

4.1.4 多维数组

数组本身只有一个维度，不过可以组合多个数组创建多维数组。多维数组很容易管理具有父子关系的数据或者与坐标系相关联的数据。声明二维数组的示例如代码清单 4-10 所示。

代码清单 4-10 声明二维数组

```
// 声明一个二维整型数组，两个维度分别存储 4 个元素和 2 个元素
var array [4][2]int

// 使用数组字面量来声明并初始化一个二维整型数组
array := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}

// 声明并初始化外层数组中索引为 1 个和 3 的元素
array := [4][2]int{1: {20, 21}, 3: {40, 41}}

// 声明并初始化外层数组和内层数组的单个元素
array := [4][2]int{1: {0: 20}, 3: {1: 41}}
```

图 4-8 展示了代码清单 4-10 中声明的二维数组在每次声明并初始化后包含的值。

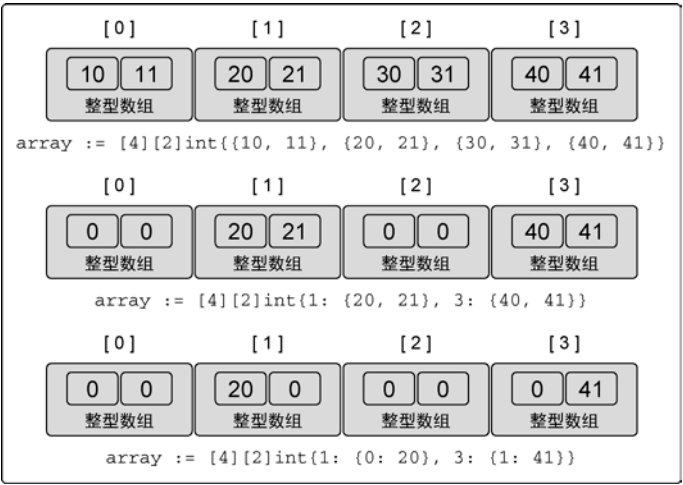


图 4-8 二维数组及其外层数组和内层数组的值

为了访问单个元素，需要反复组合使用 `[]` 运算符，如代码清单 4-11 所示。

代码清单 4-11 访问二维数组的元素

```
// 声明一个 2×2 的二维整型数组
var array [2][2]int

// 设置每个元素的整型值
array[0][0] = 10
```

```
array[0][1] = 20
array[1][0] = 30
array[1][1] = 40
```

只要类型一致，就可以将多维数组互相赋值，如代码清单 4-12 所示。多维数组的类型包括每一维度的长度以及最终存储在元素中的数据的类型。

代码清单 4-12 同样类型的多维数组赋值

```
// 声明两个不同的二维整型数组
var array1 [2][2]int
var array2 [2][2]int

// 为每个元素赋值
array2[0][0] = 10
array2[0][1] = 20
array2[1][0] = 30
array2[1][1] = 40

// 将 array2 的值复制给 array1
array1 = array2
```

因为每个数组都是一个值，所以可以独立复制某个维度，如代码清单 4-13 所示。

代码清单 4-13 使用索引为多维数组赋值

```
// 将 array1 的索引为 1 的维度复制到一个同类型的新数组里
var array3 [2]int = array1[1]

// 将外层数组的索引为 1、内层数组的索引为 0 的整型值复制到新的整型变量里
var value int = array1[1][0]
```

4.1.5 在函数间传递数组

根据内存和性能来看，在函数间传递数组是一个开销很大的操作。在函数之间传递变量时，总是以值的方式传递的。如果这个变量是一个数组，意味着整个数组，不管有多长，都会完整复制，并传递给函数。

为了考察这个操作，我们来创建一个包含 100 万个 int 类型元素的数组。在 64 位架构上，这将需要 800 万字节，即 8 MB 的内存。如果声明了这种大小的数组，并将其传递给函数，会发生什么呢？如代码清单 4-14 所示。

代码清单 4-14 使用值传递，在函数间传递大数组

```
// 声明一个需要 8 MB 的数组
var array [1e6]int

// 将数组传递给函数 foo
foo(array)
```



```
// 函数 foo 接受一个 100 万个整型值的数组
func foo(array [1e6]int) {
    ...
}
```

每次函数 `foo` 被调用时，必须在栈上分配 8 MB 的内存。之后，整个数组的值（8 MB 的内存）被复制到刚分配的内存里。虽然 Go 语言自己会处理这个复制操作，不过还有一种更好且更有效的方法来处理这个操作。可以只传入指向数组的指针，这样只需要复制 8 字节的数据而不是 8 MB 的内存数据到栈上，如代码清单 4-15 所示。

代码清单 4-15 使用指针在函数间传递大数组

```
// 分配一个需要 8 MB 的数组
var array [1e6]int

// 将数组的地址传递给函数 foo
foo(&array)

// 函数 foo 接受一个指向 100 万个整型值的数组的指针
func foo(array *[1e6]int) {
    ...
}
```

这次函数 `foo` 接受一个指向 100 万个整型值的数组的指针。现在将数组的地址传入函数，只需要在栈上分配 8 字节的内存给指针就可以。

这个操作会更有效地利用内存，性能也更好。不过要意识到，因为现在传递的是指针，所以如果改变指针指向的值，会改变共享的内存。如你所见，使用切片能更好地处理这类共享问题。

4.2 切片的内部实现和基础功能

切片是一种数据结构，这种数据结构便于使用和管理数据集合。切片是围绕动态数组的概念构建的，可以按需自动增长和缩小。切片的动态增长是通过内置函数 `append` 来实现的。这个函数可以快速且高效地增长切片。还可以通过对切片再次切片来缩小一个切片的大小。因为切片的底层内存也是在连续块中分配的，所以切片还能获得索引、迭代以及为垃圾回收优化的好处。

4.2.1 内部实现

切片是一个很小的对象，对底层数组进行了抽象，并提供相关的操作方法。切片有 3 个字段的数据结构，这些数据结构包含 Go 语言需要操作底层数组的元数据（见图 4-9）。

这 3 个字段分别是指向底层数组的指针、切片访问的元素的个数（即长度）和切片允许增长到的元素个数（即容量）。后面会进一步讲解长度和容量的区别。

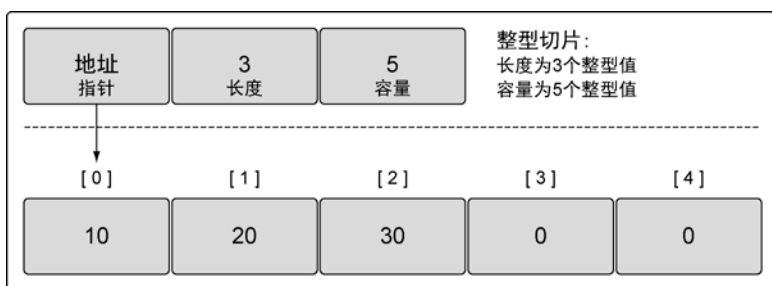


图 4-9 切片内部实现：底层数组

4.2.2 创建和初始化

Go 语言中有几种方法可以创建和初始化切片。是否能提前知道切片需要的容量通常会决定要如何创建切片。

1. make 和切片字面量

一种创建切片的方法是使用内置的 make 函数。当使用 make 时，需要传入一个参数，指定切片的长度，如代码清单 4-16 所示。

代码清单 4-16 使用长度声明一个字符串切片

```
// 创建一个字符串切片
// 其长度和容量都是 5 个元素
slice := make([]string, 5)
```

