第四章 数据类型

数据存在于我们的生活的每一个场景中。同样的数据，在不同的场景下，表达的信息也完全不同。

例如：13333333333

此数据可以表示为手机号是13333333333；也可以表示为数字一百三十三亿三千三百三十三万三千三百三十三；还可以表示为一串无意义的字符13333333333。

单纯的数据无法让在我们在认知中达到一致，在计算机系统中单纯的数据也就更无法被识别，所以我们引入了数据类型的概念，帮助计算机识别数据的具体作用及场景。根据不同的数据类型定义，系统才可以判断并将数据转换成二进制代码。

例如：同样是1，如果为整数，其ascii码为1；如果是字符串“1”，其ascii码值为49。

Go将数据类型分为四类：

* 基础类型:数值类型、字符串、布尔型、字符类型、复数类型
* 复合类型：数组、结构体
* 引用类型：指针、切片、字典、函数、通道
* 接口类型：interface

4.1基础类型

4.1.1 数值类型

Go语言的数值类型包括;÷;p;/10201254了整型数、浮点型和复数。每种数值类型都决定了对应的大小范围和是否支持正负号。

4.1.1.1 整型

Go语言为我们提供了如下的整型支持：

|  |  |  |
| --- | --- | --- |
| 类型 | 长度（字节） | 值范围 |
| int8 | 1 | -27 – 27 -1 |
| uint8（byte） | 1 | 0–28-1 |
| int16 | 2 | -215–215-1 |
| uint16 | 2 | 0–216-1 |
| int32 | 4 | -231–231-1 |
| uint32 | 4 | 0–232-1 |
| int64 | 8 | -263–263-1 |
| uint64 | 8 | 0–264-1 |
| int | 依平台变化 | 依平台变化 |
| uint | 依平台变化 | 依平台变化 |
| uintptr | 与指针相同 | 32位平台下4字节，64个平台下8字节 |
| 表4.1 | | |

类型分析

int8与int是完全不同的两种数据类型，编译器不能做自动的类型转换。

例4.1

func main() {

var v int64

v1 := 10 //系统自动推断数据类型为int

v = v1 //此处报错

fmt.Println(v)

}

报错内容如下：

cannot use v1 (type int) as type int64 in assignment

错误提示告诉我们“不能使用int类型作为int64位类型的参数”，所以如果我们要使之成立就需要使用强制类型转换解决此问题。修改后的代码如下：

例4.2

func main() {

var v int64

v1 := 10

v = int64(v1)

fmt.Println(v)

}

同样int和uint也是完全不同的两种数据类型，编译器同样不能做自动的类型转换。

大家在表4.1看到的uint8这个数据类型也就是传说中的byte类型，如下例子的方式是可行的：

例4.3

func main() {

var v byte

var v1 uint8

v1 = 10

v = v1

fmt.Println(v)

}

既然二者为同一类型，为什么要给它们提供不77同的名字呢？

一般情况下我们使用byte类型来说明此变量是一个原始的数据而不是一个小的整型数

uintptr是一种无符号整数类型，没有指定大小，但是足够容纳指针。此类型在底层编程中使用较多，与C语言函数库或者操作系统接口交互的地方会使用，其它情况使用较少。

4.1.1.2浮点型

浮点型用于表示包含小数点的数据，例如3.1415926就是一个浮点数。Go语言中的浮点类型采用IEEE-754标准的表达方式，Go语言提供了两种精度的浮点数，float32和float64

浮点数能表示的数据范围如下例所示：

例4.4

func main() {

fmt.Println(math.MaxFloat32) //输出：3.4028234663852886e+38

fmt.Println(math.MaxFloat64) //输出：1.7976931348623157e+308

}

通常情况下我们建议使用float64，毕竟flaot64的精度相对较大，累计计算的误差相对较小。float32的有效bit位只有23个，其它的用于指数和符号；当整数大于23bit能表达的范围时，float32的表示将出现误差，例如：

例4.5

func main() {

var f float32 = 1 << 24 // 1 << 24

fmt.Println(f == f+1) // "true"!

