Go 1.21 Release Notes

Table of Contents

Introduction to Go 1.21

The latest Go release, version 1.21, arrives six months after Go 1.20. Most of its changes are in the implementation of the toolchain, runtime, and libraries. As always, the release maintains the Go 1 promise of compatibility; in fact, Go 1.21 improves upon that promise. We expect almost all Go programs to continue to compile and run as before.

Go 1.21 introduces a small change to the numbering of releases. In the past, we used Go 1.*N* to refer to both the overall Go language version and release family as well as the first release in that family. Starting in Go 1.21, the first release is now Go 1.*N*.0. Today we are releasing both the Go 1.21 language and its initial implementation, the Go 1.21.0 release. These notes refer to "Go 1.21"; tools like `go version` will report "`go1.21.0`" (until you upgrade to Go 1.21.1). See "Go versions" in the "Go Toolchains" documentation for details about the new version numbering.

Changes to the language

Go 1.21 adds three new built-ins to the language.

- The new functions `min` and `max` compute the smallest (or largest, for `max`) value of a fixed number of given arguments. See the language spec for details.
- The new function `clear` deletes all elements from a map or zeroes all elements of a slice. See the language spec for details.

Package initialization order is now specified more precisely. The new algorithm is:

- Sort all packages by import path.

- Repeat until the list of packages is empty:
    - Find the first package in the list for which all imports are already initialized.
    - Initialize that package and remove it from the list.

This may change the behavior of some programs that rely on a specific initialization ordering that was not expressed by explicit imports. The behavior of such programs was not well defined by the spec in past releases. The new rule provides an unambiguous definition.

Multiple improvements that increase the power and precision of type inference have been made.

- A (possibly partially instantiated generic) function may now be called with arguments that are themselves (possibly partially instantiated) generic functions. The compiler will attempt to infer the missing type arguments of the callee (as before) and, for each argument that is a generic function that is not fully instantiated, its missing type arguments (new). Typical use cases are calls to generic functions operating on containers (such as slices.IndexFunc) where a function argument may also be generic, and where the type argument of the called function and its arguments are inferred from the container type. More generally, a generic function may now be used without explicit instantiation when it is assigned to a variable or returned as a result value if the type arguments can be inferred from the assignment.
- Type inference now also considers methods when a value is assigned to an interface: type arguments for type parameters used in method signatures may be inferred from the corresponding parameter types of matching methods.
- Similarly, since a type argument must implement all the methods of its corresponding constraint, the methods of the type argument and constraint are matched which may lead to the inference of additional type arguments.
- If multiple untyped constant arguments of different kinds (such as an untyped int and an untyped floating-point constant) are passed to parameters with the same (not otherwise specified) type parameter type, instead of an error, now type inference determines the type using the same approach as an operator with untyped constant operands. This change brings the types inferred from untyped constant arguments in line with the types of constant expressions.
- Type inference is now precise when matching corresponding types in assignments: component types (such as the elements of slices, or the parameter types in function signatures) must be identical (given suitable type arguments) to match, otherwise inference fails. This change produces more accurate error messages: where in the past type inference may have succeeded incorrectly and lead to an invalid assignment, the compiler now reports an inference error if two types can't possibly match.

More generally, the description of type inference in the language spec has been clarified. Together, all these changes make type inference more powerful and inference failures less

surprising.

Go 1.21 includes a preview of a language change we are considering for a future version of Go: making for loop variables per-iteration instead of per-loop, to avoid accidental sharing bugs. For details about how to try that language change, see the LoopvarExperiment wiki page.

Go 1.21 now defines that if a goroutine is panicking and recover was called directly by a deferred function, the return value of recover is guaranteed not to be nil. To ensure this, calling panic with a nil interface value (or an untyped nil) causes a run-time panic of type `*runtime.PanicNilError`.

To support programs written for older versions of Go, nil panics can be re-enabled by setting `GODEBUG=panicnil=1`. This setting is enabled automatically when compiling a program whose main package is in a module that declares `go 1.20` or earlier.

