



下载APP



## 09 | map: 如何实现线程安全的map类型?

2020-10-30 鬼岳攀

Go 并发编程实战课

进入课程 &gt;



讲述: 安晓辉

时长 21:50 大小 20.00M



你好，我是鸟窝。

哈希表（Hash Table）这个数据结构，我们已经非常熟悉了。它实现的就是 key-value 之间的映射关系，主要提供的方法包括 Add、Lookup、Delete 等。因为这种数据结构是一个基础的数据结构，每个 key 都会有一个唯一的索引值，通过索引可以很快地找到对应的值，所以使用哈希表进行数据的插入和读取都是很快的。Go 语言本身就内建了这样一个数据结构，也就是 **map 数据类型**。

今天呢，我们就先来学习 Go 语言内建的这个 map 类型，了解它的基本使用方法和使用技巧，然后再学习如何实现线程安全的 map 类型，最后我还会给你介绍 Go 标准库中线程安全的 sync.Map 类型。学完了这节课，你可以学会几种可以并发访问的 map 类型。

## map 的基本使用方法

Go 内建的 map 类型如下：

复制代码

```
1 map[K]V
```

其中，**key 类型的 K 必须是可比较的** (comparable)，也就是可以通过 == 和 != 操作符进行比较；value 的值和类型无所谓，可以是任意的类型，或者为 nil。

在 Go 语言中，bool、整数、浮点数、复数、字符串、指针、Channel、接口都是可比较的，包含可比较元素的 struct 和数组，这两也是可比较的，而 slice、map、函数值都是不可比较的。

那么，上面这些可比较的数据类型都可以作为 map 的 key 吗？显然不是。通常情况下，我们会选择内建的基本类型，比如整数、字符串做 key 的类型，因为这样最方便。

这里有一点需要注意，如果使用 struct 类型做 key 其实是有坑的，因为如果 struct 的某个字段值修改了，查询 map 时无法获取它 add 进去的值，如下面的例子：

复制代码

```
1 type mapKey struct {
2     key int
3 }
4
5 func main() {
6     var m = make(map[mapKey]string)
7     var key = mapKey{10}
8
9
10    m[key] = "hello"
11    fmt.Printf("m[key]=%s\n", m[key])
12
13
14    // 修改key的字段的值后再次查询map，无法获取刚才add进去的值
15    key.key = 100
16    fmt.Printf("再次查询m[key]=%s\n", m[key])
17 }
```

那该怎么办呢？如果要使用 struct 作为 key，我们要保证 struct 对象在逻辑上是不可变的，这样才会保证 map 的逻辑没有问题。

以上就是选取 key 类型的注意点了。接下来，我们看一下使用 map[key] 函数时需要注意的一个知识点。**在 Go 中，map[key]函数返回结果可以是一个值，也可以是两个值**，这是容易让人迷惑的地方。原因在于，如果获取一个不存在的 key 对应的值时，会返回零值。为了区分真正的零值和 key 不存在这两种情况，可以根据第二个返回值来区分，如下面的代码的第 6 行、第 7 行：

复制代码

```

1 func main() {
2     var m = make(map[string]int)
3     m["a"] = 0
4     fmt.Printf("a=%d; b=%d\n", m["a"], m["b"])
5
6     av, aexisted := m["a"]
7     bv, bexisted := m["b"]
8     fmt.Printf("a=%d, existed: %t; b=%d, existed: %t\n", av, aexisted, bv, bexisted)
9 }
```

map 是无序的，所以当遍历一个 map 对象的时候，迭代的元素的顺序是不确定的，无法保证两次遍历的顺序是一样的，也不能保证和插入的顺序一致。那怎么办呢？如果我们想要按照 key 的顺序获取 map 的值，需要先取出所有的 key 进行排序，然后按照这个排序的 key 依次获取对应的值。而如果我们想要保证元素有序，比如按照元素插入的顺序进行遍历，可以使用辅助的数据结构，比如 [orderedmap](#)，来记录插入顺序。

好了，总结下关于 map 我们需要掌握的内容：map 的类型是 map[key]，key 类型的 K 必须是可比较的，通常情况下，我们会选择内建的基本类型，比如整数、字符串做 key 的类型。如果要使用 struct 作为 key，我们要保证 struct 对象在逻辑上是不可变的。在 Go 中，map[key] 函数返回结果可以是一个值，也可以是两个值。map 是无序的，如果我们想要保证遍历 map 时元素有序，可以使用辅助的数据结构，比如 [orderedmap](#)。

