复合数据类型

数组

数组是具有固定长度且拥有零个或多个相同数据类型元素的有序序列

- 数组大小一旦确定，无法进行更改
- 在Go中，[3]int与[2]int是不同的类型

```
// NewArray returns a new fixed-length array type.
func NewArray(elem *Type, bound int64) *Type {
    if bound < 0 {
        Fatalf("NewArray: invalid bound %v", bound)
    }
    t := New(TARRAY)
    t.Extra = &Array{Elem: elem, Bound: bound}
    t.SetNotInHeap(elem.NotInHeap())
    return t
}
```

可以看到，元素类型和数组大小共同决定了数组的类型
source: cmd/compile/internal/types.NewArray

定义

- 声明 ⊙ var a = [2]int ⊙ 此时Go会将数组直接初始化，即此时a为[0, 0]
- 字面量声明与赋值
  - q := [3]int{1, 2, 3}
  - q := [...]{1, 2, 3, 4} ⊙ 数组长度自动推断

将数组传递给某函数时，会使用按值传递，而非引用传递，这一点与其它大多数语言均不相同

有时候会利用数组的特性做一些迭代工作，例如定义一个[5][0]int，而后对其进行下标迭代，已达到for range times的作用，要比 for i:=0;i<n;i++ 的效率稍好一些

```
var times [5][0]int
for range times {
    fmt.Println("Hello~")
}
```

二维数组[5][0]int，虽然一维数组有长度，但是二维数组长度为0，所以占用内存空间为0。在无需付出额外的内存空间情况下，能够实现高效的times次数迭代

slice(切片)

slice(切片)表示一个拥有相同类型元素的可变长度有序序列，其底层实现为数组，通过数组容量已满时的内存拷贝来实现slice的动态扩容

slice本质上是一个结构体，其中包含了指向底层数组的指针以及附属属性，所以是一种非常轻量的数据结构

- data uintptr ⊙ 指向数组的第一个可以从slice中访问的元素，但不一定是数组的第一个元素
- len int ⊙ slice中元素个数
- cap int ⊙ slice的容量，其大小通常是从slice的起始元素到底层数组的最后一个元素间元素的个数

slice可以看作是底层数组的一种view，但由于slice中的指针指向数组中某个元素，故对slice中的某个元素进行修改，其实修改的是底层数组 ⊙ 即slice非线程安全

定义

- 声明 ⊙ var s []string ⊙ 此时Go会将s进行初始化，但与数组不同，此时s为nil，因为s无任何的底层数组支持 ⊙ len(s)以及cap(s)均为0
- 字面量声明与赋值 ⊙ s := []string{"smart", "mario"} ⊙ 此时len(s)以及cap(s)的值均为2
- 使用make函数
  - make([]T, len) 此时slice的容量与len相等
    - Go会自动为用户创建一个长度为len的底层数组，并将数组初始化为类型T的零值
  - make([]T, len, cap)

append函数

原型: func append(slice []Type, elems ...Type) []Type

内置函数append用来将元素追加至slice的后面 ⊙ s = append(s, T) ⊙ 由于在追加元素的过程中，原有底层数组的容量可能不够，需要进行扩容，此时Go将开辟一块儿更大的内存空间，将原有数组内容拷贝至新数组，并返回扩容后数组对应的slice，故append函数需要主动接收

由于append函数接收不定长参数(elems)，所以可以一次性追加多个元素 ⊙ s = append(s, 1, 2, 3)

- 同时也可以追加另一个slice，通过slice"拆包"实现extend ⊙ s = append(s, k...)

对现有切片进行append可能出引发BUG

对切片执行append()时，如果追加元素数量不足以使得底层数组扩容时，返回的切片与原有切片共享同一个底层数组

```
// 以下片段执行后，slice、A1以及A2的值是什么？
slice := []int{1, 2, 3, 4, 5, 6}
A1 := slice[2:3]
A2 = append(A1, 7, 8, 9)
```

```
slice: [1 2 3 7 8 9]
A1: [3]
A2: [3 7 8 9]
```

```
// 以下片段执行后，slice、A1以及A2的值是什么？
slice := []int{1, 2, 3, 4, 5, 6}
A1 := slice[2:3]
A2 = append(A1, 7, 8, 9, 10)
```

```
slice: [1 2 3 4 5 6]
A1: [3]
A2: [3 7 8 9 10]
```

所以，当函数参数为slice类型时，在函数内部，不对slice进行append操作，除非具有明确的目的。否则会对原有slice造成难以预估的影响

Golang为数不多的陷阱之一

copy

原型: func copy(dst, src []Type) int

内置函数copy用来为两个拥有相同类型元素的slice复制元素 ⊙ copy(s[len(s):], k) ⊙ 相当于s = append(s, k...)

