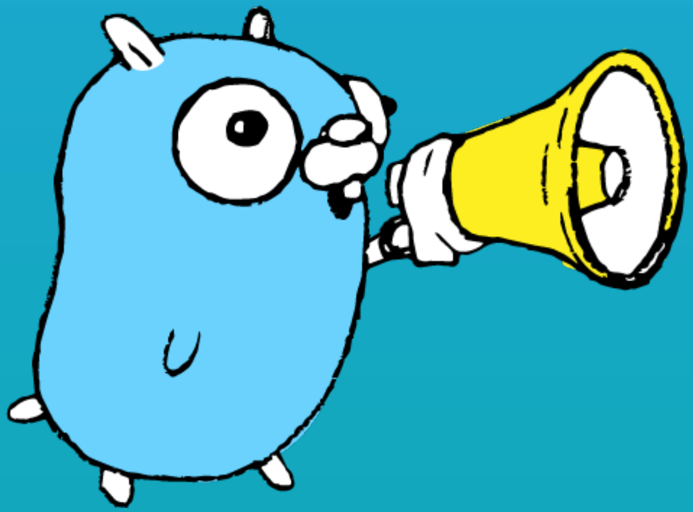
COSCUP 07/31/2022

Go Generic

Gaston Chiu

crypto.com



Today’s (glorious) blather.

Introduction 01

Overview 02

Underlying 03

Comparison 04

SECTION ONE

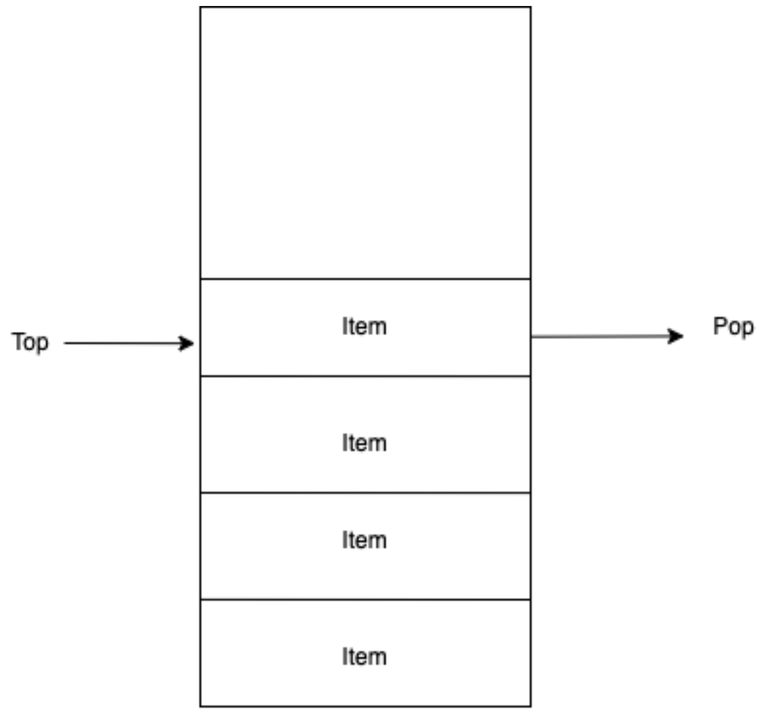
Introduction

“

Generics Definition

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.

Example - Stack



Item: int, float, struct, string, etc…

Custom function for specific type (w/o generic)

- Bad for reusable code
- Bad for abstraction as a library
- Highly couple types with algorithm

SECTION TWO

Overview

“

System One Boxing

Wrap the type to generic type in “runtime”.
E.g. interface, empty interface

Empty Interface VS Interface

Empty Interface:

```go
var z interface{}
```

Interface:

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

Type Assertion

- Couple types with algorithm
- Performance penalty in runtime for type assertion

Reflection

- Unfriendly interface and not easy to maintain.
- Unsafe code require more validation.
- Performance penalty in runtime.

