URADICAL

POCKET REFERENCE

Go's Standard Library

Alan Bradley

Go Standard Library Pocket Reference

A curated reference for application developers

Alan Bradley

uradical.io

Go Standard Library Pocket Reference

Alan Bradley

Table of Contents

database/sql

embed

encoding/base32

encoding/base64

encoding/binary

encoding/csv

encoding/gob

encoding/hex

encoding/json

encoding/pem

encoding/xml

errors

expvar

flag

fmt

hash/crc32

hash/fnv

html

html/template

io

io/fs

log

log/slog

maps

math

math/big

math/bits

math/rand/v2

mime

mime/multipart

net

net/http

net/http/httptest

net/http/pprof

net/smtp

net/url

os

os/exec

os/signal

os/user

path

path/filepath

regexp

runtime

runtime/debug

slices

sort

strconv

strings

sync

sync/atomic

testing

text/scanner

text/tabwriter

text/template

time

unicode

unicode/utf8

unicode/utf16

About this book

The Go standard library is one of the most complete and coherent standard libraries in any programming language. It provides production-capable HTTP servers, cryptographic primitives, structured logging, compression, archiving, SQL database access, and much more — all without a single third-party dependency.

This reference covers 79 packages selected for their relevance to application developers: the well-known packages treated with the depth they deserve, and the lesser-known packages surfaced with enough context to make them discoverable. Compiler internals, image processing, and platform-specific low-level packages are excluded; everything here is something a working Go developer building services, tools, or CLIs will plausibly reach for.

Each entry follows a consistent structure:

- A description that explains what the package is for and how it thinks about the problem — not a restatement of the godoc summary
- A types table covering the types you will actually work with
- A functions and methods section that explains relationships and non-obvious choices, not an inventory of signatures
- A realistic example drawn from actual application code
- A single gotcha — the one thing that bites developers, with the consequence explained

Entries are ordered alphabetically by import path. Cross-references appear in the *See also* section of each entry.

This book is a companion to the official documentation at pkg.go.dev, not a replacement for it. Use it for orientation, for the gotchas, and for the editorial judgements about what matters. Use godoc for complete API signatures and edge cases.

Excluded packages

The following packages were considered and deliberately excluded:

Compiler and toolchain internals — `go/ast`, `go/parser`, `go/token`, `go/types`, `debug/dwarf`, `debug/elf`, and related packages. Relevant only when building Go tools.

Deprecated packages — `crypto/dsa`, `crypto/rc4`. Formally deprecated in the standard library and should not be used in new code.

Platform-restricted packages — `plugin` (Linux and macOS only, with significant constraints), `syscall` (replaced by `golang.org/x/sys` for portable low-level system access).

Image processing — `image`, `image/color`, `image/draw`, `image/jpeg`, `image/png`, `image/gif`. Specialist enough to warrant their own treatment.

Network RPC — `net/rpc`. Effectively superseded by gRPC and other modern RPC frameworks.

Go version: this reference covers the standard library as of Go 1.22, with notes on features introduced in Go 1.21 and Go 1.22 where relevant.

archive/tar

Archive

The `archive/tar` package reads and writes tar archives as a stream. You work entry by entry: write a header, write the body, move to the next. There is no in-memory representation of the whole archive — which means you can produce or consume archives of arbitrary size without buffering them. The package pairs naturally with `compress/gzip` by composing over `io.Writer` and `io.Reader`; wrapping one in the other is all that is needed for `.tar.gz`.

Types

Type	Purpose
Reader	Reads entries sequentially from a tar stream
Writer	Writes entries sequentially to a tar stream
Header	Metadata for a single entry: name, size, mode, modification time

Functions and methods

The core read loop uses `(*Reader).Next()` to advance through entries, checking for `io.EOF` to detect the end of the archive. Between calls to `Next`, the `Reader` itself acts as an `io.Reader` for the entry's body — pass it directly to `io.Copy`.

On the write side, every entry requires a `WriteHeader` call followed by the body bytes. `tar.FileInfoHeader` constructs a `Header` from an `os.FileInfo`, which saves filling out fields manually. Always call `(*Writer).Close()` explicitly to write the end-of-archive marker before closing the underlying writer.

Example

```
func archive(dst io.Writer, paths []string) error {
    gw := gzip.NewWriter(dst)
    tw := tar.NewWriter(gw)

    for _, p := range paths {
        f, err := os.Open(p)
        if err != nil {
            return err
        }
        info, _ := f.Stat()
        hdr, _ := tar.FileInfoHeader(info, "")
        hdr.Name = filepath.Base(p)
        tw.WriteHeader(hdr)
        io.Copy(tw, f)
        f.Close()
    }

    // Order matters: tar must close before gzip
    if err := tw.Close(); err != nil {
        return err
    }
    return gw.Close()
}
```

Gotcha

Close the tar writer before the gzip writer — not the other way around, and not with defers that execute in reverse. Closing gzip first produces a structurally valid compressed stream with a truncated tar archive inside it. The file opens without error and fails silently when you try to extract entries.

See also

`archive/zip` · `compress/gzip` · `io/fs`

archive/zip

Archive

The `archive/zip` package reads and writes ZIP archives. Reading requires random access — ZIP's central directory lives at the end of the file — so `NewReader` takes an `io.ReaderAt` and a size rather than a plain `io.Reader`. Writing is sequential. What makes this package more broadly useful than its name suggests: ZIP is the container format for EPUB, DOCX, XLSX, JAR, and APK files. Code that needs to inspect or produce any of those formats is using this package, whether it knows it or not.

Types

Type	Purpose
Reader	Opens an existing archive; exposes entries as a []*File slice
Writer	Writes a new archive sequentially
File	A single entry in a Reader; call Open() to read the body
FileHeader	Metadata for an entry: name, compression method, timestamps

Functions and methods

`NewReader` takes an `io.ReaderAt` — an `*os.File` satisfies this directly. The resulting `Reader.File` slice gives you all entries upfront, which you can filter or seek through freely before opening any bodies.

On the write side, `(*Writer).Create(name)` is the fast path — deflate compression, current timestamp, no further configuration. When you need control over compression method or timestamps, use `CreateHeader` with a populated `FileHeader`. For already-compressed

content — images, video, existing archives — set `Method: zip.Store` to skip wasted compression work.

```
(*File).Open() (io.ReadCloser, error)
```

Opens the entry body. Always close the returned `ReadCloser` before opening another entry.

Example

```
// Unpack a DOCX and read its content
func readDocxBody(path string) ([]byte, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    info, _ := f.Stat()
    r, err := zip.NewReader(f, info.Size())
    if err != nil {
        return nil, err
    }

    for _, entry := range r.File {
        if entry.Name == "word/document.xml" {
            rc, err := entry.Open()
            if err != nil {
                return nil, err
            }
            defer rc.Close()
            return io.ReadAll(rc)
        }
    }
    return nil, errors.New("document.xml not found")
}
```

Gotcha

`(*Writer).Create` uses deflate compression by default. Storing already-compressed content through deflate wastes CPU and can

produce output that is marginally larger than the input. Use `CreateHeader` with `Method: zip.Store` for any entry you know is already compressed.

See also

`archive/tar` · `io/fs` · `encoding/binary`

bufio

I/O

The `bufio` package wraps `io.Reader` and `io.Writer` with an in-memory buffer, reducing system call frequency when reading or writing in small increments. Its `Scanner` is the standard way to process text input token by token — lines from a log file, words from stdin, records from a pipe — without loading the whole input into memory. The `Writer` matters equally when producing large volumes of output: buffering accumulates many small writes into fewer, larger flushes, which makes a measurable difference on any output that would otherwise make a syscall per line.

Types

Type	Purpose
Reader	Buffered reader with peek and single-byte read capabilities
Writer	Buffered writer; must be flushed before the underlying writer closes
Scanner	Tokenises input by lines, words, runes, or a custom split function

Functions and methods

`Scanner` is the most-reached-for type in this package. The default split function is `ScanLines`, which strips the newline and returns the text. Change it with `Split` before the first call to `Scan` — `ScanWords` and `ScanRunes` cover most other cases; a custom `SplitFunc` handles anything else. Use `Bytes()` instead of `Text()` in hot paths to avoid the string allocation.

The `Writer` has one rule that trips people: always call `Flush` before the underlying writer is closed. A deferred `bw.Flush()` on the enclosing

function is the right pattern — put it immediately after `NewWriter`.

Example

```
// Process a TSV from stdin, emit matching rows
func filterTSV(w io.Writer, r io.Reader, col int, match
string) error {
    bw := bufio.NewWriter(w)
    defer bw.Flush()

    scanner := bufio.NewScanner(r)
    for scanner.Scan() {
        fields := strings.Split(scanner.Text(), "\t")
        if col < len(fields) && fields[col] == match {
            fmt.Fprintln(bw, scanner.Text())
        }
    }
    return scanner.Err()
}
```

Gotcha

`Scanner` caps token size at 64KB by default. Files with long lines — minified JSON, base64 blobs, some log formats — will fail with `bufio.ErrTooLong`. Call `(*Scanner).Buffer(make([]byte, 0, 1<<20), 1<<20)` before the first `Scan` to raise the limit. This is not a resize after failure; it must be set before scanning begins.

See also

`io` · `os` · `strings`

bytes

Data

The `bytes` package offers the same operations on byte slices that the `strings` package offers on strings — search, split, replace, trim, compare — and adds `Buffer`, a growable byte buffer that satisfies both `io.Reader` and `io.Writer`. `Buffer` is the right choice when building binary output incrementally: protocol frames, serialised records, anything where you are assembling heterogeneous bytes before writing them out. For pure string construction without binary data, `strings.Builder` is cheaper because it avoids the `[]byte` backing.

Types

Type	Purpose
Buffer	Growable read/write buffer implementing io.Reader and io.Writer
Reader	Read-only view of a byte slice as an io.ReadSeeker and io.ReaderAt

Functions and methods

Most of the package-level functions — `Contains`, `Split`, `TrimSpace`, `Replace`, `Equal` — mirror their `strings` counterparts and behave identically. The one worth calling out is `Equal`: use it instead of `string(a) == string(b)` when comparing byte slices, because the conversion allocates.

`Buffer` is the main event. Write to it with `Write`, `WriteString`, `WriteByte`, or `WriteRune`; read from it with `Read` or drain it all with `Bytes()`. Passing a `*Buffer` to any function that expects an `io.Writer` — `fmt.Fprintf`, `json.NewEncoder`, `binary.Write` — works directly.

Example

```
// Build a length-prefixed binary frame
func encodeMessage(typ uint8, payload []byte) []byte {
    var buf bytes.Buffer
    buf.WriteByte(typ)
    binary.Write(&buf, binary.BigEndian, uint32(len(payload)))
    buf.Write(payload)
    return buf.Bytes()
}

// Decode a simple text protocol line
func parseFields(line []byte) [][]byte {
    return bytes.Fields(line) // splits on any whitespace,
trims leading/trailing
}
```

Gotcha

`(*Buffer).Bytes()` returns a slice into the buffer's internal array — not a copy. Writing to the buffer after calling `Bytes()` may invalidate that slice if the internal array is reallocated. Capture the result, copy it if you need it to outlive the next write, then proceed.

See also

`strings` · `io` · `encoding/binary`

cmp

Utilities

The `cmp` package, added in Go 1.21, provides ordered comparison for any type that supports the < operator. The central function is `Compare`, which returns -1, 0, or 1 — exactly the signature expected by `slices.SortFunc`, `slices.BinarySearchFunc`, and any code doing three-way comparison. Before `cmp`, every sort closure contained hand-rolled less-than logic. The package also provides `Or`, a small utility that returns the first non-zero value from a variadic list — useful for configuration fallback chains without if-chains.

Types

Type	Purpose
Ordered	Type constraint covering all integer, float, and string types

Functions and methods

```
Compare[T Ordered](x, y T) int
```

Returns -1, 0, or 1. Handles float NaN consistently: NaN is considered less than any non-NaN value.

```
Or[T comparable](vals ...T) T
```

Returns the first value in `vals` that is not the zero value for its type.

Example

```
// Multi-key sort: last name, then first name
slices.SortFunc(users, func(a, b User) int {
    if n := cmp.Compare(a.LastName, b.LastName); n != 0 {
        return n
    }
```

```
    return cmp.Compare(a.FirstName, b.FirstName)
})

// Config fallback: env var . config file . default
addr := cmp.Or(os.Getenv("LISTEN_ADDR"), cfg.ListenAddr,
":8080")
```

Gotcha

`Or` treats the zero value as absent — for strings that means `""`, for integers `0`, for pointers `nil`. If an empty string or zero is a legitimate intended value in your fallback chain, `Or` will skip past it silently. In those cases an explicit `if` is clearer.

See also

`slices` · `sort` · `maps`

compress/flate

Compression

The `compress/flate` package implements the DEFLATE compression format defined in RFC 1951. Most developers never use it directly — `compress/gzip` and `compress/zlib` both build on top of it and are the right entry points for those formats. Direct use of `flate` arises when you are implementing a protocol or file format that specifies raw DEFLATE without a wrapper, or when you need precise control over compression level that the higher-level packages don't expose. If you are compressing data for general use, start with `compress/gzip`.

Types

Type	Purpose
Writer	Compresses data written to it, flushing to an underlying io.Writer
Reader	Decompresses a raw DEFLATE stream; returned by NewReader

Functions and methods

```
NewWriter(w io.Writer, level int) (*Writer, error)
```

Creates a compressor. `level` runs from `BestSpeed` (1) to `BestCompression` (9), with `DefaultCompression` (-1) as a reasonable middle ground. `HuffmanOnly` disables Lempel-Ziv matching for input that is already partly compressed.

```
NewReader(r io.Reader) io.ReadCloser
```

Creates a decompressor for raw DEFLATE. Returns an `io.ReadCloser` — always close it to release internal state.

```
(*Writer).Flush() error
```

Forces buffered data to the underlying writer without closing the stream. Needed in streaming protocols where the reader must receive data before the stream ends.

Example

```
// Raw DEFLATE for a custom binary protocol
func compress(w io.Writer, data []byte) error {
    fw, err := flate.NewWriter(w, flate.BestSpeed)
    if err != nil {
        return err
    }
    if _, err := fw.Write(data); err != nil {
        return err
    }
    return fw.Close()
}

func decompress(r io.Reader) ([]byte, error) {
    fr := flate.NewReader(r)
    defer fr.Close()
    return io.ReadAll(fr)
}
```

Gotcha

`NewReader` accepts an `io.Reader`, not an `io.ReaderAt`. If your DEFLATE stream is embedded at an offset within a larger file, you need to wrap the file in an `io.SectionReader` first — passing the raw file will decompress from position zero regardless of where your data begins.

See also

`compress/gzip` · `compress/zlib` · `io`

compress/gzip

Compression

The `compress/gzip` package reads and writes gzip-compressed data as defined in RFC 1952. It wraps `compress/flate` and adds the gzip header and checksum. Because it operates over `io.Reader` and `io.Writer`, it composes with anything in the standard library that uses those interfaces — wrap a file, wrap an HTTP response body, wrap a network connection. The `Header` embedded in both `Reader` and `Writer` carries the original filename, modification time, and OS byte, which matters when producing archives that preserve file metadata.

Types

Type	Purpose
Reader	Decompresses a gzip stream; embeds Header for metadata access
Writer	Compresses data written to it; embeds Header for metadata setting
Header	Gzip header fields: Name, Comment, ModTime, OS

Functions and methods

`NewWriter` and `NewReader` are the entry points. Both wrap any `io.Writer` or `io.Reader` respectively. For compression level control, use `NewWriterLevel` — the levels are the same as `compress/flate`.

The most commonly missed method is `(*Reader).Reset(r io.Reader)`, which repositions an existing `Reader` onto a new stream without allocating a new one. When decompressing many small gzip streams — HTTP responses in a loop, for example — reset rather than creating a new reader each time.

```
(*Writer).Close() error
```

Flushes compressed data and writes the gzip footer checksum. A writer that is not closed produces a truncated stream that decompressors will reject.

Example

```
// Compress JSON responses transparently
func gzipMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        if !strings.Contains(r.Header.Get("Accept-Encoding"),
"gzip") {
            next.ServeHTTP(w, r)
            return
        }
        w.Header().Set("Content-Encoding", "gzip")
        gz := gzip.NewWriter(w)
        defer gz.Close()
        next.ServeHTTP(gzipResponseWriter{Writer: gz,
ResponseWriter: w}, r)
    })
}

type gzipResponseWriter struct {
    io.Writer
    http.ResponseWriter
}

func (g gzipResponseWriter) Write(b []byte) (int, error) {
    return g.Writer.Write(b)
}
```

Gotcha

`(*Writer).Close()` must be called explicitly — it writes the gzip checksum footer. If you only `defer gz.Close()` inside a function that also returns an error, and the error path returns before Close runs, the stream is invalid. Close explicitly on the success path; let the defer handle the error path as a safety net.

See also

`compress/flate` · `archive/tar` · `io`

compress/zlib

Compression

The `compress/zlib` package implements the zlib format defined in RFC 1950 — essentially DEFLATE with a two-byte header and an Adler-32 checksum. It is not a general-purpose compression choice; it appears in specific contexts where the format is mandated: PNG image data, PDF streams, and some network protocols. If you are not working with one of those formats, use `compress/gzip` instead — it offers the same compression with broader tooling support and OS-level interoperability.

Types

Type	Purpose
Reader	Decompresses a zlib stream
Writer	Compresses data into zlib format

Functions and methods

The API mirrors `compress/gzip` closely. `NewReader` wraps an `io.Reader`; `NewWriter` and `NewWriterLevel` wrap an `io.Writer`. The distinction from gzip is the checksum algorithm — Adler-32 rather than CRC-32 — and the absence of a filename or metadata header.

```
NewReader(r io.Reader) (io.ReadCloser, error)
```

Returns an error immediately if the zlib header is malformed, unlike `gzip.NewReader` which defers header errors to the first read.

Example

```
// Decompress a PNG IDAT chunk's zlib payload
func decompressIDAT(compressed []byte) ([]byte, error) {
    r, err := zlib.NewReader(bytes.NewReader(compressed))
    if err != nil {
```

```
        return nil, fmt.Errorf("invalid zlib header: %w", err)
    }
    defer r.Close()
    return io.ReadAll(r)
}
```

Gotcha

`zlib.NewReader` returns an error on a malformed header, but `gzip.NewReader` does not — it defers header validation to the first `Read`. If you are switching between the two, do not assume the same error-handling pattern applies to both.

See also

`compress/gzip` · `compress/flate` · `io`

context

Concurrency

The `context` package defines a single interface — `Context` — that carries a deadline, a cancellation signal, and a map of request-scoped values across API boundaries. It is the standard mechanism for propagating cancellation in Go: when a request is abandoned, a timeout is exceeded, or a shutdown signal is received, the cancellation flows through every goroutine participating in that request via its context. Almost every function that does I/O, makes network calls, or runs for an indeterminate time should accept a `context.Context` as its first parameter.

Types

Type	Purpose
Context	Interface carrying deadline, cancellation, and values
CancelFunc	Cancels the context and releases its resources
CancelCauseFunc	Cancels with an explicit error accessible via Cause

Functions and methods

`context.Background()` is the root context for any program. Pass it into the top of a call chain — an HTTP handler, a main goroutine, a test function — and derive child contexts from it as the call descends.

The three derivation functions each add something:

- `WithCancel` — returns a child that can be cancelled explicitly via `CancelFunc`

- `WithTimeout` — returns a child that cancels automatically after a duration
- `WithDeadline` — returns a child that cancels at a specific `time.Time`

Always call the returned `CancelFunc`, even when the operation succeeds. Failing to do so leaks the child context and any resources it holds until the parent is cancelled.

`WithValue` stores a request-scoped value — user ID, trace ID, authenticated principal. Use an unexported package-level key type to avoid collisions with other packages storing values in the same context.

```
ctx.Done() <-chan struct{}
```

Returns a channel that is closed when the context is cancelled or times out. Select on this channel alongside work channels to implement cancellation-aware loops.

```
ctx.Err() error
```

Returns `context.Canceled` or `context.DeadlineExceeded` after the context is done. Use `errors.Is` to distinguish them.

Example

```
type ctxKey struct{}

func processRequest(ctx context.Context, userID string) error
{
    ctx = context.WithValue(ctx, ctxKey{}, userID)
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

    result := make(chan error, 1)
    go func() { result <- callDownstream(ctx) }()

    select {
    case err := <-result:
        return err
```

```
    case <-ctx.Done():
        return fmt.Errorf("request aborted: %w", ctx.Err())
    }
}

func callDownstream(ctx context.Context) error {
    uid := ctx.Value(ctxKey{}).(string)
    // uid available throughout the call chain without
threading it as a parameter
    _ = uid
    return nil
}
```

Gotcha

Do not store contexts in structs. A context belongs to a call, not to an object — storing it on a struct makes its lifetime ambiguous and breaks cancellation semantics. If a method needs a context, it takes one as a parameter. The one exception is when implementing an interface you do not control that has no context parameter, in which case a stored context is unavoidable and should be documented as such.

See also

`sync` · `time` · `errors`

crypto/aes

Cryptography

The `crypto/aes` package implements the AES block cipher. It produces a `cipher.Block` — a low-level primitive that encrypts exactly one 16-byte block at a time. You do not use it directly for data encryption; you wrap it with a mode from `crypto/cipher`. The only decision this package requires is key length: 16 bytes for AES-128, 24 for AES-192, 32 for AES-256. AES-128 is faster on hardware without AES-NI; AES-256 is the default choice when key length is not constrained.

Types

Type	Purpose
Block	The cipher.Block interface — Encrypt and Decrypt single blocks

Functions and methods

```
NewCipher(key []byte) (cipher.Block, error)
```

The only function in the package. Returns an error only if the key length is not 16, 24, or 32 bytes. In practice, validate key length before this call so the error is impossible at runtime.

Example

```
// AES-GCM: authenticated encryption with associated data
func encrypt(key, plaintext, additionalData []byte) ([]byte,
error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }
    gcm, err := cipher.NewGCM(block)
    if err != nil {
```

```
        return nil, err
    }
    nonce := make([]byte, gcm.NonceSize())
    if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
        return nil, err
    }
    // Seal appends ciphertext to nonce so both travel
together
    return gcm.Seal(nonce, nonce, plaintext, additionalData),
nil
}

func decrypt(key, ciphertext, additionalData []byte) ([]byte,
error) {
    block, _ := aes.NewCipher(key)
    gcm, _ := cipher.NewGCM(block)
    nonceSize := gcm.NonceSize()
    if len(ciphertext) < nonceSize {
        return nil, errors.New("ciphertext too short")
    }
    nonce, ct := ciphertext[:nonceSize],
ciphertext[nonceSize:]
    return gcm.Open(nil, nonce, ct, additionalData)
}
```

Gotcha

AES-GCM authentication tags cover both the ciphertext and any additional data you pass. If you encrypt with additional data and decrypt without it — or vice versa — decryption fails with an authentication error. This is correct behaviour, not a bug, but it catches developers who add additional data in one place and forget to thread it through everywhere decryption occurs.

See also

`crypto/cipher` · `crypto/rand` · `crypto/hmac`

crypto/cipher

Cryptography

The `crypto/cipher` package provides block cipher modes — the layer between a raw block cipher like AES and usable symmetric encryption. The mode determines how the cipher processes data larger than a single block and what security properties the result has. GCM is the right default: it provides authenticated encryption, meaning tampering with the ciphertext is detected on decryption. CBC and CTR are available for interoperability with systems that require them but neither provides authentication on its own — you need a separate HMAC if you use them.

Types

Type	Purpose
AEAD	Authenticated encryption interface: Seal and Open
Block	Single-block cipher interface, provided by packages like crypto/aes
BlockMode	CBC-style mode operating on full blocks
Stream	CTR/OFB-style mode operating as a keystream

Functions and methods

`NewGCM` is the function most code reaches for. It wraps a `cipher.Block` and returns an `AEAD` — the authenticated encryption interface whose two methods, `Seal` and `Open`, handle nonce, encryption, and authentication tag in one call.

```
NewGCM(block Block) (AEAD, error)
```

Returns a GCM wrapper with a 12-byte nonce and 16-byte tag. Use `NewGCMWithNonceSize` only when interoperating with a system that uses a non-standard nonce length.

```
(AEAD).Seal(dst, nonce, plaintext, additionalData []byte)
[]byte
```

Encrypts and authenticates. Appends the result to `dst` — pass `nonce` as `dst` to prepend the nonce to the ciphertext in one allocation.

```
(AEAD).Open(dst, nonce, ciphertext, additionalData []byte)
([]byte, error)
```

Decrypts and verifies. Returns an error if the authentication tag does not match — do not use the returned plaintext if this errors.

```
NewCBCEncrypter(block Block, iv []byte) BlockMode
NewCBCDecrypter(block Block, iv []byte) BlockMode
```

CBC mode. The IV must be random and unpredictable, and must not be reused with the same key. CBC does not authenticate — pair with HMAC if you use it.

Example

```
// Rotate an encryption key: decrypt with old, re-encrypt with
new
func reencrypt(oldKey, newKey, ciphertext []byte) ([]byte,
error) {
    oldBlock, _ := aes.NewCipher(oldKey)
    oldGCM, _ := cipher.NewGCM(oldBlock)

    nonceSize := oldGCM.NonceSize()
    plaintext, err := oldGCM.Open(nil,
        ciphertext[:nonceSize],
        ciphertext[nonceSize:],
        nil,
    )
```

```
    if err != nil {
        return nil, fmt.Errorf("decrypt: %w", err)
    }

    newBlock, _ := aes.NewCipher(newKey)
    newGCM, _ := cipher.NewGCM(newBlock)
    nonce := make([]byte, newGCM.NonceSize())
    io.ReadFull(rand.Reader, nonce)
    return newGCM.Seal(nonce, nonce, plaintext, nil), nil
}
```

Gotcha

Never reuse a nonce with the same GCM key. GCM nonce reuse is catastrophic — an attacker who observes two ciphertexts encrypted with the same key and nonce can recover the plaintext of both and forge arbitrary messages. Generate a fresh random nonce for every encryption operation. At 12 bytes from `crypto/rand`, collision probability is negligible across any realistic volume of messages.

