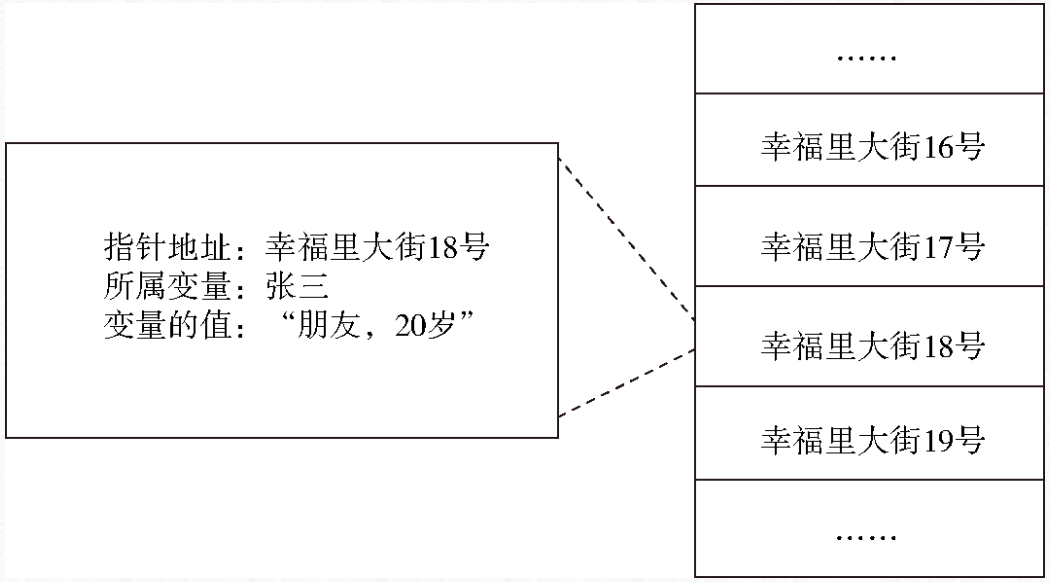
Go 语言从入门到项目实战

第3章高级数据类型

3.1 指针

- 指针本质上就是一个内存地址，这个内存地址当中存储着一个变量的值。



3.1 指针

- 声明格式：

  var var_name *var-type

- 示例：

  var intPointer *int

3.1 指针

```
/*指针类型
*/
func main() {
	var numA int=10			//声明numA变量，类型为int，值为10
	var intPointer *int		//声明intPointer变量，类型为指针变量
	intPointer=&numA		//获取numA的内存地址，将该地址赋值给intPointer
}
```

3.1 指针

```
/* 访问指针变量表示的变量的值
*/
func main() {
        var numA int=10              //声明numA变量，类型为int，值为10
        var intPointer *int          //声明intPointer变量，类型为指针变量
        intPointer=&numA             //获取numA的内存地址，将该地址赋值给intPointer
        fmt.Println(intPointer)      //输出intPointer的值
        fmt.Println(*intPointer)     //通过intPointer的值，获得numA的值，并输出
}
```

3.1 指针

```
/*空指针
*/
func main() {
    var intPointer *int              //声明intPointer变量，类型为指针变量
    fmt.Println(intPointer)          //输出intPointer的值
}
```

