# bufio

package | standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 5 | Imported by: 211,049

**Details** Valid go.mod file | Redistributable license | Tagged version | Stable version

Learn more

**Repository** cs.opensource.google/go/go

**Links** Report a Vulnerability

Documentation

## Documentation

### Overview

Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.

### Index

func (b *Reader) ReadRune() (r rune, size int, err error)
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
func (b *Reader) ReadString(delim byte) (string, error)
func (b *Reader) Reset(r io.Reader)
func (b *Reader) Size() int
func (b *Reader) UnreadByte() error
func (b *Reader) UnreadRune() error
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
type Scanner
func NewScanner(r io.Reader) *Scanner
func (s *Scanner) Buffer(buf []byte, max int)
func (s *Scanner) Bytes() []byte
func (s *Scanner) Err() error
func (s *Scanner) Scan() bool
func (s *Scanner) Split(split SplitFunc)
func (s *Scanner) Text() string
type SplitFunc
type Writer
func NewWriter(w io.Writer) *Writer
func NewWriterSize(w io.Writer, size int) *Writer
func (b *Writer) Available() int
func (b *Writer) AvailableBuffer() []byte
func (b *Writer) Buffered() int
func (b *Writer) Flush() error
func (b *Writer) ReadFrom(r io.Reader) (n int64, err error)
func (b *Writer) Reset(w io.Writer)
func (b *Writer) Size() int
func (b *Writer) Write(p []byte) (nn int, err error)
func (b *Writer) WriteByte(c byte) error
func (b *Writer) WriteRune(r rune) (size int, err error)
func (b *Writer) WriteString(s string) (int, error)

### Examples

Scanner (Custom)
Scanner (EmptyFinalToken)
Scanner (Lines)
Scanner (Words)
Scanner.Bytes
Writer
Writer.AvailableBuffer

## Constants

View Source

```
const (
    // MaxScanTokenSize is the maximum size used to buffer a token
    // unless the user provides an explicit buffer with Scanner.Buffer.
```

```
    // The actual maximum token size may be smaller as the buffer
    // may need to include, for instance, a newline.
    MaxScanTokenSize = 64 * 1024
)
```

## Variables

View Source

```
var (
    ErrInvalidUnreadByte = errors.New("bufio: invalid use of UnreadByte")
    ErrInvalidUnreadRune = errors.New("bufio: invalid use of UnreadRune")
    ErrBufferFull        = errors.New("bufio: buffer full")
    ErrNegativeCount     = errors.New("bufio: negative count")
)
```

View Source

```
var (
    ErrTooLong         = errors.New("bufio.Scanner: token too long")
    ErrNegativeAdvance = errors.New("bufio.Scanner: SplitFunc returns negative advance
    ErrAdvanceTooFar   = errors.New("bufio.Scanner: SplitFunc returns advance count bey
    ErrBadReadCount    = errors.New("bufio.Scanner: Read returned impossible count")
)
```

Errors returned by Scanner.

View Source

```
var ErrFinalToken = errors.New("final token")
```

ErrFinalToken is a special sentinel error value. It is intended to be returned by a Split function to indicate that the token being delivered with the error is the last token and scanning should stop after this one. After ErrFinalToken is received by Scan, scanning stops with no error. The value is useful to stop processing early or when it is necessary to deliver a final empty token. One could achieve the same behavior with a custom error value but providing one here is tidier. See the emptyFinalToken example for a use of this value.

## Functions

### func ScanBytes

added in go1.1

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

ScanBytes is a split function for a Scanner that returns each byte as a token.

### func ScanLines

added in go1.1

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

ScanLines is a split function for a Scanner that returns each line of text, stripped of any trailing end-of-line marker. The returned line may be empty. The end-of-line marker is one optional carriage return followed by one mandatory newline. In regular expression notation, it is `\r?\n`. The last non-empty line of input will be returned even if it has no newline.

