

Go 1.18 中的泛型

欧长坤

changkun.de/s/generics118

2022/03/30



语法和使用

什么时候需要泛型？

当使用接口作为函数的形参类型时，函数调用方传递的实际参数可以是完全不同的类型：

```
type T interface {
    Add(T) T
}

func Sum(elems ...T) (sum T) { // T 可以是任何实现 Add() 的类型
    if len(elems) == 0 { return }
    sum = elems[0]
    for _, v := range elems[1:] { sum = sum.Add(v) }
    return
}
```

当使用类型参数作为函数的形参类型时，函数调用方传递的实际参数必须是满足类型参数所约束的类型：

```
func GenericSum[S ~int](elems ...S) (sum S) { // S 的底层类型必须底层类型为 int 约束的类型
    for i := range elems { sum += elems[i] }
    return
}
```

使用泛型的根本目的是：类型安全的参数传递，以及对实现的类型进行抽象



具有类型参数 (Type Parameter) 的签名

类型参数 类型集(约束)

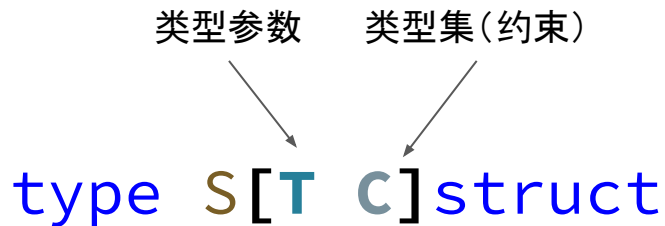
func **F**[**T** **C**] (**v** **T**) (**T**, **error**)

类型参数列表 普通参数列表 返回值列表



类型参数 类型集(约束)

