第 5 章

CHAPTER 5

类 型 系 统

Go 语言中的类型可分为两类。第一类拥有类型名，该类型名可以是一个标识符或者标识符后跟类型参数，被称作命名类型(Named Type)。命名类型包括系统预定义类型(参看 2.1.2 节中的“数据类型”)、定义类型和类型参数。另一类被称作复合类型(Composite Type)，它使用类型字面量(Type Literal)来定义。复合类型包括数组类型、切片类型、结构体类型、指针类型、函数类型、接口类型、map 类型和 chan 类型，代码如下：

```
bool                    //命名类型 系统预定义类型
type t struct {}        //t: 命名类型 定义类型
func f[T any]() {}      //T: 命名类型 类型参数
[3]float                //复合类型 数组类型
 * int                   //复合类型 指针类型
func() int              //复合类型 函数类型
interface { F() }       //复合类型 接口类型
struct{A int}           //复合类型 结构体类型
map[int]string          //复合类型 map 类型
chan  * int             //复合类型 chan 类型

type gt[T any] struct{}
gt[int]                 //命名类型(标识符 + 类型参数)实例化定义类型
```

5.1 定义类型

Go 语言中使用 type name T 的形式定义的类型 name 为定义类型(Defined Type)。Go 语言内建的数值类型(除 byte 和 rune 外)、bool、string 都为定义类型，可认为 Go 的编译器自动生成了如下代码：

```
package builtin

type bool boolImpl
type string stringImpl
type int8 int8Impl
type int32 int32Impl
```

```
type int intImpl
//... 省略更多数值类型

type byte = int8    //byte 为 int8 的别名
type rune = int32   //rune 为 int32 的别名
```

定义类型都是独特的，由于它和任何其他类型（包括它所基于的类型 T）都不同，所以 int、int8、uint 等都是不同的类型，它们之间不能直接相互赋值。

通过类型定义，可以基于现有类型生成新类型，新类型和它所基于的类型拥有共同的底层类型（Underlying Type）。底层类型相同的类型之间可以进行类型转换，代码如下：

```
type MyInt int
type MyInt2 MyInt

func Test() {
    var n int = 1
    var mn MyInt = 2
    var mn2 MyInt2 = 3
    //int MyInt MyInt2 的底层类型都为 int
    n = int(mn)
    n = int(mn2)
    mn = MyInt(mn2)
    mn = MyInt(n)
}
```

此处，int、MyInt、MyInt2 这 3 种类型的底层类型都为 int，彼此之间可以进行类型转换。

注意，MyInt2 的底层类型为 int 而不是 MyInt，因为 MyInt 不够底层，它自身的底层类型为 int。Go 语言中的底层类型包括预定义的数值类型（例如 int、float32 等）、bool、string 和复合类型，代码如下：

```
type Int int

type A bool
type B A

type C [3]int
type D [3]int

type E1 map[int]string
type E E1

type F struct{ A, B int }

type G func(int) int
```

类型 Int 的底层类型为 int。类型 A 和 B 的底层类型均为 bool。类型 C 和 D 的底层类型均为[3]int。类型 E 的底层类型为 map[int]string。类型 F 的底层类型为 struct{ A, B int }。

类型 G 的底层类型为 func(int) int。由于 Int 和 int 为两个不同的定义类型，所以[]Int 和[]int 也为两个不同的复合类型，并且二者的底层类型不相同(均为自身)，所以[]Int 和[]int 之间不能进行类型转换。

定义类型可以拥有自己的方法，例如为 MyInt 类型添加了一个 Add()方法的代码如下：

```
type MyInt int
func (n MyInt) Add(n2 MyInt) MyInt {
    return n + n2
}
```

调用此方法的代码如下：

```
var n1 MyInt = 1
var sum MyInt = n1.Add(2)       //sum 的值为 3
```