## 使用 map 的 2 种常见错误

那接下来，我们来看使用 map 最常犯的两个错误，就是 **未初始化** 和 **并发读写**。

### 常见错误一：未初始化

和 slice 或者 Mutex、 RWmutex 等 struct 类型不同， map 对象必须在使用之前初始化。如果不初始化就直接赋值的话，会出现 panic 异常，比如下面的例子， m 实例还没有初始化就直接进行操作会导致 panic (第 3 行)：

 复制代码

```
1 func main() {  
2     var m map[int]int  
3     m[100] = 100  
4 }
```

解决办法就是在第 2 行初始化这个实例 (`m := make(map[int]int)`)。

从一个 nil 的 map 对象中获取值不会 panic，而是会得到零值，所以下面的代码不会报错：

 复制代码

```
1 func main() {  
2     var m map[int]int  
3     fmt.Println(m[100])  
4 }
```

这个例子很简单，我们可以意识到 map 的初始化问题。但有时候 map 作为一个 struct 字段的时候，就很容易忘记初始化了。

 复制代码

```
1 type Counter struct {  
2     Website      string  
3     Start        time.Time  
4     PageCounters map[string]int  
5 }  
6  
7 func main() {  
8     var c Counter  
9     c.Website = "baidu.com"  
10  
11  
12     c.PageCounters["/]++  
13 }
```

所以，关于初始化这一点，我再强调一下，目前还没有工具可以检查，我们只能记住 “**别忘记初始化**” 这一条规则。

## 常见错误二：并发读写

对于 map 类型，另一个很容易犯的错误就是并发访问问题。这个易错点，相当令人讨厌，如果没有注意到并发问题，程序在运行的时候就有可能出现并发读写导致的 panic。

Go 内建的 map 对象不是线程 (goroutine) 安全的，并发读写的时候运行时会有检查，遇到并发问题就会导致 panic。

我们一起看一个并发访问 map 实例导致 panic 的例子：

 复制代码

```
1 func main() {
2     var m = make(map[int]int, 10) // 初始化一个map
3     go func() {
4         for {
5             m[1] = 1 //设置key
6         }
7     }()
8
9     go func() {
10        for {
11            _ = m[2] //访问这个map
12        }
13    }()
14    select {}
15 }
```

虽然这段代码看起来是读写 goroutine 各自操作不同的元素，貌似 map 也没有扩容的问题，但是运行时检测到同时对 map 对象有并发访问，就会直接 panic。panic 信息会告诉我们代码中哪一行有读写问题，根据这个错误信息你就能快速定位出来是哪一个 map 对象在哪里出的问题了。

```
smallnest > concurrent > master > go run map.go
fatal error: concurrent map read and map write

goroutine 6 [running]:
runtime.throw(0x478d27, 0x21)
    /usr/local/go/src/runtime/panic.go:1116 +0x72 fp=0xc00003cf80 sp=0xc00003cf50 pc=0x42a3d2
runtime.mapaccess1_fast64(0x465800, 0xc000020030, 0x2, 0x4fa680)
    /usr/local/go/src/runtime/map_fast64.go:21 +0x196 fp=0xc00003cfa8 sp=0xc00003cf80 pc=0x40d426
main.main.func2(0xc000020030)
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/map/concurrent/map.go:14 +0x40 fp=0xc00003cf80 sp=0xc00003cfa8 pc=0x458e70
runtime.goexit()
    /usr/local/go/src/runtime/asm_amd64.s:1373 +0x1 fp=0xc00003cfe0 sp=0xc00003cf80 pc=0x454351
created by main.main
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/map/concurrent/map.go:12 +0x88

goroutine 1 [select (no cases)]:
main.main()
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/map/concurrent/map.go:18 +0x8d

goroutine 5 [runnable]:
main.main.func1(0xc000020030)
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/map/concurrent/map.go:8 +0x40
created by main.main
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/map/concurrent/map.go:6 +0x66
exit status 2
```

这个错误非常常见，是几乎每个人都会踩到的坑。其实，不只是我们写代码时容易犯这个错，一些知名的项目中也是屡次出现这个问题，比如 Docker issue 40772，在删除 map 对象的元素时忘记了加锁：

```
diff --git a/builder/builder-next/builder.go b/builder/builder.go
@@ -240,7 +240,9 @@ func (b *Builder) Build(ctx context.Context, opt backend.BuildConfig) (*builder.
240     240             }
241     241
242     242             defer func() {
243 +         243                 b.mu.Lock()
244     244                     delete(b.jobs, buildID)
245 +         245                 b.mu.Unlock()
246     246             }()
247     247         }
248     248
```

