Go语言 标准库 io包

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，
- bufio实现带缓冲I/O。

其中io包中提供I/O原始操作的一系列接口。它主要包装了一些已有的实现，如 os 包中的那些，并将这些抽象成为实用性的功能和一些其他相关的接口。

错误变量

EOF

正常输入结束Read返回EOF，如果在一个结构化数据流中EOF在不期望的位置出现了，则应返回错误ErrUnexpectedEOF或者其它给出更多细节的错误。

```
var EOF = errors.New(“EOF”)
```

ErrClosedPipe

当从一个已关闭的Pipe读取或者写入时，会返回ErrClosedPipe。

```
var ErrClosedPipe = errors.New(“io: read/write on closed pipe”)
```

ErrNoProgress

某些使用io.Reader接口的客户端如果多次调用Read都不返回数据也不返回错误时，就会返回本错误，一般来说是io.Reader的实现有问题的标志。

```
var ErrNoProgress = errors.New(“multiple Read calls return no data or error”)
```

ErrShortBuffer

ErrShortBuffer表示读取操作需要大缓冲，但提供的缓冲不够大。

```
var ErrShortBuffer = errors.New(“short buffer”)
```

ErrShortWrite

ErrShortWrite表示写入操作写入的数据比提供的少，却没有显式的返回错误。

```
var ErrShortWrite = errors.New(“short write”)
```

ErrUnexpectedEOF

ErrUnexpectedEOF表示在读取一个固定尺寸的块或者数据结构时，在读取未完全时遇到了EOF。

```
var ErrUnexpectedEOF = errors.New(“unexpected EOF”)
```

基础接口

Reader接口

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

Read 将 len(p)个字节读取到 p 中。它返回读取的字节数 n（0 <= n <= len(p)）以及任何遇到的错误。即使 Read 返回的 n < len(p)，它也会在调用过程中使用 p的全部作为暂存空间。若一些数据可用但不到 len(p)个字节，Read 会照例返回可用的东西，而不是等待更多。

当 Read 在成功读取 n > 0 个字节后遇到一个错误或 EOF 情况，它就会返回读取的字节数。它会从相同的调用中返回（非nil的）错误或从随后的调用中返回错误（和 n == 0）。这种一般情况的一个例子就是 Reader 在输入流结束时会返回一个非零的字节数，可能的返回不是 err == EOF 就是 err == nil。无论如何，下一个 Read 都应当返回 0, EOF。

调用者应当总在考虑到错误 err 前处理 n > 0 的字节。这样做可以在读取一些字节，以及允许的 EOF 行为后正确地处理I/O错误。

Read 的实现会阻止返回零字节的计数和一个 nil 错误，调用者应将这种情况视作空操作。

示例：

```
package main

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

func main() {
    // 文件内容：hello world
    f, err := os.Open("a.txt")
    if err != nil {
        fmt.Printf("err: %v\n", err)
        return
    }
    defer f.Close()

    buf := make([]byte, 12) // 实例化一个长度为4的[]byte

    for {
        n, err2 := f.Read(buf) // 将内容读至buf
        if n == 0 || err2 == io.EOF {
            fmt.Println("文件以读取完毕")
            break
```

```
        }
        fmt.Println(string(buf[:n]))
    }
}
```

Writer接口

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

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。Write 不能修改此切片的数据，即便它是临时的。

示例：

```
package main

import (
    "os"
)

func main() {
    f, _ := os.OpenFile("a.txt", os.O_RDWR|os.O_APPEND, 0775) // 以读写模式打开文件，并且在写操作时将数据附加到文件尾部
    f.Write([]byte(" hello golang"))
    f.Close()
}
```

Seeker接口

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

Seeker 用来移动数据的读写指针。

Seek 设置下一次读写操作的指针位置，每次的读写操作都是从指针位置开始的。

- whence 的含义：
    - 如果 whence 为 0：表示从数据的开头开始移动指针
    - 如果 whence 为 1：表示从数据的当前指针位置开始移动指针
    - 如果 whence 为 2：表示从数据的尾部开始移动指针
- offset 是指针移动的偏移量