type **S**[**T** **C**] struct

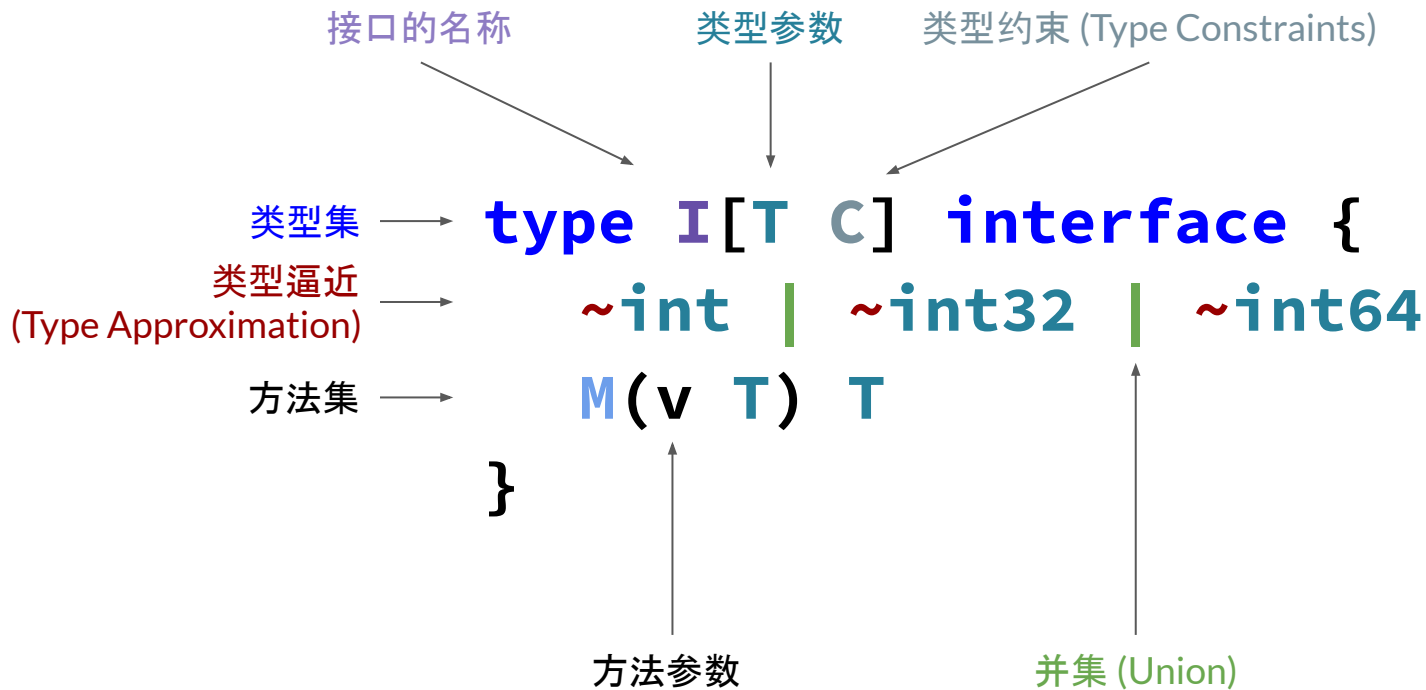


类型参数的声明紧随: 1) 函数名之后, 或者 2) 类型名之后。类型参数通过类型集进行约束。



类型集 (Type Set) 的定义

类型集本质上就是接口，类型集可以作为类型参数的约束，一个接口也可以具有类型参数。



例1: 对实现的类型进行抽象

将 Map 和 Reduce 函数的类型进行抽象:

```
func Map[T1, T2 interface{}](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 interface{}](s []T1, init T2, f func(T2, T1) T2) T2 {  
    r := init  
    for _, v := range s {  
        r = f(r, v)  
    }  
    return r  
}
```



例2: 精简调用代码

使用 `sort.Sort` 必须实现冗余的接口约束:

```
type wrapSort[T interface{}] struct {  
    s []T  
    cmp func(T, T) bool  
}  
  
func (s wrapSort[T]) Len() int           { return len(s.s) }  
func (s wrapSort[T]) Less(i, j int) bool { return s.cmp(s.s[i], s.s[j]) }  
func (s wrapSort[T]) Swap(i, j int)      { s.s[i], s.s[j] = s.s[j], s.s[i] }  
  
func Sort[T interface{}](s []T, cmp func(T, T) bool) {  
    sort.Sort(wrapSort[T]{s, cmp})  
}
```



更多例子

channel 的统一抽象 (golang.design/x/chann):

```
type Chann[T any] struct  
type Opt func(*config)  
func New[T any](opts ...Opt) *Chann[T]  
func Cap(n int) Opt
```

类型安全的 DeepCopy (golang.design/x/reflect):

```
func DeepCopy[T any](src T) (dst T)
```

等等参见 github.com/golang-design/go2generics



类型参数的声明和化简

// Pick returns a randomly selected element in a given slice.

```
func Pick[S interface{ ~[]Elem }, Elem interface{}](s S) Elem
```



类型参数的声明和化简

// Pick returns a randomly selected element in a given slice.

```
func Pick[S interface{ ~[]Elem }, Elem interface{}](s S) Elem
```

// =>

```
func Pick[S ~[]Elem, Elem interface{}](s S) Elem
```



类型参数的声明和化简

// Pick returns a randomly selected element in a given slice.

```
func Pick[S interface{ ~[]Elem }, Elem interface{}](s S) Elem
```

// =>

```
func Pick[S ~[]Elem, Elem interface{}](s S) Elem
```

// =>

```
func Pick[S ~[]Elem, Elem any](s S) Elem ← 从 1.18 开始, any 是 interface{} 的一个别名
```

// =>

```
func Pick[S ~[]any](s S) any ← 不推荐简化到这种情况
```



类型参数的声明和化简

// Pick returns a randomly selected element in a given slice.

```
func Pick[S interface{ ~[]Elem }, Elem interface{}](s S) Elem
```

// =>

```
func Pick[S ~[]Elem, Elem interface{}](s S) Elem
```

// =>

```
func Pick[S ~[]Elem, Elem any](s S) Elem ← 从 1.18 开始, any 是 interface{} 的一个别名
```

// =>

```
func Pick[S ~[]any](s S) any ← 不推荐简化到这种情况
```



ianlancetaylor commented on Aug 27, 2019

Member ...

