A Glimpse to the "Go 2.0"

Changkun Ou*

December 28, 2022

Abstract

From 2021 to 2022, Go introduced a large number of new features to its entire ecosystem. There are many exciting features, such as generics in the language, fuzzing and profile-guided optimization in the compiler toolchain, multiple error wrapping in the standard library, etc. All these changes make Go 1.20 a representative release since Go 1, which we could say that Go 1.20 is the "Go 2.0". In this article, we will walkthrough a few highlighted changes to catch our breath to the quick-moving Go community and reflect how these new features improves our future engineering experiences.

Contents

*Email: research@changkun.de

Language Changes

From Go 1.18 to Go 1.20, the Go team introduced a large number of new features to the language. These language features are mainly focused on generics and the revision of its memory model. There are also minor changes such as officially provided string/slice conversion functions in the `unsafe` package, and other minor language spec bugs.

Generics and the `comparable` constraint

There is a long strory behind the introduction of generics in Go. The Go team has been working on the generics design for more than 10 years. The first attempt was in 2011, and the second attempt was in 2017. The third attempt was in 2020, and the fourth attempt was in 2021. The Go team finally decided to introduce generics in Go 1.18, which is a major milestone in the history of Go.

However, just before Go 1.18 went out, the Go team realized that the current generics design, that included a new type constraint `comparable`, was not sound enough in its semantics, and it can introduce senseless behavior. Consider the following example:

```
package main

import (
  "fmt"
  "go/types"
)

var (
  anyType = types.Universe.Lookup("any").Type()
  anyInterface = types.Universe.Lookup("any").Type().
    Underlying().(*types.Interface)
  comparableType = types.Universe.Lookup("comparable").Type()
  comparableInterface = types.Universe.Lookup("comparable").Type().
    Underlying().(*types.Interface)
)

func main() {
  fmt.Println(types.Comparable(comparableType))                    // true
  fmt.Println(types.Comparable(anyType))                           // true
  fmt.Println(types.AssignableTo(comparableType, anyType))         // true
  fmt.Println(types.AssignableTo(anyType, comparableType))         // false
```

```
}
```

In the above example, when checking the comparability of `comparable` and `any`, the results are both true. However, when checking the assignability of `comparable` to `any` and vice versa, the results are different.

This means a comparable type is possible to assign to a value that is typed using `any`. However, a value that is typed using `any` is not possible to assign back to a comparable type. Therefore, when we use `comparable` as a type constraint in Go 1.18, a value typed using `any` will not be able to be accepted by the `comparable` constraint.

This behavior is very subtle because a value typed using `any` is comparable by spec definition, and it is somewhat non-intuitive if we can't use any to satisfy the `comparable` constraint.

With the new `comparable` semantics introduced in Go 1.20, the understanding of type constraints using interface has been separated into a concept of "implements" and a concept of "satisfies". In Go, an interface can be either considered as a type or a type constraint. When we word "implements", we mainly focus on whether an interface implements another interface or not, this is true if one interface represents a subset of another interface (interface as type constraint vs. as type constraint). When we speak "satisfies", we mainly focus on whether a type satisfies a type constraint or not, meaning a type is one of the included elements of a type constraint (interface as type vs. as type constraint).

To be clear, the following example shows the new semantics:

```
fmt.Println(types.Satisfies(comparableType, anyInterface))       // true
fmt.Println(types.Satisfies(anyInterface, comparableInterface))  // true
fmt.Println(types.Implements(comparableType, anyInterface))      // true
fmt.Println(types.Implements(anyInterface, comparableInterface)) // false
```

Translating to natural language, the following statements are true: - `comparable` satisfies `any` and `any` satisfies `comparable`. However, - `comparable` implements `any`, whereas `any` does not implement `comparable`.

To shorten the story, with Go 1.20, the following code becomes valid and permit us write generic code with more possibilities:

```
package main

func f[T comparable]() { _ = map[T]T{} }

func main() {
  // Prior Go 1.20: any does not implement comparable
  // Since GO 1.20: OK
  _ = f[any]
}
```

For more detailed community discussions, see [1, 2, 3, 4, 5, 7, 6, 8, 9, 10, 11]. In short, after a year of discussion, the [11] takes us back to the beginning of [1] but with more thought through regarding the language spec now.

Memory Model

Unsafe String/Slice Conversions

In Go, a string is immutable, and a slice is mutable. However, in some cases, we need to convert a string to a slice and vice versa. Previously, we have to understand the underlying implementation of a slice struct, and colaborate with `reflect.SliceHeader` and `reflect.StringHeader` that may be easily misused. In Go 1.20, combining with the `unsafe.Slice` that was introduced in Go 1.17, the Go team introduces three more functions in the `unsafe` package to allow us convert between string and byte slice easily:

```
package unsafe

func Slice(ptr *ArbitraryType, len IntegerType) []ArbitraryType // Go 1.17
func SliceData(slice []ArbitraryType) *ArbitraryType            // Go 1.20
func String(ptr *byte, len IntegerType) string                  // Go 1.20
func StringData(str string) *byte                               // Go 1.20
```