Tools

Go 1.21 adds improved support for backwards compatibility and forwards compatibility in the Go toolchain.

To improve backwards compatibility, Go 1.21 formalizes Go's use of the GODEBUG environment variable to control the default behavior for changes that are non-breaking according to the compatibility policy but nonetheless may cause existing programs to break. (For example, programs that depend on buggy behavior may break when a bug is fixed, but bug fixes are not considered breaking changes.) When Go must make this kind of behavior change, it now chooses between the old and new behavior based on the `go` line in the workspace's `go.work` file or else the main module's `go.mod` file. Upgrading to a new Go toolchain but leaving the `go` line set to its original (older) Go version preserves the behavior of the older toolchain. With this compatibility support, the latest Go toolchain should always be the best, most secure, implementation of an older version of Go. See "Go, Backwards Compatibility, and GODEBUG" for details.

To improve forwards compatibility, Go 1.21 now reads the `go` line in a `go.work` or `go.mod` file as a strict minimum requirement: `go 1.21.0` means that the workspace or module cannot be used with Go 1.20 or with Go 1.21rc1. This allows projects that depend on fixes made in later versions of Go to ensure that they are not used with earlier versions. It also gives better error reporting for projects that make use of new Go features: when the problem is that a newer Go version is needed, that problem is reported clearly, instead of attempting to build the code and printing errors about unresolved imports or syntax errors.

To make these new stricter version requirements easier to manage, the `go` command can now invoke not just the toolchain bundled in its own release but also other Go toolchain versions found in the PATH or downloaded on demand. If a `go.mod` or `go.work` `go` line declares a minimum requirement on a newer version of Go, the `go` command will find and run that version

automatically. The new `toolchain` directive sets a suggested minimum toolchain to use, which may be newer than the strict `go` minimum. See "Go Toolchains" for details.

Go command

The `-pgo` build flag now defaults to `-pgo=auto`, and the restriction of specifying a single main package on the command line is now removed. If a file named `default.pgo` is present in the main package's directory, the `go` command will use it to enable profile-guided optimization for building the corresponding program.

The `-C dir` flag must now be the first flag on the command-line when used.

The new `go test` option `-fullpath` prints full path names in test log messages, rather than just base names.

The `go test -c` flag now supports writing test binaries for multiple packages, each to `pkg.test` where `pkg` is the package name. It is an error if more than one test package being compiled has a given package name.

The `go test -o` flag now accepts a directory argument, in which case test binaries are written to that directory instead of the current directory.

When using an external (C) linker with cgo enabled, the `runtime/cgo` package is now supplied to the Go linker as an additional dependency to ensure that the Go runtime is compatible with any additional libraries added by the C linker.

Cgo

In files that `import "C"`, the Go toolchain now correctly reports errors for attempts to declare Go methods on C types.

Runtime

When printing very deep stacks, the runtime now prints the first 50 (innermost) frames followed by the bottom 50 (outermost) frames, rather than just printing the first 100 frames. This makes it easier to see how deeply recursive stacks started, and is especially valuable for debugging stack overflows.

On Linux platforms that support transparent huge pages, the Go runtime now manages which parts of the heap may be backed by huge pages more explicitly. This leads to better utilization of memory: small heaps should see less memory used (up to 50% in pathological cases) while large heaps should see fewer broken huge pages for dense parts of the heap, improving CPU usage and latency by up to 1%. A consequence of this change is that the runtime no longer tries to work around a particular problematic Linux configuration setting, which may result in higher memory overheads. The recommended fix is to adjust the OS's huge page settings according to

the GC guide. However, other workarounds are available as well. See the section on `max_ptes_none`.

As a result of runtime-internal garbage collection tuning, applications may see up to a 40% reduction in application tail latency and a small decrease in memory use. Some applications may also observe a small loss in throughput. The memory use decrease should be proportional to the loss in throughput, such that the previous release's throughput/memory tradeoff may be recovered (with little change to latency) by increasing `GOGC` and/or `GOMEMLIMIT` slightly.

