# **1. os — 平台无关的操作系统功能实现**

os 包提供了平台无关的操作系统功能接口。尽管错误处理是 go 风格的，但设计是 Unix 风格的；所以，失败的调用会返回 error 而非错误码。通常 error 里会包含更多信息。例如，如果使用一个文件名的调用（如 Open、Stat）失败了，打印错误时会包含该文件名，错误类型将为 \*PathError，其内部可以解包获得更多信息。

os 包规定为所有操作系统实现的接口都是一致的。有一些某个系统特定的功能，需要使用 syscall 获取。**实际上，os 依赖于 syscall。在实际编程中，我们应该总是优先使用 os 中提供的功能，而不是 syscall**。

下面是一个简单的例子，打开一个文件并从中读取一些数据：

file, err := os.Open("file.go") // For read access.

if err != nil {

log.Fatal(err)

}

如果打开失败，错误字符串是自解释的，例如：

open file.go: no such file or directory

而不像 C 语言，需要额外的函数（或宏）来解释错误码。

## **1.1. 文件 I/O**

在第一章，我们较全面的介绍了 Go 中的 I/O。本节，我们着重介绍文件相关的 I/O。因为 I/O 操作涉及到系统调用，在讲解时会涉及到 Unix 在这方面的系统调用。

在 Unix 系统调用中，所有 I/O 操作以文件描述符 ( 一个非负整数 , 通常是小整数 ) 来指代打开的文件。文件描述符用以表示所有类型的已打开文件，包括管道（pipe）、FIFO、socket、终端、设备和普通文件。这里，我们主要介绍普通文件的 I/O。

在 Go 中，文件描述符封装在 os.File 结构中，通过 File.Fd() 可以获得底层的文件描述符：fd。

按照惯例，大多数程序都期望能够使用 3 种标准的文件描述符：0- 标准输入；1- 标准输出；2- 标准错误。os 包提供了 3 个 File 对象，分别代表这 3 种标准描述符：Stdin、Stdout 和 Stderr，它们对应的文件名分别是：/dev/stdin、/dev/stdout 和 /dev/stderr。注意，这里说的文件名，并不一定存在，比如 Windows 下就没有。

### **1.1.1. 打开一个文件：OpenFile**

OpenFile 既能打开一个已经存在的文件，也能创建并打开一个新文件。

func OpenFile(name string, flag int, perm FileMode) (\*File, error)

OpenFile 是一个更一般性的文件打开函数，大多数调用者都应用 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。

要打开的文件由参数 name 指定，它可以是绝对路径或相对路径（相对于进程当前工作目录），也可以是一个符号链接（会对其进行解引用）。

位掩码参数 flag 用于指定文件的访问模式，可用的值在 os 中定义为常量（以下值并非所有操作系统都可用）：

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

)

其中，O\_RDONLY、O\_WRONLY、O\_RDWR 应该只指定一个，剩下的通过 | 操作符来指定。该函数内部会给 flags 加上 syscall.O\_CLOEXEC，在 fork 子进程时会关闭通过 OpenFile 打开的文件，即子进程不会重用该文件描述符。

*注意：由于历史原因，O\_RDONLY | O\_WRONLY 并非等于 O\_RDWR，它们的值一般是 0、1、2。*

位掩码参数 perm 指定了文件的模式和权限位，类型是 os.FileMode，文件模式位常量定义在 os 中：

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）

)

以上常量在所有操作系统都有相同的含义（可用时），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。

以上这些被定义的位是 FileMode 最重要的位。另外 9 个位（权限位）为标准 Unix rwxrwxrwx 权限（所有人都可读、写、运行）。

FileMode 还定义了几个方法，用于判断文件类型的 IsDir() 和 IsRegular()，用于获取权限的 Perm()。

返回的 error，具体实现是 \*os.PathError，它会记录具体操作、文件路径和错误原因。

另外，在 OpenFile 内部会调用 NewFile，来得到 File 对象。

****使用方法****

打开一个文件，一般通过 Open 或 Create，我们看这两个函数的实现。

func Open(name string) (\*File, error) {

return OpenFile(name, O\_RDONLY, 0)

}

func Create(name string) (\*File, error) {

return OpenFile(name, O\_RDWR|O\_CREATE|O\_TRUNC, 0666)

}

### **1.1.2. 读取文件内容：Read**

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

Read 方法从 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。

从方法声明可以知道，File 实现了 io.Reader 接口。

Read 对应的系统调用是 read。

对比下 ReadAt 方法：

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

ReadAt 从指定的位置（相对于文件开始位置）读取长度为 len(b) 个字节数据并写入 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。它对应的系统调用是 pread。

**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。

### **1.1.3. 数据写入文件：Write**

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

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。

从方法声明可以知道，File 实现了 io.Writer 接口。

Write 对应的系统调用是 write。

Write 与 WriteAt 的区别同 Read 与 ReadAt 的区别一样。为了方便，还提供了 WriteString 方法，它实际是对 Write 的封装。

注意：Write 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：

1. 打开文件时指定 `os.O\_SYNC`；

2. 调用 `File.Sync()` 方法。

说明：File.Sync() 底层调用的是 fsync 系统调用，这会将数据和元数据都刷到磁盘；如果只想刷数据到磁盘（比如，文件大小没变，只是变了文件数据），需要自己封装，调用 fdatasync 系统调用。（syscall.Fdatasync）

### **1.1.4. 关闭文件：Close**

close() 系统调用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，将自动关闭其已打开的所有文件描述符。

func (f \*File) Close() error

os.File.Close() 是对 close() 的封装。我们应该养成关闭不需要的文件的良好编程习惯。文件描述符是资源，Go 的 gc 是针对内存的，并不会自动回收资源，如果不关闭文件描述符，长期运行的服务可能会把文件描述符耗尽。

所以，通常的写法如下：

file, err := os.Open("/tmp/studygolang.txt")

if err != nil {

// 错误处理，一般会阻止程序往下执行

return

}

defer file.Close()

****关于返回值**error**

以下两种情况会导致 Close 返回错误：

1. 关闭一个未打开的文件；

2. 两次关闭同一个文件；

通常，我们不会去检查 Close 的错误。

### **1.1.5. 改变文件偏移量：Seek**

对于每个打开的文件，系统内核会记录其文件偏移量，有时也将文件偏移量称为读写偏移量或指针。文件偏移量是指执行下一个 Read 或 Write 操作的文件其实位置，会以相对于文件头部起始点的文件当前位置来表示。文件第一个字节的偏移量为 0。

文件打开时，会将文件偏移量设置为指向文件开始，以后每次 Read 或 Write 调用将自动对其进行调整，以指向已读或已写数据后的下一个字节。因此，连续的 Read 和 Write 调用将按顺序递进，对文件进行操作。

而 Seek 可以调整文件偏移量。方法定义如下：

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

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。使用中，whence 应该使用 os 包中的常量：SEEK\_SET、SEEK\_CUR 和 SEEK\_END。

注意：Seek 只是调整内核中与文件描述符相关的文件偏移量记录，并没有引起对任何物理设备的访问。

一些 Seek 的使用例子（file 为打开的文件对象），注释说明了将文件偏移量移动到的具体位置：

file.Seek(0, os.SEEK\_SET) // 文件开始处

file.Seek(0, SEEK\_END) // 文件结尾处的下一个字节

file.Seek(-1, SEEK\_END) // 文件最后一个字节

file.Seek(-10, SEEK\_CUR) // 当前位置前 10 个字节

file.Seek(1000, SEEK\_END) // 文件结尾处的下 1001 个字节

最后一个例子在文件中会产生“空洞”。

Seek 对应系统调用 lseek。该系统调用并不适用于所有类型，不允许将 lseek 应用于管道、FIFO、socket 或 终端。

## **1.2. 截断文件**

trucate 和 ftruncate 系统调用将文件大小设置为 size 参数指定的值；Go 语言中相应的包装函数是 os.Truncate 和 os.File.Truncate。

func Truncate(name string, size int64) error

func (f \*File) Truncate(size int64) error

如果文件当前长度大于参数 size，调用将丢弃超出部分，若小于参数 size，调用将在文件尾部添加一系列空字节或是一个文件空洞。

它们之间的区别在于如何指定操作文件：

1. `Truncate` 以路径名称字符串来指定文件，并要求可访问该文件（即对组成路径名的各目录拥有可执行 (x) 权限），且对文件拥有写权限。若文件名为符号链接，那么调用将对其进行解引用。

