

目 录

介绍

输入输出

`io` — 基本的 IO 接口

`ioutil` — 方便的 IO 操作函数集

`fmt` — 格式化 IO

`bufio` — 缓存 IO

文本

`strings` — 字符串操作

`bytes` — `byte slice` 便利操作

`strconv` — 字符串和基本数据类型之间转换

`regexp` — 正则表达式

`unicode` — Unicode 码点、UTF-8/16 编码

数据结构与算法

`sort` — 排序算法

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

日期与时间

主要类型概述

时区

`Time` 类型详解

定时器

数学计算

`math` — 基本数学函数

文件系统

`os` — 平台无关的操作系统功能实现

`path/filepath` — 操作路径

`io/fs` — 抽象文件系统

数据持久存储与交换

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

数据压缩与归档

`flate` - `DEFLATE` 压缩算法

测试

- testing - 单元测试
- testing - 基准测试
- testing - 子测试与子基准测试
- testing - 运行并验证示例
- testing - 其他功能
- httptest - HTTP 测试辅助工具
- 总结

进程、线程与 goroutine

- 创建进程
- 进程属性和控制
- 线程
- 进程间通信

应用构建 与 debug

- flag - 命令行参数解析
- log - 日志记录
- expvar - 公共变量的标准化接口
- runtime/debug - 运行时的调试工具

底层库介绍

- unsafe — 非类型安全操作

同步

- sync - 处理同步需求
- sync/atomic - 原子操作
- os/signal - 信号

介绍

Golang标准库文档

《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`。更多信息参考 [Go语言中文网博客中关于系统调用的文章](#)。

交流

欢迎大家加入QQ群：192706294 《Go语言实现与标准库》交流群

Go语言构建的 Go语言中文网：<http://studygolang.com>

关注作者公众号，加微信好友、进微信交流群。

阅读

为了方便阅读，Go语言中文网搭建了阅读平台，可以更友好的在线阅读。

[Go语言中文网——Go语言标准库](#)

捐赠

如果您觉得本书对您有帮助，通过微信或支付宝捐赠作者，金额随意！

由于无法从支付方获取支付者信息，请在支付的留言备注功能中附上 **Go语言中文网** 账户的昵称等信息，以便我们记录！

目录

[目录](#)

贡献者

[hikerell](#)

反馈

由于本人能力有限，书中难免有写的不对之处，且目前所写内容没有经过校正。如果阅读过程中有任何疑问或觉得不对之处，欢迎提出，谢谢！

版权声明

本书所有内容遵循 [CC-BY-SA 3.0协议](#)（署名-相同方式共享）

介绍

1. 常见误解
2. 常用手法
3. 如何理解，使用
4. 为什么接口如此组织
5. 和其它语言对比优缺点

输入输出

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。

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。

由于这些被接口包装的 I/O 原语是由不同的低级操作实现，因此，在另有声明之前不该假定它们的并行执行是安全的。

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，只要满足这两个接口，它就可以使用 IO 包的功能。

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` ($0 \leq n \leq \text{len}(p)$) 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占用 `len(p)` 个字节作为暂存空间。若可读取的数据不到 `len(p)` 个字节，`Read` 会返回可用数据，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)，它会返回读取的字节数。它可能会同时在本次的调用中返回一个 non-nil 错误，或在下一次的调用中返回这个错误（且 `n` 为 0）。一般情况下，`Reader` 会返回一个非 0 字节数 `n`，若 `n = len(p)` 个字节从输入源的结尾处由 `Read` 返回，`Read` 可能返回 `err == EOF` 或者 `err == nil`。并且之后的 `Read()` 都应该返回 `(n:0, err:EOF)`。

调用者在考虑错误之前应当首先处理返回的数据。这样做可以正确地处理在读取一些字节后产生的 I/O 错误，同时允许 EOF 的出现。

根据 Go 语言中关于接口和满足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` 方法的类型都满足 `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，当次返回的 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` ($0 \leq n \leq \text{len}(p)$) 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类型的标识（即都是 `os.File` 的实例），自然也实现了 `io.Reader` 和 `io.Writer`。

目前，Go 文档中还没有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`
- `net/conn` 分别实现了 `io.Reader` 和 `io.Writer`(`Conn`接口定义了`Read/Write`)

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` ($0 \leq n \leq \text{len}(p)$) 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。若可读取的数据不到 `len(p)` 字节，`ReadAt` 就会阻塞，直到所有数据都可用或一个错误发生。在这一点上 `ReadAt` 不同于 `Read`。

若 `n = len(p)` 个字节从输入源的结尾处由 `ReadAt` 返回，`Read` 可能返回 `err == EOF` 或者 `err == nil`

若 `ReadAt` 携带一个偏移量从输入源读取，`ReadAt` 应当既不影响偏移量也不被它所影响。

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语言中文网")
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` ($0 \leq n \leq \text{len}(p)$) 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` 携带一个偏移量写入到目标中，`WriteAt` 应当既不影响偏移量也不被它所影响。

若被写区域没有重叠，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到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")
n, err := file.WriteAt([]byte("Go语言中文网"), 24)
if err != nil {
    panic(err)
}
fmt.Println(n)
```

打开文件 `WriteAt.txt`，内容是：`Golang中文社区——Go语言中文网`。

分析：

`file.WriteString("Golang中文社区——这里是多余")` 往文件中写入 `Golang中文社区——这里是多余`，之后 `file.WriteAt([]byte("Go语言中文网"), 24)` 在文件流的 `offset=24` 处写入 `Go语言中文网`（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，有兴趣的可以实现一下。

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语言中文网"))
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** 方法是用于设置偏移量的，这样就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语言中文网")
reader.Seek(-6, io.SeekEnd)
r, _, _ := reader.ReadRune()
fmt.Printf("%c\n", r)
```

小贴士

whence 的值，在 **io** 包中定义了相应的常量，应该使用这些常量

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

而原先 **os** 包中的常量已经被标注为 **Deprecated**

```
// Deprecated: Use io.SeekStart, io.SeekCurrent, and io.SeekEnd.
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。

经过 [issue40](#) 提醒，查看源码：

```
func (f *File) Close() error {
    if f == nil {
        return ErrInvalid
    }
    return f.file.close()
}
```

可见并不会 `panic`，但在 `Close` 之前校验错误是个好习惯！

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。

这里借助 `utf8` 的 `RuneCountInString` 函数,实现代码如下：

```
// strings.Index 的 UTF-8 版本
// 即 Utf8Index("Go语言中文网", "中文") 返回 4，而不是 strings.Index 的 8
func Utf8Index(str, substr string) int {
    index := strings.Index(str, substr)
    if index < 0 {
        return -1
    }
    return utf8.RuneCountInString(str[:index])
}
```

`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，因此不提供示例代码了。

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 结构定义如下：

```

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 LimitedReader Example"
reader := strings.NewReader(content)
limitedReader := &io.LimitedReader{R: reader, N: 8}
for limitedReader.N > 0 {
    tmp := make([]byte, 2)
    limitedReader.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。结构定义如下：

```
type PipeReader struct {
    p *pipe
}
```

关于 **`PipeReader.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被关闭。如果写入端关闭时带有 `error`（即调用 `CloseWithError` 关闭），该 `Read` 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。结构定义如下：

```
type PipeWriter struct {
    p *pipe
}
```

关于 **`PipeWriter.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被关闭。如果读取端关闭时带有 `error`（即调用 `CloseWithError` 关闭），该 `Write` 返回的 `err` 就是读取端传递的 `error`；否则 `err` 为 `ErrClosedPipe`。

使用示例如下：

```
func main() {
    pipeReader, pipeWriter := io.Pipe()
    go PipeWrite(pipeWriter)
    go PipeRead(pipeReader)
    time.Sleep(30 * time.Second)
}

func PipeWrite(writer *io.PipeWriter) {
    data := []byte("Go语言中文网")
    for i := 0; i < 3; i++{
        n, err := writer.Write(data)
        if err != nil{
            fmt.Println(err)
        }
        return
    }
    fmt.Printf("写入字节 %d\n", n)
    writer.CloseWithError(errors.New("写入段已关闭"))
}

func PipeRead(reader *io.PipeReader) {
    buf := make([]byte, 128)
    for{
        fmt.Println("接口端开始阻塞5秒钟...")
        time.Sleep(5 * time.Second)
```



```

    fmt.Println("接收端开始接受")
    n, err := reader.Read(buf)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("收到字节: %d\n buf内容: %s\n", n, buf)
}
}

```

`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。

正因为是同步的，因此不能在一个 `goroutine` 中进行读和写。

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，如果有错误的话，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语言中文网"))
```

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(或到一个error)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。当且仅当 err == nil 时, written == n。

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语言中文网"), 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)
```

函数文档：

`WriteString` 将 `s` 的内容写入 `w` 中，当 `w` 实现了 `WriteString` 方法时，会直接调用该方法，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：`multiReader` 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, 128)
buf := make([]byte, 10)

for n, err := reader.Read(buf); err != io.EOF; n, err = reader.Read(buf) {
    if err != nil {
        panic(err)
    }
    data = append(data, buf[:n]...)
}
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语言中文网"))
```

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语言中文网`。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语言中文网"), os.Stdout)
reader.Read(make([]byte, 20))
```

输出结果:

```
Go语言中文网
```

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。

ioutil — 方便的 IO 操作函数集

`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。

由于这些被接口包装的 I/O 原语是由不同的低级操作实现，因此，在另有声明之前不该假定它们的并行执行是安全的。

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，只要满足这两个接口，它就可以使用 IO 包的功能。

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` ($0 \leq n \leq \text{len}(p)$) 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占用 `len(p)` 个字节作为暂存空间。若可读取的数据不到 `len(p)` 个字节，`Read` 会返回可用数据，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)，它会返回读取的字节数。它可能会同时在本次的调用中返回一个 non-nil 错误，或在下一次的调用中返回这个错误（且 `n` 为 0）。一般情况下，`Reader` 会返回一个非 0 字节数 `n`，若 `n = len(p)` 个字节从输入源的结尾处由 `Read` 返回，`Read` 可能返回 `err == EOF` 或者 `err == nil`。并且之后的 `Read()` 都应该返回 `(n:0, err:EOF)`。

调用者在考虑错误之前应当首先处理返回的数据。这样做可以正确地处理在读取一些字节后产生的 I/O 错误，同时允许 EOF 的出现。

根据 Go 语言中关于接口和满足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` 方法的类型都满足 `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，当次返回的 `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` ($0 \leq n \leq \text{len}(p)$) 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类型的标识（即都是 `os.File` 的实例），自然也实现了 `io.Reader` 和 `io.Writer`。

目前，Go 文档中还没有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`
- `net/conn` 分别实现了 `io.Reader` 和 `io.Writer`(`Conn`接口定义了`Read/Write`)

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` ($0 \leq n \leq \text{len}(p)$) 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。若可读取的数据不到 `len(p)` 字节，`ReadAt` 就会阻塞，直到所有数据都可用或一个错误发生。在这一点上 `ReadAt` 不同于 `Read`。

若 `n = len(p)` 个字节从输入源的结尾处由 `ReadAt` 返回, `Read` 可能返回 `err == EOF` 或者 `err == nil`

若 `ReadAt` 携带一个偏移量从输入源读取, `ReadAt` 应当既不影响偏移量也不被它所影响。

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语言中文网")
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` ($0 \leq n \leq \text{len}(p)$) 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` 携带一个偏移量写入到目标中, `WriteAt` 应当既不影响偏移量也不被它所影响。

若被写区域没有重叠, 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到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")
n, err := file.WriteAt([]byte("Go语言中文网"), 24)
if err != nil {
    panic(err)
}
fmt.Println(n)
```


打开文件 `WriteAt.txt`，内容是：`Golang中文社区——Go语言中文网`。

分析：

`file.WriteString("Golang中文社区——这里是多余")` 往文件中写入 `Golang中文社区——这里是多余`，之后 `file.WriteAt([]byte("Go语言中文网"), 24)` 在文件流的 `offset=24` 处写入 `Go语言中文网`（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，有兴趣的可以实现一下。

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语言中文网"))
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** 方法是用于设置偏移量的，这样就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语言中文网")
reader.Seek(-6, io.SeekEnd)
r, _, _ := reader.ReadRune()
fmt.Printf("%c\n", r)
```

小贴士

whence 的值，在 **io** 包中定义了相应的常量，应该使用这些常量

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

而原先 **os** 包中的常量已经被标注为 **Deprecated**

```
// Deprecated: Use io.SeekStart, io.SeekCurrent, and io.SeekEnd.
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。

经过 [issue40](#) 提醒，查看源码：

```
func (f *File) Close() error {
    if f == nil {
        return ErrInvalid
    }
    return f.file.close()
}
```

可见并不会 `panic`，但在 `Close` 之前校验错误是个好习惯！

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。
```

这里借助utf8的RuneCountInString函数,实现代码如下：

```
// strings.Index 的 UTF-8 版本
// 即 Utf8Index("Go语言中文网", "中文") 返回 4，而不是 strings.Index 的 8
func Utf8Index(str, substr string) int {
    index := strings.Index(str, substr)
    if index < 0 {
        return -1
    }
    return utf8.RuneCountInString(str[:index])
}
```

`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，因此不提供示例代码了。

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 结构定义如下：

```

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 LimitedReader Example"
reader := strings.NewReader(content)
limitedReader := &io.LimitedReader{R: reader, N: 8}
for limitedReader.N > 0 {
    tmp := make([]byte, 2)
    limitedReader.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。结构定义如下：

```
type PipeReader struct {
    p *pipe
}
```

关于 **`PipeReader.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被关闭。如果写入端关闭时带有 `error`（即调用 `CloseWithError` 关闭），该 `Read` 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。结构定义如下：

```
type PipeWriter struct {
    p *pipe
}
```

关于 **`PipeWriter.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被关闭。如果读取端关闭时带有 `error`（即调用 `CloseWithError` 关闭），该 `Write` 返回的 `err` 就是读取端传递的 `error`；否则 `err` 为 `ErrClosedPipe`。

使用示例如下：

```
func main() {
    pipeReader, pipeWriter := io.Pipe()
    go PipeWrite(pipeWriter)
    go PipeRead(pipeReader)
    time.Sleep(30 * time.Second)
}

func PipeWrite(writer *io.PipeWriter) {
    data := []byte("Go语言中文网")
    for i := 0; i < 3; i++{
        n, err := writer.Write(data)
        if err != nil{
            fmt.Println(err)
        }
        return
    }
    fmt.Printf("写入字节 %d\n", n)
    writer.CloseWithError(errors.New("写入段已关闭"))
}

func PipeRead(reader *io.PipeReader) {
    buf := make([]byte, 128)
    for{
        fmt.Println("接口端开始阻塞5秒钟...")
        time.Sleep(5 * time.Second)
    }
}
```

```

    fmt.Println("接收端开始接受")
    n, err := reader.Read(buf)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("收到字节: %d\n buf内容: %s\n", n, buf)
}
}

```

`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。

正因为是同步的，因此不能在一个 `goroutine` 中进行读和写。

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，如果有错误的话，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语言中文网"))
```

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(或到一个error)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。当且仅当err == nil时,written == n。

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语言中文网"), 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)
```

函数文档：

`WriteString` 将 `s` 的内容写入 `w` 中，当 `w` 实现了 `WriteString` 方法时，会直接调用该方法，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：`multiReader` 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, 128)
buf := make([]byte, 10)

for n, err := reader.Read(buf); err != io.EOF; n, err = reader.Read(buf) {
    if err != nil {
        panic(err)
    }
    data = append(data, buf[:n]...)
}
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语言中文网"))
```

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语言中文网`。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语言中文网"), os.Stdout)
reader.Read(make([]byte, 20))
```

输出结果:

```
Go语言中文网
```

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。

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
}

// 定义结构体变量
var site = Website{Name: "studygolang"}
```

