

特别放送 | Go泛型：用法、原理与最佳实践

2023-01-28 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 09:33 大小 13.09M



你好，我是郑建勋。

泛型一直是自 Go 语言诞生以来讨论最热烈的话题之一，之前，Go 语言因为没有泛型而被很多人吐槽过。🔗[经过了多年的设计](#)，我们正式迎来了 Go1.18 泛型。这节课就让我们来看一看泛型的几个重要问题，你也可以先问一问自己下面几个问题。

- 什么是泛型？
- Go 为什么需要泛型？
- Go 之前为什么没有泛型？
- Go 泛型的特性与使用方法是什么？

什么是泛型？

我们先来看看 [维基百科](#) 对泛型的定义。

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.

泛型编程的核心思想是从具体的、高效的运算中抽象出通用的运算，这些运算可以适用于不同形式的数据，从而能够适用于各种各样的场景。

显然，泛型是高级语言为了让一段代码拥有更强的抽象能力和表现力而设计出来的。

许多语言都有对泛型的支持，我们可以看看其他语言是怎么实现泛型的。在 **Java** 中对函数进行泛型抽象的代码如下所示。

 复制代码

```
1 public static <E> boolean containsElement(E [] elements, E element){
2     for (E e : elements){
3         if(e.equals(element)){
4             return true;
5         }
6     }
7     return false;
8 }
```

其中，**E** 代表类型参数，可以指代任何类型。这个函数的功能是查看任意类型的元素是否在对应的数组中。

Java 是通过 [类型擦除](#)（Type Erasure）来实现泛型的。类型擦除指的是在编译阶段，编译器会将上面的泛型函数实例化为如下的形式。

 复制代码

```
1 public static boolean containsElement(Object [] elements, Object element){
2     for (Object e : elements){
3         if(e.equals(element)){
4             return true;
5         }
6     }
7     return false;
8 }
```

在 Java 中，`java.lang.Object` 是所有类型的父类。Java 编译时的这个特性使得泛型函数并不会实例化为一个具体类型的函数，这导致了程序在运行时需要花费额外的成本处理实际类型与 `Object` 之间的转换。

同时，Java 中无处不在的 [自动装箱和拆箱机制](#) 也不可避免地在运行时消耗时间与空间。

而 C++ 中实现泛型的方式称为模版。我们举一个例子，用 C++ 求两个数的最大值。这里，C++ 中的模版只存在于编译时，编译时会将模版函数实例化为具体的类型。

 复制代码

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T> T myMax(T x, T y)
5 {
6     return (x > y) ? x : y;
7 }
8
9 int main()
10 {
11     cout << myMax<int>(3, 7) << endl;
12     cout << myMax<double>(3.0, 7.0) << endl;
13     cout << myMax<char>('g', 'e') << endl;
14
15     return 0;
16 }
```

例如，`myMax` 会将 `myMax` 函数实例化如下。

 复制代码

```
1 int myMax(int x, int y)
2 {
3     return (x > y) ? x : y;
4 }
```

而 `myMax` 会实例化另一个 `myMax`。

 复制代码


```
1 char myMax(char x, char y)
2 {
3     return (x > y) ? x : y;
4 }
```



天下无鱼

<https://shikey.com/>

C++ 模版的这一特性使得它在编译时存在代码膨胀的问题，减慢了编译的速度。而由于 C++ 允许循环依赖、模版会被实例化多次等问题，编译速度会进一步减慢。

我们可以看到，泛型在使代码变得抽象的同时，也不可避免地需要花费  额外的成本。好在这通常是值得的，因为我们的最终目标就是能够写出更加抽象、更有表现力的高级语言。

Go 为什么需要泛型？

Go 作为强类型语言，在没有泛型之前，在许多场景下书写代码都很繁琐。例如，还是刚才要判断两个数的大小这个例子，在 Go 中一般需要调用 `math.Min` 与 `math.Max` 两个函数。然而，由于函数参数定义了 `float64` 类型，所以并不适合其他的类型。

 复制代码

