



下载APP



12 | atomic: 要保证原子操作，一定要使用这几种方法

2020-11-06 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 21:27 大小 19.66M



你好，我是鸟窝。

前面我们在学习 Mutex、RWMutex 等并发原语的实现时，你可以看到，最底层是通过 atomic 包中的一些原子操作来实现的。当时，为了让你的注意力集中在这些原语的功能实现上，我并没有展开介绍这些原子操作是干什么用的。

你可能会说，这些并发原语已经可以应对大多数的并发场景了，为啥还要学习原子操作呢？其实，这是因为，在很多场景中，使用并发原语实现起来比较复杂，而原子操作可以帮助我们更轻松地实现底层的优化。



所以，现在，我会专门用一节课，带你仔细地了解一下什么是原子操作，atomic 包都提供了哪些实现原子操作的方法。另外，我还会带你实现一个基于原子操作的数据结构。好

了，接下来我们先来学习下什么是原子操作。

原子操作的基础知识

Package sync/atomic 实现了同步算法底层的原子的内存操作原语，我们把它叫做原子操作原语，它提供了一些实现原子操作的方法。

之所以叫原子操作，是因为一个原子在执行的时候，其它线程不会看到执行一半的操作结果。在其它线程看来，原子操作要么执行完了，要么还没有执行，就像一个最小的粒子 - 原子一样，不可分割。

CPU 提供了基础的原子操作，不过，不同架构的系统的原子操作是不一样的。

对于单处理器单核系统来说，如果一个操作是由一个 CPU 指令来实现的，那么它就是原子操作，比如它的 XCHG 和 INC 等指令。如果操作是基于多条指令来实现的，那么，执行的过程中可能会被中断，并执行上下文切换，这样的话，原子性的保证就被打破了，因为这个时候，操作可能只执行了一半。

在多处理器多核系统中，原子操作的实现就比较复杂了。

由于 cache 的存在，单个核上的单个指令进行原子操作的时候，你要确保其它处理器或者核不访问此原子操作的地址，或者是确保其它处理器或者核总是访问原子操作之后的最新的值。x86 架构中提供了指令前缀 LOCK，LOCK 保证了指令（比如 LOCK CMPXCHG op1、op2）不会受其它处理器或 CPU 核的影响，有些指令（比如 XCHG）本身就提供 Lock 的机制。不同的 CPU 架构提供的原子操作指令的方式也是不同的，比如对于多核的 MIPS 和 ARM，提供了 LL/SC (Load Link/Store Conditional) 指令，可以帮助实现原子操作 (ARMLL/SC 指令 LDREX 和 STREX)。

因为不同的 CPU 架构甚至不同的版本提供的原子操作的指令是不同的，所以，要用一种编程语言实现支持不同架构的原子操作是相当有难度的。不过，还好这些都不需要你操心，因为 Go 提供了一个通用的原子操作的 API，将更底层的不同的架构下的实现封装成 atomic 包，提供了修改类型的原子操作（[@atomic read-modify-write](#)，RMW）和加载存储类型的原子操作（[@Load](#) 和 [Store](#)）的 API，稍后我会一一介绍。

有的代码也会因为架构的不同而不同。有时看起来貌似一个操作是原子操作，但实际上，对于不同的架构来说，情况是不一样的。比如下面的代码的第 4 行，是将一个 64 位的值赋值给变量 i：

[复制代码](#)

```
1  const x int64 = 1 + 1<<33
2
3  func main() {
4      var i = x
5      _ = i
6  }
```

