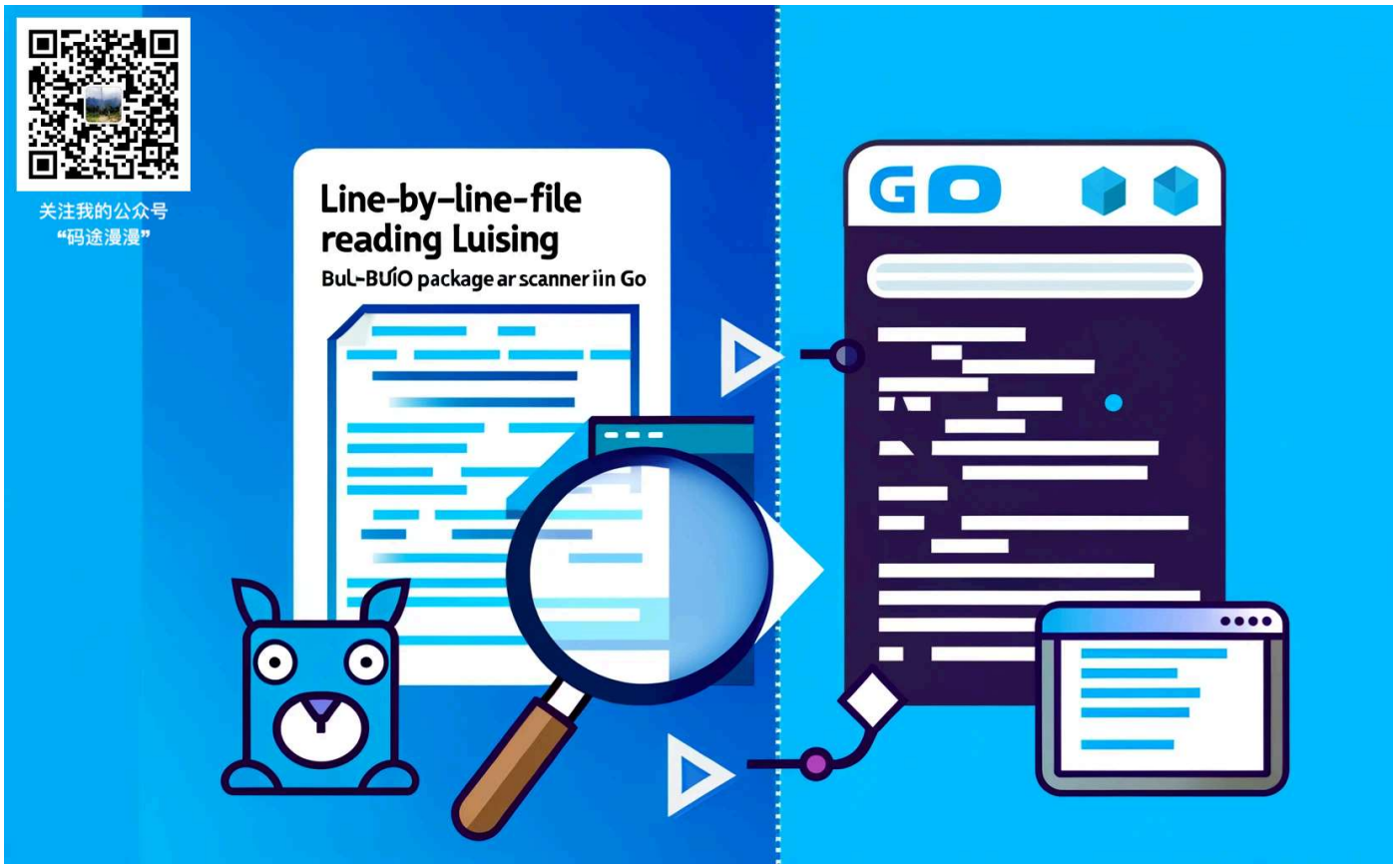
Go 如何按行读取（大）文件？尝试 bufio 包提供的几种方式

原创 波罗学 已于 2024-02-23 18:54:34 修改 阅读量1.7k 收藏 20 点赞数 19

文章标签： golang 爬虫 开发语言



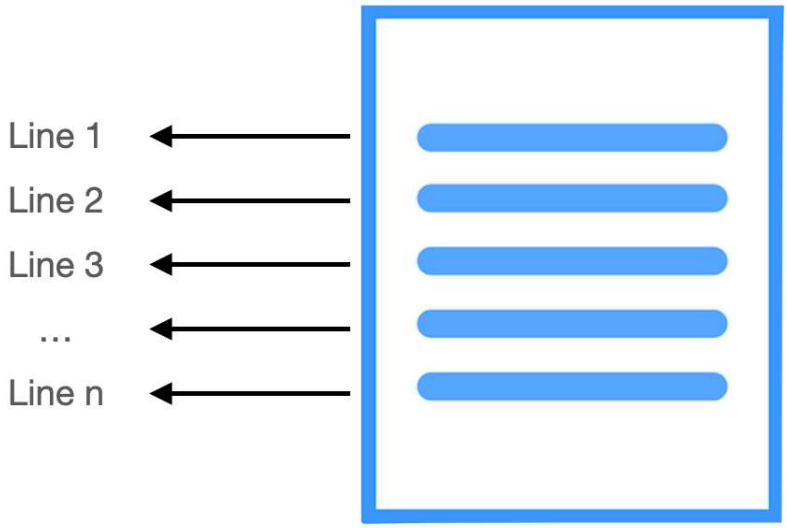
> 嗨，大家好！我是波罗学。本文是系列文章 Go 技巧第十七篇，系列文章查看：Go 语言技巧。

本文将介绍 Go 如何按行读取 文件，基于此会逐步延伸到如何按块 读取文件 。

引言

我们将要介绍的按行读取文件的方式其实是非常适合处理超大文件。

按行读取文件相较于一次性载入，有着很多优势，如内存效率高、处理速度快、实时性高、可扩展性强和灵活度高等，特别是当遇到处理大文件时，
更加明显。



稍微展开说下各个优势吧。

内存效率高，因为是按行读取，处理完一行就会丢弃，内存占用将大大减少。

处理速度快，主要体现在逐行处理时，因为无需等待全量数据，能更快开始，而且如果无顺序要求，还可 并行 计算以最大化利用计算资源，进一步度。

实时性高，因为按行读取，无需一次加载全量数据，自然有 **实时性高** 的特点，这对于处理实时流数据，如日志数据，非常有用。

可扩展性强，按行读取这种方式，不仅仅适用于小文件，大文件同样使用，有了统一的处理方式，即使未来数据量膨胀，也易于扩展。

灵活度高，因为是一行行的处理，如果想停止，随时可以。如果继续之前的流程，我们只要重新启动，从之前的位置继续处理即可。

按行读取其实只是按块读取的一种特殊形式（分隔符 是 \n），自然地，上述的优势也同样适用于按块读取文件。

