

Package base64

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)[Overview](#)[Subdirectories](#)[Versions](#)[Imports](#)[Imported By](#)[Licenses](#)

Overview

Package base64 implements base64 encoding as specified by [RFC 4648](#).

Constants

```
const (
    StdPadding rune = '=' // Standard padding character
    NoPadding  rune = -1 // No padding
)
```

Variables

```
var RawStdEncoding = StdEncoding.WithPadding(NoPadding)
```

RawStdEncoding is the standard raw, unpadded base64 encoding, as defined in [RFC 4648 section 3.2](#). This is the same as StdEncoding but omits padding characters.

```
var RawURLEncoding = URLEncoding.WithPadding(NoPadding)
```

RawURLEncoding is the unpadded alternate base64 encoding defined in [RFC 4648](#). It is typically used in URLs and file names. This is the same as URLEncoding but omits padding characters.

```
var StdEncoding = NewEncoding(encodeStd)
```

StdEncoding is the standard base64 encoding, as defined in [RFC 4648](#).

```
var URLEncoding = NewEncoding(encodeURL)
```

URLEncoding is the alternate base64 encoding defined in [RFC 4648](#). It is typically used in URLs and file names.

func NewDecoder

```
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

NewDecoder constructs a new base64 stream decoder.

func `NewEncoder`

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
```

NewEncoder returns a new base64 stream encoder. Data written to the returned writer will be encoded using enc and then written to w. Base64 encodings operate in 4-byte blocks; when finished writing, the caller must Close the returned encoder to flush any partially written blocks.

type `CorruptInputError`

```
type CorruptInputError int64
```

func (`CorruptInputError`) `Error`

```
func (e CorruptInputError) Error() string
```

type `Encoding`

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

An Encoding is a radix 64 encoding/decoding scheme, defined by a 64-character alphabet. The most common encoding is the "base64" encoding defined in [RFC 4648](#) and used in MIME ([RFC 2045](#)) and PEM ([RFC 1421](#)). [RFC 4648](#) also defines an alternate encoding, which is the standard encoding with - and _ substituted for + and /.

func `NewEncoding`

```
func NewEncoding(encoder string) *Encoding
```

NewEncoding returns a new padded Encoding defined by the given alphabet, which must be a 64-byte string that does not contain the padding character or CR / LF ('\r', '\n'). The resulting Encoding uses the default padding character ('='), which may be changed or disabled via WithPadding.

func (`*Encoding`) `Decode`

```
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)
```

Decode decodes src using the encoding enc. It writes at most DecodedLen(len(src)) bytes to dst and returns the number of bytes written. If src contains invalid base64 data, it will return the number of bytes successfully written and CorruptInputError. New line characters (\r and \n) are ignored.

func (`*Encoding`) `DecodeString`

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

DecodeString returns the bytes represented by the base64 string s.

func (*Encoding) DecodedLen

```
func (enc *Encoding) DecodedLen(n int) int
```

DecodedLen returns the maximum length in bytes of the decoded data corresponding to n bytes of base64-encoded data.

func (*Encoding) Encode

```
func (enc *Encoding) Encode(dst, src []byte)
```

Encode encodes src using the encoding enc, writing EncodedLen(len(src)) bytes to dst.

The encoding pads the output to a multiple of 4 bytes, so Encode is not appropriate for use on individual blocks of a large data stream. Use NewEncoder() instead.

func (*Encoding) EncodeToString

```
func (enc *Encoding) EncodeToString(src []byte) string
```

EncodeToString returns the base64 encoding of src.

func (*Encoding) EncodedLen

```
func (enc *Encoding) EncodedLen(n int) int
```

EncodedLen returns the length in bytes of the base64 encoding of an input buffer of length n.

func (Encoding) Strict

```
func (enc Encoding) Strict() *Encoding
```

Strict creates a new encoding identical to enc except with strict decoding enabled. In this mode, the decoder requires that trailing padding bits are zero, as described in [RFC 4648 section 3.5](#).

Note that the input is still malleable, as new line characters (CR and LF) are still ignored.

func (Encoding) WithPadding

```
func (enc Encoding) WithPadding(padding rune) *Encoding
```

WithPadding creates a new encoding identical to enc except with a specified padding character, or NoPadding to disable padding. The padding character must not be '\r' or '\n', must not be contained in the encoding's alphabet and must be a rune equal or below '\xff'.

Package bytes

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

Overview

Package bytes implements functions for the manipulation of byte slices. It is analogous to the facilities of the strings package.

Constants

```
const MinRead = 512
```

MinRead is the minimum slice size passed to a Read call by Buffer.ReadFrom. As long as the Buffer has at least MinRead bytes beyond what is required to hold the contents of r, ReadFrom will not grow the underlying buffer.

Variables

```
var ErrTooLarge = errors.New("bytes.Buffer: too large")
```

ErrTooLarge is passed to panic if memory cannot be allocated to store data in a buffer.

func Compare

```
func Compare(a, b []byte) int
```

Compare returns an integer comparing two byte slices lexicographically. The result will be 0 if a==b, -1 if a < b, and +1 if a > b. A nil argument is equivalent to an empty slice.

func Contains

```
func Contains(b, subslice []byte) bool
```

Contains reports whether subslice is within b.

func ContainsAny

```
func ContainsAny(b []byte, chars string) bool
```

ContainsAny reports whether any of the UTF-8-encoded code points in chars are within b.

func ContainsRune

```
func ContainsRune(b []byte, r rune) bool
```

ContainsRune reports whether the rune is contained in the UTF-8-encoded byte slice b.

func Count

```
func Count(s, sep []byte) int
```

Count counts the number of non-overlapping instances of sep in s. If sep is an empty slice, Count returns 1 + the number of UTF-8-encoded code points in s.

func Equal

```
func Equal(a, b []byte) bool
```

Equal reports whether a and b are the same length and contain the same bytes. A nil argument is equivalent to an empty slice.

func EqualFold

```
func EqualFold(s, t []byte) bool
```

EqualFold reports whether s and t, interpreted as UTF-8 strings, are equal under Unicode case-folding, which is a more general form of case-insensitivity.

func Fields

```
func Fields(s []byte) [][]byte
```

Fields interprets s as a sequence of UTF-8-encoded code points. It splits the slice s around each instance of one or more consecutive white space characters, as defined by `unicode.IsSpace`, returning a slice of subslices of s or an empty slice if s contains only white space.

func FieldsFunc

```
func FieldsFunc(s []byte, f func(rune) bool) [][]byte
```

FieldsFunc interprets s as a sequence of UTF-8-encoded code points. It splits the slice s at each run of code points c satisfying f(c) and returns a slice of subslices of s. If all code points in s satisfy f(c), or `len(s) == 0`, an empty slice is returned.

FieldsFunc makes no guarantees about the order in which it calls f(c) and assumes that f always returns the same value for a given c.

func HasPrefix

```
func HasPrefix(s, prefix []byte) bool
```

HasPrefix tests whether the byte slice s begins with prefix.

func HasSuffix

```
func HasSuffix(s, suffix []byte) bool
```

HasSuffix tests whether the byte slice s ends with suffix.

func Index

```
func Index(s, sep []byte) int
```

Index returns the index of the first instance of sep in s, or -1 if sep is not present in s.

func IndexAny

```
func IndexAny(s []byte, chars string) int
```

IndexAny interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index of the first occurrence in s of any of the Unicode code points in chars. It returns -1 if chars is empty or if there is no code point in common.

func IndexByte

```
func IndexByte(b []byte, c byte) int
```

IndexByte returns the index of the first instance of c in b, or -1 if c is not present in b.

func IndexFunc

```
func IndexFunc(s []byte, f func(r rune) bool) int
```

IndexFunc interprets s as a sequence of UTF-8-encoded code points. It returns the byte index in s of the first Unicode code point satisfying f(c), or -1 if none do.

func IndexRune

```
func IndexRune(s []byte, r rune) int
```

IndexRune interprets s as a sequence of UTF-8-encoded code points. It returns the byte index of the first occurrence in s of the given rune. It returns -1 if rune is not present in s. If r is utf8.RuneError, it

returns the first instance of any invalid UTF-8 byte sequence.

func `Join`

```
func Join(s [][]byte, sep []byte) []byte
```

`Join` concatenates the elements of `s` to create a new byte slice. The separator `sep` is placed between elements in the resulting slice.

func `LastIndex`

```
func LastIndex(s, sep []byte) int
```

`LastIndex` returns the index of the last instance of `sep` in `s`, or `-1` if `sep` is not present in `s`.

func `LastIndexAny`

```
func LastIndexAny(s []byte, chars string) int
```

`LastIndexAny` interprets `s` as a sequence of UTF-8-encoded Unicode code points. It returns the byte index of the last occurrence in `s` of any of the Unicode code points in `chars`. It returns `-1` if `chars` is empty or if there is no code point in common.

func `LastIndexByte`

```
func LastIndexByte(s []byte, c byte) int
```

`LastIndexByte` returns the index of the last instance of `c` in `s`, or `-1` if `c` is not present in `s`.

func `LastIndexFunc`

```
func LastIndexFunc(s []byte, f func(r rune) bool) int
```

`LastIndexFunc` interprets `s` as a sequence of UTF-8-encoded code points. It returns the byte index in `s` of the last Unicode code point satisfying `f(c)`, or `-1` if none do.

func `Map`

```
func Map(mapping func(r rune) rune, s []byte) []byte
```

`Map` returns a copy of the byte slice `s` with all its characters modified according to the mapping function. If `mapping` returns a negative value, the character is dropped from the byte slice with no replacement. The characters in `s` and the output are interpreted as UTF-8-encoded code points.

func `Repeat`

```
func Repeat(b []byte, count int) []byte
```

Repeat returns a new byte slice consisting of count copies of b.

It panics if count is negative or if the result of (len(b) * count) overflows.

func Replace

```
func Replace(s, old, new []byte, n int) []byte
```

Replace returns a copy of the slice s with the first n non-overlapping instances of old replaced by new. If old is empty, it matches at the beginning of the slice and after each UTF-8 sequence, yielding up to k+1 replacements for a k-rune slice. If n < 0, there is no limit on the number of replacements.

func ReplaceAll

```
func ReplaceAll(s, old, new []byte) []byte
```

ReplaceAll returns a copy of the slice s with all non-overlapping instances of old replaced by new. If old is empty, it matches at the beginning of the slice and after each UTF-8 sequence, yielding up to k+1 replacements for a k-rune slice.

func Runes

```
func Runes(s []byte) []rune
```

Runes interprets s as a sequence of UTF-8-encoded code points. It returns a slice of runes (Unicode code points) equivalent to s.

func Split

```
func Split(s, sep []byte) [][]byte
```

Split slices s into all subslices separated by sep and returns a slice of the subslices between those separators. If sep is empty, Split splits after each UTF-8 sequence. It is equivalent to SplitN with a count of -1.

func SplitAfter

```
func SplitAfter(s, sep []byte) [][]byte
```

SplitAfter slices s into all subslices after each instance of sep and returns a slice of those subslices. If sep is empty, SplitAfter splits after each UTF-8 sequence. It is equivalent to SplitAfterN with a count of -1.

func `SplitAfterN`

```
func SplitAfterN(s, sep []byte, n int) [][]byte
```

`SplitAfterN` slices `s` into subslices after each instance of `sep` and returns a slice of those subslices. If `sep` is empty, `SplitAfterN` splits after each UTF-8 sequence. The count determines the number of subslices to return:

```
n > 0: at most n subslices; the last subslice will be the unsplit remainder.  
n == 0: the result is nil (zero subslices)  
n < 0: all subslices
```

func `SplitN`

```
func SplitN(s, sep []byte, n int) [][]byte
```

`SplitN` slices `s` into subslices separated by `sep` and returns a slice of the subslices between those separators. If `sep` is empty, `SplitN` splits after each UTF-8 sequence. The count determines the number of subslices to return:

```
n > 0: at most n subslices; the last subslice will be the unsplit remainder.  
n == 0: the result is nil (zero subslices)  
n < 0: all subslices
```

func `Title`

```
func Title(s []byte) []byte
```

`Title` treats `s` as UTF-8-encoded bytes and returns a copy with all Unicode letters that begin words mapped to their title case.

BUG(rsc): The rule `Title` uses for word boundaries does not handle Unicode punctuation properly.

func `ToLower`

```
func ToLower(s []byte) []byte
```

`ToLower` returns a copy of the byte slice `s` with all Unicode letters mapped to their lower case.

func `ToLowerSpecial`

```
func ToLowerSpecial(c unicode.SpecialCase, s []byte) []byte
```

`ToLowerSpecial` treats `s` as UTF-8-encoded bytes and returns a copy with all the Unicode letters mapped to their lower case, giving priority to the special casing rules.

func `ToTitle`

```
func ToTitle(s []byte) []byte
```

ToTitle treats s as UTF-8-encoded bytes and returns a copy with all the Unicode letters mapped to their title case.

func `ToTitleSpecial`

```
func ToTitleSpecial(c unicode.SpecialCase, s []byte) []byte
```

ToTitleSpecial treats s as UTF-8-encoded bytes and returns a copy with all the Unicode letters mapped to their title case, giving priority to the special casing rules.

func `ToUpper`

```
func ToUpper(s []byte) []byte
```

ToUpper returns a copy of the byte slice s with all Unicode letters mapped to their upper case.

func `ToUpperSpecial`

```
func ToUpperSpecial(c unicode.SpecialCase, s []byte) []byte
```

ToUpperSpecial treats s as UTF-8-encoded bytes and returns a copy with all the Unicode letters mapped to their upper case, giving priority to the special casing rules.

func `ToValidUTF8`

```
func ToValidUTF8(s, replacement []byte) []byte
```

ToValidUTF8 treats s as UTF-8-encoded bytes and returns a copy with each run of bytes representing invalid UTF-8 replaced with the bytes in replacement, which may be empty.

func `Trim`

```
func Trim(s []byte, cutset string) []byte
```

Trim returns a subslice of s by slicing off all leading and trailing UTF-8-encoded code points contained in cutset.

func `TrimFunc`

```
func TrimFunc(s []byte, f func(r rune) bool) []byte
```

TrimFunc returns a subslice of `s` by slicing off all leading and trailing UTF-8-encoded code points `c` that satisfy `f(c)`.

func TrimLeft

```
func TrimLeft(s []byte, cutset string) []byte
```

TrimLeft returns a subslice of `s` by slicing off all leading UTF-8-encoded code points contained in `cutset`.

func TrimLeftFunc

```
func TrimLeftFunc(s []byte, f func(rune) bool) []byte
```

TrimLeftFunc treats `s` as UTF-8-encoded bytes and returns a subslice of `s` by slicing off all leading UTF-8-encoded code points `c` that satisfy `f(c)`.

func TrimPrefix

```
func TrimPrefix(s, prefix []byte) []byte
```

TrimPrefix returns `s` without the provided leading prefix string. If `s` doesn't start with `prefix`, `s` is returned unchanged.

func TrimRight

```
func TrimRight(s []byte, cutset string) []byte
```

TrimRight returns a subslice of `s` by slicing off all trailing UTF-8-encoded code points that are contained in `cutset`.

func TrimRightFunc

```
func TrimRightFunc(s []byte, f func(rune) bool) []byte
```

TrimRightFunc returns a subslice of `s` by slicing off all trailing UTF-8-encoded code points `c` that satisfy `f(c)`.

func TrimSpace

```
func TrimSpace(s []byte) []byte
```

TrimSpace returns a subslice of `s` by slicing off all leading and trailing white space, as defined by Unicode.

func TrimSuffix

```
func TrimSuffix(s, suffix []byte) []byte
```

TrimSuffix returns s without the provided trailing suffix string. If s doesn't end with suffix, s is returned unchanged.

type Buffer

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

A Buffer is a variable-sized buffer of bytes with Read and Write methods. The zero value for Buffer is an empty buffer ready to use.

func NewBuffer

```
func NewBuffer(buf []byte) *Buffer
```

NewBuffer creates and initializes a new Buffer using buf as its initial contents. The new Buffer takes ownership of buf, and the caller should not use buf after this call. NewBuffer is intended to prepare a Buffer to read existing data. It can also be used to set the initial size of the internal buffer for writing. To do that, buf should have the desired capacity but a length of zero.

In most cases, new(Buffer) (or just declaring a Buffer variable) is sufficient to initialize a Buffer.

func NewBufferString

```
func NewBufferString(s string) *Buffer
```

NewBufferString creates and initializes a new Buffer using string s as its initial contents. It is intended to prepare a buffer to read an existing string.

In most cases, new(Buffer) (or just declaring a Buffer variable) is sufficient to initialize a Buffer.

func (*Buffer) Bytes

```
func (b *Buffer) Bytes() []byte
```

Bytes returns a slice of length b.Len() holding the unread portion of the buffer. The slice is valid for use only until the next buffer modification (that is, only until the next call to a method like Read, Write, Reset, or Truncate). The slice aliases the buffer content at least until the next buffer modification, so immediate changes to the slice will affect the result of future reads.

func (*Buffer) Cap

```
func (b *Buffer) Cap() int
```

Cap returns the capacity of the buffer's underlying byte slice, that is, the total space allocated for the buffer's data.

func (*Buffer) Grow

```
func (b *Buffer) Grow(n int)
```

Grow grows the buffer's capacity, if necessary, to guarantee space for another n bytes. After Grow(n), at least n bytes can be written to the buffer without another allocation. If n is negative, Grow will panic. If the buffer can't grow it will panic with ErrTooLarge.

func (*Buffer) Len

```
func (b *Buffer) Len() int
```

Len returns the number of bytes of the unread portion of the buffer; b.Len() == len(b.Bytes()).

func (*Buffer) Next

```
func (b *Buffer) Next(n int) []byte
```

Next returns a slice containing the next n bytes from the buffer, advancing the buffer as if the bytes had been returned by Read. If there are fewer than n bytes in the buffer, Next returns the entire buffer. The slice is only valid until the next call to a read or write method.

func (*Buffer) Read

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

Read reads the next len(p) bytes from the buffer or until the buffer is drained. The return value n is the number of bytes read. If the buffer has no data to return, err is io.EOF (unless len(p) is zero); otherwise it is nil.

func (*Buffer) ReadByte

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

ReadByte reads and returns the next byte from the buffer. If no byte is available, it returns error io.EOF.

func (*Buffer) ReadBytes

```
func (b *Buffer) ReadBytes(delim byte) (line []byte, err 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`.

func (*Buffer) ReadFrom

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

`ReadFrom` reads data from `r` until `EOF` and appends it to the buffer, growing the buffer as needed. The return value `n` is the number of bytes read. Any error except `io.EOF` encountered during the read is also returned. If the buffer becomes too large, `ReadFrom` will panic with `ErrTooLarge`.

func (*Buffer) ReadRune

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

`ReadRune` reads and returns the next UTF-8-encoded Unicode code point from the buffer. If no bytes are available, the error returned is `io.EOF`. If the bytes are an erroneous UTF-8 encoding, it consumes one byte and returns `U+FFFD`, 1.

func (*Buffer) ReadString

```
func (b *Buffer) ReadString(delim byte) (line string, err 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`.

func (*Buffer) Reset

```
func (b *Buffer) Reset()
```

`Reset` resets the buffer to be empty, but it retains the underlying storage for use by future writes. `Reset` is the same as `Truncate(0)`.

func (*Buffer) String

```
func (b *Buffer) String() string
```

`String` returns the contents of the unread portion of the buffer as a string. If the `Buffer` is a nil pointer, it returns "`<nil>`".

To build strings more efficiently, see the `strings.Builder` type.

func (*Buffer) Truncate

```
func (b *Buffer) Truncate(n int)
```

Truncate discards all but the first n unread bytes from the buffer but continues to use the same allocated storage. It panics if n is negative or greater than the length of the buffer.

func (*Buffer) UnreadByte

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

UnreadByte unreads the last byte returned by the most recent successful read operation that read at least one byte. If a write has happened since the last read, if the last read returned an error, or if the read read zero bytes, UnreadByte returns an error.

func (*Buffer) UnreadRune

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

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

func (*Buffer) Write

```
func (b *Buffer) Write(p []byte) (n int, err error)
```

Write appends the contents of p to the buffer, growing the buffer as needed. The return value n is the length of p; err is always nil. If the buffer becomes too large, Write will panic with ErrTooLarge.

func (*Buffer) WriteByte

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

WriteByte appends the byte c to the buffer, growing the buffer as needed. The returned error is always nil, but is included to match bufio.Writer's WriteByte. If the buffer becomes too large, WriteByte will panic with ErrTooLarge.

func (*Buffer) WriteRune

```
func (b *Buffer) WriteRune(r rune) (n int, err error)
```

WriteRune appends the UTF-8 encoding of Unicode code point r to the buffer, returning its length and an error, which is always nil but is included to match bufio.Writer's WriteRune. The buffer is grown as needed; if it becomes too large, WriteRune will panic with ErrTooLarge.

func (*Buffer) `WriteString`

```
func (b *Buffer) WriteString(s string) (n int, err error)
```

`WriteString` appends the contents of `s` to the buffer, growing the buffer as needed. The return value `n` is the length of `s`; `err` is always `nil`. If the buffer becomes too large, `WriteString` will panic with `ErrTooLarge`.

func (*Buffer) `WriteTo`

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

`WriteTo` writes data to `w` until the buffer is drained or an error occurs. The return value `n` is the number of bytes written; it always fits into an `int`, but it is `int64` to match the `io.WriterTo` interface. Any error encountered during the write is also returned.

type `Reader`

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

A `Reader` implements the `io.Reader`, `io.ReaderAt`, `io.WriterTo`, `io.Seeker`, `io.ByteScanner`, and `io.RuneScanner` interfaces by reading from a byte slice. Unlike a `Buffer`, a `Reader` is read-only and supports seeking. The zero value for `Reader` operates like a `Reader` of an empty slice.

func `NewReader`

```
func NewReader(b []byte) *Reader
```

`NewReader` returns a new `Reader` reading from `b`.

func (*Reader) `Len`

```
func (r *Reader) Len() int
```

`Len` returns the number of bytes of the unread portion of the slice.

func (*Reader) `Read`

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

`Read` implements the `io.Reader` interface.

func (*Reader) `ReadAt`

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt implements the io.ReaderAt interface.

func (*Reader) ReadByte

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

ReadByte implements the io.ByteReader interface.

func (*Reader) ReadRune

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

ReadRune implements the io.RuneReader interface.

func (*Reader) Reset

```
func (r *Reader) Reset(b []byte)
```

Reset resets the Reader to be reading from b.

func (*Reader) Seek

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek implements the io.Seeker interface.

func (*Reader) Size

```
func (r *Reader) Size() int64
```

Size returns the original length of the underlying byte slice. Size is the number of bytes available for reading via ReadAt. The returned value is always the same and is not affected by calls to any other method.

func (*Reader) UnreadByte

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

UnreadByte complements ReadByte in implementing the io.ByteScanner interface.

func (*Reader) UnreadRune

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

UnreadRune complements ReadRune in implementing the io.RuneScanner interface.

func (*Reader) **WriteTo**

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

WriteTo implements the io.WriterTo interface.

BUGs

- The rule Title uses for word boundaries does not handle Unicode punctuation properly.

Package context

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

Overview

Package context defines the `Context` type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

Incoming requests to a server should create a `Context`, and outgoing calls to servers should accept a `Context`. The chain of function calls between them must propagate the `Context`, optionally replacing it with a derived `Context` created using `WithCancel`, `WithDeadline`, `WithTimeout`, or `WithValue`. When a `Context` is canceled, all `Contexts` derived from it are also canceled.

The `WithCancel`, `WithDeadline`, and `WithTimeout` functions take a `Context` (the parent) and return a derived `Context` (the child) and a `CancelFunc`. Calling the `CancelFunc` cancels the child and its children, removes the parent's reference to the child, and stops any associated timers. Failing to call the `CancelFunc` leaks the child and its children until the parent is canceled or the timer fires. The go vet tool checks that `CancelFuncs` are used on all control-flow paths.

Programs that use `Contexts` should follow these rules to keep interfaces consistent across packages and enable static analysis tools to check context propagation:

Do not store `Contexts` inside a struct type; instead, pass a `Context` explicitly to each function that needs it. The `Context` should be the first parameter, typically named `ctx`:

```
func DoSomething(ctx context.Context, arg Arg) error {
    // ... use ctx ...
}
```

Do not pass a nil `Context`, even if a function permits it. Pass `context.TODO` if you are unsure about which `Context` to use.

Use `context.Values` only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

The same `Context` may be passed to functions running in different goroutines; `Contexts` are safe for simultaneous use by multiple goroutines.

See <https://blog.golang.org/context> for example code for a server that uses `Contexts`.

Variables

```
var Canceled = errors.New("context canceled")
```

Canceled is the error returned by Context.Err when the context is canceled.

```
var DeadlineExceeded error = deadlineExceededError{}
```

DeadlineExceeded is the error returned by Context.Err when the context's deadline passes.

func [WithCancel](#)

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

WithCancel returns a copy of parent with a new Done channel. The returned context's Done channel is closed when the returned cancel function is called or when the parent context's Done channel is closed, whichever happens first.

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete.

func [WithDeadline](#)

```
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
```

WithDeadline returns a copy of the parent context with the deadline adjusted to be no later than d. If the parent's deadline is already earlier than d, WithDeadline(parent, d) is semantically equivalent to parent. The returned context's Done channel is closed when the deadline expires, when the returned cancel function is called, or when the parent context's Done channel is closed, whichever happens first.

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete.

func [WithTimeout](#)

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

WithTimeout returns WithDeadline(parent, time.Now().Add(timeout)).

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete:

```
func slowOperationWithTimeout(ctx context.Context) (Result, error) {
    ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
    defer cancel() // releases resources if slowOperation completes before timeout
    return slowOperation(ctx)
}
```

type [CancelFunc](#)

```
type CancelFunc func()
```

A CancelFunc tells an operation to abandon its work. A CancelFunc does not wait for the work to stop. A CancelFunc may be called by multiple goroutines simultaneously. After the first call, subsequent calls to a CancelFunc do nothing.

type Context

```
type Context interface {
    // Deadline returns the time when work done on behalf of this context
    // should be canceled. Deadline returns ok==false when no deadline is
    // set. Successive calls to Deadline return the same results.
    Deadline() (deadline time.Time, ok bool)

    // Done returns a channel that's closed when work done on behalf of this
    // context should be canceled. Done may return nil if this context can
    // never be canceled. Successive calls to Done return the same value.
    // The close of the Done channel may happen asynchronously,
    // after the cancel function returns.
    //
    // WithCancel arranges for Done to be closed when cancel is called;
    // WithDeadline arranges for Done to be closed when the deadline
    // expires; WithTimeout arranges for Done to be closed when the timeout
    // elapses.
    //
    // Done is provided for use in select statements:
    //
    // // Stream generates values with DoSomething and sends them to out
    // // until DoSomething returns an error or ctx.Done is closed.
    // func Stream(ctx context.Context, out chan<- Value) error {
    //     for {
    //         v, err := DoSomething(ctx)
    //         if err != nil {
    //             return err
    //         }
    //         select {
    //             case <-ctx.Done():
    //                 return ctx.Err()
    //             case out <- v:
    //         }
    //     }
    //
    //     // See https://blog.golang.org/pipelines for more examples of how to use
    //     // a Done channel for cancellation.
    Done() <-chan struct{}

    // If Done is not yet closed, Err returns nil.
    // If Done is closed, Err returns a non-nil error explaining why:
    // Canceled if the context was canceled
    // or DeadlineExceeded if the context's deadline passed.
    // After Err returns a non-nil error, successive calls to Err return the same err
```

```

Err() error

// Value returns the value associated with this context for key, or nil
// if no value is associated with key. Successive calls to Value with
// the same key returns the same result.
//
// Use context values only for request-scoped data that transits
// processes and API boundaries, not for passing optional parameters to
// functions.
//
// A key identifies a specific value in a Context. Functions that wish
// to store values in Context typically allocate a key in a global
// variable then use that key as the argument to contextWithValue and
// Context.Value. A key can be any type that supports equality;
// packages should define keys as an unexported type to avoid
// collisions.
//
// Packages that define a Context key should provide type-safe accessors
// for the values stored using that key:
//
// // Package user defines a User type that's stored in Contexts.
// package user
//
// import "context"
//
// // User is the type of value stored in the Contexts.
// type User struct {...}
//
// // key is an unexported type for keys defined in this package.
// // This prevents collisions with keys defined in other packages.
// type key int
//
// // userKey is the key for user.User values in Contexts. It is
// // unexported; clients use user.NewContext and user.FromContext
// // instead of using this key directly.
// var userKey key
//
// // NewContext returns a new Context that carries value u.
// func NewContext(ctx context.Context, u *User) context.Context {
//     return contextWithValue(ctx, userKey, u)
// }
//
// // FromContext returns the User value stored in ctx, if any.
// func FromContext(ctx context.Context) (*User, bool) {
//     u, ok := ctx.Value(userKey).(*User)
//     return u, ok
// }
Value(key interface{}) interface{}
}

```

A Context carries a deadline, a cancellation signal, and other values across API boundaries.

Context's methods may be called by multiple goroutines simultaneously.

func Background

```
func Background() Context
```

Background returns a non-nil, empty Context. It is never canceled, has no values, and has no deadline. It is typically used by the main function, initialization, and tests, and as the top-level Context for incoming requests.

func TODO

```
func TODO() Context
```

TODO returns a non-nil, empty Context. Code should use context.TODO when it's unclear which Context to use or it is not yet available (because the surrounding function has not yet been extended to accept a Context parameter).

func WithValue

```
func WithValue(parent Context, key, val interface{}) Context
```

WithValue returns a copy of parent in which the value associated with key is val.

Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

The provided key must be comparable and should not be of type string or any other built-in type to avoid collisions between packages using context. Users of WithValue should define their own types for keys. To avoid allocating when assigning to an interface{}, context keys often have concrete type struct{}. Alternatively, exported context key variables' static type should be a pointer or interface.

Package errors

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

Overview

Package errors implements functions to manipulate errors.

The `New` function creates errors whose only content is a text message.

The `Unwrap`, `Is` and `As` functions work on errors that may wrap other errors. An error wraps another error if its type has the method

```
Unwrap() error
```

If `e.Unwrap()` returns a non-nil error `w`, then we say that `e` wraps `w`.

`Unwrap` unpacks wrapped errors. If its argument's type has an `Unwrap` method, it calls the method once. Otherwise, it returns nil.

A simple way to create wrapped errors is to call `fmt.Errorf` and apply the `%w` verb to the error argument:

```
errors.Unwrap(fmt.Errorf("... %w ...", ..., err, ...))
```

returns `err`.

`Is` unwraps its first argument sequentially looking for an error that matches the second. It reports whether it finds a match. It should be used in preference to simple equality checks:

```
if errors.Is(err, os.ErrExist)
```

is preferable to

```
if err == os.ErrExist
```

because the former will succeed if `err` wraps `os.ErrExist`.

`As` unwraps its first argument sequentially looking for an error that can be assigned to its second argument, which must be a pointer. If it succeeds, it performs the assignment and returns true. Otherwise, it returns false. The form

```
var perr *os.PathError
if errors.As(err, &perr) {
    fmt.Println(perr.Path)
}
```

is preferable to

```
if perr, ok := err.(*os.PathError); ok {
    fmt.Println(perr.Path)
}
```

because the former will succeed if err wraps an `*os.PathError`.

func As

```
func As(err error, target interface{}) bool
```

As finds the first error in err's chain that matches target, and if so, sets target to that error value and returns true. Otherwise, it returns false.

The chain consists of err itself followed by the sequence of errors obtained by repeatedly calling `Unwrap`.

An error matches target if the error's concrete value is assignable to the value pointed to by target, or if the error has a method `As(interface{}) bool` such that `As(target)` returns true. In the latter case, the `As` method is responsible for setting target.

An error type might provide an `As` method so it can be treated as if it were a different error type.

As panics if target is not a non-nil pointer to either a type that implements `error`, or to any interface type.

func Is

```
func Is(err, target error) bool
```

Is reports whether any error in err's chain matches target.

The chain consists of err itself followed by the sequence of errors obtained by repeatedly calling `Unwrap`.

An error is considered to match a target if it is equal to that target or if it implements a method `Is(error) bool` such that `Is(target)` returns true.

An error type might provide an `Is` method so it can be treated as equivalent to an existing error. For example, if `MyError` defines

```
func (m MyError) Is(target error) bool { return target == os.ErrExist }
```

then `Is(MyError{}, os.ErrExist)` returns true. See `syscall.Errno.Is` for an example in the standard library.

func `New`

```
func New(text string) error
```

`New` returns an error that formats as the given text. Each call to `New` returns a distinct error value even if the text is identical.

func `Unwrap`

```
func Unwrap(err error) error
```

`Unwrap` returns the result of calling the `Unwrap` method on `err`, if `err`'s type contains an `Unwrap` method returning `error`. Otherwise, `Unwrap` returns `nil`.

Overview

Package flag implements command-line flag parsing.

Usage

Define flags using `flag.String()`, `Bool()`, `Int()`, etc.

This declares an integer flag, `-n`, stored in the pointer `nFlag`, with type `*int`:

```
import "flag"
var nFlag = flag.Int("n", 1234, "help message for flag n")
```

If you like, you can bind the flag to a variable using the `Var()` functions.

```
var flagvar int
func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
}
```

Or you can create custom flags that satisfy the `Value` interface (with pointer receivers) and couple them to flag parsing by

```
flag.Var(&flagVal, "name", "help message for flagname")
```

For such flags, the default value is just the initial value of the variable.

After all flags are defined, call

```
flag.Parse()
```

to parse the command line into the defined flags.

Flags may then be used directly. If you're using the flags themselves, they are all pointers; if you bind to variables, they're values.

```
fmt.Println("ip has value ", *ip)
fmt.Println("flagvar has value ", flagvar)
```

After parsing, the arguments following the flags are available as the slice `flag.Args()` or individually as `flag.Arg(i)`. The arguments are indexed from 0 through `flag.NArg()-1`.

Command line flag syntax

The following forms are permitted:

```
-flag  
-flag=x  
-flag x // non-boolean flags only
```

One or two minus signs may be used; they are equivalent. The last form is not permitted for boolean flags because the meaning of the command

```
cmd -x *
```

where `*` is a Unix shell wildcard, will change if there is a file called 0, false, etc. You must use the `-flag=false` form to turn off a boolean flag.

Flag parsing stops just before the first non-flag argument ("`-` is a non-flag argument) or after the terminator "`--`".

Integer flags accept 1234, 0664, 0x1234 and may be negative. Boolean flags may be:

```
1, 0, t, f, T, F, true, false, TRUE, FALSE, True, False
```

Duration flags accept any input valid for `time.ParseDuration`.

The default set of command-line flags is controlled by top-level functions. The `FlagSet` type allows one to define independent sets of flags, such as to implement subcommands in a command-line interface. The methods of `FlagSet` are analogous to the top-level functions for the command-line flag set.

Variables

```
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)
```

`CommandLine` is the default set of command-line flags, parsed from `os.Args`. The top-level functions such as `BoolVar`, `Arg`, and so on are wrappers for the methods of `CommandLine`.

```
var ErrHelp = errors.New("flag: help requested")
```

`ErrHelp` is the error returned if the `-help` or `-h` flag is invoked but no such flag is defined.

```
var Usage = func() {  
    fmt.Fprintf(CommandLine.Output(), "Usage of %s:\n", os.Args[0])  
}
```

```
    PrintDefaults()  
}
```

Usage prints a usage message documenting all defined command-line flags to CommandLine's output, which by default is os.Stderr. It is called when an error occurs while parsing flags. The function is a variable that may be changed to point to a custom function. By default it prints a simple header and calls PrintDefaults; for details about the format of the output and how to control it, see the documentation for PrintDefaults. Custom usage functions may choose to exit the program; by default exiting happens anyway as the command line's error handling strategy is set to ExitOnError.

func Arg

```
func Arg(i int) string
```

Arg returns the i'th command-line argument. Arg(0) is the first remaining argument after flags have been processed. Arg returns an empty string if the requested element does not exist.

func Args

```
func Args() []string
```

Args returns the non-flag command-line arguments.

func Bool

```
func Bool(name string, value bool, usage string) *bool
```

Bool defines a bool flag with specified name, default value, and usage string. The return value is the address of a bool variable that stores the value of the flag.

func BoolVar

```
func BoolVar(p *bool, name string, value bool, usage string)
```

BoolVar defines a bool flag with specified name, default value, and usage string. The argument p points to a bool variable in which to store the value of the flag.

func Duration

```
func Duration(name string, value time.Duration, usage string) *time.Duration
```

Duration defines a time.Duration flag with specified name, default value, and usage string. The return value is the address of a time.Duration variable that stores the value of the flag. The flag accepts a value acceptable to time.ParseDuration.

func DurationVar

```
func DurationVar(p *time.Duration, name string, value time.Duration, usage string)
```

DurationVar defines a time.Duration flag with specified name, default value, and usage string. The argument p points to a time.Duration variable in which to store the value of the flag. The flag accepts a value acceptable to time.ParseDuration.

func Float64

```
func Float64(name string, value float64, usage string) *float64
```

Float64 defines a float64 flag with specified name, default value, and usage string. The return value is the address of a float64 variable that stores the value of the flag.

func Float64Var

```
func Float64Var(p *float64, name string, value float64, usage string)
```

Float64Var defines a float64 flag with specified name, default value, and usage string. The argument p points to a float64 variable in which to store the value of the flag.

func Int

```
func Int(name string, value int, usage string) *int
```

Int defines an int flag with specified name, default value, and usage string. The return value is the address of an int variable that stores the value of the flag.

func Int64

```
func Int64(name string, value int64, usage string) *int64
```

Int64 defines an int64 flag with specified name, default value, and usage string. The return value is the address of an int64 variable that stores the value of the flag.

func Int64Var

```
func Int64Var(p *int64, name string, value int64, usage string)
```

Int64Var defines an int64 flag with specified name, default value, and usage string. The argument p points to an int64 variable in which to store the value of the flag.

func IntVar

```
func IntVar(p *int, name string, value int, usage string)
```

IntVar defines an int flag with specified name, default value, and usage string. The argument p points to an int variable in which to store the value of the flag.

func NArg

```
func NArg() int
```

NArg is the number of arguments remaining after flags have been processed.

func NFlag

```
func NFlag() int
```

NFlag returns the number of command-line flags that have been set.

func Parse

```
func Parse()
```

Parse parses the command-line flags from os.Args[1:]. Must be called after all flags are defined and before flags are accessed by the program.

func Parsed

```
func Parsed() bool
```

Parsed reports whether the command-line flags have been parsed.

func PrintDefaults

```
func PrintDefaults()
```

PrintDefaults prints, to standard error unless configured otherwise, a usage message showing the default settings of all defined command-line flags. For an integer valued flag x, the default output has the form

```
-x int
  usage-message-for-x (default 7)
```

The usage message will appear on a separate line for anything but a bool flag with a one-byte name. For bool flags, the type is omitted and if the flag name is one byte the usage message appears on the same line. The parenthetical default is omitted if the default is the zero value for the type. The listed type, here int, can be changed by placing a back-quoted name in the flag's usage string; the first such

item in the message is taken to be a parameter name to show in the message and the back quotes are stripped from the message when displayed. For instance, given

```
flag.String("I", "", "search `directory` for include files")
```

the output will be

```
-I directory
  search directory for include files.
```

To change the destination for flag messages, call `CommandLine.SetOutput`.

func `Set`

```
func Set(name, value string) error
```

`Set` sets the value of the named command-line flag.

func `String`

```
func String(name string, value string, usage string) *string
```

`String` defines a string flag with specified name, default value, and usage string. The return value is the address of a string variable that stores the value of the flag.

func `StringVar`

```
func StringVar(p *string, name string, value string, usage string)
```

`StringVar` defines a string flag with specified name, default value, and usage string. The argument `p` points to a string variable in which to store the value of the flag.

func `Uint`

```
func Uint(name string, value uint, usage string) *uint
```

`Uint` defines a uint flag with specified name, default value, and usage string. The return value is the address of a uint variable that stores the value of the flag.

func `Uint64`

```
func Uint64(name string, value uint64, usage string) *uint64
```

`Uint64` defines a uint64 flag with specified name, default value, and usage string. The return value is the address of a uint64 variable that stores the value of the flag.

func `Uint64Var`

```
func Uint64Var(p *uint64, name string, value uint64, usage string)
```

`Uint64Var` defines a `uint64` flag with specified name, default value, and usage string. The argument `p` points to a `uint64` variable in which to store the value of the flag.

func `UintVar`

```
func UintVar(p *uint, name string, value uint, usage string)
```

`UintVar` defines a `uint` flag with specified name, default value, and usage string. The argument `p` points to a `uint` variable in which to store the value of the flag.

func `UnquoteUsage`

```
func UnquoteUsage(flag *Flag) (name string, usage string)
```

`UnquoteUsage` extracts a back-quoted name from the usage string for a flag and returns it and the unquoted usage. Given "a `name` to show" it returns ("name", "a name to show"). If there are no back quotes, the name is an educated guess of the type of the flag's value, or the empty string if the flag is boolean.

func `Var`

```
func Var(value Value, name string, usage string)
```

`Var` defines a flag with the specified name and usage string. The type and value of the flag are represented by the first argument, of type `Value`, which typically holds a user-defined implementation of `Value`. For instance, the caller could create a flag that turns a comma-separated string into a slice of strings by giving the slice the methods of `Value`; in particular, `Set` would decompose the comma-separated string into the slice.

func `Visit`

```
func Visit(fn func(*Flag))
```

`Visit` visits the command-line flags in lexicographical order, calling `fn` for each. It visits only those flags that have been set.

func `VisitAll`

```
func VisitAll(fn func(*Flag))
```

VisitAll visits the command-line flags in lexicographical order, calling fn for each. It visits all flags, even those not set.

type ErrorHandling

```
type ErrorHandling int
```

ErrorHandling defines how FlagSet.Parse behaves if the parse fails.

```
const (
    ContinueOnError ErrorHandling = iota // Return a descriptive error.
    ExitOnError                      // Call os.Exit(2) or for -h/-help Exit(0).
    PanicOnError                     // Call panic with a descriptive error.
)
```

These constants cause FlagSet.Parse to behave as described if the parse fails.

type Flag

```
type Flag struct {
    Name      string // name as it appears on command line
    Usage     string // help message
    Value     Value  // value as set
    DefValue  string // default value (as text); for usage message
}
```

A Flag represents the state of a flag.

func Lookup

```
func Lookup(name string) *Flag
```

Lookup returns the Flag structure of the named command-line flag, returning nil if none exists.

type FlagSet

```
type FlagSet struct {
    // Usage is the function called when an error occurs while parsing flags.
    // The field is a function (not a method) that may be changed to point to
    // a custom error handler. What happens after Usage is called depends
    // on the ErrorHandling setting; for the command line, this defaults
    // to ExitOnError, which exits the program after calling Usage.
    Usage func()
    // contains filtered or unexported fields
}
```

A FlagSet represents a set of defined flags. The zero value of a FlagSet has no name and has ContinueOnError error handling.

Flag names must be unique within a FlagSet. An attempt to define a flag whose name is already in use will cause a panic.

func `NewFlagSet`

```
func NewFlagSet(name string, errorHandling ErrorHandling) *FlagSet
```

NewFlagSet returns a new, empty flag set with the specified name and error handling property. If the name is not empty, it will be printed in the default usage message and in error messages.

func `(*FlagSet) Arg`

```
func (f *FlagSet) Arg(i int) string
```

Arg returns the i'th argument. Arg(0) is the first remaining argument after flags have been processed. Arg returns an empty string if the requested element does not exist.

func `(*FlagSet) Args`

```
func (f *FlagSet) Args() []string
```

Args returns the non-flag arguments.

func `(*FlagSet) Bool`

```
func (f *FlagSet) Bool(name string, value bool, usage string) *bool
```

Bool defines a bool flag with specified name, default value, and usage string. The return value is the address of a bool variable that stores the value of the flag.

func `(*FlagSet) BoolVar`

```
func (f *FlagSet) BoolVar(p *bool, name string, value bool, usage string)
```

BoolVar defines a bool flag with specified name, default value, and usage string. The argument p points to a bool variable in which to store the value of the flag.

func `(*FlagSet) Duration`

```
func (f *FlagSet) Duration(name string, value time.Duration, usage string) *time.Dura
```

Duration defines a time.Duration flag with specified name, default value, and usage string. The return value is the address of a time.Duration variable that stores the value of the flag. The flag accepts a value acceptable to time.ParseDuration.

func (*FlagSet) DurationVar

```
func (f *FlagSet) DurationVar(p *time.Duration, name string, value time.Duration, usa
```

DurationVar defines a time.Duration flag with specified name, default value, and usage string. The argument p points to a time.Duration variable in which to store the value of the flag. The flag accepts a value acceptable to time.ParseDuration.

func (*FlagSet) ErrorHandling

```
func (f *FlagSet) ErrorHandling() ErrorHandling
```

ErrorHandling returns the error handling behavior of the flag set.

func (*FlagSet) Float64

```
func (f *FlagSet) Float64(name string, value float64, usage string) *float64
```

Float64 defines a float64 flag with specified name, default value, and usage string. The return value is the address of a float64 variable that stores the value of the flag.

func (*FlagSet) Float64Var

```
func (f *FlagSet) Float64Var(p *float64, name string, value float64, usage string)
```

Float64Var defines a float64 flag with specified name, default value, and usage string. The argument p points to a float64 variable in which to store the value of the flag.

func (*FlagSet) Init

```
func (f *FlagSet) Init(name string, errorHandling ErrorHandling)
```

Init sets the name and error handling property for a flag set. By default, the zero FlagSet uses an empty name and the ContinueOnError error handling policy.

func (*FlagSet) Int

```
func (f *FlagSet) Int(name string, value int, usage string) *int
```

Int defines an int flag with specified name, default value, and usage string. The return value is the address of an int variable that stores the value of the flag.

func (*FlagSet) Int64

```
func (f *FlagSet) Int64(name string, value int64, usage string) *int64
```

Int64 defines an int64 flag with specified name, default value, and usage string. The return value is the address of an int64 variable that stores the value of the flag.

func (*FlagSet) Int64Var

```
func (f *FlagSet) Int64Var(p *int64, name string, value int64, usage string)
```

Int64Var defines an int64 flag with specified name, default value, and usage string. The argument p points to an int64 variable in which to store the value of the flag.

func (*FlagSet) IntVar

```
func (f *FlagSet) IntVar(p *int, name string, value int, usage string)
```

IntVar defines an int flag with specified name, default value, and usage string. The argument p points to an int variable in which to store the value of the flag.

func (*FlagSet) Lookup

```
func (f *FlagSet) Lookup(name string) *Flag
```

Lookup returns the Flag structure of the named flag, returning nil if none exists.

func (*FlagSet) NArg

```
func (f *FlagSet) NArg() int
```

NArg is the number of arguments remaining after flags have been processed.

func (*FlagSet) NFlag

```
func (f *FlagSet) NFlag() int
```

NFlag returns the number of flags that have been set.

func (*FlagSet) Name

```
func (f *FlagSet) Name() string
```

Name returns the name of the flag set.

func (*FlagSet) Output

```
func (f *FlagSet) Output() io.Writer
```

Output returns the destination for usage and error messages. os.Stderr is returned if output was not set or was set to nil.

func (*FlagSet) Parse

```
func (f *FlagSet) Parse(arguments []string) error
```

Parse parses flag definitions from the argument list, which should not include the command name. Must be called after all flags in the FlagSet are defined and before flags are accessed by the program. The return value will be ErrHelp if -help or -h were set but not defined.

func (*FlagSet) Parsed

```
func (f *FlagSet) Parsed() bool
```

Parsed reports whether f.Parse has been called.

func (*FlagSet) PrintDefaults

```
func (f *FlagSet) PrintDefaults()
```

PrintDefaults prints, to standard error unless configured otherwise, the default values of all defined command-line flags in the set. See the documentation for the global function PrintDefaults for more information.

func (*FlagSet) Set

```
func (f *FlagSet) Set(name, value string) error
```

Set sets the value of the named flag.

func (*FlagSet) SetOutput

```
func (f *FlagSet) SetOutput(output io.Writer)
```

SetOutput sets the destination for usage and error messages. If output is nil, os.Stderr is used.

func (*FlagSet) String

```
func (f *FlagSet) String(name string, value string, usage string) *string
```

String defines a string flag with specified name, default value, and usage string. The return value is the address of a string variable that stores the value of the flag.

func (*FlagSet) StringVar

```
func (f *FlagSet) StringVar(p *string, name string, value string, usage string)
```

StringVar defines a string flag with specified name, default value, and usage string. The argument p points to a string variable in which to store the value of the flag.

func (*FlagSet) Uint

```
func (f *FlagSet) Uint(name string, value uint, usage string) *uint
```

Uint defines a uint flag with specified name, default value, and usage string. The return value is the address of a uint variable that stores the value of the flag.

func (*FlagSet) Uint64

```
func (f *FlagSet) Uint64(name string, value uint64, usage string) *uint64
```

Uint64 defines a uint64 flag with specified name, default value, and usage string. The return value is the address of a uint64 variable that stores the value of the flag.

func (*FlagSet) Uint64Var

```
func (f *FlagSet) Uint64Var(p *uint64, name string, value uint64, usage string)
```

Uint64Var defines a uint64 flag with specified name, default value, and usage string. The argument p points to a uint64 variable in which to store the value of the flag.

func (*FlagSet) UintVar

```
func (f *FlagSet) UintVar(p *uint, name string, value uint, usage string)
```

UintVar defines a uint flag with specified name, default value, and usage string. The argument p points to a uint variable in which to store the value of the flag.

func (*FlagSet) Var

```
func (f *FlagSet) Var(value Value, name string, usage string)
```

Var defines a flag with the specified name and usage string. The type and value of the flag are represented by the first argument, of type Value, which typically holds a user-defined implementation of Value. For instance, the caller could create a flag that turns a comma-separated string into a slice of strings by giving the slice the methods of Value; in particular, Set would decompose the comma-separated string into the slice.

func (*FlagSet) Visit

```
func (f *FlagSet) Visit(fn func(*Flag))
```

Visit visits the flags in lexicographical order, calling fn for each. It visits only those flags that have been set.

func (*FlagSet) VisitAll

```
func (f *FlagSet) VisitAll(fn func(*Flag))
```

VisitAll visits the flags in lexicographical order, calling fn for each. It visits all flags, even those not set.

type Getter

```
type Getter interface {
    Value
    Get() interface{}
}
```

Getter is an interface that allows the contents of a Value to be retrieved. It wraps the Value interface, rather than being part of it, because it appeared after Go 1 and its compatibility rules. All Value types provided by this package satisfy the Getter interface.

type Value

```
type Value interface {
    String() string
    Set(string) error
}
```

Value is the interface to the dynamic value stored in a flag. (The default value is represented as a string.)

If a Value has an `IsBoolFlag()` bool method returning true, the command-line parser makes `-name` equivalent to `-name=true` rather than using the next command-line argument.

`Set` is called once, in command line order, for each flag present. The flag package may call the `String` method with a zero-valued receiver, such as a nil pointer.

Overview

Package fmt implements formatted I/O with functions analogous to C's printf and scanf. The format 'verbs' are derived from C's but are simpler.

Printing

The verbs:

General:

```
%v the value in a default format
      when printing structs, the plus flag (%+v) adds field names
%#v a Go-syntax representation of the value
%T a Go-syntax representation of the type of the value
%% a literal percent sign; consumes no value
```

Boolean:

```
%t the word true or false
```

Integer:

```
%b base 2
%c the character represented by the corresponding Unicode code point
%d base 10
%o base 8
%0 base 8 with 0o prefix
%q a single-quoted character literal safely escaped with Go syntax.
%x base 16, with lower-case letters for a-f
%X base 16, with upper-case letters for A-F
%U Unicode format: U+1234; same as "U+%04X"
```

Floating-point and complex constituents:

```
%b decimalless scientific notation with exponent a power of two,
      in the manner of strconv.FormatFloat with the 'b' format,
      e.g. -123456p-78
%e scientific notation, e.g. -1.234456e+78
%E scientific notation, e.g. -1.234456E+78
```

```
%f decimal point but no exponent, e.g. 123.456
%F synonym for %f
%g %e for large exponents, %f otherwise. Precision is discussed below.
%G %E for large exponents, %F otherwise
%x hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
%X upper-case hexadecimal notation, e.g. -0X1.23ABCP+20
```

String and slice of bytes (treated equivalently with these verbs):

```
%s the uninterpreted bytes of the string or slice
%q a double-quoted string safely escaped with Go syntax
%x base 16, lower-case, two characters per byte
%X base 16, upper-case, two characters per byte
```

Slice:

```
%p address of 0th element in base 16 notation, with leading 0x
```

Pointer:

```
%p base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers,
formatting the value exactly as if it were an integer.
```

The default format for %v is:

bool:	%t
int, int8 etc.:	%d
uint, uint8 etc.:	%d, %#x if printed with %#v
float32, complex64, etc:	%g
string:	%s
chan:	%p
pointer:	%p

For compound objects, the elements are printed using these rules, recursively, laid out like this:

struct:	{field0 field1 ...}
array, slice:	[elem0 elem1 ...]
maps:	map[key1:value1 key2:value2 ...]
pointer to above:	&{}, &[], &map[]

Width is specified by an optional decimal number immediately preceding the verb. If absent, the width is whatever is necessary to represent the value. Precision is specified after the (optional) width by a period followed by a decimal number. If no period is present, a default precision is used. A period with no following number specifies a precision of zero. Examples:

```
%f      default width, default precision
%9f     width 9, default precision
```

```
%.2f    default width, precision 2
%9.2f   width 9, precision 2
%9.f    width 9, precision 0
```

Width and precision are measured in units of Unicode code points, that is, runes. (This differs from C's printf where the units are always measured in bytes.) Either or both of the flags may be replaced with the character '*', causing their values to be obtained from the next operand (preceding the one to format), which must be of type int.

For most values, width is the minimum number of runes to output, padding the formatted form with spaces if necessary.

For strings, byte slices and byte arrays, however, precision limits the length of the input to be formatted (not the size of the output), truncating if necessary. Normally it is measured in runes, but for these types when formatted with the %x or %X format it is measured in bytes.

For floating-point values, width sets the minimum width of the field and precision sets the number of places after the decimal, if appropriate, except that for %g/%G precision sets the maximum number of significant digits (trailing zeros are removed). For example, given 12.345 the format %6.3f prints 12.345 while %.3g prints 12.3. The default precision for %e, %f and %#g is 6; for %g it is the smallest number of digits necessary to identify the value uniquely.

For complex numbers, the width and precision apply to the two components independently and the result is parenthesized, so %f applied to 1.2+3.4i produces (1.200000+3.400000i).

Other flags:

- + always print a sign for numeric values;
guarantee ASCII-only output for %q (%+q)
- pad with spaces on the right rather than the left (left-justify the field)
- # alternate format: add leading 0b for binary (%#b), 0 for octal (%#o),
0x or 0X for hex (%#x or %#X); suppress 0x for %p (%#p);
for %q, print a raw (backquoted) string if strconv.CanBackquote
returns true;
always print a decimal point for %e, %E, %f, %F, %g and %G;
do not remove trailing zeros for %g and %G;
write e.g. U+0078 'x' if the character is printable for %U (%#U).
- ' ' (space) leave a space for elided sign in numbers (% d);
put spaces between bytes printing strings or slices in hex (% x, % X)
- 0 pad with leading zeros rather than spaces;
for numbers, this moves the padding after the sign

Flags are ignored by verbs that do not expect them. For example there is no alternate decimal format, so %#d and %d behave identically.

For each Printf-like function, there is also a Print function that takes no format and is equivalent to saying %v for every operand. Another variant Println inserts blanks between operands and appends a newline.

Regardless of the verb, if an operand is an interface value, the internal concrete value is used, not the interface itself. Thus:

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

will print 23.

Except when printed using the verbs %T and %p, special formatting considerations apply for operands that implement certain interfaces. In order of application:

1. If the operand is a reflect.Value, the operand is replaced by the concrete value that it holds, and printing continues with the next rule.
2. If an operand implements the Formatter interface, it will be invoked. Formatter provides fine control of formatting.
3. If the %v verb is used with the # flag (%#v) and the operand implements the GoStringer interface, that will be invoked.

If the format (which is implicitly %v for `Println` etc.) is valid for a string (%s %q %v %x %X), the following two rules apply:

4. If an operand implements the error interface, the `Error` method will be invoked to convert the object to a string, which will then be formatted as required by the verb (if any).
5. If an operand implements method `String()` string, that method will be invoked to convert the object to a string, which will then be formatted as required by the verb (if any).

For compound operands such as slices and structs, the format applies to the elements of each operand, recursively, not to the operand as a whole. Thus %q will quote each element of a slice of strings, and %6.2f will control formatting for each element of a floating-point array.

However, when printing a byte slice with a string-like verb (%s %q %x %X), it is treated identically to a string, as a single item.

To avoid recursion in cases such as

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

convert the value before recurring:

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

Infinite recursion can also be triggered by self-referential data structures, such as a slice that contains itself as an element, if that type has a `String` method. Such pathologies are rare, however, and the package does not protect against them.

When printing a struct, `fmt` cannot and therefore does not invoke formatting methods such as `Error` or `String` on unexported fields.

Explicit argument indexes:

In `Printf`, `Sprintf`, and `Fprintf`, the default behavior is for each formatting verb to format successive arguments passed in the call. However, the notation `[n]` immediately before the verb indicates that the `n`th one-indexed argument is to be formatted instead. The same notation before a `*` for a width or precision selects the argument index holding the value. After processing a bracketed expression `[n]`, subsequent verbs will use arguments `n+1`, `n+2`, etc. unless otherwise directed.

For example,

```
fmt.Sprintf("%[2]d %[1]d\n", 11, 22)
```

will yield "22 11", while

```
fmt.Sprintf("%[3]*.[2]*[1]f", 12.0, 2, 6)
```

equivalent to

```
fmt.Sprintf("%6.2f", 12.0)
```

will yield "12.00". Because an explicit index affects subsequent verbs, this notation can be used to print the same values multiple times by resetting the index for the first argument to be repeated:

```
fmt.Sprintf("%d %d %#[1]x %#x", 16, 17)
```

will yield "16 17 0x10 0x11".

Format errors:

If an invalid argument is given for a verb, such as providing a string to `%d`, the generated string will contain a description of the problem, as in these examples:

```
Wrong type or unknown verb: %!verb(type=value)
Printf("%d", "hi"):      %!d(string=hi)
Too many arguments: %!(EXTRA type=value)
Printf("hi", "guys"):    hi%!(EXTRA string=guys)
Too few arguments: %!verb(MISSING)
Printf("hi%d"):          hi%!d(MISSING)
Non-int for width or precision: %!(BADWIDTH) or %!(BADPREC)
Printf("%*s", 4.5, "hi"): %!(BADWIDTH)hi
Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
Invalid or invalid use of argument index: %!(BADINDEX)
Printf("%*[2]d", 7):      %!d(BADINDEX)
Printf("%.[2]d", 7):      %!d(BADINDEX)
```

All errors begin with the string "%!" followed sometimes by a single character (the verb) and end with a parenthesized description.

If an Error or String method triggers a panic when called by a print routine, the fmt package reformats the error message from the panic, decorating it with an indication that it came through the fmt package. For example, if a String method calls panic("bad"), the resulting formatted message will look like

```
%!s(PANIC=bad)
```

The %!s just shows the print verb in use when the failure occurred. If the panic is caused by a nil receiver to an Error or String method, however, the output is the undecorated string, "<nil>".

Scanning

An analogous set of functions scans formatted text to yield values. Scan, Scanf and Scanln read from os.Stdin; Fscan, Fscanf and Fscanln read from a specified io.Reader; Sscan, Sscanf and Sscanln read from an argument string.

Scan, Fscan, Sscan treat newlines in the input as spaces.

Scanln, Fscanln and Sscanln stop scanning at a newline and require that the items be followed by a newline or EOF.

Scanf, Fscanf, and Sscanf parse the arguments according to a format string, analogous to that of Printf. In the text that follows, 'space' means any Unicode whitespace character except newline.

In the format string, a verb introduced by the % character consumes and parses input; these verbs are described in more detail below. A character other than %, space, or newline in the format consumes exactly that input character, which must be present. A newline with zero or more spaces before it in the format string consumes zero or more spaces in the input followed by a single newline or the end of the input. A space following a newline in the format string consumes zero or more spaces in the input. Otherwise, any run of one or more spaces in the format string consumes as many spaces as possible in the input. Unless the run of spaces in the format string appears adjacent to a newline, the run must consume at least one space from the input or find the end of the input.

The handling of spaces and newlines differs from that of C's scanf family: in C, newlines are treated as any other space, and it is never an error when a run of spaces in the format string finds no spaces to consume in the input.

The verbs behave analogously to those of Printf. For example, %x will scan an integer as a hexadecimal number, and %v will scan the default representation format for the value. The Printf verbs %p and %T and the flags # and + are not implemented. For floating-point and complex values, all valid formatting verbs (%b %e %E %f %F %g %G %x %X and %v) are equivalent and accept both decimal and hexadecimal notation (for example: "2.3e+7", "0x4.5p-8") and digit-separating underscores (for example: "3.14159_26535_89793").

Input processed by verbs is implicitly space-delimited: the implementation of every verb except %c starts by discarding leading spaces from the remaining input, and the %s verb (and %v reading into a string) stops consuming input at the first space or newline character.

The familiar base-setting prefixes 0b (binary), 0o and 0 (octal), and 0x (hexadecimal) are accepted when scanning integers without a format or with the %v verb, as are digit-separating underscores.

Width is interpreted in the input text but there is no syntax for scanning with a precision (no %5.2f, just %5f). If width is provided, it applies after leading spaces are trimmed and specifies the maximum number of runes to read to satisfy the verb. For example,

```
Sscanf(" 1234567 ", "%5s%d", &s, &i)
```

will set s to "12345" and i to 67 while

```
Sscanf(" 12 34 567 ", "%5s%d", &s, &i)
```

will set s to "12" and i to 34.

In all the scanning functions, a carriage return followed immediately by a newline is treated as a plain newline (\r\n means the same as \n).

In all the scanning functions, if an operand implements method Scan (that is, it implements the Scanner interface) that method will be used to scan the text for that operand. Also, if the number of arguments scanned is less than the number of arguments provided, an error is returned.

All arguments to be scanned must be either pointers to basic types or implementations of the Scanner interface.

Like Scanf and Fscanf, Sscanf need not consume its entire input. There is no way to recover how much of the input string Sscanf used.

Note: Fscan etc. can read one character (rune) past the input they return, which means that a loop calling a scan routine may skip some of the input. This is usually a problem only when there is no space between input values. If the reader provided to Fscan implements ReadRune, that method will be used to read characters. If the reader also implements UnreadRune, that method will be used to save the character and successive calls will not lose data. To attach ReadRune and UnreadRune methods to a reader without that capability, use bufio.NewReader.

func **Errorf**

```
func Errorf(format string, a ...interface{}) error
```

Errorf formats according to a format specifier and returns the string as a value that satisfies error.

If the format specifier includes a %w verb with an error operand, the returned error will implement an Unwrap method returning the operand. It is invalid to include more than one %w verb or to supply it

with an operand that does not implement the error interface. The `%w` verb is otherwise a synonym for `%v`.

func `Fprint`

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
```

`Fprint` formats using the default formats for its operands and writes to `w`. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func `Fprintf`

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

`Fprintf` formats according to a format specifier and writes to `w`. It returns the number of bytes written and any write error encountered.

func `Fprintln`

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

`Fprintln` formats using the default formats for its operands and writes to `w`. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func `Fscan`

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
```

`Fscan` scans text read from `r`, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, `err` will report why.

func `Fscanf`

```
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
```

`Fscanf` scans text read from `r`, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed. Newlines in the input must match newlines in the format.

func `Fscanln`

```
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
```

FscanIn is similar to Fscan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func Print

```
func Print(a ...interface{}) (n int, err error)
```

Print formats using the default formats for its operands and writes to standard output. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func Printf

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Printf formats according to a format specifier and writes to standard output. It returns the number of bytes written and any write error encountered.

func Println

```
func Println(a ...interface{}) (n int, err error)
```

Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func Scan

```
func Scan(a ...interface{}) (n int, err error)
```

Scan scans text read from standard input, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func Scanf

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

Scanf scans text read from standard input, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why. Newlines in the input must match newlines in the format. The one exception: the verb %c always scans the next rune in the input, even if it is a space (or tab etc.) or newline.

func Scanln

```
func Scanln(a ...interface{}) (n int, err error)
```

Scanln is similar to Scan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func **Sprint**

```
func Sprint(a ...interface{}) string
```

Sprint formats using the default formats for its operands and returns the resulting string. Spaces are added between operands when neither is a string.

func **Sprintf**

```
func Sprintf(format string, a ...interface{}) string
```

Sprintf formats according to a format specifier and returns the resulting string.

func **Sprintln**

```
func Sprintln(a ...interface{}) string
```

Sprintln formats using the default formats for its operands and returns the resulting string. Spaces are always added between operands and a newline is appended.

func **Sscan**

```
func Sscan(str string, a ...interface{}) (n int, err error)
```

Sscan scans the argument string, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func **Sscanf**

```
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
```

Sscanf scans the argument string, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed. Newlines in the input must match newlines in the format.

func **Sscanln**

```
func Sscanln(str string, a ...interface{}) (n int, err error)
```

`Sscanf` is similar to `Sscanf`, but stops scanning at a newline and after the final item there must be a newline or EOF.

type `Formatter`

```
type Formatter interface {
    Format(f State, c rune)
}
```

`Formatter` is the interface implemented by values with a custom formatter. The implementation of `Format` may call `Sprint(f)` or `Fprint(f)` etc. to generate its output.

type `GoStringer`

```
type GoStringer interface {
    GoString() string
}
```

`GoStringer` is implemented by any value that has a `GoString` method, which defines the Go syntax for that value. The `GoString` method is used to print values passed as an operand to a `%#v` format.

type `ScanState`

```
type ScanState interface {
    // ReadRune reads the next rune (Unicode code point) from the input.
    // If invoked during Scanln, Fscanln, or Sscanfln, ReadRune() will
    // return EOF after returning the first '\n' or when reading beyond
    // the specified width.
    ReadRune() (r rune, size int, err error)
    // UnreadRune causes the next call to ReadRune to return the same rune.
    UnreadRune() error
    // SkipSpace skips space in the input. Newlines are treated appropriately
    // for the operation being performed; see the package documentation
    // for more information.
    SkipSpace()
    // Token skips space in the input if skipSpace is true, then returns the
    // run of Unicode code points c satisfying f(c). If f is nil,
    // !unicode.IsSpace(c) is used; that is, the token will hold non-space
    // characters. Newlines are treated appropriately for the operation being
    // performed; see the package documentation for more information.
    // The returned slice points to shared data that may be overwritten
    // by the next call to Token, a call to a Scan function using the ScanState
    // as input, or when the calling Scan method returns.
    Token(skipSpace bool, f func(rune) bool) (token []byte, err error)
    // Width returns the value of the width option and whether it has been set.
    // The unit is Unicode code points.
    Width() (wid int, ok bool)
    // Because ReadRune is implemented by the interface, Read should never be
    // called by the scanning routines and a valid implementation of
    // ScanState may choose always to return an error from Read.
}
```

```
    Read(buf []byte) (n int, err error)
}
```

ScanState represents the scanner state passed to custom scanners. Scanners may do rune-at-a-time scanning or ask the ScanState to discover the next space-delimited token.

type Scanner

```
type Scanner interface {
    Scan(state ScanState, verb rune) error
}
```

Scanner is implemented by any value that has a Scan method, which scans the input for the representation of a value and stores the result in the receiver, which must be a pointer to be useful. The Scan method is called for any argument to Scan, Scanf, or Scanln that implements it.

type State

```
type State interface {
    // Write is the function to call to emit formatted output to be printed.
    Write(b []byte) (n int, err error)
    // Width returns the value of the width option and whether it has been set.
    Width() (wid int, ok bool)
    // Precision returns the value of the precision option and whether it has been set.
    Precision() (prec int, ok bool)

    // Flag reports whether the flag c, a character, has been set.
    Flag(c int) bool
}
```

State represents the printer state passed to custom formatters. It provides access to the io.Writer interface plus information about the flags and options for the operand's format specifier.

type Stringer

```
type Stringer interface {
    String() string
}
```

Stringer is implemented by any value that has a String method, which defines the ``native'' format for that value. The String method is used to print values passed as an operand to any format that accepts a string or to an unformatted printer such as Print.

Overview

Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver). A typical use is transporting arguments and results of remote procedure calls (RPCs) such as those provided by package "net/rpc".

The implementation compiles a custom codec for each data type in the stream and is most efficient when a single Encoder is used to transmit a stream of values, amortizing the cost of compilation.

Basics

A stream of gobs is self-describing. Each data item in the stream is preceded by a specification of its type, expressed in terms of a small set of predefined types. Pointers are not transmitted, but the things they point to are transmitted; that is, the values are flattened. Nil pointers are not permitted, as they have no value. Recursive types work fine, but recursive values (data with cycles) are problematic. This may change.

To use gobs, create an Encoder and present it with a series of data items as values or addresses that can be dereferenced to values. The Encoder makes sure all type information is sent before it is needed. At the receive side, a Decoder retrieves values from the encoded stream and unpacks them into local variables.

Types and Values

The source and destination values/types need not correspond exactly. For structs, fields (identified by name) that are in the source but absent from the receiving variable will be ignored. Fields that are in the receiving variable but missing from the transmitted type or value will be ignored in the destination. If a field with the same name is present in both, their types must be compatible. Both the receiver and transmitter will do all necessary indirection and dereferencing to convert between gobs and actual Go values. For instance, a gob type that is schematically,

```
struct { A, B int }
```

can be sent from or received into any of these Go types:

```
struct { A, B int } // the same
*struct { A, B int } // extra indirection of the struct
```

```
struct { *A, **B int } // extra indirection of the fields
struct { A, B int64 } // different concrete value type; see below
```

It may also be received into any of these:

```
struct { A, B int } // the same
struct { B, A int } // ordering doesn't matter; matching is by name
struct { A, B, C int } // extra field (C) ignored
struct { B int } // missing field (A) ignored; data will be dropped
struct { B, C int } // missing field (A) ignored; extra field (C) ignored.
```

Attempting to receive into these types will draw a decode error:

```
struct { A int; B uint } // change of signedness for B
struct { A int; B float } // change of type for B
struct { } // no field names in common
struct { C, D int } // no field names in common
```

Integers are transmitted two ways: arbitrary precision signed integers or arbitrary precision unsigned integers. There is no int8, int16 etc. discrimination in the gob format; there are only signed and unsigned integers. As described below, the transmitter sends the value in a variable-length encoding; the receiver accepts the value and stores it in the destination variable. Floating-point numbers are always sent using IEEE-754 64-bit precision (see below).

Signed integers may be received into any signed integer variable: int, int16, etc.; unsigned integers may be received into any unsigned integer variable; and floating point values may be received into any floating point variable. However, the destination variable must be able to represent the value or the decode operation will fail.

Structs, arrays and slices are also supported. Structs encode and decode only exported fields. Strings and arrays of bytes are supported with a special, efficient representation (see below). When a slice is decoded, if the existing slice has capacity the slice will be extended in place; if not, a new array is allocated. Regardless, the length of the resulting slice reports the number of elements decoded.

In general, if allocation is required, the decoder will allocate memory. If not, it will update the destination variables with values read from the stream. It does not initialize them first, so if the destination is a compound value such as a map, struct, or slice, the decoded values will be merged elementwise into the existing variables.

Functions and channels will not be sent in a gob. Attempting to encode such a value at the top level will fail. A struct field of chan or func type is treated exactly like an unexported field and is ignored.

Gob can encode a value of any type implementing the GobEncoder or encoding.BinaryMarshaler interfaces by calling the corresponding method, in that order of preference.

Gob can decode a value of any type implementing the GobDecoder or encoding.BinaryUnmarshaler interfaces by calling the corresponding method, again in that order of preference.

Encoding Details

This section documents the encoding, details that are not important for most users. Details are presented bottom-up.

An unsigned integer is sent one of two ways. If it is less than 128, it is sent as a byte with that value. Otherwise it is sent as a minimal-length big-endian (high byte first) byte stream holding the value, preceded by one byte holding the byte count, negated. Thus 0 is transmitted as (00), 7 is transmitted as (07) and 256 is transmitted as (FE 01 00).

A boolean is encoded within an unsigned integer: 0 for false, 1 for true.

A signed integer, i , is encoded within an unsigned integer, u . Within u , bits 1 upward contain the value; bit 0 says whether they should be complemented upon receipt. The encode algorithm looks like this:

```
var u uint
if i < 0 {
    u = (^uint(i) << 1) | 1 // complement i, bit 0 is 1
} else {
    u = (uint(i) << 1) // do not complement i, bit 0 is 0
}
encodeUnsigned(u)
```

The low bit is therefore analogous to a sign bit, but making it the complement bit instead guarantees that the largest negative integer is not a special case. For example, $-129 = ^128 = (^{256} >> 1)$ encodes as (FE 01 01).

Floating-point numbers are always sent as a representation of a float64 value. That value is converted to a uint64 using math.Float64bits. The uint64 is then byte-reversed and sent as a regular unsigned integer. The byte-reversal means the exponent and high-precision part of the mantissa go first. Since the low bits are often zero, this can save encoding bytes. For instance, 17.0 is encoded in only three bytes (FE 31 40).

Strings and slices of bytes are sent as an unsigned count followed by that many uninterpreted bytes of the value.

All other slices and arrays are sent as an unsigned count followed by that many elements using the standard gob encoding for their type, recursively.

Maps are sent as an unsigned count followed by that many key, element pairs. Empty but non-nil maps are sent, so if the receiver has not allocated one already, one will always be allocated on receipt unless the transmitted map is nil and not at the top level.

In slices and arrays, as well as maps, all elements, even zero-valued elements, are transmitted, even if all the elements are zero.

Structs are sent as a sequence of (field number, field value) pairs. The field value is sent using the standard gob encoding for its type, recursively. If a field has the zero value for its type (except for

arrays; see above), it is omitted from the transmission. The field number is defined by the type of the encoded struct: the first field of the encoded type is field 0, the second is field 1, etc. When encoding a value, the field numbers are delta encoded for efficiency and the fields are always sent in order of increasing field number; the deltas are therefore unsigned. The initialization for the delta encoding sets the field number to -1, so an unsigned integer field 0 with value 7 is transmitted as unsigned delta = 1, unsigned value = 7 or (01 07). Finally, after all the fields have been sent a terminating mark denotes the end of the struct. That mark is a delta=0 value, which has representation (00).

Interface types are not checked for compatibility; all interface types are treated, for transmission, as members of a single "interface" type, analogous to int or []byte - in effect they're all treated as interface{}. Interface values are transmitted as a string identifying the concrete type being sent (a name that must be pre-defined by calling Register), followed by a byte count of the length of the following data (so the value can be skipped if it cannot be stored), followed by the usual encoding of concrete (dynamic) value stored in the interface value. (A nil interface value is identified by the empty string and transmits no value.) Upon receipt, the decoder verifies that the unpacked concrete item satisfies the interface of the receiving variable.

If a value is passed to Encode and the type is not a struct (or pointer to struct, etc.), for simplicity of processing it is represented as a struct of one field. The only visible effect of this is to encode a zero byte after the value, just as after the last field of an encoded struct, so that the decode algorithm knows when the top-level value is complete.

The representation of types is described below. When a type is defined on a given connection between an Encoder and Decoder, it is assigned a signed integer type id. When Encoder.Encode(v) is called, it makes sure there is an id assigned for the type of v and all its elements and then it sends the pair (typeid, encoded-v) where typeid is the type id of the encoded type of v and encoded-v is the gob encoding of the value v.

To define a type, the encoder chooses an unused, positive type id and sends the pair (-type id, encoded-type) where encoded-type is the gob encoding of a wireType description, constructed from these types:

```
type wireType struct {
    ArrayT      *ArrayType
    SliceT      *SliceType
    StructT     *StructType
    MapT        *MapType
    GobEncoderT *gobEncoderType
    BinaryMarshalerT *gobEncoderType
    TextMarshalerT  *gobEncoderType
}

type arrayType struct {
    CommonType
    ElemtypId
    Len  int
}

type CommonType struct {
    Name string // the name of the struct type
```

```

    Id int    // the id of the type, repeated so it's inside the type
}
type sliceType struct {
    CommonType
    Elemt typeId
}
type structType struct {
    CommonType
    Field []*fieldType // the fields of the struct.
}
type fieldType struct {
    Name string // the name of the field.
    Id int     // the type id of the field, which must be already defined
}
type mapType struct {
    CommonType
    Key typeId
    Elemt typeId
}
type gobEncoderType struct {
    CommonType
}

```

If there are nested type ids, the types for all inner type ids must be defined before the top-level type id is used to describe an encoded-v.

For simplicity in setup, the connection is defined to understand these types a priori, as well as the basic gob types int, uint, etc. Their ids are:

```

bool      1
int       2
uint      3
float     4
[]byte    5
string    6
complex   7
interface 8
// gap for reserved ids.
WireType  16
ArrayType  17
CommonType 18
SliceType  19
StructType 20
FieldType  21
// 22 is slice of fieldType.
MapType   23

```

Finally, each message created by a call to `Encode` is preceded by an encoded unsigned integer count of the number of bytes remaining in the message. After the initial type name, interface values are wrapped the same way; in effect, the interface value acts like a recursive invocation of `Encode`.

In summary, a gob stream looks like

```
(byteCount (-type id, encoding of a wireType)* (type id, encoding of a value))*
```

where * signifies zero or more repetitions and the type id of a value must be predefined or be defined before the value in the stream.

Compatibility: Any future changes to the package will endeavor to maintain compatibility with streams encoded using previous versions. That is, any released version of this package should be able to decode data written with any previously released version, subject to issues such as security fixes. See the Go compatibility document for background: <https://golang.org/doc/go1compat>

See "Gobs of data" for a design discussion of the gob wire format: <https://blog.golang.org/gobs-of-data>

func Register

```
func Register(value interface{})
```

Register records a type, identified by a value for that type, under its internal type name. That name will identify the concrete type of a value sent or received as an interface variable. Only types that will be transferred as implementations of interface values need to be registered. Expecting to be used only during initialization, it panics if the mapping between types and names is not a bijection.

func RegisterName

```
func RegisterName(name string, value interface{})
```

RegisterName is like Register but uses the provided name rather than the type's default.

type CommonType

```
type CommonType struct {
    Name string
    Id   typeId
}
```

CommonType holds elements of all types. It is a historical artifact, kept for binary compatibility and exported only for the benefit of the package's encoding of type descriptors. It is not intended for direct use by clients.

type Decoder

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

A Decoder manages the receipt of type and data information read from the remote side of a connection. It is safe for concurrent use by multiple goroutines.

The Decoder does only basic sanity checking on decoded input sizes, and its limits are not configurable. Take caution when decoding gob data from untrusted sources.

func `NewDecoder`

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder returns a new decoder that reads from the `io.Reader`. If `r` does not also implement `io.ByteReader`, it will be wrapped in a `bufio.Reader`.

func `(*Decoder) Decode`

```
func (dec *Decoder) Decode(e interface{}) error
```

`Decode` reads the next value from the input stream and stores it in the data represented by the empty `interface` value. If `e` is `nil`, the value will be discarded. Otherwise, the value underlying `e` must be a pointer to the correct type for the next data item received. If the input is at EOF, `Decode` returns `io.EOF` and does not modify `e`.

func `(*Decoder) DecodeValue`

```
func (dec *Decoder) DecodeValue(v reflect.Value) error
```

`DecodeValue` reads the next value from the input stream. If `v` is the zero `reflect.Value` (`v.Kind() == Invalid`), `DecodeValue` discards the value. Otherwise, it stores the value into `v`. In that case, `v` must represent a non-nil pointer to data or be an assignable `reflect.Value` (`v.CanSet()`). If the input is at EOF, `DecodeValue` returns `io.EOF` and does not modify `v`.

type `Encoder`

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

An Encoder manages the transmission of type and data information to the other side of a connection. It is safe for concurrent use by multiple goroutines.

func `NewEncoder`

```
func NewEncoder(w io.Writer) *Encoder
```

`NewEncoder` returns a new encoder that will transmit on the `io.Writer`.

func (*Encoder) Encode

```
func (enc *Encoder) Encode(e interface{}) error
```

Encode transmits the data item represented by the empty interface value, guaranteeing that all necessary type information has been transmitted first. Passing a nil pointer to Encoder will panic, as they cannot be transmitted by gob.

func (*Encoder) EncodeValue

```
func (enc *Encoder) EncodeValue(value reflect.Value) error
```

EncodeValue transmits the data item represented by the reflection value, guaranteeing that all necessary type information has been transmitted first. Passing a nil pointer to EncodeValue will panic, as they cannot be transmitted by gob.

type GobDecoder

```
type GobDecoder interface {
    // GobDecode overwrites the receiver, which must be a pointer,
    // with the value represented by the byte slice, which was written
    // by GobEncode, usually for the same concrete type.
    GobDecode([]byte) error
}
```

GobDecoder is the interface describing data that provides its own routine for decoding transmitted values sent by a GobEncoder.

type GobEncoder

```
type GobEncoder interface {
    // GobEncode returns a byte slice representing the encoding of the
    // receiver for transmission to a GobDecoder, usually of the same
    // concrete type.
    GobEncode() ([]byte, error)
}
```

GobEncoder is the interface describing data that provides its own representation for encoding values for transmission to a GobDecoder. A type that implements GobEncoder and GobDecoder has complete control over the representation of its data and may therefore contain things such as private fields, channels, and functions, which are not usually transmissible in gob streams.

Note: Since gobs can be stored permanently, it is good design to guarantee the encoding used by a GobEncoder is stable as the software evolves. For instance, it might make sense for GobEncode to include a version number in the encoding.

Package html

 go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

Overview

Package html provides functions for escaping and unescaping HTML text.

func [EscapeString](#)

```
func EscapeString(s string) string
```

`EscapeString` escapes special characters like "<" to become "<". It escapes only five such characters: <, >, &, ' and ". `UnescapeString(EscapeString(s)) == s` always holds, but the converse isn't always true.

func [UnescapeString](#)

```
func UnescapeString(s string) string
```

`UnescapeString` unescapes entities like "<" to become "<". It unescapes a larger range of entities than `EscapeString` escapes. For example, "á" unescapes to "á", as does "á" and "á". `UnescapeString(EscapeString(s)) == s` always holds, but the converse isn't always true.

Package template

 go1.15.2 Latest

Published: **Sep 9, 2020** | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

Overview

Package template (html/template) implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package text/template and should be used instead of text/template whenever the output is HTML.

The documentation here focuses on the security features of the package. For information about how to program the templates themselves, see the documentation for text/template.

Introduction

This package wraps package text/template so you can share its template API to parse and execute HTML templates safely.

```
tmpl, err := template.New("name").Parse(...)  
// Error checking elided  
err = tmpl.Execute(out, data)
```

If successful, tmpl will now be injection-safe. Otherwise, err is an error defined in the docs for ErrorCode.

HTML templates treat data values as plain text which should be encoded so they can be safely embedded in an HTML document. The escaping is contextual, so actions can appear within JavaScript, CSS, and URI contexts.

The security model used by this package assumes that template authors are trusted, while Execute's data parameter is not. More details are provided below.

Example

```
import "text/template"  
...  
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)  
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces

```
Hello, <script>alert('you have been pwned')</script>!
```

but the contextual autoescaping in html/template

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces safe, escaped HTML output

```
Hello, <script>alert('you have been pwned')</script>!
```

Contexts

This package understands HTML, CSS, JavaScript, and URLs. It adds sanitizing functions to each simple action pipeline, so given the excerpt

```
<a href="/search?q={{.}}">{{.}}</a>
```

At parse time each {{.}} is overwritten to add escaping functions as necessary. In this case it becomes

```
<a href="/search?q={{. | urlescape | attrescape}}">{{. | htmlescape}}</a>
```

where urlescape, attrescape, and htmlescape are aliases for internal escaping functions.

For these internal escaping functions, if an action pipeline evaluates to a nil interface value, it is treated as though it were an empty string.

Namespaced and data- attributes

Attributes with a namespace are treated as if they had no namespace. Given the excerpt

```
<a my:href="{{.}}"></a>
```

At parse time the attribute will be treated as if it were just "href". So at parse time the template becomes:

```
<a my:href="{{. | urlescape | attrescape}}"></a>
```

Similarly to attributes with namespaces, attributes with a "data-" prefix are treated as if they had no "data-" prefix. So given

```
<a data-href="{{.}}"></a>
```

At parse time this becomes

```
<a data-href="{{. | urlescaper | attrescaper}}"></a>
```

If an attribute has both a namespace and a "data-" prefix, only the namespace will be removed when determining the context. For example

```
<a my:mydata-href="{{.}}"></a>
```

This is handled as if "my:mydata-href" was just "mydata-href" and not "href" as it would be if the "data-" prefix were to be ignored too. Thus at parse time this becomes just

```
<a my:mydata-href="{{. | attrescaper}}"></a>
```

As a special case, attributes with the namespace "xmlns" are always treated as containing URLs. Given the excerpts

```
<a xmlns:title="{{.}}"></a>
<a xmlns:href="{{.}}"></a>
<a xmlns:onclick="{{.}}"></a>
```

At parse time they become:

```
<a xmlns:title="{{. | urlescaper | attrescaper}}"></a>
<a xmlns:href="{{. | urlescaper | attrescaper}}"></a>
<a xmlns:onclick="{{. | urlescaper | attrescaper}}"></a>
```

Errors

See the documentation of ErrorCode for details.

A fuller picture

The rest of this package comment may be skipped on first reading; it includes details necessary to understand escaping contexts and error messages. Most users will not need to understand these details.

Contexts

Assuming {{.}} is `O'Reilly: How are <i>you</i>?', the table below shows how {{.}} appears when used in the context to the left.

Context	{{.}} After
{{.}}	O'Reilly: How are <i>you</i>?
	O'Reilly: How are you?
	O'Reilly: How are %3ci%3eyou%3c/i%3e?
	O'Reilly%3a%20How%20are%3ci%3e...%3f

```
<a onx='f("{{.}}")'>          0\x27Reilly: How are \x3ci\x3eyou...?  
<a onx='f({{.}})'>           "0\x27Reilly: How are \x3ci\x3eyou...?"  
<a onx='pattern = /{{.}}/;'>  0\x27Reilly: How are \x3ci\x3eyou...\\x3f
```

If used in an unsafe context, then the value might be filtered out:

Context	{} After
	#ZgotmplZ

since "O'Reilly:" is not an allowed protocol like "http:".

If {{.}} is the innocuous word, 'left', then it can appear more widely,

Context	{} After
{}{{.}}	left
	left
	left
	left
	left
	left
	left
	left
	left
<style>p.{{.}} {color:red}</style>	left

Non-string values can be used in JavaScript contexts. If {{.}} is

```
struct{A,B string}{ "foo", "bar" }
```

in the escaped template

```
<script>var pair = {{.}};</script>
```

then the template output is

```
<script>var pair = {"A": "foo", "B": "bar"};</script>
```

See package.json to understand how non-string content is marshaled for embedding in JavaScript contexts.

Typed Strings

By default, this package assumes that all pipelines produce a plain text string. It adds escaping pipeline stages necessary to correctly and safely embed that plain text string in the appropriate context.

When a data value is not plain text, you can make sure it is not over-escaped by marking it with its type.

Types HTML, JS, URL, and others from `content.go` can carry safe content that is exempted from escaping.

The template

```
Hello, {{.}}!
```

can be invoked with

```
tmpl.Execute(out, template.HTML(`<b>World</b>`))
```

to produce

```
Hello, <b>World</b>!
```

instead of the

```
Hello, &lt;b&gt;World&lt;b&gt;!
```

that would have been produced if `{{.}}` was a regular string.

Security Model

https://rawgit.com/mikesamuel/sanitized-jquery-templates/trunk/safetemplate.html#problem_definition defines "safe" as used by this package.

This package assumes that template authors are trusted, that `Execute`'s `data` parameter is not, and seeks to preserve the properties below in the face of untrusted data:

Structure Preservation Property: "... when a template author writes an HTML tag in a safe templating language, the browser will interpret the corresponding portion of the output as a tag regardless of the values of untrusted data, and similarly for other structures such as attribute boundaries and JS and CSS string boundaries."

Code Effect Property: "... only code specified by the template author should run as a result of injecting the template output into a page and all code specified by the template author should run as a result of the same."

Least Surprise Property: "A developer (or code reviewer) familiar with HTML, CSS, and JavaScript, who knows that contextual autoescaping happens should be able to look at a `{{.}}` and correctly infer what sanitization happens."

func `HTMLEscape`

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

func [HTMLEscapeString](#)

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

func [HTMLEscaper](#)

```
func HTMLEscaper(args ...interface{}) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

func [IsTrue](#)

```
func IsTrue(val interface{}) (truth, ok bool)
```

IsTrue reports whether the value is 'true', in the sense of not the zero of its type, and whether the value has a meaningful truth value. This is the definition of truth used by if and other such actions.

func [JSEscape](#)

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

func [JSEscapeString](#)

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

func [JSEscaper](#)

```
func JSEscaper(args ...interface{}) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

func [URLQueryEscaper](#)

```
func URLQueryEscaper(args ...interface{}) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

type CSS

```
type CSS string
```

CSS encapsulates known safe content that matches any of:

1. The CSS3 stylesheet production, such as `p { color: purple }`.
2. The CSS3 rule production, such as `a[href=~"https:"].foo#bar`.
3. CSS3 declaration productions, such as `color: red; margin: 2px`.
4. The CSS3 value production, such as `rgba(0, 0, 255, 127)`.

See <https://www.w3.org/TR/css3-syntax/#parsing> and
<https://web.archive.org/web/20090211114933/http://w3.org/TR/css3-syntax#style>

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

type Error

```
type Error struct {
    // ErrorCode describes the kind of error.
    ErrorCode ErrorCode
    // Node is the node that caused the problem, if known.
    // If not nil, it overrides Name and Line.
    Node parse.Node
    // Name is the name of the template in which the error was encountered.
    Name string
    // Line is the line number of the error in the template source or 0.
    Line int
    // Description is a human-readable description of the problem.
    Description string
}
```

Error describes a problem encountered during template Escaping.

func (*Error) Error

```
func (e *Error) Error() string
```

type ErrorCode

```
type ErrorCode int
```

ErrorCode is a code for a kind of error.

```

const (
    // OK indicates the lack of an error.
    OK ErrorCode = iota

    // ErrAmbigContext: "... appears in an ambiguous context within a URL"
    // Example:
    //   <a href="
    //     {{if .C}}
    //     /path/
    //     {{else}}
    //     /search?q=
    //     {{end}}
    //     {{.X}}
    //   ">
    // Discussion:
    //   {{.X}} is in an ambiguous URL context since, depending on {{.C}},
    //   it may be either a URL suffix or a query parameter.
    //   Moving {{.X}} into the condition removes the ambiguity:
    //   <a href="{{if .C}}/path/{{.X}}{{else}}/search?q={{.X}}">
ErrAmbigContext

    // ErrBadHTML: "expected space, attr name, or end of tag, but got ...",
    //   "... in unquoted attr", "... in attribute name"
    // Example:
    //   <a href = /search?q=foo>
    //   <href=foo>
    //   <form na<e=...>
    //   <option selected<
    // Discussion:
    //   This is often due to a typo in an HTML element, but some runes
    //   are banned in tag names, attribute names, and unquoted attribute
    //   values because they can tickle parser ambiguities.
    //   Quoting all attributes is the best policy.
ErrBadHTML

    // ErrBranchEnd: "{{if}} branches end in different contexts"
    // Example:
    //   {{if .C}}<a href="{{end}}{{.X}}
    // Discussion:
    //   Package html/template statically examines each path through an
    //   {{if}}, {{range}}, or {{with}} to escape any following pipelines.
    //   The example is ambiguous since {{.X}} might be an HTML text node,
    //   or a URL prefix in an HTML attribute. The context of {{.X}} is
    //   used to figure out how to escape it, but that context depends on
    //   the run-time value of {{.C}} which is not statically known.
    //
    //   The problem is usually something like missing quotes or angle
    //   brackets, or can be avoided by refactoring to put the two contexts
    //   into different branches of an if, range or with. If the problem
    //   is in a {{range}} over a collection that should never be empty,
    //   adding a dummy {{else}} can help.
ErrBranchEnd

```

```
// ErrEndContext: "... ends in a non-text context: ..."
// Examples:
//   <div
//     <div title="no close quote>
//     <script>f()
// Discussion:
//   Executed templates should produce a DocumentFragment of HTML.
//   Templates that end without closing tags will trigger this error.
//   Templates that should not be used in an HTML context or that
//   produce incomplete Fragments should not be executed directly.
//
//   {{define "main"}} <script>{{template "helper"}}</script> {{end}}
//   {{define "helper"}} document.write(' <div title=" ' ) {{end}}
//
//   "helper" does not produce a valid document fragment, so should
//   not be Executed directly.
```

ErrEndContext

```
// ErrNoSuchTemplate: "no such template ..."
// Examples:
//   {{define "main"}}<div {{template "attrs"}}>{{end}}
//   {{define "attrs"}}href="{{.URL}}">{{end}}
// Discussion:
//   Package html/template looks through template calls to compute the
//   context.
//   Here the {{.URL}} in "attrs" must be treated as a URL when called
//   from "main", but you will get this error if "attrs" is not defined
//   when "main" is parsed.
```

ErrNoSuchTemplate

```
// ErrOutputContext: "cannot compute output context for template ..."
// Examples:
//   {{define "t"}}{{if .T}}{{template "t" .T}}{{end}}{{.H}},{{end}}
// Discussion:
//   A recursive template does not end in the same context in which it
//   starts, and a reliable output context cannot be computed.
//   Look for typos in the named template.
//   If the template should not be called in the named start context,
//   look for calls to that template in unexpected contexts.
//   Maybe refactor recursive templates to not be recursive.
```

ErrOutputContext

```
// ErrPartialCharset: "unfinished JS regexp charset in ..."
// Example:
//   <script>var pattern = /foo[{{.Chars}}]/</script>
// Discussion:
//   Package html/template does not support interpolation into regular
//   expression literal character sets.
```

ErrPartialCharset

```
// ErrPartialEscape: "unfinished escape sequence in ..."
// Example:
//   <script>alert("\{{.X}}")</script>
// Discussion:
```

```
// Package html/template does not support actions following a
// backslash.
// This is usually an error and there are better solutions; for
// example
//     <script>alert("{{.X}}")</script>
// should work, and if {{.X}} is a partial escape sequence such as
// "xA0", mark the whole sequence as safe content: JSStr(`\xA0`)
ErrPartialEscape

// ErrRangeLoopReentry: "on range loop re-entry: ..."
// Example:
//     <script>var x = [{range .}{{.}},{{end}}]</script>
// Discussion:
//     If an iteration through a range would cause it to end in a
//     different context than an earlier pass, there is no single context.
//     In the example, there is missing a quote, so it is not clear
//     whether {{.}} is meant to be inside a JS string or in a JS value
//     context. The second iteration would produce something like
//
//     <script>var x = ['firstValue,'secondValue]</script>
ErrRangeLoopReentry

// ErrSlashAmbig: '/' could start a division or regexp.
// Example:
//     <script>
//         {{if .C}}var x = 1{{end}}
//         /{{.N}}/i.test(x) ? doThis : doThat();
//     </script>
// Discussion:
//     The example above could produce `var x = 1/-2/i.test(s)...`
//     in which the first '/' is a mathematical division operator or it
//     could produce `/{{.N}}/i.test(s)` in which the first '/' starts a
//     regexp literal.
//     Look for missing semicolons inside branches, and maybe add
//     parentheses to make it clear which interpretation you intend.
ErrSlashAmbig

// ErrPredefinedEscaper: "predefined escaper ... disallowed in template"
// Example:
//     <div class={{. | html}}>Hello</div>
// Discussion:
//     Package html/template already contextually escapes all pipelines to
//     produce HTML output safe against code injection. Manually escaping
//     pipeline output using the predefined escapers "html" or "urlquery" is
//     unnecessary, and may affect the correctness or safety of the escaped
//     pipeline output in Go 1.8 and earlier.
//
//     In most cases, such as the given example, this error can be resolved by
//     simply removing the predefined escaper from the pipeline and letting the
//     contextual autoescaper handle the escaping of the pipeline. In other
//     instances, where the predefined escaper occurs in the middle of a
//     pipeline where subsequent commands expect escaped input, e.g.
//         {{.X | html | makeALink}}
//     where makeALink does
```

```

//      return `<a href="'+input+'>link</a>`
// consider refactoring the surrounding template to make use of the
// contextual autoescaper, i.e.
//      <a href="{{.X}}>link</a>
//
// To ease migration to Go 1.9 and beyond, "html" and "urlquery" will
// continue to be allowed as the last command in a pipeline. However, if the
// pipeline occurs in an unquoted attribute value context, "html" is
// disallowed. Avoid using "html" and "urlquery" entirely in new templates.
ErrPredefinedEscaper
)

```

We define codes for each error that manifests while escaping templates, but escaped templates may also fail at runtime.