Docker issue 40772, Docker issue 35588、34540、39643 等等，也都有并发读写 map 的问题。

除了 Docker 中，Kubernetes 的 issue 84431、72464、68647、64484、48045、45593、37560 等，以及 TiDB 的 issue 14960 和 17494 等，也出现了这个错误。

这么多人会踩的坑，有啥解决方案吗？肯定有，那接下来，我们就继续来看如何解决内建 map 的并发读写问题。

## 如何实现线程安全的 map 类型？

避免 map 并发读写 panic 的方式之一就是加锁，考虑到读写性能，可以使用读写锁提供性能。

## 加读写锁：扩展 map，支持并发读写

比较遗憾的是，目前 Go 还没有正式发布泛型特性，我们还不能实现一个通用的支持泛型的加锁 map。但是，将要发布的泛型方案已经可以验证测试了，离发布也不远了，也许发布之后 sync.Map 就支持泛型了。

当然了，如果没有泛型支持，我们也能解决这个问题。我们可以通过 interface{} 来模拟泛型，但还是要涉及接口和具体类型的转换，比较复杂，还不如将要发布的泛型方案更直接、性能更好。

这里我以一个具体的 map 类型为例，来演示利用读写锁实现线程安全的 map[int]int 类型：

 复制代码

```
1 type RWMap struct { // 一个读写锁保护的线程安全的map
2     sync.RWMutex // 读写锁保护下面的map字段
3     m map[int]int
4 }
5 // 新建一个RWMap
6 func NewRWMap(n int) *RWMap {
7     return &RWMap{
8         m: make(map[int]int, n),
9     }
10 }
11 func (m *RWMap) Get(k int) (int, bool) { //从map中读取一个值
12     m.RLock()
13     defer m.RUnlock()
14     v, existed := m.m[k] // 在锁的保护下从map中读取
15     return v, existed
16 }
17
18 func (m *RWMap) Set(k int, v int) { // 设置一个键值对
19     m.Lock() // 锁保护
20     defer m.Unlock()
21     m.m[k] = v
22 }
23
24 func (m *RWMap) Delete(k int) { //删除一个键
25     m.Lock() // 锁保护
26     defer m.Unlock()
27     delete(m.m, k)
```

```

28 }
29
30 func (m *RWMap) Len() int { // map的长度
31     m.RLock() // 锁保护
32     defer m.RUnlock()
33     return len(m.m)
34 }
35
36 func (m *RWMap) Each(f func(k, v int) bool) { // 遍历map
37     m.RLock() //遍历期间一直持有读锁
38     defer m.RUnlock()
39
40     for k, v := range m.m {
41         if !f(k, v) {
42             return
43         }
44     }
45 }
46

```

正如这段代码所示，对 map 对象的操作，无非就是增删改查和遍历等几种常见操作。我们可以把这些操作分为读和写两类，其中，查询和遍历可以看做读操作，增加、修改和删除可以看做写操作。如例子所示，我们可以通过读写锁对相应的操作进行保护。

## 分片加锁：更高效的并发 map

虽然使用读写锁可以提供线程安全的 map，但是在大量并发读写的情况下，锁的竞争会非常激烈。我在 [第 4 讲](#) 中提到过，锁是性能下降的万恶之源之一。

在并发编程中，我们的一条原则就是尽量减少锁的使用。一些单线程单进程的应用（比如 Redis 等），基本上不需要使用锁去解决并发线程访问的问题，所以可以取得很高的性能。但是对于 Go 开发的应用程序来说，并发是常用的一个特性，在这种情况下，我们能做的就是，**尽量减少锁的粒度和锁的持有时间**。

你可以优化业务处理的代码，以此来减少锁的持有时间，比如将串行的操作变成并行的子任务执行。不过，这就是另外的故事了，今天我们还是主要讲对同步原语的优化，所以这里我重点讲如何减少锁的粒度。

