Go 泛型运用

泛型起源

泛型编程是一种计算机编程风格，程序逻辑是根据参数指定的类型编写的，然后在调用时为作为参数提供的特定类型实例化。这种方法由 1973 年的 ML 编程语言开创，允许编写通用函数或类型，从而减少了代码重复。

什么是泛型

简而言之，类型做为参数。在 Go 语言语言中就是， 函数中携带泛型类型参数，接口 / 结构体支持泛型类型参数声明。

Go

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

func AddInt64(a int64, b int64) int64 {
    return a + b
}

func Add[T int | int32 | int64 | float32 | float64](a T, b T) T {
    return a + b
}
```

让一个函数获得了处理多种不同类型数据的能力，这种编程方式被称为 泛型编程。

为什么需要泛型

泛型能实现的功能通过接口 + 反射也基本能实现。但是使用过反射的人都知道反射机制有很多问题：

1. 用起来麻烦
2. 失去了编译时的类型检查，不仔细写容易出错
3. 性能不太理想

主流的泛型介绍

类型	实现原理	优点	缺点	代表语言
模板泛型	将代码中使用每一个类型编译成对应的类型的代码	代码运行效率最好	导致代码体积膨胀，增加编译复杂性，编译速度慢	C++，Rust
运行时泛型	编译时用占位符代替泛型参数，运行时由 JIT 编译成实际的类型	编译时代码不会膨胀，编译速度不受影响。经过 JIT 优化后泛型，性能几乎不受影响	增加了运行时的复杂性（将原来编译期的工作搬运到了运行时）。略微影响了效率	C#
擦除泛型	编译时将类型擦除，统一转换为 object/ 或者约束类	实现简单	1. 由于类型擦除，所有的检查只能在编译期，运行时无法错类型检查（例如不能用反射来检查泛型） 2. 对于值类型，做类型擦除后导致了额外的拆箱装箱成本	java
动态语言泛型	依托编译器语法转换做类型检查			typescript

C#实现原理查看链接：SharpLab

JAVA 实现原理链接：java 泛型字节码

Go 泛型实现原理

go 是基于模板的方案来实现泛型并做了改善

1. 为基础类型单独生成一份模板代码
2. 非基础类型统一生成一个虚拟方法表，在运行时调用的时候通过虚拟方法表找到实际的类型

泛型语法

- 函数名后可以附带一个方括号，包含了该函数涉及的类型参数（Type Paramters）的列表：`func F[T any](p T) { ... }`
- 这些类型参数可以在函数参数和函数体中（作为类型）被使用
- 自定义类型也可以有类型参数列表：`type M[T any] []T`
- 每个类型参数对应一个类型约束，上述的 any 就是预定义的匹配任意类型的约束
- 类型约束在语法上以 interface 的形式存在，在 interface 中嵌入类型 T 可以表示这个类型必须是 T:
- any\comparable\Ordered 约束类型

Go

```go
type Integer interface {

    int | int8 | int16 | int32 | int64

}
type Integer1 interface {
    ~int | ~int8
}
```

泛型反射

- Go 1.18 版本并没有增加泛型相关的反射 API，并且泛型是在编译期生成故推测 Go 不支持在运行时调用泛型方法 / 函数 . 这也是为什么 Go 方法不支持泛型原因之一

Go

```go
func Test_Reflect3(t *testing.T) {
    myType := &MyType[string]{"asd"}
    mtV := reflect.ValueOf(&myType).Elem()
    params := make([]reflect.Value, 0)
    params = append(params, reflect.ValueOf("OOOOOO"))
    mtV.MethodByName("Hello").Call(params)
}

type MyType[T any] struct {
    name T
}

func (mt *MyType[T]) Hello(show string) {
    println(mt.name, show)
}
```

- java 类型擦除式泛型，所以在运行是可通过反射插入不同类型的数据到泛型集合中

Java

```java
    List<String> strList=new ArrayList<String>();
    //如果存放其他类型的对象时会出现编译错误
    strList.add("sdadas");