2. 很明显，调用 `File.Truncate` 前，需要先以可写方式打开操作文件，该方法不会修改文件偏移量。

## **1.3. 文件属性**

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。函数 Stat、Lstat 和 File.Stat 可以得到该接口的实例。这三个函数对应三个系统调用：stat、lstat 和 fstat。

这三个函数的区别：

1. stat 会返回所命名文件的相关信息。
2. 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）。
3. fstat 则会返回由某个打开文件描述符（Go 中则是当前打开文件 File）所指代文件的相关信息。

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。

FileInfo 接口如下：

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 // 文件的模式位

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）

}

Sys() 底层数据的 C 语言 结构 statbuf 格式如下：

struct stat {

dev\_t st\_dev; // 设备 ID

ino\_t st\_ino; // 文件 i 节点号

mode\_t st\_mode; // 位掩码，文件类型和文件权限

nlink\_t st\_nlink; // 硬链接数

uid\_t st\_uid; // 文件属主，用户 ID

gid\_t st\_gid; // 文件属组，组 ID

dev\_t st\_rdev; // 如果针对设备 i 节点，则此字段包含主、辅 ID

off\_t st\_size; // 常规文件，则是文件字节数；符号链接，则是链接所指路径名的长度，字节为单位；对于共享内存对象，则是对象大小

blksize\_t st\_blsize; // 分配给文件的总块数，块大小为 512 字节

blkcnt\_t st\_blocks; // 实际分配给文件的磁盘块数量

time\_t st\_atime; // 对文件上次访问时间

time\_t st\_mtime; // 对文件上次修改时间

time\_t st\_ctime; // 文件状态发生改变的上次时间

}

Go 中 syscal.Stat\_t 与该结构对应。

如果我们要获取 FileInfo 接口没法直接返回的信息，比如想获取文件的上次访问时间，示例如下：

fileInfo, err := os.Stat("test.log")

if err != nil {

log.Fatal(err)

}

sys := fileInfo.Sys()

stat := sys.(\*syscall.Stat\_t)

fmt.Println(time.Unix(stat.Atimespec.Unix()))

### **1.3.1. 改变文件时间戳**

可以显式改变文件的访问时间和修改时间。

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

Chtimes 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 类型的错误。在 Unix 中，底层实现会调用 utimenstat()，它提供纳秒级别的精度。

### **1.3.2. 文件属主**

每个文件都有一个与之关联的用户 ID（UID）和组 ID（GID），籍此可以判定文件的属主和属组。系统调用 chown、lchown 和 fchown 可用来改变文件的属主和属组，Go 中对应的函数或方法：

func Chown(name string, uid, gid int) error

func Lchown(name string, uid, gid int) error

func (f \*File) Chown(uid, gid int) error

它们的区别和上文提到的 Stat 相关函数类似。

### **1.3.3. 文件权限**

这里介绍是应用于文件和目录的权限方案，尽管此处讨论的权限主要是针对普通文件和目录，但其规则可适用于所有文件类型，包括设备文件、FIFO 以及 Unix 域套接字等。

#### **普通文件的权限**

如前所述，os.FileMode 或 C 结构 stat 中的 st\_mod 的低 12 位定义了文件权限。其中前 3 位为专用位，分别是 set-user-ID 位、set-group-ID 位和 sticky 位。其余 9 位则构成了定义权限的掩码，分别授予访问文件的各类用户。文件权限掩码分为 3 类：

* Owner（亦称为 user）：授予文件属主的权限。
* Group：授予文件属组成员用户的权限。
* Other：授予其他用户的权限。

可为每一类用户授予的权限如下：

* Read：可阅读文件的内容。
* Write：可更改文件的内容。
* Execute：可以执行文件（如程序或脚本）。

Unix 中表示：rwxrwxrwx。

#### **目录权限**

目录与文件拥有相同的权限方案，只是对 3 种权限的含义另有所指。

* 读权限：可列出（比如，通过 ls 命令）目录之下的内容（即目录下的文件名）。
* 写权限：可在目录内创建、删除文件。注意，要删除文件，对文件本身无需有任何权限。
* 可执行权限：可访问目录中的文件。因此，有时也将对目录的执行权限称为 search（搜索）权限。

访问文件时，需要拥有对路径名所列所有目录的执行权限。例如，想读取文件 /home/studygolang/abc，则需拥有对目录 /、/home 以及 /home/studygolang 的执行权限（还要有对文件 abc 自身的读权限）。

#### **相关函数或方法**

在文件相关操作报错时，可以通过 os.IsPermission 检查是否是权限的问题。

func IsPermission(err error) bool

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。

另外，syscall.Access 可以获取文件的权限。这对应系统调用 access。

#### **Sticky 位**

除了 9 位用来表明属主、属组和其他用户的权限外，文件权限掩码还另设有 3 个附加位，分别是 set-user-ID(bit 04000)、set-group-ID(bit 02000) 和 sticky(bit 01000) 位。set-user-ID 和 set-group-ID 权限位将在进程章节介绍。这里介绍 sticky 位。

Sticky 位一般用于目录，起限制删除位的作用，表明仅当非特权进程具有对目录的写权限，且为文件或目录的属主时，才能对目录下的文件进行删除和重命名操作。根据这个机制来创建为多个用户共享的一个目录，各个用户可在其下创建或删除属于自己的文件，但不能删除隶属于其他用户的文件。/tmp 目录就设置了 sticky 位，正是出于这个原因。

chmod 命令或系统调用可以设置文件的 sticky 位。若对某文件设置了 sticky 位，则 ls -l 显示文件时，会在其他用户执行权限字段上看到字母 t（有执行权限时） 或 T（无执行权限时）。

os.Chmod 和 os.File.Chmod 可以修改文件权限（包括 sticky 位），分别对应系统调用 chmod 和 fchmod。

func main() {

file, err := os.Create("studygolang.txt")

if err != nil {

log.Fatal("error:", err)

}

defer file.Close()

fileMode := getFileMode(file)

log.Println("file mode:", fileMode)

file.Chmod(fileMode | os.ModeSticky)

log.Println("change after, file mode:", getFileMode(file))

}

func getFileMode(file \*os.File) os.FileMode {

fileInfo, err := file.Stat()

if err != nil {

log.Fatal("file stat error:", err)

}

return fileInfo.Mode()

}

// Output:

// 2016/06/18 15:59:06 file mode: -rw-rw-r--

// 2016/06/18 15:59:06 change after, file mode: trw-rw-r--

// ls -l 看到的 studygolang.tx 是：-rw-rw-r-T

// 当然这里是给文件设置了 sticky 位，对权限不起作用。系统会忽略它。

## **1.4. 目录与链接**

在 Unix 文件系统中，目录的存储方式类似于普通文件。目录和普通文件的区别有二：

* 在其 i-node 条目中，会将目录标记为一种不同的文件类型。
* 目录是经特殊组织而成的文件。本质上说就是一个表格，包含文件名和 i-node 标号。

### **1.4.1. 创建和移除（硬）链接**

硬链接是针对文件而言的，目录不允许创建硬链接。

link 和 unlink 系统调用用于创建和移除（硬）链接。Go 中 os.Link 对应 link 系统调用；但 os.Remove 的实现会先执行 unlink 系统调用，如果要移除的是目录，则 unlink 会失败，这时 Remove 会再调用 rmdir 系统调用。

func Link(oldname, newname string) error

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

func Remove(name string) error

Remove 删除 name 指定的文件或目录。如果出错，会返回 \*PathError 类型的错误。如果目录不为空，Remove 会返回失败。

### **1.4.2. 更改文件名**

系统调用 rename 既可以重命名文件，又可以将文件移至同一个文件系统中的另一个目录。该系统调用既可以用于文件，也可以用于目录。相关细节，请查阅相关资料。

Go 中的 os.Rename 是对应的封装函数。

func Rename(oldpath, newpath string) error

Rename 修改一个文件的名字或移动一个文件。如果 newpath 已经存在，则替换它。注意，可能会有一些个操作系统特定的限制。

### **1.4.3. 使用符号链接**

symlink 系统调用用于为指定路径名创建一个新的符号链接（想要移除符号链接，使用 unlink）。Go 中的 os.Symlink 是对应的封装函数。

func Symlink(oldname, newname string) error

Symlink 创建一个名为 newname 指向 oldname 的符号链接。如果出错，会返回 \*LinkError 类型的错误。

由 oldname 所命名的文件或目录在调用时无需存在。因为即便当时存在，也无法阻止后来将其删除。这时，newname 成为“悬空链接”，其他系统调用试图对其进行解引用操作都将错误（通常错误号是 ENOENT）。

