目錄

《Go语言标准库》The Golang Standard Library by Example

Golang标准库。对于程序员而言，标准库与语言本身同样重要，它好比一个百宝箱，能为各种常见的任务提供完美的解决方案。以示例驱动的方式讲解Golang的标准库。

标准库基于最新版本Go。注：目前 Go 标准库文档并没有标识某个 API 基于哪个版本的 Go，将来会加上这部分 issue。

讲解中涉及到特定操作系统时，针对的都是 Linux/amd64。Go 中相关系统调用在 Linux 下，对于同一个系统调用，如果有对应的 `at` 版本，使用的都是 `at` 版本，如 `open` 系统调用使用都是 `openat`。

第一章 输入输出 (Input/Output)

一般的，计算机程序是：输入 (Input) 经过算法处理产生输出 (Output)。各种语言一般都会提供IO库供开发者使用。Go语言也不例外。

Go 语言中，为了方便开发者使用，将 IO 操作封装在了如下几个包中：

- io 为 IO 原语（I/O primitives）提供基本的接口
- io/ioutil 封装一些实用的 I/O 函数
- fmt 实现格式化 I/O，类似 C 语言中的 printf 和 scanf
- bufio 实现带缓冲I/O

本章会详细介绍这些 IO 包提供的函数、类型和方法，同时通过实例讲解这些包的使用方法。

导航

- 目录
- 下一节：io — 基本的 IO 接口

1.1 io — 基本的 IO 接口

io 包为 I/O 原语提供了基本的接口。它主要包装了这些原语的已有实现。

由于这些接口和原语以不同的实现包装了低级操作，因此除非另行通知，否则客户端不应假定它们对于并行执行是安全的。

在 io 包中最重要的是两个接口：Reader 和 Writer 接口。本章所提到的各种 IO 包，都跟这两个接口有关，也就是说，只要实现了这两个接口，它就有了 IO 的功能。

Reader 接口

Reader 接口的定义如下：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> Read 将 len(p) 个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)）以及任何遇到的错误。即使 Read 返回的 n < len(p)，它也会在调用过程中使用 p 的全部作为暂存空间。若一些数据可用但不到 len(p) 个字节，Read 会照例返回可用的数据，而不是等待更多数据。
>
> 当 Read 在成功读取 n > 0 个字节后遇到一个错误或 EOF (end-of-file)，它就会返回读取的字节数。它会从相同的调用中返回（非nil的）错误或从随后的调用中返回错误（同时 n == 0）。一般情况的一个例子就是 Reader 在输入流结束时会返回一个非零的字节数，同时返回的 err 不是 EOF 就是 nil。无论如何，下一个 Read 都应当返回 0, EOF。
>
> 调用者应当总在考虑到错误 err 前处理 n > 0 的字节。这样做可以在读取一些字节，以及允许的 EOF 行为后正确地处理 I/O 错误。

也就是说，当 Read 方法返回错误时，不代表没有读取到任何数据。调用者应该处理返回的任何数据，之后才处理可能的错误。

根据 Go 语言中关于接口和实现了接口的类型的定义（Interface_types），我们知道 Reader 接口的方法集（Method_sets）只包含一个 Read 方法，因此，所有实现了 Read 方法的类型都实现了 io.Reader 接口，也就是说，在所有需要 io.Reader 的地方，可以传递实现了 Read() 方法的类型的实例。

下面，我们通过具体例子来谈谈该接口的用法。

```
func ReadFrom(reader io.Reader, num int) ([]byte, error) {
    p := make([]byte, num)
    n, err := reader.Read(p)
    if n > 0 {
        return p[:n], nil
    }
    return p, err
}
```

ReadFrom 函数将 io.Reader 作为参数，也就是说，ReadFrom 可以从任意的地方读取数据，只要来源实现了 io.Reader 接口。比如，我们可以从标准输入、文件、字符串等读取数据，示例代码如下：

```
// 从标准输入读取
data, err = ReadFrom(os.Stdin, 11)

// 从普通文件读取，其中 file 是 os.File 的实例
data, err = ReadFrom(file, 9)

// 从字符串读取
data, err = ReadFrom(strings.NewReader("from string"), 12)
```

完整的演示例子源码见 code/src/chapter01/io/reader.go

小贴士

io.EOF 变量的定义：`var EOF = errors.New("EOF")`，是 error 类型。根据 reader 接口的说明，在 n > 0 且数据被读完了的情况下，返回的 error 有可能是 EOF 也有可能是 nil。

Writer 接口

Writer 接口的定义如下：

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> Write 将 len(p) 个字节从 p 中写入到基本数据流中。它返回从 p 中被写入的字节数 n（0 <= n <= len(p)）以及任何遇到的引起写入提前停止的错误。若 Write 返回的 n < len(p)，它就必须返回一个 非nil 的错误。

同样的，所有实现了Write方法的类型都实现了 io.Writer 接口。

在上个例子中，我们是自己实现一个函数接收一个 io.Reader 类型的参数。这里，我们通过标准库的例子来学习。

在fmt标准库中，有一组函数：Fprint/Fprintf/Fprintln，它们接收一个 io.Wrtier 类型参数（第一个参数），也就是说它们将数据格式化输出到 io.Writer 中。那么，调用这组函数时，该如何传递这个参数呢？

我们以 fmt.Fprintln 为例，同时看一下 fmt.Println 函数的源码。

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

很显然，fmt.Println会将内容输出到标准输出中。下一节我们将详细介绍fmt包。

关于 io.Writer 的更多说明，可以查看笔者之前写的博文《以io.Writer为例看go中的interface{}》。

实现了 io.Reader 接口或 io.Writer 接口的类型

初学者看到函数参数是一个接口类型，很多时候有些束手无策，不知道该怎么传递参数。还有人问：标准库中有哪些类型实现了 io.Reader 或 io.Writer 接口？

通过本节上面的例子，我们可以知道，os.File 同时实现了这两个接口。我们还看到 os.Stdin/Stdout 这样的代码，它们似乎分别实现了 io.Reader/io.Writer 接口。没错，实际上在 os 包中有这样的代码：

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

也就是说，Stdin/Stdout/Stderr 只是三个特殊的文件（即都是 os.File 的实例），自然也实现了 io.Reader 和 io.Writer。

目前，Go 文档中还没法直接列出实现了某个接口的所有类型。不过，我们可以通过查看标准库文档，列出实现了 io.Reader 或 io.Writer 接口的类型（导出的类型）：（注：godoc 命令支持额外参数 -analysis ，能列出都有哪些类型实现了某个接口，相关参考 `godoc -h` 或 Static analysis features of godoc。另外，我做了一个官网镜像，能查看接口所有的实现类型，地址：http://docs.studygolang.com。

- os.File 同时实现了 io.Reader 和 io.Writer
- strings.Reader 实现了 io.Reader

- bufio.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- bytes.Buffer 同时实现了 io.Reader 和 io.Writer
- bytes.Reader 实现了 io.Reader
- compress/gzip.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- crypto/cipher.StreamReader/StreamWriter 分别实现了 io.Reader 和 io.Writer
- crypto/tls.Conn 同时实现了 io.Reader 和 io.Writer
- encoding/csv.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- mime/multipart.Part 实现了 io.Reader

除此之外，io 包本身也有这两个接口的实现类型。如：

```
实现了 Reader 的类型：LimitedReader、PipeReader、SectionReader
实现了 Writer 的类型：PipeWriter
```

以上类型中，常用的类型有：os.File、strings.Reader、bufio.Reader/Writer、bytes.Buffer、bytes.Reader

小贴士

从接口名称很容易猜到，一般地， Go 中接口的命名约定：接口名以 er 结尾。注意，这里并非强行要求，你完全可以不以 er 结尾。标准库中有些接口也不是以 er 结尾的。

ReaderAt 和 WriterAt 接口

ReaderAt 接口的定义如下：

```
type ReaderAt interface {
    ReadAt(p []byte, off int64) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> ReadAt 从基本输入源的偏移量 off 处开始，将 len(p) 个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)）以及任何遇到的错误。
>
> 当 ReadAt 返回的 n < len(p) 时，它就会返回一个 非nil 的错误来解释 为什么没有返回更多的字节。在这一点上，ReadAt 比 Read 更严格。
>
> 即使 ReadAt 返回的 n < len(p)，它也会在调用过程中使用 p 的全部作为暂存空间。若一些数据可用但不到 len(p) 字节，ReadAt 就会阻塞直到所有数据都可用或产生一个错误。在这一点上 ReadAt 不同于 Read。
>
> 若 n = len(p) 个字节在输入源的的结尾处由 ReadAt 返回，那么这时 err == EOF 或者 err == nil。
>
> 若 ReadAt 按查找偏移量从输入源读取，ReadAt 应当既不影响基本查找偏移量也不被它所影响。
>
> ReadAt 的客户端可对相同的输入源并行执行 ReadAt 调用。

可见，ReaderAt 接口使得可以从指定偏移量处开始读取数据。

简单示例代码如下：

```
reader := strings.NewReader("Go语言学习园地")
p := make([]byte, 6)
n, err := reader.ReadAt(p, 2)
if err != nil {
    panic(err)
}
fmt.Printf("%s, %d\n", p, n)
```

输出：

```
语言, 6
```

WriterAt 接口的定义如下：

```
type WriterAt interface {
    WriteAt(p []byte, off int64) (n int, err error)
}
```

官方文档中关于该接口方法的说明：

> WriteAt 从 p 中将 len(p) 个字节写入到偏移量 off 处的基本数据流中。它返回从 p 中被写入的字节数 n（0 <= n <= len(p)）以及任何遇到的引起写入提前停止的错误。若 WriteAt 返回的 n < len(p)，它就必须返回一个 非nil 的错误。
>
> 若 WriteAt 按查找偏移量写入到目标中，WriteAt 应当既不影响基本查找偏移量也不被它所影响。
>
> 若区域没有重叠，WriteAt 的客户端可对相同的目标并行执行 WriteAt 调用。

我们可以通过该接口将数据写入数据流的特定偏移量之后。

通过简单示例来演示 WriteAt 方法的使用（os.File 实现了 WriterAt 接口）：

```
file, err := os.Create("writeAt.txt")
if err != nil {
    panic(err)
}
defer file.Close()
file.WriteString("Golang中文社区—这里是多余的")
n, err := file.WriteAt([]byte("Go语言学习园地"), 24)
if err != nil {
    panic(err)
}
fmt.Println(n)
```

打开文件 WriteAt.txt，内容是：`Golang中文社区—Go语言学习园地` 。

分析：

`file.WriteString("Golang中文社区—这里是多余的")` 往文件中写入 `Golang中文社区—这里是多余的` ，之后 `file.WriteAt([]byte("Go语言学习园地"), 24)` 在文件流的 offset=24 处写入 `Go语言学习园地` （会覆盖该位置的内容）。

ReaderFrom 和 WriterTo 接口

ReaderFrom 的定义如下：

```
type ReaderFrom interface {
    ReadFrom(r Reader) (n int64, err error)
}
```

官方文档中关于该接口方法的说明：

> ReadFrom 从 r 中读取数据，直到 EOF 或发生错误。其返回值 n 为读取的字节数。除 io.EOF 之外，在读取过程中遇到的任何错误也将被返回。
>
> 如果 ReaderFrom 可用，Copy 函数就会使用它。

注意：ReadFrom 方法不会返回 err == EOF。

下面的例子简单的实现将文件中的数据全部读取（显示在标准输出）：

```
file, err := os.Open("writeAt.txt")
if err != nil {
    panic(err)
}
defer file.Close()
writer := bufio.NewWriter(os.Stdout)
writer.ReadFrom(file)
writer.Flush()
```

当然，我们可以通过 ioutil 包的 ReadFile 函数获取文件全部内容。其实，跟踪一下 ioutil.ReadFile 的源码，会发现其实也是通过 ReadFrom 方法实现（用的是 bytes.Buffer，它实现了 ReaderFrom 接口）。

如果不通过 ReadFrom 接口来做这件事，而是使用 io.Reader 接口，我们有两种思路：

1. 先获取文件的大小（File 的 Stat 方法），之后定义一个该大小的 []byte，通过 Read 一次性读取
2. 定义一个小的 []byte，不断的调用 Read 方法直到遇到 EOF，将所有读取到的 []byte 连接到一起

这里不给出实现代码了，有兴趣的可以实现以下。

提示

通过查看 bufio.Writer 或 strings.Buffer 类型的 ReadFrom 方法实现，会发现，其实它们的实现和上面说的第 2 种思路类似。

WriterTo的定义如下：

```
type WriterTo interface {
    WriteTo(w Writer) (n int64, err error)
}
```

官方文档中关于该接口方法的说明：

> WriteTo 将数据写入 w 中，直到没有数据可写或发生错误。其返回值 n 为写入的字节数。 在写入过程中遇到的任何错误也将被返回。
>
> 如果 WriterTo 可用，Copy 函数就会使用它。

读者是否发现，其实 ReaderFrom 和 WriterTo 接口的方法接收的参数是 io.Reader 和 io.Writer 类型。根据 io.Reader 和 io.Writer 接口的讲解，对该接口的使用应该可以很好的掌握。

这里只提供简单的一个示例代码：将一段文本输出到标准输出

```
reader := bytes.NewReader([]byte("Go语言学习园地"))
reader.WriteTo(os.Stdout)
```

通过 io.ReaderFrom 和 io.WriterTo 的学习，我们知道，如果这样的需求，可以考虑使用这两个接口：“一次性从某个地方读或写到某个地方去。”

Seeker 接口

接口定义如下：

```
type Seeker interface {
    Seek(offset int64, whence int) (ret int64, err error)
}
```

官方文档中关于该接口方法的说明：

> Seek 设置下一次 Read 或 Write 的偏移量为 offset，它的解释取决于 whence： 0 表示相对于文件的起始处，1 表示相对于当前的偏移，而 2 表示相对于其结尾处。 Seek 返回新的偏移量和一个错误，如果有的话。

也就是说，Seek 方法用于设置偏移量的，这样可以从某个特定位置开始操作数据流。听起来和 ReaderAt/WriteAt 接口有些类似，不过 Seeker 接口更灵活，可以更好的控制读写数据流的位置。

简单的示例代码：获取倒数第二个字符（需要考虑 UTF-8 编码，这里的代码只是一个示例）

```
reader := strings.NewReader("Go语言学习园地")
reader.Seek(-6, os.SEEK_END)
r, _, _ := reader.ReadRune()
fmt.Printf("%c\n", r)
```

小贴士

whence 的值，在 os 包中定义了相应的常量，应该使用这些常量

```
const (
    SEEK_SET int = 0 // seek relative to the origin of the file
    SEEK_CUR int = 1 // seek relative to the current offset
    SEEK_END int = 2 // seek relative to the end
)
```

Closer接口

接口定义如下：

```
type Closer interface {
    Close() error
}
```

该接口比较简单，只有一个 Close() 方法，用于关闭数据流。

文件 (os.File)、归档（压缩包）、数据库连接、Socket 等需要手动关闭的资源都实现了 Closer 接口。

实际编程中，经常将 Close 方法的调用放在 defer 语句中。

小提示

初学者容易写出这样的代码：

```
file, err := os.Open("studygolang.txt")
defer file.Close()
if err != nil {
    ...
}
```

当文件 studygolang.txt 不存在或找不到时，file.Close() 会panic，因为 file 是 nil。因此，应该将 defer file.Close() 放在错误检查之后。

其他接口

ByteReader 和 ByteWriter

通过名称大概也能猜出这组接口的用途：读或写一个字节。接口定义如下：

```
type ByteReader interface {
    ReadByte() (c byte, err error)
}

type ByteWriter interface {
    WriteByte(c byte) error
}
```

在标准库中，有如下类型实现了 io.ByteReader 或 io.ByteWriter:

- bufio.Reader/Writer 分别实现了io.ByteReader 和 io.ByteWriter
- bytes.Buffer 同时实现了 io.ByteReader 和 io.ByteWriter
- bytes.Reader 实现了 io.ByteReader
- strings.Reader 实现了 io.ByteReader

接下来的示例中，我们通过 bytes.Buffer 来一次读取或写入一个字节（主要代码）：

```
var ch byte
fmt.Scanf("%c\n", &ch)

buffer := new(bytes.Buffer)
err := buffer.WriteByte(ch)
if err == nil {
    fmt.Println("写入一个字节成功！准备读取该字节……")
    newCh, _ := buffer.ReadByte()
    fmt.Printf("读取的字节：%c\n", newCh)
} else {
    fmt.Println("写入错误")
}
```

程序从标准输入接收一个字节（ASCII 字符），调用 buffer 的 WriteByte 将该字节写入 buffer 中，之后通过 ReadByte 读取该字节。完整的代码见：code/src/chapter01/io/byterwer.go

一般地，我们不会使用 bytes.Buffer 来一次读取或写入一个字节。那么，这两个接口有哪些用处呢？

在标准库 encoding/binary 中，实现Google-ProtoBuf中的 Varints 读取，ReadVarint 就需要一个 io.ByteReader 类型的参数，也就是说，它需要一个字节一个字节的读取。关于 encoding/binary 包在后面会详细介绍。

在标准库 image/jpeg 中，Encode函数的内部实现使用了 ByteWriter 写入一个字节。

小贴士

可以通过在 Go 语言源码 src/pkg 中搜索 "io.ByteReader" 或 "io.ByteWiter"，获得哪些地方用到了这两个接口。你会发现，这两个接口在二进制数据或归档压缩时用的比较多。

ByteScanner、RuneReader 和 RuneScanner

将这三个接口放在一起，是考虑到与 ByteReader 相关或相应。

ByteScanner 接口的定义如下：

```
type ByteScanner interface {
    ByteReader
    UnreadByte() error
}
```

可见，它内嵌了 ByteReader 接口（可以理解为继承了 ByteReader 接口），UnreadByte 方法的意思是：将上一次 ReadByte 的字节还原，使得再次调用 ReadByte 返回的结果和上一次调用相同，也就是说，UnreadByte 是重置上一次的 ReadByte。注意，UnreadByte 调用之前必须调用了 ReadByte，且不能连续调用 UnreadByte。即：

```
buffer := bytes.NewBuffer([]byte{'a', 'b'})
err := buffer.UnreadByte()
```

和

```
buffer := bytes.NewBuffer([]byte{'a', 'b'})
buffer.ReadByte()
err := buffer.UnreadByte()
err = buffer.UnreadByte()
```

err 都 非nil，错误为：`bytes.Buffer: UnreadByte: previous operation was not a read`

RuneReader 接口和 ByteReader 类似，只是 ReadRune 方法读取单个 UTF-8 字符，返回其 rune 和该字符占用的字节数。该接口在 regexp 包有用到。

之前有人在QQ群中问道：

> strings.Index("行业交流群", "交流") 返回的是单字节字符的位置：6。但是想要的是 unicode 字符的位置：2。

当时以为 strings.IndexRune 可以，然而 IndexRune 还不如 Index，一方面第二个参数是 rune 类型；另一方面返回的结果跟 Index 是一样的。这里通过 RuneReader 接口来实现这个需求，代码如下：

```
// strings.Index 的 UTF-8 版本
// 即 Utf8Index("Go语言学习园地", "学习") 返回 4，而不是 strings.Index 的 8
func Utf8Index(str, substr string) int {
    asciiPos := strings.Index(str, substr)
    if asciiPos == -1 || asciiPos == 0 {
        return asciiPos
    }
    pos := 0
    totalSize := 0
    reader := strings.NewReader(str)
    for _, size, err := reader.ReadRune(); err == nil; _, size, err = reader.ReadRune(
) {
        totalSize += size
        pos++
        // 匹配到
        if totalSize == asciiPos {
            return pos
        }
    }
    return pos
}
```

该实现借助了 strings.Index。另外，此处的 strings.NewReader 可以换成 bytes.NewBufferString，不过，根据 strings.NewReader 的文档，strings.Reader 比 bytes.Buffer 效率更高，只是 strings.Reader 是只读的，而 bytes.Buffer 是可读写的（从前面介绍的实现的接口可以知道）。关于 bytes 和 strings 包，后面章节会详细介绍。

RuneScanner 接口和 ByteScanner 类似，就不赘述了。

ReadCloser、ReadSeeker、ReadWriteCloser、ReadWriteSeeker、ReadWriter、WriteCloser 和 WriteSeeker 接口

这些接口是上面介绍的接口的两个或三个组合而成的新接口。例如 ReadWriter 接口：

```
type ReadWriter interface {
    Reader
    Writer
}
```

这是 Reader 接口和 Writer 接口的简单组合（内嵌）。

这些接口的作用是：有些时候同时需要某两个接口的所有功能，即必须同时实现了某两个接口的类型才能够被传入使用。可见，io 包中有大量的“小接口”，这样方便组合为“大接口”。

SectionReader 类型

SectionReader 是一个 struct（没有任何导出的字段），实现了 Read, Seek 和 ReadAt，同时，内嵌了 ReaderAt 接口。结构定义如下：

```
type SectionReader struct {
    r     ReaderAt    // 该类型最终的 Read/ReadAt 最终都是通过 r 的 ReadAt 实现
    base  int64       // NewSectionReader 会将 base 设置为 off
    off   int64       // 从 r 中的 off 偏移处开始读取数据
    limit int64       // limit - off = SectionReader 流的长度
}
```

从名称我们可以猜到，该类型读取数据流中部分数据。看一下

```
func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReader
```

的文档说明就知道了：

> NewSectionReader 返回一个 SectionReader，它从 r 中的偏移量 off 处读取 n 个字节后以 EOF 停止。

也就是说，SectionReader 只是内部（内嵌）ReaderAt 表示的数据流的一部分：从 off 开始后的 n 个字节。

这个类型的作用是：方便重复操作某一段 (section) 数据流；或者同时需要 ReadAt 和 Seek 的功能。

由于该类型所支持的操作，前面都有介绍，因此提供示例代码了。

关于该类型在标准库中的使用，我们在 8.5 archive/zip — zip归档访问 会讲到。

LimitedReader 类型

LimitedReader 类型定义如下：

```
type LimitedReader struct {
    R Reader // underlying reader，最终的读取操作通过 R.Read 完成
    N int64  // max bytes remaining
}
```

文档说明如下：

> 从 R 读取但将返回的数据量限制为 N 字节。每调用一次 Read 都将更新 N 来反应新的剩余数量。

也就是说，最多只能返回 N 字节数据。

LimitedReader 只实现了 Read 方法（Reader 接口）。

使用示例如下：

```
content := "This Is LimitReader Example"
reader := strings.NewReader(content)
limitReader := &io.LimitedReader{R: reader, N: 8}
for limitReader.N > 0 {
    tmp := make([]byte, 2)
    limitReader.Read(tmp)
    fmt.Printf("%s", tmp)
}
```

输出：

```
This Is
```

可见，通过该类型可以达到 只允许读取一定长度数据 的目的。

在 io 包中，LimitReader 函数的实现其实就是调用 LimitedReader：

```
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }
```

PipeReader 和 PipeWriter 类型

PipeReader（一个没有任何导出字段的 struct）是管道的读取端。它实现了 io.Reader 和 io.Closer 接口。

关于 **Read** 方法的说明：从管道中读取数据。该方法会堵塞，直到管道写入端开始写入数据或写入端关闭了。如果写入端关闭时带上了 error（即调用 CloseWithError 关闭），该方法返回的 err 就是写入端传递的error；否则 err 为 EOF。

PipeWriter（一个没有任何导出字段的 struct）是管道的写入端。它实现了 io.Writer 和 io.Closer 接口。

关于 **Write** 方法的说明：写数据到管道中。该方法会堵塞，直到管道读取端读完所有数据或读取端关闭了。如果读取端关闭时带上了 error（即调用 CloseWithError 关闭），该方法返回的 err 就是读取端传递的error；否则 err 为 ErrClosedPipe。

其他方法的使用通过例子一起讲解：

```
func main() {
    Pipe()
}

func Pipe() {
    pipeReader, pipeWriter := io.Pipe()
    go PipeWrite(pipeWriter)
    go PipeRead(pipeReader)
    time.Sleep(1e7)
}

func PipeWrite(pipeWriter *io.PipeWriter) {
    var (
        i   = 0
        err error
        n int
    )
    data := []byte("Go语言学习园地")
    for _, err = pipeWriter.Write(data); err == nil; n, err = pipeWriter.Write(data) {
        i++
        if i == 3 {
            pipeWriter.CloseWithError(errors.New("输出3次后结束"))
        }
    }
    fmt.Println("close 后输出的字节数：", n, " error：",  err)
}

func PipeRead(pipeReader *io.PipeReader) {
    var (
        err error
        n   int
    )
    data := make([]byte, 1024)
    for n, err = pipeReader.Read(data); err == nil; n, err = pipeReader.Read(data) {
        fmt.Printf("%s\n", data[:n])
    }
    fmt.Println("writer 端 closewitherror 后：", err)
}
```

输出是：

```
Go语言学习园地
Go语言学习园地
Go语言学习园地
Go语言学习园地
writer 端 closewitherror 后： 输出3次后结束
close 后输出的字节数： 20  error： io: read/write on closed pipe
```

细心的读者可能发现：不是输出 3 次后结束吗？怎么“Go语言学习园地”却输出了 4 次？这个问题我们稍后讨论。我们先来分析一下例子代码。

io.Pipe() 用于创建一个同步的内存管道 (synchronous in-memory pipe)，函数签名：

```
func Pipe() (*PipeReader, *PipeWriter)
```

它将 io.Reader 连接到 io.Writer。一端的读取匹配另一端的写入，直接在这两端之间复制数据；它没有内部缓存。它对于并行调用 Read 和 Write 以及其它函数或 Close 来说都是安全的。一旦等待的 I/O 结束，Close 就会完成。并行调用 Read 或并行调用 Write 也同样安全：同种类的调用将按顺序进行控制。稍后我们会分析管道相关的源码。

正因为是同步的，因此不能在一个 goroutine 中进行读和写。

在 PipeWrite 函数中，我们循环往管道中写数据，写第三次时，我们调用 CloseWithError 方法关闭管道的写入端，之后再一次调用 Write 方法，发现返回了error，于是退出了循环。

可是，从输出结果中，我们发现，最后一次写虽然返回 error（返回的 n 并非 0），但是读取端却能读到最后一次写的数据，这让人很费解。下面我们一起来探索一下相关源码，分析问题的原因。

io 包 管道 (pipe) 源码分析

从上文知道，PipeWriter 和 PipeReader 都没有导出成员。查看源码发现，两者都只有一个成员：`p *pipe`，这两种类型的所有方法都是调用了 pipe 类型对应的方法实现的。

pipe类型的定义如下：

```
// A pipe is the shared pipe structure underlying PipeReader and PipeWriter.
type pipe struct {
    rl    sync.Mutex // gates readers one at a time
    wl    sync.Mutex // gates writers one at a time
    l     sync.Mutex // protects remaining fields
    data  []byte     // data remaining in pending write
    rwait sync.Cond  // waiting reader
    wwait sync.Cond  // waiting writer
    rerr  error      // if reader closed, error to give writes
    werr  error      // if writer closed, error to give reads
}
```

字段说明：

- rl/wl 用于控制同一时刻只能有一个读取器或写入器
- l 用于保护其他字段
- data 在管道中的数据

- rwait/wwait sync.Cond 类型（后续会讲解），分别控制读取器或写入器等待
- rerr/werr 读取器（写入器）关闭，该错误会被 Write (Read) 方法返回

pipe 的 read 方法：

```
func (p *pipe) read(b []byte) (n int, err error) {
    // One reader at a time.（控制一次只能一个读取器）
    p.rl.Lock()
    defer p.rl.Unlock()

    // 保护其他字段的读写
    p.l.Lock()
    defer p.l.Unlock()
    for {
        // Reader 端关闭后，再 Read，则返回 ErrClosedPipe
        if p.rerr != nil {
            return 0, ErrClosedPipe
        }
        // 管道中有数据，退出循环
        if p.data != nil {
            break
        }
        // Writer 端关闭，返回 p.werr
        if p.werr != nil {
            return 0, p.werr
        }
        // 没有数据或管道没有关闭，读取端等待
        p.rwait.Wait()
    }
    // 管道中有数据，将其 copy 一份到 b 中
    n = copy(b, p.data)
    p.data = p.data[n:]
    // 如果管道数据被读光，需要唤醒在等待的 Writer
    if len(p.data) == 0 {
        p.data = nil
        p.wwait.Signal()
    }
    return
}
```

加上的代码注释已经很清楚了，因此不再赘述。

pipe 的 write 方法：

