io

Package io go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package io provides basic interfaces to I/O primitives. Its primary job is to wrap existing implementations of such primitives, such as those in package os, into shared public interfaces that abstract the functionality, plus some other related primitives.

Because these interfaces and primitives wrap lower-level operations with various implementations, unless otherwise informed clients should not assume they are safe for parallel execution.

Constants

```
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
)
```

Seek whence values.

Variables

```
var EOF = errors.New("EOF")
```

EOF is the error returned by Read when no more input is available. Functions should return EOF only to signal a graceful end of input. If the EOF occurs unexpectedly in a structured data stream, the appropriate error is either ErrUnexpectedEOF or some other error giving more detail.

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

ErrClosedPipe is the error used for read or write operations on a closed pipe.

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

ErrNoProgress is returned by some clients of an io.Reader when many calls to Read have failed to return any data or error, usually the sign of a broken io.Reader implementation.

```
var ErrShortBuffer = errors.New("short buffer")
```

ErrShortBuffer means that a read required a longer buffer than was provided.

```
var ErrShortWrite = errors.New("short write")
```

ErrShortWrite means that a write accepted fewer bytes than requested but failed to return an explicit error.

```
var ErrUnexpectedEOF = errors.New("unexpected EOF")
```

ErrUnexpectedEOF means that EOF was encountered in the middle of reading a fixed-size block or data structure.

func Copy

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

Copy copies from src to dst until either EOF is reached on src or an error occurs. It returns the number of bytes copied and the first error encountered while copying, if any.

A successful Copy returns err == nil, not err == EOF. Because Copy is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

If src implements the WriterTo interface, the copy is implemented by calling src.WriteTo(dst). Otherwise, if dst implements the ReaderFrom interface, the copy is implemented by calling dst.ReadFrom(src).

func CopyBuffer

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

CopyBuffer is identical to Copy except that it stages through the provided buffer (if one is required) rather than allocating a temporary one. If buf is nil, one is allocated; otherwise if it has zero length, CopyBuffer panics.

If either src implements WriterTo or dst implements ReaderFrom, buf will not be used to perform the copy.

func CopyN

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

CopyN copies n bytes (or until an error) from src to dst. It returns the number of bytes copied and the earliest error encountered while copying. On return, written == n if and only if err == nil.

If dst implements the ReaderFrom interface, the copy is implemented using it.

func Pipe

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

Pipe creates a synchronous in-memory pipe. It can be used to connect code expecting an io.Reader with code expecting an io.Writer.

Reads and Writes on the pipe are matched one to one except when multiple Reads are needed to consume a single Write. That is, each Write to the PipeWriter blocks until it has satisfied one or more Reads from the PipeReader that fully consume the written data. The data is copied directly from the Write to the corresponding Read (or Reads); there is no internal buffering.

It is safe to call Read and Write in parallel with each other or with Close. Parallel calls to Read and parallel calls to Write are also safe: the individual calls will be gated sequentially.

func ReadAtLeast

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

ReadAtLeast reads from r into buf until it has read at least min bytes. It returns the number of bytes copied and an error if fewer bytes were read. The error is EOF only if no bytes were read. If an EOF happens after reading fewer than min bytes, ReadAtLeast returns ErrUnexpectedEOF. If min is greater than the length of buf, ReadAtLeast returns ErrShortBuffer. On return, n >= min if and only if err == nil. If r returns an error having read at least min bytes, the error is dropped.

func ReadFull

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

ReadFull reads exactly len(buf) bytes from r into buf. It returns the number of bytes copied and an error if fewer bytes were read. The error is EOF only if no bytes were read. If an EOF happens after reading some but not all the bytes, ReadFull returns ErrUnexpectedEOF. On return, n == len(buf) if and only if err == nil. If r returns an error having read at least len(buf) bytes, the error is dropped.

func WriteString

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

WriteString writes the contents of the string s to w, which accepts a slice of bytes. If w implements StringWriter, its WriteString method is invoked directly. Otherwise, w.Write is called exactly once.

type ByteReader

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

ByteReader is the interface that wraps the ReadByte method.

ReadByte reads and returns the next byte from the input or any error encountered. If ReadByte returns an error, no input byte was consumed, and the returned byte value is undefined.