fmt.Println(f) // 1.6777216e+0.7

fmt.Println(f + 1) // 1.6777216e+0.7

}

上面的例子是显示指数形式的浮点数，我们平常在使用的时候使用的是类似3.1415926这样的浮点数，那我们需要如何输出呢？

例4.6

func main() {

var f float32 = 123453.1415926

fmt.Printf("%8.2f", f) // 123453.14

fmt.Printf("%3.1f", f) // 123453.1

fmt.Printf("%3.4f", f) // 123453.1406

}

注意，如果定义的时候不加小数点，会被推导为整型。

4.1.2布尔类型

布尔类型只有两个值true和false；这两个符号代表的意义是逻辑真值及逻辑假值。定义布尔型变量使用bool关键字。

以下是几种定义及初始化布尔型变量的方法：

例4.7

func main() {

var v1 bool

fmt.Println(v1) //false

v1 = true

fmt.Println(v1) //true

v1 = (1 == 2)

fmt.Println(v1) //false

v2 := (2 == 2)

fmt.Println(v2) //true

var a, b int

a = 2

b = 2

v3 := (a == b)

fmt.Println(v3) //true

}

由例子可见，定义了v1变量后，系统默认赋值为false，同时我们在v3变量可以看到，我们将一个逻辑判断表达式赋值给v3，v3会被自动推导为bool类型。

4.1.3 字符串类型

Go语言中，字符串也是一种基本类型。Go语言中字符串的声明和初始化非常简单.

例4.8

func main() {

var s string

s1 := "this is String"

ch := s1[0]

fmt.Println(s) //这里什么也不会显示

fmt.Println(s1) //这里输出this is String

fmt.Println(reflect.TypeOf(ch)) //这里输出uint8，也就是byte类型

fmt.Println(ch) //这里输出ascii码116

fmt.Printf("%c", ch) //这里输出字母t

}

从上面的例子我们可以看到，只定义一个字符串，在不赋值的情况下，输出的是空内容；我们可以使用s1[N]的方式，来获取第N个字符（程序员的世界数字从0开始哦^\_^）；s1[N]方式取出来的值类型是uint8类型，也就是我们之前说过的byte类型，其内容为字符串的ascii码，我们使用printf的“%c”模式，将其转换成字符串，就可以得到我们获取到的字符串了。

如果我们可以使用s1[N]的方式得到字符串的第N个字符，那么我们是不是可以通过s1[N] = “a”的方式来赋值呢？我们看下面的例子：

例4.9

func main() {

var s = "this is String"

ch := s[1]

fmt.Printf("%c", ch)

s[1] = "k" //cannot assign to s[1]

}

这个例子会报一个编译错误，告诉我们不能给s[1]赋值。

cannot assign to s[1]

4.1.4字符类型

Go语言中的rune类型对应是int32类型。rune代表了单个Unicode字符。

rune类型和string类型有什么区别呢？

我们看下面的例子就能知晓一二了：

例4.10：

func main() {

var s1 rune

var s2 rune

var s3 rune

s1 = 'a'

s2 = "a"//此处报错cannot use "a" (type string) as type rune is assignment

s3 = 'abc'//此处报错 invalid character literal(more than one character)

fmt.Println(s1) //97

fmt.Printf("%c", s1) //a

}

通过上面代码我们看到，使用单引号和双引号包起来的字符对于Go语言来说是完全不同的两种数据，我们把双引号里面认为是字符串(string)，而单引号里面的认为是字符（rune）。

上例中，我们给s2赋值为”a”的时候报错为不能使用字符串为字符类型赋值，而我们给s3赋值为‘abc’的时候，编译错误为无效的字符，超过了一个字符。所以我们可以得出结论rune类型在使用及赋值的时候，必须使用单引号包起来，并且只能是一个字符，不允许赋值多个。

我们给s1赋值为’a’,实际我们刚才说过rune是int32的等价类型，所以直接输出我们得到的是字符的ascii码97。

4.1.5复数类型

复数实际上由两个实数构成，一个表示实部，一个表示虚部。如果对数学上的复数不够了解，那么就需要先学习一下数学上的复数。此数据类型我们并不常用，大家了解一下即可。

例4.11：

func main() {

var v1 complex64

v1 = 3.2 + 12i

v2 := 3.2 + 12i

v3 := complex(3.2, 12)

fmt.Println(v1) //(3.2+12i)

fmt.Println(v2) //(3.2+12i)

fmt.Println(v3) //(3.2+12i)