本文的重点在于如何使用 GO 实现按行读取，基于的是标准库的 `bufio.Reader` 和 `bufio.Scanner` 。

正式进入主题吧。

准备一个文本文件

我们先准备一个文本文件 example.txt，内容如下：

```
This post covers the Golang Interface. Let’s dive into it.

Duck Typing

To understand Go’s interfaces, it’s crucial to grasp the Duck Typing concept.

So, what’s Duck Typing?
```

基于 `bufio.Reader`

Go 中的按行读取文件，首先可通过 `bufio` 提供的 `Reader` 类型实现。

使用 `Reader.ReadLine`

`Reader` 中有一个名为 `ReadLine` 的方法，顾名思义，它的作用就是按行读取文件的。

演示代码：

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

```

```
6
7
8   reader := bufio.NewReader(file)
9   for {
10      line, _, err := reader.ReadLine() // 按行读取文件
11      if err == io.EOF { // 用于判断文件是否读取到结尾
```

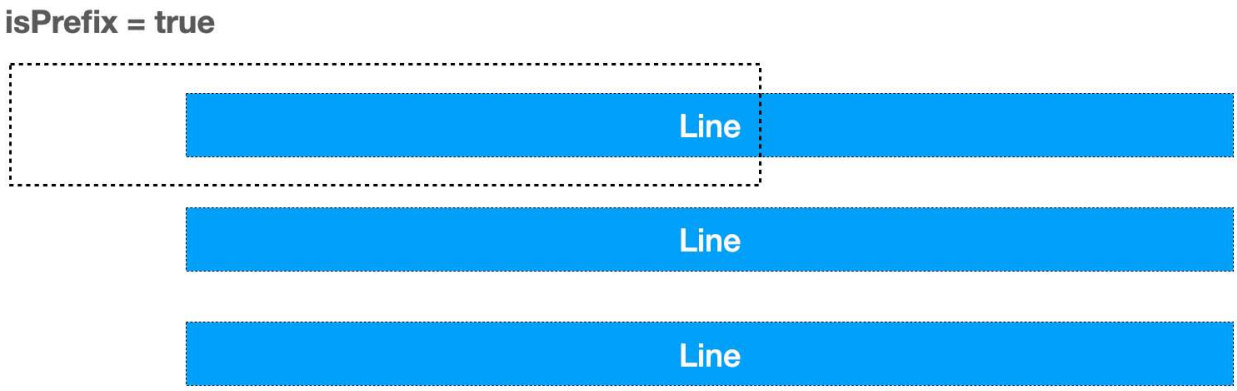
重点就是那句 `line, _, err := reader.ReadLine()`，返回值的第一值是读取的内容，第三个值是错误信息。

执行与输出：