  返回移动后的指针位置和移动过程中遇到的任何错误。

示例：

```
package main

import (
```

```
    "fmt"
    "os"
)

func main() {
    f, _ := os.Open("a.txt") // 打开文件后，光标默认在文件开头
    f.Seek(3, 0)             // 从索引值为3处开始读
    buf := make([]byte, 10)  // 设置缓冲区
    n, _ := f.Read(buf)      // 将内容读到缓冲区内
    fmt.Printf("n: %v\n", n)
    fmt.Printf("string(buf): %v\n", string(buf))
    f.Close()
}
```

Closer接口

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

Closer关闭的接口, 带有Close() 方法， 但是行为没有定义，所以 可以特定行为来实现。

在整个标准库内都没有对Closer的引用，只有实现，用法都是开启某某连接/流，在用完/报错后在进行Close的操作。

组合接口

组合接口是对多个接口进行了组合，当同时实现多个接口时，可以使用组合接口进行传递。

ReadWriter接口

ReadWriter接口聚合了基本的读写操作。

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

ReadCloser接口

ReadCloser就是Reader+Closer，例如在ioutil中的NopCloser方法返回的就是一个ReadCloser，但是里面的Close就是个空函数，毫无作用。

```
type ReadCloser interface {
    Reader
    Closer
}
```

WriteCloser接口

WriteCloser接口聚合了基本的写入和关闭操作。

```
type WriteCloser interface {
    Writer
    Closer
}
```

ReadWriteCloser接口

ReadWriteCloser接口聚合了基本的读写和关闭操作。

```
type ReadwriteCloser interface {
    Reader
    Writer
    Closer
}
```

ReadSeeker接口

ReadSeeker接口聚合了基本的读取和移位操作。

```
type ReadSeeker interface {
    Reader
    Seeker
}
```

WriteSeeker接口

WriteSeeker接口聚合了基本的写入和移位操作。

```
type WriteSeeker interface {
    Writer
    Seeker
}
```

ReadWriteSeeker接口

ReadWriteSeeker接口聚合了基本的读写和移位操作。

```
type ReadWriteSeeker interface {
    Reader
    Writer
    Seeker
}
```

指定读写器读写接口

ReaderFrom接口

ReadFrom 从 r 中读取数据到对象的数据流中，直到 r 返回 EOF 或 r 出现读取错误为止，返回值 n 是读取的字节数，返回值 err 就是 r 的返回值 err。

也就是说从输入流 r 中读取数据，写入底层输出流，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其它错误 err 也将被返回。

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

> ReadFrom`方法不会返回`err == io.EOF

示例：

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    // test.txt 文件内容：hello world!!!
    file, err := os.Open("test.txt")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    // 调用 file 的读操作从文件中读取文件，并写入标准输出
    writer := bufio.NewWriter(os.Stdout)
    n, err := writer.ReadFrom(file)
    if err != nil {
        panic(err)
    }
    fmt.Printf("read %d bytes\n", n)
    // 将缓存中的数据 flush 到控制台
    writer.Flush()
}

// 控制台输出：
hello world!!!
read 15 bytes
```

WriterTo接口

WriterTo 将对象的数据流写入到 w 中，直到对象的数据流全部写入完毕或遇到写入错误为止，返回值 n 是写入的字节数，返回值 err 就是 w 的返回值 err。

也就是说从底层输入流中读取数据，写入输出流 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。

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

示例：

```
package main

import (
    "bytes"
    "fmt"
    "os"
)

func main() {
    // 调用 r 的读操作读取数据，并写入到标准输出
    r := bytes.NewReader([]byte("Go语言是最好的语言\n"))
    n, err := r.WriteTo(os.Stdout)
    if err != nil {
        panic(err)
    }
    fmt.Printf("write %d bytes\n", n)
}