```
func (p *pipe) write(b []byte) (n int, err error) {
    // pipe uses nil to mean not available
    if b == nil {
        // zero 的定义为：var zero [0]byte
        b = zero[:]
    }

    // One writer at a time.
    p.wl.Lock()
    defer p.wl.Unlock()

    p.l.Lock()
    defer p.l.Unlock()
    // 上面说的问题来了：不管三七二十一，一上来些将数据放进管道中
    p.data = b
    // 唤醒在等待的 Reader
    p.rwait.Signal()
    for {
        // 数据被读走，退出循环
        if p.data == nil {
            break
        }
        // Reader 端关闭，设置 err = p.rerr，退出循环
        if p.rerr != nil {
            err = p.rerr
            break
        }
        // Writer 端关闭后，再 Writer，设置 err = ErrClosedPipe
        if p.werr != nil {
            err = ErrClosedPipe
        }
        // 数据没被读走（全部）或管道读取端没关闭，则等待
        p.wwait.Wait()
    }
    // 计算写入的字节数
    n = len(b) - len(p.data)
    p.data = nil // in case of rerr or werr
    return
}
```

通过上面两个方法的代码注释，应该清楚例子中为啥输出4次了吧？我们再分析一下：

当 i == 3，调用 CloseWithError 之后，程序执行 for 中的 `n, err = pipeWriter.Write(data)`，根据上面 pipe.write 方法，p.data 会被设置上数据，这个时候，Reader 被唤醒，将数据读走（第 4 次）。由于异步，多 goroutine，跟调度有关系，这个时候 Writer 可能在等待，也可能在 Reader 读完数据后将其唤醒，总之，Writer 会执行到 `if p.werr != nil`，即例子中 Write 循环结束；而 Reader 被唤醒之后，首先判断的是 `p.data != nil`，而不是 `p.werr != nil`，因此数据被正常读取，且没错误被返回，这时执行下一次循环，当然，这时候由于没有 Write，且 `p.werr != nil`，于是 Read 方法返回 err(=p.werr)。

个人认为这是一个 bug，已经向官方提出：issue5330，修复处：

1）在 pipe.write 方法的 defer p.l.Unlock() 后面增加如下代码：

```
// 写端关闭了，不让写入数据
if p.werr != nil {
    p.rwait.Signal()
    err = ErrClosedPipe
    return
}
```

同时，for 循环中如下代码没有必要，删除：

```
// Writer 端关闭后，再 Writer，设置 err = ErrClosedPipe
if p.werr != nil {
    err = ErrClosedPipe
}
```

2）在 pipe.read 方法中，调整检查 p.werr 和 p.data 的顺序，即改为：

```
if p.werr != nil {
    return 0, p.werr
}
if p.data != nil {
    break
}
```

这样不至于有错误时还把数据读走。

另外，对于管道的 close 方法（非 CloseWithError 时），err 会被置为 EOF。

Copy 和 CopyN 函数

Copy 函数的签名：

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

函数文档：

> Copy 将 src 复制到 dst，直到在 src 上到达 EOF 或发生错误。它返回复制的字节数，如果有的话，还会返回在复制时遇到的第一个错误。
>
> 成功的 Copy 返回 err == nil，而非 err == EOF。由于 Copy 被定义为从 src 读取直到 EOF 为止，因此它不会将来自 Read 的 EOF 当做错误来报告。
>
> 若 dst 实现了 ReaderFrom 接口，其复制操作可通过调用 dst.ReadFrom(src) 实现。此外，若 src 实现了 WriterTo 接口，其复制操作可通过调用 src.WriteTo(dst) 实现。

代码：

```
io.Copy(os.Stdout, strings.NewReader("Go语言学习园地"))
```

直接将内容输出（写入 Stdout 中）。

我们甚至可以这么做：

```
package main

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

func main() {
    io.Copy(os.Stdout, os.Stdin)
    fmt.Println("Got EOF -- bye")
}
```

执行：`echo "Hello, World" | go run main.go`

CopyN 函数的签名：

```
func CopyN(dst Writer, src Reader, n int64) (written int64, err error)
```

函数文档：

> CopyN 将 n 个字节从 src 复制到 dst。它返回复制的字节数以及在复制时遇到的最早的错误。由于 Read 可以返回要求的全部数量及一个错误（包括 EOF），因此 CopyN 也能如此。
>
> 若 dst 实现了 ReaderFrom 接口，复制操作也就会使用它来实现。

代码：

```
io.CopyN(os.Stdout, strings.NewReader("Go语言学习园地"), 8)
```

会输出：

```
Go语言
```

ReadAtLeast 和 ReadFull 函数

ReadAtLeast 函数的签名：

```
func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)
```

函数文档：

> ReadAtLeast 将 r 读取到 buf 中，直到读了最少 min 个字节为止。它返回复制的字节数，如果读取的字节较少，还会返回一个错误。若没有读取到字节，错误就只是 EOF。如果一个 EOF 发生在读取了少于 min 个字节之后，ReadAtLeast 就会返回 ErrUnexpectedEOF。若 min 大于 buf 的长度，ReadAtLeast 就会返回 ErrShortBuffer。对于返回值，当且仅当 err == nil 时，才有 n >= min。

一般可能不太会用到这个函数。使用时需要注意返回的 error 判断。

ReadFull 函数的签名：

```
func ReadFull(r Reader, buf []byte) (n int, err error)
```

函数文档：

> ReadFull 精确地从 r 中将 len(buf) 个字节读取到 buf 中。它返回复制的字节数，如果读取的字节较少，还会返回一个错误。若没有读取到字节，错误就只是 EOF。如果一个 EOF 发生在读取了一些但不是所有的字节后，ReadFull 就会返回 ErrUnexpectedEOF。对于返回值，当且仅当 err == nil 时，才有 n == len(buf)。

注意该函数和 ReadAtLeast 的区别：ReadFull 将 buf 读满；而 ReadAtLeast 是最少读取 min 个字节。

WriteString 函数

这是为了方便写入 string 类型提供的函数，函数签名：

```
func WriteString(w Writer, s string) (n int, err error)
```

当 w 实现了 WriteString 方法时，直接调用该方法，否则执行 w.Write([]byte(s))。

MultiReader 和 MultiWriter 函数

这两个函数的定义分别是：

```
func MultiReader(readers ...Reader) Reader
func MultiWriter(writers ...Writer) Writer
```

它们接收多个 Reader 或 Writer，返回一个 Reader 或 Writer。我们可以猜想到这两个函数就是操作多个 Reader 或 Writer 就像操作一个。

事实上，在 io 包中定义了两个非导出类型：mutilReader 和 multiWriter，它们分别实现了 io.Reader 和 io.Writer 接口。类型定义为：

```
type multiReader struct {
    readers []Reader
}

type multiWriter struct {
    writers []Writer
}
```

对于这两种类型对应的实现方法（Read 和 Write 方法）的使用，我们通过例子来演示。

MultiReader 的使用：

```
readers := []io.Reader{
    strings.NewReader("from strings reader"),
    bytes.NewBufferString("from bytes buffer"),
}
reader := io.MultiReader(readers...)
data := make([]byte, 0, 1024)
var (
    err error
    n   int
)
for err != io.EOF {
    tmp := make([]byte, 512)
    n, err = reader.Read(tmp)
    if err == nil {
        data = append(data, tmp[:n]...)
    } else {
        if err != io.EOF {
            panic(err)
        }
    }
}
fmt.Printf("%s\n", data)
```

输出：

```
from strings readerfrom bytes buffer
```

代码中首先构造了一个 io.Reader 的 slice，由 strings.Reader 和 bytes.Buffer 两个实例组成，然后通过 MultiReader 得到新的 Reader，循环读取新 Reader 中的内容。从输出结果可以看到，第一次调用 Reader 的 Read 方法获取到的是 slice 中第一个元素的内容……也就是说，MultiReader 只是逻辑上将多个 Reader 组合起来，并不能通过调用一次 Read 方法获取所有 Reader 的内容。在所有的 Reader 内容都被读完后，Reader 会返回 EOF。

MultiWriter 的使用：

```
file, err := os.Create("tmp.txt")
if err != nil {
    panic(err)
}
defer file.Close()
writers := []io.Writer{
    file,
    os.Stdout,
}
writer := io.MultiWriter(writers...)
writer.Write([]byte("Go语言学习园地"))
```

这段程序执行后在生成 tmp.txt 文件，同时在文件和屏幕中都输出：`Go语言学习园地` 。这和 Unix 中的 tee 命令类似。

动手试试

Go 实现 Unix 中 tee 命令的功能很简单吧。MultiWriter 的 Write 方法是如何实现的？有兴趣可以自己实现一个，然后对着源码比较一下。

TeeReader函数

函数签名如下：

```
func TeeReader(r Reader, w Writer) Reader
```

TeeReader 返回一个 Reader，它将从 r 中读到的数据写入 w 中。所有经由它处理的从 r 的读取都匹配于对应的对 w 的写入。它没有内部缓存，即写入必须在读取完成前完成。任何在写入时遇到的错误都将作为读取错误返回。

也就是说，我们通过 Reader 读取内容后，会自动写入到 Writer 中去。例子代码如下：

```
reader := io.TeeReader(strings.NewReader("Go语言学习园地"), os.Stdout)
reader.Read(make([]byte, 20))
```

输出结果：

```
Go语言学习园地
```

这种功能的实现其实挺简单，无非是在 Read 完后执行 Write。

至此，io 所有接口、类型和函数都讲解完成。

导航

- 目录
- 下一节：ioutil — 方便的 IO 操作函数集

1.2 ioutil — 方便的IO操作函数集

虽然 io 包提供了不少类型、方法和函数，但有时候使用起来不是那么方便。比如读取一个文件中的所有内容。为此，标准库中提供了一些常用、方便的IO操作函数。

说明：这些函数使用都相对简单，一般就不举例子了。

NopCloser 函数

有时候我们需要传递一个io.ReadCloser的实例，而我们现在有一个io.Reader的实例，比如：strings.Reader，这个时候NopCloser就派上用场了。它包装一个io.Reader，返回一个io.ReadCloser，而相应的Close方法啥也不做，只是返回nil。

比如，在标准库net/http包中的NewRequest，接收一个io.Reader的body，而实际上，Request的Body的类型是io.ReadCloser，因此，代码内部进行了判断，如果传递的io.Reader也实现了io.ReadCloser接口，则转换，否则通过ioutil.NopCloser包装转换一下。相关代码如下：

```
rc, ok := body.(io.ReadCloser)
if !ok && body != nil {
    rc = ioutil.NopCloser(body)
}
```

如果没有这个函数，我们得自己实现一个。当然，实现起来很简单，读者可以看看NopCloser的实现。

ReadAll 函数

很多时候，我们需要一次性读取io.Reader中的数据，通过上一节的讲解，我们知道有很多种实现方式。考虑到读取所有数据的需求比较多，Go提供了ReadAll这个函数，用来从io.Reader中一次读取所有数据。

```
func ReadAll(r io.Reader) ([]byte, error)
```

阅读该函数的源码发现，它是通过bytes.Buffer中的ReadFrom来实现读取所有数据的。

ReadDir 函数

笔试题：编写程序输出某目录下的所有文件（包括子目录）

是否见过这样的笔试题？

在Go中如何输出目录下的所有文件呢？首先，我们会想到查os包，看File类型是否提供了相关方法（关于os包，后面会讲解）。

其实在ioutil中提供了一个方便的函数：ReadDir，它读取目录并返回排好序的文件和子目录名（[]os.FileInfo）。通过这个方法，我们可以很容易的实现“面试题”。

下面的例子实现了类似Unix中的tree命令，不过它在windows下也运行的很好哦。

```
// 未实现-L参数功能
func main() {
    if len(os.Args) > 1 {
        Tree(os.Args[1], 1, map[int]bool{1:true})
    }
}

// 列出dirname目录中的目录树，实现类似Unix中的tree命令
// curHier 当前层级（dirname为第一层）
// hierMap 当前层级的上几层是否需要'|'的映射
func Tree(dirname string, curHier int, hierMap map[int]bool) error {
    dirAbs, err := filepath.Abs(dirname)
    if err != nil {
        return err
    }
    fileInfos, err := ioutil.ReadDir(dirAbs)
    if err != nil {
        return err
    }

    fileNum := len(fileInfos)
    for i, fileInfo := range fileInfos {
        for j := 1; j < curHier; j++ {
            if hierMap[j] {
                fmt.Print("|")
            } else {
                fmt.Print(" ")
            }
            fmt.Print(strings.Repeat(" ", 3))
        }

        // map是引用类型，所以新建一个map
        tmpMap := map[int]bool{}
        for k, v := range hierMap {
            tmpMap[k] = v
        }
        if i+1 == fileNum {
            fmt.Print("`")
            delete(tmpMap, curHier)
        } else {
            fmt.Print("|")
            tmpMap[curHier] = true
        }
        fmt.Print("-- ")
        fmt.Println(fileInfo.Name())
        if fileInfo.IsDir() {
            Tree(filepath.Join(dirAbs, fileInfo.Name()), curHier+1, tmpMap)
        }
    }
    return nil
}
```

ReadFile 和 WriteFile 函数

ReadFile 读取整个文件的内容，在上一节我们自己实现了一个函数读取文件整个内容，由于这种需求很常见，因此Go提供了ReadFile函数，方便使用。ReadFile的是实现和ReadAll类似，不过，ReadFile会先判断文件的大小，给bytes.Buffer一个预定义容量，避免额外分配内存。

WriteFile 函数的签名如下：

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

它将data写入filename文件中，当文件不存在时会创建一个（文件权限由perm指定）；否则会先清空文件内容。对于perm参数，我们一般可以指定为：0666，具体含义os包中讲解。

小提示

ReadFile 源码中先获取了文件的大小，当大小 < 1e9时，才会用到文件的大小。按源码中注释的说法是FileInfo不会很精确地得到文件大小。

TempDir 和 TempFile 函数

操作系统中一般都会提供临时目录，比如linux下的/tmp目录（通过os.TempDir()可以获取到)。有时候，我们自己需要创建临时目录，比如Go工具链源码中（src/cmd/go/build.go），通过TempDir创建一个临时目录，用于存放编译过程的临时文件：

```
b.work, err = ioutil.TempDir("", "go-build")
```

第一个参数如果为空，表明在系统默认的临时目录（os.TempDir）中创建临时目录；第二个参数指定临时目录名的前缀，该函数返回临时目录的路径。

相应的，TempFile用于创建临时文件。如gofmt命令的源码中创建临时文件：

```
f1, err := ioutil.TempFile("", "gofmt")
```

参数和ioutil.TempDir参数含义类似。

这里需要注意：创建者创建的临时文件和临时目录要负责删除这些临时目录和文件。如删除临时文件：

```
defer func() {
    f.Close()
    os.Remove(f.Name())
}()
```

Discard 变量

Discard 对应的类型（`type devNull int`）实现了io.Writer接口，同时，为了优化io.Copy到Discard，避免不必要的工作，实现了io.ReaderFrom接口。

devNull 在实现io.Writer接口时，只是简单的返回（标准库文件：src/pkg/io/ioutil.go）。

```
func (devNull) Write(p []byte) (int, error) {
    return len(p), nil
}
```

而ReadFrom的实现是读取内容到一个buf中，最大也就8192字节，其他的会丢弃（当然，这个也不会读取）。

导航

- 目录
- 上一节：io — 基本的IO接口
- 下一节：fmt — 格式化IO

1.3 fmt — 格式化IO

fmt 包实现了格式化I/O函数，类似于C的 printf 和 scanf. 格式“占位符”衍生自C，但比C更简单。

fmt 包的官方文档对Printing和Scanning有很详细的说明。这里就直接引用文档进行说明，同时附上额外的说明或例子，之后再介绍具体的函数使用。

以下例子中用到的类型或变量定义：

```
type Website struct {
    Name string
}

// 打印结构体时
var site = Website{Name:"studygolang"}
```

Printing

占位符

普通占位符

```
占位符                          说明                          举例
                输出
%v          相应值的默认格式。                                  Printf("%v", site)，Printf("%
+v", site)    {studygolang}，{Name:studygolang}
          在打印结构体时，“加号”标记（%+v）会添加字段名
%#v           相应值的Go语法表示                          Printf("#v", site)
            main.Website{Name:"studygolang"}
%T          相应值的类型的Go语法表示                      Printf("%T", site)
            main.Website
%%          字面上的百分号，并非值的占位符                  Printf("%%")
           %
```

布尔占位符

```
占位符                          说明                          举例
                输出
%t          单词 true 或 false。                            Printf("%t", true)
              true
```

整数占位符

占位符	说明	举例	输出
%b	二进制表示	Printf("%b", 5)	101
%c	相应Unicode码点所表示的字符	Printf("%c", 0x4E2D)	中
%d	十进制表示	Printf("%d", 0x12)	18
%o	八进制表示	Printf("%d", 10)	12
%q	单引号围绕的字符字面值，由Go语法安全地转义	Printf("%q", 0x4E2D)	'中'
%x	十六进制表示，字母形式为小写 a-f	Printf("%x", 13)	d
%X	十六进制表示，字母形式为大写 A-F	Printf("%x", 13)	D
%U	Unicode格式：U+1234，等同于 "U+%04X"	Printf("%U", 0x4E2D)	U+4E2D

浮点数和复数的组成部分（实部和虚部）

占位符	说明	举例	输出
%b	无小数部分的，指数为二的幂的科学计数法，与 strconv.FormatFloat 的 'b' 转换格式一致。例如 -123456p-78		
%e	科学计数法，例如 -1234.456e+78	Printf("%e", 10.2)	1.020000e+01
%E	科学计数法，例如 -1234.456E+78	Printf("%e", 10.2)	1.020000E+01
%f	有小数点而无指数，例如 123.456	Printf("%f", 10.2)	10.200000
%g	根据情况选择 %e 或 %f 以产生更紧凑的（无末尾的0）输出	Printf("%g", 10.20)	10.2
%G	根据情况选择 %E 或 %f 以产生更紧凑的（无末尾的0）输出	Printf("%G", 10.20+2i)	(10.2+2i)

字符串与字节切片

占位符	说明	举例	输出
%s	输出字符串表示（string类型或[]byte）	Printf("%s", []byte("Go语言学习园地"))	Go语言学习园地
%q	双引号围绕的字符串，由Go语法安全地转义	Printf("%q", "Go语言学习园地")	"Go语言学习园地"
%x	十六进制，小写字母，每字节两个字符	Printf("%x", "golang")	676f6c616e67
%X	十六进制，大写字母，每字节两个字符	Printf("%X", "golang")	676F6C616E67

指针

占位符	说明	举例	输出
%p	十六进制表示，前缀 0x	Printf("%p", &site)	0x4f57f0

这里没有 'u' 标记。若整数为无符号类型，他们就会被打印成无符号的。类似地，这里也不需要指定操作数的大小（int8，int64）。

宽度与精度的控制格式以Unicode码点为单位。（这点与C的 printf 不同，它以字节数为单位）二者或其中之一均可用字符 '*' 表示，此时它们的值会从下一个操作数中获取，该操作数的类型必须为 int。

对数值而言，宽度为该数值占用区域的最小宽度；精度为小数点之后的位数。 但对于 %g/%G 而言，精度为所有数字的总数。例如，对于123.45，格式 %6.2f 会打印123.45，而 %.4g 会打印123.5。%e 和 %f 的默认精度为6；但对于 %g 而言，它的默认精度为确定该值所必须的最小位数。

对大多数的值而言，宽度为输出的最小字符数，如果必要的话会为已格式化的形式填充空格。对字符串而言，精度为输出的最大字符数，如果必要的话会直接截断。

其它标记

占位符	说明	举例	输出
+	总打印数值的正负号；对于%q（%+q）保证只输出ASCII编码的字符。	Printf("%+q", "中文")	"\u4e2d\u6587"
-	在右侧而非左侧填充空格（左对齐该区域）		
#	备用格式：为八进制添加前导 0（%#o），为十六进制添加前导 0x（%#x）或 0X（%#X），为 %p（%#p）去掉前导 0x；如果可能的话，%q（%#q）会打印原始（即反引号围绕的）字符串；如果是可打印字符，%U（%#U）会写出该字符的 Unicode 编码形式（如字符 x 会被打印成 U+0078 'x'）。	Printf("%#U", '中')	U+4E2D '中'
' '	（空格）为数值中省略的正负号留出空白（% d）；以十六进制（% x, % X）打印字符串或切片时，在字节之间用空格隔开		
0	填充前导的0而非空格；对于数字，这会将填充移到正负号之后		

标记有时会被占位符忽略，所以不要指望它们。例如十进制没有备用格式，因此 %#d 与 %d 的行为相同。

对于每一个 Printf 类的函数，都有一个 Print 函数，该函数不接受任何格式化，它等价于对每一个操作数都应用 %v。另一个变参函数 Println 会在操作数之间插入空白，并在末尾追加一个换行符。

不考虑占位符的话，如果操作数是接口值，就会使用其内部的具体值，而非接口本身。 因此：

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

会打印 23。

若一个操作数实现了 Formatter 接口，该接口就能更好地用于控制格式化。

若其格式（它对于 Println 等函数是隐式的 %v）对于字符串是有效的 （%s %q %v %x %X），以下两条规则也适用：

```
1. 若一个操作数实现了 error 接口，Error 方法就能将该对象转换为字符串，随后会根据占位符的需要进行格式化。
2. 若一个操作数实现了 String() string 方法，该方法能将该对象转换为字符串，随后会根据占位符的需要进行格式化。
```

为避免以下这类递归的情况：

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

需要在递归前转换该值：

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

格式化错误

如果给占位符提供了无效的实参（例如将一个字符串提供给 %d），所生成的字符串会包含该问题的描述，如下例所示：

```
类型错误或占位符未知：%!verb(type=value)
    Printf("%d", hi):          %!d(string=hi)
实参太多：%!(EXTRA type=value)
    Printf("hi", "guys"):      hi%!(EXTRA string=guys)
实参太少： %!verb(MISSING)
    Printf("hi%d"):            hi %!d(MISSING)
宽度或精度不是int类型: %!(BADWIDTH) 或 %!(BADPREC)
    Printf("%*s", 4.5, "hi"):  %!(BADWIDTH)hi
    Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
所有错误都始于“%!”，有时紧跟着单个字符（占位符），并以小括号括住的描述结尾。
```

Scanning

一组类似的函数通过扫描已格式化的文本来产生值。Scan、Scanf 和 Scanln 从 os.Stdin 中读取；Fscan、Fscanf 和 Fscanln 从指定的 io.Reader 中读取； Sscan、Sscanf 和 Sscanln 从实参字符串中读取。Scanln、Fscanln 和 Sscanln 在换行符处停止扫描，且需要条目紧随换行符之后；Scanf、Fscanf 和 Sscanf 需要输入换行符来匹配格式中的换行符；其它函数则将换行符视为空格。

Scanf、Fscanf 和 Sscanf 根据格式字符串解析实参，类似于 Printf。例如，%x 会将一个整数扫描为十六进制数，而 %v 则会扫描该值的默认表现格式。

格式化行为类似于 Printf，但也有如下例外：

```
%p 没有实现
%T 没有实现
%e %E %f %F %g %G 都完全等价，且可扫描任何浮点数或复数数值
%s 和 %v 在扫描字符串时会将其中的空格作为分隔符
标记 # 和 + 没有实现
```

在使用 %v 占位符扫描整数时，可接受友好的进制前缀0（八进制）和0x（十六进制）。

宽度被解释为输入的文本（%5s 意为最多从输入中读取5个 rune 来扫描成字符串），而扫描函数则没有精度的语法（没有 %5.2f，只有 %5f）。

当以某种格式进行扫描时，无论在格式中还是在输入中，所有非空的连续空白字符（除换行符外）都等价于单个空格。由于这种限制，格式字符串文本必须匹配输入的文本，如果不匹配，扫描过程就会停止，并返回已扫描的实参数。

在所有的扫描参数中，若一个操作数实现了 Scan 方法（即它实现了 Scanner 接口），该操作数将使用该方法扫描其文本。此外，若已扫描的实参数少于所提供的实参数，就会返回一个错误。

所有需要被扫描的实参都必须是基本类型或 Scanner 接口的实现。

注意：Fscan 等函数会从输入中多读取一个字符（rune），因此，如果循环调用扫描函数，可能会跳过输入中的某些数据。一般只有在输入的数据中没有空白符时该问题才会出现。若提供给 Fscan 的读取器实现了 ReadRune，就会用该方法读取字符。若此读取器还实现了 UnreadRune 方法，就会用该方法保存字符，而连续的调用将不会丢失数据。若要为没有 ReadRune 和 UnreadRune 方法的读取器加上这些功能，需使用 bufio.NewReader。

Print 序列函数

这里说的 Print 序列函数包括：Fprint/Fprintf/Fprintln/Sprint/Sprintf/Sprintln/Print/Printf/Println。之所以将放在一起介绍，是因为它们的使用方式类似、参数意思也类似。

一般的，我们将Fprint/Fprintf/Fprintln归为一类；Sprint/Sprintf/Sprintln归为一类；Print/Printf/Println归为另一类。其中，Print/Printf/Println会调用相应的F开头一类函数。如：

```
func Print(a ...interface{}) (n int, err error) {
    return Fprint(os.Stdout, a...)
}
```

Fprint/Fprintf/Fprintln 函数的第一个参数接收一个io.Writer类型，会将内容输出到io.Writer中去。而Print/Printf/Println 函数是将内容输出到标准输出中，因此，直接调用F类函数做这件事，并将os.Stdout作为第一个参数传入。

Sprint/Sprintf/Sprintln 是格式化内容为string类型，而并不输出到某处，需要格式化字符串并返回时，可以用次组函数。

在这三组函数中，`S/F/Printf` 函数通过指定的格式输出或格式化内容；`S/F/Print` 函数只是使用默认的格式输出或格式化内容；`S/F/Println` 函数使用默认的格式输出或格式化内容，同时会在最后加上"换行符"。

Print 序列函数的最后一个参数都是 `a ...interface{}` 这种不定参数。对于 `S/F/Printf` 序列，这个不定参数的实参个数应该和 `formt` 参数的占位符个数一致，否则会出现格式化错误；而对于其他函数，当不定参数的实参个数为多个时，它们之间会直接（对

于 `S/F/Print` ）或通过""（空格）（对于 `S/F/Println` ）连接起来（注：对于 `S/F/Print` ，当两个参数都不是字符串时，会自动添加一个空格，否则不会加。感谢guoshanhe1983 反馈。官方 effective_go 也有说明）。利用这一点，我们可以做如下事情：

```
result1 := fmt.Sprintln("studygolang.com", 2013)
result2 := fmt.Sprint("studygolang.com", 2013)
```

result1的值是： `studygolang.com 2013` ，result2的值是： `studygolang.com2013` 。这起到了连接字符串的作用，而不需要通过strconv.Itoa()转换。

Print序列函数用的较多，而且也易于使用（可能需要掌握一些常用的占位符用法），接下来我们结合fmt包中几个相关的接口来掌握更多关于Print的内容。

Stringer 接口

Stringer接口的定义如下：

```
type Stringer interface {
    String() string
}
```

根据Go语言中实现接口的定义，一个类型只要有 `String() string` 方法，我们就说它实现了Stringer接口。而在本节开始已经说到，如果格式化输出某种类型的值，只要它实现了String()方法，那么会调用String()方法进行处理。

我们定义如下struct：

```
type Person struct {
    Name string
    Age  int
    Sex  int
}
```

我们给Person实现String方法，这个时候，我们输出Person的实例：

```
p := &Person{"polaris", 28, 0}
fmt.Println(p)
```

输出：

```
&{polaris 28 0}
```

接下来，为Person增加String方法。

```
func (this *Person) String() string {
    buffer := bytes.NewBufferString("This is ")
    buffer.WriteString(this.Name + ", ")
    if this.Sex == 0 {
        buffer.WriteString("He ")
    } else {
        buffer.WriteString("She ")
    }
    buffer.WriteString("is ")
    buffer.WriteString(strconv.Itoa(this.Age))
    buffer.WriteString(" years old.")
    return buffer.String()
}
```

这个时候运行：

```
p := &Person{"polaris", 28, 0}
fmt.Println(p)
```

输出变为：

```
This is polaris, He is 28 years old
```

可见，Stringer接口和Java中的ToString方法类似。

Formatter 接口

Formatter接口的定义如下：