### func ScanRunes

added in go1.1

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

ScanRunes is a split function for a Scanner that returns each UTF-8-encoded rune as a token. The sequence of runes returned is equivalent to that from a range loop over the input as a string, which means that erroneous UTF-8 encodings translate to U+FFFD = "\xef\xbf\xbd". Because of the Scan interface, this makes it impossible for the client to distinguish correctly encoded replacement runes from encoding errors.

### func ScanWords

added in go1.1

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

ScanWords is a split function for a Scanner that returns each space-separated word of text, with surrounding spaces deleted. It will never return an empty string. The definition of space is set by unicode.IsSpace.

## Types

### type ReadWriter

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

ReadWriter stores pointers to a Reader and a Writer. It implements io.ReadWriter.

### func NewReadWriter

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

NewReadWriter allocates a new ReadWriter that dispatches to r and w.

### type Reader

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

Reader implements buffering for an io.Reader object.

### func NewReader

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

NewReader returns a new Reader whose buffer has the default size.

### func NewReaderSize

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

NewReaderSize returns a new Reader whose buffer has at least the specified size. If the argument io.Reader is already a Reader with large enough size, it returns the underlying Reader.

### func (*Reader) Buffered

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

Buffered returns the number of bytes that can be read from the current buffer.

### func (*Reader) Discard

added in go1.5

```
func (b *Reader) Discard(n int) (discarded int, err error)
```

Discard skips the next n bytes, returning the number of bytes discarded.

If Discard skips fewer than n bytes, it also returns an error. If 0 <= n <= b.Buffered(), Discard is guaranteed to succeed without reading from the underlying io.Reader.

### func (*Reader) Peek

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

Peek returns the next n bytes without advancing the reader. The bytes stop being valid at the next read call. If Peek returns fewer than n bytes, it also returns an error explaining why the read is short. The error is ErrBufferFull if n is larger than b's buffer size.

Calling Peek prevents a UnreadByte or UnreadRune call from succeeding until the next read operation.

### func (*Reader) Read

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

Read reads data into p. It returns the number of bytes read into p. The bytes are taken from at most one Read on the underlying Reader, hence n may be less than len(p). To read exactly len(p) bytes, use io.ReadFull(b, p). If the underlying Reader can return a non-zero count with io.EOF, then this Read method can do so as well; see the io.Reader docs.

### func (*Reader) ReadByte

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

ReadByte reads and returns a single byte. If no byte is available, returns an error.

### func (*Reader) ReadBytes

```
func (b *Reader) ReadBytes(delim byte) ([]byte, error)
```

ReadBytes reads until the first occurrence of delim in the input, returning a slice containing the data up to and including the delimiter. If ReadBytes encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadBytes returns err != nil if and only if the returned data does not end in delim. For simple uses, a Scanner may be more convenient.

### func (*Reader) ReadLine

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

ReadLine is a low-level line-reading primitive. Most callers should use ReadBytes('\n') or ReadString('\n') instead or use a Scanner.

ReadLine tries to return a single line, not including the end-of-line bytes. If the line was too long for the buffer then isPrefix is set and the beginning of the line is returned. The rest of the line will be returned from future calls. isPrefix will be false when returning the last fragment of the line. The returned buffer is only valid until the next call to ReadLine. ReadLine either returns a non-nil line or it returns an error, never both.

The text returned from ReadLine does not include the line end ("\r\n" or "\n"). No indication or error is given if the input ends without a final line end. Calling UnreadByte after ReadLine will always unread the last byte read (possibly a character belonging to the line end) even if that byte is not part of the line returned by ReadLine.

### func (*Reader) ReadRune

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

ReadRune reads a single UTF-8 encoded Unicode character and returns the rune and its size in bytes. If the encoded rune is invalid, it consumes one byte and returns unicode.ReplacementChar (U+FFFD) with a size of 1.

