

44 | 使用os包中的API（上）

2018-11-21 郝林

Go语言核心36讲

[进入课程 >](#)



讲述：黄洲君

时长 10:02 大小 4.60M



我们今天要讲的是os代码包中的 API。这个代码包可以让我们拥有操控计算机操作系统的能力。

前导内容：os 包中的 API

这个代码包提供的都是平台不相关的 API。那么说，什么叫平台不相关的 API 呢？

它的意思是：这些 API 基于（或者说抽象自）操作系统，为我们使用操作系统的功能提供高层次的支持，但是，它们并不依赖于具体的操作系统。

不论是 Linux、macOS、Windows，还是 FreeBSD、OpenBSD、Plan9，os代码包都可以为之提供统一的使用接口。这使得我们可以用同样的方式，来操纵不同的操作系统，并得

到相似的结果。

`os`包中的 API 主要可以帮助我们使用操作系统中的文件系统、权限系统、环境变量、系统进程以及系统信号。

其中，操纵文件系统的 API 最为丰富。我们不但可以利用这些 API 创建和删除文件以及目录，还可以获取到它们的各种信息、修改它们的内容、改变它们的访问权限，等等。

说到这里，就不得不提及一个非常常用的数据类型：`os.File`。

从字面上来看，`os.File`类型代表了操作系统中的文件。但实际上，它可以代表的远不止于此。或许你已经知道，对于类 Unix 的操作系统（包括 Linux、macOS、FreeBSD 等），其中的一切都可以被看做是文件。

除了文本文件、二进制文件、压缩文件、目录这些常见的形式之外，还有符号链接、各种物理设备（包括内置或外接的面向块或者字符的设备）、命名管道，以及套接字（也就是 socket），等等。

因此，可以说，我们能够利用`os.File`类型操纵的东西太多了。不过，为了聚焦于`os.File`本身，同时也为了让本文讲述的内容更加通用，我们在这里主要把`os.File`类型应用于常规的文件。

下面这个问题，就是以`os.File`类型代表的最基本内容入手。**我们今天的问题是：**
`os.File`类型都实现了哪些`io`包中的接口？

这道题的**典型回答**是这样的。

`os.File`类型拥有的都是指针方法，所以除了空接口之外，它本身没有实现任何接口。而它的指针类型则实现了很多`io`代码包中的接口。

首先，对于`io`包中最核心的 3 个简单接口`io.Reader`、`io.Writer`和`io.Closer`，
`*os.File`类型都实现了它们。

其次，该类型还实现了另外的 3 个简单接口，即：`io.ReaderAt`、`io.Seeker`和 `io.WriterAt`。

正是因为`*os.File`类型实现了这些简单接口，所以它也顺便实现了`io`包的 9 个扩展接口中的 7 个。

然而，由于它并没有实现简单接口`io.ByteReader`和`io.RuneReader`，所以它没有实现分别作为这两者的扩展接口的`io.ByteScanner`和`io.RuneScanner`。

总之，`os.File`类型及其指针类型的值，不但可以通过各种方式读取和写入某个文件中的内容，还可以寻找并设定下一次读取或写入时的起始索引位置，另外还可以随时对文件进行关闭。

但是，它们并不能专门地读取文件中的下一个字节，或者下一个 Unicode 字符，也不能进行任何的读回退操作。

不过，单独读取下一个字节或字符的功能也可以通过其他方式来实现，比如，调用它的 `Read`方法并传入适当的参数值就可以做到这一点。

问题解析

这个问题其实在间接地问“`os.File`类型能够以何种方式操作文件？”我在前面的典型回答中也给出了简要的答案。

在我进一步地说明一些细节之前，我们先来看看，怎样才能获得一个`os.File`类型的指针值（以下简称`File`值）。

在`os`包中，有这样几个函数，即：`Create`、`NewFile`、`Open`和`OpenFile`。

`os.Create`函数用于根据给定的路径创建一个新的文件。它会返回一个`File`值和一个错误值。我们可以在该函数返回的`File`值之上，对相应的文件进行读操作和写操作。