```
type Formatter interface {
    Format(f State, c rune)
}
```

官方文档中关于该接口方法的说明：

> Formatter 接口由带有定制的格式化器的值所实现。 Format 的实现可调用 Sprintf 或 Fprintf(f) 等函数来生成其输出。

也就是说，通过实现Formatter接口可以做到自定义输出格式（自定义占位符）。

接着上面的例子，我们为Person增加一个方法：

```
func (this *Person) Format(f fmt.State, c rune) {
    if c == 'L' {
        f.Write([]byte(this.String()))
        f.Write([]byte(" Person has three fields."))
    } else {
        // 没有此句，会导致 fmt.Printf("%s", p) 啥也不输出
        f.Write([]byte(fmt.Sprintln(this.String())))
    }
}
```

这样，Person便实现了Formatter接口。这时再运行：

```
p := &Person{"polaris", 28, 0}
fmt.Printf("%L", p)
```

输出为：

```
This is polaris, He is 28 years old. Person has three fields.
```

这里需要解释以下几点：

1）fmt.State 是一个接口。由于Format方法是被fmt包调用的，它内部会实例化好一个fmt.State接口的实例，我们不需要关心该接口；

2）可以实现自定义占位符，同时fmt包中和类型相对应的预定义占位符会无效。因此例子中Format的实现加上了else子句；

3）实现了Formatter接口，相应的Stringer接口不起作用。但实现了Formatter接口的类型应该实现Stringer接口，这样方便在Format方法中调用String()方法。就像本例的做法；

4）Format方法的第二个参数是占位符中%后的字母（有精度和宽度会被忽略，只保留字母）；

一般地，我们不需要实现Formatter接口。如果对Formatter接口的实现感兴趣，可以看看标准库math/big包中Int类型的Formatter接口实现。

小贴士

State接口相关说明：

```
type State interface {
    // Write is the function to call to emit formatted output to be printed.
    // Write 函数用于打印出已格式化的输出。
    Write(b []byte) (ret int, err error)
    // Width returns the value of the width option and whether it has been set.
    // Width 返回宽度选项的值以及它是否已被设置。
    Width() (wid int, ok bool)
    // Precision returns the value of the precision option and whether it has been set
.
    // Precision 返回精度选项的值以及它是否已被设置。
    Precision() (prec int, ok bool)

    // Flag returns whether the flag c, a character, has been set.
    // Flag 返回标记 c（一个字符）是否已被设置。
    Flag(c int) bool
}
```

fmt包中的print.go文件中的 `type pp struct` 实现了State接口。由于State接口有Write方法，因此，实现了State接口的类型必然实现了io.Writer接口。

GoStringer 接口

GoStringer 接口定义如下；

```
type GoStringer interface {
    GoString() string
}
```

该接口定义了类型的Go语法格式。用于打印(Printf)格式化占位符为%#v的值。

用前面的例子演示。执行：

```
p := &Person{"polaris", 28, 0}
fmt.Printf("%#v", p)
```

输出：

```
&main.Person{Name:"polaris", Age:28, Sex:0}
```

接着为Person增加方法：

```
func (this *Person) GoString() string {
    return "&Person{Name is "+this.Name+", Age is "+strconv.Itoa(this.Age)+", Sex is "
+strconv.Itoa(this.Sex)+"}"
}
```

这个时候再执行

```
p := &Person{"polaris", 28, 0}
fmt.Printf("%#v", p)
```

输出：

```
&Person{Name is polaris, Age is 28, Sex is 0}
```

一般的，我们不需要实现该接口。

Scan 序列函数

该序列函数和 Print 序列函数相对应，包括：
Fscan/Fscanf/Fscanln/Sscan/Sscanf/Sscanln/Scan/Scanf/Scanln。

一般的，我们将Fscan/Fscanf/Fscanln归为一类；Sscan/Sscanf/Sscanln归为一类；Scan/Scanf/Scanln归为另一类。其中，Scan/Scanf/Scanln会调用相应的F开头一类函数。如：

```
func Scan(a ...interface{}) (n int, err error) {
    return Fscan(os.Stdin, a...)
}
```

Fscan/Fscanf/Fscanln 函数的第一个参数接收一个io.Reader类型，从其读取内容并赋值给相应的实参。而 Scan/Scanf/Scanln 正是从标准输入获取内容，因此，直接调用F类函数做这件事，并将os.Stdin作为第一个参数传入。

Sscan/Sscanf/Sscanln 则直接从字符串中获取内容。

对于Scan/Scanf/Scanln三个函数的区别，我们通过例子来说明，为了方便讲解，我们使用Sscan/Sscanf/Sscanln这组函数。

1) Scan/FScan/Sscan

```
var (
    name string
    age  int
)
n, _ := fmt.Sscan("polaris 28", &name, &age)
// 可以将"polaris 28"中的空格换成"\n"试试
// n, _ := fmt.Sscan("polaris\n28", &name, &age)
fmt.Println(n, name, age)
```

输出为：

```
2 polaris 28
```

不管"polaris 28"是用空格分隔还是"\n"分隔，输出一样。也就是说，`Scan/FScan/Sscan` 这组函数将连续由空格分隔的值存储为连续的实参（换行符也记为空格）。

2) Scanf/FScanf/Sscanf

```
var (
    name string
    age  int
)
n, _ := fmt.Sscanf("polaris 28", "%s%d", &name, &age)
// 可以将"polaris 28"中的空格换成"\n"试试
// n, _ := fmt.Sscanf("polaris\n28", "%s%d", &name, &age)
fmt.Println(n, name, age)
```

输出：

```
2 polaris 28
```

如果将"空格"分隔改为"\n"分隔，则输出为：1 polaris 0。可见，`Scanf/FScanf/Sscanf` 这组函数将连续由空格分隔的值存储为连续的实参，其格式由 `format` 决定，换行符处停止扫描(Scan)。

3) Scanln/FScanln/Sscanln

```
var (
    name string
    age  int
)
n, _ := fmt.Sscanln("polaris 28", &name, &age)
// 可以将"polaris 28"中的空格换成"\n"试试
// n, _ := fmt.Sscanln("polaris\n28", &name, &age)
fmt.Println(n, name, age)
```

输出：

```
2 polaris 28
```

`Scanln/FScanln/Sscanln`表现和上一组一样，遇到"\n"停止（对于Scanln，表示从标准输入获取内容，最后需要回车）。

一般地，我们使用 `Scan/Scanf/Scanln` 这组函数。

提示

如果你是Windows系统，在使用 `Scanf` 时，有一个地方需要注意。看下面的代码：

```
for i := 0; i < 2; i++ {
    var name string
    fmt.Print("Input Name:")
    n, err := fmt.Scanf("%s", &name)
    fmt.Println(n, err, name)
}
```

编译、运行（或直接 go run ），输入：polaris回车。控制台内如下：

```
Input Name:polaris
1 <nil> polaris
Input Name:0 unexpected newline
```

为什么不是让输入两次？第二次好像有默认值一样。

同样的代码在Linux下正常。个人认为这是go在Windows下的一个bug，已经向官方提出：issue5391。

目前的解决方法是：换用Scanln或者改为Scanf("%s\n", &name)。

Scanner 和 ScanState 接口

基本上，我们不会去自己实现这两个接口，只需要使用上文中相应的Scan函数就可以了。这里只是简单的介绍一下这两个接口的作用。

任何实现了Scan方法的对象都实现了Scanner接口，Scan方法会从输入读取数据并将处理结果存入接收端，接收端必须是有效的指针。Scan方法会被任何Scan、Scanf、Scanln等函数调用，只要对应的参数实现了该方法。Scan方法接收的第一个参数为 `ScanState` 接口类型。

ScanState是一个交给用户定制的Scanner接口的参数的接口。Scanner接口可能会进行一次一个字符的扫描或者要求ScanState去探测下一个空白分隔的token。该接口的方法基本上在io包中都有讲解，这里不赘述。

在fmt包中，scan.go文件中的 ss 结构实现了 ScanState 接口。

导航

- 目录
- 上一节：ioutil — 方便的IO操作函数集
- 下一节：bufio — 缓存IO

1.4 bufio — 缓存IO

bufio 包实现了缓存IO。它包装了 io.Reader 和 io.Writer 对象，创建了另外的Reader和Writer对象，它们也实现了io.Reader和io.Writer接口，不过它们是有缓存的。该包同时为文本I/O提供了一些便利操作。

1.4.1 Reader 类型和方法

bufio.Reader 结构包装了一个 io.Reader 对象，提供缓存功能，同时实现了 io.Reader 接口。

Reader 结构没有任何导出的字段，结构定义如下：

```
type Reader struct {
    buf          []byte        // 缓存
    rd           io.Reader    // 底层的io.Reader
    // r:从buf中读走的字节（偏移）；w:buf中填充内容的偏移；
    // w - r 是buf中可被读的长度（缓存数据的大小），也是Buffered()方法的返回值
    r, w         int
    err          error        // 读过程中遇到的错误
    lastByte     int         // 最后一次读到的字节（ReadByte/UnreadByte)
    lastRuneSize int         // 最后一次读到的Rune的大小(ReadRune/UnreadRune)
}
```

1.4.1.1 实例化

bufio 包提供了两个实例化 bufio.Reader 对象的函数：NewReader 和 NewReaderSize。其中，NewReader 函数是调用 NewReaderSize 函数实现的：

```
func NewReader(rd io.Reader) *Reader {
    // 默认缓存大小：defaultBufSize=4096
    return NewReaderSize(rd, defaultBufSize)
}
```

我们看一下NewReaderSize的源码：

```
func NewReaderSize(rd io.Reader, size int) *Reader {
    // 已经是bufio.Reader类型，且缓存大小不小于 size，则直接返回
    b, ok := rd.(*Reader)
    if ok && len(b.buf) >= size {
        return b
    }
    // 缓存大小不会小于 minReadBufferSize （16字节）
    if size < minReadBufferSize {
        size = minReadBufferSize
    }
    // 构造一个bufio.Reader实例
    return &Reader{
        buf:          make([]byte, size),
        rd:           rd,
        lastByte:     -1,
        lastRuneSize: -1,
    }
}
```

1.4.1.2 ReadSlice、ReadBytes、ReadString 和 ReadLine 方法

之所以将这几个方法放在一起，是因为他们有着类似的行为。事实上，后三个方法最终都是调用ReadSlice来实现的。所以，我们先来看看ReadSlice方法。

ReadSlice方法签名如下：

```
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
```

ReadSlice从输入中读取，直到遇到第一个界定符（delim）为止，返回一个指向缓存中字节的slice，在下次调用读操作（read）时，这些字节会无效。举例说明：

```
reader := bufio.NewReader(strings.NewReader("http://studygolang.com. \nIt is the home
of gophers"))
line, _ := reader.ReadSlice('\n')
fmt.Printf("the line:%s\n", line)
// 这里可以换上任意的 bufio 的 Read/Write 操作
n, _ := reader.ReadSlice('\n')
fmt.Printf("the line:%s\n", line)
fmt.Println(string(n))
```

输出：

```
the line:http://studygolang.com.

the line:It is the home of gophers
It is the home of gophers
```

从结果可以看出，第一次ReadSlice的结果（line），在第二次调用读操作后，内容发生了变化。也就是说，ReadSlice返回的[]byte是指向Reader中的buffer，而不是copy一份返回。正因为ReadSlice返回的数据会被下次的I/O操作重写，因此许多的客户端会选择使用ReadBytes或者ReadString来代替。读者可以将上面代码中的ReadSlice改为ReadBytes或ReadString，看看结果有什么不同。

注意，这里的界定符可以是任意的字符，可以将上面代码中的'\n'改为'm'试试。同时，返回的结果是包含界定符本身的，上例中，输出结果有一空行就是'\n'本身。

如果ReadSlice在找到界定符之前遇到了error，它就会返回缓存中所有的数据和错误本身（经常是 io.EOF）。如果在找到界定符之前缓存已经满了，ReadSlice会返回bufio.ErrBufferFull 错误。当且仅当返回的结果（line）没有以界定符结束的时候，ReadSlice返回err != nil，也就是说，如果ReadSlice返回的结果line不是以界定符delim结尾，那么返回的err也一定不等于 nil（可能是bufio.ErrBufferFull或io.EOF）。例子代码：

```
reader := bufio.NewReaderSize(strings.NewReader("http://studygolang.com"),16)
line, err := reader.ReadSlice('\n')
fmt.Printf("line:%s\terror:%s\n", line, err)
line, err = reader.ReadSlice('\n')
fmt.Printf("line:%s\terror:%s\n", line, err)
```

输出：

```
line:http://studygola    error:bufio: buffer full
line:ng.com    error:EOF
```

ReadBytes方法签名如下：

```
func (b *Reader) ReadBytes(delim byte) (line []byte, err error)
```

该方法的参数和返回值类型与ReadSlice都一样。 ReadBytes 从输入中读取直到遇到界定符（delim）为止，返回的slice包含了从当前到界定符的内容（包括界定符）。如果ReadBytes在遇到界定符之前就捕获到一个错误，它会返回遇到错误之前已经读取的数据，和这个捕获到的错误（经常是 io.EOF）。跟ReadSlice一样，如果ReadBytes返回的结果line不是以界定符delim结尾，那么返回的err也一定不等于nil（可能是bufio.ErrBufferFull或io.EOF）。

从这个说明可以看出，ReadBytes和ReadSlice功能和用法都很像，那他们有什么不同呢？

在讲解ReadSlice时说到，它返回的[]byte是指向Reader中的buffer，而不是copy一份返回，也正因为如此，通常我们会使用ReadBytes或ReadString。很显然，ReadBytes返回的[]byte不会是指向Reader中的buffer，通过查看源码可以证实这一点。

还是上面的例子，我们将ReadSlice改为ReadBytes：

```
reader := bufio.NewReader(strings.NewReader("http://studygolang.com. \nIt is the home
of gophers"))
line, _ := reader.ReadBytes('\n')
fmt.Printf("the line:%s\n", line)
// 这里可以换上任意的 bufio 的 Read/Write 操作
n, _ := reader.ReadBytes('\n')
fmt.Printf("the line:%s\n", line)
fmt.Println(string(n))
```

输出：

```
the line:http://studygolang.com.

the line:http://studygolang.com.

It is the home of gophers
```

ReadString方法

看一下该方法的源码：

```
func (b *Reader) ReadString(delim byte) (line string, err error) {
    bytes, err := b.ReadBytes(delim)
    return string(bytes), err
}
```

它调用了ReadBytes方法，并将结果的[]byte转为string类型。

ReadLine方法签名如下

```
func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
```

ReadLine是一个底层的原始行读取命令。许多调用者或许会使用ReadBytes('\n')或者ReadString('\n')来代替这个方法。

ReadLine尝试返回单独的行，不包括行尾的换行符。如果一行大于缓存，isPrefix会被设置为true，同时返回该行的开始部分（等于缓存大小的部分）。该行剩余的部分就会在下次调用的时候返回。当下次调用返回该行剩余部分时，isPrefix将会是false。跟ReadSlice一样，返回

的line只是buffer的引用，在下次执行IO操作时，line会无效。可以将ReadSlice中的例子该为ReadLine试试。

注意，返回值中，要么line不是nil，要么err非nil，两者不会同时非nil。

ReadLine返回的文本不会包含行结尾（"\r\n"或者"\n"）。如果输入中没有行尾标识符，不会返回任何指示或者错误。

从上面的讲解中，我们知道，读取一行，通常会选择ReadBytes或ReadString。不过，正常人的思维，应该用ReadLine，只是不明白为啥ReadLine的实现不是通过ReadBytes，然后清除掉行尾的\n（或\r\n），它现在的实现，用不好会出现意想不到的问题，比如丢数据。个人建议可以这么实现读取一行：

```
line, err := reader.ReadBytes('\n')
line = bytes.TrimRight(line, "\r\n")
```

这样既读取了一行，也去掉了行尾结束符（当然，如果你希望不留行尾结束符，则直接用ReadBytes即可）。

1.4.1.3 Peek 方法

从方法的名称可以猜到，该方法只是“窥探”一下Reader中没有读取的n个字节。好比栈数据结构中的取栈顶元素，但不出栈。

方法的签名如下：

```
func (b *Reader) Peek(n int) ([]byte, error)
```

同上面介绍的ReadSlice一样，返回的[]byte只是buffer中的引用，在下次IO操作后会无效，可见该方法（以及ReadSlice这样的，返回buffer引用的方法）对多goroutine是不安全的，也就是在多并发环境下，不能依赖其结果。

我们通过例子来证明一下：

```
package main

import (
    "bufio"
    "fmt"
    "strings"
    "time"
)

func main() {
    reader := bufio.NewReaderSize(strings.NewReader("http://studygolang.com.\t It is t
he home of gophers"), 14)
    go Peek(reader)
    go reader.ReadBytes('\t')
    time.Sleep(1e8)
}

func Peek(reader *bufio.Reader) {
    line, _ := reader.Peek(14)
    fmt.Printf("%s\n", line)
    // time.Sleep(1)
    fmt.Printf("%s\n", line)
}
```

输出：

```
http://studygo
http://studygo
```

输出结果和预期的一致。然而，这是由于目前的goroutine调度方式导致的结果。如果我们将例子中注释掉的time.Sleep(1)取消注释（这样调度其他goroutine执行），再次运行，得到的结果为：

```
http://studygo
ng.com.     It is
```

另外，Reader的Peek方法如果返回的[]byte长度小于n，这时返回的err为非nil，用于解释为啥会小于n。如果n大于reader的buffer长度，err会是ErrBufferFull。

1.4.1.4 其他方法

Reader的其他方法都是实现了io包中的接口，它们的使用方法在io包中都有介绍，在此不赘述。

这些方法包括：

```
func (b *Reader) Read(p []byte) (n int, err error)
func (b *Reader) ReadByte() (c byte, err error)
func (b *Reader) ReadRune() (r rune, size int, err error)
func (b *Reader) UnreadByte() error
func (b *Reader) UnreadRune() error
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
```

你应该知道它们都是哪个接口的方法吧。

1.4.2 Scanner 类型和方法

对于简单的读取一行，在Reader类型中，感觉没有让人特别满意的方法。于是，Go1.1增加了一个类型：Scanner。官方关于**Go1.1**增加该类型的说明如下：

> 在 bufio 包中有多种方式获取文本输入，ReadBytes、ReadString 和独特的 ReadLine，对于简单的目的这些都有些过于复杂了。在 Go 1.1 中，添加了一个新类型，Scanner，以便更容易的处理如按行读取输入序列或空格分隔的词等，这类简单的任务。它终结了如输入一个很长的有问题的行这样的输入错误，并且提供了简单的默认行为：基于行的输入，每行都剔除分隔标识。这里的代码展示一次输入一行：

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text()) // Println will add back the final '\n'
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading standard input:", err)
}
```

> 输入的行为可以通过一个函数控制，来控制输入的每个部分（参阅 SplitFunc 的文档），但是对于复杂的问题或持续传递错误的，可能还是需要原有接口。

Scanner 类型和 Reader 类型一样，没有任何导出的字段，同时它也包装了一个 io.Reader 对象，但它没有实现 io.Reader 接口。

Scanner 的结构定义如下：

```
type Scanner struct {
    r            io.Reader // The reader provided by the client.
    split        SplitFunc // The function to split the tokens.
    maxTokenSize int       // Maximum size of a token; modified by tests.
    token        []byte    // Last token returned by split.
    buf          []byte    // Buffer used as argument to split.
    start        int       // First non-processed byte in buf.
    end          int       // End of data in buf.
    err          error     // Sticky error.
}
```

这里 split、maxTokenSize 和 token 需要讲解一下。

然而，在讲解之前，需要先讲解 split 字段的类型 SplitFunc。

1.4.2.1 SplitFunc 类型和实例

SplitFunc 类型定义如下：

```
type SplitFunc func(data []byte, atEOF bool) (advance int, token []byte, err error)
```

SplitFunc 定义了 用于对输入进行分词的 split 函数的签名。参数 data 是还未处理的数据，atEOF 标识 Reader 是否还有更多数据（是否到了EOF）。返回值 advance 表示从输入中读取的字节数，token 表示下一个结果数据，err 则代表可能的错误。

举例说明一下这里的 token 代表的意思：

```
有数据 "studygolang\tpolaris\tgolangchina"，通过"\t"进行分词，那么会得到三个token，它们的内容分别是：studygolang、polaris 和 golangchina。而 SplitFunc 的功能是：进行分词，并返回未处理的数据中第一个 token。对于这个数据，就是返回 studygolang。
```

如果 data 中没有一个完整的 token，例如，在扫描行（scanning lines）时没有换行符，SplitFunc 会返回(0,nil,nil)通知 Scanner 读取更多数据到 slice 中，然后在这个更大的 slice 中同样的读取点处，从输入中重试读取。如下面要讲解的 split 函数的源码中有这样的代码：

```
// Request more data.
return 0, nil, nil
```

如果 err 非nil，扫描停止，同时该错误会返回。

如果参数 data 为空的 slice，除非 atEOF 为 true，否则该函数永远不会被调用。如果 atEOF 为 true，这时 data 可以非空，这时的数据是没有处理的。

bufio 包定义的 **split** 函数，即 **SplitFunc** 的实例

在 bufio 包中预定义了一些 split 函数，也就是说，在 Scanner 结构中的 split 字段，可以通过这些预定义的 split 赋值，同时 Scanner 类型的 Split 方法也可以接收这些预定义函数作为参数。所以，我们可以说，这些预定义 split 函数都是 SplitFunc 类型的实例。这些函数包括：ScanBytes、ScanRunes、ScanWords 和 ScanLines。（由于都是 SplitFunc 的实例，自然这些函数的签名都和 SplitFunc 一样）

ScanBytes 返回单个字节作为一个 token。

ScanRunes 返回单个 UTF-8 编码的 rune 作为一个 token。返回的 rune 序列（token）和 range string类型 返回的序列是等价的，也就是说，对于无效的 UTF-8 编码会解释为 U+FFFD = "\xef\xbf\xbd"。

ScanWords 返回通过“空格”分词的单词。如：study golang，调用会返回study。注意，这里的“空格”是 `unicode.IsSpace()`，即包括：'\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+00A0 (NBSP)。

ScanLines 返回一行文本，不包括行尾的换行符。这里的换行包括了Windows下的"\r\n"和Unix下的"\n"。

一般地，我们不会单独使用这些函数，而是提供给 Scanner 实例使用。现在我们回到 Scanner 的 split、maxTokenSize 和 token 字段上来。

split 字段（SplitFunc 类型实例），很显然，代表了当前 Scanner 使用的分词策略，可以使用上面介绍的预定义 SplitFunc 实例赋值，也可以自定义 SplitFunc 实例。（当然，要给 split 字段赋值，必须调用 Scanner 的 Split 方法）

maxTokenSize 字段 表示通过 split 分词后的一个 token 允许的最大长度。在该包中定义了一个常量 MaxScanTokenSize = 64 * 1024，这是允许的最大 token 长度（64k）。

token 字段 上文已经解释了这个是什么意思。

1.4.2.2 Scanner 的实例化

Scanner 没有导出任何字段，而它需要有外部的 io.Reader 对象，因此，我们不能直接实例化 Scanner 对象，必须通过 bufio 包提供的实例化函数来实例化。实例化函数签名以及内部实现：

```
func NewScanner(r io.Reader) *Scanner {
    return &Scanner{
        r:            r,
        split:        ScanLines,
        maxTokenSize: MaxScanTokenSize,
        buf:          make([]byte, 4096), // Plausible starting size; needn't be large
.
    }
}
```

可见，返回的 Scanner 实例默认的 split 函数是 ScanLines。

1.4.2.2 Scanner 的方法

Split 方法 前面我们提到过可以通过 Split 方法为 Scanner 实例设置分词行为。由于 Scanner 实例的默认 split 总是 ScanLines，如果我们想要用其他的 split，可以通过 Split 方法做到。

比如，我们想要统计一段英文有多少个单词（不排除重复），我们可以这么做：

```
const input = "This is The Golang Standard Library.\nWelcome you!"
scanner := bufio.NewScanner(strings.NewReader(input))
scanner.Split(bufio.ScanWords)
count := 0
for scanner.Scan() {
    count++
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading input:", err)
}
fmt.Println(count)
```

输出：

```
8
```

我们实例化 Scanner 后，通过调用 scanner.Split(bufio.ScanWords) 来更改 split 函数。注意，我们应该在调用 Scan 方法之前调用 Split 方法。

Scan 方法 该方法好比 iterator 中的 Next 方法，它用于将 Scanner 获取下一个 token，以便 Bytes 和 Text 方法可用。当扫描停止时，它返回false，这时候，要么是到了输入的末尾要么是遇到了一个错误。注意，当 Scan 返回 false 时，通过 Err 方法可以获取第一个遇到的错误（但如果错误是 io.EOF，Err 方法会返回 nil）。

Bytes 和 Text 方法 这两个方法的行为一致，都是返回最近的 token，无非 Bytes 返回的是 []byte，Text 返回的是 string。该方法应该在 Scan 调用后调用，而且，下次调用 Scan 会覆盖这次的 token。比如：

```
scanner := bufio.NewScanner(strings.NewReader("http://studygolang.com. \nIt is the hom
e of gophers"))
if scanner.Scan() {
    scanner.Scan()
    fmt.Printf("%s", scanner.Text())
}
```

返回的是：It is the home of gophers 而不是 http://studygolang.com.

