The Go Programming Language

The Go Blog

Go 1.6 is released

17 February 2016

Today we release Go version 1.6, the seventh major stable release of Go. You can grab it right now from the download page. Although the release of Go 1.5 six months ago contained dramatic implementation changes, this release is more incremental.

The most significant change is support for HTTP/2 in the net/http package. HTTP/2 is a new protocol, a follow-on to HTTP that has already seen widespread adoption by browser vendors and major websites. In Go 1.6, support for HTTP/2 is enabled by default for both servers and clients when using HTTPS, bringing the benefits of the new protocol to a wide range of Go projects, such as the popular Caddy web server.

The template packages have learned some new tricks, with support for trimming spaces around template actions to produce cleaner template output, and the introduction of the `{{block}}` action that can be used to create templates that build on other templates. A new template example program demonstrates these new features.

Go 1.5 introduced experimental support for a “vendor” directory that was enabled by an environment variable. In Go 1.6, the feature is now enabled by default. Source trees that contain a directory named “vendor” that is not used in accordance with the new feature will require changes to avoid broken builds (the simplest fix is to rename the directory).

The runtime has added lightweight, best-effort detection of concurrent misuse of maps. As always, if one goroutine is writing to a map, no other goroutine should be reading or writing the map concurrently. If the runtime detects this condition, it prints a diagnosis and crashes the program. The best way to find out more about the problem is to run it under the race detector, which will more reliably identify the race and give more detail.

The runtime has also changed how it prints program-ending panics. It now prints only the stack of the panicking goroutine, rather than all existing goroutines. This behavior can be configured using the GOTRACEBACK environment variable or by calling the debug.SetTraceback function.

Users of cgo should be aware of major changes to the rules for sharing pointers between Go and C code. The rules are designed to ensure that such C code can coexist with Go's garbage collector and are checked during program execution, so code may require changes to avoid crashes. See the release notes and cgo documentation for the details.

The compiler, linker, and go command have a new `-msan` flag analogous to `-race` and only available on linux/amd64, that enables interoperation with the Clang MemorySanitizer. This is useful for testing a program containing suspect C or C++ code. You might like to try it while testing your cgo code with the new pointer rules.

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

Performance of Go programs built with Go 1.6 remains similar to those built with Go 1.5. Garbage-collection pauses are even lower than with Go 1.5, but this is particularly noticeable for programs using large amounts of memory. With regard to the performance of the compiler tool chain, build times should be similar to those of Go 1.5.

The algorithm inside sort.Sort was improved to run about 10% faster, but the change may break programs that expect a specific ordering of equal but distinguishable elements. Such programs should refine their `Less` methods to indicate the desired ordering or use sort.Stable to preserve the input order for equal values.

And, of course, there are many more additions, improvements, and fixes. You can find them all in the comprehensive release notes.

To celebrate the release, Go User Groups around the world are holding release parties on the 17th of February. Online, the Go contributors are hosting a question and answer session on the golang subreddit for the next 24 hours. If you have questions about the project, the release, or just Go in general, then please join the discussion.

Thanks to everyone that contributed to the release. Happy hacking.

By Andrew Gerrand

Language and Locale Matching in Go

9 February 2016

Introduction

Consider an application, such as a web site, with support for multiple languages in its user interface. When a user arrives with a list of preferred languages, the application must decide which language it should use in its presentation to the user. This requires finding the best match between the languages the application supports and those the user prefers. This post explains why this is a difficult decision and how Go can help.

Language Tags

Language tags, also known as locale identifiers, are machine-readable identifiers for the language and/or dialect being used. The most common reference for them is the IETF BCP 47 standard, and that is the standard the Go libraries follow. Here are some examples of BCP 47 language tags and the language or dialect they represent.

Tag	Description
`en`	English
`en-US`	American English
`cmn`	Mandarin Chinese
`zh`	Chinese, typically Mandarin
`nl`	Dutch
`nl-BE`	Flemish
`es-419`	Latin American Spanish
`az, az-Latn`	both Azerbaijani written in Latin script

`az-Arab`	Azerbaijani written in Arabic

The general form of the language tag is a language code (“en”, “cmn”, “zh”, “nl”, “az” above) followed by an optional subtag for script (“-Arab”), region (“-US”, “-BE”, “-419”), variants (“-oxendict” for Oxford English Dictionary spelling), and extensions (“-u-co-phonebk” for phone-book sorting). The most common form is assumed if a subtag is omitted, for instance “az-Latn-AZ” for “az”.

