一个系列彻底搞懂map(终):并发安全map

并发安全的map在日常开发中使用极为频繁，单纯通过map与sync.RWLock结合并给整个map上粗粒度锁的方式效率并不高。在go1.9中引入sync.Map，通过快慢路径的方式提升并发读性能，除此之外还可以用大锁打为小锁的方式减少并发冲突，获得更优的性能表现。

本文就利用大锁化小锁、粗粒度变细粒度的思想，来讲解如何设计一个高性能的并发安全map，go社区的一个开源库concurrent-map使用到了此思想。虽然个人认为该库有些地方设计欠妥，但依旧有值得学习的部分，本篇文章将在此开源库的基础上展开。

最简单的设计

如果让我们自行设计一个并发安全的map，最容易想到的就是将原生的map与一个锁结合即可，让这个锁保护整个map。

```
type Map struct {
        m  map[interface{}]interface{}
        mu sync.RWMutex
}

func (m *Map) Set(k, v interface{}) {
        m.mu.Lock()
        m.m[k] = v
        m.mu.Unlock()
}

func (m *Map) Get(k interface{}) (v interface{}, ok bool) {
        m.mu.RLock()
        v, ok = m.m[k]
        m.mu.RUnlock()
        return
}

func (m *Map) Del(k interface{}) {
        m.mu.Lock()
        delete(m.m, k)
        m.mu.Unlock()
}
```

每次操作前，都对整个map上锁就可以保证并发安全，为了优化读多写少时的表现，我们使用`sync.RWMutex`读写锁替代了`sync.Mutex`。

虽然实现很简单，但每个操作都对map上锁，因此就存在频繁的资源竞争问题，所有操作的key之间互为竞争状态。

其实出现这种现象的原因就是锁的粒度过大，如果我们将map分为多个区间，每个区间单独用一个细粒度锁保护的话，这样只有映射到同一个区间的key才存在竞争问题。

源码剖析

concurent-map的思想很简单，就是维护多个map区间，每个map配备一把锁。我们将所有的key打散映射到不同的map中，这样就只有映射到同一map的key才会存在锁竞争问题。

那么如何确定key到map的映射关系呢？很简单，将所有map区间以数组方式组织，然后参照hash表中给定key找桶的方式，对key进行hash然后对区间个数取模，就可以知道key应该存取在哪个map区间中。

数据结构

```
// 区间数，每个区间对应一个小map
var SHARD_COUNT = 32

// 并发安全map，由多个小区间map组成
type ConcurrentMap []*ConcurrentMapShared

// 一个区间
type ConcurrentMapShared struct {
        items        map[string]interface{}     //这个区间存取的键值对
        sync.RWMutex // 每个小区间用读写锁保护
}

// 生成
func New() ConcurrentMap {
        m := make(ConcurrentMap, SHARD_COUNT)
        for i := 0; i < SHARD_COUNT; i++ {
                m[i] = &ConcurrentMapShared{items: make(map[string]interface{})}
        }
        return m
}
```

ConcurrentMap就是用户看到的一个并发安全的大map，它由多个小区间通过数组组织形成。

CouncurrentMap是每一个小区间，每个小区间都单独用一把锁保护，每个小区间都可存取键值对。

这个库只支持key为string类型。

给定key如何获取存取该key的map区间：

```
// 通过求模确定key存取的map区间
func (m ConcurrentMap) GetShard(key string) *ConcurrentMapShared {
        return m[uint(fnv32(key))%uint(SHARD_COUNT)]
}

// 对string类型hash得到整数
func fnv32(key string) uint32 {
        hash := uint32(2166136261)
        const prime32 = uint32(16777619)
        keyLength := len(key)
        for i := 0; i < keyLength; i++ {
                hash *= prime32
                hash ^= uint32(key[i])
        }
        return hash
}
```

其中fnv32是对字符串进行hash求值的一种算法，并非本文重点，我们不过多讲解，感兴趣的同学可阅读Fowler-Noll-Vo hash function。

很显然，如何hash函数能够将key平均打散到每个区间的话，则区间个数有几个，就能将资源竞争冲突减少多少倍。

Set

Set方法用于插入或更新键值对。实现极为简单：

```
func (m ConcurrentMap) Set(key string, value interface{}) {
        // 先获取map区间
        shard := m.GetShard(key)

    //再对小区间插入键值对
        shard.Lock()
        shard.items[key] = value
        shard.Unlock()
}
```

Get

Get方法用于获取对应key的value：

```
func (m ConcurrentMap) Get(key string) (interface{}, bool) {
        // 获取map区间
        shard := m.GetShard(key)
        shard.RLock()
        val, ok := shard.items[key]
        shard.RUnlock()
        return val, ok
}
```

Remove

Remove方法用于删除键值对：

```
func (m ConcurrentMap) Remove(key string) {
        shard := m.GetShard(key)
        shard.Lock()
        delete(shard.items, key)
        shard.Unlock()
}
```