```
1 func Max(x, y float64) float64
2 func Min(x, y float64) float64
```

同样的例子还存在于加法操作中，我们无法写出一个能够适用所有类型的加法函数。在过去，每一个类型都需要写一个单独的函数，如下所示。

 复制代码

```
1 func SumInt64(x,y int64) int64{
2     return x + y
3 }
4
5 func SumInt32(x,y int32) int32{
6     return x + y
7 }
8
9 func SumUint64(x,y uint64) uint64{
10    return x + y
11 }
12
13 func SumUint32(x,y uint32) uint32{
14    return x + y
15 }
```

过去要解决这种问题，一般会采用 `go generate` 等工具自动生成代码，或者使用如下接口的方式。但都相对比较繁琐。



📄 复制代码

```
1 func Sum(a, b interface{}) interface{} {
2     switch a.(type) {
3     case int:
4         a1 := a.(int)
5         b1 := b.(int)
6         return a1 + b1
7     case float64:
8         a1 := a.(float64)
9         b1 := b.(float64)
10        return a1 + b1
11    default:
12        return nil
13    }
14 }
```

而泛型解决了这一问题。我们以数组的加法为例，通用的泛型加法函数如下所示，这是 Go 中的新语法。Go1.18 后扩展了接口的能力，在这个例子中，`Number` 是类型的集合，用于对类型进行约束，后面我们还会详细介绍。

📄 复制代码

```
1 type Number interface {
2     int | int64 | float64
3 }
4
5 func Sum[T Number](numbers []T) T {
6     var total T
7     for _, x := range numbers {
8         total += x
9     }
10    return total
11 }
```

上面的语法可以修改成更简洁的样子，如下。这一个函数适用于 `int`、`int64`、`float64` 这三个类型数组的加法，而在过去却需要分别书写 3 个函数。

📄 复制代码

```
1 func Sum[T int | int64 | float64](numbers []T) T {
2     var total T
```

```
3   for _, x := range numbers {
4       total += x
5   }
6   return total
7 }
```



你可能会问，既然泛型有这么多好处，那为什么之前 Go 一直都没有实现泛型呢？我们可以从[官方文档FAQ](#)中找到问题的答案。总结一下，主要有下面几个方面的原因。

- Go 语言专注于软件工程本身，其目的是设计出简单、可读、可扩展的语言。因此一开始为了确保其简单性，就没有将泛型作为语言的设计目标。
- 泛型通常会带来额外的成本，这个成本可能来自编译时或者运行时的耗时，还有复杂度的上升。因此设计泛型时，需要做好成本与收益的平衡。Go 团队一开始并没有想清楚泛型的最终形态，并且 Go 的空接口也给了和泛型类似的灵活性，这就降低了泛型的紧迫性。

泛型的特性与使用方法

Go1.18 引入了泛型之后，一些之前重复的代码就可以用更简单的泛型代码来表示了。Go 的泛型有下面这些特点。

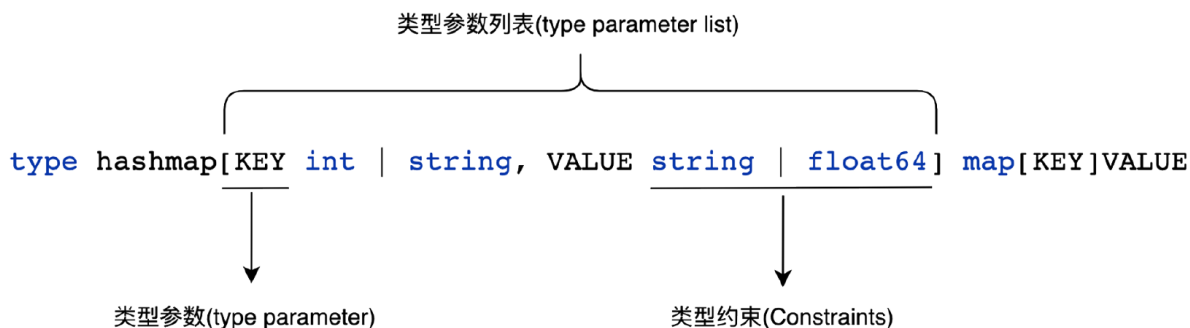
- 只存在于编译时，并不会在运行时有任何的性能损耗。
- 由于存在泛型，在编译时的编译速度会下降。但是由于 Go 对泛型的限制加上 Go 依赖管理禁止了循环依赖，Go 没有代码膨胀问题，编译速度仍然一骑绝尘。
- 与 C++ 不同，Go 中的泛型是有类型约束的，它可以限制参数的类型范围。Go 中的这种约束是通过扩展接口的能力实现的。
- 当前 Go 的泛型语法还有诸多限制，后续可能会放开。

要准确地理解泛型这个新概念，我们需要明确一些概念在中英文中的差异，这样才有进一步讨论的基础。我们以下面这个简单的泛型变量 HashMap 为例，HashMap 声明了一个带泛型的哈希表。其中 Key 可以是 int 或者 string，而 Value 必须是 string 或者 float。

复制代码