}

4.2复合类型

4.2.1数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由N（N >= 0）个元素组成。数组的长度必须是固定的不可变的，所以很少会直接使用数组。和数组对应的两种引用类型切片类型与字典类型是可以增长和收缩的动态序列，它们的使用也更灵活，所以这两种数据类型也是未来我们最常用的数据类型。但是要理解这两种数据类型，我们需要先从理解数组类型开始。

数组的每个元素都可以通过索引下标来访问，还记得我们之前说过，程序员的世界从0开始。所以索引下标也是从0开始的。

大家可以回归到4.1.3章节看一下在字符串中，我们是如何使用下标获取字符串中单个字符的。数组的基础操作，也和字符串中获取字符的方式类似。我们通过例子看一下如何构建数组，如何对数组进行基本操作。

例4.12：

func main() {

var a [8]int //定义长度为8整型数组

//定义长度为5的数组并初始化

var b [5]int = [5]int{0, 1, 2, 3, 4}

fmt.Println(a[0]) //输出：0

fmt.Println(b[len(b)-1]) //输出:4

fmt.Println(b) //输出：[0 1 2 3 4]

b[0] = 100 //修改b数组第0个元素的值

fmt.Println(b[0]) //输出：100

}

数组遍历的方法有两种：

例4.13

func main() {

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

//以下为第一种遍历方式

for i := 0; i < len(b); i++ {

fmt.Println(b[i])

//换行输出：

//0

//1

//2

//3

//4

}

//以下为第二种遍历方式

for key, value := range b {

fmt.Printf("下标是：%d 值是：%d\n", key, value)

//换行输出：

//下标是：0 值是：0

//下标是：1 值是：1

//下标是：2 值是：2

//下标是：3 值是：3

//下标是：4 值是：4

}

}

数组的长度必须是固定的，所以我们在给数组赋值的时候要注意a[8]int与b[5]int是完全不同的两种类型。

例4.14：

func main() {

var a [8]int //定义长度为8整型数组

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

a = b //编译错误：cannot use b(type [5]int) as type [8]int in assignment

}

当我们尝试将长度为5的数组赋值给长度为8的数组时，编译报错，告诉我们不允许这样的操作。

4.2.2结构体

结构体是一种聚合的数据类型，由N（N >=0）个任意类型的值聚合成的实体，每个值被称为结构体的成员或属性。

结构体的基本定义与使用方法如下：

例4.15

type Object struct {

Hash string

length int

}

func main() {

var obj Object

obj.length = 10

obj.Hash = "45454afasf4asdf1a"

fmt.Println(obj.length) //输出：10

}

我们使用type关键字定义了一个Object的结构体（struct），然后将成员用“{”、“}”包含。任何数据类型都可以成为结构体的成员的类型，接下来的内容中，我们都将使用属性来替代成员这个词，大家只要看到属性，就一定要想到，这是和结构体有关。

当属性名的首字母大写时，我们就可以在任何位置上访问这个属性；如果属性首字母是小写，那这个属性就是私有的，只能在结构体声明的包下进行访问或赋值。上例中，结构体Object是在main名声明的，所以可以在main函数中直接访问操作。如果在别的包中，就不能访问length属性，只能访问Hash属性。

关于结构体的各种使用方法，我们将在面向对象部分给大家做深入的解读。

4.3引用类型

4.3.1指针

Go语言中的指针相对于C语言中的指针简单了很多，对于开发者来说我们可以不必担心因为指针的错误使用而造成的各种各样奇奇怪怪的问题。

指针是一种指向变量内存地址的数据类型。一个指针变量可以指向任何一个变量的内存地址。

在Go语言中取地址的运算符为“&”，我将通过一个短例子查看地址与指针基本的样子。

例4.16：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

\*p = 100

fmt.Println(a) //输出：100

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：100

fmt.Println(\*p == a) //输出：true

}

上面例子中我们使用指针p指向了变量a的地址（&a），通过输出我们发现指针p的值就是变量a的地址，而\*p的值是变量a的值；当我们改变了\*p的值的时候，变量a的值也随之改变，因此我们可以认为在这种情况下，\*p等价于变量a。