**减少锁的粒度常用的方法就是分片** (Shard)，将一把锁分成几把锁，每个锁控制一个分片。Go 比较知名的分片并发 map 的实现是 [orcaman/concurrent-map](#)。

它默认采用 32 个分片, **GetShard** 是一个关键的方法, 能够根据 key 计算出分片索引。

复制代码

```

1
2     var SHARD_COUNT = 32
3
4     // 分成SHARD_COUNT个分片的map
5     type ConcurrentMap []*ConcurrentMapShared
6
7     // 通过RWMutex保护的线程安全的分片, 包含一个map
8     type ConcurrentMapShared struct {
9         items      map[string]interface{}
10        sync.RWMutex // Read Write mutex, guards access to internal map.
11    }
12
13    // 创建并发map
14    func New() ConcurrentMap {
15        m := make(ConcurrentMap, SHARD_COUNT)
16        for i := 0; i < SHARD_COUNT; i++ {
17            m[i] = &ConcurrentMapShared{items: make(map[string]interface{})}
18        }
19        return m
20    }
21
22
23    // 根据key计算分片索引
24    func (m ConcurrentMap) GetShard(key string) *ConcurrentMapShared {
25        return m[int(fnv32(key))%int(SHARD_COUNT)]
26    }

```

增加或者查询的时候, 首先根据分片索引得到分片对象, 然后对分片对象加锁进行操作:

复制代码

```

1 func (m ConcurrentMap) Set(key string, value interface{}) {
2     // 根据key计算出对应的分片
3     shard := m.GetShard(key)
4     shard.Lock() //对这个分片加锁, 执行业务操作
5     shard.items[key] = value
6     shard.Unlock()
7 }
8
9 func (m ConcurrentMap) Get(key string) (interface{}, bool) {
10    // 根据key计算出对应的分片
11    shard := m.GetShard(key)
12    shard.RLock()
13    // 从这个分片读取key的值
14    val, ok := shard.items[key]

```

```
15     shard.RUnlock()
16     return val, ok
17 }
```

当然，除了 GetShard 方法，ConcurrentMap 还提供了很多其他的方法。这些方法都是通过计算相应的分片实现的，目的是保证把锁的粒度限制在分片上。

好了，到这里我们就学会了解决 map 并发 panic 的两个方法：加锁和分片。

**在我个人使用并发 map 的过程中，加锁和分片加锁这两种方案都比较常用，如果是追求更高的性能，显然是分片加锁更好，因为它可以降低锁的粒度，进而提高访问此 map 对象的吞吐。如果并发性能要求不是那么高的场景，简单加锁方式更简单。**

接下来，我会继续给你介绍 sync.Map，这是 Go 官方线程安全 map 的标准实现。虽然是官方标准，反而是不常用的，为什么呢？一句话来说就是 map 要解决的场景很难描述，很多时候在做抉择时根本就不知道该不该用它。但是呢，确实有一些特定的场景，我们需要用到 sync.Map 来实现，所以还是很有必要学习这个知识点。具体什么场景呢，我慢慢给你道来。

## 应对特殊场景的 sync.Map

Go 内建的 map 类型不是线程安全的，所以 Go 1.9 中增加了一个线程安全的 map，也就是 sync.Map。但是，我们一定要记住，这个 sync.Map 并不是用来替换内建的 map 类型的，它只能被应用在一些特殊的场景里。

那这些特殊的场景是啥呢？[官方的文档](#)中指出，在以下两个场景中使用 sync.Map，会比使用 map+RWMutex 的方式，性能要好得多：

1. 只会增长的缓存系统中，一个 key 只写入一次而被读很多次；
2. 多个 goroutine 为不相交的键集读、写和重写键值对。

这两个场景说得都比较笼统，而且，这些场景中还包含了一些特殊的情况。所以，官方建议你针对自己的场景做性能评测，如果确实能够显著提高性能，再使用 sync.Map。

这么来看，我们能用到 sync.Map 的场景确实不多。即使是 sync.Map 的作者 Bryan C. Mills，也很少使用 sync.Map，即便是在使用 sync.Map 的时候，也是需要临时查询它的 API，才能清楚记住它的功能。所以，我们可以把 sync.Map 看成一个生产环境中很少使用的同步原语。