// 控制台输出：
Go语言是最好的语言
write 27 bytes
```

指定偏移量读写接口

ReaderAt接口

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

ReadAt 从对象数据流的 off 处读出数据到 p 中：

- 忽略数据的读写指针，从数据的起始位置偏移 off 处开始读取
- 如果对象的数据流只有部分可用，不足以填满 p则 ReadAt 将等待所有数据可用之后，继续向 p 中写入直到将 p 填满后再返回，在这点上 ReadAt 要比 Read 更严格
- 返回读取的字节数 n 和读取时遇到的错误
- 如果 n < len(p)，则需要返回一个 err 值来说明为什么没有将 p 填满（比如 EOF）
- 如果 n = len(p)，而且对象的数据没有全部读完，则err 将返回 nil
- 如果 n = len(p)，而且对象的数据刚好全部读完，则err 将返回 EOF 或者 nil（不确定）

示例：

```
package main

import (
    "fmt"
    "strings"
```

```
)

func main() {
    reader := strings.NewReader("Go语言是全世界最好的语言。")
    p := make([]byte, 6)
    n, err := reader.ReadAt(p, 2)
    if err != nil {
        panic(err)
    }
    fmt.Printf("Read %d bytes: %s\n", n, p) // Read 6 bytes: 语言
}
```

WriterAt接口

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

WriteAt 将 p 中的数据写入到对象数据流的 off 处

- 忽略数据的读写指针，从数据的起始位置偏移 off 处开始写入
- 返回写入的字节数和写入时遇到的错误
- 如果 n < len(p)，则必须返回一个 err 值来说明为什么没有将 p 完全写入

示例：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Create("writeAt.txt")
    if err != nil {
        panic(err)
    }
    defer file.Close()
    file.WriteString("Golang是最好的--这里是瞎写的!")
    n, err := file.WriteAt([]byte("PHP才是最好的!"), 20)
    if err != nil {
        panic(err)
    }
    fmt.Printf("write %d bytes", n) // write 19 bytes
    file.WriteString("hello world!!!")
}

// writeAt.txt 文件内容：
// Golang是最好的--PHP才是最好的!hello world!!!
```

单个字节读写接口

ByteReader接口

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

ByteReader是基本的ReadByte方法的包装。
ReadByte读取输入中的单个字节并返回。如果没有字节可读取，会返回错误。

ByteScanner接口

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

ByteScanner接口在基本的ReadByte方法之外还添加了UnreadByte方法。

UnreadByte方法让下一次调用ReadByte时返回之前调用ReadByte时返回的同一个字节。连续调用两次UnreadByte方法而中间没有调用ReadByte时，可能会导致错误。

ByteWriter接口

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

包装 WriteByte 单个字节写入方法的接口。

RuneReader接口

```
type RuneReader interface {
    ReadRune() (r rune, size int, err error)
}
```

ReadRune 方法的包装，读取单个UTF-8编码的Unicode字符，并返回rune及其字节大小。如果没有可用字符，将设置err。

RuneScanner接口

```
type RuneScanner interface {
    RuneReader
    UnreadRune() error
}
```

RuneScanner接口在基本的ReadRune方法之外还添加了UnreadRune方法。

UnreadRune方法让下一次调用ReadRune时返回之前调用ReadRune时返回的同一个utf-8字符。连续调用两次UnreadRune方法而中间没有调用ReadRune时，可能会导致错误。

StringWriter接口

```
type StringWriter interface {
    WriteString(s string) (n int, err error)
}
```

字符串写入方法WriteString的包装。

结构体

LimitedReader

```
type LimitedReader struct {
    R   Reader // underlying reader
    N   int64  // max bytes remaining
}
```

LimitedReader从R读取，但将返回的数据量限制为N个字节。每次读取更新N以标记剩余可以读取的字节数。Read在N<=0时或基础R返回EOF时返回EOF。

具体实现方法为：`func LimitReader(r Reader, n int64) Reader`。

PipeReader

```
type PipeReader struct {
    // 内含隐藏或非导出字段
}
```

PipeReader是一个管道的读取端。

具体实现方法有：

Read

```
func (r *PipeReader) Read(data []byte) (n int, err error)
```

Read实现了标准的读取接口：它从管道中读取数据，阻塞直到写入端到达或写入端被关闭。如果用错误关闭写入端，则返回错误为ERR；否则ERR为EOF。

Close

```
func (r *PipeReader) Close() error
```

Close关闭读取器；关闭后如果对管道的写入端进行写入操作，就会返回(0, ErrClosedPip)。

CloseWithError

```
func (r *PipeReader) CloseWithError(err error) error
```

CloseWithError类似Close方法，但将调用Write时返回的错误改为err。

PipeWriter

```
type PipeWriter struct {
    // 内含隐藏或非导出字段
}
```

PipeWriter是一个管道的写入端。

具体实现方法有：

Write

```
func (w *PipeWriter) Write(data []byte) (n int, err error)
```

Write实现了标准的写接口：它将数据写入管道，直到一个或多个读取端消耗完所有数据或读取端关闭为止。如果以错误关闭读取端，则该错误将作为ERR返回；否则ERR将为ErrClosedPipe。

