

34 | 并发安全字典sync.Map（上）

2018-10-29 郝林



在前面，我几乎已经把Go语言自带的同步工具全盘托出了。你是否已经听懂了会用了呢？无论怎样，我都希望你能够多多练习、多多使用。它们和Go语言独有的并发编程方式并不冲突，相反，配合起来使用，绝对能达到“一加一大于二”的效果。

当然了，至于怎样配合就是一门学问了。我在前面已经讲了不少的方法和技巧，不过，更多的东西可能就需要你在实践中逐渐领悟和总结了。

我们今天再来讲一个并发安全的高级数据结构：`sync.Map`。众所周知，Go语言自带的字典类型map并不是并发安全的。

换句话说，在同一时间段内，让不同goroutine中的代码，对同一个字典进行读写操作是不安全的。字典值本身可能会因这些操作而产生混乱，相关的程序也可能会因此发生不可预知的问题。

在`sync.Map`出现之前，我们如果要想实现并发安全的字典，就只能自行构建。不过，这其实也不是什么麻烦事，使用`sync.Mutex`或`sync.RWMutex`，再加上原生的map就可以轻松地做到。

GitHub网站上已经有很多库提供了类似的数据结构。我在《Go并发编程实战》的第2版中也提供了一个比较完整的并发安全字典的实现。它的性能比同类的数据结构还要好一些，因为它在很大程度上有效地避免了对锁的依赖。

尽管已经有了不少的参考实现，Go语言爱好者们还是希望Go语言官方能够发布一个标准的并发

安全字典。经过大家多年的建议和吐槽，Go语言官方终于在2017年发布的Go 1.9中正式加入了并发安全的字典类型`sync.Map`。

这个字典类型提供了一些常用的键值存取操作方法，并保证了这些操作的并发安全。同时，它的存、取、删等操作都可以基本保证在常数时间内执行完毕。换句话说，它们的算法复杂度与`map`类型一样都是 $O(1)$ 的。

在有些时候，与单纯使用原生`map`和互斥锁的方案相比，使用`sync.Map`可以显著地减少锁的争用。`sync.Map`本身虽然也用到了锁，但是，它其实在尽可能地避免使用锁。

我们都知道，使用锁就意味着要把一些并发的操作强制串行化。这往往会降低程序的性能，尤其是在计算机拥有多个CPU核心的情况下。因此，我们常说，能用原子操作就不要用锁，不过这很有局限性，毕竟原子只能对一些基本的数据类型提供支持。

无论在何种场景下使用`sync.Map`，我们都需要注意，与原生`map`明显不同，它只是Go语言标准库中的一员，而不是语言层面的东西。也正因为这一点，Go语言的编译器并不会对它的键和值进行特殊的类型检查。

如果你看过`sync.Map`的文档或者实际使用过它，那么就一定会知道，它所有的方法涉及的键和值的类型都是`interface{}`，也就是空接口，这意味着可以包罗万象。所以，我们必须在程序中自行保证它的键类型和值类型的正确性。

好了，现在第一个问题来了。**今天的问题是：并发安全字典对键的类型有要求吗？**

这道题的典型回答是：有要求。键的实际类型不能是函数类型、字典类型和切片类型。

解析一下这个问题。 我们都知道，Go语言的原生字典的键类型不能是函数类型、字典类型和切片类型。

由于并发安全字典内部使用的存储介质正是原生字典，又因为它使用的原生字典键类型也是可以包罗万象的`interface{}`，所以，我们绝对不能带着任何实际类型为函数类型、字典类型或切片类型的键值去操作并发安全字典。

由于这些键值的实际类型只有在程序运行期间才能够确定，所以Go语言编译器是无法在编译期对它们进行检查的，不正确的键值实际类型肯定会引发`panic`。

因此，我们在这里首先要做的一件事就是：一定不要违反上述规则。我们应该在每次操作并发安全字典的时候，都去显式地检查键值的实际类型。无论是存、取还是删，都应该如此。

当然，更好的做法是，把针对同一个并发安全字典的这几种操作都集中起来，然后统一地编写检查代码。除此之外，把并发安全字典封装在一个结构体类型中，往往是一个很好的选择。