Err 方法 前面已经提到，通过 Err 方法可以获取第一个遇到的错误（但如果错误是 io.EOF，Err 方法会返回 nil）。

下面，我们通过一个完整的示例来演示 Scanner 类型的使用。

1.4.2.3 Scanner 使用示例

我们经常会有这样的需求：读取文件中的数据，一次读取一行。在学习了 Reader 类型，我们可以使用它的 ReadBytes 或 ReadString来实现，甚至使用 ReadLine 来实现。然而，在 Go1.1 中，我们可以使用 Scanner 来做这件事，而且更简单好用。

```
file, err := os.Create("scanner.txt")
if err != nil {
    panic(err)
}
defer file.Close()
file.WriteString("http://studygolang.com.\nIt is the home of gophers.\nIf you are stud
ying golang, welcome you!")
// 将文件 offset 设置到文件开头
file.Seek(0, os.SEEK_SET)
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
```

输出结果：

```
http://studygolang.com.
It is the home of gophers.
If you are studying golang, welcome you!
```

1.4.3 Writer 类型和方法

bufio.Writer 结构包装了一个 io.Writer 对象，提供缓存功能，同时实现了 io.Writer 接口。

Writer 结构没有任何导出的字段，结构定义如下：

```
type Writer struct {
    err error        // 写过程中遇到的错误
    buf []byte        // 缓存
    n   int            // 当前缓存中的字节数
    wr  io.Writer    // 底层的 io.Writer 对象
}
```

相比 bufio.Reader, bufio.Writer 结构定义简单很多。

注意：如果在写数据到 Writer 的时候出现了一个错误，不会再允许有数据被写进来了，并且所有随后的写操作都会返回该错误。

1.4.3.1 实例化

和 Reader 类型一样，bufio 包提供了两个实例化 bufio.Writer 对象的函数：NewWriter 和 NewWriterSize。其中，NewWriter 函数是调用 NewWriterSize 函数实现的：

```
func NewWriter(wr io.Writer) *Writer {
    // 默认缓存大小：defaultBufSize=4096
    return NewWriterSize(wr, defaultBufSize)
}
```

我们看一下 NewWriterSize 的源码：

```
func NewWriterSize(wr io.Writer, size int) *Writer {
    // 已经是 bufio.Writer 类型，且缓存大小不小于 size，则直接返回
    b, ok := wr.(*Writer)
    if ok && len(b.buf) >= size {
        return b
    }
    if size <= 0 {
        size = defaultBufSize
    }
    // new 一个 bufio.Writer 实例
    b = new(Writer)
    b.buf = make([]byte, size)
    b.wr = wr
    return b
}
```

题外话

对比 Reader 和 Writer 的实例化函数，发现它们很相似。然而，在实际实例化时，两者用了不同的方式：&Reader{} 和 new(Writer)，让人感觉这不是同一个人写的。

1.4.3.2 Available 和 Buffered 方法

Available 方法获取缓存中还未使用的字节数（缓存大小 - 字段 n 的值）；Buffered 方法获取写入当前缓存中的字节数（字段 n 的值）

1.4.3.3 Flush 方法

该方法将缓存中的所有数据写入底层的 io.Writer 对象中。使用 bufio.Writer 时，在所有的 Write 操作完成之后，应该调用 Flush 方法使得缓存都写入 io.Writer 对象中。

1.4.3.4 其他方法

Writer 类型其他方法是一些实际的写方法：

```
// 实现了 io.ReaderFrom 接口
func (b *Writer) ReadFrom(r io.Reader) (n int64, err error)

// 实现了 io.Writer 接口
func (b *Writer) Write(p []byte) (nn int, err error)

// 实现了 io.ByteWriter 接口
func (b *Writer) WriteByte(c byte) error

// io 中没有该方法的接口，它用于写入单个 Unicode 码点，返回写入的字节数（码点占用的字节），内部实现
会根据当前 rune 的范围调用 WriteByte 或 WriteString
func (b *Writer) WriteRune(r rune) (size int, err error)

// 写入字符串，如果返回写入的字节数比 len(s) 小，返回的error会解释原因
func (b *Writer) WriteString(s string) (int, error)
```

这些写方法在缓存满了时会调用 Flush 方法。另外，这些写方法源码开始处，有这样的代码：

```
if b.err != nil {
    return b.err
}
```

也就是说，只要写的过程中遇到了错误，再次调用写操作会直接返回该错误。

1.4.4 ReadWriter 类型和实例化

ReadWriter 结构存储了 bufio.Reader 和 bufio.Writer 类型的指针（内嵌），它实现了 io.ReadWriter 结构。

```
type ReadWriter struct {
    *Reader
    *Writer
}
```

ReadWriter 的实例化可以跟普通结构类型一样，也可以通过调用 bufio.NewReadWriter 函数来实现：只是简单的实例化 ReadWriter

```
func NewReadWriter(r *Reader, w *Writer) *ReadWriter {
    return &ReadWriter{r, w}
}
```

导航

- 目录
- 上一节：fmt — 格式化IO
- 下一节：I/O 总结

第二章 文本

几乎任何程序都离不开文本（字符串）。Go 中 string 是内置类型，同时它与普通的 slice 类型有着相似的性质，例如，可以进行切片（slice）操作，这使得 Go 中少了一些处理 string 类型的函数，比如没有 substring 这样的函数，然而却能够很方便的进行这样的操作。除此之外，Go 标准库中有几个包专门用于处理文本。

strings 包提供了很多操作字符串的简单函数，通常一般的字符串操作需求都可以在这个包中找到。

strconv 包提供了基本数据类型和字符串之间的转换。在 Go 中，没有隐式类型转换，一般的类型转换可以这么做：int32(i)，将 i（比如为 int 类型）转换为 int32，然而，字符串类型和 int、float、bool 等类型之间的转换却没有这么简单。

进行复杂的文本处理必然离不开正则表达式。*regexp* 包提供了正则表达式功能，它的语法基于 RE2 ，*regexp/syntax* 子包进行正则表达式解析。

Go 代码使用 UTF-8 编码（且不能带 BOM），同时标识符支持 Unicode 字符。在标准库 *unicode* 包及其子包 utf8、utf16中，提供了对 Unicode 相关编码、解码的支持，同时提供了测试 Unicode 码点（Unicode code points）属性的功能。

在开发过程中，可能涉及到字符集的转换，作为补充，本章最后会讲解一个第三方库：mahonia — 纯 Go 语言实现的字符集转换库，以方便需要进行字符集转换的读者。

导航

- 第一章
- 下一节：strings — 字符串操作

2.1 strings — 字符串操作

字符串常见操作有：

- 字符串长度；
- 求子串；
- 是否存在某个字符或子串；
- 子串出现的次数（字符串匹配）；
- 字符串分割（切分）为[]string；
- 字符串是否有某个前缀或后缀；
- 字符或子串在字符串中首次出现的位置或最后一次出现的位置；
- 通过某个字符串将[]string连接起来；
- 字符串重复几次；
- 字符串中子串替换；
- 大小写转换；
- Trim操作；
- ...

前面已经说过，由于string类型可以看成是一种特殊的slice类型，因此获取长度可以用内置的函数len；同时支持 切片 操作，因此，子串获取很容易。

其他的字符串常见操作就是我们这小节要介绍的，由于这些操作函数的使用比较简单，只会对某些函数举例说明；但会深入这些函数的内部实现，更好的掌握它们。

说明：这里说的字符，值得是 rune 类型，即一个 UTF-8 字符（Unicode 代码点）。

2.1.1 是否存在某个字符或子串

有三个函数做这件事：

```
// 子串substr在s中，返回true
func Contains(s, substr string) bool
// chars中任何一个Unicode代码点在s中，返回true
func ContainsAny(s, chars string) bool
// Unicode代码点r在s中，返回true
func ContainsRune(s string, r rune) bool
```

这里对 ContainsAny 函数进行一下说明，看如下例子：

```
fmt.Println(strings.ContainsAny("team", "i"))
fmt.Println(strings.ContainsAny("failure", "u & i"))
fmt.Println(strings.ContainsAny("in failure", "s g"))
fmt.Println(strings.ContainsAny("foo", ""))
fmt.Println(strings.ContainsAny("", ""))
```

输出：

```
false
true
true
false
false
```

也就是说，第二个参数 chars 中任意一个字符（Unicode Code Point）如果在第一个参数 s 中存在，则返回true。

查看这三个函数的源码，发现它们只是调用了相应的Index函数（子串出现的位置），然后和 0 作比较返回true或fale。如，Contains：

```
func Contains(s, substr string) bool {
    return Index(s, substr) >= 0
}
```

关于Index相关函数的实现，我们后面介绍。

2.1.2 子串出现次数(字符串匹配)

在数据结构与算法中，可能会讲解以下字符串匹配算法：

- 朴素匹配算法
- KMP算法
- Rabin-Karp算法
- Boyer-Moore算法

还有其他的算法，这里不一一列举，感兴趣的可以网上搜一下。

在Go中，查找子串出现次数即字符串模式匹配，实现的是Rabin-Karp算法。Count 函数的签名如下：

```
func Count(s, sep string) int
```

在 Count 的实现中，处理了几种特殊情况，属于字符匹配预处理的一部分。这里要特别说明一下的是当 sep 为空时，Count 的返回值是：utf8.RuneCountInString(s) + 1

```
fmt.Println(strings.Count("five", "")) // before & after each rune
```

输出：

```
5
```

关于Rabin-Karp算法的实现，有兴趣的可以看看 Count 的源码。

另外，Count 是计算子串在字符串中出现的无重叠的次数，比如：

```
fmt.Println(strings.Count("fivevev", "vev"))
```

输出：

```
1
```

2.1.3 字符串分割为[]string

这个需求很常见，倒不一定是为了得到[]string。

该包提供了六个三组分割函数：Fields 和 FieldsFunc、Split 和 SplitAfter、SplitN 和 SplitAfterN。

2.1.3.1 Fields 和 FieldsFunc

这两个函数的签名如下：

```
func Fields(s string) []string
func FieldsFunc(s string, f func(rune) bool) []string
```

Fields 用一个或多个连续的空格分隔字符串 s，返回子字符串的数组（slice）。如果字符串 s 只包含空格，则返回空列表([]string的长度为0）。其中，空格的定义是 unicode.IsSpace，之前已经介绍过。

由于是用空格分隔，因此结果中不会含有空格或空子字符串，例如：

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz   "))
```

输出：

```
Fields are: ["foo" "bar" "baz"]
```

FieldsFunc 用这样的Unicode代码点 c 进行分隔：满足 f(c) 返回 true。该函数返回[]string。如果字符串 s 中所有的代码点(unicode code points)都满足f(c)或者 s 是空，则 FieldsFunc 返回空slice。

也就是说，我们可以通过实现一个回调函数来指定分隔字符串 s 的字符。比如上面的例子，我们通过 FieldsFunc 来实现：

```
fmt.Println(strings.FieldsFunc("  foo bar  baz   ", unicode.IsSpace))
```

实际上，Fields 函数就是调用 FieldsFunc 实现的：

```
func Fields(s string) []string {
    return FieldsFunc(s, unicode.IsSpace)
}
```

对于 FieldsFunc 源码留给读者自己阅读。

2.1.3.2 Split 和 SplitAfter、 SplitN 和 SplitAfterN

之所以将这四个函数放在一起讲，是因为它们都是通过一个同一个内部函数来实现的。它们的函数签名及其实现：

```
func Split(s, sep string) []string { return genSplit(s, sep, 0, -1) }
func SplitAfter(s, sep string) []string { return genSplit(s, sep, len(sep), -1) }
func SplitN(s, sep string, n int) []string { return genSplit(s, sep, 0, n) }
func SplitAfterN(s, sep string, n int) []string { return genSplit(s, sep, len(sep), n)
 }
```

它们都调用了 genSplit 函数。

这四个函数都是通过 sep 进行分割，返回[]string。如果 sep 为空，相当于分成一个个的 UTF-8 字符，如 `Split("abc","")` ，得到的是[a b c]。

Split(s, sep) 和 SplitN(s, sep, -1) 等价；SplitAfter(s, sep) 和 SplitAfterN(s, sep, -1) 等价。

那么，Split 和 SplitAfter 有啥区别呢？通过这两句代码的结果就知道它们的区别了：

```
fmt.Printf("%q\n", strings.Split("foo,bar,baz", ","))
fmt.Printf("%q\n", strings.SplitAfter("foo,bar,baz", ","))
```

输出：

```
["foo" "bar" "baz"]
["foo," "bar," "baz"]
```

也就是说，Split 会将 s 中的 sep 去掉，而 SplitAfter 会保留 sep。

带 N 的方法可以通过最后一个参数 n 控制返回的结果中的 slice 中的元素个数，当 n < 0 时，返回所有的子字符串；当 n == 0 时，返回的结果是 nil；当 n > 0 时，表示返回的 slice 中最多只有 n 个元素，其中，最后一个元素不会分割，比如：

```
fmt.Printf("%q\n", strings.SplitN("foo,bar,baz", ",", 2))
```

输出：

```
["foo" "bar,baz"]
```

另外看一下官方文档提供的例子，注意一下输出结果：

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
```

输出：

```
["a" "b" "c"]
["" "man " "plan " "canal panama"]
[" " "x" "y" "z" " "]
[""]
```

2.1.4 字符串是否有某个前缀或后缀

这两个函数比较简单，源码如下：

```
// s 中是否以 prefix 开始
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[0:len(prefix)] == prefix
}
// s 中是否以 suffix 结尾
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

2.1.5 字符或子串在字符串中出现的位置

有一序列函数与该功能有关：

```
// 在 s 中查找 sep 的第一次出现，返回第一次出现的索引
func Index(s, sep string) int
// chars中任何一个Unicode代码点在s中首次出现的位置
func IndexAny(s, chars string) int
// 查找字符 c 在 s 中第一次出现的位置，其中 c 满足 f(c) 返回 true
func IndexFunc(s string, f func(rune) bool) int
// Unicode 代码点 r 在 s 中第一次出现的位置
func IndexRune(s string, r rune) int

// 有三个对应的查找最后一次出现的位置
func LastIndex(s, sep string) int
func LastIndexAny(s, chars string) int
func LastIndexFunc(s string, f func(rune) bool) int
```

在 2.1.1 小节提到过，Contain 相关的函数内部调用的是响应的 Index 函数。

这一序列函数，只举 IndexFunc 的例子：

```
fmt.Printf("%d\n", strings.IndexFunc("studygolang", func(c rune) bool {
    if c > 'u' {
        return true
    }
    return false
}))
```

输出：

```
4
```

因为 y 的 Unicode 代码点大于 u 的代码点。

2.1.6 字符串 JOIN 操作

将字符串数组（或slice）连接起来可以通过 Join 实现，函数签名如下：

```
func Join(a []string, sep string) string
```

假如没有这个库函数，我们自己实现一个，我们会这么实现：

```
func Join(str []string, sep string) string {
    // 特殊情况应该做处理
    if len(str) == 0 {
        return ""
    }
    if len(str) == 1 {
        return str[0]
    }
    buffer := bytes.NewBufferString(str[0])
    for _, s := range str[1:] {
        buffer.WriteString(sep)
        buffer.WriteString(s)
    }
    return buffer.String()
}
```

这里，我们使用了 bytes 包的 Buffer 类型，避免大量的字符串连接操作（因为 Go 中字符串是不可变的）。我们再看一下标准库的实现：

```
func Join(a []string, sep string) string {
    if len(a) == 0 {
        return ""
    }
    if len(a) == 1 {
        return a[0]
    }
    n := len(sep) * (len(a) - 1)
    for i := 0; i < len(a); i++ {
        n += len(a[i])
    }

    b := make([]byte, n)
    bp := copy(b, a[0])
    for _, s := range a[1:] {
        bp += copy(b[bp:], sep)
        bp += copy(b[bp:], s)
    }
    return string(b)
}
```

标准库的实现没有用 bytes 包，当然也不会简单的通过 + 号连接字符串。Go 中是不允许循环依赖的，标准库中很多时候会出现代码拷贝，而不是引入某个包。这里 Join 的实现方式挺好，我个人猜测，不直接使用 bytes 包，也是不想依赖 bytes 包（其实 bytes 中的实现也是 copy 方式）。

简单使用示例：

```
fmt.Println(Join([]string{"name=xxx", "age=xx"}, "&"))
// 输出 name=xxx&age=xx
```

2.1.7 字符串重复几次

函数签名如下：

```
func Repeat(s string, count int) string
```

这个函数使用很简单：

```
// 输出 banana
fmt.Println("ba" + strings.Repeat("na", 2))
```

2.1.8 字符串子串替换

进行字符串替换时，考虑到性能问题，能不用正则尽量别用，应该用这里的函数。

字符串替换的函数签名如下：

```
// 用 new 替换 s 中的 old，一共替换 n 个。
// 如果 n < 0，则不限制替换次数，即全部替换
func Replace(s, old, new string, n int) string
```

使用示例：

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))
fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
```

输出：

```
oinky oinky oink
moo moo moo
```

如果我们希望一次替换多个，比如我们希望替换 `This is <b>HTML</b>` 中的 `<` 和 `>` 为 `<` 和 `>`，可以调用上面的函数两次。但标准库提供了另外的方法进行这种替换。

2.1.9 Replacer 类型

这是一个结构，没有导出任何字段，实例化通过 `func NewReplacer(oldnew ...string) *Replacer` 函数进行，其中不定参数 oldnew 是 old-new 对，即进行多个替换。

解决上面说的替换问题：

```
r := strings.NewReplacer("<", "&lt;", ">", "&gt;")
fmt.Println(r.Replace("This is <b>HTML</b>!"))
```

另外，Replacer 还提供了另外一个方法：

```
func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
```

它在替换之后将结果写入 io.Writer 中。

2.1.10 Reader 类型

看到名字就能猜到，这是实现了 `io` 包中的接口。它实现了 io.Reader（Read 方法），io.ReaderAt（ReadAt 方法），io.Seeker（Seek 方法），io.WriterTo（WriteTo 方法），io.ByteReader（ReadByte 方法），io.ByteScanner（ReadByte 和 UnreadByte 方法），io.RuneReader（ReadRune 方法）和 io.RuneScanner（ReadRune 和 UnreadRune 方法）。

Reader 结构如下：

```
type Reader struct {
    s        string    // Reader 读取的数据来源
    i        int // current reading index（当前读的索引位置）
    prevRune int // index of previous rune; or < 0（前一个读取的 rune 索引位置）
}
```

可见 Reader 结构没有导出任何字段，而是提供一个实例化方法：

```
func NewReader(s string) *Reader
```

该方法接收一个字符串，返回的 Reader 实例就是从该参数字符串读数据。在后面学习了 bytes 包之后，可以知道 bytes.NewBufferString 有类似的功能，不过，如果只是为了读取，NewReader 会更高效。

其他方法不介绍了，都是之前接口的实现，有兴趣的可以看看源码实现，大部分都是根据 i、prevRune 两个属性来控制。

导航

- 第二章 文本
- 下一节：bytes — byte slice 便利操作

2.2 bytes — byte slice 便利操作

该包定义了一些操作 byte slice 的便利操作。因为字符串可以表示为 []byte，因此，bytes 包定义的函数、方法等和 strings 包很类似，所以讲解时会和 strings 包类似甚至可以直接参考。

说明：为了方便，会称呼 []byte 为 字节数组

2.2.1 是否存在某个子slice

```
// 子slice subslice 在 b 中，返回 true
func Contains(b, subslice []byte) bool
```

该函数的内部调用了 bytes.Index 函数（在后面会讲解）：

```
func Contains(b, subslice []byte) bool {
    return Index(b, subslice) != -1
}
```

题外：对比 `strings.Contains` 你会发现，一个判断 `>=0`，一个判断 `!= -1`，可见库不是一个人写的，没有做到一致性。

2.2.2 []byte 出现次数

```
// slice sep 在 s 中出现的次数（无重叠）
func Count(s, sep []byte) int
```

和 strings 实现不同，此包中的 Count 核心代码如下：

```
count := 0
c := sep[0]
i := 0
t := s[:len(s)-n+1]
for i < len(t) {
    // 判断 sep 第一个字节是否在 t[i:] 中
    // 如果在，则比较之后相应的字节
    if t[i] != c {
        o := IndexByte(t[i:], c)
        if o < 0 {
            break
        }
        i += o
    }
    // 执行到这里表示 sep[0] == t[i]
    if n == 1 || Equal(s[i:i+n], sep) {
        count++
        i += n
        continue
    }
    i++
}
```

2.2.3 - 7 参见 strings 包对应的部分

2.2.4

导航

- 上一节：strings — 字符串操作
- 下一节：strconv — 字符串和基本数据类型之间转换

2.3 strconv — 字符串和基本数据类型之间转换

这里的基本数据类型包括：布尔、整型（包括有/无符号、二进制、八进制、十进制和十六进制）和浮点型等。

2.3.1 strconv 包转换错误处理

介绍具体的转换之前，先看看 *strconv* 中的错误处理。

由于将字符串转为其他数据类型可能会出错，*strconv* 包定义了两个 *error* 类型的变量：*ErrRange* 和 *ErrSyntax*。其中，*ErrRange* 表示值超过了类型能表示的最大范围，比如将 "128" 转为 int8 就会返回这个错误；*ErrSyntax* 表示语法错误，比如将 "" 转为 int 类型会返回这个错误。

然而，在返回错误的时候，不是直接将上面的变量值返回，而是通过构造一个 *NumError* 类型的 *error* 对象返回。*NumError* 结构的定义如下：

```
// A NumError records a failed conversion.
type NumError struct {
    Func string // the failing function (ParseBool, ParseInt, ParseUint, ParseFloat)
    Num  string // the input
    Err  error  // the reason the conversion failed (ErrRange, ErrSyntax)
}
```

可见，该结构记录了转换过程中发生的错误信息。该结构不仅包含了一个 *error* 类型的成员，记录具体的错误信息，而且它自己也实现了 *error* 接口：

```
func (e *NumError) Error() string {
    return "strconv." + e.Func + ": " + "parsing " + Quote(e.Num) + ": " + e.Err.Error
()
}
```

包的实现中，定义了两个便捷函数，用于构造 *NumError* 对象：

```
func syntaxError(fn, str string) *NumError {
    return &NumError{fn, str, ErrSyntax}
}

func rangeError(fn, str string) *NumError {
    return &NumError{fn, str, ErrRange}
}
```

在遇到 *ErrSyntax* 或 *ErrRange* 错误时，通过上面的函数构造 *NumError* 对象。

2.3.2 字符串和整型之间的转换

2.3.2.1 字符串转为整型

包括三个函数：ParseInt、ParseUint 和 Atoi，函数原型如下：

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
func ParseUint(s string, base int, bitSize int) (n uint64, err error)
func Atoi(s string) (i int, err error)
```

其中，Atoi 是 ParseInt 的便捷版，内部通过调用 *ParseInt(s, 10, 0)* 来实现的；ParseInt 转为有符号整型；ParseUint 转为无符号整型，着重介绍 ParseInt。

参数 *base* 代表字符串按照给定的进制进行解释。一般的，base 的取值为 2~36，如果 base 的值为 0，则会根据字符串的前缀来确定 base 的值："0x" 表示 16 进制；"0" 表示 8 进制；否则就是 10 进制。

参数 *bitSize* 表示的是整数取值范围，或者说整数的具体类型。取值 0、8、16、32 和 64 分别代表 int、int8、int16、int32 和 int64。

这里有必要说一下，当 bitSize==0 时的情况。

Go中，int/uint 类型，不同系统能表示的范围是不一样的，目前的实现是，32 位系统占 4 个字节；64 位系统占 8 个字节。当 bitSize==0 时，应该表示 32 位还是 64 位呢？这里没有利用 *runtime.GOARCH* 之类的方式，而是巧妙的通过如下表达式确定 intSize：

```
const intSize = 32 << uint(^uint(0)>>63)
const IntSize = intSize // number of bits in int, uint (32 or 64)
```

主要是 *^uint(0)>>63* 这个表达式。操作符 ^ 在这里是一元操作符 按位取反，而不是 按位异或。更多解释可以参考：Go位运算：取反和异或。

问题：下面的代码 n 和 err 的值分别是什么？

```
n, err := strconv.ParseInt("128", 10, 8)
```

在 *ParseInt/ParseUint* 的实现中，如果字符串表示的整数超过了 bitSize 参数能够表示的范围，则会返回 ErrRange，同时会返回 bitSize 能够表示的最大或最小值。因此，这里的 n 是 127。

另外，*ParseInt* 返回的是 int64，这是为了能够容纳所有的整型，在实际使用中，可以根据传递的 bitSize，然后将结果转为实际需要的类型。

转换的基本原理（以 "128" 转 为 10 进制 int 为例）：

```
s := "128"
n := 0
for i := 0; i < len(s); i++ {
    n *= 10    + s[i]     // base
}
```

在循环处理的过程中，会检查数据的有效性和是否越界等。

2.3.2.2 整型转为字符串

实际应用中，我们经常会遇到需要将字符串和整型连接起来，在Java中，可以通过操作符 "+" 做到。不过，在Go语言中，你需要将整型转为字符串类型，然后才能进行连接。这个时候，*strconv* 包中的整型转字符串的相关函数就派上用场了。这些函数签名如下：

```
func FormatUint(i uint64, base int) string    // 无符号整型转字符串
func FormatInt(i int64, base int) string    // 有符号整型转字符串
func Itoa(i int) string
```

其中，*Itoa* 内部直接调用 *FormatInt(i, 10)* 实现的。base 参数可以取 2~36（0-9，a-z）。

转换的基本原理（以 10 进制的 127 转 string 为例）：

```
const digits = "0123456789abcdefghijklmnopqrstuvwxyz"
u := uint64(127)
var a [65]byte
i := len(a)
b := uint64(base)
for u >= b {
    i--
    a[i] = digits[uintptr(u%b)]
    u /= b
}
i--
a[i] = digits[uintptr(u)]
return string(a[1:])
```

即将整数每一位数字对应到相应的字符，存入字符数组中，最后字符数组转为字符串即为结果。

具体实现时，当 base 是 2 的幂次方时，有优化处理（移位和掩码）；十进制也做了优化。

标准库还提供了另外两个函数：*AppendInt* 和 *AppendUint*，这两个函数不是将整数转为字符串，而是将整数转为字符数组 append 到目标字符数组中。（最终，我们也可以通过返回的 []byte 得到字符串）

除了使用上述方法将整数转为字符串外，经常见到有人使用 *fmt* 包来做这件事。如：

```
fmt.Sprintf("%d", 127)
```

那么，这两种方式我们该怎么选择呢？我们主要来考察一下性能。

```
startTime := time.Now()
for i := 0; i < 10000; i++ {
    fmt.Sprintf("%d", i)
}
fmt.Println(time.Now().Sub(startTime))

startTime := time.Now()
for i := 0; i < 10000; i++ {
    strconv.Itoa(i)
}
fmt.Println(time.Now().Sub(startTime))
```

我们分别循环转换了10000次。*Sprintf* 的时间是 3.549761ms，而 *Itoa* 的时间是 848.208us，相差 4 倍多。

Sprintf 性能差些可以预见，因为它接收的是 interface，需要进行反射等操作。个人建议使用 *strconv* 包中的方法进行转换。

注意：别想着通过 string(65) 这种方式将整数转为字符串，这样实际上得到的会是 ASCCII 值为 65 的字符，即 'A'。

思考：

```
给定一个 40 以内的正整数，如何快速判断其是否是 2 的幂次方？
```

提示：在 *strconv* 包源码 *itoa.go* 文件中找答案

2.3.3 字符串和布尔值之间的转换

Go中字符串和布尔值之间的转换比较简单，主要有三个函数：

```
// 接受 1, t, T, TRUE, true, True, 0, f, F, FALSE, false, False 等字符串；
// 其他形式的字符串会返回错误
func ParseBool(str string) (value bool, err error)
// 直接返回 "true" 或 "false"
func FormatBool(b bool) string
// 将 "true" 或 "false" append 到 dst 中
// 这里用了一个 append 函数对于字符串的特殊形式：append(dst, "true"...)
func AppendBool(dst []byte, b bool)
```

2.3.4 字符串和浮点数之间的转换

类似的，包含三个函数：

```
func ParseFloat(s string, bitSize int) (f float64, err error)
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
func AppendFloat(dst []byte, f float64, fmt byte, prec int, bitSize int)
```

函数的命名和作用跟上面讲解的其他类型一致。

关于 *FormatFloat* 的 *fmt* 参数，在第一章第三节格式化IO中有详细介绍。而 *prec* 表示有效数字（对 *fmt='b'* 无效），对于 'e', 'E' 和 'f'，有效数字用于小数点之后的位数；对于 'g' 和 'G'，则是所有的有效数字。例如：

```
strconv.FormatFloat(1223.13252, 'e', 3, 32)    // 结果：1.223e+03
strconv.FormatFloat(1223.13252, 'g', 3, 32)    // 结果：1.22e+03
```

由于浮点数有精度的问题，精度不一样，ParseFloat 和 FormatFloat 可能达不到互逆的效果。如：

```
s := strconv.FormatFloat(1234.5678, 'g', 6, 64)
strconv.ParseFloat(s, 64)
```

另外，fmt='b' 时，得到的字符串是无法通过 *ParseFloat* 还原的。

特别地（不区分大小写），+inf/inf，+infinity/infinity，-inf/-infinity 和 nan 通过 ParseFloat 转换分别返回对应的值（在 math 包中定义）。

同样的，基于性能的考虑，应该使用 *FormatFloat* 而不是 *fmt.Sprintf*。

2.3.5 其他导出的函数

如果要输出这样一句话：*This is "studygolang.com" website*. 该如何做？

So easy:

```
fmt.Println(`This is "studygolang.com" website`)
```

如果没有 `` 符号，该怎么做？转义：

```
fmt.Println("This is \"studygolang.com\" website")
```

除了这两种方法，*strconv* 包还提供了函数这做件事（Quote 函数）。我们称 "studygolang.com" 这种用双引号引起来的字符串为 Go 语言字面值字符串（Go string literal）。

上面的一句话可以这么做：

```
fmt.Println("This is", strconv.Quote("studygolang.com"), "website")
```

导航

- 上一节：bytes — byte slice 便利操作
- 下一节：regexp — 正则表达式

2.4 regexp — 正则表达式

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。正则表达式为文本处理提供了强大的功能。Go作为一门通用语言，自然提供了对正则表达式的支持。

`regexp` 包实现了正则表达式搜索。

正则表达式采用RE2语法（除了\c、\C），和Perl、Python等语言的正则基本一致。确切地说是兼容 `RE2` 语法。相关资料：http://code.google.com/p/re2/wiki/Syntax，包：regexp/syntax

注意：`regexp` 包的正则表达式实现保证运行时间随着输入大小线性增长的（即复杂度为O(n)，其中n为输入的长度），这一点，很多正则表达式的开源实现无法保证，参见：RSC的 《Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)》

另外，所有的字符都被视为utf-8编码的码值(Code Point)。

Regexp类型提供了多达16个方法，用于匹配正则表达式并获取匹配的结果。它们的名字满足如下正则表达式：

```
Find(All)?(String)?(Submatch)?(Index)?
```

2.4.1 strconv 包转换错误处理

介绍具体的转换之前，先看看 *strconv* 中的错误处理。

由于将字符串转为其他数据类型可能会出错，*strconv* 包定义了两个 *error* 类型的变量：*ErrRange* 和 *ErrSyntax*。其中，*ErrRange* 表示值超过了类型能表示的最大范围，比如将 "128" 转为 int8 就会返回这个错误；*ErrSyntax* 表示语法错误，比如将 "" 转为 int 类型会返回这个错误。

然而，在返回错误的时候，不是直接将上面的变量值返回，而是通过构造一个 *NumError* 类型的 *error* 对象返回。*NumError* 结构的定义如下：