Output: "ZgotmplZ" Example:

```

  
If the data comes from a trusted source, use content types to exempt it from filtering: URL(`javascript:...`).

## type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) argument evaluates to non-nil during execution, execution terminates and Execute returns that error. FuncMap has the same base type as FuncMap in "text/template", copied here so clients need not import "text/template".

## type HTML

```
type HTML string
```

HTML encapsulates a known safe HTML document fragment. It should not be used for HTML from a third-party, or HTML with unclosed tags or comments. The outputs of a sound HTML sanitizer and a template escaped by this package are fine for use with HTML.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

## type `HTMLAttr`

```
type HTMLAttr string
```

`HTMLAttr` encapsulates an HTML attribute from a trusted source, for example, ``dir="ltr"``.`

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

## type `JS`

```
type JS string
```

`JS` encapsulates a known safe EcmaScript5 Expression, for example, ``(x + y * z())``. Template authors are responsible for ensuring that typed expressions do not break the intended precedence and that there is no statement/expression ambiguity as when passing an expression like `"{ foo: bar() }\n['foo'] ()"`, which is both a valid Expression and a valid Program with a very different meaning.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

Using `JS` to include valid but untrusted JSON is not safe. A safe alternative is to parse the JSON with `json.Unmarshal` and then pass the resultant object into the template, where it will be converted to sanitized JSON when presented in a JavaScript context.

## type `JSStr`

```
type JSStr string
```

`JSStr` encapsulates a sequence of characters meant to be embedded between quotes in a JavaScript expression. The string must match a series of `StringCharacters`:

```
StringCharacter :: SourceCharacter but not `\\` or LineTerminator
| EscapeSequence
```

Note that `LineContinuations` are not allowed. `JSStr("foo\\nbar")` is fine, but `JSStr("foo\\\\nbar")` is not.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

## type `Srcset`

```
type Srcset string
```

`Srcset` encapsulates a known safe `srcset` attribute (see <https://w3c.github.io/html/semantics-embedded-content.html#element-attrdef-img-srcset>).

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

## type Template

```
type Template struct {

 // The underlying template's parse tree, updated to be HTML-safe.
 Tree *parse.Tree
 // contains filtered or unexported fields
}
```

Template is a specialized Template from "text/template" that produces a safe HTML document fragment.

## func Must

```
func Must(t *Template, err error) *Template
```

Must is a helper that wraps a call to a function returning (\*Template, error) and panics if the error is non-nil. It is intended for use in variable initializations such as

```
var t = template.Must(template.New("name").Parse("html"))
```

## func New

```
func New(name string) *Template
```

New allocates a new HTML template with the given name.

## func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the (base) name and (parsed) contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned \*Template is nil.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results. For instance, ParseFiles("a/foo", "b/foo") stores "b/foo" as the template named "foo", while "a/foo" is unavailable.

## func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

## func (\*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
```

AddParseTree creates a new template with the name and parse tree and associates it with t.

It returns an error if t or any associated template has already been executed.

## func (\*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

It returns an error if t has already been executed.

## func (\*Template) DefinedTemplates

```
func (t *Template) DefinedTemplates() string
```

DefinedTemplates returns a string listing the defined templates, prefixed by the string "; defined templates are: ". If there are none, it returns the empty string. Used to generate an error message.

## func (\*Template) Delims

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

## func (\*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

Execute applies a parsed template to the specified data object, writing the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

## func (\*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

## func (\*Template) Funcs

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs adds the elements of the argument map to the template's function map. It must be called before the template is parsed. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

## func (\*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

Lookup returns the template with the given name that is associated with t, or nil if there is no such template.

## func (\*Template) Name

```
func (t *Template) Name() string
```

Name returns the name of the template.

## func (\*Template) New

```
func (t *Template) New(name string) *Template
```

New allocates a new HTML template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a {{template}} action.

If a template with the given name already exists, the new HTML template will replace it. The existing template will be reset and disassociated with t.

## func (\*Template) Option

```
func (t *Template) Option(opt ...string) *Template
```

Option sets options for the template. Options are described by strings, either a simple string or "key=value". There can be at most one equals sign in an option string. If the option string is unrecognized or otherwise invalid, Option panics.

Known options:

missingkey: Control the behavior during execution if a map is indexed with a key that is not present in the map.

```
"missingkey=default" or "missingkey=invalid"
 The default behavior: Do nothing and continue execution.
 If printed, the result of the index operation is the string
 "<no value>".
"missingkey=zero"
 The operation returns the zero value for the map type's element.
"missingkey=error"
 Execution stops immediately with an error.
```

## func (\*Template) Parse

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse parses text as a template body for t. Named template definitions ({{define ...}} or {{block ...}} statements) in text define additional templates associated with t and are removed from the definition of t itself.

Templates can be redefined in successive calls to Parse, before the first use of Execute on t or any associated template. A template definition with a body containing only white space and comments is considered empty and will not replace an existing template's body. This allows using Parse to add new named template definitions without overwriting the main template body.

## func (\*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

ParseFiles returns an error if t or any associated template has already been executed.

## func (\*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseGlob parses the template definitions in the files identified by the pattern and associates the resulting templates with t. The files are matched according to the semantics of `filepath.Match`, and the pattern must match at least one file. ParseGlob is equivalent to calling `t.ParseFiles` with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

ParseGlob returns an error if t or any associated template has already been executed.

## func (\*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates returns a slice of the templates associated with t, including t itself.

## type URL

```
type URL string
```

URL encapsulates a known safe URL or URL substring (see [RFC 3986](#)). A URL like ``javascript:checkThatFormNotEditedBeforeLeavingPage()`` from a trusted source should go in the page, but by default dynamic ``javascript:`` URLs are filtered out since they are a frequently exploited injection vector.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

# Package http

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package http provides HTTP client and server implementations.

Get, Head, Post, and PostForm make HTTP (or HTTPS) requests:

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form",
 url.Values{"key": {"Value"}, "id": {"123"}})
```

The client must close the response body when finished with it:

```
resp, err := http.Get("http://example.com/")
if err != nil {
 // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

For control over HTTP client headers, redirect policy, and other settings, create a Client:

```
client := &http.Client{
 CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
// ...

req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

For control over proxies, TLS configuration, keep-alives, compression, and other settings, create a Transport:

```

tr := &http.Transport{
 MaxIdleConns: 10,
 IdleConnTimeout: 30 * time.Second,
 DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")

```

Clients and Transports are safe for concurrent use by multiple goroutines and for efficiency should only be created once and re-used.

ListenAndServe starts an HTTP server with a given address and handler. The handler is usually nil, which means to use DefaultServeMux. Handle and HandleFunc add handlers to DefaultServeMux:

```

http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))

```

More control over the server's behavior is available by creating a custom Server:

```

s := &http.Server{
 Addr: ":8080",
 Handler: myHandler,
 ReadTimeout: 10 * time.Second,
 WriteTimeout: 10 * time.Second,
 MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())

```

Starting with Go 1.6, the http package has transparent support for the HTTP/2 protocol when using HTTPS. Programs that must disable HTTP/2 can do so by setting Transport.TLSNextProto (for clients) or Server.TLSNextProto (for servers) to a non-nil, empty map. Alternatively, the following GODEBUG environment variables are currently supported:

```

GODEBUG=http2client=0 # disable HTTP/2 client support
GODEBUG=http2server=0 # disable HTTP/2 server support
GODEBUG=http2debug=1 # enable verbose HTTP/2 debug logs
GODEBUG=http2debug=2 # ... even more verbose, with frame dumps

```

The GODEBUG variables are not covered by Go's API compatibility promise. Please report any issues before disabling HTTP/2 support: <https://golang.org/s/http2bug>

The http package's Transport and Server both automatically enable HTTP/2 support for simple configurations. To enable HTTP/2 for more complex configurations, to use lower-level HTTP/2

features, or to use a newer version of Go's http2 package, import "golang.org/x/net/http2" directly and use its `ConfigureTransport` and/or `ConfigureServer` functions. Manually configuring HTTP/2 via the `golang.org/x/net/http2` package takes precedence over the `net/http` package's built-in HTTP/2 support.

## Constants

```
const (
 MethodGet = "GET"
 MethodHead = "HEAD"
 MethodPost = "POST"
 MethodPut = "PUT"
 MethodPatch = "PATCH" // RFC 5789
 MethodDelete = "DELETE"
 MethodConnect = "CONNECT"
 MethodOptions = "OPTIONS"
 MethodTrace = "TRACE"
)
```

Common HTTP methods.

Unless otherwise noted, these are defined in [RFC 7231](#) section 4.3.

```
const (
 StatusContinue = 100 // RFC 7231, 6.2.1
 StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
 StatusProcessing = 102 // RFC 2518, 10.1
 StatusEarlyHints = 103 // RFC 8297

 StatusOK = 200 // RFC 7231, 6.3.1
 StatusCreated = 201 // RFC 7231, 6.3.2
 StatusAccepted = 202 // RFC 7231, 6.3.3
 StatusNonAuthoritativeInfo = 203 // RFC 7231, 6.3.4
 StatusNoContent = 204 // RFC 7231, 6.3.5
 StatusResetContent = 205 // RFC 7231, 6.3.6
 StatusPartialContent = 206 // RFC 7233, 4.1
 StatusMultiStatus = 207 // RFC 4918, 11.1
 StatusAlreadyReported = 208 // RFC 5842, 7.1
 StatusIMUsed = 226 // RFC 3229, 10.4.1

 StatusMultipleChoices = 300 // RFC 7231, 6.4.1
 StatusMovedPermanently = 301 // RFC 7231, 6.4.2
 StatusFound = 302 // RFC 7231, 6.4.3
 StatusSeeOther = 303 // RFC 7231, 6.4.4
 StatusNotModified = 304 // RFC 7232, 4.1
 StatusUseProxy = 305 // RFC 7231, 6.4.5

 StatusTemporaryRedirect = 307 // RFC 7231, 6.4.7
 StatusPermanentRedirect = 308 // RFC 7538, 3

 StatusBadRequest = 400 // RFC 7231, 6.5.1
 StatusUnauthorized = 401 // RFC 7235, 3.1
```

```

StatusPaymentRequired = 402 // RFC 7231, 6.5.2
StatusForbidden = 403 // RFC 7231, 6.5.3
StatusNotFound = 404 // RFC 7231, 6.5.4
StatusMethodNotAllowed = 405 // RFC 7231, 6.5.5
StatusNotAcceptable = 406 // RFC 7231, 6.5.6
StatusProxyAuthRequired = 407 // RFC 7235, 3.2
StatusRequestTimeout = 408 // RFC 7231, 6.5.7
StatusConflict = 409 // RFC 7231, 6.5.8
StatusGone = 410 // RFC 7231, 6.5.9
StatusLengthRequired = 411 // RFC 7231, 6.5.10
StatusPreconditionFailed = 412 // RFC 7232, 4.2
StatusRequestEntityTooLarge = 413 // RFC 7231, 6.5.11
StatusRequestURITooLong = 414 // RFC 7231, 6.5.12
StatusUnsupportedMediaType = 415 // RFC 7231, 6.5.13
StatusRequestedRangeNotSatisfiable = 416 // RFC 7233, 4.4
StatusExpectationFailed = 417 // RFC 7231, 6.5.14
StatusTeapot = 418 // RFC 7168, 2.3.3
StatusMisdirectedRequest = 421 // RFC 7540, 9.1.2
StatusUnprocessableEntity = 422 // RFC 4918, 11.2
StatusLocked = 423 // RFC 4918, 11.3
StatusFailedDependency = 424 // RFC 4918, 11.4
StatusTooEarly = 425 // RFC 8470, 5.2.
StatusUpgradeRequired = 426 // RFC 7231, 6.5.15
StatusPreconditionRequired = 428 // RFC 6585, 3
StatusTooManyRequests = 429 // RFC 6585, 4
StatusRequestHeaderFieldsTooLarge = 431 // RFC 6585, 5
StatusUnavailableForLegalReasons = 451 // RFC 7725, 3

StatusInternalServerError = 500 // RFC 7231, 6.6.1
StatusNotImplemented = 501 // RFC 7231, 6.6.2
StatusBadGateway = 502 // RFC 7231, 6.6.3
StatusServiceUnavailable = 503 // RFC 7231, 6.6.4
StatusGatewayTimeout = 504 // RFC 7231, 6.6.5
StatusHTTPVersionNotSupported = 505 // RFC 7231, 6.6.6
StatusVariantAlsoNegotiates = 506 // RFC 2295, 8.1
StatusInsufficientStorage = 507 // RFC 4918, 11.5
StatusLoopDetected = 508 // RFC 5842, 7.2
StatusNotExtended = 510 // RFC 2774, 7
StatusNetworkAuthenticationRequired = 511 // RFC 6585, 6
)

```

HTTP status codes as registered with IANA. See: <https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>

```
const DefaultMaxHeaderBytes = 1 << 20 // 1 MB
```

DefaultMaxHeaderBytes is the maximum permitted size of the headers in an HTTP request. This can be overridden by setting Server.MaxHeaderBytes.

```
const DefaultMaxIdleConnsPerHost = 2
```

DefaultMaxIdleConnsPerHost is the default value of Transport's MaxIdleConnsPerHost.

```
const TimeFormat = "Mon, 02 Jan 2006 15:04:05 GMT"
```

TimeFormat is the time format to use when generating times in HTTP headers. It is like time.RFC1123 but hard-codes GMT as the time zone. The time being formatted must be in UTC for Format to generate the correct format.

For parsing this time format, see ParseTime.

```
const TrailerPrefix = "Trailer:"
```

TrailerPrefix is a magic prefix for ResponseWriter.Header map keys that, if present, signals that the map entry is actually for the response trailers, and not the response headers. The prefix is stripped after the ServeHTTP call finishes and the values are sent in the trailers.

This mechanism is intended only for trailers that are not known prior to the headers being written. If the set of trailers is fixed or known before the header is written, the normal Go trailers mechanism is preferred:

```
https://golang.org/pkg/net/http/#ResponseWriter
https://golang.org/pkg/net/http/#example_ResponseWriter_trailers
```

## Variables

```
var (
 // ErrNotSupported is returned by the Push method of Pusher
 // implementations to indicate that HTTP/2 Push support is not
 // available.
 ErrNotSupported = &ProtocolError{"feature not supported"}

 // Deprecated: ErrUnexpectedTrailer is no longer returned by
 // anything in the net/http package. Callers should not
 // compare errors against this variable.
 ErrUnexpectedTrailer = &ProtocolError{"trailer header without chunked transfer en

 // ErrMissingBoundary is returned by Request.MultipartReader when the
 // request's Content-Type does not include a "boundary" parameter.
 ErrMissingBoundary = &ProtocolError{"no multipart boundary param in Content-Type"

 // ErrNotMultipart is returned by Request.MultipartReader when the
 // request's Content-Type is not multipart/form-data.
 ErrNotMultipart = &ProtocolError{"request Content-Type isn't multipart/form-data"

 // Deprecated: ErrHeaderTooLong is no longer returned by
 // anything in the net/http package. Callers should not
 // compare errors against this variable.
 ErrHeaderTooLong = &ProtocolError{"header too long"}
```

```

// Deprecated: ErrShortBody is no longer returned by
// anything in the net/http package. Callers should not
// compare errors against this variable.
ErrShortBody = &ProtocolError{"entity body too short"}

// Deprecated: ErrMissingContentLength is no longer returned by
// anything in the net/http package. Callers should not
// compare errors against this variable.
ErrMissingContentLength = &ProtocolError{"missing ContentLength in HEAD response"
)

```

```

var (
 // ErrBodyNotAllowed is returned by ResponseWriter.Write calls
 // when the HTTP method or response code does not permit a
 // body.
 ErrBodyNotAllowed = errors.New("http: request method or response status code does

 // ErrHijacked is returned by ResponseWriter.Write calls when
 // the underlying connection has been hijacked using the
 // Hijacker interface. A zero-byte write on a hijacked
 // connection will return ErrHijacked without any other side
 // effects.
 ErrHijacked = errors.New("http: connection has been hijacked")

 // ErrContentLength is returned by ResponseWriter.Write calls
 // when a Handler set a Content-Length response header with a
 // declared size and then attempted to write more bytes than
 // declared.
 ErrContentLength = errors.New("http: wrote more than the declared Content-Length"

 // Deprecated: ErrWriteAfterFlush is no longer returned by
 // anything in the net/http package. Callers should not
 // compare errors against this variable.
 ErrWriteAfterFlush = errors.New("unused")
)

```

Errors used by the HTTP server.

```

var (
 // ServerContextKey is a context key. It can be used in HTTP
 // handlers with Context.Value to access the server that
 // started the handler. The associated value will be of
 // type *Server.
 ServerContextKey = &contextKey{"http-server"}

 // LocalAddrContextKey is a context key. It can be used in
 // HTTP handlers with Context.Value to access the local
 // address the connection arrived on.
 // The associated value will be of type net.Addr.
 LocalAddrContextKey = &contextKey{"local-addr"}
)

```

```
var DefaultClient = &Client{}
```

DefaultClient is the default Client and is used by Get, Head, and Post.

```
var DefaultServeMux = &defaultServeMux
```

DefaultServeMux is the default ServeMux used by Serve.

```
var ErrAbortHandler = errors.New("net/http: abort Handler")
```

ErrAbortHandler is a sentinel panic value to abort a handler. While any panic from ServeHTTP aborts the response to the client, panicking with ErrAbortHandler also suppresses logging of a stack trace to the server's error log.

```
var ErrBodyReadAfterClose = errors.New("http: invalid Read on closed Body")
```

ErrBodyReadAfterClose is returned when reading a Request or Response Body after the body has been closed. This typically happens when the body is read after an HTTP Handler calls WriteHeader or Write on its ResponseWriter.

```
var ErrHandlerTimeout = errors.New("http: Handler timeout")
```

ErrHandlerTimeout is returned on ResponseWriter Write calls in handlers which have timed out.

```
var ErrLineTooLong = internal.ErrLineTooLong
```

ErrLineTooLong is returned when reading request or response bodies with malformed chunked encoding.

```
var ErrMissingFile = errors.New("http: no such file")
```

ErrMissingFile is returned by FormFile when the provided file field name is either not present in the request or not a file field.

```
var ErrNoCookie = errors.New("http: named cookie not present")
```

ErrNoCookie is returned by Request's Cookie method when a cookie is not found.

```
var ErrNoLocation = errors.New("http: no Location header in response")
```

ErrNoLocation is returned by Response's Location method when no Location header is present.

```
var ErrServerClosed = errors.New("http: Server closed")
```

ErrServerClosed is returned by the Server's Serve, ServeTLS, ListenAndServe, and ListenAndServeTLS methods after a call to Shutdown or Close.

```
var ErrSkipAltProtocol = errors.New("net/http: skip alternate protocol")
```

ErrSkipAltProtocol is a sentinel error value defined by Transport.RegisterProtocol.

```
var ErrUseLastResponse = errors.New("net/http: use last response")
```

ErrUseLastResponse can be returned by Client.CheckRedirect hooks to control how redirects are processed. If returned, the next request is not sent and the most recent response is returned with its body unclosed.

```
var NoBody = noBody{}
```

NoBody is an io.ReadCloser with no bytes. Read always returns EOF and Close always returns nil. It can be used in an outgoing client request to explicitly signal that a request has zero bytes. An alternative, however, is to simply set Request.Body to nil.

## func CanonicalHeaderKey

```
func CanonicalHeaderKey(s string) string
```

CanonicalHeaderKey returns the canonical format of the header key s. The canonicalization converts the first letter and any letter following a hyphen to upper case; the rest are converted to lowercase. For example, the canonical key for "accept-encoding" is "Accept-Encoding". If s contains a space or invalid header field bytes, it is returned without modifications.

## func DetectContentType

```
func DetectContentType(data []byte) string
```

DetectContentType implements the algorithm described at <https://mimesniff.spec.whatwg.org/> to determine the Content-Type of the given data. It considers at most the first 512 bytes of data. DetectContentType always returns a valid MIME type: if it cannot determine a more specific one, it returns "application/octet-stream".

## func Error

```
func Error(w ResponseWriter, error string, code int)
```

Error replies to the request with the specified error message and HTTP code. It does not otherwise end the request; the caller should ensure no further writes are done to w. The error message should be plain text.

## func Handle

```
func Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

## func HandleFunc

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc registers the handler function for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

## func ListenAndServe

```
func ListenAndServe(addr string, handler Handler) error
```

ListenAndServe listens on the TCP network address addr and then calls Serve with handler to handle requests on incoming connections. Accepted connections are configured to enable TCP keep-alives.

The handler is typically nil, in which case the DefaultServeMux is used.

ListenAndServe always returns a non-nil error.

## func ListenAndServeTLS

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

ListenAndServeTLS acts identically to ListenAndServe, except that it expects HTTPS connections. Additionally, files containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

## func MaxBytesReader

```
func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.ReadCloser
```

MaxBytesReader is similar to io.LimitReader but is intended for limiting the size of incoming request bodies. In contrast to io.LimitReader, MaxBytesReader's result is a ReadCloser, returns a non-EOF error for a Read beyond the limit, and closes the underlying reader when its Close method is called.

MaxBytesReader prevents clients from accidentally or maliciously sending a large request and wasting server resources.

## func NotFound

```
func NotFound(w ResponseWriter, r *Request)
```

NotFound replies to the request with an HTTP 404 not found error.

## func ParseHTTPVersion

```
func ParseHTTPVersion(vers string) (major, minor int, ok bool)
```

ParseHTTPVersion parses an HTTP version string. "HTTP/1.0" returns (1, 0, true).

## func ParseTime

```
func ParseTime(text string) (t time.Time, err error)
```

ParseTime parses a time header (such as the Date: header), trying each of the three formats allowed by HTTP/1.1: TimeFormat, time.RFC850, and time.ANSIC.

## func ProxyFromEnvironment

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

ProxyFromEnvironment returns the URL of the proxy to use for a given request, as indicated by the environment variables HTTP\_PROXY, HTTPS\_PROXY and NO\_PROXY (or the lowercase versions thereof). HTTPS\_PROXY takes precedence over HTTP\_PROXY for https requests.

The environment values may be either a complete URL or a "host[:port]", in which case the "http" scheme is assumed. An error is returned if the value is a different form.

A nil URL and nil error are returned if no proxy is defined in the environment, or a proxy should not be used for the given request, as defined by NO\_PROXY.

As a special case, if req.URL.Host is "localhost" (with or without a port number), then a nil URL and nil error will be returned.

## func ProxyURL

```
func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
```

ProxyURL returns a proxy function (for use in a Transport) that always returns the same URL.

## func Redirect

```
func Redirect(w ResponseWriter, r *Request, url string, code int)
```

Redirect replies to the request with a redirect to url, which may be a path relative to the request path.

The provided code should be in the 3xx range and is usually StatusMovedPermanently, StatusFound or StatusSeeOther.

If the Content-Type header has not been set, Redirect sets it to "text/html; charset=utf-8" and writes a small HTML body. Setting the Content-Type header to any value, including nil, disables that behavior.

## func Serve

```
func Serve(l net.Listener, handler Handler) error
```

Serve accepts incoming HTTP connections on the listener l, creating a new service goroutine for each. The service goroutines read requests and then call handler to reply to them.

The handler is typically nil, in which case the DefaultServeMux is used.

HTTP/2 support is only enabled if the Listener returns \*tls.Conn connections and they were configured with "h2" in the TLS Config.NextProtos.

Serve always returns a non-nil error.

## func ServeContent

```
func ServeContent(w ResponseWriter, req *Request, name string, modtime time.Time, con
```

ServeContent replies to the request using the content in the provided ReadSeeker. The main benefit of ServeContent over io.Copy is that it handles Range requests properly, sets the MIME type, and handles If-Match, If-Unmodified-Since, If-None-Match, If-Modified-Since, and If-Range requests.

If the response's Content-Type header is not set, ServeContent first tries to deduce the type from name's file extension and, if that fails, falls back to reading the first block of the content and passing it to DetectContentType. The name is otherwise unused; in particular it can be empty and is never sent in the response.

If modtime is not the zero time or Unix epoch, ServeContent includes it in a Last-Modified header in the response. If the request includes an If-Modified-Since header, ServeContent uses modtime to decide whether the content needs to be sent at all.

The content's Seek method must work: ServeContent uses a seek to the end of the content to determine its size.

If the caller has set w's ETag header formatted per [RFC 7232, section 2.3](#), ServeContent uses it to handle requests using If-Match, If-None-Match, or If-Range.

Note that \*os.File implements the io.ReadSeeker interface.

## func ServeFile

```
func ServeFile(w ResponseWriter, r *Request, name string)
```

ServeFile replies to the request with the contents of the named file or directory.

If the provided file or directory name is a relative path, it is interpreted relative to the current directory and may ascend to parent directories. If the provided name is constructed from user input, it should be sanitized before calling ServeFile.

As a precaution, ServeFile will reject requests where r.URL.Path contains a ".." path element; this protects against callers who might unsafely use filepath.Join on r.URL.Path without sanitizing it and then use that filepath.Join result as the name argument.

As another special case, ServeFile redirects any request where r.URL.Path ends in "/index.html" to the same path, without the final "index.html". To avoid such redirects either modify the path or use ServeContent.

Outside of those two special cases, ServeFile does not use r.URL.Path for selecting the file or directory to serve; only the file or directory provided in the name argument is used.

## func ServeTLS

```
func ServeTLS(l net.Listener, handler Handler, certFile, keyFile string) error
```

ServeTLS accepts incoming HTTPS connections on the listener l, creating a new service goroutine for each. The service goroutines read requests and then call handler to reply to them.

The handler is typically nil, in which case the DefaultServeMux is used.

Additionally, files containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

ServeTLS always returns a non-nil error.

## func SetCookie

```
func SetCookie(w ResponseWriter, cookie *Cookie)
```

SetCookie adds a Set-Cookie header to the provided ResponseWriter's headers. The provided cookie must have a valid Name. Invalid cookies may be silently dropped.

## func StatusText

```
func StatusText(code int) string
```

StatusText returns a text for the HTTP status code. It returns the empty string if the code is unknown.

## type Client

```

type Client struct {
 // Transport specifies the mechanism by which individual
 // HTTP requests are made.
 // If nil, DefaultTransport is used.
 Transport RoundTripper

 // CheckRedirect specifies the policy for handling redirects.
 // If CheckRedirect is not nil, the client calls it before
 // following an HTTP redirect. The arguments req and via are
 // the upcoming request and the requests made already, oldest
 // first. If CheckRedirect returns an error, the Client's Get
 // method returns both the previous Response (with its Body
 // closed) and CheckRedirect's error (wrapped in a url.Error)
 // instead of issuing the Request req.
 // As a special case, if CheckRedirect returns ErrUseLastResponse,
 // then the most recent response is returned with its body
 // unclosed, along with a nil error.
 //
 // If CheckRedirect is nil, the Client uses its default policy,
 // which is to stop after 10 consecutive requests.
 CheckRedirect func(req *Request, via []*Request) error

 // Jar specifies the cookie jar.
 //
 // The Jar is used to insert relevant cookies into every
 // outbound Request and is updated with the cookie values
 // of every inbound Response. The Jar is consulted for every
 // redirect that the Client follows.
 //
 // If Jar is nil, cookies are only sent if they are explicitly
 // set on the Request.
 Jar CookieJar

 // Timeout specifies a time limit for requests made by this
 // Client. The timeout includes connection time, any
 // redirects, and reading the response body. The timer remains
 // running after Get, Head, Post, or Do return and will
 // interrupt reading of the Response.Body.
 //
 // A Timeout of zero means no timeout.
 //
 // The Client cancels requests to the underlying Transport
 // as if the Request's Context ended.
 //
 // For compatibility, the Client will also use the deprecated
 // CancelRequest method on Transport if found. New
 // RoundTripper implementations should use the Request's Context
 // for cancellation instead of implementing CancelRequest.
 Timeout time.Duration
}

```

A Client is an HTTP client. Its zero value (DefaultClient) is a usable client that uses DefaultTransport.

The Client's Transport typically has internal state (cached TCP connections), so Clients should be reused instead of created as needed. Clients are safe for concurrent use by multiple goroutines.

A Client is higher-level than a RoundTripper (such as Transport) and additionally handles HTTP details such as cookies and redirects.

When following redirects, the Client will forward all headers set on the initial Request except:

- when forwarding sensitive headers like "Authorization", "WWW-Authenticate", and "Cookie" to untrusted targets. These headers will be ignored when following a redirect to a domain that is not a subdomain match or exact match of the initial domain. For example, a redirect from "foo.com" to either "foo.com" or "sub.foo.com" will forward the sensitive headers, but a redirect to "bar.com" will not.
- when forwarding the "Cookie" header with a non-nil cookie Jar. Since each redirect may mutate the state of the cookie jar, a redirect may possibly alter a cookie set in the initial request. When forwarding the "Cookie" header, any mutated cookies will be omitted, with the expectation that the Jar will insert those mutated cookies with the updated values (assuming the origin matches). If Jar is nil, the initial cookies are forwarded without change.

## func (\*Client) **ClosidleConnections**

```
func (c *Client) CloseIdleConnections()
```

ClosidleConnections closes any connections on its Transport which were previously connected from previous requests but are now sitting idle in a "keep-alive" state. It does not interrupt any connections currently in use.

If the Client's Transport does not have a ClosidleConnections method then this method does nothing.

## func (\*Client) **Do**

```
func (c *Client) Do(req *Request) (*Response, error)
```

Do sends an HTTP request and returns an HTTP response, following policy (such as redirects, cookies, auth) as configured on the client.

An error is returned if caused by client policy (such as CheckRedirect), or failure to speak HTTP (such as a network connectivity problem). A non-2xx status code doesn't cause an error.

If the returned error is nil, the Response will contain a non-nil Body which the user is expected to close. If the Body is not both read to EOF and closed, the Client's underlying RoundTripper (typically Transport) may not be able to re-use a persistent TCP connection to the server for a subsequent "keep-alive" request.

The request Body, if non-nil, will be closed by the underlying Transport, even on errors.

On error, any Response can be ignored. A non-nil Response with a non-nil error only occurs when CheckRedirect fails, and even then the returned Response.Body is already closed.

Generally Get, Post, or PostForm will be used instead of Do.

If the server replies with a redirect, the Client first uses the CheckRedirect function to determine whether the redirect should be followed. If permitted, a 301, 302, or 303 redirect causes subsequent requests to use HTTP method GET (or HEAD if the original request was HEAD), with no body. A 307 or 308 redirect preserves the original HTTP method and body, provided that the Request.GetBody function is defined. The NewRequest function automatically sets GetBody for common standard library body types.

Any returned error will be of type \*url.Error. The url.Error value's Timeout method will report true if request timed out or was canceled.

## func (\*Client) Get

```
func (c *Client) Get(url string) (resp *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect after calling the Client's CheckRedirect function:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)
```

An error is returned if the Client's CheckRedirect function fails or if there was an HTTP protocol error. A non-2xx response doesn't cause an error. Any returned error will be of type \*url.Error. The url.Error value's Timeout method will report true if the request timed out.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

To make a request with custom headers, use NewRequest and Client.Do.

## func (\*Client) Head

```
func (c *Client) Head(url string) (resp *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the following redirect codes, Head follows the redirect after calling the Client's CheckRedirect function:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
```

307 (Temporary Redirect)  
308 (Permanent Redirect)

## func (\*Client) Post

```
func (c *Client) Post(url, contentType string, body io.Reader) (resp *Response, err error)
```

Post issues a POST to the specified URL.

Caller should close resp.Body when done reading from it.

If the provided body is an io.Closer, it is closed after the request.

To set custom headers, use NewRequest and Client.Do.

See the Client.Do method documentation for details on how redirects are handled.

## func (\*Client) PostForm

```
func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)
```

PostForm issues a POST to the specified URL, with data's keys and values URL-encoded as the request body.

The Content-Type header is set to application/x-www-form-urlencoded. To set other headers, use NewRequest and Client.Do.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

See the Client.Do method documentation for details on how redirects are handled.

## type CloseNotifier

```
type CloseNotifier interface {
 // CloseNotify returns a channel that receives at most a
 // single value (true) when the client connection has gone
 // away.
 //
 // CloseNotify may wait to notify until Request.Body has been
 // fully read.
 //
 // After the Handler has returned, there is no guarantee
 // that the channel receives a value.
 //
 // If the protocol is HTTP/1.1 and CloseNotify is called while
 // processing an idempotent request (such a GET) while
 // HTTP/1.1 pipelining is in use, the arrival of a subsequent
 // pipelined request may cause a value to be sent on the
 // returned channel. In practice HTTP/1.1 pipelining is not
}
```

```
// enabled in browsers and not seen often in the wild. If this
// is a problem, use HTTP/2 or only use CloseNotify on methods
// such as POST.
CloseNotify() <-chan bool
}
```

The CloseNotifier interface is implemented by ResponseWriters which allow detecting when the underlying connection has gone away.

This mechanism can be used to cancel long operations on the server if the client has disconnected before the response is ready.

Deprecated: the CloseNotifier interface predates Go's context package. New code should use Request.Context instead.

## type ConnState

```
type ConnState int
```

A ConnState represents the state of a client connection to a server. It's used by the optional Server.ConnState hook.

```
const (
 // StateNew represents a new connection that is expected to
 // send a request immediately. Connections begin at this
 // state and then transition to either StateActive or
 // StateClosed.
 StateNew ConnState = iota

 // StateActive represents a connection that has read 1 or more
 // bytes of a request. The Server.ConnState hook for
 // StateActive fires before the request has entered a handler
 // and doesn't fire again until the request has been
 // handled. After the request is handled, the state
 // transitions to StateClosed, StateHijacked, or StateIdle.
 // For HTTP/2, StateActive fires on the transition from zero
 // to one active request, and only transitions away once all
 // active requests are complete. That means that ConnState
 // cannot be used to do per-request work; ConnState only notes
 // the overall state of the connection.
 StateActive

 // StateIdle represents a connection that has finished
 // handling a request and is in the keep-alive state, waiting
 // for a new request. Connections transition from StateIdle
 // to either StateActive or StateClosed.
 StateIdle

 // StateHijacked represents a hijacked connection.
 // This is a terminal state. It does not transition to StateClosed.
 StateHijacked
)
```

```
// StateClosed represents a closed connection.
// This is a terminal state. Hijacked connections do not
// transition to StateClosed.
StateClosed
)
```

## func (ConnState) String

```
func (c ConnState) String() string
```

## type Cookie

```
type Cookie struct {
 Name string
 Value string

 Path string // optional
 Domain string // optional
 Expires time.Time // optional
 RawExpires string // for reading cookies only

 // MaxAge=0 means no 'Max-Age' attribute specified.
 // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
 // MaxAge>0 means Max-Age attribute present and given in seconds
 MaxAge int
 Secure bool
 HttpOnly bool
 SameSite SameSite
 Raw string
 Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

A Cookie represents an HTTP cookie as sent in the Set-Cookie header of an HTTP response or the Cookie header of an HTTP request.

See <https://tools.ietf.org/html/rfc6265> for details.

## func (\*Cookie) String

```
func (c *Cookie) String() string
```

String returns the serialization of the cookie for use in a Cookie header (if only Name and Value are set) or a Set-Cookie response header (if other fields are set). If c is nil or c.Name is invalid, the empty string is returned.

## type CookieJar

```
type CookieJar interface {
 // SetCookies handles the receipt of the cookies in a reply for the
 // given URL. It may or may not choose to save the cookies, depending
 // on the jar's policy and implementation.
 SetCookies(u *url.URL, cookies []*Cookie)

 // Cookies returns the cookies to send in a request for the given URL.
 // It is up to the implementation to honor the standard cookie use
 // restrictions such as in RFC 6265.
 Cookies(u *url.URL) []*Cookie
}
```

A CookieJar manages storage and use of cookies in HTTP requests.

Implementations of CookieJar must be safe for concurrent use by multiple goroutines.

The `net/http/cookiejar` package provides a CookieJar implementation.

## type Dir

```
type Dir string
```

A Dir implements FileSystem using the native file system restricted to a specific directory tree.

While the `FileSystem.Open` method takes `'/'`-separated paths, a Dir's string value is a filename on the native file system, not a URL, so it is separated by `filepath.Separator`, which isn't necessarily `'/'`.

Note that Dir could expose sensitive files and directories. Dir will follow symlinks pointing out of the directory tree, which can be especially dangerous if serving from a directory in which users are able to create arbitrary symlinks. Dir will also allow access to files and directories starting with a period, which could expose sensitive directories like `.git` or sensitive files like `.htpasswd`. To exclude files with a leading period, remove the files/directories from the server or create a custom FileSystem implementation.

An empty Dir is treated as `"."`.

## func (Dir) Open

```
func (d Dir) Open(name string) (File, error)
```

Open implements FileSystem using `os.Open`, opening files for reading rooted and relative to the directory d.

## type File

```
type File interface {
 io.Closer
 io.Reader
 io.Seeker
```

```
 ReadDir(count int) ([]os.FileInfo, error)
 Stat() (os.FileInfo, error)
}
```

A File is returned by a FileSystem's Open method and can be served by the FileServer implementation.

The methods should behave the same as those on an \*os.File.

## type FileSystem

```
type FileSystem interface {
 Open(name string) (File, error)
}
```

A FileSystem implements access to a collection of named files. The elements in a file path are separated by slash ('/', U+002F) characters, regardless of host operating system convention.

## type Flusher

```
type Flusher interface {
 // Flush sends any buffered data to the client.
 Flush()
}
```

The Flusher interface is implemented by ResponseWriters that allow an HTTP handler to flush buffered data to the client.

The default HTTP/1.x and HTTP/2 ResponseWriter implementations support Flusher, but ResponseWriter wrappers may not. Handlers should always test for this ability at runtime.

Note that even for ResponseWriters that support Flush, if the client is connected through an HTTP proxy, the buffered data may not reach the client until the response completes.

## type Handler

```
type Handler interface {
 ServeHTTP(ResponseWriter, *Request)
}
```

A Handler responds to an HTTP request.

ServeHTTP should write reply headers and data to the ResponseWriter and then return. Returning signals that the request is finished; it is not valid to use the ResponseWriter or read from the Request.Body after or concurrently with the completion of the ServeHTTP call.

Depending on the HTTP client software, HTTP protocol version, and any intermediaries between the client and the Go server, it may not be possible to read from the Request.Body after writing to the ResponseWriter. Cautious handlers should read the Request.Body first, and then reply.

Except for reading the body, handlers should not modify the provided Request.

If ServeHTTP panics, the server (the caller of ServeHTTP) assumes that the effect of the panic was isolated to the active request. It recovers the panic, logs a stack trace to the server error log, and either closes the network connection or sends an HTTP/2 RST\_STREAM, depending on the HTTP protocol. To abort a handler so the client sees an interrupted response but the server doesn't log an error, panic with the value ErrAbortHandler.

## func `FileServer`

```
func FileServer(root FileSystem) Handler
```

FileServer returns a handler that serves HTTP requests with the contents of the file system rooted at root.

To use the operating system's file system implementation, use http.Dir:

```
http.Handle("/", http.FileServer(http.Dir("/tmp")))
```

As a special case, the returned file server redirects any request ending in "/index.html" to the same path, without the final "index.html".

## func `NotFoundHandler`

```
func NotFoundHandler() Handler
```

NotFoundHandler returns a simple request handler that replies to each request with a ``404 page not found" reply.

## func `RedirectHandler`

```
func RedirectHandler(url string, code int) Handler
```

RedirectHandler returns a request handler that redirects each request it receives to the given url using the given status code.

The provided code should be in the 3xx range and is usually StatusMovedPermanently, StatusFound or StatusSeeOther.

## func `StripPrefix`

```
func StripPrefix(prefix string, h Handler) Handler
```

StripPrefix returns a handler that serves HTTP requests by removing the given prefix from the request URL's Path and invoking the handler h. StripPrefix handles a request for a path that doesn't begin with prefix by replying with an HTTP 404 not found error.

## func `TimeoutHandler`

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

`TimeoutHandler` returns a Handler that runs `h` with the given time limit.

The new Handler calls `h.ServeHTTP` to handle each request, but if a call runs for longer than its time limit, the handler responds with a 503 Service Unavailable error and the given message in its body. (If `msg` is empty, a suitable default message will be sent.) After such a timeout, writes by `h` to its `ResponseWriter` will return `ErrHandlerTimeout`.

`TimeoutHandler` supports the `Pusher` interface but does not support the `Hijacker` or `Flusher` interfaces.

## type `HandlerFunc`

```
type HandlerFunc func(ResponseWriter, *Request)
```

The `HandlerFunc` type is an adapter to allow the use of ordinary functions as HTTP handlers. If `f` is a function with the appropriate signature, `HandlerFunc(f)` is a Handler that calls `f`.

## func (`HandlerFunc`) `ServeHTTP`

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
```

`ServeHTTP` calls `f(w, r)`.

## type `Header`

```
type Header map[string][]string
```

A `Header` represents the key-value pairs in an HTTP header.

The keys should be in canonical form, as returned by `CanonicalHeaderKey`.

## func (`Header`) `Add`

```
func (h Header) Add(key, value string)
```

`Add` adds the key, value pair to the header. It appends to any existing values associated with key. The key is case insensitive; it is canonicalized by `CanonicalHeaderKey`.

## func (`Header`) `Clone`

```
func (h Header) Clone() Header
```

Clone returns a copy of h or nil if h is nil.

## func (Header) Del

```
func (h Header) Del(key string)
```

Del deletes the values associated with key. The key is case insensitive; it is canonicalized by CanonicalHeaderKey.

## func (Header) Get

```
func (h Header) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns "". It is case insensitive; textproto.CanonicalMIMEHeaderKey is used to canonicalize the provided key. To use non-canonical keys, access the map directly.

## func (Header) Set

```
func (h Header) Set(key, value string)
```

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key. The key is case insensitive; it is canonicalized by textproto.CanonicalMIMEHeaderKey. To use non-canonical keys, assign to the map directly.

## func (Header) Values

```
func (h Header) Values(key string) []string
```

Values returns all values associated with the given key. It is case insensitive; textproto.CanonicalMIMEHeaderKey is used to canonicalize the provided key. To use non-canonical keys, access the map directly. The returned slice is not a copy.

## func (Header) Write

```
func (h Header) Write(w io.Writer) error
```

Write writes a header in wire format.

## func (Header) WriteSubset

```
func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) error
```

WriteSubset writes a header in wire format. If exclude is not nil, keys where exclude[key] == true are not written. Keys are not canonicalized before checking the exclude map.

## type Hijacker

```
type Hijacker interface {
 // Hijack lets the caller take over the connection.
 // After a call to Hijack the HTTP server library
 // will not do anything else with the connection.
 //
 // It becomes the caller's responsibility to manage
 // and close the connection.
 //
 // The returned net.Conn may have read or write deadlines
 // already set, depending on the configuration of the
 // Server. It is the caller's responsibility to set
 // or clear those deadlines as needed.
 //
 // The returned bufio.Reader may contain unprocessed buffered
 // data from the client.
 //
 // After a call to Hijack, the original Request.Body must not
 // be used. The original Request's Context remains valid and
 // is not canceled until the Request's ServeHTTP method
 // returns.
 Hijack() (net.Conn, *bufio.ReadWriter, error)
}
```

The Hijacker interface is implemented by ResponseWriters that allow an HTTP handler to take over the connection.

The default ResponseWriter for HTTP/1.x connections supports Hijacker, but HTTP/2 connections intentionally do not. ResponseWriter wrappers may also not support Hijacker. Handlers should always test for this ability at runtime.

## type ProtocolError

```
type ProtocolError struct {
 ErrorString string
}
```

ProtocolError represents an HTTP protocol error.

Deprecated: Not all errors in the http package related to protocol errors are of type ProtocolError.

## func (\*ProtocolError) Error

```
func (pe *ProtocolError) Error() string
```

## type PushOptions

```
type PushOptions struct {
 // Method specifies the HTTP method for the promised request.
```

```

// If set, it must be "GET" or "HEAD". Empty means "GET".
Method string

// Header specifies additional promised request headers. This cannot
// include HTTP/2 pseudo header fields like ":path" and ":scheme",
// which will be added automatically.
Header Header

}

```

PushOptions describes options for Pusher.Push.

## type Pusher

```

type Pusher interface {
 // Push initiates an HTTP/2 server push. This constructs a synthetic
 // request using the given target and options, serializes that request
 // into a PUSH_PROMISE frame, then dispatches that request using the
 // server's request handler. If opts is nil, default options are used.
 //
 // The target must either be an absolute path (like "/path") or an absolute
 // URL that contains a valid host and the same scheme as the parent request.
 // If the target is a path, it will inherit the scheme and host of the
 // parent request.
 //
 // The HTTP/2 spec disallows recursive pushes and cross-authority pushes.
 // Push may or may not detect these invalid pushes; however, invalid
 // pushes will be detected and canceled by conforming clients.
 //
 // Handlers that wish to push URL X should call Push before sending any
 // data that may trigger a request for URL X. This avoids a race where the
 // client issues requests for X before receiving the PUSH_PROMISE for X.
 //
 // Push will run in a separate goroutine making the order of arrival
 // non-deterministic. Any required synchronization needs to be implemented
 // by the caller.
 //
 // Push returns ErrNotSupported if the client has disabled push or if push
 // is not supported on the underlying connection.
 Push(target string, opts *PushOptions) error
}

```

Pusher is the interface implemented by ResponseWriters that support HTTP/2 server push. For more background, see <https://tools.ietf.org/html/rfc7540#section-8.2>.

## type Request

```

type Request struct {
 // Method specifies the HTTP method (GET, POST, PUT, etc.).
 // For client requests, an empty string means GET.
 //
 // Go's HTTP client does not support sending a request with

```

```
// the CONNECT method. See the documentation on Transport for
// details.

Method string

// URL specifies either the URI being requested (for server
// requests) or the URL to access (for client requests).
//
// For server requests, the URL is parsed from the URI
// supplied on the Request-Line as stored in RequestURI. For
// most requests, fields other than Path and RawQuery will be
// empty. (See RFC 7230, Section 5.3)
//
// For client requests, the URL's Host specifies the server to
// connect to, while the Request's Host field optionally
// specifies the Host header value to send in the HTTP
// request.

URL *url.URL

// The protocol version for incoming server requests.
//
// For client requests, these fields are ignored. The HTTP
// client code always uses either HTTP/1.1 or HTTP/2.
// See the docs on Transport for details.

Proto string // "HTTP/1.0"
ProtoMajor int // 1
ProtoMinor int // 0

// Header contains the request header fields either received
// by the server or to be sent by the client.
//
// If a server received a request with header lines,
//
// Host: example.com
// accept-encoding: gzip, deflate
// Accept-Language: en-us
// fOO: Bar
// foo: two
//
// then
//
// Header = map[string][]string{
// "Accept-Encoding": {"gzip, deflate"},
// "Accept-Language": {"en-us"},
// "Foo": {"Bar", "two"},
// }

// For incoming requests, the Host header is promoted to the
// Request.Host field and removed from the Header map.
//
// HTTP defines that header names are case-insensitive. The
// request parser implements this by using CanonicalHeaderKey,
// making the first character and any characters following a
// hyphen uppercase and the rest lowercase.

//
```

```
// For client requests, certain headers such as Content-Length
// and Connection are automatically written when needed and
// values in Header may be ignored. See the documentation
// for the Request.Write method.
Header Header

// Body is the request's body.
//
// For client requests, a nil body means the request has no
// body, such as a GET request. The HTTP Client's Transport
// is responsible for calling the Close method.
//
// For server requests, the Request Body is always non-nil
// but will return EOF immediately when no body is present.
// The Server will close the request body. The ServeHTTP
// Handler does not need to.
Body io.ReadCloser

// GetBody defines an optional func to return a new copy of
// Body. It is used for client requests when a redirect requires
// reading the body more than once. Use of GetBody still
// requires setting Body.
//
// For server requests, it is unused.
GetBody func() (io.ReadCloser, error)

// ContentLength records the length of the associated content.
// The value -1 indicates that the length is unknown.
// Values >= 0 indicate that the given number of bytes may
// be read from Body.
//
// For client requests, a value of 0 with a non-nil Body is
// also treated as unknown.
ContentLength int64

// TransferEncoding lists the transfer encodings from outermost to
// innermost. An empty list denotes the "identity" encoding.
// TransferEncoding can usually be ignored; chunked encoding is
// automatically added and removed as necessary when sending and
// receiving requests.
TransferEncoding []string

// Close indicates whether to close the connection after
// replying to this request (for servers) or after sending this
// request and reading its response (for clients).
//
// For server requests, the HTTP server handles this automatically
// and this field is not needed by Handlers.
//
// For client requests, setting this field prevents re-use of
// TCP connections between requests to the same hosts, as if
// Transport.DisableKeepAlives were set.
Close bool
```

```
// For server requests, Host specifies the host on which the
// URL is sought. For HTTP/1 (per RFC 7230, section 5.4), this
// is either the value of the "Host" header or the host name
// given in the URL itself. For HTTP/2, it is the value of the
// ":authority" pseudo-header field.
// It may be of the form "host:port". For international domain
// names, Host may be in Punycode or Unicode form. Use
// golang.org/x/net/idna to convert it to either format if
// needed.
// To prevent DNS rebinding attacks, server Handlers should
// validate that the Host header has a value for which the
// Handler considers itself authoritative. The included
// ServeMux supports patterns registered to particular host
// names and thus protects its registered Handlers.
//
// For client requests, Host optionally overrides the Host
// header to send. If empty, the Request.Write method uses
// the value of URL.Host. Host may contain an international
// domain name.
Host string

// Form contains the parsed form data, including both the URL
// field's query parameters and the PATCH, POST, or PUT form data.
// This field is only available after ParseForm is called.
// The HTTP client ignores Form and uses Body instead.
Form url.Values

// PostForm contains the parsed form data from PATCH, POST
// or PUT body parameters.
//
// This field is only available after ParseForm is called.
// The HTTP client ignores PostForm and uses Body instead.
PostForm url.Values

// MultipartForm is the parsed multipart form, including file uploads.
// This field is only available after ParseMultipartForm is called.
// The HTTP client ignores MultipartForm and uses Body instead.
MultipartForm *multipart.Form

// Trailer specifies additional headers that are sent after the request
// body.
//
// For server requests, the Trailer map initially contains only the
// trailer keys, with nil values. (The client declares which trailers it
// will later send.) While the handler is reading from Body, it must
// not reference Trailer. After reading from Body returns EOF, Trailer
// can be read again and will contain non-nil values, if they were sent
// by the client.
//
// For client requests, Trailer must be initialized to a map containing
// the trailer keys to later send. The values may be nil or their final
// values. The ContentLength must be 0 or -1, to send a chunked request.
// After the HTTP request is sent the map values can be updated while
// the request body is read. Once the body returns EOF, the caller must
```

```

// not mutate Trailer.
//
// Few HTTP clients, servers, or proxies support HTTP trailers.
Trailer Header

// RemoteAddr allows HTTP servers and other software to record
// the network address that sent the request, usually for
// logging. This field is not filled in by ReadRequest and
// has no defined format. The HTTP server in this package
// sets RemoteAddr to an "IP:port" address before invoking a
// handler.
// This field is ignored by the HTTP client.
RemoteAddr string

// RequestURI is the unmodified request-target of the
// Request-Line (RFC 7230, Section 3.1.1) as sent by the client
// to a server. Usually the URL field should be used instead.
// It is an error to set this field in an HTTP client request.
RequestURI string

// TLS allows HTTP servers and other software to record
// information about the TLS connection on which the request
// was received. This field is not filled in by ReadRequest.
// The HTTP server in this package sets the field for
// TLS-enabled connections before invoking a handler;
// otherwise it leaves the field nil.
// This field is ignored by the HTTP client.
TLS *tls.ConnectionState

// Cancel is an optional channel whose closure indicates that the client
// request should be regarded as canceled. Not all implementations of
// RoundTripper may support Cancel.
//
// For server requests, this field is not applicable.
//
// Deprecated: Set the Request's context with NewRequestWithContext
// instead. If a Request's Cancel field and context are both
// set, it is undefined whether Cancel is respected.
Cancel <-chan struct{}
```

// Response is the redirect response which caused this request  
 // to be created. This field is only populated during client  
 // redirects.

Response \*Response  
 // contains filtered or unexported fields

}

A Request represents an HTTP request received by a server or to be sent by a client.

The field semantics differ slightly between client and server usage. In addition to the notes on the fields below, see the documentation for Request.Write and RoundTripper.

## func `NewRequest`

```
func NewRequest(method, url string, body io.Reader) (*Request, error)
```

`NewRequest` wraps `NewRequestWithContext` using the background context.

## func `NewRequestWithContext`

```
func NewRequestWithContext(ctx context.Context, method, url string, body io.Reader) (
```

`NewRequestWithContext` returns a new `Request` given a method, URL, and optional body.

If the provided body is also an `io.Closer`, the returned `Request.Body` is set to body and will be closed by the Client methods `Do`, `Post`, and `PostForm`, and `Transport.RoundTrip`.

`NewRequestWithContext` returns a `Request` suitable for use with `Client.Do` or `Transport.RoundTrip`. To create a request for use with testing a Server Handler, either use the `NewRequest` function in the `net/http/httptest` package, use `ReadRequest`, or manually update the `Request` fields. For an outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body. See the `Request` type's documentation for the difference between inbound and outbound request fields.

If body is of type `*bytes.Buffer`, `*bytes.Reader`, or `*strings.Reader`, the returned request's `ContentLength` is set to its exact value (instead of -1), `GetBody` is populated (so 307 and 308 redirects can replay the body), and `Body` is set to `NoBody` if the `ContentLength` is 0.

## func `ReadRequest`

```
func ReadRequest(b *bufio.Reader) (*Request, error)
```

`ReadRequest` reads and parses an incoming request from b.

`ReadRequest` is a low-level function and should only be used for specialized applications; most code should use the `Server` to read requests and handle them via the `Handler` interface. `ReadRequest` only supports HTTP/1.x requests. For HTTP/2, use [golang.org/x/net/http2](https://golang.org/x/net/http2).

## func `(*Request) AddCookie`

```
func (r *Request) AddCookie(c *Cookie)
```

`AddCookie` adds a cookie to the request. Per [RFC 6265 section 5.4](#), `AddCookie` does not attach more than one `Cookie` header field. That means all cookies, if any, are written into the same line, separated by semicolon. `AddCookie` only sanitizes c's name and value, and does not sanitize a `Cookie` header already present in the request.

## func `(*Request) BasicAuth`

```
func (r *Request) BasicAuth() (username, password string, ok bool)
```

BasicAuth returns the username and password provided in the request's Authorization header, if the request uses HTTP Basic Authentication. See [RFC 2617, Section 2](#).

## func (\*Request) Clone

```
func (r *Request) Clone(ctx context.Context) *Request
```

Clone returns a deep copy of r with its context changed to ctx. The provided ctx must be non-nil.

For an outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body.

## func (\*Request) Context

```
func (r *Request) Context() context.Context
```

Context returns the request's context. To change the context, use WithContext.

The returned context is always non-nil; it defaults to the background context.

For outgoing client requests, the context controls cancellation.

For incoming server requests, the context is canceled when the client's connection closes, the request is canceled (with HTTP/2), or when the ServeHTTP method returns.

## func (\*Request) Cookie

```
func (r *Request) Cookie(name string) (*Cookie, error)
```

Cookie returns the named cookie provided in the request or ErrNoCookie if not found. If multiple cookies match the given name, only one cookie will be returned.

## func (\*Request) Cookies

```
func (r *Request) Cookies() []*Cookie
```

Cookies parses and returns the HTTP cookies sent with the request.

## func (\*Request) FormFile

```
func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

FormFile returns the first file for the provided form key. FormFile calls ParseMultipartForm and ParseForm if necessary.

## func (\*Request) FormValue

```
func (r *Request) FormValue(key string) string
```

FormValue returns the first value for the named component of the query. POST and PUT body parameters take precedence over URL query string values. FormValue calls ParseMultipartForm and ParseForm if necessary and ignores any errors returned by these functions. If key is not present, FormValue returns the empty string. To access multiple values of the same key, call ParseForm and then inspect Request.Form directly.

## func (\*Request) MultipartReader

```
func (r *Request) MultipartReader() (*multipart.Reader, error)
```

MultipartReader returns a MIME multipart reader if this is a multipart/form-data or a multipart/mixed POST request, else returns nil and an error. Use this function instead of ParseMultipartForm to process the request body as a stream.

## func (\*Request) ParseForm

```
func (r *Request) ParseForm() error
```

ParseForm populates r.Form and r.PostForm.

For all requests, ParseForm parses the raw query from the URL and updates r.Form.

For POST, PUT, and PATCH requests, it also reads the request body, parses it as a form and puts the results into both r.PostForm and r.Form. Request body parameters take precedence over URL query string values in r.Form.

If the request Body's size has not already been limited by MaxBytesReader, the size is capped at 10MB.

For other HTTP methods, or when the Content-Type is not application/x-www-form-urlencoded, the request Body is not read, and r.PostForm is initialized to a non-nil, empty value.

ParseMultipartForm calls ParseForm automatically. ParseForm is idempotent.

## func (\*Request) ParseMultipartForm

```
func (r *Request) ParseMultipartForm(maxMemory int64) error
```

ParseMultipartForm parses a request body as multipart/form-data. The whole request body is parsed and up to a total of maxMemory bytes of its file parts are stored in memory, with the remainder stored on disk in temporary files. ParseMultipartForm calls ParseForm if necessary. After one call to ParseMultipartForm, subsequent calls have no effect.

## func (\*Request) PostFormValue

```
func (r *Request) PostFormValue(key string) string
```

PostFormValue returns the first value for the named component of the POST, PATCH, or PUT request body. URL query parameters are ignored. PostFormValue calls ParseMultipartForm and ParseForm if necessary and ignores any errors returned by these functions. If key is not present, PostFormValue returns the empty string.

## func (\*Request) ProtoAtLeast

```
func (r *Request) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast reports whether the HTTP protocol used in the request is at least major.minor.

## func (\*Request) Referer

```
func (r *Request) Referer() string
```

Referer returns the referring URL, if sent in the request.

Referer is misspelled as in the request itself, a mistake from the earliest days of HTTP. This value can also be fetched from the Header map as Header["Referer"]; the benefit of making it available as a method is that the compiler can diagnose programs that use the alternate (correct English) spelling req.Referrer() but cannot diagnose programs that use Header["Referrer"].

## func (\*Request) SetBasicAuth

```
func (r *Request) SetBasicAuth(username, password string)
```

SetBasicAuth sets the request's Authorization header to use HTTP Basic Authentication with the provided username and password.

With HTTP Basic Authentication the provided username and password are not encrypted.

Some protocols may impose additional requirements on pre-escaping the username and password. For instance, when used with OAuth2, both arguments must be URL encoded first with url.QueryEscape.

## func (\*Request) UserAgent

```
func (r *Request) UserAgent() string
```

UserAgent returns the client's User-Agent, if sent in the request.

## func (\*Request) WithContext

```
func (r *Request) WithContext(ctx context.Context) *Request
```

WithContext returns a shallow copy of r with its context changed to ctx. The provided ctx must be non-nil.

For outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body.

To create a new request with a context, use NewRequestWithContext. To change the context of a request, such as an incoming request you want to modify before sending back out, use Request.Clone. Between those two uses, it's rare to need WithContext.

## func (\*Request) Write

```
func (r *Request) Write(w io.Writer) error
```

Write writes an HTTP/1.1 request, which is the header and body, in wire format. This method consults the following fields of the request:

```
Host
URL
Method (defaults to "GET")
Header
ContentLength
TransferEncoding
Body
```

If Body is present, Content-Length is  $\leq 0$  and TransferEncoding hasn't been set to "identity", Write adds "Transfer-Encoding: chunked" to the header. Body is closed after it is sent.

## func (\*Request) WriteProxy

```
func (r *Request) WriteProxy(w io.Writer) error
```

WriteProxy is like Write but writes the request in the form expected by an HTTP proxy. In particular, WriteProxy writes the initial Request-URI line of the request with an absolute URI, per section 5.3 of [RFC 7230](#), including the scheme and host. In either case, WriteProxy also writes a Host header, using either r.Host or r.URL.Host.

## type Response

```
type Response struct {
 Status string // e.g. "200 OK"
 StatusCode int // e.g. 200
 Proto string // e.g. "HTTP/1.0"
 ProtoMajor int // e.g. 1
 ProtoMinor int // e.g. 0
```

```
// Header maps header keys to values. If the response had multiple
// headers with the same key, they may be concatenated, with comma
// delimiters. (RFC 7230, section 3.2.2 requires that multiple headers
// be semantically equivalent to a comma-delimited sequence.) When
// Header values are duplicated by other fields in this struct (e.g.,
// ContentLength, TransferEncoding, Trailer), the field values are
// authoritative.
//
// Keys in the map are canonicalized (see CanonicalHeaderKey).
Header Header

// Body represents the response body.
//
// The response body is streamed on demand as the Body field
// is read. If the network connection fails or the server
// terminates the response, Body.Read calls return an error.
//
// The http Client and Transport guarantee that Body is always
// non-nil, even on responses without a body or responses with
// a zero-length body. It is the caller's responsibility to
// close Body. The default HTTP client's Transport may not
// reuse HTTP/1.x "keep-alive" TCP connections if the Body is
// not read to completion and closed.
//
// The Body is automatically dechunked if the server replied
// with a "chunked" Transfer-Encoding.
//
// As of Go 1.12, the Body will also implement io.Writer
// on a successful "101 Switching Protocols" response,
// as used by WebSockets and HTTP/2's "h2c" mode.
Body io.ReadCloser

// ContentLength records the length of the associated content. The
// value -1 indicates that the length is unknown. Unless Request.Method
// is "HEAD", values >= 0 indicate that the given number of bytes may
// be read from Body.
ContentLength int64

// Contains transfer encodings from outer-most to inner-most. Value is
// nil, means that "identity" encoding is used.
TransferEncoding []string

// Close records whether the header directed that the connection be
// closed after reading Body. The value is advice for clients: neither
// ReadResponse nor Response.Write ever closes a connection.
Close bool

// Uncompressed reports whether the response was sent compressed but
// was decompressed by the http package. When true, reading from
// Body yields the uncompressed content instead of the compressed
// content actually set from the server, ContentLength is set to -1,
// and the "Content-Length" and "Content-Encoding" fields are deleted
// from the responseHeader. To get the original response from
```

```

// the server, set Transport.DisableCompression to true.
Uncompressed bool

// Trailer maps trailer keys to values in the same
// format as Header.
//
// The Trailer initially contains only nil values, one for
// each key specified in the server's "Trailer" header
// value. Those values are not added to Header.
//
// Trailer must not be accessed concurrently with Read calls
// on the Body.
//
// After Body.Read has returned io.EOF, Trailer will contain
// any trailer values sent by the server.

Trailer Header

// Request is the request that was sent to obtain this Response.
// Request's Body is nil (having already been consumed).
// This is only populated for Client requests.

Request *Request

// TLS contains information about the TLS connection on which the
// response was received. It is nil for unencrypted responses.
// The pointer is shared between responses and should not be
// modified.

TLS *tls.ConnectionState
}

```

Response represents the response from an HTTP request.

The Client and Transport return Responses from servers once the response headers have been received. The response body is streamed on demand as the Body field is read.

## func Get

```
func Get(url string) (resp *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect, up to a maximum of 10 redirects:

```

301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)

```

An error is returned if there were too many redirects or if there was an HTTP protocol error. A non-2xx response doesn't cause an error. Any returned error will be of type \*url.Error. The url.Error value's Timeout method will report true if request timed out or was canceled.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

Get is a wrapper around DefaultClient.Get.

To make a request with custom headers, use NewRequest and DefaultClient.Do.

## func Head

```
func Head(url string) (resp *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the following redirect codes, Head follows the redirect, up to a maximum of 10 redirects:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)
```

Head is a wrapper around DefaultClient.Head

## func Post

```
func Post(url, contentType string, body io.Reader) (resp *Response, err error)
```

Post issues a POST to the specified URL.

Caller should close resp.Body when done reading from it.

If the provided body is an io.Closer, it is closed after the request.

Post is a wrapper around DefaultClient.Post.

To set custom headers, use NewRequest and DefaultClient.Do.

See the Client.Do method documentation for details on how redirects are handled.

## func PostForm

```
func PostForm(url string, data url.Values) (resp *Response, err error)
```

PostForm issues a POST to the specified URL, with data's keys and values URL-encoded as the request body.

The Content-Type header is set to application/x-www-form-urlencoded. To set other headers, use NewRequest and DefaultClient.Do.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

PostForm is a wrapper around DefaultClient.PostForm.

See the Client.Do method documentation for details on how redirects are handled.

## func `ReadResponse`

```
func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
```

ReadResponse reads and returns an HTTP response from r. The req parameter optionally specifies the Request that corresponds to this Response. If nil, a GET request is assumed. Clients must call resp.Body.Close when finished reading resp.Body. After that call, clients can inspect resp.Trailer to find key/value pairs included in the response trailer.

## func `(*Response) Cookies`

```
func (r *Response) Cookies() []*Cookie
```

Cookies parses and returns the cookies set in the Set-Cookie headers.

## func `(*Response) Location`

```
func (r *Response) Location() (*url.URL, error)
```

Location returns the URL of the response's "Location" header, if present. Relative redirects are resolved relative to the Response's Request. ErrNoLocation is returned if no Location header is present.

## func `(*Response) ProtoAtLeast`

```
func (r *Response) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast reports whether the HTTP protocol used in the response is at least major.minor.

## func `(*Response) Write`

```
func (r *Response) Write(w io.Writer) error
```

Write writes r to w in the HTTP/1.x server response format, including the status line, headers, body, and optional trailer.

This method consults the following fields of the response r:

- StatusCode
- ProtoMajor
- ProtoMinor

```
Request.Method
TransferEncoding
Trailer
Body
ContentLength
Header, values for non-canonical keys will have unpredictable behavior
```

The Response Body is closed after it is sent.

## type ResponseWriter

```
type ResponseWriter interface {
 // Header returns the header map that will be sent by
 // WriteHeader. The Header map also is the mechanism with which
 // Handlers can set HTTP trailers.
 //
 // Changing the header map after a call to WriteHeader (or
 // Write) has no effect unless the modified headers are
 // trailers.
 //
 // There are two ways to set Trailers. The preferred way is to
 // predeclare in the headers which trailers you will later
 // send by setting the "Trailer" header to the names of the
 // trailer keys which will come later. In this case, those
 // keys of the Header map are treated as if they were
 // trailers. See the example. The second way, for trailer
 // keys not known to the Handler until after the first Write,
 // is to prefix the Header map keys with the TrailerPrefix
 // constant value. See TrailerPrefix.
 //
 // To suppress automatic response headers (such as "Date"), set
 // their value to nil.
 Header() Header

 // Write writes the data to the connection as part of an HTTP reply.
 //
 // If WriteHeader has not yet been called, Write calls
 // WriteHeader(http.StatusOK) before writing the data. If the Header
 // does not contain a Content-Type line, Write adds a Content-Type set
 // to the result of passing the initial 512 bytes of written data to
 // DetectContentType. Additionally, if the total size of all written
 // data is under a few KB and there are no Flush calls, the
 // Content-Length header is added automatically.
 //
 // Depending on the HTTP protocol version and the client, calling
 // Write or WriteHeader may prevent future reads on the
 // Request.Body. For HTTP/1.x requests, handlers should read any
 // needed request body data before writing the response. Once the
 // headers have been flushed (due to either an explicit Flusher.Flush
 // call or writing enough data to trigger a flush), the request body
 // may be unavailable. For HTTP/2 requests, the Go HTTP server permits
 // handlers to continue to read the request body while concurrently
 // writing the response. However, such behavior may not be supported
```

```

// by all HTTP/2 clients. Handlers should read before writing if
// possible to maximize compatibility.
Write([]byte) (int, error)

// WriteHeader sends an HTTP response header with the provided
// status code.
//
// If WriteHeader is not called explicitly, the first call to Write
// will trigger an implicit WriteHeader(http.StatusOK).
// Thus explicit calls to WriteHeader are mainly used to
// send error codes.
//
// The provided code must be a valid HTTP 1xx-5xx status code.
// Only one header may be written. Go does not currently
// support sending user-defined 1xx informational headers,
// with the exception of 100-continue response header that the
// Server sends automatically when the Request.Body is read.
WriteHeader(statusCode int)
}

```

A ResponseWriter interface is used by an HTTP handler to construct an HTTP response.

A ResponseWriter may not be used after the Handler.ServeHTTP method has returned.

## type RoundTripper

```

type RoundTripper interface {
 // RoundTrip executes a single HTTP transaction, returning
 // a Response for the provided Request.
 //
 // RoundTrip should not attempt to interpret the response. In
 // particular, RoundTrip must return err == nil if it obtained
 // a response, regardless of the response's HTTP status code.
 // A non-nil err should be reserved for failure to obtain a
 // response. Similarly, RoundTrip should not attempt to
 // handle higher-level protocol details such as redirects,
 // authentication, or cookies.
 //
 // RoundTrip should not modify the request, except for
 // consuming and closing the Request's Body. RoundTrip may
 // read fields of the request in a separate goroutine. Callers
 // should not mutate or reuse the request until the Response's
 // Body has been closed.
 //
 // RoundTrip must always close the body, including on errors,
 // but depending on the implementation may do so in a separate
 // goroutine even after RoundTrip returns. This means that
 // callers wanting to reuse the body for subsequent requests
 // must arrange to wait for the Close call before doing so.
 //
 // The Request's URL and Header fields must be initialized.
 RoundTrip(*Request) (*Response, error)
}

```

RoundTripper is an interface representing the ability to execute a single HTTP transaction, obtaining the Response for a given Request.

A RoundTripper must be safe for concurrent use by multiple goroutines.

```
var DefaultTransport RoundTripper = &Transport{
 Proxy: ProxyFromEnvironment,
 DialContext: (&net.Dialer{
 Timeout: 30 * time.Second,
 KeepAlive: 30 * time.Second,
 DualStack: true,
 }).DialContext,
 ForceAttemptHTTP2: true,
 MaxIdleConns: 100,
 IdleConnTimeout: 90 * time.Second,
 TLSHandshakeTimeout: 10 * time.Second,
 ExpectContinueTimeout: 1 * time.Second,
}
```

DefaultTransport is the default implementation of Transport and is used by DefaultClient. It establishes network connections as needed and caches them for reuse by subsequent calls. It uses HTTP proxies as directed by the \$HTTP\_PROXY and \$NO\_PROXY (or \$http\_proxy and \$no\_proxy) environment variables.

## func NewFileTransport

```
func NewFileTransport(fs FileSystem) RoundTripper
```

NewFileTransport returns a new RoundTripper, serving the provided FileSystem. The returned RoundTripper ignores the URL host in its incoming requests, as well as most other properties of the request.

The typical use case for NewFileTransport is to register the "file" protocol with a Transport, as in:

```
t := &http.Transport{}
t.RegisterProtocol("file", http.NewFileTransport(http.Dir("/")))
c := &http.Client{Transport: t}
res, err := c.Get("file:///etc/passwd")
...
```

## type SameSite

```
type SameSite int
```

SameSite allows a server to define a cookie attribute making it impossible for the browser to send this cookie along with cross-site requests. The main goal is to mitigate the risk of cross-origin information leakage, and provide some protection against cross-site request forgery attacks.

See <https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00> for details.

```
const (
 SameSiteDefaultMode SameSite = iota + 1
 SameSiteLaxMode
 SameSiteStrictMode
 SameSiteNoneMode
)
```

## type ServeMux

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

ServeMux is an HTTP request multiplexer. It matches the URL of each incoming request against a list of registered patterns and calls the handler for the pattern that most closely matches the URL.

Patterns name fixed, rooted paths, like "/favicon.ico", or rooted subtrees, like "/images/" (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "/images/" and "/images/thumbnails/", the latter handler will be called for paths beginning "/images/thumbnails/" and the former will receive requests for any other paths in the "/images/" subtree.

Note that since a pattern ending in a slash names a rooted subtree, the pattern "/" matches all paths not matched by other registered patterns, not just the URL with Path == "/".

If a subtree has been registered and a request is received naming the subtree root without its trailing slash, ServeMux redirects that request to the subtree root (adding the trailing slash). This behavior can be overridden with a separate registration for the path without the trailing slash. For example, registering "/images/" causes ServeMux to redirect a request for "/images" to "/images/", unless "/images" has been registered separately.

Patterns may optionally begin with a host name, restricting matches to URLs on that host only. Host-specific patterns take precedence over general patterns, so that a handler might register for the two patterns "/codesearch" and "codesearch.google.com/" without also taking over requests for "<http://www.google.com/>".

ServeMux also takes care of sanitizing the URL request path and the Host header, stripping the port number and redirecting any request containing . or .. elements or repeated slashes to an equivalent, cleaner URL.

## func NewServeMux

```
func NewServeMux() *ServeMux
```

NewServeMux allocates and returns a new ServeMux.

## func (\*ServeMux) Handle

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern. If a handler already exists for pattern, Handle panics.

## func (\*ServeMux) HandleFunc

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc registers the handler function for the given pattern.

## func (\*ServeMux) Handler

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
```

Handler returns the handler to use for the given request, consulting r.Method, r.Host, and r.URL.Path. It always returns a non-nil handler. If the path is not in its canonical form, the handler will be an internally-generated handler that redirects to the canonical path. If the host contains a port, it is ignored when matching handlers.

The path and host are used unchanged for CONNECT requests.

Handler also returns the registered pattern that matches the request or, in the case of internally-generated redirects, the pattern that will match after following the redirect.

If there is no registered handler that applies to the request, Handler returns a ``page not found'' handler and an empty pattern.

## func (\*ServeMux) ServeHTTP

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP dispatches the request to the handler whose pattern most closely matches the request URL.

## type Server

```
type Server struct {
 // Addr optionally specifies the TCP address for the server to listen on,
 // in the form "host:port". If empty, ":http" (port 80) is used.
 // The service names are defined in RFC 6335 and assigned by IANA.
 // See net.Dial for details of the address format.
 Addr string

 Handler Handler // handler to invoke, http.DefaultServeMux if nil
```

```
// TLSConfig optionally provides a TLS configuration for use
// by ServeTLS and ListenAndServeTLS. Note that this value is
// cloned by ServeTLS and ListenAndServeTLS, so it's not
// possible to modify the configuration with methods like
// tls.Config.SessionTicketKeys. To use
// SessionTicketKeys, use Server.Serve with a TLS Listener
// instead.
TLSConfig *tls.Config

// ReadTimeout is the maximum duration for reading the entire
// request, including the body.
//
// Because ReadTimeout does not let Handlers make per-request
// decisions on each request body's acceptable deadline or
// upload rate, most users will prefer to use
// ReadHeaderTimeout. It is valid to use them both.
ReadTimeout time.Duration

// ReadHeaderTimeout is the amount of time allowed to read
// request headers. The connection's read deadline is reset
// after reading the headers and the Handler can decide what
// is considered too slow for the body. If ReadHeaderTimeout
// is zero, the value of ReadTimeout is used. If both are
// zero, there is no timeout.
ReadHeaderTimeout time.Duration

// WriteTimeout is the maximum duration before timing out
// writes of the response. It is reset whenever a new
// request's header is read. Like ReadTimeout, it does not
// let Handlers make decisions on a per-request basis.
WriteTimeout time.Duration

// IdleTimeout is the maximum amount of time to wait for the
// next request when keep-alives are enabled. If IdleTimeout
// is zero, the value of ReadTimeout is used. If both are
// zero, there is no timeout.
IdleTimeout time.Duration

// MaxHeaderBytes controls the maximum number of bytes the
// server will read parsing the request header's keys and
// values, including the request line. It does not limit the
// size of the request body.
// If zero, DefaultMaxHeaderBytes is used.
MaxHeaderBytes int

// TLSNextProto optionally specifies a function to take over
// ownership of the provided TLS connection when an ALPN
// protocol upgrade has occurred. The map key is the protocol
// name negotiated. The Handler argument should be used to
// handle HTTP requests and will initialize the Request's TLS
// and RemoteAddr if not already set. The connection is
// automatically closed when the function returns.
// If TLSNextProto is not nil, HTTP/2 support is not enabled
// automatically.
```

```

TLSNextProto map[string]func(*Server, *tls.Conn, Handler)

// ConnState specifies an optional callback function that is
// called when a client connection changes state. See the
// ConnState type and associated constants for details.
ConnState func(net.Conn, ConnState)

// ErrorLog specifies an optional logger for errors accepting
// connections, unexpected behavior from handlers, and
// underlying FileSystem errors.
// If nil, logging is done via the log package's standard logger.
ErrorLog *log.Logger

// BaseContext optionally specifies a function that returns
// the base context for incoming requests on this server.
// The provided Listener is the specific Listener that's
// about to start accepting requests.
// If BaseContext is nil, the default is context.Background().
// If non-nil, it must return a non-nil context.
BaseContext func(net.Listener) context.Context

// ConnContext optionally specifies a function that modifies
// the context used for a new connection c. The provided ctx
// is derived from the base context and has a ServerContextKey
// value.
ConnContext func(ctx context.Context, c net.Conn) context.Context
// contains filtered or unexported fields
}

```

A Server defines parameters for running an HTTP server. The zero value for Server is a valid configuration.

## func (\*Server) Close

```
func (srv *Server) Close() error
```

Close immediately closes all active net.Listeners and any connections in state StateNew, StateActive, or StatIdle. For a graceful shutdown, use Shutdown.

Close does not attempt to close (and does not even know about) any hijacked connections, such as WebSockets.

Close returns any error returned from closing the Server's underlying Listener(s).

## func (\*Server) ListenAndServe

```
func (srv *Server) ListenAndServe() error
```

ListenAndServe listens on the TCP network address srv.Addr and then calls Serve to handle requests on incoming connections. Accepted connections are configured to enable TCP keep-alives.

If `srv.Addr` is blank, :http is used.

`ListenAndServe` always returns a non-nil error. After `Shutdown` or `Close`, the returned error is `ErrServerClosed`.

## func (\*Server) ListenAndServeTLS

```
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
```

`ListenAndServeTLS` listens on the TCP network address `srv.Addr` and then calls `ServeTLS` to handle requests on incoming TLS connections. Accepted connections are configured to enable TCP keep-alives.

Filenames containing a certificate and matching private key for the server must be provided if neither the Server's `TLSConfig.Certificates` nor `TLSConfig.GetCertificate` are populated. If the certificate is signed by a certificate authority, the `certFile` should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

If `srv.Addr` is blank, :https is used.

`ListenAndServeTLS` always returns a non-nil error. After `Shutdown` or `Close`, the returned error is `ErrServerClosed`.

## func (\*Server) RegisterOnShutdown

```
func (srv *Server) RegisterOnShutdown(f func())
```

`RegisterOnShutdown` registers a function to call on `Shutdown`. This can be used to gracefully shutdown connections that have undergone ALPN protocol upgrade or that have been hijacked. This function should start protocol-specific graceful shutdown, but should not wait for shutdown to complete.

## func (\*Server) Serve

```
func (srv *Server) Serve(l net.Listener) error
```

`Serve` accepts incoming connections on the Listener `l`, creating a new service goroutine for each. The service goroutines read requests and then call `srv.Handler` to reply to them.

HTTP/2 support is only enabled if the Listener returns `*tls.Conn` connections and they were configured with "h2" in the `TLSConfig.NextProtos`.

`Serve` always returns a non-nil error and closes `l`. After `Shutdown` or `Close`, the returned error is `ErrServerClosed`.

## func (\*Server) ServeTLS

```
func (srv *Server) ServeTLS(l net.Listener, certFile, keyFile string) error
```

ServeTLS accepts incoming connections on the Listener l, creating a new service goroutine for each. The service goroutines perform TLS setup and then read requests, calling srv.Handler to reply to them.

Files containing a certificate and matching private key for the server must be provided if neither the Server's TLSConfig.Certificates nor TLSConfig.GetCertificate are populated. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

ServeTLS always returns a non-nil error. After Shutdown or Close, the returned error is ErrServerClosed.

## func (\*Server) SetKeepAlivesEnabled

```
func (srv *Server) SetKeepAlivesEnabled(v bool)
```

SetKeepAlivesEnabled controls whether HTTP keep-alives are enabled. By default, keep-alives are always enabled. Only very resource-constrained environments or servers in the process of shutting down should disable them.

## func (\*Server) Shutdown

```
func (srv *Server) Shutdown(ctx context.Context) error
```

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down. If the provided context expires before the shutdown is complete, Shutdown returns the context's error, otherwise it returns any error returned from closing the Server's underlying Listener(s).

When Shutdown is called, Serve, ListenAndServe, and ListenAndServeTLS immediately return ErrServerClosed. Make sure the program doesn't exit and waits instead for Shutdown to return.

Shutdown does not attempt to close nor wait for hijacked connections such as WebSockets. The caller of Shutdown should separately notify such long-lived connections of shutdown and wait for them to close, if desired. See RegisterOnShutdown for a way to register shutdown notification functions.

Once Shutdown has been called on a server, it may not be reused; future calls to methods such as Serve will return ErrServerClosed.

## type Transport

```
type Transport struct {
```

```
 // Proxy specifies a function to return a proxy for a given
```

```
// Request. If the function returns a non-nil error, the
// request is aborted with the provided error.
//
// The proxy type is determined by the URL scheme. "http",
// "https", and "socks5" are supported. If the scheme is empty,
// "http" is assumed.
//
// If Proxy is nil or returns a nil *URL, no proxy is used.
Proxy func(*Request) (*url.URL, error)

// DialContext specifies the dial function for creating unencrypted TCP connections.
// If DialContext is nil (and the deprecated Dial below is also nil),
// then the transport dials using package net.
//
// DialContext runs concurrently with calls to RoundTrip.
// A RoundTrip call that initiates a dial may end up using
// a connection dialed previously when the earlier connection
// becomes idle before the later DialContext completes.
DialContext func(ctx context.Context, network, addr string) (net.Conn, error)

// Dial specifies the dial function for creating unencrypted TCP connections.
//
// Dial runs concurrently with calls to RoundTrip.
// A RoundTrip call that initiates a dial may end up using
// a connection dialed previously when the earlier connection
// becomes idle before the later Dial completes.
//
// Deprecated: Use DialContext instead, which allows the transport
// to cancel dials as soon as they are no longer needed.
// If both are set, DialContext takes priority.
Dial func(network, addr string) (net.Conn, error)

// DialTLSContext specifies an optional dial function for creating
// TLS connections for non-proxied HTTPS requests.
//
// If DialTLSContext is nil (and the deprecated DialTLS below is also nil),
// DialContext and TLSClientConfig are used.
//
// If DialTLSContext is set, the Dial and DialContext hooks are not used for HTTP
// requests and the TLSClientConfig and TLSHandshakeTimeout
// are ignored. The returned net.Conn is assumed to already be
// past the TLS handshake.
DialTLSContext func(ctx context.Context, network, addr string) (net.Conn, error)

// DialTLS specifies an optional dial function for creating
// TLS connections for non-proxied HTTPS requests.
//
// Deprecated: Use DialTLSContext instead, which allows the transport
// to cancel dials as soon as they are no longer needed.
// If both are set, DialTLSContext takes priority.
DialTLS func(network, addr string) (net.Conn, error)

// TLSClientConfig specifies the TLS configuration to use with
// tls.Client.
```

```
// If nil, the default configuration is used.
// If non-nil, HTTP/2 support may not be enabled by default.
TLSClientConfig *tls.Config

// TLSHandshakeTimeout specifies the maximum amount of time waiting to
// wait for a TLS handshake. Zero means no timeout.
TLSHandshakeTimeout time.Duration

// DisableKeepAlives, if true, disables HTTP keep-alives and
// will only use the connection to the server for a single
// HTTP request.
//
// This is unrelated to the similarly named TCP keep-alives.
DisableKeepAlives bool

// DisableCompression, if true, prevents the Transport from
// requesting compression with an "Accept-Encoding: gzip"
// request header when the Request contains no existing
// Accept-Encoding value. If the Transport requests gzip on
// its own and gets a gzipped response, it's transparently
// decoded in the Response.Body. However, if the user
// explicitly requested gzip it is not automatically
// uncompressed.
DisableCompression bool

// MaxIdleConns controls the maximum number of idle (keep-alive)
// connections across all hosts. Zero means no limit.
MaxIdleConns int

// MaxIdleConnsPerHost, if non-zero, controls the maximum idle
// (keep-alive) connections to keep per-host. If zero,
// DefaultMaxIdleConnsPerHost is used.
MaxIdleConnsPerHost int

// MaxConnsPerHost optionally limits the total number of
// connections per host, including connections in the dialing,
// active, and idle states. On limit violation, dials will block.
//
// Zero means no limit.
MaxConnsPerHost int

// IdleConnTimeout is the maximum amount of time an idle
// (keep-alive) connection will remain idle before closing
// itself.
// Zero means no limit.
IdleConnTimeout time.Duration

// ResponseHeaderTimeout, if non-zero, specifies the amount of
// time to wait for a server's response headers after fully
// writing the request (including its body, if any). This
// time does not include the time to read the response body.
ResponseHeaderTimeout time.Duration

// ExpectContinueTimeout, if non-zero, specifies the amount of
```

```

// time to wait for a server's first response headers after fully
// writing the request headers if the request has an
// "Expect: 100-continue" header. Zero means no timeout and
// causes the body to be sent immediately, without
// waiting for the server to approve.
// This time does not include the time to send the request header.
ExpectContinueTimeout time.Duration

// TLSNextProto specifies how the Transport switches to an
// alternate protocol (such as HTTP/2) after a TLS ALPN
// protocol negotiation. If Transport dials an TLS connection
// with a non-empty protocol name and TLSNextProto contains a
// map entry for that key (such as "h2"), then the func is
// called with the request's authority (such as "example.com"
// or "example.com:1234") and the TLS connection. The function
// must return a RoundTripper that then handles the request.
// If TLSNextProto is not nil, HTTP/2 support is not enabled
// automatically.
TLSNextProto map[string]func(authority string, c *tls.Conn) RoundTripper

// ProxyConnectHeader optionally specifies headers to send to
// proxies during CONNECT requests.
ProxyConnectHeader Header

// MaxResponseHeaderBytes specifies a limit on how many
// response bytes are allowed in the server's response
// header.
//
// Zero means to use a default limit.
MaxResponseHeaderBytes int64

// WriteBufferSize specifies the size of the write buffer used
// when writing to the transport.
// If zero, a default (currently 4KB) is used.
WriteBufferSize int

// ReadBufferSize specifies the size of the read buffer used
// when reading from the transport.
// If zero, a default (currently 4KB) is used.
ReadBufferSize int

// ForceAttemptHTTP2 controls whether HTTP/2 is enabled when a non-zero
// Dial, DialTLS, or DialContext func or TLSClientConfig is provided.
// By default, use of any those fields conservatively disables HTTP/2.
// To use a custom dialer or TLS config and still attempt HTTP/2
// upgrades, set this to true.
ForceAttemptHTTP2 bool
// contains filtered or unexported fields
}

```

Transport is an implementation of RoundTripper that supports HTTP, HTTPS, and HTTP proxies (for either HTTP or HTTPS with CONNECT).

By default, Transport caches connections for future re-use. This may leave many open connections when accessing many hosts. This behavior can be managed using Transport's `CloseIdleConnections` method and the `MaxIdleConnsPerHost` and `DisableKeepAlives` fields.

Transports should be reused instead of created as needed. Transports are safe for concurrent use by multiple goroutines.

A Transport is a low-level primitive for making HTTP and HTTPS requests. For high-level functionality, such as cookies and redirects, see Client.

Transport uses HTTP/1.1 for HTTP URLs and either HTTP/1.1 or HTTP/2 for HTTPS URLs, depending on whether the server supports HTTP/2, and how the Transport is configured. The `DefaultTransport` supports HTTP/2. To explicitly enable HTTP/2 on a transport, use `golang.org/x/net/http2` and call `ConfigureTransport`. See the package docs for more about HTTP/2.

Responses with status codes in the 1xx range are either handled automatically (100 expect-continue) or ignored. The one exception is HTTP status code 101 (Switching Protocols), which is considered a terminal status and returned by `RoundTrip`. To see the ignored 1xx responses, use the `httptrace` trace package's `ClientTrace.Got1xxResponse`.

Transport only retries a request upon encountering a network error if the request is idempotent and either has no body or has its `Request.GetBody` defined. HTTP requests are considered idempotent if they have HTTP methods GET, HEAD, OPTIONS, or TRACE; or if their Header map contains an "Idempotency-Key" or "X-Idempotency-Key" entry. If the idempotency key value is a zero-length slice, the request is treated as idempotent but the header is not sent on the wire.

## func (\*Transport) `CancelRequest`

```
func (t *Transport) CancelRequest(req *Request)
```

`CancelRequest` cancels an in-flight request by closing its connection. `CancelRequest` should only be called after `RoundTrip` has returned.

Deprecated: Use `Request.WithContext` to create a request with a cancelable context instead. `CancelRequest` cannot cancel HTTP/2 requests.

## func (\*Transport) `Clone`

```
func (t *Transport) Clone() *Transport
```

`Clone` returns a deep copy of t's exported fields.

## func (\*Transport) `CloseIdleConnections`

```
func (t *Transport) CloseIdleConnections()
```

`ClosIdleConnections` closes any connections which were previously connected from previous requests but are now sitting idle in a "keep-alive" state. It does not interrupt any connections currently in use.

## func (\*Transport) RegisterProtocol

```
func (t *Transport) RegisterProtocol(scheme string, rt RoundTripper)
```

`RegisterProtocol` registers a new protocol with `scheme`. The `Transport` will pass requests using the given `scheme` to `rt`. It is `rt`'s responsibility to simulate HTTP request semantics.

`RegisterProtocol` can be used by other packages to provide implementations of protocol schemes like "`ftp`" or "`file`".

If `rt.RoundTrip` returns `ErrSkipAltProtocol`, the `Transport` will handle the `RoundTrip` itself for that one request, as if the protocol were not registered.

## func (\*Transport) RoundTrip

```
func (t *Transport) RoundTrip(req *Request) (*Response, error)
```

`RoundTrip` implements the `RoundTripper` interface.

For higher-level HTTP client support (such as handling of cookies and redirects), see `Get`, `Post`, and the `Client` type.

Like the `RoundTripper` interface, the error types returned by `RoundTrip` are unspecified.

# Package io

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## 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 ( $0 \leq n \leq \text{len}(p)$ ) and any error encountered. Even if Read returns  $n < \text{len}(p)$ , 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 ( $0 \leq n \leq \text{len}(p)$ ) and any error encountered.

When `ReadAt` returns  $n < \text{len}(p)$ , 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  $n < \text{len}(p)$ , 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  $n = \text{len}(p)$  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  $\text{len}(p)$  bytes from  $p$  to the underlying data stream. It returns the number of bytes written from  $p$  ( $0 \leq n \leq \text{len}(p)$ ) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns  $n < \text{len}(p)$ . 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` ( $0 \leq n \leq \text{len}(p)$ ) and any error encountered that caused the write to stop early. `WriteAt` must return a non-nil error if it returns  $n < \text{len}(p)$ .

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.

# Package ioutil

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package ioutil implements some I/O utility functions.

## Variables

```
var Discard io.Writer = devNull(0)
```

Discard is an io.Writer on which all Write calls succeed without doing anything.

## func NopCloser

```
func NopCloser(r io.Reader) io.ReadCloser
```

NopCloser returns a ReadCloser with a no-op Close method wrapping the provided Reader r.

## func ReadAll

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

ReadAll reads from r until an error or EOF and returns the data it read. A successful call returns err == nil, not err == EOF. Because ReadAll is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

## func ReadDir

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

ReadDir reads the directory named by dirname and returns a list of directory entries sorted by filename.

