# **1. io — 基本的 IO 接口**

io 包为 I/O 原语提供了基本的接口。它主要包装了这些原语的已有实现。

由于这些被接口包装的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发执行是安全的。

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。

## **1.1. Reader 接口**

Reader 接口的定义如下：

type Reader interface {

Read(p []byte) (n int, err error)

}

官方文档中关于该接口方法的说明：

Read 将 len(p) 个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)） 以及任何遇到的错误。即使 Read 返回的 n < len(p)，它也会在调用过程中占用 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](http://golang.org/ref/spec" \l "Interface_types" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)），我们知道 Reader 接口的方法集（[Method\_sets](http://golang.org/ref/spec" \l "Method_sets" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)）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](http://books.studygolang.com/The-Golang-Standard-Library-by-Example/code/src/chapter01/io/reader.go)

****小贴士****

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

## **1.2. Writer 接口**

Writer 接口的定义如下：

type Writer interface {

Write(p []byte) (n int, err error)

}

官方文档中关于该接口方法的说明：

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

同样的，所有实现了Write方法的类型都实现了 io.Writer 接口。

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

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

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

func Println(a ...interface{}) (n int, err error) {

return Fprintln(os.Stdout, a...)

}

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

关于 io.Writer 的更多说明，可以查看笔者之前写的博文[《以io.Writer为例看go中的interface{}》](http://blog.studygolang.com/2013/02/%e4%bb%a5io-writer%e4%b8%ba%e4%be%8b%e7%9c%8bgo%e4%b8%ad%e7%9a%84interface/" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)。

## **1.3. 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](https://golang.org/lib/godoc/analysis/help.html" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)。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。](http://docs.studygolang.com./" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)

* 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。

## **1.4. ReaderAt 和 WriterAt 接口**

****ReaderAt 接口****的定义如下：

type ReaderAt interface {

ReadAt(p []byte, off int64) (n int, err error)

}

官方文档中关于该接口方法的说明：

ReadAt 从基本输入源的偏移量 off 处开始，将 len(p) 个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)）以及任何遇到的错误。

当 ReadAt 返回的 n < len(p) 时，它就会返回一个 非nil 的错误来解释 为什么没有返回更多的字节。在这一点上，ReadAt 比 Read 更严格。

即使 ReadAt 返回的 n < len(p)，它也会在调用过程中使用 p 的全部作为暂存空间。若可读取的数据不到 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 <= n <= len(p)）以及任何遇到的引起写入提前停止的错误。若 WriteAt 返回的 n < len(p)，它就必须返回一个 非nil 的错误。

若 WriteAt 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）。

## **1.5. 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。”

## **1.6. 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

)

## **1.7. 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](https://github.com/polaris1119/The-Golang-Standard-Library-by-Example/issues/40" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 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！

## **1.8. 其他接口**

### **1.8.1. 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](http://books.studygolang.com/The-Golang-Standard-Library-by-Example/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](https://code.google.com/p/protobuf/" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)中的 Varints 读取，[ReadVarint](http://docs.studygolang.com/pkg/encoding/binary/" \l "ReadVarint" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 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](http://docs.studygolang.com/pkg/image/jpeg/" \l "Encode" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)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。

### **1.8.2. 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](http://docs.studygolang.com/pkg/regexp" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 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 8func 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。

### **1.8.3. 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”。

## **1.9. 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](http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/01.1.html) 会讲到。

## **1.10. 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 LimitReader Example"

reader := strings.NewReader(content)

limitReader := &io.LimitedReader{R: reader, N: 8}

for limitReader.N > 0 {

tmp := make([]byte, 2)

limitReader.Read(tmp)

fmt.Printf("%s", tmp)

}

输出：

This Is

可见，通过该类型可以达到 *只允许读取一定长度数据* 的目的。

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

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

## **1.11. 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。

## **1.12. 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，因此它不会将来自 Read 的 EOF 当做错误来报告。

若 dst 实现了 ReaderFrom 接口，其复制操作可通过调用 dst.ReadFrom(src) 实现。此外，若 src 实现了 WriterTo 接口，其复制操作可通过调用 src.WriteTo(dst) 实现。

代码：

io.Copy(os.Stdout, strings.NewReader("Go语言中文网"))

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

## **1.13. 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。

## **1.14. 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))。

## **1.15. MultiReader 和 MultiWriter 函数**

这两个函数的定义分别是：

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

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

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

type multiReader struct {

readers []Reader

}

type multiWriter struct {

writers []Writer

}

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

****MultiReader 的使用****：

readers := []io.Reader{

strings.NewReader("from strings reader"),

bytes.NewBufferString("from bytes buffer"),

}

reader := io.MultiReader(readers...)