Calls from C to Go on threads created in C require some setup to prepare for Go execution. On Unix platforms, this setup is now preserved across multiple calls from the same thread. This significantly reduces the overhead of subsequent C to Go calls from ~1-3 microseconds per call to ~100-200 nanoseconds per call.

Compiler

Profile-guide optimization (PGO), added as a preview in Go 1.20, is now ready for general use. PGO enables additional optimizations on code identified as hot by profiles of production workloads. As mentioned in the Go command section, PGO is enabled by default for binaries that contain a `default.pgo` profile in the main package directory. Performance improvements vary depending on application behavior, with most programs from a representative set of Go programs seeing between 2 and 7% improvement from enabling PGO. See the PGO user guide for detailed documentation.

PGO builds can now devirtualize some interface method calls, adding a concrete call to the most common callee. This enables further optimization, such as inlining the callee.

Go 1.21 improves build speed by up to 6%, largely thanks to building the compiler itself with PGO.

Assembler

On amd64, frameless nosplit assembly functions are no longer automatically marked as `NOFRAME`. Instead, the `NOFRAME` attribute must be explicitly specified if desired, which is already the behavior on other architectures supporting frame pointers. With this, the runtime now maintains the frame pointers for stack transitions.

The verifier that checks for incorrect uses of `R15` when dynamic linking on amd64 has been improved.

Linker

On windows/amd64, the linker (with help from the compiler) now emits SEH unwinding data by default, which improves the integration of Go applications with Windows debuggers and other tools.

In Go 1.21 the linker (with help from the compiler) is now capable of deleting dead (unreferenced) global map variables, if the number of entries in the variable initializer is sufficiently large, and if the initializer expressions are side-effect free.

Standard library

New log/slog package

The new log/slog package provides structured logging with levels. Structured logging emits key-value pairs to enable fast, accurate processing of large amounts of log data. The package supports integration with popular log analysis tools and services.

New testing/slogtest package

The new testing/slogtest package can help to validate slog.Handler implementations.

New slices package

The new slices package provides many common operations on slices, using generic functions that work with slices of any element type.

New maps package

The new maps package provides several common operations on maps, using generic functions that work with maps of any key or element type.

New cmp package

The new cmp package defines the type constraint `Ordered` and two new generic functions `Less` and `Compare` that are useful with ordered types.

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. There are also various performance improvements, not enumerated here.

archive/tar

The implementation of the `io/fs.FileInfo` interface returned by `Header.FileInfo` now implements a `String` method that calls `io/fs.FormatFileInfo`.

archive/zip

The implementation of the `io/fs.FileInfo` interface returned by `FileHeader.FileInfo` now implements a `String` method that calls `io/fs.FormatFileInfo`.

The implementation of the `io/fs.DirEntry` interface returned by the `io/fs.ReadDirFile.ReadDir` method of the `io/fs.File` returned by `Reader.Open` now implements a `String` method that calls `io/fs.FormatDirEntry`.

bytes

The `Buffer` type has two new methods: `Available` and `AvailableBuffer`. These may be used along with the `Write` method to append directly to the `Buffer`.

context

The new `WithoutCancel` function returns a copy of a context that is not canceled when the original context is canceled.

The new `WithDeadlineCause` and `WithTimeoutCause` functions provide a way to set a context cancellation cause when a deadline or timer expires. The cause may be retrieved with the `Cause` function.

The new `AfterFunc` function registers a function to run after a context has been cancelled.

