GO泛型

跟着 Go 作者学泛型

增长后端 王若愚

ShareDay

Jul. 06th 2023

go没有泛型的模样

```go
func StrSliceToUintSlice(arr []string) ([]uint64, error) {
  res := make([]uint64, 0, len(arr))
  for i := range arr {
    num, err := strconv.ParseUint(arr[i], 10, 64)
    if err != nil {
      return nil, err
    }
    res = append(res, num)
  }
  return res, nil
}

func StrSliceToIntSlice(arr []string) ([]int64, error) {
  res := make([]int64, 0, len(arr))
  for i := range arr {
    num, err := strconv.ParseInt(arr[i], 10, 64)
    if err != nil {
      return nil, err
    }
    res = append(res, num)
  }
  return res, nil
}
```

Go1.18 泛型的三个特性

1. Type parameters for functions and types，即函数和类型的类型参数
2. Type sets defined by interfaces，即由接口定义的类型集合
3. Type inference，即类型推断

1、函数和类型的类型参数

1.1、类型参数列表（Type parameter lists）

类型参数列表看起来是带方括号的普通参数列表。通常，类型参数以大写字母开头，以强调它们是类型：

```
[P, Q constraint1, R constraint2]
```

非泛型版本的最小值函数

```
func min(x, y float64) float64 {
  if x < y {
    return x
  }
  return y
}
```

泛型版本的最小值函数

```
func min[T Ordered](x, y T) T {
  if x < y {
    return x
  }
  return y
}
```

那这个泛型函数如何调用呢？

```
m := min[int](2, 3)
```

1.2、实例化（Type parameter lists）

在调用时，会进行实例化过程：

1）用类型实参（type arguments）替换类型形参（type parameters）

2）检查类型实参（type arguments）是否实现了类型约束

如果第 2 步失败，实例化（调用）失败。

所以，调用过程可以分解为以下两步：

```
func min[T Ordered](x, y T) T {
  if x < y {
    return x
  }
  return y
}

fmin := min[float64]
m := fmin(2.3, 3.4)

// 和下面等价
m := min[float64](2.3, 3.4)
// 相当于 m := (min[float64])(2.3, 3.4)
```

1.3、类型的类型参数

类型也可以有类型参数。下面是一个泛型版二叉树：

```
type Tree[T any] struct {
  left, right *Tree[T]
  data        T
}

func (t *Tree[T]) Lookup(x T) *Tree[T]

var stringTree Tree[string]
```