The most common use of language tags is to select from a set of system-supported languages according to a list of the user's language preferences, for example deciding that a user who prefers Afrikaans would be best served (assuming Afrikaans is not available) by the system showing Dutch. Resolving such matches involves consulting data on mutual language comprehensibility.

The tag resulting from this match is subsequently used to obtain language-specific resources such as translations, sorting order, and casing algorithms. This involves a different kind of matching. For example, as there is no specific sorting order for Portuguese, a collate package may fall back to the sorting order for the default, or “root”, language.

The Messy Nature of Matching Languages

Handling language tags is tricky. This is partly because the boundaries of human languages are not well defined and partly because of the legacy of evolving language tag standards. In this section we will show some of the messy aspects of handling language tags.

Tags with different language codes can indicate the same language

For historical and political reasons, many language codes have changed over time, leaving languages with an older legacy code as well as a new one. But even two current codes may refer to the same language. For example, the official language code for Mandarin is “cmn”, but “zh” is by far the most commonly used designator for this language. The code “zh” is officially reserved for a so called macro language, identifying the group of Chinese languages. Tags for macro languages are often used interchangeably with the most-spoken language in the group.

Matching language code alone is not sufficient

Azerbaijani (“az”), for example, is written in different scripts depending on the country in which it is spoken: "az-Latn" for Latin (the default script), "az-Arab" for Arabic, and “az-Cyrl” for Cyrillic. If you replace "az-Arab" with just "az", the result will be in Latin script and may not be understandable to a user who only knows the Arabic form.

Also different regions may imply different scripts. For example: “zh-TW” and “zh-SG” respectively imply the use of Traditional and Simplified Han. As another example, “sr” (Serbian) defaults to Cyrillic script, but “sr-RU” (Serbian as written in Russia) implies the Latin script! A similar thing can be said for Kyrgyz and other languages.

If you ignore subtags, you might as well present Greek to the user.

The best match might be a language not listed by the user

The most common written form of Norwegian (“nb”) looks an awful lot like Danish. If Norwegian is not available, Danish may be a good second choice. Similarly, a user requesting Swiss German (“gsw”) will likely be happy to be presented German (“de”), though the converse is far from true. A user requesting Uygur may be happier to fall back to Chinese than to English. Other examples abound. If a user-requested language is not supported, falling back to English is often not the best thing to do.

The choice of language decides more than translation

Suppose a user asks for Danish, with German as a second choice. If an application chooses German, it must not only use German translations but also use German (not Danish) collation. Otherwise, for example, a list of animals might sort “Bär” before “Äffin”.

Selecting a supported language given the user’s preferred languages is like a handshaking algorithm: first you determine which protocol to communicate in (the language) and then you stick with this protocol for all communication for the duration of a session.

Using a “parent” of a language as fallback is non-trivial

Suppose your application supports Angolan Portuguese (“pt-AO”). Packages in golang.org/x/text, like collation and display, may not have specific support for this dialect. The correct course of action in such cases is to match the closest parent dialect. Languages are arranged in a hierarchy, with each specific language having a more general parent. For example, the parent of “en-GB-oxendict” is “en-GB”, whose parent is “en”, whose parent is the undefined language “und”, also known as the root language. In the case of collation, there is no specific collation order for Portugese, so the collate package will select the sorting order of the root language. The closest parent to Angolan Portuguese supported by the display package is European Portuguese (“pt-PT”) and not the more obvious “pt”, which implies Brazilian.

In general, parent relationships are non-trivial. To give a few more examples, the parent of “es-CL” is “es-419”, the parent of “zh-TW” is “zh-Hant”, and the parent of “zh-Hant” is “und”. If you compute the parent by simply removing subtags, you may select a “dialect” that is incomprehensible to the user.

