Based on Contracts (Version July 31, 2019)

Go 2 Generics? A (P)review

Changkun Ou

https://changkun.de/s/go2generics/

https://youtu.be/E16Y6bI2S08

Go 夜读 SIG 小组 | 第 80 期

March 18, 2020

主要内容

- 泛型的起源
- 泛型的早期设计
- Go 2 的「合约」
- 上手时间
- 历史性评述
- 展望

泛型的起源

Origin of Generics

当我们谈论泛型时，我们在谈论什么？

多态是同一形式表现出不同行为的一种特性。在编程语言理论中被分为两类：

临时性多态（Ad hoc Polymorphism）**根据实参类型调用对应的版本**，仅支持数量有限的调用。也被翻译为特设多态。例如：函数重载

```
func Add(a, b int) int { return a+b }
func Add(a, b float64) float64 { return a+b } // 注意：Go 语言中不允许同名函数

Add(1, 2)      // 调用第一个
Add(1.0, 2.0) // 调用第二个
Add("1", "2") // 编译时不检查，运行时找不到实现，崩溃
```

参数化多态（Parametric Polymorphism）**根据实参类型生成不同的版本**，支持任意数量的调用。即**泛型**

```
func Add(a, b T) T{ return a+b }

Add(1, 2)                // 编译器生成 T = int 的 Add
Add(float64(1.0), 2.0) // 编译器生成 T = float64 的 Add
Add("1", "2")            // 编译器生成 T = string 的 Add
```

泛型做到了什么接口做不到的事情？

当使用 interface{} 时，a、b、返回值都可以在运行时表现为不同类型，取决于内部实现如何对参数进行断言：

```
type T interface { ... }
func Max(a, b T) T { ... } // T 是接口
```

当使用泛型时，a、b、返回值必须为同一类型，类型参数施加了这一强制性保障：

```
func Max(a, b T) T { ... } // T 是类型参数
```

泛型的总体目标就是：快且安全。在这里：

快 **意味着静态类型**

安全 **意味着编译早期的错误甄别**

泛型的早期设计

Early Designs on Generics

从 Go 1 谈起

```
package main
```

来源:
https://groups.google.com/d/msg/golang-nuts/j_3n5wAZaXw/YkOdbCppAQAJ

从 Go 1 谈起

```
1   func MaxInt(a, b int) int {
2      if a > b {
3          return a
4      }
5      return b
6   }
7   func MaxFloat64(a, b float64) float64 {
8      if a > b {
9          return a
10     }
11     return b
12  }
13  func MaxUintptr(a, b uintptr) uintptr {
14     if a > b {
15         return a
16     }
17     return b
18  }
19  ...
```

动机

- Max 是一个看似简单，实则复杂的例子
- 能否将类型作为参数进行传递？
- 如何对类型参数的行为进行检查？
- 如何支持多个相同类型的参数？
- 如何支持多个不同类型的参数？
- ……

Type Functions (2010) by Ian Lance Taylor

```
1   type Greater(t) interface {
2       IsGreaterThan(t) bool
3   }
4
5   func Max(a, b t type Greater(t)) t {
6       if a.IsGreaterThan(b) {
7           return a
8       }
9       return b
10  }
```

关键设计

- 在标识符后使用 (t) 作为类型参数的缺省值，语法存在二义性
  - 既可以表示使用类型参数 Greater(t)，也可以表示实例化一个具体类型 Greater(t)，其中 t 为推导的具体类型，如 int
  - 为了解决二义性，使用 type 进行限定：Vector(t type)
    func F(arg0, arg1 t type) t { … }
- 使用接口 Greater(t) 对类型参数进行约束，跟在 type 后修饰
- 提案还包含一些其他的备选语法：
  - generic(t) func ..
  - $t // 使用类型参数
  - t // 实例化具体类型

评述

- 确实是一个糟糕的设计
- x := Vector(t)(v0) 这是两个函数调用吗？
- 尝试借用使用 C++ 的 Concepts 对类型参数的约束

Generalized Types (2011) by Ian Lance Taylor

```
1   gen [T] type Greater interface {
2      IsGreaterThan(T) bool
3   }
4   gen [T Greater[T]] func Max(arg0, arg1 T) T {
5      if arg0.IsGreaterThan(arg1) {
6         return arg0
7      }
8      return arg1
9   }
```

```
1   gen [T1, T2] (
2      type Pair struct { first T1; second T2 }
3
4      func MakePair(first T1, second T2) Pair {
5          return &Pair{first, second}
6      }
7   ) // End of gen
```