如果只指定长度，那么切片的容量和长度相等。也可以分别指定长度和容量，如代码清单 4-17 所示。

代码清单 4-17 使用长度和容量声明整型切片

```
// 创建一个整型切片
// 其长度为 3 个元素，容量为 5 个元素
slice := make([]int, 3, 5)
```

分别指定长度和容量时，创建的切片，底层数组的长度是指定的容量，但是初始化后并不能访问所有的数组元素。图 4-9 描述了代码清单 4-17 里声明的整型切片在初始化并存入一些值后的样子。

代码清单 4-17 中的切片可以访问 3 个元素，而底层数组拥有 5 个元素。剩余的 2 个元素可以在后期操作中合并到切片，可以通过切片访问这些元素。如果基于这个切片创建新的切片，新切片会和原有切片共享底层数组，也能通过后期操作来访问多余容量的元素。

不允许创建容量小于长度的切片，如代码清单 4-18 所示。

代码清单 4-18 容量小于长度的切片会在编译时报错

```
// 创建一个整型切片
// 使其长度大于容量
slice := make([]int, 5, 3)

Compiler Error:
len larger than cap in make([]int)
```

另一种常用的创建切片的方法是使用切片字面量，如代码清单 4-19 所示。这种方法和创建数组类似，只是不需要指定[]运算符里的值。初始的长度和容量会基于初始化时提供的元素的个数确定。

代码清单 4-19 通过切片字面量来声明切片

```
// 创建字符串切片
// 其长度和容量都是 5 个元素
slice := []string{"Red", "Blue", "Green", "Yellow", "Pink"}

// 创建一个整型切片
// 其长度和容量都是 3 个元素
slice := []int{10, 20, 30}
```

当使用切片字面量时，可以设置初始长度和容量。要做的就是初始化时给出所需的长度和容量作为索引。代码清单 4-20 中的语法展示了如何创建长度和容量都是 100 个元素的切片。

代码清单 4-20 使用索引声明切片

```
// 创建字符串切片
// 使用空字符串初始化第 100 个元素
slice := []string{99: ""}
```

记住，如果在[]运算符里指定了一个值，那么创建的就是数组而不是切片。只有不指定值的时候，才会创建切片，如代码清单 4-21 所示。

代码清单 4-21 声明数组和声明切片的不同

```
// 创建有 3 个元素的整型数组
array := [3]int{10, 20, 30}

// 创建长度和容量都是 3 的整型切片
slice := []int{10, 20, 30}
```

2. nil 和空切片

有时，程序可能需要声明一个值为 nil 的切片（也称 nil 切片）。只要在声明时不做任何初

始化，就会创建一个 nil 切片，如代码清单 4-22 所示。

代码清单 4-22 创建 nil 切片

```
// 创建 nil 整型切片
var slice []int
```

在 Go 语言里，nil 切片是很常见的创建切片的方法。nil 切片可以用于很多标准库和内置函数。在需要描述一个不存在的切片时，nil 切片会很好用。例如，函数要求返回一个切片但是发生异常的时候（见图 4-10）。

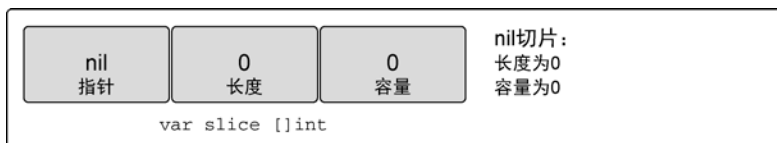


图 4-10 nil 切片的表示

利用初始化，通过声明一个切片可以创建一个空切片，如代码清单 4-23 所示。

代码清单 4-23 声明空切片

```
// 使用 make 创建空的整型切片
slice := make([]int, 0)

// 使用切片字面量创建空的整型切片
slice := []int{}
```

空切片在底层数组包含 0 个元素，也没有分配任何存储空间。想表示空集合时空切片很有用，例如，数据库查询返回 0 个查询结果时（见图 4-11）。

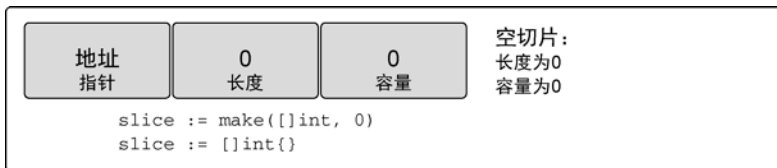


图 4-11 空切片的表示

不管是使用 nil 切片还是空切片，对其调用内置函数 append、len 和 cap 的效果都是一样的。

4.2.3 使用切片

现在知道了什么是切片，也知道如何创建切片，来看看如何在程序里使用切片。

1. 赋值和切片

对切片里某个索引指向的元素赋值和对数组里某个索引指向的元素赋值的方法完全一样。使用[]操作符就可以改变某个元素的值，如代码清单 4-24 所示。

代码清单 4-24 使用切片字面量来声明切片

```
// 创建一个整型切片
// 其容量和长度都是 5 个元素
slice := []int{10, 20, 30, 40, 50}

// 改变索引为 1 的元素的值
slice[1] = 25
```

