Go 1.2 Release Notes

- Changes to the language {#language}
  - Use of nil {#use_of_nil}
  - Three-index slices {#three_index}
- Changes to the implementations and tools {#impl}
  - Pre-emption in the scheduler {#preemption}
  - Limit on the number of threads {#thread_limit}
  - Stack size {#stack_size}
  - Cgo and C++ {#cgo_and_cpp}
  - Godoc and vet moved to the go.tools subrepository {#go_tools_godoc}
  - Status of gccgo {#gccgo}
  - Changes to the gc compiler and linker {#gc_changes}
  - Test coverage {#cover}
  - The go doc command is deleted {#go_doc}
  - Changes to the go command {#gocmd}
- Performance {#performance}
- Changes to the standard library {#library}
  - The archive/tar and archive/zip packages {#archive_tar_zip}
  - The new encoding package {#encoding}
  - The fmt package {#fmt_indexed_arguments}
  - The text/template and html/template packages {#text_template}
  - New packages {#new_packages}
  - Minor changes to the library {#minor_library_changes}

Since the release of Go version 1.1 in April, 2013, the release schedule has been shortened to make the release process more efficient. This release, Go version 1.2 or Go 1.2 for short, arrives roughly six months after 1.1, while 1.1 took over a year to appear after 1.0. Because of the shorter time scale, 1.2 is a smaller delta than the step from 1.0 to 1.1, but it still has some significant developments, including a better scheduler and one new language feature. Of course, Go 1.2 keeps the promise of compatibility. The overwhelming majority of programs built with Go 1.1 (or 1.0 for

that matter) will run without any changes whatsoever when moved to 1.2, although the introduction of one restriction to a corner of the language may expose already-incorrect code (see the discussion of the use of nil).

Changes to the language {#language}

In the interest of firming up the specification, one corner case has been clarified, with consequences for programs. There is also one new language feature.

Use of nil {#use_of_nil}

The language now specifies that, for safety reasons, certain uses of nil pointers are guaranteed to trigger a run-time panic. For instance, in Go 1.0, given code like

```
type T struct {
    X [1<<24]byte
    Field int32
}

func main() {
    var x *T
    ...
}
```

the `nil` pointer `x` could be used to access memory incorrectly: the expression `x.Field` could access memory at address `1<<24`. To prevent such unsafe behavior, in Go 1.2 the compilers now guarantee that any indirection through a nil pointer, such as illustrated here but also in nil pointers to arrays, nil interface values, nil slices, and so on, will either panic or return a correct, safe non-nil value. In short, any expression that explicitly or implicitly requires evaluation of a nil address is an error. The implementation may inject extra tests into the compiled program to enforce this behavior.

Further details are in the design document.

Updating: Most code that depended on the old behavior is erroneous and will fail when run. Such programs will need to be updated by hand.

Three-index slices {#three_index}

Go 1.2 adds the ability to specify the capacity as well as the length when using a slicing operation on an existing array or slice. A slicing operation creates a new slice by describing a contiguous section of an already-created array or slice:

```
var array [10]int
slice := array[2:4]
```

The capacity of the slice is the maximum number of elements that the slice may hold, even after reslicing; it reflects the size of the underlying array. In this example, the capacity of the `slice` variable is 8.

Go 1.2 adds new syntax to allow a slicing operation to specify the capacity as well as the length. A second colon introduces the capacity value, which must be less than or equal to the capacity of the source slice or array, adjusted for the origin. For instance,

```
slice = array[2:4:7]
```

sets the slice to have the same length as in the earlier example but its capacity is now only 5 elements (7-2). It is impossible to use this new slice value to access the last three elements of the original array.

In this three-index notation, a missing first index (`[:i:j]`) defaults to zero but the other two indices must always be specified explicitly. It is possible that future releases of Go may introduce default values for these indices.

Further details are in the design document.

Updating: This is a backwards-compatible change that affects no existing programs.

Changes to the implementations and tools {#impl}

Pre-emption in the scheduler {#preemption}

In prior releases, a goroutine that was looping forever could starve out other goroutines on the same thread, a serious problem when GOMAXPROCS provided only one user thread. In Go 1.2, this is partially addressed: The scheduler is invoked occasionally upon entry to a function. This means that any loop that includes a (non-inlined) function call can be pre-empted, allowing other goroutines to run on the same thread.