### func (*Reader) ReadSlice

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

ReadSlice reads until the first occurrence of delim in the input, returning a slice pointing at the bytes in the buffer. The bytes stop being valid at the next read. If ReadSlice encounters an error before finding a delimiter, it returns all the data in the buffer and the error itself (often io.EOF). ReadSlice fails with error ErrBufferFull if the buffer fills without a delim. Because the data returned from ReadSlice will be

overwritten by the next I/O operation, most clients should use ReadBytes or ReadString instead. ReadSlice returns err != nil if and only if line does not end in delim.

### func (*Reader) ReadString

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

ReadString reads until the first occurrence of delim in the input, returning a string containing the data up to and including the delimiter. If ReadString encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadString returns err != nil if and only if the returned data does not end in delim. For simple uses, a Scanner may be more convenient.

### func (*Reader) Reset

added in go1.2

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

Reset discards any buffered data, resets all state, and switches the buffered reader to read from r. Calling Reset on the zero value of Reader initializes the internal buffer to the default size.

### func (*Reader) Size

added in go1.10

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

Size returns the size of the underlying buffer in bytes.

### func (*Reader) UnreadByte

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

UnreadByte unreads the last byte. Only the most recently read byte can be unread.

UnreadByte returns an error if the most recent method called on the Reader was not a read operation. Notably, Peek, Discard, and WriteTo are not considered read operations.

### func (*Reader) UnreadRune

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

UnreadRune unreads the last rune. If the most recent method called on the Reader was not a ReadRune, UnreadRune returns an error. (In this regard it is stricter than UnreadByte, which will unread the last byte from any read operation.)

### func (*Reader) WriteTo

added in go1.1

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

WriteTo implements io.WriterTo. This may make multiple calls to the Read method of the underlying Reader. If the underlying reader supports the WriteTo method, this calls the underlying WriteTo without

buffering.

### type Scanner

added in go1.1

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

Scanner provides a convenient interface for reading data such as a file of newline-delimited lines of text. Successive calls to the Scan method will step through the 'tokens' of a file, skipping the bytes between the tokens. The specification of a token is defined by a split function of type SplitFunc; the default split function breaks the input into lines with line termination stripped. Split functions are defined in this package for scanning a file into lines, bytes, UTF-8-encoded runes, and space-delimited words. The client may instead provide a custom split function.

Scanning stops unrecoverably at EOF, the first I/O error, or a token too large to fit in the buffer. When a scan stops, the reader may have advanced arbitrarily far past the last token. Programs that need more control over error handling or large tokens, or must run sequential scans on a reader, should use bufio.Reader instead.

▶ Example (Custom)

▶ Example (EmptyFinalToken)

▶ Example (Lines)

▶ Example (Words)

### func NewScanner

added in go1.1

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

NewScanner returns a new Scanner to read from r. The split function defaults to ScanLines.

### func (*Scanner) Buffer

added in go1.6

```
func (s *Scanner) Buffer(buf []byte, max int)
```

Buffer sets the initial buffer to use when scanning and the maximum size of buffer that may be allocated during scanning. The maximum token size is the larger of max and cap(buf). If max <= cap(buf), Scan will use this buffer only and do no allocation.

By default, Scan uses an internal buffer and sets the maximum token size to MaxScanTokenSize.

Buffer panics if it is called after scanning has started.

### func (*Scanner) Bytes

added in go1.1

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

Bytes returns the most recent token generated by a call to Scan. The underlying array may point to data that will be overwritten by a subsequent call to Scan. It does no allocation.

▶ Example

### func (*Scanner) Err

added in go1.1

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

Err returns the first non-EOF error that was encountered by the Scanner.

### func (*Scanner) Scan

added in go1.1

```
func (s *Scanner) Scan() bool
```

Scan advances the Scanner to the next token, which will then be available through the Bytes or Text method. It returns false when the scan stops, either by reaching the end of the input or an error. After Scan returns false, the Err method will return any error that occurred during scanning, except that if it was io.EOF, Err will return nil. Scan panics if the split function returns too many empty tokens without advancing the input. This is a common error mode for scanners.