```
$ go run main.go
This post covers the Golang Interface. Let’s dive into it.

Duck Typing

To understand Go’s interfaces, it’s crucial to grasp the Duck Typing concept.

So, what’s Duck Typing?
```

和我们预期的一样，输出了完整的文本信息。

要提醒的是， `ReadLine` 读取的内容不包括行尾符（如 “\r\n” 或 “\n”）。也就是说，当读取到一行数据时，要自行处理可能的行尾符差异，尤其是在处操作系统的文本数据时。

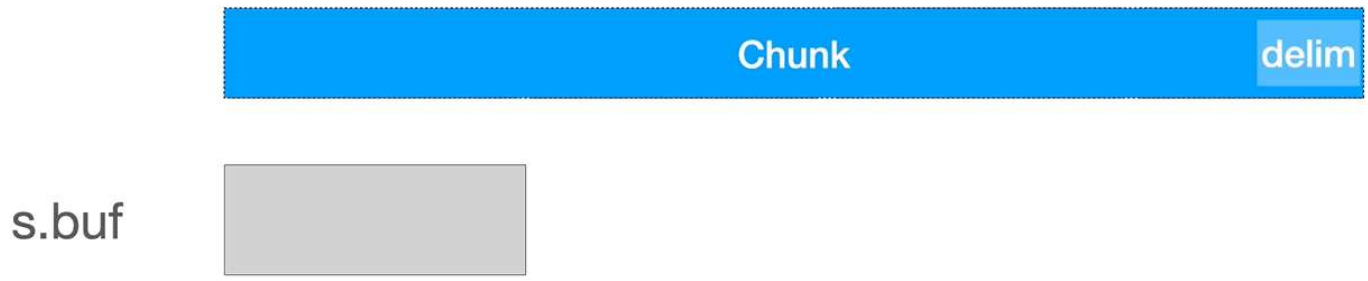


还有， `ReadLine` 省略的第二个参数，名为 `isPrefix`，它表示是否是前缀的意思，如果 `isPrefix` 为 true 表示返回的 `line` 被截断了，而截断原因行的内容大小大于缓冲区。我们可以在初始化时通过 `bufio.NewReaderSize(rd io.Reader, size int)` 调整默认缓冲区大小。

不过，这并非最优的解法。

使用 `Reader.ReadString`

解决大行读取被截断的问题，还可用 `bufio.Reader` 的另外一个方法 `ReadString` 解决。

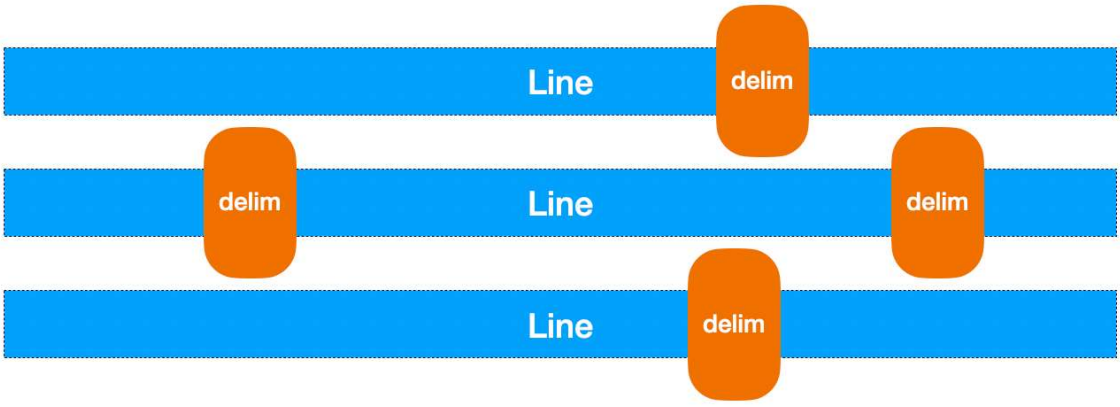


它与 `ReadLine` 类似，不过在单个 buffer 不足以容纳单行内容时，它会多次读取，直到找到目标分割符，合并多次读取的内容。

示例代码：