An optimization means that the results of calling `Background` and `TODO` and converting them to a shared type can be considered equal. In previous releases they were always different. Comparing `Context` values for equality has never been well-defined, so this is not considered to be an incompatible change.

crypto/ecdsa

`PublicKey.Equal` and `PrivateKey.Equal` now execute in constant time.

crypto/elliptic

All of the `Curve` methods have been deprecated, along with `GenerateKey`, `Marshal`, and `Unmarshal`. For ECDH operations, the new `crypto/ecdh` package should be used instead. For lower-level operations, use third-party modules such as filippo.io/nistec.

crypto/rand

The `crypto/rand` package now uses the `getrandom` system call on NetBSD 10.0 and later.

crypto/rsa

The performance of private RSA operations (decryption and signing) is now better than Go 1.19 for `GOARCH=amd64` and `GOARCH=arm64`. It had regressed in Go 1.20.

Due to the addition of private fields to `PrecomputedValues`, `PrivateKey.Precompute` must be called for optimal performance even if deserializing (for example from JSON) a previously-precomputed private key.

`PublicKey.Equal` and `PrivateKey.Equal` now execute in constant time.

The `GenerateMultiPrimeKey` function and the `PrecomputedValues.CRTValues` field have been deprecated. `PrecomputedValues.CRTValues` will still be populated when `PrivateKey.Precompute` is called, but the values will not be used during decryption operations.

crypto/sha256

SHA-224 and SHA-256 operations now use native instructions when available when `GOARCH=amd64`, providing a performance improvement on the order of 3-4x.

crypto/tls

Servers now skip verifying client certificates (including not running `Config.VerifyPeerCertificate`) for resumed connections, besides checking the expiration time. This makes session tickets larger when client certificates are in use. Clients were already skipping verification on resumption, but now check the expiration time even if `Config.InsecureSkipVerify` is set.

Applications can now control the content of session tickets.

- The new `SessionState` type describes a resumable session.
- The `SessionState.Bytes` method and `ParseSessionState` function serialize and deserialize a `SessionState`.
- The `Config.WrapSession` and `Config.UnwrapSession` hooks convert a `SessionState` to and from a ticket on the server side.
- The `Config.EncryptTicket` and `Config.DecryptTicket` methods provide a default implementation of `WrapSession` and `UnwrapSession`.
- The `ClientSessionState.ResumptionState` method and `NewResumptionState` function may be used by a `ClientSessionCache` implementation to store and resume sessions on the client side.

To reduce the potential for session tickets to be used as a tracking mechanism across connections, the server now issues new tickets on every resumption (if they are supported and not disabled) and tickets don't bear an identifier for the key that encrypted them anymore. If passing a large number of keys to `Conn.SetSessionTicketKeys`, this might lead to a noticeable performance cost.

Both clients and servers now implement the Extended Master Secret extension (RFC 7627). The deprecation of `ConnectionState.TLSUnique` has been reverted, and is now set for resumed connections that support Extended Master Secret.

The new `QUICConn` type provides support for QUIC implementations, including 0-RTT support. Note that this is not itself a QUIC implementation, and 0-RTT is still not supported in TLS.

The new `VersionName` function returns the name for a TLS version number.

The TLS alert codes sent from the server for client authentication failures have been improved. Previously, these failures always resulted in a "bad certificate" alert. Now, certain failures will result in more appropriate alert codes, as defined by RFC 5246 and RFC 8446:

- For TLS 1.3 connections, if the server is configured to require client authentication using RequireAnyClientCert or RequireAndVerifyClientCert, and the client does not provide any certificate, the server will now return the "certificate required" alert.
- If the client provides a certificate that is not signed by the set of trusted certificate authorities configured on the server, the server will return the "unknown certificate authority" alert.
- If the client provides a certificate that is either expired or not yet valid, the server will return the "expired certificate" alert.
- In all other scenarios related to client authentication failures, the server still returns "bad certificate".

crypto/x509

`RevocationList.RevokedCertificates` has been deprecated and replaced with the new `RevokedCertificateEntries` field, which is a slice of `RevocationListEntry`. `RevocationListEntry` contains all of the fields in `pkix.RevokedCertificate`, as well as the revocation reason code.