See also

`crypto/aes` · `crypto/rand` · `crypto/hmac`

crypto/ecdh

Cryptography

The `crypto/ecdh` package implements Elliptic Curve Diffie-Hellman key exchange over NIST curves and Curve25519. Added in Go 1.20, it replaces the older pattern of using `crypto/elliptic` directly for key exchange — which was error-prone and required manual point arithmetic. ECDH is used to establish a shared secret between two parties without transmitting the secret itself: each party generates an ephemeral key pair, exchanges public keys, and derives an identical shared secret independently. That secret then seeds symmetric encryption.

Types

Type	Purpose
PrivateKey	An ECDH private key; can derive a shared secret and expose its PublicKey
PublicKey	An ECDH public key suitable for transmission
Curve	Represents a specific curve: P256, P384, P521, or X25519

Functions and methods

```
(Curve).GenerateKey(rand io.Reader) (*PrivateKey, error)
```

Generates a new key pair on the curve. Always pass `crypto/rand.Reader`.

```
(*PrivateKey).ECDH(remote *PublicKey) ([]byte, error)
```

Derives the shared secret from the local private key and the remote public key. Both parties calling this with each other's public keys

produce identical output.

```
(*PrivateKey).PublicKey() *PublicKey
```

Returns the public key to be transmitted to the other party.

```
(*PublicKey).Bytes() []byte
```

Serialises the public key for transmission. Use `(Curve).NewPublicKey` to deserialise on the other side.

Example

```
// Ephemeral ECDH key exchange over X25519
func keyExchange() (sharedSecret []byte, err error) {
    curve := ecdh.X25519()

    // Each party generates an ephemeral key pair
    alicePriv, _ := curve.GenerateKey(rand.Reader)
    bobPriv, _ := curve.GenerateKey(rand.Reader)

    // Exchange public keys (over the wire in practice)
    alicePub := alicePriv.PublicKey()
    bobPub := bobPriv.PublicKey()

    // Each derives the same shared secret independently
    aliceSecret, _ := alicePriv.ECDH(bobPub)
    bobSecret, _ := bobPriv.ECDH(alicePub)

    // aliceSecret == bobSecret; use as input to a KDF, not
directly as a key
    if !bytes.Equal(aliceSecret, bobSecret) {
        return nil, errors.New("key exchange failed")
    }
    return aliceSecret, nil
}
```

Gotcha

The raw shared secret from `ECDH` should not be used directly as an encryption key. It has non-uniform distribution and reveals information

about the private keys if used directly. Pass it through a key derivation function — `golang.org/x/crypto/hkdf` is the standard choice — to produce a uniformly random key of the required length.

See also

`crypto/ecdsa` · `crypto/tls` · `crypto/rand`

crypto/ecdsa

Cryptography

The `crypto/ecdsa` package implements the Elliptic Curve Digital Signature Algorithm. ECDSA signatures are smaller than RSA signatures for equivalent security and verification is fast, which makes them the dominant choice in modern TLS certificates, JWT ES256/ES384, and code signing. The package operates on keys from `crypto/elliptic` — P-256 is the standard curve for most use cases, providing 128-bit security with wide hardware and software support.

Types

Type	Purpose
PrivateKey	Signs data; embeds PublicKey
PublicKey	Verifies signatures; embeds elliptic.Curve and point coordinates

Functions and methods

```
GenerateKey(c elliptic.Curve, rand io.Reader) (*PrivateKey, error)
```

Generates a key pair on the given curve. Use `elliptic.P256()` for new applications.

```
SignASN1(rand io.Reader, priv *PrivateKey, hash []byte) ([]byte, error)
```

Signs a hash. The caller is responsible for hashing the message first — pass the output of `sha256.Sum256` or similar, not the raw message. Returns a DER-encoded signature.

```
VerifyASN1(pub *PublicKey, hash, sig []byte) bool
```

Verifies a DER-encoded signature against a hash and public key. Returns false on any failure — there is no error to inspect.

Example

```
// Sign a build artifact and verify the signature
func signArtifact(priv *ecdsa.PrivateKey, data []byte)
([]byte, error) {
    digest := sha256.Sum256(data)
    return ecdsa.SignASN1(rand.Reader, priv, digest[:])
}

func verifyArtifact(pub *ecdsa.PublicKey, data, sig []byte)
bool {
    digest := sha256.Sum256(data)
    return ecdsa.VerifyASN1(pub, digest[:], sig)
}

func newSigningKey() (*ecdsa.PrivateKey, error) {
    return ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
}
```

Gotcha

`SignASN1` requires a hash, not a message. Passing raw data of arbitrary length directly treats those bytes as a pre-computed hash and produces a signature over them regardless of their content. The bug is invisible until a verifier that correctly hashes the message fails to verify the signature — at which point the mismatch looks like a key problem, not a hashing problem.

See also

`crypto/ed25519` · `crypto/elliptic` · `crypto/sha256`

crypto/ed25519

Cryptography

The `crypto/ed25519` package implements Ed25519 signatures over Curve25519. Ed25519 is fast, produces small 64-byte signatures, has no parameter choices that can be misconfigured, and is not vulnerable to the nonce-reuse attacks that affect ECDSA. It is the right choice for new applications that need digital signatures: SSH keys, signed tokens, artifact signing, and capability proofs. The API is deliberately minimal — there are no configuration options because there are no safe alternatives to the defaults.

Types

Type	Purpose
PrivateKey	64-byte signing key (seed concatenated with public key)
PublicKey	32-byte verification key

Functions and methods

```
GenerateKey(rand io.Reader) (PublicKey, PrivateKey, error)
```

Generates a key pair. Always pass `crypto/rand.Reader`.

```
Sign(privateKey PrivateKey, message []byte) []byte
```

Signs the full message. Unlike ECDSA, Ed25519 hashes internally — pass the raw message, not a pre-computed hash.

```
Verify(publicKey PublicKey, message, sig []byte) bool
```

Verifies a signature. Returns false on any failure.

```
NewKeyFromSeed(seed []byte) PrivateKey
```

Deterministically derives a key pair from a 32-byte seed. Useful for reproducible key generation from a master secret.

Example

```
// Issue and verify a signed capability token
func issueToken(priv ed25519.PrivateKey, claims []byte) []byte
{
    sig := ed25519.Sign(priv, claims)
    token := make([]byte, len(claims)+ed25519.SignatureSize)
    copy(token, claims)
    copy(token[len(claims):], sig)
    return token
}

func verifyToken(pub ed25519.PublicKey, token []byte) ([]byte,
bool) {
    if len(token) < ed25519.SignatureSize {
        return nil, false
    }
    claims := token[:len(token)-ed25519.SignatureSize]
    sig := token[len(token)-ed25519.SignatureSize:]
    return claims, ed25519.Verify(pub, claims, sig)
}
```

Gotcha

`ed25519.Sign` takes the raw message, not a hash — the opposite convention from `crypto/ecdsa`. Passing a pre-computed hash to `Sign` produces a valid signature, but over the hash bytes rather than the original message. Verification succeeds only if the verifier also passes the same hash bytes. The bug is silent in tests where both sides use the same code, and surfaces only when interoperating with a system that signs the message correctly.

See also

crypto/ecdsa · crypto/ecdh · crypto/rand

crypto/hmac

Cryptography

The `crypto/hmac` package computes Hash-based Message Authentication Codes. An HMAC proves both that a message has not been tampered with and that it was produced by someone holding the secret key — properties that a plain hash cannot provide. It is the standard mechanism for webhook signature verification, API request signing, cookie integrity, and anywhere a shared secret must authenticate a message without asymmetric cryptography. The hash function is pluggable: `sha256.New` is the right default for new code.

Types

Type	Purpose
Hash	The hash.Hash interface returned by New; write message bytes, call Sum

Functions and methods

```
New(h func() hash.Hash, key []byte) hash.Hash
```

Creates an HMAC using the given hash constructor and key. Write the message in one or more calls to `Write`, then call `Sum(nil)` to retrieve the MAC.

```
Equal(mac1, mac2 []byte) bool
```

Compares two MACs in constant time. This is not interchangeable with `bytes.Equal` — use `hmac.Equal` whenever comparing authentication tags to prevent timing attacks.

Example

```
// Verify an incoming webhook signature
func verifyWebhook(secret, body []byte, sigHeader string)
error {
    mac := hmac.New(sha256.New, secret)
    mac.Write(body)
    expected := mac.Sum(nil)

    // Header typically arrives as "sha256=<hex>"
    hexSig := strings.TrimPrefix(sigHeader, "sha256=")
    got, err := hex.DecodeString(hexSig)
    if err != nil {
        return fmt.Errorf("malformed signature header: %w",
err)
    }
    if !hmac.Equal(expected, got) {
        return errors.New("signature mismatch")
    }
    return nil
}
```

Gotcha

`bytes.Equal` is not a safe substitute for `hmac.Equal`. `bytes.Equal` short-circuits on the first differing byte, leaking how many bytes of the expected MAC an attacker has guessed correctly through timing. `hmac.Equal` always examines every byte. This is a real attack class — HMAC timing attacks have been demonstrated against production systems. The fix is one word: use `hmac.Equal`.

See also

`crypto/sha256` · `crypto/subtle` · `encoding/hex`

crypto/rand

Cryptography

The `crypto/rand` package provides a cryptographically secure random number generator. Its sole export of consequence is `Reader` — an `io.Reader` that produces random bytes from the operating system's entropy source: `/dev/urandom` on Unix, `BCryptGenRandom` on Windows. Use this package for anything where predictability would be a security failure: tokens, session IDs, nonces, IVs, salts, and key generation. `math/rand` is faster but deterministic given a seed — it has no place in security-sensitive code.

Types

Type	Purpose
Reader	Package-level io.Reader backed by the OS CSPRNG

Functions and methods

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

Fills `b` with random bytes. On Linux 3.17+ and recent versions of other supported OSes, this call cannot fail under normal conditions — but always check the error regardless, as the call signature requires it and future platforms may differ.

```
Int(rand io.Reader, max *big.Int) (*big.Int, error)
```

Returns a cryptographically random integer in [0, max). Used when you need an unbiased random integer in a specific range for cryptographic purposes — key generation, blinding factors.

The most common pattern is `io.ReadFull(rand.Reader, buf)` rather than calling `rand.Read` directly. `io.ReadFull` guarantees the buffer is

completely filled, whereas a bare `Read` is technically permitted to return fewer bytes than requested, though in practice `crypto/rand` never does on supported platforms.

Example

```
// Generate a URL-safe token for session IDs, API keys,
password reset links
func generateToken(n int) (string, error) {
    b := make([]byte, n)
    if _, err := io.ReadFull(rand.Reader, b); err != nil {
        return "", fmt.Errorf("generating token: %w", err)
    }
    return base64.URLEncoding.EncodeToString(b), nil
}

// Generate a 256-bit AES key
func generateKey() ([]byte, error) {
    key := make([]byte, 32)
    if _, err := io.ReadFull(rand.Reader, key); err != nil {
        return nil, err
    }
    return key, nil
}
```

Gotcha

`crypto/rand.Read` and `math/rand.Read` have identical signatures. An import swap — whether accidental, through a refactor, or via an IDE's auto-import — silently replaces cryptographic randomness with a deterministic sequence. The compiler cannot catch this. In codebases that use both packages, alias one at import time: `crand "crypto/rand"` makes the distinction explicit at every call site.

See also

`crypto/hmac` · `crypto/aes` · `encoding/base64`

crypto/rsa

Cryptography

The `crypto/rsa` package implements RSA public-key cryptography. RSA remains common in TLS certificates, JWT RS256 signatures, and legacy key infrastructure, but for new systems that need asymmetric signatures, `crypto/ed25519` is faster, produces smaller outputs, and has a simpler API. RSA's continued relevance is largely interoperability — you will encounter RSA keys in the wild and need to work with them. The two operations are distinct: OAEP for encryption, PSS for signatures. Do not use the older PKCS1v15 variants for new code unless a counterparty requires them.

Types

Type	Purpose
PrivateKey	RSA private key; embeds PublicKey
PublicKey	RSA public key with modulus N and exponent E
OAEPOptions	Options for OAEP encryption: hash function and label
PSSOptions	Options for PSS signatures: salt length and hash function

Functions and methods

```
GenerateKey(random io.Reader, bits int) (*PrivateKey, error)
```

Generates an RSA key pair. 2048 bits is the current minimum; 4096 is the conservative choice for long-lived keys. Generation is slow — cache the result.

```
EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey,
msg, label []byte) ([]byte, error)
```

Encrypts a message with OAEP padding. The message length is limited by the key size minus padding overhead — RSA is not suited to encrypting large data directly. Encrypt a symmetric key, then use that key for the data.

```
DecryptOAEP(hash hash.Hash, random io.Reader, priv
*PrivateKey, ciphertext, label []byte) ([]byte, error)
```

Decrypts an OAEP-padded ciphertext. The `label` must match the one used during encryption.

```
SignPSS(rand io.Reader, priv *PrivateKey, hash crypto.Hash,
digest []byte, opts *PSSOptions) ([]byte, error)
```

Signs a pre-computed hash with PSS padding. Pass `nil` for `opts` to use sensible defaults.

```
VerifyPSS(pub *PublicKey, hash crypto.Hash, digest []byte, sig
[]byte, opts *PSSOptions) error
```

Verifies a PSS signature. Returns nil on success.

Example

```
// Encrypt a symmetric key for transmission (hybrid
encryption)
func sealKey(pub *rsa.PublicKey, symKey []byte) ([]byte,
error) {
    return rsa.EncryptOAEP(sha256.New(), rand.Reader, pub,
symKey, nil)
}

func openKey(priv *rsa.PrivateKey, ciphertext []byte) ([]byte,
error) {
    return rsa.DecryptOAEP(sha256.New(), rand.Reader, priv,
ciphertext, nil)
}
```

```
// Sign and verify a document hash
func sign(priv *rsa.PrivateKey, data []byte) ([]byte, error) {
    digest := sha256.Sum256(data)
    return rsa.SignPSS(rand.Reader, priv, crypto.SHA256,
digest[:], nil)
}

func verify(pub *rsa.PublicKey, data, sig []byte) error {
    digest := sha256.Sum256(data)
    return rsa.VerifyPSS(pub, crypto.SHA256, digest[:], sig,
nil)
}
```

Gotcha

RSA encryption has a maximum message size determined by key length and padding. With a 2048-bit key and OAEP-SHA256, the maximum plaintext is 190 bytes. Attempting to encrypt a larger message returns an error — but the error message ("message too long for RSA public key size") is easy to misread as a key configuration problem. The correct pattern is always hybrid encryption: encrypt a random 32-byte AES key with RSA, then encrypt the actual data with AES-GCM.

See also

`crypto/ecdsa` · `crypto/ed25519` · `crypto/rand`

crypto/sha256

Cryptography

The `crypto/sha256` package implements SHA-224 and SHA-256, the most widely used hash functions in production Go code. SHA-256 produces a 32-byte digest and is used everywhere a fixed-size, collision-resistant fingerprint of data is needed: checksums, content addressing, HMAC construction, certificate fingerprints, and as the pre-hash step before signing. The package implements `hash.Hash`, so a `sha256.New()` value composes with anything that accepts that interface.

Types

Type	Purpose
Hash	The hash.Hash interface, implemented by the value returned from New()

Functions and methods

```
New() hash.Hash
```

Returns a new SHA-256 hash. Write data incrementally with `Write`, retrieve the digest with `Sum(nil)`.

```
Sum256(data []byte) [32]byte
```

One-shot SHA-256 of a byte slice. Returns a fixed-size array, not a slice — use `digest[:]` when a `[]byte` is required.

```
New224() hash.Hash
```

Returns a SHA-224 hash. Produces a 28-byte digest. Used in some certificate profiles; rarely needed in application code.

The incremental `New()` form is appropriate when hashing data that arrives in chunks — reading a file, accumulating a request body. The one-shot `Sum256` is cleaner when all the data is available at once, and avoids the allocation of a `hash.Hash` object.

Example

```
// Hash a file incrementally for integrity verification
func hashFile(path string) (string, error) {
    f, err := os.Open(path)
    if err != nil {
        return "", err
    }
    defer f.Close()

    h := sha256.New()
    if _, err := io.Copy(h, f); err != nil {
        return "", err
    }
    return hex.EncodeToString(h.Sum(nil)), nil
}

// One-shot hash for a known value
func fingerprintCert(derBytes []byte) string {
    digest := sha256.Sum256(derBytes)
    return hex.EncodeToString(digest[:])
}
```

Gotcha

`Sum256` returns `[32]byte`, a fixed-size array. Arrays in Go are value types — passing one to a function that expects `[]byte` requires an explicit slice expression: `digest[:]`. Forgetting this produces a compile error that is easy to understand but easy to encounter the first time you reach for `Sum256` after using the incremental API.

See also

`crypto/sha512` · `crypto/hmac` · `encoding/hex`

crypto/sha512

Cryptography

The `crypto/sha512` package implements SHA-384 and SHA-512. SHA-512 produces a 64-byte digest — double the output of SHA-256 — and is used when a larger security margin is required: long-lived signatures, JWT HS512/ES512, and contexts where 256-bit collision resistance is deemed insufficient. On 64-bit hardware, SHA-512 is often faster than SHA-256 due to its larger internal word size, which makes it occasionally worth considering on performance grounds alone. The API mirrors `crypto/sha256` exactly.

Types

Type	Purpose
Hash	The hash.Hash interface, returned by New() and New384()

Functions and methods

```
New() hash.Hash
```

Returns a SHA-512 hash producing a 64-byte digest.

```
New384() hash.Hash
```

Returns a SHA-384 hash producing a 48-byte digest. Used in TLS 1.2 cipher suites and some certificate profiles.

```
Sum512(data []byte) [64]byte
```

One-shot SHA-512. Returns a fixed-size array — use `digest[:]` for a slice.

```
Sum384(data []byte) [48]byte
```

One-shot SHA-384.

The package also exports `New512_224` and `New512_256` — truncated variants defined in FIPS 180-4. These appear in specialised compliance contexts and are not needed in general application code.

Example

```
// SHA-512 for a password file checksum where collision
resistance matters most
func checksumFile(path string) (string, error) {
    f, err := os.Open(path)
    if err != nil {
        return "", err
    }
    defer f.Close()

    h := sha512.New()
    if _, err := io.Copy(h, f); err != nil {
        return "", err
    }
    return hex.EncodeToString(h.Sum(nil)), nil
}
```

Gotcha

SHA-512 and SHA-256 are not interchangeable at the protocol level. Systems that verify checksums or signatures expect a specific hash algorithm and will reject output from the wrong one even if the inputs are identical. Mixing them — using SHA-512 on one side and SHA-256 on the other — produces a silent verification failure with no indication of which side is wrong.

See also

`crypto/sha256` · `crypto/hmac` · `encoding/hex`

crypto/subtle

Cryptography

The `crypto/subtle` package provides constant-time functions for security-sensitive comparisons and operations. Its name reflects its purpose: the functions here exist because the obvious alternatives — `bytes.Equal`, plain XOR loops — are subtly wrong in cryptographic contexts. Timing side-channels are real attacks. When comparing authentication tags, password hashes, or any value whose contents must not be leaked through execution time, use this package. If you are using `crypto/hmac`, you may not need `crypto/subtle` directly — `hmac.Equal` already wraps `subtle.ConstantTimeCompare`.

Types

None. The package exports functions only.

Functions and methods

```
ConstantTimeCompare(x, y []byte) int
```

Returns 1 if x and y are equal, 0 otherwise. Always examines every byte of both slices regardless of where a difference occurs. Returns 0 immediately if the lengths differ — length is not secret in most protocols, but if it is, pad to a fixed length before comparing.

```
ConstantTimeSelect(v, x, y int) int
```

Returns x if v == 1, y if v == 0. Performs the selection without branching. Used when building constant-time conditional logic.

```
XORBytes(dst, x, y []byte) int
```

XORs x and y into dst in constant time. Added in Go 1.20. Useful in stream cipher and MAC construction.

Example

```
// Compare a submitted API key against a stored hash without
timing leakage
func validateAPIKey(submitted, stored []byte) bool {
    // Hash both to fixed length before comparing,
    // so length differences do not leak information
    h1 := sha256.Sum256(submitted)
    h2 := sha256.Sum256(stored)
    return subtle.ConstantTimeCompare(h1[:], h2[:]) == 1
}
```

Gotcha

`ConstantTimeCompare` returns 0 immediately when the slice lengths differ. If the length of a value is itself sensitive — a variable-length secret where the length must not be revealed — pad both inputs to the same fixed length before comparing. In most authentication scenarios the length is not secret, but in protocol implementations where it could be, the padding step is necessary.

See also

`crypto/hmac` · `crypto/sha256` · `bytes`

database/sql

Data

The `database/sql` package provides a generic interface to SQL databases. It handles connection pooling, prepared statements, transactions, and result scanning — everything except the wire protocol, which is provided by a driver registered separately via `database/sql/driver`. The package is deliberately database-agnostic: the same code works against SQLite, PostgreSQL, MySQL, or any other database with a driver. Import the driver for its side effects only; the rest of your code touches only `database/sql`.

Types

Type	Purpose
DB	A pool of database connections; safe for concurrent use
Tx	An in-progress transaction
Stmt	A prepared statement, reusable across queries
Rows	The result set from a query; must be closed
Row	The result of a single-row query
NullString	A string that may be NULL in the database

Functions and methods

`sql.Open` does not establish a connection — it validates the driver name and data source string and returns a `*DB`. The first actual connection is made lazily on the first query. Call `db.Ping` immediately after `Open` if you need to verify connectivity at startup.

Connection pool behaviour is controlled by four methods on `*DB`: `SetMaxOpenConns`, `SetMaxIdleConns`, `SetConnMaxLifetime`, and `SetConnMaxIdleTime`. The defaults — unlimited open connections, 2 idle connections — are wrong for most production deployments. Set these explicitly.

```
(*DB).QueryContext(ctx, query, args) (*Rows, error)
```

Executes a query returning multiple rows. Always use the context variant in production code. Always close the returned `*Rows`, even on error.

```
(*DB).QueryRowContext(ctx, query, args) *Row
```

Executes a query expected to return one row. Call `(*Row).Scan` to retrieve values; it returns `sql.ErrNoRows` if nothing matched.

```
(*DB).ExecContext(ctx, query, args) (Result, error)
```

Executes a statement that does not return rows. `Result` carries the last insert ID and rows affected.

```
(*DB).BeginTx(ctx, opts) (*Tx, error)
```

Starts a transaction. The `*Tx` has the same `QueryContext`, `ExecContext`, and `PrepareContext` methods as `*DB`. Always call either `Commit` or `Rollback` — defer `Rollback` immediately after `BeginTx` and call `Commit` explicitly on success.

Example

```
func transferFunds(ctx context.Context, db *sql.DB, from, to
int64, amount int) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback() // no-op if Commit succeeds
```

```
    if _, err := tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance - $1 WHERE id =
$2",
        amount, from,
    ); err != nil {
        return fmt.Errorf("debit: %w", err)
    }
    if _, err := tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance + $1 WHERE id =
$2",
        amount, to,
    ); err != nil {
        return fmt.Errorf("credit: %w", err)
    }
    return tx.Commit()
}
```

Gotcha

`(*Rows).Close()` must be called when you are done with a result set — even if you have iterated to the end. Unclosed `Rows` hold a connection from the pool, and forgetting to close them in error paths or early returns exhausts the pool silently. Defer `rows.Close()` immediately after checking the error from `QueryContext`.

See also

`context` · `errors` · `fmt`

embed

System

The `embed` package makes files and directory trees available inside a compiled binary at build time. A single `//go:embed` directive on a variable declaration is all that is required — no asset pipeline, no bundler, no runtime file path configuration. The embedded content is immutable and available from the first line of `main`. This makes single-binary deployment genuinely single-binary: templates, static assets, migration files, and default configuration travel with the executable.

Types

Type	Purpose
FS	An in-memory filesystem implementing io/fs.FS; produced by //go:embed on a variable of this type

Functions and methods

The package exports no functions. All functionality comes from the `//go:embed` directive and the `embed.FS` type, which implements `fs.FS`, `fs.ReadFileFS`, and `fs.ReadDirFS`.

```
(FS).Open(name string) (fs.File, error)
```

Opens a file by path. Paths use forward slashes regardless of OS.

```
(FS).ReadFile(name string) ([]byte, error)
```

Reads a file's full contents. Convenience wrapper over `Open`.

```
(FS).ReadDir(name string) ([]fs.DirEntry, error)
```

Lists directory contents. Use `"."` for the root of the embedded tree.

Embed a single file into a `string` or `[]byte` variable for the simplest case. Use `embed.FS` when you need to embed a directory tree or serve files over HTTP.

Example

```
import _ "embed"

//go:embed config/defaults.yaml
var defaultConfig []byte

//go:embed static/*
var staticFiles embed.FS

//go:embed templates
var templateFiles embed.FS

func main() {
    // Serve embedded static files
    http.Handle("/static/", http.StripPrefix("/static/",
        http.FileServer(http.FS(staticFiles)),
    ))

    // Parse embedded templates
    tmpl := template.Must(
        template.ParseFS(templateFiles, "templates/*.html"),
    )
    _ = tmpl
}
```

Gotcha

`//go:embed` does not embed files whose names begin with `.` or `_`, and does not embed files in directories whose names begin with `.` or `_`. This catches developers embedding a directory that contains a `.git` folder or `.env` file — those are silently excluded. If a file you expect to be present is missing at runtime, check its name and the names of its parent

directories.