关键设计

- 使用 gen [T] 来声明一个类型参数
- 使用接口对类型进行约束
- 使用 gen [T] (…) 来复用类型参数的名称

评述

- 没有脱离糟糕设计的命运
- gen [T] (…) 引入了作用域的概念
  - 需要缩进吗？
  - 除了注释还有更好的方式快速定位作用域的 结束吗？
- 复杂的类型参数声明

Generalized Types v2 (Oct. 2013) by Ian Lance Taylor

```
gen [T] (
    type Greater interface {
        IsGreaterThan(T) bool
    }
    func Max(arg0, arg1 T) T {
        if arg0.IsGreaterThan(arg1) { return arg0 }
        return arg1
    }
)
type Int int
func (i Int) IsGreaterThan(j Int) bool {
    return i > j
}
func F() {
    a, b := 0, Int(1)
    m := Max(a, b) // 0 先被忽略，解析 b 时确认为 Int
    if m != b { panic("wrong max") }
    ...
}
```

关键设计

- 使用 gen [T] 来声明一个类型参数
- 使用 gen [T] (…) 来传播类型参数的名称
- 使用类型推导来进行约束

评述

- 语法相对简洁了许多
- 利用类型推导的想法看似很巧妙，但能够实现吗？
- gen [T] (…) 引入了作用域的概念
  - 缩进？
  - 如何快速定位作用域在何时结束？
- 企图通过实例化过程中类型推导来直接进行约束，可能吗？
- 出现多个参数时，应该选取哪个参数进行约束？
- 如果一个类型不能进行 > 将怎么处理？
- arg0/arg1 同 T 为什么推导为不同类型？

Type Parameters (Dec. 2013) by Ian Lance Taylor

```
1   type [T] Greater interface {
2      IsGreaterThan(T) bool
3   }
4   func [T] Max(arg0, arg1 T) T {
5      if arg0.IsGreaterThan(arg1) {
6          return arg0
7      }
8      return arg1
9   }
10  type Int int
11  func (i Int) IsGreaterThan(j Int) bool {
12     return i > j
13  }
14  func F() {
15     _ = Max(0, Int(1)) // 推导为 Int
16  }
```

关键设计

- 直接在类型、接口、函数名前使用 [T] 表示类型参数
- 进一步细化了类型推导作为约束的可能性

评述

- 目前为止最好的设计
- 无显式类型参数的类型推导非常复杂
- 常量究竟应该被推导为什么类型？
- [T] 的位置很诡异，声明在左，使用在右，例如：
  - `type [T1, T2] Pair struct { … }`
  - `var v Pair[T1, T2]`

//go:generate (2014) by Rob Pike

```go
import "github.com/cheekybits/genny/generic"

// cat 201401.go | genny gen "T=NUMBERS" > 201401_gen.go

type T generic.Type

func MaxT(fn func(a, b T) bool, a, b T) T {
    if fn(a, b) {
        return a
    }
    return b
}
```

关键设计

- 通过 //go:generate 编译器指示来自动生成代码
- 利用这一特性比较优秀的实现是 cheekybits/genny

评述

- 维护成本
- 需要重新生成代码
- 没有类型检查，需要程序员自行判断

First Class Types (2015) by Bryan C. Mills

```
1   const func Max(a, b gotype) gotype {
2      switch a.(type) {
3      case int, float64, uintptr:
4          if a > b { return a}
5          return b
6      default:
7          aa, ok := a.(interface{
8              IsGreaterThan(gotype) bool
9          })
10         if !ok {
11             panic("a must implements IsGreaterThan")
12         }
13         if aa.IsGreaterThan(b) {
14             return a
15         }
16         return b
17     }
18  }
```