If we are able to add generics to the language, then I suspect there will be many fewer cases where people write `interface{}`, so the alias would have correspondingly less value. Putting on hold pending a decision on generics.

👍 20



<https://go.dev/issue/33232#issuecomment-525489884>



接口的位置

当接口作为类型参数或者具体的参数时，表达的含义不一样，这也是最可能使人困惑的地方

```
func foo[T any](x T) T { return x }  
foo("string") // 返回值为 string 类型而非 interface{}
```

当接口开始包含类型集时，无法作为具体的参数使用：

```
type Ia[T any] interface{ *T }  
type Ib[T any] interface{ Foo() }  
  
func bar(T Ia[int]) {} // ERROR: interface contains type constraints  
func bar(T Ib[int]) {} // OK
```

为什么第二个 bar 不会报错？



抽象能力

泛型没有任何运行时的机制, 所以关于泛型的组件只发生在编译时期

对于下面的接口而言, 无法作为普通参数使用:

```
func Foo[T any]() {}
```

```
x := Foo          // ERROR: cannot use generic function Foo without instantiation
```

```
y := Foo[int]    // OK
```



什么时候(不)应该使用类型参数？

当函数的实现与参数的类型不强相关时，可以考虑使用

实现通用数据结构时，可以考虑使用

性能不应该是使用类型参数的理由，如果使用类型参数能够提高代码的使用场景和阅读的清晰度，则可以考虑使用

如果一个方法的实现对于不同类型都不相同，则不应该考虑使用类型参数

如果既可以用类型参数也可以用纯接口参数，则不应该考虑使用类型参数

```
func foo[T fmt.Stringer](t T) string {  
    return t.String()  
}  
  
func foo(t fmt.Stringer) string { // 更好  
    return t.String()  
}
```



其他新增内容

标准库的变化

几个新增的(准)标准库:

- golang.org/x/exp/constraints
- golang.org/x/exp/maps
- golang.org/x/exp/slices
- [go/](#)*

其中最常用的(应该是):

`package constraints`

```
// Ordered is a constraint that permits any ordered type: any type that supports  
// the operators < <= >= >.
```

```
type Ordered interface {
```

```
    ~int | ~int8 | ~int16 | ~int32 | ~int64 | // Signed
```

```
    ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr | // Unsigned
```

```
    ~float32 | ~float64 | // Float
```

```
    ~string
```

```
}
```



核心类型

接口可以嵌入类型集, 从而满足一个接口(约束)可以包含多种不同的类型. 如果一个接口只能约束一种底层类型(Underlying Type), 则这个接口称之为[核心类型](#)(Core Type).

普通类型的底层类型(也可称核心类型):

```
type X float32
type Y float32
```

接口类型的核心类型:

```
type U interface {
    *int
    String() string
}
```

```
type V interface { ~float32 }
type W interface { int | float64 }
```

U 的核心类型是 `*int`, V 的核心类型是 `float32`, W 没有核心类型.



对类型使用运算符

可以进行比较 (==, !=) 的类型允许使用 comparable 类型进行约束 (非常特殊, 有无穷多种类型可以进行比较, 其他运算符都是可穷举的):

```
// comparable is an interface that is implemented by all comparable types
// (booleans, numbers, strings, pointers, channels, arrays of comparable types,
// structs whose fields are all comparable types).
// The comparable interface may only be used as a type parameter constraint,
// not as the type of a variable.
```

```
type comparable interface{ /*compiler magic*/ }
```

```
// Index returns the index of the first occurrence of v in s,
// or -1 if not present.
```

```
func Index[E comparable](s []E, v E) int {
    for i, vs := range s {
        if v == vs { return i }
    }
    return -1
}
```



对类型使用运算符(续)