See also

`io/fs` · `html/template` · `net/http`

encoding/base32

Encoding

The `encoding/base32` package encodes binary data as a case-insensitive alphabet of 32 characters. Base32 is used where base64 is unsuitable: TOTP and HOTP secret keys as displayed in authenticator apps, some DNS encodings, and contexts where the output must survive case-folding or be entered by hand. The size cost is a reason to reach for it only when the format demands it. Base64 packs 6 bits per character, encoding 3 bytes as 4 characters — a 4/3 ratio, or 33% overhead. Base32 packs only 5 bits per character, encoding 5 bytes as 8 characters — an 8/5 ratio, or 60% overhead. Both figures are a direct consequence of the encoding ratios defined in RFC 4648. For general binary-to-text encoding where case sensitivity is not a concern, base64 is the right choice.

Types

Type	Purpose
Encoding	A base32 alphabet and padding configuration

Functions and methods

The package provides two standard encodings as package-level variables: `StdEncoding` uses the standard RFC 4648 alphabet with padding; `HexEncoding` uses an extended hex alphabet that preserves sort order. For TOTP secrets, use `StdEncoding` without padding — most authenticator apps expect unpadded base32.

```
(*Encoding).EncodeToString(src []byte) string
```

Encodes src to a base32 string with padding characters.

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

Decodes a padded base32 string.

`(*Encoding).WithPadding(padding rune) *Encoding`

Returns a copy of the encoding with a different padding character, or `NoPadding` to disable padding entirely.

Example

```
// Generate and display a TOTP secret compatible with Google
Authenticator
func newTOTPSecret() (string, error) {
    secret := make([]byte, 20) // 160 bits
    if _, err := io.ReadFull(rand.Reader, secret); err != nil
{
        return "", err
    }
    // Authenticator apps expect unpadded base32
    return
base32.StdEncoding.WithPadding(base32.NoPadding).EncodeToStrin
g(secret), nil
}
```

Gotcha

Most TOTP implementations and authenticator apps expect base32 without padding characters. `StdEncoding.EncodeToString` includes padding by default. A secret encoded with padding will be rejected by apps that do not strip = characters before decoding. Always use `WithPadding(base32.NoPadding)` when generating TOTP secrets.

See also

`encoding/base64` · `crypto/rand` · `encoding/hex`

encoding/base64

Encoding

The `encoding/base64` package encodes binary data as printable ASCII text. It is the standard mechanism for transmitting binary data in contexts that expect text: HTTP Basic authentication headers, JWT tokens, data URIs, and email attachments. The package provides four encodings — standard with padding, standard without padding, URL-safe with padding, and URL-safe without padding — because the wrong choice for the context will fail or produce subtly incorrect output.

Types

Type	Purpose
Encoding	A base64 alphabet and padding configuration

Functions and methods

Four standard encodings are available as package-level variables:

- `StdEncoding` — standard alphabet (`+`, `/`), with padding. Use for MIME and most binary-to-text contexts.
- `URLEncoding` — URL-safe alphabet (`-`, `_`), with padding. Use when the output appears in a URL query string or path.
- `RawStdEncoding` — standard alphabet, no padding. Use for JWT header and payload segments.
- `RawURLEncoding` — URL-safe alphabet, no padding. Use for JWT signatures and most modern token formats.

```
(*Encoding).EncodeToString(src []byte) string
```

Encodes src to a base64 string.

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

Decodes a base64 string. Returns an error if the input contains characters outside the encoding's alphabet.

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

Streaming base64 encoder. Must be closed to flush the final block.

Example

```
// Encode binary data for a data URI (HTML inline image)
func dataURI(mimeType string, data []byte) string {
    encoded := base64.StdEncoding.EncodeToString(data)
    return fmt.Sprintf("data:%s;base64,%s", mimeType, encoded)
}

// Decode a JWT payload (unpadded, URL-safe)
func decodeJWTPayload(segment string) ([]byte, error) {
    return base64.RawURLEncoding.DecodeString(segment)
}
```

Gotcha

The four encodings are not interchangeable. A string encoded with `StdEncoding` cannot be decoded with `URLEncoding` if it contains `+` or `/` characters — those map to `-` and `_` in the URL-safe alphabet, and the decoder will return an error. JWT tokens use `RawURLEncoding`; HTTP Basic auth uses `StdEncoding`. Using the wrong one produces errors that look like corrupted data rather than an encoding mismatch.

See also

`encoding/base32` · `encoding/hex` · `crypto/rand`

encoding/binary

Encoding

The `encoding/binary` package reads and writes fixed-size values in a specified byte order. It is the right tool whenever you are working at the wire level: parsing network protocol headers, reading binary file formats, constructing packed data structures. Unlike `encoding/json` or `encoding/gob`, there is no schema or self-description — just raw bytes in a known layout. The caller is responsible for knowing the byte order and field widths that the format requires.

Types

Type	Purpose
ByteOrder	Interface implemented by BigEndian and LittleEndian
AppendByteOrder	Extended ByteOrder with Append* methods (Go 1.21+)

Functions and methods

`BigEndian` and `LittleEndian` are package-level values implementing `ByteOrder`. Network protocols conventionally use big-endian (also called network byte order); x86 hardware is little-endian. Choose based on what the format specifies.

```
binary.Read(r io.Reader, order ByteOrder, data any) error
```

Reads structured data from `r` into `data`. `data` must be a pointer to a fixed-size value or a slice of fixed-size values. Reads exactly as many bytes as `data` requires.

```
binary.Write(w io.Writer, order ByteOrder, data any) error
```

Writes the binary representation of `data` to `w`. Same constraints as `Read`.

```
binary.Size(v any) int
```

Returns the number of bytes that `Write` would produce for `v`. Useful for pre-allocating buffers.

For performance-critical paths, the `AppendByteOrder` methods — `AppendUint16`, `AppendUint32`, `AppendUint64` — append directly to a byte slice without an `io.Writer`, avoiding allocation.

Example

```
// Parse a fixed binary protocol header
type MessageHeader struct {
    Magic   uint32
    Version uint8
    Length  uint32
}

func readHeader(r io.Reader) (*MessageHeader, error) {
    var hdr MessageHeader
    if err := binary.Read(r, binary.BigEndian, &hdr); err !=
nil {
        return nil, fmt.Errorf("reading header: %w", err)
    }
    if hdr.Magic != 0xDEADBEEF {
        return nil, errors.New("invalid magic number")
    }
    return &hdr, nil
}

// Build a frame header into a pre-allocated buffer
func writeHeader(buf []byte, length uint32) []byte {
    buf = binary.BigEndian.AppendUint32(buf, 0xDEADBEEF)
    buf = append(buf, 1) // version
    buf = binary.BigEndian.AppendUint32(buf, length)
    return buf
}
```

Gotcha

`binary.Read` and `binary.Write` only work with fixed-size types: integers, floats, booleans, arrays, and structs composed entirely of those types. Passing a struct with a `string`, `[]byte`, or interface field returns an error. If your format mixes fixed-size fields with variable-length data, read the fixed header with `binary.Read` and handle the variable portions separately with explicit slice operations.

See also

`encoding/hex` · `bytes` · `io`

encoding/csv

Encoding

The `encoding/csv` package reads and writes comma-separated values as defined in RFC 4180. It handles the parts of CSV that are easy to get wrong when rolling your own: quoted fields containing commas or newlines, escaped quotes within quoted fields, and varying line endings. The `Reader` is streaming — it processes one record at a time rather than loading the entire file — which makes it suitable for files of any size. The delimiter is configurable, so the same package handles TSV and other separated-value formats.

Types

Type	Purpose
Reader	Reads CSV records from an io.Reader one at a time
Writer	Writes CSV records to an io.Writer; must be flushed

Functions and methods

`NewReader` wraps any `io.Reader`. Two fields on the returned `Reader` are worth setting before reading begins: `r.TrimLeadingSpace = true` strips whitespace from unquoted fields, and `r.Comma = '\t'` switches to tab-separated input. Set `r.FieldsPerRecord = -1` to allow rows with varying column counts, or set it to a positive number to enforce a fixed width and return an error on deviation.

```
(*Reader).Read() ([]string, error)
```

Returns the next record as a slice of strings. Returns `io.EOF` when the input is exhausted. Each field is always a string — type conversion is the caller's responsibility.

```
(*Reader).ReadAll() ([][]string, error)
```

Reads all remaining records into memory. Convenient for small files; avoid for large ones.

```
(*Writer).Write(record []string) error
```

Writes a single record. Quoting and escaping are applied automatically.

```
(*Writer).Flush()
(*Writer).Error() error
```

`Flush` pushes buffered data to the underlying writer. Call `Error` after `Flush` to check whether any write failed — `Write` itself buffers errors rather than returning them immediately.

Example

```
// Stream a large CSV, transform, and write output
func convertCurrency(r io.Reader, w io.Writer, rate float64)
error {
    cr := csv.NewReader(r)
    cr.TrimLeadingSpace = true

    cw := csv.NewWriter(w)
    defer cw.Flush()

    // Write header unchanged
    header, err := cr.Read()
    if err != nil {
        return err
    }
    cw.Write(header)

    for {
        record, err := cr.Read()
        if err == io.EOF {
            break
        }
        if err != nil {
            return err
```

```
        }
        amount, err := strconv.ParseFloat(record[2], 64)
        if err != nil {
            return fmt.Errorf("row %v: %w", record, err)
        }
        record[2] = strconv.FormatFloat(amount*rate, 'f', 2,
64)
        cw.Write(record)
    }

    cw.Flush()
    return cw.Error()
}
```

Gotcha

`(*Writer).Write` does not return write errors directly — it buffers them. A loop that checks the error from `Write` on every iteration will never see a failure. Call `Flush` when done and then check `Error()`. Skipping `Error()` after `Flush` means a partially written file can be silently accepted as complete.

See also

`encoding/json` · `bufio` · `strconv`

encoding/gob

Encoding

The `encoding/gob` package serialises Go values into a compact binary format designed for Go-to-Go communication. Unlike JSON or XML, gob is self-describing at the type level — the encoder transmits type information alongside data, so the decoder does not need to know the schema in advance. It handles interfaces, pointers, and recursive types correctly. Gob is well suited for caching encoded values, RPC between Go processes, and persistent stores where the reader and writer are both Go programs. It is not suitable for cross-language interchange.

Types

Type	Purpose
Encoder	Encodes values to an io.Writer
Decoder	Decodes values from an io.Reader

Functions and methods

The `Encoder` and `Decoder` maintain state across calls — they track which types have already been transmitted so that type information is sent only once per connection, not once per value. This makes them unsuitable for one-shot encoding where each encoded blob must be self-contained; for that, encode into a `bytes.Buffer` and transmit the buffer.

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

Creates an encoder writing to `w`. Reuse the same encoder across multiple `Encode` calls on the same connection for efficiency.

```
(*Encoder).Encode(e any) error
```

Encodes a single value. The first call for a given type transmits type metadata; subsequent calls transmit only data.

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

Creates a decoder reading from `r`.

```
(*Decoder).Decode(e any) error
```

Decodes the next value into `e`, which must be a non-nil pointer.

To encode interface values, register the concrete types that may appear behind the interface with `gob.Register` before encoding or decoding begins. Registration must happen on both the encoder and decoder side, and must occur before the first call to `Encode` or `Decode`. The usual pattern is an `init` function:

```
func init() {
    gob.Register(ConcreteTypeA{})
    gob.Register(ConcreteTypeB{})
}
```

Omitting registration causes `Encode` to return an error and `Decode` to silently produce a nil interface value.

Example

```
// Cache a computed result to disk
type CachedResult struct {
    ComputedAt time.Time
    Values     []float64
    Metadata   map[string]string
}

func saveCache(path string, result CachedResult) error {
    f, err := os.Create(path)
    if err != nil {
        return err
```

```
    }
    defer f.Close()
    return gob.NewEncoder(f).Encode(result)
}

func loadCache(path string) (CachedResult, error) {
    f, err := os.Open(path)
    if err != nil {
        return CachedResult{}, err
    }
    defer f.Close()
    var result CachedResult
    return result, gob.NewDecoder(f).Decode(&result)
}
```

Gotcha

Gob encoding is not stable across Go versions or across changes to a type's fields. Adding, removing, or reordering exported fields can cause decoders to silently ignore fields or fail to decode values written by an older encoder. Do not use gob for data that must be read back after a code change — use a versioned format like JSON or protobuf for long-lived persistence.

See also

`encoding/json` · `encoding/binary` · `bytes`

encoding/hex

Encoding

The `encoding/hex` package encodes byte slices as hexadecimal strings and decodes them back. Hex is the universal fallback encoding: every debugger, shell tool, and network protocol understands it, it is case-insensitive, and it is trivially human-readable for short values. The cost is size — hex doubles the input (100% overhead, or 2 characters per byte), which is why it is used for short fixed-length identifiers — SHA-256 digests, UUIDs, MAC addresses — rather than bulk data.

Types

None. The package exports functions and a `Dumper` type for formatted output.

Functions and methods

```
EncodeToString(src []byte) string
```

Encodes src as a lowercase hex string. The result is always `len(src)*2` characters.

```
DecodeString(s string) ([]byte, error)
```

Decodes a hex string. Case-insensitive. Returns an error if `s` contains non-hex characters or has an odd length.

```
Dump(src []byte) string
```

Returns a formatted hex dump with offsets and ASCII representation on the right — the same layout as `xxd`. Useful for debugging binary protocols and file formats.

```
NewEncoder(w io.Writer) io.Writer
```

Streaming hex encoder. Writes two hex characters for each byte written to it.

```
NewDecoder(r io.Reader) io.Reader
```

Streaming hex decoder. Reads two characters at a time and emits one byte.

Example

```
// Display a hash in the conventional format
func checksumHex(data []byte) string {
    digest := sha256.Sum256(data)
    return hex.EncodeToString(digest[:])
}

// Debug an unexpected binary payload
func debugPayload(data []byte) {
    fmt.Println(hex.Dump(data))
}

// Decode a hex-encoded key from configuration
func loadKey(hexKey string) ([]byte, error) {
    key, err := hex.DecodeString(strings.TrimSpace(hexKey))
    if err != nil {
        return nil, fmt.Errorf("invalid key encoding: %w",
err)
    }
    if len(key) != 32 {
        return nil, fmt.Errorf("key must be 32 bytes, got %d",
len(key))
    }
    return key, nil
}
```

Gotcha

`DecodeString` requires an even number of characters — each byte is represented by exactly two hex digits. A hex string with an odd length

returns an error. When parsing hex values from external sources such as config files or API responses, trim whitespace and validate length before decoding; the error message ("encoding/hex: odd length hex string") is accurate but not immediately obvious about the cause.

See also

`encoding/base64` · `crypto/sha256` · `fmt`

encoding/json

Encoding

The `encoding/json` package marshals Go values to JSON and unmarshals JSON back into Go values. Struct field names map to JSON keys via `json:""` struct tags; without a tag, the exported field name is used as-is. The package supports streaming through `Encoder` and `Decoder`, which write to and read from any `io.Writer` and `io.Reader` — making it possible to encode directly into an HTTP response body or decode directly from a request body without buffering the entire payload. Custom marshaling behaviour is available by implementing `json.Marshaler` and `json.Unmarshaler`.

Types

Type	Purpose
Encoder	Streams JSON values to an io.Writer
Decoder	Streams JSON values from an io.Reader; supports token-level parsing via Token()
RawMessage	A raw encoded JSON value; defers or pre-computes marshaling
Number	A JSON number that preserves the original string representation

Functions and methods

For HTTP handlers, prefer `NewEncoder` and `NewDecoder` over `Marshal` and `Unmarshal` — they avoid the intermediate `[]byte` allocation and compose directly with `http.ResponseWriter` and `r.Body`.

```
Marshal(v any) ([]byte, error)
```

Encodes v to a JSON byte slice. Useful when the full encoded form is needed as a value — for logging, signing, or storage.

```
Unmarshal(data []byte, v any) error
```

Decodes JSON bytes into v. v must be a non-nil pointer.

```
NewEncoder(w io.Writer) *Encoder
(*Encoder).Encode(v any) error
```

Writes a JSON-encoded value to w, followed by a newline. Reuse the encoder for multiple values on the same writer.

```
NewDecoder(r io.Reader) *Decoder
(*Decoder).Decode(v any) error
```

Reads and decodes one JSON value from r. Call repeatedly to consume a stream of newline-delimited JSON objects.

```
(*Decoder).DisallowUnknownFields()
```

Makes the decoder return an error on JSON keys that have no corresponding struct field. Useful for strict input validation in APIs.

Example

```
type CreateOrderRequest struct {
    ProductID string  `json:"product_id"`
    Quantity  int     `json:"quantity"`
    Note      string  `json:"note,omitempty"`
}

type OrderResponse struct {
    ID        string    `json:"id"`
    CreatedAt time.Time `json:"created_at"`
}

func createOrder(w http.ResponseWriter, r *http.Request) {
    var req CreateOrderRequest
    dec := json.NewDecoder(r.Body)
    dec.DisallowUnknownFields()
```

```
    if err := dec.Decode(&req); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    resp := OrderResponse{
        ID:        uuid.New().String(),
        CreatedAt: time.Now().UTC(),
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(resp)
}
```

Gotcha

Unmarshaling JSON numbers into `any` (or `interface{}`) fields produces `float64`, not `int`. A JSON value of `1000000` decoded into an `any` field becomes the float64 `1000000` — which round-trips correctly for small integers but loses precision for integers larger than 2^53. When exact integer values matter, decode into a typed struct with explicit `int64` or `json.Number` fields rather than `any`.

See also

`encoding/csv` · `encoding/xml` · `net/http`

encoding/pem

Encoding

The `encoding/pem` package encodes and decodes PEM blocks — the `-----BEGIN CERTIFICATE-----` format used to store TLS certificates, private keys, certificate signing requests, and public keys in text files. PEM (Privacy Enhanced Mail) is base64-encoded binary data wrapped in a typed header and footer, defined in RFC 7468. Almost any code that reads or writes TLS credentials from disk, generates certificates programmatically, or communicates with a PKI will pass through this package.

Types

Type	Purpose
Block	A single PEM block with a Type string, optional Headers map, and Bytes payload

Functions and methods

```
Decode(data []byte) (p *Block, rest []byte)
```

Finds and decodes the first PEM block in data. Returns the decoded block and any bytes after it. Returns `nil` for the block if no PEM data is found. To decode a file with multiple blocks — a certificate chain — call `Decode` in a loop until the returned block is nil.

```
Encode(out io.Writer, b *Block) error
```

Writes a PEM-encoded block to `out`. The `Type` field becomes the header and footer label.

```
EncodeToMemory(b *Block) []byte
```

Returns the PEM encoding as a byte slice. Useful when the encoded form is needed as a value rather than written to a writer.

Example

```
// Load a certificate and private key from PEM files
func loadTLSCredentials(certFile, keyFile string)
(tls.Certificate, error) {
    return tls.LoadX509KeyPair(certFile, keyFile)
}

// Decode all certificates in a PEM chain
func parseCertChain(pemData []byte) ([]*x509.Certificate,
error) {
    var certs []*x509.Certificate
    for {
        block, rest := pem.Decode(pemData)
        if block == nil {
            break
        }
        if block.Type != "CERTIFICATE" {
            pemData = rest
            continue
        }
        cert, err := x509.ParseCertificate(block.Bytes)
        if err != nil {
            return nil, fmt.Errorf("parsing certificate: %w",
err)
        }
        certs = append(certs, cert)
        pemData = rest
    }
    return certs, nil
}

// Write a generated certificate to a file
func writeCertPEM(path string, derBytes []byte) error {
    f, err := os.Create(path)
    if err != nil {
        return err
    }
    defer f.Close()
```

```
    return pem.Encode(f, &pem.Block{Type: "CERTIFICATE",
Bytes: derBytes})
}
```

Gotcha

`pem.Decode` returns `nil` for the block — not an error — when no PEM data is found in the input. Code that checks only the error and proceeds on a nil block will panic or silently produce an empty result. Always check that the returned block is non-nil before accessing its fields. The common case where this bites is reading a file that contains only whitespace or a Windows line-ending variant that the PEM parser does not recognise.

See also

`crypto/tls` · `crypto/x509` · `encoding/base64`

encoding/xml

Encoding

The `encoding/xml` package marshals Go values to XML and unmarshals XML back into Go values, using struct tags to control element names, attributes, and nesting. Like `encoding/json`, it supports streaming through `Encoder` and `Decoder`. XML's verbosity and structural complexity make it a less convenient format than JSON, but it remains unavoidable in contexts where it is mandated: SOAP services, RSS and Atom feeds, configuration formats like Maven and Android manifests, and document exchange standards in regulated industries.

Types

Type	Purpose
Encoder	Streams XML to an io.Writer
Decoder	Streams XML from an io.Reader; supports token-level parsing
Token	Interface representing a single XML token: StartElement, EndElement, CharData, Comment
Attr	A single XML attribute with Name and Value

Functions and methods

Struct tags control the XML mapping. `xml:"name"` sets the element name; `xml:"name,attr"` maps to an attribute; `xml:",chardata"` maps to the text content of the element; `xml:",innerxml"` captures raw XML without parsing. Nesting is expressed through struct embedding.

`Marshal(v any) ([]byte, error)`

Encodes v to XML bytes. Does not add an XML declaration — prepend `[]byte(xml.Header)` if one is required.

`MarshalIndent(v any, prefix, indent string) ([]byte, error)`

Like `Marshal` but with indentation for human-readable output.

`Unmarshal(data []byte, v any) error`

Decodes XML into v. Unknown elements are silently ignored by default.

`(*Decoder).Token() (Token, error)`

Returns the next XML token. Use for streaming large documents or when the schema is too irregular for struct mapping.

Example

```
type Feed struct {
    XMLName xml.Name `xml:"rss"`
    Version string   `xml:"version,attr"`
    Channel Channel  `xml:"channel"`
}

type Channel struct {
    Title       string `xml:"title"`
    Link        string `xml:"link"`
    Description string `xml:"description"`
    Items       []Item `xml:"item"`
}

type Item struct {
    Title   string `xml:"title"`
    Link    string `xml:"link"`
    PubDate string `xml:"pubDate"`
}

func parseRSS(r io.Reader) (*Feed, error) {
    var feed Feed
```

```
    if err := xml.NewDecoder(r).Decode(&feed); err != nil {
        return nil, fmt.Errorf("parsing RSS: %w", err)
    }
    return &feed, nil
}
```

Gotcha

`xml.Marshal` does not emit an XML declaration (`<?xml version="1.0" encoding="UTF-8"?>`). Some XML consumers — particularly SOAP endpoints and older document parsers — require the declaration and will reject input without it. Prepend `xml.Header` (a package-level string constant containing the standard declaration) to the marshaled output when targeting those systems.

See also

`encoding/json` · `encoding/pem` · `io`

errors

Diagnostics

The `errors` package provides the tools for creating, wrapping, and inspecting errors. Go errors are values — an `error` is an interface with a single `Error() string` method — and this package builds the standard vocabulary for working with them idiomatically. Wrapping an error with `fmt.Errorf("context: %w", err)` preserves the original error while adding context; `errors.Is` and `errors.As` then unwrap the chain to find specific errors or types, regardless of how many layers of wrapping lie between the caller and the original cause.

Types

None exported directly. The package defines behaviour through functions and the `error` interface.

Functions and methods

```
errors.New(text string) error
```

Creates a simple error value. Each call to `New` with the same text returns a distinct error — two errors created with `New("not found")` are not equal. Use package-level variables for sentinel errors that callers need to test with `Is`.

```
errors.Is(err, target error) bool
```

Reports whether any error in err's chain matches target. Matching is by identity for simple errors (same pointer) and by the `Is(error) bool` method for errors that implement it. Use this instead of `==` comparison for wrapped errors.

```
errors.As(err error, target any) bool
```

Finds the first error in err's chain that can be assigned to target, and if so assigns it and returns true. target must be a non-nil pointer to either a type that implements `error`, or any interface type. Use this to extract structured error values from a wrapped chain.

```
errors.Unwrap(err error) error
```

Returns the result of calling `Unwrap() error` on err, or nil if err does not implement `Unwrap`. Rarely called directly — `Is` and `As` unwrap automatically.

```
errors.Join(errs ...error) error
```

Combines multiple errors into one, added in Go 1.20. The result's `Error` method concatenates the individual messages. `Is` and `As` inspect each wrapped error in turn. Useful for collecting validation failures or partial operation errors.

Example

```
// Sentinel errors for a storage package
var (
    ErrNotFound   = errors.New("not found")
    ErrPermission = errors.New("permission denied")
)

// Structured error with context
type QueryError struct {
    Query string
    Err   error
}

func (e *QueryError) Error() string {
    return fmt.Sprintf("query %q: %v", e.Query, e.Err)
}
func (e *QueryError) Unwrap() error { return e.Err }

func findUser(id string) (*User, error) {
    user, err := db.Lookup(id)
    if err != nil {
```

```
        return nil, &QueryError{Query: id, Err: ErrNotFound}
    }
    return user, nil
}

func handleLookup(id string) {
    _, err := findUser(id)
    if errors.Is(err, ErrNotFound) {
        // responds 404
    }
    var qe *QueryError
    if errors.As(err, &qe) {
        log.Printf("failed query: %s", qe.Query)
    }
}
```

Gotcha

`errors.Is` compares by identity, not by message text. Two errors created with `errors.New("not found")` at different call sites are distinct values — `Is` will return false when comparing them even though their messages are identical. Sentinel errors must be package-level variables, not constructed inline, for `Is` to work correctly across package boundaries.

See also

`fmt` · `log/slog` · `context`

expvar

Diagnostics

The `expvar` package exports named variables over HTTP as a JSON object at `/debug/vars`. Importing it for its side effects registers the endpoint on the default `ServeMux`; no further configuration is required. It is useful for exposing operational metrics — request counters, cache hit rates, queue depths, build information — from a running service without pulling in a metrics library. The variables are updated atomically and read safely under concurrent access.