关键设计

- 引入 gotype 内建类型
- 扩展 .(type) 的编译期特性
- const 前缀强化函数的编译期特性
- 灵感来源 C++ SFINAE

评述

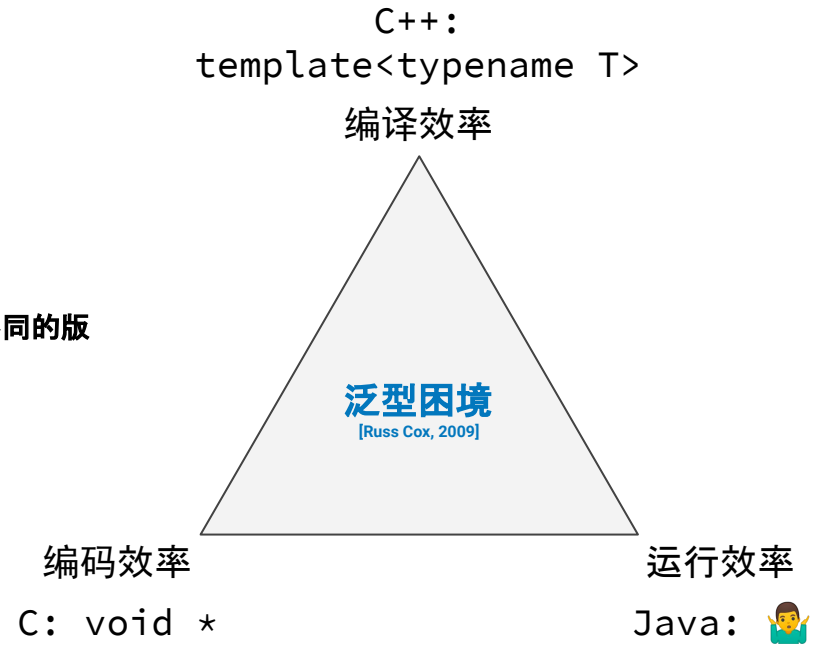
- 设计上需要额外思考 SFINAE
- 只有泛型函数的支持，泛型 结构需要通过函数来构造
- 接口二义性 interface X { Y(Z) }
  - Z 可以是类型或常量名
- 不太可能实现可类型推导

Go 2 的「合约」

"Contracts" in Go 2

Generics: Problem Overview

- 泛型从本质上是一个编译期特性
  - 「泛型困境」其实是一个**伪命题**
  - 牺牲运行时性能的做法显然不是我们所希望的
- 不加以限制的泛型机制将严重拖慢编译性能
  - **什么时候才能决定一个泛型函数应该编译多少份不同的版本？**
  - **不同的生成策略会遇到什么问题？**
- 加以限制的泛型机制将提高程序的可读性
  - **如何妥当的描述对类型的限制？**



Contracts (2018) by Ian Lance Taylor and Robert Griesemer [Revised]

```
1   contract Comparable(x T) {
2         x > x
3         x < x
4         x == x
5   }
6   func Max(type T Comparable)(v0 T, vn ...T) T {
7         switch l := len(vn); {
8         case l == 0:
9               return v0
10        case l == 1:
11              if v0 > vn[0] { return v0 }
12              return vn[0]
13        default:
14              vv := Max(vn[0], vn[1:]...)
15              if v0 > vv { return v0 }
16              return vv
17        }
18  }
```

合约是一个描述了一组类型且不会被执行的函数体。

关键设计

- 在合约中写 Go 语句对类型进行保障
- 甚至写出条件、循环、赋值语句

评述

- 复杂的合约写法（合约内的代码写法可以有多少种？）
- 「一个不会执行的函数体」太具迷惑性
- 实现上估计是一个比较麻烦的问题

Contracts (2019) by Ian Lance Taylor and Robert Griesemer [Revised]

```
1   contract Comparable(T) {
2         T int, int8, int16, int32, int64,
3         uint, uint8, uint16, uint32, uint64, uintptr,
4         float32, float64,
5         string
6   }
7   func Max(type T Comparable)(v0 T, vn ...T) T {
8         switch l := len(vn); {
9         case l == 0:
10              return v0
11        case l == 1:
12              if v0 > vn[0] { return v0 }
13              return vn[0]
14        default:
15              vv := Max(vn[0], vn[1:]...)
16              if v0 > vv { return v0 }
17              return vv
18        }
19  }
```

合约描述了一组类型的必要条件。

关键设计

- 使用方法及穷举类型来限制并描述可能的参数 类型
- comparable/arithmetic 等内建合约

评述

- 这样的代码合法吗？
  - `_ = Max(1.0, 2)`
  - 如何写出更一般的形式？
- 可变模板参数的支持情况缺失(后面会提)
- 没有算符函数、重载

- 类型参数可能出现的位置：
  - 函数 ⇒ `func F(type T C)(params ...T) T { … }`
  - 结构体 ⇒ `type S(type T C) struct { … }`
  - 接口 ⇒ `type I(type T C) interface { … }`
- 合约的形式，例：

```
contract C1(T1, T2, T3) {
      C2(T1)              // 允许与合约 C2 进行组合
      T2 int, float64     // 允许对类型 T2 进行限制
      T3 Method(T1) T1    // 允许对类型 T3 进行限制
}
```