遍历

遍历实现比较有意思点，使用管道实现：

```
// 键值对
type Tuple struct {
        Key string
        Val interface{}
}

// 返回一个可获取键值对的无缓冲管道
func (m ConcurrentMap) Iter() <-chan Tuple {
    // 返回一个管道数组，每个管道对应一个map区间，会将这个map区间的键值对发送到这个管道上。
        chans := snapshot(m)
        ch := make(chan Tuple)
    // 将chans数组里每个管道的数据往ch上发送
        go fanIn(chans, ch)
        return ch
}

// 返回一个可获取键值对的有缓冲管道
func (m ConcurrentMap) IterBuffered() <-chan Tuple {
        chans := snapshot(m)
```

```
        total := 0
        for _, c := range chans {
                total += cap(c)
        }
        ch := make(chan Tuple, total)
        go fanIn(chans, ch)
        return ch
}

// 将chans中数据往out上发送
func fanIn(chans []chan Tuple, out chan Tuple) {
        wg := sync.WaitGroup{}
        wg.Add(len(chans))
        for _, ch := range chans {
                go func(ch chan Tuple) {
                        for t := range ch {
                                out <- t
                        }
                        wg.Done()
                }(ch)
        }
        wg.Wait()
        close(out)
}
```

用户不论调用Iter还是IterBuffered，只需要从返回的管道上接受数据，就可以并发安全的遍历整个map。

看看snapshot的实现：

```
func snapshot(m ConcurrentMap) (chans []chan Tuple) {
        chans = make([]chan Tuple, SHARD_COUNT) // 每个区间对应一个管道
        wg := sync.WaitGroup{}
        wg.Add(SHARD_COUNT)
        // 遍历每个区间
        for index, shard := range m {
                go func(index int, shard *ConcurrentMapShared) {
                        // 遍历每个键值
                        shard.RLock()   // 上锁
                        chans[index] = make(chan Tuple, len(shard.items))       //分配缓存
                        wg.Done()
                        for key, val := range shard.items {
                                chans[index] <- Tuple{key, val}
                        }
                        shard.RUnlock()
                        close(chans[index])
```

```
                }(index, shard)
        }
        wg.Wait()
        return chans
}
```

逻辑很简单，但一定得理解这个`WaitGroup`的出现原因以及调用`Done`的时机。

首先搞明白为什么要给chans中的每个管道分配缓存。chans中管道数据会在`fanIn`中被消费，如果全部为无缓冲管道，则`snapshot`函数中的goroutine退出时机取决于`fanIn`中接收数据的速度，这就可能导致同一时间存在大量goroutine，消耗资源的同时延缓了锁的释放。

但如果给管道分配键值对个数的缓存，则`snapshot`的goroutine可以不阻塞的将每个区间map的键值发送到管道的缓冲上。

搞清楚上一点后，就得搞清楚给管道们分配缓存的时机，为什么要在goroutine中对区间map上锁后再分配缓存？为什么不这么做：

```
for index, shard := range m {
    chans[index] = make(chan Tuple, len(shard.items))
    go func(index int, shard *ConcurrentMapShared) {
        //...
    }(index, shard)
}
```

这个其实很好理解，如果在上述第二行分配缓存后，此时另外一个goroutine正好对这个区间map进行了插入，则设置的缓存容量存在问题，所以必须将分配过程利用锁保护。

最后，为了防止chans中缓存还未分配就交给`fanIn`函数，所以利用了`WaitGroup`进行同步。

这几个方法就是本库的核心内容，你会发现其实实现极为简单。

不合理之处

首先，对于map中元素数量的获取不太合理：

```
func (m ConcurrentMap) Count() int {
        count := 0
        for i := 0; i < SHARD_COUNT; i++ {
                shard := m[i]
                shard.RLock()
                count += len(shard.items)
                shard.RUnlock()
        }
```

```
        return count
}
```

这种方式可能得不到调用Count时刻map中的实际键值对个数。整个遍历相加的过程并非一气呵成的原子过程，遍历过程中其他goroutine对分区的插入或删除，会导致Count统计个数与实际不符。

我觉得合理的做法是，给ConcurrentMap增加一个count成员，在插入删除操作中利用atomic包下的原子操作对这个count进行修改。

其次过分滥用管道，不仅让代码逻辑不清晰还让效率降低，很典型的就是Keys方法：

```
// Keys returns all keys as []string
func (m ConcurrentMap) Keys() []string {
        count := m.Count()
        ch := make(chan string, count)
        go func() {
                // Foreach shard.
                wg := sync.WaitGroup{}
                wg.Add(SHARD_COUNT)
                for _, shard := range m {
                        go func(shard *ConcurrentMapShared) {
                                // Foreach key, value pair.
                                shard.RLock()
                                for key := range shard.items {
                                        ch <- key
                                }
                                shard.RUnlock()
                                wg.Done()
                        }(shard)
                }
                wg.Wait()
                close(ch)
        }()

        // Generate keys
        keys := make([]string, 0, count)
        for k := range ch {
                keys = append(keys, k)
        }
        return keys
}
```

此处利用管道处理多此一举，有炫技的成分，笔者如下改写后，代码简单的同时获得了性能上将近10倍的提升：

```
func (m ConcurrentMap) Keys() []string {
        count := m.Count()
        keys := make([]string, 0, count)
        for _, shard := range m {
                shard.RLock()
                for key := range shard.items {
                        keys = append(keys, key)
                }
                shard.RUnlock()
        }
        return keys
}
```

尽管库的设计有或多或少的缺漏，但最核心的大锁化小锁的思想还是值得学习。

系列目录