Limit on the number of threads {#thread_limit}

Go 1.2 introduces a configurable limit (default 10,000) to the total number of threads a single program may have in its address space, to avoid resource starvation issues in some environments. Note that goroutines are multiplexed onto threads so this limit does not directly limit the number of goroutines, only the number that may be simultaneously blocked in a system call. In practice, the limit is hard to reach.

The new `SetMaxThreads` function in the `runtime/debug` package controls the thread count limit.

Updating: Few functions will be affected by the limit, but if a program dies because it hits the limit, it could be modified to call `SetMaxThreads` to set a higher count. Even better would be to refactor the program to need fewer threads, reducing consumption of kernel resources.

Stack size {#stack_size}

In Go 1.2, the minimum size of the stack when a goroutine is created has been lifted from 4KB to 8KB. Many programs were suffering performance problems with the old size, which had a tendency to introduce expensive stack-segment switching in performance-critical sections. The new number was determined by empirical testing.

At the other end, the new function `SetMaxStack` in the `runtime/debug` package controls the *maximum* size of a single goroutine's stack. The default is 1GB on 64-bit systems and 250MB on 32-bit systems. Before Go 1.2, it was too easy for a runaway recursion to consume all the memory on a machine.

Updating: The increased minimum stack size may cause programs with many goroutines to use more memory. There is no workaround, but plans for future releases include new stack management technology that should address the problem better.

Cgo and C++ {#cgo_and_cpp}

The `cgo` command will now invoke the C++ compiler to build any pieces of the linked-to library that are written in C++; the documentation has more detail.

Godoc and vet moved to the go.tools subrepository {#go_tools_godoc}

Both binaries are still included with the distribution, but the source code for the godoc and vet commands has moved to the go.tools subrepository.

Also, the core of the godoc program has been split into a library, while the command itself is in a separate directory. The move allows the code to be updated easily and the separation into a library and command makes it easier to construct custom binaries for local sites and different deployment methods.

Updating: Since godoc and vet are not part of the library, no client Go code depends on their source and no updating is required.

The binary distributions available from golang.org include these binaries, so users of these distributions are unaffected.

When building from source, users must use "go get" to install godoc and vet. (The binaries will continue to be installed in their usual locations, not `$GOPATH/bin`.)

```
$ go get code.google.com/p/go.tools/cmd/godoc
$ go get code.google.com/p/go.tools/cmd/vet
```

Status of gccgo {#gccgo}

We expect the future GCC 4.9 release to include gccgo with full support for Go 1.2. In the current (4.8.2) release of GCC, gccgo implements Go 1.1.2.

Changes to the gc compiler and linker {#gc_changes}

Go 1.2 has several semantic changes to the workings of the gc compiler suite. Most users will be unaffected by them.

The `cgo` command now works when C++ is included in the library being linked against. See the `cgo` documentation for details.

The gc compiler displayed a vestigial detail of its origins when a program had no `package` clause: it assumed the file was in package `main`. The past has been erased, and a missing `package` clause is now an error.

On the ARM, the toolchain supports "external linking", which is a step towards being able to build shared libraries with the gc toolchain and to provide dynamic linking support for environments in which that is necessary.

In the runtime for the ARM, with `5a`, it used to be possible to refer to the runtime-internal `m` (machine) and `g` (goroutine) variables using `R9` and `R10` directly. It is now necessary to refer to them by their proper names.

Also on the ARM, the `5l` linker (sic) now defines the `MOVBS` and `MOVHS` instructions as synonyms of `MOVB` and `MOVH`, to make clearer the separation between signed and unsigned sub-word moves; the unsigned versions already existed with a `U` suffix.

Test coverage {#cover}

One major new feature of `go test` is that it can now compute and, with help from a new, separately installed "go tool cover" program, display test coverage results.

The cover tool is part of the `go.tools` subrepository. It can be installed by running

```
$ go get code.google.com/p/go.tools/cmd/cover
```

The cover tool does two things. First, when "go test" is given the `-cover` flag, it is run automatically to rewrite the source for the package and insert instrumentation statements. The test is then compiled and run as usual, and basic coverage statistics are reported:

```
$ go test -cover fmt
ok  	fmt	0.060s	coverage: 91.4% of statements
$
```

Second, for more detailed reports, different flags to "go test" can create a coverage profile file, which the cover program, invoked with "go tool cover", can then analyze.

