



Golang Programming

Code Generation and Generics in Go

Where to Find The Code and Materials?

<https://github.com/iproduct/coursego>

Agenda for This Session

- The problem
- Code generation
- Generics and their purpose
- Type-specific wrappers with type-agnostic implementations
- Alternatives – using interfaces, assertions, reflection, code generation
- Generating code with go generate
- Generics in Go – Draft Specification & Demo in go2go playground
- Q&A. Projects mentoring.

The Problem

- Many algorithms and data structures are applicable to many data types. Examples: `Stringer`, `Sorter`, `priority queue`, `sorted tree`, `concurrent hash map`
- Container data types above could hold elements of `string`, `int64`, `[]int`, `map[string]string`, or a `struct` type.
- We could implement a sorting algorithm using sorted tree for example. The requirement is that the elements should be comparable to each other.
- But what if we want to reuse the same `sorted tree implementation` with `different element types`?
- If there are `N methods` and `M element types` we should write `N x M function implementations ...`

Solutions from Other Languages – e.g. Java

```
public class SortedTree<E extends Comparable<E>> {  
    public void insert(E element) {  
        //...  
    }  
}
```

```
SortedTree<String> stringSortedTree = new SortedTree<>();  
sortedTreeOfStrings.insert("xyz");  
sortedTreeOfStrings.insert("abc");  
sortedTreeOfStrings.insert("klm");
```

The Ways to Implement Generic Code in Go

- “If C++ and Java are about type hierarchies and the taxonomy of types, Go is about composition.” (Rob Pike)
- Use multiple functions (copy & paste 😊)
- Interfaces
- Type assertions
- Reflection
- Code generation

Interfaces (1)

- Example: `sort.Interface`

```
type Interface interface {  
    Len() int  
    Less(i, j int) bool  
    Swap(i, j int)  
}
```

- Example: `io.Reader` interface:
 - many functions take an `io.Reader` as input
 - many data types, including files, network connections, ciphers, implement it
- Or even: `interface {}`. E.g.:

```
func Slice(slice interface{}, less func(i, j int) bool)
```

Interfaces (2)

```
import "sort"
```

```
type Person struct {  
    Name string  
    Age  int  
}
```

// ByAge implements sort.Interface for []Person based on the Age field.

```
type ByAge []Person
```

```
func (a ByAge) Len() int           { return len(a) }  
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }  
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }
```

```
func SortPeople(people []Person) {  
    sort.Sort(ByAge(people))  
}
```


Interfaces (3)

```
package main
import ("fmt"; "sort")
func main() {
    people := []struct {
        Name string
        Age  int
    }{
        {"Gopher", 7},
        {"Alice", 55},
        {"Vera", 24},
        {"Bob", 75},
    }
    sort.Slice(people, func(i, j int) bool { return people[i].Name < people[j].Name })
    fmt.Println("By name:", people)

    sort.Slice(people, func(i, j int) bool { return people[i].Age < people[j].Age })
    fmt.Println("By age:", people)
}
```

Type Assertions (1)

```
type MyContainer []interface{}
```

```
func (c *MyContainer) Put(elem interface{}) {  
    *c = append(*c, elem)  
}
```

```
func (c *MyContainer) Get() interface{} {  
    elem := (*c)[0]  
    *c = (*c)[1:]  
    return elem  
}
```

Type Assertions (2)

```
func main() {  
    myIntContainer := &MyContainer{}  
    myIntContainer.Put(12)  
    myIntContainer.Put(70)  
    elem, ok := myIntContainer.Get().(int)  
    if !ok {  
        fmt.Println("Error getting int from myIntContainer")  
    }  
    fmt.Printf("Got: %d (%T)\n", elem, elem)  
}
```

Reflection (1)

```
package main

import (
    "fmt"
    "reflect"
)

type MyStack struct {
    t reflect.Type
    value reflect.Value
}

func New(tp reflect.Type) *MyStack {
    return &MyStack{
        t: tp,
        value: reflect.MakeSlice(reflect.SliceOf(tp), 0, 100),
    }
}
```