“

System Two Monomorphization

Instantiation the function with type parameter in compile time.

Code Generate

- Increase maintenance cost
- Increase difficulty for debugging

Type Parameter (1.18+)

- Leverage compiler syntax check
- Handle the code generate process in compiler

Type Parameter

Ordinary Type

```
// x, y: parameter names; int: parameter type
func (x int, y int) {}
```

Type Parameter

```
// P, Q type parameter name; any type parameter constrain
func[P any, Q any] (x P, y Q) {}
```

Type Parameter (Go 1.18+)

```go
// declare a new stack
// s := NewStack[int](10)
// s2 := NewStack[string](10)

type Stack[T any] struct {
    items []T
}

func NewStack[T any](cap int) *Stack[T] {
    return &Stack[T]{
        items: make([]T, 0, cap),
    }
}

func (s *Stack[T]) Push(i T) {
    s.items = append(s.items, i)
}


func (s *Stack[T]) Pop() (T, error) {
    if len(s.items) == 0 {
        var zero T
        return zero, ErrEmptySlice
    }

    tmp := s.items[len(s.items)-1]
    s.items = s.items[:len(s.items)-1]

    return tmp, nil
}
```

Type set (Go 1.18+)

```go
type Ordered interface {
	~int | ~int8 | ~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]
}
```

SECTION THREE

Underlying

Interface in Go

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type itab struct {
    inter *interfacetype
    _type *_type
    hash  uint32 // copy of _type.hash. Used for type switches.
    _     [4]byte
    fun   [1]uintptr // variable sized. fun[0]==0 means _type does not
implement inter.
}

type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

Empty Interface in Go

```go
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

```go
type _type struct {
    size       uintptr
    ptrdata    uintptr
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}
```

Type assertion

- Couple types with algorithm
- Performance penalty in runtime for type assertion

Type Assertion (non empty interface)

```
var j uint32
var Eface interface{}

func assertion() {
      i := uint32(42)
      Eface = i
      j = Eface.(uint32)
}
```

```
type eface struct {
      _type *_type
      data  unsafe.Pointer
}


0x0030 00048 (./main.go:8)   LEAQ type.uint32(SB),
CX

func convT32(val uint32) (x unsafe.Pointer) {
      if val < uint32(len(staticuint64s)) {
            x = unsafe.Pointer(&staticuint64s[val])
            if goarch.BigEndian {
                  x = add(x, 4)
            }
      } else {
            x = mallocgc(4, uint32Type, false)
            *(*uint32)(x) = val
      }
      return
}
```

Type Assertion (non empty interface)

```go
var j uint32
var Eface interface{}


func assertion() {
      i := uint32(42)
      Eface = i
      j = Eface.(uint32)
}
```

```
0x0065 00101 MOVQ    "".Eface(SB), AX          ;; AX = Eface._type
0x006c 00108 MOVQ    "".Eface+8(SB), CX          ;; CX = Eface.data
0x0073 00115 LEAQ    type.uint32(SB), DX          ;; DX = type.uint32
0x007a 00122 CMPQ    AX, DX                     ;; Eface._type ==
type.uint32 ?
0x007d 00125 JNE    162                      ;; no? panic our way outta here
0x007f 00127 MOVL    (CX), AX                   ;; AX = *Eface.data
0x0081 00129 MOVL    AX, "".j(SB)                 ;; j = AX = *Eface.data
;; exit
0x0087 00135 MOVQ    40(SP), BP
0x008c 00140 ADDQ    $48, SP
0x0090 00144 RET
;; panic: interface conversion: <iface> is <have>, not <want>
0x00a2 00162 MOVQ    AX, (SP)                   ;; have: Eface._type
0x00a6 00166 MOVQ    DX, 8(SP)                   ;; want: type.uint32
0x00ab 00171 LEAQ    type.interface {}(SB), AX    ;; AX =
type.interface{} (eface)
0x00b2 00178 MOVQ    AX, 16(SP)                  ;; iface: AX
0x00b7 00183 CALL    runtime.panicdottypeE(SB)    ;; func
panicdottypeE(have, want, iface *_type)
0x00bc 00188 UNDEF
0x00be 00190 NOP
```

Reflection

- Unfriendly interface and not easy to maintain.
- Unsafe code require more validation.
- Performance penalty in runtime.

Reflection emptyInterface

