优化 Go 中的 map 并发存取

作者：88250
原文链接：https://ld246.com/article/1427941885853
来源网站：链滴

Catena （时序存储引擎）中有一函数的实现备受争议，它从 map 中根据指定的 <code>name</code> 获取一个 <code>metricSource</code>。每一次插入操作都会至少调用一次这个函数，现实场景中该函数调用更是频繁，并且跨多个协程的，因此我们必须要考虑同步。
该函数从 <code>map[string]*metricSource</code> 中根据指定的 <code>name</code> 获取一个指向 <code>metricSource</code> 的指针，如果获取不到则创建一个并返回。其中注意的关键点是我们只会对这个 map 进行插入操作。
简单实现如下：（为节省篇幅，省略了函数头和返回，只贴重要部分）
<pre class="prettyprint">var source *memorySource
var present bool
p.lock.Lock() // lock the mutex
defer p.lock.Unlock() // unlock the mutex at the end
if source, present = p.sources[name]; !present {
// The source wasn't found, so we'll create it.
source = &memorySource{
name: name,
metrics: map[string]*memoryMetric{},
}
<pre><code class="highlight-chroma">// Insert the newly created *memorySource.
p.sources[name] = source
</code></pre>
} </pre>
经测试，该实现大约可以达到 1,400,000 插入/秒（通过协程并发用，<code>GOMAXPROCS</code> 设置为 4）。看上去很快，但实际上它是慢于个协程的，因为多个协程间存在锁竞争。
我们简化一下情况来说明这个问题，假设两个协程分别要获取“a”、“b”，并且“a”、“b都已经存在于该 map 中。上述实现在运行时，一个协程获取到锁、拿指针、解锁、继续执行，此时一个协程会被卡在获取锁。等待锁释放是非常耗时的，并且协程越多性能越差。
让它变快的方法之一是移除锁控制，并保证只有一个协程访问这个 map。这个方法虽然简单，没有伸缩性。下面我们看看另一种简单的方法，并保证了线程安全和伸缩性。
<pre class="prettyprint">var source *memorySource
var present bool
if source, present = p.sources[name]; !present { // added this line
// The source wasn't found, so we'll create it.
<pre><code class="highlight-chroma">

p.lock.Lock() // lock the mutex

defer p.lock.Unlock() // unlock at the end

if source, present = p.sources[name]; !present {

    source = &memorySource{

        name: name,

        metrics: map[string]*memoryMetric{},

    }

    // Insert the newly created *memorySource.

    p.sources[name

原文链接：优化 Go 中的 map 并发存取

= source
}
// if present is true then another goroutine has already inserted
// the element we want, and source is set to what we want.
</code></pre>
} // added this line
// Note that if the source was present, we avoid the lock completely! </pre> </>
该实现可以达到 5,500,000 插入/秒，比第一个版本快 3.93 倍。有 4 个协程在跑测试，结果数值和预期是基本吻合的。
这个实现是 ok 的，因为我们没有删除、修改操作。在 CPU 缓存中的指针地址我们可以安全使，不过要注意的是我们还是需要加锁。如果不加，某协程在创建插入 <code>source</code> 时另一个协程可能已经正在插入，它们会处于竞争状态。这个版本中我们只是在很少情况下加锁，所性能提高了很多。
John Potocny 建议移除 <code>defe</code>，因为会延误解锁时间（要在整个函数返回时才解锁），下面给出一个“终极”版本：
<pre class="prettyprint">var source *memorySource
var present bool
if source, present = p.sources[name]; !present {
// The source wasn't found, so we'll create it.
<pre><code class="highlight-chroma">>>p.lock.Lock() // lock the mutex
>if source, present = p.sources[name]; !present {
>   source = &memorySource{
>   name: name,
>   metrics: map[string]*memoryMetric{},
>   }
>
>   // Insert the newly created *memorySource.
>   p.sources[name] = source
>}
>p.lock.Unlock() // Unlock the mutex
</code></pre>
}
// Note that if the source was present, we avoid the lock completely! </pre> </>
9,800,000 插入/秒！改了 4 行提升到 7 倍啊！有木有！！！！
<hr>
更新：（译注：原作者循序渐进非常赞）
上面实现正确么？No！通过 Go Data Race Detector 我们可以很轻松发现竟态条件，我们不能保证 map 在同时写时的完整性。
下面给出不存在竟态条件、线程安全，应该算是“正确”的版本了。使用了 <code>RW

utex</code>，读操作不会被锁，写操作保持同步。
<pre class="prettyprint">var source *memorySource
var present bool
p.lock.RLock()
if source, present = p.sources[name]; !present {
// The source wasn't found, so we'll create it.
p.lock.RUnlock()
p.lock.Lock()
if source, present = p.sources[name]; !present {
source = &memorySource{
name: name,
metrics: map[string]*memoryMetric{},
}
<pre><code class="highlight-chroma"><span class="highlight
cl">   // Insert the newly created *memorySource.
    p.sources[name
 = source
}
p.lock.Unlock()
</code></pre>
} else {
p.lock.RUnlock()
} </pre>
经测试，该版本性能为其之前版本的 93.8%，在保证正确性的前提能到达这样已经很不错了。也许我们可以认为它们之间根本没有可比性，因为之前的版本是错的。</
>

本文译自：Optimizing Concurrent Map Access in Go

原文链接：优化 Go 中的 map 并发存取