Name constraints are now correctly enforced on non-leaf certificates, and not on the certificates where they are expressed.

debug/elf

The new `File.DynValue` method may be used to retrieve the numeric values listed with a given dynamic tag.

The constant flags permitted in a `DT_FLAGS_1` dynamic tag are now defined with type `DynFlag1`. These tags have names starting with `DF_1`.

The package now defines the constant `COMPRESS_ZSTD`.

The package now defines the constant `R_PPC64_REL24_P9NOTOC`.

debug/pe

Attempts to read from a section containing uninitialized data using `Section.Data` or the reader returned by `Section.Open` now return an error.

embed

The `io/fs.File` returned by `FS.Open` now has a `ReadAt` method that implements `io.ReaderAt`.

Calling `FS.Open.Stat` will return a type that now implements a `String` method that calls `io/fs.FormatFileInfo`.

encoding/binary

The new `NativeEndian` variable may be used to convert between byte slices and integers using the current machine's native endianness.

errors

The new `ErrUnsupported` error provides a standardized way to indicate that a requested operation may not be performed because it is unsupported. For example, a call to `os.Link` when using a file system that does not support hard links.

flag

The new `BoolFunc` function and `FlagSet.BoolFunc` method define a flag that does not require an argument and calls a function when the flag is used. This is similar to `Func` but for a boolean flag.

A flag definition (via `Bool`, `BoolVar`, `Int`, `IntVar`, etc.) will panic if `Set` has already been called on a flag with the same name. This change is intended to detect cases where changes in initialization order cause flag operations to occur in a different order than expected. In many cases the fix to this problem is to introduce a explicit package dependence to correctly order the definition before any `Set` operations.

go/ast

The new `IsGenerated` predicate reports whether a file syntax tree contains the special comment that conventionally indicates that the file was generated by a tool.

The new `File.GoVersion` field records the minimum Go version required by any `//go:build` or `// +build` directives.

go/build

The package now parses build directives (comments that start with `//go:`) in file headers (before the `package` declaration). These directives are available in the new `Package` fields `Directives`, `TestDirectives`, and `XTestDirectives`.

go/build/constraint

The new `GoVersion` function returns the minimum Go version implied by a build expression.

go/token

The new `File.Lines` method returns the file's line-number table in the same form as accepted by `File.SetLines`.

go/types

The new `Package.GoVersion` method returns the Go language version used to check the package.

hash/maphash

The `hash/maphash` package now has a pure Go implementation, selectable with the `purego` build tag.

html/template

The new error `ErrJSTemplate` is returned when an action appears in a JavaScript template literal. Previously an unexported error was returned.

io/fs

The new `FormatFileInfo` function returns a formatted version of a `FileInfo`. The new `FormatDirEntry` function returns a formatted version of a `DirEntry`. The implementation of `DirEntry` returned by `ReadDir` now implements a `String` method that calls `FormatDirEntry`, and the same is true for the `DirEntry` value passed to `WalkDirFunc`.

math/big

The new `Int.Float64` method returns the nearest floating-point value to a multi-precision integer, along with an indication of any rounding that occurred.

net

On Linux, the net package can now use Multipath TCP when the kernel supports it. It is not used by default. To use Multipath TCP when available on a client, call the `Dialer.SetMultipathTCP` method before calling the `Dialer.Dial` or `Dialer.DialContext` methods. To use Multipath TCP when available on a server, call the `ListenConfig.SetMultipathTCP` method before calling the `ListenConfig.Listen` method. Specify the network as `"tcp"` or `"tcp4"` or `"tcp6"` as usual. If Multipath TCP is not supported by the kernel or the remote host, the connection will silently fall back to TCP. To test whether a particular connection is using Multipath TCP, use the `TCPConn.MultipathTCP` method.