其中的 [T interface{}] ，跟函数的类型参数语法是一样的，T 相当于是一个类型，

所以，之后用到 Tree 的地方，T 都跟随着，即 Tree[T]，包括方法的接收者（receiver）。

2、类型集合

2.1、类型约束

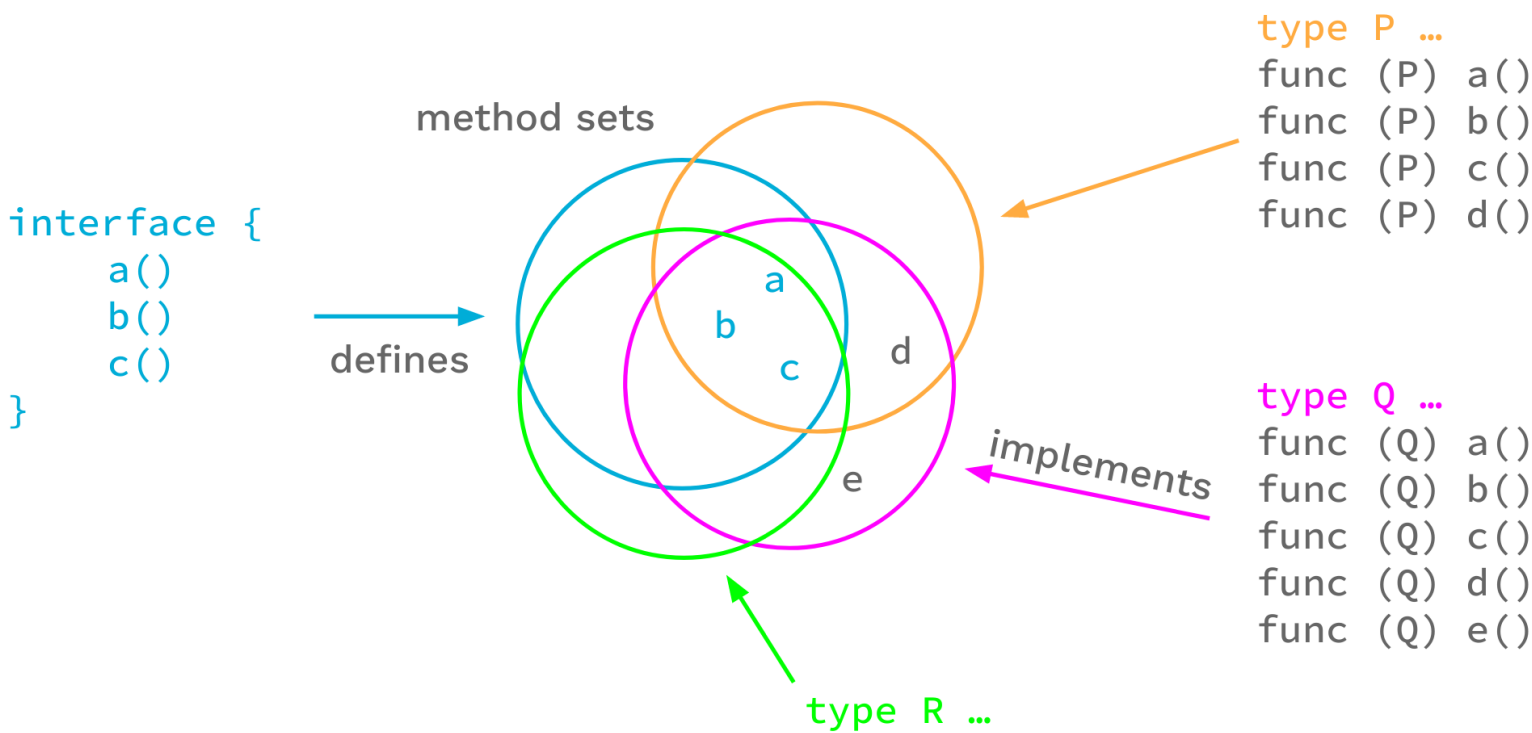
我们定义接口来约束某一个类型是否实现接口的方法

```
type IGet interface {
  Get()
}

func Test(ig IGet) {
  ig.Get()
}
```