切片之所以被称为切片，是因为创建一个新的切片就是把底层数组切出一部分，如代码清单 4-25 所示。

代码清单 4-25 使用切片创建切片

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}

// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]
```

执行完代码清单 4-25 中的切片动作后，我们有了两个切片，它们共享同一段底层数组，但通过不同的切片会看到底层数组的不同部分（见图 4-12）。

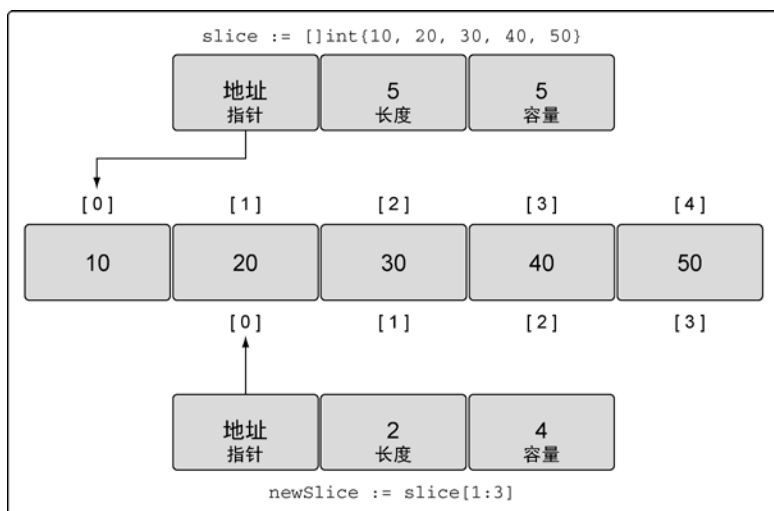


图 4-12 共享同一底层数组的两个切片

第一个切片 `slice` 能够看到底层数组全部 5 个元素的容量，不过之后的 `newSlice` 就看不到。对于 `newSlice`，底层数组的容量只有 4 个元素。`newSlice` 无法访问到它所指向的底层数组的第一个元素之前的部分。所以，对 `newSlice` 来说，之前的那些元素就是不存在的。

使用代码清单 4-26 所示的公式，可以计算出任意切片的长度和容量。

代码清单 4-26 如何计算长度和容量

对底层数组容量是 `k` 的切片 `slice[i:j]` 来说

长度： $j - i$

容量： $k - i$

对 `newSlice` 应用这个公式就能得到代码清单 4-27 所示的数字。

代码清单 4-27 计算新的长度和容量

对底层数组容量是 5 的切片 `slice[1:3]` 来说

长度： $3 - 1 = 2$

容量： $5 - 1 = 4$

可以用另一种方法来描述这几个值。第一个值表示新切片开始的元素的索引位置，这个例子中是 1。第二个值表示开始的索引位置（1），加上希望包含的元素的个数（2），1+2 的结果是 3，所以第二个值就是 3。容量是该与切片相关联的所有元素的数量。

需要记住的是，现在两个切片共享同一个底层数组。如果一个切片修改了该底层数组的共享部分，另一个切片也能感知到，如代码清单 4-28 所示。

代码清单 4-28 修改切片内容可能导致的结果

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}

// 创建一个新切片
// 其长度是 2 个元素，容量是 4 个元素
newSlice := slice[1:3]

// 修改 newSlice 索引为 1 的元素
// 同时也修改了原来的 slice 的索引为 2 的元素
newSlice[1] = 35
```

把 35 赋值给 `newSlice` 的第二个元素（索引为 1 的元素）的同时也是在修改原来的 `slice` 的第 3 个元素（索引为 2 的元素）（见图 4-13）。

切片只能访问到其长度内的元素。试图访问超出其长度的元素将会导致语言运行时异常，如代码清单 4-29 所示。与切片的容量相关联的元素只能用于增长切片。在使用这部分元素前，必须将其合并到切片的长度里。

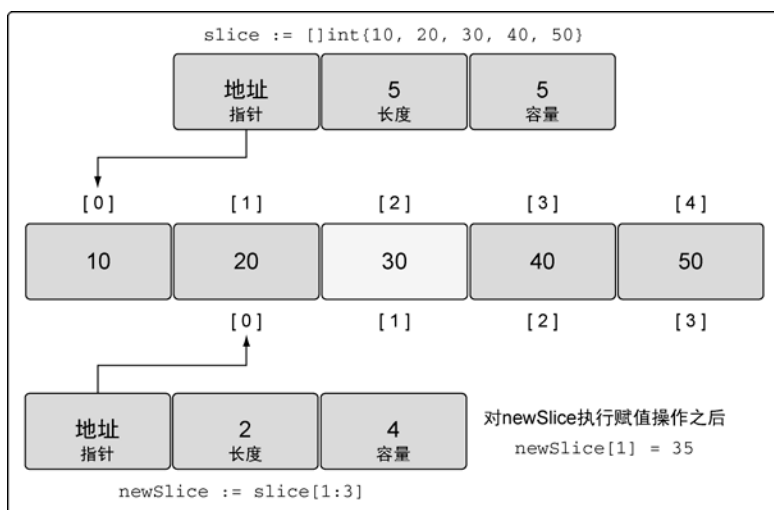


图 4-13 赋值操作之后的底层数组

代码清单 4-29 表示索引越界的语言运行时错误

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}

// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]

// 修改 newSlice 索引为 3 的元素
// 这个元素对于 newSlice 来说并不存在
newSlice[3] = 45

Runtime Exception:
panic: runtime error: index out of range
```

切片有额外的容量是很好，但是如果不能把这些容量合并到切片的长度里，这些容量就没有用处。好在可以用 Go 语言的内置函数 `append` 来做这种合并很容易。

2. 切片增长

相对于数组而言，使用切片的一个好处是，可以按需增加切片的容量。Go 语言内置的 `append` 函数会处理增加长度时的所有操作细节。

要使用 `append`，需要一个被操作的切片和一个要追加的值，如代码清单 4-30 所示。当 `append` 调用返回时，会返回一个包含修改结果的新切片。函数 `append` 总是会增加新切片的长度，而容量有可能会改变，也可能不会改变，这取决于被操作的切片的可用容量。

代码清单 4-30 使用 append 向切片增加元素

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}

// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]

// 使用原有的容量来分配一个新元素
// 将新元素赋值为 60
newSlice = append(newSlice, 60)
```

当代码清单 4-30 中的 append 操作完成后，两个切片和底层数组的布局如图 4-14 所示。

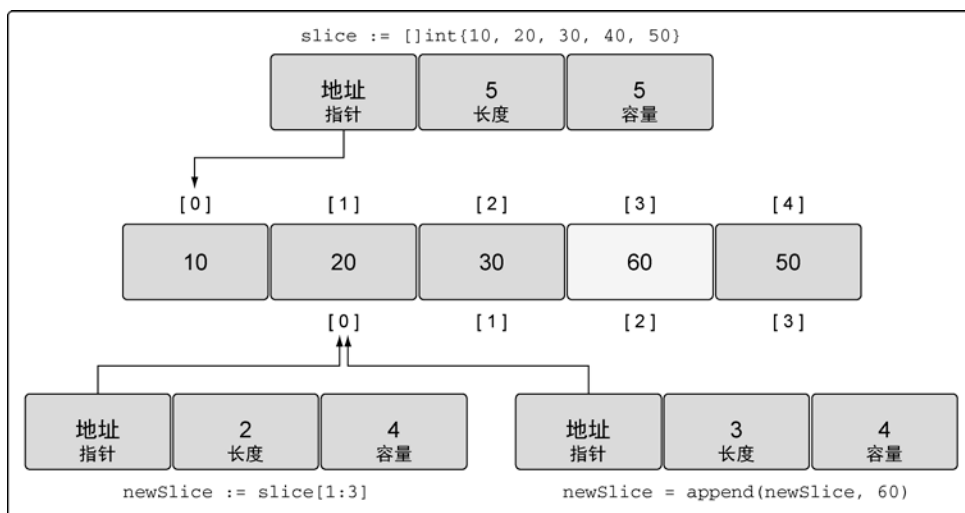


图 4-14 append 操作之后的底层数组

因为 newSlice 在底层数组里还有额外的容量可用，append 操作将可用的元素合并到切片的长度，并对其进行赋值。由于和原始的 slice 共享同一个底层数组，slice 中索引为 3 的元素的值也被改动了。