Language Matching in Go

The Go package golang.org/x/text/language implements the BCP 47 standard for language tags and adds support for deciding which language to use based on data published in the Unicode Common Locale Data Repository (CLDR).

Here is a sample program, explained below, matching a user's language preferences against an application's supported languages:

```
package main

import (
    "fmt"

    "golang.org/x/text/language"
    "golang.org/x/text/language/display"
)

var userPrefs = []language.Tag{
    language.Make("gsw"), // Swiss German
    language.Make("fr"),  // French
}

var serverLangs = []language.Tag{
    language.AmericanEnglish, // en-US fallback
    language.German,          // de
}
```

```
var matcher = language.NewMatcher(serverLangs)

func main() {
    tag, index, confidence := matcher.Match(userPrefs...)

    fmt.Printf("best match: %s (%s) index=%d confidence=%v\n",
        display.English.Tags().Name(tag),
        display.Self.Name(tag),
        index, confidence)
    // best match: German (Deutsch) index=1 confidence=High
}
```

Creating Language Tags

The simplest way to create a language.Tag from a user-given language code string is with language.Make. It extracts meaningful information even from malformed input. For example, “en-USD” will result in “en” even though USD is not a valid subtag.

Make doesn’t return an error. It is common practice to use the default language if an error occurs anyway so this makes it more convenient. Use Parse to handle any error manually.

The HTTP Accept-Language header is often used to pass a user’s desired languages. The ParseAcceptLanguage function parses it into a slice of language tags, ordered by preference.

By default, the language package does not canonicalize tags. For example, it does not follow the BCP 47 recommendation of eliminating scripts if it is the common choice in the “overwhelming majority”. It similarly ignores CLDR recommendations: “cmn” is not replaced by “zh” and “zh-Hant-HK” is not simplified to “zh-HK”. Canonicalizing tags may throw away useful information about user intent. Canonicalization is handled in the Matcher instead. A full array of canonicalization options are available if the programmer still desires to do so.

Matching User-Preferred Languages to Supported Languages

A Matcher matches user-preferred languages to supported languages. Users are strongly advised to use it if they don’t want to deal with all the intricacies of matching languages.

The Match method may pass through user settings (from BCP 47 extensions) from the preferred tags to the selected supported tag. It is therefore important that the tag returned by Match is used to obtain language-specific resources. For example, “de-u-co-phonebk” requests phone-book ordering for German. The extension is ignored for matching, but is used by the collate package to select the respective sorting order variant.

A Matcher is initialized with the languages supported by an application, which are usually the languages for which there are translations. This set is typically fixed, allowing a matcher to be created at startup. Matcher is optimized to improve the performance of Match at the expense of initialization cost.

The language package provides a predefined set of the most commonly used language tags that can be used for defining the supported set. Users generally don’t have to worry about the exact tags to pick for supported languages. For example, AmericanEnglish (“en-US”) may be used interchangeably with the more common English (“en”), which defaults to American. It is all the same for the Matcher. An application may even add both, allowing for more specific American slang for “en-US”.

Matching Example

Consider the following Matcher and lists of supported languages:

```
var supported = []language.Tag{
    language.AmericanEnglish,    // en-US: first language is fallback
    language.German,             // de
    language.Dutch,              // nl
    language.Portuguese          // pt (defaults to Brazilian)
    language.EuropeanPortuguese, // pt-pT
    language.Romanian            // ro
    language.Serbian,            // sr (defaults to Cyrillic script)
    language.SerbianLatin,       // sr-Latn
    language.SimplifiedChinese,  // zh-Hans
    language.TraditionalChinese, // zh-Hant
}
var matcher = language.NewMatcher(supported)
```

Let's look at the matches against this list of supported languages for various user preferences.

For a user preference of "he" (Hebrew), the best match is "en-US" (American English). There is no good match, so the matcher uses the fallback language (the first in the supported list).

For a user preference of "hr" (Croatian), the best match is "sr-Latn" (Serbian with Latin script), because, once they are written in the same script, Serbian and Croatian are mutually intelligible.

For a user preference of "ru, mo" (Russian, then Moldavian), the best match is "ro" (Romanian), because Moldavian is now canonically classified as "ro-MD" (Romanian in Moldova).