Types

Type	Purpose
Int	An integer counter; safe for concurrent increment
Float	A float64 gauge
String	A string value, JSON-escaped on export
Map	A string-keyed map of Var values
Func	A function called on each export; return value is JSON-encoded

Functions and methods

```
NewInt(name string) *Int
```

Creates and publishes a new integer variable. The name appears as the JSON key at `/debug/vars`. Panics if the name is already registered.

```
(*Int).Add(delta int64)
(*Int).Set(value int64)
(*Int).Value() int64
```

Increment, set, and read an `Int`. `Add` is the common operation for counters; it is atomic.

```
NewMap(name string) *Map
(*Map).Add(key string, delta int64)
(*Map).Set(key string, av Var)
```

A `Map` holds named sub-variables. Use it for per-endpoint counters or per-status-code breakdowns within a single exported variable.

```
Publish(name string, v Var)
```

Registers any `Var` implementation under a name. Use with `Func` to export computed values — uptime, memory stats, queue length — that are derived at read time rather than maintained as running counters.

Example

```
import _ "expvar" // registers /debug/vars on DefaultServeMux

var (
    requestCount  = expvar.NewInt("requests_total")
    requestErrors = expvar.NewInt("requests_errors")
    activeConns   = expvar.NewInt("connections_active")
)

func handler(w http.ResponseWriter, r *http.Request) {
    requestCount.Add(1)
    activeConns.Add(1)
    defer activeConns.Add(-1)

    if err := process(r); err != nil {
        requestErrors.Add(1)
        http.Error(w, err.Error(),
http.StatusInternalServerError)
        return
    }
    w.WriteHeader(http.StatusOK)
}

func init() {
```

```
    // Export computed value: current memory allocation
    expvar.Publish("memory_alloc_bytes", expvar.Func(func()
any {
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        return m.Alloc
    }))
}
```

Gotcha

`expvar` registers its HTTP handler on `http.DefaultServeMux`. If your service uses a custom `ServeMux` — which it should in production — the `/debug/vars` endpoint is not automatically registered on it. Either mount it explicitly with `http.Handle("/debug/vars", expvar.Handler())`, or expose a separate debug server on a non-public port that uses `DefaultServeMux`.

See also

`net/http/pprof` · `runtime` · `sync/atomic`

flag

System

The `flag` package parses command-line flags. It supports bool, int, int64, uint, uint64, float64, string, and duration values, and allows custom types via the `Value` interface. Flags can be defined as pointers — `flag.String(...)` returns `*string` — or bound to existing variables via `flag.StringVar(...)`. After all flags are defined, `flag.Parse()` processes `os.Args[1:]` and populates the values. Non-flag arguments are available via `flag.Args()` after parsing.

Types

Type	Purpose
FlagSet	An independent set of flags with its own parse method; used for subcommands
Flag	Metadata for a single flag: name, usage, default, and current value
Value	Interface for custom flag types: String() string and Set(string) error

Functions and methods

The package-level functions (`flag.String`, `flag.Int`, `flag.Bool` etc.) operate on a default `FlagSet`. For subcommands — where each command has its own set of flags — create explicit `FlagSet` instances with `flag.NewFlagSet`.

```
flag.String(name, value, usage string) *string
flag.Int(name string, value int, usage string) *int
flag.Bool(name string, value bool, usage string) *bool
flag.Duration(name string, value time.Duration, usage string)
*time.Duration
```

Define flags returning pointers. Dereference after `Parse`.

```
flag.StringVar(p *string, name, value, usage string)
```

Binds a flag to an existing variable. Preferred when the variable is declared alongside the flag definition for clarity.

```
flag.Parse()
```

Parses `os.Args[1:]`. Must be called after all flags are defined and before any flag values are used.

```
flag.Args() []string
```

Returns non-flag arguments remaining after `Parse`. Use for positional arguments — file paths, subcommand names.

```
flag.Usage
```

A `func()` variable that prints usage and exits. Override it to customise the help output.

Example

```
func main() {
    addr := flag.String("addr", ":8080", "listen address")
    workers := flag.Int("workers", runtime.NumCPU(), "number
of worker goroutines")
    verbose := flag.Bool("v", false, "enable verbose logging")
    timeout := flag.Duration("timeout", 30*time.Second,
"request timeout")

    flag.Usage = func() {
        fmt.Fprintf(os.Stderr, "Usage: %s [flags]
<input-file>\n\n", os.Args[0])
        flag.PrintDefaults()
    }
    flag.Parse()

    if flag.NArg() < 1 {
        flag.Usage()
```

```
        os.Exit(1)
    }
    inputFile := flag.Arg(0)
    _ = addr; _ = workers; _ = verbose; _ = timeout; _ =
inputFile
}
```

Gotcha

Flags must be defined before `flag.Parse()` is called, and `Parse` must be called before any flag values are read. This is enforced by convention, not the compiler — defining a flag after `Parse` registers it but it will never be set from the command line, and reading a flag value before `Parse` returns only the default. Both mistakes are silent.

See also

`os` · `fmt` · `time`

fmt

I/O

The `fmt` package implements formatted I/O using verbs modelled on C's printf family. It is the entry point for almost everything a Go developer writes to a terminal, a log, or an error value. The verbs fall into two categories: general (`%v`, `%+v`, `%#v`, `%T`) which work on any type and call the value's `String()` or `GoString()` method if available, and type-specific (`%d`, `%f`, `%s`, `%x`, `%p`) which format a value in a specific representation. `fmt.Errorf` with `%w` is the standard way to wrap errors with context.

Types

Type	Purpose
Stringer	Interface: String() string — implemented to control %v and %s formatting
GoStringer	Interface: GoString() string — implemented to control %#v formatting
Formatter	Interface for full custom verb handling

Functions and methods

The function family follows a naming convention: no prefix writes to stdout; `F` prefix writes to an `io.Writer`; `S` prefix returns a string; `Errorf` returns an `error`.

```
fmt.Printf(format string, a ...any)
fmt.Fprintf(w io.Writer, format string, a ...any)
fmt.Sprintf(format string, a ...any) string
```

The core formatted output functions. `Fprintf` is the most general — it works with any `io.Writer`, including `os.Stderr`, `bytes.Buffer`, `net.Conn`, and `http.ResponseWriter`.

```
fmt.Println(a ...any)
fmt.Fprintln(w io.Writer, a ...any)
```

Writes values separated by spaces with a trailing newline. No format string.

```
fmt.Errorf(format string, a ...any) error
```

Creates a formatted error. Use `%w` to wrap an existing error — the result supports `errors.Is` and `errors.As` unwrapping.

```
fmt.Sscanf(str, format string, a ...any) (n int, err error)
```

Scans formatted input from a string. Useful for parsing structured text without a full parser.

Verbs worth knowing

Verb	Meaning
%v	Default format for any type
%+v	Struct with field names
%#v	Go syntax representation
%T	Type name
%w	Wrap error (Errorf only)
%x	Hex encoding of bytes or integers
%q	Quoted string or rune

Example

```
// Wrap errors with context at each layer
func loadConfig(path string) (*Config, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return nil, fmt.Errorf("loadConfig %s: %w", path, err)
    }
```

```
    var cfg Config
    if err := json.Unmarshal(data, &cfg); err != nil {
        return nil, fmt.Errorf("loadConfig %s: parsing: %w",
path, err)
    }
    return &cfg, nil
}

// Implement Stringer for clean log output
type RequestID struct {
    Service string
    ID      uint64
}

func (r RequestID) String() string {
    return fmt.Sprintf("%s/%016x", r.Service, r.ID)
}
```

Gotcha

`fmt.Println` and `fmt.Printf` write to `os.Stdout`, which is line-buffered when connected to a terminal but fully buffered when redirected to a file or pipe. Output that appears correctly on the terminal may arrive out of order or be lost entirely when the program exits without flushing. For any output that must be reliable regardless of destination, write to a `bufio.Writer` and flush explicitly, or use `log` which flushes on every call.

See also

`errors` · `io` · `log/slog`

hash/crc32

Hashing

The `hash/crc32` package computes CRC-32 checksums — fast, non-cryptographic cyclic redundancy checks used for error detection in file formats and network protocols. CRC-32 is not a cryptographic hash: it is trivially reversible and collisions can be constructed deliberately. Use it only for integrity checking against accidental corruption, not against adversarial tampering. The package provides multiple polynomial tables because different formats use different CRC variants — ZIP and gzip use IEEE, iSCSI uses Castagnoli.

Types

Type	Purpose
Hash32	The hash.Hash32 interface implemented by CRC-32 values
Table	A precomputed polynomial table; required by most functions

Functions and methods

```
MakeTable(poly uint32) *Table
```

Precomputes a polynomial table. Use the package constants `IEEE`, `Castagnoli`, or `Koopman` rather than raw polynomial values. Tables are expensive to compute — create once and reuse.

```
NewIEEE() hash.Hash32
```

Returns a new CRC-32 hash using the IEEE polynomial. Equivalent to `New(IEEETable)` and the most common choice for general use.

```
Checksum(data []byte, tab *Table) uint32
```

One-shot CRC-32 of a byte slice. Faster than creating a `Hash32` when all data is available at once.

```
Update(crc uint32, tab *Table, p []byte) uint32
```

Extends an existing CRC value with additional data. Use for incremental computation without creating a `Hash32`.

Example

```
// Verify file integrity using CRC-32
var ieeeTable = crc32.MakeTable(crc32.IEEE)

func fileChecksum(path string) (uint32, error) {
    f, err := os.Open(path)
    if err != nil {
        return 0, err
    }
    defer f.Close()

    h := crc32.NewIEEE()
    if _, err := io.Copy(h, f); err != nil {
        return 0, err
    }
    return h.Sum32(), nil
}

// Incremental checksum over multiple chunks
func chunkedChecksum(chunks [][]byte) uint32 {
    var crc uint32
    for _, chunk := range chunks {
        crc = crc32.Update(crc, ieeeTable, chunk)
    }
    return crc
}
```

Gotcha

The IEEE, Castagnoli, and Koopman polynomials produce different checksums for identical input. A checksum computed with one polynomial cannot be verified with another. ZIP files use IEEE; iSCSI and ext4 use Castagnoli. If you are producing a checksum to be verified by an external system, confirm which polynomial that system expects before choosing a table.

See also

`hash/fnv` · `crypto/sha256` · `io`

hash/fnv

Hashing

The `hash/fnv` package implements the FNV (Fowler–Noll–Vo) family of non-cryptographic hash functions. FNV hashes are fast, produce well-distributed output, and have no setup cost — there are no tables to precompute. They are used when you need a quick, consistent hash for partitioning, sharding, consistent hashing rings, or building in-memory hash structures where cryptographic properties are irrelevant. The package provides 32-bit, 64-bit, and 128-bit variants in both FNV-1 and FNV-1a forms; FNV-1a is preferred for better avalanche behaviour.

Types

Type	Purpose
Hash32	32-bit FNV hash implementing hash.Hash32
Hash64	64-bit FNV hash implementing hash.Hash64

Functions and methods

```
New32a() hash.Hash32
New64a() hash.Hash64
New128a() hash.Hash
```

Returns FNV-1a hash instances. The `a` suffix denotes FNV-1a — prefer these over `New32`, `New64`, and `New128` for new code.

Write bytes to the hash with `Write`, retrieve the result with `Sum32()` or `Sum64()`. The hash implements `io.Writer`, so it composes with `fmt.Fprintf` for hashing formatted values without an intermediate allocation.

Example

```
// Shard a key across N buckets consistently
func shard(key string, n int) int {
    h := fnv.New64a()
    h.Write([]byte(key))
    return int(h.Sum64() % uint64(n))
}

// Hash multiple fields without string concatenation
func hashRequest(method, path, body string) uint64 {
    h := fnv.New64a()
    fmt.Fprintf(h, "%s\x00%s\x00%s", method, path, body)
    return h.Sum64()
}
```

Gotcha

FNV is not suitable for hash tables exposed to untrusted input. An adversary who knows you are using FNV can craft keys that all hash to the same bucket, degrading a hash table from O(1) to O(n) — a hash flooding attack. For maps keyed by user-controlled strings in a long-running server, use Go's built-in map (which uses a randomised hash seed per process) rather than implementing your own hash table with FNV.

See also

`hash/crc32` · `crypto/hmac` · `crypto/rand`

html

Text

The `html` package provides two functions: `EscapeString` and `UnescapeString`. That is the entirety of its public API. `EscapeString` replaces the five characters that have special meaning in HTML — `<`, `>`, `&`, `'`, `"` — with their entity equivalents. It exists for the cases where you are constructing HTML fragments outside of `html/template` and need to safely embed user-controlled text. If you are generating HTML, prefer `html/template`, which applies escaping automatically and context-aware — this package requires you to remember to call it.

Functions and methods

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

Replaces `<`, `>`, `&`, `'`, and `"` with HTML entities. Safe for embedding in HTML text nodes and attribute values.

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

Converts HTML entities back to their character equivalents. Handles named entities, decimal references (`"`), and hex references (`"`).

Example

```
// Safe embedding in a hand-constructed HTML fragment
func buildMeta(description string) string {
    return fmt.Sprintf(
        `<meta name="description" content="%s">`,
        html.EscapeString(description),
    )
}

// Decode entity-encoded text from an external API
func cleanText(raw string) string {
```

```
    return html.UnescapeString(raw)
}
```

Gotcha

`EscapeString` escapes for HTML text and attribute contexts but is not sufficient for all HTML contexts. It does not make a string safe for embedding inside a `<script>` tag, a `style` attribute, or a URL attribute like `href`. For those contexts, escaping rules are different and more complex. If you are generating HTML with any dynamic content, use `html/template`, which understands context and applies the correct escaping automatically.

See also

`html/template` · `text/template` · `strings`

html/template

Text

The `html/template` package provides HTML templating with automatic context-aware escaping. It uses the same syntax as `text/template` but understands the structure of HTML — it applies different escaping rules depending on whether a value appears in an HTML text node, an attribute value, a URL, a CSS property, or a JavaScript expression. This prevents XSS injection by design rather than by discipline: the template engine does the right thing regardless of what a developer remembers to escape manually. For any server-rendered HTML output, this package is the correct choice over `text/template`.

Types

Type	Purpose
Template	A parsed and compiled HTML template, safe for concurrent execution
FuncMap	A map of function names to functions available within templates
HTML	A string type that marks content as safe HTML, bypassing escaping
URL	A string type that marks a URL as trusted
JS	A string type that marks content as safe JavaScript

Functions and methods

```
template.Must(t *Template, err error) *Template
```

Wraps `New(...).Parse(...)` for package-level template variables. Panics if `err` is non-nil — appropriate at init time where a template parse failure is a programming error, not a runtime condition.

```
template.ParseFS(fs fs.FS, patterns ...string) (*Template,
error)
template.ParseFiles(filenames ...string) (*Template, error)
template.ParseGlob(pattern string) (*Template, error)
```

Load templates from files, an embedded filesystem, or a glob pattern. `ParseFS` with `embed.FS` is the idiomatic pattern for production — templates travel with the binary.

```
(*Template).Execute(wr io.Writer, data any) error
(*Template).ExecuteTemplate(wr io.Writer, name string, data
any) error
```

Render a template to a writer. `ExecuteTemplate` selects a named template from a set — use when multiple templates are parsed together as a layout system.

```
(*Template).Funcs(funcMap FuncMap) *Template
```

Registers custom functions available in templates. Must be called before parsing.

Example

```
//go:embed templates/*
var templateFS embed.FS

var templates = template.Must(
    template.New("").Funcs(template.FuncMap{
        "formatDate": func(t time.Time) string {
            return t.Format("2 January 2006")
        },
    }).ParseFS(templateFS, "templates/*.html"),
)

type PageData struct {
    Title   string
    User    *User
    Items   []Item
}
```

```
func renderPage(w http.ResponseWriter, data PageData) error {
    w.Header().Set("Content-Type", "text/html; charset=utf-8")
    return templates.ExecuteTemplate(w, "page.html", data)
}
```

Gotcha

The `template.HTML`, `template.URL`, and `template.JS` types mark content as trusted and bypass the automatic escaping. They exist for cases where the developer has already sanitised content and the template engine's escaping would double-encode it. Using them on user-controlled input defeats the XSS protection entirely. Treat these types the same way you would treat a SQL query with user input directly interpolated — only apply them to values you have explicitly verified are safe.

See also

`text/template` · `html` · `embed`

io

I/O

The `io` package defines the core interfaces for streaming I/O in Go: `Reader`, `Writer`, `Closer`, `Seeker`, and their combinations. Everything in the standard library that moves data — files, network connections, HTTP bodies, compressed streams, archive readers — does so through these interfaces. The package also provides a set of utility functions that compose readers and writers: copying between them, limiting how much can be read, reading into multiple writers simultaneously, and piping data between goroutines. Understanding `io` is understanding how data moves in Go.

Types

Type	Purpose
Reader	Read(p []byte) (n int, err error) — the fundamental input interface
Writer	Write(p []byte) (n int, err error) — the fundamental output interface
Closer	Close() error — signals end of use and releases resources
ReadWriter	Combines Reader and Writer
ReadCloser	Combines Reader and Closer; the type of http.Response.Body
WriteCloser	Combines Writer and Closer
ReadSeeker	Combines Reader and Seeker
LimitedReader	A Reader that stops after N bytes
SectionReader	Reads a segment of a ReaderAt by offset and length

PipeReader / PipeWriter	Connected pair for goroutine-to-goroutine streaming

Functions and methods

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

Copies from src to dst until EOF or error. Allocates a 32KB buffer internally. The workhorse of data movement in Go — used to stream file contents, proxy HTTP bodies, feed compressors.

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

Copies exactly n bytes. Returns an error if fewer than n bytes are available.

```
io.ReadAll(r Reader) ([]byte, error)
```

Reads all remaining bytes from r into memory. Convenient for small bodies; use `io.Copy` to a file or buffer for large ones.

```
io.LimitReader(r Reader, n int64) Reader
```

Wraps r to stop reading after n bytes. Essential for any code that reads from an untrusted source — without a limit, a malicious or malfunctioning sender can exhaust memory.

```
io.TeeReader(r Reader, w Writer) Reader
```

Returns a Reader that writes everything it reads to w. Use to inspect a stream while passing it through — logging a request body while also decoding it.

```
io.Pipe() (*PipeReader, *PipeWriter)
```

Creates a synchronous in-memory pipe. Writes block until reads consume the data. Use to connect a goroutine that produces data to

one that consumes it without buffering the whole payload.

```
io.MultiReader(readers ...Reader) Reader
```

Concatenates readers sequentially. Use to prepend or append data to a stream without copying.

```
io.MultiWriter(writers ...Writer) Writer
```

Fans out writes to multiple writers simultaneously. One write goes to all of them.

Example

```
// Proxy an HTTP response while computing its checksum
func proxyWithChecksum(w http.ResponseWriter, resp
*http.Response) (string, error) {
    defer resp.Body.Close()

    // Limit to 10MB to prevent memory exhaustion
    limited := io.LimitReader(resp.Body, 10<<20)

    h := sha256.New()
    // TeeReader feeds both the hasher and the response writer
    tee := io.TeeReader(limited, h)

    if _, err := io.Copy(w, tee); err != nil {
        return "", err
    }
    return hex.EncodeToString(h.Sum(nil)), nil
}
```

Gotcha

`io.ReadAll` reads until EOF with no size limit. Calling it on an HTTP request body, a network connection, or any externally-controlled reader without first wrapping it in `io.LimitReader` is a memory exhaustion vulnerability. A sender that never sends EOF, or sends an enormous payload, will grow the process heap until the OS kills it. Wrap with `io.LimitReader` before passing any untrusted reader to `ReadAll`.

See also

`bufio` · `os` · `bytes`

io/fs

I/O

The `io/fs` package defines a read-only filesystem interface introduced in Go 1.16. Its central type, `FS`, abstracts over any hierarchical collection of files — an `embed.FS`, an `os.DirFS`, a `zip.Reader`, or a custom implementation. Code written against `fs.FS` works unchanged across all of them, which makes it straightforward to switch between reading from disk in development and reading from an embedded binary in production. The package also defines `WalkDir`, which replaces `filepath.Walk` when the filesystem is abstract rather than OS-native.

Types

Type	Purpose
FS	Core interface: Open(name string) (File, error)
ReadFileFS	Extends FS with ReadFile(name string) ([]byte, error)
ReadDirFS	Extends FS with ReadDir(name string) ([]DirEntry, error)
GlobFS	Extends FS with Glob(pattern string) ([]string, error)
File	An open file: Read, Close, Stat
DirEntry	Metadata for a directory entry without opening the file
FileInfo	File metadata: name, size, mode, modification time

Functions and methods

```
fs.WalkDir(fsys FS, root string, fn WalkDirFunc) error
```

Walks the filesystem tree rooted at `root`, calling `fn` for each entry. Uses `DirEntry` rather than `FileInfo` — reading metadata without opening files is cheaper, especially on remote or embedded filesystems.

```
fs.ReadFile(fsys FS, name string) ([]byte, error)
```

Reads the named file from any `FS`. Uses the `ReadFileFS` fast path if available, falls back to `Open` otherwise.

```
fs.Glob(fsys FS, pattern string) ([]string, error)
```

Returns names matching the pattern. Same syntax as `filepath.Glob`.

```
os.DirFS(dir string) fs.FS
```

Defined in `os`, not `io/fs` — wraps a directory on disk as an `fs.FS`. Use this to make disk-based code accept the `fs.FS` interface and become testable with in-memory or embedded filesystems.

Example

```
//go:embed assets
var assetFS embed.FS

// Works with embed.FS in production, os.DirFS in development
func loadTemplate(fsys fs.FS, name string)
(*template.Template, error) {
    return template.ParseFS(fsys, "assets/templates/"+name)
}

// Walk any filesystem to find files by extension
func findByExt(fsys fs.FS, ext string) ([]string, error) {
    var found []string
    err := fs.WalkDir(fsys, ".", func(path string, d
fs.DirEntry, err error) error {
        if err != nil {
            return err
        }
```

```
        if !d.IsDir() && filepath.Ext(path) == ext {
            found = append(found, path)
        }
        return nil
    })
    return found, err
}
```

Gotcha

`fs.FS` path names always use forward slashes, regardless of operating system, and must not begin with a slash or contain `.` or `..` elements. Passing a path constructed with `filepath.Join` on Windows — which uses backslashes — will fail. Use `path.Join` (not `filepath.Join`) when building paths for use with `fs.FS`.

See also

`embed` · `os` · `path/filepath`

log

Diagnostics

The `log` package provides a simple logger that writes timestamped lines to `os.Stderr` by default. It is the right choice for CLIs, short-lived tools, and programs where structured output is not required. Each log call is goroutine-safe. The package-level functions (`log.Print`, `log.Printf`, `log.Println`) write to a default logger; `log.New` creates an independent logger with its own output, prefix, and flags. `log.Fatal` and `log.Fatalf` log and then call `os.Exit(1)` — use them only at the top level of a program, not inside library code.

Types

Type	Purpose
Logger	An independent logger with its own writer, prefix, and flag set

Functions and methods

```
log.SetFlags(flag int)
```

Controls which fields appear in each log line. Combine constants with `|`: `log.Ldate`, `log.Ltime`, `log.Lmicroseconds`, `log.Llongfile`, `log.Lshortfile`, `log.LUTC`, `log.Lmsgprefix`. The default is `log.LstdFlags` which is `Ldate | Ltime`.

```
log.SetPrefix(prefix string)
log.SetOutput(w io.Writer)
```

Configure the default logger's prefix string and output destination.

```
log.New(out io.Writer, prefix string, flag int) *Logger
```

Creates an independent logger. Use when different components need separate prefixes or output destinations.

```
log.Fatal(v ...any)
log.Fatalf(format string, v ...any)
```

Log and call `os.Exit(1)`. Deferred functions do not run. Never call from library code — only from `main` or top-level program logic.

Example

```
// Component-specific loggers with distinct prefixes
var (
    infoLog  = log.New(os.Stdout, "INFO  ",
log.Ldate|log.Ltime|log.Lshortfile)
    errorLog = log.New(os.Stderr, "ERROR ",
log.Ldate|log.Ltime|log.Lshortfile)
)

func processJob(id string) error {
    infoLog.Printf("starting job %s", id)
    if err := doWork(id); err != nil {
        errorLog.Printf("job %s failed: %v", id, err)
        return err
    }
    infoLog.Printf("job %s complete", id)
    return nil
}
```

Gotcha

`log.Fatal` and `log.Panic` are convenient but dangerous in anything other than `main`. `Fatal` calls `os.Exit(1)`, which bypasses all deferred functions — database connections are not closed, in-flight writes are not flushed, cleanup code does not run. In library or middleware code, return errors instead. Reserve `Fatal` for the point in `main` where the program cannot continue and a clean shutdown is not possible.

See also

log/slog · fmt · os

log/slog

Diagnostics

The `log/slog` package provides structured, levelled logging introduced in Go 1.21. Where `log` produces unstructured text lines, `slog` produces key-value records that can be emitted as JSON or logfmt and consumed by log aggregation systems. The package defines a `Logger` type backed by a `Handler` interface — the two built-in handlers are `TextHandler` and `JSONHandler`, and custom handlers can implement any output format or routing logic. A package-level default logger, configurable with `slog.SetDefault`, covers most use cases without passing a logger explicitly.

Types

Type	Purpose
Logger	The logging front-end; creates log records and passes them to a Handler
Handler	Interface for processing log records: JSONHandler, TextHandler, or custom
Record	A single log event with time, level, message, and attributes
Attr	A key-value pair attached to a log record
Level	Log severity: Debug, Info, Warn, Error
LevelVar	An atomically adjustable Level; use for runtime log level changes

Functions and methods

```
slog.SetDefault(l *Logger)
```

Sets the package-level default logger. Call once at program startup. Also redirects the output of the `log` package to the new logger, which unifies structured and unstructured logging in codebases that use both.

```
slog.New(h Handler) *Logger
```

Creates a logger backed by the given handler.

```
slog.NewJSONHandler(w io.Writer, opts *HandlerOptions)
*JSONHandler
slog.NewTextHandler(w io.Writer, opts *HandlerOptions)
*TextHandler
```

The two built-in handlers. `HandlerOptions` sets the minimum level and an optional `ReplaceAttr` function for rewriting or redacting fields before output.

```
(*Logger).Info(msg string, args ...any)
(*Logger).Error(msg string, args ...any)
(*Logger).With(args ...any) *Logger
```

Log at a level. Alternating key-value pairs after `msg` are attached as attributes. `With` returns a new logger with pre-attached attributes — use it to bind request-scoped fields (trace ID, user ID) to a logger at the start of a request.

```
(*Logger).InfoContext(ctx context.Context, msg string, args
...any)
```

The context variants pass the context to the handler, enabling handlers to extract trace IDs or span information from the context automatically.