## func ReadFile

```
func ReadFile(filename string) ([]byte, error)
```

ReadFile reads the file named by filename and returns the contents. A successful call returns err == nil, not err == EOF. Because ReadFile reads the whole file, it does not treat an EOF from Read as an error to be reported.

## func TempDir

```
func TempDir(dir, pattern string) (name string, err error)
```

TempDir creates a new temporary directory in the directory dir. The directory name is generated by taking pattern and applying a random string to the end. If pattern includes a "\*", the random string replaces the last "\*". TempDir returns the name of the new directory. If dir is the empty string, TempDir uses the default directory for temporary files (see os.TempDir). Multiple programs calling TempDir simultaneously will not choose the same directory. It is the caller's responsibility to remove the directory when no longer needed.

## func TempFile

```
func TempFile(dir, pattern string) (f *os.File, err error)
```

TempFile creates a new temporary file in the directory dir, opens the file for reading and writing, and returns the resulting \*os.File. The filename is generated by taking pattern and adding a random string to the end. If pattern includes a "\*", the random string replaces the last "\*". If dir is the empty string, TempFile uses the default directory for temporary files (see os.TempDir). Multiple programs calling TempFile simultaneously will not choose the same file. The caller can use f.Name() to find the pathname of the file. It is the caller's responsibility to remove the file when no longer needed.

## func WriteFile

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

WriteFile writes data to a file named by filename. If the file does not exist, WriteFile creates it with permissions perm (before umask); otherwise WriteFile truncates it before writing, without changing permissions.

# Package json

 go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package json implements encoding and decoding of JSON as defined in [RFC 7159](#). The mapping between JSON and Go values is described in the documentation for the Marshal and Unmarshal functions.

See "JSON and Go" for an introduction to this package:

[https://golang.org/doc/articles/json\\_and\\_go.html](https://golang.org/doc/articles/json_and_go.html)

## func `Compact`

```
func Compact(dst *bytes.Buffer, src []byte) error
```

Compact appends to dst the JSON-encoded src with insignificant space characters elided.

## func `HTMLEscape`

```
func HTMLEscape(dst *bytes.Buffer, src []byte)
```

HTMLEscape appends to dst the JSON-encoded src with <, >, &, U+2028 and U+2029 characters inside string literals changed to \u003c, \u003e, \u0026, \u2028, \u2029 so that the JSON will be safe to embed inside HTML <script> tags. For historical reasons, web browsers don't honor standard HTML escaping within <script> tags, so an alternative JSON encoding must be used.

## func `Indent`

```
func Indent(dst *bytes.Buffer, src []byte, prefix, indent string) error
```

Indent appends to dst an indented form of the JSON-encoded src. Each element in a JSON object or array begins on a new, indented line beginning with prefix followed by one or more copies of indent according to the indentation nesting. The data appended to dst does not begin with the prefix nor any indentation, to make it easier to embed inside other formatted JSON data. Although leading space characters (space, tab, carriage return, newline) at the beginning of src are dropped, trailing space characters at the end of src are preserved and copied to dst. For example, if src has no trailing spaces, neither will dst; if src ends in a trailing newline, so will dst.

## func `Marshal`

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal returns the JSON encoding of v.

Marshal traverses the value v recursively. If an encountered value implements the Marshaler interface and is not a nil pointer, Marshal calls its MarshalJSON method to produce JSON. If no MarshalJSON method is present but the value implements encoding.TextMarshaler instead, Marshal calls its MarshalText method and encodes the result as a JSON string. The nil pointer exception is not strictly necessary but mimics a similar, necessary exception in the behavior of UnmarshalJSON.

Otherwise, Marshal uses the following type-dependent default encodings:

Boolean values encode as JSON booleans.

Floating point, integer, and Number values encode as JSON numbers.

String values encode as JSON strings coerced to valid UTF-8, replacing invalid bytes with the Unicode replacement rune. So that the JSON will be safe to embed inside HTML <script> tags, the string is encoded using HTMLEscape, which replaces "<", ">", "&", U+2028, and U+2029 are escaped to "\u003c", "\u003e", "\u0026", "\u2028", and "\u2029". This replacement can be disabled when using an Encoder, by calling SetEscapeHTML(false).

Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string, and a nil slice encodes as the null JSON value.

Struct values encode as JSON objects. Each exported struct field becomes a member of the object, using the field name as the object key, unless the field is omitted for one of the reasons given below.

The encoding of each struct field can be customized by the format string stored under the "json" key in the struct field's tag. The format string gives the name of the field, possibly followed by a comma-separated list of options. The name may be empty in order to specify options without overriding the default field name.

The "omitempty" option specifies that the field should be omitted from the encoding if the field has an empty value, defined as false, 0, a nil pointer, a nil interface value, and any empty array, slice, map, or string.

As a special case, if the field tag is "-", the field is always omitted. Note that a field with name "-" can still be generated using the tag "-".

Examples of struct field tags and their meanings:

```
// Field appears in JSON as key "myName".
Field int `json:"myName"`

// Field appears in JSON as key "myName" and
// the field is omitted from the object if its value is empty,
// as defined above.
Field int `json:"myName,omitempty"
```

```
// Field appears in JSON as key "Field" (the default), but
// the field is skipped if empty.
// Note the leading comma.
Field int `json:",omitempty"`

// Field is ignored by this package.
Field int `json:"-"`

// Field appears in JSON as key "-".
Field int `json:"-,-`
```

The "string" option signals that a field is stored as JSON inside a JSON-encoded string. It applies only to fields of string, floating point, integer, or boolean types. This extra level of encoding is sometimes used when communicating with JavaScript programs:

```
Int64String int64 `json:",string"`
```

The key name will be used if it's a non-empty string consisting of only Unicode letters, digits, and ASCII punctuation except quotation marks, backslash, and comma.

Anonymous struct fields are usually marshaled as if their inner exported fields were fields in the outer struct, subject to the usual Go visibility rules amended as described in the next paragraph. An anonymous struct field with a name given in its JSON tag is treated as having that name, rather than being anonymous. An anonymous struct field of interface type is treated the same as having that type as its name, rather than being anonymous.

The Go visibility rules for struct fields are amended for JSON when deciding which field to marshal or unmarshal. If there are multiple fields at the same level, and that level is the least nested (and would therefore be the nesting level selected by the usual Go rules), the following extra rules apply:

- 1) Of those fields, if any are JSON-tagged, only tagged fields are considered, even if there are multiple untagged fields that would otherwise conflict.
- 2) If there is exactly one field (tagged or not according to the first rule), that is selected.
- 3) Otherwise there are multiple fields, and all are ignored; no error occurs.

Handling of anonymous struct fields is new in Go 1.1. Prior to Go 1.1, anonymous struct fields were ignored. To force ignoring of an anonymous struct field in both current and earlier versions, give the field a JSON tag of "-".

Map values encode as JSON objects. The map's key type must either be a string, an integer type, or implement encoding.TextMarshaler. The map keys are sorted and used as JSON object keys by applying the following rules, subject to the UTF-8 coercion described for string values above:

- keys of any string type are used directly
- encoding.TextMarshalers are marshaled
- integer keys are converted to strings

Pointer values encode as the value pointed to. A nil pointer encodes as the null JSON value.

Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON value.

Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value causes Marshal to return an `UnsupportedTypeError`.

JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an error.

## func `MarshalIndent`

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

`MarshalIndent` is like `Marshal` but applies `Indent` to format the output. Each JSON element in the output will begin on a new line beginning with `prefix` followed by one or more copies of `indent` according to the indentation nesting.

## func `Unmarshal`

```
func Unmarshal(data []byte, v interface{}) error
```

`Unmarshal` parses the JSON-encoded data and stores the result in the value pointed to by `v`. If `v` is nil or not a pointer, `Unmarshal` returns an `InvalidUnmarshalError`.

`Unmarshal` uses the inverse of the encodings that `Marshal` uses, allocating maps, slices, and pointers as necessary, with the following additional rules:

To unmarshal JSON into a pointer, `Unmarshal` first handles the case of the JSON being the JSON literal null. In that case, `Unmarshal` sets the pointer to nil. Otherwise, `Unmarshal` unmarshals the JSON into the value pointed at by the pointer. If the pointer is nil, `Unmarshal` allocates a new value for it to point to.

To unmarshal JSON into a value implementing the `Unmarshaler` interface, `Unmarshal` calls that value's `UnmarshalJSON` method, including when the input is a JSON null. Otherwise, if the value implements `encoding.TextUnmarshaler` and the input is a JSON quoted string, `Unmarshal` calls that value's `UnmarshalText` method with the unquoted form of the string.

To unmarshal JSON into a struct, `Unmarshal` matches incoming object keys to the keys used by `Marshal` (either the struct field name or its tag), preferring an exact match but also accepting a case-insensitive match. By default, object keys which don't have a corresponding struct field are ignored (see `Decoder.DisallowUnknownFields` for an alternative).

To unmarshal JSON into an interface value, `Unmarshal` stores one of these in the interface value:

```
bool, for JSON booleans
float64, for JSON numbers
```

```
string, for JSON strings
[]interface{}, for JSON arrays
map[string]interface{}, for JSON objects
nil for JSON null
```

To unmarshal a JSON array into a slice, Unmarshal resets the slice length to zero and then appends each element to the slice. As a special case, to unmarshal an empty JSON array into a slice, Unmarshal replaces the slice with a new empty slice.

To unmarshal a JSON array into a Go array, Unmarshal decodes JSON array elements into corresponding Go array elements. If the Go array is smaller than the JSON array, the additional JSON array elements are discarded. If the JSON array is smaller than the Go array, the additional Go array elements are set to zero values.

To unmarshal a JSON object into a map, Unmarshal first establishes a map to use. If the map is nil, Unmarshal allocates a new map. Otherwise Unmarshal reuses the existing map, keeping existing entries. Unmarshal then stores key-value pairs from the JSON object into the map. The map's key type must either be any string type, an integer, implement `json.Unmarshaler`, or implement `encoding.TextUnmarshaler`.

If a JSON value is not appropriate for a given target type, or if a JSON number overflows the target type, Unmarshal skips that field and completes the unmarshaling as best it can. If no more serious errors are encountered, Unmarshal returns an `UnmarshalTypeError` describing the earliest such error. In any case, it's not guaranteed that all the remaining fields following the problematic one will be unmarshaled into the target object.

The JSON null value unmarshals into an interface, map, pointer, or slice by setting that Go value to nil. Because null is often used in JSON to mean ``not present," unmarshaling a JSON null into any other Go type has no effect on the value and produces no error.

When unmarshaling quoted strings, invalid UTF-8 or invalid UTF-16 surrogate pairs are not treated as an error. Instead, they are replaced by the Unicode replacement character U+FFFD.

## func `Valid`

```
func Valid(data []byte) bool
```

`Valid` reports whether data is a valid JSON encoding.

## type `Decoder`

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

A `Decoder` reads and decodes JSON values from an input stream.

## func `NewDecoder`

```
func NewDecoder(r io.Reader) *Decoder
```

`NewDecoder` returns a new decoder that reads from `r`.

The decoder introduces its own buffering and may read data from `r` beyond the JSON values requested.

## func `(*Decoder) Buffered`

```
func (dec *Decoder) Buffered() io.Reader
```

`Buffered` returns a reader of the data remaining in the Decoder's buffer. The reader is valid until the next call to `Decode`.

## func `(*Decoder) Decode`

```
func (dec *Decoder) Decode(v interface{}) error
```

`Decode` reads the next JSON-encoded value from its input and stores it in the value pointed to by `v`.

See the documentation for `Unmarshal` for details about the conversion of JSON into a Go value.

## func `(*Decoder) DisallowUnknownFields`

```
func (dec *Decoder) DisallowUnknownFields()
```

`DisallowUnknownFields` causes the Decoder to return an error when the destination is a struct and the input contains object keys which do not match any non-ignored, exported fields in the destination.

## func `(*Decoder) InputOffset`

```
func (dec *Decoder) InputOffset() int64
```

`InputOffset` returns the input stream byte offset of the current decoder position. The offset gives the location of the end of the most recently returned token and the beginning of the next token.

## func `(*Decoder) More`

```
func (dec *Decoder) More() bool
```

`More` reports whether there is another element in the current array or object being parsed.

## func `(*Decoder) Token`

```
func (dec *Decoder) Token() (Token, error)
```

Token returns the next JSON token in the input stream. At the end of the input stream, Token returns nil, io.EOF.

Token guarantees that the delimiters [] {} it returns are properly nested and matched: if Token encounters an unexpected delimiter in the input, it will return an error.

The input stream consists of basic JSON values—bool, string, number, and null—along with delimiters [] {} of type Delim to mark the start and end of arrays and objects. Commas and colons are elided.

## func (\*Decoder) UseNumber

```
func (dec *Decoder) UseNumber()
```

UseNumber causes the Decoder to unmarshal a number into an interface{} as a Number instead of as a float64.

## type Delim

```
type Delim rune
```

A Delim is a JSON array or object delimiter, one of [] { or }.

## func (Delim) String

```
func (d Delim) String() string
```

## type Encoder

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

An Encoder writes JSON values to an output stream.

## func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that writes to w.

## func (\*Encoder) Encode

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode writes the JSON encoding of v to the stream, followed by a newline character.

See the documentation for Marshal for details about the conversion of Go values to JSON.

## func (\*Encoder) SetEscapeHTML

```
func (enc *Encoder) SetEscapeHTML(on bool)
```

SetEscapeHTML specifies whether problematic HTML characters should be escaped inside JSON quoted strings. The default behavior is to escape &, <, and > to \u0026, \u003c, and \u003e to avoid certain safety problems that can arise when embedding JSON in HTML.

In non-HTML settings where the escaping interferes with the readability of the output, SetEscapeHTML(false) disables this behavior.

## func (\*Encoder) SetIndent

```
func (enc *Encoder) SetIndent(prefix, indent string)
```

SetIndent instructs the encoder to format each subsequent encoded value as if indented by the package-level function Indent(dst, src, prefix, indent). Calling SetIndent("", "") disables indentation.

## type InvalidUTF8Error

```
type InvalidUTF8Error struct {
 S string // the whole string value that caused the error
}
```

Before Go 1.2, an InvalidUTF8Error was returned by Marshal when attempting to encode a string value with invalid UTF-8 sequences. As of Go 1.2, Marshal instead coerces the string to valid UTF-8 by replacing invalid bytes with the Unicode replacement rune U+FFFD.

Deprecated: No longer used; kept for compatibility.

## func (\*InvalidUTF8Error) Error

```
func (e *InvalidUTF8Error) Error() string
```

## type InvalidUnmarshalError

```
type InvalidUnmarshalError struct {
 Type reflect.Type
}
```

An InvalidUnmarshalError describes an invalid argument passed to Unmarshal. (The argument to Unmarshal must be a non-nil pointer.)

## func (\*InvalidUnmarshalError) Error

```
func (e *InvalidUnmarshalError) Error() string
```

## type Marshaler

```
type Marshaler interface {
 MarshalJSON() ([]byte, error)
}
```

Marshaler is the interface implemented by types that can marshal themselves into valid JSON.

## type MarshalerError

```
type MarshalerError struct {
 Type reflect.Type
 Err error
 // contains filtered or unexported fields
}
```

A MarshalerError represents an error from calling a MarshalJSON or MarshalText method.

## func (\*MarshalerError) Error

```
func (e *MarshalerError) Error() string
```

## func (\*MarshalerError) Unwrap

```
func (e *MarshalerError) Unwrap() error
```

Unwrap returns the underlying error.

## type Number

```
type Number string
```

A Number represents a JSON number literal.

## func (Number) Float64

```
func (n Number) Float64() (float64, error)
```

Float64 returns the number as a float64.

## func (Number) Int64

```
func (n Number) Int64() (int64, error)
```

Int64 returns the number as an int64.

## func (Number) String

```
func (n Number) String() string
```

String returns the literal text of the number.

## type RawMessage

```
type RawMessage []byte
```

RawMessage is a raw encoded JSON value. It implements Marshaler and Unmarshaler and can be used to delay JSON decoding or precompute a JSON encoding.

## func (RawMessage) MarshalJSON

```
func (m RawMessage) MarshalJSON() ([]byte, error)
```

MarshalJSON returns m as the JSON encoding of m.

## func (\*RawMessage) UnmarshalJSON

```
func (m *RawMessage) UnmarshalJSON(data []byte) error
```

UnmarshalJSON sets \*m to a copy of data.

## type SyntaxError

```
type SyntaxError struct {
 Offset int64 // error occurred after reading Offset bytes
 // contains filtered or unexported fields
}
```

A SyntaxError is a description of a JSON syntax error.

## func (\*SyntaxError) Error

```
func (e *SyntaxError) Error() string
```

## type Token

```
type Token interface{}
```

A Token holds a value of one of these types:

```
Delim, for the four JSON delimiters [] { }
bool, for JSON booleans
float64, for JSON numbers
Number, for JSON numbers
string, for JSON string literals
nil, for JSON null
```

## type `UnmarshalFieldError`

```
type UnmarshalFieldError struct {
 Key string
 Type reflect.Type
 Field reflect.StructField
}
```

An `UnmarshalFieldError` describes a JSON object key that led to an unexported (and therefore unwritable) struct field.

Deprecated: No longer used; kept for compatibility.

## func `(*UnmarshalFieldError) Error`

```
func (e *UnmarshalFieldError) Error() string
```

## type `UnmarshalTypeError`

```
type UnmarshalTypeError struct {
 Value string // description of JSON value - "bool", "array", "number -5"
 Type reflect.Type // type of Go value it could not be assigned to
 Offset int64 // error occurred after reading Offset bytes
 Struct string // name of the struct type containing the field
 Field string // the full path from root node to the field
}
```

An `UnmarshalTypeError` describes a JSON value that was not appropriate for a value of a specific Go type.

## func `(*UnmarshalTypeError) Error`

```
func (e *UnmarshalTypeError) Error() string
```

## type `Unmarshaler`

```
type Unmarshaler interface {
 UnmarshalJSON([]byte) error
```

```
}
```

Unmarshaler is the interface implemented by types that can unmarshal a JSON description of themselves. The input can be assumed to be a valid encoding of a JSON value. UnmarshalJSON must copy the JSON data if it wishes to retain the data after returning.

By convention, to approximate the behavior of Marshal itself, Unmarshalers implement UnmarshalJSON([]byte("null")) as a no-op.

## type **UnsupportedTypeError**

```
type UnsupportedTypeError struct {
 Type reflect.Type
}
```

An UnsupportedTypeError is returned by Marshal when attempting to encode an unsupported value type.

## func (\*UnsupportedTypeError) **Error**

```
func (e *UnsupportedTypeError) Error() string
```

## type **UnsupportedValueError**

```
type UnsupportedValueError struct {
 Value reflect.Value
 Str string
}
```

## func (\*UnsupportedValueError) **Error**

```
func (e *UnsupportedValueError) Error() string
```

# Package log

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package log implements a simple logging package. It defines a type, `Logger`, with methods for formatting output. It also has a predefined 'standard' `Logger` accessible through helper functions `Print[f|ln]`, `Fatal[f|ln]`, and `Panic[f|ln]`, which are easier to use than creating a `Logger` manually. That logger writes to standard error and prints the date and time of each logged message. Every log message is output on a separate line: if the message being printed does not end in a newline, the logger will add one. The `Fatal` functions call `os.Exit(1)` after writing the log message. The `Panic` functions call `panic` after writing the log message.

## Constants

```
const (
 Ldate = 1 << iota // the date in the local time zone: 2009/01/23
 Ltime // the time in the local time zone: 01:23:23
 Lmicroseconds // microsecond resolution: 01:23:23.123123. assume
 Llongfile // full file name and line number: /a/b/c/d.go:23
 Lshortfile // final file name element and line number: d.go:23
 LUTC // if Ldate or Ltime is set, use UTC rather than the
 Lmsgprefix // move the "prefix" from the beginning of the line
 LstdFlags = Ldate | Ltime // initial values for the standard logger
)
```

These flags define which text to prefix to each log entry generated by the `Logger`. Bits are or'ed together to control what's printed. With the exception of the `Lmsgprefix` flag, there is no control over the order they appear (the order listed here) or the format they present (as described in the comments). The prefix is followed by a colon only when `Llongfile` or `Lshortfile` is specified. For example, flags `Ldate | Ltime` (or `LstdFlags`) produce,

```
2009/01/23 01:23:23 message
```

while flags `Ldate | Ltime | Lmicroseconds | Llongfile` produce,

```
2009/01/23 01:23:23.123123 /a/b/c/d.go:23: message
```

## func Fatal

```
func Fatal(v ...interface{})
```

Fatal is equivalent to Print() followed by a call to os.Exit(1).

## func **Fatalf**

```
func Fatalf(format string, v ...interface{})
```

Fatalf is equivalent to Printf() followed by a call to os.Exit(1).

## func **Fatalln**

```
func Fatalln(v ...interface{})
```

Fatalln is equivalent to Println() followed by a call to os.Exit(1).

## func **Flags**

```
func Flags() int
```

Flags returns the output flags for the standard logger. The flag bits are Ldate, Ltime, and so on.

## func **Output**

```
func Output(calldepth int, s string) error
```

Output writes the output for a logging event. The string s contains the text to print after the prefix specified by the flags of the Logger. A newline is appended if the last character of s is not already a newline. Calldepth is the count of the number of frames to skip when computing the file name and line number if Llongfile or Lshortfile is set; a value of 1 will print the details for the caller of Output.

## func **Panic**

```
func Panic(v ...interface{})
```

Panic is equivalent to Print() followed by a call to panic().

## func **Panicf**

```
func Panicf(format string, v ...interface{})
```

Panicf is equivalent to Printf() followed by a call to panic().

## func **Panicln**

```
func Panicln(v ...interface{})
```

Panicln is equivalent to `Println()` followed by a call to `panic()`.

## func `Prefix`

```
func Prefix() string
```

`Prefix` returns the output prefix for the standard logger.

## func `Print`

```
func Print(v ...interface{})
```

`Print` calls `Output` to print to the standard logger. Arguments are handled in the manner of `fmt.Print`.

## func `Printf`

```
func Printf(format string, v ...interface{})
```

`Printf` calls `Output` to print to the standard logger. Arguments are handled in the manner of `fmt.Sprintf`.

## func `Println`

```
func Println(v ...interface{})
```

`Println` calls `Output` to print to the standard logger. Arguments are handled in the manner of `fmt.Println`.

## func `SetFlags`

```
func SetFlags(flag int)
```

`SetFlags` sets the output flags for the standard logger. The flag bits are `Ldate`, `Ltime`, and so on.

## func `SetOutput`

```
func SetOutput(w io.Writer)
```

`SetOutput` sets the output destination for the standard logger.

## func `SetPrefix`

```
func SetPrefix(prefix string)
```

`SetPrefix` sets the output prefix for the standard logger.

## func `Writer`

```
func Writer() io.Writer
```

`Writer` returns the output destination for the standard logger.

## type `Logger`

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

A `Logger` represents an active logging object that generates lines of output to an `io.Writer`. Each logging operation makes a single call to the `Writer`'s `Write` method. A `Logger` can be used simultaneously from multiple goroutines; it guarantees to serialize access to the `Writer`.

## func `New`

```
func New(out io.Writer, prefix string, flag int) *Logger
```

`New` creates a new `Logger`. The `out` variable sets the destination to which log data will be written. The `prefix` appears at the beginning of each generated log line, or after the log header if the `Lmsgprefix` flag is provided. The `flag` argument defines the logging properties.

## func `(*Logger)` `Fatal`

```
func (l *Logger) Fatal(v ...interface{})
```

`Fatal` is equivalent to `l.Print()` followed by a call to `os.Exit(1)`.

## func `(*Logger)` `Fatalf`

```
func (l *Logger) Fatalf(format string, v ...interface{})
```

`Fatalf` is equivalent to `l.Printf()` followed by a call to `os.Exit(1)`.

## func `(*Logger)` `Fatalln`

```
func (l *Logger) Fatalln(v ...interface{})
```

`Fatalln` is equivalent to `l.Println()` followed by a call to `os.Exit(1)`.

## func `(*Logger)` `Flags`

```
func (l *Logger) Flags() int
```

Flags returns the output flags for the logger. The flag bits are Ldate, Ltime, and so on.

## func (\*Logger) Output

```
func (l *Logger) Output(calldepth int, s string) error
```

Output writes the output for a logging event. The string s contains the text to print after the prefix specified by the flags of the Logger. A newline is appended if the last character of s is not already a newline. Calldepth is used to recover the PC and is provided for generality, although at the moment on all pre-defined paths it will be 2.

## func (\*Logger) Panic

```
func (l *Logger) Panic(v ...interface{})
```

Panic is equivalent to l.Println() followed by a call to panic().

## func (\*Logger) Panicf

```
func (l *Logger) Panicf(format string, v ...interface{})
```

Panicf is equivalent to l.Printf() followed by a call to panic().

## func (\*Logger) Panicln

```
func (l *Logger) Panicln(v ...interface{})
```

Panicln is equivalent to l.Println() followed by a call to panic().

## func (\*Logger) Prefix

```
func (l *Logger) Prefix() string
```

Prefix returns the output prefix for the logger.

## func (\*Logger) Print

```
func (l *Logger) Print(v ...interface{})
```

Print calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Print.

## func (\*Logger) Printf

```
func (l *Logger) Printf(format string, v ...interface{})
```

Printf calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Printf.

## func (\*Logger) Println

```
func (l *Logger) Println(v ...interface{})
```

Println calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Println.

## func (\*Logger) SetFlags

```
func (l *Logger) SetFlags(flag int)
```

SetFlags sets the output flags for the logger. The flag bits are Ldate, Ltime, and so on.

## func (\*Logger) SetOutput

```
func (l *Logger) SetOutput(w io.Writer)
```

SetOutput sets the output destination for the logger.

## func (\*Logger) SetPrefix

```
func (l *Logger) SetPrefix(prefix string)
```

SetPrefix sets the output prefix for the logger.

## func (\*Logger) Writer

```
func (l *Logger) Writer() io.Writer
```

Writer returns the output destination for the logger.

# Package math

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package math provides basic constants and mathematical functions.

This package does not guarantee bit-identical results across architectures.

## Constants

```
const (
 E = 2.71828182845904523536028747135266249775724709369995957496696763 // https://
 Pi = 3.14159265358979323846264338327950288419716939937510582097494459 // https://
 Phi = 1.61803398874989484820458683436563811772030917980576286213544862 // https://

 Sqrt2 = 1.41421356237309504880168872420969807856967187537694807317667974 // https://
 SqrtE = 1.64872127070012814684865078781416357165377610071014801157507931 // https://
 SqrtPi = 1.77245385090551602729816748334114518279754945612238712821380779 // https://
 SqrtPhi = 1.27201964951406896425242246173749149171560804184009624861664038 // https://

 Ln2 = 0.693147180559945309417232121458176568075500134360255254120680009 // https://
 Log2E = 1 / Ln2
 Ln10 = 2.30258509299404568401799145468436420760110148862877297603332790 // https://
 Log10E = 1 / Ln10
)
```

Mathematical constants.

```
const (
 MaxFloat32 = 3.40282346638528859811704183484516925440e+38 // 2**127
 SmallestNonzeroFloat32 = 1.401298464324817070923729583289916131280e-45 // 1 / 2**-126

 MaxFloat64 = 1.797693134862315708145274237317043567981e+308 // 2**102
 SmallestNonzeroFloat64 = 4.940656458412465441765687928682213723651e-324 // 1 / 2**-102
)
```

Floating-point limit values. Max is the largest finite value representable by the type. SmallestNonzero is the smallest positive, non-zero value representable by the type.

```
const (
 MaxInt8 = 1<<7 - 1
 MinInt8 = -1 << 7
 MaxInt16 = 1<<15 - 1
 MinInt16 = -1 << 15
)
```

```
MaxInt32 = 1<<31 - 1
MinInt32 = -1 << 31
MaxInt64 = 1<<63 - 1
MinInt64 = -1 << 63
MaxUint8 = 1<<8 - 1
MaxUint16 = 1<<16 - 1
MaxUint32 = 1<<32 - 1
MaxUint64 = 1<<64 - 1
)
```

Integer limit values.

## func Abs

```
func Abs(x float64) float64
```

Abs returns the absolute value of x.

Special cases are:

```
Abs(+Inf) = +Inf
Abs(NaN) = NaN
```

## func Acos

```
func Acos(x float64) float64
```

Acos returns the arccosine, in radians, of x.

Special case is:

```
Acos(x) = NaN if x < -1 or x > 1
```

## func Acosh

```
func Acosh(x float64) float64
```

Acosh returns the inverse hyperbolic cosine of x.

Special cases are:

```
Acosh(+Inf) = +Inf
Acosh(x) = NaN if x < 1
Acosh(NaN) = NaN
```

## func Asin

```
func Asin(x float64) float64
```

Asin returns the arcsine, in radians, of x.

Special cases are:

```
Asin(±0) = ±0
Asin(x) = NaN if x < -1 or x > 1
```

## func Asinh

```
func Asinh(x float64) float64
```

Asinh returns the inverse hyperbolic sine of x.

Special cases are:

```
Asinh(±0) = ±0
Asinh(±Inf) = ±Inf
Asinh(NaN) = NaN
```

## func Atan

```
func Atan(x float64) float64
```

Atan returns the arctangent, in radians, of x.

Special cases are:

```
Atan(±0) = ±0
Atan(±Inf) = ±Pi/2
```

## func Atan2

```
func Atan2(y, x float64) float64
```

Atan2 returns the arc tangent of y/x, using the signs of the two to determine the quadrant of the return value.

Special cases are (in order):

```
Atan2(y, NaN) = NaN
Atan2(NaN, x) = NaN
Atan2(+0, x>=0) = +0
Atan2(-0, x>=0) = -0
Atan2(+0, x<=-0) = +Pi
```

```
Atan2(-0, x<=-0) = -Pi
Atan2(y>0, 0) = +Pi/2
Atan2(y<0, 0) = -Pi/2
Atan2(+Inf, +Inf) = +Pi/4
Atan2(-Inf, +Inf) = -Pi/4
Atan2(+Inf, -Inf) = 3Pi/4
Atan2(-Inf, -Inf) = -3Pi/4
Atan2(y, +Inf) = 0
Atan2(y>0, -Inf) = +Pi
Atan2(y<0, -Inf) = -Pi
Atan2(+Inf, x) = +Pi/2
Atan2(-Inf, x) = -Pi/2
```

## func Atanh

```
func Atanh(x float64) float64
```

Atanh returns the inverse hyperbolic tangent of x.

Special cases are:

```
Atanh(1) = +Inf
Atanh(±0) = ±0
Atanh(-1) = -Inf
Atanh(x) = NaN if x < -1 or x > 1
Atanh(NaN) = NaN
```

## func Cbrt

```
func Cbrt(x float64) float64
```

Cbrt returns the cube root of x.

Special cases are:

```
Cbrt(±0) = ±0
Cbrt(±Inf) = ±Inf
Cbrt(NaN) = NaN
```

## func Ceil

```
func Ceil(x float64) float64
```

Ceil returns the least integer value greater than or equal to x.

Special cases are:

```
Ceil(±0) = ±0
Ceil(±Inf) = ±Inf
Ceil(NaN) = NaN
```

## func **Copysign**

```
func Copysign(x, y float64) float64
```

Copysign returns a value with the magnitude of x and the sign of y.

## func **Cos**

```
func Cos(x float64) float64
```

Cos returns the cosine of the radian argument x.

Special cases are:

```
Cos(±Inf) = NaN
Cos(NaN) = NaN
```

## func **Cosh**

```
func Cosh(x float64) float64
```

Cosh returns the hyperbolic cosine of x.

Special cases are:

```
Cosh(±0) = 1
Cosh(±Inf) = +Inf
Cosh(NaN) = NaN
```

## func **Dim**

```
func Dim(x, y float64) float64
```

Dim returns the maximum of x-y or 0.

Special cases are:

```
Dim(+Inf, +Inf) = NaN
Dim(-Inf, -Inf) = NaN
Dim(x, NaN) = Dim(NaN, x) = NaN
```

## func Erf

```
func Erf(x float64) float64
```

Erf returns the error function of x.

Special cases are:

```
Erf(+Inf) = 1
Erf(-Inf) = -1
Erf(NaN) = NaN
```

## func Erfc

```
func Erfc(x float64) float64
```

Erfc returns the complementary error function of x.

Special cases are:

```
Erfc(+Inf) = 0
Erfc(-Inf) = 2
Erfc(NaN) = NaN
```

## func Erfcinv

```
func Erfcinv(x float64) float64
```

Erfcinv returns the inverse of Erfc(x).

Special cases are:

```
Erfcinv(0) = +Inf
Erfcinv(2) = -Inf
Erfcinv(x) = NaN if x < 0 or x > 2
Erfcinv(NaN) = NaN
```

## func Erfinv

```
func Erfinv(x float64) float64
```

Erfinv returns the inverse error function of x.

Special cases are:

```
Erfinv(1) = +Inf
Erfinv(-1) = -Inf
```

```
Erfinv(x) = NaN if x < -1 or x > 1
Erfinv(NaN) = NaN
```

## func Exp

```
func Exp(x float64) float64
```

Exp returns  $e^{**x}$ , the base-e exponential of x.

Special cases are:

```
Exp(+Inf) = +Inf
Exp(NaN) = NaN
```

Very large values overflow to 0 or +Inf. Very small values underflow to 1.

## func Exp2

```
func Exp2(x float64) float64
```

Exp2 returns  $2^{**x}$ , the base-2 exponential of x.

Special cases are the same as Exp.

## func Expm1

```
func Expm1(x float64) float64
```

Expm1 returns  $e^{**x} - 1$ , the base-e exponential of x minus 1. It is more accurate than  $\text{Exp}(x) - 1$  when x is near zero.

Special cases are:

```
Expm1(+Inf) = +Inf
Expm1(-Inf) = -1
Expm1(NaN) = NaN
```

Very large values overflow to -1 or +Inf.

## func FMA

```
func FMA(x, y, z float64) float64
```

FMA returns  $x * y + z$ , computed with only one rounding. (That is, FMA returns the fused multiply-add of x, y, and z.)

## func **Float32bits**

```
func Float32bits(f float32) uint32
```

Float32bits returns the IEEE 754 binary representation of f, with the sign bit of f and the result in the same bit position.  $\text{Float32bits}(\text{Float32frombits}(x)) == x$ .

## func **Float32frombits**

```
func Float32frombits(b uint32) float32
```

Float32frombits returns the floating-point number corresponding to the IEEE 754 binary representation b, with the sign bit of b and the result in the same bit position.  $\text{Float32frombits}(\text{Float32bits}(x)) == x$ .

## func **Float64bits**

```
func Float64bits(f float64) uint64
```

Float64bits returns the IEEE 754 binary representation of f, with the sign bit of f and the result in the same bit position, and  $\text{Float64bits}(\text{Float64frombits}(x)) == x$ .

## func **Float64frombits**

```
func Float64frombits(b uint64) float64
```

Float64frombits returns the floating-point number corresponding to the IEEE 754 binary representation b, with the sign bit of b and the result in the same bit position.  $\text{Float64frombits}(\text{Float64bits}(x)) == x$ .

## func **Floor**

```
func Floor(x float64) float64
```

Floor returns the greatest integer value less than or equal to x.

Special cases are:

```
Floor(±0) = ±0
Floor(±Inf) = ±Inf
Floor(NaN) = NaN
```

## func **Frexp**

```
func Frexp(f float64) (frac float64, exp int)
```

`Frexp` breaks  $f$  into a normalized fraction and an integral power of two. It returns `frac` and `exp` satisfying  $f == \text{frac} \times 2^{\text{exp}}$ , with the absolute value of `frac` in the interval  $[\frac{1}{2}, 1)$ .

Special cases are:

```
Frexp(±0) = ±0, 0
Frexp(±Inf) = ±Inf, 0
Frexp(NaN) = NaN, 0
```

## func `Gamma`

```
func Gamma(x float64) float64
```

`Gamma` returns the Gamma function of  $x$ .

Special cases are:

```
Gamma(+Inf) = +Inf
Gamma(+0) = +Inf
Gamma(-0) = -Inf
Gamma(x) = NaN for integer x < 0
Gamma(-Inf) = NaN
Gamma(NaN) = NaN
```

## func `Hypot`

```
func Hypot(p, q float64) float64
```

`Hypot` returns  $\sqrt{p^2 + q^2}$ , taking care to avoid unnecessary overflow and underflow.

Special cases are:

```
Hypot(±Inf, q) = +Inf
Hypot(p, ±Inf) = +Inf
Hypot(NaN, q) = NaN
Hypot(p, NaN) = NaN
```

## func `Ilogb`

```
func Ilogb(x float64) int
```

`Ilogb` returns the binary exponent of  $x$  as an integer.

Special cases are:

```
Ilogb(±Inf) = MaxInt32
Ilogb(0) = MinInt32
```

```
Ilogb(NaN) = MaxInt32
```

## func Inf

```
func Inf(sign int) float64
```

Inf returns positive infinity if sign  $\geq 0$ , negative infinity if sign  $< 0$ .

## func IsInf

```
func IsInf(f float64, sign int) bool
```

IsInf reports whether f is an infinity, according to sign. If sign  $> 0$ , IsInf reports whether f is positive infinity. If sign  $< 0$ , IsInf reports whether f is negative infinity. If sign  $= 0$ , IsInf reports whether f is either infinity.

## func IsNaN

```
func IsNaN(f float64) (is bool)
```

IsNaN reports whether f is an IEEE 754 ``not-a-number" value.

## func J0

```
func J0(x float64) float64
```

J0 returns the order-zero Bessel function of the first kind.

Special cases are:

```
J0(±Inf) = 0
J0(0) = 1
J0(NaN) = NaN
```

## func J1

```
func J1(x float64) float64
```

J1 returns the order-one Bessel function of the first kind.

Special cases are:

```
J1(±Inf) = 0
J1(NaN) = NaN
```

## func `Jn`

```
func Jn(n int, x float64) float64
```

`Jn` returns the order-`n` Bessel function of the first kind.

Special cases are:

```
Jn(n, ±Inf) = 0
Jn(n, NaN) = NaN
```

## func `Ldexp`

```
func Ldexp(frac float64, exp int) float64
```

`Ldexp` is the inverse of `Frexp`. It returns `frac`  $\times$   $2^{**exp}$ .

Special cases are:

```
Ldexp(±0, exp) = ±0
Ldexp(±Inf, exp) = ±Inf
Ldexp(NaN, exp) = NaN
```

## func `Lgamma`

```
func Lgamma(x float64) (lgamma float64, sign int)
```

`Lgamma` returns the natural logarithm and sign (-1 or +1) of  $\Gamma(x)$ .

Special cases are:

```
Lgamma(+Inf) = +Inf
Lgamma(0) = +Inf
Lgamma(-integer) = +Inf
Lgamma(-Inf) = -Inf
Lgamma(NaN) = NaN
```

## func `Log`

```
func Log(x float64) float64
```

`Log` returns the natural logarithm of `x`.

Special cases are:

```
Log(+Inf) = +Inf
Log(0) = -Inf
```

```
Log(x < 0) = NaN
Log(NaN) = NaN
```

## func Log10

```
func Log10(x float64) float64
```

Log10 returns the decimal logarithm of x. The special cases are the same as for Log.

## func Log1p

```
func Log1p(x float64) float64
```

Log1p returns the natural logarithm of 1 plus its argument x. It is more accurate than Log(1 + x) when x is near zero.

Special cases are:

```
Log1p(+Inf) = +Inf
Log1p(±0) = ±0
Log1p(-1) = -Inf
Log1p(x < -1) = NaN
Log1p(NaN) = NaN
```

## func Log2

```
func Log2(x float64) float64
```

Log2 returns the binary logarithm of x. The special cases are the same as for Log.

## func Logb

```
func Logb(x float64) float64
```

Logb returns the binary exponent of x.

Special cases are:

```
Logb(±Inf) = +Inf
Logb(0) = -Inf
Logb(NaN) = NaN
```

## func Max

```
func Max(x, y float64) float64
```

Max returns the larger of x or y.

Special cases are:

```
Max(x, +Inf) = Max(+Inf, x) = +Inf
Max(x, NaN) = Max(NaN, x) = NaN
Max(+0, ±0) = Max(±0, +0) = +0
Max(-0, -0) = -0
```

## func Min

```
func Min(x, y float64) float64
```

Min returns the smaller of x or y.

Special cases are:

```
Min(x, -Inf) = Min(-Inf, x) = -Inf
Min(x, NaN) = Min(NaN, x) = NaN
Min(-0, ±0) = Min(±0, -0) = -0
```

## func Mod

```
func Mod(x, y float64) float64
```

Mod returns the floating-point remainder of x/y. The magnitude of the result is less than y and its sign agrees with that of x.

Special cases are:

```
Mod(±Inf, y) = NaN
Mod(NaN, y) = NaN
Mod(x, 0) = NaN
Mod(x, ±Inf) = x
Mod(x, NaN) = NaN
```

## func Modf

```
func Modf(f float64) (int float64, frac float64)
```

Modf returns integer and fractional floating-point numbers that sum to f. Both values have the same sign as f.

Special cases are:

```
Modf(±Inf) = ±Inf, NaN
Modf(NaN) = NaN, NaN
```

## func **NaN**

```
func NaN() float64
```

NaN returns an IEEE 754 ``not-a-number" value.

## func **Nextafter**

```
func Nextafter(x, y float64) (r float64)
```

Nextafter returns the next representable float64 value after x towards y.

Special cases are:

```
Nextafter(x, x) = x
Nextafter(NaN, y) = NaN
Nextafter(x, NaN) = NaN
```

## func **Nextafter32**

```
func Nextafter32(x, y float32) (r float32)
```

Nextafter32 returns the next representable float32 value after x towards y.

Special cases are:

```
Nextafter32(x, x) = x
Nextafter32(NaN, y) = NaN
Nextafter32(x, NaN) = NaN
```

## func **Pow**

```
func Pow(x, y float64) float64
```

Pow returns  $x^{**}y$ , the base-x exponential of y.

Special cases are (in order):

```
Pow(x, ±0) = 1 for any x
Pow(1, y) = 1 for any y
Pow(x, 1) = x for any x
Pow(NaN, y) = NaN
Pow(x, NaN) = NaN
Pow(±0, y) = ±Inf for y an odd integer < 0
Pow(±0, -Inf) = +Inf
Pow(±0, +Inf) = +0
```

```
Pow(± 0 , y) = +Inf for finite y < 0 and not an odd integer
Pow(± 0 , y) = ± 0 for y an odd integer > 0
Pow(± 0 , y) = +0 for finite y > 0 and not an odd integer
Pow(-1, $\pm \text{Inf}$) = 1
Pow(x, +Inf) = +Inf for $|x| > 1$
Pow(x, -Inf) = +0 for $|x| > 1$
Pow(x, +Inf) = +0 for $|x| < 1$
Pow(x, -Inf) = +Inf for $|x| < 1$
Pow(+Inf, y) = +Inf for y > 0
Pow(+Inf, y) = +0 for y < 0
Pow(-Inf, y) = Pow(-0, -y)
Pow(x, y) = NaN for finite x < 0 and finite non-integer y
```

## func Pow10

```
func Pow10(n int) float64
```

Pow10 returns  $10^{*n}$ , the base-10 exponential of n.

Special cases are:

```
Pow10(n) = 0 for n < -323
Pow10(n) = +Inf for n > 308
```

## func Remainder

```
func Remainder(x, y float64) float64
```

Remainder returns the IEEE 754 floating-point remainder of x/y.

Special cases are:

```
Remainder($\pm \text{Inf}$, y) = NaN
Remainder(NaN, y) = NaN
Remainder(x, 0) = NaN
Remainder(x, $\pm \text{Inf}$) = x
Remainder(x, NaN) = NaN
```

## func Round

```
func Round(x float64) float64
```

Round returns the nearest integer, rounding half away from zero.

Special cases are:

```
Round(± 0) = ± 0
Round($\pm \text{Inf}$) = $\pm \text{Inf}$
```

```
Round(NaN) = NaN
```

## func RoundToEven

```
func RoundToEven(x float64) float64
```

RoundToEven returns the nearest integer, rounding ties to even.

Special cases are:

```
RoundToEven(±0) = ±0
RoundToEven(±Inf) = ±Inf
RoundToEven(NaN) = NaN
```

## func Signbit

```
func Signbit(x float64) bool
```

Signbit reports whether x is negative or negative zero.

## func Sin

```
func Sin(x float64) float64
```

Sin returns the sine of the radian argument x.

Special cases are:

```
Sin(±0) = ±0
Sin(±Inf) = NaN
Sin(NaN) = NaN
```

## func Sincos

```
func Sincos(x float64) (sin, cos float64)
```

Sincos returns Sin(x), Cos(x).

Special cases are:

```
Sincos(±0) = ±0, 1
Sincos(±Inf) = NaN, NaN
Sincos(NaN) = NaN, NaN
```

## func Sinh

```
func Sinh(x float64) float64
```

Sinh returns the hyperbolic sine of x.

Special cases are:

```
Sinh(±0) = ±0
Sinh(±Inf) = ±Inf
Sinh(NaN) = NaN
```

## func Sqrt

```
func Sqrt(x float64) float64
```

Sqrt returns the square root of x.

Special cases are:

```
Sqrt(+Inf) = +Inf
Sqrt(±0) = ±0
Sqrt(x < 0) = NaN
Sqrt(NaN) = NaN
```

## func Tan

```
func Tan(x float64) float64
```

Tan returns the tangent of the radian argument x.

Special cases are:

```
Tan(±0) = ±0
Tan(±Inf) = NaN
Tan(NaN) = NaN
```

## func Tanh

```
func Tanh(x float64) float64
```

Tanh returns the hyperbolic tangent of x.

Special cases are:

```
Tanh(±0) = ±0
Tanh(±Inf) = ±1
Tanh(NaN) = NaN
```

## func Trunc

```
func Trunc(x float64) float64
```

Trunc returns the integer value of x.

Special cases are:

```
Trunc(±0) = ±0
Trunc(±Inf) = ±Inf
Trunc(NaN) = NaN
```

## func Y0

```
func Y0(x float64) float64
```

Y0 returns the order-zero Bessel function of the second kind.

Special cases are:

```
Y0(+Inf) = 0
Y0(0) = -Inf
Y0(x < 0) = NaN
Y0(NaN) = NaN
```

## func Y1

```
func Y1(x float64) float64
```

Y1 returns the order-one Bessel function of the second kind.

Special cases are:

```
Y1(+Inf) = 0
Y1(0) = -Inf
Y1(x < 0) = NaN
Y1(NaN) = NaN
```

## func Yn

```
func Yn(n int, x float64) float64
```

Yn returns the order-n Bessel function of the second kind.

Special cases are:

$Yn(n, +\infty) = 0$   
 $Yn(n \geq 0, 0) = -\infty$   
 $Yn(n < 0, 0) = +\infty$  if  $n$  is odd,  $-\infty$  if  $n$  is even  
 $Yn(n, x < 0) = \text{NaN}$   
 $Yn(n, \text{NaN}) = \text{NaN}$

# Package rand

 go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package rand implements pseudo-random number generators.

Random numbers are generated by a Source. Top-level functions, such as `Float64` and `Int`, use a default shared Source that produces a deterministic sequence of values each time a program is run. Use the `Seed` function to initialize the default Source if different behavior is required for each run. The default Source is safe for concurrent use by multiple goroutines, but Sources created by `NewSource` are not.

Mathematical interval notation such as  $[0, n)$  is used throughout the documentation for this package.

For random numbers suitable for security-sensitive work, see the `crypto/rand` package.

### func `ExpFloat64`

```
func ExpFloat64() float64
```

`ExpFloat64` returns an exponentially distributed `float64` in the range  $(0, +\text{math.MaxFloat64}]$  with an exponential distribution whose rate parameter (`lambda`) is 1 and whose mean is  $1/\lambda$  (1) from the default Source. To produce a distribution with a different rate parameter, callers can adjust the output using:

```
sample = ExpFloat64() / desiredRateParameter
```

### func `Float32`

```
func Float32() float32
```

`Float32` returns, as a `float32`, a pseudo-random number in  $[0.0, 1.0)$  from the default Source.

### func `Float64`

```
func Float64() float64
```

`Float64` returns, as a `float64`, a pseudo-random number in  $[0.0, 1.0)$  from the default Source.

### func `Int`

```
func Int() int
```

Int returns a non-negative pseudo-random int from the default Source.

## func Int31

```
func Int31() int32
```

Int31 returns a non-negative pseudo-random 31-bit integer as an int32 from the default Source.

## func Int31n

```
func Int31n(n int32) int32
```

Int31n returns, as an int32, a non-negative pseudo-random number in  $[0, n)$  from the default Source. It panics if  $n \leq 0$ .

## func Int63

```
func Int63() int64
```

Int63 returns a non-negative pseudo-random 63-bit integer as an int64 from the default Source.

## func Int63n

```
func Int63n(n int64) int64
```

Int63n returns, as an int64, a non-negative pseudo-random number in  $[0, n)$  from the default Source. It panics if  $n \leq 0$ .

## func Intn

```
func Intn(n int) int
```

Intn returns, as an int, a non-negative pseudo-random number in  $[0, n)$  from the default Source. It panics if  $n \leq 0$ .

## func NormFloat64

```
func NormFloat64() float64
```

NormFloat64 returns a normally distributed float64 in the range  $[-\text{math.MaxFloat64}, +\text{math.MaxFloat64}]$  with standard normal distribution (mean = 0, stddev = 1) from the default Source. To produce a different normal distribution, callers can adjust the output using:

```
sample = NormFloat64() * desiredStdDev + desiredMean
```

## func **Perm**

```
func Perm(n int) []int
```

Perm returns, as a slice of n ints, a pseudo-random permutation of the integers [0,n) from the default Source.

## func **Read**

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

Read generates len(p) random bytes from the default Source and writes them into p. It always returns len(p) and a nil error. Read, unlike the Rand.Read method, is safe for concurrent use.

## func **Seed**

```
func Seed(seed int64)
```

Seed uses the provided seed value to initialize the default Source to a deterministic state. If Seed is not called, the generator behaves as if seeded by Seed(1). Seed values that have the same remainder when divided by  $2^{31}-1$  generate the same pseudo-random sequence. Seed, unlike the Rand.Seed method, is safe for concurrent use.

## func **Shuffle**

```
func Shuffle(n int, swap func(i, j int))
```

Shuffle pseudo-randomizes the order of elements using the default Source. n is the number of elements. Shuffle panics if n < 0. swap swaps the elements with indexes i and j.

## func **Uint32**

```
func Uint32() uint32
```

Uint32 returns a pseudo-random 32-bit value as a uint32 from the default Source.

## func **Uint64**

```
func Uint64() uint64
```

Uint64 returns a pseudo-random 64-bit value as a uint64 from the default Source.

## type Rand

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

A Rand is a source of random numbers.

## func New

```
func New(src Source) *Rand
```

New returns a new Rand that uses random values from src to generate other random values.

## func (\*Rand) ExpFloat64

```
func (r *Rand) ExpFloat64() float64
```

ExpFloat64 returns an exponentially distributed float64 in the range (0, +math.MaxFloat64] with an exponential distribution whose rate parameter (lambda) is 1 and whose mean is 1/lambda (1). To produce a distribution with a different rate parameter, callers can adjust the output using:

```
sample = ExpFloat64() / desiredRateParameter
```

## func (\*Rand) Float32

```
func (r *Rand) Float32() float32
```

Float32 returns, as a float32, a pseudo-random number in [0.0,1.0].

## func (\*Rand) Float64

```
func (r *Rand) Float64() float64
```

Float64 returns, as a float64, a pseudo-random number in [0.0,1.0].

## func (\*Rand) Int

```
func (r *Rand) Int() int
```

Int returns a non-negative pseudo-random int.

## func (\*Rand) Int31

```
func (r *Rand) Int31() int32
```

Int31 returns a non-negative pseudo-random 31-bit integer as an int32.

## func (\*Rand) **Int31n**

```
func (r *Rand) Int31n(n int32) int32
```

Int31n returns, as an int32, a non-negative pseudo-random number in [0,n). It panics if n <= 0.

## func (\*Rand) **Int63**

```
func (r *Rand) Int63() int64
```

Int63 returns a non-negative pseudo-random 63-bit integer as an int64.

## func (\*Rand) **Int63n**

```
func (r *Rand) Int63n(n int64) int64
```

Int63n returns, as an int64, a non-negative pseudo-random number in [0,n). It panics if n <= 0.

## func (\*Rand) **Intn**

```
func (r *Rand) Intn(n int) int
```

Intn returns, as an int, a non-negative pseudo-random number in [0,n). It panics if n <= 0.

## func (\*Rand) **NormFloat64**

```
func (r *Rand) NormFloat64() float64
```

NormFloat64 returns a normally distributed float64 in the range -math.MaxFloat64 through +math.MaxFloat64 inclusive, with standard normal distribution (mean = 0, stddev = 1). To produce a different normal distribution, callers can adjust the output using:

```
sample = NormFloat64() * desiredStdDev + desiredMean
```

## func (\*Rand) **Perm**

```
func (r *Rand) Perm(n int) []int
```

Perm returns, as a slice of n ints, a pseudo-random permutation of the integers [0,n).

## func (\*Rand) **Read**

```
func (r *Rand) Read(p []byte) (n int, err error)
```

Read generates  $\text{len}(p)$  random bytes and writes them into  $p$ . It always returns  $\text{len}(p)$  and a nil error. Read should not be called concurrently with any other Rand method.

## func (\*Rand) Seed

```
func (r *Rand) Seed(seed int64)
```

Seed uses the provided seed value to initialize the generator to a deterministic state. Seed should not be called concurrently with any other Rand method.

## func (\*Rand) Shuffle

```
func (r *Rand) Shuffle(n int, swap func(i, j int))
```

Shuffle pseudo-randomizes the order of elements.  $n$  is the number of elements. Shuffle panics if  $n < 0$ . swap swaps the elements with indexes  $i$  and  $j$ .

## func (\*Rand) Uint32

```
func (r *Rand) Uint32() uint32
```

Uint32 returns a pseudo-random 32-bit value as a uint32.

## func (\*Rand) Uint64

```
func (r *Rand) Uint64() uint64
```

Uint64 returns a pseudo-random 64-bit value as a uint64.

## type Source

```
type Source interface {
 Int63() int64
 Seed(seed int64)
}
```

A Source represents a source of uniformly-distributed pseudo-random int64 values in the range  $[0, 1<<63)$ .

## func NewSource

```
func NewSource(seed int64) Source
```

NewSource returns a new pseudo-random Source seeded with the given value. Unlike the default Source used by top-level functions, this source is not safe for concurrent use by multiple goroutines.

## type Source64

```
type Source64 interface {
 Source
 Uint64() uint64
}
```

A Source64 is a Source that can also generate uniformly-distributed pseudo-random uint64 values in the range  $[0, 1 << 64]$  directly. If a Rand r's underlying Source s implements Source64, then r.Uint64 returns the result of one call to s.Uint64 instead of making two calls to s.Int63.

## type Zipf

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

A Zipf generates Zipf distributed variates.

## func NewZipf

```
func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf
```

NewZipf returns a Zipf variate generator. The generator generates values  $k \in [0, imax]$  such that  $P(k)$  is proportional to  $(v + k)^{-s}$ . Requirements:  $s > 1$  and  $v \geq 1$ .

## func (\*Zipf) Uint64

```
func (z *Zipf) Uint64() uint64
```

Uint64 returns a value drawn from the Zipf distribution described by the Zipf object.

# Package mime

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package mime implements parts of the MIME spec.

## Constants

```
const (
 // BEncoding represents Base64 encoding scheme as defined by RFC 2045.
 BEncoding = WordEncoder('b')
 // QEncoding represents the Q-encoding scheme as defined by RFC 2047.
 QEncoding = WordEncoder('q')
)
```

## Variables

```
var ErrInvalidMediaParameter = errors.New("mime: invalid media parameter")
```

ErrInvalidMediaParameter is returned by ParseMediaType if the media type value was found but there was an error parsing the optional parameters

## func [AddExtensionType](#)

```
func AddExtensionType(ext, typ string) error
```

AddExtensionType sets the MIME type associated with the extension ext to typ. The extension should begin with a leading dot, as in ".html".

## func [ExtensionsByType](#)

```
func ExtensionsByType(typ string) (\[\]string, error)
```

ExtensionsByType returns the extensions known to be associated with the MIME type typ. The returned extensions will each begin with a leading dot, as in ".html". When typ has no associated extensions, ExtensionsByType returns an nil slice.

## func [FormatMediaType](#)

```
func FormatMediaType(t string, param map\[string\]string) string
```

FormatMediaType serializes mediatype t and the parameters param as a media type conforming to [RFC 2045](#) and [RFC 2616](#). The type and parameter names are written in lower-case. When any of the arguments result in a standard violation then FormatMediaType returns the empty string.

## func ParseMediaType

```
func ParseMediaType(v string) (mediatype string, params map\[string\]string, err error)
```

ParseMediaType parses a media type value and any optional parameters, per [RFC 1521](#). Media types are the values in Content-Type and Content-Disposition headers ([RFC 2183](#)). On success, ParseMediaType returns the media type converted to lowercase and trimmed of white space and a non-nil map. If there is an error parsing the optional parameter, the media type will be returned along with the error ErrInvalidMediaParameter. The returned map, params, maps from the lowercase attribute to the attribute value with its case preserved.

## func TypeByExtension

```
func TypeByExtension(ext string) string
```

TypeByExtension returns the MIME type associated with the file extension ext. The extension ext should begin with a leading dot, as in ".html". When ext has no associated type, TypeByExtension returns "".

Extensions are looked up first case-sensitively, then case-insensitively.

The built-in table is small but on unix it is augmented by the local system's mime.types file(s) if available under one or more of these names:

```
/etc/mime.types
/etc/apache2/mime.types
/etc/apache/mime.types
```

On Windows, MIME types are extracted from the registry.

Text types have the charset parameter set to "utf-8" by default.

## type WordDecoder

```
type WordDecoder struct {
 // CharsetReader, if non-nil, defines a function to generate
 // charset-conversion readers, converting from the provided
 // charset into UTF-8.
 //Charsets are always lower-case. utf-8, iso-8859-1 and us-ascii charsets
 // are handled by default.
 // One of the CharsetReader's result values must be non-nil.
 CharsetReader func(charset string, input io.Reader) (io.Reader, error)
}
```

A WordDecoder decodes MIME headers containing [RFC 2047](#) encoded-words.

## func (\*WordDecoder) [Decode](#)

```
func (d *WordDecoder) Decode(word string) (string, error)
```

Decode decodes an [RFC 2047](#) encoded-word.

## func (\*WordDecoder) [DecodeHeader](#)

```
func (d *WordDecoder) DecodeHeader(header string) (string, error)
```

DecodeHeader decodes all encoded-words of the given string. It returns an error if and only if CharsetReader of d returns an error.

## type [WordEncoder](#)

```
type WordEncoder byte
```

A WordEncoder is an [RFC 2047](#) encoded-word encoder.

## func (WordEncoder) [Encode](#)

```
func (e WordEncoder) Encode(charset, s string) string
```

Encode returns the encoded-word form of s. If s is ASCII without special characters, it is returned unchanged. The provided charset is the IANA charset name of s. It is case insensitive.

# Package multipart

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package multipart implements MIME multipart parsing, as defined in RFC 2046.

The implementation is sufficient for HTTP ([RFC 2388](#)) and the multipart bodies generated by popular browsers.

## Variables

```
var ErrMessageTooLarge = errors.New("multipart: message too large")
```

ErrMsgTooLarge is returned by ReadForm if the message form data is too large to be processed.

## type File

```
type File interface {
 io.Reader
 io.ReaderAt
 io.Seeker
 io.Closer
}
```

File is an interface to access the file part of a multipart message. Its contents may be either stored in memory or on disk. If stored on disk, the File's underlying concrete type will be an `*os.File`.

## type FileHeader

```
type FileHeader struct {
 Filename string
 Header textproto.MIMEHeader
 Size int64
 // contains filtered or unexported fields
}
```

A FileHeader describes a file part of a multipart request.

## func (\*FileHeader) Open

```
func (fh *FileHeader) Open() (File, error)
```

Open opens and returns the FileHeader's associated File.

## type Form

```
type Form struct {
 Value map[string][]string
 File map[string][]*FileHeader
}
```

Form is a parsed multipart form. Its File parts are stored either in memory or on disk, and are accessible via the \*FileHeader's Open method. Its Value parts are stored as strings. Both are keyed by field name.

## func (\*Form) RemoveAll

```
func (f *Form) RemoveAll() error
```

RemoveAll removes any temporary files associated with a Form.

## type Part

```
type Part struct {
 // The headers of the body, if any, with the keys canonicalized
 // in the same fashion that the Go http.Request headers are.
 // For example, "foo-bar" changes case to "Foo-Bar"
 Header textproto.MIMEHeader
 // contains filtered or unexported fields
}
```

A Part represents a single part in a multipart body.

## func (\*Part) Close

```
func (p *Part) Close() error
```

## func (\*Part) FileName

```
func (p *Part) FileName() string
```

FileName returns the filename parameter of the Part's Content-Disposition header.

## func (\*Part) FormName

```
func (p *Part) FormName() string
```

FormName returns the name parameter if p has a Content-Disposition of type "form-data". Otherwise it returns the empty string.

## func (\*Part) Read

```
func (p *Part) Read(d []byte) (n int, err error)
```

Read reads the body of a part, after its headers and before the next part (if any) begins.

## type Reader

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

Reader is an iterator over parts in a MIME multipart body. Reader's underlying parser consumes its input as needed. Seeking isn't supported.

## func NewReader

```
func NewReader(r io.Reader, boundary string) *Reader
```

NewReader creates a new multipart Reader reading from r using the given MIME boundary.

The boundary is usually obtained from the "boundary" parameter of the message's "Content-Type" header. Use mime.ParseMediaType to parse such headers.

## func (\*Reader) NextPart

```
func (r *Reader) NextPart() (*Part, error)
```

NextPart returns the next part in the multipart or an error. When there are no more parts, the error io.EOF is returned.

As a special case, if the "Content-Transfer-Encoding" header has a value of "quoted-printable", that header is instead hidden and the body is transparently decoded during Read calls.

## func (\*Reader) NextRawPart

```
func (r *Reader) NextRawPart() (*Part, error)
```

NextRawPart returns the next part in the multipart or an error. When there are no more parts, the error io.EOF is returned.

Unlike NextPart, it does not have special handling for "Content-Transfer-Encoding: quoted-printable".

## func (\*Reader) ReadForm

```
func (r *Reader) ReadForm(maxMemory int64) (*Form, error)
```

ReadForm parses an entire multipart message whose parts have a Content-Disposition of "form-data". It stores up to maxMemory bytes + 10MB (reserved for non-file parts) in memory. File parts which can't be stored in memory will be stored on disk in temporary files. It returns ErrMessageTooLarge if all non-file parts can't be stored in memory.

## type Writer

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

A Writer generates multipart messages.

## func NewWriter

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

NewWriter returns a new multipart Writer with a random boundary, writing to w.

## func (\*Writer) Boundary

```
func (w *Writer) Boundary() string
```

Boundary returns the Writer's boundary.

## func (\*Writer) Close

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

Close finishes the multipart message and writes the trailing boundary end line to the output.

## func (\*Writer) CreateFormField

```
func (w *Writer) CreateFormField(fieldname string) (io.Writer, error)
```

CreateFormField calls CreatePart with a header using the given field name.

## func (\*Writer) CreateFormFile

```
func (w *Writer) CreateFormFile(fieldname, filename string) (io.Writer, error)
```

CreateFormFile is a convenience wrapper around CreatePart. It creates a new form-data header with the provided field name and file name.

## func (\*Writer) `CreatePart`

```
func (w *Writer) CreatePart(header textproto.MIMEHeader) (io.Writer, error)
```

CreatePart creates a new multipart section with the provided header. The body of the part should be written to the returned Writer. After calling CreatePart, any previous part may no longer be written to.

## func (\*Writer) `FormDataContentType`

```
func (w *Writer) FormDataContentType() string
```

FormDataContentType returns the Content-Type for an HTTP multipart/form-data with this Writer's Boundary.

## func (\*Writer) `SetBoundary`

```
func (w *Writer) SetBoundary(boundary string) error
```

SetBoundary overrides the Writer's default randomly-generated boundary separator with an explicit value.

SetBoundary must be called before any parts are created, may only contain certain ASCII characters, and must be non-empty and at most 70 bytes long.

## func (\*Writer) `WriteField`

```
func (w *Writer) WriteField(fieldname, value string) error
```

WriteField calls CreateFormField and then writes the given value.

# Package net

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package net provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets.

Although the package provides access to low-level networking primitives, most clients will need only the basic interface provided by the Dial, Listen, and Accept functions and the associated Conn and Listener interfaces. The crypto/tls package uses the same interfaces and similar Dial and Listen functions.

The Dial function connects to a server:

```
conn, err := net.Dial("tcp", "golang.org:80")
if err != nil {
 // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

The Listen function creates servers:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
 // handle error
}
for {
 conn, err := ln.Accept()
 if err != nil {
 // handle error
 }
 go handleConnection(conn)
}
```

## Name Resolution

The method for resolving domain names, whether indirectly with functions like Dial or directly with functions like LookupHost and LookupAddr, varies by operating system.

On Unix systems, the resolver has two options for resolving names. It can use a pure Go resolver that sends DNS requests directly to the servers listed in /etc/resolv.conf, or it can use a cgo-based resolver

that calls C library routines such as `getaddrinfo` and `getnameinfo`.

By default the pure Go resolver is used, because a blocked DNS request consumes only a goroutine, while a blocked C call consumes an operating system thread. When cgo is available, the cgo-based resolver is used instead under a variety of conditions: on systems that do not let programs make direct DNS requests (OS X), when the `LOCALDOMAIN` environment variable is present (even if empty), when the `RES_OPTIONS` or `HOSTALIASES` environment variable is non-empty, when the `ASR_CONFIG` environment variable is non-empty (OpenBSD only), when `/etc/resolv.conf` or `/etc/nsswitch.conf` specify the use of features that the Go resolver does not implement, and when the name being looked up ends in `.local` or is an mDNS name.

The resolver decision can be overridden by setting the `netdns` value of the `GODEBUG` environment variable (see package `runtime`) to `go` or `cgo`, as in:

```
export GODEBUG=netdns=go # force pure Go resolver
export GODEBUG=netdns=cgo # force cgo resolver
```

The decision can also be forced while building the Go source tree by setting the `netgo` or `netcgo` build tag.

A numeric `netdns` setting, as in `GODEBUG=netdns=1`, causes the resolver to print debugging information about its decisions. To force a particular resolver while also printing debugging information, join the two settings by a plus sign, as in `GODEBUG=netdns=go+1`.

On Plan 9, the resolver always accesses `/net/cs` and `/net/dns`.

On Windows, the resolver always uses C library functions, such as `GetAddrInfo` and `DnsQuery`.

## Constants

```
const (
 IPv4len = 4
 IPv6len = 16
)
```

IP address lengths (bytes).

## Variables

```
var (
 IPv4bcast = IPv4(255, 255, 255, 255) // limited broadcast
 IPv4allsys = IPv4(224, 0, 0, 1) // all systems
 IPv4allrouter = IPv4(224, 0, 0, 2) // all routers
 IPv4zero = IPv4(0, 0, 0, 0) // all zeros
)
```

Well-known IPv4 addresses

```
var (IPv6zero = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 IPv6unspecified = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 IPv6loopback = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
 IPv6interfacelocalallnodes = IP{0xff, 0x01, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 IPv6linklocalallnodes = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 IPv6linklocalallrouters = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0})
```

## Well-known IPv6 addresses

```
var DefaultResolver = &Resolver{}
```

DefaultResolver is the resolver used by the package-level Lookup functions and by Dialers without a specified Resolver.

```
var (
 ErrWriteToConnected = errors.New("use of WriteTo with pre-connected connection")
)
```

Various errors contained in OpError.

## func JoinHostPort

```
func JoinHostPort(host, port string) string
```

JoinHostPort combines host and port into a network address of the form "host:port". If host contains a colon, as found in literal IPv6 addresses, then JoinHostPort returns "[host]:port".

See `func Dial` for a description of the host and port parameters.

## func `LookupAddr`

```
func LookupAddr(addr string) (names []string, err error)
```

`LookupAddr` performs a reverse lookup for the given address, returning a list of names mapping to that address.

When using the host C library resolver, at most one result will be returned. To bypass the host resolver, use a custom Resolver.

## func **LookupCNAME**

```
func LookupCNAME(host string) (cname string, err error)
```

`LookupCNAME` returns the canonical name for the given host. Callers that do not care about the canonical name can call `LookupHost` or `LookupIP` directly: both take care of resolving the canonical

name as part of the lookup.

A canonical name is the final name after following zero or more CNAME records. LookupCNAME does not return an error if host does not contain DNS "CNAME" records, as long as host resolves to address records.

## func `LookupHost`

```
func LookupHost(host string) (addrs []string, err error)
```

LookupHost looks up the given host using the local resolver. It returns a slice of that host's addresses.

## func `LookupPort`

```
func LookupPort(network, service string) (port int, err error)
```

LookupPort looks up the port for the given network and service.

## func `LookupTXT`

```
func LookupTXT(name string) ([]string, error)
```

LookupTXT returns the DNS TXT records for the given domain name.

## func `ParseCIDR`

```
func ParseCIDR(s string) (IP, *IPNet, error)
```

ParseCIDR parses s as a CIDR notation IP address and prefix length, like "192.0.2.0/24" or "2001:db8::/32", as defined in [RFC 4632](#) and [RFC 4291](#).

It returns the IP address and the network implied by the IP and prefix length. For example, ParseCIDR("192.0.2.1/24") returns the IP address 192.0.2.1 and the network 192.0.2.0/24.

## func `Pipe`

```
func Pipe() (Conn, Conn)
```

Pipe creates a synchronous, in-memory, full duplex network connection; both ends implement the Conn interface. Reads on one end are matched with writes on the other, copying data directly between the two; there is no internal buffering.

## func `SplitHostPort`

```
func SplitHostPort(hostport string) (host, port string, err error)
```

SplitHostPort splits a network address of the form "host:port", "host%zone:port", "[host]:port" or "[host%zone]:port" into host or host%zone and port.

A literal IPv6 address in hostport must be enclosed in square brackets, as in "[::1]:80", "[::1%lo0]:80".

See func Dial for a description of the hostport parameter, and host and port results.

## type Addr

```
type Addr interface {
 Network() string // name of the network (for example, "tcp", "udp")
 String() string // string form of address (for example, "192.0.2.1:25", "[2001:db8:1:1:1:1:1:1]:25")
}
```

Addr represents a network end point address.

The two methods Network and String conventionally return strings that can be passed as the arguments to Dial, but the exact form and meaning of the strings is up to the implementation.

## func InterfaceAddrs

```
func InterfaceAddrs() ([]Addr, error)
```

InterfaceAddrs returns a list of the system's unicast interface addresses.

The returned list does not identify the associated interface; use Interfaces and Interface.Addrs for more detail.

## type AddrError

```
type AddrError struct {
 Err string
 Addr string
}
```

## func (\*AddrError) Error

```
func (e *AddrError) Error() string
```

## func (\*AddrError) Temporary

```
func (e *AddrError) Temporary() bool
```

## func (\*AddrError) Timeout

```
func (e *AddrError) Timeout() bool
```

## type Buffers

```
type Buffers [][]byte
```

Buffers contains zero or more runs of bytes to write.

On certain machines, for certain types of connections, this is optimized into an OS-specific batch write operation (such as "writev").

## func (\*Buffers) Read

```
func (v *Buffers) Read(p []byte) (n int, err error)
```

## func (\*Buffers) WriteTo

```
func (v *Buffers) WriteTo(w io.Writer) (n int64, err error)
```

## type Conn

```
type Conn interface {
 // Read reads data from the connection.
 // Read can be made to time out and return an error after a fixed
 // time limit; see SetDeadline and SetReadDeadline.
 Read(b []byte) (n int, err error)

 // Write writes data to the connection.
 // Write can be made to time out and return an error after a fixed
 // time limit; see SetDeadline and SetWriteDeadline.
 Write(b []byte) (n int, err error)

 // Close closes the connection.
 // Any blocked Read or Write operations will be unblocked and return errors.
 Close() error

 // LocalAddr returns the local network address.
 LocalAddr() Addr

 // RemoteAddr returns the remote network address.
 RemoteAddr() Addr

 // SetDeadline sets the read and write deadlines associated
 // with the connection. It is equivalent to calling both
 // SetReadDeadline and SetWriteDeadline.
 //
 // A deadline is an absolute time after which I/O operations
 // fail instead of blocking. The deadline applies to all future
 // and pending I/O, not just the immediately following call to
 // Read or Write. After a deadline has been exceeded, the
 // connection can be refreshed by setting a deadline in the future.
 //
}
```

```

// If the deadline is exceeded a call to Read or Write or to other
// I/O methods will return an error that wraps os.ErrDeadlineExceeded.
// This can be tested using errors.Is(err, os.ErrDeadlineExceeded).
// The error's Timeout method will return true, but note that there
// are other possible errors for which the Timeout method will
// return true even if the deadline has not been exceeded.
//
// An idle timeout can be implemented by repeatedly extending
// the deadline after successful Read or Write calls.
//
// A zero value for t means I/O operations will not time out.
SetDeadline(t time.Time) error

// SetReadDeadline sets the deadline for future Read calls
// and any currently-blocked Read call.
// A zero value for t means Read will not time out.
SetReadDeadline(t time.Time) error

// SetWriteDeadline sets the deadline for future Write calls
// and any currently-blocked Write call.
// Even if write times out, it may return n > 0, indicating that
// some of the data was successfully written.
// A zero value for t means Write will not time out.
SetWriteDeadline(t time.Time) error
}

```

Conn is a generic stream-oriented network connection.

Multiple goroutines may invoke methods on a Conn simultaneously.

## func Dial

```
func Dial(network, address string) (Conn, error)
```

Dial connects to the address on the named network.

Known networks are "tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only), "ip6" (IPv6-only), "unix", "unixgram" and "unixpacket".

For TCP and UDP networks, the address has the form "host:port". The host must be a literal IP address, or a host name that can be resolved to IP addresses. The port must be a literal port number or a service name. If the host is a literal IPv6 address it must be enclosed in square brackets, as in "[2001:db8::1]:80" or "[fe80::1%zone]:80". The zone specifies the scope of the literal IPv6 address as defined in [RFC 4007](#). The functions JoinHostPort and SplitHostPort manipulate a pair of host and port in this form. When using TCP, and the host resolves to multiple IP addresses, Dial will try each IP address in order until one succeeds.

Examples:

```
Dial("tcp", "golang.org:http")
Dial("tcp", "192.0.2.1:http")
Dial("tcp", "198.51.100.1:80")
Dial("udp", "[2001:db8::1]:domain")
Dial("udp", "[fe80::1%lo0]:53")
Dial("tcp", ":80")
```

For IP networks, the network must be "ip", "ip4" or "ip6" followed by a colon and a literal protocol number or a protocol name, and the address has the form "host". The host must be a literal IP address or a literal IPv6 address with zone. It depends on each operating system how the operating system behaves with a non-well known protocol number such as "0" or "255".

Examples:

```
Dial("ip4:1", "192.0.2.1")
Dial("ip6:ipv6-icmp", "2001:db8::1")
Dial("ip6:58", "fe80::1%lo0")
```

For TCP, UDP and IP networks, if the host is empty or a literal unspecified IP address, as in ":80", "0.0.0.0:80" or "[::]:80" for TCP and UDP, "", "0.0.0.0" or "::" for IP, the local system is assumed.

For Unix networks, the address must be a file system path.

## func DialTimeout

```
func DialTimeout(network, address string, timeout time.Duration) (Conn, error)
```

DialTimeout acts like Dial but takes a timeout.

The timeout includes name resolution, if required. When using TCP, and the host in the address parameter resolves to multiple IP addresses, the timeout is spread over each consecutive dial, such that each is given an appropriate fraction of the time to connect.

See func Dial for a description of the network and address parameters.

## func FileConn

```
func FileConn(f *os.File) (c Conn, err error)
```

FileConn returns a copy of the network connection corresponding to the open file f. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

## type DNSConfigError

```
type DNSConfigError struct {
 Err error
}
```

DNSConfigError represents an error reading the machine's DNS configuration. (No longer used; kept for compatibility.)

## func (\*DNSConfigError) Error

```
func (e *DNSConfigError) Error() string
```

## func (\*DNSConfigError) Temporary

```
func (e *DNSConfigError) Temporary() bool
```

## func (\*DNSConfigError) Timeout

```
func (e *DNSConfigError) Timeout() bool
```

## func (\*DNSConfigError) Unwrap

```
func (e *DNSConfigError) Unwrap() error
```

## type DNSError

```
type DNSError struct {
 Err string // description of the error
 Name string // name looked for
 Server string // server used
 IsTimeout bool // if true, timed out; not all timeouts set this
 IsTemporary bool // if true, error is temporary; not all errors set this
 IsNotFound bool // if true, host could not be found
}
```

DNSError represents a DNS lookup error.

## func (\*DNSError) Error

```
func (e *DNSError) Error() string
```

## func (\*DNSError) Temporary

```
func (e *DNSError) Temporary() bool
```

Temporary reports whether the DNS error is known to be temporary. This is not always known; a DNS lookup may fail due to a temporary error and return a DNSError for which Temporary returns false.

## func (\*DNSError) Timeout

```
func (e *DNSError) Timeout() bool
```

Timeout reports whether the DNS lookup is known to have timed out. This is not always known; a DNS lookup may fail due to a timeout and return a DNSError for which Timeout returns false.

## type Dialer

```
type Dialer struct {
 // Timeout is the maximum amount of time a dial will wait for
 // a connect to complete. If Deadline is also set, it may fail
 // earlier.
 //
 // The default is no timeout.
 //
 // When using TCP and dialing a host name with multiple IP
 // addresses, the timeout may be divided between them.
 //
 // With or without a timeout, the operating system may impose
 // its own earlier timeout. For instance, TCP timeouts are
 // often around 3 minutes.
 Timeout time.Duration

 // Deadline is the absolute point in time after which dials
 // will fail. If Timeout is set, it may fail earlier.
 // Zero means no deadline, or dependent on the operating system
 // as with the Timeout option.
 Deadline time.Time

 // LocalAddr is the local address to use when dialing an
 // address. The address must be of a compatible type for the
 // network being dialed.
 // If nil, a local address is automatically chosen.
 LocalAddr Addr

 // DualStack previously enabled RFC 6555 Fast Fallback
 // support, also known as "Happy Eyeballs", in which IPv4 is
 // tried soon if IPv6 appears to be misconfigured and
 // hanging.
 //
 // Deprecated: Fast Fallback is enabled by default. To
 // disable, set FallbackDelay to a negative value.
 DualStack bool

 // FallbackDelay specifies the length of time to wait before
 // spawning a RFC 6555 Fast Fallback connection. That is, this
 // is the amount of time to wait for IPv6 to succeed before
 // assuming that IPv6 is misconfigured and falling back to
 // IPv4.
 //
 // If zero, a default delay of 300ms is used.
 // A negative value disables Fast Fallback support.
 FallbackDelay time.Duration
```

```

// KeepAlive specifies the interval between keep-alive
// probes for an active network connection.
// If zero, keep-alive probes are sent with a default value
// (currently 15 seconds), if supported by the protocol and operating
// system. Network protocols or operating systems that do
// not support keep-alives ignore this field.
// If negative, keep-alive probes are disabled.
KeepAlive time.Duration

// Resolver optionally specifies an alternate resolver to use.
Resolver *Resolver

// Cancel is an optional channel whose closure indicates that
// the dial should be canceled. Not all types of dials support
// cancellation.
//
// Deprecated: Use DialContext instead.
Cancel <-chan struct{}

// If Control is not nil, it is called after creating the network
// connection but before actually dialing.
//
// Network and address parameters passed to Control method are not
// necessarily the ones passed to Dial. For example, passing "tcp" to Dial
// will cause the Control function to be called with "tcp4" or "tcp6".
Control func(network, address string, c syscall.RawConn) error
}

```

A Dialer contains options for connecting to an address.

The zero value for each field is equivalent to dialing without that option. Dialing with the zero value of Dialer is therefore equivalent to just calling the Dial function.

It is safe to call Dialer's methods concurrently.

## func (\*Dialer) Dial

```
func (d *Dialer) Dial(network, address string) (Conn, error)
```

Dial connects to the address on the named network.

See func Dial for a description of the network and address parameters.

## func (\*Dialer) DialContext

```
func (d *Dialer) DialContext(ctx context.Context, network, address string) (Conn, error)
```

DialContext connects to the address on the named network using the provided context.

The provided Context must be non-nil. If the context expires before the connection is complete, an error is returned. Once successfully connected, any expiration of the context will not affect the connection.

When using TCP, and the host in the address parameter resolves to multiple network addresses, any dial timeout (from d.Timeout or ctx) is spread over each consecutive dial, such that each is given an appropriate fraction of the time to connect. For example, if a host has 4 IP addresses and the timeout is 1 minute, the connect to each single address will be given 15 seconds to complete before trying the next one.

See func Dial for a description of the network and address parameters.

## type Error

```
type Error interface {
 error
 Timeout() bool // Is the error a timeout?
 Temporary() bool // Is the error temporary?
}
```

An Error represents a network error.

## type Flags

```
type Flags uint

const (
 FlagUp Flags = 1 << iota // interface is up
 FlagBroadcast // interface supports broadcast access capabil
 FlagLoopback // interface is a loopback interface
 FlagPointToPoint // interface belongs to a point-to-point link
 FlagMulticast // interface supports multicast access capabil
)
```

## func (Flags) String

```
func (f Flags) String() string
```

## type HardwareAddr

```
type HardwareAddr []byte
```

A HardwareAddr represents a physical hardware address.

## func ParseMAC

```
func ParseMAC(s string) (hw HardwareAddr, err error)
```

ParseMAC parses s as an IEEE 802 MAC-48, EUI-48, EUI-64, or a 20-octet IP over InfiniBand link-layer address using one of the following formats:

```
00:00:5e:00:53:01
02:00:5e:10:00:00:00:01
00:00:00:00:fe:80:00:00:00:00:00:02:00:5e:10:00:00:00:01
00-00-5e-00-53-01
02-00-5e-10-00-00-00-01
00-00-00-00-fe-80-00-00-00-00-00-02-00-5e-10-00-00-00-01
0000.5e00.5301
0200.5e10.0000.0001
0000.0000.fe80.0000.0000.0000.0200.5e10.0000.0001
```

## func (HardwareAddr) String

```
func (a HardwareAddr) String() string
```

## type IP

```
type IP []byte
```

An IP is a single IP address, a slice of bytes. Functions in this package accept either 4-byte (IPv4) or 16-byte (IPv6) slices as input.

Note that in this documentation, referring to an IP address as an IPv4 address or an IPv6 address is a semantic property of the address, not just the length of the byte slice: a 16-byte slice can still be an IPv4 address.

## func IPv4

```
func IPv4(a, b, c, d byte) IP
```

IPv4 returns the IP address (in 16-byte form) of the IPv4 address a.b.c.d.

## func LookupIP

```
func LookupIP(host string) ([]IP, error)
```

LookupIP looks up host using the local resolver. It returns a slice of that host's IPv4 and IPv6 addresses.

## func ParseIP

```
func ParseIP(s string) IP
```

`ParselP` parses `s` as an IP address, returning the result. The string `s` can be in IPv4 dotted decimal ("192.0.2.1"), IPv6 ("2001:db8::68"), or IPv4-mapped IPv6 ("::ffff:192.0.2.1") form. If `s` is not a valid textual representation of an IP address, `ParselP` returns nil.

## func (IP) DefaultMask

```
func (ip IP) DefaultMask() IPMask
```

`DefaultMask` returns the default IP mask for the IP address `ip`. Only IPv4 addresses have default masks; `DefaultMask` returns nil if `ip` is not a valid IPv4 address.

## func (IP) Equal

```
func (ip IP) Equal(x IP) bool
```

`Equal` reports whether `ip` and `x` are the same IP address. An IPv4 address and that same address in IPv6 form are considered to be equal.

## func (IP) IsGlobalUnicast

```
func (ip IP) IsGlobalUnicast() bool
```

`IsGlobalUnicast` reports whether `ip` is a global unicast address.

The identification of global unicast addresses uses address type identification as defined in [RFC 1122](#), [RFC 4632](#) and [RFC 4291](#) with the exception of IPv4 directed broadcast addresses. It returns true even if `ip` is in IPv4 private address space or local IPv6 unicast address space.

## func (IP) IsInterfaceLocalMulticast

```
func (ip IP) IsInterfaceLocalMulticast() bool
```

`IsInterfaceLocalMulticast` reports whether `ip` is an interface-local multicast address.

## func (IP) IsLinkLocalMulticast

```
func (ip IP) IsLinkLocalMulticast() bool
```

`IsLinkLocalMulticast` reports whether `ip` is a link-local multicast address.

## func (IP) IsLinkLocalUnicast

```
func (ip IP) IsLinkLocalUnicast() bool
```

`IsLinkLocalUnicast` reports whether `ip` is a link-local unicast address.

## func (IP) `IsLoopback`

```
func (ip IP) IsLoopback() bool
```

`IsLoopback` reports whether ip is a loopback address.

## func (IP) `IsMulticast`

```
func (ip IP) IsMulticast() bool
```

`IsMulticast` reports whether ip is a multicast address.

## func (IP) `IsUnspecified`

```
func (ip IP) IsUnspecified() bool
```

`IsUnspecified` reports whether ip is an unspecified address, either the IPv4 address "0.0.0.0" or the IPv6 address "::".

## func (IP) `MarshalText`

```
func (ip IP) MarshalText() ([]byte, error)
```

`MarshalText` implements the `encoding.TextMarshaler` interface. The encoding is the same as returned by `String`, with one exception: When `len(ip)` is zero, it returns an empty slice.

## func (IP) `Mask`

```
func (ip IP) Mask(mask IPMask) IP
```

`Mask` returns the result of masking the IP address `ip` with `mask`.

## func (IP) `String`

```
func (ip IP) String() string
```

`String` returns the string form of the IP address `ip`. It returns one of 4 forms:

- "<nil>", if `ip` has length 0
- dotted decimal ("192.0.2.1"), if `ip` is an IPv4 or IP4-mapped IPv6 address
- IPv6 ("2001:db8::1"), if `ip` is a valid IPv6 address
- the hexadecimal form of `ip`, without punctuation, if no other cases apply

## func (IP) `To16`

```
func (ip IP) To16() IP
```

To16 converts the IP address ip to a 16-byte representation. If ip is not an IP address (it is the wrong length), To16 returns nil.

## func (IP) To4

```
func (ip IP) To4() IP
```

To4 converts the IPv4 address ip to a 4-byte representation. If ip is not an IPv4 address, To4 returns nil.

## func (\*IP) UnmarshalText

```
func (ip *IP) UnmarshalText(text []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface. The IP address is expected in a form accepted by ParseIP.

## type IPAddr

```
type IPAddr struct {
 IP IP
 Zone string // IPv6 scoped addressing zone
}
```

IPAddr represents the address of an IP end point.

## func ResolveIPAddr

```
func ResolveIPAddr(network, address string) (*IPAddr, error)
```

ResolveIPAddr returns an address of IP end point.

The network must be an IP network name.

If the host in the address parameter is not a literal IP address, ResolveIPAddr resolves the address to an address of IP end point. Otherwise, it parses the address as a literal IP address. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See func Dial for a description of the network and address parameters.

## func (\*IPAddr) Network

```
func (a *IPAddr) Network() string
```

Network returns the address's network name, "ip".

## func (\*IPAddr) String

```
func (a *IPAddr) String() string
```

## type IPConn

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

IPConn is the implementation of the Conn and PacketConn interfaces for IP network connections.

## func DialIP

```
func DialIP(network string, laddr, raddr *IPAddr) (*IPConn, error)
```

DialIP acts like Dial for IP networks.

The network must be an IP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

## func ListenIP

```
func ListenIP(network string, laddr *IPAddr) (*IPConn, error)
```

ListenIP acts like ListenPacket for IP networks.

The network must be an IP network name; see func Dial for details.

If the IP field of laddr is nil or an unspecified IP address, ListenIP listens on all available IP addresses of the local system except multicast IP addresses.

## func (\*IPConn) Close

```
func (c *IPConn) Close() error
```

Close closes the connection.

## func (\*IPConn) File

```
func (c *IPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned `os.File`'s file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*IPConn) LocalAddr

```
func (c *IPConn) LocalAddr() Addr
```

`LocalAddr` returns the local network address. The `Addr` returned is shared by all invocations of `LocalAddr`, so do not modify it.

## func (\*IPConn) Read

```
func (c *IPConn) Read(b []byte) (int, error)
```

`Read` implements the `Conn` `Read` method.

## func (\*IPConn) ReadFrom

```
func (c *IPConn) ReadFrom(b []byte) (int, Addr, error)
```

`ReadFrom` implements the `PacketConn` `ReadFrom` method.

## func (\*IPConn) ReadFromIP

```
func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
```

`ReadFromIP` acts like `ReadFrom` but returns an `IPAddr`.

## func (\*IPConn) ReadMsgIP

```
func (c *IPConn) ReadMsgIP(b, oob []byte) (n, oobn, flags int, addr *IPAddr, err error)
```

`ReadMsgIP` reads a message from `c`, copying the payload into `b` and the associated out-of-band data into `oob`. It returns the number of bytes copied into `b`, the number of bytes copied into `oob`, the flags that were set on the message and the source address of the message.

The packages `golang.org/x/net/ipv4` and `golang.org/x/net/ipv6` can be used to manipulate IP-level socket options in `oob`.

## func (\*IPConn) RemoteAddr

```
func (c *IPConn) RemoteAddr() Addr
```

`RemoteAddr` returns the remote network address. The `Addr` returned is shared by all invocations of `RemoteAddr`, so do not modify it.

## func (\*IPConn) SetDeadline

```
func (c *IPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

## func (\*IPConn) SetReadBuffer

```
func (c *IPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

## func (\*IPConn) SetReadDeadline

```
func (c *IPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

## func (\*IPConn) SetWriteBuffer

```
func (c *IPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

## func (\*IPConn) SetWriteDeadline

```
func (c *IPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

## func (\*IPConn) SyscallConn

```
func (c *IPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

## func (\*IPConn) Write

```
func (c *IPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

## func (\*IPConn) WriteMsgIP

```
func (c *IPConn) WriteMsgIP(b, oob []byte, addr *IPAddr) (n, oobn int, err error)
```

WriteMsgIP writes a message to addr via c, copying the payload from b and the associated out-of-band data from oob. It returns the number of payload and out-of-band bytes written.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

## func (\*IPConn) WriteTo

```
func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

## func (\*IPConn) WriteToIP

```
func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, error)
```

WriteToIP acts like WriteTo but takes an IPAddr.

## type IPMask

```
type IPMask []byte
```

An IPMask is a bitmask that can be used to manipulate IP addresses for IP addressing and routing.

See type IPNet and func ParseCIDR for details.

## func CIDRMask

```
func CIDRMask(ones, bits int) IPMask
```

CIDRMask returns an IPMask consisting of 'ones' 1 bits followed by 0s up to a total length of 'bits' bits. For a mask of this form, CIDRMask is the inverse of IPMask.Size.

## func IPv4Mask

```
func IPv4Mask(a, b, c, d byte) IPMask
```

IPv4Mask returns the IP mask (in 4-byte form) of the IPv4 mask a.b.c.d.

## func (IPMask) Size

```
func (m IPMask) Size() (ones, bits int)
```

Size returns the number of leading ones and total bits in the mask. If the mask is not in the canonical form--ones followed by zeros--then Size returns 0, 0.

## func (IPMask) String

```
func (m IPMask) String() string
```

String returns the hexadecimal form of m, with no punctuation.

## type IPNet

```
type IPNet struct {
 IP IP // network number
 Mask IPMask // network mask
}
```

An IPNet represents an IP network.

## func (\*IPNet) Contains

```
func (n *IPNet) Contains(ip IP) bool
```

Contains reports whether the network includes ip.

## func (\*IPNet) Network

```
func (n *IPNet) Network() string
```

Network returns the address's network name, "ip+net".

## func (\*IPNet) String

```
func (n *IPNet) String() string
```

String returns the CIDR notation of n like "192.0.2.0/24" or "2001:db8::/48" as defined in [RFC 4632](#) and [RFC 4291](#). If the mask is not in the canonical form, it returns the string which consists of an IP address, followed by a slash character and a mask expressed as hexadecimal form with no punctuation like "198.51.100.0/c000ff00".

## type Interface

```
type Interface struct {
 Index int // positive integer that starts at one, zero is never used
 MTU int // maximum transmission unit
 Name string // e.g., "en0", "lo0", "eth0.100"
 HardwareAddr HardwareAddr // IEEE MAC-48, EUI-48 and EUI-64 form
 Flags Flags // e.g., FlagUp, FlagLoopback, FlagMulticast
}
```

Interface represents a mapping between network interface name and index. It also represents network interface facility information.

## func `InterfaceByIndex`

```
func InterfaceByIndex(index int) (*Interface, error)
```

InterfaceByIndex returns the interface specified by index.

On Solaris, it returns one of the logical network interfaces sharing the logical data link; for more precision use InterfaceByName.

## func `InterfaceByName`

```
func InterfaceByName(name string) (*Interface, error)
```

InterfaceByName returns the interface specified by name.

## func `Interfaces`

```
func Interfaces() ([]Interface, error)
```

Interfaces returns a list of the system's network interfaces.

## func `(*Interface) Addrs`

```
func (ifi *Interface) Addrs() ([]Addr, error)
```

Addrs returns a list of unicast interface addresses for a specific interface.

## func `(*Interface) MulticastAddrs`

```
func (ifi *Interface) MulticastAddrs() ([]Addr, error)
```

MulticastAddrs returns a list of multicast, joined group addresses for a specific interface.

