

# Go语言 标准库 bufio包

bufio包实现了有缓冲的I/O。它包装一个io.Reader或io.Writer接口对象，创建另一个也实现了该接口，且同时还提供了缓冲和一些文本I/O的帮助函数的对象。

它提供了 `bufio.Reader` 和 `bufio.Writer` 类型，其内部分别包装了 `io.Reader` 和 `io.Writer` 对象，同时分别实现了 `io.Reader` 和 `io.Writer` 接口。同时，该包为文本 I/O 提供了一些便利操作。

## bufio包原理

io操作本身的效率并不低，低的是频繁的访问本地磁盘的文件。所以bufio就提供了缓冲区(分配一块内存)，读和写都先在缓冲区中，最后再读写文件，来降低访问本地磁盘的次数，从而提高效率。

简单的说就是，把文件读取进缓冲（内存）之后再读取的时候就可以避免文件系统的io 从而提高速度。同理，在进行写操作时，先把文件写入缓冲（内存），然后由缓冲写入文件系统。看完以上解释有人可能会表示困惑了，直接把 内容->文件 和 内容->缓冲->文件相比，缓冲区好像没有起到作用嘛。其实缓冲区的设计是为了存储多次的写入，最后一口气把缓冲区内容写入文件。



## bufio.Reader 类型

bufio 包提供了带缓存功能的 Reader，在从硬盘中读取字节前使用内存缓存，可以节省操作硬盘 I/O 的时间。在一些情况下它也很有用，比如每次读一个字节，可以一次性的从硬盘中读取缓存大小的数据到内存缓存中，然后读取缓存中的字节，减少硬盘的磨损以及提升性能。

## 基础介绍

`bufio.Reader` 是bufio中对io.Reader 的封装。

```
// 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
type Reader struct {
    buf []byte    // 缓存
    rd io.Reader // 底层的 io.Reader
    // r: 从 buf 中读取的字节（偏移）；w: buf 中写入内容的偏移；
    // w - r 是 buf 中可被读的 length（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值，-1 是无效值
    lastRuneSize int // 最后一次读到的 Rune 的大小，只有 ReadRune 操作才会修改该值，-1 是无效值
}
```

`bufio.Read(p []byte)` 相当于读取大小 `len(p)` 的内容，思路如下：

1. 当缓存区有内容的时，将缓存区内容全部填入 `p` 并清空缓存区
2. 当缓存区没有内容的时候且 `len(p) > len(buf)`，即要读取的内容比缓存区还要大，直接去文件读取即可
3. 当缓存区没有内容的时候且 `len(p) < len(buf)`，即要读取的内容比缓存区小，缓存区从文件读取内容充满缓存区，并将 `p` 填满（此时缓存区有剩余内容）
4. 以后再次读取时缓存区有内容，将缓存区内容全部填入 `p` 并清空缓存区（此时和情况1一样）
5. `reader` 内部通过维护一个 `r, w` 即读入和写入的位置索引来判断是否缓存区内容被全部读出。

## 方法

### NewReaderSize

```
func NewReaderSize(rd io.Reader, size int) *Reader
```

`NewReaderSize` 将 `rd` 封装成一个带缓存的 `bufio.Reader` 对象，缓存大小由 `size` 指定（如果小于 16 则会被设置为 16）。

如果 `rd` 的基类型就是有足够缓存的 `bufio.Reader` 类型，则直接将 `rd` 转换为基类型返回。

### NewReader

```
func NewReader(rd io.Reader) *Reader
```

`NewReader` 相当于 `NewReaderSize(rd, 4096)`。

### Reset

```
func (b *Reader) Reset(r io.Reader)
```

`Reset` 丢弃缓冲中的数据，清除任何错误，将 `b` 重设为其下层从 `r` 读取数据。

示例：

```
package main

import (
    "bufio"
    "fmt"
    "strings"
)
```

```

func main() {
    s := strings.NewReader("ABCDEFGG")
    str := strings.NewReader("12345")
    br := bufio.NewReader(s)
    b, _ := br.ReadString('\n')
    fmt.Println(b)
    br.Reset(str)
    b, _ = br.ReadString('\n')
    fmt.Println(b)
}
#结果
ABCDEFGG
12345