Example

```
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout,
&slog.HandlerOptions{
        Level: slog.LevelInfo,
        ReplaceAttr: func(groups []string, a slog.Attr)
slog.Attr {
```

```
            if a.Key == "password" || a.Key == "token" {
                return slog.String(a.Key, "[REDACTED]")
            }
            return a
        },
    }))
    slog.SetDefault(logger)
    http.HandleFunc("/", handleRequest)
    http.ListenAndServe(":8080", nil)
}

// Bind request-scoped fields to a child logger at the handler
boundary
func handleRequest(w http.ResponseWriter, r *http.Request) {
    reqLogger := slog.With(
        "trace_id", r.Header.Get("X-Trace-Id"),
        "method", r.Method,
        "path", r.URL.Path,
    )
    reqLogger.InfoContext(r.Context(), "request received")
    // reqLogger carries trace_id, method, and path on every
subsequent call
}
```

Gotcha

Passing key-value pairs as alternating `any` arguments — `slog.Info("msg", "key", value)` — is convenient but loses type safety. Mismatched pairs (an odd number of arguments, or a non-string key) produce a malformed log record at runtime with no compile-time warning. For structured logging where correctness matters, use explicit `slog.Attr` values: `slog.Info("msg", slog.String("key", value), slog.Int("count", n))`. The `slog.String`, `slog.Int`, `slog.Duration` helpers make this readable without verbosity.

See also

`log` · `context` · `io`

maps

Data

The `maps` package provides generic operations on maps, introduced in Go 1.21. Before it existed, extracting keys, cloning a map, or deleting entries by predicate required a for-range loop every time. The package makes these operations single calls. It is small by design — it covers the common cases and leaves specialised map logic to the caller.

Types

None. The package exports functions only, all generic over `map[K]V`.

Functions and methods

```
maps.Keys[M ~map[K]V, K comparable, V any](m M) []K
```

Returns the keys of m in an unspecified order. The order is not guaranteed to be consistent across calls.

```
maps.Values[M ~map[K]V, K comparable, V any](m M) []V
```

Returns the values of m in an unspecified order.

```
maps.Clone[M ~map[K]V, K comparable, V any](m M) M
```

Returns a shallow copy of m. Values are not deep-copied — if the values are pointers or contain pointers, both maps reference the same underlying objects.

```
maps.Copy[M1 ~map[K]V, M2 ~map[K]V, K comparable, V any](dst
M1, src M2)
```

Copies all key-value pairs from src into dst, overwriting existing keys.

```
maps.DeleteFunc[M ~map[K]V, K comparable, V any](m M, del
func(K, V) bool)
```

Deletes all entries for which `del` returns true. Mutates m in place.

```
maps.Equal[M1, M2 ~map[K]V, K, V comparable](m1 M1, m2 M2)
bool
```

Reports whether two maps have identical key-value pairs.

Example

```
// Merge configuration maps with environment overrides taking
precedence
func mergeConfig(base, overrides map[string]string)
map[string]string {
    result := maps.Clone(base)
    maps.Copy(result, overrides)
    return result
}

// Extract and sort keys for deterministic output
func sortedKeys(m map[string]int) []string {
    keys := maps.Keys(m)
    slices.Sort(keys)
    return keys
}

// Remove expired cache entries
func evictExpired(cache map[string]CacheEntry, now time.Time)
{
    maps.DeleteFunc(cache, func(_ string, e CacheEntry) bool {
        return e.ExpiresAt.Before(now)
    })
}
```

Gotcha

`maps.Keys` and `maps.Values` return slices in unspecified order — and that order may differ between calls on the same map. Code that assumes a consistent order, such as building a deterministic hash or

producing sorted output, must sort the result explicitly. Go's map iteration order is intentionally randomised to prevent developers from depending on it.

See also

`slices` · `sort` · `cmp`

math

Mathematics

The `math` package provides mathematical constants and functions operating on `float64`. It covers the functions a developer reaches for without thinking: absolute value, rounding, square root, logarithms, trigonometry, and the special values `Inf` and `NaN`. The constants `math.Pi`, `math.E`, `math.Phi`, and `math.Sqrt2` are available at full `float64` precision. For integer arithmetic beyond the range of `int64`, see `math/big`.

Types

None. The package exports constants and functions only.

Functions and methods

```
math.Abs(x float64) float64
math.Round(x float64) float64
math.Floor(x float64) float64
math.Ceil(x float64) float64
```

Absolute value and rounding. `Round` rounds half away from zero; `Floor` and `Ceil` round toward negative and positive infinity respectively.

```
math.Sqrt(x float64) float64
math.Pow(x, y float64) float64
math.Log(x float64) float64
math.Log2(x float64) float64
math.Log10(x float64) float64
```

Power and logarithm functions. `Log` is the natural logarithm. `Pow(x, 0.5)` is equivalent to `Sqrt(x)` but slower.

```
math.Min(x, y float64) float64
math.Max(x, y float64) float64
```

Minimum and maximum of two float64 values. For integers, use the built-in `min` and `max` added in Go 1.21.

```
math.IsNaN(f float64) bool
math.IsInf(f float64, sign int) bool
math.NaN() float64
math.Inf(sign int) float64
```

Special value handling. `IsNaN` and `IsInf` are the correct way to test for these values — comparing with == does not work for NaN, since `NaN != NaN` by IEEE 754 definition.

Example

```
// Clamp a float64 to a range
func clamp(v, lo, hi float64) float64 {
    return math.Min(hi, math.Max(lo, v))
}

// Safe logarithm that handles non-positive input
func safeLog(x float64) (float64, error) {
    if x <= 0 || math.IsNaN(x) {
        return 0, fmt.Errorf("log undefined for %v", x)
    }
    return math.Log(x), nil
}

// Compute the number of bits needed to represent n
func bitsNeeded(n int) int {
    if n <= 0 {
        return 0
    }
    return int(math.Floor(math.Log2(float64(n)))) + 1
}
```

Gotcha

`NaN != NaN` is true by IEEE 754 — a NaN value is not equal to itself. Testing a float64 for NaN with `f == math.NaN()` always returns false, even when `f` is NaN. Use `math.IsNaN(f)` instead. The same applies to comparisons involving NaN: any comparison with NaN (<, >, <=, >=)

returns false, which can cause sorting and searching algorithms to behave incorrectly when NaN values are present in the input.

See also

`math/big` · `math/bits` · `math/rand/v2`

math/big

Mathematics

The `math/big` package implements arbitrary-precision arithmetic for integers, rational numbers, and floating-point values. It exists for situations where the fixed-size integer and float types overflow or lose precision: cryptographic key generation, financial calculations where rounding must be controlled exactly, factorial and combinatorial computations, and any algorithm that needs to work with numbers larger than 2^63. The types are mutable and method-based — operations modify the receiver and return it, enabling chaining. Allocation is minimised by reusing existing values rather than creating new ones on every operation.

Types

Type	Purpose
Int	Arbitrary-precision integer
Float	Arbitrary-precision floating-point with configurable precision and rounding
Rat	Arbitrary-precision rational number (numerator/denominator pair)

Functions and methods

`Int` is the most commonly used type. Operations follow a receiver-result pattern: `z.Add(x, y)` sets z to x+y and returns z. This allows allocation reuse — declare one `*Int` and use it as the accumulator throughout a computation rather than creating a new value for each step.

```
new(big.Int).SetInt64(n int64) *Int
new(big.Int).SetString(s string, base int) (*Int, bool)
```

Construct an `Int` from a Go integer or a string in a given base. `SetString` returns false if parsing fails.

```
(*Int).Add(x, y *Int) *Int
(*Int).Mul(x, y *Int) *Int
(*Int).Mod(x, y *Int) *Int
(*Int).Exp(x, y, m *Int) *Int
```

Arithmetic operations. `Exp` computes modular exponentiation efficiently — essential for RSA and other public-key algorithms.

```
(*Int).BitLen() int
```

Returns the number of bits in the absolute value of `x`. Used to validate key sizes in cryptographic code.

```
(*Int).Bytes() []byte
(*Int).SetBytes(buf []byte) *Int
```

Convert between `*Int` and a big-endian unsigned byte slice. The standard serialisation format for cryptographic integers.

Example

```
// Compute a large factorial exactly
func factorial(n int64) *big.Int {
    result := new(big.Int).SetInt64(1)
    for i := int64(2); i <= n; i++ {
        result.Mul(result, big.NewInt(i))
    }
    return result
}

// Modular exponentiation for RSA-like operations
func modExp(base, exp, mod *big.Int) *big.Int {
    return new(big.Int).Exp(base, exp, mod)
}

fmt.Println(factorial(50))
//
30414093201713378043612608166979581188299763898377856000000000
```

```
000
```

Gotcha

`big.Int` operations modify the receiver in place. Passing the same `*Int` as both an argument and the receiver — `x.Add(x, x)` — is defined and correct for `Add` and `Mul`, but not for all operations. `x.Div(x, x)` and similar self-referential calls on division and modulo operations produce undefined results because the implementation reads inputs and writes the receiver using overlapping memory. Use a separate accumulator variable when in doubt.

See also

`math` · `crypto/rsa` · `encoding/binary`

math/bits

Mathematics

The `math/bits` package provides bit manipulation operations on unsigned integers: counting leading and trailing zeros, counting set bits (popcount), bit length, rotation, and reversals. These operations map directly to hardware instructions on most architectures — the Go compiler recognises them and emits the corresponding CPU instruction rather than a software loop. They are used in compression algorithms, hash functions, binary protocol parsing, and any code that works with packed bit fields. Before this package existed, developers either wrote slow portable implementations or used unsafe assembly tricks.

Types

None. The package exports functions only.

Functions and methods

```
bits.Len(x uint) int
bits.Len8(x uint8) int
bits.Len16(x uint16) int
bits.Len32(x uint32) int
bits.Len64(x uint64) int
```

Returns the minimum number of bits required to represent x. `bits.Len(0)` returns 0; `bits.Len(1)` returns 1; `bits.Len(255)` returns 8.

```
bits.OnesCount(x uint) int
```

Returns the number of set bits in x (population count / popcount). Compiles to a single `POPCNT` instruction on x86.

```
bits.LeadingZeros(x uint) int
bits.TrailingZeros(x uint) int
```

Count of leading and trailing zero bits. `TrailingZeros` on a power of two gives its log base 2.

```
bits.RotateLeft(x uint, k int) uint
bits.RotateLeft32(x uint32, k int) uint32
```

Rotates x left by k bits. Negative k rotates right. Used in hash and cipher implementations.

```
bits.Add64(x, y, carry uint64) (sum, carryOut uint64)
bits.Mul64(x, y uint64) (hi, lo uint64)
```

Carry-aware addition and full 128-bit multiplication. Used when implementing big integer arithmetic or checked arithmetic without `math/big`.

Example

```
// Next power of two at or above n
func nextPow2(n uint) uint {
    if n == 0 {
        return 1
    }
    return 1 << bits.Len(n-1)
}

// Check if n is a power of two
func isPow2(n uint) bool {
    return n > 0 && bits.OnesCount(n) == 1
}

// Extract a bit field from a packed uint32
func extractField(v uint32, offset, width uint) uint32 {
    return (v >> offset) & ((1 << width) - 1)
}
```

Gotcha

`bits.Len(x)` returns 0 when x is 0. Code using `Len` to compute a log base 2 or determine an index into a power-of-two structure will produce an off-by-one or a zero result for the zero input, which may silently

corrupt data structures. Always handle the zero case explicitly before calling `Len`.

See also

`math` · `math/big` · `encoding/binary`

math/rand/v2

Mathematics

The `math/rand/v2` package provides pseudo-random number generation, introduced in Go 1.22 as a replacement for the original `math/rand`. It uses stronger algorithms — PCG and ChaCha8 — is automatically seeded with a random value at program startup, and has a cleaner API that removes the legacy global functions that required explicit seeding. It is appropriate for any use case where statistical randomness is sufficient and predictability is not a security concern: simulations, shuffling, sampling, load testing, game logic, and generating test data. For security-sensitive randomness, use `crypto/rand`.

Types

Type	Purpose
Rand	A source of pseudo-random numbers backed by a Source
Source	Interface for random number sources
PCG	Permuted congruential generator — fast, small state
ChaCha8	ChaCha8-based generator — cryptographically stronger, reproducible from seed

Functions and methods

Package-level functions use a shared, automatically-seeded global source. For reproducible sequences — tests, simulations that must replay — create an explicit `*Rand` with a fixed seed using `New(NewPCG(seed1, seed2))`.

```
rand.IntN(n int) int
rand.Int64N(n int64) int64
rand.Float64() float64
```

Generate random values in ranges. `IntN(n)` returns a value in [0, n). These replace the old `Intn`, `Int63n`, and `Float64` functions with unbiased implementations.

```
rand.N[T intType](n T) T
```

Generic version of `IntN` working across all integer types. Prefer this for type-safe code.

```
rand.Shuffle(n int, swap func(i, j int))
```

Shuffles n elements using the Fisher-Yates algorithm. Pass the swap function for your slice.

```
rand.New(src Source) *Rand
rand.NewPCG(seed1, seed2 uint64) *PCG
rand.NewChaCha8(seed [32]byte) *ChaCha8
```

Create a seeded source for reproducible sequences.

Example

```
// Shuffle a slice in place
func shuffleItems[T any](items []T) {
    rand.Shuffle(len(items), func(i, j int) {
        items[i], items[j] = items[j], items[i]
    })
}

// Reproducible random sequence for a test
func TestSimulation(t *testing.T) {
    src := rand.New(rand.NewPCG(42, 0))
    for i := range 10 {
        t.Logf("step %d: %d", i, src.IntN(100))
    }
}
```

```
// Weighted random selection
func weightedChoice(weights []float64) int {
    total := 0.0
    for _, w := range weights {
        total += w
    }
    r := rand.Float64() * total
    for i, w := range weights {
        r -= w
        if r <= 0 {
            return i
        }
    }
    return len(weights) - 1
}
```

Gotcha

`math/rand/v2` and the original `math/rand` coexist in the module system but are separate packages with different APIs. Code that imports both — common during migration — must be careful about which package's functions are called. Since Go 1.20, the global source in `math/rand` is also automatically seeded, but the v1 API remains — `Intn`, `Int63n`, deprecated seeding functions — and coexists alongside v2's cleaner `IntN`, `N[T]`, and stronger sources. Do not mix imports from both packages in the same file without aliased imports to make the distinction explicit at every call site.

See also

`crypto/rand` · `math` · `slices`

mime

Network

The `mime` package handles MIME type parsing and content type negotiation. Its most practical functions are `TypeByExtension`, which maps a file extension to its MIME type, and `ParseMediaType`, which splits a content type header into its type and parameters. Any code that serves files over HTTP, processes file uploads, or inspects content type headers without a full HTTP framework will use these functions. The database of extension-to-type mappings is sourced from the operating system and supplemented by built-in defaults.

Types

None exported directly. The package exports functions and works with plain strings and maps.

Functions and methods

```
mime.TypeByExtension(ext string) string
```

Returns the MIME type for a file extension including the leading dot — `mime.TypeByExtension(".json")` returns `"application/json"`. Returns an empty string if the extension is unknown.

```
mime.ExtensionsByType(typ string) ([]string, error)
```

Returns the file extensions associated with a MIME type. A type may have multiple extensions — `"image/jpeg"` maps to `[".jpeg", ".jpg"]`.

```
mime.ParseMediaType(v string) (mediatype string, params
map[string]string, err error)
```

Parses a MIME media type value as found in a `Content-Type` or `Content-Disposition` header. Returns the type and a map of

parameters such as `charset` or `boundary`.

```
mime.FormatMediaType(t string, params map[string]string)
string
```

Constructs a media type string from a type and parameter map. Handles quoting and escaping correctly.

Example

```
// Serve a file with the correct Content-Type
func serveFile(w http.ResponseWriter, path string) error {
    data, err := os.ReadFile(path)
    if err != nil {
        return err
    }
    ext := filepath.Ext(path)
    contentType := mime.TypeByExtension(ext)
    if contentType == "" {
        contentType = "application/octet-stream"
    }
    w.Header().Set("Content-Type", contentType)
    _, err = w.Write(data)
    return err
}

// Parse a multipart boundary from a Content-Type header
func extractBoundary(contentType string) (string, error) {
    _, params, err := mime.ParseMediaType(contentType)
    if err != nil {
        return "", fmt.Errorf("parsing content-type: %w", err)
    }
    boundary, ok := params["boundary"]
    if !ok {
        return "", errors.New("no boundary parameter")
    }
    return boundary, nil
}
```

Gotcha

`mime.TypeByExtension` returns an empty string for unknown extensions — it does not return a default or an error. Code that uses the return value directly as a `Content-Type` header without checking for an empty string will send a blank header, which causes some clients to reject the response or treat the content as plain text. Always fall back to `"application/octet-stream"` for unknown types.

See also

`mime/multipart` · `net/http` · `path/filepath`

mime/multipart

Network

The `mime/multipart` package reads and writes MIME multipart data — the format used for HTML form file uploads (`multipart/form-data`) and multipart email bodies. On the read side, `NewReader` produces a `Reader` that iterates over parts; on the write side, `NewWriter` produces a `Writer` that generates correctly-bounded parts. In HTTP servers, `(*http.Request).ParseMultipartForm` and `(*http.Request).FormFile` wrap this package, so direct use is only needed when implementing custom multipart handling or working outside of `net/http`.

Types

Type	Purpose
Reader	Iterates over parts of a multipart stream
Writer	Produces a multipart stream with generated boundaries
Part	A single part of a multipart message; implements io.Reader
FileHeader	Metadata for an uploaded file from ParseMultipartForm
Form	Parsed multipart form with values and files

Functions and methods

```
multipart.NewReader(r io.Reader, boundary string) *Reader
```

Creates a reader for a multipart stream. The boundary comes from the `Content-Type` header — use `mime.ParseMediaType` to extract it.

```
(*Reader).NextPart() (*Part, error)
```

Advances to the next part. Returns `io.EOF` when all parts are consumed. Each `Part` is an `io.Reader` for the part's body and carries its headers.

```
(*Reader).ReadForm(maxMemory int64) (*Form, error)
```

Reads all parts into a `Form`. Parts up to `maxMemory` bytes are stored in memory; larger parts are written to temporary files.

```
multipart.NewWriter(w io.Writer) *Writer
(*Writer).CreateFormFile(fieldname, filename string)
(io.Writer, error)
(*Writer).CreateFormField(fieldname string) (io.Writer, error)
(*Writer).Close() error
```

Write multipart data. `CreateFormFile` sets the `Content-Disposition` and `Content-Type` headers for a file part. Always call `Close` — it writes the terminating boundary.

Example

```
// Upload a file to a multipart endpoint
func uploadFile(url, fieldName, filePath string) error {
    f, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer f.Close()

    var body bytes.Buffer
    w := multipart.NewWriter(&body)

    part, err := w.CreateFormFile(fieldName,
filepath.Base(filePath))
    if err != nil {
        return err
    }
    if _, err := io.Copy(part, f); err != nil {
        return err
```

```
    }
    w.Close() // must close before reading body

    req, _ := http.NewRequest("POST", url, &body)
    req.Header.Set("Content-Type", w.FormDataContentType())
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    return nil
}
```

Gotcha

`(*multipart.Writer).Close()` must be called before the body buffer is read or transmitted. `Close` writes the final boundary delimiter that signals the end of the multipart stream. A receiver that reads a multipart body without the terminating boundary will either block waiting for more data or return a parse error. The `body` buffer must not be passed to the HTTP client until after `w.Close()` returns.

See also

`mime` · `net/http` · `io`

net

Network

The `net` package provides the low-level networking primitives that everything else in the network stack builds on: TCP and UDP connections, Unix domain sockets, IP address manipulation, and DNS resolution. `net/http` uses it internally; you reach for `net` directly when building custom protocols, implementing a raw TCP server, or doing anything that HTTP does not cover. The `Dial` and `Listen` functions are the two entry points — dial to connect, listen to accept.

Types

Type	Purpose
Conn	A generic network connection implementing io.Reader, io.Writer, and io.Closer
Listener	A network listener that accepts incoming connections
Dialer	Configurable dialer with timeout, local address, and keep-alive settings
TCPConn	A TCP connection with TCP-specific methods like SetNoDelay
UDPConn	A UDP connection for datagram-oriented communication
IP	An IP address as a byte slice, with parsing and formatting methods
IPNet	An IP network (CIDR block) with containment testing
Addr	Interface representing a network address

Functions and methods

```
net.Dial(network, address string) (Conn, error)
```

Connects to address on the given network. `network` is `"tcp"`, `"tcp4"`, `"tcp6"`, `"udp"`, or `"unix"`. Returns a `Conn` that satisfies `io.ReadWriteCloser` — read from and write to it like any other stream.

```
net.Listen(network, address string) (Listener, error)
```

Opens a listening socket. Call `Accept` in a loop to handle connections, each in its own goroutine.

```
net.DialTimeout(network, address string, timeout
time.Duration) (Conn, error)
```

Like `Dial` but with a connection timeout. For production code, prefer a `Dialer` with `DialContext` so the timeout is cancellable.

```
(*Dialer).DialContext(ctx context.Context, network, address
string) (Conn, error)
```

The production-correct way to dial — the context carries cancellation and deadline, and the connection attempt aborts when the context is done.

```
net.ParseIP(s string) IP
net.ParseCIDR(s string) (IP, *IPNet, error)
```

Parse IP addresses and CIDR ranges. `ParseCIDR` returns both the host address and the network address — use `IPNet.Contains` to test membership.

Example

```
// Simple TCP echo server
func runEchoServer(ctx context.Context, addr string) error {
    ln, err := net.Listen("tcp", addr)
    if err != nil {
```

```
        return err
    }
    go func() {
        <-ctx.Done()
        ln.Close()
    }()

    for {
        conn, err := ln.Accept()
        if err != nil {
            if ctx.Err() != nil {
                return nil // shutdown
            }
            return err
        }
        go func(c net.Conn) {
            defer c.Close()
            // net.Conn implements both io.Reader and
io.Writer;
            // copying it to itself echoes all input back to
the sender
            io.Copy(c, c)
        }(conn)
    }
}

// Check if an IP is within an allowed range
func isAllowed(ipStr string, allowList []string) bool {
    ip := net.ParseIP(ipStr)
    if ip == nil {
        return false
    }
    for _, cidr := range allowList {
        _, network, err := net.ParseCIDR(cidr)
        if err == nil && network.Contains(ip) {
            return true
        }
    }
    return false
}
```

Gotcha

`net.Conn` has no read or write deadline by default — a read on a connection with no data will block indefinitely. In any server that handles untrusted clients, always set deadlines with `(*Conn).SetDeadline`, `SetReadDeadline`, or `SetWriteDeadline` before reading. A slow or silent client that never sends data will otherwise hold a goroutine forever, exhausting the server's resources.

See also

`net/http` · `context` · `io`

net/http

Network

The `net/http` package provides a complete HTTP/1.1 and HTTP/2 client and server. The server side — `http.ListenAndServe`, `http.Handle`, `http.HandleFunc` — runs a production-capable HTTP server out of the box. The client side — `http.Get`, `http.Post`, `http.Client` — makes HTTP requests with connection pooling, redirect handling, and TLS managed automatically. The package is designed around the `http.Handler` interface, a single method — `ServeHTTP(ResponseWriter, *Request)` — which makes middleware composition natural and testable.

Types

Type	Purpose
Handler	Interface: ServeHTTP(ResponseWriter, *Request)
HandlerFunc	Function type implementing Handler; adapts a function to the interface
ServeMux	HTTP request multiplexer; routes requests to handlers by pattern
Server	Configurable HTTP server with timeouts, TLS, and shutdown support
Client	Configurable HTTP client with transport, timeout, and redirect policy
Request	An incoming server request or outgoing client request
Response	The response from an HTTP client request

ResponseWriter	Interface for writing an HTTP response from a handler
Transport	Low-level HTTP transport; manages connection pooling and TLS

Functions and methods

The package-level `http.Handle`, `http.HandleFunc`, and `http.ListenAndServe` use a default `ServeMux` and `Server`. For production, create explicit `*Server` and `*ServeMux` instances — the defaults have no timeouts, which makes them vulnerable to slowloris and similar attacks.

```
http.NewServeMux() *ServeMux
(*ServeMux).Handle(pattern string, handler Handler)
(*ServeMux).HandleFunc(pattern string, handler
func(ResponseWriter, *Request))
```

Register handlers. Since Go 1.22, patterns support method and wildcard routing: `"GET /users/{id}"` matches GET requests and binds `{id}` as a path parameter retrieved with `r.PathValue("id")`.

```
(*Server).ListenAndServe() error
(*Server).ListenAndServeTLS(certFile, keyFile string) error
(*Server).Shutdown(ctx context.Context) error
```

Start and stop the server. `Shutdown` stops accepting new connections and waits for active requests to complete, up to the context deadline — the correct pattern for graceful shutdown.

```
http.NewRequest(method, url string, body io.Reader) (*Request,
error)
(*Client).Do(req *Request) (*Response, error)
```

The client entry points. Always create an explicit `*Client` with a `Timeout` set — the `http.DefaultClient` has no timeout and will wait indefinitely for a response.

```
(*Response).Body io.ReadCloser
```

Always close `resp.Body`, even if you do not read it. Unclosed bodies prevent connection reuse, exhausting the transport's connection pool.

