# Go 1.18 Release Notes

Table of Contents



## Introduction to Go 1.18

The latest Go release, version 1.18, is a significant release, including changes to the language, implementation of the toolchain, runtime, and libraries. Go 1.18 arrives seven months after Go 1.17. As always, the release maintains the Go 1 promise of compatibility. We expect almost all Go programs to continue to compile and run as before.

## Changes to the language

### Generics

Go 1.18 includes an implementation of generic features as described by the Type Parameters Proposal. This includes major - but fully backward-compatible - changes to the language.

These new language changes required a large amount of new code that has not had significant testing in production settings. That will only happen as more people write and use generic code. We believe that this feature is well implemented and high quality. However, unlike most aspects of Go, we can't back up that belief with real world experience. Therefore, while we encourage the use of generics where it makes sense, please use appropriate caution when deploying generic code in production.

While we believe that the new language features are well designed and clearly specified, it is possible that we have made mistakes. We want to stress that the Go 1 compatibility guarantee says "If it becomes necessary to address an inconsistency or incompleteness in the specification, resolving the issue could affect the meaning or legality of existing programs. We reserve the right to address such issues, including updating the implementations." It also says

“If a compiler or library has a bug that violates the specification, a program that depends on the buggy behavior may break if the bug is fixed. We reserve the right to fix such bugs.” In other words, it is possible that there will be code using generics that will work with the 1.18 release but break in later releases. We do not plan or expect to make any such change. However, breaking 1.18 programs in future releases may become necessary for reasons that we cannot today foresee. We will minimize any such breakage as much as possible, but we can’t guarantee that the breakage will be zero.

The following is a list of the most visible changes. For a more comprehensive overview, see the proposal. For details see the language spec.

- The syntax for function and type declarations now accepts type parameters.
- Parameterized functions and types can be instantiated by following them with a list of type arguments in square brackets.
- The new token ~ has been added to the set of operators and punctuation.
- The syntax for Interface types now permits the embedding of arbitrary types (not just type names of interfaces) as well as union and ~T type elements. Such interfaces may only be used as type constraints. An interface now defines a set of types as well as a set of methods.
- The new predeclared identifier `any` is an alias for the empty interface. It may be used instead of `interface{}`.
- The new predeclared identifier `comparable` is an interface that denotes the set of all types which can be compared using == or `!=`. It may only be used as (or embedded in) a type constraint.

There are three experimental packages using generics that may be useful. These packages are in x/exp repository; their API is not covered by the Go 1 guarantee and may change as we gain more experience with generics.

### `golang.org/x/exp/constraints`

Constraints that are useful for generic code, such as `constraints.Ordered`.

### `golang.org/x/exp/slices`

A collection of generic functions that operate on slices of any element type.

### `golang.org/x/exp/maps`

A collection of generic functions that operate on maps of any key or element type.

The current generics implementation has the following known limitations:

- The Go compiler cannot handle type declarations inside generic functions or methods. We hope to provide support for this feature in a future release.
- The Go compiler does not accept arguments of type parameter type with the predeclared functions `real`, `imag`, and `complex`. We hope to remove this restriction in a future release.
- The Go compiler only supports calling a method `m` on a value `x` of type parameter type `P` if `m` is explicitly declared by `P`'s constraint interface. Similarly, method values `x.m` and method expressions `P.m` also are only supported if `m` is explicitly declared by `P`, even though `m` might be in the method set of `P` by virtue of the fact that all types in `P` implement `m`. We hope to remove this restriction in a future release.
- The Go compiler does not support accessing a struct field `x.f` where `x` is of type parameter type even if all types in the type parameter's type set have a field `f`. We may remove this restriction in a future release.
- Embedding a type parameter, or a pointer to a type parameter, as an unnamed field in a struct type is not permitted. Similarly, embedding a type parameter in an interface type is not permitted. Whether these will ever be permitted is unclear at present.
- A union element with more than one term may not contain an interface type with a non-empty method set. Whether this will ever be permitted is unclear at present.

Generics also represent a large change for the Go ecosystem. While we have updated several core tools with generics support, there is much more to do. It will take time for remaining tools, documentation, and libraries to catch up with these language changes.