## type `InvalidAddrError`

```
type InvalidAddrError string
```

## func `(InvalidAddrError) Error`

```
func (e InvalidAddrError) Error() string
```

## func `(InvalidAddrError) Temporary`

```
func (e InvalidAddrError) Temporary() bool
```

## func (InvalidAddrError) [Timeout](#)

```
func (e InvalidAddrError) Timeout() bool
```

## type [ListenConfig](#)

```
type ListenConfig struct {
 // If Control is not nil, it is called after creating the network
 // connection but before binding it to the operating system.
 //
 // Network and address parameters passed to Control method are not
 // necessarily the ones passed to Listen. For example, passing "tcp" to
 // Listen will cause the Control function to be called with "tcp4" or "tcp6".
 Control func(network, address string, c syscall.RawConn) error

 // KeepAlive specifies the keep-alive period for network
 // connections accepted by this listener.
 // If zero, keep-alives are enabled if supported by the protocol
 // and operating system. Network protocols or operating systems
 // that do not support keep-alives ignore this field.
 // If negative, keep-alives are disabled.
 KeepAlive time.Duration
}
```

ListenConfig contains options for listening to an address.

## func (\*ListenConfig) [Listen](#)

```
func (lc *ListenConfig) Listen(ctx context.Context, network, address string) (Listene
```

Listen announces on the local network address.

See func Listen for a description of the network and address parameters.

## func (\*ListenConfig) [ListenPacket](#)

```
func (lc *ListenConfig) ListenPacket(ctx context.Context, network, address string) (P
```

ListenPacket announces on the local network address.

See func ListenPacket for a description of the network and address parameters.

## type [Listener](#)

```
type Listener interface {
 // Accept waits for and returns the next connection to the listener.
```

```
Accept() (Conn, error)

 // Close closes the listener.
 // Any blocked Accept operations will be unblocked and return errors.
 Close() error

 // Addr returns the listener's network address.
 Addr() Addr
}
```

A Listener is a generic network listener for stream-oriented protocols.

Multiple goroutines may invoke methods on a Listener simultaneously.

## func [FileListener](#)

```
func FileListener(f *os.File) (ln Listener, err error)
```

FileListener returns a copy of the network listener corresponding to the open file f. It is the caller's responsibility to close ln when finished. Closing ln does not affect f, and closing f does not affect ln.

## func [Listen](#)

```
func Listen(network, address string) (Listener, error)
```

Listen announces on the local network address.

The network must be "tcp", "tcp4", "tcp6", "unix" or "unixpacket".

For TCP networks, if the host in the address parameter is empty or a literal unspecified IP address, Listen listens on all available unicast and anycast IP addresses of the local system. To only use IPv4, use network "tcp4". The address can use a host name, but this is not recommended, because it will create a listener for at most one of the host's IP addresses. If the port in the address parameter is empty or "0", as in "127.0.0.1:" or "[::1]:0", a port number is automatically chosen. The Addr method of Listener can be used to discover the chosen port.

See func Dial for a description of the network and address parameters.

## type [MX](#)

```
type MX struct {
 Host string
 Pref uint16
}
```

An MX represents a single DNS MX record.

## func [LookupMX](#)

```
func LookupMX(name string) ([]*MX, error)
```

LookupMX returns the DNS MX records for the given domain name sorted by preference.

## type NS

```
type NS struct {
 Host string
}
```

An NS represents a single DNS NS record.

## func LookupNS

```
func LookupNS(name string) ([]*NS, error)
```

LookupNS returns the DNS NS records for the given domain name.

## type OpError

```
type OpError struct {
 // Op is the operation which caused the error, such as
 // "read" or "write".
 Op string

 // Net is the network type on which this error occurred,
 // such as "tcp" or "udp6".
 Net string

 // For operations involving a remote network connection, like
 // Dial, Read, or Write, Source is the corresponding local
 // network address.
 Source Addr

 // Addr is the network address for which this error occurred.
 // For local operations, like Listen or SetDeadline, Addr is
 // the address of the local endpoint being manipulated.
 // For operations involving a remote network connection, like
 // Dial, Read, or Write, Addr is the remote address of that
 // connection.
 Addr Addr

 // Err is the error that occurred during the operation.
 // The Error method panics if the error is nil.
 Err error
}
```

OpError is the error type usually returned by functions in the net package. It describes the operation, network type, and address of an error.

## func (\*OpError) Error

```
func (e *OpError) Error() string
```

## func (\*OpError) Temporary

```
func (e *OpError) Temporary() bool
```

## func (\*OpError) Timeout

```
func (e *OpError) Timeout() bool
```

## func (\*OpError) Unwrap

```
func (e *OpError) Unwrap() error
```

## type PacketConn

```
type PacketConn interface {
 // ReadFrom reads a packet from the connection,
 // copying the payload into p. It returns the number of
 // bytes copied into p and the return address that
 // was on the packet.
 // It returns the number of bytes read (0 <= n <= len(p))
 // and any error encountered. Callers should always process
 // the n > 0 bytes returned before considering the error err.
 // ReadFrom can be made to time out and return an error after a
 // fixed time limit; see SetDeadline and SetReadDeadline.
 ReadFrom(p []byte) (n int, addr Addr, err error)

 // WriteTo writes a packet with payload p to addr.
 // WriteTo can be made to time out and return an Error after a
 // fixed time limit; see SetDeadline and SetWriteDeadline.
 // On packet-oriented connections, write timeouts are rare.
 WriteTo(p []byte, addr Addr) (n int, err error)

 // Close closes the connection.
 // Any blocked ReadFrom or WriteTo operations will be unblocked and return errors
 Close() error

 // LocalAddr returns the local network address.
 LocalAddr() Addr

 // SetDeadline sets the read and write deadlines associated
 // with the connection. It is equivalent to calling both
 // SetReadDeadline and SetWriteDeadline.
 //
 // A deadline is an absolute time after which I/O operations
 // fail instead of blocking. The deadline applies to all future
```

```

// and pending I/O, not just the immediately following call to
// Read or Write. After a deadline has been exceeded, the
// connection can be refreshed by setting a deadline in the future.
//
// If the deadline is exceeded a call to Read or Write or to other
// I/O methods will return an error that wraps os.ErrDeadlineExceeded.
// This can be tested using errors.Is(err, os.ErrDeadlineExceeded).
// The error's Timeout method will return true, but note that there
// are other possible errors for which the Timeout method will
// return true even if the deadline has not been exceeded.
//
// An idle timeout can be implemented by repeatedly extending
// the deadline after successful ReadFrom or WriteTo calls.
//
// A zero value for t means I/O operations will not time out.
SetDeadline(t time.Time) error

// SetReadDeadline sets the deadline for future ReadFrom calls
// and any currently-blocked ReadFrom call.
// A zero value for t means ReadFrom will not time out.
SetReadDeadline(t time.Time) error

// SetWriteDeadline sets the deadline for future WriteTo calls
// and any currently-blocked WriteTo call.
// Even if write times out, it may return n > 0, indicating that
// some of the data was successfully written.
// A zero value for t means WriteTo will not time out.
SetWriteDeadline(t time.Time) error
}

```

PacketConn is a generic packet-oriented network connection.

Multiple goroutines may invoke methods on a PacketConn simultaneously.

## func **FilePacketConn**

```
func FilePacketConn(f *os.File) (c PacketConn, err error)
```

FilePacketConn returns a copy of the packet network connection corresponding to the open file f. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

## func **ListenPacket**

```
func ListenPacket(network, address string) (PacketConn, error)
```

ListenPacket announces on the local network address.

The network must be "udp", "udp4", "udp6", "unixgram", or an IP transport. The IP transports are "ip", "ip4", or "ip6" followed by a colon and a literal protocol number or a protocol name, as in "ip:1" or

"ip:icmp".

For UDP and IP networks, if the host in the address parameter is empty or a literal unspecified IP address, ListenPacket listens on all available IP addresses of the local system except multicast IP addresses. To only use IPv4, use network "udp4" or "ip4:proto". The address can use a host name, but this is not recommended, because it will create a listener for at most one of the host's IP addresses. If the port in the address parameter is empty or "0", as in "127.0.0.1:" or "[::1]:0", a port number is automatically chosen. The LocalAddr method of PacketConn can be used to discover the chosen port.

See func Dial for a description of the network and address parameters.

## type ParseError

```
type ParseError struct {
 // Type is the type of string that was expected, such as
 // "IP address", "CIDR address".
 Type string

 // Text is the malformed text string.
 Text string
}
```

A ParseError is the error type of literal network address parsers.

## func (\*ParseError) Error

```
func (e *ParseError) Error() string
```

## type Resolver

```
type Resolver struct {
 // PreferGo controls whether Go's built-in DNS resolver is preferred
 // on platforms where it's available. It is equivalent to setting
 // GODEBUG=netdns=go, but scoped to just this resolver.
 PreferGo bool

 // StrictErrors controls the behavior of temporary errors
 // (including timeout, socket errors, and SERVFAIL) when using
 // Go's built-in resolver. For a query composed of multiple
 // sub-queries (such as an A+AAAA address lookup, or walking the
 // DNS search list), this option causes such errors to abort the
 // whole query instead of returning a partial result. This is
 // not enabled by default because it may affect compatibility
 // with resolvers that process AAAA queries incorrectly.
 StrictErrors bool

 // Dial optionally specifies an alternate dialer for use by
 // Go's built-in DNS resolver to make TCP and UDP connections
 // to DNS services. The host in the address parameter will
 // always be a literal IP address and not a host name, and the
```

```

// port in the address parameter will be a literal port number
// and not a service name.
// If the Conn returned is also a PacketConn, sent and received DNS
// messages must adhere to RFC 1035 section 4.2.1, "UDP usage".
// Otherwise, DNS messages transmitted over Conn must adhere
// to RFC 7766 section 5, "Transport Protocol Selection".
// If nil, the default dialer is used.
Dial func(ctx context.Context, network, address string) (Conn, error)
// contains filtered or unexported fields
}

```

A Resolver looks up names and numbers.

A nil \*Resolver is equivalent to a zero Resolver.

## func (\*Resolver) LookupAddr

```
func (r *Resolver) LookupAddr(ctx context.Context, addr string) (names []string, err
```

LookupAddr performs a reverse lookup for the given address, returning a list of names mapping to that address.

## func (\*Resolver) LookupCNAME

```
func (r *Resolver) LookupCNAME(ctx context.Context, host string) (cname string, err
```

LookupCNAME returns the canonical name for the given host. Callers that do not care about the canonical name can call LookupHost or LookupIP directly; both take care of resolving the canonical name as part of the lookup.

A canonical name is the final name after following zero or more CNAME records. LookupCNAME does not return an error if host does not contain DNS "CNAME" records, as long as host resolves to address records.

## func (\*Resolver) LookupHost

```
func (r *Resolver) LookupHost(ctx context.Context, host string) (addrs []string, err
```

LookupHost looks up the given host using the local resolver. It returns a slice of that host's addresses.

## func (\*Resolver) LookupIP

```
func (r *Resolver) LookupIP(ctx context.Context, network, host string) ([]IP, error)
```

LookupIP looks up host for the given network using the local resolver. It returns a slice of that host's IP addresses of the type specified by network. network must be one of "ip", "ip4" or "ip6".

## func (\*Resolver) LookupIPAddr

```
func (r *Resolver) LookupIPAddr(ctx context.Context, host string) ([]IPAddr, error)
```

LookupIPAddr looks up host using the local resolver. It returns a slice of that host's IPv4 and IPv6 addresses.

## func (\*Resolver) LookupMX

```
func (r *Resolver) LookupMX(ctx context.Context, name string) ([]*MX, error)
```

LookupMX returns the DNS MX records for the given domain name sorted by preference.

## func (\*Resolver) LookupNS

```
func (r *Resolver) LookupNS(ctx context.Context, name string) ([]*NS, error)
```

LookupNS returns the DNS NS records for the given domain name.

## func (\*Resolver) LookupPort

```
func (r *Resolver) LookupPort(ctx context.Context, network, service string) (port int, err error)
```

LookupPort looks up the port for the given network and service.

## func (\*Resolver) LookupSRV

```
func (r *Resolver) LookupSRV(ctx context.Context, service, proto, name string) (cname string, target string, port int, weight int, priority int, err error)
```

LookupSRV tries to resolve an SRV query of the given service, protocol, and domain name. The proto is "tcp" or "udp". The returned records are sorted by priority and randomized by weight within a priority.

LookupSRV constructs the DNS name to look up following [RFC 2782](#). That is, it looks up \_service.\_proto.name. To accommodate services publishing SRV records under non-standard names, if both service and proto are empty strings, LookupSRV looks up name directly.

## func (\*Resolver) LookupTXT

```
func (r *Resolver) LookupTXT(ctx context.Context, name string) ([]string, error)
```

LookupTXT returns the DNS TXT records for the given domain name.

## type SRV

```
type SRV struct {
 Target string
```

```
Port uint16
Priority uint16
Weight uint16
}
```

An SRV represents a single DNS SRV record.

## func `LookupSRV`

```
func LookupSRV(service, proto, name string) (cname string, addrs []*SRV, err error)
```

`LookupSRV` tries to resolve an SRV query of the given service, protocol, and domain name. The proto is "tcp" or "udp". The returned records are sorted by priority and randomized by weight within a priority.

`LookupSRV` constructs the DNS name to look up following [RFC 2782](#). That is, it looks up `_service._proto.name`. To accommodate services publishing SRV records under non-standard names, if both service and proto are empty strings, `LookupSRV` looks up name directly.

## type `TCPAddr`

```
type TCPAddr struct {
 IP IP
 Port int
 Zone string // IPv6 scoped addressing zone
}
```

`TCPAddr` represents the address of a TCP end point.

## func `ResolveTCPAddr`

```
func ResolveTCPAddr(network, address string) (*TCPAddr, error)
```

`ResolveTCPAddr` returns an address of TCP end point.

The network must be a TCP network name.

If the host in the address parameter is not a literal IP address or the port is not a literal port number, `ResolveTCPAddr` resolves the address to an address of TCP end point. Otherwise, it parses the address as a pair of literal IP address and port number. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See `func Dial` for a description of the network and address parameters.

## func `(*TCPAddr)` `Network`

```
func (a *TCPAddr) Network() string
```

`Network` returns the address's network name, "tcp".

## func (\*TCPAddr) String

```
func (a *TCPAddr) String() string
```

## type TCPConn

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

TCPConn is an implementation of the Conn interface for TCP network connections.

## func DialTCP

```
func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

DialTCP acts like Dial for TCP networks.

The network must be a TCP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

## func (\*TCPConn) Close

```
func (c *TCPConn) Close() error
```

Close closes the connection.

## func (\*TCPConn) CloseRead

```
func (c *TCPConn) CloseRead() error
```

CloseRead shuts down the reading side of the TCP connection. Most callers should just use Close.

## func (\*TCPConn) CloseWrite

```
func (c *TCPConn) CloseWrite() error
```

CloseWrite shuts down the writing side of the TCP connection. Most callers should just use Close.

## func (\*TCPConn) File

```
func (c *TCPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*TCPConn) LocalAddr

```
func (c *TCPConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

## func (\*TCPConn) Read

```
func (c *TCPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

## func (\*TCPConn) ReadFrom

```
func (c *TCPConn) ReadFrom(r io.Reader) (int64, error)
```

ReadFrom implements the io.ReaderFrom ReadFrom method.

## func (\*TCPConn) RemoteAddr

```
func (c *TCPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

## func (\*TCPConn) SetDeadline

```
func (c *TCPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

## func (\*TCPConn) SetKeepAlive

```
func (c *TCPConn) SetKeepAlive(keepalive bool) error
```

SetKeepAlive sets whether the operating system should send keep-alive messages on the connection.

## func (\*TCPConn) SetKeepAlivePeriod

```
func (c *TCPConn) SetKeepAlivePeriod(d time.Duration) error
```

SetKeepAlivePeriod sets period between keep-alives.

## func (\*TCPConn) `SetLinger`

```
func (c *TCPConn) SetLinger(sec int) error
```

SetLinger sets the behavior of Close on a connection which still has data waiting to be sent or to be acknowledged.

If sec < 0 (the default), the operating system finishes sending the data in the background.

If sec == 0, the operating system discards any unsent or unacknowledged data.

If sec > 0, the data is sent in the background as with sec < 0. On some operating systems after sec seconds have elapsed any remaining unsent data may be discarded.

## func (\*TCPConn) `SetNoDelay`

```
func (c *TCPConn) SetNoDelay(noDelay bool) error
```

SetNoDelay controls whether the operating system should delay packet transmission in hopes of sending fewer packets (Nagle's algorithm). The default is true (no delay), meaning that data is sent as soon as possible after a Write.

## func (\*TCPConn) `SetReadBuffer`

```
func (c *TCPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

## func (\*TCPConn) `SetReadDeadline`

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

## func (\*TCPConn) `SetWriteBuffer`

```
func (c *TCPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

## func (\*TCPConn) `SetWriteDeadline`

```
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

## func (\*TCPConn) [SyscallConn](#)

```
func (c *TCPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the [syscall.Conn](#) interface.

## func (\*TCPConn) [Write](#)

```
func (c *TCPConn) Write(b \[\]byte) (int, error)
```

Write implements the Conn Write method.

## type [TCPListener](#)

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

TCPListener is a TCP network listener. Clients should typically use variables of type Listener instead of assuming TCP.

## func [ListenTCP](#)

```
func ListenTCP(network string, laddr *TCPAddr) (*TCPListener, error)
```

ListenTCP acts like Listen for TCP networks.

The network must be a TCP network name; see func Dial for details.

If the IP field of laddr is nil or an unspecified IP address, ListenTCP listens on all available unicast and anycast IP addresses of the local system. If the Port field of laddr is 0, a port number is automatically chosen.

## func (\*TCPListener) [Accept](#)

```
func (l *TCPListener) Accept() (Conn, error)
```

Accept implements the Accept method in the Listener interface; it waits for the next call and returns a generic Conn.

## func (\*TCPListener) [AcceptTCP](#)

```
func (l *TCPListener) AcceptTCP() (*TCPConn, error)
```

AcceptTCP accepts the next incoming call and returns the new connection.

## func (\*TCPListener) Addr

```
func (l *TCPListener) Addr() Addr
```

Addr returns the listener's network address, a \*TCPAddr. The Addr returned is shared by all invocations of Addr, so do not modify it.

## func (\*TCPListener) Close

```
func (l *TCPListener) Close() error
```

Close stops listening on the TCP address. Already Accepted connections are not closed.

## func (\*TCPListener) File

```
func (l *TCPListener) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing l does not affect f, and closing f does not affect l.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*TCPListener) SetDeadline

```
func (l *TCPListener) SetDeadline(t time.Time) error
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

## func (\*TCPListener) SyscallConn

```
func (l *TCPListener) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

The returned RawConn only supports calling Control. Read and Write return an error.

## type UDPAddr

```
type UDPAddr struct {
 IP IP
 Port int}
```

```
 Zone string // IPv6 scoped addressing zone
}
```

UDPAddr represents the address of a UDP end point.

## func **ResolveUDPAddr**

```
func ResolveUDPAddr(network, address string) (*UDPAddr, error)
```

ResolveUDPAddr returns an address of UDP end point.

The network must be a UDP network name.

If the host in the address parameter is not a literal IP address or the port is not a literal port number, ResolveUDPAddr resolves the address to an address of UDP end point. Otherwise, it parses the address as a pair of literal IP address and port number. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See func Dial for a description of the network and address parameters.

## func (\***UDPAddr**) **Network**

```
func (a *UDPAddr) Network() string
```

Network returns the address's network name, "udp".

## func (\***UDPAddr**) **String**

```
func (a *UDPAddr) String() string
```

## type **UDPConn**

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

UDPConn is the implementation of the Conn and PacketConn interfaces for UDP network connections.

## func **DialUDP**

```
func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialUDP acts like Dial for UDP networks.

The network must be a UDP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

## func `ListenMulticastUDP`

```
func ListenMulticastUDP(network string, ifi *Interface, gaddr *UDPAddr) (*UDPConn, error)
```

`ListenMulticastUDP` acts like `ListenPacket` for UDP networks but takes a group address on a specific network interface.

The network must be a UDP network name; see `func Dial` for details.

`ListenMulticastUDP` listens on all available IP addresses of the local system including the group, multicast IP address. If `ifi` is nil, `ListenMulticastUDP` uses the system-assigned multicast interface, although this is not recommended because the assignment depends on platforms and sometimes it might require routing configuration. If the `Port` field of `gaddr` is 0, a port number is automatically chosen.

`ListenMulticastUDP` is just for convenience of simple, small applications. There are `golang.org/x/net/ipv4` and `golang.org/x/net/ipv6` packages for general purpose uses.

## func `ListenUDP`

```
func ListenUDP(network string, laddr *UDPAddr) (*UDPConn, error)
```

`ListenUDP` acts like `ListenPacket` for UDP networks.

The network must be a UDP network name; see `func Dial` for details.

If the IP field of `laddr` is nil or an unspecified IP address, `ListenUDP` listens on all available IP addresses of the local system except multicast IP addresses. If the `Port` field of `laddr` is 0, a port number is automatically chosen.

## func (`*UDPConn`) `Close`

```
func (c *UDPConn) Close() error
```

`Close` closes the connection.

## func (`*UDPConn`) `File`

```
func (c *UDPConn) File() (f *os.File, err error)
```

`File` returns a copy of the underlying `os.File`. It is the caller's responsibility to close `f` when finished. Closing `c` does not affect `f`, and closing `f` does not affect `c`.

The returned `os.File`'s file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*UDPConn) LocalAddr

```
func (c *UDPConn) LocalAddr() Addr
```

`LocalAddr` returns the local network address. The `Addr` returned is shared by all invocations of `LocalAddr`, so do not modify it.

## func (\*UDPConn) Read

```
func (c *UDPConn) Read(b []byte) (int, error)
```

`Read` implements the `Conn` `Read` method.

## func (\*UDPConn) ReadFrom

```
func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
```

`ReadFrom` implements the `PacketConn` `ReadFrom` method.

## func (\*UDPConn) ReadFromUDP

```
func (c *UDPConn) ReadFromUDP(b []byte) (int, *UDPAddr, error)
```

`ReadFromUDP` acts like `ReadFrom` but returns a `UDPAddr`.

## func (\*UDPConn) ReadMsgUDP

```
func (c *UDPConn) ReadMsgUDP(b, oob []byte) (n, oobn, flags int, addr *UDPAddr, err error)
```

`ReadMsgUDP` reads a message from `c`, copying the payload into `b` and the associated out-of-band data into `oob`. It returns the number of bytes copied into `b`, the number of bytes copied into `oob`, the flags that were set on the message and the source address of the message.

The packages `golang.org/x/net/ipv4` and `golang.org/x/net/ipv6` can be used to manipulate IP-level socket options in `oob`.

## func (\*UDPConn) RemoteAddr

```
func (c *UDPConn) RemoteAddr() Addr
```

`RemoteAddr` returns the remote network address. The `Addr` returned is shared by all invocations of `RemoteAddr`, so do not modify it.

## func (\*UDPConn) SetDeadline

```
func (c *UDPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

## func (\*UDPConn) SetReadBuffer

```
func (c *UDPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

## func (\*UDPConn) SetReadDeadline

```
func (c *UDPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

## func (\*UDPConn) SetWriteBuffer

```
func (c *UDPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

## func (\*UDPConn) SetWriteDeadline

```
func (c *UDPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

## func (\*UDPConn) SyscallConn

```
func (c *UDPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the [syscall.Conn](#) interface.

## func (\*UDPConn) Write

```
func (c *UDPConn) Write(b \[\]byte) (int, error)
```

Write implements the Conn Write method.

## func (\*UDPConn) WriteMsgUDP

```
func (c *UDPConn) WriteMsgUDP(b, oob \[\]byte, addr *UDPAddr) (n, oobn int, err error)
```

WriteMsgUDP writes a message to addr via c if c isn't connected, or to c's remote address if c is connected (in which case addr must be nil). The payload is copied from b and the associated out-of-band data is copied from oob. It returns the number of payload and out-of-band bytes written.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

## func (\*UDPConn) WriteTo

```
func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

## func (\*UDPConn) WriteToUDP

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP acts like WriteTo but takes a UDPAddr.

## type UnixAddr

```
type UnixAddr struct {
 Name string
 Net string
}
```

UnixAddr represents the address of a Unix domain socket end point.

## func ResolveUnixAddr

```
func ResolveUnixAddr(network, address string) (*UnixAddr, error)
```

ResolveUnixAddr returns an address of Unix domain socket end point.

The network must be a Unix network name.

See func Dial for a description of the network and address parameters.

## func (\*UnixAddr) Network

```
func (a *UnixAddr) Network() string
```

Network returns the address's network name, "unix", "unixgram" or "unixpacket".

## func (\*UnixAddr) String

```
func (a *UnixAddr) String() string
```

## type UnixConn

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

UnixConn is an implementation of the Conn interface for connections to Unix domain sockets.

## func DialUnix

```
func DialUnix(network string, laddr, raddr *UnixAddr) (*UnixConn, error)
```

DialUnix acts like Dial for Unix networks.

The network must be a Unix network name; see func Dial for details.

If laddr is non-nil, it is used as the local address for the connection.

## func ListenUnixgram

```
func ListenUnixgram(network string, laddr *UnixAddr) (*UnixConn, error)
```

ListenUnixgram acts like ListenPacket for Unix networks.

The network must be "unixgram".

## func (\*UnixConn) Close

```
func (c *UnixConn) Close() error
```

Close closes the connection.

## func (\*UnixConn) CloseRead

```
func (c *UnixConn) CloseRead() error
```

CloseRead shuts down the reading side of the Unix domain connection. Most callers should just use Close.

## func (\*UnixConn) CloseWrite

```
func (c *UnixConn) CloseWrite() error
```

CloseWrite shuts down the writing side of the Unix domain connection. Most callers should just use Close.

## func (\*UnixConn) File

```
func (c *UnixConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*UnixConn) LocalAddr

```
func (c *UnixConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

## func (\*UnixConn) Read

```
func (c *UnixConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

## func (\*UnixConn) ReadFrom

```
func (c *UnixConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

## func (\*UnixConn) ReadFromUnix

```
func (c *UnixConn) ReadFromUnix(b []byte) (int, *UnixAddr, error)
```

ReadFromUnix acts like ReadFrom but returns a UnixAddr.

## func (\*UnixConn) ReadMsgUnix

```
func (c *UnixConn) ReadMsgUnix(b, oob []byte) (n, oobn, flags int, addr *UnixAddr, er
```

ReadMsgUnix reads a message from c, copying the payload into b and the associated out-of-band data into oob. It returns the number of bytes copied into b, the number of bytes copied into oob, the flags that were set on the message and the source address of the message.

Note that if len(b) == 0 and len(oob) > 0, this function will still read (and discard) 1 byte from the connection.

## func (\*UnixConn) [RemoteAddr](#)

```
func (c *UnixConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

## func (\*UnixConn) [SetDeadline](#)

```
func (c *UnixConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

## func (\*UnixConn) [SetReadBuffer](#)

```
func (c *UnixConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

## func (\*UnixConn) [SetReadDeadline](#)

```
func (c *UnixConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

## func (\*UnixConn) [SetWriteBuffer](#)

```
func (c *UnixConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

## func (\*UnixConn) [SetWriteDeadline](#)

```
func (c *UnixConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

## func (\*UnixConn) [SyscallConn](#)

```
func (c *UnixConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

## func (\*UnixConn) [Write](#)

```
func (c *UnixConn) Write(b \[\]byte) (int, error)
```

Write implements the Conn Write method.

## func (\*UnixConn) [WriteMsgUnix](#)

```
func (c *UnixConn) WriteMsgUnix(b, oob []byte, addr *UnixAddr) (n, oobn int, err error)
```

WriteMsgUnix writes a message to addr via c, copying the payload from b and the associated out-of-band data from oob. It returns the number of payload and out-of-band bytes written.

Note that if len(b) == 0 and len(oob) > 0, this function will still write 1 byte to the connection.

## func (\*UnixConn) [WriteTo](#)

```
func (c *UnixConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

## func (\*UnixConn) [WriteToUnix](#)

```
func (c *UnixConn) WriteToUnix(b []byte, addr *UnixAddr) (int, error)
```

WriteToUnix acts like WriteTo but takes a UnixAddr.

## type [UnixListener](#)

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

UnixListener is a Unix domain socket listener. Clients should typically use variables of type Listener instead of assuming Unix domain sockets.

## func [ListenUnix](#)

```
func ListenUnix(network string, laddr *UnixAddr) (*UnixListener, error)
```

ListenUnix acts like Listen for Unix networks.

The network must be "unix" or "unixpacket".

## func (\*UnixListener) [Accept](#)

```
func (l *UnixListener) Accept() (Conn, error)
```

Accept implements the Accept method in the Listener interface. Returned connections will be of type \*UnixConn.

## func (\*UnixListener) AcceptUnix

```
func (l *UnixListener) AcceptUnix() (*UnixConn, error)
```

AcceptUnix accepts the next incoming call and returns the new connection.

## func (\*UnixListener) Addr

```
func (l *UnixListener) Addr() Addr
```

Addr returns the listener's network address. The Addr returned is shared by all invocations of Addr, so do not modify it.

## func (\*UnixListener) Close

```
func (l *UnixListener) Close() error
```

Close stops listening on the Unix address. Already accepted connections are not closed.

## func (\*UnixListener) File

```
func (l *UnixListener) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing l does not affect f, and closing f does not affect l.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

## func (\*UnixListener) SetDeadline

```
func (l *UnixListener) SetDeadline(t time.Time) error
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

## func (\*UnixListener) SetUnlinkOnClose

```
func (l *UnixListener) SetUnlinkOnClose(unlink bool)
```

SetUnlinkOnClose sets whether the underlying socket file should be removed from the file system when the listener is closed.

The default behavior is to unlink the socket file only when package net created it. That is, when the listener and the underlying socket file were created by a call to Listen or ListenUnix, then by default closing the listener will remove the socket file. but if the listener was created by a call to FileListener to use an already existing socket file, then by default closing the listener will not remove the socket file.

## func (\*UnixListener) SyscallConn

```
func (l *UnixListener) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

The returned RawConn only supports calling Control. Read and Write return an error.

## type UnknownNetworkError

```
type UnknownNetworkError string
```

## func (UnknownNetworkError) Error

```
func (e UnknownNetworkError) Error() string
```

## func (UnknownNetworkError) Temporary

```
func (e UnknownNetworkError) Temporary() bool
```

## func (UnknownNetworkError) Timeout

```
func (e UnknownNetworkError) Timeout() bool
```

## BUGs

- On JS and Windows, the FileConn, FileListener and FilePacketConn functions are not implemented.
- On JS, methods and functions related to Interface are not implemented.
- On AIX, DragonFly BSD, NetBSD, OpenBSD, Plan 9 and Solaris, the MulticastAddrs method of Interface is not implemented.
- On every POSIX platform, reads from the "ip4" network using the ReadFrom or ReadFromIP method might not return a complete IPv4 packet, including its header, even if there is space available. This can occur even in cases where Read or ReadMsgIP could return a complete packet. For this reason, it is recommended that you do not use these methods if it is important to receive a full packet.

The Go 1 compatibility guidelines make it impossible for us to change the behavior of these methods; use Read or ReadMsgIP instead.

- On JS and Plan 9, methods and functions related to IPConn are not implemented.
- On Windows, the File method of IPConn is not implemented.
- On DragonFly BSD and OpenBSD, listening on the "tcp" and "udp" networks does not listen for both IPv4 and IPv6 connections. This is due to the fact that IPv4 traffic will not be routed to an IPv6 socket - two separate sockets are required if both address families are to be supported. See `inet6(4)` for details.
- On Windows, the Write method of `syscall.RawConn` does not integrate with the runtime's network poller. It cannot wait for the connection to become writeable, and does not respect deadlines. If the user-provided callback returns false, the Write method will fail immediately.
- On JS and Plan 9, the Control, Read and Write methods of `syscall.RawConn` are not implemented.
- On JS and Windows, the File method of TCPConn and TCPLListener is not implemented.
- On Plan 9, the `ReadMsgUDP` and `WriteMsgUDP` methods of UDPConn are not implemented.
- On Windows, the File method of UDPConn is not implemented.
- On JS, methods and functions related to UDPConn are not implemented.
- On JS and Plan 9, methods and functions related to UnixConn and UnixListener are not implemented.
- On Windows, methods and functions related to UnixConn and UnixListener don't work for "unixgram" and "unixpacket".

# Package rpc

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package rpc provides access to the exported methods of an object across a network or other I/O connection. A server registers an object, making it visible as a service with the name of the type of the object. After registration, exported methods of the object will be accessible remotely. A server may register multiple objects (services) of different types but it is an error to register multiple objects of the same type.

Only methods that satisfy these criteria will be made available for remote access; other methods will be ignored:

- the method's type is exported.
- the method is exported.
- the method has two arguments, both exported (or builtin) types.
- the method's second argument is a pointer.
- the method has return type error.

In effect, the method must look schematically like

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

where T1 and T2 can be marshaled by encoding/gob. These requirements apply even if a different codec is used. (In the future, these requirements may soften for custom codecs.)

The method's first argument represents the arguments provided by the caller; the second argument represents the result parameters to be returned to the caller. The method's return value, if non-nil, is passed back as a string that the client sees as if created by errors.New. If an error is returned, the reply parameter will not be sent back to the client.

The server may handle requests on a single connection by calling ServeConn. More typically it will create a network listener and call Accept or, for an HTTP listener, HandleHTTP and http.Serve.

A client wishing to use the service establishes a connection and then invokes NewClient on the connection. The convenience function Dial (DialHTTP) performs both steps for a raw network connection (an HTTP connection). The resulting Client object has two methods, Call and Go, that specify the service and method to call, a pointer containing the arguments, and a pointer to receive the result parameters.

The Call method waits for the remote call to complete while the Go method launches the call asynchronously and signals completion using the Call structure's Done channel.

Unless an explicit codec is set up, package encoding/gob is used to transport the data.

Here is a simple example. A server wishes to export an object of type Arith:

```
package server

import "errors"

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
 *reply = args.A * args.B
 return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
 if args.B == 0 {
 return errors.New("divide by zero")
 }
 quo.Quo = args.A / args.B
 quo.Rem = args.A % args.B
 return nil
}
```

The server calls (for HTTP service):

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
 log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

At this point, clients can see a service "Arith" with methods "Arith.Multiply" and "Arith.Divide". To invoke one, a client first dials the server:

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
 log.Fatal("dialing:", err)
}
```

Then it can make a remote call:

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
 log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

or

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done // will be equal to divCall
// check errors, print, etc.
```

A server implementation will often provide a simple, type-safe wrapper for the client.

The net/rpc package is frozen and is not accepting new features.

## Constants

```
const (
 // Defaults used by HandleHTTP
 DefaultRPCPath = "/_goRPC_"
 DefaultDebugPath = "/debug/rpc"
)
```

## Variables

```
var DefaultServer = NewServer\(\)
```

DefaultServer is the default instance of `*Server`.

```
var ErrShutdown = errors.New\("connection is shut down"\)
```

## func [Accept](#)

```
func Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests to DefaultServer for each incoming connection. Accept blocks; the caller typically invokes it in a go statement.

## func [HandleHTTP](#)

```
func HandleHTTP()
```

HandleHTTP registers an HTTP handler for RPC messages to DefaultServer on DefaultRPCPath and a debugging handler on DefaultDebugPath. It is still necessary to invoke `http.Serve()`, typically in a `go` statement.

## func Register

```
func Register(rcvr interface{}) error
```

Register publishes the receiver's methods in the DefaultServer.

## func RegisterName

```
func RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

## func ServeCodec

```
func ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

## func ServeConn

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the DefaultServer on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a `go` statement. ServeConn uses the gob wire format (see package `gob`) on the connection. To use an alternate codec, use ServeCodec. See NewClient's comment for information about concurrent access.

## func ServeRequest

```
func ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

## type Call

```
type Call struct {
 ServiceMethod string // The name of the service and method to call.
 Args interface{} // The argument to the function (*struct).
 Reply interface{} // The reply from the function (*struct).
 Error error // After completion, the error status.
 Done chan *Call // Receives *Call when Go is complete.
}
```

Call represents an active RPC.

## type Client

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

Client represents an RPC Client. There may be multiple outstanding Calls associated with a single Client, and a Client may be used by multiple goroutines simultaneously.

## func Dial

```
func Dial(network, address string) (*Client, error)
```

Dial connects to an RPC server at the specified network address.

## func DialHTTP

```
func DialHTTP(network, address string) (*Client, error)
```

DialHTTP connects to an HTTP RPC server at the specified network address listening on the default HTTP RPC path.

## func DialHTTPPath

```
func DialHTTPPath(network, address, path string) (*Client, error)
```

DialHTTPPath connects to an HTTP RPC server at the specified network address and path.

## func NewClient

```
func NewClient(conn io.ReadWriteCloser) *Client
```

NewClient returns a new Client to handle requests to the set of services at the other end of the connection. It adds a buffer to the write side of the connection so the header and payload are sent as a unit.

The read and write halves of the connection are serialized independently, so no interlocking is required. However each half may be accessed concurrently so the implementation of conn should protect against concurrent reads or concurrent writes.

## func `NewClientWithCodec`

```
func NewClientWithCodec(codec ClientCodec) *Client
```

`NewClientWithCodec` is like `NewClient` but uses the specified codec to encode requests and decode responses.

## func (`*Client`) `Call`

```
func (client *Client) Call(serviceMethod string, args interface{}, reply interface{})
```

`Call` invokes the named function, waits for it to complete, and returns its error status.

## func (`*Client`) `Close`

```
func (client *Client) Close() error
```

`Close` calls the underlying codec's `Close` method. If the connection is already shutting down, `ErrShutdown` is returned.

## func (`*Client`) `Go`

```
func (client *Client) Go(serviceMethod string, args interface{}, reply interface{}, d
```

`Go` invokes the function asynchronously. It returns the `Call` structure representing the invocation. The `done` channel will signal when the call is complete by returning the same `Call` object. If `done` is `nil`, `Go` will allocate a new channel. If non-`nil`, `done` must be buffered or `Go` will deliberately crash.

## type `ClientCodec`

```
type ClientCodec interface {
 WriteRequest(*Request, interface{}) error
 ReadResponseHeader(*Response) error
 ReadResponseBody(interface{}) error

 Close() error
}
```

A `ClientCodec` implements writing of RPC requests and reading of RPC responses for the client side of an RPC session. The client calls `WriteRequest` to write a request to the connection and calls `ReadResponseHeader` and `ReadResponseBody` in pairs to read responses. The client calls `Close` when finished with the connection. `ReadResponseBody` may be called with a `nil` argument to force the body

of the response to be read and then discarded. See NewClient's comment for information about concurrent access.

## type Request

```
type Request struct {
 ServiceMethod string // format: "Service.Method"
 Seq uint64 // sequence number chosen by client
 // contains filtered or unexported fields
}
```

Request is a header written before every RPC call. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

## type Response

```
type Response struct {
 ServiceMethod string // echoes that of the Request
 Seq uint64 // echoes that of the request
 Error string // error, if any.
 // contains filtered or unexported fields
}
```

Response is a header written before every RPC return. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

## type Server

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

Server represents an RPC Server.

## func NewServer

```
func NewServer() *Server
```

NewServer returns a new Server.

## func (\*Server) Accept

```
func (server *Server) Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests for each incoming connection. Accept blocks until the listener returns a non-nil error. The caller typically invokes Accept in a go statement.

## func (\*Server) HandleHTTP

```
func (server *Server) HandleHTTP(rpcPath, debugPath string)
```

HandleHTTP registers an HTTP handler for RPC messages on rpcPath, and a debugging handler on debugPath. It is still necessary to invoke `http.Serve()`, typically in a go statement.

## func (\*Server) Register

```
func (server *Server) Register(rcvr interface{}) error
```

Register publishes in the server the set of methods of the receiver value that satisfy the following conditions:

- exported method of exported type
- two arguments, both of exported type
- the second argument is a pointer
- one return value, of type error

It returns an error if the receiver is not an exported type or has no suitable methods. It also logs the error using package log. The client accesses each method using a string of the form "Type.Method", where Type is the receiver's concrete type.

## func (\*Server) RegisterName

```
func (server *Server) RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

## func (\*Server) ServeCodec

```
func (server *Server) ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

## func (\*Server) ServeConn

```
func (server *Server) ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the server on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement. ServeConn uses the gob wire format (see package `gob`) on the connection. To use an alternate codec, use ServeCodec. See NewClient's comment for information about concurrent access.

## func (\*Server) ServeHTTP

```
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Request)
```

ServeHTTP implements an http.Handler that answers RPC requests.

## func (\*Server) ServeRequest

```
func (server *Server) ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

## type ServerCodec

```
type ServerCodec interface {
 ReadRequestHeader(*Request) error
 ReadRequestBody(interface{}) error
 WriteResponse(*Response, interface{}) error

 // Close can be called multiple times and must be idempotent.
 Close() error
}
```

A ServerCodec implements reading of RPC requests and writing of RPC responses for the server side of an RPC session. The server calls ReadRequestHeader and ReadRequestBody in pairs to read requests from the connection, and it calls WriteResponse to write a response back. The server calls Close when finished with the connection. ReadRequestBody may be called with a nil argument to force the body of the request to be read and discarded. See NewClient's comment for information about concurrent access.

## type ServerError

```
type ServerError string
```

ServerError represents an error that has been returned from the remote side of the RPC connection.

## func (ServerError) Error

```
func (e ServerError) Error() string
```

# Package jsonrpc

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package jsonrpc implements a JSON-RPC 1.0 ClientCodec and ServerCodec for the rpc package. For JSON-RPC 2.0 support, see <https://godoc.org/?q=json-rpc+2.0>

### func Dial

```
func Dial(network, address string) (*rpc.Client, error)
```

Dial connects to a JSON-RPC server at the specified network address.

### func NewClient

```
func NewClient(conn io.ReadWriteCloser) *rpc.Client
```

NewClient returns a new rpc.Client to handle requests to the set of services at the other end of the connection.

### func NewClientCodec

```
func NewClientCodec(conn io.ReadWriteCloser) rpc.ClientCodec
```

NewClientCodec returns a new rpc.ClientCodec using JSON-RPC on conn.

### func NewServerCodec

```
func NewServerCodec(conn io.ReadWriteCloser) rpc.ServerCodec
```

NewServerCodec returns a new rpc.ServerCodec using JSON-RPC on conn.

### func ServeConn

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the JSON-RPC server on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement.

# Package smtp

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package smtp implements the Simple Mail Transfer Protocol as defined in [RFC 5321](#). It also implements the following extensions:

|          |                          |
|----------|--------------------------|
| 8BITMIME | <a href="#">RFC 1652</a> |
| AUTH     | <a href="#">RFC 2554</a> |
| STARTTLS | <a href="#">RFC 3207</a> |

Additional extensions may be handled by clients.

The smtp package is frozen and is not accepting new features. Some external packages provide more functionality. See:

<https://godoc.org/?q=smtp>

## func SendMail

```
func SendMail(addr string, a Auth, from string, to \[\]string, msg \[\]byte) error
```

SendMail connects to the server at addr, switches to TLS if possible, authenticates with the optional mechanism a if possible, and then sends an email from address from, to addresses to, with message msg. The addr must include a port, as in "mail.example.com:smtp".

The addresses in the to parameter are the SMTP RCPT addresses.

The msg parameter should be an [RFC 822](#)-style email with headers first, a blank line, and then the message body. The lines of msg should be CRLF terminated. The msg headers should usually include fields such as "From", "To", "Subject", and "Cc". Sending "Bcc" messages is accomplished by including an email address in the to parameter but not including it in the msg headers.

The SendMail function and the net/smtp package are low-level mechanisms and provide no support for DKIM signing, MIME attachments (see the mime/multipart package), or other mail functionality. Higher-level packages exist outside of the standard library.

## type Auth

```
type Auth interface {
 // Start begins an authentication with a server.
```

```

// It returns the name of the authentication protocol
// and optionally data to include in the initial AUTH message
// sent to the server. It can return proto == "" to indicate
// that the authentication should be skipped.
// If it returns a non-nil error, the SMTP client aborts
// the authentication attempt and closes the connection.
Start(server *ServerInfo) (proto string, toServer []byte, err error)

// Next continues the authentication. The server has just sent
// the fromServer data. If more is true, the server expects a
// response, which Next should return as toServer; otherwise
// Next should return toServer == nil.
// If Next returns a non-nil error, the SMTP client aborts
// the authentication attempt and closes the connection.
Next(fromServer []byte, more bool) (toServer []byte, err error)
}

```

Auth is implemented by an SMTP authentication mechanism.

## func **CRAMMD5Auth**

```
func CRAMMD5Auth(username, secret string) Auth
```

CRAMMD5Auth returns an Auth that implements the CRAM-MD5 authentication mechanism as defined in [RFC 2195](#). The returned Auth uses the given username and secret to authenticate to the server using the challenge-response mechanism.

## func **PlainAuth**

```
func PlainAuth(identity, username, password, host string) Auth
```

PlainAuth returns an Auth that implements the PLAIN authentication mechanism as defined in [RFC 4616](#). The returned Auth uses the given username and password to authenticate to host and act as identity. Usually identity should be the empty string, to act as username.

PlainAuth will only send the credentials if the connection is using TLS or is connected to localhost. Otherwise authentication will fail with an error, without sending the credentials.

## type **Client**

```

type Client struct {
 // Text is the textproto.Conn used by the Client. It is exported to allow for
 // clients to add extensions.
 Text *textproto.Conn
 // contains filtered or unexported fields
}

```

A Client represents a client connection to an SMTP server.

## func Dial

```
func Dial(addr string) (*Client, error)
```

Dial returns a new Client connected to an SMTP server at addr. The addr must include a port, as in "mail.example.com:smtp".

## func NewClient

```
func NewClient(conn net.Conn, host string) (*Client, error)
```

NewClient returns a new Client using an existing connection and host as a server name to be used when authenticating.

## func (\*Client) Auth

```
func (c *Client) Auth(a Auth) error
```

Auth authenticates a client using the provided authentication mechanism. A failed authentication closes the connection. Only servers that advertise the AUTH extension support this function.

## func (\*Client) Close

```
func (c *Client) Close() error
```

Close closes the connection.

## func (\*Client) Data

```
func (c *Client) Data() (io.WriteCloser, error)
```

Data issues a DATA command to the server and returns a writer that can be used to write the mail headers and body. The caller should close the writer before calling any more methods on c. A call to Data must be preceded by one or more calls to Rcpt.

## func (\*Client) Extension

```
func (c *Client) Extension(ext string) (bool, string)
```

Extension reports whether an extension is support by the server. The extension name is case-insensitive. If the extension is supported, Extension also returns a string that contains any parameters the server specifies for the extension.

## func (\*Client) Hello

```
func (c *Client) Hello(localName string) error
```

Hello sends a HELO or EHLO to the server as the given host name. Calling this method is only necessary if the client needs control over the host name used. The client will introduce itself as "localhost" automatically otherwise. If Hello is called, it must be called before any of the other methods.

## func (\*Client) Mail

```
func (c *Client) Mail(from string) error
```

Mail issues a MAIL command to the server using the provided email address. If the server supports the 8BITMIME extension, Mail adds the BODY=8BITMIME parameter. This initiates a mail transaction and is followed by one or more Rcpt calls.

## func (\*Client) Noop

```
func (c *Client) Noop() error
```

Noop sends the NOOP command to the server. It does nothing but check that the connection to the server is okay.

## func (\*Client) Quit

```
func (c *Client) Quit() error
```

Quit sends the QUIT command and closes the connection to the server.

## func (\*Client) Rcpt

```
func (c *Client) Rcpt(to string) error
```

Rcpt issues a RCPT command to the server using the provided email address. A call to Rcpt must be preceded by a call to Mail and may be followed by a Data call or another Rcpt call.

## func (\*Client) Reset

```
func (c *Client) Reset() error
```

Reset sends the RSET command to the server, aborting the current mail transaction.

## func (\*Client) StartTLS

```
func (c *Client) StartTLS(config *tls.Config) error
```

StartTLS sends the STARTTLS command and encrypts all further communication. Only servers that advertise the STARTTLS extension support this function.

## func (\*Client) TLSConnectionState

```
func (c *Client) TLSConnectionState() (state tls.ConnectionState, ok bool)
```

TLSConnectionState returns the client's TLS connection state. The return values are their zero values if StartTLS did not succeed.

## func (\*Client) Verify

```
func (c *Client) Verify(addr string) error
```

Verify checks the validity of an email address on the server. If Verify returns nil, the address is valid. A non-nil return does not necessarily indicate an invalid address. Many servers will not verify addresses for security reasons.

## type ServerInfo

```
type ServerInfo struct {
 Name string // SMTP server name
 TLS bool // using TLS, with valid certificate for Name
 Auth []string // advertised authentication mechanisms
}
```

ServerInfo records information about an SMTP server.

# Package url

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)[Overview](#)[Subdirectories](#)[Versions](#)[Imports](#)[Imported By](#)[Licenses](#)

## Overview

Package url parses URLs and implements query escaping.

### func `PathEscape`

```
func PathEscape(s string) string
```

`PathEscape` escapes the string so it can be safely placed inside a URL path segment, replacing special characters (including `/`) with `%XX` sequences as needed.

### func `PathUnescape`

```
func PathUnescape(s string) (string, error)
```

`PathUnescape` does the inverse transformation of `PathEscape`, converting each 3-byte encoded substring of the form `"%AB"` into the hex-decoded byte `0xAB`. It returns an error if any `%` is not followed by two hexadecimal digits.

`PathUnescape` is identical to `QueryUnescape` except that it does not unescape `'+'` to `' '` (space).

### func `QueryEscape`

```
func QueryEscape(s string) string
```

`QueryEscape` escapes the string so it can be safely placed inside a URL query.

### func `QueryUnescape`

```
func QueryUnescape(s string) (string, error)
```

`QueryUnescape` does the inverse transformation of `QueryEscape`, converting each 3-byte encoded substring of the form `"%AB"` into the hex-decoded byte `0xAB`. It returns an error if any `%` is not followed by two hexadecimal digits.

### type `Error`

```
type Error struct {
 Op string
```

```
 URL string
 Err error
}
```

Error reports an error and the operation and URL that caused it.

## func (\*Error) Error

```
func (e *Error) Error() string
```

## func (\*Error) Temporary

```
func (e *Error) Temporary() bool
```

## func (\*Error) Timeout

```
func (e *Error) Timeout() bool
```

## func (\*Error) Unwrap

```
func (e *Error) Unwrap() error
```

## type EscapeError

```
type EscapeError string
```

## func (EscapeError) Error

```
func (e EscapeError) Error() string
```

## type InvalidHostError

```
type InvalidHostError string
```

## func (InvalidHostError) Error

```
func (e InvalidHostError) Error() string
```

## type URL

```
type URL struct {
 Scheme string
 Opaque string // encoded opaque data
 User *UserInfo // username and password information
```

```
Host string // host or host:port
Path string // path (relative paths may omit leading slash)
RawPath string // encoded path hint (see EscapedPath method)
ForceQuery bool // append a query ('?') even if RawQuery is empty
RawQuery string // encoded query values, without '?'
Fragment string // fragment for references, without '#'
RawFragment string // encoded fragment hint (see EscapedFragment method)
}
```

A URL represents a parsed URL (technically, a URI reference).

The general form represented is:

```
[scheme:]//[/userinfo@]host[/]path[?query][#fragment]
```

URLs that do not start with a slash after the scheme are interpreted as:

```
scheme:opaque[?query][#fragment]
```

Note that the Path field is stored in decoded form: /%47%6f%2f becomes /Go/. A consequence is that it is impossible to tell which slashes in the Path were slashes in the raw URL and which were %2f. This distinction is rarely important, but when it is, the code should use RawPath, an optional field which only gets set if the default encoding is different from Path.

URL's String method uses the EscapedPath method to obtain the path. See the EscapedPath method for more details.

## func Parse

```
func Parse(rawurl string) (*URL, error)
```

Parse parses rawurl into a URL structure.

The rawurl may be relative (a path, without a host) or absolute (starting with a scheme). Trying to parse a hostname and path without a scheme is invalid but may not necessarily return an error, due to parsing ambiguities.

## func ParseRequestURI

```
func ParseRequestURI(rawurl string) (*URL, error)
```

ParseRequestURI parses rawurl into a URL structure. It assumes that rawurl was received in an HTTP request, so the rawurl is interpreted only as an absolute URI or an absolute path. The string rawurl is assumed not to have a #fragment suffix. (Web browsers strip #fragment before sending the URL to a web server.)

## func (\*URL) EscapedFragment

```
func (u *URL) EscapedFragment() string
```

EscapedFragment returns the escaped form of u.Fragment. In general there are multiple possible escaped forms of any fragment. EscapedFragment returns u.RawFragment when it is a valid escaping of u.Fragment. Otherwise EscapedFragment ignores u.RawFragment and computes an escaped form on its own. The String method uses EscapedFragment to construct its result. In general, code should call EscapedFragment instead of reading u.RawFragment directly.

## func (\*URL) EscapedPath

```
func (u *URL) EscapedPath() string
```

EscapedPath returns the escaped form of u.Path. In general there are multiple possible escaped forms of any path. EscapedPath returns u.RawPath when it is a valid escaping of u.Path. Otherwise EscapedPath ignores u.RawPath and computes an escaped form on its own. The String and RequestURI methods use EscapedPath to construct their results. In general, code should call EscapedPath instead of reading u.RawPath directly.

## func (\*URL) Hostname

```
func (u *URL) Hostname() string
```

Hostname returns u.Host, stripping any valid port number if present.

If the result is enclosed in square brackets, as literal IPv6 addresses are, the square brackets are removed from the result.

## func (\*URL) IsAbs

```
func (u *URL) IsAbs() bool
```

IsAbs reports whether the URL is absolute. Absolute means that it has a non-empty scheme.

## func (\*URL) MarshalBinary

```
func (u *URL) MarshalBinary() (text []byte, err error)
```

## func (\*URL) Parse

```
func (u *URL) Parse(ref string) (*URL, error)
```

Parse parses a URL in the context of the receiver. The provided URL may be relative or absolute. Parse returns nil, err on parse failure, otherwise its return value is the same as ResolveReference.

## func (\*URL) Port

```
func (u *URL) Port() string
```

Port returns the port part of u.Host, without the leading colon.

If u.Host doesn't contain a valid numeric port, Port returns an empty string.

## func (\*URL) Query

```
func (u *URL) Query() Values
```

Query parses RawQuery and returns the corresponding values. It silently discards malformed value pairs. To check errors use ParseQuery.

## func (\*URL) Redacted

```
func (u *URL) Redacted() string
```

Redacted is like String but replaces any password with "xxxxx". Only the password in u.URL is redacted.

## func (\*URL) RequestURI

```
func (u *URL) RequestURI() string
```

RequestURI returns the encoded path?query or opaque?query string that would be used in an HTTP request for u.

## func (\*URL) ResolveReference

```
func (u *URL) ResolveReference(ref *URL) *URL
```

ResolveReference resolves a URI reference to an absolute URI from an absolute base URI u, per [RFC 3986 Section 5.2](#). The URI reference may be relative or absolute. ResolveReference always returns a new URL instance, even if the returned URL is identical to either the base or reference. If ref is an absolute URL, then ResolveReference ignores base and returns a copy of ref.

## func (\*URL) String

```
func (u *URL) String() string
```

String reassembles the URL into a valid URL string. The general form of the result is one of:

```
scheme:opaque?query#fragment
scheme://userinfo@host/path?query#fragment
```

If `u.Opaque` is non-empty, `String` uses the first form; otherwise it uses the second form. Any non-ASCII characters in `host` are escaped. To obtain the path, `String` uses `u.EscapedPath()`.

In the second form, the following rules apply:

- if `u.Scheme` is empty, `scheme:` is omitted.
- if `u.User` is nil, `userinfo@` is omitted.
- if `u.Host` is empty, `host/` is omitted.
- if `u.Scheme` and `u.Host` are empty and `u.User` is nil, the entire `scheme://userinfo@host/` is omitted.
- if `u.Host` is non-empty and `u.Path` begins with a `/`, the form `host/path` does not add its own `/`.
- if `u.RawQuery` is empty, `?query` is omitted.
- if `u.Fragment` is empty, `#fragment` is omitted.

## func (\*URL) `UnmarshalBinary`

```
func (u *URL) UnmarshalBinary(text []byte) error
```

## type `Userinfo`

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

The `Userinfo` type is an immutable encapsulation of username and password details for a URL. An existing `Userinfo` value is guaranteed to have a username set (potentially empty, as allowed by [RFC 2396](#)), and optionally a password.

## func `User`

```
func User(username string) *Userinfo
```

`User` returns a `Userinfo` containing the provided username and no password set.

## func `UserPassword`

```
func UserPassword(username, password string) *Userinfo
```

`UserPassword` returns a `Userinfo` containing the provided username and password.

This functionality should only be used with legacy web sites. [RFC 2396](#) warns that interpreting `Userinfo` this way ``is NOT RECOMMENDED, because the passing of authentication information in clear text (such as URI) has proven to be a security risk in almost every case where it has been used.''

## func (\*Userinfo) `Password`

```
func (u *Userinfo) Password() (string, bool)
```

Password returns the password in case it is set, and whether it is set.

## func (\*Userinfo) String

```
func (u *Userinfo) String() string
```

String returns the encoded userinfo information in the standard form of "username[:password]".

## func (\*Userinfo) Username

```
func (u *Userinfo) Username() string
```

Username returns the username.

## type Values

```
type Values map[string][]string
```

Values maps a string key to a list of values. It is typically used for query parameters and form values. Unlike in the http.Header map, the keys in a Values map are case-sensitive.

## func ParseQuery

```
func ParseQuery(query string) (Values, error)
```

ParseQuery parses the URL-encoded query string and returns a map listing the values specified for each key. ParseQuery always returns a non-nil map containing all the valid query parameters found; err describes the first decoding error encountered, if any.

Query is expected to be a list of key=value settings separated by ampersands or semicolons. A setting without an equals sign is interpreted as a key set to an empty value.

## func (Values) Add

```
func (v Values) Add(key, value string)
```

Add adds the value to key. It appends to any existing values associated with key.

## func (Values) Del

```
func (v Values) Del(key string)
```

Del deletes the values associated with key.

## func (Values) Encode

```
func (v Values) Encode() string
```

Encode encodes the values into ``URL encoded'' form ("bar=baz&foo=quux") sorted by key.

## func (Values) Get

```
func (v Values) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns the empty string. To access multiple values, use the map directly.

## func (Values) Set

```
func (v Values) Set(key, value string)
```

Set sets the key to value. It replaces any existing values.

# Package os

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)   [Overview](#)   [Subdirectories](#)   [Versions](#)   [Imports](#)   [Imported By](#)   [Licenses](#)

## Overview

Package os provides a platform-independent interface to operating system functionality. The design is Unix-like, although the error handling is Go-like; failing calls return values of type `error` rather than error numbers. Often, more information is available within the error. For example, if a call that takes a file name fails, such as `Open` or `Stat`, the error will include the failing file name when printed and will be of type `*PathError`, which may be unpacked for more information.

The `os` interface is intended to be uniform across all operating systems. Features not generally available appear in the system-specific package `syscall`.

Here is a simple example, opening a file and reading some of it.

```
file, err := os.Open("file.go") // For read access.
if err != nil {
 log.Fatal(err)
}
```

If the `open` fails, the error string will be self-explanatory, like

```
open file.go: no such file or directory
```

The file's data can then be read into a slice of bytes. `Read` and `Write` take their byte counts from the length of the argument slice.

```
data := make([]byte, 100)
count, err := file.Read(data)
if err != nil {
 log.Fatal(err)
}
fmt.Printf("read %d bytes: %q\n", count, data[:count])
```

Note: The maximum number of concurrent operations on a `File` may be limited by the OS or the system. The number should be high, but exceeding it may degrade performance or cause other issues.

## Constants

```
const (
 // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
 O_RDONLY int = syscall.O_RDONLY // open the file read-only.
```

```

O_WRONLY int = syscall.O_WRONLY // open the file write-only.
O_RDWR int = syscall.O_RDWR // open the file read-write.
// The remaining values may be or'ed in to control behavior.
O_APPEND int = syscall.O_APPEND // append data to the file when writing.
O_CREATE int = syscall.O_CREAT // create a new file if none exists.
O_EXCL int = syscall.O_EXCL // used with O_CREATE, file must not exist.
O_SYNC int = syscall.O_SYNC // open for synchronous I/O.
O_TRUNC int = syscall.O_TRUNC // truncate regular writable file when opened.
)

```

Flags to OpenFile wrapping those of the underlying system. Not all flags may be implemented on a given system.

```

const (
 SEEK_SET int = 0 // seek relative to the origin of the file
 SEEK_CUR int = 1 // seek relative to the current offset
 SEEK_END int = 2 // seek relative to the end
)

```

Seek whence values.

Deprecated: Use io.SeekStart, io.SeekCurrent, and io.SeekEnd.

```

const (
 PathSeparator = '/' // OS-specific path separator
 PathListSeparator = ':' // OS-specific path list separator
)

```

```
const DevNull = "/dev/null"
```

DevNull is the name of the operating system's ``null device.'' On Unix-like systems, it is "/dev/null"; on Windows, "NUL".

## Variables

```

var (
 // ErrInvalid indicates an invalid argument.
 // Methods on File will return this error when the receiver is nil.
 ErrInvalid = errInvalid() // "invalid argument"

 ErrPermission = errPermission() // "permission denied"
 ErrExist = errExist() // "file already exists"
 ErrNotExist = errNotExist() // "file does not exist"
 ErrClosed = errClosed() // "file already closed"
 ErrNoDeadline = errNoDeadline() // "file type does not support deadline"
 ErrDeadlineExceeded = errDeadlineExceeded() // "i/o timeout"
)

```

Portable analogs of some common system call errors.

Errors returned from this package may be tested against these errors with errors.ls.

```
var (
 Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
 Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
 Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Stdin, Stdout, and Stderr are open Files pointing to the standard input, standard output, and standard error file descriptors.

Note that the Go runtime writes to standard error for panics and crashes; closing Stderr may cause those messages to go elsewhere, perhaps to a file opened later.

```
var Args []string
```

Args hold the command-line arguments, starting with the program name.

## func Chdir

```
func Chdir(dir string) error
```

Chdir changes the current working directory to the named directory. If there is an error, it will be of type `*PathError`.

## func Chmod

```
func Chmod(name string, mode FileMode) error
```

Chmod changes the mode of the named file to mode. If the file is a symbolic link, it changes the mode of the link's target. If there is an error, it will be of type `*PathError`.

A different subset of the mode bits are used, depending on the operating system.

On Unix, the mode's permission bits, ModeSetuid, ModeSetgid, and ModeSticky are used.

On Windows, only the 0200 bit (owner writable) of mode is used; it controls whether the file's read-only attribute is set or cleared. The other bits are currently unused. For compatibility with Go 1.12 and earlier, use a non-zero mode. Use mode 0400 for a read-only file and 0600 for a readable+writable file.

On Plan 9, the mode's permission bits, ModeAppend, ModeExclusive, and ModeTemporary are used.

## func Chown

```
func Chown(name string, uid, gid int) error
```

`Chown` changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link's target. A uid or gid of -1 means to not change that value. If there is an error, it will be of type `*PathError`.

On Windows or Plan 9, `Chown` always returns the `syscall.EWINDOWS` or `EPLAN9` error, wrapped in `*PathError`.

## func `Chtimes`

```
func Chtimes(name string, atime time.Time, mtime time.Time) error
```

`Chtimes` changes the access and modification times of the named file, similar to the Unix `utime()` or `utimes()` functions.

The underlying filesystem may truncate or round the values to a less precise time unit. If there is an error, it will be of type `*PathError`.

## func `Clearenv`

```
func Clearenv()
```

`Clearenv` deletes all environment variables.

## func `Environ`

```
func Environ() []string
```

`Environ` returns a copy of strings representing the environment, in the form "key=value".

## func `Executable`

```
func Executable() (string, error)
```

`Executable` returns the path name for the executable that started the current process. There is no guarantee that the path is still pointing to the correct executable. If a symlink was used to start the process, depending on the operating system, the result might be the symlink or the path it pointed to. If a stable result is needed, `path/filepath.EvalSymlinks` might help.

`Executable` returns an absolute path unless an error occurred.

The main use case is finding resources located relative to an executable.

## func `Exit`

```
func Exit(code int)
```

Exit causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error. The program terminates immediately; deferred functions are not run.

For portability, the status code should be in the range [0, 125].

## func `Expand`

```
func Expand(s string, mapping func(string) string) string
```

Expand replaces \${var} or \$var in the string based on the mapping function. For example, os.ExpandEnv(s) is equivalent to os.Expand(s, os.Getenv).

## func `ExpandEnv`

```
func ExpandEnv(s string) string
```

ExpandEnv replaces \${var} or \$var in the string according to the values of the current environment variables. References to undefined variables are replaced by the empty string.

## func `Getegid`

```
func Getegid() int
```

Getegid returns the numeric effective group id of the caller.

On Windows, it returns -1.

## func `Getenv`

```
func Getenv(key string) string
```

Getenv retrieves the value of the environment variable named by the key. It returns the value, which will be empty if the variable is not present. To distinguish between an empty value and an unset value, use LookupEnv.

## func `Geteuid`

```
func Geteuid() int
```

Geteuid returns the numeric effective user id of the caller.

On Windows, it returns -1.

## func `Getgid`

```
func Getgid() int
```

Getgid returns the numeric group id of the caller.

On Windows, it returns -1.

## func **Getgroups**

```
func Getgroups() ([]int, error)
```

Getgroups returns a list of the numeric ids of groups that the caller belongs to.

On Windows, it returns syscall.EWINDOWS. See the os/user package for a possible alternative.

## func **Getpagesize**

```
func Getpagesize() int
```

Getpagesize returns the underlying system's memory page size.

## func **Getpid**

```
func Getpid() int
```

Getpid returns the process id of the caller.

## func **Getppid**

```
func Getppid() int
```

Getppid returns the process id of the caller's parent.

## func **Getuid**

```
func Getuid() int
```

Getuid returns the numeric user id of the caller.

On Windows, it returns -1.

## func **Getwd**

```
func Getwd() (dir string, err error)
```

Getwd returns a rooted path name corresponding to the current directory. If the current directory can be reached via multiple paths (due to symbolic links), Getwd may return any one of them.

## func **Hostname**

```
func Hostname() (name string, err error)
```

Hostname returns the host name reported by the kernel.

## func **IsExist**

```
func IsExist(err error) bool
```

IsExist returns a boolean indicating whether the error is known to report that a file or directory already exists. It is satisfied by ErrExist as well as some syscall errors.

## func **IsNotExist**

```
func IsNotExist(err error) bool
```

IsNotExist returns a boolean indicating whether the error is known to report that a file or directory does not exist. It is satisfied by ErrNotExist as well as some syscall errors.

## func **IsPathSeparator**

```
func IsPathSeparator(c uint8) bool
```

IsPathSeparator reports whether c is a directory separator character.

## func **IsPermission**

```
func IsPermission(err error) bool
```

IsPermission returns a boolean indicating whether the error is known to report that permission is denied. It is satisfied by ErrPermission as well as some syscall errors.

## func **IsTimeout**

```
func IsTimeout(err error) bool
```

IsTimeout returns a boolean indicating whether the error is known to report that a timeout occurred.

## func **Lchown**

```
func Lchown(name string, uid, gid int) error
```

Lchown changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link itself. If there is an error, it will be of type \*PathError.

On Windows, it always returns the syscall.EWINDOWS error, wrapped in \*PathError.

## func **Link**

```
func Link(oldname, newname string) error
```

Link creates newname as a hard link to the oldname file. If there is an error, it will be of type \*LinkError.

## func **LookupEnv**

```
func LookupEnv(key string) (string, bool)
```

LookupEnv retrieves the value of the environment variable named by the key. If the variable is present in the environment the value (which may be empty) is returned and the boolean is true. Otherwise the returned value will be empty and the boolean will be false.

## func **Mkdir**

```
func Mkdir(name string, perm FileMode) error
```

Mkdir creates a new directory with the specified name and permission bits (before umask). If there is an error, it will be of type \*PathError.

## func **MkdirAll**

```
func MkdirAll(path string, perm FileMode) error
```

MkdirAll creates a directory named path, along with any necessary parents, and returns nil, or else returns an error. The permission bits perm (before umask) are used for all directories that MkdirAll creates. If path is already a directory, MkdirAll does nothing and returns nil.

## func **NewSyscallError**

```
func NewSyscallError(syscall string, err error) error
```

NewSyscallError returns, as an error, a new SyscallError with the given system call name and error details. As a convenience, if err is nil, NewSyscallError returns nil.

## func **Pipe**

```
func Pipe() (r *File, w *File, err error)
```

Pipe returns a connected pair of Files; reads from r return bytes written to w. It returns the files and an error, if any.

## func **Readlink**

```
func Readlink(name string) (string, error)
```

Readlink returns the destination of the named symbolic link. If there is an error, it will be of type **\*PathError**.

## func Remove

```
func Remove(name string) error
```

Remove removes the named file or (empty) directory. If there is an error, it will be of type **\*PathError**.

## func RemoveAll

```
func RemoveAll(path string) error
```

RemoveAll removes path and any children it contains. It removes everything it can but returns the first error it encounters. If the path does not exist, RemoveAll returns nil (no error). If there is an error, it will be of type **\*PathError**.

## func Rename

```
func Rename(oldpath, newpath string) error
```

Rename renames (moves) oldpath to newpath. If newpath already exists and is not a directory, Rename replaces it. OS-specific restrictions may apply when oldpath and newpath are in different directories. If there is an error, it will be of type **\*LinkError**.

## func SameFile

```
func SameFile(fi1, fi2 FileInfo) bool
```

SameFile reports whether fi1 and fi2 describe the same file. For example, on Unix this means that the device and inode fields of the two underlying structures are identical; on other systems the decision may be based on the path names. SameFile only applies to results returned by this package's Stat. It returns false in other cases.

## func Setenv

```
func Setenv(key, value string) error
```

Setenv sets the value of the environment variable named by the key. It returns an error, if any.

## func Symlink

```
func Symlink(oldname, newname string) error
```

Symlink creates newname as a symbolic link to oldname. If there is an error, it will be of type \*LinkError.

## func TempDir

```
func TempDir() string
```

TempDir returns the default directory to use for temporary files.

On Unix systems, it returns \$TMPDIR if non-empty, else /tmp. On Windows, it uses GetTempPath, returning the first non-empty value from %TMP%, %TEMP%, %USERPROFILE%, or the Windows directory. On Plan 9, it returns /tmp.

The directory is neither guaranteed to exist nor have accessible permissions.

## func Truncate

```
func Truncate(name string, size int64) error
```

Truncate changes the size of the named file. If the file is a symbolic link, it changes the size of the link's target. If there is an error, it will be of type \*PathError.

## func Unsetenv

```
func Unsetenv(key string) error
```

Unsetenv unsets a single environment variable.

## func UserCacheDir

```
func UserCacheDir() (string, error)
```

UserCacheDir returns the default root directory to use for user-specific cached data. Users should create their own application-specific subdirectory within this one and use that.

On Unix systems, it returns \$XDG\_CACHE\_HOME as specified by

<https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html> if non-empty, else \$HOME/.cache. On Darwin, it returns \$HOME/Library/Caches. On Windows, it returns %LocalAppData%. On Plan 9, it returns \$home/lib/cache.

If the location cannot be determined (for example, \$HOME is not defined), then it will return an error.

## func UserConfigDir

```
func UserConfigDir() (string, error)
```

UserConfigDir returns the default root directory to use for user-specific configuration data. Users should create their own application-specific subdirectory within this one and use that.

On Unix systems, it returns \$XDG\_CONFIG\_HOME as specified by <https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html> if non-empty, else \$HOME/.config. On Darwin, it returns \$HOME/Library/Application Support. On Windows, it returns %AppData%. On Plan 9, it returns \$home/lib.

If the location cannot be determined (for example, \$HOME is not defined), then it will return an error.

## func `UserHomeDir`

```
func UserHomeDir() (string, error)
```

UserHomeDir returns the current user's home directory.

On Unix, including macOS, it returns the \$HOME environment variable. On Windows, it returns %USERPROFILE%. On Plan 9, it returns the \$home environment variable.

## type `File`

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

File represents an open file descriptor.

## func `Create`

```
func Create(name string) (*File, error)
```

Create creates or truncates the named file. If the file already exists, it is truncated. If the file does not exist, it is created with mode 0666 (before umask). If successful, methods on the returned File can be used for I/O; the associated file descriptor has mode O\_RDWR. If there is an error, it will be of type \*PathError.

## func `NewFile`

```
func NewFile(fd uintptr, name string) *File
```

NewFile returns a new File with the given file descriptor and name. The returned value will be nil if fd is not a valid file descriptor. On Unix systems, if the file descriptor is in non-blocking mode, NewFile will attempt to return a pollable File (one for which the SetDeadline methods work).

## func `Open`

```
func Open(name string) (*File, error)
```

Open opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O\_RDONLY. If there is an error, it will be of type \*PathError.

## func **OpenFile**

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

OpenFile is the generalized open call; most users will use Open or Create instead. It opens the named file with specified flag (O\_RDONLY etc.). If the file does not exist, and the O\_CREATE flag is passed, it is created with mode perm (before umask). If successful, methods on the returned File can be used for I/O. If there is an error, it will be of type \*PathError.

## func (\*File) **Chdir**

```
func (f *File) Chdir() error
```

Chdir changes the current working directory to the file, which must be a directory. If there is an error, it will be of type \*PathError.

## func (\*File) **Chmod**

```
func (f *File) Chmod(mode FileMode) error
```

Chmod changes the mode of the file to mode. If there is an error, it will be of type \*PathError.

## func (\*File) **Chown**

```
func (f *File) Chown(uid, gid int) error
```

Chown changes the numeric uid and gid of the named file. If there is an error, it will be of type \*PathError.

On Windows, it always returns the syscall.EWINDOWS error, wrapped in \*PathError.

## func (\*File) **Close**

```
func (f *File) Close() error
```

Close closes the File, rendering it unusable for I/O. On files that support SetDeadline, any pending I/O operations will be canceled and return immediately with an error. Close will return an error if it has already been called.

## func (\*File) **Fd**

```
func (f *File) Fd() uintptr
```

Fd returns the integer Unix file descriptor referencing the open file. The file descriptor is valid only until f.Close is called or f is garbage collected. On Unix systems this will cause the SetDeadline methods to stop working.

## func (\*File) Name

```
func (f *File) Name() string
```

Name returns the name of the file as presented to Open.

## func (\*File) Read

```
func (f *File) Read(b []byte) (n int, err error)
```

Read reads up to len(b) bytes from the File. It returns the number of bytes read and any error encountered. At end of file, Read returns 0, io.EOF.

## func (\*File) ReadAt

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt reads len(b) bytes from the File starting at byte offset off. It returns the number of bytes read and the error, if any. ReadAt always returns a non-nil error when n < len(b). At end of file, that error is io.EOF.

## func (\*File) ReadFrom

```
func (f *File) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom implements io.ReaderFrom.

## func (\*File) Readdir

```
func (f *File) Readdir(n int) ([]FileInfo, error)
```

Readdir reads the contents of the directory associated with file and returns a slice of up to n FileInfo values, as would be returned by Lstat, in directory order. Subsequent calls on the same file will yield further FileInfo values.

If n > 0, Readdir returns at most n FileInfo structures. In this case, if Readdir returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is io.EOF.

If `n <= 0`, `Readdir` returns all the `FileInfo` from the directory in a single slice. In this case, if `Readdir` succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, `Readdir` returns the `FileInfo` read until that point and a non-nil error.

## func (\*File) `Readdirnames`

```
func (f *File) Readdirnames(n int) (names []string, err error)
```

`Readdirnames` reads the contents of the directory associated with `file` and returns a slice of up to `n` names of files in the directory, in directory order. Subsequent calls on the same file will yield further names.

If `n > 0`, `Readdirnames` returns at most `n` names. In this case, if `Readdirnames` returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is `io.EOF`.

If `n <= 0`, `Readdirnames` returns all the names from the directory in a single slice. In this case, if `Readdirnames` succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, `Readdirnames` returns the names read until that point and a non-nil error.

## func (\*File) `Seek`

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

`Seek` sets the offset for the next `Read` or `Write` on `file` to `offset`, interpreted according to `whence`: 0 means relative to the origin of the file, 1 means relative to the current offset, and 2 means relative to the end. It returns the new offset and an error, if any. The behavior of `Seek` on a file opened with `O_APPEND` is not specified.

If `f` is a directory, the behavior of `Seek` varies by operating system; you can seek to the beginning of the directory on Unix-like operating systems, but not on Windows.

## func (\*File) `SetDeadline`

```
func (f *File) SetDeadline(t time.Time) error
```

`SetDeadline` sets the read and write deadlines for a `File`. It is equivalent to calling both `SetReadDeadline` and `SetWriteDeadline`.

Only some kinds of files support setting a deadline. Calls to `SetDeadline` for files that do not support deadlines will return `ErrNoDeadline`. On most systems ordinary files do not support deadlines, but pipes do.

A deadline is an absolute time after which I/O operations fail with an error instead of blocking. The deadline applies to all future and pending I/O, not just the immediately following call to `Read` or `Write`.

After a deadline has been exceeded, the connection can be refreshed by setting a deadline in the future.

If the deadline is exceeded a call to Read or Write or to other I/O methods will return an error that wraps ErrDeadlineExceeded. This can be tested using errors.Is(err, os.ErrDeadlineExceeded). That error implements the Timeout method, and calling the Timeout method will return true, but there are other possible errors for which the Timeout will return true even if the deadline has not been exceeded.

An idle timeout can be implemented by repeatedly extending the deadline after successful Read or Write calls.

A zero value for t means I/O operations will not time out.

## func (\*File) SetReadDeadline

```
func (f *File) SetReadDeadline(t time.Time) error
```

SetReadDeadline sets the deadline for future Read calls and any currently-blocked Read call. A zero value for t means Read will not time out. Not all files support setting deadlines; see SetDeadline.

## func (\*File) SetWriteDeadline

```
func (f *File) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline sets the deadline for any future Write calls and any currently-blocked Write call. Even if Write times out, it may return n > 0, indicating that some of the data was successfully written. A zero value for t means Write will not time out. Not all files support setting deadlines; see SetDeadline.

## func (\*File) Stat

```
func (f *File) Stat() (FileInfo, error)
```

Stat returns the FileInfo structure describing file. If there is an error, it will be of type \*PathError.

## func (\*File) Sync

```
func (f *File) Sync() error
```

Sync commits the current contents of the file to stable storage. Typically, this means flushing the file system's in-memory copy of recently written data to disk.

## func (\*File) SyscallConn

```
func (f *File) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw file. This implements the syscall.Conn interface.

## func (\*File) Truncate

```
func (f *File) Truncate(size int64) error
```

Truncate changes the size of the file. It does not change the I/O offset. If there is an error, it will be of type \*PathError.

## func (\*File) Write

```
func (f *File) Write(b []byte) (n int, err error)
```

Write writes len(b) bytes to the File. It returns the number of bytes written and an error, if any. Write returns a non-nil error when n != len(b).

## func (\*File) WriteAt

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt writes len(b) bytes to the File starting at byte offset off. It returns the number of bytes written and an error, if any. WriteAt returns a non-nil error when n != len(b).

If file was opened with the O\_APPEND flag, WriteAt returns an error.

## func (\*File) WriteString

```
func (f *File) WriteString(s string) (n int, err error)
```

WriteString is like Write, but writes the contents of string s rather than a slice of bytes.

## type FileInfo

```
type FileInfo interface {
 Name() string // base name of the file
 Size() int64 // length in bytes for regular files; system-dependent for others
 Mode() FileMode // file mode bits
 ModTime() time.Time // modification time
 IsDir() bool // abbreviation for Mode().IsDir()
 Sys() interface{} // underlying data source (can return nil)
}
```

A FileInfo describes a file and is returned by Stat and Lstat.

## func Lstat

```
func Lstat(name string) (FileInfo, error)
```

`Lstat` returns a `FileInfo` describing the named file. If the file is a symbolic link, the returned `FileInfo` describes the symbolic link. `Lstat` makes no attempt to follow the link. If there is an error, it will be of type `*PathError`.

## func `Stat`

```
func Stat(name string) (FileInfo, error)
```

`Stat` returns a `FileInfo` describing the named file. If there is an error, it will be of type `*PathError`.

## type `FileMode`

```
type FileMode uint32
```

A `FileMode` represents a file's mode and permission bits. The bits have the same definition on all systems, so that information about files can be moved from one system to another portably. Not all bits apply to all systems. The only required bit is `ModeDir` for directories.

```
const (
 // The single letters are the abbreviations
 // used by the String method's formatting.
 ModeDir FileMode = 1 << (32 - 1 - iota) // d: is a directory
 ModeAppend FileMode = 1 << iota // a: append-only
 ModeExclusive FileMode = 1 << (32 - 1 - iota) // l: exclusive use
 ModeTemporary FileMode = 1 << (32 - 1 - iota) // T: temporary file; Plan 9 only
 ModeSymlink FileMode = 1 << (32 - 1 - iota) // L: symbolic link
 ModeDevice FileMode = 1 << (32 - 1 - iota) // D: device file
 ModeNamedPipe FileMode = 1 << (32 - 1 - iota) // p: named pipe (FIFO)
 ModeSocket FileMode = 1 << (32 - 1 - iota) // S: Unix domain socket
 ModeSetuid FileMode = 1 << (32 - 1 - iota) // u: setuid
 ModeSetgid FileMode = 1 << (32 - 1 - iota) // g: setgid
 ModeCharDevice FileMode = 1 << (32 - 1 - iota) // c: Unix character device, when
 ModeSticky FileMode = 1 << (32 - 1 - iota) // t: sticky
 ModeIrregular FileMode = 1 << (32 - 1 - iota) // ?: non-regular file; nothing el

 // Mask for the type bits. For regular files, none will be set.
 ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket | ModeDevice | Mode
 ModePerm FileMode = 0777 // Unix permission bits
)
```

The defined file mode bits are the most significant bits of the `FileMode`. The nine least-significant bits are the standard Unix `rwxrwxrwx` permissions. The values of these bits should be considered part of the public API and may be used in wire protocols or disk representations: they must not be changed, although new bits might be added.

## func ( `FileMode`) `IsDir`

```
func (m FileMode) IsDir() bool
```

IsDir reports whether m describes a directory. That is, it tests for the ModeDir bit being set in m.

## func ( FileMode) IsRegular

```
func (m FileMode) IsRegular() bool
```

IsRegular reports whether m describes a regular file. That is, it tests that no mode type bits are set.

## func ( FileMode) Perm

```
func (m FileMode) Perm() FileMode
```

Perm returns the Unix permission bits in m.

## func ( FileMode) String

```
func (m FileMode) String() string
```

## type LinkError

```
type LinkError struct {
 Op string
 Old string
 New string
 Err error
}
```

LinkError records an error during a link or symlink or rename system call and the paths that caused it.

## func (\*LinkError) Error

```
func (e *LinkError) Error() string
```

## func (\*LinkError) Unwrap

```
func (e *LinkError) Unwrap() error
```

## type PathError

```
type PathError struct {
 Op string
 Path string
 Err error
}
```

PathError records an error and the operation and file path that caused it.

## func (\*PathError) Error

```
func (e *PathError) Error() string
```

## func (\*PathError) Timeout

```
func (e *PathError) Timeout() bool
```

Timeout reports whether this error represents a timeout.

## func (\*PathError) Unwrap

```
func (e *PathError) Unwrap() error
```

## type ProcAttr

```
type ProcAttr struct {
 // If Dir is non-empty, the child changes into the directory before
 // creating the process.
 Dir string
 // If Env is non-nil, it gives the environment variables for the
 // new process in the form returned by Environ.
 // If it is nil, the result of Environ will be used.
 Env []string
 // Files specifies the open files inherited by the new process. The
 // first three entries correspond to standard input, standard output, and
 // standard error. An implementation may support additional entries,
 // depending on the underlying operating system. A nil entry corresponds
 // to that file being closed when the process starts.
 Files []*File

 // Operating system-specific process creation attributes.
 // Note that setting this field means that your program
 // may not execute properly or even compile on some
 // operating systems.
 Sys *syscall.SysProcAttr
}
```

ProcAttr holds the attributes that will be applied to a new process started by StartProcess.

## type Process

```
type Process struct {
 Pid int
```

```
// contains filtered or unexported fields
}
```

Process stores the information about a process created by StartProcess.

## func FindProcess

```
func FindProcess(pid int) (*Process, error)
```

FindProcess looks for a running process by its pid.

The Process it returns can be used to obtain information about the underlying operating system process.

On Unix systems, FindProcess always succeeds and returns a Process for the given pid, regardless of whether the process exists.

## func StartProcess

```
func StartProcess(name string, argv []string, attr *ProcAttr) (*Process, error)
```

StartProcess starts a new process with the program, arguments and attributes specified by name, argv and attr. The argv slice will become os.Args in the new process, so it normally starts with the program name.

If the calling goroutine has locked the operating system thread with runtime.LockOSThread and modified any inheritable OS-level thread state (for example, Linux or Plan 9 name spaces), the new process will inherit the caller's thread state.

StartProcess is a low-level interface. The os/exec package provides higher-level interfaces.

If there is an error, it will be of type \*PathError.

## func (\*Process) Kill

```
func (p *Process) Kill() error
```

Kill causes the Process to exit immediately. Kill does not wait until the Process has actually exited. This only kills the Process itself, not any other processes it may have started.

## func (\*Process) Release

```
func (p *Process) Release() error
```

Release releases any resources associated with the Process p, rendering it unusable in the future. Release only needs to be called if Wait is not.

## func (\*Process) Signal

```
func (p *Process) Signal(sig Signal) error
```

Signal sends a signal to the Process. Sending Interrupt on Windows is not implemented.

## func (\*Process) Wait

```
func (p *Process) Wait() (*ProcessState, error)
```

Wait waits for the Process to exit, and then returns a ProcessState describing its status and an error, if any. Wait releases any resources associated with the Process. On most operating systems, the Process must be a child of the current process or an error will be returned.

## type ProcessState

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

ProcessState stores information about a process, as reported by Wait.

## func (\*ProcessState) ExitCode

```
func (p *ProcessState) ExitCode() int
```

ExitCode returns the exit code of the exited process, or -1 if the process hasn't exited or was terminated by a signal.

## func (\*ProcessState) Exited

```
func (p *ProcessState) Exited() bool
```

Exited reports whether the program has exited.

## func (\*ProcessState) Pid

```
func (p *ProcessState) Pid() int
```

Pid returns the process id of the exited process.

## func (\*ProcessState) String

```
func (p *ProcessState) String() string
```

## func (\*ProcessState) Success

```
func (p *ProcessState) Success() bool
```

Success reports whether the program exited successfully, such as with exit status 0 on Unix.

## func (\*ProcessState) Sys

```
func (p *ProcessState) Sys() interface{}
```

Sys returns system-dependent exit information about the process. Convert it to the appropriate underlying type, such as `syscall.WaitStatus` on Unix, to access its contents.

## func (\*ProcessState) SysUsage

```
func (p *ProcessState) SysUsage() interface{}
```

SysUsage returns system-dependent resource usage information about the exited process. Convert it to the appropriate underlying type, such as `*syscall.Rusage` on Unix, to access its contents. (On Unix, `*syscall.Rusage` matches struct `rusage` as defined in the `getrusage(2)` manual page.)

## func (\*ProcessState) SystemTime

```
func (p *ProcessState) SystemTime() time.Duration
```

SystemTime returns the system CPU time of the exited process and its children.

## func (\*ProcessState) UserTime

```
func (p *ProcessState) UserTime() time.Duration
```

UserTime returns the user CPU time of the exited process and its children.

## type Signal

```
type Signal interface {
 String() string
 Signal() // to distinguish from other Stringers
}
```

A Signal represents an operating system signal. The usual underlying implementation is operating system-dependent: on Unix it is `syscall.Signal`.

```
var (
 Interrupt Signal = syscall.SIGINT
```

```
 Kill Signal = syscall.SIGKILL
)
```

The only signal values guaranteed to be present in the `os` package on all systems are `os.Interrupt` (send the process an interrupt) and `os.Kill` (force the process to exit). On Windows, sending `os.Interrupt` to a process with `os.Process.Signal` is not implemented; it will return an error instead of sending a signal.

## **type [SyscallError](#)**

```
type SyscallError struct {
 Syscall string
 Err error
}
```

`SyscallError` records an error from a specific system call.

## **func ([\\*SyscallError](#)) [Error](#)**

```
func (e *SyscallError) Error() string
```

## **func ([\\*SyscallError](#)) [Timeout](#)**

```
func (e *SyscallError) Timeout() bool
```

`Timeout` reports whether this error represents a timeout.

## **func ([\\*SyscallError](#)) [Unwrap](#)**

```
func (e *SyscallError) Unwrap() error
```

# Package signal

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)[Overview](#)[Subdirectories](#)[Versions](#)[Imports](#)[Imported By](#)[Licenses](#)

## Overview

Package signal implements access to incoming signals.

Signals are primarily used on Unix-like systems. For the use of this package on Windows and Plan 9, see below.

## Types of signals

The signals SIGKILL and SIGSTOP may not be caught by a program, and therefore cannot be affected by this package.

Synchronous signals are signals triggered by errors in program execution: SIGBUS, SIGFPE, and SIGSEGV. These are only considered synchronous when caused by program execution, not when sent using `os.Process.Kill` or the `kill` program or some similar mechanism. In general, except as discussed below, Go programs will convert a synchronous signal into a run-time panic.

The remaining signals are asynchronous signals. They are not triggered by program errors, but are instead sent from the kernel or from some other program.

Of the asynchronous signals, the SIGHUP signal is sent when a program loses its controlling terminal. The SIGINT signal is sent when the user at the controlling terminal presses the interrupt character, which by default is `^C` (Control-C). The SIGQUIT signal is sent when the user at the controlling terminal presses the quit character, which by default is `^\\` (Control-Backslash). In general you can cause a program to simply exit by pressing `^C`, and you can cause it to exit with a stack dump by pressing `^\\`.

## Default behavior of signals in Go programs

By default, a synchronous signal is converted into a run-time panic. A SIGHUP, SIGINT, or SIGTERM signal causes the program to exit. A SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGSTKFLT, SIGEMT, or SIGSYS signal causes the program to exit with a stack dump. A SIGTSTP, SIGTTIN, or SIGTTOU signal gets the system default behavior (these signals are used by the shell for job control). The SIGPROF signal is handled directly by the Go runtime to implement `runtime.CPUProfile`. Other signals will be caught but no action will be taken.

If the Go program is started with either SIGHUP or SIGINT ignored (signal handler set to `SIG_IGN`), they will remain ignored.

If the Go program is started with a non-empty signal mask, that will generally be honored. However, some signals are explicitly unblocked: the synchronous signals, SIGILL, SIGTRAP, SIGSTKFLT, SIGCHLD, SIGPROF, and, on GNU/Linux, signals 32 (SIGCANCEL) and 33 (SIGSETXID) (SIGCANCEL and SIGSETXID are used internally by glibc). Subprocesses started by `os.Exec`, or by the `os/exec` package, will inherit the modified signal mask.

## Changing the behavior of signals in Go programs

The functions in this package allow a program to change the way Go programs handle signals.

`Notify` disables the default behavior for a given set of asynchronous signals and instead delivers them over one or more registered channels. Specifically, it applies to the signals SIGHUP, SIGINT, SIGQUIT, SIGABRT, and SIGTERM. It also applies to the job control signals SIGTSTP, SIGTTIN, and SIGTTOU, in which case the system default behavior does not occur. It also applies to some signals that otherwise cause no action: SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGCHLD, SIGCONT, SIGURG, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGWINCH, SIGIO, SIGPWR, SIGSYS, SIGINFO, SIGTHR, SIGWAITING, SIGLWP, SIGFREEZE, SIGTHAW, SIGLOST, SIGXRES, SIGJVM1, SIGJVM2, and any real time signals used on the system. Note that not all of these signals are available on all systems.

If the program was started with SIGHUP or SIGINT ignored, and `Notify` is called for either signal, a signal handler will be installed for that signal and it will no longer be ignored. If, later, `Reset` or `Ignore` is called for that signal, or `Stop` is called on all channels passed to `Notify` for that signal, the signal will once again be ignored. `Reset` will restore the system default behavior for the signal, while `Ignore` will cause the system to ignore the signal entirely.

If the program is started with a non-empty signal mask, some signals will be explicitly unblocked as described above. If `Notify` is called for a blocked signal, it will be unblocked. If, later, `Reset` is called for that signal, or `Stop` is called on all channels passed to `Notify` for that signal, the signal will once again be blocked.

## SIGPIPE

When a Go program writes to a broken pipe, the kernel will raise a SIGPIPE signal.

If the program has not called `Notify` to receive SIGPIPE signals, then the behavior depends on the file descriptor number. A write to a broken pipe on file descriptors 1 or 2 (standard output or standard error) will cause the program to exit with a SIGPIPE signal. A write to a broken pipe on some other file descriptor will take no action on the SIGPIPE signal, and the write will fail with an EPIPE error.

If the program has called `Notify` to receive SIGPIPE signals, the file descriptor number does not matter. The SIGPIPE signal will be delivered to the `Notify` channel, and the write will fail with an EPIPE error.

This means that, by default, command line programs will behave like typical Unix command line programs, while other programs will not crash with SIGPIPE when writing to a closed network connection.

# Go programs that use cgo or SWIG

In a Go program that includes non-Go code, typically C/C++ code accessed using cgo or SWIG, Go's startup code normally runs first. It configures the signal handlers as expected by the Go runtime, before the non-Go startup code runs. If the non-Go startup code wishes to install its own signal handlers, it must take certain steps to keep Go working well. This section documents those steps and the overall effect changes to signal handler settings by the non-Go code can have on Go programs. In rare cases, the non-Go code may run before the Go code, in which case the next section also applies.

If the non-Go code called by the Go program does not change any signal handlers or masks, then the behavior is the same as for a pure Go program.

If the non-Go code installs any signal handlers, it must use the SA\_ONSTACK flag with sigaction. Failing to do so is likely to cause the program to crash if the signal is received. Go programs routinely run with a limited stack, and therefore set up an alternate signal stack. Also, the Go standard library expects that any signal handlers will use the SA\_RESTART flag. Failing to do so may cause some library calls to return "interrupted system call" errors.

If the non-Go code installs a signal handler for any of the synchronous signals (SIGBUS, SIGFPE, SIGSEGV), then it should record the existing Go signal handler. If those signals occur while executing Go code, it should invoke the Go signal handler (whether the signal occurs while executing Go code can be determined by looking at the PC passed to the signal handler). Otherwise some Go run-time panics will not occur as expected.