With these functions, we can easily benchmark the performance of the two implementations, where one is using the type cast and the other is using newly introduced `unsafe` functions:

```
package main

import (
  "testing"
  "unsafe"
)

func BenchmarkSliceToString(b *testing.B) {
  x := []byte("this is a string")

  b.Run("type-cast", func(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
      s = string(x)
    }
    _ = s
  })

  b.Run("unsafe-conv", func(b *testing.B) {
    var s string
    for i := 0; i < b.N; i++ {
```

```
      s = bytes2string(x)
    }
    _ = s
  })
}

func BenchmarkStringToSlice(b *testing.B) {
  x := "this is a string"

  b.Run("type-cast", func(b *testing.B) {
    var s []byte
    for i := 0; i < b.N; i++ {
      s = []byte(x)
    }
    _ = s
  })

  b.Run("unsafe-conv", func(b *testing.B) {
    var s []byte
    for i := 0; i < b.N; i++ {
      s = string2bytes(x)
    }
    _ = s
  })
}

func string2bytes(s string) []byte {
  return unsafe.Slice(unsafe.StringData(x), len(x))
}
func bytes2string(b []byte) string {
  return unsafe.String(unsafe.SliceData(x), len(x))
}
```

The above code shows the performance difference between the type-cast and the unsafe conversion:

```
name                          time/op
SliceToString/type-cast-8     17.9ns ± 2%
SliceToString/unsafe-conv-8   0.40ns ± 0%
StringToSlice/type-cast-8     19.4ns ± 1%
StringToSlice/unsafe-conv-8   0.40ns ± 0%
```

For more detailed discussions, see [12, 13].

Compiler Toolchain

Standard Doc Comments

Fuzzing

Profile-Guided Optimization

Runtime Improvements

Heap Limit

GC Pacer

Package `arena`

Write Barrier Batching

Bitmap Heap

Metrics

Standard Library

Context

Multiple Error Wrapping

Outlook

`clear` builtin

Surprisingly, there is no way to clear a map in Go. We can write the following code to clear a map:

```
for k := range m {
    delete(m, k)
}
```

but that it only works if `m` does not contain any key values that contain `NaN`s. For instance,

```
package main

import "math"

func main() {
  m := map[any]any{}
  println(len(m)) // 0
  m[math.NaN()] = 0
  println(len(m)) // 1
  for x := range m {
    delete(m, x)
  }
```

```
  println(len(m)) // 1
}
```

We might wonder why `delete(m, k)` cannot delete a key that is `NaN`. The reason is that `NaN` is not comparable, and the Go spec requires that the key type of a map must be comparable. We might also wonder why we allow `NaN` as a key in a map. The reason is a bit complicated, and we can find the detailed discussion in [14].

With all kinds of surprising reasons, for the first time, since Go 1, the Go team is considering to add a `clear` builtin to clear a map:

```
package main

import "math"

func main() {
  m := map[any]any{}
  println(len(m)) // 0
  m[math.NaN()] = 0
  println(len(m)) // 1
  clear(m)
  println(len(m)) // 0
}
```

See discussions in [15, 16, 17, 18].

`slog` and Structured Logging

Standard iterator

Summary

In this article, we have quickly walked through a few highlighted changes in the Go 2.0 release. We hope that this article can help you to catch up with the quick-moving Go community. We are excited and looking forward to the next release of Go 1.20, which will be released in Feb 2023.

References

[1] Kei Kamikawa. proposal: spec: add comparable w/o interfaces. Nov 15, 2021. https://go.dev/issue/49587

[2] Bob Glickstein. spec: document/explain which interfaces implement comparable. Jan 16, 2022. https://go.dev/issue/50646

[3] James Billingham. spec: any no longer implements comparable. Feb 18, 2022. https://go.dev/issue/51257

[4] Ian Lance Taylor. proposal: spec: permit values to have type "comparable". Feb 24, 2022. https://go.dev/issue/51338

[5] Awad Diar. proposal: spec: add new constraint kind satisfied by types that support == (including interface types). Apr 24, 2022. https://go.dev/issue/52531

[6] Ian Lance Taylor. proposal: spec: permit non-interface types that support == to satisfy the comparable constraint. Apr 21, 2022. https://go.dev/issue/52474

[7] Robert Griesemer. proposal: type parameters are comparable unless they exclude comparable types. Apr 29, 2022. https://go.dev/issue/52614

[8] Branden J Brown. proposal: spec: allow interface types to instantiate comparable type parameters. Apr 23, 2022. https://go.dev/issue/52509

[9] Bryan C. Mills. proposal: spec: allow values of type comparable. Apr 29, 2022. https://go.dev/issue/52624

[10] Awad Diar. proposal: spec: comparable as a case of a generalized definition of basic interfaces. Jul 7, 2022. https://go.dev/issue/53734

[11] Robert Griesemer. spec: allow basic interface types to instantiate comparable type parameters. Nov 3, 2022. https://go.dev/issue/56548

[12] Matthew Dempsky. unsafe: add Slice(ptr *T, len anyIntegerType) []T. Mar 2, 2017. https://go.dev/issue/19367

[13] Travis Bischel. unsafe: add StringData, String, SliceData. May 19, 2022. https://go.dev/issue/53003

[14] Russ Cox. Random Hash Functions. April 1, 2012. https://research.swtch.com/randhash

[15] Russ Cox. spec: add clear(x) builtin, to clear map, zero content of slice. Oct 20, 2022. https://go.dev/issue/56351

[16] Axel Wagner. reflect: add Value.Clear. Sep 11, 2022. https://go.dev/issue/55002

[17] Yi Duan. proposal: runtime: add way to clear and reuse a map's working storage. Apr 1, 2021. https://go.dev/issue/45328

[18] Bryan C. Mills. proposal: spec: disallow NaN keys in maps. Jun 13, 2017. https://go.dev/issue/20660