### func (*Scanner) Split

added in go1.1

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

Split sets the split function for the Scanner. The default split function is ScanLines.

Split panics if it is called after scanning has started.

### func (*Scanner) Text

added in go1.1

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

Text returns the most recent token generated by a call to Scan as a newly allocated string holding its bytes.

### type SplitFunc

added in go1.1

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

SplitFunc is the signature of the split function used to tokenize the input. The arguments are an initial substring of the remaining unprocessed data and a flag, atEOF, that reports whether the Reader has no

more data to give. The return values are the number of bytes to advance the input and the next token to return to the user, if any, plus an error, if any.

Scanning stops if the function returns an error, in which case some of the input may be discarded. If that error is ErrFinalToken, scanning stops with no error.

Otherwise, the Scanner advances the input. If the token is not nil, the Scanner returns it to the user. If the token is nil, the Scanner reads more data and continues scanning; if there is no more data--if atEOF was true--the Scanner returns. If the data does not yet hold a complete token, for instance if it has no newline while scanning lines, a SplitFunc can return (0, nil, nil) to signal the Scanner to read more data into the slice and try again with a longer slice starting at the same point in the input.

The function is never called with an empty data slice unless atEOF is true. If atEOF is true, however, data may be non-empty and, as always, holds unprocessed text.

### type Writer

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

Writer implements buffering for an io.Writer object. If an error occurs writing to a Writer, no more data will be accepted and all subsequent writes, and Flush, will return the error. After all data has been written, the client should call the Flush method to guarantee all data has been forwarded to the underlying io.Writer.

▶ Example

### func NewWriter

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

NewWriter returns a new Writer whose buffer has the default size. If the argument io.Writer is already a Writer with large enough buffer size, it returns the underlying Writer.

### func NewWriterSize

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

NewWriterSize returns a new Writer whose buffer has at least the specified size. If the argument io.Writer is already a Writer with large enough size, it returns the underlying Writer.

### func (*Writer) Available

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

Available returns how many bytes are unused in the buffer.

### func (*Writer) AvailableBuffer

added in go1.18

```
func (b *Writer) AvailableBuffer() []byte
```

AvailableBuffer returns an empty buffer with b.Available() capacity. This buffer is intended to be appended to and passed to an immediately succeeding Write call. The buffer is only valid until the next write operation on b.

▶ Example

### func (*Writer) Buffered

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

Buffered returns the number of bytes that have been written into the current buffer.

### func (*Writer) Flush

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

Flush writes any buffered data to the underlying io.Writer.

### func (*Writer) ReadFrom

added in go1.1

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

ReadFrom implements io.ReaderFrom. If the underlying writer supports the ReadFrom method, this calls the underlying ReadFrom. If there is buffered data and an underlying ReadFrom, this fills the buffer and writes it before calling ReadFrom.

### func (*Writer) Reset

added in go1.2

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

Reset discards any unflushed buffered data, clears any error, and resets b to write its output to w. Calling Reset on the zero value of Writer initializes the internal buffer to the default size.

### func (*Writer) Size

added in go1.10

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

Size returns the size of the underlying buffer in bytes.

### func (*Writer) Write

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

Write writes the contents of p into the buffer. It returns the number of bytes written. If nn < len(p), it also returns an error explaining why the write is short.

### func (*Writer) WriteByte

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

WriteByte writes a single byte.

### func (*Writer) WriteRune

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

WriteRune writes a single Unicode code point, returning the number of bytes written and any error.

### func (*Writer) WriteString

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

WriteString writes a string. It returns the number of bytes written. If the count is less than len(s), it also returns an error explaining why the write is short.

## Source Files

View all

bufio.go scan.go