Generics and when to use them in Go

Paul Bivrell
1 November 2021

Primer

Go 1.18 slated to release in February. It will be the biggest syntactic change to the language since Go 1.

After years of work from the Go team and a little more than a year proposing and debating with the larger community, an implementation of generic programming (parametric polymorphism) has been mostly fleshed out. In just two weeks the 1.18 code freeze begins which means no new features will be added for generics initial release.

Today we are going explore all the syntax changes, some practical examples and briefly discuss my opinions as when generics might be useful in Go.

Overview of the Generic syntax

Generic Functions

Generic Function syntax

We use square brackets to write functions on generic data.

```
func Print[T any](data T) {
    fmt.Println(data)
}
```

We can call this function by passing the type of our data as a type argument

```
    Print[int](10)
    Print[string]("Apple")
    Print[float64](999.99)
```

Using type inference we can omit the type argument

```
    Print(1234)
    Print("pear")
    Print(23.45)
    Print([]int{1,2,3})
```

Run

More useful generic function

We can use generics to write functions that return us values based on the type they are provided

```
func GetFirst[T any](s []T) T {
    var zero T
    if len(s) < 1 {
        return zero
    }
    return s[0]
}
```

Example usage

```
    letter := GetFirst([]string{"a","b","c"})
    fmt.Println(letter)

    boolean := GetFirst([]bool{true, false, true, true})
    fmt.Println(boolean)

    integer := GetFirst([]int{})
    fmt.Println(integer)
```

Run

Generic Search

```
func Search[T comparable](slice []T, value T) (result T, ok bool) {

    for _, v := range slice {
        if value == v {
            return v, true
        }
    }

    return result, ok
}
```

Not all types in go can be compared with == so we must use the *comparable* type constraint

```
package main

import "fmt"

func main() {
    a, b := make([]string, 0), make([]string, 0)

    fmt.Println(a == b)
}
```

Run

Using generic search

```go
    value, ok := Search([]int{1,2,3,4,5}, 6)
    fmt.Println(value, ok)

    letter, ok := Search([]string{"a", "b", "c", "d"}, "b")
    fmt.Println(letter, ok)

    // Does this compile?
    // ints, ok := Search([][]int{ []int{1}, []int{2,3}, []int{4,5,6}}, []int{1})
    // fmt.Println(ints, ok)

    // Does this compile?
    // value, ok = Search([]string{"a", "b", "c", "d"}, "b")
    // fmt.Println(value, ok)

    // Does this compile?
    // anything, ok := Search[interface{}]([]interface{}{ "a", 1, 1.23, false}, "a")
    // fmt.Println(anything, ok)
```

Run

Constraints

Predeclared constraints

There are two predeclared type constraints any and `comparable`

- any is an alias for interface`{}`. Which is just useful short hand for all types
- comparable is all types that you can use == and != on

Constraints the concept

A type constraint defines the set of methods and operators a type must have to be usable generically

Because of their functional similarity to interfaces we can reuse that syntax.

```
type Stringer interface {
    String() string
}

func AddStrings[T Stringer](s1, s2 T) string {
    return s1.String() + s2.String()
}
```

Type Sets in Constraints

We can also now add type sets to interfaces to define the operators that are permitted.

```
type OnlySignedInt interface {
    int | int8 | int16 | int32 | int64
}

func Celsius[T OnlySignedInt](F T) T {
    return (9/5) * (F - 32)
}
```

This creates a constraint that can use all of the following operators (*, /, %, <<, >>, &, &^, +, -, |, ^, ==, !=, <, <=, >, >=)

```
func main() {
    fmt.Println(Celsius(32))
    fmt.Println(Celsius(int8(32)))
}
```

Run

Type approximation elements

The previous constraint only matches things that are exactly int, int8, int16, int32, and int64. For example OtherInt would not be acceptable as an OnlySignedInt.

```
type OtherInt int
```

To include a type who's underlying type is another type we use the approximation element syntax

```
type SignedInt interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}
```

Method and type sets in constraint

Unsurprisingly we can use combinations of methods and type sets to further specify the constraint

```
type PostiveSignedInt interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
    Positive() bool
}

type BetterInt int

func (b BetterInt) Positive() bool {
    return b > 0
}
```

Generic Types

Generic type

Performing functions on generic data in itself is useful. However go 1.18 offers us even more power with generic types.

A generic type can contain any type that matches it's constraint.

```go
// Stack represents a stack datastructure
type Stack[T any] []T

// Push appends to the end of the stack
func (s *Stack[T]) Push (elem T) {
    *s = append(*s, elem)
}

// Pop returns the last pushed item
func (s *Stack[T]) Pop() (elem T) {

    if len(*s) < 1 {
        return elem
    }

    return (*s)[len(*s)-1]
}
```

Using generic types

```
    s := Stack[int]{1,2,3}
    s.Push(4)
    fmt.Println(s.Pop())

    var s1 Stack[string]
    fmt.Println(s1.Pop())
    s1.Push("a")
    s1.Push("b")
    s1.Push("c")
    fmt.Println(s1.Pop())

    type X struct {
        A string
    }

    s2 := Stack[X]{}
    s2.Push(X{ A: "A"})
    s2.Push(X{ A: "B"})
    fmt.Println(s2.Pop())
```

Run

Important note a generic type can contain only the type specified when it's instantiated

Another container example

Type parameters can be values in struct

```
type Queue[T any] struct {
    items chan []T // non-empty slices only
    empty chan bool   // holds true if the queue is empty
}

func NewQueue[T any]() *Queue[T] {
    items := make(chan []T , 1)
    empty := make(chan bool, 1)
    empty <- true
    return &Queue[T]{items, empty}
}
```

Queue methods

```
func (q *Queue[T]) Get() T {
    items := <-q.items
    item := items[0]
    items = items[1:]
    if len(items) == 0 {
        q.empty <- true
    } else {
        q.items <- items
    }
    return item
}

func (q *Queue[T]) Put(item T) {
    var items []T
    select {
    case items = <-q.items:
    case <-q.empty:
    }
    items = append(items, item)
    q.items <- items
}
```

Queue usage

```
func main() {
    q1 := NewQueue[string]()
    q1.Put("apple")
    q1.Put("banana")
    q1.Put("kiwi")

    wg := &sync.WaitGroup{}
    wg.Add(3)
    for i := 0; i < 3; i++ {
        go func() {
            fmt.Println(q1.Get())
            wg.Done()
        }()
    }
    wg.Wait()
}
```

Run

Non container example

We can have multiple same or different type parameters

```
func Map[T1, T2 any](s []T1, f func(T1) T2) []T2 {
    r := make([]T2, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}
```

Map usage

```
func toInt(s string) int {
    i, _ := strconv.Atoi(s)
    return i
}

type Named struct {
    Name string
}

func main() {

    s := []string{"1","2","29480", "-2"}
    i := Map(s, toInt)
    fmt.Println(i)

    named := Map(s, func( x string) Named {
        return Named{
            Name: x,
        }
    })
    fmt.Println(named)
}
```

Run

Guidelines

Write non generic code

If you are tempted to write generic code ask yourself the following 3 questions

1. Are you copy and pasting existing code?
2. Is there a set (n > 1) of types that your code can operate on?
3. Will the user of my code be willing to specify a type parameter?

Final thoughts

So far I think generics are most valuable for collections and operations on collections. As such I think it is unlikely that you should often write generic code (a standard library of these will come eventually). Though you may frequently use existing generic code to reduce boiler plate.

Generics will be fantastic for the subset of problems they are meant to solve.

Thank you

Paul Bivrell
1 November 2021
Slack: @pbj (#ZgotmplZ)