思考

接口 Interface 是一组方法，描述了 值

合约 Contract 是一组条件，描述了 类型

- 加上类型参数的接口 —— 参数化的 `I(type T C)` 的与合约的本质区别是什么？

Contract v.s. Interface(type T) [Revised]

- 基于**合约**的参数化函数的写法：

```
1   contract Greater(T) {
2      IsGreaterThan(T) bool
3   }
4
5   func Max(type T Greater) (a, b T) T { ... }
```

- 基于**参数化接口**的参数化函数的写法：

```
1   type Greater(type T) interface {
2      IsGreaterThan(T)
3   }
4
5   func Max(type T Greater(T)) (a, b T) T { ... }
```

Contract v.s. Interface(type T)

- **合约 C(T) 的本质是**参数化接口 I(type T C) 的**语法糖**，一个更复杂的例子：

```
1   contract C(P1, P2) {
2      P1 m1(x P1)
3      P2 m2(x P1) P2
4      P2 int, float64
5   }
6
7   func F(type P1, P2 C) (x P1, y P2) P2 { ... }
```

```
1   type I1 (type P1) interface {
2      m1(x P1)
3   }
4   type I2 (type P1, P2) interface {
5      m2(x P1) P2
6      type int, float64
7   }
8   // 在实例化的过程中保障了 I2 中的 P1 与 I1 的 P1 是同一类型
9   func F(type P1 I1(P1), P2 I2(P1, P2)) (x P1, y P2) P2 { ... }
```

Summary [New]

合约语法的三种形式：

```
contract C1(T1, T2, T3) {
      C2(T1)              // 允许与合约 C2 进行组合
      T2 int, float64     // 允许对类型 T2 进行限制
      T3 Method(T1) T1    // 允许对类型 T3 进行限制
}
```

1. 内嵌式合约：`contract C1(T) { C2(T) … }`
2. 类型参数 + 实际类型：`T2 int, float64`
3. 类型参数 + 方法签名：`T3 Method(T1) T1`

进一步语法化简：也许只需要允许其中一种形式即可支持泛型？

上手时间

Hands-on!

go2go

```
git clone https://go.googlesource.com/go
git fetch "https://go.googlesource.com/go" refs/changes/17/187317/15 && git checkout FETCH_HEAD
```

```
$ go2go
用法: go2go <command> [arguments]

子命令包括:

        build
        run
        test
        translate  将 .go2 文件翻译为 .go 文件
```

包引入规则

```
import "x"

⇒ $GO2PATH/src/x
⇒ $GOPATH/src/x
```

Example 1: Generic Sort

在推出泛型后 sort 包需要被重写吗？可以，但（暂时）没必要

```
1   // sort wrapper operation -- written by Ian and modified by Changkun
2   type wrapSort(type T) struct {
3      s   []T
4      cmp func(T, T) bool
5   }
6
7   func (s wrapSort(T)) Len() int           { return len(s.s) }
8   func (s wrapSort(T)) Less(i, j int) bool { return s.cmp(s.s[i], s.s[j]) }
9   func (s wrapSort(T)) Swap(i, j int)      { s.s[i], s.s[j] = s.s[j], s.s[i] }
10
11  func Sort(type T)(s []T, cmp func(T, T) bool) {
12     sort.Sort(wrapSort(T){s, cmp})
13  }
```

Example 2: Map Reduce

试试写个 Filter？

```
1   // Map/Reduce operation -- written by Ian and modified by Changkun
2
3   // Map turns a []T1 to a []T2 using a mapping function.
4   func Map(type T1, T2)(s []T1, f func(T1) T2) []T2 {
5       r := make([]T2, len(s))
6       for i, v := range s {
7           r[i] = f(v)
8       }
9       return r
10  }
11
12  // Reduce reduces a []T1 to a single value using a reduction
    function.
13  func Reduce(type T1, T2)(s []T1, init T2, f func(T2, T1) T2) T2 {
14      r := init
15      for _, v := range s {
16          r = f(r, v)
17      }
18      return r
19  }
```