data := make([]byte, 0, 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。

## **1.16. 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。

# **2. ioutil — 方便的IO操作函数集**

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

说明：这些函数使用都相对简单，一般就不举例子了。

## **2.1. NopCloser 函数**

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

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

rc, ok := body.(io.ReadCloser)

if !ok && body != nil {

rc = ioutil.NopCloser(body)

}

如果没有这个函数，我们得自己实现一个。当然，实现起来很简单，读者可以看看 [NopCloser](http://docscn.studygolang.com/src/io/ioutil/ioutil.go?s=5557:5598" \l "L145" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 的实现。

## **2.2. ReadAll 函数**

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

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

阅读该函数的源码发现，它是通过 bytes.Buffer 中的 [ReadFrom](http://docscn.studygolang.com/src/bytes/buffer.go?s=5385:5444" \l "L144" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 来实现读取所有数据的。该函数成功调用后会返回 err == nil 而不是 err == EOF。(成功读取完毕应该为 err == io.EOF，这里返回 nil 由于该函数成功期望 err == io.EOF，符合无错误不处理的理念)

## **2.3. ReadDir 函数**

笔试题：编写程序输出某目录下的所有文件（包括子目录）

是否见过这样的笔试题？

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

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

func main() {

dir := os.Args[1]

listAll(dir,0)

}

func listAll(path string, curHier int){

fileInfos, err := ioutil.ReadDir(path)

if err != nil{fmt.Println(err); return}

for \_, info := range fileInfos{

if info.IsDir(){

for tmpHier := curHier; tmpHier > 0; tmpHier--{

fmt.Printf("|\t")

}

fmt.Println(info.Name(),"\\")

listAll(path + "/" + info.Name(),curHier + 1)

}else{

for tmpHier := curHier; tmpHier > 0; tmpHier--{

fmt.Printf("|\t")

}

fmt.Println(info.Name())

}

}

}

## **2.4. ReadFile 和 WriteFile 函数**

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

ReadFile 函数的签名如下:

func ReadFile(filename string) ([]byte, error)

函数文档：

ReadFile 从 filename 指定的文件中读取数据并返回文件的内容。成功的调用返回的err 为 nil 而非 EOF。因为本函数定义为读取整个文件，它不会将读取返回的 EOF 视为应报告的错误。(同 ReadAll )

WriteFile 函数的签名如下：

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

函数文档:

WriteFile 将data写入filename文件中，当文件不存在时会根据perm指定的权限进行创建一个,文件存在时会先清空文件内容。对于 perm 参数，我们一般可以指定为：0666，具体含义 os 包中讲解。

****小提示****

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

## **2.5. TempDir 和 TempFile 函数**

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

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

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

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

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

参数和 ioutil.TempDir 参数含义类似。

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

defer func() {

f.Close()

os.Remove(f.Name())

}()

## **2.6. Discard 变量**

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

devNull 在实现 io.Writer 接口时，只是简单的返回（标准库文件：[src/pkg/io/ioutil.go](http://docscn.studygolang.com/pkg/io/ioutil/" \l "pkg-variables" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank))。

func (devNull) Write(p []byte) (int, error) {

return len(p), nil

}

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

# **3. fmt — 格式化IO**

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

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

以下例子中用到的类型或变量定义：

type Website struct {

Name string

}

// 定义结构体变量

var site = Website{Name:"studygolang"}

## **3.1. Printing**

### **3.1.1. 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", 11111111) //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", "wqdew") //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

}

### **3.1.2. 占位符**

****普通占位符****

占位符 说明 举例 输出

%v 相应值的默认格式。 Printf("%v", site)，Printf("%+v", site) {studygolang}，{Name:studygolang}

在打印结构体时，“加号”标记（%+v）会添加字段名

%#v 相应值的Go语法表示 Printf("#v", site) main.Website{Name:"studygolang"}

%T 相应值的类型的Go语法表示 Printf("%T", site) main.Website

%% 字面上的百分号，并非值的占位符 Printf("%%") %

****布尔占位符****

占位符 说明 举例 输出

%t 单词 true 或 false。 Printf("%t", true) true

****整数占位符****

占位符 说明 举例 输出

%b 二进制表示 Printf("%b", 5) 101

%c 相应Unicode码点所表示的字符 Printf("%c", 0x4E2D) 中

%d 十进制表示 Printf("%d", 0x12) 18

%o 八进制表示 Printf("%o", 10) 12

%q 单引号围绕的字符字面值，由Go语法安全地转义 Printf("%q", 0x4E2D) '中'