For a user preference of "zh-TW" (Mandarin in Taiwan), the best match is "zh-Hant" (Mandarin written in Traditional Chinese), not "zh-Hans" (Mandarin written in Simplified Chinese).

For a user preference of "af, ar" (Afrikaans, then Arabic), the best match is "nl" (Dutch). Neither preference is supported directly, but Dutch is a significantly closer match to Afrikaans than the fallback language English is to either.

For a user preference of "pt-AO, id" (Angolan Portuguese, then Indonesian), the best match is "pt-PT" (European Portuguese), not "pt" (Brazilian Portuguese).

For a user preference of "gsw-u-co-phonebk" (Swiss German with phone-book collation order), the best match is "de-u-co-phonebk" (German with phone-book collation order). German is the best match for Swiss German in the server's language list, and the option for phone-book collation order has been carried over.

Confidence Scores

Go uses coarse-grained confidence scoring with rule-based elimination. A match is classified as Exact, High (not exact, but no known ambiguity), Low (probably the correct match, but maybe not), or No. In case of multiple matches, there is a set of tie-breaking rules that are executed in order. The first match is returned in the case of multiple equal matches. These confidence scores may be useful, for example, to reject relatively weak matches. They are also used to score, for example, the most likely region or script from a language tag.

Implementations in other languages often use more fine-grained, variable-scale scoring. We found that using coarse-grained scoring in the Go implementation ended up simpler to implement, more maintainable, and faster, meaning that we could handle more rules.

Displaying Supported Languages

The golang.org/x/text/language/display package allows naming language tags in many languages. It also contains a “Self” namer for displaying a tag in its own language.

For example:

```
var supported = []language.Tag{
    language.English,            // en
    language.French,             // fr
    language.Dutch,              // nl
    language.Make("nl-BE"),      // nl-BE
    language.SimplifiedChinese,  // zh-Hans
    language.TraditionalChinese, // zh-Hant
    language.Russian,            // ru
}

en := display.English.Tags()
for _, t := range supported {
    fmt.Printf("%-20s (%s)\n", en.Name(t), display.Self.Name(t))
}
```

prints

```
English              (English)
French               (français)
Dutch                (Nederlands)
Flemish              (Vlaams)
Simplified Chinese   (简体中文)
Traditional Chinese  (繁體中文)
Russian              (русский)
```

In the second column, note the differences in capitalization, reflecting the rules of the respective language.

Conclusion

At first glance, language tags look like nicely structured data, but because they describe human languages, the structure of relationships between language tags is actually quite complex. It is often tempting, especially for English-speaking programmers, to write ad-hoc language matching using nothing other than string manipulation of the language tags. As described above, this can produce awful results.

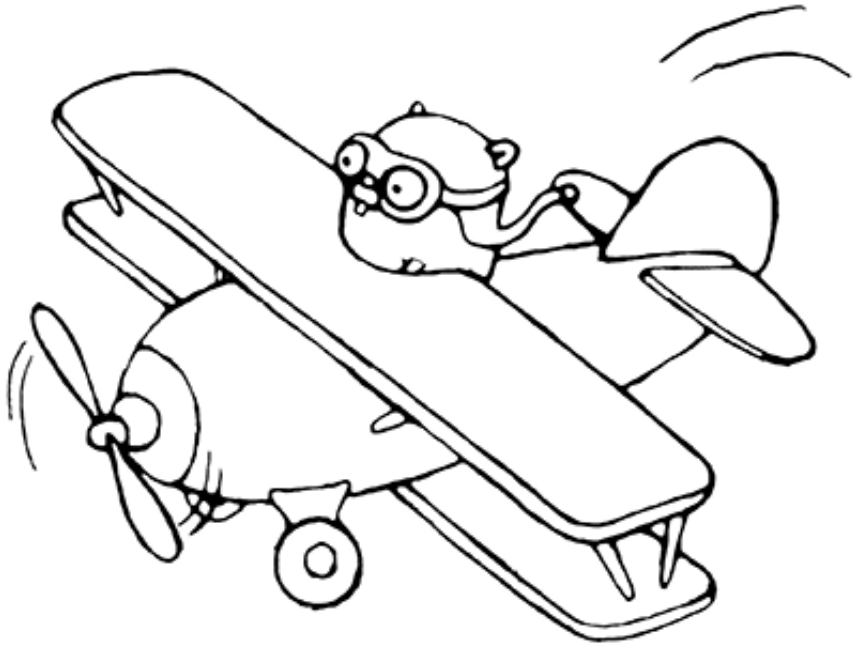
Go's golang.org/x/text/language package solves this complex problem while still presenting a simple, easy-to-use API. Enjoy.