在泛型中我们也可以限制泛型的中的类型

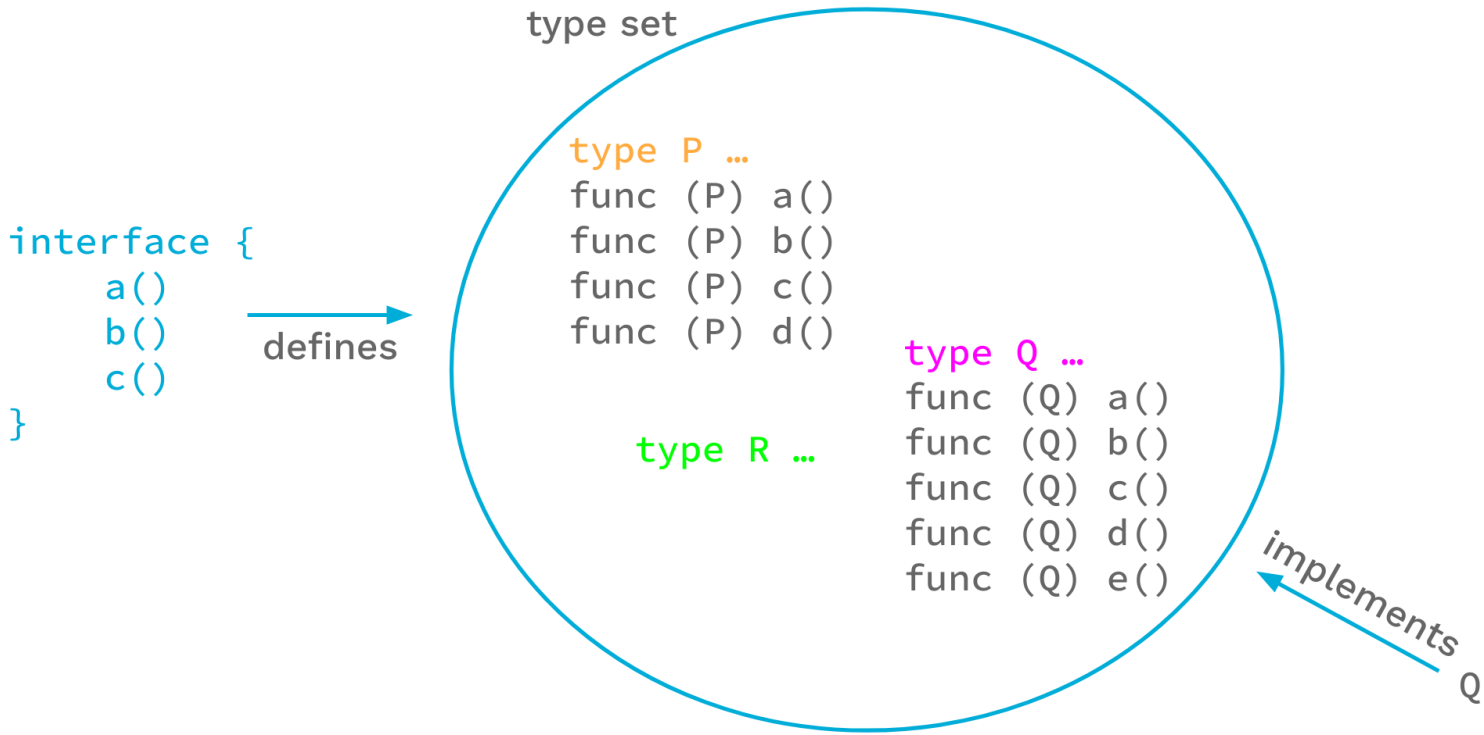
```
type Ordered interface {
  int | float64 | ~string
}

func min[T Order](x, y T) T {
  if x < y {
    return x
  }
  return y
}
```