```
// A NumError records a failed conversion.
type NumError struct {
    Func string // the failing function (ParseBool, ParseInt, ParseUint, ParseFloat)
    Num  string // the input
    Err  error  // the reason the conversion failed (ErrRange, ErrSyntax)
}
```

可见，该结构记录了转换过程中发生的错误信息。该结构不仅包含了一个 *error* 类型的成员，记录具体的错误信息，而且它自己也实现了 *error* 接口：

```
func (e *NumError) Error() string {
    return "strconv." + e.Func + ": " + "parsing " + Quote(e.Num) + ": " + e.Err.Error
()
}
```

包的实现中，定义了两个便捷函数，用于构造 *NumError* 对象：

```
func syntaxError(fn, str string) *NumError {
    return &NumError{fn, str, ErrSyntax}
}

func rangeError(fn, str string) *NumError {
    return &NumError{fn, str, ErrRange}
}
```

在遇到 *ErrSyntax* 或 *ErrRange* 错误时，通过上面的函数构造 *NumError* 对象。

https://github.com/StefanSchroeder/Golang-Regex-Tutorial

导航

- 上一节：strconv — 字符串和基本数据类型之间转换
- 下一节：unicode — Unicode码点、UTF-8/16编码

2.5 unicode — Unicode码点、UTF-8/16编码

世界中的字符有许许多多，有英文，中文，韩文等。我们强烈需要一个大大的映射表把世界上的字符映射成计算机可以阅读的二进制数字（字节）。 这样，每个字符都给予一个独一无二的编码，就不会出现写文字的人和阅读文字的人编码不同而出现无法读取的乱码现象了。

于是Unicode就出现了，它是一种所有符号的编码映射。最开始的时候，unicode认为使用两个字节，也就是16位就能包含所有的字符了。 但是非常可惜，两个字节最多只能覆盖65536个字符，这显然是不够的，于是附加了一套字符编码，即unicode4.0，附加的字符用4个字节表示。 现在为止，大概Unicode可以覆盖100多万个字符了。

Unicode只是定义了一个字符和一个编码的映射，但是呢，对应的存储却没有制定。 比如一个编码0x0041代表大写字母A，那么可能有一种存储至少有4个字节，那可能0x00000041来存储代表A。 这个就是unicode的具体实现。unicode的具体实现有很多种，UTF-8和UTF-16就是其中两种。

UTF-8表示最少用一个字节就能表示一个字符的编码实现。它采取的方式是对不同的语言使用不同的方法，将unicode编码按照这个方法进行转换。 我们只要记住最后的结果是英文占一个字节，中文占三个字节。这种编码实现方式也是现在应用最为广泛的方式了。

UTF-16表示最少用两个字节能表示一个字符的编码实现。同样是对unicode编码进行转换，它的结果是英文占用两个字节，中文占用两个或者四个字节。 实际上，UTF-16就是最严格实现了unicode4.0。但由于英文是最通用的语言，所以推广程度没有UTF-8那么普及。

回到go对unicode包的支持，由于UTF-8的作者Ken Thompson同时也是go语言的创始人，所以说，在字符支持方面，几乎没有语言的理解会高于go了。 go对unicode的支持包含三个包:

- unicode
- unicode/utf8
- unicode/utf16

unicode包包含基本的字符判断函数。utf8包主要负责rune和byte之间的转换。utf16包负责rune和uint16数组之间的转换。

由于字符的概念有的时候比较模糊，比如字符（小写a）普通显示为a，在重音字符中（grave-accented）中显示为à。 这时候字符（character）的概念就有点不准确了，因为a和à显然是两个不同的unicode编码，但是却代表同一个字符，所以引入了rune。 一个rune就代表一个unicode编码，所以上面的a和à是两个不同的rune。

这里有个要注意的事情，go语言的所有代码都是UTF8的，所以如果我们在程序中的字符串都是utf8编码的，但是我们的单个字符（单引号扩起来的）却是unicode的。

2.5.1 unicode包

unicode包含了对rune的判断。这个包把所有unicode涉及到的编码进行了分类，使用结构

```
type RangeTable struct {
    R16         []Range16
    R32         []Range32
    LatinOffset int
}
```

来表示这个功能的字符集。这些字符集都集中列表在table.go这个源码里面。

比如控制字符集：

```
var _Pc = &RangeTable{
    R16: []Range16{
        {0x005f, 0x203f, 8160},
        {0x2040, 0x2054, 20},
        {0xfe33, 0xfe34, 1},
        {0xfe4d, 0xfe4f, 1},
        {0xff3f, 0xff3f, 1},
    },
}
```

回到包的函数，我们看到有下面这些判断函数：

```
func IsControl(r rune) bool  // 是否控制字符
func IsDigit(r rune) bool  // 是否阿拉伯数字字符，即1-9
func IsGraphic(r rune) bool // 是否图形字符
func IsLetter(r rune) bool // 是否字母
func IsLower(r rune) bool // 是否小写字符
func IsMark(r rune) bool // 是否符号字符
func IsNumber(r rune) bool // 是否数字字符，比如罗马数字Ⅷ也是数字字符
func IsOneOf(ranges []*RangeTable, r rune) bool // 是否是RangeTable中的一个
func IsPrint(r rune) bool // 是否可打印字符
func IsPunct(r rune) bool // 是否标点符号
func IsSpace(r rune) bool // 是否空格
func IsSymbol(r rune) bool // 是否符号字符
func IsTitle(r rune) bool // 是否title case
func IsUpper(r rune) bool // 是否大写字符
```

看下面这个例子：

```
func main() {
    single := '\u0015'
    fmt.Println(unicode.IsControl(single))  //true
    single = '\ufe35'
    fmt.Println(unicode.IsControl(single)) // false

    digit := rune('1')
    fmt.Println(unicode.IsDigit(digit)) //true
    fmt.Println(unicode.IsNumber(digit)) //true
    letter := rune('Ⅷ')
    fmt.Println(unicode.IsDigit(letter)) //false
    fmt.Println(unicode.IsNumber(letter)) //true
}
```

2.5.2 utf8包

utf8里面的函数就有一些字节和字符的转换。

判断是否符合utf8编码的函数：

- func Valid(p []byte) bool
- func ValidRune(r rune) bool
- func ValidString(s string) bool

判断rune的长度的函数：

- func RuneLen(r rune) int

判断字节串或者字符串的rune数

- func RuneCount(p []byte) int
- func RuneCountInString(s string) (n int)

编码和解码rune到byte

- func DecodeRune(p []byte) (r rune, size int)
- func EncodeRune(p []byte, r rune) int

2.5.3 utf16包

utf16的包的函数就比较少了。

将int16和rune进行转换

- func Decode(s []uint16) []rune
- func Encode(s []rune) []uint16

unicode有个基本字符平面和增补平面的概念，基本字符平面只有65535个字符，增补平面（有16个）加上去就能表示1114112个字符。 utf16就是严格实现了unicode的这种编码规范。

而基本字符和增补平面字符就是一个代理对（Surrogate Pair）。一个代理对可以和一个rune进行转换。

- func DecodeRune(r1, r2 rune) rune
- func EncodeRune(r rune) (r1, r2 rune)

导航

- 上一节：strings — 字符串操作
- 下一节：strconv — 字符串和基本数据类型之间转换

第三章 数据结构与算法

程序设计离不开数据结构和算法。数据结构是数据组织和存储的逻辑形式，以达到方便访问和修改数据的目的。而算法是根据实际输入输出的需求设计的一系列计算过程，被认为是程序的灵魂。设计良好的算法的重要意义正如*Thomas*在《算法导论》中提到“计算机可以做得很快，但不是无限快；存储器可以做到很便宜，但不是免费的。因此，计算时间是一种有限的资源，存储空间也是一种有限的资源。这些有限的资源必须有效地使用，那些时间上和空间上有效的算法可以帮助做到这一点。“

本章内容涵盖了Go标准库中的3个包：

sort 包包含基本的排序方法，支持切片数据排序以及用户自定义数据集合排序

index/suffixary 包实现了后缀数组相关算法以支持许多常见的字符串操作

container 包提供了对heap、list和ring这3种数据结构的底层支持。任何实现了相应接口的数据结构都可以调用该结构的方法。

导航

- 第二章
- 下一节：sort - 排序算法

3.1 sort —— 排序算法

该包实现了四种基本排序算法：插入排序、归并排序、堆排序和快速排序。 但是这四种排序方法是不公开的，它们只被用于sort包内部使用。所以在对数据集合排序时不必考虑应当选择哪一种排序方法，只要实现了sort.Interface定义的三个方法：获取数据集合长度的Len()方法、比较两个元素大小的Less()方法和交换两个元素位置的Swap()方法，就可以顺利对数据集合进行排序。sort包会根据实际数据自动选择高效的排序算法。 除此之外，为了方便对常用数据类型的操作，sort包提供了对[]int切片、[]float64切片和[]string切片完整支持，主要包括：

- 对基本数据类型切片的排序支持
- 基本数据元素查找
- 判断基本数据类型切片是否已经排好序
- 对排好序的数据集合逆序

3.1.1 数据集合排序

前面已经提到过，对数据集合（包括自定义数据类型的集合）排序需要实现sort.Interface接口的三个方法，我们看以下该接口的定义：

```
type Interface interface {
        // 获取数据集合元素个数
        Len() int
        // 如果i索引的数据小于j所以的数据，返回true，不会调用
        // 下面的Swap()，即数据升序排序。
        Less(i, j int) bool
        // 交换i和j索引的两个元素的位置
        Swap(i, j int)
}
```

数据集合实现了这三个方法后，即可调用该包的Sort()方法进行排序。 Sort()方法定义如下：

```
    func Sort(data Interface)
```

Sort()方法惟一的参数就是待排序的数据集合。

该包还提供了一个方法可以判断数据集合是否已经排好顺序，该方法的内部实现依赖于我们自己实现的Len()和Less()方法：

```
func IsSorted(data Interface) bool {
    n := data.Len()
    for i := n - 1; i > 0; i-- {
        if data.Less(i, i-1) {
            return false
        }
    }
    return true
}
```

下面是一个使用sort包对学生成绩排序的示例：

```
package main

import (
    "fmt"
    "sort"
)

//学生成绩结构体
type StuScore struct {
     //姓名
    name  string
    //成绩
    score int
}

type StuScores []StuScore

//Len()
func (s StuScores) Len() int {
    return len(s)
}

//Less():成绩将有低到高排序
func (s StuScores) Less(i, j int) bool {
    return s[i].score < s[j].score
}

//Swap()
func (s StuScores) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

func main() {
    stus := StuScores{
                {"alan", 95},
                {"hikerell", 91},
                {"acmfly", 96},
                {"leao", 90}}
```

```
    fmt.Println("Default:")
    //原始顺序
    for _, v := range stus {
        fmt.Println(v.name, ":", v.score)
    }
    fmt.Println()
    //StuScores已经实现了sort.Interface接口
    sort.Sort(stus)

    fmt.Println("Sorted:")
     //排好序后的结构
    for _, v := range stus {
        fmt.Println(v.name, ":", v.score)
    }

    //判断是否已经排好顺序，将会打印true
    fmt.Println("IS Sorted?", sort.IsSorted(stus))
}
```

程序该示例程序的自定义类型StuScores实现了sort.Interface接口，所以可以将其对象作为sort.Sort()和sort.IsSorted()的参数传入。运行结果：

```
======Default======
alan : 95
hikerell : 91
acmfly : 96
leao : 90

======Sorted=======
leao : 90
hikerell : 91
alan : 95
acmfly : 96
IS Sorted? true
```

该示例实现的是升序排序，如果要得到降序排序结果，其实只要修改Less()函数：

```
//Less():成绩降序排序,只将小于号修改为大于号
func (s StuScores) Less(i, j int) bool {
    return s[i].score > s[j].score
}
```

此外，*sort*包提供了Reverse()方法，可以允许将数据按Less()定义的排序方式逆序排序，而不必修改Less()代码。方法定义如下：

```
func Reverse(data Interface) Interface
```

我们可以看到Reverse()返回的一个sort.Interface接口类型，整个Reverse()的内部实现比较有趣：

```
    //定义了一个reverse结构类型，嵌入Interface接口
    type reverse struct {
        Interface
    }

    //reverse结构类型的Less()方法拥有嵌入的Less()方法相反的行为
    //Len()和Swap()方法则会保持嵌入类型的方法行为
    func (r reverse) Less(i, j int) bool {
        return r.Interface.Less(j, i)
    }

    //返回新的实现Interface接口的数据类型
    func Reverse(data Interface) Interface {
        return &reverse{data}
    }
```

了解内部原理后，可以在学生成绩排序示例中使用Reverse()来实现成绩升序排序：

```
    sort.Sort(sort.Reverse(stus))
     for _, v := range stus {
        fmt.Println(v.name, ":", v.score)
    }
```

最后一个方法：Search()

```
func Search(n int, f func(int) bool) int
```

官方文档这样描述该方法：

> Search()方法回使用“二分查找”算法来搜索某指定切片[0:n]，并返回能够使f(i)=true的最小的i（0<=i<n）值，并且会假定，如果f(i)=true，则f(i+1)=true，即对于切片[0:n]，i之前的切片元素会使f()函数返回false，i及i之后的元素会使f()函数返回true。但是，当 在切片中无法找到时f(i)=true的i时（此时切片元素都不能使f()函数返回true），Search() 方法会返回n。

Search()函数一个常用的使用方式是搜索元素x是否在已经升序排好的切片s中：

```
    x := 11
    s := []int{3, 6, 8, 11, 45} //注意已经升序排序
    pos := sort.Search(len(s), func(i int) bool { return s[i] >= x })
    if pos < len(s) && s[pos] == x {
        fmt.Println(x, "在s中的位置为：", pos)
    } else {
        fmt.Println("s不包含元素", x)
    }
```

官方文档还给出了一个猜数字的小程序：

```
func GuessingGame() {
    var s string
    fmt.Printf("Pick an integer from 0 to 100.\n")
    answer := sort.Search(100, func(i int) bool {
        fmt.Printf("Is your number <= %d? ", i)
        fmt.Scanf("%s", &s)
        return s != "" && s[0] == 'y'
    })
    fmt.Printf("Your number is %d.\n", answer)
}
```

3.1.2 *sort*包已经支持的内部数据类型排序

前面已经提到，*sort*包原生支持[]int、[]float64和[]string三种内建数据类型切片的排序操作，即不必我们自己实现相关的Len()、Less()和Swap()方法。

1. IntSlice类型及[]int排序

由于[]int切片排序内部实现及使用方法与[]float64和[]string类似，所以只详细描述该部分。

*sort*包定义了一个IntSlice类型，并且实现了sort.Interface接口：

```
    type IntSlice []int
    func (p IntSlice) Len() int           { return len(p) }
    func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
    func (p IntSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
    //IntSlice类型定义了Sort()方法，包装了sort.Sort()函数
    func (p IntSlice) Sort() { Sort(p) }
    //IntSlice类型定义了SearchInts()方法，包装了SearchInts()函数
    func (p IntSlice) Search(x int) int { return SearchInts(p, x) }
```

并且提供的sort.Ints()方法使用了该IntSlice类型：

```
    func Ints(a []int) { Sort(IntSlice(a)) }
```

所以，对[]int切片排序是更常使用sort.Ints()，而不是直接使用IntSlice类型：

```
    s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
    sort.Ints(s)
    fmt.Println(s) //将会输出[1 2 3 4 5 6]
```

如果要使用降序排序，显然要用前面提到的Reverse()方法：

```
    s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
    sort.Sort(sort.Reverse(sort.IntSlice(s)))
    fmt.Println(s) //将会输出[6 5 4 3 2 1]
```

如果要查找整数x在切片a中的位置，相对于前面提到的Search()方法，*sort*包提供了SearchInts():

```
    func SearchInts(a []int, x int) int
```

注意，SearchInts()的使用条件为：切片**a**已经升序排序

```
    s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
    sort.Ints(s) //排序后的s为[1 2 3 4 5 6]
    fmt.Println(sort.SearchInts(s, 3)) //将会输出2
```

2. Float64Slice类型及[]float64排序

实现与Ints类似，只看一下其内部实现：

```
    type Float64Slice []float64

    func (p Float64Slice) Len() int           { return len(p) }
    func (p Float64Slice) Less(i, j int) bool { return p[i] < p[j] || isNaN(p[i]) && !
isNaN(p[j]) }
    func (p Float64Slice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
    func (p Float64Slice) Sort() { Sort(p) }
    func (p Float64Slice) Search(x float64) int { return SearchFloat64s(p, x) }
```

与Sort()、IsSorted()、Search()相对应的三个方法：

```
func Float64s(a []float64)
func Float64sAreSorted(a []float64) bool
func SearchFloat64s(a []float64, x float64) int
```

要说明一下的是，在上面Float64Slice类型定义的Less方法中，有一个内部函数isNaN()。isNaN()与*math*包中IsNaN()实现完全相同，*sort*包之所以不使用math.IsNaN()，完全是基于包依赖性的考虑，应当看到，*sort*包的实现不依赖与其他任何包。

3. StringSlice类型及[]string排序

两个string对象之间的大小比较是基于“字典序”的。

实现与Ints类似，只看一下其内部实现：

```
type StringSlice []string

func (p StringSlice) Len() int           { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
func (p StringSlice) Sort() { Sort(p) }
func (p StringSlice) Search(x string) int { return SearchStrings(p, x) }
```

与Sort()、IsSorted()、Search()相对应的三个方法：

```
func Strings(a []string)
func StringsAreSorted(a []string) bool
func SearchStrings(a []string, x string) int
```

导航

- 第三章 数据结构与算法
- 下一节：index/suffixarray — 后缀数组实现子字符串查询

3.3 container — 容器数据类型：heap、list 和ring

该包实现了三个复杂的数据结构：堆，链表，环。 这个包就意味着你使用这三个数据结构的时候不需要再费心从头开始写算法了。

3.3.1 堆

这里的堆使用的数据结构是最小二叉树，即根节点比左边子树和右边子树的所有值都小。 go 的堆包只是实现了一个接口，我们看下它的定义：

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // add x as element Len()
    Pop() interface{}   // remove and return element Len() - 1.
}
```

可以看出，这个堆结构继承自sort.Interface, 回顾下sort.Interface，它需要实现三个方法

- Len() int
- Less(i, j int) bool
- Swap(i, j int)

加上堆接口定义的两个方法

- Push(x interface{})
- Pop() interface{}

就是说你定义了一个堆，就要实现五个方法，直接拿package doc中的example做例子：

```
type IntHeap []int

func (h IntHeap) Len() int           { return len(h) }
func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }
func (h IntHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *IntHeap) Push(x interface{}) {
    *h = append(*h, x.(int))
}

func (h *IntHeap) Pop() interface{} {
    old := *h
    n := len(old)
    x := old[n-1]
    *h = old[0 : n-1]
    return x
}
```

那么IntHeap就实现了这个堆结构，我们就可以使用堆的方法来对它进行操作：

```
h := &IntHeap{2, 1, 5}
heap.Init(h)
heap.Push(h, 3)
heap.Pop(h)
```

具体说下内部实现，是使用最小堆，索引排序从根节点开始，然后左子树，右子树的顺序方式。 内部实现了down和up分别表示对堆中的某个元素向上保证最小堆和向上保证最小堆。

当往堆中插入一个元素的时候，这个元素插入到最右子树的最后一个节点中，然后调用up向上保证最小堆。

当要从堆中推出一个元素的时候，先吧这个元素和右子树最后一个节点兑换，然后弹出最后一个节点，然后对root调用down，向下保证最小堆。

3.3.2 链表

链表就是一个有prev和next指针的数组了。它维护两个type，(注意，这里不是interface)

```
type Element struct {
    next, prev *Element  // 上一个元素和下一个元素
    list *List  // 元素所在链表
    Value interface{}  // 元素
}

type List struct {
    root Element  // 链表的根元素
    len  int      // 链表的长度
}
```

基本使用是先创建list，然后往list中插入值，list就内部创建一个Element，并内部设置好Element的next,prev等。具体可以看下例子：

```
// This example demonstrates an integer heap built using the heap interface.
package main

import (
    "container/list"
    "fmt"
)

func main() {
    list := list.New()
    list.PushBack(1)
    list.PushBack(2)

    fmt.Printf("len: %v\n", list.Len());
    fmt.Printf("first: %#v\n", list.Front());
    fmt.Printf("second: %#v\n", list.Front().Next());
}

output:
len: 2
first: &list.Element{next:(*list.Element)(0x2081be1b0), prev:(*list.Element)(0x2081be1
50), list:(*list.List)(0x2081be150), Value:1}
second: &list.Element{next:(*list.Element)(0x2081be150), prev:(*list.Element)(0x2081be
180), list:(*list.List)(0x2081be150), Value:2}
```

list对应的方法有：

```
type Element
    func (e *Element) Next() *Element
    func (e *Element) Prev() *Element
type List
    func New() *List
    func (l *List) Back() *Element   // 最后一个元素
    func (l *List) Front() *Element  // 第一个元素
    func (l *List) Init() *List  // 链表初始化
    func (l *List) InsertAfter(v interface{}, mark *Element) *Element // 在某个元素前插入
    func (l *List) InsertBefore(v interface{}, mark *Element) *Element  // 在某个元素后
插入
    func (l *List) Len() int // 在链表长度
    func (l *List) MoveAfter(e, mark *Element)  // 把e元素移动到mark之后
    func (l *List) MoveBefore(e, mark *Element)  // 把e元素移动到mark之前
    func (l *List) MoveToBack(e *Element) // 把e元素移动到队列最后
    func (l *List) MoveToFront(e *Element) // 把e元素移动到队列最头部
    func (l *List) PushBack(v interface{}) *Element  // 在队列最后插入元素
    func (l *List) PushBackList(other *List)  // 在队列最后插入接上新队列
    func (l *List) PushFront(v interface{}) *Element  // 在队列头部插入元素
    func (l *List) PushFrontList(other *List) // 在队列头部插入接上新队列
    func (l *List) Remove(e *Element) interface{} // 删除某个元素
```

3.3.3 环

环的结构有点特殊，环的尾部就是头部，所以每个元素实际上就可以代表自身的这个环。 它不需要像list一样保持list和element两个结构，只需要保持一个结构就行。

```
type Ring struct {
    next, prev *Ring
    Value      interface{}
}
```

我们初始化环的时候，需要定义好环的大小，然后对环的每个元素进行赋值。环还提供一个Do方法，能便利一遍环，对每个元素执行一个function。 看下面的例子：

```
// This example demonstrates an integer heap built using the heap interface.
package main

import (
    "container/ring"
    "fmt"
)

func main() {
    ring := ring.New(3)

    for i := 1; i <= 3; i++ {
        ring.Value = i
        ring = ring.Next()
    }

    // 计算1+2+3
    s := 0
    ring.Do(func(p interface{}){
        s += p.(int)
    })
    fmt.Println("sum is", s)
}

output:
sum is 6
```

ring提供的方法有

```
type Ring
    func New(n int) *Ring  // 初始化环
    func (r *Ring) Do(f func(interface{}))  // 循环环进行操作
    func (r *Ring) Len() int // 环长度
    func (r *Ring) Link(s *Ring) *Ring // 连接两个环
    func (r *Ring) Move(n int) *Ring // 指针从当前元素开始向后移动或者向前（n可以为负数）
    func (r *Ring) Next() *Ring // 当前元素的下个元素
    func (r *Ring) Prev() *Ring // 当前元素的上个元素
    func (r *Ring) Unlink(n int) *Ring // 从当前元素开始，删除n个元素
```

导航

- 第三章 数据结构与算法
- 上一节：index/suffixarray — 后缀数组实现子字符串查询
- 下一节：container总结

第四章 日期与时间

实际开发中，经常会遇到日期和时间相关的操作，比如：格式化日期和时间，解析一个日期时间字符串等。Go语言通过标准库 `time` 包处理日期和时间相关的问题。

本章只有 `time` 这一个包，为了便于阅读，将拆为如下小结进行讲解：

- 主要类型概述
- 时区
- Time类型详解
- 定时器

导航

- 第三章
- 下一节：主要类型概述

4.1 主要类型概述

time 包提供了时间的显示和计量用的功能。日历的计算采用的是公历。提供的主要类型如下：

Location

代表一个地区，并表示该地区所在的时区（可能多个）。`Location` 通常代表地理位置的偏移，比如 CEST 和 CET 表示中欧。下一节将详细讲解 Location。

Time

代表一个纳秒精度的时间点，是公历时间。后面会详细介绍。

Duration

代表两个时间点之间经过的时间，以纳秒为单位。可表示的最长时间段大约290年，也就是说如果两个时间点相差超过 290 年，会返回 290 年，也就是 minDuration(-1 << 63) 或 maxDuration(1 << 63 - 1)。

类型定义：`type Duration int64`。

将 `Duration` 类型直接输出时，因为实现了 `fmt.Stringer` 接口，会输出人类友好的可读形式，如：72h3m0.5s。

Timer 和 Ticker

这是定时器相关类型。本章最后会讨论定时器。

Weekday 和 Month

这两个类型的原始类型都是 int，定义它们，语义更明确，同时，实现 `fmt.Stringer` 接口，方便输出。

导航

- 第四章 日期与时间
- 下一节：时区

4.2 时区

不同国家（有时甚至是同一个国家内的不同地区）使用不同的时区。对于要输入和输出时间的程序来说，必须对系统所处的时区加以考虑。Go 语言使用 `Location` 来表示地区相关的时区，一个 Location 可能表示多个时区。

`time` 包提供了 Location 的两个实例：`Local` 和 `UTC`。`Local` 代表当前系统本地时区；`UTC` 代表通用协调时间，也就是零时区。`time` 包默认（为显示提供时区）使用 `UTC` 时区。

Local 是如何做到表示本地时区的？

时区信息既浩繁又多变，Unix 系统以标准格式存于文件中，这些文件位于 /usr/share/zoneinfo，而本地时区可以通过 /etc/localtime 获取，这是一个符号链接，指向 /usr/share/zoneinfo 中某一个时区。比如我本地电脑指向的是：/usr/share/zoneinfo/Asia/Shanghai。

因此，在初始化 Local 时，通过读取 /etc/localtime 可以获取到系统本地时区。

当然，如果设置了环境变量 `TZ`，则会优先使用它。

相关代码：

```
tz, ok := syscall.Getenv("TZ")
switch {
case !ok:
    z, err := loadZoneFile("", "/etc/localtime")
    if err == nil {
        localLoc = *z
        localLoc.name = "Local"
        return
    }
case tz != "" && tz != "UTC":
    if z, err := loadLocation(tz); err == nil {
        localLoc = *z
        return
    }
}
```

获得特定时区的实例

函数 `LoadLocation` 可以根据名称获取特定时区的实例。函数声明如下：

```
func LoadLocation(name string) (*Location, error)
```

如果 name 是""或"UTC"，返回UTC；如果 name 是"Local"，返回Local；否则 name 应该是IANA时区数据库里有记录的地点名（该数据库记录了地点和对应的时区），如"America/New_York"。

LoadLocation 函数需要的时区数据库可能不是所有系统都提供，特别是非Unix系统。此时 `LoadLocation` 会查找环境变量 ZONEINFO 指定目录或解压该变量指定的zip文件（如果有该环境变量）；然后查找Unix系统的惯例时区数据安装位置，最后查找 `$GOROOT/lib/time/zoneinfo.zip` 。

可以在 Unix 系统下的 /usr/share/zoneinfo 中找到所有的名称。

总结

通常，我们使用 `time.Local` 即可，偶尔可能会需要使用 `UTC` 。在解析时间时，心中一定记得有时区这么回事。当你发现时间出现莫名的情况时，很可能是因为时区的问题，特别是当时间相差 8 小时时。

导航

- 上一节：主要类型概述
- 下一节：Time类型详解

4.3 Time 类型详解

`Time` 代表一个纳秒精度的时间点。

程序中应使用 Time 类型值来保存和传递时间，而不是指针。就是说，表示时间的变量和字段，应为time.Time类型，而不是*time.Time.类型。一个Time类型值可以被多个go程同时使用。时间点可以使用Before、After和Equal方法进行比较。Sub方法让两个时间点相减，生成一个Duration类型值（代表时间段）。Add方法给一个时间点加上一个时间段，生成一个新的Time类型时间点。