%x 十六进制表示，字母形式为小写 a-f Printf("%x", 13) d

%X 十六进制表示，字母形式为大写 A-F Printf("%x", 13) D

%U Unicode格式：U+1234，等同于 "U+%04X" Printf("%U", 0x4E2D) U+4E2D

****浮点数和复数的组成部分（实部和虚部）****

占位符 说明 举例 输出

%b 无小数部分的，指数为二的幂的科学计数法，与 strconv.FormatFloat 的 'b' 转换格式一致。例如 -123456p-78

%e 科学计数法，例如 -1234.456e+78 Printf("%e", 10.2) 1.020000e+01

%E 科学计数法，例如 -1234.456E+78 Printf("%e", 10.2) 1.020000E+01

%f 有小数点而无指数，例如 123.456 Printf("%f", 10.2) 10.200000

%g 根据情况选择 %e 或 %f 以产生更紧凑的（无末尾的0）输出 Printf("%g", 10.20) 10.2

%G 根据情况选择 %E 或 %f 以产生更紧凑的（无末尾的0）输出 Printf("%G", 10.20+2i) (10.2+2i)

****字符串与字节切片****

占位符 说明 举例 输出

%s 输出字符串表示（string类型或[]byte) Printf("%s", []byte("Go语言中文网")) 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

%X 十六进制，大写字母，每字节两个字符 Printf("%X", "golang") 676F6C616E67

****指针****

占位符 说明 举例 输出

%p 十六进制表示，前缀 0x Printf("%p", &site) 0x4f57f0

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

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

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

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

****其它标记****

占位符 说明 举例 输出

+ 总打印数值的正负号；对于%q（%+q）保证只输出ASCII编码的字符。 Printf("%+q", "中文") "\u4e2d\u6587"

- 在右侧而非左侧填充空格（左对齐该区域）

# 备用格式：为八进制添加前导 0（%#o），为十六进制添加前导 0x（%#x）或 Printf("%#U", '中') U+4E2D '中'

0X（%#X），为 %p（%#p）去掉前导 0x；如果可能的话，%q（%#q）会打印原始

（即反引号围绕的）字符串；如果是可打印字符，%U（%#U）会写出该字符的

Unicode 编码形式（如字符 x 会被打印成 U+0078 'x'）。

' ' （空格）为数值中省略的正负号留出空白（% d）；

以十六进制（% x, % X）打印字符串或切片时，在字节之间用空格隔开

0 填充前导的0而非空格；对于数字，这会将填充移到正负号之后

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

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

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

var i interface{} = 23

fmt.Printf("%v\n", i)

会打印 23。

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

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

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

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

为避免以下这类递归的情况：

type X string

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

需要在递归前转换该值：

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

****格式化错误****

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

类型错误或占位符未知：%!verb(type=value)

Printf("%d", hi): %!d(string=hi)

实参太多：%!(EXTRA type=value)

Printf("hi", "guys"): hi%!(EXTRA string=guys)

实参太少： %!verb(MISSING)

Printf("hi%d"): hi %!d(MISSING)

宽度或精度不是int类型: %!(BADWIDTH) 或 %!(BADPREC)

Printf("%\*s", 4.5, "hi"): %!(BADWIDTH)hi

Printf("%.\*s", 4.5, "hi"): %!(BADPREC)hi

所有错误都始于“%!”，有时紧跟着单个字符（占位符），并以小括号括住的描述结尾。

## **3.2. 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。

## **3.3. 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formt参数的占位符个数一致，否则会出现格式化错误；而对于其他函数，当不定参数的实参个数为多个时，它们之间会直接（对于S/F/Print）或通过" "（空格）（对于S/F/Println）连接起来（注：对于S/F/Print，当两个参数都不是字符串时，会自动添加一个空格，否则不会加。感谢guoshanhe1983 反馈。[官方 effective\_go](http://docs.studygolang.com/doc/effective_go.html" \l "Printing" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 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。

## **3.4. 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。

## **3.5. Formatter 接口**

Formatter 接口的定义如下：

type Formatter interface {

Format(f State, c rune)

}

官方文档中关于该接口方法的说明：

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

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

接着上面的例子，我们为 Person 增加一个方法：

func (this \*Person) Format(f fmt.State, c rune) {

if c == 'L' {

f.Write([]byte(this.String()))

f.Write([]byte(" Person has three fields."))

} else {

// 没有此句，会导致 fmt.Printf("%s", p) 啥也不输出

f.Write([]byte(fmt.Sprintln(this.String())))

}

}

这样，Person便实现了Formatter接口。这时再运行：

p := &Person{"polaris", 28, 0}

fmt.Printf("%L", p)

输出为：

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

这里需要解释以下几点：

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

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

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

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

一般地，我们不需要实现 Formatter 接口。如果对 Formatter 接口的实现感兴趣，可以看看标准库 [math/big](http://docscn.studygolang.com/src/math/big/floatconv.go?s=7989:8041" \l "L261" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank) 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。

## **3.6. 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。

## **3.7. 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](https://code.google.com/p/go/issues/detail?id=5391" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)。

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

## **3.8. Scanner 和 ScanState 接口**

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

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

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

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

## **3.9. fmt/print.go 阅读**

### **3.9.1. 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

}

### **3.9.2. 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 GCvar 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

