09 | 字典的操作和约束

2018-08-31 郝林

至今为止，我们讲过的集合类的高级数据类型都属于针对单一元素的容器。

它们或用连续存储，或用互存指针的方式收纳元素，这里的每个元素都代表了一个从属某一类型的独立值。

我们今天要讲的字典（map）却不同，它能存储的不是单一值的集合，而是键值对的集合。

> 什么是键值对？它是从英文key-value pair直译过来的一个词。顾名思义，一个键值对就代表了一对键和值。
>
> 注意，一个“键”和一个“值”分别代表了一个从属于某一类型的独立值，把它们两个捆绑在一起就是一个键值对了。

在Go语言规范中，应该是为了避免歧义，他们将键值对换了一种称呼，叫做：“键-元素对”。我们也沿用这个看起来更加清晰的词来讲解。

知识前导：为什么字典的键类型会受到约束？

Go语言的字典类型其实是一个哈希表（hash table）的特定实现，在这个实现中，键和元素的最大不同在于，键的类型是受限的，而元素却可以是任意类型的。

如果要探究限制的原因，我们就先要了解哈希表中最重要的一个过程：映射。

你可以把键理解为元素的一个索引，我们可以在哈希表中通过键查找与它成对的那个元素。

键和元素的这种对应关系，在数学里就被称为“映射”，这也是“map”这个词的本意，哈希表的映射过程就存在于对键-元素对的增、删、改、查的操作之中。

```
aMap := map[string]int{
 "one":   1,
 "two":   2,
 "three": 3,
}
k := "two"
v, ok := aMap[k]
if ok {
 fmt.Printf("The element of key %q: %d\n", k, v)
} else {
 fmt.Println("Not found!")
}
```

比如，我们要在哈希表中查找与某个键值对应的那个元素值，那么我们需要先把键值作为参数传给这个哈希表。

哈希表会先用哈希函数（hash function）把键值转换为哈希值。哈希值通常是一个无符号的整数。一个哈希表会持有一定数量的桶（bucket），我们也可以叫它哈希桶，这些哈希桶会均匀地储存其所属哈希表收纳的键-元素对。

因此，哈希表会先用这个键哈希值的低几位去定位到一个哈希桶，然后再去这个哈希桶中，查找这个键。

由于键-元素对总是被捆绑在一起存储的，所以一旦找到了键，就一定能找到对应的元素值。随后，哈希表就会把相应的元素值作为结果返回。

只要这个键-元素对存在哈希表中就一定会被查找到，因为哈希表增、改、删键-元素对时的映射过程，与前文所述如出一辙。

现在我们知道了，映射过程的第一步就是：把键值转换为哈希值。

在Go语言的字典中，每一个键值都是由它的哈希值代表的。也就是说，字典不会独立存储任何键的值，但会独立存储它们的哈希值。

你是不是隐约感觉到了什么？我们接着往下看。

我们今天的问题是：字典的键类型不能是哪些类型？

这个问题你可以在Go语言规范中找到答案，但却没那么简单。它的典型回答是：Go语言字典的键类型不可以是函数类型、字典类型和切片类型。

问题解析

我们来解析一下这个问题。

Go语言规范规定，在键类型的值之间必须可以施加操作符==和!=。换句话说，键类型的值必须要支持判等操作。由于函数类型、字典类型和切片类型的值并不支持判等操作，所以字典的键类型不能是这些类型。

另外，如果键的类型是接口类型的，那么键值的实际类型也不能是上述三种类型，否则在程序运行过程中会引发panic（即运行时恐慌）。

我们举个例子：

```
var badMap2 = map[interface{}]int{
 "1":   1,
 []int{2}: 2, // 这里会引发panic。
 3:    3,
}
```

这里的变量badMap2的类型是键类型为interface{}、值类型为int的字典类型。这样声明并不会引起什么错误。或者说，我通过这样的声明躲过了Go语言编译器的检查。

注意，我用字面量在声明该字典的同时对它进行了初始化，使它包含了三个键-元素对。其中第二个键-元素对的键值是[]int{2}，元素值是2。这样的键值也不会让Go语言编译器报错，因为从语法上说，这样做是可以的。

但是，当我们运行这段代码的时候，Go语言的运行时（runtime）系统就会发现这里的问题，它会抛出一个panic，并把根源指向字面量中定义第二个键-元素对的那一行。我们越晚发现问题，修正问题的成本就会越高，所以最好不要把字典的键类型设定为任何接口类型。如果非要这么做，请一定确保代码在可控的范围之内。

还要注意，如果键的类型是数组类型，那么还要确保该类型的元素类型不是函数类型、字典类型或切片类型。

比如，由于类型[1][]string的元素类型是[]string，所以它就不能作为字典类型的键类型。另外，如果键的类型是结构体类型，那么还要保证其中字段的类型的合法性。无论不合法的

类型被埋藏得有多深，比如`map[[1][2][3][]string]int`，Go语言编译器都会把它揪出来。

你可能会有疑问，为什么键类型的值必须支持判等操作？我在前面说过，Go语言一旦定位到了某一个哈希桶，那么就会试图在这个桶中查找键值。具体是怎么找的呢？

首先，每个哈希桶都会把自己包含的所有键的哈希值存起来。Go语言会用被查找键的哈希值与这些哈希值逐个对比，看看是否有相等的。如果一个相等的都没有，那么就说明这个桶中没有要查找的键值，这时Go语言就会立刻返回结果了。