If the non-Go code installs a signal handler for any of the asynchronous signals, it may invoke the Go signal handler or not as it chooses. Naturally, if it does not invoke the Go signal handler, the Go behavior described above will not occur. This can be an issue with the SIGPROF signal in particular.

The non-Go code should not change the signal mask on any threads created by the Go runtime. If the non-Go code starts new threads of its own, it may set the signal mask as it pleases.

If the non-Go code starts a new thread, changes the signal mask, and then invokes a Go function in that thread, the Go runtime will automatically unblock certain signals: the synchronous signals, SIGILL, SIGTRAP, SIGSTKFLT, SIGCHLD, SIGPROF, SIGCANCEL, and SIGSETXID. When the Go function returns, the non-Go signal mask will be restored.

If the Go signal handler is invoked on a non-Go thread not running Go code, the handler generally forwards the signal to the non-Go code, as follows. If the signal is SIGPROF, the Go handler does nothing. Otherwise, the Go handler removes itself, unblocks the signal, and raises it again, to invoke any non-Go handler or default system handler. If the program does not exit, the Go handler then reinstalls itself and continues execution of the program.

## Non-Go programs that call Go code

When Go code is built with options like `-buildmode=c-shared`, it will be run as part of an existing non-Go program. The non-Go code may have already installed signal handlers when the Go code starts (that may also happen in unusual cases when using cgo or SWIG; in that case, the discussion here

applies). For `-buildmode=c-archive` the Go runtime will initialize signals at global constructor time. For `-buildmode=c-shared` the Go runtime will initialize signals when the shared library is loaded.

If the Go runtime sees an existing signal handler for the `SIGCANCEL` or `SIGSETXID` signals (which are used only on GNU/Linux), it will turn on the `SA_ONSTACK` flag and otherwise keep the signal handler.

For the synchronous signals and `SIGPIPE`, the Go runtime will install a signal handler. It will save any existing signal handler. If a synchronous signal arrives while executing non-Go code, the Go runtime will invoke the existing signal handler instead of the Go signal handler.

Go code built with `-buildmode=c-archive` or `-buildmode=c-shared` will not install any other signal handlers by default. If there is an existing signal handler, the Go runtime will turn on the `SA_ONSTACK` flag and otherwise keep the signal handler. If `Notify` is called for an asynchronous signal, a Go signal handler will be installed for that signal. If, later, `Reset` is called for that signal, the original handling for that signal will be reinstalled, restoring the non-Go signal handler if any.

Go code built without `-buildmode=c-archive` or `-buildmode=c-shared` will install a signal handler for the asynchronous signals listed above, and save any existing signal handler. If a signal is delivered to a non-Go thread, it will act as described above, except that if there is an existing non-Go signal handler, that handler will be installed before raising the signal.

## Windows

On Windows a `^C` (Control-C) or `^BREAK` (Control-Break) normally cause the program to exit. If `Notify` is called for `os.Interrupt`, `^C` or `^BREAK` will cause `os.Interrupt` to be sent on the channel, and the program will not exit. If `Reset` is called, or `Stop` is called on all channels passed to `Notify`, then the default behavior will be restored.

Additionally, if `Notify` is called, and Windows sends `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT` or `CTRL_SHUTDOWN_EVENT` to the process, `Notify` will return `syscall.SIGTERM`. Unlike Control-C and Control-Break, `Notify` does not change process behavior when either `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT` or `CTRL_SHUTDOWN_EVENT` is received - the process will still get terminated unless it exits. But receiving `syscall.SIGTERM` will give the process an opportunity to clean up before termination.

## Plan 9

On Plan 9, signals have type `syscall.Note`, which is a string. Calling `Notify` with a `syscall.Note` will cause that value to be sent on the channel when that string is posted as a note.

## func `Ignore`

```
func Ignore(sig ...os.Signal)
```

`Ignore` causes the provided signals to be ignored. If they are received by the program, nothing will happen. `Ignore` undoes the effect of any prior calls to `Notify` for the provided signals. If no signals are

provided, all incoming signals will be ignored.

## func Ignored

```
func Ignored(sig os.Signal) bool
```

Ignored reports whether sig is currently ignored.

## func Notify

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

Notify causes package signal to relay incoming signals to c. If no signals are provided, all incoming signals will be relayed to c. Otherwise, just the provided signals will.

Package signal will not block sending to c: the caller must ensure that c has sufficient buffer space to keep up with the expected signal rate. For a channel used for notification of just one signal value, a buffer of size 1 is sufficient.

It is allowed to call Notify multiple times with the same channel: each call expands the set of signals sent to that channel. The only way to remove signals from the set is to call Stop.

It is allowed to call Notify multiple times with different channels and the same signals: each channel receives copies of incoming signals independently.

## func Reset

```
func Reset(sig ...os.Signal)
```

Reset undoes the effect of any prior calls to Notify for the provided signals. If no signals are provided, all signal handlers will be reset.

## func Stop

```
func Stop(c chan<- os.Signal)
```

Stop causes package signal to stop relaying incoming signals to c. It undoes the effect of all prior calls to Notify using c. When Stop returns, it is guaranteed that c will receive no more signals.

# Package path

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package path implements utility routines for manipulating slash-separated paths.

The path package should only be used for paths separated by forward slashes, such as the paths in URLs. This package does not deal with Windows paths with drive letters or backslashes; to manipulate operating system paths, use the path/filepath package.

## Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern indicates a pattern was malformed.

## func Base

```
func Base(path string) string
```

Base returns the last element of path. Trailing slashes are removed before extracting the last element. If the path is empty, Base returns ".". If the path consists entirely of slashes, Base returns "/".

## func Clean

```
func Clean(path string) string
```

Clean returns the shortest path name equivalent to path by purely lexical processing. It applies the following rules iteratively until no further processing can be done:

1. Replace multiple slashes with a single slash.
2. Eliminate each . path name element (the current directory).
3. Eliminate each inner .. path name element (the parent directory) along with the non-.. element that precedes it.
4. Eliminate .. elements that begin a rooted path:  
that is, replace "/.." by "/" at the beginning of a path.

The returned path ends in a slash only if it is the root "/".

If the result of this process is an empty string, Clean returns the string ".".

See also Rob Pike, ``Lexical File Names in Plan 9 or Getting Dot-Dot Right,''

<https://9p.io/sys/doc/lexnames.html>

## func Dir

```
func Dir(path string) string
```

Dir returns all but the last element of path, typically the path's directory. After dropping the final element using Split, the path is Cleaned and trailing slashes are removed. If the path is empty, Dir returns ". ". If the path consists entirely of slashes followed by non-slash bytes, Dir returns a single slash. In any other case, the returned path does not end in a slash.

## func Ext

```
func Ext(path string) string
```

Ext returns the file name extension used by path. The extension is the suffix beginning at the final dot in the final slash-separated element of path; it is empty if there is no dot.

## func IsAbs

```
func IsAbs(path string) bool
```

IsAbs reports whether the path is absolute.

## func Join

```
func Join(elem ...string) string
```

Join joins any number of path elements into a single path, separating them with slashes. Empty elements are ignored. The result is Cleaned. However, if the argument list is empty or all its elements are empty, Join returns an empty string.

## func Match

```
func Match(pattern, name string) (matched bool, err error)
```

Match reports whether name matches the shell pattern. The pattern syntax is:

```
pattern:
 { term }

term:
 '*' matches any sequence of non-/ characters
 '?' matches any single non-/ character
 '[' ['^'] { character-range } ']'
 character class (must be non-empty)
```

```
c matches character c (c != '*', '?', '\\\\', '[]')
'\\\\' c matches character c

character-range:
c matches character c (c != '\\\\', '-', ']')
'\\\\' c matches character c
lo '-' hi matches character c for lo <= c <= hi
```

Match requires pattern to match all of name, not just a substring. The only possible returned error is ErrBadPattern, when pattern is malformed.

## func **Split**

```
func Split(path string) (dir, file string)
```

Split splits path immediately following the final slash, separating it into a directory and file name component. If there is no slash in path, Split returns an empty dir and file set to path. The returned values have the property that path = dir+file.

# Package filepath

 go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package filepath implements utility routines for manipulating filename paths in a way compatible with the target operating system-defined file paths.

The filepath package uses either forward slashes or backslashes, depending on the operating system. To process paths such as URLs that always use forward slashes regardless of the operating system, see the path package.

## Constants

```
const (
 Separator = os.PathSeparator
 ListSeparator = os.PathListSeparator
)
```

## Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern indicates a pattern was malformed.

```
var SkipDir = errors.New("skip this directory")
```

SkipDir is used as a return value from WalkFuncs to indicate that the directory named in the call is to be skipped. It is not returned as an error by any function.

## func Abs

```
func Abs(path string) (string, error)
```

Abs returns an absolute representation of path. If the path is not absolute it will be joined with the current working directory to turn it into an absolute path. The absolute path name for a given file is not guaranteed to be unique. Abs calls Clean on the result.

## func Base

```
func Base(path string) string
```

Base returns the last element of path. Trailing path separators are removed before extracting the last element. If the path is empty, Base returns "..". If the path consists entirely of separators, Base returns a single separator.

## func Clean

```
func Clean(path string) string
```

Clean returns the shortest path name equivalent to path by purely lexical processing. It applies the following rules iteratively until no further processing can be done:

1. Replace multiple Separator elements with a single one.
2. Eliminate each . path name element (the current directory).
3. Eliminate each inner .. path name element (the parent directory) along with the non-.. element that precedes it.
4. Eliminate .. elements that begin a rooted path: that is, replace "/.." by "/" at the beginning of a path, assuming Separator is '/'.

The returned path ends in a slash only if it represents a root directory, such as "/" on Unix or `C:\` on Windows.

Finally, any occurrences of slash are replaced by Separator.

If the result of this process is an empty string, Clean returns the string "..".

See also Rob Pike, ``Lexical File Names in Plan 9 or Getting Dot-Dot Right,''  
<https://9p.io/sys/doc/lexnames.html>

## func Dir

```
func Dir(path string) string
```

Dir returns all but the last element of path, typically the path's directory. After dropping the final element, Dir calls Clean on the path and trailing slashes are removed. If the path is empty, Dir returns "..". If the path consists entirely of separators, Dir returns a single separator. The returned path does not end in a separator unless it is the root directory.

## func EvalSymlinks

```
func EvalSymlinks(path string) (string, error)
```

EvalSymlinks returns the path name after the evaluation of any symbolic links. If path is relative the result will be relative to the current directory, unless one of the components is an absolute symbolic link. EvalSymlinks calls Clean on the result.

## func Ext

```
func Ext(path string) string
```

Ext returns the file name extension used by path. The extension is the suffix beginning at the final dot in the final element of path; it is empty if there is no dot.

## func FromSlash

```
func FromSlash(path string) string
```

FromSlash returns the result of replacing each slash ('/') character in path with a separator character. Multiple slashes are replaced by multiple separators.

## func Glob

```
func Glob(pattern string) (matches []string, err error)
```

Glob returns the names of all files matching pattern or nil if there is no matching file. The syntax of patterns is the same as in Match. The pattern may describe hierarchical names such as /usr/\*/bin/ed (assuming the Separator is '/').

Glob ignores file system errors such as I/O errors reading directories. The only possible returned error is ErrBadPattern, when pattern is malformed.

## func HasPrefix

```
func HasPrefix(p, prefix string) bool
```

HasPrefix exists for historical compatibility and should not be used.

Deprecated: HasPrefix does not respect path boundaries and does not ignore case when required.

## func IsAbs

```
func IsAbs(path string) bool
```

IsAbs reports whether the path is absolute.

## func Join

```
func Join(elem ...string) string
```

Join joins any number of path elements into a single path, separating them with an OS specific Separator. Empty elements are ignored. The result is Cleaned. However, if the argument list is empty or all its elements are empty, Join returns an empty string. On Windows, the result will only be a UNC path if the first non-empty element is a UNC path.

## func Match

```
func Match(pattern, name string) (matched bool, err error)
```

Match reports whether name matches the shell file name pattern. The pattern syntax is:

```
pattern:
 { term }

term:
 '*' matches any sequence of non-Separator characters
 '?' matches any single non-Separator character
 '[' ['^'] { character-range } ']'
 character class (must be non-empty)
 c matches character c (c != '*', '?', '\\\\', '[')
 '\\\\' c matches character c

character-range:
 c matches character c (c != '\\\\', '-', ']')
 '\\\\' c matches character c
 lo '-' hi matches character c for lo <= c <= hi
```

Match requires pattern to match all of name, not just a substring. The only possible returned error is ErrBadPattern, when pattern is malformed.

On Windows, escaping is disabled. Instead, '\\\' is treated as path separator.

## func Rel

```
func Rel(basepath, targpath string) (string, error)
```

Rel returns a relative path that is lexically equivalent to targpath when joined to basepath with an intervening separator. That is, Join(basepath, Rel(basepath, targpath)) is equivalent to targpath itself. On success, the returned path will always be relative to basepath, even if basepath and targpath share no elements. An error is returned if targpath can't be made relative to basepath or if knowing the current working directory would be necessary to compute it. Rel calls Clean on the result.

## func Split

```
func Split(path string) (dir, file string)
```

Split splits path immediately following the final Separator, separating it into a directory and file name component. If there is no Separator in path, Split returns an empty dir and file set to path. The returned values have the property that path = dir+file.

## func SplitList

```
func SplitList(path string) []string
```

SplitList splits a list of paths joined by the OS-specific ListSeparator, usually found in PATH or GOPATH environment variables. Unlike strings.Split, SplitList returns an empty slice when passed an empty string.

## func `ToSlash`

```
func ToSlash(path string) string
```

ToSlash returns the result of replacing each separator character in path with a slash ('/') character. Multiple separators are replaced by multiple slashes.

## func `VolumeName`

```
func VolumeName(path string) string
```

VolumeName returns leading volume name. Given "C:\foo\bar" it returns "C:" on Windows. Given "\\host\share\foo" it returns "\\host\share". On other platforms it returns "".

## func `Walk`

```
func Walk(root string, walkFn WalkFunc) error
```

Walk walks the file tree rooted at root, calling walkFn for each file or directory in the tree, including root. All errors that arise visiting files and directories are filtered by walkFn. The files are walked in lexical order, which makes the output deterministic but means that for very large directories Walk can be inefficient. Walk does not follow symbolic links.

## type `WalkFunc`

```
type WalkFunc func(path string, info os.FileInfo, err error) error
```

WalkFunc is the type of the function called for each file or directory visited by Walk. The path argument contains the argument to Walk as a prefix; that is, if Walk is called with "dir", which is a directory containing the file "a", the walk function will be called with argument "dir/a". The info argument is the os.FileInfo for the named path.

If there was a problem walking to the file or directory named by path, the incoming error will describe the problem and the function can decide how to handle that error (and Walk will not descend into that directory). In the case of an error, the info argument will be nil. If an error is returned, processing stops. The sole exception is when the function returns the special value SkipDir. If the function returns SkipDir when invoked on a directory, Walk skips the directory's contents entirely. If the function returns SkipDir when invoked on a non-directory file, Walk skips the remaining files in the containing directory.

## Overview

Package reflect implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type `interface{}` and extract its dynamic type information by calling `TypeOf`, which returns a `Type`.

A call to `ValueOf` returns a `Value` representing the run-time data. `Zero` takes a `Type` and returns a `Value` representing a zero value for that type.

See "The Laws of Reflection" for an introduction to reflection in Go:

[https://golang.org/doc/articles/laws\\_of\\_reflection.html](https://golang.org/doc/articles/laws_of_reflection.html)

## func `Copy`

```
func Copy(dst, src Value) int
```

`Copy` copies the contents of `src` into `dst` until either `dst` has been filled or `src` has been exhausted. It returns the number of elements copied. `Dst` and `src` each must have kind `Slice` or `Array`, and `dst` and `src` must have the same element type.

As a special case, `src` can have kind `String` if the element type of `dst` is kind `Uint8`.

## func `DeepEqual`

```
func DeepEqual(x, y interface{}) bool
```

`DeepEqual` reports whether `x` and `y` are ``deeply equal," defined as follows. Two values of identical type are deeply equal if one of the following cases applies. Values of distinct types are never deeply equal.

Array values are deeply equal when their corresponding elements are deeply equal.

Struct values are deeply equal if their corresponding fields, both exported and unexported, are deeply equal.

Func values are deeply equal if both are nil; otherwise they are not deeply equal.

Interface values are deeply equal if they hold deeply equal concrete values.

Map values are deeply equal when all of the following are true: they are both nil or both non-nil, they have the same length, and either they are the same map object or their corresponding keys (matched

using Go equality) map to deeply equal values.

Pointer values are deeply equal if they are equal using Go's == operator or if they point to deeply equal values.

Slice values are deeply equal when all of the following are true: they are both nil or both non-nil, they have the same length, and either they point to the same initial entry of the same underlying array (that is, &x[0] == &y[0]) or their corresponding elements (up to length) are deeply equal. Note that a non-nil empty slice and a nil slice (for example, []byte{} and []byte(nil)) are not deeply equal.

Other values - numbers, bools, strings, and channels - are deeply equal if they are equal using Go's == operator.

In general DeepEqual is a recursive relaxation of Go's == operator. However, this idea is impossible to implement without some inconsistency. Specifically, it is possible for a value to be unequal to itself, either because it is of func type (uncomparable in general) or because it is a floating-point NaN value (not equal to itself in floating-point comparison), or because it is an array, struct, or interface containing such a value. On the other hand, pointer values are always equal to themselves, even if they point at or contain such problematic values, because they compare equal using Go's == operator, and that is a sufficient condition to be deeply equal, regardless of content. DeepEqual has been defined so that the same short-cut applies to slices and maps: if x and y are the same slice or the same map, they are deeply equal regardless of content.

As DeepEqual traverses the data values it may find a cycle. The second and subsequent times that DeepEqual compares two pointer values that have been compared before, it treats the values as equal rather than examining the values to which they point. This ensures that DeepEqual terminates.

## func **Swapper**

```
func Swapper(slice interface{}) func(i, j int)
```

Swapper returns a function that swaps the elements in the provided slice.

Swapper panics if the provided interface is not a slice.

## type **ChanDir**

```
type ChanDir int
```

ChanDir represents a channel type's direction.

```
const (
 RecvDir ChanDir = 1 << iota // <-chan
 SendDir = iota // chan<-
 BothDir = RecvDir | SendDir // chan
)
```

## func (ChanDir) String

```
func (d ChanDir) String() string
```

## type Kind

```
type Kind uint
```

A Kind represents the specific kind of type that a Type represents. The zero Kind is not a valid kind.

```
const (
 Invalid Kind = iota
 Bool
 Int
 Int8
 Int16
 Int32
 Int64
 Uint
 Uint8
 Uint16
 Uint32
 Uint64
 Uintptr
 Float32
 Float64
 Complex64
 Complex128
 Array
 Chan
 Func
 Interface
 Map
 Ptr
 Slice
 String
 Struct
 UnsafePointer
)
```

## func (Kind) String

```
func (k Kind) String() string
```

String returns the name of k.

## type MapIter

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

A MapIter is an iterator for ranging over a map. See Value.MapRange.

## func (\*MapIter) Key

```
func (it *MapIter) Key() Value
```

Key returns the key of the iterator's current map entry.

## func (\*MapIter) Next

```
func (it *MapIter) Next() bool
```

Next advances the map iterator and reports whether there is another entry. It returns false when the iterator is exhausted; subsequent calls to Key, Value, or Next will panic.

## func (\*MapIter) Value

```
func (it *MapIter) Value() Value
```

Value returns the value of the iterator's current map entry.

## type Method

```
type Method struct {
 // Name is the method name.
 // PkgPath is the package path that qualifies a lower case (unexported)
 // method name. It is empty for upper case (exported) method names.
 // The combination of PkgPath and Name uniquely identifies a method
 // in a method set.
 // See https://golang.org/ref/spec#Uniqueness_of_identifiers
 Name string
 PkgPath string

 Type Type // method type
 Func Value // func with receiver as first argument
 Index int // index for Type.Method
}
```

Method represents a single method.

## type SelectCase

```
type SelectCase struct {
 Dir SelectDir // direction of case
```

```
 Chan Value // channel to use (for send or receive)
 Send Value // value to send (for send)
}
```

A SelectCase describes a single case in a select operation. The kind of case depends on Dir, the communication direction.

If Dir is SelectDefault, the case represents a default case. Chan and Send must be zero Values.

If Dir is SelectSend, the case represents a send operation. Normally Chan's underlying value must be a channel, and Send's underlying value must be assignable to the channel's element type. As a special case, if Chan is a zero Value, then the case is ignored, and the field Send will also be ignored and may be either zero or non-zero.

If Dir is SelectRecv, the case represents a receive operation. Normally Chan's underlying value must be a channel and Send must be a zero Value. If Chan is a zero Value, then the case is ignored, but Send must still be a zero Value. When a receive operation is selected, the received Value is returned by Select.

## type **SelectDir**

```
type SelectDir int
```

A SelectDir describes the communication direction of a select case.

```
const (
 SelectSend SelectDir // case Chan <- Send
 SelectRecv // case <-Chan:
 SelectDefault // default
)
```

## type **SliceHeader**

```
type SliceHeader struct {
 Data uintptr
 Len int
 Cap int
}
```

SliceHeader is the runtime representation of a slice. It cannot be used safely or portably and its representation may change in a later release. Moreover, the Data field is not sufficient to guarantee the data it references will not be garbage collected, so programs must keep a separate, correctly typed pointer to the underlying data.

## type **StringHeader**

```
type StringHeader struct {
 Data uintptr
}
```

```
 Len int
}
```

StringHeader is the runtime representation of a string. It cannot be used safely or portably and its representation may change in a later release. Moreover, the Data field is not sufficient to guarantee the data it references will not be garbage collected, so programs must keep a separate, correctly typed pointer to the underlying data.

## type StructField

```
type StructField struct {
 // Name is the field name.
 Name string
 // PkgPath is the package path that qualifies a lower case (unexported)
 // field name. It is empty for upper case (exported) field names.
 // See https://golang.org/ref/spec#Uniqueness_of_identifiers
 PkgPath string

 Type Type // field type
 Tag StructTag // field tag string
 Offset uintptr // offset within struct, in bytes
 Index []int // index sequence for Type.FieldByIndex
 Anonymous bool // is an embedded field
}
```

A StructField describes a single field in a struct.

## type StructTag

```
type StructTag string
```

A StructTag is the tag string in a struct field.

By convention, tag strings are a concatenation of optionally space-separated key:"value" pairs. Each key is a non-empty string consisting of non-control characters other than space (U+0020 ' '), quote (U+0022 ""), and colon (U+003A ':'). Each value is quoted using U+0022 "" characters and Go string literal syntax.

## func (StructTag) Get

```
func (tag StructTag) Get(key string) string
```

Get returns the value associated with key in the tag string. If there is no such key in the tag, Get returns the empty string. If the tag does not have the conventional format, the value returned by Get is unspecified. To determine whether a tag is explicitly set to the empty string, use Lookup.

## func (StructTag) Lookup

```
func (tag StructTag) Lookup(key string) (value string, ok bool)
```

Lookup returns the value associated with key in the tag string. If the key is present in the tag the value (which may be empty) is returned. Otherwise the returned value will be the empty string. The ok return value reports whether the value was explicitly set in the tag string. If the tag does not have the conventional format, the value returned by Lookup is unspecified.

## type Type

```
type Type interface {

 // Align returns the alignment in bytes of a value of
 // this type when allocated in memory.
 Align() int

 // FieldAlign returns the alignment in bytes of a value of
 // this type when used as a field in a struct.
 FieldAlign() int

 // Method returns the i'th method in the type's method set.
 // It panics if i is not in the range [0, NumMethod()].
 //
 // For a non-interface type T or *T, the returned Method's Type and Func
 // fields describe a function whose first argument is the receiver.
 //
 // For an interface type, the returned Method's Type field gives the
 // method signature, without a receiver, and the Func field is nil.
 //
 // Only exported methods are accessible and they are sorted in
 // lexicographic order.
 Method(int) Method

 // MethodByName returns the method with that name in the type's
 // method set and a boolean indicating if the method was found.
 //
 // For a non-interface type T or *T, the returned Method's Type and Func
 // fields describe a function whose first argument is the receiver.
 //
 // For an interface type, the returned Method's Type field gives the
 // method signature, without a receiver, and the Func field is nil.
 MethodByName(string) (Method, bool)

 // NumMethod returns the number of exported methods in the type's method set.
 NumMethod() int

 // Name returns the type's name within its package for a defined type.
 // For other (non-defined) types it returns the empty string.
 Name() string

 // PkgPath returns a defined type's package path, that is, the import path
 // that uniquely identifies the package, such as "encoding/base64".
 // If the type was predeclared (string, error) or not defined (*T, struct{}),
}
```

```
// []int, or A where A is an alias for a non-defined type), the package path
// will be the empty string.
PkgPath() string

// Size returns the number of bytes needed to store
// a value of the given type; it is analogous to unsafe.Sizeof.
Size() uintptr

// String returns a string representation of the type.
// The string representation may use shortened package names
// (e.g., base64 instead of "encoding/base64") and is not
// guaranteed to be unique among types. To test for type identity,
// compare the Types directly.
String() string

// Kind returns the specific kind of this type.
Kind() Kind

// Implements reports whether the type implements the interface type u.
Implements(u Type) bool

// AssignableTo reports whether a value of the type is assignable to type u.
AssignableTo(u Type) bool

// ConvertibleTo reports whether a value of the type is convertible to type u.
ConvertibleTo(u Type) bool

// Comparable reports whether values of this type are comparable.
Comparable() bool

// Bits returns the size of the type in bits.
// It panics if the type's Kind is not one of the
// sized or unsized Int, Uint, Float, or Complex kinds.
Bits() int

// ChanDir returns a channel type's direction.
// It panics if the type's Kind is not Chan.
ChanDir() ChanDir

// IsVariadic reports whether a function type's final input parameter
// is a "..." parameter. If so, t.In(t.NumIn() - 1) returns the parameter's
// implicit actual type []T.
//
// For concreteness, if t represents func(x int, y ... float64), then
//
// t.NumIn() == 2
// t.In(0) is the reflect.Type for "int"
// t.In(1) is the reflect.Type for "[]float64"
// t.IsVariadic() == true
//
// IsVariadic panics if the type's Kind is not Func.
IsVariadic() bool

// Elem returns a type's element type.
```

```
// It panics if the type's Kind is not Array, Chan, Map, Ptr, or Slice.
Elem() Type
```

```
// Field returns a struct type's i'th field.
// It panics if the type's Kind is not Struct.
// It panics if i is not in the range [0, NumField()).
Field(i int) StructField
```

```
// FieldByIndex returns the nested field corresponding
// to the index sequence. It is equivalent to calling Field
// successively for each index i.
// It panics if the type's Kind is not Struct.
FieldByIndex(index []int) StructField
```

```
// FieldByName returns the struct field with the given name
// and a boolean indicating if the field was found.
FieldByName(name string) (StructField, bool)
```

```
// FieldByNameFunc returns the struct field with a name
// that satisfies the match function and a boolean indicating if
// the field was found.
//
// FieldByNameFunc considers the fields in the struct itself
// and then the fields in any embedded structs, in breadth first order,
// stopping at the shallowest nesting depth containing one or more
// fields satisfying the match function. If multiple fields at that depth
// satisfy the match function, they cancel each other
// and FieldByNameFunc returns no match.
// This behavior mirrors Go's handling of name lookup in
// structs containing embedded fields.
FieldByNameFunc(match func(string) bool) (StructField, bool)
```

```
// In returns the type of a function type's i'th input parameter.
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumIn()).
In(i int) Type
```

```
// Key returns a map type's key type.
// It panics if the type's Kind is not Map.
Key() Type
```

```
// Len returns an array type's length.
// It panics if the type's Kind is not Array.
Len() int
```

```
// NumField returns a struct type's field count.
// It panics if the type's Kind is not Struct.
NumField() int
```

```
// NumIn returns a function type's input parameter count.
// It panics if the type's Kind is not Func.
NumIn() int
```

```
// NumOut returns a function type's output parameter count.
```

```
// It panics if the type's Kind is not Func.
NumOut() int

// Out returns the type of a function type's i'th output parameter.
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumOut()).
Out(i int) Type
// contains filtered or unexported methods
}
```

Type is the representation of a Go type.

Not all methods apply to all kinds of types. Restrictions, if any, are noted in the documentation for each method. Use the Kind method to find out the kind of type before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run-time panic.

Type values are comparable, such as with the == operator, so they can be used as map keys. Two Type values are equal if they represent identical types.

## func **ArrayOf**

```
func ArrayOf(count int, elem Type) Type
```

ArrayOf returns the array type with the given count and element type. For example, if t represents int, ArrayOf(5, t) represents [5]int.

If the resulting type would be larger than the available address space, ArrayOf panics.

## func **ChanOf**

```
func ChanOf(dir ChanDir, t Type) Type
```

ChanOf returns the channel type with the given direction and element type. For example, if t represents int, ChanOf(RecvDir, t) represents <-chan int.

The gc runtime imposes a limit of 64 kB on channel element types. If t's size is equal to or exceeds this limit, ChanOf panics.

## func **FuncOf**

```
func FuncOf(in, out []Type, variadic bool) Type
```

FuncOf returns the function type with the given argument and result types. For example if k represents int and e represents string, FuncOf([]Type{k}, []Type{e}, false) represents func(int) string.

The variadic argument controls whether the function is variadic. FuncOf panics if the in[len(in)-1] does not represent a slice and variadic is true.

## func MapOf

```
func MapOf(key, elem Type) Type
```

MapOf returns the map type with the given key and element types. For example, if k represents int and e represents string, MapOf(k, e) represents map[int]string.

If the key type is not a valid map key type (that is, if it does not implement Go's == operator), MapOf panics.

## func PtrTo

```
func PtrTo(t Type) Type
```

PtrTo returns the pointer type with element t. For example, if t represents type Foo, PtrTo(t) represents \*Foo.

## func SliceOf

```
func SliceOf(t Type) Type
```

SliceOf returns the slice type with element type t. For example, if t represents int, SliceOf(t) represents []int.

## func StructOf

```
func StructOf(fields []StructField) Type
```

StructOf returns the struct type containing fields. The Offset and Index fields are ignored and computed as they would be by the compiler.

StructOf currently does not generate wrapper methods for embedded fields and panics if passed unexported StructFields. These limitations may be lifted in a future version.

## func TypeOf

```
func TypeOf(i interface{}) Type
```

TypeOf returns the reflection Type that represents the dynamic type of i. If i is a nil interface value, TypeOf returns nil.

## type Value

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

Value is the reflection interface to a Go value.

Not all methods apply to all kinds of values. Restrictions, if any, are noted in the documentation for each method. Use the Kind method to find out the kind of value before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run time panic.

The zero Value represents no value. Its IsValid method returns false, its Kind method returns Invalid, its String method returns "<invalid Value>", and all other methods panic. Most functions and methods never return an invalid value. If one does, its documentation states the conditions explicitly.

A Value can be used concurrently by multiple goroutines provided that the underlying Go value can be used concurrently for the equivalent direct operations.

To compare two Values, compare the results of the Interface method. Using == on two Values does not compare the underlying values they represent.

## func Append

```
func Append(s Value, x ...Value) Value
```

Append appends the values x to a slice s and returns the resulting slice. As in Go, each x's value must be assignable to the slice's element type.

## func AppendSlice

```
func AppendSlice(s, t Value) Value
```

AppendSlice appends a slice t to a slice s and returns the resulting slice. The slices s and t must have the same element type.

## func Indirect

```
func Indirect(v Value) Value
```

Indirect returns the value that v points to. If v is a nil pointer, Indirect returns a zero Value. If v is not a pointer, Indirect returns v.

## func MakeChan

```
func MakeChan(typ Type, buffer int) Value
```

MakeChan creates a new channel with the specified type and buffer size.

## func MakeFunc

```
func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Value
```

MakeFunc returns a new function of the given Type that wraps the function fn. When called, that new function does the following:

- converts its arguments to a slice of Values.
- runs results := fn(args).
- returns the results as a slice of Values, one per formal result.

The implementation fn can assume that the argument Value slice has the number and type of arguments given by typ. If typ describes a variadic function, the final Value is itself a slice representing the variadic arguments, as in the body of a variadic function. The result Value slice returned by fn must have the number and type of results given by typ.

The Value.Call method allows the caller to invoke a typed function in terms of Values; in contrast, MakeFunc allows the caller to implement a typed function in terms of Values.

The Examples section of the documentation includes an illustration of how to use MakeFunc to build a swap function for different types.

## func MakeMap

```
func MakeMap(typ Type) Value
```

MakeMap creates a new map with the specified type.

## func MakeMapWithSize

```
func MakeMapWithSize(typ Type, n int) Value
```

MakeMapWithSize creates a new map with the specified type and initial space for approximately n elements.

## func MakeSlice

```
func MakeSlice(typ Type, len, cap int) Value
```

MakeSlice creates a new zero-initialized slice value for the specified slice type, length, and capacity.

## func New

```
func New(typ Type) Value
```

New returns a Value representing a pointer to a new zero value for the specified type. That is, the returned Value's Type is PtrTo(typ).

## func NewAt

```
func NewAt(typ Type, p unsafe.Pointer) Value
```

NewAt returns a Value representing a pointer to a value of the specified type, using p as that pointer.

## func **Select**

```
func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)
```

Select executes a select operation described by the list of cases. Like the Go select statement, it blocks until at least one of the cases can proceed, makes a uniform pseudo-random choice, and then executes that case. It returns the index of the chosen case and, if that case was a receive operation, the value received and a boolean indicating whether the value corresponds to a send on the channel (as opposed to a zero value received because the channel is closed). Select supports a maximum of 65536 cases.

## func **ValueOf**

```
func ValueOf(i interface{}) Value
```

ValueOf returns a new Value initialized to the concrete value stored in the interface i. ValueOf(nil) returns the zero Value.

## func **Zero**

```
func Zero(typ Type) Value
```

Zero returns a Value representing the zero value for the specified type. The result is different from the zero value of the Value struct, which represents no value at all. For example, Zero(TypeOf(42)) returns a Value with Kind Int and value 0. The returned value is neither addressable nor settable.

## func **(Value) Addr**

```
func (v Value) Addr() Value
```

Addr returns a pointer value representing the address of v. It panics if CanAddr() returns false. Addr is typically used to obtain a pointer to a struct field or slice element in order to call a method that requires a pointer receiver.

## func **(Value) Bool**

```
func (v Value) Bool() bool
```

Bool returns v's underlying value. It panics if v's kind is not Bool.

## func **(Value) Bytes**

```
func (v Value) Bytes() []byte
```

Bytes returns v's underlying value. It panics if v's underlying value is not a slice of bytes.

## func (Value) Call

```
func (v Value) Call(in []Value) []Value
```

Call calls the function v with the input arguments in. For example, if `len(in) == 3`, `v.Call(in)` represents the Go call `v(in[0], in[1], in[2])`. Call panics if v's Kind is not Func. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter. If v is a variadic function, Call creates the variadic slice parameter itself, copying in the corresponding values.

## func (Value) CallSlice

```
func (v Value) CallSlice(in []Value) []Value
```

CallSlice calls the variadic function v with the input arguments in, assigning the slice `in[len(in)-1]` to v's final variadic argument. For example, if `len(in) == 3`, `v.CallSlice(in)` represents the Go call `v(in[0], in[1], in[2]...)`. CallSlice panics if v's Kind is not Func or if v is not variadic. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter.

## func (Value) CanAddr

```
func (v Value) CanAddr() bool
```

CanAddr reports whether the value's address can be obtained with Addr. Such values are called addressable. A value is addressable if it is an element of a slice, an element of an addressable array, a field of an addressable struct, or the result of dereferencing a pointer. If CanAddr returns false, calling Addr will panic.

## func (Value) CanInterface

```
func (v Value) CanInterface() bool
```

CanInterface reports whether Interface can be used without panicking.

## func (Value) CanSet

```
func (v Value) CanSet() bool
```

CanSet reports whether the value of v can be changed. A Value can be changed only if it is addressable and was not obtained by the use of unexported struct fields. If CanSet returns false,

calling Set or any type-specific setter (e.g., SetBool, SetInt) will panic.

## func (Value) Cap

```
func (v Value) Cap() int
```

Cap returns v's capacity. It panics if v's Kind is not Array, Chan, or Slice.

## func (Value) Close

```
func (v Value) Close()
```

Close closes the channel v. It panics if v's Kind is not Chan.

## func (Value) Complex

```
func (v Value) Complex() complex128
```

Complex returns v's underlying value, as a complex128. It panics if v's Kind is not Complex64 or Complex128

## func (Value) Convert

```
func (v Value) Convert(t Type) Value
```

Convert returns the value v converted to type t. If the usual Go conversion rules do not allow conversion of the value v to type t, Convert panics.

## func (Value) Elem

```
func (v Value) Elem() Value
```

Elem returns the value that the interface v contains or that the pointer v points to. It panics if v's Kind is not Interface or Ptr. It returns the zero Value if v is nil.

## func (Value) Field

```
func (v Value) Field(i int) Value
```

Field returns the i'th field of the struct v. It panics if v's Kind is not Struct or i is out of range.

## func (Value) FieldByIndex

```
func (v Value) FieldByIndex(index []int) Value
```

FieldByIndex returns the nested field corresponding to index. It panics if v's Kind is not struct.

## func (Value) FieldByName

```
func (v Value) FieldByName(name string) Value
```

FieldByName returns the struct field with the given name. It returns the zero Value if no field was found. It panics if v's Kind is not struct.

## func (Value) FieldByNameFunc

```
func (v Value) FieldByNameFunc(match func(string) bool) Value
```

FieldByNameFunc returns the struct field with a name that satisfies the match function. It panics if v's Kind is not struct. It returns the zero Value if no field was found.

## func (Value) Float

```
func (v Value) Float() float64
```

Float returns v's underlying value, as a float64. It panics if v's Kind is not Float32 or Float64

## func (Value) Index

```
func (v Value) Index(i int) Value
```

Index returns v's i'th element. It panics if v's Kind is not Array, Slice, or String or i is out of range.

## func (Value) Int

```
func (v Value) Int() int64
```

Int returns v's underlying value, as an int64. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64.

## func (Value) Interface

```
func (v Value) Interface() (i interface{})
```

Interface returns v's current value as an interface{}. It is equivalent to:

```
var i interface{} = (v's underlying value)
```

It panics if the Value was obtained by accessing unexported struct fields.

## func (Value) InterfaceData

```
func (v Value) InterfaceData() [2]uintptr
```

InterfaceData returns the interface v's value as a uintptr pair. It panics if v's Kind is not Interface.

## func (Value) **IsNil**

```
func (v Value) IsNil() bool
```

IsNil reports whether its argument v is nil. The argument must be a chan, func, interface, map, pointer, or slice value; if it is not, IsNil panics. Note that IsNil is not always equivalent to a regular comparison with nil in Go. For example, if v was created by calling ValueOf with an uninitialized interface variable i, `i==nil` will be true but `v.IsNil` will panic as v will be the zero Value.

## func (Value) **IsValid**

```
func (v Value) IsValid() bool
```

IsValid reports whether v represents a value. It returns false if v is the zero Value. If IsValid returns false, all other methods except String panic. Most functions and methods never return an invalid Value. If one does, its documentation states the conditions explicitly.

## func (Value) **IsZero**

```
func (v Value) IsZero() bool
```

IsZero reports whether v is the zero value for its type. It panics if the argument is invalid.

## func (Value) **Kind**

```
func (v Value) Kind() Kind
```

Kind returns v's Kind. If v is the zero Value (IsValid returns false), Kind returns Invalid.

## func (Value) **Len**

```
func (v Value) Len() int
```

Len returns v's length. It panics if v's Kind is not Array, Chan, Map, Slice, or String.

## func (Value) **MapIndex**

```
func (v Value) MapIndex(key Value) Value
```

MapIndex returns the value associated with key in the map v. It panics if v's Kind is not Map. It returns the zero Value if key is not found in the map or if v represents a nil map. As in Go, the key's value must

be assignable to the map's key type.

## func (Value) MapKeys

```
func (v Value) MapKeys() []Value
```

MapKeys returns a slice containing all the keys present in the map, in unspecified order. It panics if v's Kind is not Map. It returns an empty slice if v represents a nil map.

## func (Value) MapRange

```
func (v Value) MapRange() *MapIter
```

MapRange returns a range iterator for a map. It panics if v's Kind is not Map.

Call Next to advance the iterator, and Key/Value to access each entry. Next returns false when the iterator is exhausted. MapRange follows the same iteration semantics as a range statement.

Example:

```
iter := reflect.ValueOf(m).MapRange()
for iter.Next() {
 k := iter.Key()
 v := iter.Value()
 ...
}
```

## func (Value) Method

```
func (v Value) Method(i int) Value
```

Method returns a function value corresponding to v's i'th method. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. Method panics if i is out of range or if v is a nil interface value.

## func (Value) MethodByName

```
func (v Value) MethodByName(name string) Value
```

MethodByName returns a function value corresponding to the method of v with the given name. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. It returns the zero Value if no method was found.

## func (Value) NumField

```
func (v Value) NumField() int
```

NumField returns the number of fields in the struct v. It panics if v's Kind is not Struct.

## func (Value) NumMethod

```
func (v Value) NumMethod() int
```

NumMethod returns the number of exported methods in the value's method set.

## func (Value) OverflowComplex

```
func (v Value) OverflowComplex(x complex128) bool
```

OverflowComplex reports whether the complex128 x cannot be represented by v's type. It panics if v's Kind is not Complex64 or Complex128.

## func (Value) OverflowFloat

```
func (v Value) OverflowFloat(x float64) bool
```

OverflowFloat reports whether the float64 x cannot be represented by v's type. It panics if v's Kind is not Float32 or Float64.

## func (Value) OverflowInt

```
func (v Value) OverflowInt(x int64) bool
```

OverflowInt reports whether the int64 x cannot be represented by v's type. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64.

## func (Value) OverflowUint

```
func (v Value) OverflowUint(x uint64) bool
```

OverflowUint reports whether the uint64 x cannot be represented by v's type. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

## func (Value) Pointer

```
func (v Value) Pointer() uintptr
```

Pointer returns v's value as a uintptr. It returns uintptr instead of unsafe.Pointer so that code using reflect cannot obtain unsafe.Pointers without importing the unsafe package explicitly. It panics if v's Kind is not Chan, Func, Map, Ptr, Slice, or UnsafePointer.

If v's Kind is Func, the returned pointer is an underlying code pointer, but not necessarily enough to identify a single function uniquely. The only guarantee is that the result is zero if and only if v is a nil

func Value.

If v's Kind is Slice, the returned pointer is to the first element of the slice. If the slice is nil the returned value is 0. If the slice is empty but non-nil the return value is non-zero.

## func (Value) Recv

```
func (v Value) Recv() (x Value, ok bool)
```

Recv receives and returns a value from the channel v. It panics if v's Kind is not Chan. The receive blocks until a value is ready. The boolean value ok is true if the value x corresponds to a send on the channel, false if it is a zero value received because the channel is closed.

## func (Value) Send

```
func (v Value) Send(x Value)
```

Send sends x on the channel v. It panics if v's kind is not Chan or if x's type is not the same type as v's element type. As in Go, x's value must be assignable to the channel's element type.

## func (Value) Set

```
func (v Value) Set(x Value)
```

Set assigns x to the value v. It panics if CanSet returns false. As in Go, x's value must be assignable to v's type.

## func (Value) SetBool

```
func (v Value) SetBool(x bool)
```

SetBool sets v's underlying value. It panics if v's Kind is not Bool or if CanSet() is false.

## func (Value) SetBytes

```
func (v Value) SetBytes(x []byte)
```

SetBytes sets v's underlying value. It panics if v's underlying value is not a slice of bytes.

## func (Value) SetCap

```
func (v Value) SetCap(n int)
```

SetCap sets v's capacity to n. It panics if v's Kind is not Slice or if n is smaller than the length or greater than the capacity of the slice.

## func (Value) SetComplex

```
func (v Value) SetComplex(x complex128)
```

SetComplex sets v's underlying value to x. It panics if v's Kind is not Complex64 or Complex128, or if CanSet() is false.

## func (Value) SetFloat

```
func (v Value) SetFloat(x float64)
```

SetFloat sets v's underlying value to x. It panics if v's Kind is not Float32 or Float64, or if CanSet() is false.

## func (Value) SetInt

```
func (v Value) SetInt(x int64)
```

SetInt sets v's underlying value to x. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64, or if CanSet() is false.

## func (Value) SetLen

```
func (v Value) SetLen(n int)
```

SetLen sets v's length to n. It panics if v's Kind is not Slice or if n is negative or greater than the capacity of the slice.

## func (Value) SetMapIndex

```
func (v Value) SetMapIndex(key, elem Value)
```

SetMapIndex sets the element associated with key in the map v to elem. It panics if v's Kind is not Map. If elem is the zero Value, SetMapIndex deletes the key from the map. Otherwise if v holds a nil map, SetMapIndex will panic. As in Go, key's elem must be assignable to the map's key type, and elem's value must be assignable to the map's elem type.

## func (Value) SetPointer

```
func (v Value) SetPointer(x unsafe.Pointer)
```

SetPointer sets the unsafe.Pointer value v to x. It panics if v's Kind is not UnsafePointer.

## func (Value) SetString

```
func (v Value) SetString(x string)
```

SetString sets v's underlying value to x. It panics if v's Kind is not String or if CanSet() is false.

## func (Value) SetUint

```
func (v Value) SetUint(x uint64)
```

SetUint sets v's underlying value to x. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64, or if CanSet() is false.

## func (Value) Slice

```
func (v Value) Slice(i, j int) Value
```

Slice returns v[i:j]. It panics if v's Kind is not Array, Slice or String, or if v is an unaddressable array, or if the indexes are out of bounds.

## func (Value) Slice3

```
func (v Value) Slice3(i, j, k int) Value
```

Slice3 is the 3-index form of the slice operation: it returns v[i:j:k]. It panics if v's Kind is not Array or Slice, or if v is an unaddressable array, or if the indexes are out of bounds.

## func (Value) String

```
func (v Value) String() string
```

String returns the string v's underlying value, as a string. String is a special case because of Go's String method convention. Unlike the other getters, it does not panic if v's Kind is not String. Instead, it returns a string of the form "<T value>" where T is v's type. The fmt package treats Values specially. It does not call their String method implicitly but instead prints the concrete values they hold.

## func (Value) TryRecv

```
func (v Value) TryRecv() (x Value, ok bool)
```

TryRecv attempts to receive a value from the channel v but will not block. It panics if v's Kind is not Chan. If the receive delivers a value, x is the transferred value and ok is true. If the receive cannot finish without blocking, x is the zero Value and ok is false. If the channel is closed, x is the zero value for the channel's element type and ok is false.

## func (Value) TrySend

```
func (v Value) TrySend(x Value) bool
```

TrySend attempts to send x on the channel v but will not block. It panics if v's Kind is not Chan. It reports whether the value was sent. As in Go, x's value must be assignable to the channel's element type.

## func (Value) Type

```
func (v Value) Type() Type
```

Type returns v's type.

## func (Value) Uint

```
func (v Value) Uint() uint64
```

Uint returns v's underlying value, as a uint64. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

## func (Value) UnsafeAddr

```
func (v Value) UnsafeAddr() uintptr
```

UnsafeAddr returns a pointer to v's data. It is for advanced clients that also import the "unsafe" package. It panics if v is not addressable.

## type ValueError

```
type ValueError struct {
 Method string
 Kind Kind
}
```

A ValueError occurs when a Value method is invoked on a Value that does not support it. Such cases are documented in the description of each method.

## func (\*ValueError) Error

```
func (e *ValueError) Error() string
```

## BUGs

- FieldByName and related functions consider struct field names to be equal if the names are equal, even if they are unexported names originating in different packages. The practical effect of this is that the result of t.FieldByName("x") is not well defined if the struct type t contains multiple fields

named x (embedded from different packages). FieldByName may return one of the fields named x or may report that there are none. See <https://golang.org/issue/4876> for more details.

# Package regexp

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package regexp implements regular expression search.

The syntax of the regular expressions accepted is the same general syntax used by Perl, Python, and other languages. More precisely, it is the syntax accepted by RE2 and described at <https://golang.org/s/re2syntax>, except for \C. For an overview of the syntax, run

```
go doc regexp/syntax
```

The regexp implementation provided by this package is guaranteed to run in time linear in the size of the input. (This is a property not guaranteed by most open source implementations of regular expressions.) For more information about this property, see

<https://swtch.com/~rsc/regexp/regexp1.html>

or any book about automata theory.

All characters are UTF-8-encoded code points.

There are 16 methods of `Regexp` that match a regular expression and identify the matched text. Their names are matched by this regular expression:

```
Find(All)?(String)?(Submatch)?(Index)?
```

If 'All' is present, the routine matches successive non-overlapping matches of the entire expression. Empty matches abutting a preceding match are ignored. The return value is a slice containing the successive return values of the corresponding non-'All' routine. These routines take an extra integer argument, `n`. If `n >= 0`, the function returns at most `n` matches/submatches; otherwise, it returns all of them.

If 'String' is present, the argument is a string; otherwise it is a slice of bytes; return values are adjusted as appropriate.

If 'Submatch' is present, the return value is a slice identifying the successive submatches of the expression. Submatches are matches of parenthesized subexpressions (also known as capturing groups) within the regular expression, numbered from left to right in order of opening parenthesis. Submatch 0 is the match of the entire expression, submatch 1 the match of the first parenthesized subexpression, and so on.

If 'Index' is present, matches and submatches are identified by byte index pairs within the input string: `result[2*n:2*n+1]` identifies the indexes of the nth submatch. The pair for `n==0` identifies the match of the entire expression. If 'Index' is not present, the match is identified by the text of the match/submatch. If an index is negative or text is nil, it means that subexpression did not match any string in the input. For 'String' versions an empty string means either no match or an empty match.

There is also a subset of the methods that can be applied to text read from a RuneReader:

```
MatchReader, FindReaderIndex, FindReaderSubmatchIndex
```

This set may grow. Note that regular expression matches may need to examine text beyond the text returned by a match, so the methods that match text from a RuneReader may read arbitrarily far into the input before returning.

(There are a few other methods that do not match this pattern.)

## func Match

```
func Match(pattern string, b []byte) (matched bool, err error)
```

Match reports whether the byte slice `b` contains any match of the regular expression pattern. More complicated queries need to use `Compile` and the full `Regexp` interface.

## func MatchReader

```
func MatchReader(pattern string, r io.RuneReader) (matched bool, err error)
```

MatchReader reports whether the text returned by the RuneReader contains any match of the regular expression pattern. More complicated queries need to use `Compile` and the full `Regexp` interface.

## func MatchString

```
func MatchString(pattern string, s string) (matched bool, err error)
```

MatchString reports whether the string `s` contains any match of the regular expression pattern. More complicated queries need to use `Compile` and the full `Regexp` interface.

## func QuoteMeta

```
func QuoteMeta(s string) string
```

QuoteMeta returns a string that escapes all regular expression metacharacters inside the argument text; the returned string is a regular expression matching the literal text.

## type Regexp

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

Regexp is the representation of a compiled regular expression. A Regexp is safe for concurrent use by multiple goroutines, except for configuration methods, such as Longest.

## func `Compile`

```
func Compile(expr string) (*Regexp, error)
```

Compile parses a regular expression and returns, if successful, a Regexp object that can be used to match against text.

When matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses the one that a backtracking search would have found first. This so-called leftmost-first matching is the same semantics that Perl, Python, and other implementations use, although this package implements it without the expense of backtracking. For POSIX leftmost-longest matching, see `CompilePOSIX`.

## func `CompilePOSIX`

```
func CompilePOSIX(expr string) (*Regexp, error)
```

`CompilePOSIX` is like `Compile` but restricts the regular expression to POSIX ERE (egrep) syntax and changes the match semantics to leftmost-longest.

That is, when matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses a match that is as long as possible. This so-called leftmost-longest matching is the same semantics that early regular expression implementations used and that POSIX specifies.

However, there can be multiple leftmost-longest matches, with different submatch choices, and here this package diverges from POSIX. Among the possible leftmost-longest matches, this package chooses the one that a backtracking search would have found first, while POSIX specifies that the match be chosen to maximize the length of the first subexpression, then the second, and so on from left to right. The POSIX rule is computationally prohibitive and not even well-defined. See <https://swtch.com/~rsc/regexp/regexp2.html#posix> for details.

## func `MustCompile`

```
func MustCompile(str string) *Regexp
```

`MustCompile` is like `Compile` but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

## func MustCompilePOSIX

```
func MustCompilePOSIX(str string) *Regexp
```

MustCompilePOSIX is like CompilePOSIX but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

## func (\*Regexp) Copy

```
func (re *Regexp) Copy() *Regexp
```

Copy returns a new Regexp object copied from re. Calling Longest on one copy does not affect another.

Deprecated: In earlier releases, when using a Regexp in multiple goroutines, giving each goroutine its own copy helped to avoid lock contention. As of Go 1.12, using Copy is no longer necessary to avoid lock contention. Copy may still be appropriate if the reason for its use is to make two copies with different Longest settings.

## func (\*Regexp) Expand

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, match []int) []byte
```

Expand appends template to dst and returns the result; during the append, Expand replaces variables in the template with corresponding matches drawn from src. The match slice should have been returned by FindSubmatchIndex.

In the template, a variable is denoted by a substring of the form \${name} or \${name}, where name is a non-empty sequence of letters, digits, and underscores. A purely numeric name like \$1 refers to the submatch with the corresponding index; other names refer to capturing parentheses named with the (?P<name>...) syntax. A reference to an out of range or unmatched index or a name that is not present in the regular expression is replaced with an empty slice.

In the \${name} form, name is taken to be as long as possible: \${1x} is equivalent to \${1x}, not \${1}x, and, \${10} is equivalent to \${10}, not \${1}0.

To insert a literal \$ in the output, use \$\$ in the template.

## func (\*Regexp) ExpandString

```
func (re *Regexp) ExpandString(dst []byte, template string, src string, match []int)
```

ExpandString is like Expand but the template and source are strings. It appends to and returns a byte slice in order to give the calling code control over allocation.

## func (\*Regexp) Find

```
func (re *Regexp) Find(b []byte) []byte
```

Find returns a slice holding the text of the leftmost match in b of the regular expression. A return value of nil indicates no match.

## func (\*Regexp) FindAll

```
func (re *Regexp) FindAll(b []byte, n int) [][]byte
```

FindAll is the 'All' version of Find; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllIndex

```
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
```

FindAllIndex is the 'All' version of FindIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllString

```
func (re *Regexp) FindAllString(s string, n int) []string
```

FindAllString is the 'All' version of FindString; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllStringIndex

```
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
```

FindAllStringIndex is the 'All' version of FindStringIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllStringSubmatch

```
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
```

FindAllStringSubmatch is the 'All' version of FindStringSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllStringSubmatchIndex

```
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [] []int
```

FindAllStringSubmatchIndex is the 'All' version of FindStringSubmatchIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllSubmatch

```
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][] []byte
```

FindAllSubmatch is the 'All' version of FindSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindAllSubmatchIndex

```
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
```

FindAllSubmatchIndex is the 'All' version of FindSubmatchIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindIndex

```
func (re *Regexp) FindIndex(b []byte) (loc []int)
```

FindIndex returns a two-element slice of integers defining the location of the leftmost match in b of the regular expression. The match itself is at b[loc[0]:loc[1]]. A return value of nil indicates no match.

## func (\*Regexp) FindReaderIndex

```
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
```

FindReaderIndex returns a two-element slice of integers defining the location of the leftmost match of the regular expression in text read from the RuneReader. The match text was found in the input stream at byte offset loc[0] through loc[1]-1. A return value of nil indicates no match.

## func (\*Regexp) FindReaderSubmatchIndex

```
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
```

FindReaderSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression of text read by the RuneReader, and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindString

```
func (re *Regexp) FindString(s string) string
```

FindString returns a string holding the text of the leftmost match in s of the regular expression. If there is no match, the return value is an empty string, but it will also be empty if the regular expression successfully matches an empty string. Use FindStringIndex or FindStringSubmatch if it is necessary to distinguish these cases.

## func (\*Regexp) FindStringIndex

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

FindStringIndex returns a two-element slice of integers defining the location of the leftmost match in s of the regular expression. The match itself is at s[loc[0]:loc[1]]. A return value of nil indicates no match.

## func (\*Regexp) FindStringSubmatch

```
func (re *Regexp) FindStringSubmatch(s string) []string
```

FindStringSubmatch returns a slice of strings holding the text of the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' description in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindStringSubmatchIndex

```
func (re *Regexp) FindStringSubmatchIndex(s string) []int
```

FindStringSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindSubmatch

```
func (re *Regexp) FindSubmatch(b []byte) [][]byte
```

FindSubmatch returns a slice of slices holding the text of the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' descriptions in the package comment. A return value of nil indicates no match.

## func (\*Regexp) FindSubmatchIndex

```
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

FindSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (\*Regexp) LiteralPrefix

```
func (re *Regexp) LiteralPrefix() (prefix string, complete bool)
```

LiteralPrefix returns a literal string that must begin any match of the regular expression re. It returns the boolean true if the literal string comprises the entire regular expression.

## func (\*Regexp) Longest

```
func (re *Regexp) Longest()
```

Longest makes future searches prefer the leftmost-longest match. That is, when matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses a match that is as long as possible. This method modifies the Regexp and may not be called concurrently with any other methods.

## func (\*Regexp) Match

```
func (re *Regexp) Match(b []byte) bool
```

Match reports whether the byte slice b contains any match of the regular expression re.

## func (\*Regexp) MatchReader

```
func (re *Regexp) MatchReader(r io.RuneReader) bool
```

MatchReader reports whether the text returned by the RuneReader contains any match of the regular expression re.

## func (\*Regexp) MatchString

```
func (re *Regexp) MatchString(s string) bool
```

MatchString reports whether the string s contains any match of the regular expression re.

## func (\*Regexp) NumSubexp

```
func (re *Regexp) NumSubexp() int
```

NumSubexp returns the number of parenthesized subexpressions in this Regexp.

## func (\*Regexp) ReplaceAll

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
```

ReplaceAll returns a copy of src, replacing matches of the Regexp with the replacement text repl. Inside repl, \$ signs are interpreted as in Expand, so for instance \$1 represents the text of the first submatch.

## func (\*Regexp) ReplaceAllFunc

```
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []byte) []byte
```

ReplaceAllFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of function repl applied to the matched byte slice. The replacement returned by repl is substituted directly, without using Expand.

## func (\*Regexp) ReplaceAllLiteral

```
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
```

ReplaceAllLiteral returns a copy of src, replacing matches of the Regexp with the replacement bytes repl. The replacement repl is substituted directly, without using Expand.

## func (\*Regexp) ReplaceAllLiteralString

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
```

ReplaceAllLiteralString returns a copy of src, replacing matches of the Regexp with the replacement string repl. The replacement repl is substituted directly, without using Expand.

## func (\*Regexp) ReplaceAllString

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

ReplaceAllString returns a copy of src, replacing matches of the Regexp with the replacement string repl. Inside repl, \$ signs are interpreted as in Expand, so for instance \$1 represents the text of the first submatch.

## func (\*Regexp) ReplaceAllStringFunc

```
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string) string) string
```

ReplaceAllStringFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of function repl applied to the matched substring. The replacement returned by repl is substituted directly, without using Expand.

## func (\*Regexp) Split

```
func (re *Regexp) Split(s string, n int) []string
```

Split slices s into substrings separated by the expression and returns a slice of the substrings between those expression matches.

The slice returned by this method consists of all the substrings of s not contained in the slice returned by FindAllString. When called on an expression that contains no metacharacters, it is equivalent to strings.SplitN.

Example:

```
s := regexp.MustCompile("a*").Split("abaabaccadaaae", 5)
// s: ["", "b", "b", "c", "cadaaae"]
```

The count determines the number of substrings to return:

```
n > 0: at most n substrings; the last substring will be the unsplit remainder.
n == 0: the result is nil (zero substrings)
n < 0: all substrings
```

## func (\*Regexp) String

```
func (re *Regexp) String() string
```

String returns the source text used to compile the regular expression.

## func (\*Regexp) SubexpIndex

```
func (re *Regexp) SubexpIndex(name string) int
```

SubexpIndex returns the index of the first subexpression with the given name, or -1 if there is no subexpression with that name.

Note that multiple subexpressions can be written using the same name, as in `(?P<bob>a+)?P<bob>b+`, which declares two subexpressions named "bob". In this case, SubexpIndex returns the index of the leftmost such subexpression in the regular expression.

## func (\*Regexp) SubexpNames

```
func (re *Regexp) SubexpNames() []string
```

SubexpNames returns the names of the parenthesized subexpressions in this Regexp. The name for the first sub-expression is names[1], so that if m is a match slice, the name for m[i] is SubexpNames()

[i]. Since the Regexp as a whole cannot be named, `names[0]` is always the empty string. The slice should not be modified.

# Package strconv

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package strconv implements conversions to and from string representations of basic data types.

## Numeric Conversions

The most common numeric conversions are `Atoi` (string to int) and `Itoa` (int to string).

```
i, err := strconv.Atoi("-42")
s := strconv.Itoa(-42)
```

These assume decimal and the Go `int` type.

`ParseBool`, `ParseFloat`, `ParseInt`, and `ParseUint` convert strings to values:

```
b, err := strconv.ParseBool("true")
f, err := strconv.ParseFloat("3.1415", 64)
i, err := strconv.ParseInt("-42", 10, 64)
u, err := strconv.ParseUint("42", 10, 64)
```

The parse functions return the widest type (`float64`, `int64`, and `uint64`), but if the size argument specifies a narrower width the result can be converted to that narrower type without data loss:

```
s := "2147483647" // biggest int32
i64, err := strconv.ParseInt(s, 10, 32)
...
i := int32(i64)
```

`FormatBool`, `FormatFloat`, `FormatInt`, and `FormatUint` convert values to strings:

```
s := strconv.FormatBool(true)
s := strconv.FormatFloat(3.1415, 'E', -1, 64)
s := strconv.FormatInt(-42, 16)
s := strconv.FormatUint(42, 16)
```

`AppendBool`, `AppendFloat`, `AppendInt`, and `AppendUint` are similar but append the formatted value to a destination slice.

## String Conversions

Quote and QuoteToASCII convert strings to quoted Go string literals. The latter guarantees that the result is an ASCII string, by escaping any non-ASCII Unicode with \u:

```
q := strconv.Quote("Hello, 世界")
q := strconv.QuoteToASCII("Hello, 世界")
```

QuoteRune and QuoteRuneToASCII are similar but accept runes and return quoted Go rune literals.

Unquote and UnquoteChar unquote Go string and rune literals.

## Constants

```
const IntSize = intSize
```

IntSize is the size in bits of an int or uint value.

## Variables

```
var ErrRange = errors.New("value out of range")
```

ErrRange indicates that a value is out of range for the target type.

```
var ErrSyntax = errors.New("invalid syntax")
```

ErrSyntax indicates that a value does not have the right syntax for the target type.

## func AppendBool

```
func AppendBool(dst []byte, b bool) []byte
```

AppendBool appends "true" or "false", according to the value of b, to dst and returns the extended buffer.

## func AppendFloat

```
func AppendFloat(dst []byte, f float64, fmt byte, prec, bitSize int) []byte
```

AppendFloat appends the string form of the floating-point number f, as generated by FormatFloat, to dst and returns the extended buffer.

## func AppendInt

```
func AppendInt(dst []byte, i int64, base int) []byte
```

AppendInt appends the string form of the integer *i*, as generated by FormatInt, to *dst* and returns the extended buffer.

## func AppendQuote

```
func AppendQuote(dst []byte, s string) []byte
```

AppendQuote appends a double-quoted Go string literal representing *s*, as generated by Quote, to *dst* and returns the extended buffer.

## func AppendQuoteRune

```
func AppendQuoteRune(dst []byte, r rune) []byte
```

AppendQuoteRune appends a single-quoted Go character literal representing the rune, as generated by QuoteRune, to *dst* and returns the extended buffer.

## func AppendQuoteRuneToASCII

```
func AppendQuoteRuneToASCII(dst []byte, r rune) []byte
```

AppendQuoteRuneToASCII appends a single-quoted Go character literal representing the rune, as generated by QuoteRuneToASCII, to *dst* and returns the extended buffer.

## func AppendQuoteRuneToGraphic

```
func AppendQuoteRuneToGraphic(dst []byte, r rune) []byte
```

AppendQuoteRuneToGraphic appends a single-quoted Go character literal representing the rune, as generated by QuoteRuneToGraphic, to *dst* and returns the extended buffer.

## func AppendQuoteToASCII

```
func AppendQuoteToASCII(dst []byte, s string) []byte
```

AppendQuoteToASCII appends a double-quoted Go string literal representing *s*, as generated by QuoteToASCII, to *dst* and returns the extended buffer.

## func AppendQuoteToGraphic

```
func AppendQuoteToGraphic(dst []byte, s string) []byte
```

AppendQuoteToGraphic appends a double-quoted Go string literal representing *s*, as generated by QuoteToGraphic, to *dst* and returns the extended buffer.

## func AppendUint

```
func AppendUint(dst []byte, i uint64, base int) []byte
```

AppendUint appends the string form of the unsigned integer i, as generated by FormatUint, to dst and returns the extended buffer.

## func Atoi

```
func Atoi(s string) (int, error)
```

Atoi is equivalent to ParseInt(s, 10, 0), converted to type int.

## func CanBackquote

```
func CanBackquote(s string) bool
```

CanBackquote reports whether the string s can be represented unchanged as a single-line backquoted string without control characters other than tab.

## func FormatBool

```
func FormatBool(b bool) string
```

FormatBool returns "true" or "false" according to the value of b.

## func FormatComplex

```
func FormatComplex(c complex128, fmt byte, prec, bitSize int) string
```

FormatComplex converts the complex number c to a string of the form (a+bi) where a and b are the real and imaginary parts, formatted according to the format fmt and precision prec.

The format fmt and precision prec have the same meaning as in FormatFloat. It rounds the result assuming that the original was obtained from a complex value of bitSize bits, which must be 64 for complex64 and 128 for complex128.

## func FormatFloat

```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

FormatFloat converts the floating-point number f to a string, according to the format fmt and precision prec. It rounds the result assuming that the original was obtained from a floating-point value of bitSize bits (32 for float32, 64 for float64).

The format fmt is one of 'b' (-ddddp±ddd, a binary exponent), 'e' (-d.ddde±dd, a decimal exponent), 'E' (-d.dddE±dd, a decimal exponent), 'f' (-ddd.dddd, no exponent), 'g' ('e' for large exponents, 'f' otherwise), 'G' ('E' for large exponents, 'f' otherwise), 'x' (-0xd.dddp±ddd, a hexadecimal fraction and binary exponent), or 'X' (-0Xd.dddP±ddd, a hexadecimal fraction and binary exponent).

The precision prec controls the number of digits (excluding the exponent) printed by the 'e', 'E', 'f', 'g', 'G', 'x', and 'X' formats. For 'e', 'E', 'f', 'x', and 'X', it is the number of digits after the decimal point. For 'g' and 'G' it is the maximum number of significant digits (trailing zeros are removed). The special precision -1 uses the smallest number of digits necessary such that ParseFloat will return f exactly.

## func FormatInt

```
func FormatInt(i int64, base int) string
```

FormatInt returns the string representation of i in the given base, for  $2 \leq \text{base} \leq 36$ . The result uses the lower-case letters 'a' to 'z' for digit values  $\geq 10$ .

## func FormatUint

```
func FormatUint(i uint64, base int) string
```

FormatUint returns the string representation of i in the given base, for  $2 \leq \text{base} \leq 36$ . The result uses the lower-case letters 'a' to 'z' for digit values  $\geq 10$ .

## func IsGraphic

```
func IsGraphic(r rune) bool
```

IsGraphic reports whether the rune is defined as a Graphic by Unicode. Such characters include letters, marks, numbers, punctuation, symbols, and spaces, from categories L, M, N, P, S, and Zs.

## func IsPrint

```
func IsPrint(r rune) bool
```

IsPrint reports whether the rune is defined as printable by Go, with the same definition as unicode.IsPrint: letters, numbers, punctuation, symbols and ASCII space.

## func Itoa

```
func Itoa(i int) string
```

Itoa is equivalent to FormatInt(int64(i), 10).

## func ParseBool

```
func ParseBool(str string) (bool, error)
```

ParseBool returns the boolean value represented by the string. It accepts 1, t, T, TRUE, true, True, 0, f, F, FALSE, false, False. Any other value returns an error.

## func ParseComplex

```
func ParseComplex(s string, bitSize int) (complex128, error)
```

ParseComplex converts the string s to a complex number with the precision specified by bitSize: 64 for complex64, or 128 for complex128. When bitSize=64, the result still has type complex128, but it will be convertible to complex64 without changing its value.

The number represented by s must be of the form N, Ni, or N±Ni, where N stands for a floating-point number as recognized by ParseFloat, and i is the imaginary component. If the second N is unsigned, a + sign is required between the two components as indicated by the ±. If the second N is NaN, only a + sign is accepted. The form may be parenthesized and cannot contain any spaces. The resulting complex number consists of the two components converted by ParseFloat.

The errors that ParseComplex returns have concrete type \*NumError and include err.Num = s.

If s is not syntactically well-formed, ParseComplex returns err.Err = ErrSyntax.

If s is syntactically well-formed but either component is more than 1/2 ULP away from the largest floating point number of the given component's size, ParseComplex returns err.Err = ErrRange and c = ±Inf for the respective component.

## func ParseFloat

```
func ParseFloat(s string, bitSize int) (float64, error)
```

ParseFloat converts the string s to a floating-point number with the precision specified by bitSize: 32 for float32, or 64 for float64. When bitSize=32, the result still has type float64, but it will be convertible to float32 without changing its value.

ParseFloat accepts decimal and hexadecimal floating-point number syntax. If s is well-formed and near a valid floating-point number, ParseFloat returns the nearest floating-point number rounded using IEEE754 unbiased rounding. (Parsing a hexadecimal floating-point value only rounds when there are more bits in the hexadecimal representation than will fit in the mantissa.)

The errors that ParseFloat returns have concrete type \*NumError and include err.Num = s.

If s is not syntactically well-formed, ParseFloat returns err.Err = ErrSyntax.

If s is syntactically well-formed but is more than 1/2 ULP away from the largest floating point number of the given size, ParseFloat returns f = ±Inf, err.Err = ErrRange.

ParseFloat recognizes the strings "NaN", and the (possibly signed) strings "Inf" and "Infinity" as their respective special floating point values. It ignores case when matching.

## func `Parselnt`

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
```

Parselnt interprets a string s in the given base (0, 2 to 36) and bit size (0 to 64) and returns the corresponding value i.

If the base argument is 0, the true base is implied by the string's prefix: 2 for "0b", 8 for "0" or "0o", 16 for "0x", and 10 otherwise. Also, for argument base 0 only, underscore characters are permitted as defined by the Go syntax for integer literals.

The bitSize argument specifies the integer type that the result must fit into. Bit sizes 0, 8, 16, 32, and 64 correspond to int, int8, int16, int32, and int64. If bitSize is below 0 or above 64, an error is returned.

The errors that Parselnt returns have concrete type `*NumError` and include err.Num = s. If s is empty or contains invalid digits, err.Err = ErrSyntax and the returned value is 0; if the value corresponding to s cannot be represented by a signed integer of the given size, err.Err = ErrRange and the returned value is the maximum magnitude integer of the appropriate bitSize and sign.

## func `ParseUint`

```
func ParseUint(s string, base int, bitSize int) (uint64, error)
```

ParseUint is like Parselnt but for unsigned numbers.

## func `Quote`

```
func Quote(s string) string
```

Quote returns a double-quoted Go string literal representing s. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for control characters and non-printable characters as defined by IsPrint.

## func `QuoteRune`

```
func QuoteRune(r rune) string
```

QuoteRune returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for control characters and non-printable characters as defined by IsPrint.

## func `QuoteRuneToASCII`

```
func QuoteRuneToASCII(r rune) string
```

QuoteRuneToASCII returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for non-ASCII characters and non-printable characters as defined by IsPrint.

## func `QuoteRuneToGraphic`

```
func QuoteRuneToGraphic(r rune) string
```

QuoteRuneToGraphic returns a single-quoted Go character literal representing the rune. If the rune is not a Unicode graphic character, as defined by IsGraphic, the returned string will use a Go escape sequence (\t, \n, \xFF, \u0100).

## func `QuoteToASCII`

```
func QuoteToASCII(s string) string
```

QuoteToASCII returns a double-quoted Go string literal representing s. The returned string uses Go escape sequences (\t, \n, \xFF, \u0100) for non-ASCII characters and non-printable characters as defined by IsPrint.

## func `QuoteToGraphic`

```
func QuoteToGraphic(s string) string
```

QuoteToGraphic returns a double-quoted Go string literal representing s. The returned string leaves Unicode graphic characters, as defined by IsGraphic, unchanged and uses Go escape sequences (\t, \n, \xFF, \u0100) for non-graphic characters.

## func `Unquote`

```
func Unquote(s string) (string, error)
```

Unquote interprets s as a single-quoted, double-quoted, or backquoted Go string literal, returning the string value that s quotes. (If s is single-quoted, it would be a Go character literal; Unquote returns the corresponding one-character string.)

## func `UnquoteChar`

```
func UnquoteChar(s string, quote byte) (value rune, multibyte bool, tail string, err)
```

UnquoteChar decodes the first character or byte in the escaped string or character literal represented by the string s. It returns four values:

- 1) value, the decoded Unicode code point or byte value;
- 2) multibyte, a boolean indicating whether the decoded character requires a multibyte sequence;
- 3) tail, the remainder of the string after the character; and
- 4) an error that will be nil if the character is syntactically valid.

The second argument, quote, specifies the type of literal being parsed and therefore which escaped quote character is permitted. If set to a single quote, it permits the sequence \' and disallows unescaped '. If set to a double quote, it permits \" and disallows unescaped ". If set to zero, it does not permit either escape and allows both quote characters to appear unescaped.

## type NumError

```
type NumError struct {
 Func string // the failing function (ParseBool, ParseInt, ParseUint, ParseFloat,
 Num string // the input
 Err error // the reason the conversion failed (e.g. ErrRange, ErrSyntax, etc.)
}
```

A NumError records a failed conversion.

## func (\*NumError) Error

```
func (e *NumError) Error() string
```

## func (\*NumError) Unwrap

```
func (e *NumError) Unwrap() error
```

# Package strings

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package strings implements simple functions to manipulate UTF-8 encoded strings.

For information about UTF-8 strings in Go, see <https://blog.golang.org/strings>.

### func `Compare`

```
func Compare(a, b string) int
```

Compare returns an integer comparing two strings lexicographically. The result will be 0 if  $a==b$ , -1 if  $a < b$ , and +1 if  $a > b$ .

Compare is included only for symmetry with package bytes. It is usually clearer and always faster to use the built-in string comparison operators `==`, `<`, `>`, and so on.

### func `Contains`

```
func Contains(s, substr string) bool
```

Contains reports whether substr is within s.

### func `ContainsAny`

```
func ContainsAny(s, chars string) bool
```

ContainsAny reports whether any Unicode code points in chars are within s.

### func `ContainsRune`

```
func ContainsRune(s string, r rune) bool
```

ContainsRune reports whether the Unicode code point r is within s.

### func `Count`

```
func Count(s, substr string) int
```

Count counts the number of non-overlapping instances of substr in s. If substr is an empty string, Count returns 1 + the number of Unicode code points in s.

## func EqualFold

```
func EqualFold(s, t string) bool
```

EqualFold reports whether s and t, interpreted as UTF-8 strings, are equal under Unicode case-folding, which is a more general form of case-insensitivity.

## func Fields

```
func Fields(s string) []string
```

Fields splits the string s around each instance of one or more consecutive white space characters, as defined by `unicode.IsSpace`, returning a slice of substrings of s or an empty slice if s contains only white space.

## func FieldsFunc

```
func FieldsFunc(s string, f func(rune) bool) []string
```

FieldsFunc splits the string s at each run of Unicode code points c satisfying f(c) and returns an array of slices of s. If all code points in s satisfy f(c) or the string is empty, an empty slice is returned.

FieldsFunc makes no guarantees about the order in which it calls f(c) and assumes that f always returns the same value for a given c.

## func HasPrefix

```
func HasPrefix(s, prefix string) bool
```

HasPrefix tests whether the string s begins with prefix.

## func HasSuffix

```
func HasSuffix(s, suffix string) bool
```

HasSuffix tests whether the string s ends with suffix.

## func Index

```
func Index(s, substr string) int
```

Index returns the index of the first instance of substr in s, or -1 if substr is not present in s.

## func `IndexAny`

```
func IndexAny(s, chars string) int
```

`IndexAny` returns the index of the first instance of any Unicode code point from `chars` in `s`, or `-1` if no Unicode code point from `chars` is present in `s`.

## func `IndexByte`

```
func IndexByte(s string, c byte) int
```

`IndexByte` returns the index of the first instance of `c` in `s`, or `-1` if `c` is not present in `s`.

## func `IndexFunc`

```
func IndexFunc(s string, f func(rune) bool) int
```

`IndexFunc` returns the index into `s` of the first Unicode code point satisfying `f(c)`, or `-1` if none do.

## func `IndexRune`

```
func IndexRune(s string, r rune) int
```

`IndexRune` returns the index of the first instance of the Unicode code point `r`, or `-1` if `rune` is not present in `s`. If `r` is `utf8.RuneError`, it returns the first instance of any invalid UTF-8 byte sequence.

## func `Join`

```
func Join(elems []string, sep string) string
```

`Join` concatenates the elements of its first argument to create a single string. The separator string `sep` is placed between elements in the resulting string.

## func `LastIndex`

```
func LastIndex(s, substr string) int
```

`LastIndex` returns the index of the last instance of `substr` in `s`, or `-1` if `substr` is not present in `s`.

## func `LastIndexAny`

```
func LastIndexAny(s, chars string) int
```

`LastIndexAny` returns the index of the last instance of any Unicode code point from `chars` in `s`, or `-1` if no Unicode code point from `chars` is present in `s`.

## func `LastIndexByte`

```
func LastIndexByte(s string, c byte) int
```

`LastIndexByte` returns the index of the last instance of `c` in `s`, or `-1` if `c` is not present in `s`.

## func `LastIndexFunc`

```
func LastIndexFunc(s string, f func(rune) bool) int
```

`LastIndexFunc` returns the index into `s` of the last Unicode code point satisfying `f(c)`, or `-1` if none do.

## func `Map`

```
func Map(mapping func(rune) rune, s string) string
```

`Map` returns a copy of the string `s` with all its characters modified according to the mapping function. If `mapping` returns a negative value, the character is dropped from the string with no replacement.

## func `Repeat`

```
func Repeat(s string, count int) string
```

`Repeat` returns a new string consisting of `count` copies of the string `s`.

It panics if `count` is negative or if the result of `(len(s) * count)` overflows.

## func `Replace`

```
func Replace(s, old, new string, n int) string
```

`Replace` returns a copy of the string `s` with the first `n` non-overlapping instances of `old` replaced by `new`. If `old` is empty, it matches at the beginning of the string and after each UTF-8 sequence, yielding up to `k+1` replacements for a `k`-rune string. If `n < 0`, there is no limit on the number of replacements.

## func `ReplaceAll`

```
func ReplaceAll(s, old, new string) string
```

`ReplaceAll` returns a copy of the string `s` with all non-overlapping instances of `old` replaced by `new`. If `old` is empty, it matches at the beginning of the string and after each UTF-8 sequence, yielding up to `k+1` replacements for a `k`-rune string.

## func `Split`

```
func Split(s, sep string) []string
```

Split slices `s` into all substrings separated by `sep` and returns a slice of the substrings between those separators.

If `s` does not contain `sep` and `sep` is not empty, `Split` returns a slice of length 1 whose only element is `s`.

If `sep` is empty, `Split` splits after each UTF-8 sequence. If both `s` and `sep` are empty, `Split` returns an empty slice.

It is equivalent to `SplitN` with a count of `-1`.

## func `SplitAfter`

```
func SplitAfter(s, sep string) []string
```

`SplitAfter` slices `s` into all substrings after each instance of `sep` and returns a slice of those substrings.

If `s` does not contain `sep` and `sep` is not empty, `SplitAfter` returns a slice of length 1 whose only element is `s`.

If `sep` is empty, `SplitAfter` splits after each UTF-8 sequence. If both `s` and `sep` are empty, `SplitAfter` returns an empty slice.

It is equivalent to `SplitAfterN` with a count of `-1`.

## func `SplitAfterN`

```
func SplitAfterN(s, sep string, n int) []string
```

`SplitAfterN` slices `s` into substrings after each instance of `sep` and returns a slice of those substrings.

The count determines the number of substrings to return:

```
n > 0: at most n substrings; the last substring will be the unsplit remainder.
n == 0: the result is nil (zero substrings)
n < 0: all substrings
```

Edge cases for `s` and `sep` (for example, empty strings) are handled as described in the documentation for `SplitAfter`.

## func `SplitN`

```
func SplitN(s, sep string, n int) []string
```

`SplitN` slices `s` into substrings separated by `sep` and returns a slice of the substrings between those separators.

The count determines the number of substrings to return:

```
n > 0: at most n substrings; the last substring will be the unsplit remainder.
n == 0: the result is nil (zero substrings)
n < 0: all substrings
```

Edge cases for `s` and `sep` (for example, empty strings) are handled as described in the documentation for `Split`.

## func `Title`

```
func Title(s string) string
```

`Title` returns a copy of the string `s` with all Unicode letters that begin words mapped to their Unicode title case.

BUG(rsc): The rule `Title` uses for word boundaries does not handle Unicode punctuation properly.

## func `ToLower`

```
func ToLower(s string) string
```

`ToLower` returns `s` with all Unicode letters mapped to their lower case.

## func `ToLowerSpecial`

```
func ToLowerSpecial(c unicode.SpecialCase, s string) string
```

`ToLowerSpecial` returns a copy of the string `s` with all Unicode letters mapped to their lower case using the case mapping specified by `c`.

## func `ToTitle`

```
func ToTitle(s string) string
```

`ToTitle` returns a copy of the string `s` with all Unicode letters mapped to their Unicode title case.

## func `ToTitleSpecial`

```
func ToTitleSpecial(c unicode.SpecialCase, s string) string
```

`ToTitleSpecial` returns a copy of the string `s` with all Unicode letters mapped to their Unicode title case, giving priority to the special casing rules.

## func `ToUpper`

```
func ToUpper(s string) string
```

ToUpper returns s with all Unicode letters mapped to their upper case.

## func **ToUpperSpecial**

```
func ToUpperSpecial(c unicode.SpecialCase, s string) string
```

ToUpperSpecial returns a copy of the string s with all Unicode letters mapped to their upper case using the case mapping specified by c.

## func **ToValidUTF8**

```
func ToValidUTF8(s, replacement string) string
```

ToValidUTF8 returns a copy of the string s with each run of invalid UTF-8 byte sequences replaced by the replacement string, which may be empty.

## func **Trim**

```
func Trim(s, cutset string) string
```

Trim returns a slice of the string s with all leading and trailing Unicode code points contained in cutset removed.

## func **TrimFunc**

```
func TrimFunc(s string, f func(rune) bool) string
```

TrimFunc returns a slice of the string s with all leading and trailing Unicode code points c satisfying f(c) removed.

## func **TrimLeft**

```
func TrimLeft(s, cutset string) string
```

TrimLeft returns a slice of the string s with all leading Unicode code points contained in cutset removed.

To remove a prefix, use TrimPrefix instead.

## func **TrimLeftFunc**

```
func TrimLeftFunc(s string, f func(rune) bool) string
```

TrimLeftFunc returns a slice of the string `s` with all leading Unicode code points `c` satisfying `f(c)` removed.

## func TrimPrefix

```
func TrimPrefix(s, prefix string) string
```

TrimPrefix returns `s` without the provided leading prefix string. If `s` doesn't start with prefix, `s` is returned unchanged.

## func TrimRight

```
func TrimRight(s, cutset string) string
```

TrimRight returns a slice of the string `s`, with all trailing Unicode code points contained in `cutset` removed.

To remove a suffix, use TrimSuffix instead.

## func TrimRightFunc

```
func TrimRightFunc(s string, f func(rune) bool) string
```

TrimRightFunc returns a slice of the string `s` with all trailing Unicode code points `c` satisfying `f(c)` removed.

## func TrimSpace

```
func TrimSpace(s string) string
```

TrimSpace returns a slice of the string `s`, with all leading and trailing white space removed, as defined by Unicode.

## func TrimSuffix

```
func TrimSuffix(s, suffix string) string
```

TrimSuffix returns `s` without the provided trailing suffix string. If `s` doesn't end with suffix, `s` is returned unchanged.

## type Builder

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

A Builder is used to efficiently build a string using Write methods. It minimizes memory copying. The zero value is ready to use. Do not copy a non-zero Builder.

## func (\*Builder) Cap

```
func (b *Builder) Cap() int
```

Cap returns the capacity of the builder's underlying byte slice. It is the total space allocated for the string being built and includes any bytes already written.

## func (\*Builder) Grow

```
func (b *Builder) Grow(n int)
```

Grow grows b's capacity, if necessary, to guarantee space for another n bytes. After Grow(n), at least n bytes can be written to b without another allocation. If n is negative, Grow panics.

## func (\*Builder) Len

```
func (b *Builder) Len() int
```

Len returns the number of accumulated bytes; b.Len() == len(b.String()).

## func (\*Builder) Reset

```
func (b *Builder) Reset()
```

Reset resets the Builder to be empty.

## func (\*Builder) String

```
func (b *Builder) String() string
```

String returns the accumulated string.

## func (\*Builder) Write

```
func (b *Builder) Write(p []byte) (int, error)
```

Write appends the contents of p to b's buffer. Write always returns len(p), nil.

## func (\*Builder) WriteByte

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

WriteByte appends the byte c to b's buffer. The returned error is always nil.

## func (\*Builder) WriteRune

```
func (b *Builder) WriteRune(r rune) (int, error)
```

WriteRune appends the UTF-8 encoding of Unicode code point r to b's buffer. It returns the length of r and a nil error.

## func (\*Builder) WriteString

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

WriteString appends the contents of s to b's buffer. It returns the length of s and a nil error.

## type Reader

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

A Reader implements the io.Reader, io.ReaderAt, io.Seeker, io.WriterTo, io.ByteScanner, and io.RuneScanner interfaces by reading from a string. The zero value for Reader operates like a Reader of an empty string.

## func NewReader

```
func NewReader(s string) *Reader
```

NewReader returns a new Reader reading from s. It is similar to bytes.NewBufferString but more efficient and read-only.

## func (\*Reader) Len

```
func (r *Reader) Len() int
```

Len returns the number of bytes of the unread portion of the string.

## func (\*Reader) Read

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

## func (\*Reader) ReadAt

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

## func (\*Reader) ReadByte

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

## func (\*Reader) ReadRune

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

## func (\*Reader) Reset

```
func (r *Reader) Reset(s string)
```

Reset resets the Reader to be reading from s.

## func (\*Reader) Seek

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek implements the io.Seeker interface.

## func (\*Reader) Size

```
func (r *Reader) Size() int64
```

Size returns the original length of the underlying string. Size is the number of bytes available for reading via ReadAt. The returned value is always the same and is not affected by calls to any other method.

## func (\*Reader) UnreadByte

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

## func (\*Reader) UnreadRune

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

## func (\*Reader) WriteTo

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

WriteTo implements the io.WriterTo interface.

## type Replacer

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

Replacer replaces a list of strings with replacements. It is safe for concurrent use by multiple goroutines.

## func **NewReplacer**

```
func NewReplacer(oldnew ...string) *Replacer
```

NewReplacer returns a new Replacer from a list of old, new string pairs. Replacements are performed in the order they appear in the target string, without overlapping matches. The old string comparisons are done in argument order.

NewReplacer panics if given an odd number of arguments.

## func (\*Replacer) **Replace**

```
func (r *Replacer) Replace(s string) string
```

Replace returns a copy of s with all replacements performed.

## func (\*Replacer) **WriteString**

```
func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
```

WriteString writes s to w with all replacements performed.

## BUGs

- The rule Title uses for word boundaries does not handle Unicode punctuation properly.

# Package sync

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.

Values containing the types defined in this package should not be copied.

### type Cond

```
type Cond struct {

 // L is held while observing or changing the condition
 L Locker
 // contains filtered or unexported fields
}
```

Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Each Cond has an associated Locker L (often a \*Mutex or \*RWMutex), which must be held when changing the condition and when calling the Wait method.

A Cond must not be copied after first use.

### func NewCond

```
func NewCond(l Locker) *Cond
```

NewCond returns a new Cond with Locker l.

### func (\*Cond) Broadcast

```
func (c *Cond) Broadcast()
```

Broadcast wakes all goroutines waiting on c.

It is allowed but not required for the caller to hold c.L during the call.

### func (\*Cond) Signal

```
func (c *Cond) Signal()
```

Signal wakes one goroutine waiting on c, if there is any.

It is allowed but not required for the caller to hold c.L during the call.

## func (\*Cond) Wait

```
func (c *Cond) Wait()
```

Wait atomically unlocks c.L and suspends execution of the calling goroutine. After later resuming execution, Wait locks c.L before returning. Unlike in other systems, Wait cannot return unless awoken by Broadcast or Signal.

Because c.L is not locked when Wait first resumes, the caller typically cannot assume that the condition is true when Wait returns. Instead, the caller should Wait in a loop:

```
c.L.Lock()
for !condition() {
 c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

## type Locker

```
type Locker interface {
 Lock()
 Unlock()
}
```

A Locker represents an object that can be locked and unlocked.

## type Map

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

Map is like a Go map[interface{}][interface{}] but is safe for concurrent use by multiple goroutines without additional locking or coordination. Loads, stores, and deletes run in amortized constant time.

The Map type is specialized. Most code should use a plain Go map instead, with separate locking or coordination, for better type safety and to make it easier to maintain other invariants along with the map content.

The Map type is optimized for two common use cases: (1) when the entry for a given key is only ever written once but read many times, as in caches that only grow, or (2) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys. In these two cases, use of a Map may significantly reduce lock contention compared to a Go map paired with a separate Mutex or RWMutex.

The zero Map is empty and ready for use. A Map must not be copied after first use.

## func (\*Map) Delete

```
func (m *Map) Delete(key interface{})
```

Delete deletes the value for a key.

## func (\*Map) Load

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool)
```

Load returns the value stored in the map for a key, or nil if no value is present. The ok result indicates whether value was found in the map.

## func (\*Map) LoadAndDelete

```
func (m *Map) LoadAndDelete(key interface{}) (value interface{}, loaded bool)
```

LoadAndDelete deletes the value for a key, returning the previous value if any. The loaded result reports whether the key was present.

## func (\*Map) LoadOrStore

```
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)
```

LoadOrStore returns the existing value for the key if present. Otherwise, it stores and returns the given value. The loaded result is true if the value was loaded, false if stored.

## func (\*Map) Range

```
func (m *Map) Range(f func(key, value interface{}) bool)
```

Range calls f sequentially for each key and value present in the map. If f returns false, range stops the iteration.

Range does not necessarily correspond to any consistent snapshot of the Map's contents: no key will be visited more than once, but if the value for any key is stored or deleted concurrently, Range may reflect any mapping for that key from any point during the Range call.

Range may be  $O(N)$  with the number of elements in the map even if  $f$  returns false after a constant number of calls.

## func (\*Map) Store

```
func (m *Map) Store(key, value interface{})
```

Store sets the value for a key.

## type Mutex

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

A Mutex is a mutual exclusion lock. The zero value for a Mutex is an unlocked mutex.

A Mutex must not be copied after first use.

## func (\*Mutex) Lock

```
func (m *Mutex) Lock()
```

Lock locks  $m$ . If the lock is already in use, the calling goroutine blocks until the mutex is available.

## func (\*Mutex) Unlock

```
func (m *Mutex) Unlock()
```

Unlock unlocks  $m$ . It is a run-time error if  $m$  is not locked on entry to Unlock.

A locked Mutex is not associated with a particular goroutine. It is allowed for one goroutine to lock a Mutex and then arrange for another goroutine to unlock it.

## type Once

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

Once is an object that will perform exactly one action.

## func (\*Once) Do

```
func (o *Once) Do(f func())
```

Do calls the function f if and only if Do is being called for the first time for this instance of Once. In other words, given

```
var once Once
```

if once.Do(f) is called multiple times, only the first call will invoke f, even if f has a different value in each invocation. A new instance of Once is required for each function to execute.

Do is intended for initialization that must be run exactly once. Since f is niladic, it may be necessary to use a function literal to capture the arguments to a function to be invoked by Do:

```
config.once.Do(func() { config.init(filename) })
```

Because no call to Do returns until the one call to f returns, if f causes Do to be called, it will deadlock.

If f panics, Do considers it to have returned; future calls of Do return without calling f.

## type Pool

```
type Pool struct {

 // New optionally specifies a function to generate
 // a value when Get would otherwise return nil.
 // It may not be changed concurrently with calls to Get.
 New func() interface{}
 // contains filtered or unexported fields
}
```

A Pool is a set of temporary objects that may be individually saved and retrieved.

Any item stored in the Pool may be removed automatically at any time without notification. If the Pool holds the only reference when this happens, the item might be deallocated.

A Pool is safe for use by multiple goroutines simultaneously.

Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector. That is, it makes it easy to build efficient, thread-safe free lists. However, it is not suitable for all free lists.

An appropriate use of a Pool is to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package. Pool provides a way to amortize allocation overhead across many clients.

An example of good use of a Pool is in the fmt package, which maintains a dynamically-sized store of temporary output buffers. The store scales under load (when many goroutines are actively printing) and shrinks when quiescent.

On the other hand, a free list maintained as part of a short-lived object is not a suitable use for a Pool, since the overhead does not amortize well in that scenario. It is more efficient to have such objects implement their own free list.

A Pool must not be copied after first use.

## func (\*Pool) Get

```
func (p *Pool) Get() interface{}
```

Get selects an arbitrary item from the Pool, removes it from the Pool, and returns it to the caller. Get may choose to ignore the pool and treat it as empty. Callers should not assume any relation between values passed to Put and the values returned by Get.

If Get would otherwise return nil and p.New is non-nil, Get returns the result of calling p.New.