```
reader := bufio.NewReader(file)
for {
    line, err := reader.ReadString('\n')
    if err == io.EOF {
        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Printf("%s\n", line)
}
```

重点就是那句 `reader.ReadString('\n')`，它的入参是分割符（delim），即 '\n'，而返回值分别读取内容（line）和错误（err）。



相较于 `ReadLine`， `ReadString` 显然是更加灵活，无大行读取被截断的问题，而且分割符也可自定义。但只支持单一字节的分割符自定义，还不够们想按多个字符（如 `.|,` 等等）分割文本，或者按照大小分块读取，就没有那么方便了。

我们继续引入另一个 Go 标准提供的按行读取文件的方案，即 `bufio.Scanner`。

使用 bufio.Scanner

为了由浅入深地介绍 `bufio.Scanner` 的使用，我们还是先从 `bufio.Scanner` 实现按行读取讲起吧。

一个示例代码了解 `bufio.Scanner` 的基本使用。

```
// 创建文件的扫描器，用于逐行读取文件
scanner := bufio.NewScanner(file)
// 循环，直到文件结束
for scanner.Scan() {
```

```
    // 处理每行的内容：打印
    fmt.Println(scanner.Text())
}

// 最后，检查扫描过程中是否有错误发生
if err := scanner.Err(); err != nil {
    panic(err)
}
```

这个例子中，我们基于打开的文件描述符 `file`，创建了一个 `bufio.Scanner` 变量 `scanner`，它通过 `scanner.Scan()` 逐行扫描文件和 `scanner` buffer 中获取扫描内容，直到结束。

毫无疑问，相对于 `bufio.Reader`，以上通过 `bufio.Scanner` 实现的代码简洁很多，而且，错误处理 也是集中在 for 循环完成后统一进行。

如何读取大行?

`bufio.Scanner` 如何处理特别长的行呢?

默认情况下，`bufio.Scanner` 初始缓冲区是 4KB，而最大 token 大小是 64KB，即无法处理超过 64KB 的行。

来自源码中的定义，如下所示：

```
// `MaxScanTokenSize` 可定义 buffer 中 token 的最大 size，
// 除非用户通过 `Scanner.Buffer` 显式修改
// 缓冲区初始大小和 token 最大 size，
// 实际的最大标记大小可能会更小，因为
// 缓冲区可能需要包含例如换行符之类的内容。
MaxScanTokenSize = 64 * 1024
// 缓冲区的初始大小
startBufSize = 4096
```

`bufio.Scanner` 中提供了 `Scanner.Buffer()` 方法可用于调整默认的缓冲区。

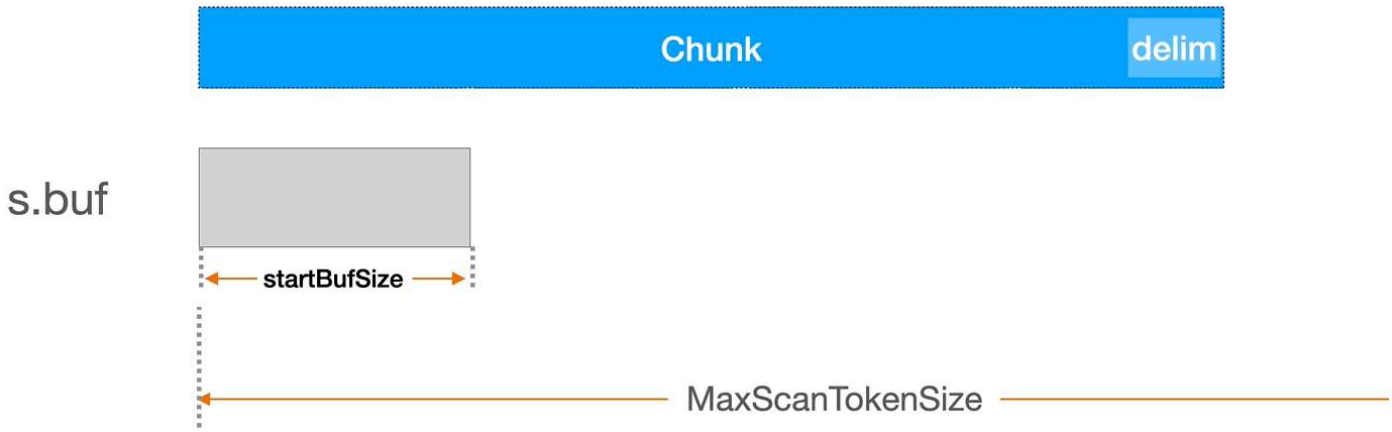
示例代码：