Printing

Sample

```
type user struct {
    name string
}

func main() {
    u := user{"tang"}
    //Printf 格式化输出
    fmt.Printf("% + v\n", u) //格式化输出结构
    fmt.Printf("#v\n", u) //输出值的 Go 语言表示方法
    fmt.Printf("T\n", u) //输出值的类型的 Go 语言表示
    fmt.Printf("t\n", true) //输出值的 true 或 false
    fmt.Printf("b\n", 1024) //二进制表示
    fmt.Printf("c\n", 1111111) //数值对应的 Unicode 编码字符
    fmt.Printf("d\n", 10) //十进制表示
    fmt.Printf("o\n", 8) //八进制表示
    fmt.Printf("q\n", 22) //转化为十六进制并附上单引号
    fmt.Printf("x\n", 1223) //十六进制表示, 用a-f表示
    fmt.Printf("X\n", 1223) //十六进制表示, 用A-F表示
    fmt.Printf("U\n", 1233) //Unicode表示
    fmt.Printf("b\n", 12.34) //无小数部分, 两位指数的科学计数法6946802425218990p-49
    fmt.Printf("e\n", 12.345) //科学计数法, e表示
    fmt.Printf("E\n", 12.34455) //科学计数法, E表示
    fmt.Printf("f\n", 12.3456) //有小数部分, 无指数部分
    fmt.Printf("g\n", 12.3456) //根据实际情况采用%e或%f输出
    fmt.Printf("G\n", 12.3456) //根据实际情况采用%E或%F输出
    fmt.Printf("s\n", "wqde") //直接输出字符串或者[]byte
    fmt.Printf("q\n", "dedede") //双引号括起来的字符串
    fmt.Printf("x\n", "abczxc") //每个字节用两字节十六进制表示, a-f表示
    fmt.Printf("X\n", "asdzxc") //每个字节用两字节十六进制表示, A-F表示
    fmt.Printf("p\n", 0x123) //0x开头的十六进制数表示
}
```

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。 在打印结构体时，“加号”标记(%+v)会添加字段名	Printf("%v", site), Printf("%+v", site)	{study golang}, {Name:studygolang}
%#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("%o", 10)	12
%q	单引号围绕的字符串面值，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语言中文网"))
Go语言中文网		
%q	双引号围绕的字符串，由Go语法安全地转义	Printf("%q", "Go语言中文网")
"Go语言中文网"		
%x	十六进制，小写字母，每字节两个字符	Printf("%x", "golang")
676f6c616e67		

占位符	说明	举例
%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
%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
+	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	Printf("%#U", '中')
U+4E2D '中'		
OX (%#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'）。	
' '	（空格）为数值中省略的正负号留出空白 (% d)；	
x, X	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，可以用这组函数。

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 `fmt` 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.Sprintf("%s", p)))
    }
}
```

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。

fmt/print.go 阅读

Fprint

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter() // 实际工作结构
    p.doPrint(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

newPrinter

```
// printer 状态结构
type pp struct {
    buf buffer

    arg interface{}

    value reflect.Value

    fmt fmt

    reordered bool

    goodArgNum bool

    panicking bool

    erroring bool
}

// 通过 sync.Pool 复用, 避免回收造成 GC
var ppFree = sync.Pool{
    New: func() interface{} { return new(pp) },
}

// 分配或重用 pp 结构
func newPrinter() *pp {
    p := ppFree.Get().(*pp)
    p.panicking = false
    p.erroring = false
    p.fmt.init(&p.buf)
    return p
}
```

doPrint

```
func (p *pp) doPrint(a []interface{}) {
    prevString := false

    // 获取可变参数索引及参数
    for argNum, arg := range a {
```

```

// reflect.TypeOf.Kind
isString := arg != nil && reflect.TypeOf(arg).Kind() == reflect.String

// 判断是否需要一个空格
if argNum > 0 && !isString && !prevString {
    p.buf.WriteByte(' ')
}
p.printArg(arg, 'v')
prevString = isString
}
}

```

printArg

```

func (p *pp) printArg(arg interface{}, verb rune) {
    p.arg = arg
    p.value = reflect.Value{}

    if arg == nil {
        switch verb {
            case 'T', 'v':
                p.fmt.padString(nilAngleString)
            default:
                p.badVerb(verb)
        }
        return
    }

    switch verb {
        case 'T':
            p.fmt.fmt_s(reflect.TypeOf(arg).String())
            return
        case 'p':
            p.fmtPointer(reflect.ValueOf(arg), 'p')
            return
    }

    // 类型判断
    switch f := arg.(type) {
        case bool:
            p.fmtBool(f, verb)
        case float32:
            p.fmtFloat(float64(f), 32, verb)
        case float64:
            p.fmtFloat(f, 64, verb)
        case complex64:
            p.fmtComplex(complex128(f), 64, verb)
        case complex128:
            p.fmtComplex(f, 128, verb)
        case int:
            p.fmtInteger(uint64(f), signed, verb)
        case int8:
            p.fmtInteger(uint64(f), signed, verb)
        case int16:
            p.fmtInteger(uint64(f), signed, verb)
        case int32:
            p.fmtInteger(uint64(f), signed, verb)
        case int64:
            p.fmtInteger(uint64(f), signed, verb)
        case uint:
    }
}

```

```

    p.fmtInteger(uint64(f), unsigned, verb)
case uint8:
    p.fmtInteger(uint64(f), unsigned, verb)
case uint16:
    p.fmtInteger(uint64(f), unsigned, verb)
case uint32:
    p.fmtInteger(uint64(f), unsigned, verb)
case uint64:
    p.fmtInteger(f, unsigned, verb)
case uintptr:
    p.fmtInteger(uint64(f), unsigned, verb)
case string:
    p.fmtString(f, verb)
case []byte:
    p.fmtBytes(f, verb, "[]byte")
case reflect.Value:
    // Handle extractable values with special methods
    // since printValue does not handle them at depth 0.
    if f.IsValid() && f.CanInterface() {
        p.arg = f.Interface()
        if p.handleMethods(verb) {
            return
        }
    }
    p.printValue(f, verb, 0)
default:
    // If the type is not simple, it might have methods.
    if !p.handleMethods(verb) {
        // Need to use reflection, since the type had no
        // interface methods that could be used for formatting.
        p.printValue(reflect.ValueOf(f), verb, 0)
    }
}
}
}

```


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。(感觉这一段直接看源码较好)

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，这些字节会失效。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(line携带一个'\n',printf又追加了一个'\n')。

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（“`\n\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，如果你希望留下行尾结束符，只用 `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 the 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 != 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显得有些过于复杂了。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单词等，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：studygol
ang、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 != 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'`、`'\u0085 (NEL)`、`'\u00A0 (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 home 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 studying 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
    }
    return &Writer{
        buf: make([]byte, size),
        wr:  wr,
    }
}
```

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.NewReaderWriter` 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}  
}
```

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。

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，指的是 `rune` 类型，即一个 UTF-8 字符（Unicode 代码点）。

2.1.1 字符串比较

```
// Compare 函数，用于比较两个字符串的大小，如果两个字符串相等，返回为 0。如果 a 小于 b，返回 -1，反之返回 1。不推荐使用这个函数，直接使用 == != > < >= <= 等一系列运算符更加直观。
func Compare(a, b string) int
// EqualFold 函数，计算 s 与 t 忽略字母大小写后是否相等。
func EqualFold(s, t string) bool
```

示例：

```

a := "gopher"
b := "hello world"
fmt.Println(strings.Compare(a, b))
fmt.Println(strings.Compare(a, a))
fmt.Println(strings.Compare(b, a))

fmt.Println(strings.EqualFold("GO", "go"))
fmt.Println(strings.EqualFold("壹", "一"))

```

输出结果:

```

-1
0
1
true
false

```

2.1.2 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 `false`。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。

2.1.3 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，属于字符串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("cheese", "e"))  
fmt.Println(len("谷歌中国"))  
fmt.Println(strings.Count("谷歌中国", ""))
```

输出：

```
3  
12  
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
```

2.1.4 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。

常见间隔符包括：`'\t'`、`'\n'`、`'\v'`、`'\f'`、`'\r'`、`'\u0085'` (NEL)、`'\u00A0'` (NBSP)

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 "))
```

输出结果：

```
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)
}
```

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
}
```

如果 `prefix` 或 `suffix` 为 `""`，返回值总是 `true`。

示例：

```
fmt.Println(strings.HasPrefix("Gopher", "Go"))
fmt.Println(strings.HasPrefix("Gopher", "C"))
fmt.Println(strings.HasPrefix("Gopher", ""))
fmt.Println(strings.HasSuffix("Amigo", "go"))
fmt.Println(strings.HasSuffix("Amigo", "Ami"))
fmt.Println(strings.HasSuffix("Amigo", ""))
```


输出结果:

```
true
false
true
true
false
true
```

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
// 在 s 中查找字节 c 的第一次出现, 返回第一次出现的索引
func IndexByte(s string, c byte) 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 LastIndexByte(s string, c byte) 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:

```
han := func(c rune) bool {
    return unicode.Is(unicode.Han, c) // 汉字
}
fmt.Println(strings.IndexFunc("Hello, world", han))
fmt.Println(strings.IndexFunc("Hello, 世界", han))
```

输出:

```
-1
7
```

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"}, "&"))
```

输出结果：

```
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
```

将 `s` 重复 `count` 次，如果 `count` 为负数或返回值长度 `len(s)*count` 超出 `string` 上限会导致 `panic`，这个函数使用很简单：

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

输出结果：

```
banana
```

2.1.8 字符替换

```
func Map(mapping func(rune) rune, s string) string
```

`Map` 函数，将 `s` 的每一个字符按照 `mapping` 的规则做映射替换，如果 `mapping` 返回值 `<0`，则舍弃该字符。该方法只能对每一个字符做处理，但处理方式很灵活，可以方便的过滤，筛选汉字等。

示例：

```
mapping := func(r rune) rune {
    switch {
    case r >= 'A' && r <= 'Z': // 大写字母转小写
        return r + 32
    case r >= 'a' && r <= 'z': // 小写字母不处理
        return r
    case unicode.Is(unicode.Han, r): // 汉字换行
        return '\n'
    }
    return -1 // 过滤所有非字母、汉字的字符
}
fmt.Println(strings.Map(mapping, "Hello你#¥%……\n (' World\n,好Hello^(&(*界gopher..."))
```

输出结果：

```
hello
world
hello
gopher
```

2.1.9 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
// 该函数内部直接调用了函数 Replace(s, old, new, -1)
func ReplaceAll(s, old, new string) 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))
fmt.Println(strings.ReplaceAll("oink oink oink", "oink", "moo"))
```

输出:

```
oinky oinky oink
moo moo moo
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 HTML` 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。

2.1.10 大小写转换

```
func ToLower(s string) string
func ToLowerSpecial(c unicode.SpecialCase, s string) string
func ToUpper(s string) string
func ToUpperSpecial(c unicode.SpecialCase, s string) string
```

大小写转换包含了 4 个相关函数, `ToLower`, `ToUpper` 用于大小写转换。 `ToLowerSpecial`, `ToUpperSpecial` 可以转换特殊字符的大小写。

举个例子:

```
fmt.Println(strings.ToLower("HELLO WORLD"))
fmt.Println(strings.ToLower("Ä Ä Ä Ä"))
fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "壹"))
fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "HELLO WORLD"))
fmt.Println(strings.ToLower("Önnek İŞ"))
fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "Önnek İŞ"))

fmt.Println(strings.ToUpper("hello world"))
fmt.Println(strings.ToUpper("ä ä ä ä"))
fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "一"))
fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "hello world"))
fmt.Println(strings.ToUpper("örnek iş"))
fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "örnek iş"))
```

输出结果:

```
hello world
ä ä ä ä
壹
hello world
önnek iş
önnek iş
HELLO WORLD
Ä Ä Ä Ä // 汉字拼音有效
一 // 汉字无效
HELLO WORLD
ÖRNEK İŞ
ÖRNEK İŞ // 有细微差别
```

2.1.10 标题处理

```
func Title(s string) string
func ToTitle(s string) string
func ToTitleSpecial(c unicode.SpecialCase, s string) string
```

标题处理包含 3 个相关函数，其中 `Title` 会将 `s` 每个单词的首字母大写，不处理该单词的后续字符。`ToTitle` 将 `s` 的每个字母大写。`ToTitleSpecial` 将 `s` 的每个字母大写，并且会将一些特殊字母转换为其对应的特殊大写字母。

举个例子：

```
fmt.Println(strings.Title("hELLo wOrLd"))
fmt.Println(strings.ToTitle("hELLo wOrLd"))
fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "hELLo wOrLd"))
fmt.Println(strings.Title("ääää öööö êééèè"))
fmt.Println(strings.ToTitle("ääää öööö êééèè"))
fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "ääää öööö êééèè"))
fmt.Println(strings.Title("dünyanın ilk borsa yapısı Aizonai kabul edilir"))
fmt.Println(strings.ToTitle("dünyanın ilk borsa yapısı Aizonai kabul edilir"))
fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "dünyanın ilk borsa yapısı Aizonai kabul edilir"))
```

输出结果：

```
HELLO wOrLd
HELLO WORLD
HELLO WORLD
ÄÄÄÄ ÖÖÖÖ ÊÉÉÈÈ
ÄÄÄÄ ÖÖÖÖ ÊÉÉÈÈ
ÄÄÄÄ ÖÖÖÖ ÊÉÉÈÈ
Dünyanın İlk Borsa Yapısı Aizonai Kabul Edilir
DÜNYANIN ILK BORSA YAPISI AIZONAI KABUL EDILIR
DÜNYANIN İLK BORSA YAPISI AİZONAI KABUL EDİLİR
```

2.1.11 修剪

```
// 将 s 左侧和右侧中匹配 cutset 中的任一字符的字符去掉
func Trim(s string, cutset string) string
// 将 s 左侧的匹配 cutset 中的任一字符的字符去掉
func TrimLeft(s string, cutset string) string
// 将 s 右侧的匹配 cutset 中的任一字符的字符去掉
func TrimRight(s string, cutset string) string
// 如果 s 的前缀为 prefix 则返回去掉前缀后的 string，否则 s 没有变化。
func TrimPrefix(s, prefix string) string
// 如果 s 的后缀为 suffix 则返回去掉后缀后的 string，否则 s 没有变化。
func TrimSuffix(s, suffix string) string
// 将 s 左侧和右侧的间隔符去掉。常见间隔符包括：' |t', '|n', '|v', '|f', '|r', '|', ' ', U+0085 (NEL)
func TrimSpace(s string) string
// 将 s 左侧和右侧的匹配 f 的字符去掉
func TrimFunc(s string, f func(rune) bool) string
// 将 s 左侧的匹配 f 的字符去掉
func TrimLeftFunc(s string, f func(rune) bool) string
// 将 s 右侧的匹配 f 的字符去掉
func TrimRightFunc(s string, f func(rune) bool) string
```

包含了 9 个相关函数用于修剪字符串。

举个例子:

```
x := "!!!@@@你好,!@$ Gophers####$$"
fmt.Println(strings.Trim(x, "@#$!%^&*()_+!="))
fmt.Println(strings.TrimLeft(x, "@#$!%^&*()_+!="))
fmt.Println(strings.TrimRight(x, "@#$!%^&*()_+!="))
fmt.Println(strings.TrimSpace("\t\n Hello, Gophers \n\t\r\n"))
fmt.Println(strings.TrimPrefix(x, "!"))
fmt.Println(strings.TrimSuffix(x, "$"))

f := func(r rune) bool {
    return !unicode.Is(unicode.Han, r) // 非汉字返回 true
}
fmt.Println(strings.TrimFunc(x, f))
fmt.Println(strings.TrimLeftFunc(x, f))
fmt.Println(strings.TrimRightFunc(x, f))
```

输出结果:

```
你好,!@$ Gophers
你好,!@$ Gophers####$$
!!!@@@你好,!@$ Gophers
Hello, Gophers
!!!@@@你好,!@$ Gophers####$$
!!!@@@你好,!@$ Gophers####$$
你好
你好,!@$ Gophers####$$
!!!@@@你好
```

2.1.12 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。如果 `oldnew` 长度与奇数，会导致 `panic`。

示例:

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

输出结果:

```
This is &lt;b&gt;HTML&lt;/b&gt;!
```

另外，`Replacer` 还提供了另外一个方法，它在替换之后将结果写入 `io.Writer` 中。

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

2.1.13 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。

2.1.14 Builder 类型

```
type Builder struct {
    addr *Builder // of receiver, to detect copies by value
    buf []byte
}
```

该类型实现了 io 包下的 Writer, ByteWriter, StringWriter 等接口, 可以向该对象内写入数据, Builder 没有实现 Reader 等接口, 所以该类型不可读, 但提供了 String 方法可以获取对象内的数据。