### Bug fixes

The Go 1.18 compiler now correctly reports `declared but not used` errors for variables that are set inside a function literal but are never used. Before Go 1.18, the compiler did not report an error in such cases. This fixes long-outstanding compiler issue #8560. As a result of this change, (possibly incorrect) programs may not compile anymore. The necessary fix is straightforward: fix the program if it was in fact incorrect, or use the offending variable, for instance by assigning it to the blank identifier `_`. Since `go vet` always pointed out this error, the number of affected programs is likely very small.

The Go 1.18 compiler now reports an overflow when passing a rune constant expression such as `'1' << 32` as an argument to the predeclared functions `print` and `println`, consistent with the behavior of user-defined functions. Before Go 1.18, the compiler did not report an error in such cases but silently accepted such constant arguments if they fit into an `int64`. As a result of this change, (possibly incorrect) programs may not compile anymore. The necessary fix is straightforward: fix the program if it was in fact incorrect, or explicitly convert the offending argument to the correct type. Since `go vet` always pointed out this error, the number of affected programs is likely very small.

# Ports

## AMD64

Go 1.18 introduces the new `GOAMD64` environment variable, which selects at compile time a minimum target version of the AMD64 architecture. Allowed values are `v1`, `v2`, `v3`, or `v4`. Each higher level requires, and takes advantage of, additional processor features. A detailed description can be found here.

The `GOAMD64` environment variable defaults to `v1`.

## RISC-V

The 64-bit RISC-V architecture on Linux (the `linux/riscv64` port) now supports the `c-archive` and `c-shared` build modes.

## Linux

Go 1.18 requires Linux kernel version 2.6.32 or later.

## Windows

The `windows/arm` and `windows/arm64` ports now support non-cooperative preemption, bringing that capability to all four Windows ports, which should hopefully address subtle bugs encountered when calling into Win32 functions that block for extended periods of time.

## iOS

On iOS (the `ios/arm64` port) and iOS simulator running on AMD64-based macOS (the `ios/amd64` port), Go 1.18 now requires iOS 12 or later; support for previous versions has been discontinued.

## FreeBSD

Go 1.18 is the last release that is supported on FreeBSD 11.x, which has already reached end-of-life. Go 1.19 will require FreeBSD 12.2+ or FreeBSD 13.0+. FreeBSD 13.0+ will require a kernel with the COMPAT_FREEBSD12 option set (this is the default).

# Tools

## Fuzzing

Go 1.18 includes an implementation of fuzzing as described by the fuzzing proposal.

See the fuzzing landing page to get started.

Please be aware that fuzzing can consume a lot of memory and may impact your machine's performance while it runs. Also be aware that the fuzzing engine writes values that expand test coverage to a fuzz cache directory within `$GOCACHE/fuzz` while it runs. There is currently no limit to the number of files or total bytes that may be written to the fuzz cache, so it may occupy a large amount of storage (possibly several GBs).

## Go command

### `go get`

`go get` no longer builds or installs packages in module-aware mode. `go get` is now dedicated to adjusting dependencies in `go.mod`. Effectively, the `-d` flag is always enabled. To install the latest version of an executable outside the context of the current module, use `go install example.com/cmd@latest`. Any version query may be used instead of `latest`. This form of `go install` was added in Go 1.16, so projects supporting older versions may need to provide install instructions for both `go install` and `go get`. `go get` now reports an error when used outside a module, since there is no `go.mod` file to update. In GOPATH mode (with `GO111MODULE=off`), `go get` still builds and installs packages, as before.

### Automatic `go.mod` and `go.sum` updates

The `go mod graph`, `go mod vendor`, `go mod verify`, and `go mod why` subcommands no longer automatically update the `go.mod` and `go.sum` files. (Those files can be updated explicitly using `go get`, `go mod tidy`, or `go mod download`.)

### `go version`

The `go` command now embeds version control information in binaries. It includes the currently checked-out revision, commit time, and a flag indicating whether edited or untracked files are present. Version control information is embedded if the `go` command is invoked in a directory within a Git, Mercurial, Fossil, or Bazaar repository, and the `main` package and its containing main module are in the same repository. This information may be omitted using the flag `-buildvcs=false`.

Additionally, the `go` command embeds information about the build, including build and tool tags (set with `-tags`), compiler, assembler, and linker flags (like `-gcflags`), whether cgo was enabled, and if it was, the values of the cgo environment variables (like `CGO_CFLAGS`). Both VCS and build information may be read together with module information using `go version -m file` or `runtime/debug.ReadBuildInfo` (for the currently running binary) or the new `debug/buildinfo` package.