我们刚才说到，每个变量都有一个地址，指针本身也是一个变量，那么指针是不是也有地址呢？指针是不是也可以被另一个指针指向呢？我们看下面的例子。

例4.17：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

fmt.Println(&p) //输出：0xc42000c028

//定义一个指向指针的指针

var ptr \*\*int

//将指针ptr指向指针p的地址

ptr = &p

fmt.Println(ptr) //输出：指针ptr的地址为0xc42000c028

fmt.Println(\*ptr) //输出：0xc420014100

fmt.Println(\*\*ptr) //输出：10

}

上例中我们定义了一个指向指针的指针ptr，并使其指向指针p的地址。

我们在输出的时候发现

ptr的值为指针p的地址；

\*ptr的值为指针p的值，也就是变量a的地址；

\*\*ptr的值为指针\*p的值，也就是变量a的值10。

4.3.2切片

切片也叫数组切片，通过上面对数组的介绍，我们知道数组的长度一 旦确定就不可以更改了。这种情况下是无法满足我们的真实需求的。

Go语言为我们提供了数组切片（slice）来补充数组的不足。

切片是一个可以随时动态扩充的存储空间，用于存放相同类型的数据；并且切片可以随意传递而不会导致所管理的元素被重复复制。

4.3.2.1创建切片

创建切片的方式有两种，一种是通过数组创建；一种是通过make命令创建。

例4.18：通过数组创建

func main() {

//定义一个数组

arr := [...]string{0: "zero", 1: "one", 2: "two", 3: "three", 5: "five", 4: "four"}

//通过数组生成一个切片

s := arr[:5]

//给切片追加数据

s = append(s, "six")

fmt.Println(s) //[zero one two three four six]

arr[0] = "111"

fmt.Println(s) //[111 one two three four six]

}

通过上面的代码，我们发现我们随后修改了数组的首元素的值，然后切片的首元素的值也跟着改变了，这是因为我们通过数组生成的切片，实际上是与数组本身共享了数据，没有为数组做数据的副本，所以在这种情况下，我们要引起我们的注意。

例4.19：通过make创建

func main() {

//创建一个预留长度为10，初始元素个数为5的数组切片

s := make([]string, 5, 10)

fmt.Println(s) //输出:[ ]

fmt.Println(cap(s)) //输出：10

fmt.Println(len(s)) //输出：5

s = append(s, "1")

s[0] = "zero"

s[1] = "one"

fmt.Println(s) //[zero one 1]

fmt.Println(cap(s)) //输出：10

fmt.Println(len(s)) //输出：6

}

通过上面的例子我们发现make也可以创建切片，创建切片的同时，系统也为我们创建了一个匿名数组，用于存储我们的数据。

4.3.2.2append（）函数

在上一节中，我们使用了append函数为切片增加了元素。这一节中，我们将对append函数进行讲解。

例4.20：

func main() {

//创建一个预留长度为10，初始原素个数为5的数组切片

s := make([]string, 5, 10)

fmt.Println(s) //输出:[ ]

fmt.Println(cap(s)) //输出：10

fmt.Println(len(s)) //输出：5

s = append(s, "1") //增加一个元素

s[0] = "zero" //修改第1个元素的值

s[1] = "one" //修改第2个元素的值

s[7] = "seven" //这里执行时错误：index out of range

fmt.Println(s) //[zero one 1]

fmt.Println(cap(s)) //输出：10

fmt.Println(len(s)) //输出：6

}

例子中我们创建了一个预留长度为10，初始元素为5个的切片，我们通过append()函数增加了第6个元素，这种情况下是编译通过的。然后我们通过下标对元素正常进行修改，当我们尝试对第8个元素进行修改的时候，我们得到一个下标越界的执行报错，也就是说，我们无法对未扩容的下标数据进行直接修改，必须通过append函数增加了以后，才可以可以变更里面的值。通过len()函数我们可以得到 当前切片的长度，通过cap()函数我们可以得到切片的预留总长度，这也是我们判断是否可以直接修改当前切片数据的依据。