Reflection (2)

```
func (m *MyStack) Push(v interface{}) {
    if reflect.TypeOf(v).Type() != m.value.Type().Elem() {
        panic(fmt.Sprintf("Error putting %T into %s", v, m.value.Type().Elem()))
    }
    m.value = reflect.Append(m.value, reflect.ValueOf(v))
}

func (m *MyStack) Pop() interface{} {
    v := m.value.Index(0)
    m.value = m.value.Slice(1, m.value.Len())
    return v.Interface()
}

func main() {
    val := 2.88
    stack := New(reflect.TypeOf(val))
    stack.Push(val)
    fmt.Println(stack.value.Index(0))
    result := stack.Pop()
    fmt.Printf("Result: %[1]f (%[1]T)\n", result)
}
```

Problem: Reflection Performance – Getting Map Keys

https://github.com/iproduct/coursego/blob/master/13-generation/reflection-performance/invoc_test.go

```
func keysPrealloc(m map[uint64]string) {  
    k := make([]uint64, len(m))  
    var i uint64  
    for key := range m {  
        k[i] = key  
        i++  
    }  
}
```

```
func keysAppend(m map[uint64]string) {  
    keys := make([]uint64, 0)  
    for key := range m {  
        keys = append(keys, key)  
    }  
}
```

```
func keysReflect(m map[uint64]string) {  
    reflect.ValueOf(m).MapKeys()  
}
```

populating
filling 1000000000 slots
done in 25.6922994s
running prealloc
took: **1.5210013s**
running append
took: **2.908973s**
running reflect
took: **6.4179969s**

Problem: Reflection Performance

<https://github.com/iproduct/coursego/blob/master/13-generation/reflection-performance/reflection.performance.go>

```
func BenchmarkReflectMethodCall(b *testing.B) {  
    i := new(myint)  
    incnReflectCall(i.inc, b.N)  
}
```

```
func incnReflectCall(v interface{}, n int) {  
    for k := 0; k < n; k++ {  
        reflect.ValueOf(v).Call(nil)  
    }  
}
```

```
func incnReflectOnceCall(v interface{}, n int) {  
    fn := reflect.ValueOf(v)  
    for k := 0; k < n; k++ {  
        fn.Call(nil)  
    }  
}
```



goos: windows

goarch: amd64

pkg: github.com/iproduct/coursego/generation/reflection-performance

BenchmarkReflectMethodCall

BenchmarkReflectMethodCall-12	11213013	95.7 ns/op
--------------------------------------	-----------------	-------------------

BenchmarkReflectOnceMethodCall

BenchmarkReflectOnceMethodCall-12	12766160	94.8 ns/op
--	-----------------	-------------------

BenchmarkStructMethodCall

BenchmarkStructMethodCall-12	944913882	1.26 ns/op
-------------------------------------	------------------	-------------------

BenchmarkInterfaceMethodCall

BenchmarkInterfaceMethodCall-12	666554833	1.80 ns/op
--	------------------	-------------------

BenchmarkTypeSwitchMethodCall

BenchmarkTypeSwitchMethodCall-12	1000000000	0.957 ns/op
---	-------------------	--------------------

BenchmarkTypeAssertionMethodCall

BenchmarkTypeAssertionMethodCall-12	1000000000	1.18 ns/op
--	-------------------	-------------------

PASS

Code Generation - Rob Pike

[\[https://blog.golang.org/generate\]](https://blog.golang.org/generate)

- A property of universal computation—Turing completeness—is that a computer program can write a computer program. This is a powerful idea that is not appreciated as often as it might be, even though it happens frequently.
- It's a big part of the definition of a compiler, for instance. It's also how the `go test` command works: it scans the packages to be tested, writes out a Go program containing a test harness customized for the package, and then compiles and runs it.
- Modern computers are so fast this expensive-sounding sequence can complete in a fraction of a second.
- Examples: [Yacc](#), [Protocol Buffers](#)

Code Generation Example - Rob Pike

[\[https://blog.golang.org/generate\]](https://blog.golang.org/generate)

- Get Go Yacc tool:

```
go get golang.org/x/tools/cmd/goyacc
```

- You can run it directly:

```
goyacc -o gopher.go -p parser gopher.y
```

- Go generate - add this comment anywhere in the non-generated go file:

```
//go:generate goyacc -o gopher.go -p parser gopher.y
```

```
$ cd $GOPATH/myrepo/gopher
```

```
$ go generate
```

```
$ go build
```

```
$ go test
```

Code Generation - Rob Pike

[\[https://blog.golang.org/generate\]](https://blog.golang.org/generate)