```go
// emptyInterface is the header for an interface{} value.
type emptyInterface struct {
	typ  *rtype
	word unsafe.Pointer
}


type rtype struct {
	size       uintptr
	ptrdata    uintptr // number of bytes in the type that can contain pointers
	hash       uint32  // hash of type; avoids computation in hash tables
	tflag      tflag   // extra type information flags
	align      uint8   // alignment of variable with this type
	fieldAlign uint8   // alignment of struct field with this type
	kind       uint8   // enumeration for C
	// function for comparing objects of this type
	// (ptr to object A, ptr to object B) -> ==?
	equal     func(unsafe.Pointer, unsafe.Pointer) bool
	gcdata    *byte   // garbage collection data
	str       nameOff // string form
	ptrToThis typeOff // type for pointer to this type, may be zero
}
```

Rtype in reflection

```go
func (t *rtype) Kind() Kind { return Kind(t.kind & kindMask) }

func (t *rtype) Size() uintptr { return t.size }
```

Reflection TypeOf()

```go
func TypeOf(i any) Type {
	eface := *(*emptyInterface)(unsafe.Pointer(&i))
	return toType(eface.typ)
}
```

Reflection Value

```go
type Value struct {
	typ *rtype

	ptr unsafe.Pointer

	flag
}


// ValueOf returns a new Value initialized to the concrete value
// stored in the interface i. ValueOf(nil) returns the zero Value.
func ValueOf(i any) Value {
	if i == nil {
		return Value{}
	}

	escapes(i)

	return unpackEface(i)
}


// unpackEface converts the empty interface i to a Value.
func unpackEface(i any) Value {
	e := (*emptyInterface)(unsafe.Pointer(&i))
	// NOTE: don't read e.word until we know whether it is really a pointer or not.
	t := e.typ
	if t == nil {
		return Value{}
	}
	f := flag(t.Kind())
	if ifaceIndir(t) {
		f |= flagIndir
	}
	return Value{t, e.word, f}
}
```

Reflection ValueOf()