## sync.Map 的实现

那 sync.Map 是怎么实现的呢？它是如何解决并发问题提升性能的呢？其实 sync.Map 的实现有几个优化点，这里先列出来，我们后面慢慢分析。

空间换时间。通过冗余的两个数据结构（只读的 read 字段、可写的 dirty），来减少加锁对性能的影响。对只读字段（read）的操作不需要加锁。

优先从 read 字段读取、更新、删除，因为对 read 字段的读取不需要锁。

动态调整。miss 次数多了之后，将 dirty 数据提升为 read，避免总是从 dirty 中加锁读取。

double-checking。加锁之后先还要再检查 read 字段，确定真的不存在才操作 dirty 字段。

延迟删除。删除一个键值只是打标记，只有在提升 dirty 字段为 read 字段的时候才清理删除的数据。

要理解 sync.Map 这些优化点，我们还是得深入到它的设计和实现上，去学习它的处理方式。

我们先看一下 map 的数据结构：

 复制代码

```

1 type Map struct {
2     mu Mutex
3     // 基本上你可以把它看成一个安全的只读的map
4     // 它包含的元素其实也是通过原子操作更新的，但是已删除的entry就需要加锁操作了
5     read atomic.Value // readOnly
6
7     // 包含需要加锁才能访问的元素
8     // 包括所有在read字段中但未被expunged（删除）的元素以及新加的元素
9     dirty map[interface{}]*entry
10
11    // 记录从read中读取miss的次数，一旦miss数和dirty长度一样了，就会把dirty提升为read,
12

```

```

13     misses int
14 }
15
16 type readOnly struct {
17     m      map[interface{}]*entry
18     amended bool // 当dirty中包含read没有的数据时为true, 比如新增一条数据
19 }
20
21 // expunged是用来标识此项已经删掉的指针
22 // 当map中的一个项目被删除了, 只是把它的值标记为expunged, 以后才有机会真正删除此项
23 var expunged = unsafe.Pointer(new(interface{}))
24
25 // entry代表一个值
26 type entry struct {
27     p unsafe.Pointer // *interface{}
    }

```

如果 dirty 字段非 nil 的话, map 的 read 字段和 dirty 字段会包含相同的非 expunged 的项, 所以如果通过 read 字段更改了这个项的值, 从 dirty 字段中也会读取到这个项的新值, 因为本来它们指向的就是同一个地址。

dirty 包含重复项目的好处就是, 一旦 miss 数达到阈值需要将 dirty 提升为 read 的话, 只需简单地把 dirty 设置为 read 对象即可。不好的一点就是, 当创建新的 dirty 对象的时候, 需要逐条遍历 read, 把非 expunged 的项复制到 dirty 对象中。

接下来, 我们就深入到源码去看看 sync.map 的实现。在看这部分源码的过程中, 我们只要重点关注 Store、Load 和 Delete 这 3 个核心的方法就可以了。

Store、Load 和 Delete 这三个核心函数的操作都是先从 read 字段中处理的, 因为读取 read 字段的时候不用加锁。

## Store 方法

我们先来看 Store 方法, 它是用来设置一个键值对, 或者更新一个键值对的。

 复制代码

```

1 func (m *Map) Store(key, value interface{}) {
2     read, _ := m.read.Load().(readOnly)
3     // 如果read字段包含这个项, 说明是更新, cas更新项目的值即可
4     if e, ok := read.m[key]; ok && e.tryStore(&value) {
5         return
6     }

```

```

7 // read中不存在, 或者cas更新失败, 就需要加锁访问dirty了
8 m.mu.Lock()
9 read, _ = m.read.Load().(readOnly)
10 if e, ok := read.m[key]; ok { // 双检查, 看看read是否已经存在了
11     if e.unexpungeLocked() {
12         // 此项目先前已经被删除了, 通过将它的值设置为nil, 标记为unexpunged
13         m.dirty[key] = e
14     }
15     e.storeLocked(&value) // 更新
16 } else if e, ok := m.dirty[key]; ok { // 如果dirty中有此项
17     e.storeLocked(&value) // 直接更新
18 } else { // 否则就是一个新的key
19     if !read.amended { //如果dirty为nil
20         // 需要创建dirty对象, 并且标记read的amended为true,
21         // 说明有元素它不包含而dirty包含
22         m.dirtyLocked()
23         m.read.Store(readOnly{m: read.m, amended: true})
24     }
25     m.dirty[key] = newEntry(value) //将新值增加到dirty对象中
26 }
27 m.mu.Unlock()
28 }
29 }
```

可以看出, Store 既可以是新增元素, 也可以是更新元素。如果运气好的话, 更新的是已存在的未被删除的元素, 直接更新即可, 不会用到锁。如果运气不好, 需要更新(重用)删除的对象、更新还未提升的 dirty 中的对象, 或者新增加元素的时候就会使用到了锁, 这个时候, 性能就会下降。

所以从这一点来看, sync.Map 适合那些只会增长的缓存系统, 可以进行更新, 但是不要删除, 并且不要频繁地增加新元素。

新加的元素需要放入到 dirty 中, 如果 dirty 为 nil, 那么需要从 read 字段中复制出来一个 dirty 对象:

 复制代码

```

1 func (m *Map) dirtyLocked() {
2     if m.dirty != nil { // 如果dirty字段已经存在, 不需要创建了
3         return
4     }
5
6     read, _ := m.read.Load().(readOnly) // 获取read字段
7     m.dirty = make(map[interface{}]*entry, len(read.m))
8     for k, e := range read.m { // 遍历read字段
9         if !e.tryExpungeLocked() { // 把非punged的键值对复制到dirty中
10    }
```

```

10         m.dirty[k] = e
11     }
12 }
13 }
```

## Load 方法

Load 方法用来读取一个 key 对应的值。它也是从 read 开始处理，一开始并不需要锁。

复制代码

```

1 func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
2     // 首先从read处理
3     read, _ := m.read.Load().(readOnly)
4     e, ok := read.m[key]
5     if !ok && read.amended { // 如果不存在并且dirty不为nil(有新的元素)
6         m.mu.Lock()
7         // 双检查，看看read中现在是否存在此key
8         read, _ = m.read.Load().(readOnly)
9         e, ok = read.m[key]
10        if !ok && read.amended { //依然不存在，并且dirty不为nil
11            e, ok = m.dirty[key] // 从dirty中读取
12            // 不管dirty中存不存在，miss数都加1
13            m.missLocked()
14        }
15        m.mu.Unlock()
16    }
17    if !ok {
18        return nil, false
19    }
20    return e.load() //返回读取的对象，e既可能是从read中获得的，也可能是从dirty中获得的
21 }
```

如果幸运的话，我们从 read 中读取到了这个 key 对应的值，那么就不需要加锁了，性能会非常好。但是，如果请求的 key 不存在或者是新加的，就需要加锁从 dirty 中读取。所以，读取不存在的 key 会因为加锁而导致性能下降，读取还没有提升的新值的情况下也会因为加锁性能下降。

其中，missLocked 增加 miss 的时候，如果 miss 数等于 dirty 长度，会将 dirty 提升为 read，并将 dirty 置空。

复制代码

```
1 func (m *Map) missLocked() {
```

```

2     m.misses++ // misses计数加一
3     if m.misses < len(m.dirty) { // 如果没达到阈值(dirty字段的长度),返回
4         return
5     }
6     m.read.Store(readOnly{m: m.dirty}) //把dirty字段的内存提升为read字段
7     m.dirty = nil // 清空dirty
8     m.misses = 0 // misses数重置为0
9 }
```

## Delete 方法

sync.map 的第 3 个核心方法是 Delete 方法。在 Go 1.15 中欧长坤提供了一个 LoadAndDelete 的实现 ([go#issue 33762](#))，所以 Delete 方法的核心改在了对 LoadAndDelete 中实现了。

同样地，Delete 方法是先从 read 操作开始，原因我们已经知道了，因为不需要锁。

 复制代码

```

1 func (m *Map) LoadAndDelete(key interface{}) (value interface{}, loaded bool)
2     read, _ := m.read.Load().(readOnly)
3     e, ok := read.m[key]
4     if !ok && read.amended {
5         m.mu.Lock()
6         // 双检查
7         read, _ = m.read.Load().(readOnly)
8         e, ok = read.m[key]
9         if !ok && read.amended {
10             e, ok = m.dirty[key]
11             // 这一行长坤在1.15中实现的时候忘记加上了，导致在特殊的场景下有些key总是没有被
12             delete(m.dirty, key)
13             // miss数加1
14             m.missLocked()
15         }
16         m.mu.Unlock()
17     }
18     if ok {
19         return e.delete()
20     }
21     return nil, false
22 }
23
24 func (m *Map) Delete(key interface{}) {
25     m.LoadAndDelete(key)
26 }
27 func (e *entry) delete() (value interface{}, ok bool) {
28     for {
29         p := atomic.LoadPointer(&e.p)
```

```
30     if p == nil || p == expunged {
31         return nil, false
32     }
33     if atomic.CompareAndSwapPointer(&e.p, p, nil) {
34         return *(*interface{})(p), true
35     }
36 }
37 }
```

如果 read 中不存在，那么就需要从 dirty 中寻找这个项目。最终，如果项目存在就删除（将它的值标记为 nil）。如果项目不为 nil 或者没有被标记为 expunged，那么还可以把它的值返回。