//但是通过反射方案可以实现插入数字     strList.add(123)
```

- C#是运行时泛型，所以反射时仍然可以做到类型检查

C#

```
    var strList=new List<String>();
    //如果存放其他类型的对象时会出现编译错误
    strList.Add("sdadas");

//反射插入数字时会报运行时错误     strList.add(123)//error
```

泛型运用

1. 三元表达式

Go

```
func If[T any](flag bool, f1, f2 T) T {
    if flag {
        return f1
    }
    return f2
}

func IfFunc[T any](flag bool, f1, f2 func() T) T {
    if flag {
        return f1()
    }
    return f2()
}
//调用
If(len(arr)>0,arr[0],0)
IfFunc(len(arr) > 0 ,func()int64{return arr[0]},0)
```

2. 泛型类型转换

Go

```go
func ToNumber[N Number](strNumber string) N {
    var num N
    switch (interface{})(num).(type) {
    case int:
        cn, _ := strconv.Atoi(strNumber)
        return N(cn)
    case int32:
        cn, _ := strconv.ParseInt(strNumber, 10, 32)
        return N(cn)
    case int64:
        cn, _ := strconv.ParseInt(strNumber, 10, 64)
        return N(cn)
    case uint32:
        cn, _ := strconv.ParseUint(strNumber, 10, 32)
        return N(cn)
    case uint64:
        cn, _ := strconv.ParseUint(strNumber, 10, 64)
        return N(cn)
    case float32:
        cn, _ := strconv.ParseFloat(strNumber, 64)
        return N(cn)
    case float64:
        cn, _ := strconv.ParseFloat(strNumber, 64)
        return N(cn)
    }
    return num
}

//调用
a := ToNumber[int]("123")
b := ToNumber[int64]("123")
c := ToNumber[float64]("123")
d := ToNumber[int32]("123")
```

3. linq 集合操作 仓库地址

Go

```go
type Student struct {
    Id     int
    Name   string
    Age    int
    Sex    bool
    Score  int
}

var list = []*Student{
        {Id: 1, Name: "张三", Age: 18, Sex: true, Score: 80},
        {Id: 2, Name: "李四", Age: 19, Sex: true, Score: 88},
        {Id: 3, Name: "王五", Age: 20, Sex: true, Score: 87},
        {Id: 4, Name: "赵六", Age: 18, Sex: false, Score: 67},
        {Id: 5, Name: "李娟", Age: 17, Sex: false, Score: 89},
        {Id: 6, Name: "王芊", Age: 18, Sex: false, Score: 99},
        {Id: 7, Name: "赵月", Age: 19, Sex: false, Score: 72},
        {Id: 7, Name: "赵月", Age: 19, Sex: false, Score: 72},
    }
```

a. 集合过滤

Go

```go
//返回年龄大于19岁的学生
res := linq.Where(list, func(s *Student) bool { return s.Age > 19 })
    // Id: 3, Name: "王五"
```

b. 取第一个或者最后一个

Go

```
    b := linq.First(list) //输出 1
   // 获取指定条件的第一个
    b = linq.First(list, func(s *Student) bool { return s.Sex }) //输出 1

    b = linq.Last(list) //输出 7
    // 获取指定条件的最后一个
    b = linq.Last(list, func(s *Student) bool { return s.Sex })
   //输出 3
```

c. 去重

Go

```
//引用类型需要指定 去重的Key值
  c := linq.Distinct(list, func(s *Student) int { return s.Id })
   fmt.Println(c) //输出 1，2，3，4，5，6，7

   //值类型可直接调用
   var list1 = []Student{
       {Id: 1, Name: "张三", Age: 18, Sex: true, Score: 80},
       {Id: 2, Name: "李四", Age: 19, Sex: true, Score: 88},
       {Id: 3, Name: "王五", Age: 20, Sex: true, Score: 87},
       {Id: 4, Name: "赵六", Age: 18, Sex: false, Score: 67},
       {Id: 5, Name: "李娟", Age: 17, Sex: false, Score: 89},
       {Id: 6, Name: "王芊", Age: 18, Sex: false, Score: 99},
       {Id: 7, Name: "赵月", Age: 19, Sex: false, Score: 72},
       {Id: 7, Name: "赵月", Age: 19, Sex: false, Score: 72},
   }
   d := linq.DistinctComparable(list1)
   fmt.Println(d) //输出 1，2，3，4，5，6，7
```

d. 投影成新对象

Go