```

## Read

```
func (b *Reader) Read(p []byte) (n int, err error)
```

Read读取数据写入p。本方法返回去写入p的字节数。本方法一次调用最多会调用下层Reader接口一次Read方法，因此返回值n可能小于len(p)。读取到达结尾时，返回值n将为0而err将为io.EOF。

示例：

```

package main

import (
    "bufio"
    "fmt"
    "io"
    "strings"
)

func main() {
    s := strings.NewReader("ABCDEFGHJKLMNOPQRSTUVWXYZ1234567890")
    br := bufio.NewReader(s)
    p := make([]byte, 10)

    for {
        n, err := br.Read(p)
        if err == io.EOF {
            break
        } else {
            fmt.Printf("string(p[0:n]): %v\n", string(p[0:n]))
        }
    }
}
#结果
string(p[0:n]): ABCDEFGHIJ
string(p[0:n]): KLMNOPQRST
string(p[0:n]): UVWXYZ1234
string(p[0:n]): 567890

```

## Peek

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

Peek 返回缓存的一个切片，该切片引用缓存中前 n 个字节的数据，该操作不会将数据读出，只是引用，引用的数据在下次读取操作之前是有效的。如果切片长度小于 n，则返回一个错误信息说明原因。

如果 n 大于缓存的总大小，则返回 ErrBufferFull。

## ReadByte

```
func (b *Reader) ReadByte() (c byte, err error)
```

ReadByte 读取并返回一个字节。如果没有可用的数据，会返回错误。

## UnreadByte

```
func (b *Reader) UnreadByte() error
```

UnreadByte 吐出最近一次读取操作读取的最后一个字节。（只能吐出最后一个，多次调用会出问题）。

示例：

```
package main

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

func main() {
    s := strings.NewReader("ABCDEFG")
    br := bufio.NewReader(s)

    c, _ := br.ReadByte()
    fmt.Printf("%c\n", c)

    c, _ = br.ReadByte()
    fmt.Printf("%c\n", c)

    br.UnreadByte()
    c, _ = br.ReadByte()
    fmt.Printf("%c\n", c)
}

#结果
A
B
B
```

## ReadRune

```
func (b *Reader) ReadRune() (r rune, size int, err error)
```

ReadRune读取一个utf-8编码的unicode码值，返回该码值、其编码长度和可能的错误。如果utf-8编码非法，读取位置只移动1字节，返回U+FFFD，返回值size为1而err为nil。如果没有可用的数据，会返回错误。

## UnreadRune

```
func (b *Reader) UnreadRune() error
```

UnreadRune吐出最后一次ReadRune调用读取的unicode码值。如果最近一次读取不是调用的ReadRune，会返回错误。（从这点看，UnreadRune比unreadByte严格很多）。

示例：

```
package main

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

func main() {
    s := strings.NewReader("你好，世界！")
    br := bufio.NewReader(s)

    c, size, _ := br.ReadRune()
    fmt.Printf("%c %v\n", c, size)

    c, size, _ = br.ReadRune()
    fmt.Printf("%c %v\n", c, size)

    br.UnreadRune()
    c, size, _ = br.ReadRune()
    fmt.Printf("%c %v\n", c, size)
}
```

## ReadLine

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

ReadLine尝试返回一行数据，不包括行尾标志的字节。如果行太长超过了缓冲，返回值isPrefix会被设为true，并返回行的前面一部分。该行剩下的部分将在之后的调用中返回。返回值isPrefix会在返回该行最后一个片段时才设为false。返回切片是缓冲的子切片，只在下一次读取操作之前有效。ReadLine要么返回一个非nil的line，要么返回一个非nil的err，两个返回值至少一个非nil。