在 Go 语言中，只能为定义在本包中的类型添加方法，例如不能为 int 类型添加方法，因为 int 并非定义在当前包中。

定义类型并不具有它所绑定类型 T 的方法，但如果 T 为接口，则新类型依然拥有接口中的方法；如果 T 有元素或字段(例如数组、切片或结构体)，则元素或字段在新类型中依然有效，代码如下：

```
//Mutex 有 Lock() 和 Unlock()两个方法
type Mutex struct          { /* Mutex 的字段 */ }
func (m *Mutex) Lock()     { /* Lock 的实现 */ }
func (m *Mutex) Unlock()   { /* Unlock 的实现 */ }

//NewMutex 拥有和 Mutex 一样的字段，但是它没有任何方法
type NewMutex Mutex

// *Mutex 拥有 Mutex 的所有方法(它不是定义类型)
//PtrMutex 没有任何方法
type PtrMutex *Mutex

type Block interface {
    Block()
}

//MyBlock 拥有和 Block 同样的方法集
type MyBlock Block

//IntArray 的长度为 3，元素类型为 int
type IntArray [3]int
```

NewMutex 和 PtrMutex 都是定义类型，它们都不继承所绑定类型的方法，由于它们也没有定义自己的方法，所以它们的方法集为空。*Mutex 不是定义类型，它拥有 Mutex 的所有方法。由于 MyBlock 被绑定到接口 Block，所以它和 Block 的方法集相同。IntArray

虽然和[3]int 不是同一种类型，但是它同样拥有 3 个 int 元素。

5.2 类型别名

Go 语言中可以为类型定义别名(Alias)，别名并不引入新的类型，别名和它所绑定的类型为同一种类型。当别名出现在代码中时，它所代表的就是其绑定的类型。也可理解为编译器在遇到类型别名时会自动将其替换为所绑定的类型，代码如下：

```
package main

//Integer 为 int 的别名
//所有出现 Integer 的地方都相当于 int
type Integer = int

//非法：无法为 int 添加方法
//func (i Integer) F() {}       //相当于 func (i int) F() {}

func main() {
    var i Integer               //等价于 var i int
    var i2 int
    i += i2                     //i 和 i2 的类型都为 int
}
```

要注意区分 type Integer = int(类型别名)和 type Integer int(定义类型)。这是两种完全不同的类型定义方式。

在 Go 语言预定义类型中，byte 为 int8 的别名，rune 为 int32 的别名。

5.3 类型匹配

在 Go 语言中，两个类型要么相同，要么不同。首先，命名类型都是独特的，和任何其他类型都不相同，其次，复合类型只有在它们的类型字面量一致时才彼此相同。具体来讲：

(1) 对于两个数组类型，只有当它们的元素类型相同且长度相同时才相同。

(2) 对于两个切片类型，当它们的元素类型相同时才相同。

(3) 对于两个结构体类型，只有当它们有相同的字段序列，并且对应字段的名称、类型、标签都相同时才相同。注意，位于不同包中的未导出字段名均不相同，即不同包中的两个结构体类型，如果有未导出成员，则它们一定不相同。

(4) 对于两个指针类型，当它们的元素类型相同时才相同。

(5) 对于两个函数类型，只有当它们的参数和返回值的个数和类型都相同时才相同。注意，参数和返回值的名称不要求一定匹配。

(6) 对于两个接口类型，当它们的定义的类型集相同时才相同。

(7) 对于两个 map 类型,当它们的键和值类型都相同时才相同。

(8) 对于两个 channel 类型,当它们的元素类型和方向都相同时才相同。

示例代码如下:

```
type (
    A0 = []string
    A1 = A0
    A2 = struct{ a, b int }
    A3 = int
    A4 = func(A3, float64) * A0
    A5 = func(x int, _ float64) *[]string

    B0 A0
    B1 []string
    B2 struct{ a, b int }
    B3 struct{ a, c int }
    B4 func(int, float64) * B0
    B5 func(x int, y float64) * A1

    C0 = B0

    D0 struct {
        a int `example:"tag"`
        c int
    }
)
```

下列类型相同:

(1) A0、A1 和[]string。

(2) A2 和 struct{ a, b int }。

(3) A3 和 int。

(4) A4、func(int, float64) *[]string 和 A5。

(5) B0 和 C0。

(6) []int 和[]int。

(7) struct{ a, b *B5 }和 struct{ a *B5; b *B5 }。

(8) func(x int, y float64) *[]string、func(int, float64) (result *[]string)和 A5。

B0 和 B1 不相同,因为它们是命名类型(定义类型)。D0 和 B3 不同,因为字段 a 的标签不同。

5.4 可赋值性

一个类型的变量,在下列情况下可以赋值给另一个类型的变量:

(1) 当两个变量的类型完全相同时,彼此可相互赋值。

(2) 复合类型 T 的变量可以和以 T 为底层类型的变量相互赋值,代码如下:

```
type S struct {A int}
var named S
var composite struct {A int}
named = composite
composite = named

type A []int
var named2 A
var composite2 = []int{1, 2}
named2 = composite2
composite2 = named2
```

(3) 元素类型相同的 chan 之间,只要方向兼容就可以赋值,但两个命名类型除外,代码如下:

```
var rw chan int
var r <- chan int
r = rw

type RWC chan int
type RC <- chan int
var rc RC
var rwc RWC

rc = r
rc = rw
rwc = rw

//两个命名类型是不同的
rc = rwc //!! 语法错误 !!
```

(4) 实现了接口 T 的类型的变量可以赋值给 T 类型的变量,代码如下:

```
// * bytes.Buffer 和 * os.File 都实现了 io.Reader 接口
var r io.Reader = &bytes.Buffer{}
r = ( * os.File)(nil)
```

(5) 预定义标识符 nil 可以赋值给指针、函数、切片、map、channel 或接口类型,代码如下:

```
var _ * int = nil
var _ func() = nil
var _ []int = nil
var _ map[int]int = nil
var _ chan int = nil
var _ error = nil
```

(6) 无类型常量可以赋值给兼容的数值类型,代码如下:

```
const c = 1
var _ int = c
var _ byte = c
```

```
var _ float32 = c
type Int int
var _ Int = c
var _ complex128 = c
```

5.5 可转换性

一个变量 x 在满足下列条件之一时，可以显式地转换为另一个类型 T。

(1) x 可赋值给 T(参见 5.4 节)。

(2) x 的类型和 T 有一样的底层类型。结构体字段的标签不影响可转换性，代码如下：

```
type (
    A struct{ A int }
    B A
    C struct {
        A int `example:"tag"`
    }
)
//A, B, C 的底层类型一致(不考虑标签)，可以相互转换
var a A = A(B{})
var b B = B(a)
var c C = C(b)
```

(3) x 的类型和 T 都是指针，并且它们的指针元素类型有一样的底层类型。结构体字段的标签不影响可转换性，代码如下：

```
type (
    A struct{ A int }
    B A
    C struct {
        A int `example:"tag"`
    }
)
//A, B, C 的底层类型一致(不考虑标签)
// *A, *B, *C 可以相互转换
var a *A = (*A)(&B{})
var b *B = (*B)(a)
var c *C = (*C)(b)
```

(4) x 的类型和 T 均为整数或浮点数，或 x 的类型和 T 都是复数，代码如下：

```
var a int = 1
var b uint = uint(a)
var c float32 = float32(a)
a = int(c)

var d complex64 = 1.0 + 2i
```

```
var e complex128 = complex128(d)
d = complex64(e)
```

(5) x 的类型为整数、[]byte 或[]rune，T 为 string，代码如下：

```
var a string = string(97)          //"a"
var b string = string('a')         //"a"
var c string = string([]byte{97})  //"a"
var d string = string([]rune{'a'}) //"a"
```

(6) x 的类型为 string，T 为[]byte 或[]rune，代码如下：

```
var a []byte = []byte("a")        //[97]
var b []rune = []rune("a")        //[97]
```

(7) x 的类型为[]E，T 为 E 的数组或数组指针，代码如下：

```
var a = []byte{1, 2, 3}
//如果数组的长度大于切片长度，则会引发运行时 panic
var b [3]byte = ([3]byte)(a)
var c *[3]byte = (*[3]byte)(a)
```

5.6 零值

Go 语言中，未初始化的变量值为其类型的零值。bool 类型的零值为 false；数值类型的零值为 0；string 的零值为空字符串；指针、函数、接口、切片、chan 和 map 的零值都为 nil。数组和结构体的零值包含每个元素或字段的零值。

指针、函数和接口的零值不可用。对 nil 指针脱引用、调用 nil 函数或者 nil 接口的方法都会引发运行时 panic。

切片的零值为 nil，是一个可用的空切片，其 len()和 cap()都为 0，可以对其调用 append()。当然对其使用索引[]运算符会导致索引越界错误，但这是因为其长度为 0 而不是因为其值为 nil。

chan 的零值为 nil，是一个可用的空 chan，其 len()和 cap()都为 0。向其中写入值或从其中读取值都会永远阻塞。close()一个 nil chan 会引发运行时 panic。

map 的零值是一个可用的空 map，但向其中添加元素会导致运行时 panic。