file.Readdirnames(-1)

if err != nil {

fmt.Println(err)

}

for \_, f := range filename {

fmt.Println(f)　　　　　//输出所有文件的名字

}

}

func (f \*File) Seek(offset int64, whence int) (ret int64, err error)　　　　//Seek设置下一次读或写操作的偏移量offset，根据whence来解析：0意味着相对于文件的原始位置，1意味着相对于当前偏移量，2意味着相对于文件结尾。它返回新的偏移量和错误（如果存在）。

func main() {

s := make([]byte, 10)

file, \_ := os.Open("tmp.txt")

defer file.Close()

file.Seek(-12, 2) // 从离最后位置12的地方开始

n, \_ := file.Read(s)

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

}

func (f \*File) Stat() (fi FileInfo, err error)　　//返回文件描述相关信息，包括大小，名字等．等价于os.Stat(filename string)  
func (f \*File) Sync() (err error)                        //同步操作，将当前存在内存中的文件内容写入硬盘．  
func (f \*File) Truncate(size int64) error        //类似  os.Truncate(name, size),，将文件进行截断  
func (f \*File) Write(b []byte) (n int, err error)　　//将b中的数据写入f文件  
func (f \*File) WriteAt(b []byte, off int64) (n int, err error)　//将b中数据写入f文件中，写入时从offset位置开始进行写入操作  
func (f \*File) WriteString(s string) (ret int, err error)　　　//将字符串s写入文件中

func main() {

file, \_ := os.Create("tmp.txt")

a := "hellobyte"

file.WriteAt([]byte(a), 10) //在文件file偏移量10处开始写入hellobyte

file.WriteString("string")　　//在文件file偏移量0处开始写入string

file.Write([]byte(a)) //在文件file偏移量string之后开始写入hellobyte，这个时候就会把开始利用WriteAt在offset为10处开始写入的hellobyte进行部分覆盖

b := make([]byte, 20)

file.Seek(0, 0) 　　　　//file指针指向文件开始位置

n, \_ := file.Read(b)

fmt.Println(string(b[:n])) //stringhellobytebyte，这是由于在写入过程中存在覆盖造成的

}

type FileInfo

type FileInfo interface {

Name() string //文件名字

Size() int64 // length in bytes for regular files; system-dependent for others，文件大小

Mode() FileMode // file mode bits，文件权限

ModTime() time.Time // modification time　文件更改时间

IsDir() bool // abbreviation for Mode().IsDir()　文件是否为目录

Sys() interface{} // underlying data source (can return nil)　　基础数据源

}

FileInfo经常被Stat和Lstat返回来描述一个文件  
func Lstat(name string) (fi FileInfo, err error)      //返回描述文件的FileInfo信息。如果文件是符号链接，返回的FileInfo描述的符号链接。Lstat不会试着去追溯link。如果出错，将是 \*PathError类型。  
func Stat(name string) (fi FileInfo, err error)       //返回描述文件的FileInfo信息。如果出错，将是 \*PathError类型。

type FileMode

type FileMode uint32

FileMode代表文件的模式和权限标志位。标志位在所有的操作系统有相同的定义，因此文件的信息可以从一个操作系统移动到另外一个操作系统。不是所有的标志位是用所有的系统。唯一要求的标志位是目录的ModeDir。

const (

// The single letters are the abbreviations

// used by the String method's formatting.

ModeDir FileMode = 1 << (32 - 1 - iota) // d: is a directory

ModeAppend // a: append-only

ModeExclusive // l: exclusive use

ModeTemporary // T: temporary file (not backed up)

ModeSymlink // L: symbolic link

ModeDevice // D: device file

ModeNamedPipe // p: named pipe (FIFO)

ModeSocket // S: Unix domain socket

ModeSetuid // u: setuid

ModeSetgid // g: setgid

ModeCharDevice // c: Unix character device, when ModeDevice is set

ModeSticky // t: sticky

// Mask for the type bits. For regular files, none will be set.

ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket | ModeDevice

ModePerm FileMode = 0777 // permission bits

)

所定义的文件标志位最重要的位是FileMode。9个次重要的位是标准Unix rwxrwxrwx权限。这些位的值应该被认为公开API的一部分，可能用于连接协议或磁盘表示：它们必须不能被改变，尽管新的标志位有可能增加。

FileModel的方法主要用来进行判断和输出权限  
func (m FileMode) IsDir() bool              //判断m是否是目录，也就是检查文件是否有设置的ModeDir位  
func (m FileMode) IsRegular() bool　　//判断m是否是普通文件，也就是说检查m中是否有设置mode type  
func (m FileMode) Perm() FileMode　　//返回m的权限位  
func (m FileMode) String() string　　　　//返回m的字符串表示

func main() {

fd, err := os.Stat("tmp.txt")

if err != nil {

fmt.Println(err)

}

fm := fd.Mode()

fmt.Println(fm.IsDir()) //false

fmt.Println(fm.IsRegular()) //true

fmt.Println(fm.Perm()) //-rwxrwxrwx

fmt.Println(fm.String()) //-rwxrwxrwx

}

type LinkError

type LinkError struct {

Op string

Old string

New string

Err error

}

func (e \*LinkError) Error() string　　　　//LinkError记录了一个在链接或者符号连接或者重命名的系统调用中发生的错误和引起错误的文件的路径。

type PathError

type PathError struct {

Op string

Path string

Err error

}

func (e \*PathError) Error() string　　//返回一个有操作者，路径以及错误组成的字符串形式

进程相关操作：  
type ProcAttr

type ProcAttr struct {

Dir string //如果dir不是空，子进程在创建之前先进入该目录

Env []string　　　//如果Env不是空，则将里面的内容赋值给新进程的环境变量，如果他为空，则使用默认的环境变量

Files []\*File　　　 //Files指定新进程打开文件，前三个实体分别为标准输入，标准输出和标准错误输出，可以添加额外的实体，这取决于底层的操作系统，当进程开始时，文

　　　　　　　　　　//件对应的空实体将被关闭

Sys \*syscall.SysProcAttr //操作系统特定进程的属性，设置该值也许会导致你的程序在某些操作系统上无法运行或者编译

}

ProcAttr包含属性，这些属性将会被应用在被StartProcess启动的新进程上type Process

Process存储了通过StartProcess创建的进程信息。

type Process struct {

Pid int

handle uintptr 　　//处理指针

isdone uint32 // 如果进程正在等待则该值非０，没有等待该值为０

}

func FindProcess(pid int) (p \*Process, err error)　　　　//通过进程pid查找运行的进程，返回相关进程信息及在该过程中遇到的errorfunc StartProcess(name string, argv []string, attr \*ProcAttr) (\*Process, error)  //StartProcess启动一个新的进程，其传入的name、argv和addr指定了程序、参数和属性；StartProcess是一个低层次的接口。os/exec包提供了高层次的接口；如果出错，将会是\*PathError错误。func (p \*Process) Kill() error　　　　　　　　　　　//杀死进程并直接退出func (p \*Process) Release() error　　　　　　　　//释放进程p的所有资源，之后进程p便不能够再被使用，只有Wati没有被调用时，才需要调用Release释放资源

func main() {

    attr := &os.ProcAttr{

        Files: []\*os.File{os.Stdin, os.Stdout, os.Stderr}, //其他变量如果不清楚可以不设定

    }

    p, err := os.StartProcess("/usr/bin/vim", []string{"/usr/bin/vim", "tmp.txt"}, attr) //vim 打开tmp.txt文件

    if err != nil {

        fmt.Println(err)

    }

    fmt.Println(p)                  //&{5488 240 0}

    pro, \_ := os.FindProcess(p.Pid) //查找进程

    fmt.Println(pro)                //&{5488 240 0}

    err = p.Kill() 　　　　　//杀死进程但不释放进程相关资源

    fmt.Println(err)

    err = p.Release() 　　　//<span style="color:#FF0000;">释放进程相关资源，因为资源释放凋之后进程p就不能进行任何操作，此后进程Ｐ的任何操作都会被报错</span>

    fmt.Println(err)