```
// 该方法向 b 写入一个字节
func (b *Builder) WriteByte(c byte) error
// WriteRune 方法向 b 写入一个字符
func (b *Builder) WriteRune(r rune) (int, error)
// WriteRune 方法向 b 写入字节数组 p
func (b *Builder) Write(p []byte) (int, error)
// WriteRune 方法向 b 写入字符串 s
func (b *Builder) WriteString(s string) (int, error)
// Len 方法返回 b 的数据长度。
func (b *Builder) Len() int
// Cap 方法返回 b 的 cap。
func (b *Builder) Cap() int
// Grow 方法将 b 的 cap 至少增加 n (可能会更多)。如果 n 为负数, 会导致 panic。
func (b *Builder) Grow(n int)
// Reset 方法将 b 清空 b 的所有内容。
func (b *Builder) Reset()
// String 方法将 b 的数据以 string 类型返回。
func (b *Builder) String() string
```

Builder 有 4 个与写入相关的方法, 这 4 个方法的 error 都总是为 nil。

Builder 的 cap 会自动增长, 一般不需要手动调用 Grow 方法。

String 方法可以方便的获取 Builder 的内容。

举个例子:

```
b := strings.Builder{}
_ = b.WriteByte('7')
n, _ := b.WriteRune('夕')
fmt.Println(n)
n, _ = b.Write([]byte("Hello, World"))
fmt.Println(n)
n, _ = b.WriteString("你好, 世界")
fmt.Println(n)
fmt.Println(b.Len())
fmt.Println(b.Cap())
b.Grow(100)
fmt.Println(b.Len())
fmt.Println(b.Cap())
fmt.Println(b.String())
b.Reset()
fmt.Println(b.String())
```

输出结果:

```
3
12
15
31
32
31
164
7夕Hello, World你好, 世界
```


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 Runes 类型转换

```
// 将 []byte 转换为 []rune
func Runes(s []byte) []rune
```

该函数将 []byte 转换为 []rune，适用于汉字等多字节字符，示例：

```
b:=[]byte("你好，世界")
for k,v:=range b{
    fmt.Printf("%d:%s |",k,string(v))
}
r:=bytes.Runes(b)
for k,v:=range r{
    fmt.Printf("%d:%s |",k,string(v))
}
```

运行结果：

```
0:ä |1:½ |2: |3:ã |4:¥ |5:½ |6:ï |7:¼ |8: |9:ä |10:, |11: |12:ç |13: |14: |
0:你|1:好|2:, |3:世|4:界|
```

2.2.4 Reader 类型

```
type Reader struct {
    s []byte
    i int64 // 当前读取下标
    prevRune int // 前一个字符的下标，也可能 < 0
}
```

bytes 包下的 Reader 类型实现了 io 包下的 Reader, ReaderAt, RuneReader, RuneScanner, ByteReader, ByteScanner, ReadSeeker, Seeker, WriterTo 等多个接口。主要用于 Read 数据。

我们需要通过 bytes.NewReader 方法来初始化 bytes.Reader 类型的对象。初始化时传入 []byte 类型的数据。NewReader 函数签名如下：

```
func NewReader(b []byte) *Reader
```

如果直接声明该对象了，可以通过 Reset 方法重新写入数据，示例：

```
x:=[]byte("你好，世界")

r1:=bytes.NewReader(x)
d1:=make([]byte, len(x))
n,_:=r1.Read(d1)
fmt.Println(n,string(d1))

r2:=bytes.Reader{}
r2.Reset(x)
d2:=make([]byte, len(x))
```

```
n, _ = r2.Read(d2)
fmt.Println(n, string(d2))
```

输出结果:

```
15 你好, 世界
15 你好, 世界
```

`Reader` 包含了 8 个读取相关的方法, 实现了前面提到的 `io` 包下的 9 个接口 (`ReaderSeeker` 接口内嵌 `Reader` 和 `Seeker` 两个接口):

```
// 读取数据至 b
func (r *Reader) Read(b []byte) (n int, err error)
// 读取一个字节
func (r *Reader) ReadByte() (byte, error)
// 读取一个字符
func (r *Reader) ReadRune() (ch rune, size int, err error)
// 读取数据至 w
func (r *Reader) WriteTo(w io.Writer) (n int64, err error)
// 进度下标指向前一个字节, 如果 r.i <= 0 返回错误。
func (r *Reader) UnreadByte()
// 进度下标指向前一个字符, 如果 r.i <= 0 返回错误, 且只能在每次 ReadRune 方法后使用一次, 否则返回错误。
func (r *Reader) UnreadRune()
// 读取 r.s[off:] 的数据至b, 该方法忽略进度下标 i, 不使用也不修改。
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
// 根据 whence 的值, 修改并返回进度下标 i, 当 whence == 0, 进度下标修改为 off, 当 whence == 1, 进度下标修改为 i+off, 当 whence == 2, 进度下标修改为 len[s]+off。
// off 可以为负数, whence 的只能为 0, 1, 2, 当 whence 为其他值或计算后的进度下标越界, 则返回错误。
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

示例:

```
x := []byte("你好, 世界")
r1 := bytes.NewReader(x)

ch, size, _ := r1.ReadRune()
fmt.Println(size, string(ch))
_ = r1.UnreadRune()
ch, size, _ = r1.ReadRune()
fmt.Println(size, string(ch))
_ = r1.UnreadRune()

by, _ := r1.ReadByte()
fmt.Println(by)
_ = r1.UnreadByte()
by, _ = r1.ReadByte()
fmt.Println(by)
_ = r1.UnreadByte()

d1 := make([]byte, 6)
n, _ := r1.Read(d1)
fmt.Println(n, string(d1))

d2 := make([]byte, 6)
n, _ = r1.ReadAt(d2, 0)
fmt.Println(n, string(d2))

w1 := &bytes.Buffer{}
```

```
_, _ = r1.Seek(0, 0)
_, _ = r1.WriteTo(w1)
fmt.Println(w1.String())
```

运行结果:

```
3 你
3 你
228
228
6 你好
6 你好
你好, 世界
```

2.2.5 Buffer 类型

```
type Buffer struct {
    buf []byte
    off int
    lastRead readOp
}
```

在上一个示例的最后，我们使用了 `bytes.Buffer` 类型，该类型实现了 `io` 包下的 `ByteScanner`, `ByteWriter`, `ReadWriter`, `Reader`, `ReaderFrom`, `RuneReader`, `RuneScanner`, `StringWriter`, `Writer`, `WriterTo` 等接口，可以方便的进行读写操作。

对象可读取数据为 `buf[off : len(buf)]`, `off` 表示进度下标，`lastRead` 表示最后读取的一个字符所占字节数，方便 `Unread*` 相关操作。

`Buffer` 可以通过 3 中方法初始化对象:

```
a := bytes.NewBufferString("Hello World")
b := bytes.NewBuffer([]byte("Hello World"))
c := bytes.Buffer{}

fmt.Println(a)
fmt.Println(b)
fmt.Println(c)
}
```

输出结果:

```
Hello World
Hello World
{[] 0 0}
```

`Buffer` 包含了 21 个读写相关的方法，大部分同名方法的用法与前面讲的类似，这里只讲演示其中的 3 个方法:

```
// 读取到字节 delim 后，以字节数组的形式返回该字节及前面读取到的字节。如果遍历 b.buf 也找不到匹配的字节，则返回错误(一般是 EOF)
func (b *Buffer) ReadBytes(delim byte) (line []byte, err error)
// 读取到字节 delim 后，以字符串的形式返回该字节及前面读取到的字节。如果遍历 b.buf 也找不到匹配的字节，则返回错误(一般是 EOF)
func (b *Buffer) ReadString(delim byte) (line string, err error)
// 截断 b.buf，舍弃 b.off+n 之后的数据。n == 0 时，调用 Reset 方法重置该对象，当 n 越界时 (n < 0 || n > b.Le
```

```
n() ) 方法会触发 panic.
func (b *Buffer) Truncate(n int)
```

示例:

```
a := bytes.NewBufferString("Good Night")

x, err := a.ReadBytes('t')
if err != nil {
    fmt.Println("delim:t err:", err)
} else {
    fmt.Println(string(x))
}

a.Truncate(0)
a.WriteString("Good Night")
fmt.Println(a.Len())
a.Truncate(5)
fmt.Println(a.Len())
y, err := a.ReadString('N')
if err != nil {
    fmt.Println("delim:N err:", err)
} else {
    fmt.Println(y)
}
```

输出结果:

```
Good Night
10
5
delim:N err: EOF
```

2.2.6 其它函数

其它大部分函数、方法与 `strings` 包下的函数、方法类似，只是数据源从 `string` 变为了 `[]byte`，请参考 `strings` 包的用法。

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 ASCII 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")
```

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)?
```

未完待续。。。

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

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},
  },
}
```

比如对国内开发者很实用的汉字字符集:

```
var _Han = &RangeTable{
  R16: []Range16{
    {0x2e80, 0x2e99, 1},
    {0x2e9b, 0x2ef3, 1},
    {0x2f00, 0x2fd5, 1},
    {0x3005, 0x3005, 1},
    {0x3007, 0x3007, 1},
    {0x3021, 0x3029, 1},
    {0x3038, 0x303b, 1},
    {0x3400, 0x4db5, 1},
    {0x4e00, 0x9fea, 1},
    {0xf900, 0xfa6d, 1},
    {0xfa70, 0xfad9, 1},
  },
  R32: []Range32{
    {0x20000, 0x2a6d6, 1},
    {0x2a700, 0x2b734, 1},
    {0x2b740, 0x2b81d, 1},
    {0x2b820, 0x2ceal, 1},
    {0x2ceb0, 0x2ebe0, 1},
    {0x2f800, 0x2fa1d, 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 0-9
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 // 是否小写字母
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 // 是否大写字母
func Is(r rune, rangeTab *RangeTable) bool // r 是否为 rangeTab 类型的字符
func In(r rune, ranges ...*RangeTable) bool // r 是否为 ranges 中任意一个类型的字符
```

看下面这个例子:

```

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

    digit := '1'
    fmt.Println(unicode.IsDigit(digit))
    fmt.Println(unicode.IsNumber(digit))

    letter := 'Ⅷ'
    fmt.Println(unicode.IsDigit(letter))
    fmt.Println(unicode.IsNumber(letter))

    han := '你'
    fmt.Println(unicode.IsDigit(han))
    fmt.Println(unicode.Is(unicode.Han, han))
    fmt.Println(unicode.In(han, unicode.Gujarati, unicode.White_Space))
}

```

输出结果:

```

true
false
true
true
false
true
false
true
false

```

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 所占字节数:

- 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)

编码和解码到 rune:

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

- `func DecodeRune(p []byte) (r rune, size int)`
- `func DecodeRuneInString(s string) (r rune, size int)`
- `func DecodeLastRune(p []byte) (r rune, size int)`
- `func DecodeLastRuneInString(s string) (r rune, size int)`

是否为完整 rune:

- `func FullRune(p []byte) bool`
- `func FullRuneInString(s string) bool`

是否为 rune 第一个字节:

- `func RuneStart(b byte) bool`

示例:

```
word:=[]byte("界")

fmt.Println(utf8.Valid(word[:2]))
fmt.Println(utf8.ValidRune('界'))
fmt.Println(utf8.ValidString("世界"))

fmt.Println(utf8.RuneLen('界'))

fmt.Println(utf8.RuneCount(word))
fmt.Println(utf8.RuneCountInString("世界"))

p:=make([]byte,3)
utf8.EncodeRune(p,'好')
fmt.Println(p)
fmt.Println(utf8.DecodeRune(p))
fmt.Println(utf8.DecodeRuneInString("你好"))
fmt.Println(utf8.DecodeLastRune([]byte("你好")))
fmt.Println(utf8.DecodeLastRuneInString("你好"))

fmt.Println(utf8.FullRune(word[:2]))
fmt.Println(utf8.FullRuneInString("你好"))

fmt.Println(utf8.RuneStart(word[1]))
fmt.Println(utf8.RuneStart(word[0]))
```

运行结果:

```
false
true
true
3
1
2
[229 165 189]
22909 3
20320 3
22909 3
22909 3
false
true
```

```
false
true
```

2.5.3 utf16 包

utf16 的包的函数就比较少了。

将 uint16 和 rune 进行转换

```
func Encode(s []rune) []uint16
func EncodeRune(r rune) (r1, r2 rune)
func Decode(s []uint16) []rune
func DecodeRune(r1, r2 rune) rune
func IsSurrogate(r rune) bool // 是否为有效代理对
```

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。

示例：

```
words :=[]rune{' ', ' ', ' '}

u16:=utf16.Encode(words)
fmt.Println(u16)
fmt.Println(utf16.Decode(u16))

r1,r2:=utf16.EncodeRune(' ')
fmt.Println(r1,r2)
fmt.Println(utf16.DecodeRune(r1,r2))
fmt.Println(utf16.IsSurrogate(r1))
fmt.Println(utf16.IsSurrogate(r2))
fmt.Println(utf16.IsSurrogate(1234))
```

输出结果：

```
[55349 56528 55349 56529]
[120016 120017]
55349 56528
120016
true
true
false
```


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。

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
- 对排好序的数据集逆序

3.1.1 数据集排序

前面已经提到过，对数据集（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 {
    // 获取数据集元素个数
    Len() int
    // 如果 i 索引的数据小于 j 索引的数据，返回 true，且不会调用下面的 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()` 方法唯一的参数就是待排序的数据集。

该包还提供了一个方法可以判断数据集是否已经排好顺序，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 {
    name string // 姓名
    score int // 成绩
}

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},
    }

    // 打印未排序的 stus 数据
    fmt.Println("Default:\n\t", stus)
    //StuScores 已经实现了 sort.Interface 接口，所以可以调用 Sort 函数进行排序
    sort.Sort(stus)
    // 判断是否已经排好顺序，将会打印 true
    fmt.Println("IS Sorted?\n\t", sort.IsSorted(stus))
    // 打印排序后的 stus 数据
    fmt.Println("Sorted:\n\t", stus)
}

```

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}]
IS Sorted?
    true
Sorted:
    [{leao 90} {hikerell 91} {alan 95} {acmfly 96}]

```

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))
fmt.Println(stus)
```

最后一个方法: `Search()`

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

该方法会使用“二分查找”算法来找出能使 $f(x)(0 \leq x < n)$ 返回 `true` 的最小值 i 。
前提条件: $f(x)(0 \leq x < i)$ 均返回 `false`, $f(x)(i \leq x < n)$ 均返回 `true`。
如果不存在 i 可以使 $f(i)$ 返回 `true`, 则返回 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 以下是一个错误使用的例子：

```

s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
fmt.Println(sort.SearchInts(s, 2)) // 将会输出 0 而不是 1

```

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
```

3.1.3 []interface 排序与查找

通过前面的内容我们可以知道，只要实现了 `sort.Interface` 接口，即可通过 `sort` 包内的函数完成排序，查找等操作。并且 `sort` 包已经帮我们把 `[]int`，`[]float64`，`[]string` 三种类型都实现了该接口，我们可以方便的调用。但是这种用法对于其它数据类型的 `slice` 不友好，可能我们需要为大量的 `struct` 定义一个单独的 `[]struct` 类型，再为其实现 `sort.Interface` 接口，类似这样：

```
type Person struct {
    Name string
    Age  int
}
type Persons []Person
```

```
func (p Persons) Len() int {
    panic("implement me")
}

func (p Persons) Less(i, j int) bool {
    panic("implement me")
}

func (p Persons) Swap(i, j int) {
    panic("implement me")
}
```

思考一个问题：为什么 `sort` 包可以完成 `[]int` 的排序，而不能完成 `[]struct` 的排序？

因为排序涉及到比较两个变量的值，而 `struct` 可能包含多个属性，程序并不知道你想以哪一个属性或哪几个属性作为衡量大小的标准。如果你能帮助程序完成比较，并将结果返回，`sort` 包内的方法就可以完成排序，判断，查找等。`sort` 包提供了以下函数：

```
func Slice(slice interface{}, less func(i, j int) bool)
func SliceStable(slice interface{}, less func(i, j int) bool)
func SliceIsSorted(slice interface{}, less func(i, j int) bool) bool
func Search(n int, f func(int) bool) int
```

通过函数签名可以看到，排序相关的三个函数都接收 `[]interface`，并且需要传入一个比较函数，用于为程序比较两个变量的大小，因为函数签名和作用域的原因，这个函数只能是 `匿名函数`。

1. sort.Slice

该函数完成 `[]interface` 的排序，举个例子：

```
people := []struct {
    Name string
    Age  int
}{
    {"Gopher", 7},
    {"Alice", 55},
    {"Vera", 24},
    {"Bob", 75},
}

sort.Slice(people, func(i, j int) bool { return people[i].Age < people[j].Age }) // 按年龄升序排序
fmt.Println("Sort by age:", people)
```

输出结果：

```
By age: [{Gopher 7} {Vera 24} {Alice 55} {Bob 75}]
```

2. sort.SliceStable

该函数完成 `[]interface` 的稳定排序，举个例子：