```
    //将学生数组转换为名称数组
    e := linq.Select(list, func(s *Student) string { return s.Name })
// 输出 ["张三","李四","王五",...]
```

e. 取前几条数据

Go

```
    //取前几条数据
    f := linq.Take(list,3)
    fmt.Println(f)
```

f. 判断 元素，集合，条件 是否存在 / 满足

Go

```
    //判断是否存在分数大于99分的同学
    linq.Any(list, func(s *Student) bool { return s.Score > 99 }) // false

    var arr = []int{1,2,3,4,4,5}
    linq.AnyComparable(arr, 8)//false
    linq.AnyComparable(arr, 8,12,1)//true

    //判断所有同学分数大于60
    linq.All(list, func(s *Student) bool { return s.Score > 60 }) //true
    //判断所有同学分数大于60
    linq.All(list, func(s *Student) bool { return s.Score > 70 }) //false

    linq.AnyComparable(arr,1) //判断所有元素都为1
```

g. 求和 / 最大值 / 最小值

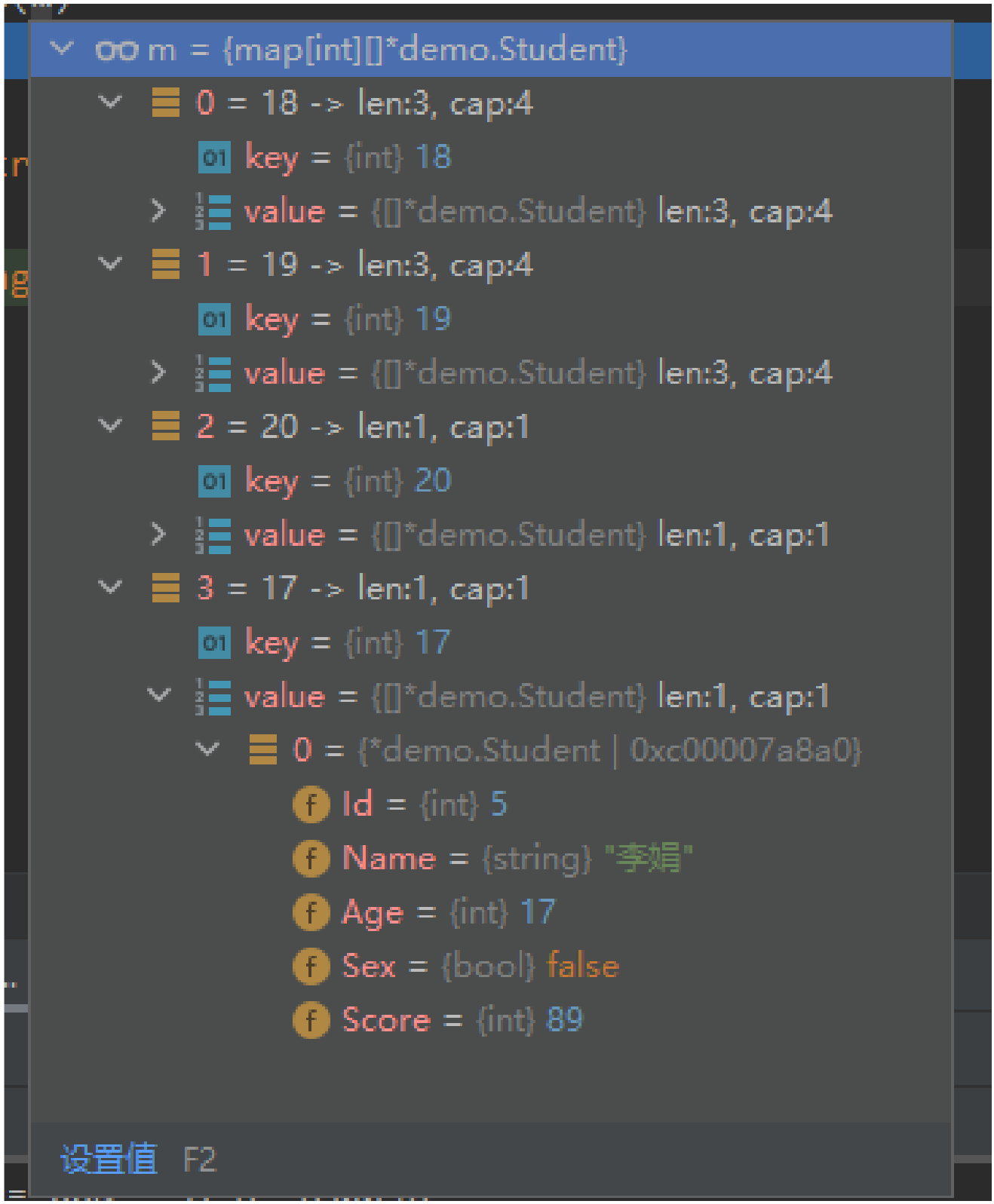
Go