既然append()函数是动态的增加元素，我们需要了解它是如何扩充存储空间的。请看以下代码。

例4.21：

func main() {

s := make([]string, 1, 3)

fmt.Println(cap(s))

fmt.Println(len(s))

s[0] = "0"

s1 := make([]string, 2, 2)

s1[0] = "1"

s1[1] = "2"

for i := 0; i < 5; i++ {

s = append(s, s1...)

fmt.Println(s)

fmt.Println("cap:", cap(s))

fmt.Println("len:", len(s))

}

}

以上代码输出结果为：

3

1

[0 1 2]

cap: 3

len: 3

[0 1 2 1 2]

cap: 6

len: 5

[0 1 2 1 2 1 2]

cap: 12

len: 7

[0 1 2 1 2 1 2 1 2]

cap: 12

len: 9

[0 1 2 1 2 1 2 1 2 1 2]

cap: 12

len: 11

我们看到cap每次增加都是原有长度的倍数，也就是说，每当写入的内容超过了原来的cap容量，就会复制一个与原有长度一样的长度追加到后面，再将数据写入到切片中。

4.3.2.3获取切片中的元素

切片为我们提供了更丰富的操作方法，我们除了可以获取单个元素之外，还可以同时获取多个连续的元素，我们先看一下具体的规则：

格式：sliceExample[startIndex:endIndex]

返回类型：slice

我们通过示例详细的了解一下这种方法如何使用

例4.22：

**func** main() {

s := [...]string{**0**: **"zero"**, **1**: **"one"**, **2**: **"two"**, **3**: **"three"**, **5**: **"five"**, **4**: **"four"**}

fmt.Println(s[**0**:**6**]) //输出：[zero one two three four five]

fmt.Println(s[**2**:**4**]) //输出：[two three]

fmt.Println(s[:**4**]) //输出：[zero one two three]

fmt.Println(s[**3**:]) //输出：[three four five]

fmt.Println(s[**0**:**8**]) //编译错误：./main.go:19:15: invalid slice index 8 (out of bounds for 6-element array)

通过例子我们看到使用这种方法可以得到一个新的切片，同时切片本身也是有长度限制的，一旦我们的取值范围超出界限，会得到编译错误，整个程序无法编译通过。上例中我们使用数组的方式创建切片，还有一各方法是使用make函数创建切片，这种两创建方式对于这种方法有什么不同，请读者自己进行尝试和挖掘，这里有一个小坑，暂且留给大家去填。

接下来我们将利用这个特性帮我们解决一些小问题。

例4.23：

**func** main() {

s := [...]string{**0**: **"zero"**, **1**: **"one"**, **2**: **"two"**, **3**: **"three"**, **5**: **"five"**, **4**: **"four"**}

fmt.Println(s[:**2**]) //输出：[zero one]

fmt.Println(s[**3**:]) //输出：[three four five]

s1 := **append**(s[:**2**], s[**3**:]...) //将[three four five] 连接到[zero one]后面生成新的切片

fmt.Println(s1) //输出：[zero one three four five]我们通过这种写法删除掉了2:"two"这个元素，由此我们实现了删除一个元素的功能

接下来，我们尝试入切片中插入数据。

例4.24

**func** main() {

//声明一个数组

arr := [...]string{**0**: **"zero"**, **1**: **"one"**, **2**: **"two"**, **3**: **"three"**, **5**: **"five"**, **4**: **"four"**}

//创建一个切片

s := **make**([]string, **1**, **10**)

//将数组追加的方式加入到切片中

s = **append**(s, arr[**0**:]...)

//提取插入点之后的数据

insertAfter := **append**([]string{}, s[**4**:]...)

//将要插入的数据写入切片

s = **append**(s[**0**:**4**], **"inserted"**)

//组合数据

s = **append**(s, insertAfter...)

fmt.Println(s) //输出：[ zero one two inserted three four five]

}

我们的思路是将切片分成了三部分：插入点之前、插入点、插入点之后。先将插入点之后的数据取出来，找中间变量存储，然后将插入点之前的数据与要插入的数据进行连接，最后再连接上已经取出来的原始的插入点之后的数据，就生成了新的切片，同时我们也将目标数据插入到了切片中。

4.3.3字典（map）

map是一种键/值（key/value）对形式的数据结构，它是一种未排序的集合，所有的key都是唯一的，通过key可以精准定位数据的位置，并对其进行相关操作。

在Go语言中，一个map就是一个哈希表的引用，所有的key的类型必须相同，value的类型也必须相同。要注意所有的的key都必须使用支持”==”比较运算符的类型。