```
1 type hashmap [KEY int|string, VALUE string|float64] map[KEY]VALUE
```

在这个例子中，[]括号中的这一串叫做类型参数列表（Type Parameter List），类型参数列表是由多个类型参数组成的，中间用逗号隔开。这个例子中的 Key 与 Value 被叫做类型参数（Type Parameter），可以取任意的名字。它们后方的 int|string 与 string|float64 叫做类型约束（Constraints）。



此外，学习泛型时，还有几个重要知识点，分别是：

1. 泛型的声明
2. 泛型的类型约束
3. 泛型的实例化
4. 泛型的自动类型推断
5. 泛型类型参数的操作与限制
6. 泛型类型的转换

下面我们挨个来介绍一下这些知识点。

泛型的声明

我们先来看泛型的声明。泛型可以用在切片、哈希表、结构体、方法、函数、接口等类型中，我们先来看看泛型在各个类型中的表现，就能对泛型的声明方式有大体的了解。

- 泛型切片的声明。

 复制代码

```
1 type Slice1 [T int|float64|string] []T
```

该泛型切片可以指代下面的 3 种切片类型。

复制代码

```
1 type SliceInt []int
2 type SliceFloat []float64
3 type SliceString []string
```

- 泛型哈希表的声明。

复制代码

```
1 type Map1 [KEY int|string, VALUE string|float64] map[KEY]VALUE
```

该泛型哈希表可以指代下面的 4 种切片类型。

复制代码

```
1 type Map2 map[int]string
2 type Map3 map[int]float64
3 type Map4 map[string]string
4 type Map5 map[string]float64
```

- 泛型结构体的声明。

复制代码

```
1 type Struct1 [T string|int|float64] struct {
2     Title string
3     Content T
4 }
```

该泛型结构体可以指代下面的 3 种类型。

复制代码

```
1 // 结构体
2 type Struct3 struct {
3     Title string
4     Content string
```



```

5 }
6
7 type Struct4 struct {
8     Title string
9     Content int
10 }
11
12 type Struct5 struct {
13     Title string
14     Content float64
15 }

```



- 泛型方法的声明。

下面的泛型方法可以灵活地对任意类型的 **Data** 进行加锁，并执行 **f** 函数。

复制代码

```

1 type Lockable[T any] struct {
2     mu sync.Mutex
3     data T
4 }
5
6 func (l *Lockable[T]) Do(f func(*T)) {
7     l.mu.Lock()
8     defer l.mu.Unlock()
9     f(&l.data)
10 }

```

- 泛型函数的声明。

下面的函数 **NoDiff** 可以判断可变长度数组中的每一个元素是不是都是相同的。

复制代码

```

1 // 函数
2 func NoDiff[V comparable](vs ...V) bool {
3     if len(vs) == 0 {
4         return true
5     }
6
7     v := vs[0]
8     for _, x := range vs[1:] {
9         if v != x {
10             return false
11         }
12     }
13     return true
14 }

```

- 泛型接口的声明。

下面的泛型接口可以指代不同类型的方法。

[复制代码](#)