```
    //分数汇总
    linq.Sum(list, func(s *Student) int { return s.Score }) // 654
    linq.Max(list, func(s *Student) int { return s.Score }) // Id: 6, Name: "王芊"
    linq.Min(list, func(s *Student) int { return s.Score }) //Id: 4, Name: "赵六"
    var arr1 = []int{1, 2, 3, 4, 4, 5}
    linq.SumOrdered(arr1) // 19
    linq.MaxOrdered(arr1) // 5
    linq.MinOrdered(arr1) // 1
```

h. 按照条件分组

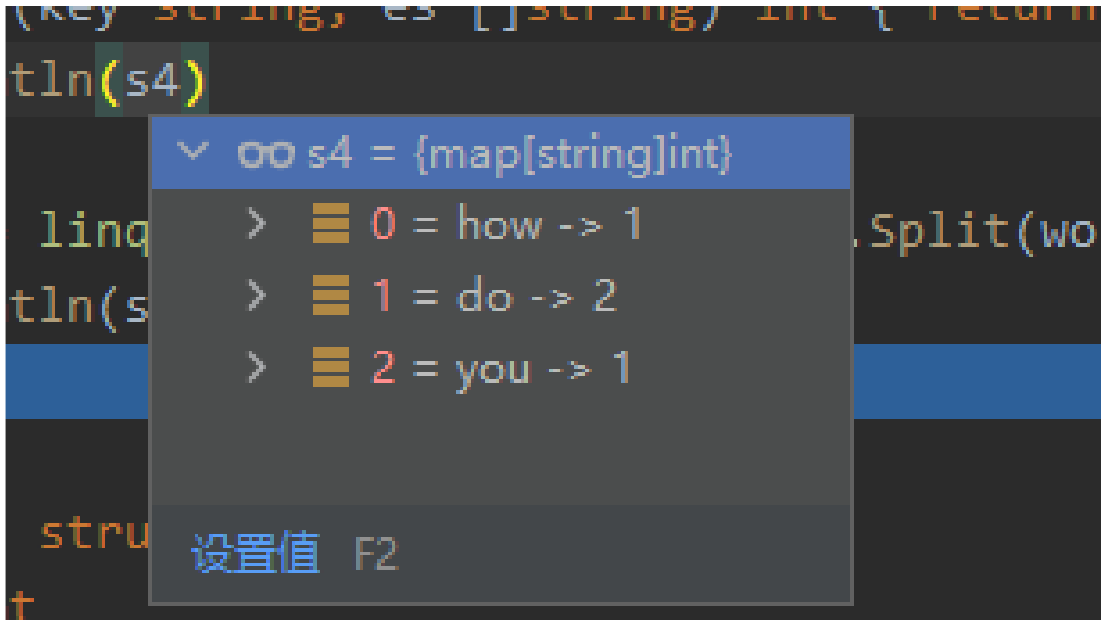
Go