如果切片的底层数组没有足够的可用容量，append 函数会创建一个新的底层数组，将被引用的现有的值复制到新数组里，再追加新的值，如代码清单 4-31 所示。

代码清单 4-31 使用 append 同时增加切片的长度和容量

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
```

```
// 向切片追加一个新元素
// 将新元素赋值为 50
newSlice := append(slice, 50)
```

当这个 `append` 操作完成后, `newSlice` 拥有一个全新的底层数组, 这个数组的容量是原来的两倍 (见图 4-15)。

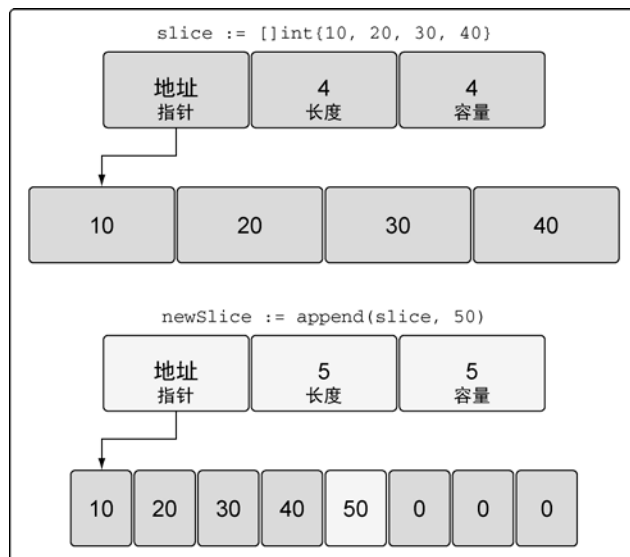


图 4-15 `append` 操作之后的新的底层数组

函数 `append` 会智能地处理底层数组的容量增长。在切片的容量小于 1000 个元素时, 总是会成倍地增加容量。一旦元素个数超过 1000, 容量的增长因子会设为 1.25, 也就是会每次增加 25% 的容量。随着语言的演化, 这种增长算法可能会有所改变。

3. 创建切片时的 3 个索引

在创建切片时, 还可以使用之前我们没有提及的第三个索引选项。第三个索引可以用来控制新切片的容量。其目的并不是要增加容量, 而是要限制容量。可以看到, 允许限制新切片的容量为底层数组提供了一定的保护, 可以更好地控制追加操作。

让我们看看一个包含 5 个元素的字符串切片。这个切片包含了本地超市能找到的水果名字, 如代码清单 4-32 所示。

代码清单 4-32 使用切片字面量声明一个字符串切片

```
// 创建字符串切片
// 其长度和容量都是 5 个元素
source := []string{"Apple", "Orange", "Plum", "Banana", "Grape"}
```

如果查看这个包含水果的切片的值, 就像图 4-16 所展示的样子。

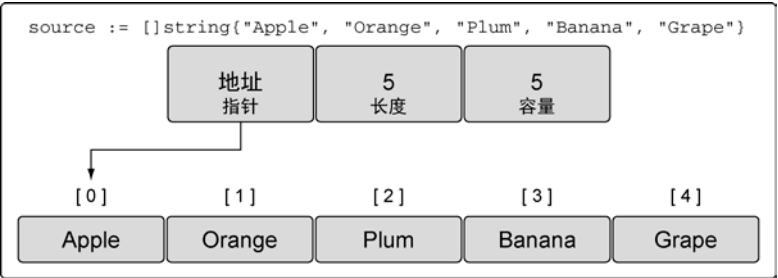


图 4-16 字符串切片的表示

现在，让我们试着用第三个索引选项来完成切片操作，如代码清单 4-33 所示。

代码清单 4-33 使用 3 个索引创建切片

```
// 将第三个元素切片，并限制容量
// 其长度为 1 个元素，容量为 2 个元素
slice := source[2:3:4]
```

这个切片操作执行后，新切片里从底层数组引用了 1 个元素，容量是 2 个元素。具体来说，新切片引用了 Plum 元素，并将容量扩展到 Banana 元素，如图 4-17 所示。

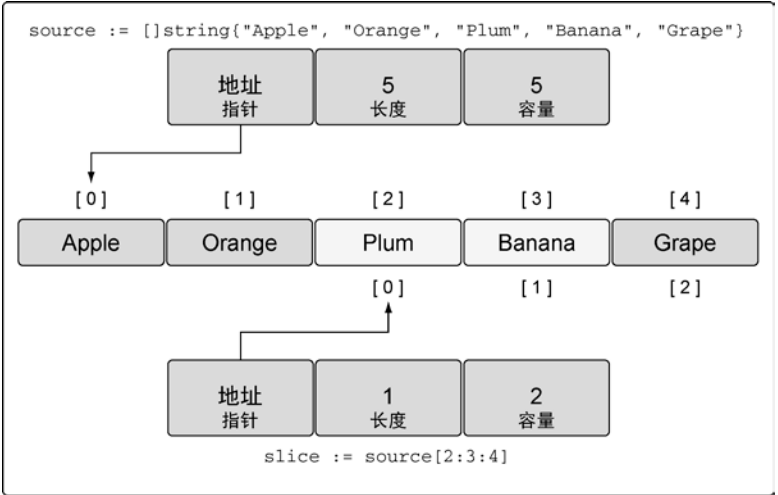


图 4-17 操作之后的新切片的表示

我们可以应用之前定义的公式来计算新切片的长度和容量，如代码清单 4-34 所示。

代码清单 4-34 如何计算长度和容量

```
对于 slice[i:j:k] 或 [2:3:4]
```

```
长度: j - i 或 3 - 2 = 1
容量: k - i 或 4 - 2 = 2
```

和之前一样，第一个值表示新切片开始的元素的索引位置，这个例子中是 2。第二个值表示开始的索引位置 (2) 加上希望包括的元素的个数 (1)，2+1 的结果是 3，所以第二个值就是 3。为了设置容量，从索引位置 2 开始，加上希望容量中包含的元素的个数 (2)，就得到了第三个值 4。

如果试图设置的容量比可用的容量还大，就会得到一个语言运行时错误，如代码清单 4-35 所示。

代码清单 4-35 设置容量大于已有容量的语言运行时错误

```
// 这个切片操作试图设置容量为 4
// 这比可用的容量大
slice := source[2:3:6]