```
people := []struct {
    Name string
    Age  int
}{
    {"Gopher", 7},
```

```

    {"Alice", 55},
    {"Vera", 24},
    {"Bob", 75},
}

sort.SliceStable(people, func(i, j int) bool { return people[i].Age > people[j].Age }) // 按年龄降序排序
fmt.Println("Sort by age:", people)

```

输出结果:

```
By age: [{Bob 75} {Alice 55} {Vera 24} {Gopher 7}]
```

3. sort.SliceIsSorted

该函数判断 []interface 是否为有序，举个例子:

```

people := []struct {
    Name string
    Age  int
}{
    {"Gopher", 7},
    {"Alice", 55},
    {"Vera", 24},
    {"Bob", 75},
}

sort.Slice(people, func(i, j int) bool { return people[i].Age > people[j].Age }) // 按年龄降序排序
fmt.Println("Sort by age:", people)
fmt.Println("Sorted:", sort.SliceIsSorted(people, func(i, j int) bool { return people[i].Age < people[j].Age
}))

```

输出结果:

```
Sort by age: [{Bob 75} {Alice 55} {Vera 24} {Gopher 7}]
Sorted: false
```

sort 包没有为 []interface 提供反序函数，但是从 1 和 2 可以看出，我们传入的比较函数已经决定了排序结果是升序还是降序。

判断 slice 是否为有序，同样取决于我们传入的比较函数，从 3 可以看出，虽然 slice 已经按年龄降序排序，但我们在判断 slice 是否为有序时给定的比较函数是判断其是否为升序有序，所以最终得到的结果为 false。

4. sort.Search

该函数判断 []interface 是否存在指定元素，举个例子:

- 升序 slice

sort 包为 []int, []float64, []string 提供的 Search 函数其实也是调用的该函数，因为该函数是使用的二分查找法，所以要求 slice 为升序排序状态。并且判断条件必须为 `>=`，这也是官方库提供的三个查找相关函数的的写法。

举个例子:

```

a := []int{2, 3, 4, 200, 100, 21, 234, 56}
x := 21

```



```
sort.Slice(a, func(i, j int) bool { return a[i] < a[j] }) // 升序排序
index := sort.Search(len(a), func(i int) bool { return a[i] >= x }) // 查找元素

if index < len(a) && a[index] == x {
    fmt.Printf("found %d at index %d in %v\n", x, index, a)
} else {
    fmt.Printf("%d not found in %v, index:%d\n", x, a, index)
}
```

输出结果:

```
found 21 at index 3 in [2 3 4 21 56 100 200 234]
```

- 降序 slice

如果 slice 是降序状态，而我们又不想将其变为升序，只需将判断条件由 `>=` 变更为 `<=` 即可。此处就不给代码了，有兴趣的同学可以自己去尝试一下。推荐采用升序排列及相应的判断条件，与官方函数保持风格一致。

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。索引布局如下：

```
0 1 2 3 4 5 6 7 8 9 10 11
```

假设 (`heap[1]== 小明`) 它的左子树 (`heap[3]== 小黑`) 和右子树 (`heap[4]== 大黄`) 且 `小明 > 小黑 > 大黄`；

堆内部实现了 `down` 和 `up` 函数：

`down` 函数用于将索引 `i` 处存储的值（设 `i=1`，即小明）与它的左子树（小黑）和右子树（大黄）相比，将三者最小的值大黄与小明的位置交换，交换后小明继续与交换后的子树 (`heap[9]`和 `heap[10]`) 相比，重复以上步骤，直到小明位置不变。

`up` 函数用于将索引 `i` 处的值（设 `i=3`，即小黑）与他的父节点（小明）比较，将两者较小的值放到父节点上，本例中即交换小黑和小明的位置，之后小黑继续与其父节点比较，重复以上步骤，直到小黑位置不变。

假设 `heap[11]== 阿花`

当从堆中 `Pop` 一个元素的时候，先把元素和最后一个节点的值（阿花）交换，然后弹出，然后对阿花调用 `down`，向下保证最小堆。

当往堆中 `Push` 一个元素的时候，这个元素插入到最后一个节点，本例中为 `heap[12]`，即作为 `heap[5]`的右子树，然后调用 `up` 函数向上比较。

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)(0x2081be150), list:(*list.List)(
0x2081be150), Value:1}
second: &list.Element{next:(*list.Element)(0x2081be150), prev:(*list.Element)(0x2081be180), 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 // 在某个元素后插入
func (l *List) InsertBefore(v interface{}, mark *Element) *Element // 在某个元素前插入
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，能遍历一遍环，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 // 当前元素的下一个元素  
func (r *Ring) Prev() *Ring // 当前元素的上一个元素  
func (r *Ring) Unlink(n int) *Ring // 从当前元素开始，删除 n 个元素
```

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](#)

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。

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。

时区

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` 协程同时使用。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 `Time` 都具有一个地点信息（即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。（最多到微秒）

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 15:00:00"), 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 `Truncate`，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))
```

Time类型详解

```
// 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)
```

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` 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未来某一点，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（heap）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, elapse:", 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, elapse: 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 `Ticker` 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`》。

数学计算

math — 基本数学函数

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 Expml(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 uint32 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
```

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称。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。

os — 平台无关的操作系统功能实现

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

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

下面是一个简单的例子，打开一个文件并从中读取一些数据：

```
file, err := os.Open("file.go") // For read access.
if err != nil {
    log.Fatal(err)
}
```

如果打开失败，错误字符串是自解释的，例如：

```
open file.go: no such file or directory
```

而不像 C 语言，需要额外的函数（或宏）来解释错误码。

文件 I/O

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

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

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

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

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，如果不关闭文件描述符，长期运行的服务可能会把文件描述符耗尽。

所以，通常的写法如下：

```
file, err := os.Open("/tmp/studygolang.txt")
if err != nil {
    // 错误处理，一般会阻止程序往下执行
    return
}
defer file.Close()
```

关于返回值 `error`

以下两种情况会导致 `Close` 返回错误：

1. 关闭一个未打开的文件；
2. 两次关闭同一个文件；

通常，我们不会去检查 `Close` 的错误。

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` 操作的文件实际位置，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

`truncate` 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_blksize; // 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 `syscall.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

可以显式改变文件的访问时间和修改时间。

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

`Chtimes` 修改 `name` 指定的文件对象的访问时间和修改时间，类似 Unix 的 `utime()` 或 `utimes()` 函数。底层的文件系统可能会截断 / 舍入时间单位到更低的精确度。如果出错，会返回 `*PathError` 类型的错误。在 Unix 中，底层实现会调用 `utimensat()`，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` 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` 修改一个文件的名称或移动一个文件。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 := 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` 填满。这里采用的验证方法是分配一个更大的 `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](#)。

path/filepath — 操作路径

`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杠构成的，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/logic/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`。当 `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 studygolang.txt 建一个符号链接 studygolang.txt.2
fmt.Println(filepath.EvalSymlinks("symlink/studygolang.txt.2"))
fmt.Println(os.Readlink("symlink/studygolang.txt.2"))

// Output:
// 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。

io/fs — 抽象文件系统

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

本节除了讲解标准库的相关内容，还会实现一个文件系统作为例子。

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

三个核心接口

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

fs.FS

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

```
type FS interface {
    // Open opens the named file.
    //
    // When Open returns an error, it should be of type *PathError
    // with the Op field set to "open", the Path field set to name,
    // and the Err field describing the problem.
    //
    // Open should reject attempts to open names that do not satisfy
    // ValidPath(name), returning a *PathError with Err set to
    // ErrInvalid or ErrNotExist.
    Open(name string) (File, error)
}
```

该接口只有一个方法，即打开一个命名文件，该方法的实现要求如下：

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

```
type PathError struct {
    Op    string
    Path  string
    Err   error
}
```

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

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

```
type PathError = fs.PathError
```

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

```
func ValidPath(name string) bool
```

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

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

fs.File

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

```
type File interface {
    Stat() (FileInfo, error)
    Read([]byte) (int, error)
    Close() error
}
```

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

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

在 `Go` 中，你应该始终记住，打开文件，进行操作后，记得关闭文件，否则会泄露文件描述符。所以，`fs.File` 的第第三个方法就是 `Close` 方法，它的签名和 `io.Closer` 是一致的。

fs.FileInfo

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

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

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

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

实现一个文件系统

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

实现 fs.File 和 fs.FileInfo

实现文件系统先实现这两个接口。我们通过一个类型来实现：

```
type file struct {
    name      string
    content   *bytes.Buffer
    modTime   time.Time
    closed    bool
}
```

```

func (f *file) Read(p []byte) (int, error) {
    if f.closed {
        return 0, errors.New("file closed")
    }

    return f.content.Read(p)
}

func (f *file) Stat() (fs.FileInfo, error) {
    if f.closed {
        return nil, errors.New("file closed")
    }

    return f, nil
}

// Close 关闭文件，可以调用多次。
func (f *file) Close() error {
    f.closed = true
    return nil
}

// 实现 fs.FileInfo

func (f *file) Name() string {
    return f.name
}

func (f *file) Size() int64 {
    return int64(f.content.Len())
}

func (f *file) Mode() fs.FileMode {
    // 固定为 0444
    return 0444
}

func (f *file) ModTime() time.Time {
    return f.modTime
}

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

func (f *file) Sys() interface{} {
    return nil
}

```

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

实现 fs.FS

实现了 fs.File，通过它可以实现 fs.FS:

```

type FS struct {
    files map[string]*file
}

func NewFS() *FS {
    return &FS{
        files: make(map[string]*file),
    }
}

func (fsys *FS) Open(name string) (fs.File, error) {
    if !fs.ValidPath(name) {
        return nil, &fs.PathError{
            Op: "open",
            Path: name,
            Err: fs.ErrInvalid,
        }
    }

    if f, ok := fsys.files[name]; !ok {
        return nil, &fs.PathError{
            Op: "open",
            Path: name,
            Err: fs.ErrNotExist,
        }
    } else {
        return f, nil
    }
}

```

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

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

```

func (fsys *FS) WriteFile(name, content string) error {
    if !fs.ValidPath(name) {
        return &fs.PathError{
            Op: "write",
            Path: name,
            Err: fs.ErrInvalid,
        }
    }

    f := &file{
        name: name,
        content: bytes.NewBufferString(content),
        modTime: time.Now(),
    }

    fsys.files[name] = f

    return nil
}

```

WriteFile 方法就是生成一个 file 然后存入 files 中。

验证

一个基于内存的文件系统已经实现完成，接下来需要验证下。

```
func TestMemFS(t *testing.T) {
    name := "x/y/name.txt"
    content := "This is polarisxu, welcome."
    memFS := memfs.NewFS()
    err := memFS.WriteFile(name, content)
    if err != nil {
        t.Fatal(err)
    }

    f, err := memFS.Open(name)
    if err != nil {
        t.Fatal(err)
    }
    defer f.Close()

    fi, err := f.Stat()
    if err != nil {
        t.Fatal(err)
    }

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

    var result = make([]byte, int(fi.Size()))
    n, err := f.Read(result)
    if err != nil {
        t.Fatal(err)
    }

    if string(result[:n]) != content {
        t.Errorf("expect: %s, actual: %s", content, result[:n])
    }
}
```

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

增强型接口

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

fs.DirEntry 和相关接口

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

```
type DirEntry interface {
    // Name returns the name of the file (or subdirectory) described by the entry.
    // This name is only the final element of the path (the base name), not the entire path.
    // For example, Name would return "hello.go" not "/home/gopher/hello.go".
    Name() string

    // IsDir reports whether the entry describes a directory.
    IsDir() bool

    // Type returns the type bits for the entry.
    // The type bits are a subset of the usual FileMode bits, those returned by the FileMode.Type method.
    Type() FileMode
}
```

```

// Info returns the FileInfo for the file or subdirectory described by the entry.
// The returned FileInfo may be from the time of the original directory read
// or from the time of the call to Info. If the file has been removed or renamed
// since the directory read, Info may return an error satisfying errors.Is(err, ErrNotExist).
// If the entry denotes a symbolic link, Info reports the information about the link itself,
// not the link's target.
Info() (FileInfo, error)
}

```

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

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

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

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

```

type ReadDirFS interface {
    FS

    // ReadDir reads the named directory
    // and returns a list of directory entries sorted by filename.
    ReadDir(name string) ([]DirEntry, error)
}

```

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

```

type ReadDirFile interface {
    File

    // ReadDir reads the contents of the directory and returns
    // a slice of up to n DirEntry values in directory order.
    // Subsequent calls on the same file will yield further DirEntry values.
    //
    // If n > 0, ReadDir returns at most n DirEntry structures.
    // In this case, if ReadDir returns an empty slice, it will return
    // a non-nil error explaining why.
    // At the end of a directory, the error is io.EOF.
    //
    // If n <= 0, ReadDir returns all the DirEntry values from the directory
    // in a single slice. In this case, if ReadDir succeeds (reads all the way
    // to the end of the directory), it returns the slice and a nil error.
    // If it encounters an error before the end of the directory,
    // ReadDir returns the DirEntry list read until that point and a non-nil error.
    ReadDir(n int) ([]DirEntry, error)
}

```

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

其他 fs.FS 相关的接口

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

fs.ReadFileFS

该接口的定义如下：

```
type ReadFileFS interface {
    FS

    // ReadFile reads the named file and returns its contents.
    // A successful call returns a nil error, not io.EOF.
    // (Because ReadFile reads the whole file, the expected EOF
    // from the final Read is not treated as an error to be reported.)
    ReadFile(name string) ([]byte, error)
}
```

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

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

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

fs.StatFS

该接口的定义如下：

```
type StatFS interface {
    FS

    // Stat returns a FileInfo describing the file.
    // If there is an error, it should be of type *PathError.
    Stat(name string) (FileInfo, error)
}
```

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

fs.GlobFS

该接口的定义如下：

```
type GlobFS interface {
    FS

    // Glob returns the names of all files matching pattern,
    // providing an implementation of the top-level
    // Glob function.
    Glob(pattern string) ([]string, error)
}
```

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

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

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

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

fs.SubFS

该接口的定义如下：

```
type SubFS interface {
    FS

    // Sub returns an FS corresponding to the subtree rooted at dir.
    Sub(dir string) (FS, error)
}
```

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

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

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

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

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

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

设计思想

上面啰嗦讲了几个 fs.FS 相关接口，其中目的之一是希望理解其设计思想。

io/fs 包中和 fs.FS 相关的接口如下：

- fs.ReadDirFS
- fs.ReadFileFS
- fs.StatFS
- fs.SubFS
- fs.GlobFS

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

完善内存文件系统

是时候完善我们上面实现的内存文件系统了。

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

如何设计

先通过一个类图表示 `io/fs` 包相关接口的关系。

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

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

所以，我们在实现 `fs.FS` 接口时，定义的类型 `FS` 有一个根目录字段：

```
type FS struct {
    rootDir *dir
}
```

文件的实现

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

- `fs.File`
- `fs.FileInfo`
- `fs.DirEntry`

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

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

目录的实现

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

- `fs.DirEntry`
- `fs.File`
- `fs.ReadDirFile`
- `fs.FileInfo`

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

```
// dir 代表一个目录
type dir struct {
    name      string
    modTime  time.Time

    // 存放该目录下的子项, value 可能是 *dir 或 *file
    children map[string]fs.DirEntry
}
```

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

fs.FS 接口的实现

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

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

创建目录和文件的实现

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

先看创建目录的实现。

创建目录，实际上是构建一个层级关系。方法签名如下：

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

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

```
cur := fsys.rootDir
parts := strings.Split(path, "/")
for _, part := range parts {
    child := cur.children[part]
    if child == nil {
        childDir := &dir{
            name:      part,
            modTime:  time.Now(),
            children:  make(map[string]fs.DirEntry),
        }
        cur.children[part] = childDir
        cur = childDir
    } else {
        childDir, ok := child.(*dir)
        if !ok {
            return fmt.Errorf("%s is not directory", part)
        }
    }
}
```

```

    cur = childDir
  }
}

```

文件的创建和内容写入通过 `WriteFile` 方法实现。签名如下：

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

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

```

// getDir 通过一个路径获取其 dir 类型实例
func (fsys *FS) getDir(path string) (*dir, error) {
    parts := strings.Split(path, "/")

    cur := fsys.rootDir
    for _, part := range parts {
        child := cur.children[part]
        if child == nil {
            return nil, fmt.Errorf("%s is not exists", path)
        }

        childDir, ok := child.(*dir)
        if !ok {
            return nil, fmt.Errorf("%s is not directory", path)
        }

        cur = childDir
    }

    return cur, nil
}

```

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

```

filename := filepath.Base(name)

dir.children[filename] = &file{
    name: filename,
    content: bytes.NewBufferString(content),
    modTime: time.Now(),
}