The underlying data format of the embedded build information can change with new go releases, so an older version of `go` may not handle the build information produced with a newer version of `go`. To read the version information from a binary built with `go` 1.18, use the `go version` command and the `debug/buildinfo` package from `go` 1.18+.

#### `go mod download`

If the main module's `go.mod` file specifies `go 1.17` or higher, `go mod download` without arguments now downloads source code for only the modules explicitly required in the main module's `go.mod` file. (In a `go 1.17` or higher module, that set already includes all dependencies needed to build the packages and tests in the main module.) To also download source code for transitive dependencies, use `go mod download all`.

#### `go mod vendor`

The `go mod vendor` subcommand now supports a `-o` flag to set the output directory. (Other `go` commands still read from the `vendor` directory at the module root when loading packages with `-mod=vendor`, so the main use for this flag is for third-party tools that need to collect package source code.)

#### `go mod tidy`

The `go mod tidy` command now retains additional checksums in the `go.sum` file for modules whose source code is needed to verify that each imported package is provided by only one module in the build list. Because this condition is rare and failure to apply it results in a build error, this change is *not* conditioned on the `go` version in the main module's `go.mod` file.

#### `go work`

The `go` command now supports a "Workspace" mode. If a `go.work` file is found in the working directory or a parent directory, or one is specified using the `GOWORK` environment variable, it will put the `go` command into workspace mode. In workspace mode, the `go.work` file will be used to determine the set of main modules used as the roots for module resolution, instead of using the normally-found `go.mod` file to specify the single main module. For more information see the `go work` documentation.

#### `go build -asan`

The `go build` command and related commands now support an `-asan` flag that enables interoperation with C (or C++) code compiled with the address sanitizer (C compiler option `-fsanitize=address`).

#### `go test`

The `go` command now supports additional command line options for the new fuzzing support described above:

- `go test` supports `-fuzz`, `-fuzztime`, and `-fuzzminimizetime` options. For documentation on these see `go help testflag`.
- `go clean` supports a `-fuzzcache` option. For documentation see `go help clean`.

### `//go:build` lines

Go 1.17 introduced `//go:build` lines as a more readable way to write build constraints, instead of `// +build` lines. As of Go 1.17, `gofmt` adds `//go:build` lines to match existing `+build` lines and keeps them in sync, while `go vet` diagnoses when they are out of sync.

Since the release of Go 1.18 marks the end of support for Go 1.16, all supported versions of Go now understand `//go:build` lines. In Go 1.18, `go fix` now removes the now-obsolete `// +build` lines in modules declaring `go 1.18` or later in their `go.mod` files.

For more information, see go.dev/design/draft-gobuild.

## Gofmt

`gofmt` now reads and formats input files concurrently, with a memory limit proportional to `GOMAXPROCS`. On a machine with multiple CPUs, `gofmt` should now be significantly faster.

## Vet

### Updates for Generics

The `vet` tool is updated to support generic code. In most cases, it reports an error in generic code whenever it would report an error in the equivalent non-generic code after substituting for type parameters with a type from their type set. For example, `vet` reports a format error in

```
func Print[T ~int|~string](t T) {
    fmt.Printf("%d", t)
}
```

because it would report a format error in the non-generic equivalent of `Print[string]`:

```
func PrintString(x string) {
    fmt.Printf("%d", x)
}
```

### Precision improvements for existing checkers

The `cmd/vet` checkers `copylock`, `printf`, `sortslice`, `testinggoroutine`, and `tests` have all had moderate precision improvements to handle additional code patterns. This may lead to newly reported errors in existing packages. For example, the `printf` checker now tracks formatting strings created by concatenating string constants. So `vet` will report an error in:

```
// fmt.Printf formatting directive %d is being passed to Println.
fmt.Println("%d"+` ≡ x (mod 2)`+"\n", x%2)
```

## Runtime

The garbage collector now includes non-heap sources of garbage collector work (e.g., stack scanning) when determining how frequently to run. As a result, garbage collector overhead is more predictable when these sources are significant. For most applications these changes will be negligible; however, some Go applications may now use less memory and spend more time on garbage collection, or vice versa, than before. The intended workaround is to tweak `GOGC` where necessary.