有时候，我们希望通过符号链接，能获取其所指向的路径名。系统调用 readlink 能做到，Go 的封装函数是 os.Readlink：

func Readlink(name string) (string, error)

Readlink 获取 name 指定的符号链接指向的文件的路径。如果出错，会返回 \*PathError 类型的错误。我们看看 Readlink 的实现。

func Readlink(name string) (string, error) {

for len := 128; ; len \*= 2 {

b := make([]byte, len)

n, e := fixCount(syscall.Readlink(name, b))

if e != nil {

return "", &PathError{"readlink", name, e}

}

if n < len {

return string(b[0:n]), nil

}

}

}

这里之所以用循环，是因为我们没法知道文件的路径到底多长，如果 b 长度不够，文件名会被截断，而 readlink 系统调用无非分辨所返回的字符串到底是经过截断处理，还是恰巧将 b 填满。这里采用的验证方法是分配一个更大的（两倍）b 并再次调用 readlink。

### **1.4.4. 创建和移除目录**

mkdir 系统调用创建一个新目录，Go 中的 os.Mkdir 是对应的封装函数。

func Mkdir(name string, perm FileMode) error

Mkdir 使用指定的权限和名称创建一个目录。如果出错，会返回 \*PathError 类型的错误。

name 参数指定了新目录的路径名，可以是相对路径，也可以是绝对路径。如果已经存在，则调用失败并返回 os.ErrExist 错误。

perm 参数指定了新目录的权限。对该位掩码值的指定方式和 os.OpenFile 相同，也可以直接赋予八进制数值。注意，perm 值还将于进程掩码相与（&）。如果 perm 中设置了 sticky 位，那么将对新目录设置该权限。

因为 Mkdir 所创建的只是路径名中的最后一部分，如果父目录不存在，创建会失败。os.MkdirAll 用于递归创建所有不存在的目录。

建议读者阅读下 os.MkdirAll 的源码，了解其实现方式、技巧。

rmdir 系统调用移除一个指定的目录，目录可以是绝对路径或相对路径。在讲解 unlink 时，已经介绍了 Go 中的 os.Remove。注意，这里要求目录必须为空。为了方便使用，Go 中封装了一个 os.RemoveAll 函数：

func RemoveAll(path string) error

RemoveAll 删除 path 指定的文件，或目录及它包含的任何下级对象。它会尝试删除所有东西，除非遇到错误并返回。如果 path 指定的对象不存在，RemoveAll 会返回 nil 而不返回错误。

RemoveAll 的内部实现逻辑如下：

1. 调用 Remove 尝试进行删除，如果成功或返回 path 不存在，则直接返回 nil；
2. 调用 Lstat 获取 path 信息，以便判断是否是目录。注意，这里使用 Lstat，表示不对符号链接解引用；
3. 调用 Open 打开目录，递归读取目录中内容，执行删除操作。

阅读 RemoveAll 源码，可以掌握马上要介绍的读目录内容或遍历目录。

### **1.4.5. 读目录**

POSIX 与 SUS 定义了读取目录相关的 C 语言标准，各个操作系统提供的系统调用却不尽相同。Go 没有基于 C 语言，而是自己通过系统调用实现了读目录功能。

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

Readdirnames 读取目录 f 的内容，返回一个最多有 n 个成员的[]string，切片成员为目录中文件对象的名字，采用目录顺序。对本函数的下一次调用会返回上一次调用未读取的内容的信息。

如果 n>0，Readdirnames 函数会返回一个最多 n 个成员的切片。这时，如果 Readdirnames 返回一个空切片，它会返回一个非 nil 的错误说明原因。如果到达了目录 f 的结尾，返回值 err 会是 io.EOF。

如果 n<=0，Readdirnames 函数返回目录中剩余所有文件对象的名字构成的切片。此时，如果 Readdirnames 调用成功（读取所有内容直到结尾），它会返回该切片和 nil 的错误值。如果在到达结尾前遇到错误，会返回之前成功读取的名字构成的切片和该错误。

func (f \*File) Readdir(n int) (fi []FileInfo, err error)

Readdir 内部会调用 Readdirnames，将得到的 names 构造路径，通过 Lstat 构造出 []FileInfo。

列出某个目录的文件列表示例程序见 [dirtree](http://books.studygolang.com/The-Golang-Standard-Library-by-Example/code/src/chapter06/os/dirtree/main.go)。

# **2. path/filepath — 兼容操作系统的文件路径操作**

path/filepath 包涉及到路径操作时，路径分隔符使用 os.PathSeparator。不同系统，路径表示方式有所不同，比如 Unix 和 Windows 差别很大。本包能够处理所有的文件路径，不管是什么系统。

注意，路径操作函数并不会校验路径是否真实存在。

## **2.1. 解析路径名字符串**

Dir() 和 Base() 函数将一个路径名字符串分解成目录和文件名两部分。（注意一般情况，这些函数与 Unix 中 dirname 和 basename 命令类似，但如果路径以 / 结尾，Dir 的行为和 dirname 不太一致。）

func Dir(path string) string

func Base(path string) string

Dir 返回路径中除去最后一个路径元素的部分，即该路径最后一个元素所在的目录。在使用 Split 去掉最后一个元素后，会简化路径并去掉末尾的斜杠。如果路径是空字符串，会返回 "."；如果路径由 1 到多个斜杠后跟 0 到多个非斜杠字符组成，会返回 "/"；其他任何情况下都不会返回以斜杠结尾的路径。

Base 函数返回路径的最后一个元素。在提取元素前会去掉末尾的斜杠。如果路径是 ""，会返回 "."；如果路径是只有一个斜杆构成的，会返回 "/"。

比如，给定路径名 /home/polaris/studygolang.go，Dir 返回 /home/polaris，而 Base 返回 studygolang.go。

如果给定路径名 /home/polaris/studygolang/，Dir 返回 /home/polaris/studygolang（这与 Unix 中的 dirname 不一致，dirname 会返回 /home/polaris），而 Base 返回 studygolang。

有人提出此问题，见[issue13199](https://github.com/golang/go/issues/13199" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter06/_blank)，不过官方认为这不是问题，如果需要和 dirname 一样的功能，应该自己处理，比如在调用 Dir 之前，先将末尾的 / 去掉。

此外，Ext 可以获得路径中文件名的扩展名。

func Ext(path string) string

Ext 函数返回 path 文件扩展名。扩展名是路径中最后一个从 . 开始的部分，包括 .。如果该元素没有 . 会返回空字符串。

## **2.2. 相对路径和绝对路径**

某个进程都会有当前工作目录（进程相关章节会详细介绍），一般的相对路径，就是针对进程当前工作目录而言的。当然，可以针对某个目录指定相对路径。

绝对路径，在 Unix 中，以 / 开始；在 Windows 下以某个盘符开始，比如 C:\Program Files。

func IsAbs(path string) bool

IsAbs 返回路径是否是一个绝对路径。而

func Abs(path string) (string, error)

Abs 函数返回 path 代表的绝对路径，如果 path 不是绝对路径，会加入当前工作目录以使之成为绝对路径。因为硬链接的存在，不能保证返回的绝对路径是唯一指向该地址的绝对路径。在 os.Getwd 出错时，Abs 会返回该错误，一般不会出错，如果路径名长度超过系统限制，则会报错。

func Rel(basepath, targpath string) (string, error)

Rel 函数返回一个相对路径，将 basepath 和该路径用路径分隔符连起来的新路径在词法上等价于 targpath。也就是说，Join(basepath, Rel(basepath, targpath)) 等价于 targpath。如果成功执行，返回值总是相对于 basepath 的，即使 basepath 和 targpath 没有共享的路径元素。如果两个参数一个是相对路径而另一个是绝对路径，或者 targpath 无法表示为相对于 basepath 的路径，将返回错误。

fmt.Println(filepath.Rel("/home/polaris/studygolang", "/home/polaris/studygolang/src/logic/topic.go"))

fmt.Println(filepath.Rel("/home/polaris/studygolang", "/data/studygolang"))

// Output:

// src/logic/topic.go <nil>

// ../../../data/studygolang <nil>

## **2.3. 路径的切分和拼接**

对于一个常规文件路径，我们可以通过 Split 函数得到它的目录路径和文件名：

func Split(path string) (dir, file string)