}

### **3.9.3. 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

}

}

### **3.9.4. 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)

}

}

}

# **4. bufio — 缓存IO**

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

## **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)

}

### **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,

}

}

### **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，这些字节会无效。举例说明：

reader := bufio.NewReader(strings.NewReader("http://studygolang.com. \nIt is the home of gophers"))

line, \_ := reader.ReadSlice('\n')

fmt.Printf("the line:%s\n", line)

// 这里可以换上任意的 bufio 的 Read/Write 操作

n, \_ := reader.ReadSlice('\n')

fmt.Printf("the line:%s\n", line)

fmt.Println(string(n))

输出：

the line:http://studygolang.com.

the line:It is the home of gophers

It is the home of gophers

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

注意，这里的界定符可以是任意的字符，可以将上面代码中的'\n'改为'm'试试。同时，返回的结果是包含界定符本身的，上例中，输出结果有一空行就是'\n'本身(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 er r也一定不等于 nil（可能是bufio.ErrBufferFull或io.EOF）。 例子代码：

reader := bufio.NewReaderSize(strings.NewReader("http://studygolang.com"),16)

line, err := reader.ReadSlice('\n')

fmt.Printf("line:%s\terror:%s\n", line, err)

line, err = reader.ReadSlice('\n')

fmt.Printf("line:%s\terror:%s\n", line, err)

输出：

line:http://studygola error:bufio: buffer full

line:ng.com error:EOF

****ReadBytes方法签名****如下：

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

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

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

在讲解ReadSlice时说到，它返回的 []byte 是指向 Reader 中的 buffer，而不是 copy 一份返回，也正因为如此，通常我们会使用 ReadBytes 或 ReadString。很显然，ReadBytes 返回的 []byte 不会是指向 Reader 中的 buffer，通过[查看源码](http://docscn.studygolang.com/src/bufio/bufio.go?s=10277:10340" \l "L338" \t "http://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter01/_blank)可以证实这一点。

还是上面的例子，我们将 ReadSlice 改为 ReadBytes：

reader := bufio.NewReader(strings.NewReader("http://studygolang.com. \nIt is the home of gophers"))

line, \_ := reader.ReadBytes('\n')

fmt.Printf("the line:%s\n", line)

// 这里可以换上任意的 bufio 的 Read/Write 操作

n, \_ := reader.ReadBytes('\n')

fmt.Printf("the line:%s\n", line)

fmt.Println(string(n))

输出：

the line:http://studygolang.com.

the line:http://studygolang.com.

It is the home of gophers

****ReadString方法****

看一下该方法的源码：

func (b \*Reader) ReadString(delim byte) (line string, err error) {

bytes, err := b.ReadBytes(delim)

return string(bytes), err

}

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

****ReadLine方法签名****如下

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

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

ReadLine 尝试返回单独的行，不包括行尾的换行符。如果一行大于缓存，isPrefix 会被设置为 true，同时返回该行的开始部分（等于缓存大小的部分）。该行剩余的部分就会在下次调用的时候返回。当下次调用返回该行剩余部分时，isPrefix 将会是 false 。跟 ReadSlice 一样，返回的 line 只是 buffer 的引用，在下次执行IO操作时，line 会无效。可以将 ReadSlice 中的例子该为 ReadLine 试试。

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

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

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

line, err := reader.ReadBytes('\n')

line = bytes.TrimRight(line, "\r\n")

这样既读取了一行，也去掉了行尾结束符（当然，如果你希望留下行尾结束符，只用ReadBytes即可）。

### **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。

### **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。

## **4.2. Scanner 类型和方法**

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

在 bufio 包中有多种方式获取文本输入，ReadBytes、ReadString 和独特的 ReadLine，对于简单的目的这些都有些过于复杂了。在 Go 1.1 中，添加了一个新类型，Scanner，以便更容易的处理如按行读取输入序列或空格分隔单词等，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。

### **4.2.1 SplitFunc 类型和实例**

****SplitFunc 类型定义****如下：

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

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

举例说明一下这里的 token 代表的意思：

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

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

// Request more data.

return 0, nil, nil

如果 err != nil，扫描停止，同时该错误会返回。

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

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

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

****ScanBytes**** 返回单个字节作为一个 token。

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

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

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

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

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

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

****token 字段**** 上文已经解释了这个是什么意思。

### **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。

### **4.2.3 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。

### **4.2.4 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!

## **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。

### **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: w,

}

}

### **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）

### **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。

### **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。

## **4.4 ReadWriter 类型和实例化**

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

type ReadWriter struct {

\*Reader

\*Writer

}

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

func NewReadWriter(r \*Reader, w \*Writer) \*ReadWriter {

return &ReadWriter{r, w}

}