Example

```
// Production server with timeouts and graceful shutdown
func runServer(ctx context.Context, addr string, handler
http.Handler) error {
    srv := &http.Server{
        Addr:         addr,
        Handler:      handler,
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
        IdleTimeout:  120 * time.Second,
    }

    go func() {
        <-ctx.Done()
        shutCtx, cancel :=
context.WithTimeout(context.Background(), 30*time.Second)
        defer cancel()
        srv.Shutdown(shutCtx)
    }()

    if err := srv.ListenAndServe(); err !=
http.ErrServerClosed {
        return err
    }
    return nil
}

// Middleware pattern
func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        slog.Info("request", "method", r.Method, "path",
r.URL.Path,
            "ms", time.Since(start).Milliseconds())
```

```
    })
}
```

Gotcha

`http.ListenAndServe` and `http.DefaultClient` have no timeouts. A server without `ReadTimeout` and `WriteTimeout` is vulnerable to connection exhaustion from slow clients. A client without `Timeout` will wait indefinitely for a response — one hung upstream service will cascade into goroutine accumulation and eventual process death. Set timeouts explicitly on both the server and any client your code creates. The defaults are wrong for production.

See also

`net/url` · `context` · `crypto/tls`

net/http/httptest

Network

The `net/http/httptest` package provides two tools for testing HTTP code without a real network: `ResponseRecorder`, which captures what a handler writes, and `Server`, which runs a real HTTP server on a local port for integration testing. Together they cover the full range of HTTP testing needs — unit testing individual handlers, and end-to-end testing of code that makes real HTTP requests. Both work with the standard `testing` package and require no configuration.

Types

Type	Purpose
ResponseRecorder	Captures status code, headers, and body written by a handler
Server	A real local HTTP server for integration and client testing

Functions and methods

```
httptest.NewRecorder() *ResponseRecorder
```

Creates a `ResponseRecorder` that implements `http.ResponseWriter`. Pass it to a handler under test, then inspect `Code`, `Header()`, and `Body` on the recorder.

```
(*ResponseRecorder).Result() *http.Response
```

Returns the recorded response as an `*http.Response`. Use this when you need the full response object — including trailers — rather than accessing `Code` and `Body` directly.

```
httptest.NewRequest(method, target string, body io.Reader)
*http.Request
```

Creates a synthetic `*http.Request` for handler testing. Sets `r.RequestURI` correctly, which `http.NewRequest` does not — use this for handler tests, not for client tests.

```
httptest.NewServer(handler http.Handler) *Server
(*Server).Close()
(*Server).URL string
```

Starts a real HTTP server on a random local port. `URL` gives the base URL — pass it to an `http.Client` to make real requests against the handler. Always call `Close` when the test is done.

```
httptest.NewTLSServer(handler http.Handler) *Server
```

Like `NewServer` but with TLS. The server uses a self-signed certificate; use `Server.Client()` to get a pre-configured `*http.Client` that trusts it.

Example

```
// Unit test an individual handler
func TestHealthHandler(t *testing.T) {
    req := httptest.NewRequest(http.MethodGet, "/health", nil)
    rec := httptest.NewRecorder()

    HealthHandler(rec, req)

    if rec.Code != http.StatusOK {
        t.Errorf("expected 200, got %d", rec.Code)
    }
    var body map[string]string
    json.NewDecoder(rec.Body).Decode(&body)
    if body["status"] != "ok" {
        t.Errorf("unexpected body: %v", body)
    }
}
```

```
// Integration test using a real server
func TestClientIntegration(t *testing.T) {
    srv := httptest.NewServer(http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(map[string]string{"status":
"ok"})
    }))
    defer srv.Close()

    client := &MyAPIClient{BaseURL: srv.URL}
    result, err := client.Health(context.Background())
    if err != nil || result.Status != "ok" {
        t.Fatalf("unexpected result: %v, %v", result, err)
    }
}
```

Gotcha

`httptest.NewRequest` and `http.NewRequest` are not interchangeable. `http.NewRequest` is for constructing outgoing client requests — it does not set `RequestURI`, which some handlers inspect. `httptest.NewRequest` is for constructing incoming server requests in handler tests — it sets `RequestURI` correctly and sets `r.Body` to `http.NoBody` when body is nil. Using `http.NewRequest` in handler tests produces requests that subtly differ from what a real server would deliver.

See also

`net/http` · `testing` · `encoding/json`

net/http/pprof

Diagnostics

The `net/http/pprof` package exposes Go's runtime profiling data over HTTP by registering handlers on `http.DefaultServeMux` when imported. A running service with this package imported can be profiled live without restarting or recompiling — attach `go tool pprof` to the running process and capture CPU profiles, heap snapshots, goroutine dumps, and execution traces. This is one of the most powerful diagnostic capabilities in the Go ecosystem and costs nothing to include in development and staging builds.

Types

None. The package works entirely through HTTP handler registration as a side effect of import.

Functions and methods

The package registers the following endpoints on `http.DefaultServeMux` at import time:

Endpoint	Profile
/debug/pprof/	Index of available profiles
/debug/pprof/heap	Memory allocation profile
/debug/pprof/goroutine	Stack traces of all goroutines
/debug/pprof/cpu	CPU profile (collected over a duration)
/debug/pprof/trace	Execution trace
/debug/pprof/block	Goroutine blocking profile
/debug/pprof/mutex	Mutex contention profile

Example

```
import _ "net/http/pprof" // registers handlers as a side
effect

// Expose pprof on a separate debug server, never on the
public port
func startDebugServer(addr string) {
    go func() {
        // DefaultServeMux has pprof handlers registered
        if err := http.ListenAndServe(addr, nil); err != nil {
            log.Printf("debug server: %v", err)
        }
    }()
}

// Capture a heap profile from the command line:
// go tool pprof http://localhost:6060/debug/pprof/heap
//
// Capture a 30-second CPU profile:
// go tool pprof
http://localhost:6060/debug/pprof/profile?seconds=30
```

Gotcha

`net/http/pprof` registers on `http.DefaultServeMux`. If your service uses a custom `ServeMux` — which it should — the pprof endpoints are not reachable through your main server. Run a separate `http.ListenAndServe` on an internal port using `nil` as the handler (which uses `DefaultServeMux`) and restrict access to that port at the network level. Never expose pprof endpoints on a public-facing port — they reveal internal memory layout and goroutine state.

See also

`expvar` · `runtime` · `net/http`

net/smtp

Network

The `net/smtp` package implements an SMTP client for sending email. It supports PLAIN authentication, STARTTLS, and the full SMTP command set. For transactional email from an application — password resets, notifications, alerts — `net/smtp` is sufficient without a third-party library, provided your SMTP relay accepts the connection. It does not handle MIME encoding for HTML email or attachments; for those, construct the MIME message manually using `mime/multipart` and `encoding/base64`, or use a higher-level library.

Types

Type	Purpose
Client	An SMTP client connection with methods for each protocol command
Auth	Interface for SMTP authentication mechanisms
ServerInfo	Server details passed to an Auth implementation for validation

Functions and methods

```
smtp.SendMail(addr string, a Auth, from string, to []string, msg []byte) error
```

The one-call entry point. Connects to addr, authenticates with a, and sends msg from from to all recipients in to. msg must be a complete RFC 5322 message including headers — `Subject:`, `To:`, `From:`, `MIME-Version:`, and the body.

`smtp.PlainAuth(identity, username, password, host string) Auth`

PLAIN authentication. Pass an empty string for identity. The host parameter is verified against the server's hostname during authentication to prevent credential forwarding to the wrong server.

`smtp.NewClient(conn net.Conn, host string) (*Client, error)`

Creates an SMTP client over an existing connection. Use when you need control over TLS setup, connection reuse, or custom transport.

Example

```
func sendAlert(cfg SMTPConfig, subject, body string) error {
    msg := fmt.Sprintf(
        "From: %s\r\nTo: %s\r\nSubject: %s\r\nMIME-Version:
1.0\r\n"+
            "Content-Type: text/plain;
charset=utf-8\r\n\r\n%s",
        cfg.From, strings.Join(cfg.To, ", "), subject, body,
    )

    auth := smtp.PlainAuth("", cfg.Username, cfg.Password,
cfg.Host)
    return smtp.SendMail(
        fmt.Sprintf("%s:%d", cfg.Host, cfg.Port),
        auth,
        cfg.From,
        cfg.To,
        []byte(msg),
    )
}
```

Gotcha

`smtp.SendMail` constructs the message envelope from the `from` and `to` parameters, but the `From:` and `To:` headers in the message body are independent — they are what the recipient's email client displays. A mismatch between the envelope and the headers does not cause an error but can trigger spam filters or confuse recipients. Always ensure

the `From:` header in the message body matches the envelope `from` argument.

See also

`mime/multipart` · `net` · `crypto/tls`

net/url

Network

The `net/url` package parses, constructs, and manipulates URLs. It handles the encoding rules that are easy to get wrong by hand — which characters must be percent-encoded in a path versus a query string, how to correctly join a base URL with a relative reference, how to encode and decode query parameters. Any code that constructs URLs programmatically, parses them from user input, or manipulates query strings should use this package rather than string operations.

Types

Type	Purpose
URL	A parsed URL with fields for scheme, host, path, query, and fragment
Values	A map of query string parameters; url.Values is map[string][]string
Error	A URL error with the operation and the URL that caused it

Functions and methods

```
url.Parse(rawURL string) (*URL, error)
```

Parses a URL. Lenient — accepts relative references and URLs without schemes. Use `url.ParseRequestURI` when the input must be an absolute URL with a scheme.

```
(*URL).String() string
```

Reconstructs the URL string from its components. Safe to call after modifying individual fields.

```
url.JoinPath(base string, elem ...string) (string, error)
(*URL).JoinPath(elem ...string) *URL
```

Appends path elements to a base URL, handling slashes and percent-encoding correctly. Added in Go 1.19. The package-level function takes a base URL string and returns a string; the method operates on an existing `*URL` and returns a new `*URL`. Both replace the common but fragile pattern of string-concatenating path segments.

```
url.Values.Set(key, value string)
url.Values.Get(key string) string
url.Values.Encode() string
```

Manipulate query parameters. `Encode` produces a URL-encoded query string with keys in sorted order — use this to build query strings, not `fmt.Sprintf`.

```
url.QueryEscape(s string) string
url.QueryUnescape(s string) (string, error)
url.PathEscape(s string) string
```

Escape individual values for query strings or path segments. `QueryEscape` and `PathEscape` use different rules — a space is `+` in a query string and `%20` in a path.

Example

```
// Construct an API request URL safely
func buildSearchURL(base, query string, page, limit int)
(string, error) {
    u, err := url.Parse(base)
    if err != nil {
        return "", err
    }
    // (*URL).JoinPath returns a new *URL with path elements
appended
    u = u.JoinPath("search")
    q := u.Query()
    q.Set("q", query)
    q.Set("page", strconv.Itoa(page))
```

```
    q.Set("limit", strconv.Itoa(limit))
    u.RawQuery = q.Encode()
    return u.String(), nil
}

// Parse and inspect an incoming request URL
func extractTenant(r *http.Request) string {
    return r.URL.Query().Get("tenant")
}
```

Gotcha

`QueryEscape` and `PathEscape` encode spaces differently and are not interchangeable. `QueryEscape` encodes a space as `+` (valid in query strings per HTML form encoding); `PathEscape` encodes it as `%20`. Using `QueryEscape` to encode a path segment produces URLs that some servers reject or misparse. Always use the function that matches the part of the URL you are encoding.

See also

`net/http` · `strconv` · `strings`

os

System

The `os` package provides a portable interface to the operating system: file I/O, environment variables, process information, directory operations, and standard streams. It is the primary boundary between a Go program and the system it runs on. The file operations return `*os.File`, which implements `io.Reader`, `io.Writer`, `io.Seeker`, and `io.Closer`, composing naturally with the rest of the standard library. For path manipulation, use `path/filepath`; for subprocess execution, use `os/exec`.

Types

Type	Purpose
File	An open file descriptor implementing io.ReadWriteCloser and io.Seeker
FileInfo	File metadata: name, size, mode, modification time, whether it is a directory
FileMode	A file's mode and permission bits
ProcessState	The status of a completed process
PathError	An error with the operation and path that caused it

Functions and methods

```
os.Open(name string) (*File, error)
```

Opens a file for reading. The most common file operation — returns a `*File` that satisfies `io.Reader`.

```
os.Create(name string) (*File, error)
```

Creates or truncates a file for writing. For append-only or more controlled modes, use `os.OpenFile`.

```
os.OpenFile(name string, flag int, perm FileMode) (*File,
error)
```

The general file opener. Combine flag constants: `os.O_RDWR|os.O_CREATE|os.O_APPEND` opens for read-write, creates if absent, appends on write.

```
os.ReadFile(name string) ([]byte, error)
os.WriteFile(name string, data []byte, perm FileMode) error
```

One-shot file read and write. Convenient for small files; use streaming for large ones.

```
os.MkdirAll(path string, perm FileMode) error
```

Creates a directory and all necessary parents. The idempotent version of `os.Mkdir` — does not error if the directory already exists.

```
os.Remove(name string) error
os.RemoveAll(path string) error
```

Delete a file or an entire directory tree. `RemoveAll` does not error if path does not exist.

```
os.Getenv(key string) string
os.LookupEnv(key string) (string, bool)
```

Read environment variables. `LookupEnv` distinguishes between a missing variable and one set to an empty string — use it when an empty string is a valid value.

```
os.Exit(code int)
```

Terminates the process immediately with the given exit code. Deferred functions do not run. Use only in `main` or top-level CLI entry points.

Example

```
// Atomically replace a file using a temp file and rename
func writeAtomic(path string, data []byte) error {
    dir := filepath.Dir(path)
    tmp, err := os.CreateTemp(dir, ".tmp-*")
    if err != nil {
        return err
    }
    tmpName := tmp.Name()
    defer os.Remove(tmpName) // clean up if rename fails

    if _, err := tmp.Write(data); err != nil {
        tmp.Close()
        return err
    }
    if err := tmp.Close(); err != nil {
        return err
    }
    return os.Rename(tmpName, path)
}

// Read configuration with explicit missing-vs-empty
distinction
func getPort() string {
    if port, ok := os.LookupEnv("PORT"); ok {
        return port
    }
    return "8080"
}
```

Gotcha

`os.Getenv` returns an empty string for both a missing variable and a variable set to `""`. If your code needs to distinguish between "not set" and "explicitly set to empty" — a common requirement in configuration — use `os.LookupEnv`, which returns a boolean indicating presence. Relying on `Getenv` returning empty to mean "not configured" will silently

mishandle the case where an operator intentionally sets the variable to an empty string.

See also

`path/filepath` · `io` · `os/exec`

os/exec

System

The `os/exec` package runs external commands and manages their input, output, and environment. It wraps the OS process API into a `Cmd` struct whose fields control every aspect of the subprocess: standard streams, working directory, environment, and process group. The package does not invoke a shell — arguments are passed directly to the executable, which means shell metacharacters in arguments are not interpreted and do not need escaping. This makes it safe to pass user-controlled strings as arguments without the injection risks of shell invocation.

Types

Type	Purpose
Cmd	A prepared external command; configure before calling Run, Start, or Output
Error	Returned when a binary is not found on PATH
ExitError	Returned when a command exits with a non-zero status; carries ProcessState

Functions and methods

```
exec.Command(name string, arg ...string) *Cmd
```

Creates a `Cmd`. The first argument is the binary name or path; subsequent arguments are passed to the binary directly, not through a shell.

```
exec.LookPath(file string) (string, error)
```

Searches `PATH` for a binary. Use to check for tool availability before attempting to run it, or to resolve the full path for logging.

```
(*Cmd).Output() ([]byte, error)
```

Runs the command and returns its stdout. If the command exits with a non-zero status, the error is an `*ExecError` and stdout may still contain partial output.

```
(*Cmd).CombinedOutput() ([]byte, error)
```

Like `Output` but merges stderr into stdout. Useful for commands that write errors to stderr — you get the full output regardless of stream.

```
(*Cmd).Run() error
```

Runs the command and waits for it to complete. Use when you have set `Stdout` and `Stderr` explicitly rather than capturing them.

```
(*Cmd).Start() error
(*Cmd).Wait() error
```

Asynchronous execution. `Start` launches the process; `Wait` blocks until it completes and releases resources. Always call `Wait` after `Start`.

Example

```
// Run a command with context-based cancellation
func runWithContext(ctx context.Context, name string, args
...string) ([]byte, error) {
    cmd := exec.CommandContext(ctx, name, args...)
    out, err := cmd.Output()
    if err != nil {
        var exitErr *exec.ExitError
        if errors.As(err, &exitErr) {
            return nil, fmt.Errorf("%s exited %d: %s",
                name, exitErr.ExitCode(), exitErr.Stderr)
        }
        return nil, err
    }
```

```
    return out, nil
}

// Stream command output in real time
func streamCommand(ctx context.Context, w io.Writer, name
string, args ...string) error {
    cmd := exec.CommandContext(ctx, name, args...)
    cmd.Stdout = w
    cmd.Stderr = w
    return cmd.Run()
}
```

Gotcha

`exec.Command` does not invoke a shell. Writing `exec.Command("ls -la")` passes the string `"ls -la"` as the binary name and fails with a "not found" error — the space and flags are not interpreted. Shell features like pipes, redirects, and glob expansion are not available. If you genuinely need a shell, call `exec.Command("sh", "-c", shellCommand)` explicitly, and be acutely aware that any user-controlled content in `shellCommand` is a command injection vulnerability.

See also

`os` · `context` · `io`

os/signal

System

The `os/signal` package routes OS signals to Go channels. It is the standard mechanism for graceful shutdown — intercepting `SIGINT` and `SIGTERM` so the program can finish in-flight work, close connections, and flush buffers before exiting. The `signal.NotifyContext` function, added in Go 1.16, returns a context that is cancelled when a signal arrives, which integrates directly with the context-based cancellation patterns used throughout the standard library.

Types

Type	Purpose
Signal	Interface representing an OS signal; os.Signal

Functions and methods

```
signal.NotifyContext(parent context.Context, signals ...os.Signal) (ctx context.Context, stop context.CancelFunc)
```

Returns a child context that is cancelled when any of the listed signals arrive. The `stop` function unregisters the signal handlers and should be deferred. This is the idiomatic pattern for signal-driven shutdown in Go 1.16+.

```
signal.Notify(c chan<- os.Signal, sig ...os.Signal)
```

Routes the listed signals to channel c. The channel must be buffered — a signal delivered to an unbuffered channel is dropped if the receiver is not ready. Use `NotifyContext` for new code; `Notify` is useful when you need to handle signals multiple times or distinguish between different signal types.

```
signal.Stop(c chan<- os.Signal)
```

Stops routing signals to c. Call this when shutting down to avoid leaking the signal registration.

```
signal.Reset(sig ...os.Signal)
```

Resets the handling of the listed signals to the default OS behaviour. Useful before re-execing the process.

Example

```
func main() {
    ctx, stop := signal.NotifyContext(context.Background(),
        syscall.SIGINT, syscall.SIGTERM,
    )
    defer stop()

    server := &http.Server{Addr: ":8080", Handler:
buildRouter()}

    go func() {
        if err := server.ListenAndServe(); err !=
http.ErrServerClosed {
            log.Printf("server error: %v", err)
        }
    }()

    // Block until signal or other cancellation
    <-ctx.Done()
    stop() // release signal resources before shutdown

    shutCtx, cancel :=
context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()
    if err := server.Shutdown(shutCtx); err != nil {
        log.Printf("shutdown error: %v", err)
    }
}
```

Gotcha

`signal.Notify` requires a buffered channel. If the signal arrives while your goroutine is not reading from the channel, the signal is dropped silently — the program does not receive it and does not shut down. A buffer of 1 is sufficient for most uses since signals do not arrive faster than they can be processed, but it must not be zero. `NotifyContext` handles this correctly internally.

See also

`context` · `os` · `net/http`

os/user

System

The `os/user` package looks up user and group information from the operating system. It provides the current user, lookup by username or UID, and group membership queries. The package is useful in tools that need to operate relative to the running user — finding home directories, checking permissions, dropping privileges — and in services that validate or impersonate users. On most Unix systems the lookups use the native `getpwuid_r` and `getgrnam_r` system calls; on others they fall back to parsing `/etc/passwd` and `/etc/group`.

Types

Type	Purpose
User	OS user account with Uid, Gid, Username, Name, and HomeDir
Group	OS group with Gid and Name

Functions and methods

```
user.Current() (*User, error)
```

Returns the current user. The most common call — used to find the home directory or validate that the process is running as an expected user.

```
user.Lookup(username string) (*User, error)
user.LookupId(uid string) (*User, error)
```

Look up a user by name or UID. UIDs are strings, not integers, matching the underlying system representation.

```
user.LookupGroup(name string) (*Group, error)
user.LookupGroupId(gid string) (*Group, error)
```

Look up a group by name or GID.

```
(*User).GroupIds() ([]string, error)
```

Returns the GIDs of all groups the user belongs to. The result includes the primary GID from `User.Gid`.

Example

```
// Resolve a path relative to the current user's home
directory
func expandHome(path string) (string, error) {
    if !strings.HasPrefix(path, "~/") {
        return path, nil
    }
    u, err := user.Current()
    if err != nil {
        return "", fmt.Errorf("resolving home: %w", err)
    }
    return filepath.Join(u.HomeDir, path[2:]), nil
}

// Check that the process is not running as root
func requireNonRoot() error {
    u, err := user.Current()
    if err != nil {
        return err
    }
    if u.Uid == "0" {
        return errors.New("must not run as root")
    }
    return nil
}
```

Gotcha

`os/user` may not function correctly in minimal container environments — Alpine Linux images using musl libc, scratch-based images, or

containers without `/etc/passwd`. The CGO-based implementation calls into libc; the pure Go fallback parses `/etc/passwd` directly but requires that file to exist. In distroless or scratch containers, lookups will fail with errors that look like system configuration problems rather than missing files. If your containerised service needs user information, either use a base image with a full user database or pass user information through environment variables.

See also

`os` · `path/filepath` · `os/exec`

path

System

The `path` package manipulates slash-delimited paths — not OS filesystem paths, but virtual paths as they appear in URLs, import paths, and embedded filesystems. It is intentionally minimal: `Join`, `Split`, `Base`, `Dir`, `Ext`, `Clean`, and `Match`. For OS filesystem paths, use `path/filepath`, which handles OS-specific separators and volume names. For URL path manipulation, `path` is the correct choice since URLs always use forward slashes regardless of platform.

Types

None. The package exports functions only.

Functions and methods

```
path.Join(elem ...string) string
```

Joins path elements with slashes, cleaning the result. Unlike `filepath.Join`, always produces a forward-slash path regardless of OS.

```
path.Base(path string) string
```

Returns the last element of path. Equivalent to the filename portion of a URL path.

```
path.Dir(path string) string
```

Returns all but the last element of path. Always ends with a slash unless the result is `"."`.

```
path.Ext(path string) string
```

Returns the file extension including the leading dot, or an empty string if there is none.

```
path.Clean(path string) string
```

Returns the shortest equivalent path by eliminating `.`, `..`, and repeated slashes.

```
path.Match(pattern, name string) (matched bool, err error)
```

Tests whether a path matches a shell pattern. Uses the same syntax as `filepath.Match` — `*` matches any sequence of non-slash characters, `?` matches one character, `[abc]` matches a character class.

Example

```
// Build paths for an embedded filesystem
func assetPath(category, name string) string {
    return path.Join("assets", category, name)
}

// Extract the extension from a URL path for content
negotiation
func extensionFromURL(rawURL string) string {
    u, _ := url.Parse(rawURL)
    return path.Ext(u.Path)
}

// Check if an fs.FS path matches a pattern
func matchesPattern(fsPath, pattern string) bool {
    matched, _ := path.Match(pattern, fsPath)
    return matched
}
```

Gotcha

`path.Join` and `filepath.Join` produce identical output on Unix for most inputs, which makes the mistake easy to miss in development. The difference surfaces on Windows, where `filepath.Join` handles drive letters, UNC paths, and backslash separators — none of which `path.Join` understands. Code that uses `path.Join` for filesystem paths will silently produce wrong results on Windows. The rule is simple: `path` for virtual paths and URLs, `filepath` for anything touching the OS

filesystem.

See also

`path/filepath` · `io/fs` · `net/url`

path/filepath

System

The `path/filepath` package manipulates OS filesystem paths with full awareness of the platform's separator and volume conventions. On Unix it uses forward slashes; on Windows it uses backslashes and handles drive letters and UNC paths. `WalkDir` traverses a directory tree, calling a function for each entry — it is the standard way to process files recursively. For virtual paths in URLs or embedded filesystems, use `path` instead.

Types

Type	Purpose
WalkDirFunc	Function signature for WalkDir callbacks

Functions and methods

```
filepath.Join(elem ...string) string
```

Joins path elements using the OS separator and cleans the result. The most common path operation — always prefer this over string concatenation.

```
filepath.Abs(path string) (string, error)
```

Returns the absolute path by resolving relative references against the current working directory.

```
filepath.Rel(basepath, targpath string) (string, error)
```

Returns a relative path from `basepath` to `targpath`. Useful for producing portable relative references in configs and archives.

```
filepath.Dir(path string) string
filepath.Base(path string) string
filepath.Ext(path string) string
```

Directory, filename, and extension components. `Ext` includes the leading dot.

```
filepath.Glob(pattern string) ([]string, error)
```

Returns paths matching a shell pattern. Searches the real filesystem — for embedded or abstract filesystems use `fs.Glob`.

```
filepath.WalkDir(root string, fn fs.WalkDirFunc) error
```

Walks the directory tree rooted at `root`. Calls `fn` for each file and directory in lexical order. Return `filepath.SkipDir` from `fn` to skip a directory without stopping the walk, or `filepath.SkipAll` (Go 1.20) to stop the entire walk early.

```
filepath.Match(pattern, name string) (matched bool, err error)
```

Tests whether a path matches a shell pattern.