根据 Go 的规则，类型 P、Q、R 方法中包含了 a、b、c，因此它们实现了接口

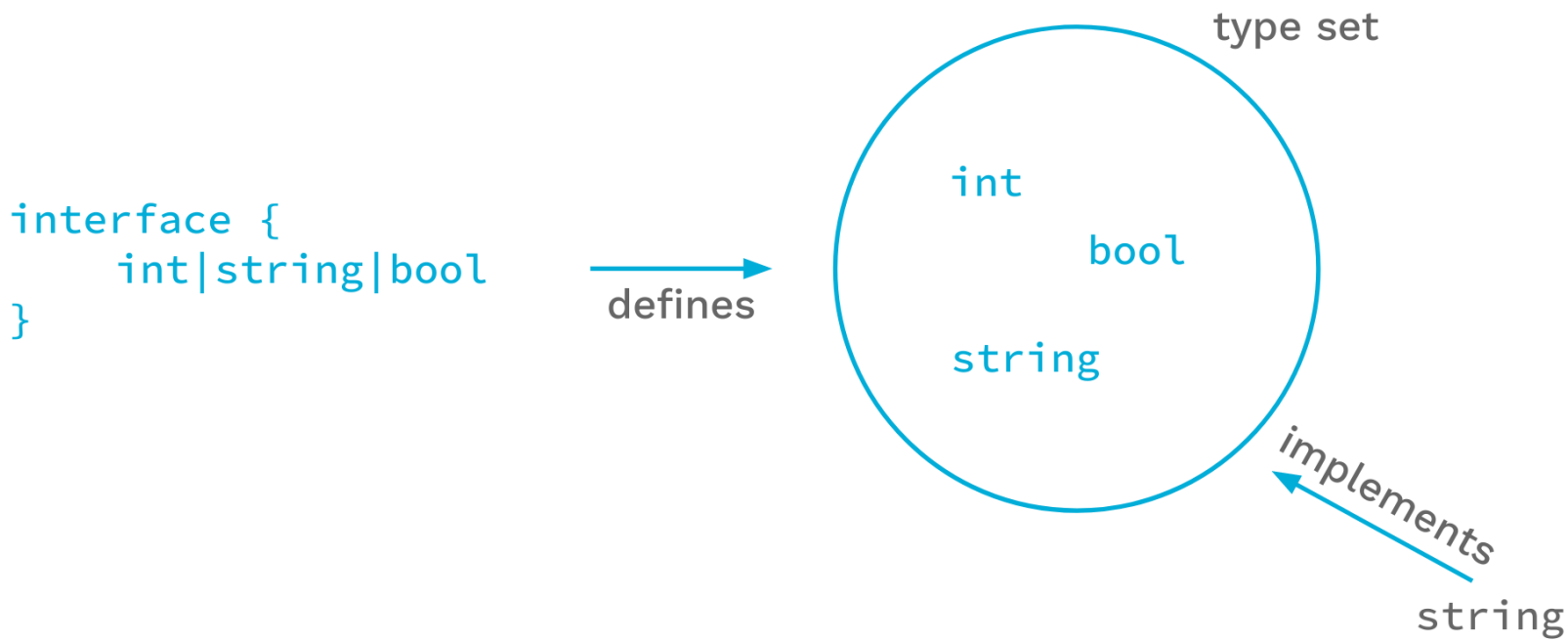


反过来可以说，接口也定义了类型集（type sets）

类型 P、Q、R 都实现了左边的接口（因为都实现了接口的方法集），因此我们可以说该接口定义了类型集。



既然接口是定义类型集，只不过是间接定义的：类型实现接口的方法集。而类型约束是类型集，因此完全可以重用接口的语义，只不过这次是直接定义类型集：



3、类型推断

3、类型推断

```go
// 在调用泛型函数时，提供类型实参感觉有点多余。
// Go 虽然是静态类型语言，但擅长类型推断。
// 因此泛型这里，Go 也实现了类型推断。
// 在这理不需要提供类型实参 m = min[int](a, b)
var a, b, m int
m = min(a, b)
```

```go
// 这个函数的目的是希望对 s 中的每个元素都乘以参数 c，
// 最后返回一个新的切片。
func Scale[E constraints.Integer](s []E, c E) []E {
  r := make([]E, len(s))
  for i, v := range s {
    r[i] = v * c
  }
  return r
}
```

```go
// 定义一个结构体
type Point []int32

func (p Point) String() string {
  return "point"
}
```

```go
// 很显然，Point 类型的切片可以传递给 Scale
// 我们希望对 p 进行 Scale，得到一个新的 p，
// 但发现返回的 r 根本不是 Point
func ScaleAndPrint(p Point) {
  r := Scale(p, 2)
  fmt.Println(r.String()) // r.String undefined (type []int32 has no fi
}

func main() {
  p := Point{3, 2, 4}
  ScaleAndPrint(p)
}
```

```go
// 加入了泛型 S，以及额外的类型约束 ~[]E
// 调用 Scale 时，不需要 r := Scale[Point, int32](p, 2),
// 因为 Go 会进行类型推断
func Scale[S ~[]E, E constraints.Integer](s S, c E) S {
  r := make(S, len(s))
  for i, v := range s {
    r[i] = v * c
  }
  return r
}
```

4、一些不足

4.1、不支持泛型方法

主要原因Go泛型的处理是在编译的时候实现的，泛型方法在编译的时候，如果没有上下文的分析推断，很难判断泛型方案该如何实例化，甚至判断不了，导致目前Go实现中不支持泛型方法：

```
type StudentModel struct{}
type Client struct{  }
type Querier struct{
  client *Client
}
// Identity 一个泛型方法，支持任意类型.
func (q *Querier) All[T any](ctx context) ([]T, error) { return nil, nil } // method must have no type parameters
```