The runtime now returns memory to the operating system more efficiently and has been tuned to work more aggressively as a result.

Go 1.17 generally improved the formatting of arguments in stack traces, but could print inaccurate values for arguments passed in registers. This is improved in Go 1.18 by printing a question mark (`?`) after each value that may be inaccurate.

The built-in function `append` now uses a slightly different formula when deciding how much to grow a slice when it must allocate a new underlying array. The new formula is less prone to sudden transitions in allocation behavior.

## Compiler

Go 1.17 implemented a new way of passing function arguments and results using registers instead of the stack on 64-bit x86 architecture on selected operating systems. Go 1.18 expands the supported platforms to include 64-bit ARM (`GOARCH=arm64`), big- and little-endian 64-bit PowerPC (`GOARCH=ppc64`, `ppc64le`), as well as 64-bit x86 architecture (`GOARCH=amd64`) on all operating systems. On 64-bit ARM and 64-bit PowerPC systems, benchmarking shows typical performance improvements of 10% or more.

As mentioned in the Go 1.17 release notes, this change does not affect the functionality of any safe Go code and is designed to have no impact on most assembly code. See the Go 1.17 release notes for more details.

The compiler now can inline functions that contain range loops or labeled for loops.

The new `–asan` compiler option supports the new `go` command `–asan` option.

Because the compiler's type checker was replaced in its entirety to support generics, some error messages now may use different wording than before. In some cases, pre-Go 1.18 error messages provided more detail or were phrased in a more helpful way. We intend to address these cases in Go 1.19.

Because of changes in the compiler related to supporting generics, the Go 1.18 compile speed can be roughly 15% slower than the Go 1.17 compile speed. The execution time of the compiled code is not affected. We intend to improve the speed of the compiler in future releases.

## Linker

The linker emits far fewer relocations. As a result, most codebases will link faster, require less memory to link, and generate smaller binaries. Tools that process Go binaries should use Go 1.18's `debug/gosym` package to transparently handle both old and new binaries.

The new `–asan` linker option supports the new `go` command `–asan` option.

## Bootstrap

When building a Go release from source and `GOROOT_BOOTSTRAP` is not set, previous versions of Go looked for a Go 1.4 or later bootstrap toolchain in the directory `$HOME/go1.4` (`%HOMEDRIVE%%HOMEPATH%\go1.4` on Windows). Go now looks first for `$HOME/go1.17` or `$HOME/sdk/go1.17` before falling back to `$HOME/go1.4`. We intend for Go 1.19 to require Go 1.17 or later for bootstrap, and this change should make the transition smoother. For more details, see go.dev/issue/44505.

## Standard library

### New `debug/buildinfo` package

The new `debug/buildinfo` package provides access to module versions, version control information, and build flags embedded in executable files built by the `go` command. The same information is also available via `runtime/debug.ReadBuildInfo` for the currently running binary and via `go version –m` on the command line.

### New `net/netip` package

The new `net/netip` package defines a new IP address type, `Addr`. Compared to the existing `net.IP` type, the `netip.Addr` type takes less memory, is immutable, and is comparable so it supports == and can be used as a map key.

In addition to `Addr`, the package defines `AddrPort`, representing an IP and port, and `Prefix`, representing a network CIDR prefix.

The package also defines several functions to create and examine these new types: `AddrFrom4`, `AddrFrom16`, `AddrFromSlice`, `AddrPortFrom`, `IPv4Unspecified`, `IPv6LinkLocalAllNodes`, `IPv6Unspecified`, `MustParseAddr`, `MustParseAddrPort`, `MustParsePrefix`, `ParseAddr`, `ParseAddrPort`, `ParsePrefix`, `PrefixFrom`.

The `net` package includes new methods that parallel existing methods, but return `netip.AddrPort` instead of the heavier-weight `net.IP` or `*net.UDPAddr` types: `Resolver.LookupNetIP`, `UDPConn.ReadFromUDPAddrPort`,

`UDPConn.ReadMsgUDPAddrPort`, `UDPConn.WriteToUDPAddrPort`, `UDPConn.WriteMsgUDPAddrPort`. The new `UDPConn` methods support allocation-free I/O.