返回的文本不包含行尾的标志字节（“\r\n”或“\n”）。如果输入流结束时没有行尾标志字节，方法不会出错，也不会指出这一情况。在调用ReadLine之后调用UnreadByte会总是吐出最后一个读取的字节（很可能是该行的行尾标志字节），即使该字节不是ReadLine返回值的一部分。

示例：

```
package main

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

func main() {
    s := strings.NewReader("ABC\\nDEF\\r\\nGHI\\r\\nGHI")
    br := bufio.NewReader(s)

    w, isPrefix, _ := br.ReadLine()
    fmt.Printf("%q %v\\n", w, isPrefix)

    w, isPrefix, _ = br.ReadLine()
    fmt.Printf("%q %v\\n", w, isPrefix)

    w, isPrefix, _ = br.ReadLine()
    fmt.Printf("%q %v\\n", w, isPrefix)

    w, isPrefix, _ = br.ReadLine()
    fmt.Printf("%q %v\\n", w, isPrefix)
}

#结果
"ABC" false
"DEF" false
"GHI" false
"GHI" false
```

## ReadSlice

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

ReadSlice读取直到第一次遇到delim字节，返回缓冲里的包含已读取的数据和delim字节的切片。该返回值只在下一次读取操作之前合法。如果ReadSlice放在在读取到delim之前遇到了错误，它会返回在错误之前读取的数据在缓冲中的切片以及该错误（一般是io.EOF）。如果在读取delim之前缓冲就写满了，ReadSlice失败并返回ErrBufferFull。因为ReadSlice的返回值会被下一次I/O操作重写，调用者应尽量使用ReadBytes或ReadString替代本方法。当且仅当ReadBytes方法返回的切片不以delim结尾时，会返回一次非nil的错误。

示例：

```
package main

import (
    "bufio"
    "fmt"
    "strings"
)
```

```
func main() {
    s := strings.NewReader("ABC,DEF,GHI,JKL")
    br := bufio.NewReader(s)

    w, _ := br.ReadSlice(',')
    fmt.Printf("%q\n", w)

    w, _ = br.ReadSlice(',')
    fmt.Printf("%q\n", w)

    w, _ = br.ReadSlice(',')
    fmt.Printf("%q\n", w)
}
#结果
"ABC,"
"DEF,"
"GHI,"
```

## ReadString

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

ReadString读取直到第一次遇到delim字节，返回一个包含已读取的数据和delim字节的字符串。如果ReadString方法在读取到delim之前遇到了错误，它会返回在错误之前读取的数据以及该错误（一般是io.EOF）。当且仅当ReadString方法返回的切片不以delim结尾时，会返回一个非nil的错误。

示例：

```
package main

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

func main() {
    s := strings.NewReader("ABC DEF GHI JKL")
    br := bufio.NewReader(s)

    w, _ := br.ReadString(' ')
    fmt.Printf("%q\n", w)

    w, _ = br.ReadString(' ')
    fmt.Printf("%q\n", w)

    w, _ = br.ReadString(' ')
    fmt.Printf("%q\n", w)
}
#结果
"ABC "
"DEF "
"GHI "
```

## WriteTo

```
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo方法实现了io.WriterTo接口。

示例：

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
    "strings"
)

func main() {
    s := strings.NewReader("ABCDEFGHIJKLMN")
    br := bufio.NewReader(s)
    b := bytes.NewBuffer(make([]byte, 0))

    br.WriteTo(b)
    fmt.Printf("%s\n", b)
}
#结果
ABCDEFGHIJKLMN
```