3.1 指针

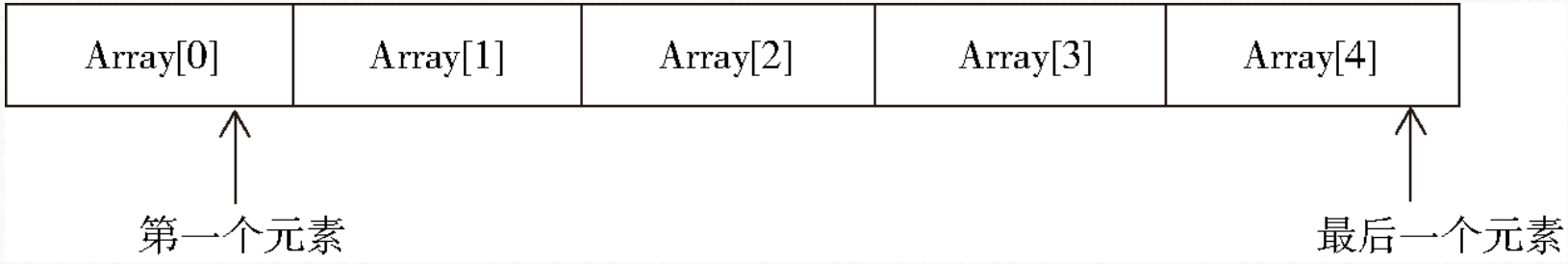
- 指向指针的指针变量
  - 声明格式：

    var var_name **var-type
  - 示例：

```
func main() {
        var c int=100        //声明整型变量c，值为100
        var b *int=&c        //声明指针变量 b，用来存放整型变量c的地址
        var a **int=&b       //声明指针变量a，用来存放指针变量b的地址
        fmt.Println(**a)     //输出a所表示的变量的值所表示的变量的值（b所表示的变量的值，即整型变量c的值）
}
```

3.2 数组

- 数组是相同数据类型的且长度固定的有序数据项序列；



3.2 数组

- 声明格式：

  var variable_name [SIZE] variable_type

- 示例：

```
//通过var关键字声明数组arrZeroInt
//仅声明数组，编译器会自动初始化数组的元素为数据项类型的零值
var arrZeroInt = [4]int32{}
fmt.Println("arrZeroInt=", arrZeroInt)
//对数组变量使用Go语言内置的len()函数来获取数组长度
fmt.Println("arrZeroInt的长度为", len(arrZeroInt))
```

3.2 数组

```
//指定“元素索引值:元素值”
var arrString := [6]string{0: "张三", 3: "李四"}
//指定数组第一个元素为99,第6个元素即索引值为5的元素值为128
var arrInt64 := [7]int64{99, 5: 128}
//数组长度由编译器自动推导，数组长度就是给出的最大索引值+1
var arrAutoInit := [...]int{10, 5: 100}
//通过new关键字声明一个整型数组，此时的arrPointer的数据类型为数组指针
var arrPointer := new([20]int)
```

3.2 数组

```
arrInt := [6]int{11, 15, 25, 23, 19, 78}
//获取arrInt的第一个元素
fmt.Println("arrInt的第一个元素为", arrInt[0])
//可以通过Go语言标准库的reflect包中的TypeOf函数查看数组arrInt的数据类型
fmt.Println("arrInt的数据类型为", reflect.TypeOf(arrInt))
```

3.2 数组

```
//定义长度由编译器自动推导，数据项类型为float64的arrFloat64
arrAutoFloat64 := [...]float64{1.31, 3.14, 5.28, 6.78}
fmt.Println("arrAutoFloat64=", arrAutoFloat64)
fmt.Println("arrAutoFloat64的长度为", len(arrAutoFloat64))
```

3.2 数组

- 二维数组：可将其看作是数组元素为一维数组的一维数组。

```
arrMulti := [2][3]int{{1, 2, 3}, {4, 5, 6}}
fmt.Println("arrMulti=", arrMulti)
fmt.Println("arrMulti的长度为", len(arrMulti)
fmt.Println("arrMulti[0][1]=", arrMulti[0][1])
arrMulti[0][1]=100
fmt.Println("arrMulti=", arrMulti)
fmt.Println("arrMulti的数据类型为", reflect.TypeOf(arrMulti))
```

3.3 切片

- 切片：对数组（array）的抽象。
- 声明方式：

```
var variable_name []variable_type
make([]T, length, capacity)
arr[startIndex:endIndex]
[]type {value}
```

- 示例：

```
//声明并使用make()函数初始化arrZeroInt切片变量
var arrZeroInt=make([]int32, 4, 6)
fmt.Println("arrZeroInt=", arrZeroInt)
//对切片变量使用Go语言内置的len()函数来获取切片长度
fmt.Println("arrZeroInt的长度为", len(arrZeroInt))
//对切片变量使用Go语言内置的cap()函数来获取切片容量
fmt.Println("arrZeroInt的容量为", cap(arrZeroInt))
```

3.3 切片

- 示例：

```
//声明arrInt数组变量，并初始化
var arrInt=[6]int{11, 15, 25, 23, 19, 78}
fmt.Println("arrInt=", arrInt)
fmt.Println("arrInt的长度为", len(arrInt))
//声明并节选arrInt数组用作初始化sliceInt切片
var sliceInt=arrInt[1:5]
fmt.Println("sliceInt=", sliceInt)
//对切片变量使用Go语言内置的len()函数来获取切片长度
fmt.Println("sliceInt的长度为", len(sliceInt))
//对切片变量使用Go语言内置的cap()函数来获取切片容量
fmt.Println("sliceInt的容量为", cap(sliceInt))
```