```

完整的实现代码

以下是 `dir` 类型的实现，代表一个目录，注意注释。

```

// dir 代表一个目录
type dir struct {
    name string
    modTime time.Time

    // 存放该目录下的子项，value 可能是 *dir 或 *file
    children map[string]fs.DirEntry

    // ReadDir 遍历用
}

```

```

    idx int
}

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

func (d *dir) Read(p []byte) (int, error) {
    return 0, &fs.PathError{
        Op: "read",
        Path: d.name,
        Err: errors.New("is directory"),
    }
}

func (d *dir) Stat() (fs.FileInfo, error) {
    return d, nil
}

func (d *dir) Close() error {
    return nil
}

// ReadDir 实现 fs.ReadDirFile 接口，方便遍历目录
func (d *dir) ReadDir(n int) ([]fs.DirEntry, error) {
    names := make([]string, 0, len(d.children))
    for name := range d.children {
        names = append(names, name)
    }

    totalEntry := len(names)
    if n <= 0 {
        n = totalEntry
    }

    dirEntries := make([]fs.DirEntry, 0, n)
    for i := d.idx; i < n && i < totalEntry; i++ {
        name := names[i]
        child := d.children[name]

        f, isFile := child.(*file)
        if isFile {
            dirEntries = append(dirEntries, f)
        } else {
            dirEntry := child.(*dir)
            dirEntries = append(dirEntries, dirEntry)
        }

        d.idx = i
    }

    return dirEntries, nil
}

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

func (d *dir) Name() string {
    return d.name
}

func (d *dir) Size() int64 {
    return 0
}

```

```

func (d *dir) Mode() fs.FileMode {
    return fs.ModeDir | 0444
}

func (d *dir) ModTime() time.Time {
    return d.modTime
}

func (d *dir) IsDir() bool {
    return true
}

func (d *dir) Sys() interface{} {
    return nil
}

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

func (d *dir) Type() fs.FileMode {
    return d.Mode()
}

func (d *dir) Info() (fs.FileInfo, error) {
    return d, nil
}

```

接着是 `file` 的实现，代表一个文件，注意注释。

```

// file 代表一个文件
type file struct {
    name      string
    // 存放文件内容
    content *bytes.Buffer
    modTime  time.Time
    closed   bool
}

// 实现 fs.File 接口

func (f *file) Read(p []byte) (int, error) {
    if f.closed {
        return 0, errors.New("file closed")
    }

    return f.content.Read(p)
}

func (f *file) Stat() (fs.FileInfo, error) {
    if f.closed {
        return nil, errors.New("file closed")
    }

    return f, nil
}

// Close 关闭文件，可以调用多次。
func (f *file) Close() error {
    f.closed = true
    return nil
}

```

```

}

// 实现 fs.FileInfo 接口

func (f *file) Name() string {
    return f.name
}

func (f *file) Size() int64 {
    return int64(f.content.Len())
}

func (f *file) Mode() fs.FileMode {
    // 固定为 0444
    return 0444
}

func (f *file) ModTime() time.Time {
    return f.modTime
}

func (f *file) IsDir() bool {
    return false
}

func (f *file) Sys() interface{} {
    return nil
}

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

func (f *file) Type() fs.FileMode {
    return f.Mode()
}

func (f *file) Info() (fs.FileInfo, error) {
    return f, nil
}

```

有了目录（`dir`）和文件（`file`），看 `fs.FS` 的实现。

```

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

// NewFS 创建一个内存文件系统的实例
func NewFS() *FS {
    return &FS{
        rootDir: &dir{
            children: make(map[string]fs.DirEntry),
        },
    }
}

// Open 实现 fs.FS 的 Open 方法
func (fsys *FS) Open(name string) (fs.File, error) {
    // 1、校验 name
    if !fs.ValidPath(name) {
        return nil, &fs.PathError{

```



```

        Op: "open",
        Path: name,
        Err: fs.ErrInvalid,
    }
}

// 2、根目录处理
if name == "." || name == "/" {
    // 重置目录的遍历
    fsys.rootDir.idx = 0
    return fsys.rootDir, nil
}

// 3、根据 name 在目录树中进行查找
cur := fsys.rootDir
parts := strings.Split(name, "/")
for i, part := range parts {
    // 不存在返回错误
    child := cur.children[part]
    if child == nil {
        return nil, &fs.PathError{
            Op: "open",
            Path: name,
            Err: fs.ErrNotExist,
        }
    }

    // 是否是文件
    f, ok := child.(*file)
    if ok {
        // 文件名是最后一项
        if i == len(parts)-1 {
            return f, nil
        }

        return nil, &fs.PathError{
            Op: "open",
            Path: name,
            Err: fs.ErrNotExist,
        }
    }

    // 是否是目录
    d, ok := child.(*dir)
    if !ok {
        return nil, &fs.PathError{
            Op: "open",
            Path: name,
            Err: errors.New("not a directory"),
        }
    }

    // 重置，避免遍历问题
    d.idx = 0

    cur = d
}

return cur, nil
}

// MkdirAll 这不是 io/fs 的要求，但一个文件系统目录树需要可以构建
// 这个方法就是用来创建目录

```

```

func (fsys *FS) MkdirAll(path string) error {
    if !fs.ValidPath(path) {
        return errors.New("Invalid path")
    }

    if path == "." {
        return nil
    }

    cur := fsys.rootDir
    parts := strings.Split(path, "/")
    for _, part := range parts {
        child := cur.children[part]
        if child == nil {
            childDir := &dir{
                name:    part,
                modTime: time.Now(),
                children: make(map[string]fs.DirEntry),
            }
            cur.children[part] = childDir
            cur = childDir
        } else {
            childDir, ok := child.(*dir)
            if !ok {
                return fmt.Errorf("%s is not directory", part)
            }
            cur = childDir
        }
    }

    return nil
}

// WriteFile 也不是 io/fs 的要求, 和 MkdirAll 类似, 文件内容也需要有接口写入
func (fsys *FS) WriteFile(name, content string) error {
    if !fs.ValidPath(name) {
        return &fs.PathError{
            Op:    "write",
            Path: name,
            Err:   fs.ErrInvalid,
        }
    }

    var err error
    dir := fsys.rootDir

    path := filepath.Dir(name)
    if path != "." {
        dir, err = fsys.getDir(path)
        if err != nil {
            return err
        }
    }
    filename := filepath.Base(name)

    dir.children[filename] = &file{
        name:    filename,
        content: bytes.NewBufferString(content),
        modTime: time.Now(),
    }
}

```

```

return nil
}

// getDir 通过一个路径获取其 dir 类型实例
func (fsys *FS) getDir(path string) (*dir, error) {
    parts := strings.Split(path, "/")

    cur := fsys.rootDir
    for _, part := range parts {
        child := cur.children[part]
        if child == nil {
            return nil, fmt.Errorf("%s is not exists", path)
        }

        childDir, ok := child.(*dir)
        if !ok {
            return nil, fmt.Errorf("%s is not directory", path)
        }

        cur = childDir
    }

    return cur, nil
}

```

为了完整性，我把所有代码列出来了，关键的地方加上了注释。

验证正确性并学习 fs.WalkDir

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

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

```

|— a
|   |— b
|   |   |— z
|   |— x
|   |   |— y
|   |   |— z
|   |— name.txt

```

即执行如下代码：

```

memFS := memfs.NewFS()
memFS.MkdirAll("a/b/z")
memFS.MkdirAll("x/y/z")
memFS.WriteFile("x/name.txt", "This is polarisxu, welcome.")

```

基于这个目录树，我们该怎么遍历它？

自己实现遍历方法

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

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

```
func walk(fsys fs.FS, parent, base string) error {
    dirEntries, err := fs.ReadDir(fsys, filepath.Join(parent, base))
    if err != nil {
        return err
    }
    for _, dirEntry := range dirEntries {
        name := dirEntry.Name()
        fmt.Println(name)

        if dirEntry.IsDir() {
            err = walk(fsys, filepath.Join(parent, base), name)
        }
    }
    return err
}
```

然后用根目录调用它：

```
walk(memFS, "", ".")
```

使用 fs.WalkDir 实现

其实标准库为我们实现了这样的功能，通过它提供的方法可以很容易的实现遍历。

```
fs.WalkDir(memFS, ".", func(path string, d fs.DirEntry, err error) error {
    fmt.Println(path)
    return nil
})
```

是不是不要太简单？

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

小结

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

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

- <https://github.com/jhchabran/gistfs>: 用于读取 GitHub gists 的文件；
- <https://github.com/benbjohnson/hashfs>: hash 文件，以便 HTTP Cache；
- <https://github.com/psanford/memfs>: 内存文件系统的实现；本小节文件系统的实现参考了它。

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、网盘等，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。

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，要么传递给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 2 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，但连接池中并没有空闲的可用，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。

数据压缩与归档

在计算机算法中，经常会有以空间换时间的做法；而无损压缩算法以压缩或解压缩数据花费的时间来换取存储空间。除此之外，压缩还有另外一个重要用途，那就是减少网络传输的数据量，进而减少网络传输的时间。

Go 标准库实现了一些最流行的压缩标准。`zlib` 和 `gzip` 提供了 GNU zip 库，`bzip2` 用于读写 `bzip2` 格式。这些格式都可以处理数据流而不考虑输入格式，并且提供了接口可以透明地读写压缩文件。除此之外，标准库还提供了 DEFLATE 压缩算法的实现，`gzip` 和 `zlib` 可以读取基于 DEFLATE 的文件格式。

标准库提供了 LZW 压缩算法（串表压缩算法）的实现，该算法常用的文件格式：`GIF` 和 `PDF`。另外，`TIFF` 文件格式使用和该算法类似，但和目前 LZW 算法版本不兼容。

标准库还提供一些包管理归档 (`archive`) 格式，将多个文件合并到一个文件，从而将其作为一个单元管理。`archive/tar` 读写 `UNIX` 磁带归档格式，这是一种老标准，但由于其灵活性，当前仍得到广泛使用。`archive/zip` 根据 `zip` 格式来处理归档，这种格式因 `PC` 程序 `PKZIP` 得以普及，原先在 `MS-DOS` 和 `Windows` 下使用，不过由于其 `API` 的简单性以及这种格式的可移植性，现在也用于其他平台。

flate - DEFLATE 压缩算法

DEFLATE 是同时使用了哈夫曼编码（Huffman Coding）与 LZ77 算法的一个无损数据压缩算法，是一种压缩数据流的算法。任何需要流式压缩的地方都可以用。目前 zip 压缩文件默认使用的就是该算法。

关于算法的原理，以及哈夫曼编码（Huffman Coding）与 LZ77 算法，感兴趣的读者可以查询相关资料，这里推荐 [GZIP 压缩原理分析——第五章 Deflate 算法详解](#) 序列文章。

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 2 个包：

- `testing` 方便进行 Go 包的自动化单元测试、基准测试
- `net/http/httptest` 提供测试 HTTP 的工具

testing - 单元测试

`testing` 为 Go 语言 `package` 提供自动化测试的支持。通过 `go test` 命令，能够自动执行如下形式的任何函数：

```
func TestXxx(*testing.T)
```

注意：Xxx 可以是任何字母数字字符串，但是第一个字母不能是小写字母。

在这些函数中，使用 `Error`、`Fail` 或相关方法来发出失败信号。

要编写一个新的测试套件，需要创建一个名称以 `_test.go` 结尾的文件，该文件包含 `TestXxx` 函数，如上所述。将该文件放在与被测试文件相同的包中。该文件将被排除在正常的程序包之外，但在运行 `go test` 命令时将被包含。有关详细信息，请运行 `go help test` 和 `go help testflag` 了解。

如果有需要，可以调用 `*T` 和 `*B` 的 `Skip` 方法，跳过该测试或基准测试：

```
func TestTimeConsuming(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping test in short mode.")
    }
    ...
}
```

第一个单元测试

要测试的代码：

```
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

测试代码：

```
func TestFib(t *testing.T) {
    var (
        in      = 7
        expected = 13
    )
    actual := Fib(in)
    if actual != expected {
        t.Errorf("Fib(%d) = %d; expected %d", in, actual, expected)
    }
}
```

执行 `go test .`，输出：

```
$ go test .
ok      chapter09/testing 0.007s
```

表示测试通过。

我们将 `Sum` 函数改为：

```
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-1)
}
```

再执行 `go test .`，输出：

```
$ go test .
--- FAIL: TestSum (0.00s)
    t_test.go:16: Fib(10) = 64; expected 13
FAIL
FAIL    chapter09/testing    0.009s
```

Table-Driven Test

测试讲究 case 覆盖，按上面的方式，当我们要覆盖更多 case 时，显然通过修改代码的方式很笨拙。这时我们可以采用 Table-Driven 的方式写测试，标准库中有很多测试是使用这种方式写的。

```
func TestFib(t *testing.T) {
    var fibTests = []struct {
        in      int // input
        expected int // expected result
    }{
        {1, 1},
        {2, 1},
        {3, 2},
        {4, 3},
        {5, 5},
        {6, 8},
        {7, 13},
    }

    for _, tt := range fibTests {
        actual := Fib(tt.in)
        if actual != tt.expected {
            t.Errorf("Fib(%d) = %d; expected %d", tt.in, actual, tt.expected)
        }
    }
}
```

由于我们使用的是 `t.Errorf`，即使其中某个 case 失败，也不会终止测试执行。

T 类型

单元测试中，传递给测试函数的参数是 `*testing.T` 类型。它用于管理测试状态并支持格式化测试日志。测试日志会在执行测试的过程中不断累积，并在测试完成时转储至标准输出。

当测试函数返回时，或者当测试函数调用 `FailNow`、`Fatal`、`Fatalf`、`SkipNow`、`Skip`、`Skipf` 中的任意一个时，则宣告该测试函数结束。跟 `Parallel` 方法一样，以上提到的这些方法只能在运行测试函数的 goroutine 中调用。

至于其他报告方法，比如 `Log` 以及 `Error` 的变种，则可以在多个 `goroutine` 中同时进行调用。

报告方法

上面提到的系列包括方法，带 `f` 的是格式化的，格式化语法参考 `fmt` 包。

T 类型内嵌了 `common` 类型，`common` 提供这一系列方法，我们经常会用到的（注意，这里说的测试中断，都是指当前测试函数）：

1) 当我们遇到一个断言错误的时候，标识这个测试失败，会使用到：

`Fail` : 测试失败，测试继续，也就是之后的代码依然会执行
`FailNow` : 测试失败，测试中断

在 `FailNow` 方法实现的内部，是通过调用 `runtime.Goexit()` 来中断测试的。

2) 当我们遇到一个断言错误，只希望跳过这个错误，但是不希望标识测试失败，会使用到：

`SkipNow` : 跳过测试，测试中断

在 `SkipNow` 方法实现的内部，是通过调用 `runtime.Goexit()` 来中断测试的。

3) 当我们只希望打印信息，会用到：

`Log` : 输出信息
`Logf` : 输出格式化的信息

注意：默认情况下，单元测试成功时，它们打印的信息不会输出，可以通过加上 `-v` 选项，输出这些信息。但对于基准测试，它们总是会被输出。

4) 当我们希望跳过这个测试，并且打印出信息，会用到：

`Skip` : 相当于 `Log + SkipNow`
`Skipf` : 相当于 `Logf + SkipNow`

5) 当我们希望断言失败的时候，标识测试失败，并打印出必要的信息，但是测试继续，会用到：

`Error` : 相当于 `Log + Fail`
`Errorf` : 相当于 `Logf + Fail`

6) 当我们希望断言失败的时候，标识测试失败，打印出必要的信息，但中断测试，会用到：

`Fatal` : 相当于 `Log + FailNow`
`Fatalf` : 相当于 `Logf + FailNow`

Parallel 测试

包中的 `Parallel` 方法表示当前测试只会与其他带有 `Parallel` 方法的测试并行进行测试。

下面例子将演示 `Parallel` 的使用方法：

```

var (
    data = make(map[string]string)
    locker sync.RWMutex
)

func WriteToMap(k, v string) {
    locker.Lock()
    defer locker.Unlock()
    data[k] = v
}

func ReadFromMap(k string) string {
    locker.RLock()
    defer locker.RUnlock()
    return data[k]
}

```

测试代码:

```

var pairs = []struct {
    k string
    v string
}{
    {"polaris", "徐新华"},
    {"studygolang", "Go 语言中文网"},
    {"stdlib", "Go 语言标准库"},
    {"polaris1", "徐新华 1"},
    {"studygolang1", "Go 语言中文网 1"},
    {"stdlib1", "Go 语言标准库 1"},
    {"polaris2", "徐新华 2"},
    {"studygolang2", "Go 语言中文网 2"},
    {"stdlib2", "Go 语言标准库 2"},
    {"polaris3", "徐新华 3"},
    {"studygolang3", "Go 语言中文网 3"},
    {"stdlib3", "Go 语言标准库 3"},
    {"polaris4", "徐新华 4"},
    {"studygolang4", "Go 语言中文网 4"},
    {"stdlib4", "Go 语言标准库 4"},
}

// 注意 TestWriteToMap 需要在 TestReadFromMap 之前
func TestWriteToMap(t *testing.T) {
    t.Parallel()
    for _, tt := range pairs {
        WriteToMap(tt.k, tt.v)
    }
}

func TestReadFromMap(t *testing.T) {
    t.Parallel()
    for _, tt := range pairs {
        actual := ReadFromMap(tt.k)
        if actual != tt.v {
            t.Errorf("the value of key(%s) is %s, expected: %s", tt.k, actual, tt.v)
        }
    }
}

```

试验步骤:

1. 注释掉 `WriteToMap` 和 `ReadFromMap` 中 `locker` 保护的代码，同时注释掉测试代码中的 `t.Parallel`，执行测试，测试通过，即使加上 `-race`，测试依然通过；
2. 只注释掉 `WriteToMap` 和 `ReadFromMap` 中 `locker` 保护的代码，执行测试，测试失败（如果未失败，加上 `-race` 一定会失败）；

如果代码能够进行并行测试，在写测试时，尽量加上 `Parallel`，这样可以测试出一些可能的问题。

关于 `Parallel` 的更多内容，会在 [子测试](#) 中介绍。

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` 的形式存在。

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("empty 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 的结构体

- 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：内部使用的结构体
- InternalExample：内部使用的结构体
- InternalTest：内部使用的结构体
- M：main 测试使用的结构体
- PB：Parallel benchmarks 并行测试使用的结构体
- 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：

1) 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
```

2) 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
```

3) 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
```

4) 当我们希望跳过这个用例, 并且打印出信息:

```
Skip : Log + SkipNow  
Skipf : Logf + SkipNow
```

5) 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
```

6) 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
```

扩展阅读

[GO 中如何进行单元测试](#)
[GoDoc - testing](#)
[testing/testing.go 源代码](#)

testing - 基准测试

在 `_test.go` 结尾的测试文件中，如下形式的函数：

```
func BenchmarkXxx(*testing.B)
```

被认为是基准测试，通过 `go test` 命令，加上 `-bench` 标志来执行。多个基准测试按照顺序运行。

基准测试函数的形式如下：

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

基准函数会运行目标代码 `b.N` 次。在基准执行期间，程序会自动调整 `b.N` 直到基准测试函数持续足够长的时间。输出结果形如：

```
BenchmarkHello 10000000 282 ns/op
```

意味着循环执行了 10000000 次，每次循环花费 282 纳秒 (ns)。

如果基准测试在循环前需要一些耗时的配置，则可以先重置定时器：

```
func BenchmarkBigLen(b *testing.B) {
    big := NewBig()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        big.Len()
    }
}
```

如果基准测试需要在并行设置中测试性能，则可以使用 `RunParallel` 辅助函数；这样的基准测试一般与 `go test -cpu` 标志一起使用：

```
func BenchmarkTemplateParallel(b *testing.B) {
    templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
    b.RunParallel(func(pb *testing.PB) {
        // 每个 goroutine 有属于自己的 bytes.Buffer.
        var buf bytes.Buffer
        for pb.Next() {
            // 循环体在所有 goroutine 中总共执行 b.N 次
            buf.Reset()
            templ.Execute(&buf, "World")
        }
    })
}
```

基准测试示例

接着上一节的例子，我们对 `Fib` 进行基准测试：

```
func BenchmarkFib10(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(10)
    }
}
```

执行 `go test -bench=.`，输出：

```
$ go test -bench=.
BenchmarkFib10-4      3000000      424 ns/op
PASS
ok   chapter09/testing  1.724s
```

这里测试了 `Fib(10)` 的情况，我们可能需要测试更多不同的情况，这时可以改写我们的测试代码：

```
func BenchmarkFib1(b *testing.B) { benchmarkFib(1, b) }
func BenchmarkFib2(b *testing.B) { benchmarkFib(2, b) }
func BenchmarkFib3(b *testing.B) { benchmarkFib(3, b) }
func BenchmarkFib10(b *testing.B) { benchmarkFib(10, b) }
func BenchmarkFib20(b *testing.B) { benchmarkFib(20, b) }
func BenchmarkFib40(b *testing.B) { benchmarkFib(40, b) }

func benchmarkFib(i int, b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(i)
    }
}
```

再次执行 `go test -bench=.`，输出：

```
$ go test -bench=.
BenchmarkFib1-4      1000000000    2.58 ns/op
BenchmarkFib2-4      200000000    7.38 ns/op
BenchmarkFib3-4      100000000    13.0 ns/op
BenchmarkFib10-4     3000000      429 ns/op
BenchmarkFib20-4     30000        54335 ns/op
BenchmarkFib40-4     2            805759850 ns/op
PASS
ok   chapter09/testing  15.361s
```

默认情况下，每个基准测试最少运行 1 秒。如果基准测试函数返回时，还不到 1 秒钟，`b.N` 的值会按照序列 `1,2,5,10,20,50,...` 增加，同时再次运行基准测试函数。

我们注意到 `BenchmarkFib40` 一共才运行 2 次。为了更精确的结果，我们可以通过 `-benchtime` 标志指定运行时间，从而使它运行更多次。

```
$ go test -bench=Fib40 -benchtime=20s
BenchmarkFib40-4     30            838675800 ns/op
```

B 类型

`B` 是传递给基准测试函数的一种类型，它用于管理基准测试的计时行为，并指示应该迭代地运行测试多少次。

当基准测试函数返回时，或者当基准测试函数调用

`FailNow`、`Fatal`、`Fatalf`、`SkipNow`、`Skip`、`Skipf` 中的任何一个方法时，则宣告测

测试函数结束。至于其他报告方法，比如 `Log` 和 `Error` 的变种，则可以在其他 `goroutine` 中同时进行调用。

跟单元测试一样，基准测试会在执行的过程中积累日志，并在测试完毕时将日志转储到标准错误。但跟单元测试不一样的是，为了避免基准测试的结果受到日志打印操作的影响，基准测试总是会把日志打印出来。

B 类型中的报告方法使用方式和 T 类型是一样的，一般来说，基准测试中也不需要使用，毕竟主要是测性能。这里我们对 B 类型中其他的一些方法进行讲解。

计时方法

有三个方法用于计时：

1. `StartTimer`：开始对测试进行计时。该方法会在基准测试开始时自动被调用，我们也可以在调用 `StopTimer` 之后恢复计时；
2. `StopTimer`：停止对测试进行计时。当你需要执行一些复杂的初始化操作，并且你不想对这些操作进行测量时，就可以使用这个方法暂时地停止计时；
3. `ResetTimer`：对已经逝去的基准测试时间以及内存分配计数器进行清零。对于正在运行中的计时器，这个方法不会产生任何效果。本节开头有使用示例。

并行执行

通过 `RunParallel` 方法能够并行地执行给定的基准测试。`RunParallel` 会创建出多个 `goroutine`，并将 `b.N` 分配给这些 `goroutine` 执行，其中 `goroutine` 数量的默认值为 `GOMAXPROCS`。用户如果想要增加非 CPU 受限（non-CPU-bound）基准测试的并行性，那么可以在 `RunParallel` 之前调用 `SetParallelism`（如 `SetParallelism(2)`，则 `goroutine` 数量为 `2*GOMAXPROCS`）。`RunParallel` 通常会与 `-cpu` 标志一同使用。

`body` 函数将在每个 `goroutine` 中执行，这个函数需要设置所有 `goroutine` 本地的状态，并迭代直到 `pb.Next` 返回 `false` 值为止。因为 `StartTimer`、`StopTime` 和 `ResetTimer` 这三个方法都带有全局作用，所以 `body` 函数不应该调用这些方法；除此之外，`body` 函数也不应该调用 `Run` 方法。

具体的使用示例，在本节开头已经提供！

内存统计

`ReportAllocs` 方法用于打开当前基准测试的内存统计功能，与 `go test` 使用 `-benchmem` 标志类似，但只影响那些调用了该函数的基准测试。

测试示例：

```
func BenchmarkTmplExucte(b *testing.B) {
    b.ReportAllocs()
    templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
    b.RunParallel(func(pb *testing.PB) {
        // Each goroutine has its own bytes.Buffer.
        var buf bytes.Buffer
        for pb.Next() {
            // The loop body is executed b.N times total across all goroutines.
            buf.Reset()
            templ.Execute(&buf, "World")
        }
    })
}
```

测试结果类似这样：

```
BenchmarkTmplExucte-4      2000000      898 ns/op      368 B/op      9 allocs/op
```

基准测试结果

对上述结果中的每一项，你是否都清楚是什么意思呢？

- `2000000` : 基准测试的迭代总次数 `b.N`
- `898 ns/op` : 平均每次迭代所消耗的纳秒数
- `368 B/op` : 平均每次迭代内存所分配的字节数
- `9 allocs/op` : 平均每次迭代的内存分配次数

`testing` 包中的 `BenchmarkResult` 类型能为你提供帮助，它保存了基准测试的结果，定义如下：

```
type BenchmarkResult struct {
    N      int           // The number of iterations. 基准测试的迭代总次数，即 b.N
    T      time.Duration // The total time taken. 基准测试的总耗时
    Bytes  int64        // Bytes processed in one iteration. 一次迭代处理的字节数，通过 b.SetBytes 设置
    MemAllocs uint64       // The total number of memory allocations. 内存分配的总次数
    MemBytes uint64       // The total number of bytes allocated. 内存分配的总字节数
}
```

该类型还提供了每次迭代操作所消耗资源的计算方法，示例如下：

```
package main

import (
    "bytes"
    "fmt"
    "testing"
    "text/template"
)

func main() {
    benchmarkResult := testing.Benchmark(func(b *testing.B) {
        templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
        // RunParallel will create GOMAXPROCS goroutines
        // and distribute work among them.
        b.RunParallel(func(pb *testing.PB) {
            // Each goroutine has its own bytes.Buffer.
            var buf bytes.Buffer
            for pb.Next() {
                // The loop body is executed b.N times total across all goroutines.
                buf.Reset()
                templ.Execute(&buf, "World")
            }
        })
    })

    // fmt.Printf("%8d\t%10d ns/op\t%10d B/op\t%10d allocs/op\n", benchmarkResult.N, benchmarkResult.NsPerOp(),
    benchmarkResult.AllocatedBytesPerOp(), benchmarkResult.AllocsPerOp())
    fmt.Printf("%s\t%s\n", benchmarkResult.String(), benchmarkResult.MemString())
}
```

testing - 子测试与子基准测试

从 Go 1.7 开始，引入了一个新特性：子测试（**subtests**）与子基准测试（**sub-benchmarks**），它意味着您现在可以拥有嵌套测试，这对于过滤执行特定测试用例非常有用。

T 和 B 的 `Run` 方法允许定义子单元测试和子基准测试，而不必为它们单独定义函数。这便于创建基于 **Table-Driven** 的基准测试和层级测试。它还提供了一种共享通用 `setup` 和 `tear-down` 代码的方法：

```
func TestFoo(t *testing.T) {
    // <setup code>
    t.Run("A=1", func(t *testing.T) { ... })
    t.Run("A=2", func(t *testing.T) { ... })
    t.Run("B=1", func(t *testing.T) { ... })
    // <tear-down code>
}
```

每个子测试和子基准测试都有一个唯一的名称：由顶层测试的名称与传递给 `Run` 的名称组成，以斜杠分隔，并具有可选的尾随序列号，用于消除歧义。

命令行标志 `-run` 和 `-bench` 的参数是非固定的正则表达式，用于匹配测试名称。对于由斜杠分隔的测试名称，例如子测试的名称，它名称本身即可作为参数，依次匹配由斜杠分隔的每部分名称。因为参数是非固定的，一个空的表达式匹配任何字符串，所以下述例子中的“匹配”意味着“顶层/子测试名称包含有”：

```
go test -run '' # 执行所有测试。
go test -run Foo # 执行匹配 "Foo" 的顶层测试，例如 "TestFooBar"。
go test -run Foo/A= # 对于匹配 "Foo" 的顶层测试，执行其匹配 "A=" 的子测试。
go test -run /A=1 # 执行所有匹配 "A=1" 的子测试。
```

子测试也可用于程序并行控制。只有子测试全部执行完毕后，父测试才会完成。在下述例子中，所有子测试之间并行运行，此处的“并行”只限于这些子测试之间，并不影响定义在其他顶层测试中的子测试：

```
func TestGroupedParallel(t *testing.T) {
    for _, tc := range tests {
        tc := tc // capture range variable
        t.Run(tc.Name, func(t *testing.T) {
            t.Parallel()
            ...
        })
    }
}
```

在所有子测试并行运行完毕之前，`Run` 方法不会返回。下述例子提供了一种方法，用于在子测试并行运行完毕后清理资源：

```
func TestTeardownParallel(t *testing.T) {
    // This Run will not return until the parallel tests finish.
    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })
    // <tear-down code>
}
```

testing - 运行并验证示例

`testing` 包除了测试，还提供了运行并验证示例的功能。示例，一方面是文档的效果，是关于某个功能的使用例子；另一方面，可以被当做测试运行。

一个示例的例子如下：

```
func ExampleHello() {
    fmt.Println("Hello")
    // Output: Hello
}
```

如果 `Output: Hello` 改为：`Output: hello`，运行测试会失败，提示：

```
got:
Hello
want:
hello
```

一个示例函数以 `Example` 开头，如果示例函数包含以“`Output:`”开头的行注释，在运行测试时，`go` 会将示例函数的输出和“`Output:`”注释中的值做比较，就如上面的例子。

有时候，输出顺序可能不确定，比如循环输出 `map` 的值，那么可以使用“`Unordered output:`”开头的注释。

如果示例函数没有上述输出注释，该示例函数只会被编译而不会被运行。

命名约定

`Go` 语言通过大量的命名约定来简化工具的复杂度，规范代码的风格。对示例函数的命名有如下约定：

- 包级别的示例函数，直接命名为 `func Example() { ... }`
- 函数 `F` 的示例，命名为 `func ExampleF() { ... }`
- 类型 `T` 的示例，命名为 `func ExampleT() { ... }`
- 类型 `T` 上的方法 `M` 的示例，命名为 `func ExampleT_M() { ... }`

有时，我们想要给包 / 类型 / 函数 / 方法 提供多个示例，可以通过在示例函数名称后附加一个不同的后缀来实现，但这种后缀必须以小写字母开头，如：

```
func Example_suffix() { ... }
func ExampleF_suffix() { ... }
func ExampleT_suffix() { ... }
func ExampleT_M_suffix() { ... }
```

通常，示例代码会放在单独的示例文件中，命名为 `example_test.go`。可以查看 `io` 包中的 `example_test.go` 了解示例的编写。

实现原理

本节开头提到了示例的两个作用，它们分别是由 `godoc` 和 `go test` 这两个命令实现的。

在执行 `go test` 时，会运行示例。具体的实现原理，可以通过阅读 `go test` 命令源码和 `testing` 包中的 `example.go` 文件了解。

testing - 其他功能

TestMain

在写测试时，有时需要在测试之前或之后进行额外的设置（**setup**）或拆卸（**teardown**）；有时，测试还需要控制在主线程上运行的代码。为了支持这些需求，`testing` 包提供了 `TestMain` 函数：

```
func TestMain(m *testing.M)
```

如果测试文件中包含该函数，那么生成的测试将调用 `TestMain(m)`，而不是直接运行测试。`TestMain` 运行在主 `goroutine` 中，可以在调用 `m.Run` 前后做任何设置和拆卸。注意，在 `TestMain` 函数的最后，应该使用 `m.Run` 的返回值作为参数去调用 `os.Exit`。

另外，在调用 `TestMain` 时，`flag.Parse` 并没有被调用。所以，如果 `TestMain` 依赖于 `command-line` 标志（包括 `testing` 包的标志），则应该显式地调用 `flag.Parse`。注意，这里的依赖是指，若 `TestMain` 函数内需要用到 `command-line` 标志，则必须显式地调用 `flag.Parse`，否则不需要，因为 `m.Run` 中调用 `flag.Parse`。