## Buffered

```
func (b *Reader) Buffered() int
```

返回缓存中现有的可读取的字节数。

## bufio.Writer 类型

### 基础介绍

bufio 包提供了带缓存功能的 Writer，在写字节到硬盘前使用内存缓存，可以节省操作硬盘 I/O 的时间。在一些情况下它也很有用，比如每次写一个字节，可以把它们攒在内存缓存中，然后一次写入到硬盘中，减少硬盘的磨损以及提升性能。

```
// 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
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
}
```

bufio.Write(p []byte) 的思路如下

1. 判断buf中可用容量是否可以放下 p



2. 如果能放下，直接把p拼接到buf后面，即把内容放到缓冲区
3. 如果缓冲区的可用容量不足以放下，且此时缓冲区是空的，直接把p写入文件即可
4. 如果缓冲区的可用容量不足以放下，且此时缓冲区有内容，则用p把缓冲区填满，把缓冲区所有内容写入文件，并清空缓冲区
5. 判断p的剩余内容大小能否放到缓冲区，如果能放下（此时和步骤1情况一样）则把内容放到缓冲区
6. 如果p的剩余内容依旧大于缓冲区，（注意此时缓冲区是空的，情况和步骤3一样）则把p的剩余内容直接写入文件

## 方法

### NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建一个具有默认大小缓冲、写入w的\*Writer。NewWriter相当于NewWriterSize(wr, 4096)。

### NewWriterSize

```
func NewWriterSize(w io.Writer, size int) *Writer
```

NewWriterSize创建一个具有最少有size尺寸的缓冲、写入w的Writer。如果参数w已经是一个具有足够大缓冲的Writer类型值，会返回w。

### Reset

```
func (b *Writer) Reset(w io.Writer)
```

Reset丢弃缓冲中的数据，清除任何错误，将b重设为将其输出写入w。

示例：

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
)

func main() {
    b := bytes.NewBuffer(make([]byte, 0))
    bw := bufio.NewWriter(b)
    bw.WriteString("123456789")
    c := bytes.NewBuffer(make([]byte, 0))
    bw.Reset(c)
    bw.WriteString("456")
    bw.Flush()
    fmt.Println(b)
    fmt.Println(c)
}

#结果
```

456

## Buffered

```
func (b *Writer) Buffered() int
```

Buffered返回缓冲中已使用的字节数。

## Available

```
func (b *Writer) Available() int
```

Available返回缓冲中海油多少字节未使用。

## Write

```
func (b *Writer) Write(p []byte) (nn int, err error)
```

Write将p的内容写入缓冲。返回写入的字节数。如果返回值 $nn < len(s)$ ，还会返回一个错误说明原因。

## WriteString

```
func (b *Writer) WriteString(s string) (int, error)
```

WriteString写入一个字符串。返回写入的字节数。如果返回值 $m < len(s)$ ，还会返回一个错误说明原因。

## WriteByte

```
func (b *Writer) WriteByte(c byte) error
```

WriteByte写入单个字节。

## WriteRune

```
func (b *Writer) WriteRune(r rune) (size int, err error)
```

WriteRune写入一个unicode码值（的utf-8编码），返回写入的字节数和可能的错误。

## Flush

```
func (b *Writer) Flush() error
```

Flush方法将缓冲中的数据写入下层的io.Writer接口。

## ReadFrom

```
func (b *Writer) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom实现了io.ReaderFrom接口。

示例一：

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
)

func main() {
    b := bytes.NewBuffer(make([]byte, 0))
    bw := bufio.NewWriter(b)
    fmt.Println(bw.Available()) // 4096
    fmt.Println(bw.Buffered())  // 0

    bw.WriteString("ABCDEFGHIJKLMN")
    fmt.Println(bw.Available())
    fmt.Println(bw.Buffered())
    fmt.Printf("%q\n", b)

    bw.Flush()
    fmt.Println(bw.Available())
    fmt.Println(bw.Buffered())
    fmt.Printf("%q\n", b)
}