Example

```
// Find all Go source files under a directory
func findGoFiles(root string) ([]string, error) {
    var files []string
    err := filepath.WalkDir(root, func(path string, d
fs.DirEntry, err error) error {
        if err != nil {
            return err
        }
        if d.IsDir() && d.Name() == "vendor" {
            return filepath.SkipDir
        }
        if !d.IsDir() && filepath.Ext(path) == ".go" {
            files = append(files, path)
        }
        return nil
```

```
    })
    return files, err
}

// Safely expand a user-provided path relative to a base
directory
func safePath(base, userInput string) (string, error) {
    joined := filepath.Join(base, userInput)
    abs, err := filepath.Abs(joined)
    if err != nil {
        return "", err
    }
    if !strings.HasPrefix(abs,
filepath.Clean(base)+string(filepath.Separator)) {
        return "", errors.New("path escapes base directory")
    }
    return abs, nil
}
```

Gotcha

`filepath.Join` silently drops leading slashes in path elements after the first. `filepath.Join("/base", "/user/input")` produces `/base/user/input`, not `/user/input`. This means a user-supplied absolute path passed as a second argument to `Join` does not override the base — but it also means you cannot use `Join` alone to detect path traversal. Always check that the resulting absolute path still begins with the intended base directory, as shown in `safePath` above.

See also

`path` · `os` · `io/fs`

regexp

Text

The `regexp` package implements regular expressions using RE2 syntax, which guarantees linear-time matching in the size of the input. Unlike PCRE-based engines, RE2 does not support backtracking constructs — no lookahead, no backreferences. This is a deliberate tradeoff: RE2 cannot be made to exhibit catastrophic backtracking, which makes it safe for use with untrusted input. Compile patterns once at package initialisation; `regexp.MustCompile` panics on an invalid pattern, which is appropriate for patterns embedded in source code.

Types

Type	Purpose
Regexp	A compiled regular expression; safe for concurrent use

Functions and methods

```
regexp.Compile(expr string) (*Regexp, error)
regexp.MustCompile(str string) *Regexp
```

Compile a pattern. `MustCompile` is for package-level variables where a bad pattern is a programming error. `Compile` is for patterns derived from input or configuration.

```
(*Regexp).MatchString(s string) bool
```

Reports whether s contains any match.

```
(*Regexp).FindString(s string) string
(*Regexp).FindAllString(s string, n int) []string
```

Return the first or all matches. Pass `n = -1` to `FindAllString` for all matches.

```
(*Regexp).FindStringSubmatch(s string) []string
```

Returns the full match and all captured subgroups. Index 0 is the whole match; indices 1+ are the capture groups in order.

```
(*Regexp).ReplaceAllString(src, repl string) string
(*Regexp).ReplaceAllStringFunc(src string, repl func(string)
string) string
```

Replace matches with a literal string or a function. In `repl`, `$1` refers to the first capture group; `${name}` refers to a named group.

```
(*Regexp).Split(s string, n int) []string
```

Splits s by the pattern. Analogous to `strings.SplitN`.

Example

```
// Extract structured data from log lines
var logPattern = regexp.MustCompile(
    `^(\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}Z)\s+(\w+)\s+(.+)$`,
)

type LogEntry struct {
    Time    string
    Level   string
    Message string
}

func parseLogLine(line string) (LogEntry, bool) {
    m := logPattern.FindStringSubmatch(line)
    if m == nil {
        return LogEntry{}, false
    }
    return LogEntry{Time: m[1], Level: m[2], Message: m[3]},
true
}
```

```
// Redact sensitive values in a config string
var secretPattern =
regexp.MustCompile(`(?i)(password|token|secret)\s*=\s*\S+`)

func redactSecrets(config string) string {
    return secretPattern.ReplaceAllStringFunc(config,
func(match string) string {
        eq := strings.Index(match, "=")
        return match[:eq+1] + " [REDACTED]"
    })
}
```

Gotcha

`FindStringSubmatch` returns `nil` when there is no match, not an empty slice. Code that indexes into the result without checking for nil — `m[1]` where m is nil — panics. Always check `if m == nil` before accessing submatch indices. The same applies to `FindSubmatch`, `FindStringSubmatchIndex`, and the other submatch variants.

See also

`strings` · `strconv` · `unicode`

runtime

System

The `runtime` package exposes introspection into the Go runtime: goroutine management, garbage collection, memory statistics, and stack traces. Most application code never calls it directly — it is primarily used in diagnostics, performance tuning, and tooling. The functions that do appear in application code are `runtime.GOMAXPROCS`, `runtime.NumCPU`, `runtime.NumGoroutine`, and `runtime.ReadMemStats`. The package also provides `runtime.Caller` and `runtime.Callers` for capturing stack frame information in custom error and logging implementations.

Types

Type	Purpose
MemStats	Memory allocator statistics: heap size, GC counts, allocation rates
Frame	A single stack frame with function name, file, and line number

Functions and methods

```
runtime.GOMAXPROCS(n int) int
```

Sets the number of OS threads that can execute Go code simultaneously. Returns the previous value. Pass `runtime.NumCPU()` to use all available cores — this is the default since Go 1.5 and rarely needs to be called explicitly.

```
runtime.NumGoroutine() int
```

Returns the number of goroutines that currently exist. Useful as a metric to detect goroutine leaks — a count that grows monotonically over time

without plateauing indicates goroutines that are not terminating.

```
runtime.ReadMemStats(m *MemStats)
```

Populates m with current memory statistics. Stops the world briefly to collect accurate data — do not call in hot paths. Use for periodic diagnostic snapshots.

```
runtime.GC()
```

Triggers a garbage collection cycle. Rarely needed in application code — the runtime manages GC automatically. Useful in benchmarks to establish a clean heap state before timing.

```
runtime.Caller(skip int) (pc uintptr, file string, line int,
ok bool)
```

Returns the file and line number of the calling function, skipping `skip` frames. Used in custom logging and error types to capture call site information.

```
runtime.Stack(buf []byte, all bool) int
```

Writes a stack trace to buf. `all = true` includes all goroutines. Useful for dumping goroutine state on receipt of a signal.

Example

```
// Periodic goroutine leak detector
func monitorGoroutines(ctx context.Context, interval
time.Duration) {
    ticker := time.NewTicker(interval)
    defer ticker.Stop()
    prev := runtime.NumGoroutine()
    for {
        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            current := runtime.NumGoroutine()
```

```
            if current > prev*2 {
                slog.Warn("goroutine count doubled",
                    "previous", prev, "current", current)
            }
            prev = current
        }
    }
}

// Dump all goroutine stacks on SIGUSR1
func dumpGoroutines() {
    buf := make([]byte, 1<<20)
    n := runtime.Stack(buf, true)
    os.Stderr.Write(buf[:n])
}
```

Gotcha

`runtime.ReadMemStats` stops all goroutines to collect consistent statistics. Calling it frequently — in a per-request handler, for example — introduces stop-the-world pauses that degrade throughput under load. Collect memory statistics on a timer at intervals of seconds or minutes, not on the critical path. For lightweight per-request metrics, use `expvar` counters maintained by the application code itself.

See also

`runtime/debug` · `net/http/pprof` · `expvar`

runtime/debug

Diagnostics

The `runtime/debug` package provides runtime debugging utilities that are not part of the core `runtime` package: stack traces for individual goroutines, control over GC behaviour, memory limit enforcement, and build information. Its most practically useful function is `debug.ReadBuildInfo`, which returns the module path, Go version, and dependency list embedded in the binary at compile time — useful for exposing version information in health endpoints or logs without maintaining a separate version variable.

Types

Type	Purpose
BuildInfo	Module path, Go version, and dependency information embedded in the binary
Module	A single module dependency with path, version, and checksum

Functions and methods

```
debug.ReadBuildInfo() (*BuildInfo, bool)
```

Returns build information embedded by the Go toolchain. Returns false if the binary was not built with module support.
`BuildInfo.Main.Version` is the module version;
`BuildInfo.GoVersion` is the Go toolchain version.

```
debug.SetMemoryLimit(limit int64) int64
```

Sets a soft memory limit for the Go runtime, introduced in Go 1.19. When the heap approaches the limit, the GC runs more aggressively to

stay within it. Useful in container environments where the OS will OOM-kill the process if memory is exceeded. Pass `math.MaxInt64` to remove the limit.

```
debug.SetGCPercent(percent int) int
```

Controls GC trigger threshold. The default of 100 means GC runs when live heap doubles. Lower values trigger more frequent GC with lower peak memory; higher values reduce GC frequency at the cost of higher peak memory.

```
debug.Stack() []byte
```

Returns the stack trace of the current goroutine as a byte slice. More convenient than `runtime.Stack` for capturing a single goroutine's trace in a log or error.

```
debug.PrintStack()
```

Writes the current goroutine's stack trace to stderr. Useful in panic recovery handlers.

Example

```
// Expose build and version information in a health endpoint
func healthHandler(w http.ResponseWriter, r *http.Request) {
    info, ok := debug.ReadBuildInfo()
    if !ok {
        http.Error(w, "build info unavailable",
http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]string{
        "version":    info.Main.Version,
        "go_version": info.GoVersion,
        "path":       info.Main.Path,
    })
}
```

```
// Set a container-aware memory limit at startup
func init() {
    // Limit to 512MB; adjust to match container memory
allocation
    debug.SetMemoryLimit(512 << 20)
}
```

Gotcha

`debug.ReadBuildInfo` returns `(nil, false)` when the binary was built without module support — typically via `go build` in GOPATH mode, or from a test binary built with certain flags. Do not assume the function always succeeds. In production, build info is reliably available, but in test environments and CI pipelines using non-standard build configurations it may not be. Always check the boolean before accessing the returned `*BuildInfo`.

See also

`runtime` · `net/http/pprof` · `log/slog`

slices

Data

The `slices` package provides generic operations on slices, introduced in Go 1.21. It eliminates the boilerplate of common slice tasks that previously required either a for loop or a dependency on a utility library: sorting, searching, filtering, deduplication, insertion, and deletion. All functions are generic — they work on slices of any type that satisfies the relevant constraints. For sorted slices, the binary search functions replace the manual `sort.Search` pattern with a direct, readable call.

Types

None. The package exports functions only, all generic over `[]E`.

Functions and methods

```
slices.Sort[S ~[]E, E cmp.Ordered](x S)
slices.SortFunc[S ~[]E, E any](x S, cmp func(a, b E) int)
slices.SortStableFunc[S ~[]E, E any](x S, cmp func(a, b E)
int)
```

Sort a slice in place. `Sort` works on any ordered type; `SortFunc` takes a comparison function returning -1, 0, or 1 — use `cmp.Compare` for single-field comparison and compose multiple calls for multi-key sorting.

```
slices.Contains[S ~[]E, E comparable](s S, v E) bool
slices.Index[S ~[]E, E comparable](s S, v E) int
```

Linear search. `Contains` reports presence; `Index` returns the first index or -1.

```
slices.BinarySearch[S ~[]E, E cmp.Ordered](x S, target E)
(int, bool)
slices.BinarySearchFunc[S ~[]E, E, T any](x S, target T, cmp
func(E, T) int) (int, bool)
```

Binary search on a sorted slice. Returns the index and whether an exact match was found. When no match exists, the returned index is where `target` would be inserted to maintain order.

```
slices.Compact[S ~[]E, E comparable](s S) S
slices.CompactFunc[S ~[]E, E any](s S, eq func(E, E) bool) S
```

Remove consecutive duplicate elements. For deduplication of all duplicates, sort first then compact.

```
slices.Reverse[S ~[]E, E any](s S)
```

Reverses the elements of s in place.

```
slices.Insert[S ~[]E, E any](s S, i int, v ...E) S
slices.Delete[S ~[]E, E any](s S, i, j int) S
```

Insert elements at index i; delete elements from index i to j (exclusive). Both return a new slice header — reassign the result.

```
slices.Equal[S ~[]E, E comparable](s1, s2 S) bool
slices.Clone[S ~[]E, E any](s S) S
```

Equality check and shallow copy.

Example

```
type Event struct {
    Time     time.Time
    Severity int
    Message  string
}

// Sort events by severity descending, then time ascending
func sortEvents(events []Event) {
    slices.SortFunc(events, func(a, b Event) int {
        if n := cmp.Compare(b.Severity, a.Severity); n != 0 {
            return n
        }
        return a.Time.Compare(b.Time)
    })
```

```
}

// Deduplicate a sorted list of strings
func dedupe(sorted []string) []string {
    return slices.Compact(sorted)
}

// Find the insertion point for a new event by time
func insertionPoint(events []Event, t time.Time) int {
    idx, _ := slices.BinarySearchFunc(events, t, func(e Event,
target time.Time) int {
        return e.Time.Compare(target)
    })
    return idx
}
```

Gotcha

`slices.Delete` modifies the underlying array of the slice and returns a new slice header with a shorter length. The original slice variable still points to the same array — elements beyond the new length are not zeroed, and if the slice contains pointers, those elements continue to hold references that prevent garbage collection. After deleting pointer-containing elements, zero the tail explicitly: `clear(s[len(result):])` before discarding the original slice.

See also

`sort` · `cmp` · `maps`

sort

Data

The `sort` package sorts slices and user-defined collections. Since Go 1.21, `slices.Sort` and `slices.SortFunc` are the preferred entry points for slice sorting — they are generic, require no interface implementation, and are generally faster. The `sort` package remains relevant for `sort.Search` (binary search on any sorted sequence via a predicate), `sort.Stable` (sort preserving original order of equal elements via the `Interface`), and sorting custom collection types that are not slices.

Types

Type	Purpose
Interface	Three-method interface for sorting custom collections: Len, Less, Swap

Functions and methods

```
sort.Slice(x any, less func(i, j int) bool)
sort.SliceStable(x any, less func(i, j int) bool)
```

Sort a slice with a comparison function. `Slice` does not preserve the relative order of equal elements; `SliceStable` does. For new code, prefer `slices.SortFunc`.

```
sort.Search(n int, f func(int) bool) int
```

Binary search via predicate. Returns the smallest index i in [0, n) for which `f(i)` is true, assuming f is false for smaller indices and true for larger ones. Returns n if f is never true. The general-purpose binary search that works on any sorted sequence, not just slices.

```
sort.Ints(x []int)
sort.Strings(x []string)
sort.Float64s(x []float64)
```

Convenience sort functions for common types. Equivalent to `slices.Sort` for those types.

```
sort.IntsAreSorted(x []int) bool
sort.StringsAreSorted(x []string) bool
```

Check whether a slice is already sorted. Useful in tests and precondition checks.

Example

```
// Binary search on a custom sorted structure
type IntervalTree []Interval

type Interval struct {
    Start, End int
}

func (t IntervalTree) findFirst(point int) (Interval, bool) {
    // Find the first interval whose End >= point
    i := sort.Search(len(t), func(i int) bool {
        return t[i].End >= point
    })
    if i < len(t) && t[i].Start <= point {
        return t[i], true
    }
    return Interval{}, false
}

// Sort a slice of structs — still readable with sort.Slice
sort.SliceStable(records, func(i, j int) bool {
    return records[i].Priority > records[j].Priority
})
```

Gotcha

`sort.Search` requires that the predicate transitions from false to true exactly once across the range — it must be monotone. Passing a predicate that is true, then false, then true again produces an undefined result — `Search` may return any index. Ensure the slice or structure is actually sorted by the same criterion the predicate tests before calling `Search`.

See also

`slices` · `cmp` · `maps`

strconv

Text

The `strconv` package converts between strings and primitive types: integers, floats, booleans, and quoted strings. It is the correct tool for type conversion when `fmt.Sprintf` and `fmt.Sscanf` are too heavy — `strconv` functions have no formatting overhead and return typed errors. The two functions developers reach for most are `strconv.Atoi` (string to int) and `strconv.Itoa` (int to string). For float parsing and formatting with precision control, `strconv.ParseFloat` and `strconv.FormatFloat` provide the full range of options.

Types

Type	Purpose
NumError	Error returned by parsing functions; carries the function name, input, and error code

Functions and methods

```
strconv.Atoi(s string) (int, error)
strconv.Itoa(i int) string
```

The most common conversions. `Atoi` is equivalent to `ParseInt(s, 10, 0)` but returns a plain `int`.

```
strconv.ParseInt(s string, base, bitSize int) (int64, error)
strconv.ParseUint(s string, base, bitSize int) (uint64, error)
```

Parse integers with explicit base (2–36) and bit size (0, 8, 16, 32, 64). Use `base = 0` to infer from prefix: `0x` for hex, `0o` for octal, `0b` for binary.

```
strconv.FormatInt(i int64, base int) string
strconv.FormatUint(i uint64, base int) string
```

Format integers in any base. `FormatInt(n, 16)` is the efficient way to produce hex strings without `fmt.Sprintf`.

```
strconv.ParseFloat(s string, bitSize int) (float64, error)
strconv.FormatFloat(f float64, fmt byte, prec, bitSize int)
string
```

Parse and format floating-point values. `fmt` controls notation: `'f'` for decimal, `'e'` for scientific, `'g'` for shortest representation. `prec = -1` in `FormatFloat` uses the minimum number of digits to represent the value exactly.

```
strconv.ParseBool(str string) (bool, error)
strconv.FormatBool(b bool) string
```

Parse and format booleans. Accepts `"1"`, `"t"`, `"true"`, `"TRUE"` and their false equivalents.

```
strconv.Quote(s string) string
strconv.Unquote(s string) (string, error)
```

Add and remove Go string literal quoting, including escape sequences. Useful when generating code or parsing config formats that use Go-style string literals.

Example

```
// Parse a config value with a meaningful error
func parsePort(s string) (int, error) {
    n, err := strconv.Atoi(s)
    if err != nil {
        return 0, fmt.Errorf("invalid port %q: %w", s, err)
    }
    if n < 1 || n > 65535 {
        return 0, fmt.Errorf("port %d out of range [1,
65535]", n)
    }
```

```
    return n, nil
}

// Format a float with controlled precision for display
func formatPrice(price float64) string {
    return strconv.FormatFloat(price, 'f', 2, 64)
}

// Parse a hex color code
func parseHexColor(s string) (r, g, b uint8, err error) {
    s = strings.TrimPrefix(s, "#")
    n, err := strconv.ParseUint(s, 16, 32)
    if err != nil {
        return 0, 0, 0, fmt.Errorf("invalid color %q: %w", s,
err)
    }
    return uint8(n >> 16), uint8(n >> 8), uint8(n), nil
}
```

Gotcha

`strconv.ParseFloat` accepts strings like `"NaN"`, `"Inf"`, and `"+Inf"` without error. Code that parses user-supplied float values and then uses them in arithmetic or comparisons without checking for NaN or infinity will produce silent calculation errors or comparison failures. After parsing, check with `math.IsNaN` and `math.IsInf` if the input source is untrusted or the downstream arithmetic cannot tolerate special values.

See also

`strings` · `fmt` · `math`

strings

Text

The `strings` package provides all string manipulation operations: searching, splitting, joining, trimming, replacing, and case conversion. Its `Builder` type constructs strings incrementally without the allocation cost of repeated concatenation. The package mirrors `bytes` for byte slices — almost every function has a counterpart in `bytes` that operates on `[]byte`. For Unicode-aware operations at the rune level, use `unicode` and `unicode/utf8` alongside `strings`.

Types

Type	Purpose
Builder	Accumulates strings efficiently; implements io.Writer
Reader	Reads from a string as an io.Reader, io.Seeker, and io.ReaderAt
Replacer	Performs multiple string replacements in a single pass

Functions and methods

```
strings.Contains(s, substr string) bool
strings.HasPrefix(s, prefix string) bool
strings.HasSuffix(s, suffix string) bool
strings.Count(s, substr string) int
strings.Index(s, substr string) int
```

Searching. `Index` returns -1 when not found.

```
strings.Split(s, sep string) []string
strings.SplitN(s, sep string, n int) []string
strings.Fields(s string) []string
```

Splitting. `Fields` splits on any whitespace and discards empty tokens — it is the correct function for tokenising space-separated input. `SplitN` limits the number of substrings returned.

```
strings.Join(elems []string, sep string) string
```

Joins a slice with a separator. More efficient than a loop with concatenation.

```
strings.TrimSpace(s string) string
strings.Trim(s, cutset string) string
strings.TrimPrefix(s, prefix string) string
strings.TrimSuffix(s, suffix string) string
strings.TrimFunc(s string, f func(rune) bool) string
```

Trimming. `Trim` removes any characters in `cutset` from both ends; `TrimFunc` removes characters matching a predicate.

```
strings.ReplaceAll(s, old, new string) string
strings.NewReplacer(oldnew ...string) *Replacer
```

Replacement. `NewReplacer` performs multiple replacements in one pass — more efficient than chaining `ReplaceAll` calls.

```
strings.ToLower(s string) string
strings.ToUpper(s string) string
strings.EqualFold(s, t string) bool
```

Case conversion and case-insensitive comparison. `EqualFold` is the correct way to compare strings case-insensitively — it handles Unicode folding correctly, unlike `strings.ToLower(a) == strings.ToLower(b)`.

```
(*Builder).WriteString(s string) (int, error)
(*Builder).String() string
```

Accumulate and retrieve. `WriteString` never returns an error — the signature satisfies `io.StringWriter`.

Example

```
// Parse a simple key=value config format
func parseConfig(input string) map[string]string {
    cfg := make(map[string]string)
    for _, line := range strings.Split(input, "\n") {
        line = strings.TrimSpace(line)
        if line == "" || strings.HasPrefix(line, "#") {
            continue
        }
        key, value, found := strings.Cut(line, "=")
        if !found {
            continue
        }
        cfg[strings.TrimSpace(key)] = strings.TrimSpace(value)
    }
    return cfg
}

// Build a query string efficiently
func buildQuery(terms []string) string {
    var b strings.Builder
    for i, term := range terms {
        if i > 0 {
            b.WriteString(" AND ")
        }
        b.WriteString(strings.ToUpper(term))
    }
    return b.String()
}
```

Gotcha

`strings.Split(s, sep)` returns a slice of length 1 containing the original string when `sep` is not found — not an empty slice and not an error. Code that assumes the result has at least two elements, or that uses index 1 directly, will panic on input that contains no separator. Always check `len(parts) > 1` or use `strings.Cut`, which returns a boolean indicating whether the separator was present. `strings.Cut` was added in Go 1.18 and is the correct function for splitting on the first occurrence of a separator.

See also

`bytes` · `strconv` · `unicode`

sync

Concurrency

The `sync` package provides synchronisation primitives for coordinating goroutines that share state: mutexes, wait groups, once-initialisation, and concurrent maps. These primitives operate on shared memory — they are the correct tool when goroutines need to access a common data structure safely. For communication between goroutines, channels are idiomatic; for protecting shared state, `sync` is the right package. The two are complementary, not alternatives.

Types

Type	Purpose
Mutex	Mutual exclusion lock; only one goroutine holds it at a time
RWMutex	Reader-writer lock; multiple concurrent readers or one writer
WaitGroup	Waits for a collection of goroutines to finish
Once	Ensures a function is called exactly once, even under concurrent access
Map	Concurrent map safe for simultaneous reads and writes without external locking
Pool	A pool of reusable objects; reduces allocation pressure
Cond	Condition variable for goroutines waiting on a shared condition

Functions and methods

```
(*Mutex).Lock()
(*Mutex).Unlock()
```

Acquire and release a mutex. Always unlock with `defer mu.Unlock()` immediately after `Lock()` to ensure release on all return paths including panics.

```
(*RWMutex).RLock()
(*RWMutex).RUnlock()
(*RWMutex).Lock()
(*RWMutex).Unlock()
```

Reader-writer locking. Use `RLock`/`RUnlock` for read-only access; `Lock`/`Unlock` for writes. Multiple goroutines can hold `RLock` simultaneously; `Lock` waits for all readers to finish.

```
(*WaitGroup).Add(delta int)
(*WaitGroup).Done()
(*WaitGroup).Wait()
```

Coordinate goroutine completion. Call `Add` before starting each goroutine; `Done` (typically deferred) when it finishes; `Wait` to block until all are done.

```
(*Once).Do(f func())
```

Calls f exactly once, regardless of how many goroutines call `Do` concurrently. Subsequent calls block until the first completes, then return without calling f again. Use for lazy initialisation of shared resources.

```
(*Pool).Get() any
(*Pool).Put(x any)
```

Borrow and return objects. The pool may discard objects between GC cycles — do not use for objects that must persist. Best for short-lived allocations like `bytes.Buffer` instances in request handlers.

Example

```
// Thread-safe cache with read-biased locking
type Cache struct {
    mu    sync.RWMutex
    items map[string]Item
}

func (c *Cache) Get(key string) (Item, bool) {
    c.mu.RLock()
    defer c.mu.RUnlock()
    item, ok := c.items[key]
    return item, ok
}

func (c *Cache) Set(key string, item Item) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items[key] = item
}

// Fan out work across goroutines and collect results
func processAll(items []string) []Result {
    results := make([]Result, len(items))
    var wg sync.WaitGroup
    for i, item := range items {
        wg.Add(1)
        go func(idx int, val string) {
            defer wg.Done()
            results[idx] = process(val)
        }(i, item)
    }
    wg.Wait()
    return results
}
```

Gotcha

A `sync.Mutex` must not be copied after first use. Copying a mutex copies its internal state — if it was locked at the time of copy, the copy is permanently locked. This applies to any struct containing a mutex: pass such structs by pointer, not by value. The `go vet` tool detects this with

the `copylocks` check, but only when the copy is syntactically visible — copies through interface assignments or reflection are not caught.

See also

`sync/atomic` · `context` · `runtime`

sync/atomic

Concurrency

The `sync/atomic` package provides lock-free atomic operations on integers, booleans, pointers, and arbitrary values. Atomic operations are implemented using CPU instructions that guarantee indivisibility — a read-modify-write that cannot be interrupted by another goroutine. They are faster than mutexes for the specific case of a single shared value: a counter, a flag, a pointer swap. For anything more complex than a single value — a struct with multiple fields that must be updated together — a mutex is the correct tool because atomics provide no way to compose multiple operations into a single indivisible unit.