Time 零值代表时间点 January 1, year 1, 00:00:00.000000000 UTC。因为本时间点一般不会出现在使用中，IsZero 方法提供了检验时间是否是显式初始化的一个简单途径。

每一个时间都具有一个地点信息（及对应地点的时区信息），当计算时间的表示格式时，如Format、Hour和Year等方法，都会考虑该信息。Local、UTC和In方法返回一个指定时区（但指向同一时间点）的Time。修改地点/时区信息只是会改变其表示；不会修改被表示的时间点，因此也不会影响其计算。

通过 == 比较 Time 时，Location 信息也会参与比较，因此 Time 不应该作为 map 的 key。

Time 的内部结构

```
type Time struct {
    // sec gives the number of seconds elapsed since
    // January 1, year 1 00:00:00 UTC.
    sec int64

    // nsec specifies a non-negative nanosecond
    // offset within the second named by Seconds.
    // It must be in the range [0, 999999999].
    nsec int32

    // loc specifies the Location that should be used to
    // determine the minute, hour, month, day, and year
    // that correspond to this Time.
    // Only the zero Time has a nil Location.
    // In that case it is interpreted to mean UTC.
    loc *Location
}
```

要讲解 `time.Time` 的内部结构，得先看 `time.Now()` 函数。

```
// Now returns the current local time.
func Now() Time {
    sec, nsec := now()
    return Time{sec + unixToInternal, nsec, Local}
}
```

now() 的具体实现在 `runtime` 包中，以 linux/amd64 为例，在 sys_linux_amd64.s 中的 `time·now`，这是汇编实现的：

- 调用系统调用 `clock_gettime` 获取时钟值（这是 POSIX 时钟）。其中 clockid_t 时钟类型是 CLOCK_REALTIME，也就是可设定的系统级实时时钟。得到的是 struct timespec 类型。（可以到纳秒）
- 如果 `clock_gettime` 不存在，则使用精度差些的系统调用 `gettimeofday`。得到的是 struct timeval 类型。（最多到微妙）

注意：这里使用了 Linux 的 vdso 特性，不了解的，可以查阅相关知识。

虽然 `timespec` 和 `timeval` 不一样，但结构类似。因为 `now()` 函数返回两个值：sec(秒)和 nsec(纳秒)，所以，`time·now` 的实现将这两个结构转为需要的返回值。需要注意的是，Linux 系统调用返回的 sec(秒) 是 Unix 时间戳，也就是从 1970-1-1 算起的。

回到 `time.Now()` 的实现，现在我们得到了 sec 和 nsec，从 `Time{sec + unixToInternal, nsec, Local}` 这句可以看出，Time 结构的 sec 并非 Unix 时间戳，实际上，加上的 `unixToInternal` 是 1-1-1 到 1970-1-1 经历的秒数。也就是 `Time` 中的 sec 是从 1-1-1 算起的秒数，而不是 Unix 时间戳。

`Time` 的最后一个字段表示地点时区信息。本章后面会专门介绍。

常用函数或方法

`Time` 相关的函数和方法较多，有些很容易理解，不赘述，查文档即可。

零值的判断

因为 `Time` 的零值是 sec 和 nsec 都是0，表示 1年1月1日。

Time.IsZero() 函数用于判断 Time 表示的时间是否是 0 值。

与 **Unix** 时间戳的转换

相关函数或方法：

- time.Unix(sec, nsec int64) 通过 Unix 时间戳生成 `time.Time` 实例；

- time.Time.Unix() 得到 Unix 时间戳；
- time.Time.UnixNano() 得到 Unix 时间戳的纳秒表示；

格式化和解析

这是实际开发中常用到的。

- time.Parse 和 time.ParseInLocation
- time.Time.Format

解析

对于解析，要特别注意时区问题，否则很容易出 bug。比如：

```
t, _ := time.Parse("2006-01-02 15:04:05", "2016-06-13 09:14:00")
fmt.Println(time.Now().Sub(t).Hours())
```

`2016-06-13 09:14:00` 这个时间可能是参数传递过来的。这段代码的结果跟预期的不一样。

原因是 `time.Now()` 的时区是 `time.Local`，而 `time.Parse` 解析出来的时区却是 time.UTC（可以通过 `Time.Location()` 函数知道是哪个时区）。在中国，它们相差 8 小时。

所以，一般的，我们应该总是使用 `time.ParseInLocation` 来解析时间，并给第三个参数传递 `time.Local`。

为什么是 2006-01-02 15:04:05

可能你已经注意到：`2006-01-02 15:04:05` 这个字符串了。没错，这是固定写法，类似于其他语言中 `Y-m-d H:i:s` 等。为什么采用这种形式？又为什么是这个时间点而不是其他时间点？

- 官方说，使用具体的时间，比使用 `Y-m-d H:i:s` 更容易理解和记忆；这么一说还真是~
- 而选择这个时间点，也是出于好记的考虑，官方的例子：`Mon Jan 2 15:04:05 MST 2006`，另一种形式 `01/02 03:04:05PM '06 -0700`，对应是 1、2、3、4、5、6、7；常见的格式：`2006-01-02 15:04:05`，很好记：2006年1月2日3点4分5秒~

如果你是 *PHPer*，喜欢 *PHP* 的格式，可以试试 *times* 这个包。

格式化

时间格式化输出，使用 `Format` 方法，`layout` 参数和 `Parse` 的一样。`Time.String()` 方法使用了 `2006-01-02 15:04:05.999999999 -0700 MST` 这种 `layout`。

实现 序列化/反序列化 相关接口

`Time` 实现了 `encoding` 包中的 `BinaryMarshaler`、`BinaryUnmarshaler`、`TextMarshaler` 和 `TextUnmarshaler` 接口；`encoding/json` 包中的 `Marshaler` 和 `Unmarshaler` 接口。

它还实现了 `gob` 包中的 `GobEncoder` 和 `GobDecoder` 接口。

对于文本序列化/反序列化，通过 `Parse` 和 `Format` 实现；对于二进制序列化，需要单独实现。

对于 json，使用的是 `time.RFC3339Nano` 这种格式。通常程序中不使用这种格式。解决办法是定义自己的类型。如：

```
type OftenTime time.Time

func (self OftenTime) MarshalJSON() ([]byte, error) {
    t := time.Time(self)
    if y := t.Year(); y < 0 || y >= 10000 {
        return nil, errors.New("Time.MarshalJSON: year outside of range [0,9999]")
    }
    // 注意 `"2006-01-02 15:04:05"`。因为是 JSON，双引号不能少
    return []byte(t.Format(`"2006-01-02 15:04:05"`)), nil
}
```

Round 和 Truncate 方法

比如，有这么个需求：获取当前时间整点的 `Time` 实例。例如，当前时间是 15:54:23，需要的是 15:00:00。我们可以这么做：

```
t, _ := time.ParseInLocation("2006-01-02 15:04:05", time.Now().Format("2006-01-02 15:0
0:00"), time.Local)
fmt.Println(t)
```

实际上，`time` 包给我们提供了专门的方法，功能更强大，性能也更好，这就是 `Round` 和 `Trunate`，它们区别，一个是取最接近的，一个是向下取整。

使用示例：

```
t, _ := time.ParseInLocation("2006-01-02 15:04:05", "2016-06-13 15:34:39", time.Local)
// 整点（向下取整）
fmt.Println(t.Truncate(1 * time.Hour))
// 整点（最接近）
fmt.Println(t.Round(1 * time.Hour))

// 整分（向下取整）
fmt.Println(t.Truncate(1 * time.Minute))
// 整分（最接近）
fmt.Println(t.Round(1 * time.Minute))

t2, _ := time.ParseInLocation("2006-01-02 15:04:05", t.Format("2006-01-02 15:00:00"),
time.Local)
fmt.Println(t2)
```

导航

- 上一节：时区
- 下一节：定时器

4.4 定时器

定时器是进程规划自己在未来某一时刻接获通知的一种机制。本节介绍两种定时器：`Timer`（到达指定时间触发且只触发一次）和 `Ticker`（间隔特定时间触发）。

Timer

内部实现源码分析

`Timer` 类型代表单次时间事件。当 `Timer` 到期时，当时的时间会被发送给 C (channel)，除非 `Timer` 是被 `AfterFunc` 函数创建的。

注意：`Timer` 的实例必须通过 `NewTimer` 或 `AfterFunc` 获得。

类型定义如下：

```
type Timer struct {
    C <-chan Time     // The channel on which the time is delivered.
    r runtimeTimer
}
```

C 已经解释了，我们看看 `runtimeTimer`。它定义在 sleep.go 文件中，必须和 `runtime` 包中 `time.go` 文件中的 `timer` 必须保持一致：

```
type timer struct {
    i int // heap index

    // Timer wakes up at when, and then at when+period, ... (period > 0 only)
    // each time calling f(now, arg) in the timer goroutine, so f must be
    // a well-behaved function and not block.
    when   int64
    period int64
    f      func(interface{}, uintptr)
    arg    interface{}
    seq    uintptr
}
```

我们通过 `NewTimer()` 来看这些字段都怎么赋值，是什么用途。

```
// NewTimer creates a new Timer that will send
// the current time on its channel after at least duration d.
func NewTimer(d Duration) *Timer {
    c := make(chan Time, 1)
    t := &Timer{
        C: c,
        r: runtimeTimer{
            when: when(d),
            f:    sendTime,
            arg:  c,
        },
    }
    startTimer(&t.r)
    return t
}
```

在 `when` 表示的时间到时，会往 Timer.C 中发送当前时间。`when` 表示的时间是纳秒时间，正常通过 `runtimeNano() + int64(d)` 赋值。跟上一节中讲到的 `now()` 类似，`runtimeNano()` 也在 `runtime` 中实现（`runtime·nanotime`）：

- 调用系统调用 `clock_gettime` 获取时钟值（这是 POSIX 时钟）。其中 clockid_t 时钟类型是 CLOCK_MONOTONIC，也就是不可设定的恒定态时钟。具体的是什么时间，SUSv3 规定始于未予规范的过去某一点，Linux 上，始于系统启动。
- 如果 `clock_gettime` 不存在，则使用精度差些的系统调用 `gettimeofday`。

`f` 参数的值是 `sendTime`，定时器时间到时，会调用 f，并将 `arg` 和 `seq` 传给 `f`。

因为 `Timer` 是一次性的，所以 `period` 保留默认值 0。

定时器的具体实现逻辑，都在 `runtime` 中的 `time.go` 中，它的实现，没有采用经典 Unix 间隔定时器 `setitimer` 系统调用，也没有 采用 POSIX 间隔式定时器（相关系统调用：`timer_create`、`timer_settime` 和 `timer_delete`），而是通过四叉树堆(heep)实现的（`runtimeTimer` 结构中的 `i` 字段，表示在堆中的索引）。通过构建一个最小堆，保证最快拿到到期了的定时器执行。定时器的执行，在专门的 `goroutine` 中进行的：`go timerproc()`。有兴趣的同学，可以阅读 `runtime/time.go` 的源码。

Timer 相关函数或方法的使用

通过 **`time.After`** 模拟超时：

```
c := make(chan int)

go func() {
    // time.Sleep(1 * time.Second)
    time.Sleep(3 * time.Second)
    <-c
}()

select {
case c <- 1:
    fmt.Println("channel...")
case <-time.After(2 * time.Second):
    close(c)
    fmt.Println("timeout...")
}
```

`time.Stop` 停止定时器 或 **`time.Reset`** 重置定时器

```
start := time.Now()
timer := time.AfterFunc(2*time.Second, func() {
    fmt.Println("after func callback, elaspe:", time.Now().Sub(start))
})

time.Sleep(1 * time.Second)
// time.Sleep(3*time.Second)
// Reset 在 Timer 还未触发时返回 true；触发了或Stop了，返回false
if timer.Reset(3 * time.Second) {
    fmt.Println("timer has not trigger!")
} else {
    fmt.Println("timer had expired or stop!")
}

time.Sleep(10 * time.Second)

// output:
// timer has not trigger!
// after func callback, elaspe: 4.00026461s
```

如果定时器还未触发，`Stop` 会将其移除，并返回 true；否则返回 false；后续再对该 `Timer` 调用 `Stop`，直接返回 false。

`Reset` 会先调用 `stopTimer` 再调用 `startTimer`，类似于废弃之前的定时器，重新启动一个定时器。返回值和 `Stop` 一样。

Sleep 的内部实现

查看 `runtime/time.go` 文件中的 `timeSleep` 可知，`Sleep` 的是通过 `Timer` 实现的，把当前 goroutine 作为 `arg` 参数（ `getg()` ）。

Ticker 相关函数或方法的使用

`Ticker` 和 `Timer` 类似，区别是：`Ticker` 中的 `runtimeTimer` 字段的 `period` 字段会赋值为 `NewTicker(d Duration)` 中的 `d`，表示每间隔 `d` 纳秒，定时器就会触发一次。

除非程序终止前定时器一直需要触发，否则，不需要时应该调用 `Ticker.Stop` 来释放相关资源。

如果程序终止前需要定时器一直触发，可以使用更简单方便的 `time.Tick` 函数，因为 `Ticker` 实例隐藏起来了，因此，该函数启动的定时器无法停止。

定时器的实际应用

在实际开发中，定时器用的较多的会是 `Timer`，如模拟超时，而需要类似 `Tiker` 的功能时，可以使用实现了 `cron spec` 的库 cron，感兴趣的可以参考文章：《Go语言版 crontab》。

导航

- 上一节：Time类型详解
- 下一节：Unix 时间相关系统调用

5.1 math — 基本数学函数

math包实现的就是数学函数计算。

5.1.1 三角函数

正弦函数，反正弦函数，双曲正弦，反双曲正弦

- func Sin(x float64) float64
- func Asin(x float64) float64
- func Sinh(x float64) float64
- func Asinh(x float64) float64

一次性返回sin,cos

- func Sincos(x float64) (sin, cos float64)

余弦函数，反余弦函数，双曲余弦，反双曲余弦

- func Cos(x float64) float64
- func Acos(x float64) float64
- func Cosh(x float64) float64
- func Acosh(x float64) float64

正切函数，反正切函数，双曲正切，反双曲正切

- func Tan(x float64) float64
- func Atan(x float64) float64 和 func Atan2(y, x float64) float64
- func Tanh(x float64) float64
- func Atanh(x float64) float64

5.1.2 幂次函数

- func Cbrt(x float64) float64 //立方根函数
- func Pow(x, y float64) float64 // x的幂函数
- func Pow10(e int) float64 // 10根的幂函数
- func Sqrt(x float64) float64 // 平方根
- func Log(x float64) float64 // 对数函数
- func Log10(x float64) float64 // 10为底的对数函数
- func Log2(x float64) float64 // 2为底的对数函数

- func Log1p(x float64) float64 // log(1 + x)
- func Logb(x float64) float64 // 相当于log2(x)的绝对值
- func Ilogb(x float64) int // 相当于log2(x)的绝对值的整数部分
- func Exp(x float64) float64 // 指数函数
- func Exp2(x float64) float64 // 2为底的指数函数
- func Expm1(x float64) float64 // Exp(x) - 1

5.1.3 特殊函数

- func Inf(sign int) float64 // 正无穷
- func IsInf(f float64, sign int) bool // 是否正无穷
- func NaN() float64 // 无穷值
- func IsNaN(f float64) (is bool) // 是否是无穷值
- func Hypot(p, q float64) float64 // 计算直角三角形的斜边长

5.1.4 类型转化函数

- func Float32bits(f float32) uint32 // float32和unit32的转换
- func Float32frombits(b uint32) float32 // uint32和float32的转换
- func Float64bits(f float64) uint64 // float64和uint64的转换
- func Float64frombits(b uint64) float64 // uint64和float64的转换

5.1.5 其他函数

- func Abs(x float64) float64 // 绝对值函数
- func Ceil(x float64) float64 // 向上取整
- func Floor(x float64) float64 // 向下取整
- func Mod(x, y float64) float64 // 取模
- func Modf(f float64) (int float64, frac float64) // 分解f，以得到f的整数和小数部分
- func Frexp(f float64) (frac float64, exp int) // 分解f，得到f的位数和指数
- func Max(x, y float64) float64 // 取大值
- func Min(x, y float64) float64 // 取小值
- func Dim(x, y float64) float64 // 复数的维数
- func J0(x float64) float64 // 0阶贝塞尔函数
- func J1(x float64) float64 // 1阶贝塞尔函数
- func Jn(n int, x float64) float64 // n阶贝塞尔函数
- func Y0(x float64) float64 // 第二类贝塞尔函数0阶
- func Y1(x float64) float64 // 第二类贝塞尔函数1阶

- func Yn(n int, x float64) float64 // 第二类贝塞尔函数n阶
- func Erf(x float64) float64 // 误差函数
- func Erfc(x float64) float64 // 余补误差函数
- func Copysign(x, y float64) float64 // 以y的符号返回x值
- func Signbit(x float64) bool // 获取x的符号
- func Gamma(x float64) float64 // 伽玛函数
- func Lgamma(x float64) (lgamma float64, sign int) // 伽玛函数的自然对数
- func Ldexp(frac float64, exp int) float64 // value乘以2的exp次幂
- func Nextafter(x, y float64) (r float64) //返回参数x在参数y方向上可以表示的最接近的数值，若x等于y，则返回x
- func Nextafter32(x, y float32) (r float32) //返回参数x在参数y方向上可以表示的最接近的数值，若x等于y，则返回x
- func Remainder(x, y float64) float64 // 取余运算
- func Trunc(x float64) float64 // 截取函数

导航

第六章 文件系统

Go 的标准库提供了很多工具，可以处理文件系统中的文件、构造和解析文件名等。

处理文件的第一步是确定要处理的文件的名字。Go 将文件名表示为简单的字符串，提供了 `path`、`filepath` 等库来操作文件名或路径。用 `os` 中 `File` 结构的 `Readdir` 可以列出一个目录中的内容。

可以用 `os.Stat` 或 `os.Lstat` 来检查文件的一些特性，如权限、大小等。

有时需要创建草稿文件来保存临时数据，或将数据移动到一个永久位置之前需要临时文件存储，`os.TempDir` 可以返回默认的临时目录，用于存放临时文件。关于临时文件，在 `ioutil` 中已经讲解了。

`os` 包还包含了很多其他文件系统相关的操作，比如创建目录、重命名、移动文件等等。

由于本章探讨文件系统相关知识，`os` 包中关于进程相关的知识会在后续章节讲解。

导航

- 第五章
- 下一节：os — 平台无关的操作系统功能实现

6.1 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包的接口规定实现为在所有操作系统中都是一致的。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。

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，所有执行 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，并不说一定存在的文件名，比如 Windows 下就没有。

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)
}
```

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。

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` ）

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，我们不回去检查 `Close` 的错误。

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。

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。

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()))
```

改变文件时间戳

可以显示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。

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。

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。
```

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。

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。

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。

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`。

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。

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。

导航

- 第六章
- 下一节：path — 操作路径

6.2 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。

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，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。

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/l
ogic/topic.go"))
fmt.Println(filepath.Rel("/home/polaris/studygolang", "/data/studygolang"))

// Output:
// src/logic/topic.go <nil>
// ../../../data/studygolang <nil>
```

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`。

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：

1. 将连续的多个路径分隔符替换为单个路径分隔符
2. 剔除每一个 `.` 路径名元素（代表当前目录）
3. 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
4. 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。

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 studygola
ng.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>
```

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")
```

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，程序简单很多。

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。

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。

导航

- 下一节：os — 平台无关的操作系统功能实现
- 第七章：数据持久存储与交换

第七章 数据持久存储与交换

现代程序离不开数据存储，现阶段很热的所谓大数据处理、云盘等，更是以存储为依托。有数据存储，自然需要进行数据交换，已达到数据共享等目的。

关系型数据库发展了很长一段时间，SQL/SQL-like 已经很成熟，使用也很广泛，Go 语言标准库提供了对 SQL/SQL-like 数据库的操作的标准接口，即 database/sql 包。

在数据交换方面，有很多成熟的协议可以使用，常用的有：JSON、XML等，似乎 Java 社区更喜欢 XML，而目前似乎使用更多的是 JSON。在交换协议选择方面，考虑的主要这几个方面因素：性能、跨语言（通用性）、传输量等。因此，对于性能要求高的场景，会使用 protobuf、msgpack 之类的协议。由于 JSON 和 XML 使用很广泛，Go 语言提供了解析它们的标准库；同时，为了方便 Go 程序直接数据交换，Go 专门提供了 gob 这种交换协议。

导航

- 目录
- 下一节：database/sql — SQL/SQL-Like 数据库操作接口

7.1 database/sql — SQL/SQL-Like 数据库操作接口

这是 Go 提供的操作 SQL/SQL-Like 数据库的通用接口，但 Go 标准库并没有提供具体数据库的实现，需要结合第三方的驱动来使用该接口。本书使用的是 mysql 的驱动：github.com/go-sql-driver/mysql。

注：该包有一个子包：*driver*，它定义了一些接口供数据库驱动实现，一般业务代码中使用 *database/sql* 包即可，尽量避免使用 *driver* 这个子包。

7.1.1 database/sql 是什么？

很明显，database/sql 首先是 Go 标准库提供的一个包，用于和 SQL/SQL-Like 数据库(关系或类似关系数据库）通讯。它提供了和 ODBC、Perl的DBI、Java的JDBC和PHP的PDO 类似的功能。然而，它的设计却不太一样，掌握了它有利于构建健壮、高性能的基于 database 的应用。

另一方面，database/sql 提供的是抽象概念，和具体数据库无关，具体的数据库实现，有驱动来做，这样可以很方便的更换数据库。

该包提供了一些类型（概括性的），每个类型可能包括一个或多个概念。

- DB
  sql.DB 类型代表了一个数据库。这点和很多其他语言不同，它并不代表一个到数据库的具体连接，而是一个能操作的数据库对象，具体的连接在内部通过连接池来管理，对外不暴露。这点是很多人容易误解的：每一次数据库操作，都产生一个 sql.DB 实例，操作完 Close。
- Results
  定义了三种结果类型：sql.Rows、sql.Row 和 sql.Result，分别用于获取多个多行结果、一行结果和修改数据库影响的行数（或其返回last insert id）。
- Statements
  sql.Stmt 代表一个语句，如：DDL、DML等。
- Transactions
  sql.Tx 代表带有特定属性的一个事务。

7.1.2 sql.DB 的使用

官方文档关于 DB 的描述：

> 是一个数据库句柄，代表一个具有零到多个底层连接的连接池，它可以安全的被多个 goroutine 同时使用。
> sql包会自动创建和释放连接；它也会维护一个闲置连接的连接池。如果数据库具有单连接状态的概念，该状态只有在事务中被观察时才可信。一旦调用了BD.Begin，返回的Tx会绑定到单个连接。当调用事务Tx的Commit或Rollback后，该事务使用的连接会归还到DB的闲置连接池中。连接池的大小可以用SetMaxIdleConns方法控制。

由于 DB 并非一个实际的到数据库的连接，而且可以被多个 goroutine 并发使用，因此，程序中只需要拥有一个全局的实例即可。所以，经常见到的示例代码：

```
db, err := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")
if err != nil {
    panic(err)
}
defer db.Close()
```

实际中，`defer db.Close()` 可以不调用，官方文档关于 DB.Close 的说明也提到了：Close 用于关闭数据库，释放任何打开的资源。一般不会关闭 DB，因为 DB 句柄通常被多个 goroutine 共享，并长期活跃。当然，如果你确定 DB 只会被使用一次，之后不会使用了，应该调用 Close。

所以，实际的 Go 程序，应该在一个go文件中的 init 函数中调用 `sql.Open` 初始化全局的 sql.DB 对象，供程序中所有需要进行数据库操作的地方使用。

前面说过，sql.DB 并不是实际的数据库连接，因此，sql.Open 函数并没有进行数据库连接，只有在驱动未注册时才会返回 `err != nil`。

例如：`db, err := sql.Open("mysql", "root:@tcp23(localhost233:3306)/test?charset=utf8")`。虽然这里的 dsn 是错误的，但依然 `err == nil`，只有在实际操作数据库（查询、更新等）或调用 `Ping` 时才会报错。

关于 Open 函数的参数，第一个是驱动名，为了避免混淆，一般和驱动包名一致，在驱动实现中，会有类似这样的代码：

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

其中 mysql 即是注册的驱动名。由于注册驱动是在 init 函数中进行的，这也就是为什么采用 `_ "github.com/go-sql-driver/mysql"` 这种方式引入驱动包。第二个参数是 DSN（数据源名称），这个是和具体驱动相关的，database/sql 包并没有规定，具体书写方式参见驱动文档。

7.1.2.1 连接池的工作原理

获取 DB 对象后，连接池是空的，第一个连接在需要的时候才会创建。可以通过下面的代码验证这一点：

```
db, _ := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")
fmt.Println("please exec show processlist")
time.Sleep(10 * time.Second)
fmt.Println("please exec show processlist again")
db.Ping()
time.Sleep(10 * time.Second)
```

在 Ping 执行之前和之后，show processlist 多了一条记录，即多了一个连接，Command 列是 Sleep。

连接池的工作方式：当调用一个函数，需要访问数据库时，该函数会请求从连接池中获取一个连接，如果连接池中存在一个空闲连接，它会将该空闲连接给该函数；否则，会打开一个新的连接。当该函数结束时，该连接要么返回给连接池，要么传递个某个需要该连接的对象，知道该对象完成时，连接才会返回给连接池。相关方法的处理说明（假设 sql.DB 的对象是 db）：

- **db.Ping()** 会将连接立马返回给连接池。
- **db.Exec()** 会将连接立马返回给连接池，但是它返回的 Result 对象会引用该连接，所以，之后可能会再次被使用。
- **db.Query()** 会传递连接给 sql.Rows 对象，直到完全遍历了所有的行或 Rows 的 Close 方法被调用了，连接才会返回给连接池。
- **db.QueryRow()** 会传递连接给 sql.Row 对象，当该对象的 Scan 方法被调用时，连接会返回给连接池。
- **db.Begin()** 会传递连接给 sql.Tx 对象，当该对象的 Commit 或 Rollback 方法被调用时，该链接会返回给连接池。

从上面的解释可以知道，大部分时候，我们不需要关心连接不释放问题，它们会自动返回给连接池，只有 Query 方法有点特殊，后面讲解如何处理。

注意：如果某个连接有问题（broken connection)，database/sql 内部会进行最多10次的重试，从连接池中获取或新开一个连接来服务，因此，你的代码中不需要重试的逻辑。

7.1.2.2 控制连接池

Go1.2.1 之前，没法控制连接池，Go1.2.1 之后，提供了两个方法来控制连接池（Go1.2 提供了控制，不过有bug）。

- **db.SetMaxOpenConns(n int)** 设置连接池中最多保存打开多少个数据库连接。注意，它包括在使用的和空闲的。如果某个方法调用需要一个连接，但连接池中没有空闲的可用，且打开的连接数达到了该方法设置的最大值，该方法调用将堵塞。默认限制是0，表示最大打开数没有限制。

- **db.SetMaxIdleConns(n int)** 设置连接池中能够保持的最大空闲连接的数量。默认值是2

上面的两个设置，可以用程序实际测试。比如通过下面的代码，可以验证 MaxIdleConns 是 2：

```
db, _ := sql.Open("mysql", "root:@tcp(localhost:3306)/test?charset=utf8")

// 去掉注释，可以看看相应的空闲连接是不是变化了
// db.SetMaxIdleConns(3)

for i := 0; i < 10; i++ {
    go func() {
        db.Ping()
    }()
}

time.Sleep(20 * time.Second)
```

通过 show processlist 命令，可以看到有两个是 Sleep 的连接。

导航

- 第七章 数据持久存储与交换
- 下一节：encoding/json — json 解析

第九章 测试

Go语言从开发初期就注意了测试用例的编写。特别是静态语言，由于调试没有动态语言那么方便，所以能最快最方便地编写一个测试用例就显得非常重要了。

本章内容涵盖了Go标准库中的3的包：

testing 包含基本的testing使用接口

导航

- 第二章
- 下一节：testing - 测试基本使用接口

testing - 测试基本使用接口

当你写完一个函数，结构体，main之后，你下一步需要的就是测试了。testing包提供了很简单易用的测试包。

写一个基本的测试用例

测试文件的文件名需要以_test.go为结尾，测试用例需要以TestXxxx的样式存在。

比如我要测试utils包的sql.go中的函数：

```
func GetOne(db *sql.DB, query string, args ...interface{}) (map[string][]byte, error)
{
```