一个包含 `TestMain` 的例子如下：

```
package mytestmain

import (
    "flag"
    "fmt"
    "os"
    "testing"
)

var db struct {
    Dns string
}

func TestMain(m *testing.M) {
    db.Dns = os.Getenv("DATABASE_DNS")
    if db.Dns == "" {
        db.Dns = "root:123456@tcp(localhost:3306)/?charset=utf8&parseTime=True&loc=Local"
    }

    flag.Parse()
    exitCode := m.Run()

    db.Dns = ""

    // 退出
    os.Exit(exitCode)
}

func TestDatabase(t *testing.T) {
    fmt.Println(db.Dns)
}
```

对 `m.Run` 感兴趣的可以阅读源码，了解其原理。

Test Coverage

测试覆盖率，这里讨论的是基于代码的测试覆盖率。

Go 从 1.2 开始，引入了对测试覆盖率的支持，使用的是与 `cover` 相关的工具（`go test -cover`、`go tool cover`）。虽然 `testing` 包提供了 `cover` 相关函数，不过它们是给 `cover` 的工具使用的。

关于测试覆盖率的更多信息，可以参考官方的博文：[The cover story](#)

httpstest - HTTP 测试辅助工具

由于 Go 标准库的强大支持，Go 可以很容易的进行 Web 开发。为此，Go 标准库专门提供了 `net/http/httpstest` 包专门用于进行 http Web 开发测试。

本节我们通过一个社区帖子的增删改查的例子来学习该包。

简单的 Web 应用

我们首先构建一个简单的 Web 应用。

为了简单起见，数据保存在内存，并且没有考虑并发问题。

```
// 保存 Topic, 没有考虑并发问题
var TopicCache = make([]*Topic, 0, 16)

type Topic struct {
    Id      int    `json:"id"`
    Title   string `json:"title"`
    Content string `json:"content"`
    CreatedAt time.Time `json:"created_at"`
}
```

对于 `Topic` 的增删改查代码很简单，可以查看[完整代码](#)。

接下来，是通过 `net/http` 包来实现一个 Web 应用。

```
func main() {
    http.HandleFunc("/topic/", handleRequest)
    http.ListenAndServe(":2017", nil)
}
...
```

`/topic/` 开头的请求都交由 `handleRequest` 处理，它根据不同的 `Method` 执行相应的增删改查，详细代码可以查看 [server.go](#)。

准备好 Web 应用后，我们启动它。

```
go run server.go data.go
```

通过 `curl` 进行简单的测试：

```
增: curl -i -X POST http://localhost:2017/topic/ -H 'content-type: application/json' -d '{"title":"The Go Standard Library","content":"It contains many packages."}'
```

```
查: curl -i -X GET http://localhost:2017/topic/1
```

```
改: curl -i -X PUT http://localhost:2017/topic/1 -H 'content-type: application/json' -d '{"title":"The Go Standard Library By Example","content":"It contains many packages, enjoying it."}'
```

```
删: curl -i -X DELETE http://localhost:2017/topic/1
```

通过 httpstest 进行测试

上面，我们通过 `curl` 对我们的 Web 应用的接口进行了测试。现在，我们通过 `net/http/httpstest` 包进行测试。

我们先测试创建帖子，也就是测试 `handlePost` 函数。

```
func TestHandlePost(t *testing.T) {
    mux := http.NewServeMux()
    mux.HandleFunc("/topic/", handleRequest)

    reader := strings.NewReader(`{"title":"The Go Standard Library","content":"It contains many packages."}`)
    r, _ := http.NewRequest(http.MethodPost, "/topic/", reader)

    w := httpstest.NewRecorder()

    mux.ServeHTTP(w, r)

    resp := w.Result()
    if resp.StatusCode != http.StatusOK {
        t.Errorf("Response code is %v", resp.StatusCode)
    }
}
```

首先跟待测试代码一样，配置上路由，对 `/topic/` 的请求都交由 `handleRequest` 处理。

```
mux := http.NewServeMux()
mux.HandleFunc("/topic/", handleRequest)
```

因为 `handlePost` 的函数签名是 `func handlePost(w http.ResponseWriter, r *http.Request) error`，为了测试它，我们必须创建 `http.ResponseWriter` 和 `http.Request` 的实例。

接下来的代码就是创建一个 `http.Request` 实例 和一个 `http.ResponseWriter` 的实例。这里的关键是，通过 `httpstest.NewRecorder()` 可以获得 `httpstest.ResponseRecorder` 结构，而此结构实现了 `http.ResponseWriter` 接口。

```
reader := strings.NewReader(`{"title":"The Go Standard Library","content":"It contains many packages."}`)
r, _ := http.NewRequest(http.MethodPost, "/topic/", reader)

w := httpstest.NewRecorder()
```

准备好之后，可以测试目标函数了。这里，我们没有直接调用 `handlePost(w, r)`，而是调用 `mux.ServeHTTP(w, r)`，实际上这里直接调用 `handlePost(w, r)` 也是可以的，但调用 `mux.ServeHTTP(w, r)` 会更完整地测试整个流程。`mux.ServeHTTP(w, r)` 最终也会调用到 `handlePost(w, r)`。

最后，通过 `go test -v` 运行测试。

查、改和删帖子的接口测试代码类似，比如，`handleGet` 的测试代码如下：

```
func TestHandleGet(t *testing.T) {
    mux := http.NewServeMux()
    mux.HandleFunc("/topic/", handleRequest)

    r, _ := http.NewRequest(http.MethodGet, "/topic/1", nil)

    w := httpstest.NewRecorder()
```

```
    mux.ServeHTTP(w, r)

    resp := w.Result()
    if resp.StatusCode != http.StatusOK {
        t.Errorf("Response code is %v", resp.StatusCode)
    }

    topic := new(Topic)
    json.Unmarshal(w.Body.Bytes(), topic)
    if topic.Id != 1 {
        t.Errorf("Cannot get topic")
    }
}
```

注意：因为数据没有落地存储，为了保证后面的测试正常，请将 `TestHandlePost` 放在最前面。

测试代码改进

细心的朋友应该会发现，上面的测试代码有重复，比如：

```
mux := http.NewServeMux()
mux.HandleFunc("/topic/", handleRequest)
```

以及：

```
w := httptest.NewRecorder()
```

这正好是前面学习的 `setup` 可以做的事情，因此可以使用 `TestMain` 来做重构。

```
var w *httptest.ResponseRecorder

func TestMain(m *testing.M) {
    http.DefaultServeMux.HandleFunc("/topic/", handleRequest)

    w = httptest.NewRecorder()

    os.Exit(m.Run())
}
```

总结

除了标准库提供的测试包，还有很多优秀的第三方测试包，比如 [testify](#) 可有助于 `testing` 进行单元测试，进行 `mock` 等，有机会会写文章介绍它的使用；再比如，[ginkgo](#) 有助于集成测试等等。

参考阅读：[Testing Go at Stream](#)

进程、线程与 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。

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，如果子进程尚未调用 `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 `syscall.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;

`syscall.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 中读取数据，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。

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 (effective 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 变更为此专用 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 `fchdir`（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，这是文件系统的真正根目录。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。

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 (`CLONEVM`)、文件系统资源 (`CLONEFS`)、文件描述符 (`CLONEFILES`) 和信号处理程序 (`CLONESIGHAND`)。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。

进程间通信

进程之间用来相互通讯和同步

应用构建 与 debug

本章我们来研究应用程序构建相关的标准库，包括用于命令行参数解析的 `flag` 包，简单的日志记录包 `log`（以及 `syslog`），公共变量标准接口 `expvar` 包，以及运行时的调试工具 `runtime/debug` 包。

flag - 命令行参数解析

在写命令行程序（工具、server）时，对命令参数进行解析是常见的需求。各种语言一般都会提供解析命令行参数的方法或库，以方便程序员使用。如果命令参数纯粹自己写代码解析，对于比较复杂的，还是挺费劲的。在 go 标准库中提供了一个包：`flag`，方便进行命令行解析。

注：区分几个概念

1. 命令行参数（或参数）：是指运行程序提供的参数
2. 已定义命令行参数：是指程序中通过 `flag.Xxx` 等这种形式定义了参数
3. 非 flag（non-flag）命令行参数（或保留的命令行参数）：后文解释

使用示例

我们以 nginx 为例，执行 `nginx -h`，输出如下：

```
nginx version: nginx/1.10.0
Usage: nginx [-?hvVtTq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
  -?, -h      : this help
  -v          : show version and exit
  -V          : show version and configure options then exit
  -t          : test configuration and exit
  -T          : test configuration, dump it and exit
  -q          : suppress non-error messages during configuration testing
  -s signal   : send signal to a master process: stop, quit, reopen, reload
  -p prefix   : set prefix path (default: /usr/local/nginx/)
  -c filename : set configuration file (default: conf/nginx.conf)
  -g directives : set global directives out of configuration file
```

我们通过 `flag` 实现类似 nginx 的这个输出，创建文件 `nginx.go`，内容如下：

```
package main

import (
    "flag"
    "fmt"
    "os"
)

// 实际中应该用更好的变量名
var (
    h bool

    v, V bool
    t, T bool
    q *bool

    s string
    p string
    c string
    g string
)

func init() {
    flag.BoolVar(&h, "h", false, "this help")
```



```

flag.BoolVar(&v, "v", false, "show version and exit")
flag.BoolVar(&V, "V", false, "show version and configure options then exit")

flag.BoolVar(&t, "t", false, "test configuration and exit")
flag.BoolVar(&T, "T", false, "test configuration, dump it and exit")

// 另一种绑定方式
q = flag.Bool("q", false, "suppress non-error messages during configuration testing")

// 注意 `signal`。默认是 -s string, 有了 `signal` 之后, 变为 -s signal
flag.StringVar(&s, "s", "", "send `signal` to a master process: stop, quit, reopen, reload")
flag.StringVar(&p, "p", "/usr/local/nginx/", "set `prefix` path")
flag.StringVar(&c, "c", "conf/nginx.conf", "set configuration `file`")
flag.StringVar(&g, "g", "conf/nginx.conf", "set global `directives` out of configuration file")

// 改变默认的 Usage
flag.Usage = usage
}

func main() {
    flag.Parse()

    if h {
        flag.Usage()
    }
}

func usage() {
    fmt.Fprintf(os.Stderr, `nginx version: nginx/1.10.0
Usage: nginx [-hvTtq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
`)
    flag.PrintDefaults()
}

```

执行: `go run nginx.go -h`, (或 `go build -o nginx && ./nginx -h`) 输出如下:

```

nginx version: nginx/1.10.0
Usage: nginx [-hvTtq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
-T    test configuration, dump it and exit
-V    show version and configure options then exit
-c file
     set configuration file (default "conf/nginx.conf")
-g directives
     set global directives out of configuration file (default "conf/nginx.conf")
-h    this help
-p prefix
     set prefix path (default "/usr/local/nginx/")
-q    suppress non-error messages during configuration testing
-s signal
     send signal to a master process: stop, quit, reopen, reload
-t    test configuration and exit
-v    show version and exit

```

仔细理解以上例子, 如果有不理解的, 看完下文的讲解再回过头来看。

flag 包概述

`flag` 包实现了命令行参数的解析。

定义 flags 有两种方式

1) `flag.Xxx()`, 其中 `Xxx` 可以是 `Int`、`String` 等; 返回一个相应类型的指针, 如:

```
var ip = flag.Int("flagname", 1234, "help message for flagname")
```

2) `flag.XxxVar()`, 将 `flag` 绑定到一个变量上, 如:

```
var flagvar int
flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
```

自定义 Value

另外, 还可以创建自定义 `flag`, 只要实现 `flag.Value` 接口即可 (要求 `receiver` 是指针), 这时候可以通过如下方式定义该 `flag`:

```
flag.Var(&flagVal, "name", "help message for flagname")
```

例如, 解析我喜欢的编程语言, 我们希望直接解析到 `slice` 中, 我们可以定义如下 `Value`:

```
type sliceValue []string

func newSliceValue(vals []string, p *[]string) *sliceValue {
    *p = vals
    return (*sliceValue)(p)
}

func (s *sliceValue) Set(val string) error {
    *s = sliceValue(strings.Split(val, ","))
    return nil
}

func (s *sliceValue) Get() interface{} { return []string(*s) }

func (s *sliceValue) String() string { return strings.Join([]string(*s), ",") }
```

之后可以这么使用:

```
var languages []string
flag.Var(newSliceValue([]string{}, &languages), "slice", "I like programming `languages`")
```

这样通过 `-slice "go, php"` 这样的形式传递参数, `languages` 得到的就是 `[go, php]`。

`flag` 中对 `Duration` 这种非基本类型的支持, 使用的就是类似这样的方式。

解析 flag

在所有的 `flag` 定义完成之后, 可以通过调用 `flag.Parse()` 进行解析。

命令行 `flag` 的语法有如下三种形式：

```
-flag // 只支持 bool 类型
-flag=x
-flag x // 只支持非 bool 类型
```

其中第三种形式只能用于非 `bool` 类型的 `flag`，原因是：如果支持，那么对于这样的命令 `cmd -x *`，如果有一个文件名字是 `0` 或 `false` 等，则命令的原意会改变（之所以这样，是因为 `bool` 类型支持 `-flag` 这种形式，如果 `bool` 类型不支持 `-flag` 这种形式，则 `bool` 类型可以和其他类型一样处理。也正因为这样，`Parse()` 中，对 `bool` 类型进行了特殊处理）。默认的，提供了 `-flag`，则对应的值为 `true`，否则为 `flag.Bool/BoolVar` 中指定的默认值；如果希望显示设置为 `false` 则使用 `-flag=false`。

`int` 类型可以是十进制、十六进制、八进制甚至是负数；`bool` 类型可以是 `1, 0, t, f, true, false, TRUE, FALSE, True, False`。`Duration` 可以接受任何 `time.ParseDuration` 能解析的类型。

类型和函数

在看类型和函数之前，先看一下变量。

ErrHelp: 该错误类型用于当命令行指定了 `-help` 参数但没有定义时。

Usage: 这是一个函数，用于输出所有定义了命令行参数和帮助信息 (`usage message`)。一般，当命令行参数解析出错时，该函数会被调用。我们可以指定自己的 `Usage` 函数，即：`flag.Usage = func() {}`

函数

`go` 标准库中，经常这么做：

```
定义了一个类型，提供了很多方法；为了方便使用，会实例化一个该类型的实例（通用），这样便可以直接使用该实例调用方法。比如：encoding/base64 中提供了 StdEncoding 和 URLEncoding 实例，使用时：
base64.StdEncoding.Encode()
```

在 `flag` 包使用了有类似的方法，比如 `CommandLine` 实例，只不过 `flag` 进行了进一步封装：将 `FlagSet` 的方法都重新定义了一遍，也就是提供了一序列函数，而函数中只是简单的调用已经实例化好了的 `FlagSet` 实例：`CommandLine` 的方法。这样，使用者是这么调用：`flag.Parse()` 而不是 `flag.CommandLine.Parse()`。（`Go 1.2` 起，将 `CommandLine` 导出，之前是非导出的）

这里不详细介绍各个函数，在类型方法中介绍。

类型（数据结构）

1) ErrorHandler

```
type ErrorHandler int
```

该类型定义了了在参数解析出错时错误处理方式。定义了三个该类型的常量：

```
const (
    ContinueOnError ErrorHandler = iota
    ExitOnError
    PanicOnError
)
```

三个常量在源码的 `FlagSet` 的方法 `parseOne()` 中使用了。

2) Flag

```
// A Flag represents the state of a flag.
type Flag struct {
    Name      string // name as it appears on command line
    Usage     string // help message
    Value     Value  // value as set
    DefValue  string // default value (as text); for usage message
}
```

Flag 类型代表一个 flag 的状态。

比如，对于命令：`./nginx -c /etc/nginx.conf`，相应代码是：

```
flag.StringVar(&c, "c", "conf/nginx.conf", "set configuration file")
```

则该 Flag 实例（可以通过 `flag.Lookup("c")` 获得）相应各个字段的值为：

```
&Flag{
    Name: c,
    Usage: set configuration file,
    Value: /etc/nginx.conf,
    DefValue: conf/nginx.conf,
}
```

3) FlagSet

```
// A FlagSet represents a set of defined flags.
type FlagSet struct {
    // Usage is the function called when an error occurs while parsing flags.
    // The field is a function (not a method) that may be changed to point to
    // a custom error handler.
    Usage func()

    name string // FlagSet 的名字。CommandLine 给的是 os.Args[0]
    parsed bool // 是否执行过 Parse()
    actual map[string]*Flag // 存放实际传递了的参数（即命令行参数）
    formal map[string]*Flag // 存放所有已定义命令行参数
    args []string // arguments after flags // 开始存放所有参数，最后保留 非 flag (non-flag) 参数
    exitOnError bool // does the program exit if there's an error?
    errorHandling ErrorHandling // 当解析出错时，处理错误的方式
    output io.Writer // nil means stderr; use out() accessor
}
```