我们要想实现Client，只能在结构上加泛型：

```
type Querier[T any] struct {
  client *Client
}
func NewQuerier[T any](c *Client) *Querier[T] {
  return &Querier[T]{
    client: c
  }
}
func (q *Querier[T]) All(ctx context.Context) ([]T, error) {return nil, nil}
```

```go
package p1
// S 是一个普通的struct,但是包含一个泛型方法Identity.
type S struct{}
// Identity 一个泛型方法，支持任意类型.
func (S) Identity[T any](v T) T { return v }
```

```go
package p2
// HasIdentity 定义了一个接口
type HasIdentity interface {
  Identity[T any](T) T
}
```

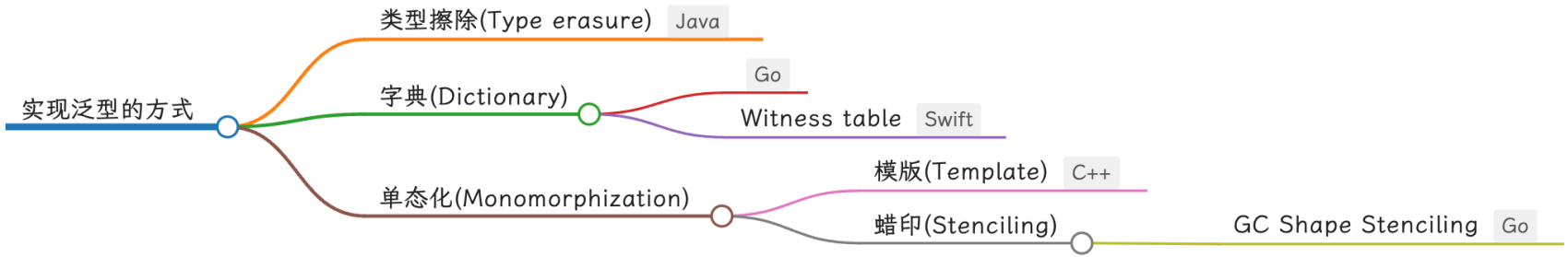
```go
package p3
import "p2"
// CheckIdentity 是一个普通函数,
// 检查实参是不是实现了HasIdentity接口,
// 如果是，则调用这个接口的泛型方法Identity.
func CheckIdentity(v interface{}) {
  if vi, ok := v.(p2.HasIdentity); ok {
    if got := vi.Identity[int](0); got != 0 {
      panic(got)
    }
  }
}
```

```go
package p4
import (
  "p1"
  "p3"
)
// CheckSIdentity 传参S给CheckIdentity.
func CheckSIdentity() {
  p3.CheckIdentity(p1.S{})
}
```

一切看起来都没有问题，但是问题是package p3不知道p1.S类型，整个程序中如果也没有其它地方调用p1.S.Identity,依照现在的Go编译器的实现，是没有办法为p1.S.Identity[int]生成对应的代码的。

4、go泛型的实现

4.1 实现泛型的方式



- 字典(Dictionary)

编译器在编译泛型函数时只生成了一份函数副本，通过新增一个字典参数来供调用方传递类型参数(Type Parameters)，这种实现方式称为字典传递(Dictionary passing)。

- 蜡印(Stenciling)

GC Shape这种技术就是通过对类型的底层内存布局（从内存分配器或垃圾回收器的视角）分组，对拥有相同的类型内存布局的类型参数进行蜡印，这样就可以避免生成大量重复的代码。