The `net` package also now includes functions and methods to convert between the existing `TCPAddr`/`UDPAddr` types and `netip.AddrPort`: `TCPAddrFromAddrPort`, `UDPAddrFromAddrPort`, `TCPAddr.AddrPort`, `UDPAddr.AddrPort`.

### TLS 1.0 and 1.1 disabled by default client-side

If `Config.MinVersion` is not set, it now defaults to TLS 1.2 for client connections. Any safely up-to-date server is expected to support TLS 1.2, and browsers have required it since 2020. TLS 1.0 and 1.1 are still supported by setting `Config.MinVersion` to `VersionTLS10`. The server-side default is unchanged at TLS 1.0.

The default can be temporarily reverted to TLS 1.0 by setting the `GODEBUG=tls10default=1` environment variable. This option will be removed in Go 1.19.

### Rejecting SHA-1 certificates

`crypto/x509` will now reject certificates signed with the SHA-1 hash function. This doesn't apply to self-signed root certificates. Practical attacks against SHA-1 have been demonstrated since 2017 and publicly trusted Certificate Authorities have not issued SHA-1 certificates since 2015.

This can be temporarily reverted by setting the `GODEBUG=x509sha1=1` environment variable. This option will be removed in a future release.

### Minor changes to the library

As always, there are various minor changes and updates to the library, made with the Go 1 promise of compatibility in mind.

#### bufio

The new `Writer.AvailableBuffer` method returns an empty buffer with a possibly non-empty capacity for use with append-like APIs. After appending, the buffer can be provided to a succeeding `Write` call and possibly avoid any copying.

The `Reader.Reset` and `Writer.Reset` methods now use the default buffer size when called on objects with a `nil` buffer.

#### bytes

The new `Cut` function slices a `[]byte` around a separator. It can replace and simplify many common uses of `Index`, `IndexByte`, `IndexRune`, and `SplitN`.

`Trim`, `TrimLeft`, and `TrimRight` are now allocation free and, especially for small ASCII cutsets, up to 10 times faster.

The `Title` function is now deprecated. It doesn't handle Unicode punctuation and language-specific capitalization rules, and is superseded by the golang.org/x/text/cases package.

### crypto/elliptic

The `P224`, `P384`, and `P521` curve implementations are now all backed by code generated by the addchain and fiat-crypto projects, the latter of which is based on a formally-verified model of the arithmetic operations. They now use safer complete formulas and internal APIs. P-224 and P-384 are now approximately four times faster. All specific curve implementations are now constant-time.

Operating on invalid curve points (those for which the `IsOnCurve` method returns false, and which are never returned by `Unmarshal` or a `Curve` method operating on a valid point) has always been undefined behavior, can lead to key recovery attacks, and is now unsupported by the new backend. If an invalid point is supplied to a `P224`, `P384`, or `P521` method, that method will now return a random point. The behavior might change to an explicit panic in a future release.

### crypto/tls

The new `Conn.NetConn` method allows access to the underlying `net.Conn`.

### crypto/x509

`Certificate.Verify` now uses platform APIs to verify certificate validity on macOS and iOS when it is called with a nil `VerifyOpts.Roots` or when using the root pool returned from `SystemCertPool`.

`SystemCertPool` is now available on Windows.

On Windows, macOS, and iOS, when a `CertPool` returned by `SystemCertPool` has additional certificates added to it, `Certificate.Verify` will do two verifications: one using the platform verifier APIs and the system roots, and one using the Go verifier and the additional roots. Chains returned by the platform verifier APIs will be prioritized.

`CertPool.Subjects` is deprecated. On Windows, macOS, and iOS the `CertPool` returned by `SystemCertPool` will return a pool which does not include system roots in the slice returned by `Subjects`, as a static list can't appropriately represent the platform policies and might not be available at all from the platform APIs.

Support for signing certificates using signature algorithms that depend on the MD5 hash (`MD5WithRSA`) may be removed in Go 1.19.

### debug/dwarf

The `StructField` and `BasicType` structs both now have a `DataBitOffset` field, which holds the value of the `DW_AT_data_bit_offset` attribute if present.

### debug/elf

The `R_PPC64_RELATIVE` constant has been added.

### debug/plan9obj

The File.Symbols method now returns the new exported error value ErrNoSymbols if the file has no symbol section.

### embed

A `go:embed` directive may now start with `all:` to include files whose names start with dot or underscore.