不但如此，我们使用这个函数创建的文件，对于操作系统中的所有用户来说，都是可以读和写的。

换句话说，一旦这样的文件被创建出来，任何能够登录其所属的操作系统的用户，都可以在任意时刻读取该文件中的内容，或者向该文件写入内容。

注意，如果在我们给予`os.Create`函数的路径之上，已经存在了一个文件，那么该函数会先清空现有文件中的全部内容，然后再把它作为第一个结果值返回。

另外，`os.Create`函数是有可能返回非`nil`的错误值的。


比如，如果我们给定的路径上的某一级父目录并不存在，那么该函数就会返回一个`*os.PathError`类型的错误值，以表示“不存在的文件或目录”。

再来看`os.NewFile`函数。该函数在被调用的时候，需要接受一个代表文件描述符的、`uintptr`类型的值，以及一个用于表示文件名的字符串值。

如果我们给定的文件描述符并不是有效的，那么这个函数将会返回`nil`，否则，它将会返回一个代表了相应文件的`File`值。


注意，不要被这个函数的名称误导了，它的功能并不是创建一个新的文件，而是依据一个已经存在的文件的描述符，来新建一个包装了该文件的`File`值。

例如，我们可以像这样拿到一个包装了标准错误输出的`File`值：

 复制代码

```
1 file3 := os.NewFile(uintptr(syscall.Stderr), "/dev/stderr")
```

然后，通过这个`File`值向标准错误输出上写入一些内容：

 复制代码

```
1 if file3 != nil {
2     defer file3.Close()
3     file3.WriteString(
4         "The Go language program writes the contents into stderr.\n")
5 }
```

`os.Open`函数会打开一个文件并返回包装了该文件的`File`值。然而，该函数只能以只读模式打开文件。换句话说，我们只能从该函数返回的`File`值中读取内容，而不能向它写入任何内容。

如果我们调用了这个`File`值的任何一个写入方法，那么都将会得到一个表示了“坏的文件描述符”的错误值。实际上，我们刚刚说的只读模式，正是应用在`File`值所持有的文件描述符之上的。

所谓的文件描述符，是由通常很小的非负整数代表的。它一般会由 I/O 相关的系统调用返回，并作为某个文件的一个标识存在。

从操作系统的层面看，针对任何文件的 I/O 操作都需要用到这个文件描述符。只不过，Go 语言中的一些数据类型，为我们隐匿掉了这个描述符，如此一来我们就无需时刻关注和辨别它了（就像`os.File`类型这样）。

实际上，我们在调用前文所述的`os.Create`函数、`os.Open`函数以及将会提到的`os.OpenFile`函数的时候，它们都会执行同一个系统调用，并且在成功之后得到这样一个文件描述符。这个文件描述符将会被储存在它们返回的`File`值中。

`os.File`类型有一个指针方法，名叫`Fd`。它在被调用之后将会返回一个`uintptr`类型的值。这个值就代表了当前的`File`值所持有的那个文件描述符。