```
1 type MyInterface[T int | string] interface {  
2     WriteOne(data T) T  
3     ReadOne() T  
4 }
```

从上面的例子可以看出，不同的泛型声明都有相似的表示形式。无外乎在[]中添加类型参数，然后为类型参数加上类型约束，从而利用类型参数指代多种不同的类型。下面我们来看一看类型约束。

泛型的类型约束

正如之前看到的，Go 可以为类型参数加上约束。其实我们可以将约束视为类型的类型，这种约束使类型参数成为了一种类型的集合，这个集合可能很大也可能很小。

Go 引入了一些新的符号来表示类型的约束，我们具体来看一看。

- ~T 表示一个类型集，它包括所有基础类型为 T 的类型。

[复制代码](#)

```
1 ~int  
2 ~[]byte  
3 ~map[int]string  
4 ~chan struct{}  
5 ~struct{x int}
```

举一个例子，type flag int 中的 flag 是一个新的类型，但是它的基础类型仍然是 int。

golang.org/x/exp/constraints 库也为我们预置了几个类型：Complex、Float、Integer、Ordered、Signed、Unsigned。以 Ordered 为例，它本身的定义如下，它约束了类型必须可以进行 <、<=、>=、> 等比较操作。

[复制代码](#)

```
1 type Ordered interface {
2     Integer | Float | ~string
3 }
```



天下无鱼

<https://shikey.com/>

- **Comparable** 为 Go 中的预定义类型，约束了类型可以进行等于和不等于是的判断（`==`、`!=`）。
- **Any** 也是 Go 中的预定义类型，它其实就是空接口的别名，在 Go 源码中，已经将所有空接口都替换为了 **Any**。
- **T1 | T2 | ... | Tn** 表示集合类型，它指的是所有这些类型的并集，**Tn** 可以是上方的 `~T` 类型、基础类型或者是类型名。

复制代码

```
1 uint8 | uint16 | uint32 | uint64
2 ~[]byte | ~string
3 map[int]int | []int | [16]int | any
4 chan struct{} | ~struct{x int}
```

类型约束是通过接口来实现的，Go1.18 放宽了接口的定义，可以在接口中包含类型的集合，如下所示。

复制代码

```
1 type L interface {
2     Run() error
3     Stop()
4 }
5
6 type M interface {
7     L
8     Step() error
9 }
10
11 type N interface {
12     M
13     interface{ Resume() }
14     ~map[int]bool
15     ~[]byte | string
16 }
17
18 type O interface {
19     Pause()
20     N
21     string
```

```
22     int64 | ~chan int | any
23 }
```



天下无鱼

<https://shikey.com/>

接口中用空行分割的类型则标识了类型的交集。在下面这个例子中，约束 `generic` 是类型 `int | ~float64` 与 `float64` 的交集，所以 `generic` 只能够代表 `float64` 类型。所以下例中尝试使用 `int` 类型实例化时，编译会报错。

 复制代码

```
1 type generic interface {
2     int | ~float64
3     float64
4 }
5
6 func Sum[T generic](numbers []T) T {
7     var total T
8     for _, x := range numbers {
9         total += x
10    }
11    return total
12 }
13
14 // int does not implement generic (int missing in float64)
15 func main() {
16     xs := []int{3, 5, 10}
17     total := Sum(xs)
18     fmt.Println(total)
19 }
```

另外，约束虽然靠的是扩展后的接口，但书写却可以简化，像下面两个类型参数列表就是等价的。

 复制代码

```
1 [X interface{string|[]byte}, Y interface{~int}]
2 [X string|[]byte, Y ~int]
```

泛型的实例化

泛型类型必须被实例化才能够使用。和声明一样，让我们看看各个泛型类型是如何被实例化的。

• 泛型切片实例化