Runtime Error:
panic: runtime error: slice bounds out of range
```

我们之前讨论过，内置函数 `append` 会首先使用可用容量。一旦没有可用容量，会分配一个新的底层数组。这导致很容易忘记切片间正在共享同一个底层数组。一旦发生这种情况，对切片进行修改，很可能会导致随机且奇怪的问题。对切片内容的修改会影响多个切片，却很难找到问题的原因。

如果在创建切片时设置切片的容量和长度一样，就可以强制让新切片的第一个 `append` 操作创建新的底层数组，与原有的底层数组分离。新切片与原有的底层数组分离后，可以安全地进行后续修改，如代码清单 4-36 所示。

代码清单 4-36 设置长度和容量一样的好处

```
// 创建字符串切片
// 其长度和容量都是 5 个元素
source := []string{"Apple", "Orange", "Plum", "Banana", "Grape"}

// 对第三个元素做切片，并限制容量
// 其长度和容量都是 1 个元素
slice := source[2:3:3]

// 向 slice 追加新字符串
slice = append(slice, "Kiwi")
```

如果不加第三个索引，由于剩余的所有容量都属于 `slice`，向 `slice` 追加 `Kiwi` 会改变原有底层数组索引为 3 的元素的值 `Banana`。不过在代码清单 4-36 中我们限制了 `slice` 的容量为 1。当我们第一次对 `slice` 调用 `append` 的时候，会创建一个新的底层数组，这个数组包括 2 个元素，并将水果 `Plum` 复制进来，再追加新水果 `Kiwi`，并返回一个引用了这个底层数组的新切片，如图 4-18 所示。

因为新的切片 `slice` 拥有了自己的底层数组，所以杜绝了可能发生的問題。我们可以继续向新切片里追加水果，而不用担心会不小心修改了其他切片里的水果。同时，也保持了为切片申

请新的底层数组的简洁。

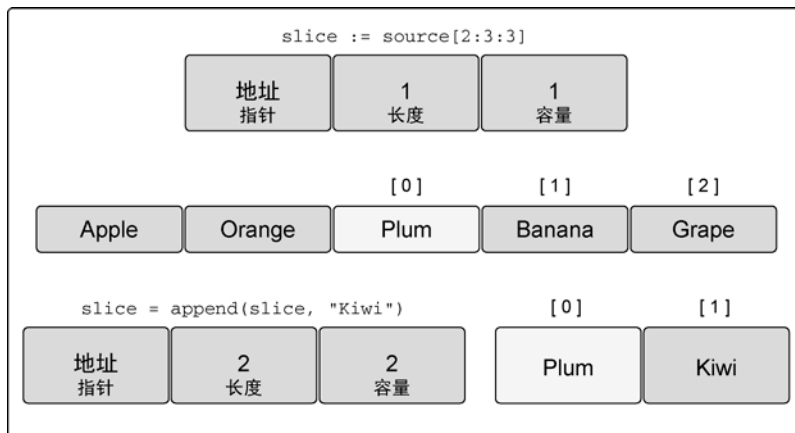


图 4-18 append 操作之后的新切片的表示

内置函数 `append` 也是一个可变参数的函数。这意味着可以在一次调用传递多个追加的值。如果使用 `...` 运算符，可以将一个切片的所有元素追加到另一个切片里，如代码清单 4-37 所示。

代码清单 4-37 将一个切片追加到另一个切片

```
// 创建两个切片，并分别用两个整数进行初始化
s1 := []int{1, 2}
s2 := []int{3, 4}

// 将两个切片追加在一起，并显示结果
fmt.Printf("%v\n", append(s1, s2...))
```

Output:
[1 2 3 4]

就像通过输出看到的那样，切片 `s2` 里的所有值都追加到了切片 `s1` 的后面。使用 `Printf` 时用来显示 `append` 函数返回的新切片的值。

4. 迭代切片

既然切片是一个集合，可以迭代其中的元素。Go 语言有个特殊的关键字 `range`，它可以配合关键字 `for` 来迭代切片里的元素，如代码清单 4-38 所示。

代码清单 4-38 使用 `for range` 迭代切片

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
```

```
// 迭代每一个元素，并显示其值
for index, value := range slice {
    fmt.Printf("Index: %d Value: %d\n", index, value)
}
```

Output:

```
Index: 0 Value: 10
Index: 1 Value: 20
Index: 2 Value: 30
Index: 3 Value: 40
```

当迭代切片时，关键字 `range` 会返回两个值。第一个值是当前迭代到的索引位置，第二个值是该位置对应元素值的一份副本（见图 4-19）。

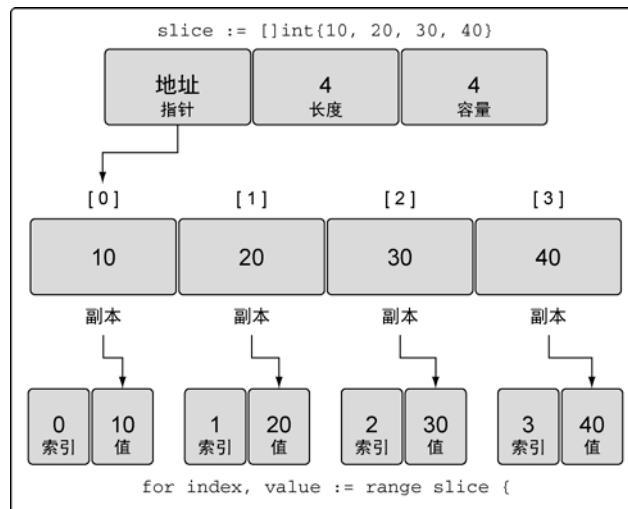


图 4-19 使用 `range` 迭代切片会创建每个元素的副本

需要强调的是，`range` 创建了每个元素的副本，而不是直接返回对该元素的引用，如代码清单 4-39 所示。如果使用该值变量的地址作为指向每个元素的指针，就会造成错误。让我们看看是什么原因。

代码清单 4-39 `range` 提供了每个元素的副本

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}

// 迭代每个元素，并显示值和地址
for index, value := range slice {
    fmt.Printf("Value: %d Value-Addr: %X ElemAddr: %X\n",
        value, &value, &slice[index])
}
```

```
Output:
Value: 10 Value-Addr: 10500168 ElemAddr: 1052E100
Value: 20 Value-Addr: 10500168 ElemAddr: 1052E104
Value: 30 Value-Addr: 10500168 ElemAddr: 1052E108
Value: 40 Value-Addr: 10500168 ElemAddr: 1052E10C
```

因为迭代返回的变量是一个迭代过程中根据切片依次赋值的新变量，所以 `value` 的地址总是相同的。要想获取每个元素的地址，可以使用切片变量和索引值。

如果不需要索引值，可以使用占位字符来忽略这个值，如代码清单 4-40 所示。

代码清单 4-40 使用空白标识符（下划线）来忽略索引值

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}

// 迭代每个元素，并显示其值
for _, value := range slice {
    fmt.Printf("Value: %d\n", value)
}
```

```
Output:
Value: 10
Value: 20
Value: 30
Value: 40
```