ReadByte provides an efficient interface for byte-at-time processing. A Reader that does not implement ByteReader can be wrapped using bufio.NewReader to add this method.

type ByteScanner

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

ByteScanner is the interface that adds the UnreadByte method to the basic ReadByte method.

UnreadByte causes the next call to ReadByte to return the same byte as the previous call to ReadByte. It may be an error to call UnreadByte twice without an intervening call to ReadByte.

type ByteWriter

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

ByteWriter is the interface that wraps the WriteByte method.

type Closer

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

Closer is the interface that wraps the basic Close method.

The behavior of Close after the first call is undefined. Specific implementations may document their own behavior.

type LimitedReader

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

A LimitedReader reads from R but limits the amount of data returned to just N bytes. Each call to Read updates N to reflect the new amount remaining. Read returns EOF when N <= 0 or when the underlying R returns EOF.

func (*LimitedReader) Read

```
func (l *LimitedReader) Read(p []byte) (n int, err error)
```

type PipeReader

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

A PipeReader is the read half of a pipe.

func (*PipeReader) Close

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

Close closes the reader; subsequent writes to the write half of the pipe will return the error ErrClosedPipe.

func (*PipeReader) CloseWithError

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

CloseWithError closes the reader; subsequent writes to the write half of the pipe will return the error err.

CloseWithError never overwrites the previous error if it exists and always returns nil.

func (*PipeReader) Read

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

Read implements the standard Read interface: it reads data from the pipe, blocking until a writer arrives or the write end is closed. If the write end is closed with an error, that error is returned as err; otherwise err is EOF.

type PipeWriter

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

A PipeWriter is the write half of a pipe.

func (*PipeWriter) Close

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

Close closes the writer; subsequent reads from the read half of the pipe will return no bytes and EOF.

func (*PipeWriter) CloseWithError

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

CloseWithError closes the writer; subsequent reads from the read half of the pipe will return no bytes and the error err, or EOF if err is nil.

CloseWithError never overwrites the previous error if it exists and always returns nil.

func (*PipeWriter) Write

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

Write implements the standard Write interface: it writes data to the pipe, blocking until one or more readers have consumed all the data or the read end is closed. If the read end is closed with an error, that err is returned as err; otherwise err is ErrClosedPipe.

type ReadCloser

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

ReadCloser is the interface that groups the basic Read and Close methods.

type ReadSeeker

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

ReadSeeker is the interface that groups the basic Read and Seek methods.

type ReadWriteCloser

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

ReadWriteCloser is the interface that groups the basic Read, Write and Close methods.

type ReadWriteSeeker

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

ReadWriteSeeker is the interface that groups the basic Read, Write and Seek methods.

type ReadWriter

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

ReadWriter is the interface that groups the basic Read and Write methods.

type Reader

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

Reader is the interface that wraps the basic Read method.

Read reads up to len(p) bytes into p. It returns the number of bytes read (0 <= n <= len(p)) and any error encountered. Even if Read returns n < len(p), it may use all of p as scratch space during the call. If some data is available but not len(p) bytes, Read conventionally returns what is available instead of waiting for more.

When Read encounters an error or end-of-file condition after successfully reading n > 0 bytes, it returns the number of bytes read. It may return the (non-nil) error from the same call or return the error (and n == 0) from a subsequent call. An instance of this general case is that a Reader returning a non-zero number of bytes at the end of the input stream may return either err == EOF or err == nil. The next Read should return 0, EOF.

Callers should always process the n > 0 bytes returned before considering the error err. Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviors.

Implementations of Read are discouraged from returning a zero byte count with a nil error, except when len(p) == 0. Callers should treat a return of 0 and nil as indicating that nothing happened; in particular it does not indicate EOF.

Implementations must not retain p.

func LimitReader

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

LimitReader returns a Reader that reads from r but stops with EOF after n bytes. The underlying implementation is a *LimitedReader.

func MultiReader

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

MultiReader returns a Reader that's the logical concatenation of the provided input readers. They're read sequentially. Once all inputs have returned EOF, Read will return EOF. If any of the readers return a non-nil, non-EOF error, Read will return that error.

func TeeReader

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

TeeReader returns a Reader that writes to w what it reads from r. All reads from r performed through it are matched with corresponding writes to w. There is no internal buffering - the write must complete before the read completes. Any error encountered while writing is reported as a read error.

type ReaderAt

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

ReaderAt is the interface that wraps the basic ReadAt method.