```
1 type Slice1 [T int|float64|string] []T
2 var MySlice1 Slice1[int] = []int{1,2,3}
3 var MySlice3 Slice1[string] = []string{"hello", "small", "yang"}
4 var MySlice5 Slice1[float64] = []float64{1.222, 3.444, 5.666}
```

• 泛型 Map 实例化

复制代码

```
1 type Map1[KEY int | string, VALUE string | float64] map[KEY]VALUE
2
3 var MyMap1 Map1[int, string] = map[int]string{
4     1: "hello",
5     2: "small",
6 }
7
8 var MyMap3 Map1[string, string] = map[string]string{
9     "one": "hello",
10    "two": "small",
11 }
```

• 泛型结构体实例化

复制代码

```
1 type Aticle [T string|int|float64] struct {
2     Title string
3     Content T
4 }
5
6 var s = Aticle[string]{
7     Title: "hello",
8     Content: "small",
9 }
10
11 // 复杂结构体的实例化
12 type MyStruct[S int | string, P map[S]string] struct {
13     Name string
14     Content S
15     Job P
16 }
17
18 var MyStruct1 = MyStruct[int, map[int]string]{
```

```
19 Name:      "small",
20 Content: 1,
21 Job:      map[int]string{1: "ss"},
22 }
```



天下无鱼
<https://shikey.com/>

• 泛型函数实例化

复制代码

```
1 // 函数实例化
2 package main
3 func NoDiff[V comparable](vs ...V) bool {
4     if len(vs) == 0 {
5         return true
6     }
7
8     v := vs[0]
9     for _, x := range vs[1:] {
10         if v != x {
11             return false
12         }
13     }
14     return true
15 }
16
17 func main() {
18     var NoDiffString = NoDiff[string]
19     println(NoDiffString("Go", "go")) // false
20     println(NoDiff[int](123, 123, 789)) // false
21 }
22 // 函数实例化, 例子2
23 type Ordered interface {
24     ~int | ~uint | ~int8 | ~uint8 | ~int16 | ~uint16 |
25     ~int32 | ~uint32 | ~int64 | ~uint64 | ~uintptr |
26     ~float32 | ~float64 | ~string
27 }
28
29 func Max[S ~[]E, E Ordered](vs S) E {
30     if len(vs) == 0 {
31         panic("no elements")
32     }
33
34     var r = vs[0]
35     for i := range vs[1:] {
36         if vs[i] > r {
37             r = vs[i]
38         }
39     }
40     return r
41 }
```

```

42 type Age int
43 var ages = []Age{99, 12, 55, 67, 32, 3}
44
45 var langs = []string {"C", "Go", "C++"}
46
47 func main() {
48     var maxAge = Max([]Age, Age)
49     println(maxAge(ages)) // 99
50
51     var maxStr = Max([]string, string)
52     println(maxStr(langs)) // Go
53 }
54

```



• 泛型方法实例化

复制代码

```

1 package main
2
3 import "sync"
4
5 type Lockable[T any] struct {
6     mu sync.Mutex
7     data T
8 }
9
10 func (l *Lockable[T]) Do(f func(*T)) {
11     l.mu.Lock()
12     defer l.mu.Unlock()
13     f(&l.data)
14 }
15
16 func main() {
17     var n Lockable[uint32]
18     n.Do(func(v *uint32) {
19         *v++
20     })
21
22     var f Lockable[float64]
23     f.Do(func(v *float64) {
24         *v += 1.23
25     })
26
27     var b Lockable[bool]
28     b.Do(func(v *bool) {
29         *v = !*v
30     })
31
32     var bs Lockable[[]byte]
33     bs.Do(func(v *[]byte) {

```