Types

Type	Purpose
Int32	Atomic int32 with Load, Store, Add, Swap, CompareAndSwap
Int64	Atomic int64
Uint32	Atomic uint32
Uint64	Atomic uint64
Bool	Atomic bool with Load, Store, Swap, CompareAndSwap
Pointer[T]	Atomic pointer to T, generic since Go 1.19
Value	Atomic store for any value of a consistent type

Functions and methods

The typed atomic types (`Int64`, `Bool`, `Pointer[T]`) are the correct API for new code — they are safer and more ergonomic than the

package-level functions. The package-level functions (`atomic.AddInt64`, `atomic.LoadUint32` etc.) remain for compatibility but are more error-prone.

```
(*Int64).Add(delta int64) int64
```

Atomically adds delta to the value and returns the new value. The standard operation for counters — no mutex required.

```
(*Int64).Load() int64
(*Int64).Store(val int64)
```

Read and write the value atomically. A plain assignment or read on a shared integer without these is a data race even if the operation appears to be one instruction.

```
(*Int64).CompareAndSwap(old, new int64) bool
(*Int64).Swap(new int64) (old int64)
```

CAS and unconditional swap. CAS is the building block of lock-free data structures — it updates the value only if it currently equals `old`, returning whether the swap occurred.

```
(*Value).Store(v any)
(*Value).Load() any
(*Value).CompareAndSwap(old, new any) bool
```

Atomically store and load values of any type. All values stored must have the same concrete type — storing a different type after the first `Store` panics.

Example

```
// Request metrics without mutex overhead
type Metrics struct {
    Requests atomic.Int64
    Errors   atomic.Int64
    InFlight atomic.Int64
}
```

```
func (m *Metrics) RecordRequest(err error) {
    m.Requests.Add(1)
    m.InFlight.Add(1)
    defer m.InFlight.Add(-1)
    if err != nil {
        m.Errors.Add(1)
    }
}

// Atomic config hot-swap: update config without stopping the
server
type Server struct {
    config atomic.Pointer[Config]
}

func (s *Server) UpdateConfig(cfg *Config) {
    s.config.Store(cfg)
}

func (s *Server) handleRequest(r *http.Request) {
    cfg := s.config.Load() // always sees a complete Config,
never a partial write
    _ = cfg
}
```

Gotcha

`sync/atomic.Value` requires that all stored values have the same concrete type. Storing a `*ConfigV1` and then a `*ConfigV2` — even if both implement the same interface — panics at runtime. If the type of the stored value needs to change, use `atomic.Pointer[T]` with an interface type for T, or restructure to store a versioned wrapper struct. The panic message ("sync/atomic: store of inconsistently typed value into Value") is clear but only appears at runtime, not at compile time.

See also

`sync` · `runtime` · `context`

testing

Testing

The `testing` package provides Go's built-in test framework. Tests, benchmarks, and fuzz tests all live in `_test.go` files and are discovered and executed by `go test`. No external test runner, no configuration files, no test framework dependency. The `T` type drives unit tests; `B` drives benchmarks; `F` drives fuzz tests. Subtests via `t.Run` enable table-driven tests and parallel execution within a single test function.

Types

Type	Purpose
T	Unit test state; Fail, Error, Fatal, Log, Run, Parallel, Helper
B	Benchmark state; N is the iteration count set by the framework
F	Fuzz test state; Add seeds the corpus, Fuzz runs the fuzz function
M	Test main; Run executes all tests, allowing setup and teardown
TB	Interface satisfied by both T and B; use in shared test helpers

Functions and methods

```
(*T).Run(name string, f func(t *T)) bool
```

Runs f as a subtest named name. Subtests appear as `TestFoo/name` in output. Return value indicates whether the subtest passed. Subtests are run sequentially by default; call `t.Parallel()` inside f to run them concurrently.

```
(*T).Fatal(args ...any)
(*T).Fatalf(format string, args ...any)
```

Log a message and stop the current test immediately. Use for errors that make further test execution meaningless — a nil pointer that would cause a panic on the next line.

```
(*T).Error(args ...any)
(*T).Errorf(format string, args ...any)
```

Log a failure but continue execution. Use when multiple independent assertions should all be checked even if one fails.

```
(*T).Helper()
```

Marks the calling function as a test helper. When the test fails, the file and line number reported points to the caller of the helper, not the helper itself. Always call `t.Helper()` as the first line of any assertion helper function.

```
(*T).Cleanup(f func())
```

Registers f to be called when the test completes, regardless of outcome. Equivalent to `defer` but correctly scoped to the test rather than the function — useful in helper functions that set up resources.

```
(*B).ResetTimer()
(*B).ReportAllocs()
```

Control benchmark timing. Call `ResetTimer` after expensive setup to exclude it from the measured time. `ReportAllocs` reports allocations per operation — essential for allocation-sensitive code.

Example

```
// Table-driven test with subtests
func TestParsePort(t *testing.T) {
    cases := []struct {
        name    string
```

```
        input   string
        want    int
        wantErr bool
    }{
        {"valid", "8080", 8080, false},
        {"zero", "0", 0, true},
        {"negative", "-1", 0, true},
        {"too large", "99999", 0, true},
        {"non-numeric", "abc", 0, true},
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            got, err := parsePort(tc.input)
            if (err != nil) != tc.wantErr {
                t.Errorf("parsePort(%q) error = %v, wantErr
%v", tc.input, err, tc.wantErr)
            }
            if !tc.wantErr && got != tc.want {
                t.Errorf("parsePort(%q) = %d, want %d",
tc.input, got, tc.want)
            }
        })
    }
}

// Benchmark with allocation reporting
func BenchmarkParsePort(b *testing.B) {
    b.ReportAllocs()
    for range b.N {
        parsePort("8080")
    }
}
```

Gotcha

`t.Fatal` and `t.FailNow` stop the current goroutine by calling `runtime.Goexit`. If called from a goroutine other than the one running the test — inside a spawned goroutine — the test function itself continues running and the failure may not be reported correctly. In goroutines spawned during a test, collect errors through a channel or use `t.Errorf` rather than `t.Fatalf`, and signal completion through a

`sync.WaitGroup`.

See also

`net/http/httptest` · `os` · `fmt`

text/scanner

Text

The `text/scanner` package provides a lexical scanner for Go-like source text. It tokenises input into identifiers, integers, floats, characters, strings, comments, and raw strings — the basic vocabulary of programming language syntax. It is the right tool for implementing simple configuration file parsers, DSL readers, and expression evaluators where the token structure resembles Go or a C-family language. For languages with significantly different syntax, a hand-written tokeniser or a parser generator like `go/scanner` or `flex` is more appropriate.

Types

Type	Purpose
Scanner	Lexical scanner; initialise with Init, then call Scan repeatedly
Position	Source position with filename, offset, line, and column

Functions and methods

```
(*Scanner).Init(src io.Reader) *Scanner
```

Initialises the scanner with a source reader. Configure behaviour before scanning by setting fields on the scanner: `Mode` controls which token types are recognised; `Error` installs an error handler; `Filename` sets the source name for position reporting.

```
(*Scanner).Scan() rune
```

Advances to the next token and returns its type as a rune. Negative values are the predefined token constants: `scanner.Ident`,

`scanner.Int`, `scanner.Float`, `scanner.Char`, `scanner.String`, `scanner.RawString`, `scanner.Comment`. Returns `scanner.EOF` when the input is exhausted. The token text is available in `(*Scanner).TokenText()`.

```
(*Scanner).TokenText() string
```

Returns the text of the most recently scanned token.

```
(*Scanner).Pos() Position
```

Returns the position of the most recently scanned token.

Example

```
// Parse a simple key = value DSL
func parseDSL(input string) (map[string]string, error) {
    var s scanner.Scanner
    s.Init(strings.NewReader(input))
    s.Filename = "config"

    result := make(map[string]string)
    for tok := s.Scan(); tok != scanner.EOF; tok = s.Scan() {
        if tok != scanner.Ident {
            return nil, fmt.Errorf("%s: expected identifier,
got %q", s.Pos(), s.TokenText())
        }
        key := s.TokenText()

        if tok = s.Scan(); tok != '=' {
            return nil, fmt.Errorf("%s: expected '=', got %q",
s.Pos(), s.TokenText())
        }

        if tok = s.Scan(); tok != scanner.String {
            return nil, fmt.Errorf("%s: expected string, got
%q", s.Pos(), s.TokenText())
        }
        value, _ := strconv.Unquote(s.TokenText())
        result[key] = value
    }
```

```
    return result, nil
}
```

Gotcha

`text/scanner` skips whitespace and comments by default, which is convenient but means the scanner position reported by `Pos()` refers to the start of the next token, not the current one. To get the position of the token just scanned, call `(*Scanner).Pos()` before calling `Scan` again — or use the `Position` field directly after each `Scan` call. The asymmetry between `Scan` and `Pos` is the most common source of incorrect position reporting in DSL parsers built on this package.

See also

`bufio` · `strings` · `strconv`

text/tabwriter

Text

The `text/tabwriter` package implements an elastic tabstop writer that aligns columns in tab-separated text. It buffers output and flushes it with columns aligned once it has seen enough of the input to determine the required widths. The Go toolchain uses it internally for `go list`, `go vet`, and similar commands. It is the correct tool for any CLI output that presents tabular data — build tools, status displays, report generators — where manual padding would be fragile and error-prone.

Types

Type	Purpose
Writer	Buffering writer that aligns tab-separated columns before flushing

Functions and methods

```
tabwriter.NewWriter(output io.Writer, minwidth, tabwidth,
padding int,
    padchar byte, flags uint) *Writer
```

Creates a new tabwriter. Key parameters:

- `minwidth` — minimum column width in characters
- `tabwidth` — width of tab characters (only relevant when `TabIndent` flag is set)
- `padding` — extra padding added between columns
- `padchar` — the character used for padding, typically `' '`
- `flags` — `tabwriter.AlignRight` right-aligns numeric columns; `tabwriter.Debug` adds column separators for inspection

```
(*Writer).Flush() error
```

Flushes buffered output to the underlying writer with columns aligned. Must be called after all rows are written — output is not visible until `Flush` is called.

Example

```
// Tabular CLI output with aligned columns
func printServices(w io.Writer, services []ServiceStatus) {
    tw := tabwriter.NewWriter(w, 0, 0, 2, ' ', 0)
    fmt.Fprintln(tw, "NAME\tSTATUS\tUPTIME\tRESTARTS")
    for _, s := range services {
        fmt.Fprintf(tw, "%s\t%s\t%s\t%d\n",
            s.Name,
            s.Status,
            s.Uptime.Round(time.Second),
            s.Restarts,
        )
    }
    tw.Flush()
}
```

Gotcha

`tabwriter` aligns columns by buffering all output until `Flush` is called. If you write rows incrementally and call `Flush` after each row, each row is formatted independently with no alignment between rows — every column is as wide as the content of that row alone. Write all rows first, then flush once. For streaming output where alignment across all rows is not required, `tabwriter` adds unnecessary complexity and buffering — write pre-formatted strings directly instead.

See also

`fmt` · `bufio` · `os`

text/template

Text

The `text/template` package provides general-purpose templating using the same syntax as `html/template` but without HTML-specific escaping. It is the correct choice for generating any text that is not HTML: configuration files, code generation, email bodies, markdown, shell scripts, and structured text output. Because it applies no automatic escaping, it is the developer's responsibility to ensure that dynamic values are safe for the output format. For HTML output, always use `html/template`.

Types

Type	Purpose
Template	A parsed template; safe for concurrent execution after parsing
FuncMap	A map of function names to functions callable within templates

Functions and methods

```
template.New(name string) *Template
(*Template).Parse(text string) (*Template, error)
template.Must(t *Template, err error) *Template
```

Parse a template from a string. `Must` wraps `New(...).Parse(...)` for package-level variables where a parse error is a programming bug. For templates loaded from files or embedded filesystems, use `ParseFiles`, `ParseGlob`, or `ParseFS`.

```
(*Template).Funcs(funcMap FuncMap) *Template
```

Registers custom functions available within template actions. Must be called before parsing — functions referenced in a template must be

registered before the template is parsed, not before it is executed.

```
(*Template).Execute(wr io.Writer, data any) error
(*Template).ExecuteTemplate(wr io.Writer, name string, data
any) error
```

Renders the template to wr with data as the dot value. `ExecuteTemplate` selects a named template from a set parsed together.

```
(*Template).ParseFS(fs fs.FS, patterns ...string) (*Template,
error)
```

Parses templates from an embedded or abstract filesystem. The idiomatic pattern for production — templates embedded in the binary via `embed.FS`.

Example

```
// Generate a Go source file from a template
const structTemplate = `// Code generated by scaffold. DO NOT
EDIT.
package {{.Package}}

// {{.Name}} represents a {{.Description}}.
type {{.Name}} struct {
{{- range .Fields}}
    {{.Name}} {{.Type}} ` + "`" + `json:"{{.JSONName}}"` + "`"
+ `
{{- end}}
}

func New{{.Name}}() *{{.Name}} {
    return &{{.Name}}{}
}
`

var structTmpl =
template.Must(template.New("struct").Parse(structTemplate))

type StructDef struct {
```

```
    Package     string
    Name        string
    Description string
    Fields      []FieldDef
}

type FieldDef struct {
    Name     string
    Type     string
    JSONName string
}

func generateStruct(w io.Writer, def StructDef) error {
    return structTmpl.Execute(w, def)
}
```

Gotcha

`(*Template).Funcs` must be called before parsing, not before execution. A function registered after the template is parsed is not visible to that template — the parser resolves function names at parse time, not at execution time. If you register a function and find the template returning `"function ... not defined"` at execution, the registration call is in the wrong order. Register all functions, then parse.

See also

`html/template` · `embed` · `os`

time

System

The `time` package handles time values, durations, timers, and tickers. Its design is unusual: formatting and parsing use a reference time — `Mon Jan 2 15:04:05 MST 2006` — rather than strftime-style format codes. Each component of the reference time has a unique value (month=1, day=2, hour=15, minute=4, second=5, year=6) that makes the layout self-documenting. The package uses a monotonic clock for elapsed time measurement and a wall clock for absolute time — `time.Now()` carries both, which means duration arithmetic is accurate even across wall clock adjustments.

Types

Type	Purpose
Time	A point in time with nanosecond precision; carries both wall and monotonic readings
Duration	A span of time as an int64 nanosecond count
Location	A time zone; time.UTC and time.Local are the two standard values
Timer	A single-shot timer; fires once after a duration
Ticker	A repeating timer; fires at regular intervals

Functions and methods

```
time.Now() Time
```

Returns the current local time with both wall and monotonic clock readings. Use for all time measurement — do not use

`time.Now().Unix()` differences for elapsed time, use `time.Since`.

```
time.Since(t Time) Duration
time.Until(t Time) Duration
```

Elapsed time since t, and remaining time until t. Both use the monotonic clock when available — prefer these over `time.Now().Sub(t)` for interval measurement.

```
(*Time).Format(layout string) string
time.Parse(layout, value string) (Time, error)
```

Format and parse using the reference time layout. Common layouts are available as constants: `time.RFC3339`, `time.RFC822`, `time.DateTime`, `time.DateOnly`, `time.TimeOnly`.

```
(*Time).In(loc *Location) Time
time.LoadLocation(name string) (*Location, error)
```

Convert a time to a different time zone. Location names follow the IANA database: `"America/New_York"`, `"Europe/London"`, `"Asia/Tokyo"`.

```
time.NewTimer(d Duration) *Timer
(*Timer).Stop() bool
(*Timer).Reset(d Duration) bool
```

Single-shot timer. Always stop a timer that is no longer needed to release its resources. The return value of `Stop` indicates whether the timer fired before being stopped.

```
time.NewTicker(d Duration) *Ticker
(*Ticker).Stop()
```

Repeating timer. Always call `Stop` when done — an unstopped ticker runs until the program exits.

```
time.Sleep(d Duration)
time.After(d Duration) <-chan Time
```

`Sleep` blocks the current goroutine. `After` returns a channel that receives once after d — convenient for select statements, but leaks a timer if the channel is never read before the goroutine exits. Prefer `time.NewTimer` with explicit `Stop` in long-lived code.

Example

```
// Rate limiter using a ticker
func rateLimitedProcessor(ctx context.Context, items <-chan
Item, ratePerSec int) {
    ticker := time.NewTicker(time.Second /
time.Duration(ratePerSec))
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            select {
            case item, ok := <-items:
                if !ok {
                    return
                }
                process(item)
            default:
            }
        }
    }
}

// Parse and format timestamps with explicit timezone
func normaliseTimestamp(raw, layout, zone string) (string,
error) {
    loc, err := time.LoadLocation(zone)
    if err != nil {
        return "", fmt.Errorf("unknown timezone %q: %w", zone,
err)
    }
    t, err := time.Parse(layout, raw)
    if err != nil {
        return "", fmt.Errorf("parsing %q: %w", raw, err)
```

```
    }
    return t.In(loc).Format(time.RFC3339), nil
}
```

Gotcha

`time.After(d)` creates a `time.Timer` internally that is not garbage collected until it fires, even if the surrounding goroutine exits or the channel is never read. In code that creates many short-lived goroutines with `time.After` in a select — a common timeout pattern — this leaks timers for the full duration d before they are collected. Replace `time.After(d)` with `time.NewTimer(d)` and defer `timer.Stop()` to release the timer immediately when it is no longer needed.

See also

`context` · `sync` · `os/signal`

unicode

Text

The `unicode` package provides Unicode character classification and case conversion at the rune level. Its functions test whether a rune belongs to a category — letter, digit, space, punctuation, mark, symbol — and convert between cases. It is the correct package for validating or transforming text character by character when the rules must be Unicode-correct rather than ASCII-only. For byte-level operations on UTF-8 encoded strings, use `unicode/utf8` alongside `strings`.

Types

Type	Purpose
RangeTable	A set of Unicode code points defined by ranges; used by classification functions
SpecialCase	Custom case mapping for languages with non-standard rules (Turkish, Azerbaijani)

Functions and methods

```
unicode.IsLetter(r rune) bool
unicode.IsDigit(r rune) bool
unicode.IsSpace(r rune) bool
unicode.IsPunct(r rune) bool
unicode.IsUpper(r rune) bool
unicode.IsLower(r rune) bool
unicode.IsPrint(r rune) bool
```

Classification functions. `IsSpace` recognises all Unicode whitespace, not just ASCII space and tab. `IsPrint` reports whether a rune is printable — visible and not a control character.

```
unicode.ToUpper(r rune) rune
unicode.ToLower(r rune) rune
unicode.ToTitle(r rune) rune
```

Case conversion at the rune level. For string-level case conversion, use `strings.ToUpper` and `strings.ToLower`, which call these internally.

```
unicode.In(r rune, rangeTab ...*RangeTable) bool
unicode.Is(rangeTab *RangeTable, r rune) bool
```

Test membership in a Unicode category. The package exports range tables for all Unicode categories: `unicode.Latin`, `unicode.Cyrillic`, `unicode.Han`, `unicode.Arabic`, and many others.

Example

```
// Validate that a string contains only printable non-space
characters
func isValidIdentifier(s string) bool {
    if s == "" {
        return false
    }
    for i, r := range s {
        if i == 0 && !unicode.IsLetter(r) && r != '_' {
            return false
        }
        if i > 0 && !unicode.IsLetter(r) &&
!unicode.IsDigit(r) && r != '_' {
            return false
        }
    }
    return true
}

// Count words in text, handling Unicode whitespace correctly
func wordCount(s string) int {
    inWord := false
    count := 0
    for _, r := range s {
        if unicode.IsSpace(r) {
            inWord = false
```

```
        } else if !inWord {
            inWord = true
            count++
        }
    }
    return count
}
```

Gotcha

`unicode.IsDigit` returns true for Unicode digits beyond the ASCII range — Eastern Arabic numerals (.....), Devanagari digits (....), and others. Code that validates a "numeric" string with `IsDigit` and then passes it to `strconv.Atoi` or `strconv.ParseInt` will find that `Atoi` rejects the non-ASCII digits even though `IsDigit` accepted them. For validating ASCII-only numeric strings, test `r >= '0' && r <= '9'` directly rather than `unicode.IsDigit`.

See also

`unicode/utf8` · `strings` · `strconv`

unicode/utf8

Text

The `unicode/utf8` package provides functions for encoding and decoding UTF-8 at the byte level. Go strings are UTF-8 encoded byte sequences, and ranging over a string with `for _, r := range s` decodes runes correctly — but when you need to inspect or manipulate the encoding directly, this package is the tool. It is used when working with byte slices rather than strings, when you need to detect invalid UTF-8, when counting the number of characters rather than bytes, or when implementing a parser or encoder that must work at the rune boundary level.

Types

None. The package exports functions and constants only.

Functions and methods

```
utf8.RuneCountInString(s string) int
utf8.RuneCount(p []byte) int
```

Returns the number of runes in a string or byte slice — the character count, not the byte count. `len(s)` returns byte count; `utf8.RuneCountInString(s)` returns character count.

```
utf8.ValidString(s string) bool
utf8.Valid(p []byte) bool
```

Reports whether s or p consists entirely of valid UTF-8 encoded runes. Use to validate input from external sources before processing.

```
utf8.DecodeRuneInString(s string) (r rune, size int)
utf8.DecodeLastRuneInString(s string) (r rune, size int)
```

Decodes the first or last rune from s, returning the rune and its byte width. Used when iterating manually rather than with range, or when you

need the byte width for indexing.

```
utf8.EncodeRune(p []byte, r rune) int
```

Encodes r into p and returns the number of bytes written. p must be large enough — `utf8.UTFMax` (4) bytes is always sufficient.

```
utf8.RuneLen(r rune) int
```

Returns the number of bytes required to encode r. Returns -1 for invalid runes.

Constants

```
utf8.UTFMax = 4          // maximum bytes per encoded rune
utf8.RuneError = '\uFFFD' // the replacement character for
invalid sequences
```

Example

```
// Truncate a string to a maximum number of runes, not bytes
func truncate(s string, maxRunes int) string {
    count := 0
    for i, r := range s {
        if count == maxRunes {
            return s[:i]
        }
        _ = r
        count++
    }
    return s
}

// Validate and sanitise UTF-8 input, replacing invalid
sequences
func sanitiseUTF8(s string) string {
    if utf8.ValidString(s) {
        return s
    }
    var b strings.Builder
    for i := 0; i < len(s); {
```

```
        r, size := utf8.DecodeRuneInString(s[i:])
        if r == utf8.RuneError && size == 1 {
            b.WriteRune(utf8.RuneError) // replacement
character
        } else {
            b.WriteRune(r)
        }
        i += size
    }
    return b.String()
}
```

Gotcha

`len(s)` returns the number of bytes in a string, not the number of characters. For ASCII strings these are equal, which masks the bug. For strings containing multibyte characters — any non-English text, emoji, or symbols — `len(s)` and `utf8.RuneCountInString(s)` diverge. Code that uses `len(s)` to validate a maximum character count, to index into a string by character position, or to compare string lengths will silently produce wrong results for non-ASCII input. Always use `utf8.RuneCountInString` for character counts.

See also

`unicode` · `strings` · `bytes`

unicode/utf16

Text

The `unicode/utf16` package encodes and decodes UTF-16, the encoding used natively by Windows APIs, Java, JavaScript strings, and some file formats including NTFS filenames and Microsoft Office documents. Go strings are UTF-8 internally, so UTF-16 is only encountered at system boundaries. Direct use of this package is uncommon in most Go programs — it appears when calling Windows system APIs via `golang.org/x/sys/windows`, reading Windows registry values, processing Office Open XML documents, or handling Java serialisation formats.

Types

None. The package exports functions only.

Functions and methods

```
utf16.Encode(s []rune) []uint16
```

Encodes a slice of runes as UTF-16 code units. Runes outside the Basic Multilingual Plane are encoded as surrogate pairs.

```
utf16.Decode(s []uint16) []rune
```

Decodes UTF-16 code units to runes. Surrogate pairs are combined into the corresponding rune.

```
utf16.EncodeRune(r rune) (r1, r2 rune)
utf16.DecodeRune(r1, r2 rune) rune
```

Encode and decode a single rune as a surrogate pair. `IsSurrogate(r rune) bool` tests whether a rune is in the surrogate range.

```
utf16.AppendRune(a []uint16, r rune) []uint16
```

Appends the UTF-16 encoding of r to a, added in Go 1.20. The incremental version of `Encode` for building UTF-16 strings one rune at a time.

Example

```
// Convert a Go string to a null-terminated UTF-16 slice for a
Windows API call
func toUTF16Ptr(s string) *uint16 {
    encoded := utf16.Encode([]rune(s + "\x00"))
    return &encoded[0]
}

// Read a UTF-16LE encoded file (common in Windows
environments)
func readUTF16File(path string) (string, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return "", err
    }
    // Strip BOM if present
    if len(data) >= 2 && data[0] == 0xFF && data[1] == 0xFE {
        data = data[2:]
    }
    // Convert []byte to []uint16
    u16 := make([]uint16, len(data)/2)
    for i := range u16 {
        u16[i] = uint16(data[2*i]) | uint16(data[2*i+1])<<8
    }
    return string(utf16.Decode(u16)), nil
}
```

Gotcha

UTF-16 comes in two byte orders — little-endian (UTF-16LE) and big-endian (UTF-16BE) — distinguished by a Byte Order Mark (BOM: `0xFF 0xFE` for LE, `0xFE 0xFF` for BE) at the start of the data. `utf16.Decode` operates on `[]uint16` values and has no concept of byte order — the caller is responsible for converting the raw bytes to

`uint16` in the correct order. Reading a UTF-16BE file as if it were UTF-16LE swaps every pair of bytes, producing garbage with no error. Always check for and handle the BOM before decoding.

See also

`unicode/utf8` · `unicode` · `encoding/binary`