Split 函数根据最后一个路径分隔符将路径 path 分隔为目录和文件名两部分（dir 和 file）。如果路径中没有路径分隔符，函数返回值 dir 为空字符串，file 等于 path；反之，如果路径中最后一个字符是 /，则 dir 等于 path，file 为空字符串。返回值满足 path == dir+file。dir 非空时，最后一个字符总是 /。

// dir == /home/polaris/，file == studygolang

filepath.Split("/home/polaris/studygolang")

// dir == /home/polaris/studygolang/，file == ""

filepath.Split("/home/polaris/studygolang/")

// dir == ""，file == studygolang

filepath.Split("studygolang")

相对路径到绝对路径的转变，需要经过路径的拼接。Join 用于将多个路径拼接起来，会根据情况添加路径分隔符。

func Join(elem ...string) string

Join 函数可以将任意数量的路径元素放入一个单一路径里，会根据需要添加路径分隔符。结果是经过 Clean 的，所有的空字符串元素会被忽略。对于拼接路径的需求，我们应该总是使用 Join 函数来处理。

有时，我们需要分割 PATH 或 GOPATH 之类的环境变量（这些路径被特定于 OS 的列表分隔符连接起来），filepath.SplitList 就是这个用途：

func SplitList(path string) []string

注意，与 strings.Split 函数的不同之处是：对 ""，SplitList 返回[]string{}，而 strings.Split 返回 []string{""}。SplitList 内部调用的是 strings.Split。

## **2.4. 规整化路径**

func Clean(path string) string

Clean 函数通过单纯的词法操作返回和 path 代表同一地址的最短路径。

它会不断的依次应用如下的规则，直到不能再进行任何处理：

* 将连续的多个路径分隔符替换为单个路径分隔符
* 剔除每一个 . 路径名元素（代表当前目录）
* 剔除每一个路径内的 .. 路径名元素（代表父目录）和它前面的非 .. 路径名元素
* 剔除开始于根路径的 .. 路径名元素，即将路径开始处的 /.. 替换为 /（假设路径分隔符是 /）

返回的路径只有其代表一个根地址时才以路径分隔符结尾，如 Unix 的 / 或 Windows 的 C:\。

如果处理的结果是空字符串，Clean 会返回 .，代表当前路径。

## **2.5. 符号链接指向的路径名**

在上一节 os 包中介绍了 Readlink，可以读取符号链接指向的路径名。不过，如果原路径中又包含符号链接，Readlink 却不会解析出来。filepath.EvalSymlinks 会将所有路径的符号链接都解析出来。除此之外，它返回的路径，是直接可访问的。

func EvalSymlinks(path string) (string, error)

如果 path 或返回值是相对路径，则是相对于进程当前工作目录。

os.Readlink 和 filepath.EvalSymlinks 区别示例程序：

// 在当前目录下创建一个 studygolang.txt 文件和一个 symlink 目录，在 symlink 目录下对 studygolang.txt 建一个符号链接 studygolang.txt.2

fmt.Println(filepath.EvalSymlinks("symlink/studygolang.txt.2"))

fmt.Println(os.Readlink("symlink/studygolang.txt.2"))

// Ouput:

// studygolang.txt <nil>

// ../studygolang.txt <nil>

## **2.6. 文件路径匹配**

func Match(pattern, name string) (matched bool, err error)

Match 指示 name 是否和 shell 的文件模式匹配。模式语法如下：

pattern:

{ term }

term:

'\*' 匹配 0 或多个非路径分隔符的字符

'?' 匹配 1 个非路径分隔符的字符

'[' [ '^' ] { character-range } ']'

字符组（必须非空）

c 匹配字符 c（c != '\*', '?', '\\', '['）

'\\' c 匹配字符 c

character-range:

c 匹配字符 c（c != '\\', '-', ']'）

'\\' c 匹配字符 c

lo '-' hi 匹配区间[lo, hi]内的字符

匹配要求 pattern 必须和 name 全匹配上，不只是子串。在 Windows 下转义字符被禁用。

Match 函数很少使用，搜索了一遍，标准库没有用到这个函数。而 Glob 函数在模板标准库中被用到了。

func Glob(pattern string) (matches []string, err error)

Glob 函数返回所有匹配了 模式字符串 pattern 的文件列表或者 nil（如果没有匹配的文件）。pattern 的语法和 Match 函数相同。pattern 可以描述多层的名字，如 /usr/\*/bin/ed（假设路径分隔符是 /）。

注意，Glob 会忽略任何文件系统相关的错误，如读目录引发的 I/O 错误。唯一的错误和 Match 一样，在 pattern 不合法时，返回 filepath.ErrBadPattern。返回的结果是根据文件名字典顺序进行了排序的。

Glob 的常见用法，是读取某个目录下所有的文件，比如写单元测试时，读取 testdata 目录下所有测试数据：

filepath.Glob("testdata/\*.input")

## **2.7. 遍历目录**

在介绍 os 时，讲解了读取目录的方法，并给出了一个遍历目录的示例。在 filepath 中，提供了 Walk 函数，用于遍历目录树。

func Walk(root string, walkFn WalkFunc) error

Walk 函数会遍历 root 指定的目录下的文件树，对每一个该文件树中的目录和文件都会调用 walkFn，包括 root 自身。所有访问文件 / 目录时遇到的错误都会传递给 walkFn 过滤。文件是按字典顺序遍历的，这让输出更漂亮，但也导致处理非常大的目录时效率会降低。Walk 函数不会遍历文件树中的符号链接（快捷方式）文件包含的路径。

walkFn 的类型 WalkFunc 的定义如下：

type WalkFunc func(path string, info os.FileInfo, err error) error

Walk 函数对每一个文件 / 目录都会调用 WalkFunc 函数类型值。调用时 path 参数会包含 Walk 的 root 参数作为前缀；就是说，如果 Walk 函数的 root 为 "dir"，该目录下有文件 "a"，将会使用 "dir/a" 作为调用 walkFn 的参数。walkFn 参数被调用时的 info 参数是 path 指定的地址（文件 / 目录）的文件信息，类型为 os.FileInfo。

如果遍历 path 指定的文件或目录时出现了问题，传入的参数 err 会描述该问题，WalkFunc 类型函数可以决定如何去处理该错误（Walk 函数将不会深入该目录）；如果该函数返回一个错误，Walk 函数的执行会中止；只有一个例外，如果 Walk 的 walkFn 返回值是 SkipDir，将会跳过该目录的内容而 Walk 函数照常执行处理下一个文件。

和 os 遍历目录树的示例对应，使用 Walk 遍历目录树的示例程序在 [walk](http://books.studygolang.com/The-Golang-Standard-Library-by-Example/code/src/chapter06/filepath/walk/main.go)，程序简单很多。

## **2.8. Windows 起作用的函数**

filepath 中有三个函数：VolumeName、FromSlash 和 ToSlash，针对非 Unix 平台的。

## **2.9. 关于 path 包**

path 包提供了对 / 分隔的路径的实用操作函数。

在 Unix 中，路径的分隔符是 /，但 Windows 是 \。在使用 path 包时，应该总是使用 /，不论什么系统。

path 包中提供的函数，filepath 都有提供，功能类似，但实现不同。

一般应该总是使用 filepath 包，而不是 path 包。

# **3. io/fs — 抽象文件系统**

Go 语言从 1.16 开始增加了 io/fs 包，该包定义了一个文件系统需要的相关基础接口，因此我们称之为抽象文件系统。该文件系统是层级文件系统或叫树形文件系统，Unix 文件系统就是这种类型。

本节除了讲解标准库的相关内容，还会实现一个文件系统作为例子。

注意，因为抽象了一个文件系统，之前 os 包中和文件系统相关的功能都移到 io/fs 包了，os 中的原类型只是 io/fs 对应类型的别名。如果你的系统要求 Go1.16，应该优先使用 io/fs 包。

## **3.1. 三个核心接口**

一个文件系统有些必要的元素，io/fs 包提供两个最小的接口来表示，即 fs.FS 和 fs.File。但因为 fs.File 依赖 fs.FileInfo 接口，因此实际上是三个接口。

### **3.1.1. fs.FS**

该接口提供了对层级文件系统的访问。一个文件系统的最低要求是必须实现 fs.FS 接口，但一般还会实现额外的接口，比如 ReadFileFS，该接口在后文讲解。

type FS interface {

// Open opens the named file.

//

// When Open returns an error, it should be of type \*PathError

// with the Op field set to "open", the Path field set to name,

// and the Err field describing the problem.

//

// Open should reject attempts to open names that do not satisfy

// ValidPath(name), returning a \*PathError with Err set to

// ErrInvalid or ErrNotExist.