## func (\*Pool) Put

```
func (p *Pool) Put(x interface{})
```

Put adds x to the pool.

## type RWMutex

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

A RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a RWMutex is an unlocked mutex.

A RWMutex must not be copied after first use.

If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released. In particular, this prohibits recursive read locking. This is to ensure that the lock eventually becomes available; a blocked Lock call excludes new readers from acquiring the lock.

## func (\*RWMutex) Lock

```
func (rw *RWMutex) Lock()
```

Lock locks rw for writing. If the lock is already locked for reading or writing, Lock blocks until the lock is available.

## func (\*RWMutex) RLock

```
func (rw *RWMutex) RLock()
```

RLock locks rw for reading.

It should not be used for recursive read locking; a blocked Lock call excludes new readers from acquiring the lock. See the documentation on the RWMutex type.

## func (\*RWMutex) **RLocker**

```
func (rw *RWMutex) RLocker() Locker
```

RLocker returns a Locker interface that implements the Lock and Unlock methods by calling rw.RLock and rw.RUnlock.

## func (\*RWMutex) **RUnlock**

```
func (rw *RWMutex) RUnlock()
```

RUnlock undoes a single RLock call; it does not affect other simultaneous readers. It is a run-time error if rw is not locked for reading on entry to RUnlock.

## func (\*RWMutex) **Unlock**

```
func (rw *RWMutex) Unlock()
```

Unlock unlocks rw for writing. It is a run-time error if rw is not locked for writing on entry to Unlock.

As with Mutexes, a locked RWMutex is not associated with a particular goroutine. One goroutine may RLock (Lock) a RWMutex and then arrange for another goroutine to RUnlock (Unlock) it.

## type **WaitGroup**

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

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

A WaitGroup must not be copied after first use.

## func (\*WaitGroup) **Add**

```
func (wg *WaitGroup) Add(delta int)
```

Add adds delta, which may be negative, to the WaitGroup counter. If the counter becomes zero, all goroutines blocked on Wait are released. If the counter goes negative, Add panics.

Note that calls with a positive delta that occur when the counter is zero must happen before a Wait. Calls with a negative delta, or calls with a positive delta that start when the counter is greater than zero, may happen at any time. Typically this means the calls to Add should execute before the statement creating the goroutine or other event to be waited for. If a WaitGroup is reused to wait for several independent sets of events, new Add calls must happen after all previous Wait calls have returned. See the WaitGroup example.

## func (\*WaitGroup) Done

```
func (wg *WaitGroup) Done()
```

Done decrements the WaitGroup counter by one.

## func (\*WaitGroup) Wait

```
func (wg *WaitGroup) Wait()
```

Wait blocks until the WaitGroup counter is zero.

# Package atomic

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package. Share memory by communicating; don't communicate by sharing memory.

The swap operation, implemented by the SwapT functions, is the atomic equivalent of:

```
old = *addr
*addr = new
return old
```

The compare-and-swap operation, implemented by the CompareAndSwapT functions, is the atomic equivalent of:

```
if *addr == old {
 *addr = new
 return true
}
return false
```

The add operation, implemented by the AddT functions, is the atomic equivalent of:

```
*addr += delta
return *addr
```

The load and store operations, implemented by the LoadT and StoreT functions, are the atomic equivalents of "return \*addr" and "\*addr = val".

### func AddInt32

```
func AddInt32(addr *int32, delta int32) (new int32)
```

AddInt32 atomically adds delta to \*addr and returns the new value.

### func AddInt64

```
func AddInt64(addr *int64, delta int64) (new int64)
```

AddInt64 atomically adds delta to \*addr and returns the new value.

## func AddUint32

```
func AddUint32(addr *uint32, delta uint32) (new uint32)
```

AddUint32 atomically adds delta to \*addr and returns the new value. To subtract a signed positive constant value c from x, do AddUint32(&x, ^uint32(c-1)). In particular, to decrement x, do AddUint32(&x, ^uint32(0)).

## func AddUint64

```
func AddUint64(addr *uint64, delta uint64) (new uint64)
```

AddUint64 atomically adds delta to \*addr and returns the new value. To subtract a signed positive constant value c from x, do AddUint64(&x, ^uint64(c-1)). In particular, to decrement x, do AddUint64(&x, ^uint64(0)).

## func AddUintptr

```
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
```

AddUintptr atomically adds delta to \*addr and returns the new value.

## func CompareAndSwapInt32

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
```

CompareAndSwapInt32 executes the compare-and-swap operation for an int32 value.

## func CompareAndSwapInt64

```
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
```

CompareAndSwapInt64 executes the compare-and-swap operation for an int64 value.

## func CompareAndSwapPointer

```
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
```

CompareAndSwapPointer executes the compare-and-swap operation for a unsafe.Pointer value.

## func CompareAndSwapUint32

```
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
```

CompareAndSwapUint32 executes the compare-and-swap operation for a uint32 value.

## func CompareAndSwapUint64

```
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
```

CompareAndSwapUint64 executes the compare-and-swap operation for a uint64 value.

## func CompareAndSwapUintptr

```
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)
```

CompareAndSwapUintptr executes the compare-and-swap operation for a uintptr value.

## func LoadInt32

```
func LoadInt32(addr *int32) (val int32)
```

LoadInt32 atomically loads \*addr.

## func LoadInt64

```
func LoadInt64(addr *int64) (val int64)
```

LoadInt64 atomically loads \*addr.

## func LoadPointer

```
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
```

LoadPointer atomically loads \*addr.

## func LoadUint32

```
func LoadUint32(addr *uint32) (val uint32)
```

LoadUint32 atomically loads \*addr.

## func LoadUint64

```
func LoadUint64(addr *uint64) (val uint64)
```

LoadUint64 atomically loads \*addr.

## func `LoadUintptr`

```
func LoadUintptr(addr *uintptr) (val uintptr)
```

`LoadUintptr` atomically loads `*addr`.

## func `StoreInt32`

```
func StoreInt32(addr *int32, val int32)
```

`StoreInt32` atomically stores `val` into `*addr`.

## func `StoreInt64`

```
func StoreInt64(addr *int64, val int64)
```

`StoreInt64` atomically stores `val` into `*addr`.

## func `StorePointer`

```
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
```

`StorePointer` atomically stores `val` into `*addr`.

## func `StoreUint32`

```
func StoreUint32(addr *uint32, val uint32)
```

`StoreUint32` atomically stores `val` into `*addr`.

## func `StoreUint64`

```
func StoreUint64(addr *uint64, val uint64)
```

`StoreUint64` atomically stores `val` into `*addr`.

## func `StoreUintptr`

```
func StoreUintptr(addr *uintptr, val uintptr)
```

`StoreUintptr` atomically stores `val` into `*addr`.

## func `SwapInt32`

```
func SwapInt32(addr *int32, new int32) (old int32)
```

SwapInt32 atomically stores new into \*addr and returns the previous \*addr value.

## func SwapInt64

```
func SwapInt64(addr *int64, new int64) (old int64)
```

SwapInt64 atomically stores new into \*addr and returns the previous \*addr value.

## func SwapPointer

```
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
```

SwapPointer atomically stores new into \*addr and returns the previous \*addr value.

## func SwapUint32

```
func SwapUint32(addr *uint32, new uint32) (old uint32)
```

SwapUint32 atomically stores new into \*addr and returns the previous \*addr value.

## func SwapUint64

```
func SwapUint64(addr *uint64, new uint64) (old uint64)
```

SwapUint64 atomically stores new into \*addr and returns the previous \*addr value.

## func SwapUintptr

```
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

SwapUintptr atomically stores new into \*addr and returns the previous \*addr value.

## type Value

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

A Value provides an atomic load and store of a consistently typed value. The zero value for a Value returns nil from Load. Once Store has been called, a Value must not be copied.

A Value must not be copied after first use.

## func (\*Value) Load

```
func (v *Value) Load() (x interface{})
```

Load returns the value set by the most recent Store. It returns nil if there has been no call to Store for this Value.

## func (\*Value) Store

```
func (v *Value) Store(x interface{})
```

Store sets the value of the Value to x. All calls to Store for a given Value must use values of the same concrete type. Store of an inconsistent type panics, as does Store(nil).

## BUGs

- On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.

On non-Linux ARM, the 64-bit functions use instructions unavailable before the ARMv6k core.

On ARM, x86-32, and 32-bit MIPS, it is the caller's responsibility to arrange for 64-bit alignment of 64-bit words accessed atomically. The first word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned.

# Package syscall

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package syscall contains an interface to the low-level operating system primitives. The details vary depending on the underlying system, and by default, godoc will display the syscall documentation for the current system. If you want godoc to display syscall documentation for another system, set \$GOOS and \$GOARCH to the desired system. For example, if you want to view documentation for freebsd/arm on linux/amd64, set \$GOOS to freebsd and \$GOARCH to arm. The primary use of syscall is inside other packages that provide a more portable interface to the system, such as "os", "time" and "net". Use those packages rather than this one if you can. For details of the functions and data types in this package consult the manuals for the appropriate operating system. These calls return err == nil to indicate success; otherwise err is an operating system error describing the failure. On most systems, that error has type syscall.Errno.

Deprecated: this package is locked down. Callers should use the corresponding package in the [golang.org/x/sys](https://golang.org/x/sys) repository instead. That is also where updates required by new systems or versions should be applied. See <https://golang.org/s/go1.4-syscall> for more information.

## Constants

```
const (
 AF_ALG = 0x26
 AF_APPLETALK = 0x5
 AF_ASH = 0x12
 AF_ATMPVC = 0x8
 AF_ATMSVC = 0x14
 AF_AX25 = 0x3
 AF_BLUETOOTH = 0x1f
 AF_BRIDGE = 0x7
 AF_CAIF = 0x25
 AF_CAN = 0x1d
 AF_DECnet = 0xc
 AF_ECONET = 0x13
 AF_FILE = 0x1
 AF_IEEE802154 = 0x24
 AF_INET = 0x2
 AF_INET6 = 0xa
 AF_IPX = 0x4
 AF_IRDA = 0x17
 AF_ISDN = 0x22
 AF_IUCV = 0x20
 AF_KEY = 0xf
 AF_LLC = 0x1a
```

|                            |         |
|----------------------------|---------|
| AF_LOCAL                   | = 0x1   |
| AF_MAX                     | = 0x27  |
| AF_NETBEUI                 | = 0xd   |
| AF_NETLINK                 | = 0x10  |
| AF_NETROM                  | = 0x6   |
| AF_PACKET                  | = 0x11  |
| AF_PHONET                  | = 0x23  |
| AF_PPPOX                   | = 0x18  |
| AF_RDS                     | = 0x15  |
| AF_ROSE                    | = 0xb   |
| AF_ROUTE                   | = 0x10  |
| AF_RXRPC                   | = 0x21  |
| AF_SECURITY                | = 0xe   |
| AF_SNA                     | = 0x16  |
| AF_TIPC                    | = 0x1e  |
| AF_UNIX                    | = 0x1   |
| AF_UNSPEC                  | = 0x0   |
| AF_WANPIPE                 | = 0x19  |
| AF_X25                     | = 0x9   |
| ARPHRD_ADAPTER             | = 0x108 |
| ARPHRD_APPLETALK           | = 0x8   |
| ARPHRD_ARCNET              | = 0x7   |
| ARPHRD_ASH                 | = 0x30d |
| ARPHRD_ATM                 | = 0x13  |
| ARPHRD_AX25                | = 0x3   |
| ARPHRD_BIF                 | = 0x307 |
| ARPHRD_CHAOS               | = 0x5   |
| ARPHRD_CISCO               | = 0x201 |
| ARPHRD_CSCLIP              | = 0x101 |
| ARPHRD_CSCLIP6             | = 0x103 |
| ARPHRD_DDCMP               | = 0x205 |
| ARPHRD_DLCl                | = 0xf   |
| ARPHRD_ECONET              | = 0x30e |
| ARPHRD_EETHER              | = 0x2   |
| ARPHRD_ETHER               | = 0x1   |
| ARPHRD_EUI64               | = 0x1b  |
| ARPHRD_FCAL                | = 0x311 |
| ARPHRD_FCFABRIC            | = 0x313 |
| ARPHRD_FCPL                | = 0x312 |
| ARPHRD_FCPP                | = 0x310 |
| ARPHRD_FDDI                | = 0x306 |
| ARPHRD_FRAD                | = 0x302 |
| ARPHRD_HDLC                | = 0x201 |
| ARPHRD_HIPPI               | = 0x30c |
| ARPHRD_HWX25               | = 0x110 |
| ARPHRD_IEEE1394            | = 0x18  |
| ARPHRD_IEEE802             | = 0x6   |
| ARPHRD_IEEE80211           | = 0x321 |
| ARPHRD_IEEE80211_PRISM     | = 0x322 |
| ARPHRD_IEEE80211_RADIO TAP | = 0x323 |
| ARPHRD_IEEE802154          | = 0x324 |
| ARPHRD_IEEE802154_PHY      | = 0x325 |
| ARPHRD_IEEE802_TR          | = 0x320 |
| ARPHRD_INFINIBAND          | = 0x20  |

|                   |           |
|-------------------|-----------|
| ARPHRD_IPDDP      | = 0x309   |
| ARPHRD_IPGRE      | = 0x30a   |
| ARPHRD_IRDA       | = 0x30f   |
| ARPHRD_LAPB       | = 0x204   |
| ARPHRD_LOCALTLK   | = 0x305   |
| ARPHRD_LOOPBACK   | = 0x304   |
| ARPHRD_METRICOM   | = 0x17    |
| ARPHRD_NETROM     | = 0x0     |
| ARPHRD_NONE       | = 0xffffe |
| ARPHRD_PIMREG     | = 0x30b   |
| ARPHRD_PPP        | = 0x200   |
| ARPHRD_PRONET     | = 0x4     |
| ARPHRD_RAWHDLC    | = 0x206   |
| ARPHRD_ROSE       | = 0x10e   |
| ARPHRD_RSRVD      | = 0x104   |
| ARPHRD_SIT        | = 0x308   |
| ARPHRD_SKIP       | = 0x303   |
| ARPHRD_SLIP       | = 0x100   |
| ARPHRD_SLIP6      | = 0x102   |
| ARPHRD_TUNNEL     | = 0x300   |
| ARPHRD_TUNNEL6    | = 0x301   |
| ARPHRD_VOID       | = 0xfffff |
| ARPHRD_X25        | = 0x10f   |
| BPF_A             | = 0x10    |
| BPF_ABS           | = 0x20    |
| BPF_ADD           | = 0x0     |
| BPF_ALU           | = 0x4     |
| BPF_AND           | = 0x50    |
| BPF_B             | = 0x10    |
| BPF_DIV           | = 0x30    |
| BPF_H             | = 0x8     |
| BPF_IMM           | = 0x0     |
| BPF_IND           | = 0x40    |
| BPF_JA            | = 0x0     |
| BPF_JEQ           | = 0x10    |
| BPF_JGE           | = 0x30    |
| BPF_JGT           | = 0x20    |
| BPF_JMP           | = 0x5     |
| BPF_JSET          | = 0x40    |
| BPF_K             | = 0x0     |
| BPF_LD            | = 0x0     |
| BPF_LDX           | = 0x1     |
| BPF_LEN           | = 0x80    |
| BPF_LSH           | = 0x60    |
| BPF_MAJOR_VERSION | = 0x1     |
| BPF_MAXINSNS      | = 0x1000  |
| BPF_MEM           | = 0x60    |
| BPF_MEMWORDS      | = 0x10    |
| BPF_MINOR_VERSION | = 0x1     |
| BPF_MISC          | = 0x7     |
| BPF_MSH           | = 0xa0    |
| BPF_MUL           | = 0x20    |
| BPF_NEG           | = 0x80    |
| BPF_OR            | = 0x40    |

|                      |                |
|----------------------|----------------|
| BPF_RET              | = 0x6          |
| BPF_RSH              | = 0x70         |
| BPF_ST               | = 0x2          |
| BPF_STX              | = 0x3          |
| BPF_SUB              | = 0x10         |
| BPF_TAX              | = 0x0          |
| BPF_TXA              | = 0x80         |
| BPF_W                | = 0x0          |
| BPF_X                | = 0x8          |
| CLONE_CHILD_CLEARTID | = 0x2000000    |
| CLONE_CHILD_SETTID   | = 0x1000000    |
| CLONE_DETACHED       | = 0x400000     |
| CLONE_FILES          | = 0x400        |
| CLONE_FS             | = 0x200        |
| CLONE_IO             | = 0x800000000  |
| CLONE_NEWIPC         | = 0x8000000    |
| CLONE_NEWWET         | = 0x400000000  |
| CLONE_NEWNS          | = 0x20000      |
| CLONE_NEWPID         | = 0x200000000  |
| CLONE_NEWUSER        | = 0x100000000  |
| CLONE_NEWUTS         | = 0x40000000   |
| CLONE_PARENT         | = 0x8000       |
| CLONE_PARENT_SETTID  | = 0x100000     |
| CLONE_PTRACE         | = 0x2000       |
| CLONE_SETTLS         | = 0x80000      |
| CLONE_SIGHAND        | = 0x800        |
| CLONE_SYSVSEM        | = 0x40000      |
| CLONE_THREAD         | = 0x10000      |
| CLONE_UNTRACED       | = 0x800000     |
| CLONE_VFORK          | = 0x4000       |
| CLONE_VM             | = 0x100        |
| DT_BLK               | = 0x6          |
| DT_CHR               | = 0x2          |
| DT_DIR               | = 0x4          |
| DT_FIFO              | = 0x1          |
| DT_LNK               | = 0xa          |
| DT_REG               | = 0x8          |
| DT_SOCK              | = 0xc          |
| DT_UNKNOWN           | = 0x0          |
| DT_WHT               | = 0xe          |
| EPOLLERR             | = 0x8          |
| EPOLLET              | = -0x800000000 |
| EPOLLHUP             | = 0x10         |
| EPOLLIN              | = 0x1          |
| EPOLLMSG             | = 0x400        |
| EPOLLONESHOT         | = 0x400000000  |
| EPOLLOUT             | = 0x4          |
| EPOLLPRI             | = 0x2          |
| EPOLLRDBAND          | = 0x80         |
| EPOLLRDHUP           | = 0x2000       |
| EPOLLRDNORM          | = 0x40         |
| EPOLLWRBAND          | = 0x200        |
| EPOLLWRNORM          | = 0x100        |
| EPOLL_CLOEXEC        | = 0x80000      |

|                  |          |
|------------------|----------|
| EPOLL_CTL_ADD    | = 0x1    |
| EPOLL_CTL_DEL    | = 0x2    |
| EPOLL_CTL_MOD    | = 0x3    |
| EPOLL_NONBLOCK   | = 0x800  |
| ETH_P_1588       | = 0x88f7 |
| ETH_P_8021Q      | = 0x8100 |
| ETH_P_802_2      | = 0x4    |
| ETH_P_802_3      | = 0x1    |
| ETH_P_AARP       | = 0x80f3 |
| ETH_P_ALL        | = 0x3    |
| ETH_P_AOE        | = 0x88a2 |
| ETH_P_ARCNET     | = 0x1a   |
| ETH_P_ARP        | = 0x806  |
| ETH_P_ATALK      | = 0x809b |
| ETH_P_ATMFATE    | = 0x8884 |
| ETH_P_ATMMPOA    | = 0x884c |
| ETH_P_AX25       | = 0x2    |
| ETH_P_BPQ        | = 0x8ff  |
| ETH_P_CAIF       | = 0xf7   |
| ETH_P_CAN        | = 0xc    |
| ETH_P_CONTROL    | = 0x16   |
| ETH_P_CUST       | = 0x6006 |
| ETH_P_DDCMP      | = 0x6    |
| ETH_P_DEC        | = 0x6000 |
| ETH_P_DIAG       | = 0x6005 |
| ETH_P_DNA_DL     | = 0x6001 |
| ETH_P_DNA_RC     | = 0x6002 |
| ETH_P_DNA_RT     | = 0x6003 |
| ETH_P_DSA        | = 0x1b   |
| ETH_P_ECONET     | = 0x18   |
| ETH_P_EDSA       | = 0xdada |
| ETH_P_FCOE       | = 0x8906 |
| ETH_P_FIP        | = 0x8914 |
| ETH_P_HDLC       | = 0x19   |
| ETH_P_IEEE802154 | = 0xf6   |
| ETH_P_IEEEPUP    | = 0xa00  |
| ETH_P_IEEEPUPAT  | = 0xa01  |
| ETH_P_IP         | = 0x800  |
| ETH_P_IPV6       | = 0x86dd |
| ETH_P_IPX        | = 0x8137 |
| ETH_P_IRDA       | = 0x17   |
| ETH_P_LAT        | = 0x6004 |
| ETH_P_LINK_CTL   | = 0x886c |
| ETH_P_LOCALTALK  | = 0x9    |
| ETH_P_LOOP       | = 0x60   |
| ETH_P_MOBITEX    | = 0x15   |
| ETH_P_MPLS_MC    | = 0x8848 |
| ETH_P_MPLS_UC    | = 0x8847 |
| ETH_P_PAE        | = 0x888e |
| ETH_P_PAUSE      | = 0x8808 |
| ETH_P_PHONET     | = 0xf5   |
| ETH_P_PPPTALK    | = 0x10   |
| ETH_P_PPP_DISC   | = 0x8863 |
| ETH_P_PPP_MP     | = 0x8    |

|                   |          |
|-------------------|----------|
| ETH_P_PPP_SES     | = 0x8864 |
| ETH_P_PUP         | = 0x200  |
| ETH_P_PUPAT       | = 0x201  |
| ETH_P_RARP        | = 0x8035 |
| ETH_P_SCA         | = 0x6007 |
| ETH_P_SLOW        | = 0x8809 |
| ETH_P_SNAP        | = 0x5    |
| ETH_P_TEB         | = 0x6558 |
| ETH_P_TIPC        | = 0x88ca |
| ETH_P_TRAILER     | = 0x1c   |
| ETH_P_TR_802_2    | = 0x11   |
| ETH_P_WAN_PPP     | = 0x7    |
| ETH_P_WCCP        | = 0x883e |
| ETH_P_X25         | = 0x805  |
| FD_CLOEXEC        | = 0x1    |
| FD_SETSIZE        | = 0x400  |
| F_DUPFD           | = 0x0    |
| F_DUPFD_CLOEXEC   | = 0x406  |
| F_EXLCK           | = 0x4    |
| F_GETFD           | = 0x1    |
| F_GETFL           | = 0x3    |
| F_GETLEASE        | = 0x401  |
| F_GETLK           | = 0x5    |
| F_GETLK64         | = 0x5    |
| F_GETOWN          | = 0x9    |
| F_GETOWN_EX       | = 0x10   |
| F_GETPIPE_SZ      | = 0x408  |
| F_GETSIG          | = 0xb    |
| F_LOCK            | = 0x1    |
| F_NOTIFY          | = 0x402  |
| F_OK              | = 0x0    |
| F_RDLCK           | = 0x0    |
| F_SETFD           | = 0x2    |
| F_SETFL           | = 0x4    |
| F_SETLEASE        | = 0x400  |
| F_SETLK           | = 0x6    |
| F_SETLK64         | = 0x6    |
| F_SETLKW          | = 0x7    |
| F_SETLKW64        | = 0x7    |
| F_SETOWN          | = 0x8    |
| F_SETOWN_EX       | = 0xf    |
| F_SETPIPE_SZ      | = 0x407  |
| F_SETSIG          | = 0xa    |
| F_SHLCK           | = 0x8    |
| F_TEST            | = 0x3    |
| F_TLOCK           | = 0x2    |
| F_ULOCK           | = 0x0    |
| F_UNLCK           | = 0x2    |
| F_WRLCK           | = 0x1    |
| ICMPV6_FILTER     | = 0x1    |
| IFA_F_DADFAILED   | = 0x8    |
| IFA_F_DEPRECATED  | = 0x20   |
| IFA_F_HOMEADDRESS | = 0x10   |
| IFA_F_NODAD       | = 0x2    |

|                  |                |
|------------------|----------------|
| IFA_F_OPTIMISTIC | = 0x4          |
| IFA_F_PERMANENT  | = 0x80         |
| IFA_F_SECONDARY  | = 0x1          |
| IFA_F_TEMPORARY  | = 0x1          |
| IFA_F_TENTATIVE  | = 0x40         |
| IFA_MAX          | = 0x7          |
| IFF_ALLMULTI     | = 0x200        |
| IFF_AUTOMEDIA    | = 0x4000       |
| IFF_BROADCAST    | = 0x2          |
| IFF_DEBUG        | = 0x4          |
| IFF_DYNAMIC      | = 0x8000       |
| IFF_LOOPBACK     | = 0x8          |
| IFF_MASTER       | = 0x400        |
| IFF_MULTICAST    | = 0x1000       |
| IFF_NOARP        | = 0x80         |
| IFF_NOTRAILERS   | = 0x20         |
| IFF_NO_PI        | = 0x1000       |
| IFF_ONE_QUEUE    | = 0x2000       |
| IFF_POINTOPOINT  | = 0x10         |
| IFF_PORTSEL      | = 0x2000       |
| IFF_PROMISC      | = 0x100        |
| IFF_RUNNING      | = 0x40         |
| IFF_SLAVE        | = 0x800        |
| IFF_TAP          | = 0x2          |
| IFF_TUN          | = 0x1          |
| IFF_TUN_EXCL     | = 0x8000       |
| IFF_UP           | = 0x1          |
| IFF_VNET_HDR     | = 0x4000       |
| IFNAMSIZ         | = 0x10         |
| IN_ACCESS        | = 0x1          |
| IN_ALL_EVENTS    | = 0xffff       |
| IN_ATTRIB        | = 0x4          |
| IN_CLASSA_HOST   | = 0xffffffff   |
| IN_CLASSA_MAX    | = 0x80         |
| IN_CLASSA_NET    | = 0xff000000   |
| IN_CLASSA_NSHIFT | = 0x18         |
| IN_CLASSB_HOST   | = 0xffff       |
| IN_CLASSB_MAX    | = 0x10000      |
| IN_CLASSB_NET    | = 0xffff0000   |
| IN_CLASSB_NSHIFT | = 0x10         |
| IN_CLASSC_HOST   | = 0xff         |
| IN_CLASSC_NET    | = 0xffffffff00 |
| IN_CLASSC_NSHIFT | = 0x8          |
| IN_CLOEXEC       | = 0x80000      |
| IN_CLOSE         | = 0x18         |
| IN_CLOSE_NOWRITE | = 0x10         |
| IN_CLOSE_WRITE   | = 0x8          |
| IN_CREATE        | = 0x100        |
| IN_DELETE        | = 0x200        |
| IN_DELETE_SELF   | = 0x400        |
| IN_DONT_FOLLOW   | = 0x20000000   |
| IN_EXCL_UNLINK   | = 0x40000000   |
| IN_IGNORED       | = 0x8000       |
| IN_ISDIR         | = 0x40000000   |

|                      |               |
|----------------------|---------------|
| IN_LOOPBACKNET       | = 0x7f        |
| IN_MASK_ADD          | = 0x200000000 |
| IN MODIFY            | = 0x2         |
| IN_MOVE              | = 0xc0        |
| IN_MOVED_FROM        | = 0x40        |
| IN_MOVED_TO          | = 0x80        |
| IN_MOVE_SELF         | = 0x800       |
| IN_NONBLOCK          | = 0x800       |
| IN_ONESHOT           | = 0x800000000 |
| IN_ONLYDIR           | = 0x1000000   |
| IN_OPEN              | = 0x20        |
| IN_Q_OVERFLOW        | = 0x4000      |
| IN_UNMOUNT           | = 0x2000      |
| IPPROTO_AH           | = 0x33        |
| IPPROTO_COMP         | = 0x6c        |
| IPPROTO_DCCP         | = 0x21        |
| IPPROTO_DSTOPTS      | = 0x3c        |
| IPPROTO_EGP          | = 0x8         |
| IPPROTO_ENCAP        | = 0x62        |
| IPPROTO_ESP          | = 0x32        |
| IPPROTO_FRAGMENT     | = 0x2c        |
| IPPROTO_GRE          | = 0x2f        |
| IPPROTO_HOPOPTS      | = 0x0         |
| IPPROTO_ICMP         | = 0x1         |
| IPPROTO_ICMPV6       | = 0x3a        |
| IPPROTO_IDP          | = 0x16        |
| IPPROTO_IGMP         | = 0x2         |
| IPPROTO_IP           | = 0x0         |
| IPPROTO_IPIP         | = 0x4         |
| IPPROTO_IPV6         | = 0x29        |
| IPPROTO_MTP          | = 0x5c        |
| IPPROTO_NONE         | = 0x3b        |
| IPPROTO_PIM          | = 0x67        |
| IPPROTO_PUP          | = 0xc         |
| IPPROTO_RAW          | = 0xff        |
| IPPROTO_ROUTING      | = 0x2b        |
| IPPROTO_RSVP         | = 0x2e        |
| IPPROTO_SCTP         | = 0x84        |
| IPPROTO_TCP          | = 0x6         |
| IPPROTO_TP           | = 0x1d        |
| IPPROTO_UDP          | = 0x11        |
| IPPROTO_UDPLITE      | = 0x88        |
| IPV6_2292DSTOPTS     | = 0x4         |
| IPV6_2292HOPLIMIT    | = 0x8         |
| IPV6_2292HOPOPTS     | = 0x3         |
| IPV6_2292PKTINFO     | = 0x2         |
| IPV6_2292PKTOPTIONS  | = 0x6         |
| IPV6_2292RTHDR       | = 0x5         |
| IPV6_ADDRFORM        | = 0x1         |
| IPV6_ADD_MEMBERSHIP  | = 0x14        |
| IPV6_AUTHHDR         | = 0xa         |
| IPV6_CHECKSUM        | = 0x7         |
| IPV6_DROP_MEMBERSHIP | = 0x15        |
| IPV6_DSTOPTS         | = 0x3b        |

|                           |          |
|---------------------------|----------|
| IPV6_HOPLIMIT             | = 0x34   |
| IPV6_HOPOPTS              | = 0x36   |
| IPV6_IPSEC_POLICY         | = 0x22   |
| IPV6_JOIN_ANYCAST         | = 0x1b   |
| IPV6_JOIN_GROUP           | = 0x14   |
| IPV6_LEAVE_ANYCAST        | = 0x1c   |
| IPV6_LEAVE_GROUP          | = 0x15   |
| IPV6_MTU                  | = 0x18   |
| IPV6_MTU_DISCOVER         | = 0x17   |
| IPV6_MULTICAST_HOPS       | = 0x12   |
| IPV6_MULTICAST_IF         | = 0x11   |
| IPV6_MULTICAST_LOOP       | = 0x13   |
| IPV6_NEXTHOP              | = 0x9    |
| IPV6_PKTINFO              | = 0x32   |
| IPV6_PMTUDISC_DO          | = 0x2    |
| IPV6_PMTUDISC_DONT        | = 0x0    |
| IPV6_PMTUDISC_PROBE       | = 0x3    |
| IPV6_PMTUDISC_WANT        | = 0x1    |
| IPV6_RECVDSTOPTS          | = 0x3a   |
| IPV6_RECVVERR             | = 0x19   |
| IPV6_RECVHOPLIMIT         | = 0x33   |
| IPV6_RECVHOPOPTS          | = 0x35   |
| IPV6_RECVPKTINFO          | = 0x31   |
| IPV6_RECVRTHDR            | = 0x38   |
| IPV6_RECVTCLASS           | = 0x42   |
| IPV6_ROUTER_ALERT         | = 0x16   |
| IPV6_RTHDR                | = 0x39   |
| IPV6_RTHDRDSTOPTS         | = 0x37   |
| IPV6_RTHDR_LOOSE          | = 0x0    |
| IPV6_RTHDR_STRICT         | = 0x1    |
| IPV6_RTHDR_TYPE_0         | = 0x0    |
| IPV6_RXDSTOPTS            | = 0x3b   |
| IPV6_RXHOPOPTS            | = 0x36   |
| IPV6_TCLASS               | = 0x43   |
| IPV6_UNICAST_HOPS         | = 0x10   |
| IPV6_V6ONLY               | = 0x1a   |
| IPV6_XFRM_POLICY          | = 0x23   |
| IP_ADD_MEMBERSHIP         | = 0x23   |
| IP_ADD_SOURCE_MEMBERSHIP  | = 0x27   |
| IP_BLOCK_SOURCE           | = 0x26   |
| IP_DEFAULT_MULTICAST_LOOP | = 0x1    |
| IP_DEFAULT_MULTICAST_TTL  | = 0x1    |
| IP_DF                     | = 0x4000 |
| IP_DROP_MEMBERSHIP        | = 0x24   |
| IP_DROP_SOURCE_MEMBERSHIP | = 0x28   |
| IP_FREEBIND               | = 0xf    |
| IP_HDRINCL                | = 0x3    |
| IP_IPSEC_POLICY           | = 0x10   |
| IP_MAXPACKET              | = 0xffff |
| IP_MAX_MEMBERSHIPS        | = 0x14   |
| IP_MF                     | = 0x2000 |
| IP_MINTTL                 | = 0x15   |
| IP_MSFILTER               | = 0x29   |
| IP_MSS                    | = 0x240  |

|                             |              |
|-----------------------------|--------------|
| IP_MTU                      | = 0xe        |
| IP_MTU_DISCOVER             | = 0xa        |
| IP_MULTICAST_IF             | = 0x20       |
| IP_MULTICAST_LOOP           | = 0x22       |
| IP_MULTICAST_TTL            | = 0x21       |
| IP_OFFSET                   | = 0x1fff     |
| IP_OPTIONS                  | = 0x4        |
| IP_ORIGDSTADDR              | = 0x14       |
| IP_PASSEC                   | = 0x12       |
| IP_PKTINFO                  | = 0x8        |
| IP_PKTOPTIONS               | = 0x9        |
| IP_PMTUDISC                 | = 0xa        |
| IP_PMTUDISC_DO              | = 0x2        |
| IP_PMTUDISC_DONT            | = 0x0        |
| IP_PMTUDISC_PROBE           | = 0x3        |
| IP_PMTUDISC_WANT            | = 0x1        |
| IP_RECVERR                  | = 0xb        |
| IP_RECVOPTS                 | = 0x6        |
| IP_RECVORIGDSTADDR          | = 0x14       |
| IP_RECVRETOPTS              | = 0x7        |
| IP_RECVTOS                  | = 0xd        |
| IP_RECVTTL                  | = 0xc        |
| IP_RECVTOPTS                | = 0x7        |
| IP_RF                       | = 0x8000     |
| IP_ROUTER_ALERT             | = 0x5        |
| IP_TOS                      | = 0x1        |
| IP_TRANSPARENT              | = 0x13       |
| IP_TTL                      | = 0x2        |
| IP_UNBLOCK_SOURCE           | = 0x25       |
| IP_XFRM_POLICY              | = 0x11       |
| LINUX_REBOOT_CMD_CAD_OFF    | = 0x0        |
| LINUX_REBOOT_CMD_CAD_ON     | = 0x89abcdef |
| LINUX_REBOOT_CMD_HALT       | = 0xcdef0123 |
| LINUX_REBOOT_CMD_KEXEC      | = 0x45584543 |
| LINUX_REBOOT_CMD_POWER_OFF  | = 0x4321fedc |
| LINUX_REBOOT_CMD_RESTART    | = 0x1234567  |
| LINUX_REBOOT_CMD_RESTART2   | = 0xa1b2c3d4 |
| LINUX_REBOOT_CMD_SW_SUSPEND | = 0xd000fce2 |
| LINUX_REBOOT_MAGIC1         | = 0xfee1dead |
| LINUX_REBOOT_MAGIC2         | = 0x28121969 |
| LOCK_EX                     | = 0x2        |
| LOCK_NB                     | = 0x4        |
| LOCK_SH                     | = 0x1        |
| LOCK_UN                     | = 0x8        |
| MADV_DOFORK                 | = 0xb        |
| MADV_DONTFORK               | = 0xa        |
| MADV_DONTNEED               | = 0x4        |
| MADV_HUGEPAGE               | = 0xe        |
| MADV_HWPOISON               | = 0x64       |
| MADV_MERGEABLE              | = 0xc        |
| MADV_NOHUGEPAGE             | = 0xf        |
| MADV_NORMAL                 | = 0x0        |
| MADV_RANDOM                 | = 0x1        |
| MADV_REMOVE                 | = 0x9        |

|                  |               |
|------------------|---------------|
| MADV_SEQUENTIAL  | = 0x2         |
| MADV_UNMERGEABLE | = 0xd         |
| MADV_WILLNEED    | = 0x3         |
| MAP_32BIT        | = 0x40        |
| MAP_ANON         | = 0x20        |
| MAP_ANONYMOUS    | = 0x20        |
| MAP_DENYWRITE    | = 0x800       |
| MAP_EXECUTABLE   | = 0x1000      |
| MAP_FILE         | = 0x0         |
| MAP_FIXED        | = 0x10        |
| MAP_GROWSDOWN    | = 0x100       |
| MAP_HUGETLB      | = 0x40000     |
| MAP_LOCKED       | = 0x2000      |
| MAP_NONBLOCK     | = 0x10000     |
| MAP_NORESERVE    | = 0x4000      |
| MAP_POPULATE     | = 0x8000      |
| MAP_PRIVATE      | = 0x2         |
| MAP_SHARED       | = 0x1         |
| MAP_STACK        | = 0x20000     |
| MAP_TYPE         | = 0xf         |
| MCL_CURRENT      | = 0x1         |
| MCL_FUTURE       | = 0x2         |
| MNT_DETACH       | = 0x2         |
| MNT_EXPIRE       | = 0x4         |
| MNT_FORCE        | = 0x1         |
| MSG_CMSG_CLOEXEC | = 0x40000000  |
| MSG_CONFIRM      | = 0x800       |
| MSG_CTRUNC       | = 0x8         |
| MSG_DONTROUTE    | = 0x4         |
| MSG_DONTWAIT     | = 0x40        |
| MSG_EOR          | = 0x80        |
| MSG_ERRQUEUE     | = 0x2000      |
| MSG_FASTOPEN     | = 0x20000000  |
| MSG_FIN          | = 0x200       |
| MSG_MORE         | = 0x8000      |
| MSG_NOSIGNAL     | = 0x4000      |
| MSG_OOB          | = 0x1         |
| MSG_PEEK         | = 0x2         |
| MSG_PROXY        | = 0x10        |
| MSG_RST          | = 0x1000      |
| MSG_SYN          | = 0x400       |
| MSG_TRUNC        | = 0x20        |
| MSG_TRYHARD      | = 0x4         |
| MSG_WAITALL      | = 0x100       |
| MSG_WAITFORONE   | = 0x10000     |
| MS_ACTIVE        | = 0x40000000  |
| MS_ASYNC         | = 0x1         |
| MS_BIND          | = 0x1000      |
| MS_DIRSYNC       | = 0x80        |
| MS_INVALIDATE    | = 0x2         |
| MS_I_VERSION     | = 0x800000    |
| MS_KERNMOUNT     | = 0x400000    |
| MS_MANDLOCK      | = 0x40        |
| MS_MGC_MSK       | = 0xfffff0000 |

|                         |               |
|-------------------------|---------------|
| MS_MGC_VAL              | = 0xc0ed0000  |
| MS_MOVE                 | = 0x2000      |
| MS_NOATIME              | = 0x400       |
| MS_NODEV                | = 0x4         |
| MS_NODIRATIME           | = 0x800       |
| MS_NOEXEC               | = 0x8         |
| MS_NOSUID               | = 0x2         |
| MS_NOUSER               | = -0x80000000 |
| MS_POSIXACL             | = 0x10000     |
| MS_PRIVATE              | = 0x40000     |
| MS_RDONLY               | = 0x1         |
| MS_REC                  | = 0x4000      |
| MS_RELATIME             | = 0x200000    |
| MS_REMOUNT              | = 0x20        |
| MS_RMT_MASK             | = 0x800051    |
| MS_SHARED               | = 0x100000    |
| MS_SILENT               | = 0x8000      |
| MS_SLAVE                | = 0x80000     |
| MS_STRICTATIME          | = 0x1000000   |
| MS_SYNC                 | = 0x4         |
| MS_SYNCHRONOUS          | = 0x10        |
| MS_UNBINDABLE           | = 0x20000     |
| NAME_MAX                | = 0xff        |
| NETLINK_ADD_MEMBERSHIP  | = 0x1         |
| NETLINK_AUDIT           | = 0x9         |
| NETLINK_BROADCAST_ERROR | = 0x4         |
| NETLINK_CONNECTOR       | = 0xb         |
| NETLINK_DNRMSG          | = 0xe         |
| NETLINK_DROP_MEMBERSHIP | = 0x2         |
| NETLINK_ECRYPTFS        | = 0x13        |
| NETLINK_FIB_LOOKUP      | = 0xa         |
| NETLINK_FIREWALL        | = 0x3         |
| NETLINK_GENERIC         | = 0x10        |
| NETLINK_INET_DIAG       | = 0x4         |
| NETLINK_IP6_FW          | = 0xd         |
| NETLINK_ISCSI           | = 0x8         |
| NETLINK_KOBJECT_UEVENT  | = 0xf         |
| NETLINK_NETFILTER       | = 0xc         |
| NETLINK_NFLOG           | = 0x5         |
| NETLINK_NO_ENOBUFS      | = 0x5         |
| NETLINK_PKTINFO         | = 0x3         |
| NETLINK_ROUTE           | = 0x0         |
| NETLINK_SCSITRANSPORT   | = 0x12        |
| NETLINK_SELINUX         | = 0x7         |
| NETLINK_UNUSED          | = 0x1         |
| NETLINK_USERSOCK        | = 0x2         |
| NETLINK_XFRM            | = 0x6         |
| NLA_ALIGNTO             | = 0x4         |
| NLA_F_NESTED            | = 0x8000      |
| NLA_F_NET_BYTEORDER     | = 0x4000      |
| NLA_HDRLEN              | = 0x4         |
| NLMSG_ALIGNTO           | = 0x4         |
| NLMSG_DONE              | = 0x3         |
| NLMSG_ERROR             | = 0x2         |

|                        |            |
|------------------------|------------|
| NLMSG_HDRLEN           | = 0x10     |
| NLMSG_MIN_TYPE         | = 0x10     |
| NLMSG_NOOP             | = 0x1      |
| NLMSG_OVERRUN          | = 0x4      |
| NLM_F_ACK              | = 0x4      |
| NLM_F_APPEND           | = 0x800    |
| NLM_F_ATOMIC           | = 0x400    |
| NLM_F_CREATE           | = 0x400    |
| NLM_F_DUMP             | = 0x300    |
| NLM_F_ECHO             | = 0x8      |
| NLM_F_EXCL             | = 0x200    |
| NLM_F_MATCH            | = 0x200    |
| NLM_F_MULTI            | = 0x2      |
| NLM_F_REPLACE          | = 0x100    |
| NLM_F_REQUEST          | = 0x1      |
| NLM_F_ROOT             | = 0x100    |
| O_ACCMODE              | = 0x3      |
| O_APPEND               | = 0x400    |
| O_ASYNC                | = 0x2000   |
| O_CLOEXEC              | = 0x80000  |
| O_CREAT                | = 0x40     |
| O_DIRECT               | = 0x4000   |
| O_DIRECTORY            | = 0x10000  |
| O_DSYNC                | = 0x1000   |
| O_EXCL                 | = 0x80     |
| O_FSYNC                | = 0x101000 |
| O_LARGEFILE            | = 0x0      |
| O_NDELAY               | = 0x800    |
| O_NOATIME              | = 0x40000  |
| O_NOCTTY               | = 0x100    |
| O_NOFOLLOW             | = 0x20000  |
| O_NONBLOCK             | = 0x800    |
| O_RDONLY               | = 0x0      |
| O_RDWR                 | = 0x2      |
| O_RSYNC                | = 0x101000 |
| O_SYNC                 | = 0x101000 |
| O_TRUNC                | = 0x200    |
| O_WRONLY               | = 0x1      |
| PACKET_ADD_MEMBERSHIP  | = 0x1      |
| PACKET_BROADCAST       | = 0x1      |
| PACKET_DROP_MEMBERSHIP | = 0x2      |
| PACKET_FASTROUTE       | = 0x6      |
| PACKET_HOST            | = 0x0      |
| PACKET_LOOPBACK        | = 0x5      |
| PACKET_MR_ALLMULTI     | = 0x2      |
| PACKET_MR_MULTICAST    | = 0x0      |
| PACKET_MR_PROMISC      | = 0x1      |
| PACKET_MULTICAST       | = 0x2      |
| PACKET_OTHERHOST       | = 0x3      |
| PACKET_OUTGOING        | = 0x4      |
| PACKET_RECV_OUTPUT     | = 0x3      |
| PACKET_RX_RING         | = 0x5      |
| PACKET_STATISTICS      | = 0x6      |
| PRI_O_PGRP             | = 0x1      |

|                      |              |
|----------------------|--------------|
| PRI0_PROCESS         | = 0x0        |
| PRI0_USER            | = 0x2        |
| PROT_EXEC            | = 0x4        |
| PROT_GROWSDOWN       | = 0x1000000  |
| PROT_GROWSUP         | = 0x2000000  |
| PROT_NONE            | = 0x0        |
| PROT_READ            | = 0x1        |
| PROT_WRITE           | = 0x2        |
| PR_CAPBSET_DROP      | = 0x18       |
| PR_CAPBSET_READ      | = 0x17       |
| PR_ENDIAN_BIG        | = 0x0        |
| PR_ENDIAN_LITTLE     | = 0x1        |
| PR_ENDIAN_PPC_LITTLE | = 0x2        |
| PR_FPEMU_NOPRINT     | = 0x1        |
| PR_FPEMU_SIGFPE      | = 0x2        |
| PR_FP_EXC_ASYNC      | = 0x2        |
| PR_FP_EXC_DISABLED   | = 0x0        |
| PR_FP_EXC_DIV        | = 0x10000    |
| PR_FP_EXC_INV        | = 0x100000   |
| PR_FP_EXC_NONRECOV   | = 0x1        |
| PR_FP_EXC_OVF        | = 0x20000    |
| PR_FP_EXC_PRECISE    | = 0x3        |
| PR_FP_EXC_RES        | = 0x80000    |
| PR_FP_EXC_SW_ENABLE  | = 0x80       |
| PR_FP_EXC_UND        | = 0x40000    |
| PR_GET_DUMPABLE      | = 0x3        |
| PR_GET_ENDIAN        | = 0x13       |
| PR_GET_FPEMU         | = 0x9        |
| PR_GET_FPEXC         | = 0xb        |
| PR_GET_KEEPcaps      | = 0x7        |
| PR_GET_NAME          | = 0x10       |
| PR_GET_PDEATHSIG     | = 0x2        |
| PR_GET_SECCOMP       | = 0x15       |
| PR_GET_SECUREBITS    | = 0x1b       |
| PR_GET_TIMERSLACK    | = 0x1e       |
| PR_GET_TIMING        | = 0xd        |
| PR_GET_TSC           | = 0x19       |
| PR_GET_UNALIGN       | = 0x5        |
| PR_MCE_KILL          | = 0x21       |
| PR_MCE_KILL_CLEAR    | = 0x0        |
| PR_MCE_KILL_DEFAULT  | = 0x2        |
| PR_MCE_KILL_EARLY    | = 0x1        |
| PR_MCE_KILL_GET      | = 0x22       |
| PR_MCE_KILL_LATE     | = 0x0        |
| PR_MCE_KILL_SET      | = 0x1        |
| PR_SET_DUMPABLE      | = 0x4        |
| PR_SET_ENDIAN        | = 0x14       |
| PR_SET_FPEMU         | = 0xa        |
| PR_SET_FPEXC         | = 0xc        |
| PR_SET_KEEPcaps      | = 0x8        |
| PR_SET_NAME          | = 0xf        |
| PR_SET_PDEATHSIG     | = 0x1        |
| PR_SET_PTRACER       | = 0x59616d61 |
| PR_SET_SECCOMP       | = 0x16       |

|                             |          |
|-----------------------------|----------|
| PR_SET_SECUREBITS           | = 0x1c   |
| PR_SET_TIMERSLACK           | = 0x1d   |
| PR_SET_TIMING               | = 0xe    |
| PR_SET_TSC                  | = 0x1a   |
| PR_SET_UNALIGN              | = 0x6    |
| PR_TASK_PERF_EVENTS_DISABLE | = 0x1f   |
| PR_TASK_PERF_EVENTS_ENABLE  | = 0x20   |
| PR_TIMING_STATISTICAL       | = 0x0    |
| PR_TIMING_TIMESTAMP         | = 0x1    |
| PR_TSC_ENABLE               | = 0x1    |
| PR_TSC_SIGSEGV              | = 0x2    |
| PR_UNALIGN_NOPRINT          | = 0x1    |
| PR_UNALIGN_SIGBUS           | = 0x2    |
| PTRACE_ARCH_PRCTL           | = 0x1e   |
| PTRACE_ATTACH               | = 0x10   |
| PTRACE_CONT                 | = 0x7    |
| PTRACE_DETACH               | = 0x11   |
| PTRACE_EVENT_CLONE          | = 0x3    |
| PTRACE_EVENT_EXEC           | = 0x4    |
| PTRACE_EVENT_EXIT           | = 0x6    |
| PTRACE_EVENT_FORK           | = 0x1    |
| PTRACE_EVENT_VFORK          | = 0x2    |
| PTRACE_EVENT_VFORK_DONE     | = 0x5    |
| PTRACE_GETEVENTMSG          | = 0x4201 |
| PTRACE_GETFPREGS            | = 0xe    |
| PTRACE_GETFPXREGS           | = 0x12   |
| PTRACE_GETREGS              | = 0xc    |
| PTRACE_GETREGSET            | = 0x4204 |
| PTRACE_GETSIGINFO           | = 0x4202 |
| PTRACE_GET_THREAD_AREA      | = 0x19   |
| PTRACE_KILL                 | = 0x8    |
| PTRACE_OLDSETOPTIONS        | = 0x15   |
| PTRACE_O_MASK               | = 0x7f   |
| PTRACE_O_TRACECLONE         | = 0x8    |
| PTRACE_O_TRACEEXEC          | = 0x10   |
| PTRACE_O_TRACEEXIT          | = 0x40   |
| PTRACE_O_TRACEFORK          | = 0x2    |
| PTRACE_O_TRACESYSGOOD       | = 0x1    |
| PTRACE_O_TRACEVFORK         | = 0x4    |
| PTRACE_O_TRACEVFORKDONE     | = 0x20   |
| PTRACE_PEEKDATA             | = 0x2    |
| PTRACE_PEEKTEXT             | = 0x1    |
| PTRACE_PEEKUSR              | = 0x3    |
| PTRACE_POKEDATA             | = 0x5    |
| PTRACE_POKETEXT             | = 0x4    |
| PTRACE_POKEUSR              | = 0x6    |
| PTRACE_SETFPREGS            | = 0xf    |
| PTRACE_SETFPXREGS           | = 0x13   |
| PTRACE_SETOPTIONS           | = 0x4200 |
| PTRACE_SETREGS              | = 0xd    |
| PTRACE_SETREGSET            | = 0x4205 |
| PTRACE_SETSIGINFO           | = 0x4203 |
| PTRACE_SET_THREAD_AREA      | = 0x1a   |
| PTRACE_SINGLEBLOCK          | = 0x21   |

|                          |               |
|--------------------------|---------------|
| PTRACE_SINGLESTEP        | = 0x9         |
| PTRACE_SYSCALL           | = 0x18        |
| PTRACE_SYSEMU            | = 0x1f        |
| PTRACE_SYSEMU_SINGLESTEP | = 0x20        |
| PTRACE_TRACEME           | = 0x0         |
| RLIMIT_AS                | = 0x9         |
| RLIMIT_CORE              | = 0x4         |
| RLIMIT_CPU               | = 0x0         |
| RLIMIT_DATA              | = 0x2         |
| RLIMIT_FSIZE             | = 0x1         |
| RLIMIT_NOFILE            | = 0x7         |
| RLIMIT_STACK             | = 0x3         |
| RLIM_INFINITY            | = -0x1        |
| RTAX_ADV MSS             | = 0x8         |
| RTAX_CWND                | = 0x7         |
| RTAX_FEATURES            | = 0xc         |
| RTAX_FEATURE_ALLFRAG     | = 0x8         |
| RTAX_FEATURE_ECN         | = 0x1         |
| RTAX_FEATURE_SACK        | = 0x2         |
| RTAX_FEATURE_TIMESTAMP   | = 0x4         |
| RTAX_HOPLIMIT            | = 0xa         |
| RTAX_INITCWND            | = 0xb         |
| RTAX_INITRWND            | = 0xe         |
| RTAX_LOCK                | = 0x1         |
| RTAX_MAX                 | = 0xe         |
| RTAX_MTU                 | = 0x2         |
| RTAX_REORDERING          | = 0x9         |
| RTAX_RTO_MIN             | = 0xd         |
| RTAX_RTT                 | = 0x4         |
| RTAX_RTTVAR              | = 0x5         |
| RTAX_SSTHRESH            | = 0x6         |
| RTAX_UNSPEC              | = 0x0         |
| RTAX_WINDOW              | = 0x3         |
| RTA_ALIGN TO             | = 0x4         |
| RTA_MAX                  | = 0x10        |
| RTCF_DIRECTSRC           | = 0x40000000  |
| RTCF_DOREDIRECT          | = 0x10000000  |
| RTCF_LOG                 | = 0x20000000  |
| RTCF_MASQ                | = 0x400000    |
| RTCF_NAT                 | = 0x800000    |
| RTCF_VALVE               | = 0x200000    |
| RTF_ADDRCMASK            | = 0xf80000000 |
| RTF_ADDRCNF              | = 0x40000     |
| RTF_ALLONLINK            | = 0x20000     |
| RTF_BROADCAST            | = 0x100000000 |
| RTF_CACHE                | = 0x1000000   |
| RTF_DEFAULT              | = 0x10000     |
| RTF_DYNAMIC              | = 0x10        |
| RTF_FLOW                 | = 0x20000000  |
| RTF_GATEWAY              | = 0x2         |
| RTF_HOST                 | = 0x4         |
| RTF_INTERFACE            | = 0x400000000 |
| RTF_IRTT                 | = 0x100       |
| RTF_LINKRT               | = 0x100000    |

|                  |              |
|------------------|--------------|
| RTF_LOCAL        | = 0x80000000 |
| RTF_MODIFIED     | = 0x20       |
| RTF_MSS          | = 0x40       |
| RTF_MTU          | = 0x40       |
| RTF_MULTICAST    | = 0x20000000 |
| RTF_NAT          | = 0x8000000  |
| RTF_NOFORWARD    | = 0x1000     |
| RTF_NONEXTHOP    | = 0x200000   |
| RTF_NOPMTUDISC   | = 0x4000     |
| RTF_POLICY       | = 0x4000000  |
| RTF_REINSTATE    | = 0x8        |
| RTF_REJECT       | = 0x200      |
| RTF_STATIC       | = 0x400      |
| RTF_THROW        | = 0x2000     |
| RTF_UP           | = 0x1        |
| RTF_WINDOW       | = 0x80       |
| RTF_XRESOLVE     | = 0x800      |
| RTM_BASE         | = 0x10       |
| RTM_DELACTION    | = 0x31       |
| RTM_DELADDR      | = 0x15       |
| RTM_DELADDRLABEL | = 0x49       |
| RTM_DELLINK      | = 0x11       |
| RTM_DELNEIGH     | = 0x1d       |
| RTM_DELQDISC     | = 0x25       |
| RTM_DELROUTE     | = 0x19       |
| RTM_DELRULE      | = 0x21       |
| RTM_DELTCLASS    | = 0x29       |
| RTM_DELTFILTER   | = 0x2d       |
| RTM_F_CLONED     | = 0x200      |
| RTM_F_EQUALIZE   | = 0x400      |
| RTM_F_NOTIFY     | = 0x100      |
| RTM_F_PREFIX     | = 0x800      |
| RTM_GETACTION    | = 0x32       |
| RTM_GETADDR      | = 0x16       |
| RTM_GETADDRLABEL | = 0x4a       |
| RTM_GETANYCAST   | = 0x3e       |
| RTM_GETDCB       | = 0x4e       |
| RTM_GETLINK      | = 0x12       |
| RTM_GETMULTICAST | = 0x3a       |
| RTM_GETNEIGH     | = 0x1e       |
| RTM_GETNEIGHTBL  | = 0x42       |
| RTM_GETQDISC     | = 0x26       |
| RTM_GETROUTE     | = 0x1a       |
| RTM_GETRULE      | = 0x22       |
| RTM_GETTCLASS    | = 0x2a       |
| RTM_GETTFILTER   | = 0x2e       |
| RTM_MAX          | = 0x4f       |
| RTM_NEWACTION    | = 0x30       |
| RTM_NEWADDR      | = 0x14       |
| RTM_NEWADDRLABEL | = 0x48       |
| RTM_NEWLINK      | = 0x10       |
| RTM_NEWNDUSEROPT | = 0x44       |
| RTM_NEWNEIGH     | = 0x1c       |
| RTM_NEWNEIGHTBL  | = 0x40       |

|                  |          |
|------------------|----------|
| RTM_NEWPREFIX    | = 0x34   |
| RTM_NEWQDISC     | = 0x24   |
| RTM_NEWRUTE      | = 0x18   |
| RTM_NEWRULE      | = 0x20   |
| RTM_NEWTCLASS    | = 0x28   |
| RTM_NEWTFILTER   | = 0x2c   |
| RTM_NR_FAMILIES  | = 0x10   |
| RTM_NR_MSGTYPES  | = 0x40   |
| RTM_SETDCB       | = 0x4f   |
| RTM_SETLINK      | = 0x13   |
| RTM_SETNEIGHTBL  | = 0x43   |
| RTNH_ALIGNTO     | = 0x4    |
| RTNH_F_DEAD      | = 0x1    |
| RTNH_F_ONLINK    | = 0x4    |
| RTNH_F_PERVASIVE | = 0x2    |
| RTN_MAX          | = 0xb    |
| RTPROT_BIRD      | = 0xc    |
| RTPROT_BOOT      | = 0x3    |
| RTPROT_DHCP      | = 0x10   |
| RTPROT_DNRouted  | = 0xd    |
| RTPROT_GATED     | = 0x8    |
| RTPROT_KERNEL    | = 0x2    |
| RTPROT_MRT       | = 0xa    |
| RTPROT_NTK       | = 0xf    |
| RTPROT_RA        | = 0x9    |
| RTPROT_REDIRECT  | = 0x1    |
| RTPROT_STATIC    | = 0x4    |
| RTPROT_UNSPEC    | = 0x0    |
| RTPROT_XORP      | = 0xe    |
| RTPROT_ZEBRA     | = 0xb    |
| RT_CLASS_DEFAULT | = 0xfd   |
| RT_CLASS_LOCAL   | = 0xff   |
| RT_CLASS_MAIN    | = 0xfe   |
| RT_CLASS_MAX     | = 0xff   |
| RT_CLASS_UNSPEC  | = 0x0    |
| RUSAGE_CHILDREN  | = -0x1   |
| RUSAGE_SELF      | = 0x0    |
| RUSAGE_THREAD    | = 0x1    |
| SCM_CREDENTIALS  | = 0x2    |
| SCM_RIGHTS       | = 0x1    |
| SCM_TIMESTAMP    | = 0x1d   |
| SCM_TIMESTAMPING | = 0x25   |
| SCM_TIMESTAMPNS  | = 0x23   |
| SHUT_RD          | = 0x0    |
| SHUT_RDWR        | = 0x2    |
| SHUT_WR          | = 0x1    |
| SIOCADDLCI       | = 0x8980 |
| SIOCADDMULTI     | = 0x8931 |
| SIOCADDRT        | = 0x890b |
| SIOCATMARK       | = 0x8905 |
| SIOCDARP         | = 0x8953 |
| SIOCDELDLCI      | = 0x8981 |
| SIOCDELMULTI     | = 0x8932 |
| SIOCDELRT        | = 0x890c |

|                    |           |
|--------------------|-----------|
| SIOCDEVPRIVATE     | = 0x89f0  |
| SIOCDIFADDR        | = 0x8936  |
| SIOCDRARP          | = 0x8960  |
| SIOCGARP           | = 0x8954  |
| SIOCGIFADDR        | = 0x8915  |
| SIOCGIFBR          | = 0x8940  |
| SIOCGIFBRDADDR     | = 0x8919  |
| SIOCGIFCONF        | = 0x8912  |
| SIOCGIFCOUNT       | = 0x8938  |
| SIOCGIFDSTADDR     | = 0x8917  |
| SIOCGIFENCAP       | = 0x8925  |
| SIOCGIFFLAGS       | = 0x8913  |
| SIOCGIFHWADDR      | = 0x8927  |
| SIOCGIFINDEX       | = 0x8933  |
| SIOCGIFMAP         | = 0x8970  |
| SIOCGIFMEM         | = 0x891f  |
| SIOCGIFMETRIC      | = 0x891d  |
| SIOCGIFMTU         | = 0x8921  |
| SIOCGIFNAME        | = 0x8910  |
| SIOCGIFNETMASK     | = 0x891b  |
| SIOCGIFPFLAGS      | = 0x8935  |
| SIOCGIFSLAVE       | = 0x8929  |
| SIOCGIFTXQLEN      | = 0x8942  |
| SIOCGRPGRP         | = 0x8904  |
| SIOCGRARP          | = 0x8961  |
| SIOCGSTAMP         | = 0x8906  |
| SIOCGSTAMPNS       | = 0x8907  |
| SIOCPRIVATE        | = 0x89e0  |
| SIOCRTMSG          | = 0x890d  |
| SIOCSARP           | = 0x8955  |
| SIOCSIFADDR        | = 0x8916  |
| SIOCSIFBR          | = 0x8941  |
| SIOCSIFBRDADDR     | = 0x891a  |
| SIOCSIFDSTADDR     | = 0x8918  |
| SIOCSIFENCAP       | = 0x8926  |
| SIOCSIFFLAGS       | = 0x8914  |
| SIOCSIFHWADDR      | = 0x8924  |
| SIOCSIFHWBROADCAST | = 0x8937  |
| SIOCSIFLINK        | = 0x8911  |
| SIOCSIFMAP         | = 0x8971  |
| SIOCSIFMEM         | = 0x8920  |
| SIOCSIFMETRIC      | = 0x891e  |
| SIOCSIFMTU         | = 0x8922  |
| SIOCSIFNAME        | = 0x8923  |
| SIOCSIFNETMASK     | = 0x891c  |
| SIOCSIFPFLAGS      | = 0x8934  |
| SIOCSIFSLAVE       | = 0x8930  |
| SIOCSIFTXQLEN      | = 0x8943  |
| SIOCSPGRP          | = 0x8902  |
| SIOCSRARP          | = 0x8962  |
| SOCK_CLOEXEC       | = 0x80000 |
| SOCK_DCCP          | = 0x6     |
| SOCK_DGRAM         | = 0x2     |
| SOCK_NONBLOCK      | = 0x800   |

|                                  |         |
|----------------------------------|---------|
| SOCK_PACKET                      | = 0xa   |
| SOCK_RAW                         | = 0x3   |
| SOCK_RDM                         | = 0x4   |
| SOCK_SEQPACKET                   | = 0x5   |
| SOCK_STREAM                      | = 0x1   |
| SOL_AAL                          | = 0x109 |
| SOL_ATM                          | = 0x108 |
| SOL_DECNET                       | = 0x105 |
| SOL_ICMPV6                       | = 0x3a  |
| SOL_IP                           | = 0x0   |
| SOL_IPV6                         | = 0x29  |
| SOL_IRDA                         | = 0x10a |
| SOL_PACKET                       | = 0x107 |
| SOL_RAW                          | = 0xff  |
| SOL_SOCKET                       | = 0x1   |
| SOL_TCP                          | = 0x6   |
| SOL_X25                          | = 0x106 |
| SOMAXCONN                        | = 0x80  |
| SO_ACCEPTCONN                    | = 0x1e  |
| SO_ATTACH_FILTER                 | = 0x1a  |
| SO_BINDTODEVICE                  | = 0x19  |
| SO_BROADCAST                     | = 0x6   |
| SO_BSDCOMPAT                     | = 0xe   |
| SO_DEBUG                         | = 0x1   |
| SO_DETACH_FILTER                 | = 0x1b  |
| SO_DOMAIN                        | = 0x27  |
| SO_DONTROUTE                     | = 0x5   |
| SO_ERROR                         | = 0x4   |
| SO_KEEPALIVE                     | = 0x9   |
| SO_LINGER                        | = 0xd   |
| SO_MARK                          | = 0x24  |
| SO_NO_CHECK                      | = 0xb   |
| SO_OOBINLINE                     | = 0xa   |
| SO_PASSCRED                      | = 0x10  |
| SO_PASSEC                        | = 0x22  |
| SO_PEERCRED                      | = 0x11  |
| SO_PEERNAME                      | = 0x1c  |
| SO_PEERSEC                       | = 0x1f  |
| SO_PRIORITY                      | = 0xc   |
| SO_PROTOCOL                      | = 0x26  |
| SO_RCVBUF                        | = 0x8   |
| SO_RCVBUFFORCE                   | = 0x21  |
| SO_RCVLOWAT                      | = 0x12  |
| SO_RCVTIMEO                      | = 0x14  |
| SO_REUSEADDR                     | = 0x2   |
| SO_RXQ_OVFL                      | = 0x28  |
| SO_SECURITY_AUTHENTICATION       | = 0x16  |
| SO_SECURITY_ENCRYPTION_NETWORK   | = 0x18  |
| SO_SECURITY_ENCRYPTION_TRANSPORT | = 0x17  |
| SO_SNDBUF                        | = 0x7   |
| SO_SNDBUFFORCE                   | = 0x20  |
| SO SNDLOWAT                      | = 0x13  |
| SO SNDTIMEO                      | = 0x15  |
| SO_TIMESTAMP                     | = 0x1d  |

|                      |          |
|----------------------|----------|
| SO_TIMESTAMPING      | = 0x25   |
| SO_TIMESTAMPNS       | = 0x23   |
| SO_TYPE              | = 0x3    |
| S_BLKSIZE            | = 0x200  |
| S_IEXEC              | = 0x40   |
| S_IFBLK              | = 0x6000 |
| S_IFCHR              | = 0x2000 |
| S_IFDIR              | = 0x4000 |
| S_IFIFO              | = 0x1000 |
| S_IFLNK              | = 0xa000 |
| S_IFMT               | = 0xf000 |
| S_IFREG              | = 0x8000 |
| S_IFSOCK             | = 0xc000 |
| S_IREAD              | = 0x100  |
| S_IRGRP              | = 0x20   |
| S_IROTH              | = 0x4    |
| S_IRUSR              | = 0x100  |
| S_IRWXG              | = 0x38   |
| S_IRWXO              | = 0x7    |
| S_IRWXU              | = 0x1c0  |
| S_ISGID              | = 0x400  |
| S_ISUID              | = 0x800  |
| S_ISVTX              | = 0x200  |
| S_IWGRP              | = 0x10   |
| S_IWOTH              | = 0x2    |
| S_IWRITE             | = 0x80   |
| S_IWUSR              | = 0x80   |
| S_IXGRP              | = 0x8    |
| S_IXOTH              | = 0x1    |
| S_IXUSR              | = 0x40   |
| TCIFLUSH             | = 0x0    |
| TCIOFLUSH            | = 0x2    |
| TCOFLUSH             | = 0x1    |
| TCP_CONGESTION       | = 0xd    |
| TCP_CORK             | = 0x3    |
| TCP_DEFER_ACCEPT     | = 0x9    |
| TCP_INFO             | = 0xb    |
| TCP_KEEPCNT          | = 0x6    |
| TCP_KEEPIDLE         | = 0x4    |
| TCP_KEEPINTVL        | = 0x5    |
| TCP_LINGER2          | = 0x8    |
| TCP_MAXSEG           | = 0x2    |
| TCP_MAXWIN           | = 0xffff |
| TCP_MAX_WINSHIFT     | = 0xe    |
| TCP_MD5SIG           | = 0xe    |
| TCP_MD5SIG_MAXKEYLEN | = 0x50   |
| TCP_MSS              | = 0x200  |
| TCP_NODELAY          | = 0x1    |
| TCP_QUICKACK         | = 0xc    |
| TCP_SYNCNT           | = 0x7    |
| TCP_WINDOW_CLAMP     | = 0xa    |
| TIOCCBRK             | = 0x5428 |
| TIOCCONS             | = 0x541d |
| TIOCEXCL             | = 0x540c |

|                    |              |
|--------------------|--------------|
| TIOCGDEV           | = 0x80045432 |
| TIOCGETD           | = 0x5424     |
| TIOCGICOUNT        | = 0x545d     |
| TIOCGLCKTRMIOS     | = 0x5456     |
| TIOCGRPGRP         | = 0x540f     |
| TIOCPTN            | = 0x80045430 |
| TIOCGRS485         | = 0x542e     |
| TIOCGSERIAL        | = 0x541e     |
| TIOCGSID           | = 0x5429     |
| TIOCGRSOFTCAR      | = 0x5419     |
| TIOCGWINSZ         | = 0x5413     |
| TIOCINQ            | = 0x541b     |
| TIOCLINUX          | = 0x541c     |
| TIOCMBIC           | = 0x5417     |
| TIOCMBIS           | = 0x5416     |
| TIOCGET            | = 0x5415     |
| TIOCMIWAIT         | = 0x545c     |
| TIOCSET            | = 0x5418     |
| TIOC_CAR           | = 0x40       |
| TIOC_CD            | = 0x40       |
| TIOCCTS            | = 0x20       |
| TIOCDSR            | = 0x100      |
| TIOC_DTR           | = 0x2        |
| TIOC_LE            | = 0x1        |
| TIOC_RI            | = 0x80       |
| TIOC_RNG           | = 0x80       |
| TIOC_RTS           | = 0x4        |
| TIOC_SR            | = 0x10       |
| TIOC_ST            | = 0x8        |
| TIOCNOTTY          | = 0x5422     |
| TIOCNXCL           | = 0x540d     |
| TIOCOUTQ           | = 0x5411     |
| TIOCPTK            | = 0x5420     |
| TIOCPTK_DATA       | = 0x0        |
| TIOCPTK_DOSTOP     | = 0x20       |
| TIOCPTK_FLUSHREAD  | = 0x1        |
| TIOCPTK_FLUSHWRITE | = 0x2        |
| TIOCPTK_IOCTL      | = 0x40       |
| TIOCPTK_NOSTOP     | = 0x10       |
| TIOCPTK_START      | = 0x8        |
| TIOCPTK_STOP       | = 0x4        |
| TIOCSBRK           | = 0x5427     |
| TIOCSCTTY          | = 0x540e     |
| TIOCSERCOMFIG      | = 0x5453     |
| TIOCSEGETLSR       | = 0x5459     |
| TIOCSEGETMULTI     | = 0x545a     |
| TIOCSERGSTRUCT     | = 0x5458     |
| TIOCSERGWILD       | = 0x5454     |
| TIOCSESETMULTI     | = 0x545b     |
| TIOCSERSWILD       | = 0x5455     |
| TIOCSETEMPT        | = 0x1        |
| TIOCSETD           | = 0x5423     |
| TIOCSIG            | = 0x40045436 |
| TIOCSLCKTRMIOS     | = 0x5457     |

```

TIOCSPGRP = 0x5410
TIOCSPTLCK = 0x40045431
TIOCSRS485 = 0x542f
TIOCSSERIAL = 0x541f
TIOCSSOFTCAR = 0x541a
TIOCSTI = 0x5412
TIOCSWINSZ = 0x5414
TUNATTACHFILTER = 0x401054d5
TUNDETACHFILTER = 0x401054d6
TUNGETFEATURES = 0x800454cf
TUNGETIFF = 0x800454d2
TUNGETSNDBUF = 0x800454d3
TUNGETVNETHDRSZ = 0x800454d7
TUNSETDEBUG = 0x400454c9
TUNSETGROUP = 0x400454ce
TUNSETIFF = 0x400454ca
TUNSETLINK = 0x400454cd
TUNSETNOCSUM = 0x400454c8
TUNSETOFFLOAD = 0x400454d0
TUNSETOwner = 0x400454cc
TUNSETPERSIST = 0x400454cb
TUNSETSNDBUF = 0x400454d4
TUNSETTXFILTER = 0x400454d1
TUNSETVNETHDRSZ = 0x400454d8
WALL = 0x40000000
WCLONE = 0x80000000
WCONTINUED = 0x8
WEXITED = 0x4
WNOHANG = 0x1
WNOTHREAD = 0x20000000
WNOWAIT = 0x10000000
WORDSIZE = 0x40
WSTOPPED = 0x2
WUNTRACED = 0x2
)

```

```

const (
 E2BIG = Errno(0x7)
 EACCES = Errno(0xd)
 EADDRINUSE = Errno(0x62)
 EADDRNOTAVAIL = Errno(0x63)
 EADV = Errno(0x44)
 EAFNOSUPPORT = Errno(0x61)
 EAGAIN = Errno(0xb)
 EALREADY = Errno(0x72)
 EBADE = Errno(0x34)
 EBADF = Errno(0x9)
 EBADFD = Errno(0x4d)
 EBADMSG = Errno(0x4a)
 EBADR = Errno(0x35)
 EBADRQC = Errno(0x38)
 EBADSLT = Errno(0x39)
 EBFONT = Errno(0x3b)
)

```

|              |                                |
|--------------|--------------------------------|
| EBUSY        | = <a href="#">Errno</a> (0x10) |
| ECANCELED    | = <a href="#">Errno</a> (0x7d) |
| ECHILD       | = <a href="#">Errno</a> (0xa)  |
| ECHRNG       | = <a href="#">Errno</a> (0x2c) |
| ECOMM        | = <a href="#">Errno</a> (0x46) |
| ECONNABORTED | = <a href="#">Errno</a> (0x67) |
| ECONNREFUSED | = <a href="#">Errno</a> (0x6f) |
| ECONNRESET   | = <a href="#">Errno</a> (0x68) |
| EDEADLK      | = <a href="#">Errno</a> (0x23) |
| EDEADLOCK    | = <a href="#">Errno</a> (0x23) |
| EDESTADDRREQ | = <a href="#">Errno</a> (0x59) |
| EDOM         | = <a href="#">Errno</a> (0x21) |
| EDOTDOT      | = <a href="#">Errno</a> (0x49) |
| EDQUOT       | = <a href="#">Errno</a> (0x7a) |
| EEXIST       | = <a href="#">Errno</a> (0x11) |
| EFAULT       | = <a href="#">Errno</a> (0xe)  |
| EFBIG        | = <a href="#">Errno</a> (0x1b) |
| EHOSTDOWN    | = <a href="#">Errno</a> (0x70) |
| EHOSTUNREACH | = <a href="#">Errno</a> (0x71) |
| EIDRM        | = <a href="#">Errno</a> (0x2b) |
| EILSEQ       | = <a href="#">Errno</a> (0x54) |
| EINPROGRESS  | = <a href="#">Errno</a> (0x73) |
| EINTR        | = <a href="#">Errno</a> (0x4)  |
| EINVAL       | = <a href="#">Errno</a> (0x16) |
| EIO          | = <a href="#">Errno</a> (0x5)  |
| EISCONN      | = <a href="#">Errno</a> (0x6a) |
| EISDIR       | = <a href="#">Errno</a> (0x15) |
| EISNAM       | = <a href="#">Errno</a> (0x78) |
| EKEYEXPIRED  | = <a href="#">Errno</a> (0x7f) |
| EKEYREJECTED | = <a href="#">Errno</a> (0x81) |
| EKEYREVOKED  | = <a href="#">Errno</a> (0x80) |
| EL2HLT       | = <a href="#">Errno</a> (0x33) |
| EL2NSYNC     | = <a href="#">Errno</a> (0x2d) |
| EL3HLT       | = <a href="#">Errno</a> (0x2e) |
| EL3RST       | = <a href="#">Errno</a> (0x2f) |
| ELIBACC      | = <a href="#">Errno</a> (0x4f) |
| ELIBBAD      | = <a href="#">Errno</a> (0x50) |
| ELIBEXEC     | = <a href="#">Errno</a> (0x53) |
| ELIBMAX      | = <a href="#">Errno</a> (0x52) |
| ELIBSCN      | = <a href="#">Errno</a> (0x51) |
| ELNRNG       | = <a href="#">Errno</a> (0x30) |
| ELOOP        | = <a href="#">Errno</a> (0x28) |
| EMEDIUMTYPE  | = <a href="#">Errno</a> (0x7c) |
| EMFILE       | = <a href="#">Errno</a> (0x18) |
| EMLINK       | = <a href="#">Errno</a> (0x1f) |
| EMSGSIZE     | = <a href="#">Errno</a> (0x5a) |
| EMULTIHOP    | = <a href="#">Errno</a> (0x48) |
| ENAMETOOLONG | = <a href="#">Errno</a> (0x24) |
| ENAVAIL      | = <a href="#">Errno</a> (0x77) |
| ENETDOWN     | = <a href="#">Errno</a> (0x64) |
| ENETRESET    | = <a href="#">Errno</a> (0x66) |
| ENETUNREACH  | = <a href="#">Errno</a> (0x65) |
| ENFILE       | = <a href="#">Errno</a> (0x17) |
| ENOANO       | = <a href="#">Errno</a> (0x37) |

|                 |                                |
|-----------------|--------------------------------|
| ENOBUFS         | = <a href="#">Errno</a> (0x69) |
| ENOCXI          | = <a href="#">Errno</a> (0x32) |
| ENODATA         | = <a href="#">Errno</a> (0x3d) |
| ENODEV          | = <a href="#">Errno</a> (0x13) |
| ENOENT          | = <a href="#">Errno</a> (0x2)  |
| ENOEXEC         | = <a href="#">Errno</a> (0x8)  |
| ENOKEY          | = <a href="#">Errno</a> (0x7e) |
| ENOLCK          | = <a href="#">Errno</a> (0x25) |
| ENOLINK         | = <a href="#">Errno</a> (0x43) |
| ENOMEDIUM       | = <a href="#">Errno</a> (0x7b) |
| ENOMEM          | = <a href="#">Errno</a> (0xc)  |
| ENOMSG          | = <a href="#">Errno</a> (0x2a) |
| ENONET          | = <a href="#">Errno</a> (0x40) |
| ENOPKG          | = <a href="#">Errno</a> (0x41) |
| ENOPROTOOPT     | = <a href="#">Errno</a> (0x5c) |
| ENOSPC          | = <a href="#">Errno</a> (0x1c) |
| ENOSR           | = <a href="#">Errno</a> (0x3f) |
| ENOSTR          | = <a href="#">Errno</a> (0x3c) |
| ENOSYS          | = <a href="#">Errno</a> (0x26) |
| ENOTBLK         | = <a href="#">Errno</a> (0xf)  |
| ENOTCONN        | = <a href="#">Errno</a> (0x6b) |
| ENOTDIR         | = <a href="#">Errno</a> (0x14) |
| ENOTEMPTY       | = <a href="#">Errno</a> (0x27) |
| ENOTNAM         | = <a href="#">Errno</a> (0x76) |
| ENOTRECOVERABLE | = <a href="#">Errno</a> (0x83) |
| ENOTSOCK        | = <a href="#">Errno</a> (0x58) |
| ENOTSUP         | = <a href="#">Errno</a> (0x5f) |
| ENOTTY          | = <a href="#">Errno</a> (0x19) |
| ENOTUNIQ        | = <a href="#">Errno</a> (0x4c) |
| ENXIO           | = <a href="#">Errno</a> (0x6)  |
| EOPNOTSUPP      | = <a href="#">Errno</a> (0x5f) |
| EOVERFLOW       | = <a href="#">Errno</a> (0x4b) |
| EOWNERDEAD      | = <a href="#">Errno</a> (0x82) |
| EPERM           | = <a href="#">Errno</a> (0x1)  |
| EPFNOSUPPORT    | = <a href="#">Errno</a> (0x60) |
| EPIPE           | = <a href="#">Errno</a> (0x20) |
| EPROTO          | = <a href="#">Errno</a> (0x47) |
| EPROTONOSUPPORT | = <a href="#">Errno</a> (0x5d) |
| EPROTOTYPE      | = <a href="#">Errno</a> (0x5b) |
| ERANGE          | = <a href="#">Errno</a> (0x22) |
| EREMCHG         | = <a href="#">Errno</a> (0x4e) |
| EREMOTE         | = <a href="#">Errno</a> (0x42) |
| EREMOTEIO       | = <a href="#">Errno</a> (0x79) |
| ERESTART        | = <a href="#">Errno</a> (0x55) |
| ERFKILL         | = <a href="#">Errno</a> (0x84) |
| EROFS           | = <a href="#">Errno</a> (0x1e) |
| ESHUTDOWN       | = <a href="#">Errno</a> (0x6c) |
| ESOCKTNOSUPPORT | = <a href="#">Errno</a> (0x5e) |
| ESPIPE          | = <a href="#">Errno</a> (0x1d) |
| ESRCH           | = <a href="#">Errno</a> (0x3)  |
| ESRMNT          | = <a href="#">Errno</a> (0x45) |
| ESTALE          | = <a href="#">Errno</a> (0x74) |
| ESTRPIPE        | = <a href="#">Errno</a> (0x56) |
| ETIME           | = <a href="#">Errno</a> (0x3e) |

```

ETIMEDOUT = Errno(0x6e)
ETOOMANYREFS = Errno(0x6d)
ETXTBSY = Errno(0x1a)
EUCLEAN = Errno(0x75)
EUNATCH = Errno(0x31)
EUSERS = Errno(0x57)
EWOULDBLOCK = Errno(0xb)
EXDEV = Errno(0x12)
EXFULL = Errno(0x36)
)

```

## Errors

```

const (
 SIGABRT = Signal(0x6)
 SIGALRM = Signal(0xe)
 SIGBUS = Signal(0x7)
 SIGCHLD = Signal(0x11)
 SIGCLD = Signal(0x11)
 SIGCONT = Signal(0x12)
 SIGFPE = Signal(0x8)
 SIGHUP = Signal(0x1)
 SIGILL = Signal(0x4)
 SIGINT = Signal(0x2)
 SIGIO = Signal(0x1d)
 SIGIOT = Signal(0x6)
 SIGKILL = Signal(0x9)
 SIGPIPE = Signal(0xd)
 SIGPOLL = Signal(0x1d)
 SIGPROF = Signal(0x1b)
 SIGPWR = Signal(0x1e)
 SIGQUIT = Signal(0x3)
 SIGSEGV = Signal(0xb)
 SIGSTKFLT = Signal(0x10)
 SIGSTOP = Signal(0x13)
 SIGSYS = Signal(0x1f)
 SIGTERM = Signal(0xf)
 SIGTRAP = Signal(0x5)
 SIGTSTP = Signal(0x14)
 SIGTTIN = Signal(0x15)
 SIGTTOU = Signal(0x16)
 SIGUNUSED = Signal(0x1f)
 SIGURG = Signal(0x17)
 SIGUSR1 = Signal(0xa)
 SIGUSR2 = Signal(0xc)
 SIGVTALRM = Signal(0x1a)
 SIGWINCH = Signal(0x1c)
 SIGXCPU = Signal(0x18)
 SIGXFSZ = Signal(0x19)
)

```

## Signals

```
const (
 SYS_READ = 0
 SYS_WRITE = 1
 SYS_OPEN = 2
 SYS_CLOSE = 3
 SYS_STAT = 4
 SYS_FSTAT = 5
 SYS_LSTAT = 6
 SYS_POLL = 7
 SYS_LSEEK = 8
 SYS_MMAP = 9
 SYS_MPROTECT = 10
 SYS_MUNMAP = 11
 SYS_BRK = 12
 SYS_RT_SIGACTION = 13
 SYS_RT_SIGPROCMASK = 14
 SYS_RT_SIGRETURN = 15
 SYS_IOCTL = 16
 SYS_PREAD64 = 17
 SYS_PWRITE64 = 18
 SYS_READV = 19
 SYS_WRITEV = 20
 SYS_ACCESS = 21
 SYS_PIPE = 22
 SYS_SELECT = 23
 SYS_SCHED_YIELD = 24
 SYS_MREMAP = 25
 SYS_MSYNC = 26
 SYS_MINCORE = 27
 SYS_MADVISE = 28
 SYS_SHMGET = 29
 SYS_SHMAT = 30
 SYS_SHMCTL = 31
 SYS_DUP = 32
 SYS_DUP2 = 33
 SYS_PAUSE = 34
 SYS_NANOSLEEP = 35
 SYS_GETITIMER = 36
 SYS_ALARM = 37
 SYS_SETITIMER = 38
 SYS_GETPID = 39
 SYS_SENDFILE = 40
 SYS_SOCKET = 41
 SYS_CONNECT = 42
 SYS_ACCEPT = 43
 SYS_SENDTO = 44
 SYS_RECVFROM = 45
 SYS_SENDSMSG = 46
 SYS_RECVMSGS = 47
 SYS_SHUTDOWN = 48
 SYS_BIND = 49
 SYS_LISTEN = 50
 SYS_GETSOCKNAME = 51
```

|                  |       |
|------------------|-------|
| SYS_GETPEERNAME  | = 52  |
| SYS_SOCKETPAIR   | = 53  |
| SYS_SETSOCKOPT   | = 54  |
| SYS_GETSOCKOPT   | = 55  |
| SYS_CLONE        | = 56  |
| SYS_FORK         | = 57  |
| SYS_VFORK        | = 58  |
| SYS_EXECVE       | = 59  |
| SYS_EXIT         | = 60  |
| SYS_WAIT4        | = 61  |
| SYS_KILL         | = 62  |
| SYS_UNAME        | = 63  |
| SYS_SEMGET       | = 64  |
| SYS_SEMOP        | = 65  |
| SYS_SEMCTL       | = 66  |
| SYS_SHMDT        | = 67  |
| SYS_MSGGET       | = 68  |
| SYS_MSGSND       | = 69  |
| SYS_MSGRCV       | = 70  |
| SYS_MSGCTL       | = 71  |
| SYS_FCNTL        | = 72  |
| SYS_FLOCK        | = 73  |
| SYS_FSYNC        | = 74  |
| SYS_FDATASYNC    | = 75  |
| SYS_TRUNCATE     | = 76  |
| SYS_FTRUNCATE    | = 77  |
| SYS_GETDENTS     | = 78  |
| SYS_GETCWD       | = 79  |
| SYS_CHDIR        | = 80  |
| SYS_FCHDIR       | = 81  |
| SYS_RENAME       | = 82  |
| SYS_MKDIR        | = 83  |
| SYS_RMDIR        | = 84  |
| SYS_CREAT        | = 85  |
| SYS_LINK         | = 86  |
| SYS_UNLINK       | = 87  |
| SYS_SYMLINK      | = 88  |
| SYS_READLINK     | = 89  |
| SYS_CHMOD        | = 90  |
| SYS_FCHMOD       | = 91  |
| SYS_CHOWN        | = 92  |
| SYS_FCHOWN       | = 93  |
| SYS_LCHOWN       | = 94  |
| SYS_UMASK        | = 95  |
| SYS_GETTIMEOFDAY | = 96  |
| SYS_GETRLIMIT    | = 97  |
| SYS_GETRUSAGE    | = 98  |
| SYS_SYSINFO      | = 99  |
| SYS_TIMES        | = 100 |
| SYS_PTRACE       | = 101 |
| SYS_GETUID       | = 102 |
| SYS_SYSLOG       | = 103 |
| SYS_GETGID       | = 104 |
| SYS_SETUID       | = 105 |

|                            |       |
|----------------------------|-------|
| SYS_SETGID                 | = 106 |
| SYS_GETEUID                | = 107 |
| SYS_GETEGID                | = 108 |
| SYS_SETPGID                | = 109 |
| SYS_GETPPID                | = 110 |
| SYS_GETPGRP                | = 111 |
| SYS_SETSID                 | = 112 |
| SYS_SETREUID               | = 113 |
| SYS_SETREGID               | = 114 |
| SYS_GETGROUPS              | = 115 |
| SYS_SETGROUPS              | = 116 |
| SYS_SETRESUID              | = 117 |
| SYS_GETRESUID              | = 118 |
| SYS_SETRESGID              | = 119 |
| SYS_GETRESGID              | = 120 |
| SYS_GETPGID                | = 121 |
| SYS_SETFSUID               | = 122 |
| SYS_SETFSGID               | = 123 |
| SYS_GETSID                 | = 124 |
| SYS_CAPGET                 | = 125 |
| SYS_CAPSET                 | = 126 |
| SYS_RT_SIGPENDING          | = 127 |
| SYS_RT_SIGTIMEDWAIT        | = 128 |
| SYS_RT_SIGQUEUEINFO        | = 129 |
| SYS_RT_SIGSUSPEND          | = 130 |
| SYS_SIGALTSTACK            | = 131 |
| SYS_UTIME                  | = 132 |
| SYS_MKNOD                  | = 133 |
| SYS_USELIB                 | = 134 |
| SYS_PERSONALITY            | = 135 |
| SYS_USTAT                  | = 136 |
| SYS_STATFS                 | = 137 |
| SYS_FSTATFS                | = 138 |
| SYS_SYSFS                  | = 139 |
| SYS_GETPRIORITY            | = 140 |
| SYS_SETPRIORITY            | = 141 |
| SYS_SCHED_SETPARAM         | = 142 |
| SYS_SCHED_GETPARAM         | = 143 |
| SYS_SCHED_SETSCHEDULER     | = 144 |
| SYS_SCHED_GETSCHEDULER     | = 145 |
| SYS_SCHED_GET_PRIORITY_MAX | = 146 |
| SYS_SCHED_GET_PRIORITY_MIN | = 147 |
| SYS_SCHED_RR_GET_INTERVAL  | = 148 |
| SYS_MLOCK                  | = 149 |
| SYS_MUNLOCK                | = 150 |
| SYS_MLOCKALL               | = 151 |
| SYS_MUNLOCKALL             | = 152 |
| SYS_VHANGUP                | = 153 |
| SYS MODIFY_LDT             | = 154 |
| SYS_PIVOT_ROOT             | = 155 |
| SYS__SYSCTL                | = 156 |
| SYS_PRCTL                  | = 157 |
| SYS_ARCH_PRCTL             | = 158 |
| SYS_ADJTIMEX               | = 159 |

|                       |       |
|-----------------------|-------|
| SYS_SETRLIMIT         | = 160 |
| SYS_CHROOT            | = 161 |
| SYS_SYNC              | = 162 |
| SYS_ACCT              | = 163 |
| SYS_SETTIMEOFDAY      | = 164 |
| SYS_MOUNT             | = 165 |
| SYS_UNMOUNT2          | = 166 |
| SYS_SWAPON            | = 167 |
| SYS_SWAPOFF           | = 168 |
| SYS_REBOOT            | = 169 |
| SYS_SETHOSTNAME       | = 170 |
| SYS_SETDOMAINNAME     | = 171 |
| SYS_IOPL              | = 172 |
| SYS_IOPERM            | = 173 |
| SYS_CREATE_MODULE     | = 174 |
| SYS_INIT_MODULE       | = 175 |
| SYS_DELETE_MODULE     | = 176 |
| SYS_GET_KERNEL_SYMS   | = 177 |
| SYS_QUERY_MODULE      | = 178 |
| SYS_QUOTACTL          | = 179 |
| SYS_NFSSERVCTL        | = 180 |
| SYS_GETPMSG           | = 181 |
| SYS_PUTPMSG           | = 182 |
| SYS_AFS_SYSCALL       | = 183 |
| SYS_TUXCALL           | = 184 |
| SYS_SECURITY          | = 185 |
| SYS_GETTID            | = 186 |
| SYS_READAHEAD         | = 187 |
| SYS_SETXATTR          | = 188 |
| SYS_LSETXATTR         | = 189 |
| SYS_FSETXATTR         | = 190 |
| SYS_GETXATTR          | = 191 |
| SYS_LGETXATTR         | = 192 |
| SYS_FGETXATTR         | = 193 |
| SYS_LISTXATTR         | = 194 |
| SYS_LLISTXATTR        | = 195 |
| SYS_FLISTXATTR        | = 196 |
| SYS_REMOVEXATTR       | = 197 |
| SYS_LREMOVEXATTR      | = 198 |
| SYS_FREMOVEXATTR      | = 199 |
| SYS_TKILL             | = 200 |
| SYS_TIME              | = 201 |
| SYS_FUTEX             | = 202 |
| SYS_SCHED_SETAFFINITY | = 203 |
| SYS_SCHED_GETAFFINITY | = 204 |
| SYS_SET_THREAD_AREA   | = 205 |
| SYS_IO_SETUP          | = 206 |
| SYS_IO_DESTROY        | = 207 |
| SYS_IO_GETEVENTS      | = 208 |
| SYS_IO_SUBMIT         | = 209 |
| SYS_IO_CANCEL         | = 210 |
| SYS_GET_THREAD_AREA   | = 211 |
| SYS_LOOKUP_DCOOKIE    | = 212 |
| SYS_EPOLL_CREATE      | = 213 |

|                       |       |
|-----------------------|-------|
| SYS_EPOLL_CTL_OLD     | = 214 |
| SYS_EPOLL_WAIT_OLD    | = 215 |
| SYS_REMAP_FILE_PAGES  | = 216 |
| SYS_GETDENTS64        | = 217 |
| SYS_SET_TID_ADDRESS   | = 218 |
| SYS_RESTART_SYSCALL   | = 219 |
| SYS_SEMTIMEDOP        | = 220 |
| SYS_FADVISE64         | = 221 |
| SYS_TIMER_CREATE      | = 222 |
| SYS_TIMER_SETTIME     | = 223 |
| SYS_TIMER_GETTIME     | = 224 |
| SYS_TIMER_GETOVERRUN  | = 225 |
| SYS_TIMER_DELETE      | = 226 |
| SYS_CLOCK_SETTIME     | = 227 |
| SYS_CLOCK_GETTIME     | = 228 |
| SYS_CLOCK_GETRES      | = 229 |
| SYS_CLOCK_NANOSLEEP   | = 230 |
| SYS_EXIT_GROUP        | = 231 |
| SYS_EPOLL_WAIT        | = 232 |
| SYS_EPOLL_CTL         | = 233 |
| SYS_TGKILL            | = 234 |
| SYS_UTIMES            | = 235 |
| SYS_VSERVER           | = 236 |
| SYS_MBIND             | = 237 |
| SYS_SET_MEMPOLICY     | = 238 |
| SYS_GET_MEMPOLICY     | = 239 |
| SYS_MQ_OPEN           | = 240 |
| SYS_MQ_UNLINK         | = 241 |
| SYS_MQ_TIMEDSEND      | = 242 |
| SYS_MQ_TIMEDRECEIVE   | = 243 |
| SYS_MQ_NOTIFY         | = 244 |
| SYS_MQ_GETSETATTR     | = 245 |
| SYS_KEXEC_LOAD        | = 246 |
| SYS_WAITID            | = 247 |
| SYS_ADD_KEY           | = 248 |
| SYS_REQUEST_KEY       | = 249 |
| SYS_KEYCTL            | = 250 |
| SYS_IOPRIO_SET        | = 251 |
| SYS_IOPRIO_GET        | = 252 |
| SYS_INOTIFY_INIT      | = 253 |
| SYS_INOTIFY_ADD_WATCH | = 254 |
| SYS_INOTIFY_RM_WATCH  | = 255 |
| SYS_MIGRATE_PAGES     | = 256 |
| SYS_OPENAT            | = 257 |
| SYS_MKDIRAT           | = 258 |
| SYS_MKNODAT           | = 259 |
| SYS_FCHOWNAT          | = 260 |
| SYS_FUTIMESAT         | = 261 |
| SYS_NEWFSTATAT        | = 262 |
| SYS_UNLINKAT          | = 263 |
| SYS_RENAMEAT          | = 264 |
| SYS_LINKAT            | = 265 |
| SYS_SYMLINKAT         | = 266 |
| SYS_READLINKAT        | = 267 |

|                       |       |
|-----------------------|-------|
| SYS_FCHMODAT          | = 268 |
| SYS_FACCESSAT         | = 269 |
| SYS_PSELECT6          | = 270 |
| SYS_PPOLL             | = 271 |
| SYS_UNSHARE           | = 272 |
| SYS_SET_ROBUST_LIST   | = 273 |
| SYS_GET_ROBUST_LIST   | = 274 |
| SYS_SPLICE            | = 275 |
| SYS_TEE               | = 276 |
| SYS_SYNC_FILE_RANGE   | = 277 |
| SYS_VMSPLICE          | = 278 |
| SYS_MOVE_PAGES        | = 279 |
| SYS_UTIMENSAT         | = 280 |
| SYS_EPOLL_PWAIT       | = 281 |
| SYS_SIGNALFD          | = 282 |
| SYS_TIMERFD_CREATE    | = 283 |
| SYS_EVENTFD           | = 284 |
| SYS_FALLOCATE         | = 285 |
| SYS_TIMERFD_SETTIME   | = 286 |
| SYS_TIMERFD_GETTIME   | = 287 |
| SYS_ACCEPT4           | = 288 |
| SYS_SIGNALFD4         | = 289 |
| SYS_EVENTFD2          | = 290 |
| SYS_EPOLL_CREATE1     | = 291 |
| SYS_DUP3              | = 292 |
| SYS_PIPE2             | = 293 |
| SYS_INOTIFY_INIT1     | = 294 |
| SYS_PREADV            | = 295 |
| SYS_PWRITEV           | = 296 |
| SYS_RT_TGSIGQUEUEINFO | = 297 |
| SYS_PERF_EVENT_OPEN   | = 298 |
| SYS_RECVMMMSG         | = 299 |
| SYS_FANOTIFY_INIT     | = 300 |
| SYS_FANOTIFY_MARK     | = 301 |
| SYS_PRLIMIT64         | = 302 |

)

|                         |        |
|-------------------------|--------|
| const (                 |        |
| SizeofSockaddrInet4     | = 0x10 |
| SizeofSockaddrInet6     | = 0x1c |
| SizeofSockaddrAny       | = 0x70 |
| SizeofSockaddrUnix      | = 0x6e |
| SizeofSockaddrLinklayer | = 0x14 |
| SizeofSockaddrNetlink   | = 0xc  |
| SizeofLinger            | = 0x8  |
| SizeofIPMreq            | = 0x8  |
| SizeofIPMreqn           | = 0xc  |
| SizeofIPv6Mreq          | = 0x14 |
| SizeofMsghdr            | = 0x38 |
| SizeofCmsghdr           | = 0x10 |
| SizeofInet4Pktinfo      | = 0xc  |
| SizeofInet6Pktinfo      | = 0x14 |
| SizeofIPv6MTUInfo       | = 0x20 |

```
SizeofICMPv6Filter = 0x20
SizeofUcred = 0xc
SizeofTCPIInfo = 0x68
)
```

```
const (
 IFA_UNSPEC = 0x0
 IFA_ADDRESS = 0x1
 IFA_LOCAL = 0x2
 IFA_LABEL = 0x3
 IFA_BROADCAST = 0x4
 IFA_ANYCAST = 0x5
 IFA_CACHEINFO = 0x6
 IFA_MULTICAST = 0x7
 IFLA_UNSPEC = 0x0
 IFLA_ADDRESS = 0x1
 IFLA_BROADCAST = 0x2
 IFLA_IFNAME = 0x3
 IFLA_MTU = 0x4
 IFLA_LINK = 0x5
 IFLA_QDISC = 0x6
 IFLA_STATS = 0x7
 IFLA_COST = 0x8
 IFLA_PRIORITY = 0x9
 IFLA_MASTER = 0xa
 IFLA_WIRELESS = 0xb
 IFLA_PROTINFO = 0xc
 IFLA_TXQLEN = 0xd
 IFLA_MAP = 0xe
 IFLA_WEIGHT = 0xf
 IFLA_OPERSTATE = 0x10
 IFLA_LINKMODE = 0x11
 IFLA_LINKINFO = 0x12
 IFLA_NET_NS_PID = 0x13
 IFLA_IFALIAS = 0x14
 IFLA_MAX = 0x1d
 RT_SCOPE_UNIVERSE = 0x0
 RT_SCOPE_SITE = 0xc8
 RT_SCOPE_LINK = 0xfd
 RT_SCOPE_HOST = 0xfe
 RT_SCOPE_NOWHERE = 0xff
 RT_TABLE_UNSPEC = 0x0
 RT_TABLE_COMPAT = 0xfc
 RT_TABLE_DEFAULT = 0xfd
 RT_TABLE_MAIN = 0xfe
 RT_TABLE_LOCAL = 0xff
 RT_TABLE_MAX = 0xffffffff
 RTA_UNSPEC = 0x0
 RTA_DST = 0x1
 RTA_SRC = 0x2
 RTA_IIF = 0x3
 RTA_OIF = 0x4
 RTA_GATEWAY = 0x5
```

```

RTA_PRIORITY = 0x6
RTA_PREFSRC = 0x7
RTA_METRICS = 0x8
RTA_MULTIPATH = 0x9
RTA_FLOW = 0xb
RTA_CACHEINFO = 0xc
RTA_TABLE = 0xf
RTN_UNSPEC = 0x0
RTN_UNICAST = 0x1
RTN_LOCAL = 0x2
RTN_BROADCAST = 0x3
RTN_ANYCAST = 0x4
RTN_MULTICAST = 0x5
RTN_BLACKHOLE = 0x6
RTN_UNREACHABLE = 0x7
RTN_PROHIBIT = 0x8
RTN_THROW = 0x9
RTN_NAT = 0xa
RTN_XRESOLVE = 0xb
RTNLGRP_NONE = 0x0
RTNLGRP_LINK = 0x1
RTNLGRP_NOTIFY = 0x2
RTNLGRP_NEIGH = 0x3
RTNLGRP_TC = 0x4
RTNLGRP_IPV4_IFADDR = 0x5
RTNLGRP_IPV4_MROUTE = 0x6
RTNLGRP_IPV4_ROUTE = 0x7
RTNLGRP_IPV4_RULE = 0x8
RTNLGRP_IPV6_IFADDR = 0x9
RTNLGRP_IPV6_MROUTE = 0xa
RTNLGRP_IPV6_ROUTE = 0xb
RTNLGRP_IPV6_IFINFO = 0xc
RTNLGRP_IPV6_PREFIX = 0x12
RTNLGRP_IPV6_RULE = 0x13
RTNLGRP_ND_USEROPT = 0x14
SizeofNlMsgHdr = 0x10
SizeofNlMsgerr = 0x14
SizeofRtGenmsg = 0x1
SizeofNlAttr = 0x4
SizeofRtAttr = 0x4
SizeofIfInfomsg = 0x10
SizeofIfAddrmsg = 0x8
SizeofRtMsg = 0xc
SizeofRtNexthop = 0x8
)

```

```

const (
 SizeofSockFilter = 0x8
 SizeofSockFprog = 0x10
)

```

```

const (
 VINTR = 0x0
)

```

|          |          |
|----------|----------|
| VQUIT    | = 0x1    |
| VERASE   | = 0x2    |
| VKILL    | = 0x3    |
| VEOF     | = 0x4    |
| VTIME    | = 0x5    |
| VMIN     | = 0x6    |
| VSWTC    | = 0x7    |
| VSTART   | = 0x8    |
| VSTOP    | = 0x9    |
| VSUSP    | = 0xa    |
| VEOL     | = 0xb    |
| VREPRINT | = 0xc    |
| VDISCARD | = 0xd    |
| VWERASE  | = 0xe    |
| VLNEXT   | = 0xf    |
| VEOL2    | = 0x10   |
| IGNBRK   | = 0x1    |
| BRKINT   | = 0x2    |
| IGNPAR   | = 0x4    |
| PARMRK   | = 0x8    |
| INPCK    | = 0x10   |
| ISTRIP   | = 0x20   |
| INLCR    | = 0x40   |
| IGNCR    | = 0x80   |
| ICRNL    | = 0x100  |
| IUCLC    | = 0x200  |
| IXON     | = 0x400  |
| IXANY    | = 0x800  |
| IXOFF    | = 0x1000 |
| IMAXBEL  | = 0x2000 |
| IUTF8    | = 0x4000 |
| OPOST    | = 0x1    |
| OLCUC    | = 0x2    |
| ONLCR    | = 0x4    |
| OCRNL    | = 0x8    |
| ONOCR    | = 0x10   |
| ONLRET   | = 0x20   |
| OFILL    | = 0x40   |
| OFDEL    | = 0x80   |
| B0       | = 0x0    |
| B50      | = 0x1    |
| B75      | = 0x2    |
| B110     | = 0x3    |
| B134     | = 0x4    |
| B150     | = 0x5    |
| B200     | = 0x6    |
| B300     | = 0x7    |
| B600     | = 0x8    |
| B1200    | = 0x9    |
| B1800    | = 0xa    |
| B2400    | = 0xb    |
| B4800    | = 0xc    |
| B9600    | = 0xd    |
| B19200   | = 0xe    |

```
B38400 = 0xf
CSIZE = 0x30
CS5 = 0x0
CS6 = 0x10
CS7 = 0x20
CS8 = 0x30
CSTOPB = 0x40
CREAD = 0x80
PARENB = 0x100
PARODD = 0x200
HUPCL = 0x400
CLOCAL = 0x800
B57600 = 0x1001
B115200 = 0x1002
B230400 = 0x1003
B460800 = 0x1004
B500000 = 0x1005
B576000 = 0x1006
B921600 = 0x1007
B1000000 = 0x1008
B1152000 = 0x1009
B1500000 = 0x100a
B2000000 = 0x100b
B2500000 = 0x100c
B3000000 = 0x100d
B3500000 = 0x100e
B4000000 = 0x100f
ISIG = 0x1
ICANON = 0x2
XCASE = 0x4
ECHO = 0x8
ECHOE = 0x10
ECHOK = 0x20
ECHONL = 0x40
NOFLSH = 0x80
TOSTOP = 0x100
ECHOCTL = 0x200
ECHOPRT = 0x400
ECHOKE = 0x800
FLUSHO = 0x1000
PENDIN = 0x4000
IEXTEN = 0x8000
TCGETS = 0x5401
TCSETS = 0x5402
)
```

```
const ImplementsGetwd = true
```

```
const (
 PathMax = 0x1000
)
```

```
const SizeofInotifyEvent = 0x10
```

## Variables

```
var (
 Stdin = 0
 Stdout = 1
 Stderr = 2
)
```

```
var ForkLock sync.RWMutex
```

```
var SocketDisableIPv6 bool
```

For testing: clients can set this flag to force creation of IPv6 sockets to return EAFNOSUPPORT.