```
m := linq.GroupBy(list, func(s *Student) int { return s.Age })
```


m = {map[int][]*demo.Student}
0 = 18 -> len:3, cap:4
key = {int} 18
value = {[]*demo.Student} len:3, cap:4
1 = 19 -> len:3, cap:4
key = {int} 19
value = {[]*demo.Student} len:3, cap:4
2 = 20 -> len:1, cap:1
key = {int} 20
value = {[]*demo.Student} len:1, cap:1
3 = 17 -> len:1, cap:1
key = {int} 17
value = {[]*demo.Student} len:1, cap:1
0 = {*demo.Student | 0xc00007a8a0}
Id = {int} 5
Name = {string} "李娟"
Age = {int} 17
Sex = {bool} false
Score = {int} 89
设置值 F2

Go

```
// 单词出现频率统计
    var words = "how do you do"
    //var s = words.Split(' ').GroupBy(x => x).ToDictionary(x => x.Key, x => x.Count());
   var s5 = linq.ToComparableMap(strings.Split(words, " "),
 func(key string, es []string) int { return len(es) })
    fmt.Println(s5)
```



i. 链式操作，如果有集合的组合操作需求，可以先将切片转换为链式操作对象，然后持续操作如下图

Go

```
    s1 := linq.AsEnumerable([]string{"d", "a", "b", "c", "a"}).
        Where(func(s string) bool { return s != "b" }).
        Select(func(s string) string { return "class_" + s }).
        Sort(func(s1, s2 string) bool { return s1 < s2 }).
        ToSlice()
    fmt.Println(s1)

    s2 := linq.AsComparableEnumerable([]string{"d", "a", "b", "c",
"a"}).
        Where(func(s string) bool { return s != "b" }).
        Select(func(s string) string { return "class_" + s }).
        Distinct().
        Take(3).
        ToSlice()
    fmt.Println(s2)

    s3 := linq.AsOrderEnumerable([]string{"d", "a", "b", "c", "a"}).
        Where(func(s string) bool { return s != "b" }).
        Select(func(s string) string { return "class_" + s }).
        Sort().
        Distinct().
        Take(3).
        ToSlice()
    fmt.Println(s3)

    s5 := linq.AsSelectEnumerable([]int{1, 2, 3, 4, 5}).
        Where(func(v int) bool { return v > 3 }).
        Select(func(v int) string { return "mapping_" + strconv.Itoa
(v) }).
        Take(2).
        ToSlice<>()

    fmt.Println(s5)

```

```go
func MapToSlice[K comparable, V any, R any](source map[K]V, fun func(K, V) R) []R {...}

func MapSelect[K comparable, V any, R any](source map[K]V, fun func(K, V) (K, R)) map[K]R {...}

func MapWhere[K comparable, E any](values map[K]E, predicate func(K, E) bool) map[K]E {...}

//MapWhereSelect  按照条件过滤筛选返回切片集合
func MapWhereSelect[K comparable, E any, R any](values map[K]E, predicate func(K, E) (R, bool)) []R {...}

func Distinct[E any, K comparable](values []E, fun func(E) K) []E {...}

func DistinctComparable[E comparable](values []E) []E {...}

func Select[S any, T any](source []S, fun func(S) T) []T {...}

func Take[E any](values []E, size int) []E {...}

func Fist[E any](values []E, predicates ...func(E) bool) (e E) {...}

func Where[E any](values []E, predicate func(E) bool) []E {...}

func All[E any](values []E, predicate func(E) bool) bool {...}

func AllComparable[E comparable](values []E, e E) bool {...}

func Any[E any](values []E, predicate func(E) bool) bool {...}

//AnyComparable 查找元素是否存在，如果查找对象是个切片，就判断是否相交
func AnyComparable[E comparable](values []E, es ...E) bool {...}