Details on how to generate and analyze coverage statistics can be found by running the commands

```
$ go help testflag
$ go tool cover -help
```

The go doc command is deleted {#go_doc}

The "go doc" command is deleted. Note that the `godoc` tool itself is not deleted, just the wrapping of it by the `go` command. All it did was show the documents for a package by package path, which godoc itself already does with more flexibility. It has therefore been deleted to reduce the number of documentation tools and, as part of the restructuring of godoc, encourage better options in future.

Updating: For those who still need the precise functionality of running

```
$ go doc
```

in a directory, the behavior is identical to running

```
$ godoc .
```

Changes to the go command {#gocmd}

The `go get` command now has a `-t` flag that causes it to download the dependencies of the tests run by the package, not just those of the package itself. By default, as before, dependencies of the tests are not downloaded.

Performance {#performance}

There are a number of significant performance improvements in the standard library; here are a few of them.

- The `compress/bzip2` decompresses about 30% faster.
- The `crypto/des` package is about five times faster.
- The `encoding/json` package encodes about 30% faster.
- Networking performance on Windows and BSD systems is about 30% faster through the use of an integrated network poller in the runtime, similar to what was done for Linux and OS X in Go 1.1.

Changes to the standard library {#library}

The archive/tar and archive/zip packages {#archive_tar_zip}

The `archive/tar` and `archive/zip` packages have had a change to their semantics that may break existing programs. The issue is that they both provided an implementation of the `os.FileInfo` interface that was not compliant with the specification for that interface. In particular, their `Name` method returned the full path name of the entry, but the interface specification requires that the method return only the base name (final path element).

Updating: Since this behavior was newly implemented and a bit obscure, it is possible that no code depends on the broken behavior. If there are programs that do depend on it, they will need to be identified and fixed manually.

The new encoding package {#encoding}

There is a new package, `encoding`, that defines a set of standard encoding interfaces that may be used to build custom marshalers and unmarshalers for packages such as `encoding/xml`, `encoding/json`, and `encoding/binary`. These new interfaces have been used to tidy up some implementations in the standard library.

The new interfaces are called `BinaryMarshaler`, `BinaryUnmarshaler`, `TextMarshaler`, and `TextUnmarshaler`. Full details are in the documentation for the package and a separate design document.

The fmt package {#fmt_indexed_arguments}

The `fmt` package's formatted print routines such as `Printf` now allow the data items to be printed to be accessed in arbitrary order by using an indexing operation in the formatting specifications. Wherever an argument is to be fetched from the argument list for formatting, either as the value to be formatted or as a width or specification integer, a new optional indexing notation `[`*n*`]` fetches argument *n* instead. The value of *n* is 1-indexed. After such an indexing operating, the next

argument to be fetched by normal processing will be *n*+1.

For example, the normal `Printf` call

```
fmt.Sprintf("%c %c %c\n", 'a', 'b', 'c')
```

would create the string `"a b c"`, but with indexing operations like this,

```
fmt.Sprintf("%[3]c %[1]c %c\n", 'a', 'b', 'c')
```

the result is "`"c a b"`. The `[3]` index accesses the third formatting argument, which is `'c'`, `[1]` accesses the first, `'a'`, and then the next fetch accesses the argument following that one, `'b'`.

The motivation for this feature is programmable format statements to access the arguments in different order for localization, but it has other uses:

```
log.Printf("trace: value %v of type %[1]T\n", expensiveFunction(a.b[c]))
```

Updating: The change to the syntax of format specifications is strictly backwards compatible, so it affects no working programs.

The text/template and html/template packages {#text_template}

The `text/template` package has a couple of changes in Go 1.2, both of which are also mirrored in the `html/template` package.

First, there are new default functions for comparing basic types. The functions are listed in this table, which shows their names and the associated familiar comparison operator.

	Name	Operator
	`eq`	`==`
	`ne`	`!=`
	`lt`	`<`
	`le`	`<=`
	`gt`	`>`
	`ge`	`>=`

These functions behave slightly differently from the corresponding Go operators. First, they operate only on basic types (`bool`, `int`, `float64`, `string`, etc.). (Go allows comparison of arrays and structs as well, under some circumstances.) Second, values can be compared as long as they are

the same sort of value: any signed integer value can be compared to any other signed integer value for example. (Go does not permit comparing an `int8` and an `int16`). Finally, the `eq` function (only) allows comparison of the first argument with one or more following arguments. The template in this example,

```
{{if eq .A 1 2 3}} equal {{else}} not equal {{end}}
```

reports "equal" if `.A` is equal to *any* of 1, 2, or 3.