```go
func (v Value) Float() float64 {
        k := v.kind()
        switch k {
        case Float32:
                return float64(*(*float32)(v.ptr))
        case Float64:
                return *(*float64)(v.ptr)
        }
        panic(&ValueError{"reflect.Value.Float", v.kind()})
}
```

Type parameter

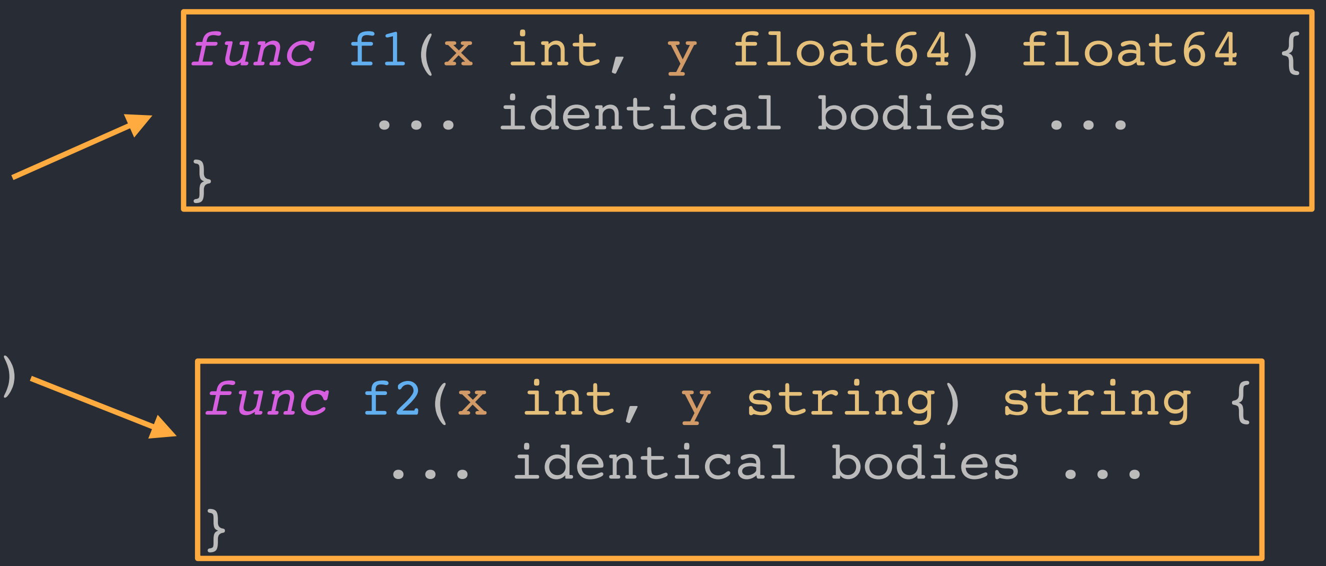
- Friendly for maintenance
- Compile time overhead.

Stenciling

```
func f[T1, T2 any](x T1, y T2) T2 {
      ...
}

var a float64 = f[int, float64](7, 8.0)


var b string = f[int, string](7, “aaaa")
```



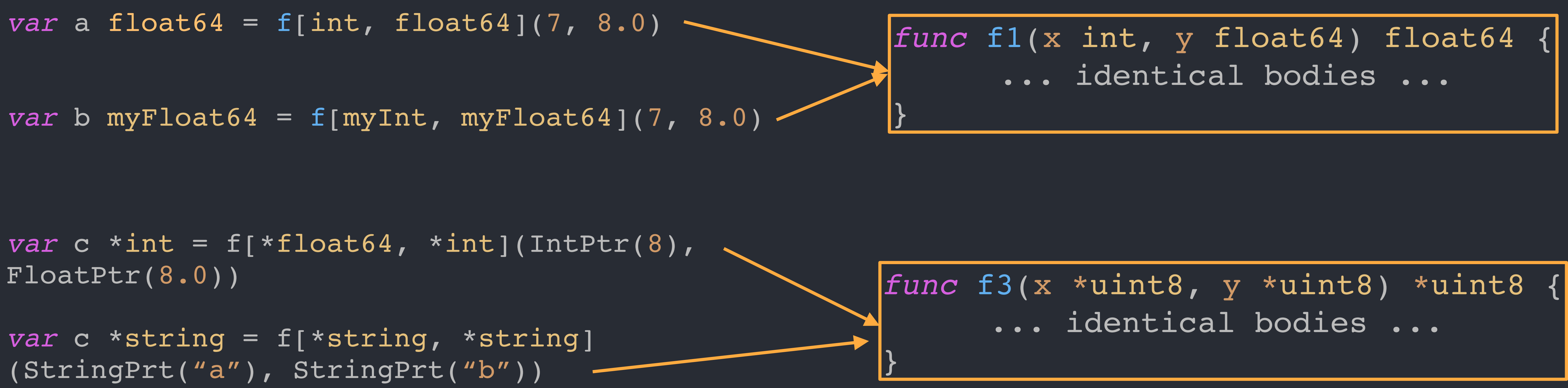
```
func f1(x int, y float64) float64 {
      ... identical bodies ...
}
```

```
func f2(x int, y string) string {
      ... identical bodies ...
}
```

GC Shape Grouping