可以进行排序 (<, <=, >, >=) 的类型允许使用 constraints.Ordered 接口类型进行约束 (只有有限多个类型可以使用排序符号):

```
// IsSorted reports whether x is sorted in ascending order.
```

```
func IsSorted[E constraints.Ordered](x []E) bool {  
    for i := len(x) - 1; i > 0; i-- {  
        if x[i] < x[i-1] { return false }  
    }  
    return true  
}
```



为什么不能用运算符方法？

新增的 comparable 看起来很别扭，为什么不直接使用运算符方法？例如：

```
type Comparable[T any] interface {  
    ==(T) bool  
}
```

主要原因是可比较性本质上应该被定义为

```
type Comparable[T any] interface {  
    ==(T) untyped bool  
    != (T) untyped bool  
}
```

而且比较对象的类型无法定义为自身类型。



部分进阶话题

应用场景
理论基础
限制及原因

部分使用场景和解决方法

返回满足接口约束的具体类型

编写一个获得返回不同类型的方法：

```
// Want returns a pointer that points to a value of the specified type.
```

```
// Usage:
```

```
// Want[A]() // returns *A
```

```
// Want[B]() // returns *B
```

```
func Want[T interface{ Foo() }]() (x *T) {
```

```
    switch any(x).(type) {
```

```
    case A:
```

```
        return any(&A{}).(*T)
```

```
    case B:
```

```
        return any(&B{}).(*T)
```

```
    default:
```

```
        panic("unsupported")
```

```
    }
```

```
}
```

```
type A struct{}
```

```
func (a A) Foo() {}
```

```
type B struct{}
```

```
func (b B) Foo() {}
```



类型 T 没有实现接口 I

具有指针接收者方法的类型必须使用指针类型作为类型参数:

```
type A struct{}  
func (a A) Foo() {}  
func (a A) Bar() {}  
type B struct{}  
func (b B) Foo() {}  
func (b *B) Bar() {}
```

```
type C[T any] interface {  
    Foo()  
    Bar()  
}
```

```
func Want[T C]() (x T) { return }
```

```
func main() {  
    Want[A]() // OK  
    Want[B]() // ERROR: B does not implement C (Bar method has  
pointer receiver)  
}
```



对应的解决方案: 将指针指定为约束的核心类型

```
type A struct{}
```

```
func (a A) Foo() {}
```

```
func (a A) Bar() {}
```

```
type B struct{}
```

```
type C[T any] interface {
```

```
    *T
```

```
    Foo()
```

```
    Bar()
```

```
}
```

```
func (b B) Foo() {}
```

```
func (b *B) Bar() {}
```

```
// Guarantee U must be a pointer.
```

```
func Want[U any, V C[U]](x V) {
```

```
    return
```

```
}
```

```
func main() {
```

```
    a := Want[A]()
```

```
    b := Want[B]()
```

```
    fmt.Printf("%T, %T\n", a, b) // *A, *B
```

```
}
```



强制转换为类型参数

符合类型约束的类型不能被强转为类型参数

```
func Foo[T int]() T {  
    x := 42  
    return T(x)    // OK  
}
```

```
func Bar[T int]() T {  
    x := 42  
    return int(x) // ERROR: cannot use int(x) (value of type int) as type T in return statement  
}
```



无类型常量的类型推导

math.MaxFloat32 是 float64 类型:

```
type Vec2[T ~float32 | ~float64] struct {
    X, Y T
}

func NewV[T ~float32 | ~float64](x, y T) Vec2[T] {
    return Vec2[T]{x, y}
}

func main() {
    v := NewV(math.MaxFloat32, math.MaxFloat32)
    switch (any)(v).(type) {
    case Vec2[float32]:
    case Vec2[float64]:
        panic("?") // 会执行 panic
    }
}
```

```
package math
```

```
const MaxFloat32 = 0x1p127 * (1 + (1 - 0x1p-23))
```

go.dev/ref/spec: [...] The default type of an untyped constant is bool, rune, int, **float64**, complex128 or string respectively, depending on whether it is a boolean, rune, integer, floating-point, complex, or string constant.