需要注意的是浮点型最好不要使用在key中，一些误差会导致判断失败，影响程序的正常运行。value的类型则没有任何限制。

例4.25：

func main() {

//创建一个map,key为string类型，value为string类型

m := make(map[string]string)

//给map增加值

m["username"] = "admin"

m["sex"] = "man"

m["age"] = "20"

fmt.Println(m) //输出：map[username:admin sex:man age:20]

//删除键值

delete(m, "age")

fmt.Println(m) //输出：map[username:admin sex:man]

//查询键值是否存在

value, ok := m["username"]

if ok {

fmt.Println(value) //输出：value的结果，此处为admin

} else {

fmt.Println("nil") //输出：如果ok返回false，则输出nil

}

}

通过上面的例子我们可以了解到如何创建map，如何给map增加数据、修改数据、删除数据以及查询数据是否存在 。这一个示例，已经基本上包含了map的大部分操作。

4.3.4函数

在Go语言中，函数也作为一种数据类型。既然是数据类型那么函数也就可以当作一个值来传递和使用。

例4.26：

func main() {

//定义一个匿名函数，赋值给fun变量

fun := func(a, b int) int {

if a > b {

return a

}

return b

}

//调用匿名函数并保存返回值

fmt.Println(fun(5, 10)) //输出：10

fmt.Println(reflect.TypeOf(fun)) //输出:func()int,int)int

}

在上面例子中，我们定义了一个匿名函数，将其赋值给变量fun，我们在程序结尾输出类型的时候，看到类型是func类型，也就是函数类型；函数类型的调用 方式也与常规函数的方法一样，关于函数详细的使用方式，我们将在第六章函数中进行详细的解读。

4.3.5通道（channel）

通道(channel)是Go语言中为我们提供的用于在进程间通信的一种方式，我们使用channel在多个不同的进程间传递消息。

channel本身有类型的限制，一个channel只能传递一种类型的数据，在声明的时候就必须指定好要传递的类型。本书第16章并发编程中将会对channel的使用进行详细的解释说明，需要提前了解的读者请移步第16章

4.4接口类型

Go语言中的接口有别于其它编程语言中所提供的接口概念。

例如java中的接口，是作为开放标准存在的，用于约束对象之间的关系，而且接口必须强制实现。一旦声明了实现某个接口就必须要实现接口里的方法，也就是说当我们实现多个接口的时候，其实接口中有些方法我们在当前业务场景下未必使用的到。这种强制性的实现，为我们增加了很多烦恼。我们举个例子：一个按钮对象，我们可以拥有的方法包括单击、双击、长按等一系列操作，与之对应的我们就需要提供单击监听器（OnClickListener），双击监听器（OnDoubleClickListener），长按监听器（OnLongClickListener）；其中长按监听器有两个回调方法onLongClick()、onCancel()，很多的时候我们并不需要onCancel()，但是每次需要绑定监听的时候，我们都必须空实现这个方法，这样就会造成很多的代码冗余。

而在Go语言中，一个类只要实现了接口要求的所有函数，我们就认为这个类实现了这个接口；即使不实现所有的方法，我们也依旧可以正常的使用这个类。换言之就是说Go语言中，对于接口的实现不是强制声明的，而是你实现了所有的方法就是实现了接口，并不需要刻意的知道接口里有什么方法，取你所需就可以了。具体的例子，我们会在面向对象部分再进行深入的讲解。

以下例子仅供参考：

例4.27：

type ICar interface { //普通汽车

reduceGas()

}

type IECar interface { //电动汽车

reducePower()

}

type Car struct {

Name string

Price uint64

}

func (car \*Car) reduceGas() {

fmt.Println("this is Gas Car")

}

func (car \*Car) reducePower() {

fmt.Println("this is Electric Car")

}

func main() {

var gasCar ICar = new(Car) //这是一辆普通汽油汽车

var eCar IECar = new(Car) //这是一辆电动汽车

var car Car //这是一辆混合动力汽车

gasCar.reduceGas()

eCar.reducePower()

car.reduceGas()

car.reducePower()

}