```
func f[T1, T2 any](x T1, y T2) T2 {
        ...
}


type myInt int
type myFloat64 float64
```



```
var a float64 = f[int, float64](7, 8.0)

var b myFloat64 = f[myInt, myFloat64](7, 8.0)
```

```
func f1(x int, y float64) float64 {
      ... identical bodies ...
}
```

```
var c *int = f[*float64, *int](IntPtr(8),
FloatPtr(8.0))

var c *string = f[*string, *string]
(StringPrt(“a”), StringPrt(“b”))
```

```
func f3(x *uint8, y *uint8) *uint8 {
      ... identical bodies ...
}
```

Compatible with interface (not recommended)

```go
type Person interface {
        GetName() string
}

type PersonImp struct {
        Name string
}

func (p PersonImp) GetName() string {
        return p.Name
}

func GenericPerson[T Person](p T) T {
        p.GetName()

        return p
}
```

Dictionaries

```
type dictionary struct {
      T1 *runtime._type
      T2 *runtime._type
      …
      tab *itab // non empty interface
}


type itab struct {
      inter *interfacetype
      _type *_type
      hash  uint32 // copy of _type.hash. Used for type switches.
      _     [4]byte
      fun   [1]uintptr // variable sized. fun[0]==0 means _type does not
implement inter.
}
```

Q&A

SECTION FOUR

Comparison

Template Programing (C++)

```
template <class T> T myMax(T x, T y)
{
     return (x > y) ? x : y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl; // Call myMax for int

    return 0;
}
```

Operator Overload (C++)

```
class Rectangle {
public:
    Rectangle(int w, int l) : w_(w), l_(l) {}

    int GetSize() { return w_ * l_; }

    int GetWid() { return w_; }

    int GetLen() { return l_; }

private:
    int w_;
    int l_;
};



bool operator> (Rectangle& ra, Rectangle& rb) {
    return ra.GetSize() > rb.GetSize();
}
```

Operator Overload (C++)

```cpp
class Hack
{
};

Hack& operator< (Hack &a , Hack &b)
{
    cout << "less than operator" << endl;
    return a;
}

Hack& operator> (Hack &a, Hack &b)
{
    cout <<  "greater than operator" << endl;
    return a;
}

int main(int argc, char ** argv)
{
    Hack vector;
    Hack UINT4;
    Hack foo;

    vector<UINT4> foo;

    return(0);
}
```

Class template specialization (C++)

```cpp
template<class T>
T Max(T a, T b) {
    return a > b ? a : b;
}

template<>
string Max<string>(string a, string b) {
    return a.length() > b.length() ? a : b;
}

int main() {
    cout << Max<int>(3, 2) << endl;
    cout << Max<string>(string("aaac"), string ("bbb"));
}
```

References

- Go reflection https://halfrost.com/go_reflection/
- Go interface https://halfrost.com/go_interface/
- Go internal interface: https://cmc.gitbook.io/go-internals/chapter-ii-interfaces
- Go assembly https://go.dev/doc/asm
- Go assembly https://9p.io/sys/doc/asm.html
- Go reflection https://go101.org/article/reflection.html
- Go Map Reduce https://coolshell.cn/articles/21164.html
- Go generic performance https://planetscale.com/blog/generics-can-make-your-go-code-slower
- Go generic implement proposal ttps://github.com/golang/proposal/blob/master/design/generics-implementation-dictionaries-go1.18.md
- Models of Generics and Metaprogramming https://www.gushiciku.cn/pl/2SvT
- 恐怖的C++语言 https://coolshell.cn/articles/1724.html
- C++ operator overloading: https://www.geeksforgeeks.org/operator-overloading-c/
- Go assembly https://github.com/cch123/asmshare/blob/master/layout.md
- GO编程模式 ： 泛型编程 https://coolshell.cn/articles/21615.html

Thanks