关键字 `range` 总是会从切片头部开始迭代。如果想对迭代做更多的控制，依旧可以使用传统的 `for` 循环，如代码清单 4-41 所示。

代码清单 4-41 使用传统的 `for` 循环对切片进行迭代

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}

// 从第三个元素开始迭代每个元素
for index := 2; index < len(slice); index++ {
    fmt.Printf("Index: %d Value: %d\n", index, slice[index])
}
```

```
Output:
Index: 2 Value: 30
Index: 3 Value: 40
```

有两个特殊的内置函数 `len` 和 `cap`，可以用于处理数组、切片和通道。对于切片，函数 `len` 返回切片的长度，函数 `cap` 返回切片的容量。在代码清单 4-41 里，我们使用函数 `len` 来决定什么时候停止对切片的迭代。

现在知道了如何创建和使用切片。可以组合多个切片成为多维切片，并对其进行迭代。

4.2.4 多维切片

和数组一样，切片是一维的。不过，和之前对数组的讨论一样，可以组合多个切片形成多维切片，如代码清单 4-42 所示。

代码清单 4-42 声明多维切片

```
// 创建一个整型切片的切片
slice := [][]int{{10}, {100, 200}}
```

我们有了一个包含两个元素的外层切片，每个元素包含一个内层的整型切片。切片 `slice` 的值看起来像图 4-20 展示的样子。

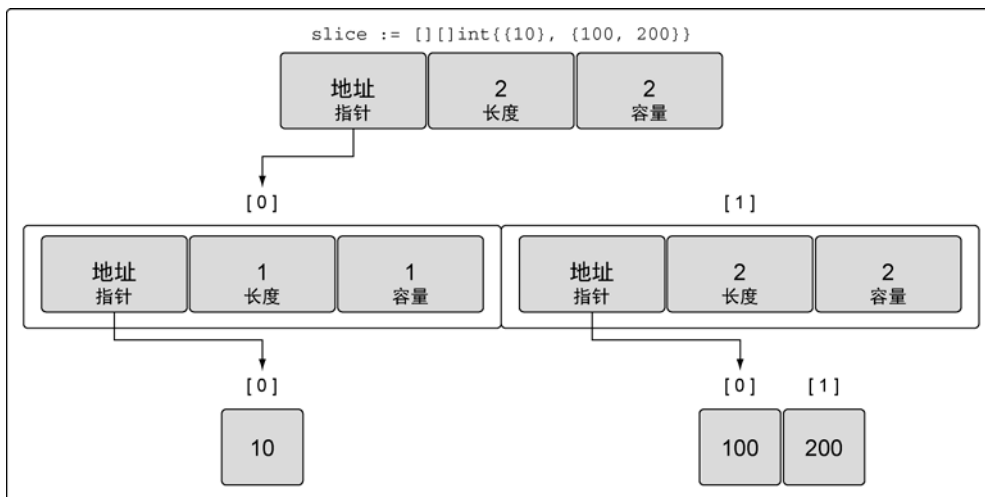


图 4-20 整型切片的切片的值

在图 4-20 里，可以看到组合切片的操作是如何将一个切片嵌入到另一个切片中的。外层切片包括两个元素，每个元素都是一个切片。第一个元素中的切片使用单个整数 10 来初始化，第二个元素中的切片包括两个整数，即 100 和 200。

这种组合可以让用户创建非常复杂且强大的数据结构。已经学过的关于内置函数 `append` 的规则也可以应用到组合后的切片上，如代码清单 4-43 所示。

代码清单 4-43 组合切片的切片

```
// 创建一个整型切片的切片
slice := [][]int{{10}, {100, 200}}

// 为第一个切片追加值为 20 的元素
slice[0] = append(slice[0], 20)
```

Go 语言里使用 `append` 函数处理追加的方式很简明：先增长切片，再将新的整型切片赋值给外层切片的第一个元素。当代码清单 4-43 中的操作完成后，会为新的整型切片分配新的底层数组，然后将切片复制到外层切片的索引为 0 的元素，如图 4-21 所示。

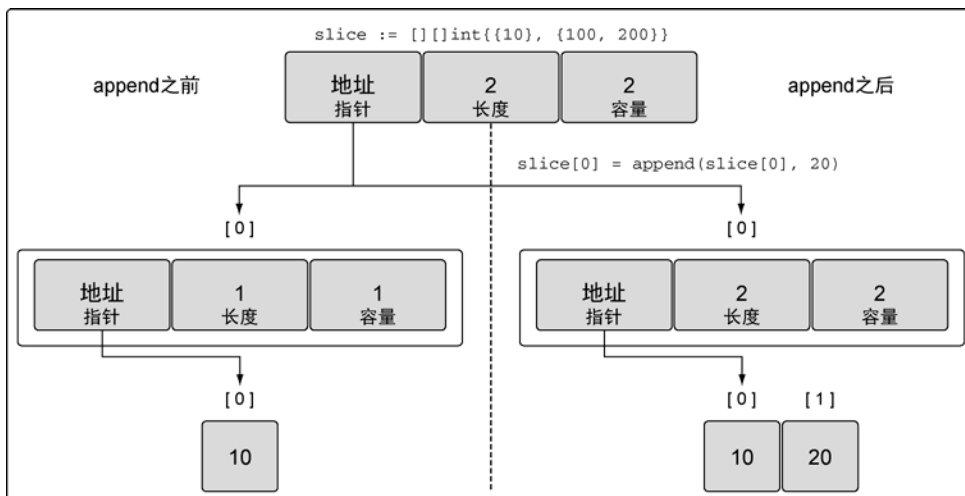


图 4-21 `append` 操作之后外层切片索引为 0 的元素的布局

即便是这么简单的多维切片，操作时也会涉及众多布局和值。看起来在函数间像这样传递数据结构也会很复杂。不过切片本身结构很简单，可以以很小的成本在函数间传递。

4.2.5 在函数间传递切片

在函数间传递切片就是要在函数间以值的方式传递切片。由于切片的尺寸很小，在函数间复制和传递切片成本也很低。让我们创建一个大切片，并将这个切片以值的方式传递给函数 `foo`，如代码清单 4-44 所示。

代码清单 4-44 在函数间传递切片

```
// 分配包含 100 万个整型值的切片
slice := make([]int, 1e6)

// 将 slice 传递到函数 foo
slice = foo(slice)