如果有相等的，那就再用键值本身去对比一次。为什么还要对比？原因是，不同值的哈希值是可能相同的。这有个术语，叫做“哈希碰撞”。

所以，即使哈希值一样，键值也不一定一样。如果键类型的值之间无法判断相等，那么此时这个映射的过程就没办法继续下去了。最后，只有键的哈希值和键值都相等，才能说明查找到了匹配的键-元素对。

以上内容涉及的示例都在demo18.go中。

知识扩展

问题1：应该优先考虑哪些类型作为字典的键类型？

你现在已经清楚了，在Go语言中，有些类型的值是支持判等的，有些是不支持的。那么在这些值支持判等的类型当中，哪些更适合作为字典的键类型呢？

这里先抛开我们使用字典时的上下文，只从性能的角度看。在前文所述的映射过程中，“把键值转换为哈希值”以及“把要查找的键值与哈希桶中的键值做对比”， 明显是两个重要且比较耗时的操作。

因此，可以说，**求哈希和判等操作的速度越快，对应的类型就越适合作为键类型。**

对于所有的基本类型、指针类型，以及数组类型、结构体类型和接口类型，Go语言都有一套算法与之对应。这套算法中就包含了哈希和判等。以求哈希的操作为例，宽度越小的类型速度通常越快。对于布尔类型、整数类型、浮点数类型、复数类型和指针类型来说都是如此。对于字符串类型，由于它的宽度是不定的，所以要看它的值的具体长度，长度越短求哈希越快。

类型的宽度是指它的单个值需要占用的字节数。比如，`bool`、`int8`和`uint8`类型的一个值需要占用的字节数都是`1`，因此这些类型的宽度就都是`1`。

以上说的都是基本类型，再来看高级类型。对数组类型的值求哈希实际上是依次求得它的每个元素的哈希值并进行合并，所以速度就取决于它的元素类型以及它的长度。细则同上。

与之类似，对结构体类型的值求哈希实际上就是对它的所有字段值求哈希并进行合并，所以关键在于它的各个字段的类型以及字段的数量。而对于接口类型，具体的哈希算法，则由值的实际类型决定。

我不建议你使用这些高级数据类型作为字典的键类型，不仅仅是因为对它们的值求哈希，以及判等的速度较慢，更是因为在它们的值中存在变数。

比如，对一个数组来说，我可以任意改变其中的元素值，但在变化前后，它却代表了两个不同的键值。

对于结构体类型的值情况可能会好一些，因为如果我可以控制其中各字段的访问权限的话，就可以阻止外界修改它了。把接口类型作为字典的键类型最危险。

还记得吗？如果在这种情况下Go运行时系统发现某个键值不支持判等操作，那么就会立即抛出一个panic。在最坏的情况下，这足以使程序崩溃。

那么，在那些基本类型中应该优先选择哪一个？答案是，优先选用数值类型和指针类型，通常情况下类型的宽度越小越好。如果非要选择字符串类型的话，最好对键值的长度进行额外的约束。

那什么是不通常的情况？笼统地说，Go语言有时会对字典的增、删、改、查操作做一些优化。

比如，在字典的键类型为字符串类型的情况下；又比如，在字典的键类型为宽度为4或8的整数类型的情况下。

问题2：在值为nil的字典上执行读操作会成功吗，那写操作呢？

好了，为了避免烧脑太久，我们再来说一个简单些的问题。由于字典是引用类型，所以当我们仅声明而不初始化一个字典类型的变量的时候，它的值会是nil。

在这样一个变量上试图通过键值获取对应的元素值，或者添加键-元素对，会成功吗？这个问题虽然简单，但却是我们必须铭记于心的，因为这涉及程序运行时的稳定性。

我来说一下答案。除了添加键-元素对，我们在一个值为nil的字典上做任何操作都不会引起错误。当我们试图在一个值为nil的字典中添加键-元素对的时候，Go语言的运行时系统就会立即抛出一个panic。你可以运行一下demo19.go文件试试看。

总结

我们这次主要讨论了与字典类型有关的，一些容易让人困惑的问题。比如，为什么字典的键类型会受到约束？又比如，我们通常应该选取什么样的类型作为字典的键类型。

我以Go语言规范为起始，并以Go语言源码为依据回答了这些问题。认真看了这篇文章之后，你应该对字典中的映射过程有了一定的理解。

另外，对于Go语言在那些合法的键类型上所做的求哈希和判等的操作，你也应该有所了解了。

再次强调，永远要注意那些可能引发panic的操作，比如像一个值为`nil`的字典添加键-元素对。

思考题

今天的思考题是关于并发安全性的。更具体地说，在同一时间段内但在不同的goroutine（或者说go程）中对同一个值进行操作是否是安全的。这里的安全是指，该值不会因这些操作而产生混乱，或其它不可预知的问题。

具体的思考题是：字典类型的值是并发安全的吗？如果不是，那么在我们只在字典上添加或删除键-元素对的情况下，依然不安全吗？感谢你的收听，我们下期再见。

戳此查看Go语言专栏文章配套详细代码。