Open(name string) (File, error)

}

该接口只有一个方法，即打开一个命名文件，该方法的实现要求如下：

* 如果 Open 方法出错，应该返回 \*PathError 类型的错误，该类型定义如下：

type PathError struct {

Op string

Path string

Err error

}

返回该类型错误时，Op 字段设置为 "open"，Path 字段设置为文件名，而 Err 字段描述错误原因。

注：在 os 那小节提到过该类型，Go 1.16 后，os.PathError 只是 fs.PathError 的别名。

type PathError = fs.PathError

* 对于指定的文件名，需要满足 ValidPath(name) 函数，如果不满足，则返回 \*PathError 的 Err 为 fs.ErrInvalid 或 fs.ErrNotExist 的错误。

func ValidPath(name string) bool

传递给该函数的 name 应该是一个非根，且是 / 分隔的，例如 x/y/z。除了只包含 .，其他情况不能有 . 和 ..。

因为 Open 方法返回一个 fs.File 接口类型，因此一个文件系统只实现 fs.FS 还不够，需要同时实现 fs.File 接口。

### **3.1.2. fs.File**

该接口提供对单个文件的访问。File 接口是文件的最低实现要求。一个文件可以实现其他接口，例如fs.ReadDirFile，fs.ReaderAt 或 fs.Seeker，以提供额外或优化的功能。

type File interface {

Stat() (FileInfo, error)

Read([]byte) (int, error)

Close() error

}

通过 fs.FS 接口的 Open 打开文件后，通过 fs.File 接口的 Read 方法进行读操作，这个方法和 io.Reader 接口的 Read 方法签名一样。

对操作系统有所了解的读者应该知晓（特别是 Unix 系统），目录也是文件，只是特殊的文件。因此，在遍历文件目录树时，我们通常需要判断文件是什么类型，也可能需要获取文件的一些元数据信息，比如文件名、大小、修改时间等，而这就是 Stat 方法的功能。该方法会返回一个 FileInfo 类型，它也是一个接口。这就是文件系统需要实现的第三接口，稍后讲解。

在 Go 中，你应该始终记住，打开文件，进行操作后，记得关闭文件，否则会泄露文件描述符。所以，fs.File 的第是三个方法就是 Close 方法，它的签名和 io.Closer 是一致的。

### **3.1.3. fs.FileInfo**

该接口描述一个文件的元数据信息，它由 Stat 返回。为了方便，在 io/fs 包有一个 Stat 函数：

func Stat(fsys FS, name string) (FileInfo, error)

该函数接受任意的 FS 文件系统和该系统下的任意一个文件。如果 fsys 实现了 StatFS，则直接通过 StatFS 的 Stat 方法获取 FileInfo，否则需要 Open 文件，然后调用 File 的 Stat 方法来获取 FileInfo。关于 fs.StatFS 接口后文讲解。

本节开头提到了，Go1.16 开始，os 包中和文件系统相关的类型移到 io/fs 包中了，fs.FileInfo 就是其中之一。因为在 os 中已经讲过该接口了，此处不再赘述。

## **3.2. 实现一个文件系统**

介绍完这三个核心接口，我们尝试实现一个文件系统。这是一个基于内存的文件系统，这个实现相对比较简陋。

### **3.2.1. 实现 fs.File 和 fs.FileInfo**

实现文件系统先实现这两个接口。我们通过一个类型来实现：

type file struct {

name string

content \*bytes.Buffer

modTime time.Time

closed bool

}

func (f \*file) Read(p []byte) (int, error) {

if f.closed {

return 0, errors.New("file closed")

}

return f.content.Read(p)

}

func (f \*file) Stat() (fs.FileInfo, error) {

if f.closed {

return nil, errors.New("file closed")

}

return f, nil

}

// Close 关闭文件，可以调用多次。

func (f \*file) Close() error {

f.closed = true

return nil

}

// 实现 fs.FileInfo

func (f \*file) Name() string {

return f.name

}

func (f \*file) Size() int64 {

return int64(f.content.Len())

}

func (f \*file) Mode() fs.FileMode {

// 固定为 0444

return 0444

}

func (f \*file) ModTime() time.Time {

return f.modTime

}

// IsDir 目前未实现目录功能func (f \*file) IsDir() bool {

return false

}

func (f \*file) Sys() interface{} {

return nil

}

* file 同时实现 fs.File 和 fs.FileInfo；
* 文件内容放在 file 的 bytes.Buffer 类型中，它实现了 io.Reader，因此 file 的 Read 可以直接通过它实现；
* 目前是一个简化实现，因此 IsDir 未实现目录功能，只返回 false；

### **3.2.2. 实现 fs.FS**

实现了 fs.File，通过它可以实现 fs.FS：

type FS struct {

files map[string]\*file

}

func NewFS() \*FS {

return &FS{

files: make(map[string]\*file),

}

}

func (fsys \*FS) Open(name string) (fs.File, error) {

if !fs.ValidPath(name) {

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: fs.ErrInvalid,

}

}

if f, ok := fsys.files[name]; !ok {

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: fs.ErrNotExist,

}

} else {

return f, nil

}

}

* FS 类型中的 files 存放所有的文件；
* 按照前面 Open 方法的实现要求，先通过 ValidPath 函数进行校验，接着通过 name 查找 file；

细心的读者应该会发现，io/fs 并没有提供 Write 相关的功能，那我们读什么呢？为此，我们实现一个 Write 的功能。

func (fsys \*FS) WriteFile(name, content string) error {

if !fs.ValidPath(name) {

return &fs.PathError{

Op: "write",

Path: name,

Err: fs.ErrInvalid,

}

}

f := &file{

name: name,

content: bytes.NewBufferString(content),

modTime: time.Now(),

}

fsys.files[name] = f

return nil

}

WriteFile 方法就是生成一个 file 然后存入 files 中。

### **3.2.3. 验证**

一个基于内存的文件系统已经实现完成，接下来需要验证下。

func TestMemFS(t \*testing.T) {

name := "x/y/name.txt"

content := "This is polarisxu, welcome."

memFS := memfs.NewFS()

err := memFS.WriteFile(name, content)

if err != nil {

t.Fatal(err)

}

f, err := memFS.Open(name)

if err != nil {

t.Fatal(err)

}

defer f.Close()

fi, err := f.Stat()

if err != nil {

t.Fatal(err)

}

t.Log(fi.Name(), fi.Size(), fi.ModTime())

var result = make([]byte, int(fi.Size()))

n, err := f.Read(result)

if err != nil {

t.Fatal(err)

}

if string(result[:n]) != content {

t.Errorf("expect: %s, actual: %s", content, result[:n])

}

}

如果测试通过，说明基于内存的简单文件系统已经完成。至于缺失的功能，本节后面再完善。

## **3.3. 增强型接口**

上面实现的内存文件系统中，目录功能是有问题的，比如我们没法遍历整个文件系统。要实现一个更完整的文件系统，需要实现 io/fs 包中的其他接口。

### **3.3.1. fs.DirEntry 和相关接口**

在文件系统中，一个目录下可能会有子目录或文件，这称为 entry，在 io/fs 包中用 DirEntry 接口表示：

type DirEntry interface {

// Name returns the name of the file (or subdirectory) described by the entry.

// This name is only the final element of the path (the base name), not the entire path.

// For example, Name would return "hello.go" not "/home/gopher/hello.go".

Name() string

// IsDir reports whether the entry describes a directory.

IsDir() bool

// Type returns the type bits for the entry.

// The type bits are a subset of the usual FileMode bits, those returned by the FileMode.Type method.

Type() FileMode

// Info returns the FileInfo for the file or subdirectory described by the entry.

// The returned FileInfo may be from the time of the original directory read

// or from the time of the call to Info. If the file has been removed or renamed

// since the directory read, Info may return an error satisfying errors.Is(err, ErrNotExist).

// If the entry denotes a symbolic link, Info reports the information about the link itself,

// not the link's target.

Info() (FileInfo, error)

}

* Name() 方法和 FileInfo 接口的 Name() 方法类似，代表的是 base name，而我们上面实现的文件系统没有处理这一点；
* Type() 方法返回一个 fs.FileMode，表示 entry 的位类型，关于 FileMode 的详细信息在 os 包中有讲解；
* Info() 方法和 Stat 有点类似，获取元数据信息；如果 entry 是软链接，Info() 返回的 FileInfo 是链接本身的信息，而不是目标文件；

为了方便遍历文件系统（目录），io/fs 包提供了 ReadDir 函数，用来获取某个目录下的所有目录项：

func ReadDir(fsys FS, name string) ([]DirEntry, error)

对于这个函数的实现，如果第一个参数实现了 fs.ReadDirFS 接口，直接调用该接口的 ReadDir 方法：

type ReadDirFS interface {

FS

// ReadDir reads the named directory

// and returns a list of directory entries sorted by filename.