In a future Go release we may enable Multipath TCP by default on systems that support it.

net/http

The new `ResponseController.EnableFullDuplex` method allows server handlers to concurrently read from an HTTP/1 request body while writing the response. Normally, the HTTP/1 server automatically consumes any remaining request body before starting to write the response, to avoid deadlocking clients which attempt to write a complete request before reading the response. The `EnableFullDuplex` method disables this behavior.

The new `ErrSchemeMismatch` error is returned by `Client` and `Transport` when the server responds to an HTTPS request with an HTTP response.

The net/http package now supports `errors.ErrUnsupported`, in that the expression `errors.Is(http.ErrNotSupported, errors.ErrUnsupported)` will return true.

os

Programs may now pass an empty `time.Time` value to the `Chtimes` function to leave either the access time or the modification time unchanged.

On Windows the `File.Chdir` method now changes the current directory to the file, rather than always returning an error.

On Unix systems, if a non-blocking descriptor is passed to `NewFile`, calling the `File.Fd` method will now return a non-blocking descriptor. Previously the descriptor was converted to blocking mode.

On Windows calling `Truncate` on a non-existent file used to create an empty file. It now returns an error indicating that the file does not exist.

On Windows calling `TempDir` now uses GetTempPath2W when available, instead of GetTempPathW. The new behavior is a security hardening measure that prevents temporary files created by processes running as SYSTEM to be accessed by non-SYSTEM processes.

On Windows the os package now supports working with files whose names, stored as UTF-16, can't be represented as valid UTF-8.

On Windows `Lstat` now resolves symbolic links for paths ending with a path separator, consistent with its behavior on POSIX platforms.

The implementation of the `io/fs.DirEntry` interface returned by the `ReadDir` function and the `File.ReadDir` method now implements a `String` method that calls `io/fs.FormatDirEntry`.

The implementation of the `io/fs.FS` interface returned by the `DirFS` function now implements the `io/fs.ReadFileFS` and the `io/fs.ReadDirFS` interfaces.

path/filepath

The implementation of the `io/fs.DirEntry` interface passed to the function argument of `WalkDir` now implements a `String` method that calls `io/fs.FormatDirEntry`.

reflect

In Go 1.21, `ValueOf` no longer forces its argument to be allocated on the heap, allowing a `Value`'s content to be allocated on the stack. Most operations on a `Value` also allow the underlying value to be stack allocated.

The new `Value` method `Value.Clear` clears the contents of a map or zeros the contents of a slice. This corresponds to the new `clear` built-in added to the language.

The `SliceHeader` and `StringHeader` types are now deprecated. In new code prefer `unsafe.Slice`, `unsafe.SliceData`, `unsafe.String`, or `unsafe.StringData`.

regexp

`Regexp` now defines `MarshalText` and `UnmarshalText` methods. These implement `encoding.TextMarshaler` and `encoding.TextUnmarshaler` and will be used by packages such as encoding/json.

runtime

Textual stack traces produced by Go programs, such as those produced when crashing, calling `runtime.Stack`, or collecting a goroutine profile with `debug=2`, now include the IDs of the goroutines that created each goroutine in the stack trace.

Crashing Go applications can now opt-in to Windows Error Reporting (WER) by setting the environment variable `GOTRACEBACK=wer` or calling `debug.SetTraceback("wer")` before the crash. Other than enabling WER, the runtime will behave as with `GOTRACEBACK=crash`. On non-Windows systems, `GOTRACEBACK=wer` is ignored.

`GODEBUG=cgocheck=2`, a thorough checker of cgo pointer passing rules, is no longer available as a debug option. Instead, it is available as an experiment using `GOEXPERIMENT=cgocheck2`. In particular this means that this mode has to be selected at build time instead of startup time.

`GODEBUG=cgocheck=1` is still available (and is still the default).

A new type `Pinner` has been added to the runtime package. `Pinners` may be used to "pin" Go memory such that it may be used more freely by non-Go code. For instance, passing Go values that reference pinned Go memory to C code is now allowed. Previously, passing any such nested reference was disallowed by the cgo pointer passing rules. See the docs for more details.

runtime/metrics

A few previously-internal GC metrics, such as live heap size, are now available. `GOGC` and `GOMEMLIMIT` are also now available as metrics.

runtime/trace

Collecting traces on amd64 and arm64 now incurs a substantially smaller CPU cost: up to a 10x improvement over the previous release.