// 函数 foo 接收一个整型切片，并返回这个切片
func foo(slice []int) []int {
    ...
    return slice
}
```

在 64 位架构的机器上，一个切片需要 24 字节的内存：指针字段需要 8 字节，长度和容量

字段分别需要 8 字节。由于与切片关联的数据包含在底层数组里，不属于切片本身，所以将切片复制到任意函数的时候，对底层数组大小都不会有影响。复制时只会复制切片本身，不会涉及底层数组（见图 4-22）。

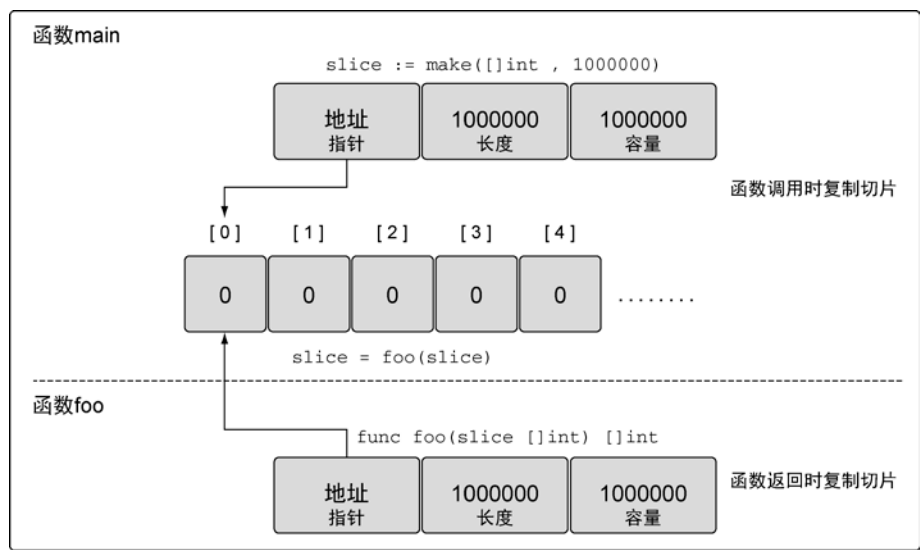


图 4-22 函数调用之后两个切片指向同一个底层数组

在函数间传递 24 字节的数据会非常快速、简单。这也是切片效率高的地方。不需要传递指针和处理复杂的语法，只需要复制切片，按想要的方式修改数据，然后传递回一份新的切片副本。

4.3 映射的内部实现和基础功能

映射是一种数据结构，用于存储一系列无序的键值对。映射里基于键来存储值。图 4-23 通过一个例子展示了映射里键值对是如何存储的。映射功能强大的地方是，能够基于键快速检索数据。键就像索引一样，指向与该键关联的值。

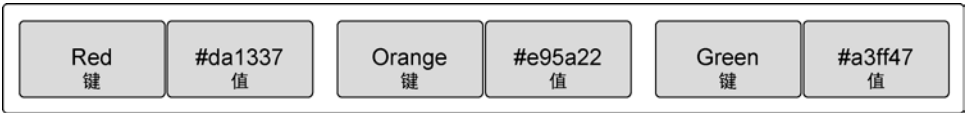


图 4-23 键值对的关系

4.3.1 内部实现

映射是一个集合，可以使用类似处理数组和切片的方式迭代映射中的元素。但映射是无序的

集合，意味着没有办法预测键值对被返回的顺序。即便使用同样的顺序保存键值对，每次迭代映射的时候顺序也可能不一样。无序的原因是映射的实现使用了散列表，见图 4-24。

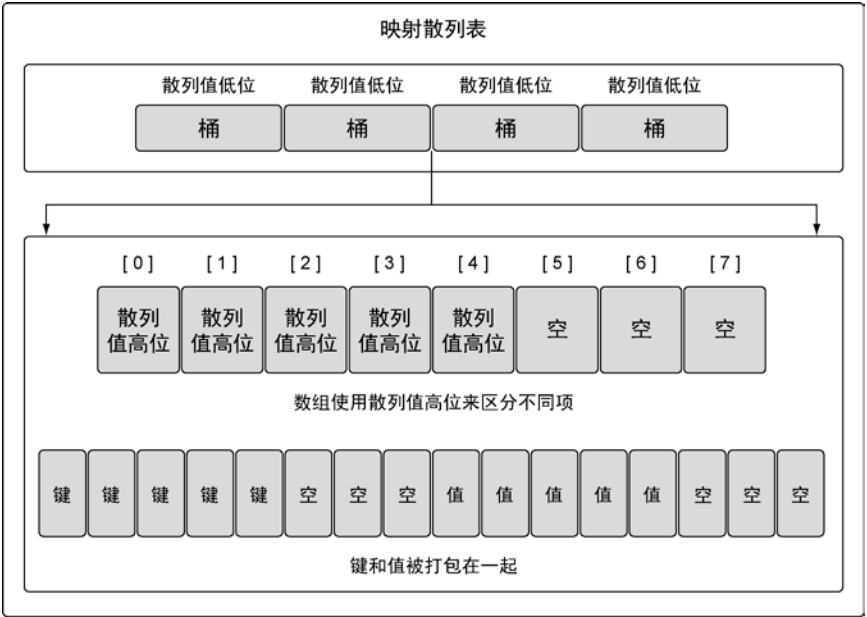


图 4-24 映射的内部结构的简单表示

映射的散列表包含一组桶。在存储、删除或者查找键值对的时候，所有操作都要先选择一个桶。把操作映射时指定的键传给映射的散列函数，就能选中对应的桶。这个散列函数的目的是生成一个索引，这个索引最终将键值对分布到所有可用的桶里。

随着映射存储的增加，索引分布越均匀，访问键值对的速度就越快。如果你在映射里存储了 10 000 个元素，你不希望每次查找都要访问 10 000 个键值对才能找到需要的元素，你希望查找键值对的次数越少越好。对于有 10 000 个元素的映射，每次查找只需要查找 8 个键值对才是一个分布得比较好的映射。映射通过合理数量的桶来平衡键值对的分布。

Go 语言的映射生成散列键的过程比图 4-25 展示的过程要稍微长一些，不过大体过程是类似的。在我们的例子里，键是字符串，代表颜色。这些字符串会转换为一个数值（散列值）。这个数值落在映射已有桶的序号范围内表示一个可以用于存储的桶的序号。之后，这个数值就被用于选择桶，用于存储或者查找指定的键值对。对 Go 语言的映射来说，生成的散列键的一部分，具体来说是低位（LOB），被用来选择桶。

如果再仔细看看图 4-24，就能看出桶的内部实现。映射使用两个数据结构来存储数据。第一个数据结构是一个数组，内部存储的是用于选择桶的散列键的高八位值。这个数组用于区分每个键值对要存在哪个桶里。第二个数据结构是一个字节数组，用于存储键值对。该字节数组先依次

存储了这个桶里所有的键，之后依次存储了这个桶里所有的值。实现这种键值对的存储方式目的在于减少每个桶所需的内存。

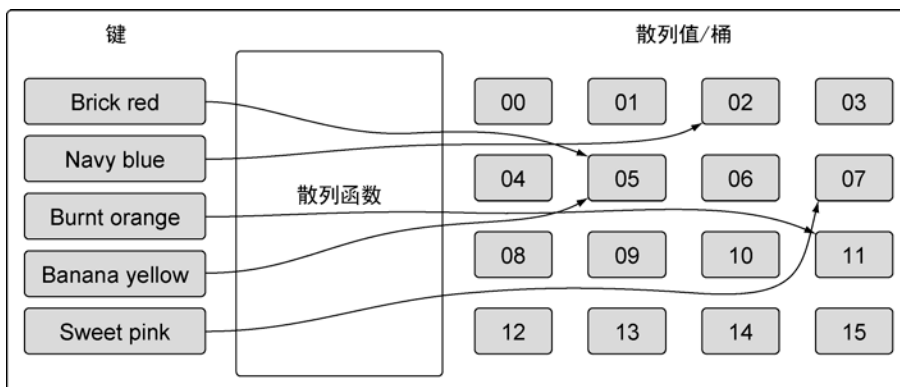


图 4-25 简单描述散列函数是如何工作的

映射底层的实现还有很多细节，不过这些细节超出了本书的范畴。创建并使用映射并不需要了解所有的细节，只要记住一件事：映射是一个存储键值对的无序集合。

4.3.2 创建和初始化

Go 语言中有很多方法可以创建并初始化映射，可以使用内置的 `make` 函数（如代码清单 4-45 所示），也可以使用映射字面量。

代码清单 4-45 使用 `make` 声明映射