就需要创建一个sql_test.go

```
package utils

import (
    "database/sql"
    _ "fmt"
    _ "github.com/go-sql-driver/mysql"
    "strconv"
    "testing"
)

func Test_GetOne(t *testing.T) {
    db, err := sql.Open("mysql", "root:123.abc@tcp(192.168.33.10:3306)/test")
    defer func() {
        db.Close()
    }()
    if err != nil {
        t.Fatal(err)
    }

    // 测试empty
    car_brand, err := GetOne(db, "select * from user where id = 999999")
    if (car_brand != nil) || (err != nil) {
        t.Fatal("emtpy测试错误")
    }
}
```

testing的测试用例形式

测试用例有四种形式： TestXxxx(t *testing.T) // 基本测试用例 *BenchmarkXxxx(b* testing.B) // 压力测试的测试用例 Example_Xxx() // 测试控制台输出的例子 TestMain(m *testing.M) // 测试Main函数

给个Example的例子:（Example需要在最后用注释的方式确认控制台输出和预期是不是一致的）

```
func Example_GetScore() {
    score := getScore(100, 100, 100, 2.1)
    fmt.Println(score)
    // Output:
    // 31.1
}
```

testing的变量

gotest的变量有这些：

- test.short : 一个快速测试的标记，在测试用例中可以使用testing.Short()来绕开一些测试
- test.outputdir : 输出目录
- test.coverprofile : 测试覆盖率参数，指定输出文件
- test.run : 指定正则来运行某个/某些测试用例
- test.memprofile : 内存分析参数，指定输出文件
- test.memprofilerate : 内存分析参数，内存分析的抽样率
- test.cpuprofile : cpu分析输出参数，为空则不做cpu分析
- test.blockprofile : 阻塞事件的分析参数，指定输出文件
- test.blockprofilerate : 阻塞事件的分析参数，指定抽样频率
- test.timeout : 超时时间
- test.cpu : 指定cpu数量
- test.parallel : 指定运行测试用例的并行数

testing包内的结构

- B : 压力测试
- BenchmarkResult : 压力测试结果
- Cover : 代码覆盖率相关结构体
- CoverBlock : 代码覆盖率相关结构体

- InternalBenchmark : 内部使用的结构
- InternalExample : 内部使用的结构
- InternalTest : 内部使用的结构
- M : main测试使用的结构
- PB : Parallel benchmarks 并行测试使用结果
- T : 普通测试用例
- TB : 测试用例的接口

testing的通用方法

T结构内部是继承自common结构，common结构提供集中方法，是我们经常会用到的：

当我们遇到一个断言错误的时候，我们就会判断这个测试用例失败，就会使用到：

```
Fail  : case失败，测试用例继续
FailedNow : case失败，测试用例中断
```

当我们遇到一个断言错误，只希望跳过这个错误，但是不希望标示测试用例失败，会使用到：

```
SkipNow : case跳过，测试用例不继续
```

当我们只希望在一个地方打印出信息，我们会用到:

```
Log : 输出信息
Logf : 输出有format的信息
```

当我们希望跳过这个用例，并且打印出信息:

```
Skip : Log + SkipNow
Skipf : Logf + SkipNow
```

当我们希望断言失败的时候，测试用例失败，打印出必要的信息，但是测试用例继续：

```
Error : Log + Fail
Errorf : Logf + Fail
```

当我们希望断言失败的时候，测试用例失败，打印出必要的信息，测试用例中断：

```
Fatal : Log + FailNow
Fatalf : Logf + FailNow
```

第十章 进程、线程和 goroutine

本章将研究 Go 语言进程、线程和 goroutine，会涉及到操作系统关于进程、线程的知识，同时研究 Go 语言提供的相关标准库 API；goroutine 作为 Go 的一个核心特性，本章会重点介绍。

虽然标准库中能操作进程、线程和 goroutine 的API不多，但它们是深入学习、理解 Go 语言必须掌握的知识。本章从操作系统和 Go 源码层面深入探讨它们。

导航

- 第九章
- 下一节：创建进程

10.1 创建进程

`os` 包及其子包 `os/exec` 提供了创建进程的方法。

一般的，应该优先使用 `os/exec` 包。因为 `os/exec` 包依赖 `os` 包中关键创建进程的 API，为了便于理解，我们先探讨 `os` 包中和进程相关的部分。

进程的创建

在 Unix 中，创建一个进程，通过系统调用 `fork` 实现（及其一些变种，如 vfork、clone）。在 Go 语言中，Linux 下创建进程使用的系统调用是 `clone`。

很多时候，系统调用 `fork`、`execve`、`wait` 和 `exit` 会在一起出现。此处先简要介绍这 4 个系统调用及其典型用法。

- fork：允许一进程（父进程）创建一新进程（子进程）。具体做法是，新的子进程几近于对父进程的翻版：子进程获得父进程的栈、数据段、堆和执行文本段的拷贝。可将此视为把父进程一分为二。
- exit(status)：终止一进程，将进程占用的所有资源（内存、文件描述符等）归还内核，交其进行再次分配。参数 `status` 为一整型变量，表示进程的退出状态。父进程可使用系统调用 `wait()` 来获取该状态。
- wait(&status)目的有二：其一，如果进程尚未调用 `exit()` 终止，那么 `wait` 会挂起父进程直至子进程终止；其二，子进程的终止状态通过 `wait` 的 `status` 参数返回。
- execve(pathname, argv, envp) 加载一个新程序（路径名为 pathname，参数列表为 argv，环境变量列表为 envp）到当前进程的内存。这将丢弃现存的程序文本段，并为新程序重新创建栈、数据段以及堆。通常将这一动作称为执行一个新程序。

在 Go 语言中，没有直接提供 `fork` 系统调用的封装，而是将 `fork` 和 `execve` 合二为一，提供了 `syscall.ForkExec`。如果想只调用 `fork`，得自己通过 `syscall.Syscall(syscall.SYS_FORK, 0, 0, 0)` 实现。

Process 及其相关方法

`os.Process` 存储了通过 `StartProcess` 创建的进程的相关信息。

```
type Process struct {
    Pid    int
    handle uintptr // handle is accessed atomically on Windows
    isdone uint32  // process has been successfully waited on, non zero if true
}
```

一般通过 `StartProcess` 创建 `Process` 的实例，函数声明如下：

```
func StartProcess(name string, argv []string, attr *ProcAttr) (*Process, error)
```

它使用提供的程序名、命令行参数、属性开始一个新进程。`StartProcess` 是一个低级别的接口。`os/exec` 包提供了高级别的接口，一般应该尽量使用 `os/exec` 包。如果出错，错误的类型会是 `*PathError` 。

其中的参数 `attr` ，类型是 `ProcAttr` 的指针，用于为 `StartProcess` 创建新进程提供一些属性。定义如下：

```
type ProcAttr struct {
    // 如果 Dir 非空，子进程会在创建 Process 实例前先进入该目录。（即设为子进程的当前工作目录）
    Dir string
    // 如果 Env 非空，它会作为新进程的环境变量。必须采用 Environ 返回值的格式。
    // 如果 Env 为 nil，将使用 Environ 函数的返回值。
    Env []string
    // Files 指定被新进程继承的打开文件对象。
    // 前三个绑定为标准输入、标准输出、标准错误输出。
    // 依赖底层操作系统的实现可能会支持额外的文件对象。
    // nil 相当于在进程开始时关闭的文件对象。
    Files []*File
    // 操作系统特定的创建属性。
    // 注意设置本字段意味着你的程序可能会执行异常甚至在某些操作系统中无法通过编译。这时候可以通过为特定系统设置。
    // 看 syscall.SysProcAttr 的定义，可以知道用于控制进程的相关属性。
    Sys *syscall.SysProcAttr
}
```

`FindProcess` 可以通过 `pid` 查找一个运行中的进程。该函数返回的 `Process` 对象可以用于获取关于底层操作系统进程的信息。在 Unix 系统中，此函数总是成功，即使 `pid` 对应的进程不存在。

```
func FindProcess(pid int) (*Process, error)
```

`Process` 提供了四个方法：`Kill` 、`Signal` 、`Wait` 和 `Release` 。其中 `Kill` 和 `Signal` 跟信号相关，而 `Kill` 实际上就是调用 `Signal` ，发送了 `SIGKILL` 信号，强制进程退出，关于信号，后续章节会专门讲解。

`Release` 方法用于释放 `Process` 对象相关的资源，以便将来可以被再使用。该方法只有在确定没有调用 `Wait` 时才需要调用。Unix 中，该方法的内部实现只是将 `Process` 的 `pid` 置为 -1。

我们重点看看 `Wait` 方法。

```
func (p *Process) Wait() (*ProcessState, error)
```

在多进程应用程序的设计中，父进程需要知道某个子进程何时改变了状态 —— 子进程终止或因收到信号而停止。`Wait` 方法就是一种用于监控子进程的技术。

`Wait` 方法阻塞直到进程退出，然后返回一个 `ProcessState` 描述进程的状态和可能的错误。`Wait` 方法会释放绑定到 `Process` 的所有资源。在大多数操作系统中，`Process` 必须是当前进程的子进程，否则会返回错误。

看看 `ProcessState` 的内部结构：

```
type ProcessState struct {
    pid    int                // The process's id.
   status syscall.WaitStatus // System-dependent status info.
   rusage *syscall.Rusage
}
```

`ProcessState` 保存了 `Wait` 函数报告的某个进程的信息。`status` 记录了状态原因，通过 `syscal.WaitStatus` 类型定义的方法可以判断：

- Exited()：是否正常退出，如调用 `os.Exit` ；
- Signaled()：是否收到未处理信号而终止；
- CoreDump()：是否收到未处理信号而终止，同时生成 coredump 文件，如 SIGABRT；
- Stopped()：是否因信号而停止（SIGSTOP）；
- Continued()：是否因收到信号 SIGCONT 而恢复；

`syscal.WaitStatus` 还提供了其他一些方法，比如获取退出状态、信号、停止信号和中断（Trap）原因。

因为 Linux 下 `Wait` 的内部实现使用的是 `wait4` 系统调用，因此，`ProcessState` 中包含了 `rusage` ，用于统计进程的各类资源信息。一般情况下，`syscall.Rusage` 中定义的信息都用不到，如果实际中需要使用，可以查阅 Linux 系统调用 `getrusage` 获得相关说明（`getrusage(2)`）。

`ProcessState` 结构内部字段是私有的，我们可以通过它提供的方法来获得一些基本信息，比如：进程是否退出、Pid、进程是否是正常退出、进程CPU时间、用户时间等等。

实现类似 Linux 中 `time` 命令的功能：

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "path/filepath"
    "time"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Printf("Usage: %s [command]\n", os.Args[0])
```

```
        os.Exit(1)
    }

    cmdName := os.Args[1]
    if filepath.Base(os.Args[1]) == os.Args[1] {
        if lp, err := exec.LookPath(os.Args[1]); err != nil {
            fmt.Println("look path error:", err)
            os.Exit(1)
        } else {
            cmdName = lp
        }
    }

    procAttr := &os.ProcAttr{
        Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
    }

    cwd, err := os.Getwd()
    if err != nil {
        fmt.Println("look path error:", err)
        os.Exit(1)
    }

    start := time.Now()
    process, err := os.StartProcess(cmdName, []string{cwd}, procAttr)
    if err != nil {
        fmt.Println("start process error:", err)
        os.Exit(2)
    }

    processState, err := process.Wait()
    if err != nil {
        fmt.Println("wait error:", err)
        os.Exit(3)
    }

    fmt.Println()
    fmt.Println("real", time.Now().Sub(start))
    fmt.Println("user", processState.UserTime())
    fmt.Println("system", processState.SystemTime())
}

// go build main.go && ./main ls
// Output:
//
// real 4.994739ms
// user 1.177ms
// system 2.279ms
```

运行外部命令

通过 `os` 包可以做到运行外部命令，如前面的例子。不过，Go 标准库为我们封装了更好用的包： `os/exec` ，运行外部命令，应该优先使用它，它包装了 `os.StartProcess` 函数以便更容易的重定向标准输入和输出，使用管道连接I/O，以及作其它的一些调整。

查找可执行程序

`exec.LookPath` 函数在 `PATH` 指定目录中搜索可执行程序，如 `file` 中有 `/` ，则只在当前目录搜索。该函数返回完整路径或相对于当前路径的一个相对路径。

```
func LookPath(file string) (string, error)
```

如果在 `PATH` 中没有找到可执行文件，则返回 `exec.ErrNotFound` 。

Cmd 及其相关方法

`Cmd` 结构代表一个正在准备或者在执行中的外部命令，调用了 `Run` 、 `Output` 或 `CombinedOutput` 后， `Cmd` 实例不能被重用。

```
type Cmd struct {
    // Path 是将要执行的命令路径。
    // 该字段不能为空（也是唯一一个不能为空的字段），如为相对路径会相对于 Dir 字段。
    // 通过 Command 初始化时，会在需要时调用 LookPath 获得完整的路径。
    Path string

    // Args 存放着命令的参数，第一个值是要执行的命令（Args[0]）；如果为空切片或者nil，使用 {Path}
运行。
    // 一般情况下，Path 和 Args 都应被 Command 函数设定。
    Args []string

    // Env 指定进程的环境变量，如为 nil，则使用当前进程的环境变量，即 os.Environ()，一般就是当前系
统的环境变量。
    Env []string

    // Dir 指定命令的工作目录。如为空字符串，会在调用者的进程当前工作目录下执行。
    Dir string

    // Stdin 指定进程的标准输入，如为 nil，进程会从空设备读取（os.DevNull）
    // 如果 Stdin 是 *os.File 的实例，进程的标准输入会直接指向这个文件
    // 否则，会在一个单独的 goroutine 中从 Stdin 中读数据，然后将数据通过管道传递到该命令中（也就
是从 Stdin 读到数据后，写入管道，该命令可以从管道读到这个数据）。在 goroutine 停止数据拷贝之前（停止
的原因如遇到EOF或其他错误，或管道的 write 端错误），Wait 方法会一直堵塞。
    Stdin io.Reader

    // Stdout 和 Stderr 指定进程的标准输出和标准错误输出。
    // 如果任一个为 nil，Run 方法会将对应的文件描述符关联到空设备（os.DevNull）
    // 如果两个字段相同，同一时间最多有一个线程可以写入。
    Stdout io.Writer
    Stderr io.Writer

    // ExtraFiles 指定额外被新进程继承的已打开文件，不包括标准输入、标准输出、标准错误输出。
    // 如果本字段非 nil，其中的元素 i 会变成文件描述符 3+i。
    //
    // BUG: 在OS X 10.6系统中，子进程可能会继承不期望的文件描述符。
    // http://golang.org/issue/2603
    ExtraFiles []*os.File

    // SysProcAttr 提供可选的、各操作系统特定的 sys 属性。
    // Run 方法会将它作为 os.ProcAttr 的 Sys 字段传递给os.StartProcess 函数。
    SysProcAttr *syscall.SysProcAttr

    // Process 是底层的，只执行一次的进程。
    Process *os.Process

    // ProcessState 包含一个已经存在的进程的信息，只有在调用 Wait 或 Run 后才可用。
    ProcessState *os.ProcessState
}
```

Command

一般的，应该通过 `exec.Command` 函数产生 `Cmd` 实例：

```
func Command(name string, arg ...string) *Cmd
```

该函数返回一个 `*Cmd`，用于使用给出的参数执行 `name` 指定的程序。返回的 `*Cmd` 只设定了 `Path` 和 `Args` 两个字段。

如果 `name` 不含路径分隔符，将使用 `LookPath` 获取完整路径；否则直接使用 `name`。参数 `arg` 不应包含命令名。

得到 `*Cmd` 实例后，接下来一般有两种写法：

1. 调用 `Start()`，接着调用 `Wait()`，然后会阻塞直到命令执行完成；
2. 调用 `Run()`，它内部会先调用 `Start()`，接着调用 `Wait()`；

Start

```
func (c *Cmd) Start() error
```

开始执行 `c` 包含的命令，但并不会等待该命令完成即返回。`Wait` 方法会返回命令的退出状态码并在命令执行完后释放相关的资源。内部调用 `os.StartProcess`，执行 `forkExec`。

Wait

```
func (c *Cmd) Wait() error
```

`Wait` 会阻塞直到该命令执行完成，该命令必须是先通过 `Start` 执行。

如果命令成功执行，stdin、stdout、stderr 数据传递没有问题，并且返回状态码为 0，方法的返回值为 nil；如果命令没有执行或者执行失败，会返回 `*ExitError` 类型的错误；否则返回的 error 可能是表示 I/O 问题。

如果 `c.Stdin` 不是 `*os.File` 类型，`Wait` 会等待，直到数据从 `c.Stdin` 拷贝到进程的标准输入。

`Wait` 方法会在命令返回后释放相关的资源。

Output

除了 `Run()` 是 `Start` + `Wait` 的简便写法，`Output()` 更是 `Run()` 的简便写法，外加获取外部命令的输出。

```
func (c *Cmd) Output() ([]byte, error)
```

它要求 `c.Stdout` 必须是 `nil`，内部会将 `bytes.Buffer` 赋值给 `c.Stdout`，在 `Run()` 成功返回后，会将 `Buffer` 的结果返回（`stdout.Bytes()`）。

CombinedOutput

`Output()` 只返回 `Stdout` 的结果，而 `CombinedOutput` 组合 `Stdout` 和 `Stderr` 的输出，即 `Stdout` 和 `Stderr` 都赋值为同一个 `bytes.Buffer`。

StdoutPipe、StderrPipe 和 StdinPipe

除了上面介绍的 `Output` 和 `CombinedOutput` 直接获取命令输出结果外，还可以通过 `StdoutPipe` 返回 `io.ReadCloser` 来获取输出；相应的 `StderrPipe` 得到错误信息；而 `StdinPipe` 则可以往命令写入数据。

```
func (c *Cmd) StdoutPipe() (io.ReadCloser, error)
```

`StdoutPipe` 方法返回一个在命令 `Start` 执行后与命令标准输出关联的管道。`Wait` 方法会在命令结束后会关闭这个管道，所以一般不需要手动关闭该管道。但是在从管道读取完全部数据之前调用 `Wait` 出错了，则必须手动关闭。

```
func (c *Cmd) StderrPipe() (io.ReadCloser, error)
```

`StderrPipe` 方法返回一个在命令 `Start` 执行后与命令标准错误输出关联的管道。`Wait` 方法会在命令结束后会关闭这个管道，一般不需要手动关闭该管道。但是在从管道读取完全部数据之前调用 `Wait` 出错了，则必须手动关闭。

```
func (c *Cmd) StdinPipe() (io.WriteCloser, error)
```

`StdinPipe` 方法返回一个在命令 `Start` 执行后与命令标准输入关联的管道。`Wait` 方法会在命令结束后会关闭这个管道。必要时调用者可以调用 `Close` 方法来强行关闭管道。例如，标准输入已经关闭了，命令执行才完成，这时调用者需要显示关闭管道。

因为 `Wait` 之后，会将管道关闭，所以，要使用这些方法，只能使用 `Start` + `Wait` 组合，不能使用 `Run`。

执行外部命令示例

前面讲到，通过 `Cmd` 实例后，有两种方式运行命令。有时候，我们不只是简单的运行命令，还希望能控制命令的输入和输出。通过上面的 API 介绍，控制输入输出有几种方法：

- 得到 `Cmd` 实例后，直接给它的字段 `Stdin`、`Stdout` 和 `Stderr` 赋值；
- 通过 `Output` 或 `CombinedOutput` 获得输出；
- 通过带 `Pipe` 后缀的方法获得管道，用于输入或输出；

直接赋值 `Stdin`、`Stdout` 和 `Stderr`

```
func FillStd(name string, arg ...string) ([]byte, error) {
    cmd := exec.Command(name, arg...)
    var out = new(bytes.Buffer)

    cmd.Stdout = out
    cmd.Stderr = out

    err := cmd.Run()
    if err != nil {
        return nil, err
    }

    return out.Bytes(), nil
}
```

使用 `Output`

```
func UseOutput(name string, arg ...string) ([]byte, error) {
    return exec.Command(name, arg...).Output()
}
```

使用 Pipe

```
func UsePipe(name string, arg ...string) ([]byte, error) {
    cmd := exec.Command(name, arg...)
    stdout, err := cmd.StdoutPipe()
    if err != nil {
        return nil, err
    }

    if err = cmd.Start(); err != nil {
        return nil, err
    }

    var out = make([]byte, 0, 1024)
    for {
        tmp := make([]byte, 128)
        n, err := stdout.Read(tmp)
        out = append(out, tmp[:n]...)
        if err != nil {
            break
        }
    }

    if err = cmd.Wait(); err != nil {
        return nil, err
    }

    return out, nil
}
```

完整代码见 os_exec。

进程终止

`os.Exit()` 函数会终止当前进程，对应的系统调用不是 `_exit`，而是 `exit_group`。

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

`Exit` 让当前进程以给出的状态码 `code` 退出。一般来说，状态码 0 表示成功，非 0 表示出错。进程会立刻终止，defer 的函数不会被执行。

导航

- 第十章
- 下一节：进程属性和控制

10.2 进程属性和控制

每个进程都有一些属性，`os` 包提供了一些函数可以获取进程属性。

进程 ID

每个进程都会有一个进程ID，可以通过 `os.Getpid` 获得。同时，每个进程都有创建自己的父进程，通过 `os.Getppid` 获得。

进程凭证

Unix 中进程都有一套数字表示的用户 ID(UID) 和组 ID(GID)，有时也将这些 ID 称之为进程凭证。Windows 下总是 -1。

实际用户 ID 和实际组 ID

实际用户 ID（real user ID）和实际组 ID（real group ID）确定了进程所属的用户和组。登录 shell 从 `/etc/passwd` 文件读取用户 ID 和组 ID。当创建新进程时（如 shell 执行程序），将从其父进程中继承这些 ID。

可通过 `os.Getuid()` 和 `os.Getgid()` 获取当前进程的实际用户 ID 和实际组 ID；

有效用户 ID 和有效组 ID

大多数 Unix 实现中，当进程尝试执行各种操作（即系统调用）时，将结合有效用户 ID、有效组 ID，连同辅助组 ID 一起来确定授予进程的权限。内核还会使用有效用户 ID 来决定一个进程是否能向另一个进程发送信号。

有效用户 ID 为 0（root 的用户 ID）的进程拥有超级用户的所有权限。这样的进程又称为特权级进程（privileged process）。某些系统调用只能由特权级进程执行。

可通过 `os.Geteuid()` 和 `os.Getegid()` 获取当前进程的有效用户 ID（effective user ID）和有效组 ID（effectvie group ID）。

通常，有效用户 ID 及组 ID 与其相应的实际 ID 相等，但有两种方法能够致使二者不同。一是使用相关系统调用；二是执行 set-user-ID 和 set-group-ID 程序。

Set-User-ID 和 Set-Group-ID 程序

`set-user-ID` 程序会将进程的有效用户 ID 置为可执行文件的用户ID（属主），从而获得常规情况下并不具有的权限。`set-group-ID` 程序对进程有效组 ID 实现类似任务。（有时也将这程序简称为 set-UID 程序和 set-GID 程序。）

与其他文件一样，可执行文件的用户 ID 和组 ID 决定了该文件的所有权。在 6.1 os — 平台无关的操作系统功能实现 中提到过，文件还拥有两个特别的权限位 set-user-ID 位和 set-group-ID 位，可以使用 `os.Chmod` 修改这些权限位（非特权用户进程只能修改其自身文件，而特权用户进程能修改任何文件）。

文件设置了 set-user-ID 位后，`ls -l` 显示文件后，会在属主用户执行权限字段上看到字母 s（有执行权限时） 或 S（无执行权限时）；相应的 set-group-ID 则是在组用户执行位上看到 s 或 S。

当运行 set-user-ID 程序时，内核会将进程的有效用户 ID 设置为可执行文件的用户 ID。set-group-ID 程序对进程有效组 ID 的操作与之类似。通过这种方法修改进程的有效用户 ID 或组 ID，能够使进程（换言之，执行该程序的用户）获得常规情况下所不具有的权限。例如，如果一个可执行文件的属主为 root，且为此程序设置了 set-user-ID 权限位，那么当运行该程序时，进程会取得超级用户权限。

也可以利用程序的 set-user-ID 和 set-group-ID 机制，将进程的有效 ID 修改为 root 之外的其他用户。例如，为提供一个受保护文件的访问，可采用如下方案：创建一个具有对该文件访问权限的专有用户（组）ID，然后再创建一个 set-user-ID（set-group-ID）程序，将进程有效用户（组）ID 变更为这个专用 ID。这样，无需拥有超级用户的所有权限，程序就能访问该文件。

Linux 系统中经常使用的 set-user-ID 程序，如 passwd。

测试 **set-user-ID** 程序

在 Linux 的某个目录下，用 root 账号创建一个文件：

```
echo "This is my shadow, studygolang." > my_shadow.txt
```

然后将所有权限都去掉：`chmod 0 my_shadow.txt`。ls -l 结果类似如下：

```
---------- 1 root root 32 6月 24 17:31 my_shadow.txt
```

这时，如果非 root 用户是无法查看文件内容的。

接着，用 root 账号创建一个 `main.go` 文件，内容如下：

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "os"
)

func main() {
    file, err := os.Open("my_shadow.txt")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    data, err := ioutil.ReadAll(file)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("my_shadow:%s\n", data)
}
```

就是简单地读取 `my_shadow` 文件内容。`go build main.go` 后，生成的 `main` 可执行文件，权限是：`-rwxrwxr-x` 。

这时，切换到非 root 用户，执行 `./main` ，会输出：

```
open my_shadow.txt: permission denied
```

因为这时的 `main` 程序生成的进程有效用户 ID 是当前用户的（非 root）。

接着，给 `main` 设置 `set-user-ID` 位：`chmod u+s main` ，权限变为 `-rwsrwxr-x` ，非 root 下再次执行 `./main` ，输出：

```
my_shadow:This is my shadow, studygolang.
```

因为设置了 `set-user-ID` 位，这时 `main` 程序生成的进程有效用户是 `main` 文件的属主，即 root 的 ID，因此有权限读 `my_shadow.txt` 。

修改进程的凭证

`os` 包没有提供相应的功能修改进程的凭证，在 `syscall` 包对这些系统调用进行了封装。因为 https://golang.org/s/go1.4-syscall，用户程序不建议直接使用该包，应该使用 `golang.org/x/sys` 包代替。

该包提供了修改进程各种 ID 的系统调用封装，这里不一一介绍。

此外，`os` 还提供了获取辅助组 ID 的函数：`os.Getgroups()`。

操作系统用户

包 `os/user` 允许通过名称或 ID 查询用户账号。用户结构定义如下：

```
type User struct {
    Uid      string // user id
    Gid      string // primary group id
    Username string
    Name     string
    HomeDir  string
}
```

`User` 代表一个用户帐户。

在 POSIX 系统中 Uid 和 Gid 字段分别包含代表 uid 和 gid 的十进制数字。在 Windows 系统中 Uid 和 Gid 包含字符串格式的安全标识符（SID）。在 Plan 9 系统中，Uid、Gid、Username 和 Name 字段是 /dev/user 的内容。

`Current` 函数可以获取当前用户账号。而 `Lookup` 和 `LookupId` 则分别根据用户名和用户 ID 查询用户。如果对应的用户不存在，则返回 `user.UnknownUserError` 或 `user.UnknownUserIdError`。

```
package main

import (
    "fmt"
    "os/user"
)

func main() {
    fmt.Println(user.Current())
    fmt.Println(user.Lookup("xuxinhua"))
    fmt.Println(user.LookupId("0"))
}

// Output:
// &{502 502 xuxinhua  /home/xuxinhua} <nil>
// &{502 502 xuxinhua  /home/xuxinhua} <nil>
// &{0 0 root root /root} <nil>
```

进程的当前工作目录

一个进程的当前工作目录（current working directory）定义了该进程解析相对路径名的起点。新进程的当前工作目录继承自其父进程。

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

`Getwd` 返回一个对应当前工作目录的根路径。如果当前目录可以经过多条路径抵达（比如符号链接），`Getwd` 会返回其中一个。对应系统调用：`getcwd`。