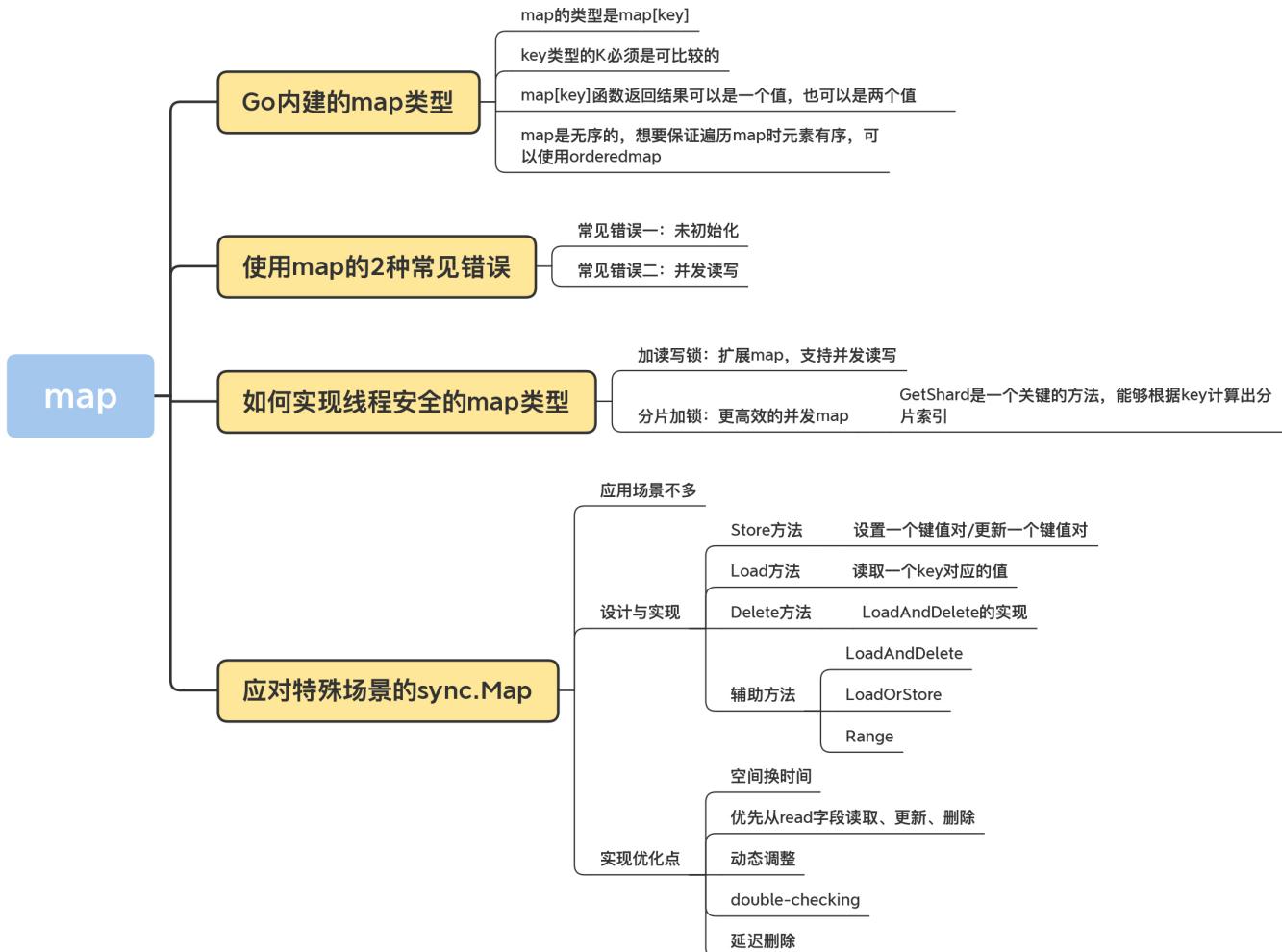
最后，我补充一点，sync.map 还有一些 LoadAndDelete、LoadOrStore、Range 等辅助方法，但是没有 Len 这样查询 sync.Map 的包含项目数量的方法，并且官方也不准备提供。如果你想得到 sync.Map 的项目数量的话，你可能不得不通过 Range 逐个计数。

## 总结

Go 内置的 map 类型使用起来很方便，但是它有一个非常致命的缺陷，那就是它存在着并发问题，所以如果有多个 goroutine 同时并发访问这个 map，就会导致程序崩溃。所以 Go 官方 Blog 很早就提供了一种加锁的 [方法](#)，还有后来提供了适用特定场景的线程安全的 sync.Map，还有第三方实现的分片式的 map，这些方法都可以应用于并发访问的场景。