//AnyComparableChild 是否存在子集
func AnyComparableChild[E comparable](values []E, es ...E) bool {...}

func Join[E any](values []E, fun func(E) string, sep string) string {...}

func Sum[E any, R constraints.Ordered](values []E, fun func(E) R) R {...}
```

```go
260
261     users := linq.AsSelectEnumerable[*wps.NewUserInfo, *NewUserInner](newUserInfos).
262         Select(func(s *wps.NewUserInfo) *NewUserInner {
263             return &NewUserInner{
264                 CorpId:          s.CompanyId,
265                 CorpName:        s.CompanyName,
266                 Logo:            s.AvatarUrl,
267                 Enable:          corpMaps[s.CompanyId] != nil,
268                 NewUserId:       s.UserId,
269                 CanChange:       s.IsCompanyAccount,
270                 IsCompanyAccount: s.IsCompanyAccount,
271                 NewUserName:     s.NickName,
272             }
273         }).
274         Sort(func(a, b *NewUserInner) bool { return util.BoolToInt(a.Enable) > util.BoolToInt(b.Enable) }).
275         ToSlice()
```

4. 实现泛型 Set，泛型 LRU, 泛型 orm, 泛型 Redis 返回


package linq
type Set[T comparable] map[T]struct{}
func NewSet[T comparable](es ...T) Set[T] {...}
func (s Set[T]) Len() int {...}
func (s Set[T]) IsEmpty() bool {...}
func (s Set[T]) Add(es ...T) bool {...}
func (s Set[T]) Remove(es ...T) {...}
func (s Set[T]) Contains(v T) bool {...}
func (s *Set[T]) Clone() Set[T] {...}
func (s Set[T]) ToSlice() []T {...}

不足

- 方法不支持泛型

Go

```go
type A[T any] struct {
}

//编译通过
func (a A[T]) SelectString() string {
    return ""
}

//编译不通过
func (a A[T]) Select2[E any]() E {
    retrun nil
}

type Foo struct {}
//编译不通过
func (Foo) bar[T any](t T) {}
```

- Go 没有提供在编译期操作类型的能力，只能在运行时操作

Go

```go
type Number interface {
    int | int32 | int64 | uint32 | uint64 | float64 | float32
}

func ToNumber[N Number](strNumber string) N {
    var num N
    switch (interface{})(num).(type) {
    case int:
        cn, _ := strconv.Atoi(strNumber)
        return N(cn)
    case int32:
        cn, _ := strconv.ParseInt(strNumber, 10, 32)
        return N(cn)
    case int64:
        cn, _ := strconv.ParseInt(strNumber, 10, 64)
        return N(cn)
    case uint32:
        cn, _ := strconv.ParseUint(strNumber, 10, 32)
        return N(cn)
    case uint64:
        cn, _ := strconv.ParseUint(strNumber, 10, 64)
        return N(cn)
    case float32:
        cn, _ := strconv.ParseFloat(strNumber, 64)
        return N(cn)
    case float64:
        cn, _ := strconv.ParseFloat(strNumber, 64)
        return N(cn)
    }
    return num
}
```

- 类型转换，不支持转换为带类型约束的 interface

Go

```
type Number interface {
    int | int32 | int64 | uint32 | uint64 | float64 | float32
}
    K, _ := v.(int)   //编译通过
    g, _ := v.(Number) //编译不通过
```

- 更加智能类型推导，函数lamda化

Go

```
f(x) = ax + b
y = f(x)

joinCorpIds := linq.Select(newUserInfos, func(s *wps.NewUserInfo) int64 { return s.CompanyId })

joinCorpIds := linq.Select(newUserInfos, s = > s.CompanyId)

joinCorpIds = newUserInfos.Select(s = > s.CompanyId)

```

参考文档

https://go.dev/doc/tutorial/generics

https://go.dev/blog/when-generics

GitHub - ahmetb/go-linq: .NET LINQ capabilities in Go

C++ 模板与 C# 泛型的区别 Microsoft Docs