The second change is that a small addition to the grammar makes "if else if" chains easier to write. Instead of writing,

```
{{if eq .A 1}} X {{else}} {{if eq .A 2}} Y {{end}} {{end}}
```

one can fold the second "if" into the "else" and have only one "end", like this:

```
{{if eq .A 1}} X {{else if eq .A 2}} Y {{end}}
```

The two forms are identical in effect; the difference is just in the syntax.

Updating: Neither the "else if" change nor the comparison functions affect existing programs. Those that already define functions called `eq` and so on through a function map are unaffected because the associated function map will override the new default function definitions.

New packages {#new_packages}

There are two new packages.

- The `encoding` package is described above.
- The `image/color/palette` package provides standard color palettes.

Minor changes to the library {#minor_library_changes}

The following list summarizes a number of minor changes to the library, mostly additions. See the relevant package documentation for more information about each change.

- The `archive/zip` package adds the `DataOffset` accessor to return the offset of a file's (possibly compressed) data within the archive.
- The `bufio` package adds `Reset` methods to `Reader` and `Writer`. These methods allow the `Readers` and `Writers` to be re-used on new input and output readers and writers, saving allocation overhead.

- The `compress/bzip2` can now decompress concatenated archives.
- The `compress/flate` package adds a `Reset` method on the `Writer`, to make it possible to reduce allocation when, for instance, constructing an archive to hold multiple compressed files.
- The `compress/gzip` package's `Writer` type adds a `Reset` so it may be reused.
- The `compress/zlib` package's `Writer` type adds a `Reset` so it may be reused.
- The `container/heap` package adds a `Fix` method to provide a more efficient way to update an item's position in the heap.
- The `container/list` package adds the `MoveBefore` and `MoveAfter` methods, which implement the obvious rearrangement.
- The `crypto/cipher` package adds the new GCM mode (Galois Counter Mode), which is almost always used with AES encryption.
- The `crypto/md5` package adds a new `Sum` function to simplify hashing without sacrificing performance.
- Similarly, the `crypto/sha1` package adds a new `Sum` function.
- Also, the `crypto/sha256` package adds `Sum256` and `Sum224` functions.
- Finally, the `crypto/sha512` package adds `Sum512` and `Sum384` functions.
- The `crypto/x509` package adds support for reading and writing arbitrary extensions.
- The `crypto/tls` package adds support for TLS 1.1, 1.2 and AES-GCM.
- The `database/sql` package adds a `SetMaxOpenConns` method on `DB` to limit the number of open connections to the database.
- The `encoding/csv` package now always allows trailing commas on fields.
- The `encoding/gob` package now treats channel and function fields of structures as if they were unexported, even if they are not. That is, it ignores them completely. Previously they would trigger an error, which could cause unexpected compatibility problems if an embedded structure added such a field. The package also now supports the generic `BinaryMarshaler` and `BinaryUnmarshaler` interfaces of the `encoding` package described above.
- The `encoding/json` package now will always escape ampersands as "\u0026" when printing strings. It will now accept but correct invalid UTF-8 in `Marshal` (such input was previously rejected). Finally, it now supports the generic encoding interfaces of the `encoding` package described above.
- The `encoding/xml` package now allows attributes stored in pointers to be marshaled. It also supports the generic encoding interfaces of the `encoding` package described above through the new `Marshaler`, `Unmarshaler`, and related `MarshalerAttr` and `UnmarshalerAttr` interfaces. The package also adds a `Flush` method to the `Encoder` type for use by custom encoders. See the documentation for `EncodeToken` to see how to use it.
- The `flag` package now has a `Getter` interface to allow the value of a flag to be retrieved. Due to the Go 1 compatibility guidelines, this method cannot be added to the existing `Value` interface, but all the existing standard flag types implement it. The package also now exports the `CommandLine` flag set, which holds the flags from the command line.
- The `go/ast` package's `SliceExpr` struct has a new boolean field, `Slice3`, which is set to true when representing a slice expression with three indices (two colons). The default is false, representing the usual two-index form.
- The `go/build` package adds the `AllTags` field to the `Package` type, to make it easier to process build tags.