Example 3: Stack

实现大部分通用容器不需要使用合 约定义

```
1   // generic stack -- written by Ian and modified by Changkun
2   type Stack(type E) []E
3   func NewStack(type E) () Stack(E) {
4      return Stack(E){}
5   }
6   func (s *Stack(E)) Pop() (r E, success bool) {
7      l := len(*s)
8      if l == 0 { return }
9      r, *s = (*s)[l - 1], (*s)[:l - 1]
10     success = true
11     return
12  }
13  func (s *Stack(E)) Push(e E)      { *s = append(*s, e) }
14  func (s *Stack(E)) IsEmpty() bool { return len(*s) == 0 }
15  func (s *Stack(E)) Len() int      { return len(*s) }
```

go2go 支持情况 [new]

- **2020-03-13：首次发布 go2go 工具**
  - 此时不支持 <-、...、switch、select，不支持第三方包 import
  - []P(T) 存在二义，但 []P(T1，T2) 不会出现二义仍无法直接使用，见 changkun/go2generics/bugs/1/
- **2020-03-19：改进**
  - 支持编写 <-、...、switch、select，支持第三方包 import
  - 但使用 testing 时部分包导入功能失效，见 changkun/go2generics/bugs/2/
  - 不明原因无法导入 errors 包 `can't find any importable name in package "errors"`
- **2020-04-03 & 2020-04-09：改进**
  - 仍然不支持泛型指针 contract C(T) { *T M() }，见 changkun/go2generics/bugs/2/
  - 修复 changkun/go2generics/bugs/2/
  - 可以导入 errors 包

Example 4: Generic map[K]V Container

[new]

```
1   // naive generic map[k]v -- by changkun
2   type Pair(type T1, T2) struct {
3      Key   T1
4      Value T2
5   }
6   type Map(type T1, T2 contracts.Comparable(T1)) struct {
7      s []Pair(T1, T2)
8   }
9   func NewMap(type T1, T2) () Map(T1, T2) {
10     return Map(T1, T2){s: [](Pair(T1, T2)){}}
11  }
12  func (m *Map(T1, T2)) Set(k T1, v T2) {
13     m.s = append(m.s, Pair(T1, T2){k, v})
14  }
15  func (m *Map(T1, T2)) Get(k T1) (v T2, ok bool) {
16     for _, p := range m.s {
17         if p.Key == k {
18             return p.Value, true
19         }
20     }
21     return
22  }
```

试试写个 append()？

Example 5: Fan-in Fan-out

[new]

```go
// generic fan-in -- by changkun
func Fanin(type T)(ins ...<-chan T) <-chan T {
    buf := 0
    for _, ch := range ins {
        if len(ch) > buf { buf = len(ch) }
    }
    out := make(chan T, buf)
    wg := sync.WaitGroup{}
    wg.Add(len(chans))
    for _, ch := range ins {
        go func(ch <-chan T) {
            for v := range ch { out <- v }
            wg.Done()
        }(ch)
    }
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

```go
// generic fan-out -- by changkun
func Fanout(type T)(r func(max int) int, in <-chan
T, outs ...chan T) {
    l := len(outs)
    for v := range in {
        i := r(l)
        if i < 0 || i > l { i = rand.Intn(l) }
        outs[i] <- v
    }
    for i := range outs {
        close(Outs[i])
    }
}
```

试试写个 Load Balancer?

历史性评述: 以 C++ 为例

Historical Review: C++ Case Study

About Conservative Attitude

「对于与大多数人而言，（在 1988 年）**使用 C++ 最大的问题就是缺乏一个 扩充的标准库**。要编写这种库，遇到的最主要 问题就是，**C++ 没有提供一种充分一般的机制，以便与定 义容器类**。如：表、向量和关 联数组等。」

「回过头看，模板恰好成 为精炼一种新语言特征的两种策略之 间的分界线。**在模板之前，**我（Bjarne Stroustrup）**一直通过实现、使用、讨论、再实现的过程去精炼一个语言特征。**而在模板之后，[...] 实现通常是和这些并行讨论的。**有关模板的讨论并没有像他所 应该做的那样广泛，我也缺乏批判性的 实现经验。**这就**导致后来**基于实现和使用经验又对模板**进行了多方面的修 订**。」

「我确实认为，在开始**描述模板机制 时自己是过于谨慎和保守**了。**我们原来就应该把许多特性加 进来，**[...] **这些特性并没有 给实现者增加多少负担，但是却 对用户特别有帮助。**」

——"The Design and Evolution of C++" Chapter 15: Templates, 15.2 Templates

About Parametric Constraints

『模板参数并没有提出任何限制。相反，所有 类型检查都被推迟到模板实例化的时刻进行（1988 年）。

「模板的用户是否应该要求其使用者 说明满足什么 样操作的类型，才能用于模板参数 吗？例如：