Close

```
func (w *PipeWriter) Close() error
```

Close关闭写入器；关闭后如果对管道的读取端进行读取操作，就会返回(0, EOF)。

CloseWithError

```
func (w *PipeWriter) CloseWithError(err error) error
```

CloseWithError类似Close方法，但将调用Read时返回的错误改为err。

> 注：以上两个结构体PipeWriter与PipeReader是结合使用的需要用Pipe()方法进行创建。

SectionReader

```
type SectionReader struct {
    // contains filtered or unexported fields
}
```

SectionReader在ReaderAt的基础上实现了Read，Seek和ReadAt。

具体实现方法有：

NewSectionReader

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

结构体SectionReader的创建方法
NewSectionReader返回一个SectionReader，它从r开始读取，偏移量为off，并在n个字节后以EOF停止。

Read

```
func (s *SectionReader) Read(p []byte) (n int, err error)
```

实现了接口Reader的Read方法

ReadAt

```
func (s *SectionReader) ReadAt(p []byte, off int64) (n int, err error)
```

实现了接口ReaderAt的ReadAt方法

Seek

```
func (s *SectionReader) Seek(offset int64, whence int) (int64, error)
```

实现了接口Seeker的Seek方法

Size

```
func (s *SectionReader) Size() int64
```

Size返回以字节为单位的片段大小。

供外部调用的函数

Copy

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

将副本从src复制到dst，直到在src上达到EOF或发生错误。它返回复制的字节数和复制时遇到的第一个错误（如果有）。 成功的复制将返回err == nil而不是err == EOF。因为复制被定义为从src读取直到EOF，所以它不会将读取的EOF视为要报告的错误。 如果src实现WriterTo接口，则通过调用src.WriteTo（dst）实现该副本。否则，如果dst实现了ReaderFrom接口，则通过调用dst.ReadFrom（src）实现该副本。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    if _, err := io.Copy(os.Stdout, r); err != nil {
        // os.Stdout将内容输出到控制台
```

```
        log.Fatal(err)
        // log.Fatal函数完成：
        // 1. 打印输出err
        // 2. 退出应用程序
    }
}
#结果
some io.Reader stream to be read
```

CopyBuffer

```
func CopyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error)
```

CopyBuffer与Copy相同，区别在于CopyBuffer逐步遍历提供的缓冲区（如果需要），而不是分配临时缓冲区。如果buf为nil，则分配一个；如果长度为零，则CopyBuffer会panic报错。 如果src实现WriterTo或dst实现ReaderFrom，则buf将不用于执行复制。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r1 := strings.NewReader("first reader\n")
    r2 := strings.NewReader("second reader\n")
    buf := make([]byte, 8)

    // buf is used here...
    if _, err := io.CopyBuffer(os.Stdout, r1, buf); err != nil {
        log.Fatal(err)
    }

    // ... reused here also. No need to allocate an extra buffer.
    if _, err := io.CopyBuffer(os.Stdout, r2, buf); err != nil {
        log.Fatal(err)
    }
}

#结果
first reader
second reader
```

CopyN

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

CopyN将n个字节（或直到出错）从src复制到dst。它返回复制的字节数以及复制时遇到的最早错误。返回时，只有err == nil时，writte == n。 如果dst实现了ReaderFrom接口，则使用该接口实现副本。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read")

    if _, err := io.CopyN(os.Stdout, r, 4); err != nil {
        log.Fatal(err)
    }
}
#结果
some
```

LimitReader