```

34     *v = append(*v, "Go"... )
35 })
36 }
37
38 // 方法实例化, 例子2
39 type Number interface{
40     int | int32 | int64 | float64 | float32
41 }
42
43 //定义一个泛型结构体, 表示堆栈
44 type Stack[V Number] struct {
45     size int
46     value []V
47 }
48
49 //加上Push方法
50 func (s *Stack[V]) Push(v V) {
51     s.value = append(s.value, v)
52     s.size++
53 }
54
55 //加上Pop方法
56 func (s *Stack[V]) Pop() V {
57     e := s.value[s.size-1]
58     if s.size != 0 {
59         s.value = s.value[:s.size-1]
60         s.size--
61     }
62     return e
63 }
64
65 //实例化成一个int型的结构体堆栈
66 s1 := &Stack[int]{}
67
68 //入栈
69 s1.Push(1)
70 s1.Push(2)
71 s1.Push(3)
72 fmt.Println(s1.size, s1.value) // 3 [1 2 3]
73
74 //出栈
75 fmt.Println(s1.Pop()) //3
76 fmt.Println(s1.Pop()) //2
77 fmt.Println(s1.Pop()) //1

```



• 泛型接口实例化

复制代码

```
1 type MyInterface[T int | string] interface {
```



```

2   WriteOne(data T) T
3   ReadOne() T
4 }
5
6 type Note struct {
7
8 }
9
10 func (n Note) WriteOne(one string) string {
11     return "hello"
12 }
13
14 func (n Note) ReadOne() string {
15     return "small"
16 }
17
18 var one MyInterface[string] = Note{}
19 fmt.Println(one.WriteOne("hello"))
20 fmt.Println(one.ReadOne())

```



从上面泛型实例化的例子中我们可以看出，在实际使用泛型时，我们需要在[]中明确泛型的具体类型，这样编译器就能够为我们生成具体类型的函数或者类型了。不过，每一次都需要在[]中指定类型还是比较繁琐的，好在借助编译器的自动推断能力，我们就可以简化泛型实例化的书写方式。

泛型的自动类型推断

以泛型函数实例化为例，借助编译时的自动类型推断，泛型函数的调用可以像调用正常的函数一样自然。

复制代码

```

1 // 函数实例化
2 package main
3 func NoDiff[V comparable](vs ...V) bool {
4     if len(vs) == 0 {
5         return true
6     }
7
8     v := vs[0]
9     for _, x := range vs[1:] {
10         if v != x {
11             return false
12         }
13     }
14     return true
15 }

```

```

16 func main() {
17     println(NoDiff("Go", "Go", "Go")) // true,自动推断
18     println(NoDiff[string]("Go", "go")) // false
19
20     println(NoDiff(123, 123, 123, 123)) // true, 自动推断
21     println(NoDiff[int](123, 123, 789)) // false
22
23     type A = [2]int
24     println(NoDiff(A{}, A{}, A{})) // true, 自动推断
25     println(NoDiff(A{}, A{}, A{1, 2})) // false,自动推断
26
27     println(NoDiff(new(int))) // true,自动推断
28     println(NoDiff(new(int), new(int))) // false,自动推断
29 }
30

```



泛型类型参数的操作与限制

在使用泛型的过程中，我们还不可避免地会遇到一个问题，那就是在对类型参数进行操作时，哪些操作是有效的，哪些是无效的？下面我们来看一些类型参数允许的重要操作。

- 类型断言

类型参数本质是扩展了接口的能力实现的，因此它仍然可以进行类型的断言，判断实际的类型以给出不同的操作。

复制代码

```

1  import "fmt"
2
3  func nel[T int | string](x any) {
4      if v, ok := x.(T); ok {
5          fmt.Printf("x is a %T\n", v)
6      } else {
7          fmt.Printf("x is not a %T\n", v)
8      }
9  }
10
11 func wua[T int | string](x any) {
12     switch v := x.(type) {
13     case T:
14         fmt.Println(v)
15     case int:
16         fmt.Println("int")
17     case string:
18         fmt.Println("string")
19     }

```

• 核心类型限制

由于类型参数是类型的集合，原则上只有当类型参数中的所有类型都可以执行这个操作时，才被认为是有效的。下面这类操作在编译时就会直接报错，因为 `Any` 是所有类型的集合，但是并不是所有类型都可以进行加法操作。

[复制代码](#)