这里我给你的建议，也是 Go 开发者给的建议，就是通过性能测试，看看某种线程安全的 map 实现是否满足你的需求。

当然还有一些扩展其它功能的 map 实现，比如带有过期功能的 [timedmap](#)、使用红黑树实现的 key 有序的 [treemap](#) 等，因为和并发问题没有关系，就不详细介绍。这里我给你提供了链接，你可以自己探索。



## 思考题

- 为什么 sync.Map 中的集合核心方法的实现中，如果 read 中项目不存在，加锁后还要双检查，再检查一次 read？
- 你看到 sync.map 元素删除的时候只是把它的值设置为 nil，那么什么时候这个 key 才会真正从 map 对象中删除？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

[提建议](#)

# Go并发编程实战课

## 鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金奖励**。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | Once：一个简约而不简单的并发原语

下一篇 10 | Pool：性能提升大杀器

### 精选留言 (10)

写留言



Junes

2020-10-30

1. 双检查主要是针对高并发的场景：

第一次先用CAS快速尝试，失败后进行加锁，然后进行第二次CAS检查，再进行修改；

在高并发的情况下，存在多个goroutine在修改同一个Key，第一次CAS都失败了，在竞争锁；如果不进行第二次CAS检查就直接修改，这个Key就会被多次修改；

...

展开 ▼



6



我来也

2020-10-30

看到本文的标题,就让我想到之前看过的一篇文章:

[踩了 Golang sync.Map 的一个坑](<https://gocn.vip/topics/10860>)

就是老师文章代码中的一行注释的由来:

`这一行长坤在1.15中实现的时候忘记加上了，导致在特殊的场景下有些key总是没有被...

展开 ▾



**NULL**

2020-11-02

感觉有两个地方写的有点模糊，导致对后面的内容有些不明所以\_(:3」∠)\_

1是 “通过冗余的两个数据结构（只读的 read 字段、可以 dirty）”，可以 dirty 是笔误吗

2是 “动态调整。miss 次数多了之后”， miss是什么？

...

展开 ▾



**蝴蝶**

2020-11-12

老师好，我看到 read 中 key 被删除会有两个状态：nil 和 expunged。我会有些不明白，要么都用 nil 或者都用 expunged，这样会不会更好一些？

展开 ▾

作者回复: 第一你说的没错：nil和expunged都代表元素被删除了，只不过expunged比较特殊，如果被删除的元素是expunged,代表它只存在于readonly之中，不存在于dirty中。这样如果重新设置这个key的话，需要往dirty增加key



**约书亚**

2020-11-13

应该着重说明一下为什么有expunged这种状态,这点比较迷惑。我能理解expunged的entry代表read中存在而dirty中不存在。但为什么在read向dirty复制时，需要将nil的entry变为expunged?

作者回复: nil和expunged都代表元素被删除了，只不过expunged比较特殊，如果被删除的元素是expunged,代表它只存在于readonly之中，不存在于dirty中。这样如果重新设置这个key的话，需要往dirty增加key



**Bug? Feature!**

2020-11-07

没有竞争，就没有伤害！  
为啥要再次加锁？  
为了安全，我知道java就有重排序啥的  
展开 ▾

**NULL**

2020-11-02

“通过冗余的两个数据结构（只读的 read 字段、可以 dirty）”  
这里的“可以 dirty”不太通顺吧

展开 ▾

作者回复: 嗯，可以写的dirty字段

**橙子888**

2020-10-31

看完打卡。

展开 ▾

**Panmax**

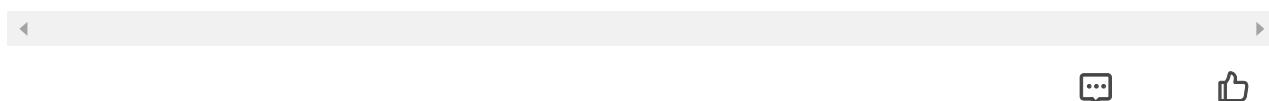
2020-10-31

文章中写到「所以，这里我们就超前一把，我带你直接体验这个即将要发布的泛型方案。」

是我对泛型的理解有什么误会吗，下文中并没有看到使用泛型的地方??????。

展开 ▾

作者回复: 这句话应该删除，谢谢指出

**asdf100**

2020-10-30

分片索引值计算也消耗一部分时间的吧？

展开 ▼

1