```
template <class T {
      T& operator=(const T&);
      int operator==(const T&, const T&);
      int operator<=(const T&, const T&);
      int operator<(const T&, const T&);
};> class vector { /*…*/ };
```

不！如此要求用 户就会降低参数机制的灵活性，又不会使 实现变得简单，或使这种功能更安全……」

（1994 年）回头再看，我明白了 这些限制对于可读性和早期错误检测的重要性。』

——"The Design and Evolution of C++" Chapter 15: Templates, 15.4 Constraints on Template Arguments

About Syntax Design

『语法总是一个问题。开始时我希望把模板参数直接放在模板名字的后面，但是 这种方式无法很清晰地 扩展到函数模板。**初看起来，不另外使用关 键字的函数语法似乎好一些：**

```
T& index<class T>(vector<T>& v, int i) { /*…*/ }
int i = index(vi, 10);
char* p = index(vpx, 29);
```

这种 "简洁" 的语法设计非常精巧，很 难在程序中识别一个模板的声明，此外还会**对某些函数模板 进行语法分析可能非常 难。**[...] 最后的模板语法被设计为：

```
template<class T> T& index(vector<T>& v, int i) { /*…*/ }
```

我也严肃的讨论过**将返回值放在参数表之后 进而很好的解决 语法分析问题，**

```
index<class T>(vector<T>& v, int i) : T& { /*…*/ }
```

但大部分人宁愿要一个关 键字来帮助识别模板，[...]
选择尖括号 <…> 而不是圆括号（…），是因为用户发现这样更容易阅读，因为圆括号在 C/C++ 里已被过度使用。事实证明，使用圆括号进行语法分析也并不困 难，但读者（reader）总是喜欢尖括号 <…>。』

——"The Design and Evolution of C++" Chapter 15: Templates, 15.7 Syntax

展望

Outlooks

Open Questions

非类型参数合约

变长参数合约

运算符重载

...

Variadic Generics?

```
1   type Tuple (type Ts ...comparable) struct {
2      elements ...Ts
3   }
4
5   func (t *Tuple(Ts...)) Set(es ...Ts) {
6      t.elements(Ts...){es...}
7   }
8
9   func (t Tuple) PirntAll() {
10     for _, e := range t.elements {
11         fmt.Println(e)
12     }
13  }
14
15  // func (t Tuple(Ts...)) Get(i int) T?!?
```

- 目前的设计不支持变长类型参数
- 目前的设计未实现变长参数表达式(即 ...)
- 进而无法实现 Tuple
- 单纯引入 ...C 的语法不能够解决多个类型的索引问题
  - 考虑 Tuple 的 Get 方法
- 单纯从索引的角度来看，但是会产生歧义
  - 可以使用 v1, v2 := t.elements[0], t.elements[1]
  - 可以使用 for _, e := range t.elements
  - 可以使用 reflect
- 索引的边界检查问题也不简单，考虑
  - 编译期索引
  - 运行时索引

Conclusions

- 回顾来看，Go 2 中基于合约的泛型设计，是可以理解的，经过多次迭代、吸取了诸多决策失误的经验
  - 目前的实现粗略的说是一种基于特设多态实现的参数化多态
- 目前的实现相对完整，但存在一些功能性的缺失，但更像是有意为之（语言更加复杂）
- 还存在非常多可改进的空间
- 会像 try proposal 一样被废弃吗？个人看法：形势还不够明朗（例如：社区反馈不够丰富），但被接受的概率很大
- 会修改语法吗？个人看法：可能不会。
- 什么时候会正式上线？个人看法：
  - 取决于社区的反馈和大量的实践
  - 以 C++ 的历史经验来看，在模板特性草案被正式定稿时，已经有大量的泛型实现，如 STL
  - Go 也需要这种社区的力量（尽管 Go 团队喜欢「一意孤行」🤷）
- 引入泛型会打破向前兼容性吗？
  - 从现在的设计来看，不会
  - 但从 C++ 的历史经验来看，已经积累的代码的迁移过程将是痛苦且漫长的

这么多不同版本的泛型设计里，你最喜欢哪一个？

进一步阅读的参考文献

References

github.com/changkun/go2generics

- 演示文稿中的示例代码参见 `demo` 文件夹
- 仓库中还包含更多示例

Go 夜读 微信公众号