By Marcel van Lohuizen

Six years of Go

10 November 2015

Six years ago today the Go language was released as an open source project. Since then, more than 780 contributors have made over 30,000 commits to the project's 22 repositories. The ecosystem continues to grow, with GitHub reporting more than 90,000 Go repositories. And, offline, we see new Go events and user groups pop up around the world with regularity.



In August we released Go 1.5, the most significant release since Go 1. It features a completely redesigned garbage collector that makes the language more suitable for latency-sensitive applications; it marks the transition from a C-based compiler tool chain to one written entirely in Go; and it includes ports to new architectures, with better support for ARM processors (the chips that power most smartphones). These improvements make Go better suited to a broader range of tasks, a trend that we hope will continue over the coming years.

Improvements to tools continue to boost developer productivity. We introduced the execution tracer and the "go doc" command, as well as more enhancements to our various static analysis tools. We are also working on an official Go plugin for Sublime Text, with better support for other editors in the pipeline.

Early next year we will release more improvements in Go 1.6, including HTTP/2 support for net/http servers and clients, an official package vendoring mechanism, support for blocks in text and HTML templates, a memory sanitizer that checks both Go and C/C++ code, and the usual assortment of other improvements and fixes.

This is the sixth time we have had the pleasure of writing a birthday blog post for Go, and we would not be doing so if not for the wonderful and passionate people in our community. The Go team would like to thank everyone who has contributed code, written an open source library, authored a blog post, helped a new gopher, or just given Go a try. Without you, Go would not be as complete, useful, or successful as it is today. Thank you, and celebrate!

By Andrew Gerrand

GolangUK 2015

9 October 2015

On August 21st the Go community gathered in London for the first edition of Golang UK. The conference featured two parallel tracks and nearly 400 gophers attended.

The conference started with the opening keynote by David Calavera called Crossing the Language Chasm (video) and continued with two concurrently executed tracks.

Main track:

- Stupid Gopher Tricks, by Andrew Gerrand (video)
- Complex Concurrency Patterns in Go, by Evan Huus (video)
- Code Analysis [no reading required], by Francesc Campoy (video)
- Go kit: a toolkit for microservices Peter Bourgon (video)
- Dependency Management Conundrum, by William Kennedy (video)

Side track:

- Building APIs, by Mat Ryer (video)
- Building a Bank with Go, by Matt Heath (video)
- CockroachDB: Make Data Easy, by Ben Darnell (video)
- Understanding memory allocation in Go, by Dean Elbaz (video)
- Whispered Secrets, by Eleanor McHugh (video)

Finally Damian Gryski took the stage for the closing keynote (video), giving an overview of how the Go community has evolved over time and hinting to what the future might look like.

On the day before the conference William Kennedy gave a full day Go workshop.

It was a great conference, so congratulations to the organizers and see you next year in London!

By Francesc Campoy

Go GC: Prioritizing low latency and simplicity

31 August 2015

The Setup

Go is building a garbage collector (GC) not only for 2015 but for 2025 and beyond: A GC that supports today's software development and scales along with new software and hardware throughout the next decade. Such a future has no place for stop-the-world GC pauses, which have been an impediment to broader uses of safe and secure languages such as Go.

Go 1.5, the first glimpse of this future, achieves GC latencies well below the 10 millisecond goal we set a year ago. We presented some impressive numbers in a talk at Gophercon. The latency improvements have generated a lot of attention; Robin Verlangen's blog post *Billions of requests per day meet Go 1.5* validates our direction with end to end results. We also particularly enjoyed Alan Shreve's production server graphs and his "Holy 85% reduction" comment.