## func Access

```
func Access(path string, mode uint32) (err error)
```

## func Acct

```
func Acct(path string) (err error)
```

## func Adjtimex

```
func Adjtimex(buf *Timex) (state int, err error)
```

## func AttachLsf

```
func AttachLsf(fd int, i []SockFilter) error
```

Deprecated: Use [golang.org/x/net/bpf](https://golang.org/x/net/bpf) instead.

## func Bind

```
func Bind(fd int, sa Sockaddr) (err error)
```

## func BindToDevice

```
func BindToDevice(fd int, device string) (err error)
```

BindToDevice binds the socket associated with fd to device.

## func **BytePtrFromString**

```
func BytePtrFromString(s string) (*byte, error)
```

BytePtrFromString returns a pointer to a NUL-terminated array of bytes containing the text of s. If s contains a NUL byte at any location, it returns (nil, EINVAL).

## func **ByteSliceFromString**

```
func ByteSliceFromString(s string) ([]byte, error)
```

ByteSliceFromString returns a NUL-terminated slice of bytes containing the text of s. If s contains a NUL byte at any location, it returns (nil, EINVAL).

## func **Chdir**

```
func Chdir(path string) (err error)
```

## func **Chmod**

```
func Chmod(path string, mode uint32) (err error)
```

## func **Chown**

```
func Chown(path string, uid int, gid int) (err error)
```

## func **Chroot**

```
func Chroot(path string) (err error)
```

## func **Clearenv**

```
func Clearenv()
```

## func **Close**

```
func Close(fd int) (err error)
```

## func **CloseOnExec**

```
func CloseOnExec(fd int)
```

## func **CmsgLen**

```
func CmsgLen(datalen int) int
```

CmsgLen returns the value to store in the Len field of the Cmsghdr structure, taking into account any necessary alignment.

## func **CmsgSpace**

```
func CmsgSpace(datalen int) int
```

CmsgSpace returns the number of bytes an ancillary element with payload of the passed data length occupies.

## func **Connect**

```
func Connect(fd int, sa Sockaddr) (err error)
```

## func **Create**

```
func Create(path string, mode uint32) (fd int, err error)
```

## func **DetachLsf**

```
func DetachLsf(fd int) error
```

Deprecated: Use [golang.org/x/net/bpf](https://golang.org/x/net/bpf) instead.

## func **Dup**

```
func Dup(olddfd int) (fd int, err error)
```

## func **Dup2**

```
func Dup2(olddfd int, newfd int) (err error)
```

## func **Dup3**

```
func Dup3(olddfd int, newfd int, flags int) (err error)
```

## func **Environ**

```
func Environ() []string
```

## func **EpollCreate**

```
func EpollCreate(size int) (fd int, err error)
```

## func **EpollCreate1**

```
func EpollCreate1(flag int) (fd int, err error)
```

## func **EpollCtl**

```
func EpollCtl(epfd int, op int, fd int, event *EpollEvent) (err error)
```

## func **EpollWait**

```
func EpollWait(epfd int, events []EpollEvent, msec int) (n int, err error)
```

## func **Exec**

```
func Exec(argv0 string, argv []string, envv []string) (err error)
```

Exec invokes the execve(2) system call.

## func **Exit**

```
func Exit(code int)
```

## func **Faccessat**

```
func Faccessat(dirfd int, path string, mode uint32, flags int) (err error)
```

## func **Fallocate**

```
func Fallocate(fd int, mode uint32, off int64, len int64) (err error)
```

## func **Fchdir**

```
func Fchdir(fd int) (err error)
```

## func **Fchmod**

```
func Fchmod(fd int, mode uint32) (err error)
```

## func **Fchmodat**

```
func Fchmodat(dirfd int, path string, mode uint32, flags int) (err error)
```

## func Fchown

```
func Fchown(fd int, uid int, gid int) (err error)
```

## func Fchownat

```
func Fchownat(dirfd int, path string, uid int, gid int, flags int) (err error)
```

## func FcntlFlock

```
func FcntlFlock(fd uintptr, cmd int, lk *Flock_t) error
```

FcntlFlock performs a fcntl syscall for the F\_GETLK, F\_SETLK or F\_SETLKW command.

## func Fdatasync

```
func Fdatasync(fd int) (err error)
```

## func Flock

```
func Flock(fd int, how int) (err error)
```

## func ForkExec

```
func ForkExec(argv0 string, argv []string, attr *ProcAttr) (pid int, err error)
```

Combination of fork and exec, careful to be thread safe.

## func Fstat

```
func Fstat(fd int, stat *Stat_t) (err error)
```

## func Fstatfs

```
func Fstatfs(fd int, buf *Statfs_t) (err error)
```

## func Fsync

```
func Fsync(fd int) (err error)
```

## func **Ftruncate**

```
func Ftruncate(fd int, length int64) (err error)
```

## func **Futimes**

```
func Futimes(fd int, tv []Timeval) (err error)
```

## func **Futimesat**

```
func Futimesat(dirfd int, path string, tv []Timeval) (err error)
```

## func **Getcwd**

```
func Getcwd(buf []byte) (n int, err error)
```

## func **Getdents**

```
func Getdents(fd int, buf []byte) (n int, err error)
```

## func **Getegid**

```
func Getegid() (egid int)
```

## func **Getenv**

```
func Getenv(key string) (value string, found bool)
```

## func **Geteuid**

```
func Geteuid() (euid int)
```

## func **Getgid**

```
func Getgid() (gid int)
```

## func **Getgroups**

```
func Getgroups() (gids []int, err error)
```

## func **Getpagesize**

```
func Getpagesize() int
```

## func **Getpgid**

```
func Getpgid(pid int) (pgid int, err error)
```

## func **Getpgrp**

```
func Getpgrp() (pid int)
```

## func **Getpid**

```
func Getpid() (pid int)
```

## func **Getppid**

```
func Getppid() (ppid int)
```

## func **Getpriority**

```
func Getpriority(which int, who int) (prio int, err error)
```

## func **Getrlimit**

```
func Getrlimit(resource int, rlim *Rlimit) (err error)
```

## func **Getrusage**

```
func Getrusage(who int, rusage *Rusage) (err error)
```

## func **GetsockoptInet4Addr**

```
func GetsockoptInet4Addr(fd, level, opt int) (value [4]byte, err error)
```

## func **GetsockoptInt**

```
func GetsockoptInt(fd, level, opt int) (value int, err error)
```

## func **Gettid**

```
func Gettid() (tid int)
```

## func **Gettimeofday**

```
func Gettimeofday(tv *Timeval) (err error)
```

## func **Getuid**

```
func Getuid() (uid int)
```

## func **Getwd**

```
func Getwd() (wd string, err error)
```

## func **Getxattr**

```
func Getxattr(path string, attr string, dest []byte) (sz int, err error)
```

## func **InotifyAddWatch**

```
func InotifyAddWatch(fd int, pathname string, mask uint32) (watchdesc int, err error)
```

## func **InotifyInit**

```
func InotifyInit() (fd int, err error)
```

## func **InotifyInit1**

```
func InotifyInit1(flags int) (fd int, err error)
```

## func **InotifyRmWatch**

```
func InotifyRmWatch(fd int, watchdesc uint32) (success int, err error)
```

## func **Ioperm**

```
func Ioperm(from int, num int, on int) (err error)
```

## func **Iopl**

```
func Iopl(level int) (err error)
```

## func **Kill**

```
func Kill(pid int, sig Signal) (err error)
```

## func Klogctl

```
func Klogctl(typ int, buf []byte) (n int, err error)
```

## func Lchown

```
func Lchown(path string, uid int, gid int) (err error)
```

## func Link

```
func Link(oldpath string, newpath string) (err error)
```

## func Listen

```
func Listen(s int, n int) (err error)
```

## func Listxattr

```
func Listxattr(path string, dest []byte) (sz int, err error)
```

## func LsfSocket

```
func LsfSocket(ifindex, proto int) (int, error)
```

Deprecated: Use [golang.org/x/net/bpf](https://golang.org/x/net/bpf) instead.

## func Lstat

```
func Lstat(path string, stat *Stat_t) (err error)
```

## func Madvise

```
func Madvise(b []byte, advice int) (err error)
```

## func Mkdir

```
func Mkdir(path string, mode uint32) (err error)
```

## func Mkdirat

```
func Mkdirat(dirfd int, path string, mode uint32) (err error)
```

## func **Mkfifo**

```
func Mkfifo(path string, mode uint32) (err error)
```

## func **Mknod**

```
func Mknod(path string, mode uint32, dev int) (err error)
```

## func **Mknodat**

```
func Mknodat(dirfd int, path string, mode uint32, dev int) (err error)
```

## func **Mlock**

```
func Mlock(b []byte) (err error)
```

## func **Mlockall**

```
func Mlockall(flags int) (err error)
```

## func **Mmap**

```
func Mmap(fd int, offset int64, length int, prot int, flags int) (data []byte, err er
```

## func **Mount**

```
func Mount(source string, target string, fstype string, flags uintptr, data string) (
```

## func **Mprotect**

```
func Mprotect(b []byte, prot int) (err error)
```

## func **Munlock**

```
func Munlock(b []byte) (err error)
```

## func **Munlockall**

```
func Munlockall() (err error)
```

## func Munmap

```
func Munmap(b []byte) (err error)
```

## func Nanosleep

```
func Nanosleep(time *Timespec, leftover *Timespec) (err error)
```

## func NetlinkRIB

```
func NetlinkRIB(proto, family int) ([]byte, error)
```

NetlinkRIB returns routing information base, as known as RIB, which consists of network facility information, states and parameters.

## func Open

```
func Open(path string, mode int, perm uint32) (fd int, err error)
```

## func Openat

```
func Openat(dirfd int, path string, flags int, mode uint32) (fd int, err error)
```

## func ParseDirent

```
func ParseDirent(buf []byte, max int, names []string) (consumed int, count int, newnames []string)
```

ParseDirent parses up to max directory entries in buf, appending the names to names. It returns the number of bytes consumed from buf, the number of entries added to names, and the new names slice.

## func ParseUnixRights

```
func ParseUnixRights(m *SocketControlMessage) ([]int, error)
```

ParseUnixRights decodes a socket control message that contains an integer array of open file descriptors from another process.

## func Pause

```
func Pause() (err error)
```

## func Pipe

```
func Pipe(p []int) (err error)
```

## func Pipe2

```
func Pipe2(p []int, flags int) (err error)
```

## func PivotRoot

```
func PivotRoot(newroot string, putold string) (err error)
```

## func Pread

```
func Pread(fd int, p []byte, offset int64) (n int, err error)
```

## func PtraceAttach

```
func PtraceAttach(pid int) (err error)
```

## func PtraceCont

```
func PtraceCont(pid int, signal int) (err error)
```

## func PtraceDetach

```
func PtraceDetach(pid int) (err error)
```

## func PtraceGetEventMsg

```
func PtraceGetEventMsg(pid int) (msg uint, err error)
```

## func PtraceGetRegs

```
func PtraceGetRegs(pid int, regsout *PtraceRegs) (err error)
```

## func PtracePeekData

```
func PtracePeekData(pid int, addr uintptr, out []byte) (count int, err error)
```

## func PtracePeekText

```
func PtracePeekText(pid int, addr uintptr, out []byte) (count int, err error)
```

## func **PtracePokeData**

```
func PtracePokeData(pid int, addr uintptr, data []byte) (count int, err error)
```

## func **PtracePokeText**

```
func PtracePokeText(pid int, addr uintptr, data []byte) (count int, err error)
```

## func **PtraceSetOptions**

```
func PtraceSetOptions(pid int, options int) (err error)
```

## func **PtraceSetRegs**

```
func PtraceSetRegs(pid int, regs *PtraceRegs) (err error)
```

## func **PtraceSingleStep**

```
func PtraceSingleStep(pid int) (err error)
```

## func **PtraceSyscall**

```
func PtraceSyscall(pid int, signal int) (err error)
```

## func **Pwrite**

```
func Pwrite(fd int, p []byte, offset int64) (n int, err error)
```

## func **Read**

```
func Read(fd int, p []byte) (n int, err error)
```

## func **ReadDirent**

```
func ReadDirent(fd int, buf []byte) (n int, err error)
```

## func **Readlink**

```
func Readlink(path string, buf []byte) (n int, err error)
```

## func **Reboot**

```
func Reboot(cmd int) (err error)
```

## func **Removexattr**

```
func Removexattr(path string, attr string) (err error)
```

## func **Rename**

```
func Rename(oldpath string, newpath string) (err error)
```

## func **Renameat**

```
func Renameat(olddirfd int, oldpath string, newdirfd int, newpath string) (err error)
```

## func **Rmdir**

```
func Rmdir(path string) error
```

## func **Seek**

```
func Seek(fd int, offset int64, whence int) (off int64, err error)
```

## func **Select**

```
func Select(nfd int, r *FdSet, w *FdSet, e *FdSet, timeout *Timeval) (n int, err error)
```

## func **Sendfile**

```
func Sendfile(outfd int, infd int, offset *int64, count int) (written int, err error)
```

## func **Sendmsg**

```
func Sendmsg(fd int, p, oob []byte, to Sockaddr, flags int) (err error)
```

## func **SendmsgN**

```
func SendmsgN(fd int, p, oob []byte, to Sockaddr, flags int) (n int, err error)
```

## func **Sendto**

```
func Sendto(fd int, p []byte, flags int, to Sockaddr) (err error)
```

## func **SetLsfPromisc**

```
func SetLsfPromisc(name string, m bool) error
```

Deprecated: Use [golang.org/x/net/bpf](https://golang.org/x/net/bpf) instead.

## func **SetNonblock**

```
func SetNonblock(fd int, nonblocking bool) (err error)
```

## func **Setdomainname**

```
func Setdomainname(p []byte) (err error)
```

## func **Setenv**

```
func Setenv(key, value string) error
```

## func **Setfsgid**

```
func Setfsgid(gid int) (err error)
```

## func **Setfsuid**

```
func Setfsuid(uid int) (err error)
```

## func **Setgid**

```
func Setgid(gid int) (err error)
```

## func **Setgroups**

```
func Setgroups(gids []int) (err error)
```

## func **Sethostname**

```
func Sethostname(p []byte) (err error)
```

## func **Setpgid**

```
func Setpgid(pid int, pgid int) (err error)
```

## func **Setpriority**

```
func Setpriority(which int, who int, prio int) (err error)
```

## func **Setregid**

```
func Setregid(rgid int, egid int) (err error)
```

## func **Setresgid**

```
func Setresgid(rgid int, egid int, sgid int) (err error)
```

## func **Setresuid**

```
func Setresuid(ruid int, euid int, suid int) (err error)
```

## func **Setreuid**

```
func Setreuid(ruid int, euid int) (err error)
```

## func **Setrlimit**

```
func Setrlimit(resource int, rlim *Rlimit) (err error)
```

## func **Setsid**

```
func Setsid() (pid int, err error)
```

## func **SetsockoptByte**

```
func SetsockoptByte(fd, level, opt int, value byte) (err error)
```

## func **SetsockoptICMPv6Filter**

```
func SetsockoptICMPv6Filter(fd, level, opt int, filter *ICMPv6Filter) error
```

## func **SetsockoptIPMreq**

```
func SetsockoptIPMreq(fd, level, opt int, mreq *IPMreq) (err error)
```

## func **SetsockoptIPMreqn**

```
func SetsockoptIPMreqn(fd, level, opt int, mreq *IPMreqn) (err error)
```

## func **SetsockoptIPv6Mreq**

```
func SetsockoptIPv6Mreq(fd, level, opt int, mreq *IPv6Mreq) (err error)
```

## func **SetsockoptInet4Addr**

```
func SetsockoptInet4Addr(fd, level, opt int, value [4]byte) (err error)
```

## func **SetsockoptInt**

```
func SetsockoptInt(fd, level, opt int, value int) (err error)
```

## func **SetsockoptLinger**

```
func SetsockoptLinger(fd, level, opt int, l *Linger) (err error)
```

## func **SetsockoptString**

```
func SetsockoptString(fd, level, opt int, s string) (err error)
```

## func **SetsockoptTimeval**

```
func SetsockoptTimeval(fd, level, opt int, tv *Timeval) (err error)
```

## func **Settimeofday**

```
func Settimeofday(tv *Timeval) (err error)
```

## func **Setuid**

```
func Setuid(uid int) (err error)
```

## func **Setxattr**

```
func Setxattr(path string, attr string, data []byte, flags int) (err error)
```

## func **Shutdown**

```
func Shutdown(fd int, how int) (err error)
```

## func SlicePtrFromStrings

```
func SlicePtrFromStrings(ss []string) ([]*byte, error)
```

SlicePtrFromStrings converts a slice of strings to a slice of pointers to NUL-terminated byte arrays. If any string contains a NUL byte, it returns (nil, EINVAL).

## func Socket

```
func Socket(domain, typ, proto int) (fd int, err error)
```

## func Socketpair

```
func Socketpair(domain, typ, proto int) (fd [2]int, err error)
```

## func Splice

```
func Splice(rfd int, roff *int64, wfd int, woff *int64, len int, flags int) (n int64,
```

## func StartProcess

```
func StartProcess(argv0 string, argv []string, attr *ProcAttr) (pid int, handle uintp
```

StartProcess wraps ForkExec for package os.

## func Stat

```
func Stat(path string, stat *Stat_t) (err error)
```

## func Statfs

```
func Statfs(path string, buf *Statfs_t) (err error)
```

## func StringBytePtr

```
func StringBytePtr(s string) *byte
```

StringBytePtr returns a pointer to a NUL-terminated array of bytes. If s contains a NUL byte this function panics instead of returning an error.

Deprecated: Use BytePtrFromString instead.

## func StringByteSlice

```
func StringByteSlice(s string) []byte
```

StringByteSlice converts a string to a NUL-terminated **[]byte**. If **s** contains a NUL byte this function panics instead of returning an error.

Deprecated: Use **ByteSliceFromString** instead.

## func **StringSlicePtr**

```
func StringSlicePtr(ss []string) []*byte
```

StringSlicePtr converts a slice of strings to a slice of pointers to NUL-terminated byte arrays. If any string contains a NUL byte this function panics instead of returning an error.

Deprecated: Use **SlicePtrFromStrings** instead.

## func **Symlink**

```
func Symlink(oldpath string, newpath string) (err error)
```

## func **Sync**

```
func Sync()
```

## func **SyncFileRange**

```
func SyncFileRange(fd int, off int64, n int64, flags int) (err error)
```

## func **Sysinfo**

```
func Sysinfo(info *Sysinfo_t) (err error)
```

## func **Tee**

```
func Tee(rfd int, wfd int, len int, flags int) (n int64, err error)
```

## func **Tgkill**

```
func Tgkill(tgid int, tid int, sig Signal) (err error)
```

## func **Times**

```
func Times(tms *Tms) (ticks uintptr, err error)
```

## func TimespecToNsec

```
func TimespecToNsec(ts Timespec) int64
```

TimespecToNsec converts a Timespec value into a number of nanoseconds since the Unix epoch.

## func TimevalToNsec

```
func TimevalToNsec(tv Timeval) int64
```

TimevalToNsec converts a Timeval value into a number of nanoseconds since the Unix epoch.

## func Truncate

```
func Truncate(path string, length int64) (err error)
```

## func Umask

```
func Umask(mask int) (oldmask int)
```

## func Uname

```
func Uname(buf *Utsname) (err error)
```

## func UnixCredentials

```
func UnixCredentials(ucred *Ucred) []byte
```

UnixCredentials encodes credentials into a socket control message for sending to another process. This can be used for authentication.

## func UnixRights

```
func UnixRights(fds ...int) []byte
```

UnixRights encodes a set of open file descriptors into a socket control message for sending to another process.

## func Unlink

```
func Unlink(path string) error
```

## func Unlinkat

```
func Unlinkat(dirfd int, path string) error
```

## func **Unmount**

```
func Unmount(target string, flags int) (err error)
```

## func **Unsetenv**

```
func Unsetenv(key string) error
```

## func **Unshare**

```
func Unshare(flags int) (err error)
```

## func **Ustat**

```
func Ustat(dev int, ubuf *Ustat_t) (err error)
```

## func **Utime**

```
func Utime(path string, buf *Utimbuf) (err error)
```

## func **Utimes**

```
func Utimes(path string, tv []Timeval) (err error)
```

## func **UtimesNano**

```
func UtimesNano(path string, ts []Timespec) (err error)
```

## func **Wait4**

```
func Wait4(pid int, wstatus *WaitStatus, options int, rusage *Rusage) (wpid int, err
```

## func **Write**

```
func Write(fd int, p []byte) (n int, err error)
```

## type **Cmsghdr**

```
type Cmsghdr struct {
 Len uint64
```

```
 Level int32
 Type int32
}
```

## func (\*Cmsghdr) SetLen

```
func (cmsg *Cmsghdr) SetLen(length int)
```

## type Conn

```
type Conn interface {
 // SyscallConn returns a raw network connection.
 SyscallConn() (RawConn, error)
}
```

Conn is implemented by some types in the net and os packages to provide access to the underlying file descriptor or handle.

## type Credential

```
type Credential struct {
 Uid uint32 // User ID.
 Gid uint32 // Group ID.
 Groups []uint32 // Supplementary group IDs.
 NoSetGroups bool // If true, don't set supplementary groups
}
```

Credential holds user and group identities to be assumed by a child process started by StartProcess.

## type Dirent

```
type Dirent struct {
 Ino uint64
 Off int64
 Reclen uint16
 Type uint8
 Name [256]int8
 Pad_cgo_0 [5]byte
}
```

## type EpollEvent

```
type EpollEvent struct {
 Events uint32
 Fd int32
 Pad int32
}
```

## type Errno

```
type Errno uintptr
```

An Errno is an unsigned number describing an error condition. It implements the error interface. The zero Errno is by convention a non-error, so code to convert from Errno to error should use:

```
err = nil
if errno != 0 {
 err = errno
}
```

Errno values can be tested against error values from the os package using errors.Is. For example:

```
_, _, err := syscall.Syscall(...)
if errors.Is(err, os.ErrNotExist) ...
```

## func RawSyscall

```
func RawSyscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno)
```

## func RawSyscall6

```
func RawSyscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr, err Errno)
```

## func Syscall

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno)
```

## func Syscall6

```
func Syscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr, err Errno)
```

## func (Errno) Error

```
func (e Errno) Error() string
```

## func (Errno) Is

```
func (e Errno) Is(target error) bool
```

## func (Errno) Temporary

```
func (e Errno) Temporary() bool
```

## func (Errno) Timeout

```
func (e Errno) Timeout() bool
```

## type FdSet

```
type FdSet struct {
 Bits [16]int64
}
```

## type Flock\_t

```
type Flock_t struct {
 Type int16
 Whence int16
 Pad_cgo_0 [4]byte
 Start int64
 Len int64
 Pid int32
 Pad_cgo_1 [4]byte
}
```

## type Fsid

```
type Fsid struct {
 X_val [2]int32
}
```

## type ICMPv6Filter

```
type ICMPv6Filter struct {
 Data [8]uint32
}
```

## func GetsockoptICMPv6Filter

```
func GetsockoptICMPv6Filter(fd, level, opt int) (*ICMPv6Filter, error)
```

## type IPMreq

```
type IPMreq struct {
 Multiaddr [4]byte /* in_addr */
 Interface [4]byte /* in_addr */
}
```

## func `GetsockoptIPMreq`

```
func GetsockoptIPMreq(fd, level, opt int) (*IPMreq, error)
```

## type `IPMreqn`

```
type IPMreqn struct {
 Multiaddr [4]byte /* in_addr */
 Address [4]byte /* in_addr */
 Ifindex int32
}
```

## func `GetsockoptIPMreqn`

```
func GetsockoptIPMreqn(fd, level, opt int) (*IPMreqn, error)
```

## type `IPv6MTUInfo`

```
type IPv6MTUInfo struct {
 Addr RawSockaddrInet6
 Mtu uint32
}
```

## func `GetsockoptIPv6MTUInfo`

```
func GetsockoptIPv6MTUInfo(fd, level, opt int) (*IPv6MTUInfo, error)
```

## type `IPv6Mreq`

```
type IPv6Mreq struct {
 Multiaddr [16]byte /* in6_addr */
 Interface uint32
}
```

## func `GetsockoptIPv6Mreq`

```
func GetsockoptIPv6Mreq(fd, level, opt int) (*IPv6Mreq, error)
```

## type `IfAddrmsg`

```
type IfAddrmsg struct {
 Family uint8
 Prefixlen uint8
 Flags uint8
 Scope uint8
}
```

```
 Index uint32
}
```

## type IfInfomsg

```
type IfInfomsg struct {
 Family uint8
 X_ifi_pad uint8
 Type uint16
 Index int32
 Flags uint32
 Change uint32
}
```

## type Inet4Pktinfo

```
type Inet4Pktinfo struct {
 Ifindex int32
 Spec_dst [4]byte /* in_addr */
 Addr [4]byte /* in_addr */
}
```

## type Inet6Pktinfo

```
type Inet6Pktinfo struct {
 Addr [16]byte /* in6_addr */
 Ifindex uint32
}
```

## type InotifyEvent

```
type InotifyEvent struct {
 Wd int32
 Mask uint32
 Cookie uint32
 Len uint32
 Name [0]uint8
}
```

## type Iovec

```
type Iovec struct {
 Base *byte
 Len uint64
}
```

## func (\*Iovec) SetLen

```
func (iov *Iovec) SetLen(length int)
```

## type Linger

```
type Linger struct {
 Onoff int32
 Linger int32
}
```

## type Msghdr

```
type Msghdr struct {
 Name *byte
 Namelen uint32
 Pad_cgo_0 [4]byte
 Iov *Iovec
 Iovlen uint64
 Control *byte
 Controllen uint64
 Flags int32
 Pad_cgo_1 [4]byte
}
```

## func (\*Msghdr) SetControllen

```
func (msghdr *Msghdr) SetControllen(length int)
```

## type NetlinkMessage

```
type NetlinkMessage struct {
 Header NlMsghdr
 Data []byte
}
```

NetlinkMessage represents a netlink message.

## func ParseNetlinkMessage

```
func ParseNetlinkMessage(b []byte) ([]NetlinkMessage, error)
```

ParseNetlinkMessage parses b as an array of netlink messages and returns the slice containing the NetlinkMessage structures.

## type NetlinkRouteAttr

```
type NetlinkRouteAttr struct {
 Attr RtAttr
 Value []byte
}
```

NetlinkRouteAttr represents a netlink route attribute.

## func **ParseNetlinkRouteAttr**

```
func ParseNetlinkRouteAttr(m *NetlinkMessage) ([]NetlinkRouteAttr, error)
```

ParseNetlinkRouteAttr parses m's payload as an array of netlink route attributes and returns the slice containing the NetlinkRouteAttr structures.

## type **NetlinkRouteRequest**

```
type NetlinkRouteRequest struct {
 Header NlMsgHdr
 Data RtGenmsg
}
```

NetlinkRouteRequest represents a request message to receive routing and link states from the kernel.

## type **NlAttr**

```
type NlAttr struct {
 Len uint16
 Type uint16
}
```

## type **NlMsgerr**

```
type NlMsgerr struct {
 Error int32
 Msg NlMsgHdr
}
```

## type **NlMsgHdr**

```
type NlMsgHdr struct {
 Len uint32
 Type uint16
 Flags uint16
 Seq uint32
 Pid uint32
}
```

## type ProcAttr

```
type ProcAttr struct {
 Dir string // Current working directory.
 Env []string // Environment.
 Files []uintptr // File descriptors.
 Sys *SysProcAttr
}
```

ProcAttr holds attributes that will be applied to a new process started by StartProcess.

## type PtraceRegs

```
type PtraceRegs struct {
 R15 uint64
 R14 uint64
 R13 uint64
 R12 uint64
 Rbp uint64
 Rbx uint64
 R11 uint64
 R10 uint64
 R9 uint64
 R8 uint64
 Rax uint64
 Rcx uint64
 Rdx uint64
 Rsi uint64
 Rdi uint64
 Orig_rax uint64
 Rip uint64
 Cs uint64
 Eflags uint64
 Rsp uint64
 Ss uint64
 Fs_base uint64
 Gs_base uint64
 Ds uint64
 Es uint64
 Fs uint64
 Gs uint64
}
```

## func (\*PtraceRegs) PC

```
func (r *PtraceRegs) PC() uint64
```

## func (\*PtraceRegs) SetPC

```
func (r *PtraceRegs) SetPC(pc uint64)
```

## type RawConn

```
type RawConn interface {
 // Control invokes f on the underlying connection's file
 // descriptor or handle.
 // The file descriptor fd is guaranteed to remain valid while
 // f executes but not after f returns.
 Control(f func(fd uintptr)) error

 // Read invokes f on the underlying connection's file
 // descriptor or handle; f is expected to try to read from the
 // file descriptor.
 // If f returns true, Read returns. Otherwise Read blocks
 // waiting for the connection to be ready for reading and
 // tries again repeatedly.
 // The file descriptor is guaranteed to remain valid while f
 // executes but not after f returns.
 Read(f func(fd uintptr) (done bool)) error

 // Write is like Read but for writing.
 Write(f func(fd uintptr) (done bool)) error
}
```

A RawConn is a raw network connection.

## type RawSockaddr

```
type RawSockaddr struct {
 Family uint16
 Data [14]int8
}
```

## type RawSockaddrAny

```
type RawSockaddrAny struct {
 Addr RawSockaddr
 Pad [96]int8
}
```

## type RawSockaddrInet4

```
type RawSockaddrInet4 struct {
 Family uint16
 Port uint16
 Addr [4]byte /* in_addr */
 Zero [8]uint8
}
```

## type RawSockaddrInet6

```
type RawSockaddrInet6 struct {
 Family uint16
 Port uint16
 Flowinfo uint32
 Addr [16]byte /* in6_addr */
 Scope_id uint32
}
```

## type RawSockaddrLinklayer

```
type RawSockaddrLinklayer struct {
 Family uint16
 Protocol uint16
 Ifindex int32
 Hatype uint16
 Pkttype uint8
 Halen uint8
 Addr [8]uint8
}
```

## type RawSockaddrNetlink

```
type RawSockaddrNetlink struct {
 Family uint16
 Pad uint16
 Pid uint32
 Groups uint32
}
```

## type RawSockaddrUnix

```
type RawSockaddrUnix struct {
 Family uint16
 Path [108]int8
}
```

## type Rlimit

```
type Rlimit struct {
 Cur uint64
 Max uint64
}
```

## type RtAttr

```
type RtAttr struct {
 Len uint16
```

```
Type uint16
}
```

## type RtGenmsg

```
type RtGenmsg struct {
 Family uint8
}
```

## type RtMsg

```
type RtMsg struct {
 Family uint8
 Dst_len uint8
 Src_len uint8
 Tos uint8
 Table uint8
 Protocol uint8
 Scope uint8
 Type uint8
 Flags uint32
}
```

## type RtNexthop

```
type RtNexthop struct {
 Len uint16
 Flags uint8
 Hops uint8
 Ifindex int32
}
```

## type Rusage

```
type Rusage struct {
 Utime Timeval
 Stime Timeval
 Maxrss int64
 Ixrss int64
 Idrss int64
 Isrss int64
 Minflt int64
 Majflt int64
 Nswap int64
 Inblock int64
 Oublock int64
 Msgsnd int64
 Msgrcv int64
 Nsignals int64
}
```

```
Nvcsw int64
Nivcsw int64
}
```

## type Signal

```
type Signal int
```

A Signal is a number describing a process signal. It implements the os.Signal interface.

## func (Signal) Signal

```
func (s Signal) Signal()
```

## func (Signal) String

```
func (s Signal) String() string
```

## type SockFilter

```
type SockFilter struct {
 Code uint16
 Jt uint8
 Jf uint8
 K uint32
}
```

## func LsfJump

```
func LsfJump(code, k, jt, jf int) *SockFilter
```

Deprecated: Use golang.org/x/net/bpf instead.

## func LsfStmt

```
func LsfStmt(code, k int) *SockFilter
```

Deprecated: Use golang.org/x/net/bpf instead.

## type SockFprog

```
type SockFprog struct {
 Len uint16
 Pad_cgo_0 [6]byte
 Filter *SockFilter
}
```

## type Sockaddr

```
type Sockaddr interface {
 // contains filtered or unexported methods
}
```

## func Accept

```
func Accept(fd int) (nfd int, sa Sockaddr, err error)
```

## func Accept4

```
func Accept4(fd int, flags int) (nfd int, sa Sockaddr, err error)
```

## func Getpeername

```
func Getpeername(fd int) (sa Sockaddr, err error)
```

## func Getsockname

```
func Getsockname(fd int) (sa Sockaddr, err error)
```

## func Recvfrom

```
func Recvfrom(fd int, p []byte, flags int) (n int, from Sockaddr, err error)
```

## func Recvmsg

```
func Recvmsg(fd int, p, oob []byte, flags int) (n, oobn int, recvflags int, from Sockaddr)
```

## type SockaddrInet4

```
type SockaddrInet4 struct {
 Port int
 Addr [4]byte
 // contains filtered or unexported fields
}
```

## type SockaddrInet6

```
type SockaddrInet6 struct {
 Port int
 ZoneId uint32
 Addr [16]byte
}
```

```
// contains filtered or unexported fields
}
```

## type SockaddrLinklayer

```
type SockaddrLinklayer struct {
 Protocol uint16
 Ifindex int
 Hatype uint16
 Pkttype uint8
 Halen uint8
 Addr [8]byte
 // contains filtered or unexported fields
}
```

## type SockaddrNetlink

```
type SockaddrNetlink struct {
 Family uint16
 Pad uint16
 Pid uint32
 Groups uint32
 // contains filtered or unexported fields
}
```

## type SockaddrUnix

```
type SockaddrUnix struct {
 Name string
 // contains filtered or unexported fields
}
```

## type SocketControlMessage

```
type SocketControlMessage struct {
 Header Cmsghdr
 Data []byte
}
```

SocketControlMessage represents a socket control message.

## func ParseSocketControlMessage

```
func ParseSocketControlMessage(b []byte) ([]SocketControlMessage, error)
```

ParseSocketControlMessage parses b as an array of socket control messages.

## type Stat\_t

```
type Stat_t struct {
 Dev uint64
 Ino uint64
 Nlink uint64
 Mode uint32
 Uid uint32
 Gid uint32
 X__pad0 int32
 Rdev uint64
 Size int64
 Blksize int64
 Blocks int64
 Atim Timespec
 Mtim Timespec
 Ctim Timespec
 X__unused [3]int64
}
```

## type Statfs\_t

```
type Statfs_t struct {
 Type int64
 Bsize int64
 Blocks uint64
 Bfree uint64
 Bavail uint64
 Files uint64
 Ffree uint64
 Fsid Fsid
 Namelen int64
 Frsize int64
 Flags int64
 Spare [4]int64
}
```

## type SysProcAttr

```
type SysProcAttr struct {
 Chroot string // Chroot.
 Credential *Credential // Credential.
 // Ptrace tells the child to call ptrace(PTRACE_TRACEME).
 // Call runtime.LockOSThread before starting a process with this set,
 // and don't call UnlockOSThread until done with PtraceSyscall calls.
 Ptrace bool
 Setsid bool // Create session.
 // Setpgid sets the process group ID of the child to Pgid,
 // or, if Pgid == 0, to the new child's process ID.
 Setpgid bool
 // Setctty sets the controlling terminal of the child to
```

```

// file descriptor Ctty. Ctty must be a descriptor number
// in the child process: an index into ProcAttr.Files.
// This is only meaningful if Setsid is true.
Setctty bool
Noctty bool // Detach fd 0 from controlling terminal
Ctty int // Controlling TTY fd
// Foreground places the child process group in the foreground.
// This implies Setpgid. The Ctty field must be set to
// the descriptor of the controlling TTY.
// Unlike Setctty, in this case Ctty must be a descriptor
// number in the parent process.
Foreground bool
Pgid int // Child's process group ID if Setpgid.
Pdeathsig Signal // Signal that the process will get when its parent d
Cloneflags uintptr // Flags for clone calls (Linux only)
Unshareflags uintptr // Flags for unshare calls (Linux only)
UidMappings []SysProcIDMap // User ID mappings for user namespaces.
GidMappings []SysProcIDMap // Group ID mappings for user namespaces.
// GidMappingsEnableSetgroups enabling setgroups syscall.
// If false, then setgroups syscall will be disabled for the child process.
// This parameter is no-op if GidMappings == nil. Otherwise for unprivileged
// users this should be set to false for mappings work.
GidMappingsEnableSetgroups bool
AmbientCaps []uintptr // Ambient capabilities (Linux only)
}

```

## type SysProcIDMap

```

type SysProcIDMap struct {
 ContainerID int // Container ID.
 HostID int // Host ID.
 Size int // Size.
}

```

SysProcIDMap holds Container ID to Host ID mappings used for User Namespaces in Linux. See user\_namespaces(7).

## type Sysinfo\_t

```

type Sysinfo_t struct {
 Uptime int64
 Loads [3]uint64
 Totalram uint64
 Freeram uint64
 Sharedram uint64
 Bufferram uint64
 Totalswap uint64
 Freeswap uint64
 Procs uint16
 Pad uint16
 Pad_cgo_0 [4]byte
}

```

```
Totalhigh uint64
Freehigh uint64
Unit uint32
X_f [0]byte
Pad_cgo_1 [4]byte
}
```

## type TCPInfo

```
type TCPInfo struct {
 State uint8
 Ca_state uint8
 Retransmits uint8
 Probes uint8
 Backoff uint8
 Options uint8
 Pad_cgo_0 [2]byte
 Rto uint32
 Ato uint32
 Snd_mss uint32
 Rcv_mss uint32
 Unacked uint32
 Sacked uint32
 Lost uint32
 Retrans uint32
 Fackets uint32
 Last_data_sent uint32
 Last_ack_sent uint32
 Last_data_recv uint32
 Last_ack_recv uint32
 Pmtu uint32
 Rcv_sssthresh uint32
 Rtt uint32
 Rttvar uint32
 Snd_sssthresh uint32
 Snd_cwnd uint32
 Advmss uint32
 Reordering uint32
 Rcv_rtt uint32
 Rcv_space uint32
 Total_retrans uint32
}
```

## type Termios

```
type Termios struct {
 Iflag uint32
 Oflag uint32
 Cflag uint32
 Lflag uint32
 Line uint8
 Cc [32]uint8
}
```

```
Pad_cgo_0 [3]byte
Ispeed uint32
Ospeed uint32
}
```

## type Time\_t

```
type Time_t int64
```

## func Time

```
func Time(t *Time_t) (tt Time_t, err error)
```

## type Timespec

```
type Timespec struct {
 Sec int64
 Nsec int64
}
```

## func NsecToTimespec

```
func NsecToTimespec(nsec int64) Timespec
```

NsecToTimespec takes a number of nanoseconds since the Unix epoch and returns the corresponding Timespec value.

## func (\*Timespec) Nano

```
func (ts *Timespec) Nano() int64
```

Nano returns ts as the number of nanoseconds elapsed since the Unix epoch.

## func (\*Timespec) Unix

```
func (ts *Timespec) Unix() (sec int64, nsec int64)
```

Unix returns ts as the number of seconds and nanoseconds elapsed since the Unix epoch.

## type Timeval

```
type Timeval struct {
 Sec int64
 Usec int64
}
```

## func [NsecToTimeval](#)

```
func NsecToTimeval(nsec int64) Timeval
```

NsecToTimeval takes a number of nanoseconds since the Unix epoch and returns the corresponding Timeval value.

## func [\(\\*Timeval\) Nano](#)

```
func (tv *Timeval) Nano() int64
```

Nano returns tv as the number of nanoseconds elapsed since the Unix epoch.

## func [\(\\*Timeval\) Unix](#)

```
func (tv *Timeval) Unix() (sec int64, nsec int64)
```

Unix returns tv as the number of seconds and nanoseconds elapsed since the Unix epoch.

## type [Timex](#)

```
type Timex struct {
 Modes uint32
 Pad_cgo_0 [4]byte
 Offset int64
 Freq int64
 Maxerror int64
 Esterror int64
 Status int32
 Pad_cgo_1 [4]byte
 Constant int64
 Precision int64
 Tolerance int64
 Time Timeval
 Tick int64
 Ppsfreq int64
 Jitter int64
 Shift int32
 Pad_cgo_2 [4]byte
 Stabil int64
 Jitcnt int64
 Calcnt int64
 Errcnt int64
 Stbcnt int64
 Tai int32
 Pad_cgo_3 [44]byte
}
```

## type [Tms](#)

```
type Tms struct {
 Utime int64
 Stime int64
 Cutime int64
 Cstime int64
}
```

## type Ucred

```
type Ucred struct {
 Pid int32
 Uid uint32
 Gid uint32
}
```

## func GetsockoptUcred

```
func GetsockoptUcred(fd, level, opt int) (*Ucred, error)
```

## func ParseUnixCredentials

```
func ParseUnixCredentials(m *SocketControlMessage) (*Ucred, error)
```

ParseUnixCredentials decodes a socket control message that contains credentials in a Ucred structure. To receive such a message, the SO\_PASSCRED option must be enabled on the socket.

## type Ustat\_t

```
type Ustat_t struct {
 Tfree int32
 Pad_cgo_0 [4]byte
 Tinode uint64
 Fname [6]int8
 Fpack [6]int8
 Pad_cgo_1 [4]byte
}
```

## type Utimbuf

```
type Utimbuf struct {
 Actime int64
 Modtime int64
}
```

## type Utsname

```
type Utsname struct {
 Sysname [65]int8
 Nodename [65]int8
 Release [65]int8
 Version [65]int8
 Machine [65]int8
 Domainname [65]int8
}
```

## type WaitStatus

```
type WaitStatus uint32
```

## func (WaitStatus) Continued

```
func (w WaitStatus) Continued() bool
```

## func (WaitStatus) CoreDump

```
func (w WaitStatus) CoreDump() bool
```

## func (WaitStatus) ExitStatus

```
func (w WaitStatus) ExitStatus() int
```

## func (WaitStatus) Exited

```
func (w WaitStatus) Exited() bool
```

## func (WaitStatus) Signal

```
func (w WaitStatus) Signal() Signal
```

## func (WaitStatus) Signaled

```
func (w WaitStatus) Signaled() bool
```

## func (WaitStatus) StopSignal

```
func (w WaitStatus) StopSignal() Signal
```

## func (WaitStatus) Stopped

```
func (w WaitStatus) Stopped() bool
```

## func (WaitStatus) TrapCause

```
func (w WaitStatus) TrapCause() int
```

# Package testing

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package testing provides support for automated testing of Go packages. It is intended to be used in concert with the "go test" command, which automates execution of any function of the form

```
func TestXxx(*testing.T)
```

where Xxx does not start with a lowercase letter. The function name serves to identify the test routine.

Within these functions, use the Error, Fail or related methods to signal failure.

To write a new test suite, create a file whose name ends \_test.go that contains the TestXxx functions as described here. Put the file in the same package as the one being tested. The file will be excluded from regular package builds but will be included when the "go test" command is run. For more detail, run "go help test" and "go help testflag".

A simple test function looks like this:

```
func TestAbs(t *testing.T) {
 got := Abs(-1)
 if got != 1 {
 t.Errorf("Abs(-1) = %d; want 1", got)
 }
}
```

## Benchmarks

Functions of the form

```
func BenchmarkXxx(*testing.B)
```

are considered benchmarks, and are executed by the "go test" command when its -bench flag is provided. Benchmarks are run sequentially.

For a description of the testing flags, see [https://golang.org/cmd/go/#hdr-Testing\\_flags](https://golang.org/cmd/go/#hdr-Testing_flags)

A sample benchmark function looks like this:

```
func BenchmarkRandInt(b *testing.B) {
 for i := 0; i < b.N; i++ {
```

```
 rand.Int()
}
}
```

The benchmark function must run the target code `b.N` times. During benchmark execution, `b.N` is adjusted until the benchmark function lasts long enough to be timed reliably. The output

|                    |          |            |
|--------------------|----------|------------|
| BenchmarkRandInt-8 | 68453040 | 17.8 ns/op |
|--------------------|----------|------------|

means that the loop ran 68453040 times at a speed of 17.8 ns per loop.

If a benchmark needs some expensive setup before running, the timer may be reset:

```
func BenchmarkBigLen(b *testing.B) {
 big := NewBig()
 b.ResetTimer()
 for i := 0; i < b.N; i++ {
 big.Len()
 }
}
```

If a benchmark needs to test performance in a parallel setting, it may use the `RunParallel` helper function; such benchmarks are intended to be used with the `go test -cpu` flag:

```
func BenchmarkTemplateParallel(b *testing.B) {
 templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
 b.RunParallel(func(pb *testing.PB) {
 var buf bytes.Buffer
 for pb.Next() {
 buf.Reset()
 templ.Execute(&buf, "World")
 }
 })
}
```

## Examples

The package also runs and verifies example code. Example functions may include a concluding line comment that begins with "Output:" and is compared with the standard output of the function when the tests are run. (The comparison ignores leading and trailing space.) These are examples of an example:

```
func ExampleHello() {
 fmt.Println("hello")
 // Output: hello
}

func ExampleSalutations() {
```

```
fmt.Println("hello, and")
fmt.Println("goodbye")
// Output:
// hello, and
// goodbye
}
```

The comment prefix "Unordered output:" is like "Output:", but matches any line order:

```
func ExamplePerm() {
 for _, value := range Perm(5) {
 fmt.Println(value)
 }
 // Unordered output: 4
 // 2
 // 1
 // 3
 // 0
}
```

Example functions without output comments are compiled but not executed.

The naming convention to declare examples for the package, a function F, a type T and method M on type T are:

```
func Example() { ... }
func ExampleF() { ... }
func ExampleT() { ... }
func ExampleT_M() { ... }
```

Multiple example functions for a package/type/function/method may be provided by appending a distinct suffix to the name. The suffix must start with a lower-case letter.

```
func Example_suffix() { ... }
func ExampleF_suffix() { ... }
func ExampleT_suffix() { ... }
func ExampleT_M_suffix() { ... }
```

The entire test file is presented as the example when it contains a single example function, at least one other function, type, variable, or constant declaration, and no test or benchmark functions.

## Skipping

Tests or benchmarks may be skipped at run time with a call to the Skip method of \*T or \*B:

```
func TestTimeConsuming(t *testing.T) {
 if testing.Short() {
 t.Skip("skipping test in short mode.")
 }
}
```

```
...
}
```

## Subtests and Sub-benchmarks

The `Run` methods of `T` and `B` allow defining subtests and sub-benchmarks, without having to define separate functions for each. This enables uses like table-driven benchmarks and creating hierarchical tests. It also provides a way to share common setup and tear-down code:

```
func TestFoo(t *testing.T) {
 // <setup code>
 t.Run("A=1", func(t *testing.T) { ... })
 t.Run("A=2", func(t *testing.T) { ... })
 t.Run("B=1", func(t *testing.T) { ... })
 // <tear-down code>
}
```

Each subtest and sub-benchmark has a unique name: the combination of the name of the top-level test and the sequence of names passed to `Run`, separated by slashes, with an optional trailing sequence number for disambiguation.

The argument to the `-run` and `-bench` command-line flags is an unanchored regular expression that matches the test's name. For tests with multiple slash-separated elements, such as subtests, the argument is itself slash-separated, with expressions matching each name element in turn. Because it is unanchored, an empty expression matches any string. For example, using "matching" to mean "whose name contains":

```
go test -run '' # Run all tests.
go test -run Foo # Run top-level tests matching "Foo", such as "TestFooBar".
go test -run Foo/A= # For top-level tests matching "Foo", run subtests matching "A=".
go test -run /A=1 # For all top-level tests, run subtests matching "A=1".
```

Subtests can also be used to control parallelism. A parent test will only complete once all of its subtests complete. In this example, all tests are run in parallel with each other, and only with each other, regardless of other top-level tests that may be defined:

```
func TestGroupedParallel(t *testing.T) {
 for _, tc := range tests {
 tc := tc // capture range variable
 t.Run(tc.Name, func(t *testing.T) {
 t.Parallel()
 ...
 })
 }
}
```

The race detector kills the program if it exceeds 8192 concurrent goroutines, so use care when running parallel tests with the `-race` flag set.

`Run` does not return until parallel subtests have completed, providing a way to clean up after a group of parallel tests:

```
func TestTeardownParallel(t *testing.T) {
 // This Run will not return until the parallel tests finish.
 t.Run("group", func(t *testing.T) {
 t.Run("Test1", parallelTest1)
 t.Run("Test2", parallelTest2)
 t.Run("Test3", parallelTest3)
 })
 // <tear-down code>
}
```

## Main

It is sometimes necessary for a test program to do extra setup or teardown before or after testing. It is also sometimes necessary for a test to control which code runs on the main thread. To support these and other cases, if a test file contains a function:

```
func TestMain(m *testing.M)
```

then the generated test will call `TestMain(m)` instead of running the tests directly. `TestMain` runs in the main goroutine and can do whatever setup and teardown is necessary around a call to `m.Run`. `m.Run` will return an exit code that may be passed to `os.Exit`. If `TestMain` returns, the test wrapper will pass the result of `m.Run` to `os.Exit` itself.

When `TestMain` is called, `flag.Parse` has not been run. If `TestMain` depends on command-line flags, including those of the testing package, it should call `flag.Parse` explicitly. Command line flags are always parsed by the time `test` or `benchmark` functions run.

A simple implementation of `TestMain` is:

```
func TestMain(m *testing.M) {
 // call flag.Parse() here if TestMain uses flags
 os.Exit(m.Run())
}
```

## func AllocsPerRun

```
func AllocsPerRun(runs int, f func()) (avg float64)
```

`AllocsPerRun` returns the average number of allocations during calls to `f`. Although the return value has type `float64`, it will always be an integral value.

To compute the number of allocations, the function will first be run once as a warm-up. The average number of allocations over the specified number of runs will then be measured and returned.

AllocsPerRun sets GOMAXPROCS to 1 during its measurement and will restore it before returning.

## func `CoverMode`

```
func CoverMode() string
```

CoverMode reports what the test coverage mode is set to. The values are "set", "count", or "atomic". The return value will be empty if test coverage is not enabled.

## func `Coverage`

```
func Coverage() float64
```

Coverage reports the current code coverage as a fraction in the range [0, 1]. If coverage is not enabled, Coverage returns 0.

When running a large set of sequential test cases, checking Coverage after each one can be useful for identifying which test cases exercise new code paths. It is not a replacement for the reports generated by 'go test -cover' and 'go tool cover'.

## func `Init`

```
func Init()
```

Init registers testing flags. These flags are automatically registered by the "go test" command before running test functions, so Init is only needed when calling functions such as Benchmark without using "go test".

Init has no effect if it was already called.

## func `Main`

```
func Main(matchString func(pat, str string) (bool, error), tests []InternalTest, benc
```

Main is an internal function, part of the implementation of the "go test" command. It was exported because it is cross-package and predates "internal" packages. It is no longer used by "go test" but preserved, as much as possible, for other systems that simulate "go test" using Main, but Main sometimes cannot be updated as new functionality is added to the testing package. Systems simulating "go test" should be updated to use MainStart.

## func `RegisterCover`

```
func RegisterCover(c Cover)
```

RegisterCover records the coverage data accumulators for the tests. NOTE: This function is internal to the testing infrastructure and may change. It is not covered (yet) by the Go 1 compatibility guidelines.

## func RunBenchmarks

```
func RunBenchmarks(matchString func(pat, str string) (bool, error), benchmarks []InternalTest)
```

RunBenchmarks is an internal function but exported because it is cross-package; it is part of the implementation of the "go test" command.

## func RunExamples

```
func RunExamples(matchString func(pat, str string) (bool, error), examples []InternalExample)
```

RunExamples is an internal function but exported because it is cross-package; it is part of the implementation of the "go test" command.

## func RunTests

```
func RunTests(matchString func(pat, str string) (bool, error), tests []InternalTest)
```

RunTests is an internal function but exported because it is cross-package; it is part of the implementation of the "go test" command.

## func Short

```
func Short() bool
```

Short reports whether the `-test.short` flag is set.

## func Verbose

```
func Verbose() bool
```

Verbose reports whether the `-test.v` flag is set.

## type B

```
type B struct {
 N int
 // contains filtered or unexported fields
}
```

B is a type passed to Benchmark functions to manage benchmark timing and to specify the number of iterations to run.

A benchmark ends when its Benchmark function returns or calls any of the methods FailNow, Fatal, Fatalf, SkipNow, Skip, or Skipf. Those methods must be called only from the goroutine running the Benchmark function. The other reporting methods, such as the variations of Log and Error, may be called simultaneously from multiple goroutines.

Like in tests, benchmark logs are accumulated during execution and dumped to standard output when done. Unlike in tests, benchmark logs are always printed, so as not to hide output whose existence may be affecting benchmark results.

## func (\*B) **Cleanup**

```
func (c *B) Cleanup(f func())
```

Cleanup registers a function to be called when the test and all its subtests complete. Cleanup functions will be called in last added, first called order.

## func (\*B) **Error**

```
func (c *B) Error(args ...interface{})
```

Error is equivalent to Log followed by Fail.

## func (\*B) **Errorf**

```
func (c *B) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf followed by Fail.

## func (\*B) **Fail**

```
func (c *B) Fail()
```

Fail marks the function as having failed but continues execution.

## func (\*B) **FailNow**

```
func (c *B) FailNow()
```

FailNow marks the function as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine). Execution will continue at the next test or benchmark. FailNow must be called from the goroutine running the test or benchmark function, not from other goroutines created during the test. Calling FailNow does not stop those other goroutines.

## func (\*B) **Failed**

```
func (c *B) Failed() bool
```

Failed reports whether the function has failed.

## func (\*B) Fatal

```
func (c *B) Fatal(args ...interface{})
```

Fatal is equivalent to Log followed by FailNow.

## func (\*B) Fatalf

```
func (c *B) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf followed by FailNow.

## func (\*B) Helper

```
func (c *B) Helper()
```

Helper marks the calling function as a test helper function. When printing file and line information, that function will be skipped. Helper may be called simultaneously from multiple goroutines.

## func (\*B) Log

```
func (c *B) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to `Println`, and records the text in the error log. For tests, the text will be printed only if the test fails or the `-test.v` flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the `-test.v` flag.

## func (\*B) Logf

```
func (c *B) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to `Printf`, and records the text in the error log. A final newline is added if not provided. For tests, the text will be printed only if the test fails or the `-test.v` flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the `-test.v` flag.

## func (\*B) Name

```
func (c *B) Name() string
```

Name returns the name of the running test or benchmark.

## func (\*B) ReportAllocs

```
func (b *B) ReportAllocs()
```

ReportAllocs enables malloc statistics for this benchmark. It is equivalent to setting `-test.benchmem`, but it only affects the benchmark function that calls ReportAllocs.

## func (\*B) ReportMetric

```
func (b *B) ReportMetric(n float64, unit string)
```

ReportMetric adds "n unit" to the reported benchmark results. If the metric is per-iteration, the caller should divide by `b.N`, and by convention units should end in `/op`. ReportMetric overrides any previously reported value for the same unit. ReportMetric panics if `unit` is the empty string or if `unit` contains any whitespace. If `unit` is a unit normally reported by the benchmark framework itself (such as `"allocs/op"`), ReportMetric will override that metric. Setting `"ns/op"` to 0 will suppress that built-in metric.

## func (\*B) ResetTimer

```
func (b *B) ResetTimer()
```

ResetTimer zeroes the elapsed benchmark time and memory allocation counters and deletes user-reported metrics. It does not affect whether the timer is running.

## func (\*B) Run

```
func (b *B) Run(name string, f func(b *B)) bool
```

Run benchmarks `f` as a subbenchmark with the given name. It reports whether there were any failures.

A subbenchmark is like any other benchmark. A benchmark that calls Run at least once will not be measured itself and will be called once with `N=1`.

## func (\*B) RunParallel

```
func (b *B) RunParallel(body func(*PB))
```

RunParallel runs a benchmark in parallel. It creates multiple goroutines and distributes `b.N` iterations among them. The number of goroutines defaults to `GOMAXPROCS`. To increase parallelism for non-CPU-bound benchmarks, call `SetParallelism` before `RunParallel`. `RunParallel` is usually used with the go test `-cpu` flag.

The body function will be run in each goroutine. It should set up any goroutine-local state and then iterate until `pb.Next` returns false. It should not use the `StartTimer`, `StopTimer`, or `ResetTimer` functions, because they have global effect. It should also not call `Run`.

## func (\*B) SetBytes

```
func (b *B) SetBytes(n int64)
```

SetBytes records the number of bytes processed in a single operation. If this is called, the benchmark will report ns/op and MB/s.

## func (\*B) SetParallelism

```
func (b *B) SetParallelism(p int)
```

SetParallelism sets the number of goroutines used by RunParallel to p\*GOMAXPROCS. There is usually no need to call SetParallelism for CPU-bound benchmarks. If p is less than 1, this call will have no effect.

## func (\*B) Skip

```
func (c *B) Skip(args ...interface{})
```

Skip is equivalent to Log followed by SkipNow.

## func (\*B) SkipNow

```
func (c *B) SkipNow()
```

SkipNow marks the test as having been skipped and stops its execution by calling runtime.Goexit. If a test fails (see Error, Errorf, Fail) and is then skipped, it is still considered to have failed. Execution will continue at the next test or benchmark. See also FailNow. SkipNow must be called from the goroutine running the test, not from other goroutines created during the test. Calling SkipNow does not stop those other goroutines.

## func (\*B) Skipf

```
func (c *B) Skipf(format string, args ...interface{})
```

Skipf is equivalent to Logf followed by SkipNow.

## func (\*B) Skipped

```
func (c *B) Skipped() bool
```

Skipped reports whether the test was skipped.

## func (\*B) StartTimer

```
func (b *B) StartTimer()
```

StartTimer starts timing a test. This function is called automatically before a benchmark starts, but it can also be used to resume timing after a call to StopTimer.

## func (\*B) StopTimer

```
func (b *B) StopTimer()
```

StopTimer stops timing a test. This can be used to pause the timer while performing complex initialization that you don't want to measure.

## func (\*B) TempDir

```
func (c *B) TempDir() string
```

TempDir returns a temporary directory for the test to use. The directory is automatically removed by Cleanup when the test and all its subtests complete. Each subsequent call to t.TempDir returns a unique directory; if the directory creation fails, TempDir terminates the test by calling Fatal.

## type BenchmarkResult

```
type BenchmarkResult struct {
 N int // The number of iterations.
 T time.Duration // The total time taken.
 Bytes int64 // Bytes processed in one iteration.
 MemAllocs uint64 // The total number of memory allocations.
 MemBytes uint64 // The total number of bytes allocated.

 // Extra records additional metrics reported by ReportMetric.
 Extra map[string]float64
}
```

BenchmarkResult contains the results of a benchmark run.

## func Benchmark

```
func Benchmark(f func(b *B)) BenchmarkResult
```

Benchmark benchmarks a single function. It is useful for creating custom benchmarks that do not use the "go test" command.

If f depends on testing flags, then Init must be used to register those flags before calling Benchmark and before calling flag.Parse.

If f calls Run, the result will be an estimate of running all its subbenchmarks that don't call Run in sequence in a single benchmark.

## func (BenchmarkResult) AllocatedBytesPerOp

```
func (r BenchmarkResult) AllocatedBytesPerOp() int64
```

AllocatedBytesPerOp returns the "B/op" metric, which is calculated as r.MemBytes / r.N.

## func (BenchmarkResult) AllocsPerOp

```
func (r BenchmarkResult) AllocsPerOp() int64
```

AllocsPerOp returns the "allocs/op" metric, which is calculated as r.MemAllocs / r.N.

## func (BenchmarkResult) MemString

```
func (r BenchmarkResult) MemString() string
```

MemString returns r.AllocatedBytesPerOp and r.AllocsPerOp in the same format as 'go test'.

## func (BenchmarkResult) NsPerOp

```
func (r BenchmarkResult) NsPerOp() int64
```

NsPerOp returns the "ns/op" metric.

## func (BenchmarkResult) String

```
func (r BenchmarkResult) String() string
```

String returns a summary of the benchmark results. It follows the benchmark result line format from <https://golang.org/design/14313-benchmark-format>, not including the benchmark name. Extra metrics override built-in metrics of the same name. String does not include allocs/op or B/op, since those are reported by MemString.

## type Cover

```
type Cover struct {
 Mode string
 Counters map[string][]uint32
 Blocks map[string][]CoverBlock
 CoveredPackages string
}
```

Cover records information about test coverage checking. NOTE: This struct is internal to the testing infrastructure and may change. It is not covered (yet) by the Go 1 compatibility guidelines.

## type CoverBlock

```
type CoverBlock struct {
 Line0 uint32 // Line number for block start.
 Col0 uint16 // Column number for block start.
 Line1 uint32 // Line number for block end.
 Col1 uint16 // Column number for block end.
 Stmts uint16 // Number of statements included in this block.
}
```

CoverBlock records the coverage data for a single basic block. The fields are 1-indexed, as in an editor: The opening line of the file is number 1, for example. Columns are measured in bytes. NOTE: This struct is internal to the testing infrastructure and may change. It is not covered (yet) by the Go 1 compatibility guidelines.

## type InternalBenchmark

```
type InternalBenchmark struct {
 Name string
 F func(b *B)
}
```

InternalBenchmark is an internal type but exported because it is cross-package; it is part of the implementation of the "go test" command.

## type InternalExample

```
type InternalExample struct {
 Name string
 F func()
 Output string
 Unordered bool
}
```

## type InternalTest

```
type InternalTest struct {
 Name string
 F func(*T)
}
```

InternalTest is an internal type but exported because it is cross-package; it is part of the implementation of the "go test" command.

## type M

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

M is a type passed to a TestMain function to run the actual tests.

## func MainStart

```
func MainStart(deps testDeps, tests []InternalTest, benchmarks []InternalBenchmark, e
```

MainStart is meant for use by tests generated by 'go test'. It is not meant to be called directly and is not subject to the Go 1 compatibility document. It may change signature from release to release.

## func (\*M) Run

```
func (m *M) Run() (code int)
```

Run runs the tests. It returns an exit code to pass to os.Exit.

## type PB

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

A PB is used by RunParallel for running parallel benchmarks.

## func (\*PB) Next

```
func (pb *PB) Next() bool
```

Next reports whether there are more iterations to execute.

## type T

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

T is a type passed to Test functions to manage test state and support formatted test logs.

A test ends when its Test function returns or calls any of the methods FailNow, Fatal, Fatalf, SkipNow, Skip, or Skipf. Those methods, as well as the Parallel method, must be called only from the goroutine running the Test function.

The other reporting methods, such as the variations of Log and Error, may be called simultaneously from multiple goroutines.

## func (\*T) Cleanup

```
func (c *T) Cleanup(f func())
```

Cleanup registers a function to be called when the test and all its subtests complete. Cleanup functions will be called in last added, first called order.

## func (\*T) Deadline

```
func (t *T) Deadline() (deadline time.Time, ok bool)
```

Deadline reports the time at which the test binary will have exceeded the timeout specified by the -timeout flag.

The ok result is false if the -timeout flag indicates “no timeout” (0).

## func (\*T) Error

```
func (c *T) Error(args ...interface{})
```

Error is equivalent to Log followed by Fail.

## func (\*T) Errorf

```
func (c *T) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf followed by Fail.

## func (\*T) Fail

```
func (c *T) Fail()
```

Fail marks the function as having failed but continues execution.

## func (\*T) FailNow

```
func (c *T) FailNow()
```

FailNow marks the function as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine). Execution will continue at the next test or benchmark. FailNow must be called from the goroutine running the test or benchmark function, not from other goroutines created during the test. Calling FailNow does not stop those other goroutines.

## func (\*T) Failed

```
func (c *T) Failed() bool
```

Failed reports whether the function has failed.

## func (\*T) Fatal

```
func (c *T) Fatal(args ...interface{})
```

Fatal is equivalent to Log followed by FailNow.

## func (\*T) Fatalf

```
func (c *T) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf followed by FailNow.

## func (\*T) Helper

```
func (c *T) Helper()
```

Helper marks the calling function as a test helper function. When printing file and line information, that function will be skipped. Helper may be called simultaneously from multiple goroutines.

## func (\*T) Log

```
func (c *T) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to `Println`, and records the text in the error log. For tests, the text will be printed only if the test fails or the `-test.v` flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the `-test.v` flag.

## func (\*T) Logf

```
func (c *T) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to `Printf`, and records the text in the error log. A final newline is added if not provided. For tests, the text will be printed only if the test fails or the `-test.v` flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the `-test.v` flag.

## func (\*T) Name

```
func (c *T) Name() string
```

Name returns the name of the running test or benchmark.

## func (\*T) Parallel

```
func (t *T) Parallel()
```

Parallel signals that this test is to be run in parallel with (and only with) other parallel tests. When a test is run multiple times due to use of `-test.count` or `-test.cpu`, multiple instances of a single test never run in parallel with each other.

## func (\*T) Run

```
func (t *T) Run(name string, f func(t *T)) bool
```

`Run` runs `f` as a subtest of `t` called `name`. It runs `f` in a separate goroutine and blocks until `f` returns or calls `t.Parallel` to become a parallel test. `Run` reports whether `f` succeeded (or at least did not fail before calling `t.Parallel`).

`Run` may be called simultaneously from multiple goroutines, but all such calls must return before the outer test function for `t` returns.

## func (\*T) Skip

```
func (c *T) Skip(args ...interface{})
```

`Skip` is equivalent to `Log` followed by `SkipNow`.

## func (\*T) SkipNow

```
func (c *T) SkipNow()
```

`SkipNow` marks the test as having been skipped and stops its execution by calling `runtime.Goexit`. If a test fails (see `Error`, `Errorf`, `Fail`) and is then skipped, it is still considered to have failed. Execution will continue at the next test or benchmark. See also `FailNow`. `SkipNow` must be called from the goroutine running the test, not from other goroutines created during the test. Calling `SkipNow` does not stop those other goroutines.

## func (\*T) Skipf

```
func (c *T) Skipf(format string, args ...interface{})
```

`Skipf` is equivalent to `Logf` followed by `SkipNow`.

## func (\*T) Skipped

```
func (c *T) Skipped() bool
```

`Skipped` reports whether the test was skipped.

## func (\*T) TempDir

```
func (c *T) TempDir() string
```

TempDir returns a temporary directory for the test to use. The directory is automatically removed by Cleanup when the test and all its subtests complete. Each subsequent call to t.TempDir returns a unique directory; if the directory creation fails, TempDir terminates the test by calling Fatal.

## type TB

```
type TB interface {
 Cleanup(func())
 Error(args ...interface{})
 Errorf(format string, args ...interface{})
 Fail()
 FailNow()
 Failed() bool
 Fatal(args ...interface{})
 Fatalf(format string, args ...interface{})
 Helper()
 Log(args ...interface{})
 Logf(format string, args ...interface{})
 Name() string
 Skip(args ...interface{})
 SkipNow()
 Skipf(format string, args ...interface{})
 Skipped() bool
 TempDir() string
 // contains filtered or unexported methods
}
```

TB is the interface common to T and B.

# Package template

 go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package template implements data-driven templates for generating textual output.

To generate HTML output, see package `html/template`, which has the same interface as this package but automatically secures HTML output against certain attacks.

Templates are executed by applying them to a data structure. Annotations in the template refer to elements of the data structure (typically a field of a struct or a key in a map) to control execution and derive values to be displayed. Execution of the template walks the structure and sets the cursor, represented by a period '.' and called "dot", to the value at the current location in the structure as execution proceeds.

The input text for a template is UTF-8-encoded text in any format. "Actions"--data evaluations or control structures--are delimited by "{{" and "}}"; all text outside actions is copied to the output unchanged. Except for raw strings, actions may not span newlines, although comments can.

Once parsed, a template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

Here is a trivial example that prints "17 items are made of wool".

```
type Inventory struct {
 Material string
 Count uint
}
sweaters := Inventory{"wool", 17}
tmpl, err := template.New("test").Parse("{{.Count}} items are made of {{.Material}}")
if err != nil { panic(err) }
err = tmpl.Execute(os.Stdout, sweaters)
if err != nil { panic(err) }
```

More intricate examples appear below.

## Text and spaces

By default, all text between actions is copied verbatim when the template is executed. For example, the string " items are made of " in the example above appears on standard output when the program is run.

However, to aid in formatting template source code, if an action's left delimiter (by default "{{") is followed immediately by a minus sign and ASCII space character ("{{- "), all trailing white space is trimmed from the immediately preceding text. Similarly, if the right delimiter ("}}") is preceded by a space and minus sign (" -}}"), all leading white space is trimmed from the immediately following text. In these trim markers, the ASCII space must be present; "{{-3}}" parses as an action containing the number -3.

For instance, when executing the template whose source is

```
"{{23 -}} < {{- 45}}"
```

the generated output would be

```
"23<45"
```

For this trimming, the definition of white space characters is the same as in Go: space, horizontal tab, carriage return, and newline.

## Actions

Here is the list of actions. "Arguments" and "pipelines" are evaluations of data, defined in detail in the corresponding sections that follow.

```
{{{/* a comment */}}}
{{{- /* a comment with white space trimmed from preceding and following text */ -}}
 A comment; discarded. May contain newlines.
 Comments do not nest and must start and end at the
 delimiters, as shown here.

{{pipeline}}
 The default textual representation (the same as would be
 printed by fmt.Print) of the value of the pipeline is copied
 to the output.

{{if pipeline}} T1 {{end}}
 If the value of the pipeline is empty, no output is generated;
 otherwise, T1 is executed. The empty values are false, 0, any
 nil pointer or interface value, and any array, slice, map, or
 string of length zero.
 Dot is unaffected.

{{if pipeline}} T1 {{else}} T0 {{end}}
 If the value of the pipeline is empty, T0 is executed;
 otherwise, T1 is executed. Dot is unaffected.

{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
 To simplify the appearance of if-else chains, the else action
 of an if may include another if directly; the effect is exactly
 the same as writing
```

```
 {{if pipeline}} T1 {{else}}{{if pipeline}} T0 {{end}}{{end}}
```

```
 {{range pipeline}} T1 {{end}}
```

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, nothing is output; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed. If the value is a map and the keys are of basic type with a defined order, the elements will be visited in sorted key order.

```
 {{range pipeline}} T1 {{else}} T0 {{end}}
```

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, dot is unaffected and T0 is executed; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed.

```
 {{template "name"}}
```

The template with the specified name is executed with nil data.

```
 {{template "name" pipeline}}
```

The template with the specified name is executed with dot set to the value of the pipeline.

```
 {{block "name" pipeline}} T1 {{end}}
```

A block is shorthand for defining a template

```
 {{define "name"}} T1 {{end}}
```

and then executing it in place

```
 {{template "name" pipeline}}
```

The typical use is to define a set of root templates that are then customized by redefining the block templates within.

```
 {{with pipeline}} T1 {{end}}
```

If the value of the pipeline is empty, no output is generated; otherwise, dot is set to the value of the pipeline and T1 is executed.

```
 {{with pipeline}} T1 {{else}} T0 {{end}}
```

If the value of the pipeline is empty, dot is unaffected and T0 is executed; otherwise, dot is set to the value of the pipeline and T1 is executed.

## Arguments

An argument is a simple value, denoted by one of the following.

- A boolean, string, character, integer, floating-point, imaginary or complex constant in Go syntax. These behave like Go's untyped constants. Note that, as in Go, whether a large integer constant overflows when assigned or passed to a function can depend on whether the host machine's ints are 32 or 64 bits.
- The keyword `nil`, representing an untyped Go `nil`.
- The character `'.'` (period):

The result is the value of dot.

- A variable name, which is a (possibly empty) alphanumeric string preceded by a dollar sign, such as

`$piOver2`

or

`$`

The result is the value of the variable.

Variables are described below.

- The name of a field of the data, which must be a struct, preceded by a period, such as

`.Field`

The result is the value of the field. Field invocations may be chained:

`.Field1.Field2`

Fields can also be evaluated on variables, including chaining:

`$x.Field1.Field2`

- The name of a key of the data, which must be a map, preceded by a period, such as

`.Key`

The result is the map element value indexed by the key.

Key invocations may be chained and combined with fields to any depth:

`.Field1.Key1.Field2.Key2`

Although the key must be an alphanumeric identifier, unlike with field names they do not need to start with an upper case letter.

Keys can also be evaluated on variables, including chaining:

`$x.key1.key2`

- The name of a niladic method of the data, preceded by a period, such as

`.Method`

The result is the value of invoking the method with dot as the receiver, `dot.Method()`. Such a method must have one return value (of any type) or two return values, the second of which is an error.

If it has two and the returned error is non-nil, execution terminates and an error is returned to the caller as the value of `Execute`.

Method invocations may be chained and combined with fields and keys to any depth:

`.Field1.Key1.Method1.Field2.Key2.Method2`

Methods can also be evaluated on variables, including chaining:

`$x.Method1.Field`

- The name of a niladic function, such as

`fun`

The result is the value of invoking the function, `fun()`. The return types and values behave as in methods. Functions and function names are described below.

- A parenthesized instance of one the above, for grouping. The result may be accessed by a field or map key invocation.

`print (.F1 arg1) (.F2 arg2)`

`(.StructValuedMethod "arg").Field`

Arguments may evaluate to any type; if they are pointers the implementation automatically indirections to the base type when required. If an evaluation yields a function value, such as a function-valued field of

a struct, the function is not invoked automatically, but it can be used as a truth value for an if action and the like. To invoke it, use the call function, defined below.

## Pipelines

A pipeline is a possibly chained sequence of "commands". A command is a simple value (argument) or a function or method call, possibly with multiple arguments:

### Argument

The result is the value of evaluating the argument.

### .Method [Argument...]

The method can be alone or the last element of a chain but, unlike methods in the middle of a chain, it can take arguments.

The result is the value of calling the method with the arguments:

dot.Method(Argument1, etc.)

### functionName [Argument...]

The result is the value of calling the function associated with the name:

function(Argument1, etc.)

Functions and function names are described below.

A pipeline may be "chained" by separating a sequence of commands with pipeline characters '|'. In a chained pipeline, the result of each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline.

The output of a command will be either one value or two values, the second of which has type error. If that second value is present and evaluates to non-nil, execution terminates and the error is returned to the caller of Execute.

## Variables

A pipeline inside an action may initialize a variable to capture the result. The initialization has syntax

```
$variable := pipeline
```

where \$variable is the name of the variable. An action that declares a variable produces no output.

Variables previously declared can also be assigned, using the syntax

```
$variable = pipeline
```

If a "range" action initializes a variable, the variable is set to the successive elements of the iteration. Also, a "range" may declare two variables, separated by a comma:

```
range $index, $element := pipeline
```

in which case `$index` and `$element` are set to the successive values of the array/slice index or map key and element, respectively. Note that if there is only one variable, it is assigned the element; this is opposite to the convention in Go range clauses.

A variable's scope extends to the "end" action of the control structure ("if", "with", or "range") in which it is declared, or to the end of the template if there is no such control structure. A template invocation does not inherit variables from the point of its invocation.

When execution begins, `$` is set to the data argument passed to `Execute`, that is, to the starting value of `dot`.

## Examples

Here are some example one-line templates demonstrating pipelines and variables. All produce the quoted word "output":

```
{{"\"output\""}}
 A string constant.
{{`output`}}
 A raw string constant.
{{printf "%q" "output"}}
 A function call.
{{"output" | printf "%q"}}
 A function call whose final argument comes from the previous
 command.
{{printf "%q" (print "out" "put")}}
 A parenthesized argument.
{{"put" | printf "%s%s" "out" | printf "%q"}}
 A more elaborate call.
{{"output" | printf "%s" | printf "%q"}}
 A longer chain.
{{with "output"}{{printf "%q" .}}}{end}
 A with action using dot.
{{with $x := "output" | printf "%q"}{{$x}}}{end}
 A with action that creates and uses a variable.
{{with $x := "output"}{{printf "%q" $x}}}{end}
 A with action that uses the variable in another action.
{{with $x := "output"}{{$x | printf "%q"}}}{end}
 The same, but pipelined.
```

## Functions

During execution functions are found in two function maps: first in the template, then in the global function map. By default, no functions are defined in the template but the `Funcs` method can be used to add them.

Predefined global functions are named as follows.

and

Returns the boolean AND of its arguments by returning the first empty argument or the last argument, that is, "and x y" behaves as "if x then y else x". All the arguments are evaluated.

call

Returns the result of calling the first argument, which must be a function, with the remaining arguments as parameters.

Thus "call .X.Y 1 2" is, in Go notation, dot.X.Y(1, 2) where Y is a func-valued field, map entry, or the like.

The first argument must be the result of an evaluation that yields a value of function type (as distinct from a predefined function such as print). The function must return either one or two result values, the second of which is of type error. If the arguments don't match the function or the returned error value is non-nil, execution stops.

html

Returns the escaped HTML equivalent of the textual representation of its arguments. This function is unavailable in html/template, with a few exceptions.

index

Returns the result of indexing its first argument by the following arguments. Thus "index x 1 2 3" is, in Go syntax, x[1][2][3]. Each indexed item must be a map, slice, or array.

slice

slice returns the result of slicing its first argument by the remaining arguments. Thus "slice x 1 2" is, in Go syntax, x[1:2], while "slice x" is x[:], "slice x 1" is x[1:], and "slice x 1 2 3" is x[1:2:3]. The first argument must be a string, slice, or array.

js

Returns the escaped JavaScript equivalent of the textual representation of its arguments.

len

Returns the integer length of its argument.

not

Returns the boolean negation of its single argument.

or

Returns the boolean OR of its arguments by returning the first non-empty argument or the last argument, that is, "or x y" behaves as "if x then x else y". All the arguments are evaluated.

print

An alias for fmt.Sprint

printf

An alias for fmt.Sprintf

println

An alias for fmt.Sprintln

urlquery

Returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query. This function is unavailable in html/template, with a few exceptions.

The boolean functions take any zero value to be false and a non-zero value to be true.

There is also a set of binary comparison operators defined as functions:

```
eq Returns the boolean truth of arg1 == arg2
ne Returns the boolean truth of arg1 != arg2
lt Returns the boolean truth of arg1 < arg2
le Returns the boolean truth of arg1 <= arg2
gt Returns the boolean truth of arg1 > arg2
ge Returns the boolean truth of arg1 >= arg2
```

For simpler multi-way equality tests, eq (only) accepts two or more arguments and compares the second and subsequent to the first, returning in effect

```
arg1==arg2 || arg1==arg3 || arg1==arg4 ...
```

(Unlike with || in Go, however, eq is a function call and all the arguments will be evaluated.)

The comparison functions work on any values whose type Go defines as comparable. For basic types such as integers, the rules are relaxed: size and exact type are ignored, so any integer value, signed or unsigned, may be compared with any other integer value. (The arithmetic value is compared, not the bit pattern, so all negative integers are less than all unsigned integers.) However, as usual, one may not compare an int with a float32 and so on.

## Associated templates

Each template is named by a string specified when it is created. Also, each template is associated with zero or more other templates that it may invoke by name; such associations are transitive and form a name space of templates.

A template may use a template invocation to instantiate another associated template; see the explanation of the "template" action above. The name must be that of a template associated with the template that contains the invocation.

## Nested template definitions

When parsing a template, another template may be defined and associated with the template being parsed. Template definitions must appear at the top level of the template, much like global variables in a Go program.

The syntax of such definitions is to surround each template declaration with a "define" and "end" action.

The define action names the template being created by providing a string constant. Here is a simple example:

```
`{{define "T1"}}ONE{{end}}
{{define "T2"}}TWO{{end}}
{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}
{{template "T3"}}`
```

This defines two templates, T1 and T2, and a third T3 that invokes the other two when it is executed. Finally it invokes T3. If executed this template will produce the text

```
ONE TWO
```

By construction, a template may reside in only one association. If it's necessary to have a template addressable from multiple associations, the template definition must be parsed multiple times to create distinct \*Template values, or must be copied with the Clone or AddParseTree method.

Parse may be called multiple times to assemble the various associated templates; see the ParseFiles and ParseGlob functions and methods for simple ways to parse related templates stored in files.

A template may be executed directly or through ExecuteTemplate, which executes an associated template identified by name. To invoke our example above, we might write,

```
err := tmpl.Execute(os.Stdout, "no data needed")
if err != nil {
 log.Fatalf("execution failed: %s", err)
}
```

or to invoke a particular template explicitly by name,

```
err := tmpl.ExecuteTemplate(os.Stdout, "T2", "no data needed")
if err != nil {
 log.Fatalf("execution failed: %s", err)
}
```

## func **HTMLEscape**

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

## func **HTMLEscapeString**

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

## func HTMLEscaper

```
func HTMLEscaper(args ...interface{}) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

## func IsTrue

```
func IsTrue(val interface{}) (truth, ok bool)
```

IsTrue reports whether the value is 'true', in the sense of not the zero of its type, and whether the value has a meaningful truth value. This is the definition of truth used by if and other such actions.

## func JSEscape

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

## func JSEscapeString

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

## func JSEscaper

```
func JSEscaper(args ...interface{}) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

## func URLQueryEscaper

```
func URLQueryEscaper(args ...interface{}) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

## type ExecError

```
type ExecError struct {
 Name string // Name of template.
```

```
 Err error // Pre-formatted error.
}
```

ExecError is the custom error type returned when Execute has an error evaluating its template. (If a write error occurs, the actual error is returned; it will not be of type ExecError.)

## func (ExecError) Error

```
func (e ExecError) Error() string
```

## func (ExecError) Unwrap

```
func (e ExecError) Unwrap() error
```

## type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) return value evaluates to non-nil during execution, execution terminates and Execute returns that error.

When template execution invokes a function with an argument list, that list must be assignable to the function's parameter types. Functions meant to apply to arguments of arbitrary type can use parameters of type interface{} or of type reflect.Value. Similarly, functions meant to return a result of arbitrary type can return interface{} or reflect.Value.

## type Template

```
type Template struct {
 *parse.Tree
 // contains filtered or unexported fields
}
```

Template is the representation of a parsed template. The \*parse.Tree field is exported only for use by html/template and should be treated as unexported by all other clients.

## func Must

```
func Must(t *Template, err error) *Template
```

Must is a helper that wraps a call to a function returning (\*Template, error) and panics if the error is non-nil. It is intended for use in variable initializations such as

```
var t = template.Must(template.New("name").Parse("text"))
```

## func New

```
func New(name string) *Template
```

New allocates a new, undefined template with the given name.

## func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the base name and parsed contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned \*Template is nil.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results. For instance, ParseFiles("a/foo", "b/foo") stores "b/foo" as the template named "foo", while "a/foo" is unavailable.