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

}

否则看是否实现了 fs.ReadDirFile 接口，没实现则报错；否则调用该接口的 ReadDir 方法：

type ReadDirFile interface {

File

// ReadDir reads the contents of the directory and returns

// a slice of up to n DirEntry values in directory order.

// Subsequent calls on the same file will yield further DirEntry values.

//

// If n > 0, ReadDir returns at most n DirEntry structures.

// In this case, if ReadDir returns an empty slice, it will return

// a non-nil error explaining why.

// At the end of a directory, the error is io.EOF.

//

// If n <= 0, ReadDir returns all the DirEntry values from the directory

// in a single slice. In this case, if ReadDir succeeds (reads all the way

// to the end of the directory), it returns the slice and a nil error.

// If it encounters an error before the end of the directory,

// ReadDir returns the DirEntry list read until that point and a non-nil error.

ReadDir(n int) ([]DirEntry, error)

}

这个接口的 ReadDir 比 ReadDirFS 复杂多了，但 ReadDirFS 的 ReadDir 必须自己对 entry 进行排序。此外，如果目录下内容特别多，ReadDirFile 接口会更适合，它可以分段读取。而且目录应该实现 ReadDirFile 接口。

## **3.4. 其他 fs.FS 相关的接口**

在讲解 fs.FS 接口时提到还有其他接口，用于增强 fs.FS，即嵌入了 fs.FS 接口，除了已经介绍的 ReadDirFS 接口，还有如下接口。

### **3.4.1. fs.ReadFileFS**

该接口的定义如下：

type ReadFileFS interface {

FS

// ReadFile reads the named file and returns its contents.

// A successful call returns a nil error, not io.EOF.

// (Because ReadFile reads the whole file, the expected EOF

// from the final Read is not treated as an error to be reported.)

ReadFile(name string) ([]byte, error)

}

也就是说这是一个支持 ReadFile 的文件系统，如果一个文件系统实现了该接口，则 fs.ReadFile 函数会先直接使用该接口的 ReadFile 方法来实现：

func ReadFile(fsys FS, name string) ([]byte, error)

如果没实现该接口，则通过 fs.FS 的 Open 方法获取 fs.File 类型，然后调用 fs.File 的 Read 方法来实现。有兴趣可以查看 fs.ReadFile 函数的实现。

### **3.4.2. fs.StatFS**

该接口的定义如下：

type StatFS interface {

FS

// Stat returns a FileInfo describing the file.

// If there is an error, it should be of type \*PathError.

Stat(name string) (FileInfo, error)

}

如果一个文件系统支持 Stat 功能，则 fs.Stat 函数会优先使用该文件系统的 Stat 方法，否则通过 fs.FS 的 Open 方法获取 fs.File 类型，然后调用 fs.File 的 Stat 方法来实现。

### **3.4.3. fs.GlobFS**

该接口的定义如下：

type GlobFS interface {

FS

// Glob returns the names of all files matching pattern,

// providing an implementation of the top-level

// Glob function.

Glob(pattern string) ([]string, error)

}

类似的，实现了该接口，表示文件系统支持 Glob 方法。对应的，io/fs 提供了 Glob 函数：

func Glob(fsys FS, pattern string) (matches []string, err error)

* 这是用于文件模式匹配的；
* 语法和 path.Match 相同；
* 模式（pattern）可以描述层级，比如：/usr/\*/bin/ed；
* 该函数会忽略文件系统错误，比如 IO 错误；唯一的错误是模式语法错误；

和其他 fs.FS 相关接口对应的函数一样，Glob 函数内部实现优先调用 fs.GlobFS 接口，如果没实现该接口，则使用 ReadDir 遍历目录树来查找匹配的目标。

### **3.4.4. fs.SubFS**

该接口的定义如下：

type SubFS interface {

FS

// Sub returns an FS corresponding to the subtree rooted at dir.

Sub(dir string) (FS, error)

}

这个接口的作用主要是让一个文件系统支持定义子文件系统。io/fs 包也提供了一个相应的函数 Sub：

func Sub(fsys FS, dir string) (FS, error)

通过该函数可以获得一个子文件系统，该子文件系统的根由第二个参数 dir 指定。

类似的，该函数的实现会优先判断 fsys 是否实现了 fs.SubFS 接口，以便调用其 Sub 方法。如果未实现，同时 dir 是 .，则原样返回 fsys，否则返回一个新实现的 fs.FS。

不过有一点需要注意，对于 os 实现的 fs.FS 文件系统（磁盘文件系统），Sub 并不能提到 chroot 的进制，它不会限制子文件系统根之外的操作，典型的，子文件系统内部的文件软连到根之外，Sub 得到的子文件系统不会阻止这种行为。

查看 fs.Sub 函数的源码可以发现，如果 fsys 没有实现 fs.SubFS，Sub 函数返回的 FS 实现了不少 FS 相关接口。

### **3.4.5. 设计思想**

上面啰啰嗦嗦讲了好几个 fs.FS 相关接口，其中目的之一是希望理解其设计思想。

io/fs 包中和 fs.FS 相关的接口如下：

* fs.ReadDirFS
* fs.ReadFileFS
* fs.StatFS
* fs.SubFS
* fs.GlobFS

Go 以简单著称，大道至简。Go 强调定义小接口。fs.FS 接口只有一个方法：Open，其他 fs.FS 相关接口都内嵌了 fs.FS 接口，以此来扩展文件系统的功能。同时 io/fs 包辅以相关便捷函数（比如 Stat、Sub、Glob 等），达到操作 fs.FS 的目的。