#结果
4096
0
4082
14
""
4096
0
"ABCDEFGHIJKLMN"
```

示例二：

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
    "strings"
)

func main() {
```

```
b := bytes.NewBuffer(make([]byte, 0))
s := strings.NewReader("Hello 世界!")
bw := bufio.NewWriter(b)
bw.ReadFrom(s)
// bw.Flush() // ReadFrom无需使用Flush, 其自己已经写入
fmt.Println(b) // Hello 世界!
}
#结果
Hello 世界!
```

## bufio.ReadWriter 类型

### 基础介绍

ReadWriter类型保管了指向Reader和Writer类型的指针，（因此）实现了io.ReadWriter接口。

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

### 方法

#### NewReadWriter

```
func NewReadWriter(r *Reader, w *Writer) *ReadWriter
```

NewReadWriter申请创建一个新的、将读写操作分派给r和w的ReadWriter。

示例：

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
    "strings"
)

func main() {
    b := bytes.NewBuffer(make([]byte, 0))
    bw := bufio.NewWriter(b)
    s := strings.NewReader("123")
    br := bufio.NewReader(s)
    rw := bufio.NewReadWriter(br, bw)
    p, _ := rw.ReadString('\n')
    fmt.Println(string(p)) // 123
    rw.WriteString("asdf")
    rw.Flush()
    fmt.Println(b) // asdf
}
#结果
```

```
123
asdf
```

## bufio.SplitFunc 类型

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

SplitFunc类型代表用于对输出作词法分析的分割函数。

参数data是尚未处理的数据的一个开始部分的切片，参数atEOF表示是否Reader接口不能提供更多的数据。返回值是解析位置前进的字节数，将要返回给调用者的token切片，以及可能遇到的错误。如果数据不足以（保证）生成一个完整的token，例如需要一整行数据但data里没有换行符，SplitFunc可以返回(0, nil, nil)来告诉Scanner读取更多的数据写入切片然后用从同一位置起始、长度更长的切片再试一次（调用SplitFunc类型函数）。

如果返回值err非nil，扫描将终止并将该错误返回给Scanner的调用者。

除非atEOF为真，永远不会使用空切片data调用SplitFunc类型函数。然而，如果atEOF为真，data却可能是非空的、且包含着未处理的文本。

SplitFunc的作用很简单，从data中找出你感兴趣的数据，然后返回并告诉调用者，data中有多少数据你已经处理过了。

## bufio.Scanner 类型

### 基础介绍