对一个编程范式的破坏

考虑下面的代码是否适合使用泛型？

```
type Window struct {
    cfg1, cfg2 any
}

func NewWindow(opts ...Option) *Window {
    w := &Window{}
    for _, opt := range opts {
        opt(w)
    }
    return w
}
```

```
type Option func(w *Window)

func Config1(cfg1 any) Option {
    return func(w *Window) {
        w.cfg1 = cfg1
    }
}

func Config2(cfg2 any) Option {
    return func(w *Window) {
        w.cfg2 = cfg2
    }
}
```



底层细节

一些关键的实现算法

类型推断使用的方法是 类型合一 (Unification by Substitution):

- https://go.dev/ref/spec#Type_inference
- <https://github.com/golang/go/blob/go1.18/src/cmd/compile/internal/types2/infer.go>
- <https://github.com/golang/go/blob/go1.18/src/go/types/unify.go>

泛型的实例化通过字典和 Gcshape 模板:

- <https://go.dev/design/generics-implementation-dictionaries-go1.18>
- <https://go.dev/design/generics-implementation-gcshape>

类型集的计算:

- <https://github.com/golang/go/blob/go1.18/src/cmd/compile/internal/types2/instantiate.go#L156>

我们不考虑具体的实现细节, 有兴趣可以自行阅读



限制

comparable 的困境

Issue [#49587](#), [#50646](#), [#50791](#), [#51257](#), [#51338](#)

运行下段代码会输出什么？

```
func Equal[T comparable](v1, v2 T) bool {  
    return v1 == v2  
}
```

```
func main() {  
    v1 := interface{}(func() {})  
    v2 := interface{}(func() {})  
    Equal(v1, v2)  
}
```



comparable 的困境

Issue [#49587](#), [#50646](#), [#50791](#), [#51257](#), [#51338](#)

运行下段代码会输出什么？

```
func Equal[T comparable](v1, v2 T) bool {  
    return v1 == v2  
}
```

```
func main() {  
    v1 := interface{}(func() {})  
    v2 := interface{}(func() {})  
    Equal(v1, v2) // ERROR: interface{} does not implement comparable  
}
```

go.dev/ref/spec: “**Interface values are comparable**. Two interface values are equal if they have **identical** dynamic types and equal dynamic values or if both have value `nil`.”



comparable 的困境

Issue [#49587](#), [#50646](#), [#50791](#), [#51257](#), [#51338](#)

Go 1.18 之前: 接口类型的值是可比较的

type parameter 提案对 comparable 的定义: 所有可比较类型的集合 (The type set of the comparable constraint is the set of all comparable types.) 因此空接口 (应该) 属于 comparable, 但实际上不是.

如果允许空接口属于 comparable, 下面代码会发生什么?

```
func Equal[T comparable](v1, v2 T) bool {  
    return v1 == v2  
}
```

```
func main() {  
    v1 := interface{}(func() {})  
    v2 := interface{}(func() {})  
    Equal(v1, v2) // 如果不产生编译错误, 则会引发运行时错误 panic: runtime error: comparing  
uncomparable type func()  
}
```



comparable 的困境

Issue [#49587](#), [#50646](#), [#50791](#), [#51257](#), [#51338](#)

在 Go1.18 的 spec 中, comparable 的定义:

The `predeclared interface type` `comparable` denotes the set of all **non-interface types** that are `comparable`. Specifically, a type `T` implements `comparable` if:

- `T` is not an interface type and `T` supports the operations `==` and `!=`; or
- `T` is an interface type and each type in `T`'s type set implements `comparable`.

这也导致了我们无法写出这样的代码:

```
type P map[any]struct{}           // OK
type R[T comparable] map[T]struct{} // OK
// type Q[T any] map[T]struct{}    // ERROR: incomparable map key type T
```



还不允许方法上的类型参数

目前还不允许这种写法

```
type X[U any] struct {  
    u U  
}
```

```
func (x X[U]) Foo(v any) {} // OK
```

```
func (x X[U]) Bar[V any](v V) {} // ERROR: methods cannot have type parameters
```

主要原因是一个潜在的语言设计问题(运行时的类型断言), 例如:

```
func f(x any) {  
    if _, ok := x.(interface{ Bar(int) }); ok {  
        // ...  
    }  
}
```