如果你使用 GOARCH=386 的架构去编译这段代码，那么，第 5 行其实是被拆成了两个指令，分别操作低 32 位和高 32 位（使用 GOARCH=386 go tool compile -N -l test.go；GOARCH=386 go tool objdump -gnu test.o 反编译试试）：

```
main.go:5      0x3e0      83ec08      SUBL $0x8, SP      // sub $0x8,%esp
main.go:6      0x3e3      c7042401000000    MOVL $0x1, 0(SP)   // movl $0x1, (%esp)
main.go:6      0x3ea      c744240402000000    MOVL $0x2, 0x4(SP) // movl $0x2, 0x4(%esp)
main.go:8      0x3f2      83c408      ADDL $0x8, SP      // add $0x8,%esp
```

如果 GOARCH=amd64 的架构去编译这段代码，那么，第 5 行其中的赋值操作其实是一条指令：

```
smallnest ...op/3.atomic/assignment master GOARCH=amd64 go tool compile -N -l main.go
smallnest ...op/3.atomic/assignment master GOARCH=amd64 go tool objdump -gnu main.o
EXT %22%22.main(SB) gofile../mnt/c/chaos/go/src/github.com/smallnest/dive-to-gosync-workshop/3.atomic/assignment/main.go
main.go:5      0x38e      4883ec10      SUBQ $0x10, SP      // sub $0x10,%rsp
main.go:5      0x392      48896c2408     MOVQ BP, 0x8(SP)     // mov %rbp,0x8(%rsp)
main.go:5      0x397      488d6c2408     LEAQ 0x8(SP), BP     // lea 0x8(%rsp),%rbp
main.go:6      0x39c      48b80100000002000000    MOVQ $0x200000001, AX // mov $0x200000001,%rax
main.go:6      0x3a6      48890424      MOVQ AX, 0(SP)       // mov %rax, (%rsp)
main.go:8      0x3aa      488b6c2408     MOVQ 0x8(SP), BP     // mov 0x8(%rsp),%rbp
main.go:8      0x3af      4883c410      ADDQ $0x10, SP       // add $0x10,%rsp
main.go:8      0x3b3      c3            RET                  // retq
```

所以，如果要想保证原子操作，切记一定要使用 atomic 提供的方法。

好了，了解了什么是原子操作以及不同系统的不同原子操作，接下来，我来介绍下 atomic 原子操作的应用场景。

atomic 原子操作的应用场景

开篇我说过，使用 atomic 的一些方法，我们可以实现更底层的一些优化。如果使用 Mutex 等并发原语进行这些优化，虽然可以解决问题，但是这些并发原语的实现逻辑比较复杂，对性能还是有一定的影响的。

举个例子：假设你想在程序中使用一个标志（flag，比如一个 bool 类型的变量），来标识一个定时任务是否已经启动执行了，你会怎么做呢？

我们先来看看加锁的方法。如果使用 Mutex 和 RWMutex，在读取和设置这个标志的时候加锁，是可以做到互斥的、保证同一时刻只有一个定时任务在执行的，所以使用 Mutex 或者 RWMutex 是一种解决方案。

其实，这个场景中的问题不涉及到对资源复杂的竞争逻辑，只是会并发地读写这个标志，这类场景就适合使用 atomic 的原子操作。具体怎么做呢？你可以使用一个 uint32 类型的变量，如果这个变量的值是 0，就标识没有任务在执行，如果它的值是 1，就标识已经有任务在完成了。你看，是不是很简单呢？

再来看一个例子。假设你在开发应用程序的时候，需要从配置服务器中读取一个节点的配置信息。而且，在这个节点的配置发生变更的时候，你需要重新从配置服务器中拉取一份新的配置并更新。你的程序中可能有多个 goroutine 都依赖这份配置，涉及到对这个配置对象的并发读写，你可以使用读写锁实现对配置对象的保护。在大部分情况下，你也可以利用 atomic 实现配置对象的更新和加载。

分析到这里，可以看到，这两个例子都可以使用基本并发原语来实现的，只不过，我们不需要这些基本并发原语里面的复杂逻辑，而是只需要其中的简单原子操作，所以，这些场景可以直接使用 atomic 包中的方法去实现。