使用append+copy实现slice的insert,pop以及remove

- insert
  - slice插入元素首要步骤就是为slice添加一个扩展位 ⊙ s = append(s, T)
  - 而后移动下标i后面的所有元素 ⊙ copy(s[i+1:], s[i:])
  - 最终于下标i位置填充要插入的元素 ⊙ s[i] = T
- pop
  - 删除下标为index(index<len(s)-1)的元素，可直接使用copy函数 ⊙ copy(s[i:], s[i+1:])
  - 对于指针类型的slice，使用copy后，最后一个元素应设置为nil，避免内存泄露 ⊙ s[len(s)-1] = nil

```
temp := s[:0]
for _, v := range s {
    if !f(v) {
        temp = append(temp, v)
    }
}
// 对于指针等类型，可将底层数组指针设置为nil，无其余引用时进行gc回收
for i := len(temp); i < len(s); i++ {
    s[i] = nil
}
```

- remove ⊙ **删除slice中某一类元素，可以借助s[:0]这一空slice来对slice进行就地修改** ⊙

Go没有泛型的概念，所以针对不同的slice需要定义看起来完全相同的方法，除了函数参数不同以外

map

字典或者散列表，与Python和Java一样，作为内置数据结构实现

定义

- 声明 ⊙ var m map[stirng]string
  - 此时m为nil，且其key的数量为0
  - 使用此类声明的map只能作为接收者，对m的元素设置将导致错误 ⊙ m["name" = smart] //panic
- 字面量声明并赋值 ⊙ m := map[string]int{"smart": 10}
- 使用make函数 ⊙ make(map[string]int) ⊙ 此时将创建一个空map，但不是nil，所以make创建map使用更加普遍

迭代 ⊙ for key, value := range m ⊙ 使用`_`忽略key或者是value

获取 ⊙ 当使用map[key]获取value值时，将会有2个返回值: value以及isExist标识位，类型为布尔值。为false时表示该map中不存在该key ⊙ value, ok := m["smart"]

Go并没有内置的set(集合)实现，所以只能使用map实现集合

结构体

结构体是将零个或多个任意类型的命名变量组合在一起的聚合数据类型，是Go组合数据类型的究极体

结构体中的变量称为结构体变量

```
type User struct {
    Nickname string
    Gender uint8
    Mobile string
}
```

Go结构体与C结构体基本类型，均可包含结构体函数。可以类比结构体为Java(Python)中的类，毕竟这是Go提供的唯一封装方式 ⊙

初始化

- 顺序初始化 ⊙ user := User{"smart", 2, "13666666666"} ⊙ 必须对所有的结构体变量均进行赋值
- 命名初始化 ⊙ user := User{Nickname: "smart", Gender: 2, Mobile: "13666666666"} ⊙ 可选择地对某些结构体变量进行赋值，并且更加易读

结构体嵌套与匿名成员

- 结构体嵌套 ⊙ 一个结构体可以包含另一个命名结构体

```
type Address struct {
    DetailAddress string
}

type User struct {
    Nickname string
    Address
}
```

- 匿名成员
  - 结构体嵌套的一种特殊使用方式 ⊙ ⊙ ⊙
  - 如右侧所示，此时可直接使用user.DetailAddress来访问Address匿名结构体中的内容
  - 但初始化时则必须一层一层的初始 ⊙ user := User{Address: Address{"广东深圳"}, Nickname: "smart"}

JSON

在Go中，JSON的准确反序列化必须使用结构体进行接收，并通过field tag控制序列化/反序列化的元信息

- 序列化 ⊙ json.Marshal ⊙ func Marshal(v interface{}) ([]byte, error) ⊙ 需要注意的是,Marshal返回的是字节数组
- 反序列化 ⊙ json.Unmarshal ⊙ func Unmarshal(data []byte, v interface{}) error

struct field tag

- serialization name ⊙ 由于结构体变量必须首字母大写才可进行序列化，但json格式通常采用小写字母+下划线的方式，故需要指定序列化的名称 ⊙ UserAddress string `json:"user_address"`
- omitempty ⊙ 当结构体变量的值为空或者零值时，添加omitempty则不会在序列化的结果中出现 ⊙ UserAddress string `json:"user_address, omitempty"`
- not serialization ⊙ 控制结构体变量是否需要序列化至结果中 ⊙ UserAddress string `json:"-"`

流式解码器

流式解码器可依次从字节流中解码出多个JSON实体

- json.Encoder ⊙ 通常采用json.NewEncoder()方法生成Encoder结构体指针
- json.Decoder ⊙ 通常采用json.NewDecoder()方法生成Decoder结构体指针

方法均接收io.Reader类型参数

控制结构体的序列化/反序列化方式

Go提供了MarshalJSON以及UnmarshalJSON两个接口以控制结构体的序列化以及反序列化行为

- MarshalJSON
  - 原型: MarshalJSON() ([]byte, error)
  - 结构体可实现该方法用于控制序列化成JSON对象的行为
- UnmarshalJSON
  - 原型: UnmarshalJSON([]byte) error
  - 结构体可实现该方法用于控制JSON对象反序列化成结构体对象的行为

关于时间

Go语言中time.Time是以RFC 3339为标准建立的，包括了时区以及小数点秒数，如2020-02-05T11:29:59.986677232+08:00

然而绝大多数应用并不会采用如此精细的时间表示，通常使用Y-m-d H:M:S的方式，然而Go无法将此格式时间解析成time.Time类型

所以，我们需要添加一层抽象，封装time.Time结构体，并实现上述的MarshalJSON以及UnmarshalJSON方法 ⊙ 由于Go无继承语法，所以使用type的方式扩展某一内置类型

```
type JSONTime time.Time

func (p *JSONTime) UnmarshalJSON(data []byte) error {
    local, err := time.ParseInLocation(`"2006-01-02 15:04:05"`, string(data), time.Local)
    *p = JSONTime(local)
    return err
}

func (p JSONTime) MarshalJSON() ([]byte, error) {
    stamp := fmt.Sprintf("\"%s\"", time.Time(p).Format("2006-01-02 15:04:05"))
    return []byte(stamp), nil
}
```