编译器无法提前得知是否应该给 X 生成 Bar(int) 方法



类型集的本质

类型集的基本想法是设计一种能够表达一类类型的机制，这就是集合论的基本想法。设计这种机制的核心难点是保证不会出现罗素悖论。例如不加以限制的[分类公理](#)是朴素集合论中的导致矛盾的根源。

类型集可以从 ZF 系统来考虑：

外延公理：两个类型集相等，当且仅当他们包含的类型相同

分类公理：给出一个类型集和一个普通接口，存在同时满足他们的子集

并集公理：两个类型集可以求并集

空集公理：存在一个不满足任何类型的类型集

无穷公理：存在一个包含无穷多个类型的类型集

等等...

```
type TypeSet[T any] interface {
    *T
}
type Iface interface {
    Foo()
}

type Subset[T any] interface {
    TypeSet[T]
    Iface
}
```



类型集的困境

检查类型是否满足接口描述的类型集, 是一个典型的 [可满足性问题\(SAT\)](#):

```
type Constraint interface {  
    ConstraintA | ConstraintB // 并集  
    ConstraintMethodC()      // 交集  
}
```

这个问题是一个 NP 完全问题([Cook-Levin 定理](#))。

编译器在编译期间执行这类检查, 如果不对规则加以限制, 则将在某些情况下极大的增加 编译时间。

这最终导致了 "并集元素中不能包含具有方法集的参数 类型" 这一限制。

Go 1.18 中的类型集设计并不是完备的, 即某些类型集无法表示。

例如不可能写出一个 类型集来涵盖字符串或者可以字符串化的 类型的总和:

```
type Stringish interface {  
    ~string | fmt.Stringer // ERROR: cannot use fmt.Stringer in union  
}
```



不支持变长类型参数

目前的设计不支持(好像也很难支持)变长参数的写法, 从而导致无法编写变长元组(如果有 []any 似乎也不需要), 如果一定需要, 做边界检查(bound check)也是一个挑战:

```
type Tuple[Ts ...comparable] struct {
    elements ...Ts
}

func (t *Tuple[Ts...]) Set(es ...Ts) {
    t.elements[Ts...]{es...}
}

func (t Tuple) PirntAll() {
    for _, e := range t.elements {
        fmt.Println(e)
    }
}

// func (t Tuple[Ts...]) Get(i int) T???
```



更多限制

泛型函数方法中不能定义类型 <https://go.dev/issue/47631>

不能访问类型的结构字段 <https://go.dev/issue/49030#issuecomment-954336867>



更加高层的泛型机制同样不存在

特化

元编程

柯里化

非类型类型参数

运算符方法

...



结论

类型参数更广泛的支持了基于接口的参数化多态

类型集使用了公理化集合论(更特别一点类型论)的方法扩展了接口的定义实现了类型约束

类型推导和类型合一简化了泛型的使用

但是

Go 目前的泛型(1.18, 但很可能在 1.20 之前都)还比较基础且限制较多(也不排除永远不会解除这些限制)

很多原因是权衡了编译器编译速度的结果(而非标新立异)

期待未来的改进空间

1. 取消前面提到的限制
2. 支持 parameterized aliases
3. 编译器改善



讨论

泛型在哪些基础库得到广泛应用？

泛型对于 Go 语言的简单性是一种破坏吗？

使用泛型会对性能有提升吗？

Go 的泛型设计解决了泛型的困境吗？

哪里能看到泛型下个阶段的规划？



进一步阅读的参考

一些可参考的泛型代码

<https://github.com/polyred/polyred/tree/main/math>

<https://github.com/golang-design/chann>

<https://github.com/golang-design/reflect>

<https://github.com/golang-design/go2generics>

<https://github.com/samber/lo>

官方教程 <https://go.dev/blog/intro-generics>

类型参数的提案 <https://go.dev/design/43651-type-parameters>

类型参数的理论基础 <https://arxiv.org/abs/2005.11710>