Today 16 gigabytes of RAM costs $100 and CPUs come with many cores, each with multiple hardware threads. In a decade this hardware will seem quaint but the software being built in Go today will need to scale to meet expanding needs and the next big thing. Given that hardware will provide the power to increase throughput, Go's garbage collector is being designed to favor low latency and tuning via only a single knob. Go 1.5 is the first big step down this path and these first steps will forever influence Go and the applications it best supports. This blog post gives a high-level overview of what we have done for the Go 1.5 collector.

The Embellishment

To create a garbage collector for the next decade, we turned to an algorithm from decades ago. Go's new garbage collector is a *concurrent*, *tri-color*, *mark-sweep* collector, an idea first proposed by Dijkstra in 1978. This is a deliberate divergence from most "enterprise" grade garbage collectors of today, and one that we believe is well suited to the properties of modern hardware and the latency requirements of modern software.

In a tri-color collector, every object is either white, grey, or black and we view the heap as a graph of connected objects. At the start of a GC cycle all objects are white. The GC visits all *roots*, which are objects directly accessible by the application such as globals and things on the stack, and colors these grey. The GC then chooses a grey object, blackens it, and then scans it for pointers to other objects. When this scan finds a pointer to a white object, it turns that object grey. This process repeats until there are no more grey objects. At this point, white objects are known to be unreachable and can be reused.

This all happens concurrently with the application, known as the *mutator*, changing pointers while the collector is running. Hence, the mutator must maintain the invariant that no black object points to a white object, lest the garbage collector lose track of an object installed in a part of the heap it has already visited. Maintaining this invariant is the job of the *write barrier*, which is a small function run by the mutator whenever a pointer in the heap is modified. Go's write barrier colors the now-reachable object grey if it is currently white, ensuring that the garbage collector will eventually scan it for pointers.

Deciding when the job of finding all grey objects is done is subtle and can be expensive and complicated if we want to avoid blocking the mutators. To keep things simple Go 1.5 does as much work as it can concurrently and then briefly stops the world to inspect all potential sources of grey objects. Finding the sweet spot between the time needed for this final stop-the-world and the total amount of work that this GC does is a major deliverable for Go 1.6.

Of course the devil is in the details. When do we start a GC cycle? What metrics do we use to make that decision? How should the GC interact with the Go scheduler? How do we pause a mutator thread long enough to scan its stack? How do we represent white, grey, and black so we can efficiently find and scan grey objects? How do we know where the roots are? How do we know where in an object pointers are located? How do we minimize memory fragmentation? How do we deal with cache performance issues? How big should the heap be? And on and on, some related to allocation, some to finding reachable objects, some related to scheduling, but many related to performance. Low-level discussions of each of these areas are beyond the scope of this blog post.

At a higher level, one approach to solving performance problems is to add GC knobs, one for each performance issue. The programmer can then turn the knobs in search of appropriate settings for their application. The downside is that after a decade with one or two new knobs each year you end up with the GC Knobs Turner Employment Act. Go is not going down that path. Instead we provide a single knob, called GOGC. This value controls the total size of the heap relative to the size of reachable objects. The default value of 100 means that total heap size is now 100% bigger than (i.e., twice) the size of the reachable objects after the last collection. 200 means total heap size is 200% bigger than (i.e., three times) the size of the reachable objects. If you want to lower the total time spent in GC, increase GOGC. If you want to trade more GC time for less memory, lower GOGC.

More importantly as RAM doubles with the next generation of hardware, simply doubling GOGC will halve the number of GC cycles. On the other hand since GOGC is based on reachable object size, doubling the load by doubling the reachable objects requires no retuning. The application just scales. Furthermore, unencumbered by ongoing support for dozens of knobs, the runtime team can focus on improving the runtime based on feedback from real customer applications.

The Punchline

Go 1.5's GC ushers in a future where stop-the-world pauses are no longer a barrier to moving to a safe and secure language. It is a future where applications scale effortlessly along with hardware and as hardware becomes more powerful the GC will not be an impediment to better, more scalable software. It's a good place to be for the next decade and beyond. For more details about the 1.5 GC and how we eliminated latency issues see the Go GC: Latency Problem Solved presentation or the slides.

By Richard Hudson

See the index for more articles.