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

相应的，`Chdir` 将当前工作目录修改为 `dir` 指定的目录。如果出错，会返回 `*PathError` 类型的错误。对应系统调用 `chdir`。

另外，`os.File` 有一个方法：`Chdir`，对应系统调用 `fchidr`（以文件描述符为参数），也可以改变当前工作目录。

改变进程的根目录

每个进程都有一个根目录，该目录是解释绝对路径（即那些以/开始的目录）时的起点。默认情况下，这是文件系统的真是根目录。新进程从其父进程处继承根目录。有时可能需要改变一个进程的根目录（比如 ftp 服务就是一个典型的例子）。系统调用 `chroot` 能改变一个进程的根目录，Go 中对应的封装在 `syscall.Chroot`。

除此之外，在 `fork` 子进程时，可以通过给 `syscall.SysProcAttr` 结构的 `Chroot` 字段指定一个路径，来初始化子进程的根目录。

进程环境列表

每个进程都有与其相关的称之为环境列表（environment list）的字符串数组，或简称环境（environment）。其中每个字符串都以 名称=值（name=value）形式定义。因此，环境是“名称-值”的成对集合，可存储任何信息。常将列表中的名称称为环境变量（environment variables）。

新进程在创建之时，会继承其父进程的环境副本。这是一种原始的进程间通信方式，却颇为常用。环境（environment）提供了将信息和父进程传递给子进程的方法。创建后，父子进程的环境相互独立，互不影响。

环境变量的常见用途之一是在 shell 中，通过在自身环境中放置变量值，shell 就可确保把这些值传递给其所创建的进程，并以此来执行用户命令。

在程序中，可以通过 `os.Environ` 获取环境列表：

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

返回的 `[]string` 中每个元素是 `key=value` 的形式。

`func Getenv(key string) string`

`Getenv` 检索并返回名为 `key` 的环境变量的值。如果不存在该环境变量会返回空字符串。有时候，可能环境变量存在，只是值刚好是空。为了区分这种情况，提供了另外一个函数 `LookupEnv()`：

`func LookupEnv(key string) (string, bool)`

如果变量名存在，第二个参数返回 `true`，否则返回 `false`。

`func Setenv(key, value string) error`

`Setenv` 设置名为 `key` 的环境变量，值为 `value`。如果出错会返回该错误。（如果值之前存在，会覆盖）

`func Unsetenv(key string) error`

`Unsetenv` 删除名为 `key` 的环境变量。

`func Clearenv()`

`Clearenv` 删除所有环境变量。

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("The num of environ:", len(os.Environ()))
    godebug, ok := os.LookupEnv("GODEBUG")
    if ok {
        fmt.Println("GODEBUG==", godebug)
    } else {
        fmt.Println("GODEBUG not exists!")
        os.Setenv("GODEBUG", "gctrace=1")
        fmt.Println("after setenv:", os.Getenv("GODEBUG"))
    }

    os.Clearenv()
    fmt.Println("clearenv, the num:", len(os.Environ()))
}

// Output:
// The num of environ: 25
// GODEBUG not exists!
// after setenv: gctrace=1
// clearenv, the num: 0
```

另外，`ExpandEnv` 和 `Getenv` 功能类似，不过，前者使用变量方式，如：

os.ExpandEnv("$GODEBUG") 和 os.Getenv("GODEBUG") 是一样的。

实际上，`os.ExpandEnv` 调用的是 `os.Expand(s, os.Getenv)`。

```
func Expand(s string, mapping func(string) string) string
```

`Expand` 能够将 ${var} 或 $var 形式的变量，经过 mapping 处理，得到结果。

导航

- 上一节：创建进程
- 下一节：进程间通信

10.3 线程

与进程类似，线程是允许应用程序并发执行多个任务的一种机制。一个进程可以包含多个线程。同一个程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域。

同一进程中的多个线程可以并发执行。在多处理器环境下，多个线程可以同时并行。如果一个线程因等待 I/O 操作而遭阻塞，那么其他线程依然可以继续运行。

在 Linux 中，通过系统调用 `clone()` 来实现线程的。从前面的介绍，我们知道，该系统调用也可以用来创建进程。实际上，从内核的角度来说，它并没有线程这个概念。Linux 把所有的线程都当作进程来实现。内核并没有准备特别的调度算法或是定义特别的数据结构来表征线程。相反，线程仅仅被视为一个使用某些共享资源的进程。所以，在内核中，它看起来就是一个普通的进程（只是该进程和其他一些进程共享某些资源，如地址空间）。

在 Go 中，通过 `clone()` 系统调用来创建线程，其中的 `clone_flags` 为：

```
cloneFlags = _CLONE_VM | /* share memory */
        _CLONE_FS | /* share cwd, etc */
        _CLONE_FILES | /* share fd table */
        _CLONE_SIGHAND | /* share sig handler table */
        _CLONE_THREAD /* revisit - okay for now */
```

也就是说，父子俩共享了地址空间(_CLONE_VM)、文件系统资源(_CLONE_FS)、文件描述符(_CLONE_FILES)和信号处理程序(_CLONE_SIGHAND)。而 `_CLONE_THREAD` 则会将父子进程放入相同的线程组。这样一来，新建的进程和父进程都叫做线程。

导航

- 上一节：进程属性和控制
- 下一节：进程间通信

10.3 进程间通信

进程之间用来相互通讯和同步

导航

- 上一节：创建进程
- 下一节：进程间通信

15.2 — 非类型安全操作

*unsafe*库徘徊在“类型安全”边缘，由于它们绕过了Golang的内存安全原则，一般被认为使用该库是不安全的。但是，在许多情况下，*unsafe*库的作用又是不可替代的，灵活地使用它们可以实现对内存的直接读写操作。在*reflect*库、*syscall*库以及其他许多需要操作内存的开源项目中都有对它的引用。

*unsafe*库源码极少，只有两个类型的定义和三个方法的声明。

Arbitrary 类型

官方导出这个类型只是出于完善文档的考虑，在其他的库和任何项目中都没有使用价值，除非程序员故意使用它。

Pointer 类型

这个类型比较重要，它是实现定位欲读写的内存的基础。官方文档对该类型有四个重要描述：

- （1）任何类型的指针都可以被转化为Pointer
- （2）Pointer可以被转化为任何类型的指针
- （3）uintptr可以被转化为Pointer
- （4）Pointer可以被转化为uintptr

举例来说，该类型可以这样使用：

```
func main() {
    i := 100
    fmt.Println(i)  // 100
    p := (*int)unsafe.Pointer(&i)
    fmt.Println(*p) // 100
    *p = 0
    fmt.Println(i)  // 0
    fmt.Println(*p) // 0
}
```

Sizeof 函数

该函数的定义如下：

```
func Sizeof(v ArbitraryType) uintptr
```

Sizeof函数返回变量v占用的内存空间的字节数，该字节数不是按照变量v实际占用的内存计算，而是按照v的“top level”内存计算。比如，在64位系统中，如果变量v是int类型，会返回16，因为v的“top level”内存就是它的值使用的内存；如果变量v是string类型，会返回16，因为v的“top level”内存不是存放着实际的字符串，而是该字符串的地址；如果变量v是slice类型，会返回24，这是因为slice的描述符就占了24个字节。

Offsetof 函数

该函数的定义如下：

```
func Offsetof(v ArbitraryType) uintptr
```

该函数返回由v所指示的某结构体中的字段在该结构体中的位置偏移字节数，注意，v的表达方式必须是“struct.filed”形式。 举例说明，在64为系统中运行以下代码：

```
type Datas struct{
    c0 byte
    c1 int
    c2 string
    c3 int
}
func main(){
    var d Datas
    fmt.Println(unsafe.Offset(d.c0))    // 0
    fmt.Println(unsafe.Offset(d.c1))    // 8
    fmt.Println(unsafe.Offset(d.c2))    // 16
    fmt.Println(unsafe.Offset(d.c3))    // 32
}
```

如果知道的结构体的起始地址和字段的偏移值，就可以直接读写内存：

```
d.c3 = 13
p := unsafe.Pointer(&d)
offset := unsafe.Offsetof(d.c3)
q := (*int)(unsafe.Pointer(uintptr(p) + offset))
fmt.Println(*q) // 13
*p = 1013
fmt.Println(d.c3)   // 1013
```

导航

- 目录
- 上一节：buildin
- 下一节：暂未确定

sync - 处理同步需求

golang是一门语言级别支持并发的程序语言。golang中使用go语句来开启一个新的协程。goroutine是非常轻量的，除了给它分配栈空间，它所占用的内存空间是微乎其微的。

但当多个goroutine同时进行处理的时候，就会遇到比如同时抢占一个资源，某个goroutine等待另一个goroutine处理完某一个步骤之后才能继续的需求。 在golang的官方文档上，作者明确指出，golang并不希望依靠共享内存的方式进行进程的协同操作。而是希望通过管道channel的方式进行。 当然，golang也提供了共享内存，锁，等机制进行协同操作的包。sync包就是为了这个目的而出现的。

锁

sync包中定义了Locker结构来代表锁。

```
type Locker interface {
    Lock()
    Unlock()
}
```

并且创造了两个结构来实现Locker接口：Mutex 和 RWMutex。

Mutex就是互斥锁，互斥锁代表着当数据被加锁了之后，除了加锁的程序，其他程序不能对数据进行读操作和写操作。 这个当然能解决并发程序对资源的操作。但是，效率上是个问题。当加锁后，其他程序要读取操作数据，就只能进行等待了。 这个时候就需要使用读写锁。

读写锁分为读锁和写锁，读数据的时候上读锁，写数据的时候上写锁。有写锁的时候，数据不可读不可写。有读锁的时候，数据可读，不可写。 互斥锁就不举例子，读写锁可以看下面的例子：

```
package main

import (
    "sync"
    "time"
)

var m *sync.RWMutex
var val = 0

func main() {
    m = new(sync.RWMutex)
    go read(1)
    go write(2)
    go read(3)
    time.Sleep(5 * time.Second)
}

func read(i int) {
    m.RLock()
    time.Sleep(1 * time.Second)
    println("val: ", val)
    time.Sleep(1 * time.Second)
    m.RUnlock()
}

func write(i int) {
    m.Lock()
    val = 10
    time.Sleep(1 * time.Second)
    m.Unlock()
}

返回：
val:  0
val:  10
```

但是如果我们把read中的RLock和RUnlock两个函数给注释了，就返回了:

```
val:  10
val:  10
```

这个就是由于读的时候没有加读锁，在准备读取val的时候，val被write函数进行修改了。

临时对象池

当多个goroutine都需要创建同一个对象的时候，如果goroutine过多，可能导致对象的创建数目剧增。而对象又是占用内存的，进而导致的就是内存回收的GC压力徒增。造成“并发大－占用内存大－GC缓慢－处理并发能力降低－并发更大”这样的恶性循环。在这个时候，我们非常迫切需要有一个对象池，每个goroutine不再自己单独创建对象，而是从对象池中获取出一个对象（如果池中已经有的话）。这就是sync.Pool出现的目的了。

sync.Pool的使用非常简单，提供两个方法:Get和Put 和一个初始化回调函数New。

看下面这个例子（取自gomemcache）：

```
// keyBufPool returns []byte buffers for use by PickServer's call to
// crc32.ChecksumIEEE to avoid allocations. (but doesn't avoid the
// copies, which at least are bounded in size and small)
var keyBufPool = sync.Pool{
    New: func() interface{} {
        b := make([]byte, 256)
        return &b
    },
}

func (ss *ServerList) PickServer(key string) (net.Addr, error) {
    ss.mu.RLock()
    defer ss.mu.RUnlock()
    if len(ss.addrs) == 0 {
        return nil, ErrNoServers
    }
    if len(ss.addrs) == 1 {
        return ss.addrs[0], nil
    }
    bufp := keyBufPool.Get().(*[]byte)
    n := copy(*bufp, key)
    cs := crc32.ChecksumIEEE((*bufp)[:n])
    keyBufPool.Put(bufp)

    return ss.addrs[cs%uint32(len(ss.addrs))], nil
}
```

这是实际项目中的一个例子，这里使用keyBufPool的目的是为了让crc32.ChecksumIEEE所使用的[]bytes数组可以重复使用，减少GC的压力。

但是这里可能会有一个问题，我们没有看到Pool的手动回收函数。那么是不是就意味着，如果我们的并发量不断增加，这个Pool的体积会不断变大，或者一直维持在很大的范围内呢？

答案是不会的，sync.Pool的回收是有的，它是在系统自动GC的时候，触发pool.go中的poolCleanup函数。

```
func poolCleanup() {
    for i, p := range allPools {
        allPools[i] = nil
        for i := 0; i < int(p.localSize); i++ {
            l := indexLocal(p.local, i)
            l.private = nil
            for j := range l.shared {
                l.shared[j] = nil
            }
            l.shared = nil
        }
        p.local = nil
        p.localSize = 0
    }
    allPools = []*Pool{}
}
```

这个函数会把Pool中所有goroutine创建的对象都进行销毁。

那这里另外一个问题也凸显出来了，很可能我上一步刚往pool中PUT一个对象之后，下一步GC触发，导致pool的GET函数获取不到PUT进去的对象。 这个时候，GET函数就会调用New函数，临时创建出一个对象，并存放到pool中。

根据以上结论，sync.Pool其实不适合用来做持久保存的对象池（比如连接池）。它更适合用来做临时对象池，目的是为了降低GC的压力。

连接池性能测试

```
package main

import (
    "sync"
    "testing"
)

var bytePool = sync.Pool{
    New: newPool,
}

func newPool() interface{} {
    b := make([]byte, 1024)
    return &b
}
func BenchmarkAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        obj := make([]byte, 1024)
        _ = obj
    }
}

func BenchmarkPool(b *testing.B) {
    for i := 0; i < b.N; i++ {
        obj := bytePool.Get().(*[]byte)
        _ = obj
        bytePool.Put(obj)
    }
}
```

文件目录下执行 `go test -bench .`

```
E:\MyGo\sync>go test -bench .
testing: warning: no tests to run
PASS
BenchmarkAlloc-4        50000000                39.3 ns/op
BenchmarkPool-4         50000000                25.4 ns/op
ok      _/E_/MyGo/sync  3.345s
```

通过性能测试可以清楚地看到，使用连接池消耗的CPU时间远远小于每次手动分配内存。

Once

有的时候，我们多个goroutine都要过一个操作，但是这个操作我只希望被执行一次，这个时候Once就上场了。比如下面的例子:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var once sync.Once
    onceBody := func() {
        fmt.Println("Only once")
    }
    for i := 0; i < 10; i++ {
        go func() {
            once.Do(onceBody)
        }()
    }
    time.Sleep(3e9)
}
```

只会打出一次"Only once"。

WaitGroup 和 Cond

一个goroutine需要等待一批goroutine执行完毕以后才继续执行，那么这种多线程等待的问题就可以使用WaitGroup了。

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    wp := new(sync.WaitGroup)
    wp.Add(10);

    for i := 0; i < 10; i++ {
        go func() {
            fmt.Println("done ", i)
            wp.Done()
        }()
    }

    wp.Wait()
    fmt.Println("wait end")
}
```

还有个sync.Cond是用来控制某个条件下，goroutine进入等待时期，等待信号到来，然后重新启动。比如：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    locker := new(sync.Mutex)
    cond := sync.NewCond(locker)
    done := false

    cond.L.Lock()

    go func() {
        time.Sleep(2e9)
        done = true
        cond.Signal()
    }()

    if (!done) {
        cond.Wait()
    }

    fmt.Println("now done is ", done);
}
```

这里当主goroutine进入cond.Wait的时候，就会进入等待，当从goroutine发出信号之后，主goroutine才会继续往下面走。

sync.Cond还有一个BroadCast方法，用来通知唤醒所有等待的gouroutine。

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var locker = new(sync.Mutex)
var cond = sync.NewCond(locker)

func test(x int) {

    cond.L.Lock() // 获取锁
    cond.Wait()   // 等待通知  暂时阻塞
    fmt.Println(x)
    time.Sleep(time.Second * 1)
    cond.L.Unlock() // 释放锁，不释放的话将只会有一次输出
}
func main() {
    for i := 0; i < 40; i++ {
        go test(i)
    }
    fmt.Println("start all")
    cond.Broadcast() //  下发广播给所有等待的goroutine
    time.Sleep(time.Second * 60)
}
```

主gouroutine开启后，可以创建多个从gouroutine，从gouroutine获取锁后，进入cond.Wait状态，当主gouroutine执行完任务后，通过BroadCast广播信号。 处于cond.Wait状态的所有gouroutine收到信号后将全部被唤醒并往下执行。需要注意的是，从gouroutine执行完任务后，需要通过cond.L.Unlock释放锁， 否则其它被唤醒的gouroutine将没法继续执行。 通过查看cond.Wait 的源码就明白为什么需要需要释放锁了

```
func (c *Cond) Wait() {
    c.checker.check()
    if raceenabled {
        raceDisable()
    }
    atomic.AddUint32(&c.waiters, 1)
    if raceenabled {
        raceEnable()
    }
    c.L.Unlock()
    runtime_Syncsemacquire(&c.sema)
    c.L.Lock()
}
```

Cond.Wait会自动释放锁等待信号的到来，当信号到来后，第一个获取到信号的Wait将继续往下执行并从新上锁，如果不释放锁， 其它收到信号的gouroutine将阻塞无法继续执行。 由于各个Wait收到信号的时间是不确定的，因此每次的输出顺序也都是随机的。

导航

- 目录
- 上一节：buildin
- 下一节：暂未确定

sync/atomic - 原子操作

对于并发操作而言，原子操作是个非常现实的问题。典型的就是i++的问题。 当两个CPU同时对内存中的i进行读取，然后把加一之后的值放入内存中，可能两次i++的结果，这个i只增加了一次。 如何保证多CPU对同一块内存的操作是原子的。 golang中sync/atomic就是做这个使用的。

具体的原子操作在不同的操作系统中实现是不同的。比如在Intel的CPU架构机器上，主要是使用总线锁的方式实现的。 大致的意思就是当一个CPU需要操作一个内存块的时候，向总线发送一个LOCK信号，所有CPU收到这个信号后就不对这个内存块进行操作了。 等待操作的CPU执行完操作后，发送UNLOCK信号，才结束。 在AMD的CPU架构机器上就是使用MESI一致性协议的方式来保证原子操作。 所以我们在看atomic源码的时候，我们看到它针对不同的操作系统有不同汇编语言文件。

如果我们善用原子操作，它会比锁更为高效。

CAS

原子操作中最经典的CAS(compare-and-swap)在atomic包中是Compare开头的函数。

- func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
- func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
- func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
- func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
- func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
- func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)

CAS的意思是判断内存中的某个值是否等于old值，如果是的话，则赋new值给这块内存。 CAS是一个方法，并不局限在CPU原子操作中。 CAS比互斥锁乐观，但是也就代表CAS是有赋值不成功的时候，调用CAS的那一方就需要处理赋值不成功的后续行为了。

这一系列的函数需要比较后再进行交换，也有不需要进行比较就进行交换的原子操作。

- func SwapInt32(addr *int32, new int32) (old int32)
- func SwapInt64(addr *int64, new int64) (old int64)
- func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
- func SwapUint32(addr *uint32, new uint32) (old uint32)
- func SwapUint64(addr *uint64, new uint64) (old uint64)
- func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)

增加或减少

对一个数值进行增加或者减少的行为也需要保证是原子的，它对应于atomic包的函数就是

- func AddInt32(addr *int32, delta int32) (new int32)
- func AddInt64(addr *int64, delta int64) (new int64)
- func AddUint32(addr *uint32, delta uint32) (new uint32)
- func AddUint64(addr *uint64, delta uint64) (new uint64)
- func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)

读取或写入

当我们要读取一个变量的时候，很有可能这个变量正在被写入，这个时候，我们就很有可能读取到写到一半的数据。 所以读取操作是需要一个原子行为的。在atomic包中就是Load开头的函数群。

- func LoadInt32(addr *int32) (val int32)
- func LoadInt64(addr *int64) (val int64)
- func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
- func LoadUint32(addr *uint32) (val uint32)
- func LoadUint64(addr *uint64) (val uint64)
- func LoadUintptr(addr *uintptr) (val uintptr)

好了，读取我们是完成了原子性，那写入呢？也是同样的，如果有多个CPU往内存中一个数据块写入数据的时候，可能导致这个写入的数据不完整。 在atomic包对应的是Store开头的函数群。

- func StoreInt32(addr *int32, val int32)
- func StoreInt64(addr *int64, val int64)
- func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
- func StoreUint32(addr *uint32, val uint32)
- func StoreUint64(addr *uint64, val uint64)
- func StoreUintptr(addr *uintptr, val uintptr)

导航

- 目录
- 上一节：sync - 处理同步需求
- 下一节：os/signal - 信号

os/signal - 信号

基本概念

信号是事件发生时对进程的通知机制。有时也称之为软件中断。信号与硬件中断的相似之处在于打断了程序执行的正常流程，大多数情况下，无法预测信号到达的精确时间。

因为一个具有合适权限的进程可以向另一个进程发送信号，这可以称为进程间的一种同步技术。当然，进程也可以向自身发送信号。然而，发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下。

- 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令（除0，引用无法访问的内存区域）。
- 用户键入了能够产生信号的终端特殊字符。如中断字符（通常是 Control-C）、暂停字符（通常是 Control-Z）。
- 发生了软件事件。如调整了终端窗口大小，定时器到期等。

针对每个信号，都定义了一个唯一的（小）整数，从 1 开始顺序展开。系统会用相应常量表示。Linux 中，1-31 为标准信号；32-64 为实时信号（通过 `kill -l` 可以查看）。

信号达到后，进程视具体信号执行如下默认操作之一。

- 忽略信号，也就是内核将信号丢弃，信号对进程不产生任何影响。
- 终止（杀死）进程。
- 产生 coredump 文件，同时进程终止。
- 暂停（Stop）进程的执行。
- 恢复进程执行。

当然，对于有些信号，程序是可以改变默认行为的，这也就是 `os/signal` 包的用途。

兼容性问题：信号的概念来自于 Unix-like 系统。Windows 下只支持 os.SIGINT 信号。

Go 对信号的处理

程序无法捕获信号 SIGKILL 和 SIGSTOP （终止和暂停进程），因此 `os/signal` 包对这两个信号无效。

Go 程序对信号的默认行为

Go 语言实现了自己的运行时，因此，对信号的默认处理方式和普通的 C 程序不太一样。

- SIGBUS（总线错误）, SIGFPE（算术错误）和 SIGSEGV（段错误）称为同步信号，它们在程序执行错误时触发，而不是通过 `os.Process.Kill` 之类的触发。通常，Go 程序会将这类信号转为 run-time panic。
- SIGHUP（挂起）, SIGINT（中断）或 SIGTERM（终止）默认会使得程序退出。
- SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGSTKFLT, SIGEMT 或 SIGSYS 默认会使得程序退出，同时生成 stack dump。
- SIGTSTP, SIGTTIN 或 SIGTTOU，这是 shell 使用的，作业控制的信号，执行系统默认的行为。
- SIGPROF（性能分析定时器，记录 CPU 时间，包括用户态和内核态）， Go 运行时使用该信号实现 `runtime.CPUProfile` 。
- 其他信号，Go 捕获了，但没有做任何处理。

信号可以被忽略或通过掩码阻塞（屏蔽字 mask）。忽略信号通过 signal.Ignore，没有导出 API 可以直接修改阻塞掩码，虽然 Go 内部有实现 sigprocmask 等。Go 中的信号被 runtime 控制，在使用时和 C 是不太一样的。

改变信号的默认行为

这就是 `os/signal` 包的功能。

`Notify` 改变信号处理，可以改变信号的默认行为； `Ignore` 可以忽略信号； `Reset` 重置信号为默认行为； `Stop` 则停止接收信号，但并没有重置为默认行为。

SIGPIPE

文档中对这个信号单独进行了说明。如果 Go 程序往一个 broken pipe 写数据，内核会产生一个 SIGPIPE 信号。

如果 Go 程序没有为 SIGPIPE 信号调用 Notify，对于标准输出或标准错误（文件描述符1或2），该信号会使得程序退出；但其他文件描述符对该信号是啥也不做，当然 write 会返回错误 EPIPE。

如果 Go 程序为 SIGPIPE 调用了 Notify，不论什么文件描述符，SIGPIPE 信号都会传递给 Notify channel，当然 write 依然会返回 EPIPE。

也就是说，默认情况下，Go 的命令行程序跟传统的 Unix 命令行程序行为一致；但当往一个关闭的网络连接写数据时，传统 Unix 程序会 crash，但 Go 程序不会。

cgo 注意事项

如果非 Go 代码使用信号相关功能，需要仔细阅读掌握 `os/signal` 包中相关文档：Go programs that use cgo or SWIG 和 Non-Go programs that call Go code

signal 中 API 详解

Ignore 函数

```
func Ignore(sig ...os.Signal)
```

忽略一个、多个或全部（不提供任何信号）信号。如果程序接收到了被忽略的信号，则什么也不做。对一个信号，如果先调用 `Notify` ，再调用 `Ignore` ， `Notify` 的效果会被取消；如果先调用 `Ignore` ，在调用 `Notify` ，接着调用 `Reset/Stop` 的话，会回到 Ingore 的效果。注意，如果 Notify 作用于多个 chan，则 Stop 需要对每个 chan 都调用才能起到该作用。

Notify 函数

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

类似于绑定信号处理程序。将输入信号转发到 chan c。如果没有列出要传递的信号，会将所有输入信号传递到c；否则只传递列出的输入信号。

channel c 缓存如何决定？因为 `signal` 包不会为了向c发送信息而阻塞（就是说如果发送时 c 阻塞了，signal包会直接放弃）：调用者应该保证 c 有足够的缓存空间可以跟上期望的信号频率。对使用单一信号用于通知的channel，缓存为1就足够了。

相关源码：

```
// src/os/signal/signal.go process 函数
for c, h := range handlers.m {
    if h.want(n) {
        // send but do not block for it
        select {
        case c <- sig:
        default:    // 保证不会阻塞，直接丢弃
        }
    }
}
```

可以使用同一 channel 多次调用 `Notify` ：每一次都会扩展该 channel 接收的信号集。唯一从信号集去除信号的方法是调用 `Stop` 。可以使用同一信号和不同 channel 多次调用 `Notify` ：每一个 channel 都会独立接收到该信号的一个拷贝。

Stop 函数

```
func Stop(c chan<- os.Signal)
```

让 signal 包停止向 c 转发信号。它会取消之前使用 c 调用的所有 `Notify` 的效果。当 `Stop` 返回后，会保证 c 不再接收到任何信号。

Reset 函数

```
func Reset(sig ...os.Signal)
```

取消之前使用 `Notify` 对信号产生的效果；如果没有参数，则所有信号处理都被重置。

使用示例

注：syscall 包中定义了所有的信号常量

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
)

var firstSigusr1 = true

func main() {
    // 忽略 Control-C (SIGINT)
    // os.Interrupt 和 syscall.SIGINT 是同义词
    signal.Ignore(os.Interrupt)

    c1 := make(chan os.Signal, 2)
    // Notify SIGHUP
    signal.Notify(c1, syscall.SIGHUP)
    // Notify SIGUSR1
    signal.Notify(c1, syscall.SIGUSR1)
    go func() {
        for {
            switch <-c1 {
            case syscall.SIGHUP:
                fmt.Println("sighup, reset sighup")
                signal.Reset(syscall.SIGHUP)
            case syscall.SIGUSR1:
                if firstSigusr1 {
                    fmt.Println("first usr1, notify interrupt which had ignore!")
                    c2 := make(chan os.Signal, 1)
                    // Notify Interrupt
                    signal.Notify(c2, os.Interrupt)
                    go handlerInterrupt(c2)
                }
            }
        }
    }()

    select {}
}

func handlerInterrupt(c <-chan os.Signal) {
    for {
        switch <-c {
        case os.Interrupt:
            fmt.Println("signal interrupt")
        }
    }
}
```

编译后运行，先后给该进程发送如下信号：SIGINT、SIGUSR1、SIGINT、SIGHUP、SIGHUP，看输出是不是和你预期的一样。

关于信号的额外说明

1. 查看 Go 中 Linux/amd64 信号的实现，发现大量使用的是 rt 相关系统调用，这是支持实时信号处理的 API。
2. C 语言中信号处理涉及到可重入函数和异步信号安全函数问题；Go 中不存在此问题。
3. Unix 和信号处理相关的很多系统调用，Go 都隐藏起来了，Go 中对信号的处理，`signal` 包中的函数基本就能搞定。

导航

- 目录
- 上一节：sync/atomic - 原子操作
- 下一节：暂未确定