4.2 两个具体类型具有相同的基础类型

```go
package main

import "fmt"

type A struct{}

func (*A) A() A {
    return A{}
}

type B struct{}

func (*B) B() B {
    return B{}
}

func foo[T A | B](data T) {
    fmt.Println(data)
}

func main() {
    foo(A{})
    foo(B{})
}
```

main.main

file: /Users/wangruoyu/Documents/project/go/ddd/main.go

```
CMPQ 0x10(R14), SP
JBE 0x48b6bb
SUBQ $0x10, SP
MOVQ BP, 0x8(SP)
LEAQ 0x8(SP), BP
LEAQ main..dict.foo[main.A](SB), AX
NOPL 0(AX)(AX*1)
CALL main.foo[go.shape.struct {}_0](SB)
LEAQ main..dict.foo[main.B](SB), AX
CALL main.foo[go.shape.struct {}_0](SB)
MOVQ 0x8(SP), BP
ADDQ $0x10, SP
RET

NOPL 0(AX)(AX*1)
CALL runtime.morestack_noctxt.abi0(SB)
JMP main.main(SB)
```

```
/Users/wangruoyu/Documents/project/go/ddd/main.go
18        fmt.Println(data)
19    }
20
21    func main() {
22        foo(A{})
23        foo(B{})
24    }
25
```

4.3 两个具体类型具有不同的基础类型

```go
package main

import "fmt"

type A struct {
    data int    field data i
}

func (*A) A() A {
    return A{}
}

type B struct {
    data string    field dat
}

func (*B) B() B {
    return B{}
}

func foo[T A | B](data T) {
    fmt.Println(data)
}

func main() {
    foo(A{})
    foo(B{})
}

```

main.main

file: /Users/wangruoyu/Documents/project/go/ddd/main.go

```
CMPQ 0x10(R14), SP
JBE 0x48b6bf
SUBQ $0x20, SP
MOVQ BP, 0x18(SP)
LEAQ 0x18(SP), BP
LEAQ main..dict.foo[main.A](SB), AX
XORL BX, BX
NOPL 0(AX)
CALL main.foo[go.shape.struct { main.data int }_0](SB)
LEAQ main..dict.foo[main.B](SB), AX
XORL BX, BX
XORL CX, CX
CALL main.foo[go.shape.struct { main.data string }_0](SB)
MOVQ 0x18(SP), BP
ADDQ $0x20, SP
RET

NOPL
CALL runtime.morestack_noctxt.abi0(SB)
JMP main.main(SB)
```

```
/Users/wangruoyu/Documents/project/go/ddd/main.go
22        fmt.Println(data)
23    }
24
25    func main() {
26        foo(A{})
27        foo(B{})
28    }
29
```

4.4 两个具体类型的指针

```go
package main

import "fmt"

type A struct {
    data int    field data is u
}

func (*A) A() A {
    return A{}
}

type B struct {
    data string    field data i
}

func (*B) B() B {
    return B{}
}

func foo[T *A | *B](data T) {
    fmt.Println(data)
}

```

main.main

file: /Users/wangruoyu/Documents/project/go/ddd/main.go

```
CMPQ 0x10(R14), SP
JBE 0x48b6e5
SUBQ $0x18, SP
MOVQ BP, 0x10(SP)
LEAQ 0x10(SP), BP
LEAQ type.*+50400(SB), AX
NOPL 0(AX)(AX*1)
CALL runtime.newobject(SB)
MOVQ AX, BX
LEAQ main..dict.foo[*main.A](SB), AX
CALL main.foo[go.shape.*uint8_0](SB)
LEAQ type.*+50528(SB), AX
NOPL 0(AX)(AX*1)
CALL runtime.newobject(SB)
MOVQ $0x0, 0(AX)
MOVQ AX, BX
LEAQ main..dict.foo[*main.B](SB), AX
CALL main.foo[go.shape.*uint8_0](SB)
MOVQ 0x10(SP), BP
ADDQ $0x18, SP
RET

CALL runtime.morestack_noctxt.abi0(SB)
JMP main.main(SB)
```

```
/Users/wangruoyu/Documents/project/go/ddd/main.go
22      fmt.Println(data)
23  }
24
25  func main() {
26      foo(&A{})
27      foo(&B{})
28  }
29
```

参考

- https://go.dev/blog/intro-generics
- https://changkun.de/research/talks/generics118.pdf
- https://deepsource.com/blog/go-1-18-generics-implementation
- https://mytechshares.com/2022/05/01/generics-can-make-your-go-code-slower/

THANKS!