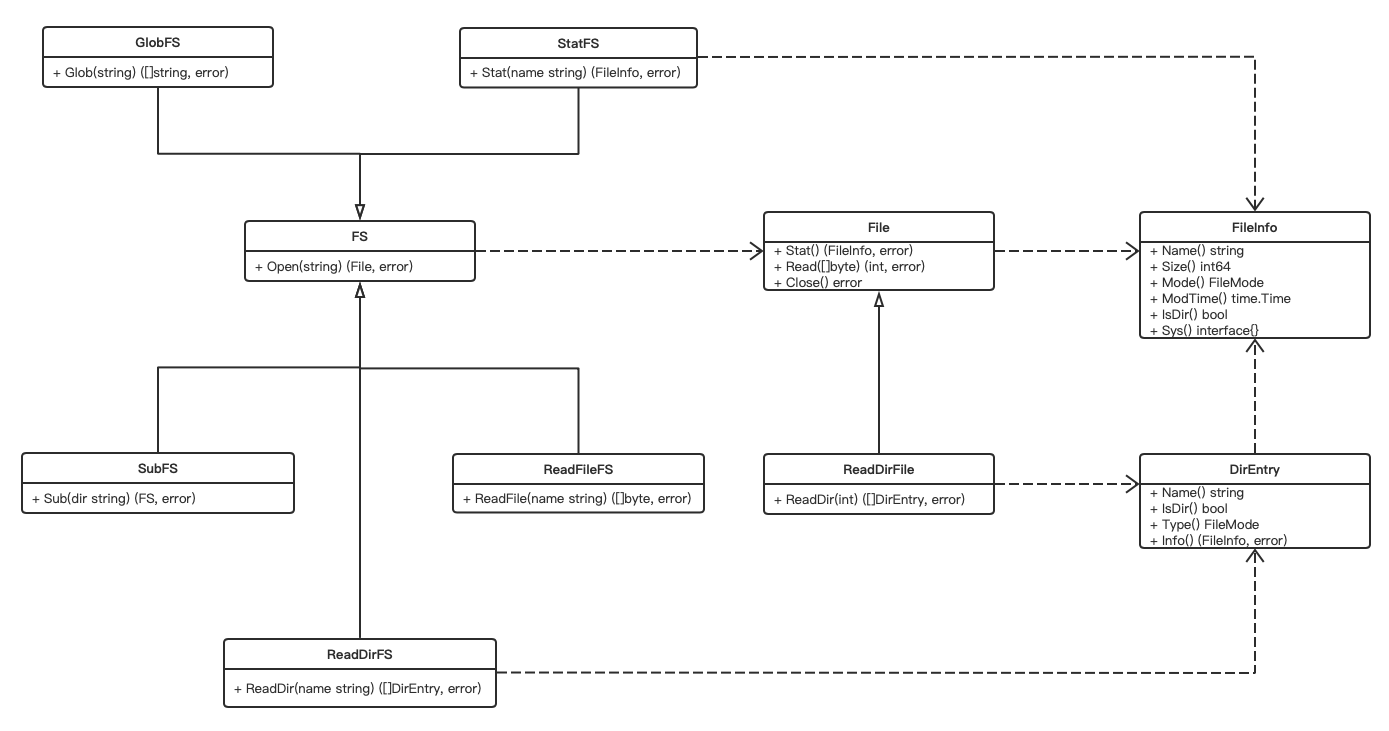
## **3.5. 完善内存文件系统**

是时候完善我们上面实现的内存文件系统了。

前面只是实现了文件的写入、读取，并没有实现文件系统该有的目录树。现在补充完善这部分内容，实现一个较完整的内存文件系统。具体看代码前，看看如何设计。

### **3.5.1. 如何设计**

先通过一个类图表示 io/fs 包相关接口的关系。



要基于这些接口实现一个文件系统，我们需要先了解一些文件系统相关的知识（前面有提到，这里总结下）。

* 文件系统是一个树形结构，有一个根目录；
* 一个目录下的目录项，可以是文件或子目录；
* 一切皆文件，所以目录也是文件；（虽然如此，但两者还是有不小区别，因此实现时不一定适合使用嵌入）

所以，我们在实现 fs.FS 接口时，定义的类型 FS 有一个根目录字段：

type FS struct {

rootDir \*dir

}

#### **文件的实现**

从上面类图可以看出，一个文件需要实现 fs.File 接口，同时因为该接口依赖 fs.FileInfo 接口，我们可以选择用一个单独的类型实现 fs.FileInfo 接口，也可以直接用这个文件类型（file）实现该接口，内存文件系统直接使用文件类型实现了 fs.FileInfo 接口。此外，一个文件还是其所在目录的目录项，因此还需要实现 fs.DirEntry 接口。因此内存文件系统的 file 类型实现了以下接口：

* fs.File
* fs.FileInfo
* fs.DirEntry

具体如何实现这些接口，需要先思考一个问题：文件内容用什么表示？因为是内存文件系统，因此一切都在内存中。文件内容本质上是字节数组，但因为要实现 fs.File 接口，这其中关键的是 Read 方法，它的签名和 io.Reader 接口的 Read 方法是一样的，因此在 file 类型中，我们用一个 bytes.Buffer 字段来存放文件内容。

至于其他接口的实现相对较简单，这里不赘述。值得一提的是，因为 file 类型实现了 fs.FileInfo 接口，所以在实现 Stat 方法时，直接返回 file 的实例即可。

#### **目录的实现**

对于目录，我们用类型 dir 表示，它首先是其所在目录的目录项，因此需要实现 fs.DirEntry 接口；其次目录也是文件，因此它需要实现 fs.File 接口。同时，读取目录的内容，即读取其目录项，不应该通过 Read 读取，而 fs.ReadDirFile 接口是用来读目录的，因此 dir 应该实现它。同样的，因为 fs.DirEntry 和 fs.File 都依赖 fs.FileInfo 接口，跟 file 一样，我们不单独实现，而是让 dir 直接实现它。因此内存文件系统的 dir 类型实现了以下接口：

* fs.DirEntry
* fs.File
* fs.ReadDirFile
* fs.FileInfo

因为目录涉及到有目录项，构成了一个树形结构。这里使用一个 map 来存放所有的目录项，key 是目录项的名称，value 是目录项的实例。

// dir 代表一个目录

type dir struct {

name string

modTime time.Time

// 存放该目录下的子项，value 可能是 \*dir 或 \*file

children map[string]fs.DirEntry

}

因为 Read 对于目录来说没有实际价值，因此它的实现返回错误即可。dir 的难点在于实现 fs.ReadDirFile 接口中的 ReadDir 方法：给定一个目录，该方法需要返回该目录下的所有目录项。而且，根据 fs.ReadDirFile 中 ReadDir 方法的实现要求，它应该支持分步读取目录项。所以，在 dir 类型中增加一个字段：idx，用来表示当前读取到什么位置的目录项了。具体实现代码见后文。

#### **fs.FS 接口的实现**

对于内存文件系统，如何实现 Open 方法呢？我们需要根据参数 name 在文件系统的目录树中找到该文件所在位置。因此，我们将该文件用 / 分隔，从左到右，一部分一部分，从文件系统的根开始，在目录树中查找，直到找到对应的文件，然后返回该文件。如果没找到，返回错误。

具体来说，在遍历文件系统目录树时，如果某个目录项是文件，且是 name 的最后一部分，表示找到了该文件；如果某个目录项是目录，则递归遍历它的目录项。

#### **创建目录和文件的实现**

io/fs 没有定义创建目录和文件的接口，从这个维度看，io/fs 定义的文件系统是一个只读文件系统。但实际的文件系统，必然要有写入的接口。因此我们还需要实现创建目录和创建文件（写入内容）的功能。

先看创建目录的实现。

创建目录，实际上是构建一个层级关系。方法签名如下：

func (fsys \*FS) MkdirAll(path string) error

根据传入的 path，比如 x/y/z，能够创建对应的目录结构。因此我们将 path 通过 / 分隔，从左到右，一步步从文件系统的根开始在对应的层级创建目录。创建时，需要判断是否已经存在对应的目录。关键代码如下：

cur := fsys.rootDir

parts := strings.Split(path, "/")for \_, part := range parts {

child := cur.children[part]

if child == nil {

childDir := &dir{

name: part,

modTime: time.Now(),

children: make(map[string]fs.DirEntry),

}

cur.children[part] = childDir

cur = childDir

} else {

childDir, ok := child.(\*dir)

if !ok {

return fmt.Errorf("%s is not directory", part)

}

cur = childDir

}

}

文件的创键和内容写入通过 WriteFile 方法实现。签名如下：

func (fsys \*FS) WriteFile(name, content string) error

在非完善版本中，粗暴的直接将传递的文件名（包括路径）和 file 实例关联，没有处理目录层级关系。因此，这里的实现的关键是要找到该文件（name 对应）的目录 dir 实例。和上面创建目录的思路类似，一步步处理。

// getDir 通过一个路径获取其 dir 类型实例func (fsys \*FS) getDir(path string) (\*dir, error) {

parts := strings.Split(path, "/")

cur := fsys.rootDir

for \_, part := range parts {

child := cur.children[part]

if child == nil {

return nil, fmt.Errorf("%s is not exists", path)

}

childDir, ok := child.(\*dir)

if !ok {

return nil, fmt.Errorf("%s is not directory", path)

}

cur = childDir

}

return cur, nil

}

得到了文件应该放置的目录（dir）后，就可以构建一个 file 实例，并将该实例放置到其目录的目录项中。

filename := filepath.Base(name)

dir.children[filename] = &file{

name: filename,

content: bytes.NewBufferString(content),

modTime: time.Now(),

}

### **3.5.2. 完整的实现代码**

以下是 dir 类型的实现，代表一个目录，注意注释。

// dir 代表一个目录type dir struct {

name string

modTime time.Time

// 存放该目录下的子项，value 可能是 \*dir 或 \*file

children map[string]fs.DirEntry

// ReadDir 遍历用

idx int

}

// dir 虽然是一个目录，但根据一切皆文件的思想，目录也是文件，因此需要实现 fs.File 接口// 这样，fs.FS 的 Open 方法可以对目录起作用。

func (d \*dir) Read(p []byte) (int, error) {

return 0, &fs.PathError{

Op: "read",

Path: d.name,

Err: errors.New("is directory"),

}

}

func (d \*dir) Stat() (fs.FileInfo, error) {

return d, nil

}

func (d \*dir) Close() error {

return nil

}

// ReadDir 实现 fs.ReadDirFile 接口，方便遍历目录func (d \*dir) ReadDir(n int) ([]fs.DirEntry, error) {

names := make([]string, 0, len(d.children))

for name := range d.children {

names = append(names, name)

}

totalEntry := len(names)

if n <= 0 {

n = totalEntry

}

dirEntries := make([]fs.DirEntry, 0, n)

for i := d.idx; i < n && i < totalEntry; i++ {

name := names[i]

child := d.children[name]

f, isFile := child.(\*file)

if isFile {

dirEntries = append(dirEntries, f)