### go/ast

Per the proposal Additions to go/ast and go/token to support parameterized functions and types the following additions are made to the `go/ast` package:

- the `FuncType` and `TypeSpec` nodes have a new field `TypeParams` to hold type parameters, if any.
- The new expression node `IndexListExpr` represents index expressions with multiple indices, used for function and type instantiations with more than one explicit type argument.

### go/constant

The new `Kind.String` method returns a human-readable name for the receiver kind.

### go/token

The new constant `TILDE` represents the ~ token per the proposal Additions to go/ast and go/token to support parameterized functions and types .

### go/types

The new `Config.GoVersion` field sets the accepted Go language version.

Per the proposal Additions to go/types to support type parameters the following additions are made to the `go/types` package:

- The new type `TypeParam`, factory function `NewTypeParam`, and associated methods are added to represent a type parameter.

- The new type `TypeParamList` holds a list of type parameters.
- The new type `TypeList` holds a list of types.
- The new factory function `NewSignatureType` allocates a `Signature` with (receiver or function) type parameters. To access those type parameters, the `Signature` type has two new methods `Signature.RecvTypeParams` and `Signature.TypeParams`.
- `Named` types have four new methods: `Named.Origin` to get the original parameterized types of instantiated types, `Named.TypeArgs` and `Named.TypeParams` to get the type arguments or type parameters of an instantiated or parameterized type, and `Named.SetTypeParams` to set the type parameters (for instance, when importing a named type where allocation of the named type and setting of type parameters cannot be done simultaneously due to possible cycles).
- The `Interface` type has four new methods: `Interface.IsComparable` and `Interface.IsMethodSet` to query properties of the type set defined by the interface, and `Interface.MarkImplicit` and `Interface.IsImplicit` to set and test whether the interface is an implicit interface around a type constraint literal.
- The new types `Union` and `Term`, factory functions `NewUnion` and `NewTerm`, and associated methods are added to represent type sets in interfaces.
- The new function `Instantiate` instantiates a parameterized type.
- The new `Info.Instances` map records function and type instantiations through the new `Instance` type.
- The new type `ArgumentError` and associated methods are added to represent an error related to a type argument.
- The new type `Context` and factory function `NewContext` are added to facilitate sharing of identical type instances across type-checked packages, via the new `Config.Context` field.

The predicates `AssignableTo`, `ConvertibleTo`, `Implements`, `Identical`, `IdenticalIgnoreTags`, and `AssertableTo` now also work with arguments that are or contain generalized interfaces, i.e. interfaces that may only be used as type constraints in Go code. Note that the behavior of `AssignableTo`, `ConvertibleTo`, `Implements`, and `AssertableTo` is undefined with arguments that are uninstantiated generic types, and `AssertableTo` is undefined if the first argument is a generalized interface.

### html/template

Within a `range` pipeline the new `{{break}}` command will end the loop early and the new `{{continue}}` command will immediately start the next loop iteration.

The `and` function no longer always evaluates all arguments; it stops evaluating arguments after the first argument that evaluates to false. Similarly, the `or` function now stops evaluating

arguments after the first argument that evaluates to true. This makes a difference if any of the arguments is a function call.

### image/draw

The `Draw` and `DrawMask` fallback implementations (used when the arguments are not the most common image types) are now faster when those arguments implement the optional `draw.RGBA64Image` and `image.RGBA64Image` interfaces that were added in Go 1.17.

### net

`net.Error.Temporary` has been deprecated.

### net/http

On WebAssembly targets, the `Dial`, `DialContext`, `DialTLS` and `DialTLSContext` method fields in `Transport` will now be correctly used, if specified, for making HTTP requests.

The new `Cookie.Valid` method reports whether the cookie is valid.

The new `MaxBytesHandler` function creates a `Handler` that wraps its `ResponseWriter` and `Request.Body` with a `MaxBytesReader`.

When looking up a domain name containing non-ASCII characters, the Unicode-to-ASCII conversion is now done in accordance with Nontransitional Processing as defined in the Unicode IDNA Compatibility Processing standard (UTS #46). The interpretation of four distinct runes are changed: ß, ς, zero-width joiner U+200D, and zero-width non-joiner U+200C. Nontransitional Processing is consistent with most applications and web browsers.