- The `image/draw` package now exports an interface, `Drawer`, that wraps the standard `Draw` method. The Porter-Duff operators now implement this interface, in effect binding an operation to the draw operator rather than providing it explicitly. Given a paletted image as its destination, the new `FloydSteinberg` implementation of the `Drawer` interface will use the Floyd-Steinberg error diffusion algorithm to draw the image. To create palettes suitable for such processing, the new `Quantizer` interface represents implementations of quantization algorithms that choose a palette given a full-color image. There are no implementations of this interface in the library.
- The `image/gif` package can now create GIF files using the new `Encode` and `EncodeAll` functions. Their options argument allows specification of an image `Quantizer` to use; if it is `nil`, the generated GIF will use the `Plan9` color map (palette) defined in the new `image/color/palette` package. The options also specify a `Drawer` to use to create the output image; if it is `nil`, Floyd-Steinberg error diffusion is used.
- The `Copy` method of the `io` package now prioritizes its arguments differently. If one argument implements `WriterTo` and the other implements `ReaderFrom`, `Copy` will now invoke `WriterTo` to do the work, so that less intermediate buffering is required in general.
- The `net` package requires cgo by default because the host operating system must in general mediate network call setup. On some systems, though, it is possible to use the network without cgo, and useful to do so, for instance to avoid dynamic linking. The new build tag `netgo` (off by default) allows the construction of a `net` package in pure Go on those systems where it is possible.
- The `net` package adds a new field `DualStack` to the `Dialer` struct for TCP connection setup using a dual IP stack as described in RFC 6555.
- The `net/http` package will no longer transmit cookies that are incorrect according to RFC 6265. It just logs an error and sends nothing. Also, the `net/http` package's `ReadResponse` function now permits the `*Request` parameter to be `nil`, whereupon it assumes a GET request. Finally, an HTTP server will now serve HEAD requests transparently, without the need for special casing in handler code. While serving a HEAD request, writes to a `Handler`'s `ResponseWriter` are absorbed by the `Server` and the client receives an empty body as required by the HTTP specification.
- The `os/exec` package's `Cmd.StdinPipe` method returns an `io.WriteCloser`, but has changed its concrete implementation from `*os.File` to an unexported type that embeds `*os.File`, and it is now safe to close the returned value. Before Go 1.2, there was an unavoidable race that this change fixes. Code that needs access to the methods of `*os.File` can use an interface type assertion, such as `wc.(interface{ Sync() error })`.
- The `runtime` package relaxes the constraints on finalizer functions in `SetFinalizer`: the actual argument can now be any type that is assignable to the formal type of the function, as is the case for any normal function call in Go.
- The `sort` package has a new `Stable` function that implements stable sorting. It is less efficient than the normal sort algorithm, however.
- The `strings` package adds an `IndexByte` function for consistency with the `bytes` package.
- The `sync/atomic` package adds a new set of swap functions that atomically exchange the argument with the value stored in the pointer, returning the old value. The functions are `SwapInt32`, `SwapInt64`, `SwapUint32`, `SwapUint64`, `SwapUintptr`, and `SwapPointer`, which swaps an `unsafe.Pointer`.

- The `syscall` package now implements `Sendfile` for Darwin.
- The `testing` package now exports the `TB` interface. It records the methods in common with the `T` and `B` types, to make it easier to share code between tests and benchmarks. Also, the `AllocsPerRun` function now quantizes the return value to an integer (although it still has type `float64`), to round off any error caused by initialization and make the result more repeatable.
- The `text/template` package now automatically dereferences pointer values when evaluating the arguments to "escape" functions such as "html", to bring the behavior of such functions in agreement with that of other printing functions such as "printf".
- In the `time` package, the `Parse` function and `Format` method now handle time zone offsets with seconds, such as in the historical date "1871-01-01T05:33:02+00:34:08". Also, pattern matching in the formats for those routines is stricter: a non-lowercase letter must now follow the standard words such as "Jan" and "Mon".
- The `unicode` package adds `In`, a nicer-to-use but equivalent version of the original `IsOneOf`, to see whether a character is a member of a Unicode category.

□□□ #2
□ □□□ □□□ 2026-10-26 20:26:29 UTC
□ □□□ □□□ 2026-10-26 20:26:29 UTC