- A property of universal computation—Turing completeness—is that a computer program can write a computer program. This is a powerful idea that is not appreciated as often as it might be, even though it happens frequently.
- It's a big part of the definition of a compiler, for instance. It's also how the `go test` command works: it scans the packages to be tested, writes out a Go program containing a test harness customized for the package, and then compiles and runs it.
- Modern computers are so fast this expensive-sounding sequence can complete in a fraction of a second.
- Examples: [Yacc](#), [Protocol Buffers](#)

Code Generation Examples

<https://github.com/iproduct/coursego/tree/master/13-generation>

<https://github.com/iproduct/coursego/tree/master/generationlab>



Generics in Go 1.18

experiments with generics

Type Parameters - Draft Design - I

Ian Lance Taylor, Robert Griesemer, January 11, 2021:

<https://go.gogglesource.com/proposal/+refs/heads/master/design/go2draft-type-parameters.md>

- Functions can have an additional type parameter list that uses square brackets but otherwise looks like an ordinary parameter list:
func F[T any](p T) { ... }
- These type parameters can be used by the regular parameters and in the function body.
- Types can also have a type parameter list: **type M[T any] []T**
- Each type parameter has a type constraint, just as each ordinary parameter has a type: **func F[T Constraint](p T) { ... }**
- Type constraints are **interface types**.

Type Parameters - Draft Design - II

Ian Lance Taylor, Robert Griesemer, January 11, 2021:

<https://go.gogglesource.com/proposal/+refs/heads/master/design/go2draft-type-parameters.md>

- The new predeclared name **any** is a type constraint that permits **any type**.
- Interface types used as **type constraints** can have a list of **predeclared types**; only type arguments that match one of those types satisfy the constraint.
- Generic functions may **only use operations permitted by the type constraint**.
- Using a **generic function** or type requires **passing type arguments**.
- **Type inference** permits omitting the type arguments of a function call in common cases.

Simple Generics Demo

```
package main
```

```
import "fmt"
```

```
func PrintSlice [T any] (s []T) {  
    for _,v := range s{  
        fmt.Print(v)  
    }  
}
```

```
func main() {  
    PrintSlice([]int {1,2,3,4,5,6,7,8,9})  
    fmt.Println()  
    PrintSlice([]string {"a","b","c","d"})  
    fmt.Println()  
}
```

- Enable experimental support in Goland
- Rename to .go2 => Ctrl+Alt+Shift+S to run code in [The go2go Playground](#)

Map-Filter-Reduce Demo - I

```
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  
}
```

```
func Reduce [T1, T2 any] (s []T1, initializer T2, f func(T2, T1) T2) T2 {  
    r := initializer  
    for _, v := range s { r = f(r, v) }  
    return r  
}
```

```
func Filter [T any] (s []T, f func(T) bool) []T {  
    var r []T  
    for _, v := range s {  
        if f(v) { r = append(r, v) }  
    }  
    return r  
}
```


Map-Filter-Reduce Demo - II

```
func main() {  
    s := []int{1, 2, 3}  
  
    floats := Map(s, func(i int) float64 { return float64(i) })  
    fmt.Println(floats) // Now floats is []float64{1.0, 2.0, 3.0}.  
  
    sum := Reduce(s, 0, func(i, j int) int { return i + j })  
    fmt.Println(sum) // Now sum is 6.  
  
    evens := Filter(s, func(i int) bool { return i%2 == 0 })  
    fmt.Println(evens) // Now evens is []int{2}.  
}
```

Generic Sorting - OrderedSlice Demo

```
type Ordered interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        string
}

type orderedSlice[T Ordered] []T
func (s orderedSlice[T]) Len() int      { return len(s) }
func (s orderedSlice[T]) Less(i, j int) bool { return s[i] < s[j] }
func (s orderedSlice[T]) Swap(i, j int)    { s[i], s[j] = s[j], s[i] }
func OrderedSlice[T Ordered](s []T) {
    sort.Sort(orderedSlice[T](s))
}

func main() {
    s1 := []int32{3, 5, 2}
    OrderedSlice(s1)
    fmt.Println(s1) // Now s1 is []int32{2, 3, 5}
    s2 := []string{"a", "c", "b"}
    OrderedSlice(s2)
    fmt.Println(s2) // Now s2 is []string{"a", "b", "c"}
}
```

Q&A. Projects mentoring



Recommended Literature

- The Go Documentation - <https://golang.org/doc/>
- The Go Bible: Effective Go - https://golang.org/doc/effective_go.html
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>