```
// bufio.Scanner 类型提供了方便的读取数据的接口，如从换行符分隔的文本里读取每一行
type Scanner struct {
    r          io.Reader // The reader provided by the client.
    split      SplitFunc // The function to split the token.
    maxTokenSize int      // Maximum size of a token; modified by tests.
    token      []byte   // Last token returned by split.
    buf        []byte   // Buffer used an argument to split.
    start      int      // First non-processed byte in buf.
    end        int      // End of data in buf.
    err        error    // Sticky error.
}
```

Scanner类型提供了方便的读取数据的接口，如从换行符分隔的文本里读取每一行。成功调用的Scan方法会逐步提供文件的token，跳过token之间的字节。token由SplitFunc类型的分割函数指定；默认的分割函数会将输入分割为多个行，并去掉行尾的换行标志。本包预定义的分割函数可以将文件分割为行、字节、unicode码值、空白分隔的word。调用者可以定制自己的分割函数。扫描会在抵达输入流结尾、遇到的第一个I/O错误、token过大不能保存进缓冲时，不可恢复的停止。当扫描停止后，当前读取位置可能会在最后一个获得的token后面。需要更多对错误管理的控制或token很大，或必须从reader连续扫描的程序，应使用bufio.Reader代替。

Scanner 是 bufio 包下的类型，在处理文件中以分隔符分隔的文本时很有用。通常我们使用换行符作为分隔符将文件内容分成多行。在 CSV 文件中，逗号一般作为分隔符。`os.File` 文件可以被包装成 `bufio.Scanner`，它就像一个缓存 Reader。我们会调用 `Scan()` 方法去读取下一个分隔符，使用 `Text()` 或者 `Bytes()` 获取读取的数据。

分隔符可以不是一个简单的字节或者字符，有一个特殊的方法可以实现分隔符的功能，以及将指针移动多少，返回什么数据。如果没有定制的 `SplitFunc` 提供，缺省的 `ScanLines()` 会使用 `newline` 字符作为分隔符，其它的分隔函数还包括 `ScanRunes()` 和 `ScanWords()`，皆在 `bufio` 包中。

## 方法

### ScanBytes

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

`ScanBytes`是用于`Scanner`类型的分割函数（符合`SplitFunc`），本函数会将每个字节作为一个`token`返回。

### ScanRunes

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

`ScanRunes`是用于`Scanner`类型的分割函数（符合`SplitFunc`），本函数将每个utf-8编码的unicode码值作为一个`token`返回。本函数返回的`rune`序列和`range`一个字符串的输出`rune`序列相同。错误的utf-8编码会翻译为`U+FFFD`为“`\xef\xbf\xbd`”，但只会消耗一个字节。调用者无法区分正确编码的`rune`和错误编码的`rune`。

### ScanWords

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

`ScanRunes`是用于`Scanner`类型的分割函数（符合`SplitFunc`），本函数会将空白（参见`unicode.IsSpace`）分隔的片段（去掉前后空白后）作为一个`token`返回。本函数永远不会返回空字符串。用来找出`data`中的单行数据并返回（包括空行）。

### ScanLines

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

`ScanLines`是用于`Scanner`类型的分割函数（符合`SplitFunc`），本函数会将每一行文本去掉末尾的换行标记作为一个`token`返回。返回的行可以是空字符串。换行标记为一个可选的回车后跟一个必选的换行符。最后一行即使没有换行符也会作为一个`token`返回。

### NewScanner

```
func NewScanner(r io.Reader) *Scanner
```

`NewScanner`创建并返回一个从`r`读取数据的`Scanner`，默认的分隔函数是`ScanLines`。

### Split

```
func (s *Scanner) Split(split SplitFunc)
```

`Split`设置该`Scanner`的分隔函数。本方法必须在`Scan`之前调用。

示例：

```

package main

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

func main() {
    s := strings.NewReader("ABC DEF GHI JKL")
    bs := bufio.NewScanner(s)
    bs.Split(bufio.Scanwords)
    for bs.Scan() {
        fmt.Println(bs.Text())
    }
}
#结果
ABC
DEF
GHI
JKL

```

## Scan

```

func (s *Scanner) Scan() bool

```

Scan方法获取当前位置的token（该token可以通过Bytes或Text方法获得），并让Scanner的扫描位置移动到下一个token。当扫描因为抵达输入流结尾或者遇到错误而停止时，本方法会返回false。在Scan方法返回false后，Err方法将返回扫描时遇到的任何错误；除非是io.EOF，此时Err会返回nil。

示例：

```

package main

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

func main() {
    s := strings.NewReader("Hello 世界!")
    bs := bufio.NewScanner(s)
    bs.Split(bufio.ScanBytes)
    for bs.Scan() {
        fmt.Printf("%s", bs.Text())
    }
}
#结果
Hello 世界!

```

## Bytes

```
func (s *Scanner) Bytes() []byte
```

Bytes方法返回最近一次Scan调用生成的token。底层数组指向的数据可能会被下一次Scan的调用重写。

示例:

```
package main

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

func main() {
    s := strings.NewReader("Hello 世界!")
    bs := bufio.NewScanner(s)
    bs.Split(bufio.ScanRunes)
    for bs.Scan() {
        fmt.Printf("%s", bs.Text())
    }
}

#结果
Hello 世界!
```

## Text

```
func (s *Scanner) Text() string
```

返回最近一次 Scan 调用生成的 token，会申请创建一个字符串保存 token 并返回该字符串

## Err

```
func (s *Scanner) Err() error
```

Err返回Scanner遇到的第一个非EOF的错误。