```
const maxCapacity = 1024 * 1024  // 例如，1MB，可读取任何 1MB 的行。
buf := make([]byte, maxCapacity) // 初始缓冲大小 1MB，无需多次扩容
scanner.Buffer(buf, maxCapacity)
```

在 `scanner` 扫描前，加上这段代码，会重新设置缓冲区，将初始缓冲大小和最大容易都设置为 1MB，这样就可以处理异常长的大行（size <= 1MB）由于初始缓冲区大小就是最大容量，也无需多次扩容缓冲。

缓冲区逻辑

为了更好理解上面的缓冲区配置，我简单介绍下 `bufio.Scanner` 是的 `Scan` 文件读取逻辑以及缓冲区是如何用的。



`bufio.Scanner` 内部有一个 `s.buf` 缓冲区，当我们调用 `scannder.Scan` 方法时，它会尝试用 `io.Reader`（即示例中的 `file` 文件描述符）中读取大小的内容。它的具体实现是在 `bufio.Scanner` 的 `Scan` 方法中。如果当缓冲区大小不足以容纳一个完整的 token，[illegible]冲区的大

接下来，让我们实现 `bufio.Scanner` 按单词读取。

扩展思路

如果每次都读取这么大块的一整行，和一次载入没有什么区别，这明显已经失去了开头介绍的一行行读取的优势了。

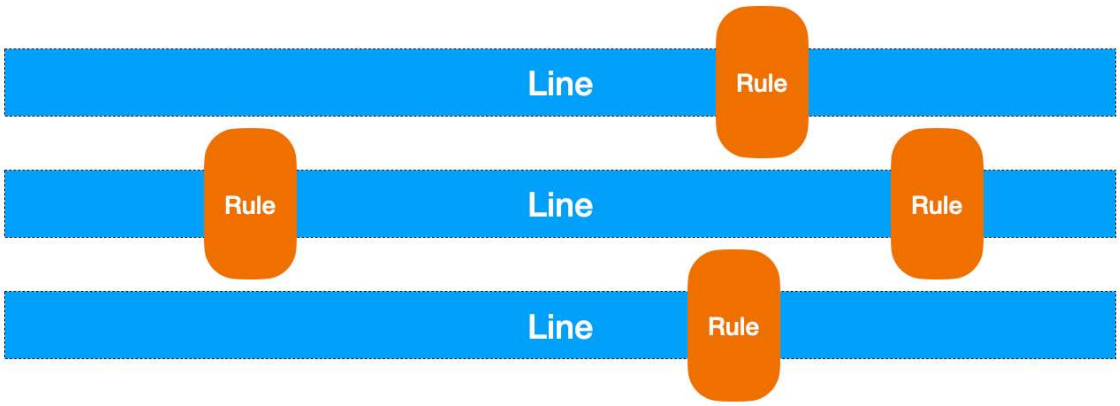
除了直接读取整行，是否还有什么更好的方法处理大行呢？

我们可以尝试解放一些思路，是否还有其他方式定义一次读取内容呢？我们只要保证读取的内容有实际含义即可，如按一句话，一个单词或者固定的
割，而非是纠结于是不是一整行。

分割规则定义

在正式介绍切割规则前，先说明下什么是完整 token。前面一直在说 token，如 `MaxScanTokenSize` 定义的就是 token 最大 size。

token 定义其实就是对一次读取内容的定义，如一行文本，一个单词，或者一个固定大小的块。相对于特定分隔符，分割规则更加灵活，可以定义任
式。



而 `bufio.Scanner` 是一个非常灵活的工具，它提供了自定义切割文本规则的 函数 - `Scanner.Split`。