不过，在`os`包中，除了`NewFile`函数需要用到它，它也没有什么别的用武之地了。所以，如果你操作的只是常规的文件或者目录，那么就无需特别地在意它了。

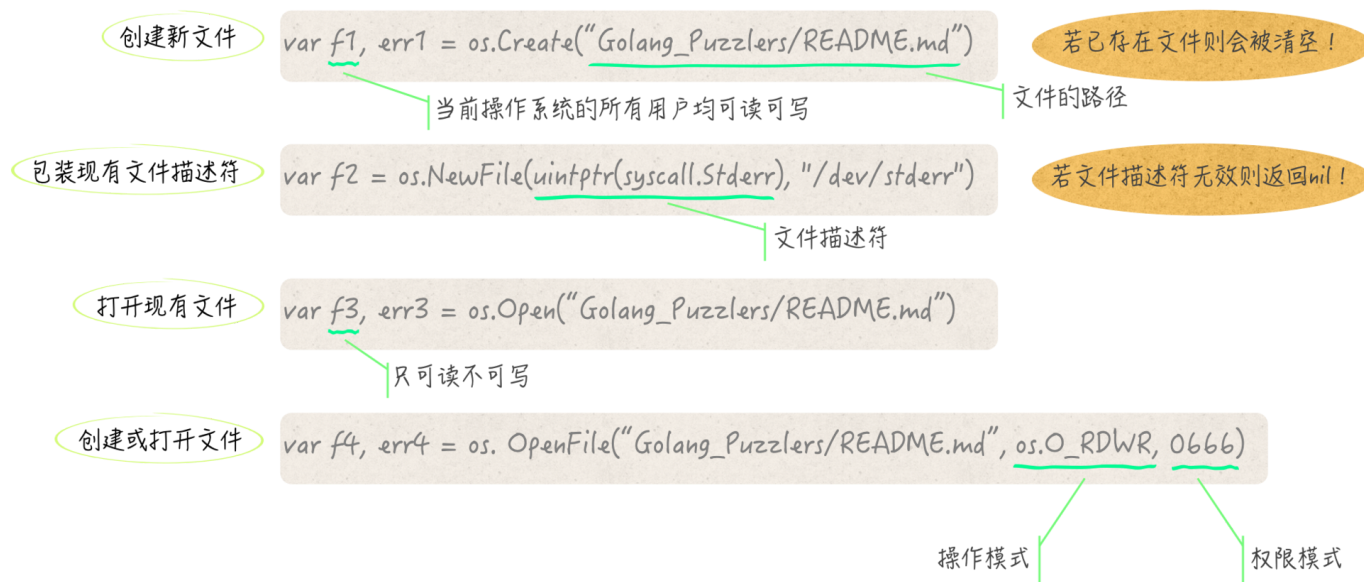
最后，再说一下`os.OpenFile`函数。这个函数其实是`os.Create`函数和`os.Open`函数的底层支持，它最为灵活。

这个函数有 3 个参数，分别名为`name`、`flag`和`perm`。其中的`name`指代的的就是文件的路径。而`flag`参数指的则是需要施加在文件描述符之上的模式，我在前面提到的只读模式就是这里的一个可选项。

在 Go 语言中，这个只读模式由常量`os.O_RDONLY`代表，它是`int`类型的。当然了，这里除了只读模式之外，还有几个别的模式可选，我们稍后再细说。

`os.OpenFile`函数的参数`perm`代表的也是模式，它的类型是`os.FileMode`，此类型是一个基于`uint32`类型的再定义类型。

为了加以区别，我们把参数`flag`指代的模式叫做操作模式，而把参数`perm`指代的模式叫做权限模式。可以这么说，操作模式限定了操作文件的方式，而权限模式则可以控制文件的访问权限。关于权限模式的更多细节我们将在后面讨论。



(获得 `os.File` 类型的指针值的几种方式)

到这里，你需要记住的是，通过`os.File`类型的值，我们不但可以对文件进行读取、写入、关闭等操作，还可以设定下一次读取或写入时的起始索引位置。

此外，`os`包中还有用于创建全新文件的`Create`函数，用于包装现存文件的`NewFile`函数，以及可被用来打开已存在的文件的`Open`函数和`OpenFile`函数。

总结

我们今天讲的是`os`代码包以及其中的程序实体。我们首先讨论了`os`包存在的意义，和它的主要用途。代码包中所包含的 API，都是对操作系统的某方面功能的高层次抽象，这使得我们可以通过它以统一的方式，操纵不同的操作系统，并得到相似的结果。

在这个代码包中，操纵文件系统的 API 最为丰富，最有代表性的就是数据类型`os.File`。`os.File`类型不但可以代表操作系统中的文件，还可以代表很多其他的東西。尤其是在类 Unix 的操作系统中，它几乎可以代表一切可以操纵的软件和硬件。

在下一期的文章中，我会继续讲解 os 包中的 API 的内容。如果你对这部分的知识有什么问题，可以给我留言，感谢你的收听，我们下期再见。

[戳此查看 Go 语言专栏文章配套详细代码。](#)

 极客时间

GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 43 | bufio包中的数据类型（下）

下一篇 45 | 使用os包中的API（下）

精选留言 (3)

 写留言



Walking I...

2018-11-22

 11

最希望老师把net包内极相关的包讲解一下，这部分用的最频繁，但是总有一种似懂非懂的感觉，只是知道是这样用，不知道为什么，对底层知识不清晰，没有一个轮廓



思维





2019-04-17

打卡

展开 ∨



王小勃

2019-03-13

打卡

展开 ∨