} else {

dirEntry := child.(\*dir)

dirEntries = append(dirEntries, dirEntry)

}

d.idx = i

}

return dirEntries, nil

}

// 因为 fs.Stat 对目录也是有效的，因此 dir 需要实现 fs.FileInfo 接口

func (d \*dir) Name() string {

return d.name

}

func (d \*dir) Size() int64 {

return 0

}

func (d \*dir) Mode() fs.FileMode {

return fs.ModeDir | 0444

}

func (d \*dir) ModTime() time.Time {

return d.modTime

}

func (d \*dir) IsDir() bool {

return true

}

func (d \*dir) Sys() interface{} {

return nil

}

// 因为 dir 是一个目录项，因此需要实现 fs.DirEntry 接口

func (d \*dir) Type() fs.FileMode {

return d.Mode()

}

func (d \*dir) Info() (fs.FileInfo, error) {

return d, nil

}

接着是 file 的实现，代表一个文件，注意注释。

// file 代表一个文件type file struct {

name string

// 存放文件内容

content \*bytes.Buffer

modTime time.Time

closed bool

}

// 实现 fs.File 接口

func (f \*file) Read(p []byte) (int, error) {

if f.closed {

return 0, errors.New("file closed")

}

return f.content.Read(p)

}

func (f \*file) Stat() (fs.FileInfo, error) {

if f.closed {

return nil, errors.New("file closed")

}

return f, nil

}

// Close 关闭文件，可以调用多次。func (f \*file) Close() error {

f.closed = true

return nil

}

// 实现 fs.FileInfo 接口

func (f \*file) Name() string {

return f.name

}

func (f \*file) Size() int64 {

return int64(f.content.Len())

}

func (f \*file) Mode() fs.FileMode {

// 固定为 0444

return 0444

}

func (f \*file) ModTime() time.Time {

return f.modTime

}

func (f \*file) IsDir() bool {

return false

}

func (f \*file) Sys() interface{} {

return nil

}

// 文件也是某个目录下的目录项，因此需要实现 fs.DirEntry 接口

func (f \*file) Type() fs.FileMode {

return f.Mode()

}

func (f \*file) Info() (fs.FileInfo, error) {

return f, nil

}

有了目录（dir）和文件（file），看 fs.FS 的实现。

// FS 是 fs.FS 的内存文件系统实现type FS struct {

rootDir \*dir

}

// NewFS 创建一个内存文件系统的实例func NewFS() \*FS {

return &FS{

rootDir: &dir{

children: make(map[string]fs.DirEntry),

},

}

}

// Open 实现 fs.FS 的 Open 方法func (fsys \*FS) Open(name string) (fs.File, error) {

// 1、校验 name

if !fs.ValidPath(name) {

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: fs.ErrInvalid,

}

}

// 2、根目录处理

if name == "." || name == "" {

// 重置目录的遍历

fsys.rootDir.idx = 0

return fsys.rootDir, nil

}

// 3、根据 name 在目录树中进行查找

cur := fsys.rootDir

parts := strings.Split(name, "/")

for i, part := range parts {

// 不存在返回错误

child := cur.children[part]

if child == nil {

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: fs.ErrNotExist,

}

}

// 是否是文件

f, ok := child.(\*file)

if ok {

// 文件名是最后一项

if i == len(parts)-1 {

return f, nil

}

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: fs.ErrNotExist,

}

}

// 是否是目录

d, ok := child.(\*dir)

if !ok {

return nil, &fs.PathError{

Op: "open",

Path: name,

Err: errors.New("not a directory"),

}

}

// 重置，避免遍历问题

d.idx = 0

cur = d

}

return cur, nil

}

// MkdirAll 这不是 io/fs 的要求，但一个文件系统目录树需要可以构建// 这个方法就是用来创建目录func (fsys \*FS) MkdirAll(path string) error {

if !fs.ValidPath(path) {

return errors.New("Invalid path")

}

if path == "." {

return nil

}

cur := fsys.rootDir

parts := strings.Split(path, "/")

for \_, part := range parts {

child := cur.children[part]

if child == nil {

childDir := &dir{

name: part,

modTime: time.Now(),

children: make(map[string]fs.DirEntry),

}

cur.children[part] = childDir

cur = childDir

} else {

childDir, ok := child.(\*dir)

if !ok {

return fmt.Errorf("%s is not directory", part)

}

cur = childDir

}

}

return nil

}

// WriteFile 也不是 io/fs 的要求，和 MkdirAll 类似，文件内容也需要有接口写入func (fsys \*FS) WriteFile(name, content string) error {

if !fs.ValidPath(name) {

return &fs.PathError{

Op: "write",

Path: name,

Err: fs.ErrInvalid,

}

}

var err error

dir := fsys.rootDir

path := filepath.Dir(name)

if path != "." {

dir, err = fsys.getDir(path)

if err != nil {

return err

}

}

filename := filepath.Base(name)

dir.children[filename] = &file{

name: filename,

content: bytes.NewBufferString(content),

modTime: time.Now(),

}

return nil

}

// getDir 通过一个路径获取其 dir 类型实例func (fsys \*FS) getDir(path string) (\*dir, error) {

parts := strings.Split(path, "/")

cur := fsys.rootDir

for \_, part := range parts {

child := cur.children[part]

if child == nil {

return nil, fmt.Errorf("%s is not exists", path)

}

childDir, ok := child.(\*dir)

if !ok {

return nil, fmt.Errorf("%s is not directory", path)

}

cur = childDir

}

return cur, nil

}

为了完整性，我把所有代码列出来了，关键的地方加上了注释。

## **1.6. 验证正确性并学习 fs.WalkDir**

用心的读者可能会发现，io/fs 包还有一个类型和函数没有介绍，那就是 fs.WalkDir 函数和 WalkDirFunc 类型。它们是遍历目录用的。这里通过验证上面内存文件系统的正确性来学习它们。

首先，我们使用 MkdirAll 和 WriteFile 创建如下的目录树：

├── a

│ ├── b

│ │ └── z

├── x

│ └── y

│ │ └── z

│ └── name.txt

即执行如下代码：

memFS := memfs.NewFS()

memFS.MkdirAll("a/b/z")

memFS.MkdirAll("x/y/z")

memFS.WriteFile("x/name.txt", "This is polarisxu, welcome.")

基于这个目录树，我们该怎么遍历它？

### **1.6.1. 自己实现遍历方法**

遍历目录树也是一个面试常考的基础题目。熟悉的朋友应该知道，这需要用到递归。基于上面的内存文件系统 API，我们实现遍历目录树。

通过 io/fs 包的 ReadDir 函数读取目录下所有目录项，然后遍历这些目录项，如果某个目录项是目录，递归处理它。

func walk(fsys fs.FS, parent, base string) error {

dirEntries, err := fs.ReadDir(fsys, filepath.Join(parent, base))

if err != nil {

return err

}

for \_, dirEntry := range dirEntries {

name := dirEntry.Name()

fmt.Println(name)

if dirEntry.IsDir() {

err = walk(fsys, filepath.Join(parent, base), name)

}

}

return err

}

然后用根目录调用它：

walk(memFS, "", ".")

### **1.6.2. 使用 fs.WalkDir 实现**

其实标准库为我们实现了这样的功能，通过它提供的方法可以很容易的实现遍历。

fs.WalkDir(memFS, ".", func(path string, d fs.DirEntry, err error) error {

fmt.Pritnln(path)

return nil

})

是不是不要太简单？

关于 fs.WalkDir 和 fs.WalkDirFunc 有一大段文字说明，介绍其中的一些细节。比如在回调函数中，如果返回 fs.SkipDir，则会停止该目录的遍历。这里细说了。

## **1.7. 小结**

io/fs 包基本上是在 os 包的基础上抽象出来的。之所以抽象，是因为 Go1.16 的 embed 功能，它需要文件系统，但又不同于 os 的文件系统。所以做了这个抽象。

基于 io/fs 包的接口，标准库不少地方做了改动，以支持 fs.FS 接口。此外还有第三方实现了它的文件系统：

* [https://github.com/jhchabran/gistfs](https://github.com/jhchabran/gistfs" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter06/_blank)：用于读取 GitHub gists 的文件；
* [https://github.com/benbjohnson/hashfs](https://github.com/benbjohnson/hashfs" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter06/_blank)：hash 文件，以便 HTTP Cache；
* [https://github.com/psanford/memfs](https://github.com/psanford/memfs" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter06/_blank)：内存文件系统的实现；本小节文件系统的实现参考了它。