```
1 // invalid operation: operator + not defined on total (variable of type T) const
2 func Sum[T any](numbers []T) T {
3     var total T
4     for _, x := range numbers {
5         total += x
6     }
7     return total
8 }
```

此外，即便是类型参数中的所有类型都可以执行的操作，在 `Go` 中也有一些限制。例如，对于下面的函数类型，调用时必须具有相同的核心类型，否则会在编译时报错。一般来说，如果一个类型参数的所有类型都共享一个相同的底层类型，这个相同的底层类型就被称为类型参数的核心类型。

[复制代码](#)

```
1 func foo[F func(int) | func(any)] (f F, x int) {
2     f(x) // error: invalid operation: cannot call non-function f
3 }
4
5 func bar[F func(int) | func(int)int] (f F, x int) {
6     f(x) // error: invalid operation: cannot call non-function f
7 }
8
9 type Fun func(int)
10
11 func tag[F func(int) | Fun] (f F, x int) {
12     f(x) // okay
13 }
```

执行切片的截取操作时，类型参数也必须具有相同的核心类型。

```

1 func foo[T []int | [2]int](c T) {
2     _ = c[:] // invalid operation: cannot slice c: T has no core type
3 }
4
5 func bar[T [8]int | [2]int](c T) {
6     _ = c[:] // invalid operation: cannot slice c: T has no core type
7 }

```



同样的道理也存在于 `for-range` 循环中。

```

1 func values[T []E | map[int]E, E any](kvs T) []E {
2     r := make([]E, 0, len(kvs))
3     // error: cannot range over kvs (T has no core type)
4     for _, v := range kvs {
5         r = append(r, v)
6     }
7     return r
8 }

```

泛型类型的转换

类型参数还可以完成类型的转换操作。例如，要想让 **From** 类型顺利转换为 **To** 类型，必须确保 **From** 中的所有类型都能够强制转换为 **To** 中的所有类型，只有这样操作才是有效的。

```

1 func pet[A ~int32 | ~int64, B ~float32 | ~float64](x A, y B){
2     x = A(y)
3     y = B(x)
4 }
5
6 func dig[From ~byte | ~rune, To ~string | ~int](x From) To {
7     return To(x)
8 }
9
10 func cov[V ~[]byte | ~[]rune](x V) string {
11     return string(x)
12 }
13
14 func voc[V ~[]byte | ~[]rune](x string) V {
15     return V(x)
16 }
17
18 func eve[X, Y int | string](x X) Y {

```

```
19     return Y(x) // error
20 }
```



天下无鱼

<https://shikey.com/>

相似的情况还出现在内置的 `len` 函数、`make` 函数、`close` 函数中，这里就不一一赘述了。

除了转换，当前 Go 泛型中还有一个重要的限制，就是方法的接受者可以有类型参数，但是方法中不能够有类型参数。例如下面的代码在编译时会直接报错。

复制代码

```
1 type MyThing[T any] struct {
2     value T
3 }
4
5 // syntax error: method must have no type parameters
6 func (m MyThing[T]) Perform[R any](f func(T) R) R {
7     return f(m.value)
8 }
9
10 func main() {
11     fmt.Println("Hello, playground")
12 }
```

总结

Go1.18 之后，我们期待已久的泛型终于问世了。泛型使 Go 能够书写更加简单、抽象的代码。但泛型通常也是成本与收益之间的权衡。

Go 的泛型只存在于编译时，它以增加少量编译时间为代价，换来了更方便的代码书写。泛型的使用方法需要重点考虑几个问题，即泛型的声明、泛型的约束、泛型的实例化和泛型的操作。

Go 泛型通过扩展接口的能力实现了类型的集合与约束，但是当前 Go 中的泛型仍然有不少的限制。在实际使用时，我建议你像往常一样书写代码，当真正需要代码抽象时替换为泛型也不迟。

分享给需要的人，Ta 购买本课程，你将得 20 元

生成海报并分享

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 49 | 服务治理：如何进行限流、熔断与认证？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。