```
func LimitReader(r Reader, n int64) Reader
```

LimitedReader从r读取，但将返回的数据量限制为n个字节。每次读取更新n以标记剩余可以读取的字节数。Read在n<=0时或基础r返回EOF时返回EOF。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    lr := io.LimitReader(r, 4)

    if _, err := io.Copy(os.Stdout, lr); err != nil {
        log.Fatal(err)
    }
```

```
}
#结果
some
```

MultiReader

```
func MultiReader(readers ...Reader) Reader
```

MultiReader返回一个Reader，它是所提供的输入阅读器的逻辑串联。它们被顺序读取。一旦所有输入均返回EOF，读取将返回EOF。如果任何读取器返回非零，非EOF错误，则Read将返回该错误。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r1 := strings.NewReader("first reader ")
    r2 := strings.NewReader("second reader ")
    r3 := strings.NewReader("third reader\n")
    r := io.MultiReader(r1, r2, r3)

    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }
}
#结果
first reader second reader third reader
```

MultiWriter

```
func MultiWriter(writers ...Writer) Writer
```

MultiWriter创建一个Writers，将其写入复制到所有提供的写入器中，类似于Unix tee（1）命令。 每个写入一次写入每个列出的写入器。如果列出的写程序返回错误，则整个写操作将停止并返回错误；它不会在列表中继续下去。

示例：

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "log"
    "strings"
```

```
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    var buf1, buf2 bytes.Buffer
    w := io.MultiWriter(&buf1, &buf2)

    if _, err := io.Copy(w, r); err != nil {
        log.Fatal(err)
    }

    fmt.Print(buf1.String())
    fmt.Print(buf2.String())
}
#结果
some io.Reader stream to be read
some io.Reader stream to be read
```

Pipe

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

Pipe创建一个同步的内存管道。
可用于连接期望io.Reader的代码和期望io.Writer的代码。

管道上的读和写是一对一匹配的，除非需要多次读取才能使用单次写入。也就是说，每次对PipeWriter的写入都将阻塞，直到它满足从PipeReader读取的一个或多个读取，这些读取会完全消耗已写入的数据。

数据直接从Write复制到相应的Read (或Reads)；没有内部缓冲。
对读的并行调用和对写的并行调用也是安全的：单个调用将按顺序执行。

示例：

```
package main

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

func main() {
    r, w := io.Pipe()

    go func() {
        fmt.Fprint(w, "some io.Reader stream to be read\n")
        w.Close()
    }()

    if _, err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
```

```
    }

}
#结果
some io.Reader stream to be read
```

ReadAll

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

ReadAll从r读取，直到出现错误或EOF，并返回其读取的数据。成功的调用返回errnil，而不是errEOF。由于ReadAll定义为从src读取直到EOF，因此它不会将读取的EOF视为要报告的错误。

示例：

```
package main

import (
    "fmt"
    "io"
    "log"
    "strings"
)

func main() {
    r := strings.NewReader("Go is a general-purpose language designed with
systems programming in mind.")

    b, err := io.ReadAll(r)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s", b)

}
#结果
Go is a general-purpose language designed with systems programming in mind.
```

ReadAtLeast

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

ReadAtLeast从r读取到buf，直到它至少读取了min字节。它返回复制的字节数n，如果读取的字节数少则返回错误。仅当未读取任何字节时，错误才是EOF。如果在读取少于最小字节后发生EOF，则ReadAtLeast返回ErrUnexpectedEOF。如果min大于buf的长度，则ReadAtLeast返回ErrShortBuffer。返回时，当且仅当err == nil时，n> = min。

示例：

```
import (
    "fmt"
    "io"
```

```
    "log"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    buf := make([]byte, 14)
    if _, err := io.ReadAtLeast(r, buf, 4); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s\n", buf)

    // buffer smaller than minimal read size.
    shortBuf := make([]byte, 3)
    if _, err := io.ReadAtLeast(r, shortBuf, 4); err != nil {
        fmt.Println("error:", err)
    }

    // minimal read size bigger than io.Reader stream
    longBuf := make([]byte, 64)
    if _, err := io.ReadAtLeast(r, longBuf, 64); err != nil {
        fmt.Println("error:", err)
    }

}
#结果
some io.Reader
error: short buffer
error: unexpected EOF
```