Traces now contain explicit stop-the-world events for every reason the Go runtime might stop-the-world, not just garbage collection.

sync

The new `OnceFunc`, `OnceValue`, and `OnceValues` functions capture a common use of `Once` to lazily initialize a value on first use.

syscall

On Windows the `Fchdir` function now changes the current directory to its argument, rather than always returning an error.

On FreeBSD `SysProcAttr` has a new field `Jail` that may be used to put the newly created process in a jailed environment.

On Windows the syscall package now supports working with files whose names, stored as UTF-16, can't be represented as valid UTF-8. The `UTF16ToString` and `UTF16FromString` functions now convert between UTF-16 data and `WTF-8` strings. This is backward compatible as WTF-8 is a superset of the UTF-8 format that was used in earlier releases.

Several error values match the new `errors.ErrUnsupported`, such that `errors.Is(err, errors.ErrUnsupported)` returns true.

- `ENOSYS`
- `ENOTSUP`
- `EOPNOTSUPP`
- `EPLAN9` (Plan 9 only)
- `ERROR_CALL_NOT_IMPLEMENTED` (Windows only)
- `ERROR_NOT_SUPPORTED` (Windows only)
- `EWINDOWS` (Windows only)

testing

The new `-test.fullpath` option will print full path names in test log messages, rather than just base names.

The new `Testing` function reports whether the program is a test created by `go test`.

testing/fstest

Calling `Open.Stat` will return a type that now implements a `String` method that calls `io/fs.FormatFileInfo`.

unicode

The `unicode` package and associated support throughout the system has been upgraded to Unicode 15.0.0.

Ports

Darwin

As announced in the Go 1.20 release notes, Go 1.21 requires macOS 10.15 Catalina or later; support for previous versions has been discontinued.

Windows

As announced in the Go 1.20 release notes, Go 1.21 requires at least Windows 10 or Windows Server 2016; support for previous versions has been discontinued.

ARM

When building the Go distribution with `GOARCH=arm` when not running on an ARM system (that is, when building a cross-compiler to ARM), the default value for the `GOARM` environment variable is now always set to `7`. Previously the default depended on characteristics of the build system.

When not building a cross-compiler, the default value is determined by examining the build system. That was true before and remains true in Go 1.21. What has changed is the behavior when building a cross-compiler.

WebAssembly

The new `go:wasmimport` directive can now be used in Go programs to import functions from the WebAssembly host.

The Go scheduler now interacts much more efficiently with the JavaScript event loop, especially in applications that block frequently on asynchronous events.

WebAssembly System Interface

Go 1.21 adds an experimental port to the WebAssembly System Interface (WASI), Preview 1 (`GOOS=wasip1`, `GOARCH=wasm`).

As a result of the addition of the new `GOOS` value "`wasip1`", Go files named `*_wasip1.go` will now be ignored by Go tools except when that `GOOS` value is being used. If you have existing filenames matching that pattern, you will need to rename them.

ppc64/ppc64le

On Linux, `GOPPC64=power10` now generates PC-relative instructions, prefixed instructions, and other new Power10 instructions. On AIX, `GOPPC64=power10` generates Power10 instructions, but does not generate PC-relative instructions.

When building position-independent binaries for `GOPPC64=power10` `GOOS=linux` `GOARCH=ppc64le`, users can expect reduced binary sizes in most cases, in some cases 3.5%. Position-independent binaries are built for ppc64le with the following `-buildmode` values: `c-archive`, `c-shared`, `shared`, `pie`, `plugin`.

loong64

The `linux/loong64` port now supports `-buildmode=c-archive`, `-buildmode=c-shared` and `-buildmode=pie`.