}

func (p \*Process) Signal(sig Signal) error　　　　//发送一个信号给进程p, 在windows中没有实现发送中断interrupt  
func (p \*Process) Wait() (\*ProcessState, error)　　//Wait等待进程退出，并返回进程状态和错误。Wait释放进程相关的资源。在大多数的系统上，进程p必须是当前进程的子进程，否则会返回一个错误。

func main() {

    attr := &os.ProcAttr{

        Files: []\*os.File{os.Stdin, os.Stdout, os.Stderr}, //其他变量如果不清楚可以不设定

    }

    p, err := os.StartProcess("/usr/bin/vim", []string{"/usr/bin/vim", "tmp.txt"}, attr) //vim 打开tmp.txt文件

    if err != nil {

        fmt.Println(err)

    }

    go func() {

        p.Signal(os.Kill) //kill process

    }()

    pstat, err := p.Wait()

    if err != nil {

        fmt.Println(err)

    }

    fmt.Println(pstat) //signal: killed

}

type ProcessState     //ProcessState存储了Wait函数报告的进程信息

type ProcessState struct {

pid int

status syscall.WaitStatus

rusage \*syscall.Rusage

}

func (p \*ProcessState) Exited() bool　　　　　　// 判断程序是否已经退出  
func (p \*ProcessState) Pid() int                                //返回退出进程的pid  
func (p \*ProcessState) String() string                     //获取进程状态的字符串表示  
func (p \*ProcessState) Success() bool                    //判断程序是否成功退出，而Exited则仅仅只是判断其是否退出  
func (p \*ProcessState) Sys() interface{}                //返回有关进程的系统独立的退出信息。并将它转换为恰当的底层类型（比如Unix上的syscall.WaitStatus），主要用来获取进程退出相关资源。  
func (p \*ProcessState) SysUsage() interface{}         //SysUsage返回关于退出进程的系统独立的资源使用信息。主要用来获取进程使用系统资源  
func (p \*ProcessState) SystemTime() time.Duration      //获取退出进程的内核态cpu使用时间  
func (p \*ProcessState) UserTime() time.Duration     //返回退出进程和子进程的用户态CPU使用时间

func main() {

attr := &os.ProcAttr{

Files: []\*os.File{os.Stdin, os.Stdout, os.Stderr}, //其他变量如果不清楚可以不设定

}

p, err := os.StartProcess("/usr/bin/vim", []string{"/usr/bin/vim", "tmp.txt"}, attr) //vim 打开tmp.txt文件

if err != nil {

fmt.Println(err)

}

ps, \_ := p.Wait()　　//当关闭vim打开的tmp.txt时，进程就结束了

fmt.Println(ps) 　　//exit status 0

if ps.Exited() {

fmt.Println(ps.Pid()) //536

fmt.Println(ps.String()) // exit status 0

fmt.Println(ps.Success()) //true

fmt.Println(ps.Sys()) //0

fmt.Println(ps.SysUsage()) //&{{0 313157} {0 42898} 29164 0 0 0 7745 41 0 10736 240 0 0 0 1238 143}

fmt.Println(ps.SystemTime()) // 42.898ms

fmt.Println(ps.UserTime()) //313.157ms

}

}

type Signal

type Signal interface {

String() string

Signal() // 同其他字符串做区别

}

代表操作系统的信号．底层的实现是操作系统独立的：在Unix中是syscal.Signal．

var (

Interrupt Signal = syscall.SIGINT

Kill Signal = syscall.SIGKILL

)

在所有系统中都能够使用的是interrupe,给进程发送一个信号，强制杀死该进程  
type SyscallError　//SyscallError记录了一个特定系统调用的错误，主要用来返回SyscallError的字符串表示

type SyscallError struct {

Syscall string

Err error

}

func (e \*SyscallError) Error() string　　　//返回SyscallError的字符串表示