ReadAt reads len(p) bytes into p starting at offset off in the underlying input source. It returns the number of bytes read (0 <= n <= len(p)) and any error encountered.

When ReadAt returns n < len(p), it returns a non-nil error explaining why more bytes were not returned. In this respect, ReadAt is stricter than Read.

Even if ReadAt returns n < len(p), it may use all of p as scratch space during the call. If some data is available but not len(p) bytes, ReadAt blocks until either all the data is available or an error occurs. In this respect ReadAt is different from Read.

If the n = len(p) bytes returned by ReadAt are at the end of the input source, ReadAt may return either err == EOF or err == nil.

If ReadAt is reading from an input source with a seek offset, ReadAt should not affect nor be affected by the underlying seek offset.

Clients of ReadAt can execute parallel ReadAt calls on the same input source.

Implementations must not retain p.

type ReaderFrom

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

ReaderFrom is the interface that wraps the ReadFrom method.

ReadFrom reads data from r until EOF or error. The return value n is the number of bytes read. Any error except io.EOF encountered during the read is also returned.

The Copy function uses ReaderFrom if available.

type RuneReader

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

RuneReader is the interface that wraps the ReadRune method.

ReadRune reads a single UTF-8 encoded Unicode character and returns the rune and its size in bytes. If no character is available, err will be set.

type RuneScanner

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

RuneScanner is the interface that adds the UnreadRune method to the basic ReadRune method.

UnreadRune causes the next call to ReadRune to return the same rune as the previous call to ReadRune. It may be an error to call UnreadRune twice without an intervening call to ReadRune.

type SectionReader

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

SectionReader implements Read, Seek, and ReadAt on a section of an underlying ReaderAt.

func NewSectionReader

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

NewSectionReader returns a SectionReader that reads from r starting at offset off and stops with EOF after n bytes.

func (*SectionReader) Read

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

func (*SectionReader) ReadAt

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

func (*SectionReader) Seek

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

func (*SectionReader) Size

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

Size returns the size of the section in bytes.

type Seeker

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

Seeker is the interface that wraps the basic Seek method.

Seek sets the offset for the next Read or Write to offset, interpreted according to whence: SeekStart means relative to the start of the file, SeekCurrent means relative to the current offset, and SeekEnd means relative to the end. Seek returns the new offset relative to the start of the file and an error, if any.

Seeking to an offset before the start of the file is an error. Seeking to any positive offset is legal, but the behavior of subsequent I/O operations on the underlying object is implementation-dependent.

type StringWriter

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

StringWriter is the interface that wraps the WriteString method.

type WriteCloser

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

WriteCloser is the interface that groups the basic Write and Close methods.

type WriteSeeker

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

WriteSeeker is the interface that groups the basic Write and Seek methods.

type Writer

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

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p). Write must not modify the slice data, even temporarily.

Implementations must not retain p.

func MultiWriter

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

MultiWriter creates a writer that duplicates its writes to all the provided writers, similar to the Unix tee(1) command.

Each write is written to each listed writer, one at a time. If a listed writer returns an error, that overall write operation stops and returns the error; it does not continue down the list.

type WriterAt

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

WriterAt is the interface that wraps the basic WriteAt method.

WriteAt writes len(p) bytes from p to the underlying data stream at offset off. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. WriteAt must return a non-nil error if it returns n < len(p).

If WriteAt is writing to a destination with a seek offset, WriteAt should not affect nor be affected by the underlying seek offset.

Clients of WriteAt can execute parallel WriteAt calls on the same destination if the ranges do not overlap.

Implementations must not retain p.

type WriterTo

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

WriterTo is the interface that wraps the WriteTo method.

WriteTo writes data to w until there's no more data to write or when an error occurs. The return value n is the number of bytes written. Any error encountered during the write is also returned.

The Copy function uses WriterTo if available.