ReadFull

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

ReadFull将r中的len（buf）个字节准确地读取到buf中。它返回复制的字节数，如果读取的字节数少则返回错误。仅当未读取任何字节时，错误才是EOF。如果在读取了一些但不是全部字节后发生EOF，则ReadFull返回ErrUnexpectedEOF。返回时，当且仅当err == nil时，n == len（buf）。

示例：

```
package main

import (
    "fmt"
    "io"
    "log"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")

    buf := make([]byte, 4)
```

```
    if _, err := io.ReadFull(r, buf); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s\n", buf)

    // minimal read size bigger than io.Reader stream
    longBuf := make([]byte, 64)
    if _, err := io.ReadFull(r, longBuf); err != nil {
        fmt.Println("error:", err)
    }
}
#结果
some
error: unexpected EOF
```

SectionReader

SectionReader在ReaderAt的基础上实现了Read，Seek和ReadAt。

具体实现方法有：

NewSectionReader

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

结构体SectionReader的创建方法
NewSectionReader返回一个SectionReader，它从r开始读取，偏移量为off，并在n个字节后以EOF停止。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    if _, err := io.Copy(os.Stdout, s); err != nil {
        log.Fatal(err)
    }

}
#结果
io.Reader stream
```

SectionReader.Read

```
func (s *SectionReader) Read(p []byte) (n int, err error)
```

实现了接口Reader的Read方法。

```
import (
    "fmt"
    "io"
    "log"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    buf := make([]byte, 9)
    if _, err := s.Read(buf); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s\n", buf)

}
#结果
io.Reader
```

SectionReader.ReadAt

```
func (s *SectionReader) ReadAt(p []byte, off int64) (n int, err error)
```

实现了接口ReaderAt的ReadAt方法。

示例：

```
package main

import (
    "fmt"
    "io"
    "log"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    buf := make([]byte, 6)
    if _, err := s.ReadAt(buf, 10); err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Printf("%s\n", buf)

}
#结果
stream
```

SectionReader.Seek

```
func (s *SectionReader) Seek(offset int64, whence int) (int64, error)
```

实现了接口Seeker的Seek方法。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    if _, err := s.Seek(10, io.SeekStart); err != nil {
        log.Fatal(err)
    }

    if _, err := io.Copy(os.Stdout, s); err != nil {
        log.Fatal(err)
    }

}
#结果
stream
```

> 可以看的出来SectionReader是根据ReaderAt实现的，而非Seeker，虽然两者的效果很像，但是ReaderAt读取内容是无视Seeker偏移量的。且在读取数据大小上ReadAt是要比Read严格的，同样的Bytes在Read上即使设大了也会没事，但在ReadAt会报错。

SectionReader.Size

```
func (s *SectionReader) Size() int64
```

Size返回以字节为单位的片段大小。

示例：

```
package main
```

```
import (
    "fmt"
    "io"
    "strings"
)

func main() {
    r := strings.NewReader("some io.Reader stream to be read\n")
    s := io.NewSectionReader(r, 5, 17)

    fmt.Println(s.Size())

}
#结果
17
```

TeeReader

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

TeeReader返回一个Reader，该Reader向w写入从r读取的内容。通过r执行的所有r读取均与对w的相应写入匹配。没有内部缓冲-写入必须在读取完成之前完成。写入时遇到的任何错误均报告为读取错误。

示例：

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    var r io.Reader = strings.NewReader("some io.Reader stream to be read\n")

    r = io.TeeReader(r, os.Stdout)

    // Everything read from r will be copied to stdout.
    if _, err := io.ReadAll(r); err != nil {
        log.Fatal(err)
    }

}
#结果
some io.Reader stream to be read
```

WriteString

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

WriteString将字符串s的内容写入w，w接受字节片。如果w实现StringWriter，则直接调用其WriteString方法。否则，w.Write只调用一次。

示例：

```
package main

import (
    "io"
    "log"
    "os"
)

func main() {
    if _, err := io.WriteString(os.Stdout, "Hello World"); err != nil {
        log.Fatal(err)
    }

}
#结果
Hello World
```