3.3 切片

- 示例：

```
var sliceInt=[]int{1,2,3,4,5}
fmt.Println("sliceInt=", sliceInt)
//对切片变量使用Go语言内置的len()函数来获取切片长度
fmt.Println("sliceInt的长度为", len(sliceInt))
//对切片变量使用Go语言内置的cap()函数来获取切片容量
fmt.Println("sliceInt的容量为", cap(sliceInt))
```

3.3 切片

- append()函数

```
//向切片中追加1个元素
sliceInt=append(sliceInt,1)
//向切片中追加多个元素
sliceInt=append(sliceInt,2,3,4)
//向切片中追加另一个切片
sliceInt=append(sliceInt,[]int{5,6,7}...)
```

3.3 切片

- copy()函数

```
//声明切片变量sliceA和slice2B并赋初始值
sliceA := []int{1, 2, 3, 4, 5}
sliceB := []int{5, 4, 3}
// 只会复制sliceA的前3个元素到sliceB中
copy(sliceB, sliceA)
// 只会复制sliceB的3个元素到sliceA的前3个位置
copy(sliceA, sliceB)
```

3.4 集合

- 集合（Map）是一种特殊的数据结构，一种键值对（Pair）的无序集合。一个键值对包含键（Key）和值（Value），所以这个结构也称为字典。
- 声明格式：

  var variable_name map[key_type]value_type

  make([]T, length)

  给定具体值初始化

3.4 集合

- 示例：

```
//声明map型变量mapValueA
var mapValueA map[string]string
```

3.4 集合

- 示例：

```
//使用make()函数初始化集合
mapValueB := make(map[string]string)
mapValueB["key1"] = "value1"
mapValueB["key2"] = "value2"
```

3.4 集合

- 示例：

```
//直接给定具体值初始化集合
var mapValueC = map[string]string{"key1": "value1", "key2": "value2"}
```

3.4 集合

- 元素的检索：

  map[key_value]

- 示例：

```
//直接给定具体值初始化集合
var mapValueC = map[string]string{"key1": "value1", "key2": "value2"}
fmt.Println("mapValueC的值为：", mapValueC)
fmt.Println("mapValueC的长度为：", len(mapValueC))
fmt.Println("获取键key1的值", mapValueC["key1"])
fmt.Println("获取键key3的值", mapValueC["key3"])
fmt.Println(mapValueC["key3"]=="")
```

3.4 集合

- 添加、删除和修改元素 ：

```
//直接给定具体值初始化集合
var mapValueD = map[string]string{
                "key1": "value1",
                "key2": "value2",
                "key3": "value3"}
fmt.Println("mapValueD的值为：", mapValueD)
//向mapValueD中添加元素
mapValueD["key4"]="value4"
mapValueD["key5"]="value5"
fmt.Println("mapValueD的值为：", mapValueD)
//删除键为key5的键值对
delete(mapValueD,"key5")
fmt.Println("mapValueD的值为：", mapValueD)
//修改键为key1的值
mapValueD["key1"]="VALUE1"
fmt.Println("mapValueD的值为：", mapValueD)
```

3.5 结构体

- 结构体（Struct）是通过自定义的方式形成的新的复合数据类型，由零个或多个任意类型的值聚合而成，每个值都可以称为结构体的成员（也被称为字段）。
- 这些字段有以下特性：
  - 字段拥有自己的类型和值；
  - 字段名必须唯一；
  - 字段的类型也可以是结构体，甚至是字段所在结构体的类型。

3.5 结构体

- 声明格式：

```
type struct_name struct {
    field_name definition
    field_name definition
    ...
}
```

3.5 结构体

- 示例：

```
//结构体Person
type Person struct {
//string型字段name
  name string
  //int型字段age
  age int
  //int型字段gender，0表示男，1表示女
  gender int
}
```

3.5 结构体

```
//实例化Person类型变量alice
var alice Person
alice.name="alice"
alice.gender=1
alice.age=25
//输出alice各字段的值
fmt.Println("姓名：",alice.name)
fmt.Println("性别：",alice.gender)
fmt.Println("年龄：",alice.age)
```