### os/user

`User.GroupIds` now uses a Go native implementation when cgo is not available.

### reflect

The new `Value.SetIterKey` and `Value.SetIterValue` methods set a Value using a map iterator as the source. They are equivalent to `Value.Set(iter.Key())` and `Value.Set(iter.Value())`, but do fewer allocations.

The new `Value.UnsafePointer` method returns the Value's value as an `unsafe.Pointer`. This allows callers to migrate from `Value.UnsafeAddr` and `Value.Pointer` to eliminate the need to perform uintptr to unsafe.Pointer conversions at the callsite (as unsafe.Pointer rules require).

The new `MapIter.Reset` method changes its receiver to iterate over a different map. The use of `MapIter.Reset` allows allocation-free iteration over many maps.

A number of methods ( `Value.CanInt`, `Value.CanUint`, `Value.CanFloat`, `Value.CanComplex` ) have been added to `Value` to test if a conversion is safe.

`Value.FieldByIndexErr` has been added to avoid the panic that occurs in `Value.FieldByIndex` when stepping through a nil pointer to an embedded struct.

`reflect.Ptr` and `reflect.PtrTo` have been renamed to `reflect.Pointer` and `reflect.PointerTo`, respectively, for consistency with the rest of the reflect package. The old names will continue to work, but will be deprecated in a future Go release.

### regexp

`regexp` now treats each invalid byte of a UTF-8 string as `U+FFFD`.

### runtime/debug

The `BuildInfo` struct has two new fields, containing additional information about how the binary was built:

- `GoVersion` holds the version of Go used to build the binary.
- `Settings` is a slice of `BuildSettings` structs holding key/value pairs describing the build.

### runtime/pprof

The CPU profiler now uses per-thread timers on Linux. This increases the maximum CPU usage that a profile can observe, and reduces some forms of bias.

### strconv

`strconv.Unquote` now rejects Unicode surrogate halves.

### strings

The new `Cut` function slices a `string` around a separator. It can replace and simplify many common uses of `Index`, `IndexByte`, `IndexRune`, and `SplitN`.

The new `Clone` function copies the input `string` without the returned cloned `string` referencing the input string's memory.

`Trim`, `TrimLeft`, and `TrimRight` are now allocation free and, especially for small ASCII cutsets, up to 10 times faster.

The `Title` function is now deprecated. It doesn't handle Unicode punctuation and language-specific capitalization rules, and is superseded by the golang.org/x/text/cases package.

### sync

The new methods `Mutex.TryLock`, `RWMutex.TryLock`, and `RWMutex.TryRLock`, will acquire the lock if it is not currently held.

### syscall

The new function `SyscallN` has been introduced for Windows, allowing for calls with arbitrary number of arguments. As a result, `Syscall`, `Syscall6`, `Syscall9`, `Syscall12`, `Syscall15`, and `Syscall18` are deprecated in favor of `SyscallN`.

`SysProcAttr.Pdeathsig` is now supported in FreeBSD.

### syscall/js

The `Wrapper` interface has been removed.

### testing

The precedence of `/` in the argument for `-run` and `-bench` has been increased. `A/B|C/D` used to be treated as `A/(B|C)/D` and is now treated as `(A/B)|(C/D)`.

If the `-run` option does not select any tests, the `-count` option is ignored. This could change the behavior of existing tests in the unlikely case that a test changes the set of subtests that are run each time the test function itself is run.

The new `testing.F` type is used by the new fuzzing support described above. Tests also now support the command line options `-test.fuzz`, `-test.fuzztime`, and `-test.fuzzminimizetime`.

### text/template

Within a `range` pipeline the new `{{break}}` command will end the loop early and the new `{{continue}}` command will immediately start the next loop iteration.

The `and` function no longer always evaluates all arguments; it stops evaluating arguments after the first argument that evaluates to false. Similarly, the `or` function now stops evaluating arguments after the first argument that evaluates to true. This makes a difference if any of the arguments is a function call.

### text/template/parse

The package supports the new text/template and html/template `{{break}}` command via the new constant `NodeBreak` and the new type `BreakNode`, and similarly supports the new `{{continue}}` command via the new constant `NodeContinue` and the new type `ContinueNode`.

### unicode/utf8

The new AppendRune function appends the UTF-8 encoding of a `rune` to a `[]byte`.