Go maps & slices & cmp

标准库 std

maps、slices 和 cmp 包。

package maps

Clone

克隆

```
函数签名:
func Clone[M ~map[K]V, K comparable, V any](m M) M

例:
m := map[string]int{
    "one": 1,
    "two": 2,
    "three": 3,
}
cm := maps.Clone(m)
```

Copy

复制

```
函数签名:
func Copy[M1 ~map[K]V, M2 ~map[K]V, K comparable, V any
    ↪ ](dst M1, src M2)

例:
srcM := map[string]int{
    ...
}
dstM := make(map[string]int, 3)
maps.Copy(dstM, srcM)
```

DeleteFunc

删除符合条件的元素

```
函数签名:
func DeleteFunc(m M, del func(K, V) bool)

例:
m := map[string]int{
    ...
}
maps.DeleteFunc(m, func(k string, v int) bool {
    return v%2 != 0 // 删除奇数
})
```

Equal

相等性检查

```
函数签名:
func Equal(m1 M1, m2 M2) bool

例:
m1 := map[string]int{
    ...
}
m2 := map[string]int{
    "one": 1,
    "two": 2,
}
fmt.Println(maps.Equal(m1, m2)) // false
```

EqualFunc

提供自定义比较函数进行相等性检查

```
函数签名:
func EqualFunc(m1 M1, m2 M2, eq func(V1, V2) bool) bool

例:
m1 := map[int]string{
        1: "one",
        10: "Ten",
        1000: "THOUSAND",
}
m2 := map[int][]byte{
        1: []byte("One"),
        10: []byte("Ten"),
        1000: []byte("Thousand"),
}
eq := maps.EqualFunc(m1, m2, func(v1 string, v2 []byte)
    ↪  bool {
        return strings.ToLower(v1) == strings.ToLower(
            ↪ string(v2))
})
fmt.Println(eq) // true
```

package slices

- **BinarySearch**: 二分查找
- **BinarySearchFunc**: 二分查找，提供自定义比较函数
- **Clip**: 裁剪，移除多余的 capacity, $s[:len(s):len(s)]$
- **Clone**: 克隆，浅复制
- **Compact**: 压缩，保持元素唯一，在源 slice 上操作，注意内存泄露问题
- **CompactFunc**: 压缩，提供自定义比较函数
- **Compare**: 比较，使用 $cmp.Compare$, 按索引逐个比较
- **CompareFunc**: 比较，提供自定义比较函数
- **Contains**: 是否包含某个元素
- **ContainsFunc**: 是否包含符合条件的元素
- **Delete**: 删除 $s[i:j]$ 的元素, 修改源 slice，注意内存泄露问题
- **DeleteFunc**: 删除符合条件的元素
- **Equal**: 相等性检查, 长度相同，每个索引值都相同
- **EqualFunc**: 相等性检查，提供自定义比较函数
- **Grow**: 扩容至少 n 个元素
- **Index**: 查找元素的索引 (第一个)，如果没有返回-1
- **IndexFunc**: 查找符合条件的元素的索引 (第一个)，如果没有返回-1
- **Insert**: 往指定的索引处插入多个元素，原索引处的元素后移
- **IsSorted**: 是否已排好序
- **IsSortedFu**: 是否已排好序，提供自定义比较函数
- **Max**: 返回元素中最大值
- **MaxFunc**: 返回元素中最大值，提供自定义比较函数
- **Min**: 返回元素中最小值
- **MinFunc**: 返回元素中最小值，提供自定义比较函数
- **Replace**: 替换 $s[i:j]$ 的元素，如果 i j 数量和提供的元素数量不一致，会返回怪异的结果
- **Reverse**: 反转 slice
- **Sort**: 排序, 元素必须是可排序的 ($cmp.Ordered$])
- **SortFunc**: 排序，提供自定义比较函数
- **SortStableFunc**: 稳定排序，提供自定义比较函数

package cmp

Compare

比较两个值

```
函数签名：
func Compare[T Ordered](x, y T) int

返回值：
-1 如果x小于y，
 0 如果x等于y，
+1 如果x大于y.
```

Less

判断值 x 是否小于 y

```
函数签名：
func Less[T Ordered](x, y T) bool

对于浮点数，NaN总是小于non-NaN。
-0.0 不小于(等于)0.0
```

Ordered

可排序的类型

```
type Ordered interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64 |
        ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64
            ↪ | ~uintptr |
        ~float32 | ~float64 |
        ~string
}
```

扩展库 golang.org/x/exp

maps、slices 和 constraints 包。

package maps

- **Clear**: 清空 map,Go 1.21 中可以使用内建函数 clear 替代
- **Clone**: 复制一个 map
- **Copy**: 把源 map 的键值对复制到目的 map 中
- **DeleteFunc**: 删除符合条件的元素
- **Equal**: 相等性检查
- **EqualFunc**: 相等性检查，提供自定义比较函数
- **Keys**: 返回 map 的所有键
- **Values**: 返回 map 的所有值

package slices

和标准库中的 slices 相同。

- **BinarySearch**: 二分查找
- **BinarySearchFunc**: 二分查找，提供自定义比较函数
- **Clip**: 裁剪，移除多余的 capacity, $s[:len(s):len(s)]$
- **Clone**: 克隆，浅复制
- **Compact**: 压缩，保持元素唯一，在源 slice 上操作，注意内存泄露问题
- **CompactFunc**: 压缩，提供自定义比较函数
- **Compare**: 比较，使用 *cmp.Compare*, 按索引逐个比较
- **CompareFunc**: 比较，提供自定义比较函数
- **Contains**: 是否包含某个元素
- **ContainsFunc**: 是否包含符合条件的元素
- **Delete**: 删除 $s[i:j]$ 的元素, 修改源 slice，注意内存泄露问题
- **DeleteFunc**: 删除符合条件的元素
- **Equal**: 相等性检查, 长度相同，每个索引值都相同
- **EqualFunc**: 相等性检查，提供自定义比较函数
- **Grow**: 扩容至少 n 个元素
- **Index**: 查找元素的索引 (第一个)，如果没有返回-1
- **IndexFunc**: 查找符合条件的元素的索引 (第一个)，如果没有返回-1
- **Insert**: 往指定的索引处插入多个元素，原索引处的元素后移
- **IsSorted**: 是否已排好序
- **IsSortedFunc**: 是否已排好序，提供自定义比较函数
- **Max**: 返回元素中最大值
- **MaxFunc**: 返回元素中最大值，提供自定义比较函数
- **Min**: 返回元素中最小值
- **MinFunc**: 返回元素中最小值，提供自定义比较函数
- **Replace**: 替换 $s[i:j]$ 的元素，如果 i j 数量和提供的元素数量不一致，会返回怪异的结果
- **Reverse**: 反转 slice
- **Sort**: 排序, 元素必须是可排序的 (*cmp.Ordered*])
- **SortFunc**: 排序，提供自定义比较函数
- **SortStableFunc**: 稳定排序，提供自定义比较函数

package constraints

Complex 复数

```
type Complex interface {
    ~complex64 | ~complex128
}
```

Float 浮点数

```
type Complex interface {
    ~complex64 | ~complex128
}
```

Integer 整数

```
type Integer interface {
    Signed | Unsigned
}
```

Ordered 可排序类型

```
type Ordered interface {
    Integer | Float | ~string
}
```

Signed 有符号整数

```
type Signed interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}
```

Unsigned 无符号整数

```
type Unsigned interface {
    ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~
        ↪ uintptr
}
```

2023 整理 by 晁岳攀 (@ 鸟窝) https://colobu.com。