4) Value 接口

```
// Value is the interface to the dynamic value stored in a flag.
// (The default value is represented as a string.)
type Value interface {
    String() string
    Set(string) error
}
```

所有参数类型需要实现 Value 接口，flag 包中，为 int、float、bool 等实现了该接口。借助该接口，我们可以自定义 flag。
（上文已经给了具体的例子）

主要类型的方法（包括类型实例化）

flag 包中主要是 FlagSet 类型。

实例化方式

`NewFlagSet()` 用于实例化 FlagSet。预定义的 FlagSet 实例 `CommandLine` 的定义方式：

```
// The default set of command-line flags, parsed from os.Args.
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)
```

可见，默认的 FlagSet 实例在解析出错时会退出程序。

由于 FlagSet 中的字段没有 export，其他方式获得 FlagSet 实例后，比如：FlagSet{} 或 new(FlagSet)，应该调用 Init() 方法，以初始化 name 和 errorHandler，否则 name 为空，errorHandler 为 ContinueOnError。

定义 flag 参数的方法

这一序列的方法都有两种形式，在一开始已经说了两种方式的区别。这些方法用于定义某一类型的 flag 参数。

解析参数（Parse）

```
func (f *FlagSet) Parse(arguments []string) error
```

从参数列表中解析定义的 flag。方法参数 arguments 不包括命令名，即应该是 os.Args[1:]。事实上，`flag.Parse()` 函数就是这么做的：

```
// Parse parses the command-line flags from os.Args[1:]. Must be called
// after all flags are defined and before flags are accessed by the program.
func Parse() {
    // Ignore errors; CommandLine is set for ExitOnError.
    CommandLine.Parse(os.Args[1:])
}
```

该方法应该在 flag 参数定义后而具体参数值被访问前调用。

如果提供了 `-help` 参数（命令中给了）但没有定义（代码中没有），该方法返回 `ErrHelp` 错误。默认的 `CommandLine`，在 Parse 出错时会退出程序（ExitOnError）。

为了更深入的理解，我们看一下 `Parse(arguments []string)` 的源码：

```
func (f *FlagSet) Parse(arguments []string) error {
    f.parsed = true
    f.args = arguments
    for {
        seen, err := f.parseOne()
        if seen {
            continue
        }
        if err == nil {
            break
        }
        switch f.errorHandling {
        case ContinueOnError:
            return err
        }
    }
}
```

```

        case ExitOnError:
            os.Exit(2)
        case PanicOnError:
            panic(err)
    }
}
return nil
}

```

真正解析参数的方法是非导出方法 `parseOne`。

结合 `parseOne` 方法，我们来解释 `non-flag` 以及包文档中的这句话：

Flag parsing stops just before the first non-flag argument (“-” is a non-flag argument) or after the terminator “-”.

我们需要了解解析什么时候停止。

根据 `Parse()` 中 `for` 循环终止的条件（不考虑解析出错），我们知道，当 `parseOne` 返回 `false, nil` 时，`Parse` 解析终止。正常解析完成我们不考虑。看一下 `parseOne` 的源码发现，有两处会返回 `false, nil`。

1) 第一个 non-flag 参数

```

s := f.args[0]
if len(s) == 0 || s[0] != '-' || len(s) == 1 {
    return false, nil
}

```

也就是，当遇到单独的一个“-”或不是“-”开始时，会停止解析。比如：

```
./nginx - -c 或 ./nginx build -c
```

这两种情况，`-c` 都不会被正确解析。像该例子中的“-”或 `build`（以及之后的参数），我们称之为 `non-flag` 参数。

2) 两个连续的“-”

```

if s[1] == '-' {
    num_minuses++
    if len(s) == 2 { // "-" terminates the flags
        f.args = f.args[1:]
        return false, nil
    }
}
}

```

也就是，当遇到连续的两个“-”时，解析停止。

说明：这里说的“-”和“-”，位置和“-c”这种的一样。也就是说，下面这种情况并不是这里说的：

```
./nginx -c -
```

这里的“-”会被当成是 `c` 的值

`parseOne` 方法中接下来是处理 `-flag=x` 这种形式，然后是 `-flag` 这种形式（`bool` 类型）（这里对 `bool` 进行了特殊处理），接着是 `-flag x` 这种形式，最后，将解析成功的 `Flag` 实例存入 `FlagSet` 的 `actual map` 中。

另外，在 `parseOne` 中有这么一句：

```
f. args = f. args[1:]
```

也就是说，每执行成功一次 `parseOne`，`f.args` 会少一个。所以，`FlagSet` 中的 `args` 最后留下来的就是所有 `non-flag` 参数。

Arg(i int) 和 Args()、NArg()、NFlag()

`Arg(i int)` 和 `Args()` 这两个方法就是获取 `non-flag` 参数的；`NArg()` 获得 `non-flag` 的个数；`NFlag()` 获得 `FlagSet` 中 `actual` 长度（即被设置了的参数个数）。

Visit/VisitAll

这两个函数分别用于访问 `FlatSet` 的 `actual` 和 `formal` 中的 `Flag`，而具体的访问方式由调用者决定。

PrintDefaults()

打印所有已定义参数的默认值（调用 `VisitAll` 实现），默认输出到标准错误，除非指定了 `FlagSet` 的 `output`（通过 `SetOutput()` 设置）

Set(name, value string)

设置某个 `flag` 的值（通过 `name` 查找到对应的 `Flag`）

总结

使用建议：虽然上面讲了那么多，一般来说，我们只简单的定义 `flag`，然后 `parse`，就如同开始的例子一样。

如果项目需要复杂或更高级的命令行解析方式，可以使用 <https://github.com/urfave/cli> 或者 <https://github.com/spf13/cobra> 这两个强大的库。

log - 日志记录

expvar - 公共变量的标准化接口

expvar 挺简单的，然而，它也是很有用的。但不幸的是，貌似了解它的人不多。来自 godoc.org 的数据表明，没有多少人知道这个包。截止目前（2017-6-18），该包被公开的项目 import 2207 次，相比较而言，连 image 包都被 import 3491 次之多。

如果你看到了这里，希望以后你的项目中能使用上 expvar 这个包。

包简介

包 expvar 为公共变量提供了一个标准化的接口，如服务器中的操作计数器。它以 JSON 格式通过 `/debug/vars` 接口以 HTTP 的方式公开这些公共变量。

设置或修改这些公共变量的操作是原子的。

除了为程序增加 HTTP handler，此包还注册以下变量：

```
cmdline  os.Args
memstats runtime.Memstats
```

导入该包有时只是为注册其 HTTP handler 和上述变量。要以这种方式使用，请将此包通过如下形式引入到程序中：

```
import _ "expvar"
```

例子

在继续介绍此包的详细信息之前，我们演示使用 expvar 包可以做什么。以下代码创建一个在监听 8080 端口的 HTTP 服务器。每个请求到达 handler() 后，在向访问者发送响应消息之前增加计数器。

```
package main

import (
    "expvar"
    "fmt"
    "net/http"
)

var visits = expvar.NewInt("visits")

func handler(w http.ResponseWriter, r *http.Request) {
    visits.Add(1)
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

导入 expvar 包后，它将为 `http.DefaultServeMux` 上的 PATH `/debug/vars` 注册一个处理函数。此处理程序返回已在 expvar 包中注册的所有公共变量。运行代码并访问 `http://localhost:8080/debug/vars`，您将看到如下所示的内容（输出被截断以增加可读性）：

```

{
  "cmdline": [
    "/var/folders/qv/2jztyc09357ddtxn_bvgh8j0000gn/T/go-build146580631/command-line-arguments/_obj/exe/test"
  ],
  "memstats": {
    "Alloc": 414432,
    "TotalAlloc": 414432,
    "Sys": 3084288,
    "Lookups": 13,
    "Mallocs": 5111,
    "Frees": 147,
    "HeapAlloc": 414432,
    "HeapSys": 1703936,
    "HeapIdle": 835584,
    "HeapInuse": 868352,
    "HeapReleased": 0,
    "HeapObjects": 4964,
    "StackInuse": 393216,
    "StackSys": 393216,
    "MSpanInuse": 15504,
    "MSpanSys": 16384,
    "MCacheInuse": 4800,
    "MCacheSys": 16384,
    "BuckHashSys": 2426,
    "GCSys": 137216,
    "OtherSys": 814726,
    "NextGC": 4473924,
    "LastGC": 0,
    "PauseTotalNs": 0,
    "PauseNs": [
      0,
      0,
    ],
    "PauseEnd": [
      0,
      0,
    ],
    "GCCPUFraction": 0,
    "EnableGC": true,
    "DebugGC": false,
    "BySize": [
      {
        "Size": 16640,
        "Mallocs": 0,
        "Frees": 0
      },
      {
        "Size": 17664,
        "Mallocs": 0,
        "Frees": 0
      }
    ]
  },
  "visits": 0
}

```

信息真不少。这是因为默认情况下该包注册了 `os.Args` 和 `runtime.Memstats` 两个指标。因为我们还没有访问到增加 `visits` 的路径，所以它的值仍然为 `0`。现在通过访问 `http://localhost:8080/golang` 来增加计数器，然后返回。计数器不再为 `0`。

expvar.Publish 函数

expvar 包相当小且容易理解。它主要由两个部分组成。第一个是函数 `expvar.Publish(name string, v expvar.Var)`。该函数可用于在未导出的全局注册表中注册具有特定名称 (name) 的 v。以下代码段显示了具体实现。接下来的 3 个代码段是从 expvar 包的源代码中截取的。

先看下全局注册表:

```
var (
    mutex sync.RWMutex
    vars   = make(map[string]Var)
    varKeys []string // sorted
)
```

全局注册表实际就是一个 map: vars。

Publish 函数的实现:

```
// Publish declares a named exported variable. This should be called from a
// package's init function when it creates its Vars. If the name is already
// registered then this will log.Panic.
func Publish(name string, v Var) {
    mutex.Lock()
    defer mutex.Unlock()

    // Check if name has been taken already. If so, panic.
    if _, existing := vars[name]; existing {
        log.Panicln("Reuse of exported var name:", name)
    }

    // vars is the global registry. It is defined somewhere else in the
    // expvar package like this:
    //
    // vars = make(map[string]Var)
    vars[name] = v
    // 一方面, 该包中所有公共变量, 放在 vars 中, 同时, 通过 varKeys 保存了所有变量名, 并且按字母序排序,
    // 即实现了一个有序的、线程安全的 map
    varKeys = append(varKeys, name)
    sort.Strings(varKeys)
}
```

expvar 包内置的两个公共变量就是通过 Publish 注册的:

```
Publish("cmdline", Func(cmdline))
Publish("memstats", Func(memstats))
```

expvar.Var 接口

另一个重要的组成部分是 `expvar.Var` 接口。这个接口只有一个方法:

```
// Var is an abstract type for all exported variables.
type Var interface {
    // String returns a valid JSON value for the variable.
    // Types with String methods that do not return valid JSON
    // (such as time.Time) must not be used as a Var.
}
```

```
String() string  
}
```

所以你可以在有 `String() string` 方法的所有类型上调用 `Publish()` 函数，但需要注意的是，这里的 `String()` 要求返回的是一个有效的 JSON 字符串。

expvar.Int 类型

`expvar` 包提供了其他几个类型，它们实现了 `expvar.Var` 接口。其中一个就是 `expvar.Int`，我们已经在演示代码中通过 `expvar.NewInt("visits")` 方式使用它了，它会创建一个新的 `expvar.Int`，并使用 `expvar.Publish` 注册它，然后返回一个指向新创建的 `expvar.Int` 的指针。

```
func NewInt(name string) *Int {  
    v := new(Int)  
    Publish(name, v)  
    return v  
}
```

`expvar.Int` 包装一个 `int64`，并有两个函数 `Add(delta int64)` 和 `Set(value int64)`，它们以线程安全的方式（通过 `atomic` 包实现）修改包装的 `int64`。另外通过 `Value() int64` 函数获得包装的 `int64`。

```
type Int struct {  
    i int64  
}
```

其他类型

除了 `expvar.Int`，该包还提供了一些实现 `expvar.Var` 接口的其他类型：

- [expvar.Float](#)
- [expvar.String](#)
- [expvar.Map](#)
- [expvar.Func](#)

前两个类型包装了 `float64` 和 `string`。后两种类型需要稍微解释下。

`expvar.Map` 类型可用于使公共变量出现在某个名称空间下。可以这样用：

```
var stats = expvar.NewMap("http")  
var requests, requestsFailed expvar.Int  
  
func init() {  
    stats.Set("req_succ", &requests)  
    stats.Set("req_failed", &requestsFailed)  
}
```

这段代码使用名称空间 `http` 注册了两个指标 `req_succ` 和 `req_failed`。它将显示在 JSON 响应中，如下所示：

```
{  
  "http": {  
    "req_succ": 18,  
    "req_failed": 21  
  }  
}
```

```
}  
}
```

当要注册某个函数的执行结果到某个公共变量时，您可以使用 `expvar.Func`。假设您希望计算应用程序的正常运行时间，每次有人访问 `http://localhost:8080/debug/vars` 时，都必须重新计算此值。

```
var start = time.Now()  
  
func calculateUptime() interface {  
    return time.Since(start).String()  
}  
  
expvar.Publish("uptime", expvar.Func(calculateUptime))
```

实际上，内置的两个指标 `cmdline` 和 `memstats` 就是通过这种方式注册的。注意，函数签名有如下要求：没有参数，返回 `interface{}`

```
type Func func() interface{}
```

关于 Handler 函数

本文开始时提到，可以简单的导入 `expvar` 包，然后使用其副作用，导出路径 `/debug/vars`。然而，如果我们使用了一些框架，并非使用 `http.DefaultServeMux`，而是框架自己定义的 `Mux`，这时直接导入使用副作用可能不会生效。我们可以按照使用的框架，定义自己的路径，比如也叫 `/debug/vars`，然后，这对应的处理程序中，按如下的两种方式处理：

1) 将处理直接交给 `expvar.Handler`，比如：

```
handler := expvar.Handler()  
handler.ServeHTTP(w, req)
```

2) 自己遍历 `expvar` 中的公共变量，构造输出，甚至可以过滤 `expvar` 默认提供的 `cmdline` 和 `memstats`，我们看下 `expvarHandler` 的源码就明白了：（通过 `expvar.Do` 函数来遍历的）

```
func expvarHandler(w http.ResponseWriter, r *http.Request) {  
    w.Header().Set("Content-Type", "application/json; charset=utf-8")  
    fmt.Fprintf(w, "\n")  
    first := true  
    Do(func(kv KeyValue) {  
        if !first {  
            fmt.Fprintf(w, ",\n")  
        }  
        first = false  
        fmt.Fprintf(w, "%q: %s", kv.Key, kv.Value)  
    })  
    fmt.Fprintf(w, "\n}\n")  
}
```

Go 语言中文网 因为使用了 Echo 框架，使用第 1 种方式来处理的。

定义自己的 expvar.Var 类型

`expvar` 包提供了 `int`、`float` 和 `string` 这三种基本数据类型的 `expvar.Var` 实现，以及 `Func` 和 `Map`。有时，我们自己有一个复杂的类型，想要实现 `expvar.Var` 接口，怎么做呢？

从上面的介绍，应该很容易实现吧，如果您遇到了具体的需求，可以试试。

总结

综上所述，通过 `expvar` 包，可以非常容易的展示应用程序指标。建议在您的每个应用程序中使用它来展示一些指示应用程序运行状况的指标，通过它和其他的一些工具来监控应用程序。

runtime/debug - 运行时的调试工具

底层库介绍

unsafe — 非类型安全操作

unsafe — 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.field`”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
```

同步

同步

sync - 处理同步需求

sync/atomic - 原子操作

os/signal - 信号

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)  
}
```

只会打出一行“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 `goroutine`。

```
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 `goroutine` 开启后, 可以创建多个从 `goroutine`, 从 `goroutine` 获取锁后, 进入 `cond.Wait` 状态, 当主 `goroutine` 执行完任务后, 通过 `BroadCast` 广播信号。

处于 `cond.Wait` 状态的所有 `goroutine` 收到信号后将全部被唤醒并往下执行。需要注意的是, 从 `goroutine` 执行完任务后, 需要通过 `cond.L.Unlock` 释放锁, 否则其它被唤醒的 `goroutine` 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 `goroutine` 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。

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)`

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 `Ignore` 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。