## func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

## func (\*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
```

AddParseTree associates the argument parse tree with the template t, giving it the specified name. If the template has not been defined, this tree becomes its definition. If it has been defined and already has that name, the existing definition is replaced; otherwise a new template is created, defined, and returned.

## func (\*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

## func (\*Template) **DefinedTemplates**

```
func (t *Template) DefinedTemplates() string
```

DefinedTemplates returns a string listing the defined templates, prefixed by the string "; defined templates are: ". If there are none, it returns the empty string. For generating an error message here and in html/template.

## func (\*Template) **Delims**

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

## func (\*Template) **Execute**

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

Execute applies a parsed template to the specified data object, and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

If data is a reflect.Value, the template applies to the concrete value that the reflect.Value holds, as in fmt.Println.

## func (\*Template) **ExecuteTemplate**

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

## func (\*Template) **Funcs**

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs adds the elements of the argument map to the template's function map. It must be called before the template is parsed. It panics if a value in the map is not a function with appropriate return type or if the name cannot be used syntactically as a function in a template. It is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

## func (\*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

Lookup returns the template with the given name that is associated with t. It returns nil if there is no such template or the template has no definition.

## func (\*Template) Name

```
func (t *Template) Name() string
```

Name returns the name of the template.

## func (\*Template) New

```
func (t *Template) New(name string) *Template
```

New allocates a new, undefined template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a {{template}} action.

Because associated templates share underlying data, template construction cannot be done safely in parallel. Once the templates are constructed, they can be executed in parallel.

## func (\*Template) Option

```
func (t *Template) Option(opt ...string) *Template
```

Option sets options for the template. Options are described by strings, either a simple string or "key=value". There can be at most one equals sign in an option string. If the option string is unrecognized or otherwise invalid, Option panics.

Known options:

missingkey: Control the behavior during execution if a map is indexed with a key that is not present in the map.

"missingkey=default" or "missingkey=invalid"

The default behavior: Do nothing and continue execution.

If printed, the result of the index operation is the string

```
"<no value>".
"missingkey=zero"
 The operation returns the zero value for the map type's element.
"missingkey=error"
 Execution stops immediately with an error.
```

## func (\*Template) Parse

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse parses text as a template body for t. Named template definitions ({{define ...}} or {{block ...}} statements) in text define additional templates associated with t and are removed from the definition of t itself.

Templates can be redefined in successive calls to Parse. A template definition with a body containing only white space and comments is considered empty and will not replace an existing template's body. This allows using Parse to add new named template definitions without overwriting the main template body.

## func (\*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file. Since the templates created by ParseFiles are named by the base names of the argument files, t should usually have the name of one of the (base) names of the files. If it does not, depending on t's contents before calling ParseFiles, t.Execute may fail. In that case use t.ExecuteTemplate to execute a valid template.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

## func (\*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseGlob parses the template definitions in the files identified by the pattern and associates the resulting templates with t. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. ParseGlob is equivalent to calling t.ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

## func (\*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates returns a slice of defined templates associated with t.

## Overview

Package time provides functionality for measuring and displaying time.

The calendrical calculations always assume a Gregorian calendar, with no leap seconds.

## Monotonic Clocks

Operating systems provide both a “wall clock,” which is subject to changes for clock synchronization, and a “monotonic clock,” which is not. The general rule is that the wall clock is for telling time and the monotonic clock is for measuring time. Rather than split the API, in this package the Time returned by `time.Now` contains both a wall clock reading and a monotonic clock reading; later time-telling operations use the wall clock reading, but later time-measuring operations, specifically comparisons and subtractions, use the monotonic clock reading.

For example, this code always computes a positive elapsed time of approximately 20 milliseconds, even if the wall clock is changed during the operation being timed:

```
start := time.Now()
... operation that takes 20 milliseconds ...
t := time.Now()
elapsed := t.Sub(start)
```

Other idioms, such as `time.Since(start)`, `time.Until(deadline)`, and `time.Now().Before(deadline)`, are similarly robust against wall clock resets.

The rest of this section gives the precise details of how operations use monotonic clocks, but understanding those details is not required to use this package.

The Time returned by `time.Now` contains a monotonic clock reading. If Time `t` has a monotonic clock reading, `t.Add` adds the same duration to both the wall clock and monotonic clock readings to compute the result. Because `t.AddDate(y, m, d)`, `t.Round(d)`, and `t.Truncate(d)` are wall time computations, they always strip any monotonic clock reading from their results. Because `t.In`, `t.Local`, and `t.UTC` are used for their effect on the interpretation of the wall time, they also strip any monotonic clock reading from their results. The canonical way to strip a monotonic clock reading is to use `t = t.Round(0)`.

If Times `t` and `u` both contain monotonic clock readings, the operations `t.After(u)`, `t.Before(u)`, `t.Equal(u)`, and `t.Sub(u)` are carried out using the monotonic clock readings alone, ignoring the wall

clock readings. If either `t` or `u` contains no monotonic clock reading, these operations fall back to using the wall clock readings.

On some systems the monotonic clock will stop if the computer goes to sleep. On such a system, `t.Sub(u)` may not accurately reflect the actual time that passed between `t` and `u`.

Because the monotonic clock reading has no meaning outside the current process, the serialized forms generated by `t.GobEncode`, `t.MarshalBinary`, `t.MarshalJSON`, and `t.MarshalText` omit the monotonic clock reading, and `t.Format` provides no format for it. Similarly, the constructors `time.Date`, `time.Parse`, `time.ParseInLocation`, and `time.Unix`, as well as the unmarshalers `t.GobDecode`, `t.UnmarshalBinary`, `t.UnmarshalJSON`, and `t.UnmarshalText` always create times with no monotonic clock reading.

Note that the `Go ==` operator compares not just the time instant but also the `Location` and the monotonic clock reading. See the documentation for the `Time` type for a discussion of equality testing for `Time` values.

For debugging, the result of `t.String` does include the monotonic clock reading if present. If `t != u` because of different monotonic clock readings, that difference will be visible when printing `t.String()` and `u.String()`.

## Constants

```
const (
 ANSIC = "Mon Jan _2 15:04:05 2006"
 UnixDate = "Mon Jan _2 15:04:05 MST 2006"
 RubyDate = "Mon Jan 02 15:04:05 -0700 2006"
 RFC822 = "02 Jan 06 15:04 MST"
 RFC822Z = "02 Jan 06 15:04 -0700" // RFC822 with numeric zone
 RFC850 = "Monday, 02-Jan-06 15:04:05 MST"
 RFC1123 = "Mon, 02 Jan 2006 15:04:05 MST"
 RFC1123Z = "Mon, 02 Jan 2006 15:04:05 -0700" // RFC1123 with numeric zone
 RFC3339 = "2006-01-02T15:04:05Z07:00"
 RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
 Kitchen = "3:04PM"
 // Handy time stamps.
 Stamp = "Jan _2 15:04:05"
 StampMilli = "Jan _2 15:04:05.000"
 StampMicro = "Jan _2 15:04:05.000000"
 StampNano = "Jan _2 15:04:05.000000000"
)
```

These are predefined layouts for use in `Time.Format` and `time.Parse`. The reference time used in the layouts is the specific time:

```
Mon Jan 2 15:04:05 MST 2006
```

which is Unix time 1136239445. Since MST is GMT-0700, the reference time can be thought of as

To define your own format, write down what the reference time would look like formatted your way; see the values of constants like ANSIC, StampMicro or Kitchen for examples. The model is to demonstrate what the reference time looks like so that the Format and Parse methods can apply the same transformation to a general time value.

Some valid layouts are invalid time values for time.Parse, due to formats such as \_ for space padding and Z for zone information.

Within the format string, an underscore \_ represents a space that may be replaced by a digit if the following number (a day) has two digits; for compatibility with fixed-width Unix time formats.

A decimal point followed by one or more zeros represents a fractional second, printed to the given number of decimal places. A decimal point followed by one or more nines represents a fractional second, printed to the given number of decimal places, with trailing zeros removed. When parsing (only), the input may contain a fractional second field immediately after the seconds field, even if the layout does not signify its presence. In that case a decimal point followed by a maximal series of digits is parsed as a fractional second.

Numeric time zone offsets format as follows:

```
-0700 ±hhmm
-07:00 ±hh:mm
-07 ±hh
```

Replacing the sign in the format with a Z triggers the ISO 8601 behavior of printing Z instead of an offset for the UTC zone. Thus:

```
Z0700 Z or ±hhmm
Z07:00 Z or ±hh:mm
Z07 Z or ±hh
```

The recognized day of week formats are "Mon" and "Monday". The recognized month formats are "Jan" and "January".

The formats 2, \_2, and 02 are unpadded, space-padded, and zero-padded day of month. The formats \_\_2 and 002 are space-padded and zero-padded three-character day of year; there is no unpadded day of year format.

Text in the format string that is not recognized as part of the reference time is echoed verbatim during Format and expected to appear verbatim in the input to Parse.

The executable example for Time.Format demonstrates the working of the layout string in detail and is a good reference.

Note that the RFC822, RFC850, and RFC1123 formats should be applied only to local times. Applying them to UTC times will use "UTC" as the time zone abbreviation, while strictly speaking those RFCs require the use of "GMT" in that case. In general RFC1123Z should be used instead of RFC1123 for servers that insist on that format, and RFC3339 should be preferred for new protocols. RFC3339, RFC822, RFC822Z, RFC1123, and RFC1123Z are useful for formatting; when used with time.Parse they do not accept all the time formats permitted by the RFCs and they do accept time formats not formally defined. The RFC3339Nano format removes trailing zeros from the seconds field and thus may not sort correctly once formatted.

```
const (
 Nanosecond Duration = 1
 Microsecond = 1000 * Nanosecond
 Millisecond = 1000 * Microsecond
 Second = 1000 * Millisecond
 Minute = 60 * Second
 Hour = 60 * Minute
)
```

Common durations. There is no definition for units of Day or larger to avoid confusion across daylight savings time zone transitions.

To count the number of units in a Duration, divide:

```
second := time.Second
fmt.Println(int64(second/time.Millisecond)) // prints 1000
```

To convert an integer number of units to a Duration, multiply:

```
seconds := 10
fmt.Println(time.Duration(seconds)*time.Second) // prints 10s
```

## func After

```
func After(d Duration) <-chan Time
```

After waits for the duration to elapse and then sends the current time on the returned channel. It is equivalent to NewTimer(d).C. The underlying Timer is not recovered by the garbage collector until the timer fires. If efficiency is a concern, use NewTimer instead and call Timer.Stop if the timer is no longer needed.

## func Sleep

```
func Sleep(d Duration)
```

Sleep pauses the current goroutine for at least the duration d. A negative or zero duration causes Sleep to return immediately.

## func Tick

```
func Tick(d Duration) <-chan Time
```

Tick is a convenience wrapper for NewTicker providing access to the ticking channel only. While Tick is useful for clients that have no need to shut down the Ticker, be aware that without a way to shut it down the underlying Ticker cannot be recovered by the garbage collector; it "leaks". Unlike NewTicker, Tick will return nil if d <= 0.

## type Duration

```
type Duration int64
```

A Duration represents the elapsed time between two instants as an int64 nanosecond count. The representation limits the largest representable duration to approximately 290 years.

## func ParseDuration

```
func ParseDuration(s string) (Duration, error)
```

ParseDuration parses a duration string. A duration string is a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as "300ms", "-1.5h" or "2h45m". Valid time units are "ns", "us" (or "μs"), "ms", "s", "m", "h".

## func Since

```
func Since(t Time) Duration
```

Since returns the time elapsed since t. It is shorthand for time.Now().Sub(t).

## func Until

```
func Until(t Time) Duration
```

Until returns the duration until t. It is shorthand for t.Sub(time.Now()).

## func (Duration) Hours

```
func (d Duration) Hours() float64
```

Hours returns the duration as a floating point number of hours.

## func (Duration) Microseconds

```
func (d Duration) Microseconds() int64
```

Microseconds returns the duration as an integer microsecond count.

## func (Duration) **Milliseconds**

```
func (d Duration) Milliseconds() int64
```

Milliseconds returns the duration as an integer millisecond count.

## func (Duration) **Minutes**

```
func (d Duration) Minutes() float64
```

Minutes returns the duration as a floating point number of minutes.

## func (Duration) **Nanoseconds**

```
func (d Duration) Nanoseconds() int64
```

Nanoseconds returns the duration as an integer nanosecond count.

## func (Duration) **Round**

```
func (d Duration) Round(m Duration) Duration
```

Round returns the result of rounding d to the nearest multiple of m. The rounding behavior for halfway values is to round away from zero. If the result exceeds the maximum (or minimum) value that can be stored in a Duration, Round returns the maximum (or minimum) duration. If m <= 0, Round returns d unchanged.

## func (Duration) **Seconds**

```
func (d Duration) Seconds() float64
```

Seconds returns the duration as a floating point number of seconds.

## func (Duration) **String**

```
func (d Duration) String() string
```

String returns a string representing the duration in the form "72h3m0.5s". Leading zero units are omitted. As a special case, durations less than one second format use a smaller unit (milli-, micro-, or nanoseconds) to ensure that the leading digit is non-zero. The zero duration formats as 0s.

## func (Duration) **Truncate**

```
func (d Duration) Truncate(m Duration) Duration
```

Truncate returns the result of rounding d toward zero to a multiple of m. If m <= 0, Truncate returns d unchanged.

## type Location

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

A Location maps time instants to the zone in use at that time. Typically, the Location represents the collection of time offsets in use in a geographical area. For many Locations the time offset varies depending on whether daylight savings time is in use at the time instant.

```
var Local *Location = &localLoc
```

Local represents the system's local time zone. On Unix systems, Local consults the TZ environment variable to find the time zone to use. No TZ means use the system default /etc/localtime. TZ="" means use UTC. TZ="foo" means use file foo in the system timezone directory.

```
var UTC *Location = &utcLoc
```

UTC represents Universal Coordinated Time (UTC).

## func FixedZone

```
func FixedZone(name string, offset int) *Location
```

FixedZone returns a Location that always uses the given zone name and offset (seconds east of UTC).

## func LoadLocation

```
func LoadLocation(name string) (*Location, error)
```

LoadLocation returns the Location with the given name.

If the name is "" or "UTC", LoadLocation returns UTC. If the name is "Local", LoadLocation returns Local.

Otherwise, the name is taken to be a location name corresponding to a file in the IANA Time Zone database, such as "America/New\_York".

The time zone database needed by LoadLocation may not be present on all systems, especially non-Unix systems. LoadLocation looks in the directory or uncompressed zip file named by the ZONEINFO

environment variable, if any, then looks in known installation locations on Unix systems, and finally looks in \$GOROOT/lib/time/zoneinfo.zip.

## func `LoadLocationFromTZData`

```
func LoadLocationFromTZData(name string, data []byte) (*Location, error)
```

`LoadLocationFromTZData` returns a `Location` with the given name initialized from the IANA Time Zone database-formatted data. The data should be in the format of a standard IANA time zone file (for example, the content of `/etc/localtime` on Unix systems).

## func `(*Location) String`

```
func (l *Location) String() string
```

`String` returns a descriptive name for the time zone information, corresponding to the name argument to `LoadLocation` or `FixedZone`.

## type `Month`

```
type Month int
```

A `Month` specifies a month of the year (January = 1, ...).

```
const (
 January Month = 1 + iota
 February
 March
 April
 May
 June
 July
 August
 September
 October
 November
 December
)
```

## func `(Month) String`

```
func (m Month) String() string
```

`String` returns the English name of the month ("January", "February", ...).

## type `ParseError`

```
type ParseError struct {
 Layout string
 Value string
 LayoutElem string
 ValueElem string
 Message string
}
```

ParseError describes a problem parsing a time string.

## func (\*ParseError) Error

```
func (e *ParseError) Error() string
```

Error returns the string representation of a ParseError.

## type Ticker

```
type Ticker struct {
 C <-chan Time // The channel on which the ticks are delivered.
 // contains filtered or unexported fields
}
```

A Ticker holds a channel that delivers 'ticks' of a clock at intervals.

## func NewTicker

```
func NewTicker(d Duration) *Ticker
```

NewTicker returns a new Ticker containing a channel that will send the time with a period specified by the duration argument. It adjusts the intervals or drops ticks to make up for slow receivers. The duration d must be greater than zero; if not, NewTicker will panic. Stop the ticker to release associated resources.

## func (\*Ticker) Reset

```
func (t *Ticker) Reset(d Duration)
```

Reset stops a ticker and resets its period to the specified duration. The next tick will arrive after the new period elapses.

## func (\*Ticker) Stop

```
func (t *Ticker) Stop()
```

Stop turns off a ticker. After Stop, no more ticks will be sent. Stop does not close the channel, to prevent a concurrent goroutine reading from the channel from seeing an erroneous "tick".

## type Time

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

A Time represents an instant in time with nanosecond precision.

Programs using times should typically store and pass them as values, not pointers. That is, time variables and struct fields should be of type `time.Time`, not `*time.Time`.

A Time value can be used by multiple goroutines simultaneously except that the methods `GobDecode`, `UnmarshalBinary`, `UnmarshalJSON` and `UnmarshalText` are not concurrency-safe.

Time instants can be compared using the `Before`, `After`, and `Equal` methods. The `Sub` method subtracts two instants, producing a Duration. The `Add` method adds a Time and a Duration, producing a Time.

The zero value of type Time is January 1, year 1, 00:00:00.000000000 UTC. As this time is unlikely to come up in practice, the `IsZero` method gives a simple way of detecting a time that has not been initialized explicitly.

Each Time has associated with it a Location, consulted when computing the presentation form of the time, such as in the `Format`, `Hour`, and `Year` methods. The methods `Local`, `UTC`, and `In` return a Time with a specific location. Changing the location in this way changes only the presentation; it does not change the instant in time being denoted and therefore does not affect the computations described in earlier paragraphs.

Representations of a Time value saved by the `GobEncode`, `MarshalBinary`, `MarshalJSON`, and `MarshalText` methods store the `Time.Location`'s offset, but not the location name. They therefore lose information about Daylight Saving Time.

In addition to the required "wall clock" reading, a Time may contain an optional reading of the current process's monotonic clock, to provide additional precision for comparison or subtraction. See the "Monotonic Clocks" section in the package documentation for details.

Note that the `Go ==` operator compares not just the time instant but also the Location and the monotonic clock reading. Therefore, Time values should not be used as map or database keys without first guaranteeing that the identical Location has been set for all values, which can be achieved through use of the `UTC` or `Local` method, and that the monotonic clock reading has been stripped by setting `t = t.Round(0)`. In general, prefer `t.Equal(u)` to `t == u`, since `t.Equal` uses the most accurate comparison available and correctly handles the case when only one of its arguments has a monotonic clock reading.

## func Date

```
func Date(year int, month Month, day, hour, min, sec, nsec int, loc *Location) Time
```

Date returns the Time corresponding to

```
yyyy-mm-dd hh:mm:ss + nsec nanoseconds
```

in the appropriate zone for that time in the given location.

The month, day, hour, min, sec, and nsec values may be outside their usual ranges and will be normalized during the conversion. For example, October 32 converts to November 1.

A daylight savings time transition skips or repeats times. For example, in the United States, March 13, 2011 2:15am never occurred, while November 6, 2011 1:15am occurred twice. In such cases, the choice of time zone, and therefore the time, is not well-defined. Date returns a time that is correct in one of the two zones involved in the transition, but it does not guarantee which.

Date panics if loc is nil.

## func Now

```
func Now() Time
```

Now returns the current local time.

## func Parse

```
func Parse(layout, value string) (Time, error)
```

Parse parses a formatted string and returns the time value it represents. The layout defines the format by showing how the reference time, defined to be

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

would be interpreted if it were the value; it serves as an example of the input format. The same interpretation will then be made to the input string.

Predefined layouts ANSIC, UnixDate, RFC3339 and others describe standard and convenient representations of the reference time. For more information about the formats and the definition of the reference time, see the documentation for ANSIC and the other constants defined by this package. Also, the executable example for Time.Format demonstrates the working of the layout string in detail and is a good reference.

Elements omitted from the value are assumed to be zero or, when zero is impossible, one, so parsing "3:04pm" returns the time corresponding to Jan 1, year 0, 15:04:00 UTC (note that because the year is 0, this time is before the zero Time). Years must be in the range 0000..9999. The day of the week is checked for syntax but it is otherwise ignored.

For layouts specifying the two-digit year 06, a value NN >= 69 will be treated as 19NN and a value NN < 69 will be treated as 20NN.

In the absence of a time zone indicator, Parse returns a time in UTC.

When parsing a time with a zone offset like -0700, if the offset corresponds to a time zone used by the current location (Local), then Parse uses that location and zone in the returned time. Otherwise it records the time as being in a fabricated location with time fixed at the given zone offset.

When parsing a time with a zone abbreviation like MST, if the zone abbreviation has a defined offset in the current location, then that offset is used. The zone abbreviation "UTC" is recognized as UTC regardless of location. If the zone abbreviation is unknown, Parse records the time as being in a fabricated location with the given zone abbreviation and a zero offset. This choice means that such a time can be parsed and reformatted with the same layout losslessly, but the exact instant used in the representation will differ by the actual zone offset. To avoid such problems, prefer time layouts that use a numeric zone offset, or use ParseInLocation.

## func `ParseInLocation`

```
func ParseInLocation(layout, value string, loc *Location) (Time, error)
```

ParseInLocation is like Parse but differs in two important ways. First, in the absence of time zone information, Parse interprets a time as UTC; ParseInLocation interprets the time as in the given location. Second, when given a zone offset or abbreviation, Parse tries to match it against the Local location; ParseInLocation uses the given location.

## func `Unix`

```
func Unix(sec int64, nsec int64) Time
```

Unix returns the local Time corresponding to the given Unix time, sec seconds and nsec nanoseconds since January 1, 1970 UTC. It is valid to pass nsec outside the range [0, 999999999]. Not all sec values have a corresponding time value. One such value is 1<<63-1 (the largest int64 value).

## func (`Time`) `Add`

```
func (t Time) Add(d Duration) Time
```

Add returns the time t+d.

## func (`Time`) `AddDate`

```
func (t Time) AddDate(years int, months int, days int) Time
```

AddDate returns the time corresponding to adding the given number of years, months, and days to t. For example, AddDate(-1, 2, 3) applied to January 1, 2011 returns March 4, 2010.

AddDate normalizes its result in the same way that Date does, so, for example, adding one month to October 31 yields December 1, the normalized form for November 31.

## func (Time) After

```
func (t Time) After(u Time) bool
```

After reports whether the time instant t is after u.

## func (Time) AppendFormat

```
func (t Time) AppendFormat(b []byte, layout string) []byte
```

AppendFormat is like Format but appends the textual representation to b and returns the extended buffer.

## func (Time) Before

```
func (t Time) Before(u Time) bool
```

Before reports whether the time instant t is before u.

## func (Time) Clock

```
func (t Time) Clock() (hour, min, sec int)
```

Clock returns the hour, minute, and second within the day specified by t.

## func (Time) Date

```
func (t Time) Date() (year int, month Month, day int)
```

Date returns the year, month, and day in which t occurs.

## func (Time) Day

```
func (t Time) Day() int
```

Day returns the day of the month specified by t.

## func (Time) Equal

```
func (t Time) Equal(u Time) bool
```

Equal reports whether t and u represent the same time instant. Two times can be equal even if they are in different locations. For example, 6:00 +0200 and 4:00 UTC are Equal. See the documentation on the Time type for the pitfalls of using == with Time values; most code should use Equal instead.

## func (Time) Format

```
func (t Time) Format(layout string) string
```

Format returns a textual representation of the time value formatted according to layout, which defines the format by showing how the reference time, defined to be

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

would be displayed if it were the value; it serves as an example of the desired output. The same display rules will then be applied to the time value.

A fractional second is represented by adding a period and zeros to the end of the seconds section of layout string, as in "15:04:05.000" to format a time stamp with millisecond precision.

Predefined layouts ANSIC, UnixDate, RFC3339 and others describe standard and convenient representations of the reference time. For more information about the formats and the definition of the reference time, see the documentation for ANSIC and the other constants defined by this package.

## func (\*Time) GobDecode

```
func (t *Time) GobDecode(data []byte) error
```

GobDecode implements the gob.GobDecoder interface.

## func (Time) GobEncode

```
func (t Time) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

## func (Time) Hour

```
func (t Time) Hour() int
```

Hour returns the hour within the day specified by t, in the range [0, 23].

## func (Time) ISOWeek

```
func (t Time) ISOWeek() (year, week int)
```

ISOWeek returns the ISO 8601 year and week number in which t occurs. Week ranges from 1 to 53. Jan 01 to Jan 03 of year n might belong to week 52 or 53 of year n-1, and Dec 29 to Dec 31 might belong to week 1 of year n+1.

## func (Time) In

```
func (t Time) In(loc *Location) Time
```

In returns a copy of t representing the same time instant, but with the copy's location information set to loc for display purposes.

In panics if loc is nil.

## func (Time) IsZero

```
func (t Time) IsZero() bool
```

IsZero reports whether t represents the zero time instant, January 1, year 1, 00:00:00 UTC.

## func (Time) Local

```
func (t Time) Local() Time
```

Local returns t with the location set to local time.

## func (Time) Location

```
func (t Time) Location() *Location
```

Location returns the time zone information associated with t.

## func (Time) MarshalBinary

```
func (t Time) MarshalBinary() ([]byte, error)
```

MarshalBinary implements the encoding.BinaryMarshaler interface.

## func (Time) MarshalJSON

```
func (t Time) MarshalJSON() ([]byte, error)
```

MarshalJSON implements the json.Marshaler interface. The time is a quoted string in [RFC 3339](#) format, with sub-second precision added if present.

## func (Time) MarshalText

```
func (t Time) MarshalText() ([]byte, error)
```

MarshalText implements the encoding.TextMarshaler interface. The time is formatted in [RFC 3339](#) format, with sub-second precision added if present.

## func (Time) Minute

```
func (t Time) Minute() int
```

Minute returns the minute offset within the hour specified by t, in the range [0, 59].

## func (Time) Month

```
func (t Time) Month() Month
```

Month returns the month of the year specified by t.

## func (Time) Nanosecond

```
func (t Time) Nanosecond() int
```

Nanosecond returns the nanosecond offset within the second specified by t, in the range [0, 999999999].

## func (Time) Round

```
func (t Time) Round(d Duration) Time
```

Round returns the result of rounding t to the nearest multiple of d (since the zero time). The rounding behavior for halfway values is to round up. If d <= 0, Round returns t stripped of any monotonic clock reading but otherwise unchanged.

Round operates on the time as an absolute duration since the zero time; it does not operate on the presentation form of the time. Thus, Round(Hour) may return a time with a non-zero minute, depending on the time's Location.

## func (Time) Second

```
func (t Time) Second() int
```

Second returns the second offset within the minute specified by t, in the range [0, 59].

## func (Time) String

```
func (t Time) String() string
```

String returns the time formatted using the format string

```
"2006-01-02 15:04:05.999999999 -0700 MST"
```

If the time has a monotonic clock reading, the returned string includes a final field "m=±<value>", where value is the monotonic clock reading formatted as a decimal number of seconds.

The returned string is meant for debugging; for a stable serialized representation, use `t.MarshalText`, `t.MarshalBinary`, or `t.Format` with an explicit format string.

## func (Time) Sub

```
func (t Time) Sub(u Time) Duration
```

`Sub` returns the duration `t-u`. If the result exceeds the maximum (or minimum) value that can be stored in a `Duration`, the maximum (or minimum) duration will be returned. To compute `t-d` for a duration `d`, use `t.Add(-d)`.

## func (Time) Truncate

```
func (t Time) Truncate(d Duration) Time
```

`Truncate` returns the result of rounding `t` down to a multiple of `d` (since the zero time). If `d <= 0`, `Truncate` returns `t` stripped of any monotonic clock reading but otherwise unchanged.

`Truncate` operates on the time as an absolute duration since the zero time; it does not operate on the presentation form of the time. Thus, `Truncate(Hour)` may return a time with a non-zero minute, depending on the time's `Location`.

## func (Time) UTC

```
func (t Time) UTC() Time
```

`UTC` returns `t` with the location set to UTC.

## func (Time) Unix

```
func (t Time) Unix() int64
```

`Unix` returns `t` as a Unix time, the number of seconds elapsed since January 1, 1970 UTC. The result does not depend on the location associated with `t`. Unix-like operating systems often record time as a 32-bit count of seconds, but since the method here returns a 64-bit value it is valid for billions of years into the past or future.

## func (Time) UnixNano

```
func (t Time) UnixNano() int64
```

UnixNano returns t as a Unix time, the number of nanoseconds elapsed since January 1, 1970 UTC. The result is undefined if the Unix time in nanoseconds cannot be represented by an int64 (a date before the year 1678 or after 2262). Note that this means the result of calling UnixNano on the zero Time is undefined. The result does not depend on the location associated with t.

## func (\*Time) **UnmarshalBinary**

```
func (t *Time) UnmarshalBinary(data []byte) error
```

UnmarshalBinary implements the encoding.BinaryUnmarshaler interface.

## func (\*Time) **UnmarshalJSON**

```
func (t *Time) UnmarshalJSON(data []byte) error
```

UnmarshalJSON implements the json.Unmarshaler interface. The time is expected to be a quoted string in [RFC 3339](#) format.

## func (\*Time) **UnmarshalText**

```
func (t *Time) UnmarshalText(data []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface. The time is expected to be in [RFC 3339](#) format.

## func (Time) **Weekday**

```
func (t Time) Weekday() Weekday
```

Weekday returns the day of the week specified by t.

## func (Time) **Year**

```
func (t Time) Year() int
```

Year returns the year in which t occurs.

## func (Time) **YearDay**

```
func (t Time) YearDay() int
```

YearDay returns the day of the year specified by t, in the range [1,365] for non-leap years, and [1,366] in leap years.

## func (Time) Zone

```
func (t Time) Zone() (name string, offset int)
```

Zone computes the time zone in effect at time t, returning the abbreviated name of the zone (such as "CET") and its offset in seconds east of UTC.

## type Timer

```
type Timer struct {
 C <-chan Time
 // contains filtered or unexported fields
}
```

The Timer type represents a single event. When the Timer expires, the current time will be sent on C, unless the Timer was created by AfterFunc. A Timer must be created with NewTimer or AfterFunc.

## func AfterFunc

```
func AfterFunc(d Duration, f func()) *Timer
```

AfterFunc waits for the duration to elapse and then calls f in its own goroutine. It returns a Timer that can be used to cancel the call using its Stop method.

## func NewTimer

```
func NewTimer(d Duration) *Timer
```

NewTimer creates a new Timer that will send the current time on its channel after at least duration d.

## func (\*Timer) Reset

```
func (t *Timer) Reset(d Duration) bool
```

Reset changes the timer to expire after duration d. It returns true if the timer had been active, false if the timer had expired or been stopped.

Reset should be invoked only on stopped or expired timers with drained channels. If a program has already received a value from t.C, the timer is known to have expired and the channel drained, so t.Reset can be used directly. If a program has not yet received a value from t.C, however, the timer must be stopped and—if Stop reports that the timer expired before being stopped—the channel explicitly drained:

```
if !t.Stop() {
 <-t.C
}
t.Reset(d)
```

This should not be done concurrent to other receives from the Timer's channel.

Note that it is not possible to use Reset's return value correctly, as there is a race condition between draining the channel and the new timer expiring. Reset should always be invoked on stopped or expired channels, as described above. The return value exists to preserve compatibility with existing programs.

## func (\*Timer) Stop

```
func (t *Timer) Stop() bool
```

Stop prevents the Timer from firing. It returns true if the call stops the timer, false if the timer has already expired or been stopped. Stop does not close the channel, to prevent a read from the channel succeeding incorrectly.

To ensure the channel is empty after a call to Stop, check the return value and drain the channel. For example, assuming the program has not received from t.C already:

```
if !t.Stop() {
 <-t.C
}
```

This cannot be done concurrent to other receives from the Timer's channel or other calls to the Timer's Stop method.

For a timer created with AfterFunc(d, f), if t.Stop returns false, then the timer has already expired and the function f has been started in its own goroutine; Stop does not wait for f to complete before returning. If the caller needs to know whether f is completed, it must coordinate with f explicitly.

## type Weekday

```
type Weekday int
```

A Weekday specifies a day of the week (Sunday = 0, ...).

```
const (
 Sunday Weekday = iota
 Monday
 Tuesday
 Wednesday
 Thursday
 Friday
 Saturday
)
```

## func (Weekday) String

```
func (d Weekday) String() string
```

String returns the English name of the day ("Sunday", "Monday", ...).

# Package unicode

go1.15.2    Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)

[Doc](#)    [Overview](#)    [Subdirectories](#)    [Versions](#)    [Imports](#)    [Imported By](#)    [Licenses](#)

## Overview

Package unicode provides data and functions to test some properties of Unicode code points.

## Constants

```
const (
 MaxRune = '\U0010FFFF' // Maximum valid Unicode code point.
 ReplacementChar = '\uFFFD' // Represents invalid code points.
 MaxASCII = '\u007F' // maximum ASCII value.
 MaxLatin1 = '\u00FF' // maximum Latin-1 value.
)
```

```
const (
 Uppercase = iota
 Lowercase
 Titlecase
 Maxcase
)
```

Indices into the Delta arrays inside CaseRanges for case mapping.

```
const (
 UpperLower = MaxRune + 1 // (Cannot be a valid delta.)
)
```

If the Delta field of a CaseRange is UpperLower, it means this CaseRange represents a sequence of the form (say) Upper Lower Upper Lower.

```
const Version = "12.0.0"
```

Version is the Unicode edition from which the tables are derived.

## Variables

```
var (
 Cc = _Cc // Cc is the set of Unicode characters in category Cc (Other, control, format, punctuation, symbol, separator, other)
 Cf = _Cf // Cf is the set of Unicode characters in category Cf (Other, format, punctuation, symbol, separator, other)
 Co = _Co // Co is the set of Unicode characters in category Co (Other, private use, other)
 Cs = _Cs // Cs is the set of Unicode characters in category Cs (Other, surrogate, other)
```

```

Digit = _Nd // Digit is the set of Unicode characters with the "decimal digit" property.
Nd = _Nd // Nd is the set of Unicode characters in category Nd (Number, decimal).
Letter = _L // Letter/L is the set of Unicode letters, category L.
L = _L
Lm = _Lm // Lm is the set of Unicode characters in category Lm (Letter, modifier).
Lo = _Lo // Lo is the set of Unicode characters in category Lo (Letter, other).
Lower = _Ll // Lower is the set of Unicode lower case letters.
Ll = _Ll // Ll is the set of Unicode characters in category Ll (Letter, lower case).
Mark = _M // Mark/M is the set of Unicode mark characters, category M.
M = _M
Mc = _Mc // Mc is the set of Unicode characters in category Mc (Mark, spacing).
Me = _Me // Me is the set of Unicode characters in category Me (Mark, enclosing).
Mn = _Mn // Mn is the set of Unicode characters in category Mn (Mark, nonspacing).
Nl = _Nl // Nl is the set of Unicode characters in category Nl (Number, letter).
No = _No // No is the set of Unicode characters in category No (Number, other).
Number = _N // Number/N is the set of Unicode number characters, category N.
N = _N
Other = _C // Other/C is the set of Unicode control and special characters, category C.
C = _C
Pc = _Pc // Pc is the set of Unicode characters in category Pc (Punctuation, connector).
Pd = _Pd // Pd is the set of Unicode characters in category Pd (Punctuation, dash).
Pe = _Pe // Pe is the set of Unicode characters in category Pe (Punctuation, end-of-sentence).
Pf = _Pf // Pf is the set of Unicode characters in category Pf (Punctuation, form).
Pi = _Pi // Pi is the set of Unicode characters in category Pi (Punctuation, ideographic).
Po = _Po // Po is the set of Unicode characters in category Po (Punctuation, other).
Ps = _Ps // Ps is the set of Unicode characters in category Ps (Punctuation, separator).
Punct = _P // Punct/P is the set of Unicode punctuation characters, category P.
P = _P
Sc = _Sc // Sc is the set of Unicode characters in category Sc (Symbol, currency).
Sk = _Sk // Sk is the set of Unicode characters in category Sk (Symbol, modifier).
Sm = _Sm // Sm is the set of Unicode characters in category Sm (Symbol, math).
So = _So // So is the set of Unicode characters in category So (Symbol, other).
Space = _Z // Space/Z is the set of Unicode space characters, category Z.
Z = _Z
Symbol = _S // Symbol/S is the set of Unicode symbol characters, category S.
S = _S
Title = _Lt // Title is the set of Unicode title case letters.
Lt = _Lt // Lt is the set of Unicode characters in category Lt (Letter, title case).
Upper = _Lu // Upper is the set of Unicode upper case letters.
Lu = _Lu // Lu is the set of Unicode characters in category Lu (Letter, upper case).
Zl = _Zl // Zl is the set of Unicode characters in category Zl (Separator, line).
Zp = _Zp // Zp is the set of Unicode characters in category Zp (Separator, page).
Zs = _Zs // Zs is the set of Unicode characters in category Zs (Separator, space).
)

```

These variables have type `*RangeTable`.

```

var (
 Adlam = _Adlam // Adlam is the set of Unicode characters in category Adlam.
 Ahom = _Ahom // Ahom is the set of Unicode characters in category Ahom.
 Anatolian_Hieroglyphs = _Anatolian_Hieroglyphs // Anatolian_Hieroglyphs is the set of Unicode characters in category Anatolian_Hieroglyphs.
 Arabic = _Arabic // Arabic is the set of Unicode characters in category Arabic.
 Armenian = _Armenian // Armenian is the set of Unicode characters in category Armenian.
)

```

|                        |                           |                                                 |
|------------------------|---------------------------|-------------------------------------------------|
| Avestan                | = _Avestan                | // Avestan is the set of Unicode                |
| Balinese               | = _Balinese               | // Balinese is the set of Unicode               |
| Bamum                  | = _Bamum                  | // Bamum is the set of Unicode                  |
| Bassa_Vah              | = _Bassa_Vah              | // Bassa_Vah is the set of Unicode              |
| Batak                  | = _Batak                  | // Batak is the set of Unicode                  |
| Bengali                | = _Bengali                | // Bengali is the set of Unicode                |
| Bhaiksuki              | = _Bhaiksuki              | // Bhaiksuki is the set of Unicode              |
| Bopomofo               | = _Bopomofo               | // Bopomofo is the set of Unicode               |
| Brahmi                 | = _Brahmi                 | // Brahmi is the set of Unicode                 |
| Braille                | = _Braille                | // Braille is the set of Unicode                |
| Buginese               | = _Buginese               | // Buginese is the set of Unicode               |
| Buhid                  | = _Buhid                  | // Buhid is the set of Unicode                  |
| Canadian_Aboriginal    | = _Canadian_Aboriginal    | // Canadian_Aboriginal is the set of Unicode    |
| Carian                 | = _Carian                 | // Carian is the set of Unicode                 |
| Caucasian_Albanian     | = _Caucasian_Albanian     | // Caucasian_Albanian is the set of Unicode     |
| Chakma                 | = _Chakma                 | // Chakma is the set of Unicode                 |
| Cham                   | = _Cham                   | // Cham is the set of Unicode                   |
| Cherokee               | = _Cherokee               | // Cherokee is the set of Unicode               |
| Common                 | = _Common                 | // Common is the set of Unicode                 |
| Coptic                 | = _Coptic                 | // Coptic is the set of Unicode                 |
| Cuneiform              | = _Cuneiform              | // Cuneiform is the set of Unicode              |
| Cypriot                | = _Cypriot                | // Cypriot is the set of Unicode                |
| Cyrillic               | = _Cyrillic               | // Cyrillic is the set of Unicode               |
| Deseret                | = _Deseret                | // Deseret is the set of Unicode                |
| Devanagari             | = _Devanagari             | // Devanagari is the set of Unicode             |
| Dogra                  | = _Dogra                  | // Dogra is the set of Unicode                  |
| Duployan               | = _Duployan               | // Duployan is the set of Unicode               |
| Egyptian_Hieroglyphs   | = _Egyptian_Hieroglyphs   | // Egyptian_Hieroglyphs is the set of Unicode   |
| Elbasan                | = _Elbasan                | // Elbasan is the set of Unicode                |
| Elymaic                | = _Elymaic                | // Elymaic is the set of Unicode                |
| Ethiopic               | = _Ethiopic               | // Ethiopic is the set of Unicode               |
| Georgian               | = _Georgian               | // Georgian is the set of Unicode               |
| Glagolitic             | = _Glagolitic             | // Glagolitic is the set of Unicode             |
| Gothic                 | = _Gothic                 | // Gothic is the set of Unicode                 |
| Grantha                | = _Grantha                | // Grantha is the set of Unicode                |
| Greek                  | = _Greek                  | // Greek is the set of Unicode                  |
| Gujarati               | = _Gujarati               | // Gujarati is the set of Unicode               |
| Gunjala_Gondi          | = _Gunjala_Gondi          | // Gunjala_Gondi is the set of Unicode          |
| Gurmukhi               | = _Gurmukhi               | // Gurmukhi is the set of Unicode               |
| Han                    | = _Han                    | // Han is the set of Unicode                    |
| Hangul                 | = _Hangul                 | // Hangul is the set of Unicode                 |
| Hanifi_Rohingya        | = _Hanifi_Rohingya        | // Hanifi_Rohingya is the set of Unicode        |
| Hanunoo                | = _Hanunoo                | // Hanunoo is the set of Unicode                |
| Hatran                 | = _Hatran                 | // Hatran is the set of Unicode                 |
| Hebrew                 | = _Hebrew                 | // Hebrew is the set of Unicode                 |
| Hiragana               | = _Hiragana               | // Hiragana is the set of Unicode               |
| Imperial_Aramaic       | = _Imperial_Aramaic       | // Imperial_Aramaic is the set of Unicode       |
| Inherited              | = _Inherited              | // Inherited is the set of Unicode              |
| Inscriptional_Pahlavi  | = _Inscriptional_Pahlavi  | // Inscriptional_Pahlavi is the set of Unicode  |
| Inscriptional_Parthian | = _Inscriptional_Parthian | // Inscriptional_Parthian is the set of Unicode |
| Javanese               | = _Javanese               | // Javanese is the set of Unicode               |
| Kaithi                 | = _Kaithi                 | // Kaithi is the set of Unicode                 |
| Kannada                | = _Kannada                | // Kannada is the set of Unicode                |
| Katakana               | = _Katakana               | // Katakana is the set of Unicode               |

|                        |                           |                                                            |
|------------------------|---------------------------|------------------------------------------------------------|
| Kayah_Li               | = _Kayah_Li               | // Kayah_Li is the set of Unicode                          |
| Kharoshthi             | = _Kharoshthi             | // Kharoshthi is the set of Unicode                        |
| Khmer                  | = _Khmer                  | // Khmer is the set of Unicode characters                  |
| Khojki                 | = _Khojki                 | // Khojki is the set of Unicode characters                 |
| Khudawadi              | = _Khudawadi              | // Khudawadi is the set of Unicode characters              |
| Lao                    | = _Lao                    | // Lao is the set of Unicode characters                    |
| Latin                  | = _Latin                  | // Latin is the set of Unicode characters                  |
| Lepcha                 | = _Lepcha                 | // Lepcha is the set of Unicode characters                 |
| Limbu                  | = _Limbu                  | // Limbu is the set of Unicode characters                  |
| Linear_A               | = _Linear_A               | // Linear_A is the set of Unicode characters               |
| Linear_B               | = _Linear_B               | // Linear_B is the set of Unicode characters               |
| Lisu                   | = _Lisu                   | // Lisu is the set of Unicode characters                   |
| Lycian                 | = _Lycian                 | // Lycian is the set of Unicode characters                 |
| Lydian                 | = _Lydian                 | // Lydian is the set of Unicode characters                 |
| Mahajani               | = _Mahajani               | // Mahajani is the set of Unicode characters               |
| Makasar                | = _Makasar                | // Makasar is the set of Unicode characters                |
| Malayalam              | = _Malayalam              | // Malayalam is the set of Unicode characters              |
| Mandaic                | = _Mandaic                | // Mandaic is the set of Unicode characters                |
| Manichaean             | = _Manichaean             | // Manichaean is the set of Unicode characters             |
| Marchen                | = _Marchen                | // Marchen is the set of Unicode characters                |
| Masaram_Gondi          | = _Masaram_Gondi          | // Masaram_Gondi is the set of Unicode characters          |
| Medefaidrin            | = _Medefaidrin            | // Medefaidrin is the set of Unicode characters            |
| Meetei_Mayek           | = _Meetei_Mayek           | // Meetei_Mayek is the set of Unicode characters           |
| Mende_Kikakui          | = _Mende_Kikakui          | // Mende_Kikakui is the set of Unicode characters          |
| Meroitic_Cursive       | = _Meroitic_Cursive       | // Meroitic_Cursive is the set of Unicode characters       |
| Meroitic_Hieroglyphs   | = _Meroitic_Hieroglyphs   | // Meroitic_Hieroglyphs is the set of Unicode characters   |
| Miao                   | = _Miao                   | // Miao is the set of Unicode characters                   |
| Modi                   | = _Modi                   | // Modi is the set of Unicode characters                   |
| Mongolian              | = _Mongolian              | // Mongolian is the set of Unicode characters              |
| Mro                    | = _Mro                    | // Mro is the set of Unicode characters                    |
| Multani                | = _Multani                | // Multani is the set of Unicode characters                |
| Myanmar                | = _Myanmar                | // Myanmar is the set of Unicode characters                |
| Nabataean              | = _Nabataean              | // Nabataean is the set of Unicode characters              |
| Nandinagari            | = _Nandinagari            | // Nandinagari is the set of Unicode characters            |
| New_Tai_Lue            | = _New_Tai_Lue            | // New_Tai_Lue is the set of Unicode characters            |
| Newa                   | = _Newa                   | // Newa is the set of Unicode characters                   |
| Nko                    | = _Nko                    | // Nko is the set of Unicode characters                    |
| Nushu                  | = _Nushu                  | // Nushu is the set of Unicode characters                  |
| Nyiakeng_Puachue_Hmong | = _Nyiakeng_Puachue_Hmong | // Nyiakeng_Puachue_Hmong is the set of Unicode characters |
| Ogham                  | = _Ogham                  | // Ogham is the set of Unicode characters                  |
| Ol_Chiki               | = _Ol_Chiki               | // Ol_Chiki is the set of Unicode characters               |
| Old_Hungarian          | = _Old_Hungarian          | // Old_Hungarian is the set of Unicode characters          |
| Old_Italic             | = _Old_Italic             | // Old_Italic is the set of Unicode characters             |
| Old_North_Arabian      | = _Old_North_Arabian      | // Old_North_Arabian is the set of Unicode characters      |
| Old_Permic             | = _Old_Permic             | // Old_Permic is the set of Unicode characters             |
| Old_Persian            | = _Old_Persian            | // Old_Persian is the set of Unicode characters            |
| Old_Sogdian            | = _Old_Sogdian            | // Old_Sogdian is the set of Unicode characters            |
| Old_South_Arabian      | = _Old_South_Arabian      | // Old_South_Arabian is the set of Unicode characters      |
| Old_Turkic             | = _Old_Turkic             | // Old_Turkic is the set of Unicode characters             |
| Oriya                  | = _Oriya                  | // Oriya is the set of Unicode characters                  |
| Osage                  | = _Osage                  | // Osage is the set of Unicode characters                  |
| Osmanya                | = _Osmanya                | // Osmanya is the set of Unicode characters                |
| Pahawh_Hmong           | = _Pahawh_Hmong           | // Pahawh_Hmong is the set of Unicode characters           |
| Palmyrene              | = _Palmyrene              | // Palmyrene is the set of Unicode characters              |

```

Pau_Cin_Hau = _Pau_Cin_Hau // Pau_Cin_Hau is the set of Unicode
Phags_Pa = _Phags_Pa // Phags_Pa is the set of Unicode
Phoenician = _Phoenician // Phoenician is the set of Unicode
Psalter_Pahlavi = _Psalter_Pahlavi // Psalter_Pahlavi is the set of Unicode
Rejang = _Rejang // Rejang is the set of Unicode
Runic = _Runic // Runic is the set of Unicode characters
Samaritan = _Samaritan // Samaritan is the set of Unicode
Saurashtra = _Saurashtra // Saurashtra is the set of Unicode
Sharada = _Sharada // Sharada is the set of Unicode
Shavian = _Shavian // Shavian is the set of Unicode
Siddham = _Siddham // Siddham is the set of Unicode
SignWriting = _SignWriting // SignWriting is the set of Unicode
Sinhala = _Sinhala // Sinhala is the set of Unicode
Sogdian = _Sogdian // Sogdian is the set of Unicode
Sora_Sompeng = _Sora_Sompeng // Sora_Sompeng is the set of Unicode
Soyombo = _Soyombo // Soyombo is the set of Unicode
Sundanese = _Sundanese // Sundanese is the set of Unicode
Syloti_Nagri = _Syloti_Nagri // Syloti_Nagri is the set of Unicode
Syriac = _Syriac // Syriac is the set of Unicode
Tagalog = _Tagalog // Tagalog is the set of Unicode
Tagbanwa = _Tagbanwa // Tagbanwa is the set of Unicode
Tai_Le = _Tai_Le // Tai_Le is the set of Unicode
Tai_Tham = _Tai_Tham // Tai_Tham is the set of Unicode
Tai_Viet = _Tai_Viet // Tai_Viet is the set of Unicode
Takri = _Takri // Takri is the set of Unicode characters
Tamil = _Tamil // Tamil is the set of Unicode characters
Tangut = _Tangut // Tangut is the set of Unicode
Telugu = _Telugu // Telugu is the set of Unicode
Thaana = _Thaana // Thaana is the set of Unicode
Thai = _Thai // Thai is the set of Unicode characters
Tibetan = _Tibetan // Tibetan is the set of Unicode
Tifinagh = _Tifinagh // Tifinagh is the set of Unicode
Tirhuta = _Tirhuta // Tirhuta is the set of Unicode
Ugaritic = _Ugaritic // Ugaritic is the set of Unicode
Vai = _Vai // Vai is the set of Unicode characters
Wancho = _Wancho // Wancho is the set of Unicode
Warang_Citi = _Warang_Citi // Warang_Citi is the set of Unicode
Yi = _Yi // Yi is the set of Unicode characters
Zanabazar_Square = _Zanabazar_Square // Zanabazar_Square is the set of Unicode
)

```

These variables have type `*RangeTable`.

```

var (
 ASCII_Hex_Digit = _ASCII_Hex_Digit // ASCII
 Bidi_Control = _Bidi_Control // Bidi_
 Dash = _Dash // Dash
 Deprecated = _Deprecated // Depre
 Diacritic = _Diacritic // Diacr
 Extender = _Extender // Exten
 Hex_Digit = _Hex_Digit // Hex_D
 Hyphen = _Hyphen // Hyphe
 IDS_Binary_Operator = _IDS_Binary_Operator // IDS_B
)

```

```

IDS_Trinary_Operator = _IDS_Trinary_Operator // IDS_T
Ideographic = _Ideographic // Ideog
Join_Control = _Join_Control // Join_
Logical_Order_Exception = _Logical_Order_Exception // Logic
Noncharacter_Code_Point = _Noncharacter_Code_Point // Nonch
Other_Alphabetic = _Other_Alphabetic // Other
Other_Default_Ignorable_Code_Point = _Other_Default_Ignorable_Code_Point // Other
Other_Grapheme_Extend = _Other_Grapheme_Extend // Other
Other_ID_Continue = _Other_ID_Continue // Other
Other_ID_Start = _Other_ID_Start // Other
Other_Lowercase = _Other_Lowercase // Other
Other_Math = _Other_Math // Other
Other_Uppercase = _Other_Uppercase // Other
Pattern_Syntax = _Pattern_Syntax // Patte
Pattern_White_Space = _Pattern_White_Space // Patte
Prepended_Concatenation_Mark = _Prepended_Concatenation_Mark // Prepe
Quotation_Mark = _Quotation_Mark // Quota
Radical = _Radical // Radic
Regional_Indicator = _Regional_Indicator // Regio
STerm = _Sentence_Terminal // STerm
Sentence_Terminal = _Sentence_Terminal // Sente
Soft_Dotted = _Soft_Dotted // Soft_
Terminal_Punctuation = _Terminal_Punctuation // Termi
Unified_Ideograph = _Unified_Ideograph // Unifi
Variation_Selector = _Variation_Selector // Varia
White_Space = _White_Space // White
)

```

These variables have type `*RangeTable`.

```
var CaseRanges = _CaseRanges
```

`CaseRanges` is the table describing case mappings for all letters with non-self mappings.

```
var Categories = map[string]*RangeTable{
 "C": C,
 "Cc": Cc,
 "Cf": Cf,
 "Co": Co,
 "Cs": Cs,
 "L": L,
 "Ll": Ll,
 "Lm": Lm,
 "Lo": Lo,
 "Lt": Lt,
 "Lu": Lu,
 "M": M,
 "Mc": Mc,
 "Me": Me,
 "Mn": Mn,
 "N": N,
 "Nd": Nd,
```

```
"Nl": Nl,
"No": No,
"P": P,
"Pc": Pc,
"Pd": Pd,
"Pe": Pe,
"Pf": Pf,
"Pi": Pi,
"Po": Po,
"Ps": Ps,
"S": S,
"Sc": Sc,
"Sk": Sk,
"Sm": Sm,
"So": So,
"Z": Z,
"Zl": Zl,
"Zp": Zp,
"Zs": Zs,
}
```

Categories is the set of Unicode category tables.

```
var FoldCategory = map[string]*RangeTable{
 "L": foldL,
 "Ll": foldLl,
 "Lt": foldLt,
 "Lu": foldLu,
 "M": foldM,
 "Mn": foldMn,
}
```

FoldCategory maps a category name to a table of code points outside the category that are equivalent under simple case folding to code points inside the category. If there is no entry for a category name, there are no such points.

```
var FoldScript = map[string]*RangeTable{
 "Common": foldCommon,
 "Greek": foldGreek,
 "Inherited": foldInherited,
}
```

FoldScript maps a script name to a table of code points outside the script that are equivalent under simple case folding to code points inside the script. If there is no entry for a script name, there are no such points.

```
var GraphicRanges = []*RangeTable{
 L, M, N, P, S, Zs,
}
```

GraphicRanges defines the set of graphic characters according to Unicode.

```
var PrintRanges = []*RangeTable{
 L, M, N, P, S,
}
```

PrintRanges defines the set of printable characters according to Go. ASCII space, U+0020, is handled separately.

```
var Properties = map[string]*RangeTable{
 "ASCII_Hex_Digit": ASCII_Hex_Digit,
 "Bidi_Control": Bidi_Control,
 "Dash": Dash,
 "Deprecated": Deprecated,
 "Diacritic": Diacritic,
 "Extender": Extender,
 "Hex_Digit": Hex_Digit,
 "Hyphen": Hyphen,
 "IDS_Binary_Operator": IDS_Binary_Operator,
 "IDS_Triinary_Operator": IDS_Triinary_Operator,
 "Ideographic": Ideographic,
 "Join_Control": Join_Control,
 "Logical_Order_Exception": Logical_Order_Exception,
 "Noncharacter_Code_Point": Noncharacter_Code_Point,
 "Other_Alphabetic": Other_Alphabetic,
 "Other_Default_Ignorable_Code_Point": Other_Default_Ignorable_Code_Point,
 "Other_Grapheme_Extend": Other_Grapheme_Extend,
 "Other_ID_Continue": Other_ID_Continue,
 "Other_ID_Start": Other_ID_Start,
 "Other_Lowercase": Other_Lowercase,
 "Other_Math": Other_Math,
 "Other_Uppercase": Other_Uppercase,
 "Pattern_Syntax": Pattern_Syntax,
 "Pattern_White_Space": Pattern_White_Space,
 "Prepended_Concatenation_Mark": Prepended_Concatenation_Mark,
 "Quotation_Mark": Quotation_Mark,
 "Radical": Radical,
 "Regional_Indicator": Regional_Indicator,
 "Sentence_Terminal": Sentence_Terminal,
 "STerm": Sentence_Terminal,
 "Soft_Dotted": Soft_Dotted,
 "Terminal_Punctuation": Terminal_Punctuation,
 "Unified_Ideograph": Unified_Ideograph,
 "Variation_Selector": Variation_Selector,
 "White_Space": White_Space,
}
```

Properties is the set of Unicode property tables.

```
var Scripts = map[string]*RangeTable{ /* 152 elements not displayed */
```

```
}
```

Scripts is the set of Unicode script tables.

## func **In**

```
func In(r rune, ranges ...*RangeTable) bool
```

In reports whether the rune is a member of one of the ranges.

## func **Is**

```
func Is(rangeTab *RangeTable, r rune) bool
```

Is reports whether the rune is in the specified table of ranges.

## func **IsControl**

```
func IsControl(r rune) bool
```

IsControl reports whether the rune is a control character. The C (Other) Unicode category includes more code points such as surrogates; use Is(C, r) to test for them.

## func **IsDigit**

```
func IsDigit(r rune) bool
```

IsDigit reports whether the rune is a decimal digit.

## func **IsGraphic**

```
func IsGraphic(r rune) bool
```

IsGraphic reports whether the rune is defined as a Graphic by Unicode. Such characters include letters, marks, numbers, punctuation, symbols, and spaces, from categories L, M, N, P, S, Zs.

## func **IsLetter**

```
func IsLetter(r rune) bool
```

IsLetter reports whether the rune is a letter (category L).

## func **IsLower**

```
func IsLower(r rune) bool
```

IsLower reports whether the rune is a lower case letter.

## func **IsMark**

```
func IsMark(r rune) bool
```

IsMark reports whether the rune is a mark character (category M).

## func **IsNumber**

```
func IsNumber(r rune) bool
```

IsNumber reports whether the rune is a number (category N).

## func **IsOneOf**

```
func IsOneOf(ranges []*RangeTable, r rune) bool
```

IsOneOf reports whether the rune is a member of one of the ranges. The function "In" provides a nicer signature and should be used in preference to IsOneOf.

## func **IsPrint**

```
func IsPrint(r rune) bool
```

IsPrint reports whether the rune is defined as printable by Go. Such characters include letters, marks, numbers, punctuation, symbols, and the ASCII space character, from categories L, M, N, P, S and the ASCII space character. This categorization is the same as IsGraphic except that the only spacing character is ASCII space, U+0020.

## func **IsPunct**

```
func IsPunct(r rune) bool
```

IsPunct reports whether the rune is a Unicode punctuation character (category P).

## func **IsSpace**

```
func IsSpace(r rune) bool
```

IsSpace reports whether the rune is a space character as defined by Unicode's White Space property; in the Latin-1 space this is

```
'\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+00A0 (NBSP).
```

Other definitions of spacing characters are set by category Z and property Pattern\_White\_Space.

## func `IsSymbol`

```
func IsSymbol(r rune) bool
```

`IsSymbol` reports whether the rune is a symbolic character.

## func `IsTitle`

```
func IsTitle(r rune) bool
```

`IsTitle` reports whether the rune is a title case letter.

## func `IsUpper`

```
func IsUpper(r rune) bool
```

`IsUpper` reports whether the rune is an upper case letter.

## func `SimpleFold`

```
func SimpleFold(r rune) rune
```

`SimpleFold` iterates over Unicode code points equivalent under the Unicode-defined simple case folding. Among the code points equivalent to `rune` (including `rune` itself), `SimpleFold` returns the smallest `rune > r` if one exists, or else the smallest `rune >= 0`. If `r` is not a valid Unicode code point, `SimpleFold(r)` returns `r`.

For example:

```
SimpleFold('A') = 'a'
SimpleFold('a') = 'A'

SimpleFold('K') = 'k'
SimpleFold('k') = '\u212A' (Kelvin symbol, K)
SimpleFold('\u212A') = 'K'

SimpleFold('1') = '1'

SimpleFold(-2) = -2
```

## func `To`

```
func To(_case int, r rune) rune
```

`To` maps the `rune` to the specified case: `UpperCase`, `LowerCase`, or `TitleCase`.

## func `ToLower`

```
func ToLower(r rune) rune
```

`ToLower` maps the rune to lower case.

## func `ToTitle`

```
func ToTitle(r rune) rune
```

`ToTitle` maps the rune to title case.

## func `ToUpper`

```
func ToUpper(r rune) rune
```

`ToUpper` maps the rune to upper case.

## type `CaseRange`

```
type CaseRange struct {
 Lo uint32
 Hi uint32
 Delta d
}
```

`CaseRange` represents a range of Unicode code points for simple (one code point to one code point) case conversion. The range runs from `Lo` to `Hi` inclusive, with a fixed stride of 1. `Delta`s are the number to add to the code point to reach the code point for a different case for that character. They may be negative. If zero, it means the character is in the corresponding case. There is a special case representing sequences of alternating corresponding Upper and Lower pairs. It appears with a fixed `Delta` of

```
{UpperLower, UpperLower, UpperLower}
```

The constant `UpperLower` has an otherwise impossible delta value.

## type `Range16`

```
type Range16 struct {
 Lo uint16
 Hi uint16
 Stride uint16
}
```

Range16 represents of a range of 16-bit Unicode code points. The range runs from Lo to Hi inclusive and has the specified stride.

## type Range32

```
type Range32 struct {
 Lo uint32
 Hi uint32
 Stride uint32
}
```

Range32 represents of a range of Unicode code points and is used when one or more of the values will not fit in 16 bits. The range runs from Lo to Hi inclusive and has the specified stride. Lo and Hi must always be  $\geq 1 \ll 16$ .

## type RangeTable

```
type RangeTable struct {
 R16 []Range16
 R32 []Range32
 LatinOffset int // number of entries in R16 with Hi <= MaxLatin1
}
```

RangeTable defines a set of Unicode code points by listing the ranges of code points within the set. The ranges are listed in two slices to save space: a slice of 16-bit ranges and a slice of 32-bit ranges. The two slices must be in sorted order and non-overlapping. Also, R32 should contain only values  $\geq 0x10000$  ( $1 \ll 16$ ).

## type SpecialCase

```
type SpecialCase []CaseRange
```

SpecialCase represents language-specific case mappings such as Turkish. Methods of SpecialCase customize (by overriding) the standard mappings.

```
var AzeriCase SpecialCase = _TurkishCase
```

```
var TurkishCase SpecialCase = _TurkishCase
```

## func (SpecialCase) ToLower

```
func (special SpecialCase) ToLower(r rune) rune
```

ToLower maps the rune to lower case giving priority to the special mapping.

## func (SpecialCase) ToTitle

```
func (special SpecialCase) ToTitle(r rune) rune
```

ToTitle maps the rune to title case giving priority to the special mapping.

## func (SpecialCase) **ToUpper**

```
func (special SpecialCase) ToUpper(r rune) rune
```

ToUpper maps the rune to upper case giving priority to the special mapping.

## BUGs

- There is no mechanism for full case folding, that is, for characters that involve multiple runes in the input or output.

# Package utf8

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package utf8 implements functions and constants to support text encoded in UTF-8. It includes functions to translate between runes and UTF-8 byte sequences. See <https://en.wikipedia.org/wiki/UTF-8>

## Constants

```
const (
 RuneError = '\uFFFD' // the "error" Rune or "Unicode replacement character"
 RuneSelf = 0x80 // characters below RuneSelf are represented as themselves
 MaxRune = '\U0010FFFF' // Maximum valid Unicode code point.
 UTFMax = 4 // maximum number of bytes of a UTF-8 encoded Unicode character
)
```

Numbers fundamental to the encoding.

## func `DecodeLastRune`

```
func DecodeLastRune(p []byte) (r rune, size int)
```

`DecodeLastRune` unpacks the last UTF-8 encoding in `p` and returns the rune and its width in bytes. If `p` is empty it returns (`RuneError`, 0). Otherwise, if the encoding is invalid, it returns (`RuneError`, 1). Both are impossible results for correct, non-empty UTF-8.

An encoding is invalid if it is incorrect UTF-8, encodes a rune that is out of range, or is not the shortest possible UTF-8 encoding for the value. No other validation is performed.

## func `DecodeLastRunesInString`

```
func DecodeLastRunesInString(s string) (r rune, size int)
```

`DecodeLastRunesInString` is like `DecodeLastRune` but its input is a string. If `s` is empty it returns (`RuneError`, 0). Otherwise, if the encoding is invalid, it returns (`RuneError`, 1). Both are impossible results for correct, non-empty UTF-8.

An encoding is invalid if it is incorrect UTF-8, encodes a rune that is out of range, or is not the shortest possible UTF-8 encoding for the value. No other validation is performed.

## func `DecodeRune`

```
func DecodeRune(p []byte) (r rune, size int)
```

`DecodeRune` unpacks the first UTF-8 encoding in `p` and returns the rune and its width in bytes. If `p` is empty it returns (`RuneError`, 0). Otherwise, if the encoding is invalid, it returns (`RuneError`, 1). Both are impossible results for correct, non-empty UTF-8.

An encoding is invalid if it is incorrect UTF-8, encodes a rune that is out of range, or is not the shortest possible UTF-8 encoding for the value. No other validation is performed.

## func `DecodeRuneInString`

```
func DecodeRuneInString(s string) (r rune, size int)
```

`DecodeRuneInString` is like `DecodeRune` but its input is a string. If `s` is empty it returns (`RuneError`, 0). Otherwise, if the encoding is invalid, it returns (`RuneError`, 1). Both are impossible results for correct, non-empty UTF-8.

An encoding is invalid if it is incorrect UTF-8, encodes a rune that is out of range, or is not the shortest possible UTF-8 encoding for the value. No other validation is performed.

## func `EncodeRune`

```
func EncodeRune(p []byte, r rune) int
```

`EncodeRune` writes into `p` (which must be large enough) the UTF-8 encoding of the rune. It returns the number of bytes written.

## func `FullRune`

```
func FullRune(p []byte) bool
```

`FullRune` reports whether the bytes in `p` begin with a full UTF-8 encoding of a rune. An invalid encoding is considered a full Rune since it will convert as a width-1 error rune.

## func `FullRuneInString`

```
func FullRuneInString(s string) bool
```

`FullRuneInString` is like `FullRune` but its input is a string.

## func `RuneCount`

```
func RuneCount(p []byte) int
```

RuneCount returns the number of runes in p. Erroneous and short encodings are treated as single runes of width 1 byte.

## func RuneCountInString

```
func RuneCountInString(s string) (n int)
```

RuneCountInString is like RuneCount but its input is a string.

## func RuneLen

```
func RuneLen(r rune) int
```

RuneLen returns the number of bytes required to encode the rune. It returns -1 if the rune is not a valid value to encode in UTF-8.

## func RuneStart

```
func RuneStart(b byte) bool
```

RuneStart reports whether the byte could be the first byte of an encoded, possibly invalid rune. Second and subsequent bytes always have the top two bits set to 10.

## func Valid

```
func Valid(p []byte) bool
```

Valid reports whether p consists entirely of valid UTF-8-encoded runes.

## func ValidRune

```
func ValidRune(r rune) bool
```

ValidRune reports whether r can be legally encoded as UTF-8. Code points that are out of range or a surrogate half are illegal.

## func ValidString

```
func ValidString(s string) bool
```

ValidString reports whether s consists entirely of valid UTF-8-encoded runes.

# Package unsafe

go1.15.2

Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package unsafe contains operations that step around the type safety of Go programs.

Packages that import unsafe may be non-portable and are not protected by the Go 1 compatibility guidelines.

### func Alignof

```
func Alignof(x ArbitraryType) uintptr
```

Alignof takes an expression x of any type and returns the required alignment of a hypothetical variable v as if v was declared via var v = x. It is the largest value m such that the address of v is always zero mod m. It is the same as the value returned by reflect.TypeOf(x).Align(). As a special case, if a variable s is of struct type and f is a field within that struct, then Alignof(s.f) will return the required alignment of a field of that type within a struct. This case is the same as the value returned by reflect.TypeOf(s.f).FieldAlign(). The return value of Alignof is a Go constant.

### func Offsetof

```
func Offsetof(x ArbitraryType) uintptr
```

Offsetof returns the offset within the struct of the field represented by x, which must be of the form structValue.field. In other words, it returns the number of bytes between the start of the struct and the start of the field. The return value of Offsetof is a Go constant.

### func Sizeof

```
func Sizeof(x ArbitraryType) uintptr
```

Sizeof takes an expression x of any type and returns the size in bytes of a hypothetical variable v as if v was declared via var v = x. The size does not include any memory possibly referenced by x. For instance, if x is a slice, Sizeof returns the size of the slice descriptor, not the size of the memory referenced by the slice. The return value of Sizeof is a Go constant.

### type ArbitraryType

```
type ArbitraryType int
```

ArbitraryType is here for the purposes of documentation only and is not actually part of the unsafe package. It represents the type of an arbitrary Go expression.

## type Pointer

```
type Pointer *ArbitraryType
```

Pointer represents a pointer to an arbitrary type. There are four special operations available for type Pointer that are not available for other types:

- A pointer value of any type can be converted to a Pointer.
- A Pointer can be converted to a pointer value of any type.
- A uintptr can be converted to a Pointer.
- A Pointer can be converted to a uintptr.

Pointer therefore allows a program to defeat the type system and read and write arbitrary memory. It should be used with extreme care.

The following patterns involving Pointer are valid. Code not using these patterns is likely to be invalid today or to become invalid in the future. Even the valid patterns below come with important caveats.

Running "go vet" can help find uses of Pointer that do not conform to these patterns, but silence from "go vet" is not a guarantee that the code is valid.

### (1) Conversion of a \*T1 to Pointer to \*T2.

Provided that T2 is no larger than T1 and that the two share an equivalent memory layout, this conversion allows reinterpreting data of one type as data of another type. An example is the implementation of math.Float64bits:

```
func Float64bits(f float64) uint64 {
 return *(*uint64)(unsafe.Pointer(&f))
}
```

### (2) Conversion of a Pointer to a uintptr (but not back to Pointer).

Converting a Pointer to a uintptr produces the memory address of the value pointed at, as an integer. The usual use for such a uintptr is to print it.

Conversion of a uintptr back to Pointer is not valid in general.

A uintptr is an integer, not a reference. Converting a Pointer to a uintptr creates an integer value with no pointer semantics. Even if a uintptr holds the address of some object, the garbage collector will not update that uintptr's value if the object moves, nor will that uintptr keep the object from being reclaimed.

The remaining patterns enumerate the only valid conversions from uintptr to Pointer.

### (3) Conversion of a Pointer to a uintptr and back, with arithmetic.

If `p` points into an allocated object, it can be advanced through the object by conversion to `uintptr`, addition of an offset, and conversion back to `Pointer`.

```
p = unsafe.Pointer(uintptr(p) + offset)
```

The most common use of this pattern is to access fields in a struct or elements of an array:

```
// equivalent to f := unsafe.Pointer(&s.f)
f := unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f))

// equivalent to e := unsafe.Pointer(&x[i])
e := unsafe.Pointer(uintptr(unsafe.Pointer(&x[0])) + i*unsafe.Sizeof(x[0]))
```

It is valid both to add and to subtract offsets from a pointer in this way. It is also valid to use `&^` to round pointers, usually for alignment. In all cases, the result must continue to point into the original allocated object.

Unlike in C, it is not valid to advance a pointer just beyond the end of its original allocation:

```
// INVALID: end points outside allocated space.
var s thing
end = unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Sizeof(s))

// INVALID: end points outside allocated space.
b := make([]byte, n)
end = unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(n))
```

Note that both conversions must appear in the same expression, with only the intervening arithmetic between them:

```
// INVALID: uintptr cannot be stored in variable
// before conversion back to Pointer.
u := uintptr(p)
p = unsafe.Pointer(u + offset)
```

Note that the pointer must point into an allocated object, so it may not be nil.

```
// INVALID: conversion of nil pointer
u := unsafe.Pointer(nil)
p := unsafe.Pointer(uintptr(u) + offset)
```

### (4) Conversion of a Pointer to a uintptr when calling `syscall.Syscall`.

The `Syscall` functions in package `syscall` pass their `uintptr` arguments directly to the operating system, which then may, depending on the details of the call, reinterpret some of them as pointers. That is, the system call implementation is implicitly converting certain arguments back from `uintptr` to pointer.

If a pointer argument must be converted to `uintptr` for use as an argument, that conversion must appear in the call expression itself:

```
syscall.Syscall(SYS_READ, uintptr(fd), uintptr(unsafe.Pointer(p)), uintptr(n))
```

The compiler handles a `Pointer` converted to a `uintptr` in the argument list of a call to a function implemented in assembly by arranging that the referenced allocated object, if any, is retained and not moved until the call completes, even though from the types alone it would appear that the object is no longer needed during the call.

For the compiler to recognize this pattern, the conversion must appear in the argument list:

```
// INVALID: uintptr cannot be stored in variable
// before implicit conversion back to Pointer during system call.
u := uintptr(unsafe.Pointer(p))
syscall.Syscall(SYS_READ, uintptr(fd), u, uintptr(n))
```

(5) Conversion of the result of `reflect.Value.Pointer` or `reflect.Value.UnsafeAddr` from `uintptr` to `Pointer`.

Package `reflect`'s `Value` methods named `Pointer` and `UnsafeAddr` return type `uintptr` instead of `unsafe.Pointer` to keep callers from changing the result to an arbitrary type without first importing `"unsafe"`. However, this means that the result is fragile and must be converted to `Pointer` immediately after making the call, in the same expression:

```
p := (*int)(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))
```

As in the cases above, it is invalid to store the result before the conversion:

```
// INVALID: uintptr cannot be stored in variable
// before conversion back to Pointer.
u := reflect.ValueOf(new(int)).Pointer()
p := (*int)(unsafe.Pointer(u))
```

(6) Conversion of a `reflect.SliceHeader` or `reflect.StringHeader` Data field to or from `Pointer`.

As in the previous case, the `reflect` data structures `SliceHeader` and `StringHeader` declare the field `Data` as a `uintptr` to keep callers from changing the result to an arbitrary type without first importing `"unsafe"`. However, this means that `SliceHeader` and `StringHeader` are only valid when interpreting the content of an actual slice or string value.

```
var s string
hdr := (*reflect.StringHeader)(unsafe.Pointer(&s)) // case 1
hdr.Data = uintptr(unsafe.Pointer(p)) // case 6 (this case)
hdr.Len = n
```

In this usage `hdr.Data` is really an alternate way to refer to the underlying pointer in the string header, not a `uintptr` variable itself.

In general, `reflect.SliceHeader` and `reflect.StringHeader` should be used only as `*reflect.SliceHeader` and `*reflect.StringHeader` pointing at actual slices or strings, never as plain structs. A program should not declare or allocate variables of these struct types.

```
// INVALID: a directly-declared header will not hold Data as a reference.
var hdr reflect.StringHeader
hdr.Data = uintptr(unsafe.Pointer(p))
hdr.Len = n
s := *(*string)(unsafe.Pointer(&hdr)) // p possibly already lost
```

# Package xml

go1.15.2 Latest

Published: Sep 9, 2020 | License: [BSD-3-Clause](#) | [Standard library](#)[Doc](#) [Overview](#) [Subdirectories](#) [Versions](#) [Imports](#) [Imported By](#) [Licenses](#)

## Overview

Package `xml` implements a simple XML 1.0 parser that understands XML name spaces.

## Constants

```
const (
 // Header is a generic XML header suitable for use with the output of Marshal.
 // This is not automatically added to any output of this package,
 // it is provided as a convenience.
 Header = `<?xml version="1.0" encoding="UTF-8"?>` + "\n"
)
```

## Variables

```
var HTMLAutoClose []string = htmlAutoClose
```

`HTMLAutoClose` is the set of HTML elements that should be considered to close automatically.

See the `Decoder.Strict` and `Decoder.Entity` fields' documentation.

```
var HTMLEntity map[string]string = htmlEntity
```

`HTMLEntity` is an entity map containing translations for the standard HTML entity characters.

See the `Decoder.Strict` and `Decoder.Entity` fields' documentation.

## func `Escape`

```
func Escape(w io.Writer, s []byte)
```

`Escape` is like `EscapeText` but omits the error return value. It is provided for backwards compatibility with Go 1.0. Code targeting Go 1.1 or later should use `EscapeText`.

## func `EscapeText`

```
func EscapeText(w io.Writer, s []byte) error
```

`EscapeText` writes to `w` the properly escaped XML equivalent of the plain text data `s`.

## func Marshal

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal returns the XML encoding of v.

Marshal handles an array or slice by marshaling each of the elements. Marshal handles a pointer by marshaling the value it points at or, if the pointer is nil, by writing nothing. Marshal handles an interface value by marshaling the value it contains or, if the interface value is nil, by writing nothing. Marshal handles all other data by writing one or more XML elements containing the data.

The name for the XML elements is taken from, in order of preference:

- the tag on the XMLName field, if the data is a struct
- the value of the XMLName field of type Name
- the tag of the struct field used to obtain the data
- the name of the struct field used to obtain the data
- the name of the marshaled type

The XML element for a struct contains marshaled elements for each of the exported fields of the struct, with these exceptions:

- the XMLName field, described above, is omitted.
- a field with tag "-" is omitted.
- a field with tag "name,attr" becomes an attribute with the given name in the XML element.
- a field with tag ",attr" becomes an attribute with the field name in the XML element.
- a field with tag ",chardata" is written as character data, not as an XML element.
- a field with tag ",cdata" is written as character data wrapped in one or more <![CDATA[ ... ]]> tags, not as an XML element.
- a field with tag ",innerxml" is written verbatim, not subject to the usual marshaling procedure.
- a field with tag ",comment" is written as an XML comment, not subject to the usual marshaling procedure. It must not contain the "--" string within it.
- a field with a tag including the "omitempty" option is omitted if the field value is empty. The empty values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero.
- an anonymous struct field is handled as if the fields of its value were part of the outer struct.
- a field implementing Marshaler is written by calling its MarshalXML method.
- a field implementing encoding.TextMarshaler is written by encoding the result of its MarshalText method as text.

If a field uses a tag "a>b>c", then the element c will be nested inside parent elements a and b. Fields that appear next to each other that name the same parent will be enclosed in one XML element.

If the XML name for a struct field is defined by both the field tag and the struct's XMLName field, the names must match.

See `MarshalIndent` for an example.

`Marshal` will return an error if asked to marshal a channel, function, or map.

## func `MarshalIndent`

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

`MarshalIndent` works like `Marshal`, but each XML element begins on a new indented line that starts with `prefix` and is followed by one or more copies of `indent` according to the nesting depth.

## func `Unmarshal`

```
func Unmarshal(data []byte, v interface{}) error
```

`Unmarshal` parses the XML-encoded data and stores the result in the value pointed to by `v`, which must be an arbitrary struct, slice, or string. Well-formed data that does not fit into `v` is discarded.

Because `Unmarshal` uses the `reflect` package, it can only assign to exported (upper case) fields. `Unmarshal` uses a case-sensitive comparison to match XML element names to tag values and struct field names.

`Unmarshal` maps an XML element to a struct using the following rules. In the rules, the tag of a field refers to the value associated with the key 'xml' in the struct field's tag (see the example above).

- \* If the struct has a field of type `[]byte` or `string` with tag `",innerxml"`, `Unmarshal` accumulates the raw XML nested inside the element in that field. The rest of the rules still apply.
- \* If the struct has a field named `XMLName` of type `Name`, `Unmarshal` records the element name in that field.
- \* If the `XMLName` field has an associated tag of the form `"name"` or `"namespace-URL name"`, the XML element must have the given name (and, optionally, name space) or else `Unmarshal` returns an error.
- \* If the XML element has an attribute whose name matches a struct field name with an associated tag containing `",attr"` or the explicit name in a struct field tag of the form `"name,attr"`, `Unmarshal` records the attribute value in that field.
- \* If the XML element has an attribute not handled by the previous rule and the struct has a field with an associated tag containing `",any,attr"`, `Unmarshal` records the attribute value in the first such field.

- \* If the XML element contains character data, that data is accumulated in the first struct field that has tag ",chardata". The struct field may have type []byte or string. If there is no such field, the character data is discarded.
- \* If the XML element contains comments, they are accumulated in the first struct field that has tag ",comment". The struct field may have type []byte or string. If there is no such field, the comments are discarded.
- \* If the XML element contains a sub-element whose name matches the prefix of a tag formatted as "a" or "a>b>c", unmarshal will descend into the XML structure looking for elements with the given names, and will map the innermost elements to that struct field. A tag starting with ">" is equivalent to one starting with the field name followed by ">".
- \* If the XML element contains a sub-element whose name matches a struct field's XMLName tag and the struct field has no explicit name tag as per the previous rule, unmarshal maps the sub-element to that struct field.
- \* If the XML element contains a sub-element whose name matches a field without any mode flags (",attr", ",chardata", etc), Unmarshal maps the sub-element to that struct field.
- \* If the XML element contains a sub-element that hasn't matched any of the above rules and the struct has a field with tag ",any", unmarshal maps the sub-element to that struct field.
- \* An anonymous struct field is handled as if the fields of its value were part of the outer struct.
- \* A struct field with tag "--" is never unmarshaled into.

If Unmarshal encounters a field type that implements the Unmarshaler interface, Unmarshal calls its UnmarshalXML method to produce the value from the XML element. Otherwise, if the value implements encoding.TextUnmarshaler, Unmarshal calls that value's UnmarshalText method.

Unmarshal maps an XML element to a string or []byte by saving the concatenation of that element's character data in the string or []byte. The saved []byte is never nil.

Unmarshal maps an attribute value to a string or []byte by saving the value in the string or slice.

Unmarshal maps an attribute value to an Attr by saving the attribute, including its name, in the Attr.

Unmarshal maps an XML element or attribute value to a slice by extending the length of the slice and mapping the element or attribute to the newly created value.

Unmarshal maps an XML element or attribute value to a bool by setting it to the boolean value represented by the string. Whitespace is trimmed and ignored.

Unmarshal maps an XML element or attribute value to an integer or floating-point field by setting the field to the result of interpreting the string value in decimal. There is no check for overflow. Whitespace is trimmed and ignored.

Unmarshal maps an XML element to a Name by recording the element name.

Unmarshal maps an XML element to a pointer by setting the pointer to a freshly allocated value and then mapping the element to that value.

A missing element or empty attribute value will be unmarshaled as a zero value. If the field is a slice, a zero value will be appended to the field. Otherwise, the field will be set to its zero value.

## type Attr

```
type Attr struct {
 Name Name
 Value string
}
```

An Attr represents an attribute in an XML element (Name=Value).

## type CharData

```
type CharData []byte
```

A CharData represents XML character data (raw text), in which XML escape sequences have been replaced by the characters they represent.

## func (CharData) Copy

```
func (c CharData) Copy() CharData
```

Copy creates a new copy of CharData.

## type Comment

```
type Comment []byte
```

A Comment represents an XML comment of the form <!--comment-->. The bytes do not include the <!-- and --> comment markers.

## func (Comment) Copy

```
func (c Comment) Copy() Comment
```

Copy creates a new copy of Comment.

## type Decoder

```
type Decoder struct {
 // Strict defaults to true, enforcing the requirements
 // of the XML specification.
 // If set to false, the parser allows input containing common
 // mistakes:
 // * If an element is missing an end tag, the parser invents
 // end tags as necessary to keep the return values from Token
 // properly balanced.
 // * In attribute values and character data, unknown or malformed
 // character entities (sequences beginning with &) are left alone.
 //
 // Setting:
 //
 // d.Strict = false
 // d.AutoClose = xml.HTMLAutoClose
 // d.Entity = xml.HTMLEntity
 //
 // creates a parser that can handle typical HTML.
 //
 // Strict mode does not enforce the requirements of the XML name spaces TR.
 // In particular it does not reject name space tags using undefined prefixes.
 // Such tags are recorded with the unknown prefix as the name space URL.
 Strict bool

 // When Strict == false, AutoClose indicates a set of elements to
 // consider closed immediately after they are opened, regardless
 // of whether an end element is present.
 AutoClose []string

 // Entity can be used to map non-standard entity names to string replacements.
 // The parser behaves as if these standard mappings are present in the map,
 // regardless of the actual map content:
 //
 // "lt": "<",
 // "gt": ">",
 // "amp": "&",
 // "apos": "'",
 // "quot": ``,
 Entity map[string]string

 // CharsetReader, if non-nil, defines a function to generate
 // charset-conversion readers, converting from the provided
 // non-UTF-8 charset into UTF-8. If CharsetReader is nil or
 // returns an error, parsing stops with an error. One of the
 // CharsetReader's result values must be non-nil.
 CharsetReader func(charset string, input io.Reader) (io.Reader, error)

 // DefaultSpace sets the default name space used for unadorned tags,
 // as if the entire XML stream were wrapped in an element containing
 // the attribute xmlns="DefaultSpace".
 DefaultSpace string
```

```
// contains filtered or unexported fields
}
```

A Decoder represents an XML parser reading a particular input stream. The parser assumes that its input is encoded in UTF-8.

## func **NewDecoder**

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder creates a new XML parser reading from r. If r does not implement [io.ByteReader](#), NewDecoder will do its own buffering.

## func **NewTokenDecoder**

```
func NewTokenDecoder(t TokenReader) *Decoder
```

NewTokenDecoder creates a new XML parser using an underlying token stream.

## func (\*Decoder) **Decode**

```
func (d *Decoder) Decode(v interface{}) error
```

Decode works like [Unmarshal](#), except it reads the decoder stream to find the start element.

## func (\*Decoder) **DecodeElement**

```
func (d *Decoder) DecodeElement(v interface{}, start *StartElement) error
```

DecodeElement works like [Unmarshal](#) except that it takes a pointer to the start XML element to decode into v. It is useful when a client reads some raw XML tokens itself but also wants to defer to [Unmarshal](#) for some elements.

## func (\*Decoder) **InputOffset**

```
func (d *Decoder) InputOffset() int64
```

InputOffset returns the input stream byte offset of the current decoder position. The offset gives the location of the end of the most recently returned token and the beginning of the next token.

## func (\*Decoder) **RawToken**

```
func (d *Decoder) RawToken() (Token, error)
```

RawToken is like Token but does not verify that start and end elements match and does not translate name space prefixes to their corresponding URLs.

## func (\*Decoder) Skip

```
func (d *Decoder) Skip() error
```

Skip reads tokens until it has consumed the end element matching the most recent start element already consumed. It recurs if it encounters a start element, so it can be used to skip nested structures. It returns nil if it finds an end element matching the start element; otherwise it returns an error describing the problem.

## func (\*Decoder) Token

```
func (d *Decoder) Token() (Token, error)
```

Token returns the next XML token in the input stream. At the end of the input stream, Token returns nil, io.EOF.

Slices of bytes in the returned token data refer to the parser's internal buffer and remain valid only until the next call to Token. To acquire a copy of the bytes, call CopyToken or the token's Copy method.

Token expands self-closing elements such as <br/> into separate start and end elements returned by successive calls.

Token guarantees that the StartElement and EndElement tokens it returns are properly nested and matched: if Token encounters an unexpected end element or EOF before all expected end elements, it will return an error.

Token implements XML name spaces as described by <https://www.w3.org/TR/REC-xml-names/>. Each of the Name structures contained in the Token has the Space set to the URL identifying its name space when known. If Token encounters an unrecognized name space prefix, it uses the prefix as the Space rather than report an error.

## type Directive

```
type Directive []byte
```

A Directive represents an XML directive of the form <!text>. The bytes do not include the <! and > markers.

## func (Directive) Copy

```
func (d Directive) Copy() Directive
```

Copy creates a new copy of Directive.

## type Encoder

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

An Encoder writes XML data to an output stream.

## func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that writes to w.

## func (\*Encoder) Encode

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode writes the XML encoding of v to the stream.

See the documentation for Marshal for details about the conversion of Go values to XML.

Encode calls Flush before returning.

## func (\*Encoder) EncodeElement

```
func (enc *Encoder) EncodeElement(v interface{}, start StartElement) error
```

EncodeElement writes the XML encoding of v to the stream, using start as the outermost tag in the encoding.

See the documentation for Marshal for details about the conversion of Go values to XML.

EncodeElement calls Flush before returning.

## func (\*Encoder) EncodeToken

```
func (enc *Encoder) EncodeToken(t Token) error
```

EncodeToken writes the given XML token to the stream. It returns an error if StartElement and EndElement tokens are not properly matched.

EncodeToken does not call Flush, because usually it is part of a larger operation such as Encode or EncodeElement (or a custom Marshaler's MarshalXML invoked during those), and those will call Flush when finished. Callers that create an Encoder and then invoke EncodeToken directly, without using Encode or EncodeElement, need to call Flush when finished to ensure that the XML is written to the underlying writer.

EncodeToken allows writing a ProInst with Target set to "xml" only as the first token in the stream.

## func (\*Encoder) Flush

```
func (enc *Encoder) Flush() error
```

Flush flushes any buffered XML to the underlying writer. See the EncodeToken documentation for details about when it is necessary.

## func (\*Encoder) Indent

```
func (enc *Encoder) Indent(prefix, indent string)
```

Indent sets the encoder to generate XML in which each element begins on a new indented line that starts with prefix and is followed by one or more copies of indent according to the nesting depth.

## type EndElement

```
type EndElement struct {
 Name Name
}
```

An EndElement represents an XML end element.

## type Marshaler

```
type Marshaler interface {
 MarshalXML(e *Encoder, start StartElement) error
}
```

Marshaler is the interface implemented by objects that can marshal themselves into valid XML elements.

MarshalXML encodes the receiver as zero or more XML elements. By convention, arrays or slices are typically encoded as a sequence of elements, one per entry. Using start as the element tag is not required, but doing so will enable Unmarshal to match the XML elements to the correct struct field. One common implementation strategy is to construct a separate value with a layout corresponding to the desired XML and then to encode it using e.EncodeElement. Another common strategy is to use repeated calls to e.EncodeToken to generate the XML output one token at a time. The sequence of encoded tokens must make up zero or more valid XML elements.

## type MarshalerAttr

```
type MarshalerAttr interface {
 MarshalXMLAttr(name Name) (Attr, error)
}
```

MarshalerAttr is the interface implemented by objects that can marshal themselves into valid XML attributes.

MarshalXMLAttr returns an XML attribute with the encoded value of the receiver. Using name as the attribute name is not required, but doing so will enable Unmarshal to match the attribute to the correct struct field. If MarshalXMLAttr returns the zero attribute Attr{}, no attribute will be generated in the output. MarshalXMLAttr is used only for struct fields with the "attr" option in the field tag.

## type Name

```
type Name struct {
 Space, Local string
}
```

A Name represents an XML name (Local) annotated with a name space identifier (Space). In tokens returned by Decoder.Token, the Space identifier is given as a canonical URL, not the short prefix used in the document being parsed.

## type ProInst

```
type ProInst struct {
 Target string
 Inst []byte
}
```

A ProInst represents an XML processing instruction of the form <?target inst?>

## func (ProInst) Copy

```
func (p ProInst) Copy() ProInst
```

Copy creates a new copy of ProInst.

## type StartElement

```
type StartElement struct {
 Name Name
 Attr []Attr
}
```

A StartElement represents an XML start element.

## func (StartElement) Copy

```
func (e StartElement) Copy() StartElement
```

Copy creates a new copy of StartElement.

## func (StartElement) End

```
func (e StartElement) End() EndElement
```

End returns the corresponding XML end element.

## type SyntaxError

```
type SyntaxError struct {
 Msg string
 Line int
}
```

A SyntaxError represents a syntax error in the XML input stream.

## func (\*SyntaxError) Error

```
func (e *SyntaxError) Error() string
```

## type TagPathError

```
type TagPathError struct {
 Struct reflect.Type
 Field1, Tag1 string
 Field2, Tag2 string
}
```

A TagPathError represents an error in the unmarshaling process caused by the use of field tags with conflicting paths.

## func (\*TagPathError) Error

```
func (e *TagPathError) Error() string
```

## type Token

```
type Token interface{}
```

A Token is an interface holding one of the token types: StartElement, EndElement, CharData, Comment, ProInst, or Directive.

## func CopyToken

```
func CopyToken(t Token) Token
```

CopyToken returns a copy of a Token.

## type TokenReader

```
type TokenReader interface {
 Token() (Token, error)
}
```

A TokenReader is anything that can decode a stream of XML tokens, including a Decoder.

When Token encounters an error or end-of-file condition after successfully reading a token, it returns the token. It may return the (non-nil) error from the same call or return the error (and a nil token) from a subsequent call. An instance of this general case is that a TokenReader returning a non-nil token at the end of the token stream may return either io.EOF or a nil error. The next Read should return nil, io.EOF.

Implementations of Token are discouraged from returning a nil token with a nil error. Callers should treat a return of nil, nil as indicating that nothing happened; in particular it does not indicate EOF.

## type UnmarshalError

```
type UnmarshalError string
```

An UnmarshalError represents an error in the unmarshaling process.

## func (UnmarshalError) Error

```
func (e UnmarshalError) Error() string
```

## type Unmarshaler

```
type Unmarshaler interface {
 UnmarshalXML(d *Decoder, start StartElement) error
}
```

Unmarshaler is the interface implemented by objects that can unmarshal an XML element description of themselves.

UnmarshalXML decodes a single XML element beginning with the given start element. If it returns an error, the outer call to Unmarshal stops and returns that error. UnmarshalXML must consume exactly one XML element. One common implementation strategy is to unmarshal into a separate value with a layout matching the expected XML using d.DecodeElement, and then to copy the data from that value into the receiver. Another common strategy is to use d.Token to process the XML object one token at a time. UnmarshalXML may not use d.RawToken.

## type UnmarshalerAttr

```
type UnmarshalerAttr interface {
 UnmarshalXMLAttr(attr Attr) error
}
```

```
}
```

UnmarshalerAttr is the interface implemented by objects that can unmarshal an XML attribute description of themselves.

UnmarshalXMLAttr decodes a single XML attribute. If it returns an error, the outer call to Unmarshal stops and returns that error. UnmarshalXMLAttr is used only for struct fields with the "attr" option in the field tag.

## type **UnsupportedTypeError**

```
type UnsupportedTypeError struct {
 Type reflect.Type
}
```

UnsupportedTypeError is returned when Marshal encounters a type that cannot be converted into XML.

## func (\*UnsupportedTypeError) **Error**

```
func (e *UnsupportedTypeError) Error() string
```

## BUGs

- Mapping between XML elements and data structures is inherently flawed: an XML element is an order-dependent collection of anonymous values, while a data structure is an order-independent collection of named values. See package json for a textual representation more suitable to data structures.