```
// 创建一个映射，键的类型是 string，值的类型是 int
dict := make(map[string]int)

// 创建一个映射，键和值的类型都是 string
// 使用两个键值对初始化映射
dict := map[string]string{"Red": "#da1337", "Orange": "#e95a22"}
```

创建映射时，更常用的方法是使用映射字面量。映射的初始长度会根据初始化时指定的键值对的数量来确定。

映射的键可以是任何值。这个值的类型可以是内置的类型，也可以是结构类型，只要这个值可以使用 `==` 运算符做比较。切片、函数以及包含切片的结构类型这些类型由于具有引用语义，不能作为映射的键，使用这些类型会造成编译错误，如代码清单 4-46 所示。

代码清单 4-46 使用映射字面量声明空映射

```
// 创建一个映射，使用字符串切片作为映射的键
dict := map[[]string]int{}
```

```
Compiler Exception:
invalid map key type []string
```

没有任何理由阻止用户使用切片作为映射的值，如代码清单 4-47 所示。这个在使用一个映射键对应一组数据时，会非常有用。

代码清单 4-47 声明一个存储字符串切片的映射

```
// 创建一个映射，使用字符串切片作为值
dict := map[int][]string{}
```

4.3.3 使用映射

键值对赋值给映射，是通过指定适当类型的键并给这个键赋一个值来完成的，如代码清单 4-48 所示。

代码清单 4-48 为映射赋值

```
// 创建一个空映射，用来存储颜色以及颜色对应的十六进制代码
colors := map[string]string{}

// 将 Red 的代码加入到映射
colors["Red"] = "#d1337"
```

可以通过声明一个未初始化的映射来创建一个值为 `nil` 的映射（称为 `nil` 映射）。`nil` 映射不能用于存储键值对，否则，会产生一个语言运行时错误，如代码清单 4-49 所示。

代码清单 4-49 对 `nil` 映射赋值时的语言运行时错误

```
// 通过声明映射创建一个 nil 映射
var colors map[string]string

// 将 Red 的代码加入到映射
colors["Red"] = "#d1337"

Runtime Error:
panic: runtime error: assignment to entry in nil map
```

测试映射里是否存在某个键是映射的一个重要操作。这个操作允许用户写一些逻辑来确定是否完成了某个操作或者是否在映射里缓存了特定数据。这个操作也可以用来比较两个映射，来确定哪些键值对互相匹配，哪些键值对不匹配。

从映射取值时有两个选择。第一个选择是，可以同时获得值，以及一个表示这个键是否存在的标志，如代码清单 4-50 所示。

代码清单 4-50 从映射获取值并判断键是否存在

```
// 获取键 Blue 对应的值
value, exists := colors["Blue"]
```

```
// 这个键存在吗?
if exists {
    fmt.Println(value)
}
```

另一个选择是，只返回键对应的值，然后通过判断这个值是不是零值来确定键是否存在，如代码清单 4-51 所示。^①

代码清单 4-51 从映射获取值，并通过该值判断键是否存在

```
// 获取键 Blue 对应的值
value := colors["Blue"]

// 这个键存在吗?
if value != "" {
    fmt.Println(value)
}
```

在 Go 语言里，通过键来索引映射时，即便这个键不存在也总会返回一个值。在这种情况下，返回的是该值对应的类型的零值。

迭代映射里的所有值和迭代数组或切片一样，使用关键字 `range`，如代码清单 4-52 所示。但对映射来说，`range` 返回的不是索引和值，而是键值对。

代码清单 4-52 使用 `range` 迭代映射

```
// 创建一个映射，存储颜色以及颜色对应的十六进制代码
colors := map[string]string{
    "AliceBlue":    "#f0f8ff",
    "Coral":        "#ff7f50",
    "DarkGray":     "#a9a9a9",
    "ForestGreen": "#228b22",
}

// 显示映射里的所有颜色
for key, value := range colors {
    fmt.Printf("Key: %s Value: %s\n", key, value)
}
```

如果想把一个键值对从映射里删除，就使用内置的 `delete` 函数，如代码清单 4-53 所示。

代码清单 4-53 从映射中删除一项

```
// 删除键为 Coral 的键值对
delete(colors, "Coral")

// 显示映射里的所有颜色
for key, value := range colors {
    fmt.Printf("Key: %s Value: %s\n", key, value)
}
```

^① 这种方法只能用在映射存储的值都是非零值的情况。——译者注

这次在迭代映射时，颜色 Coral 不会显示在屏幕上。

4.3.4 在函数间传递映射

在函数间传递映射并不会制造出该映射的一个副本。实际上，当传递映射给一个函数，并对这个映射做了修改时，所有对这个映射的引用都会察觉到这个修改，如代码清单 4-54 所示。

代码清单 4-54 在函数间传递映射

```
func main() {
    // 创建一个映射，存储颜色以及颜色对应的十六进制代码
    colors := map[string]string{
        "AliceBlue":    "#f0f8ff",
        "Coral":        "#ff7f50",
        "DarkGray":     "#a9a9a9",
        "ForestGreen":  "#228b22",
    }

    // 显示映射里的所有颜色
    for key, value := range colors {
        fmt.Printf("Key: %s Value: %s\n", key, value)
    }

    // 调用函数来移除指定的键
    removeColor(colors, "Coral")

    // 显示映射里的所有颜色
    for key, value := range colors {
        fmt.Printf("Key: %s Value: %s\n", key, value)
    }
}

// removeColor 将指定映射里的键删除
func removeColor(colors map[string]string, key string) {
    delete(colors, key)
}
```

如果运行这个程序，会得到代码清单 4-55 所示的输出。

代码清单 4-55 代码清单 4-54 的输出

```
Key: AliceBlue Value: #F0F8FF
Key: Coral Value: #FF7F50
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22

Key: AliceBlue Value: #F0F8FF
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22
```

可以看到，在调用了 `removeColor` 之后，`main` 函数里引用的映射中不再有 Coral 颜色