总之，我们必须保证键的类型是可比较的（或者说可判等的）。如果你实在拿不准，那么可以先通过调用`reflect.TypeOf`函数得到一个键值对应的反射类型值（即：`reflect.Type`类型的值），然后再调用这个值的`Comparable`方法，得到确切的判断结果。

知识扩展

问题1：怎样保证并发安全字典中的键和值的类型正确性？（方案一）

简单地说，可以使用类型断言表达式或者反射操作来保证它们的类型正确性。

为了进一步明确并发安全字典中键值的实际类型，这里大致有两种方案可选。

第一种方案是，让并发安全字典只能存储某个特定类型的键。

比如，指定这里的键只能是`int`类型的，或者只能是字符串，又或是某类结构体。一旦完全确定了键的类型，你就可以在进行存、取、删操作的时候，使用类型断言表达式去对键的类型做检查了。

一般情况下，这种检查并不繁琐。而且，你要是把并发安全字典封装在一个结构体类型里面，那就更加方便了。你这时完全可以让Go语言编译器帮助你做类型检查。请看下面的代码：

```

type IntStrMap struct {
    m sync.Map
}

func (iMap *IntStrMap) Delete(key int) {
    iMap.m.Delete(key)
}

func (iMap *IntStrMap) Load(key int) (value string, ok bool) {
    v, ok := iMap.m.Load(key)
    if v != nil {
        value = v.(string)
    }
    return
}

func (iMap *IntStrMap) LoadOrStore(key int, value string) (actual string, loaded bool) {
    a, loaded := iMap.m.LoadOrStore(key, value)
    actual = a.(string)
    return
}

func (iMap *IntStrMap) Range(f func(key int, value string) bool) {
    f1 := func(key, value interface{}) bool {
        return f(key.(int), value.(string))
    }
    iMap.m.Range(f1)
}

func (iMap *IntStrMap) Store(key int, value string) {
    iMap.m.Store(key, value)
}

```

如上所示，我编写了一个名为IntStrMap的结构体类型，它代表了键类型为int、值类型为string的并发安全字典。在这个结构体类型中，只有一个sync.Map类型的字段m。并且，这个类型拥有的所有方法，都与sync.Map类型的方法非常类似。

两者对应的方法名称完全一致，方法签名也非常相似，只不过，与键和值相关的那些参数和结果的类型不同而已。在`IntStrMap`类型的方法签名中，明确了键的类型为`int`，且值的类型为`string`。

显然，这些方法在接受键和值的时候就不用再做类型检查了。另外，这些方法在从`m`中取出键和值的时候，完全不用担心它们的类型会不正确，因为它的正确性在当初存入的时候，就已经由Go语言编译器保证了。

稍微总结一下。第一种方案适用于我们可以完全确定键和值的具体类型的情况。在这种情况下，我们可以利用Go语言编译器去做类型检查，并用类型断言表达式作为辅助，就像`IntStrMap`那样。

总结

我们今天讨论的是`sync.Map`类型，它是一种并发安全的字典。它提供了一些常用的键、值存取操作方法，并保证了这些操作的并发安全。同时，它还保证了存、取、删等操作的常数级执行时间。

与原生的字典相同，并发安全字典对键的类型也是有要求的。它们同样不能是函数类型、字典类型和切片类型。另外，由于并发安全字典提供的方法涉及的键和值的类型都是`interface{}`，所以我们在调用这些方法的时候，往往还需要对键和值的实际类型进行检查。

这里大致有两个方案。我们今天主要提到了第一种方案，这是在编码时就完全确定键和值的类型，然后利用Go语言的编译器帮我们做检查。在下一次的文章中，我们会提到另外一种方案，并对比这两种方案的优劣。除此之外，我会继续探讨并发安全字典的相关问题。

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。](#)

GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人