```
// 参数
//  data []byte: 未处理数据的初始子串，当前需要处理的输入数据。
//  atEOF bool: 一个标志，如果为 true，则表示没有更多数据可处理。
// 返回值
//  advance int: 需要在输入中前进多少以到达下一个标记的起始位置。
//  token []byte: 要返回给用户的内容（如果有）。
//  err error: 扫描过程中遇到的错误。
type SplitFunc func(data []byte, atEOF bool) (advance int, token []byte, err error)
```

它的返回是分别读取内容的长度、读取的内容和错误信息。

默认情况下，Scanner 按行分割（ScanLines）。

```
scanner.Split(bufio.ScanLines) // 默认配置，按行读取
```

我们可以通过自定义的 Split 函数改变这个默认行为，如按单词分割。

示例代码：

```
const input = "This is a test. This is only a test."
scanner := bufio.NewScanner(strings.NewReader(input))

// 设置分割函数为按单词分割
scanner.Split(bufio.ScanWords)

// 逐个读取单词
for scanner.Scan() {
    fmt.Println(scanner.Text())
}

if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading input:", err)
}
```

输出：

```
This
is
a
test.
This
is
only
a
test.
```

现在，无论多大的文件，我们都可以通过巧妙定义切割方式来避免一次性读取的缺点了。

我之前利用 whispwer 识别油管视频的字幕，有些视频的内容非常长，超长字幕，都在一行。现在我就可以通过如句号、问号、感叹号分割即可。现在的定义这样一个 `ScanSentences` 函数。

示例代码：

```go
func ScanSentences(data []byte, atEOF bool) (advance int, token []byte, err error) {
    // 如果我们处于 EOF 并且有数据，则返回剩余的数据
    if atEOF && len(data) > 0 {
        return len(data), data, nil
    }

    // 定义一个查找任意句子结束符的函数
    findSentenceEnd := func(data []byte) int {
        // 检查每个可能的句子结束符
        endIndex := -1
        for _, sep := range []byte{'.', '?', '!'} {
```

我们写个 `main` 函数测试下 `ScanSentences` 的正确性吧。

示例代码：

```go
func main() {
    const input = "This is a test. This is only a test. Is this a test? \n" +
        "Wow, what a brilliant test! Thanks for your help."
    scanner := bufio.NewScanner(strings.NewReader(input))
    scanner.Split(ScanSentences)
    for scanner.Scan() {
        text := scanner.Text()
        fmt.Printf("%s\n", strings.TrimSpace(text))
    }

    if err := scanner.Err(); err != nil {
        panic(err)
    }
}
```

执行输出：

```
$ go run main.go
This is a test.
This is only a test.
Is this a test?
Wow, what a brilliant test!
Thanks for your help.
```

或者按照固定大小分批读取文件， `SplitBatchSize` 示例代码：

```go
// ScanBatchSize 返回一个 bufio.SplitFunc 函数，该函数按照固定的大小分割数据。
// 如果数据大小不足一个完整的批次，并且已经到
func ScanBatchSize(batchSize int) bufi
```

```
    return func(data []byte, atEOF bool) (advance int, token []byte, err error) {
        // 如果数据大小达到或超过批次大小，或者在 EOF 时有剩余数据
        if len(data) >= batchSize || (atEOF && len(data) > 0) {
            // 如果当前批次大小超过剩余数据大小，则只返回剩余数据
            if len(data) < batchSize {
                return len(data), data[:], nil
            }
```

`SplitBatchSize` 是一个闭包，它的返回值是我们期待的 `SplitFunc` 。我们可传递参数配置每次读取内容的大小。具体可自行测试，这里就演示了。

不得不说

到这里，我还是想再提一点，每次从文件中读取内容大小是由传入系统调用 read() 函数时传入参数 buf 大小决定的，而不是由所谓按行还是按块确定块是基于读取出来的二次处理的结果。

之所以要提这点，因为我之前看到一些文章说，按块相比按行读取减少了读取的次数。

结论

本文详细介绍了在 Go 中如何使用 bufio.Reader 和 bufio.Scanner 按行或按块读取文件，通过利用 GO 的标准库能力，我们有了更加灵活、高效处文件的策略。

最后，感谢阅读，希望本文对你有所帮助。