有时候，你也可以使用 atomic 实现自己定义的基本并发原语，比如 Go issue 有人提议的 CondMutex、Mutex.LockContext、WaitGroup.Go 等，我们可以使用 atomic 或者基于它的更高一级的并发原语去实现。我先前讲的几种基本并发原语的底层（比如 Mutex），就是基于通过 atomic 的方法实现的。

除此之外，atomic 原子操作还是实现 lock-free 数据结构的基石。

在实现 lock-free 的数据结构时，我们可以不使用互斥锁，这样就不会让线程因为等待互斥锁而阻塞休眠，而是让线程保持继续处理的状态。另外，不使用互斥锁的话，lock-free 的数据结构还可以提供并发的性能。

不过，lock-free 的数据结构实现起来比较复杂，需要考虑的东西很多，有兴趣的同学可以看一位微软专家写的一篇经验分享：[🔗 Lockless Programming Considerations for Xbox 360 and Microsoft Windows](#)，这里我们不细谈了。不过，这节课的最后我会带你开发一个 lock-free 的 queue，来学习下使用 atomic 操作实现 lock-free 数据结构的方法，你可以拿它和使用互斥锁实现的 queue 做性能对比，看看在性能上是否有所提升。

看到这里，你是不是觉得 atomic 非常重要呢？不过，要想能够灵活地应用 atomic，我们首先得知道 atomic 提供的所有方法。

atomic 提供的方法

目前的 Go 的泛型的特性还没有发布，Go 的标准库中的很多实现会显得非常啰嗦，多个类型会实现很多类似的方法，尤其是 atomic 包，最为明显。相信泛型支持之后，atomic 的 API 会清爽很多。

atomic 为了支持 int32、int64、uint32、uint64、uintptr、Pointer (Add 方法不支持) 类型，分别提供了 AddXXX、CompareAndSwapXXX、SwapXXX、LoadXXX、StoreXXX 等方法。不过，你也不要担心，你只要记住了一种数据类型的方法的意义，其它数据类型的方法也是一样的。

关于 atomic，还有一个地方你一定要记住，**atomic 操作的对象是一个地址，你需要把可寻址的变量的地址作为参数传递给方法，而不是把变量的值传递给方法。**

好了，下面我就来给你介绍一下 atomic 提供的方法。掌握了这些，你就可以说完全掌握了 atomic 包。

Add

首先，我们来看 Add 方法的签名：

```
func AddInt32(addr *int32, delta int32) (new int32)
func AddInt64(addr *int64, delta int64) (new int64)
func AddUint32(addr *uint32, delta uint32) (new uint32)
func AddUint64(addr *uint64, delta uint64) (new uint64)
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
```

其实，Add 方法就是给第一个参数地址中的值增加一个 delta 值。

对于有符号的整数来说，delta 可以是一个负数，相当于减去一个值。对于无符号的整数和 uintptr 类型来说，怎么实现减去一个值呢？毕竟，atomic 并没有提供单独的减法操作。

我来跟你说一种方法。你可以利用计算机补码的规则，把减法变成加法。以 uint32 类型为例：

```
1 AddUint32(&x, ^uint32(c-1)).
```

[复制代码](#)

如果是对 uint64 的值进行操作，那么，就把上面的代码中的 uint32 替换成 uint64。

尤其是减 1 这种特殊的操作，我们可以简化为：

```
1 AddUint32(&x, ^uint32(0))
```

[复制代码](#)

好了，我们再来看看 CAS 方法。

CAS (CompareAndSwap)

以 int32 为例，我们学习一下 CAS 提供的功能。在 CAS 的方法签名中，需要提供要操作的地址、原数据值、新值，如下所示：


```
1 func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
```

[复制代码](#)

我们来看下这个方法的功能。

这个方法会比较当前 addr 地址里的值是不是 old，如果不等于 old，就返回 false；如果等于 old，就把此地址的值替换成 new 值，返回 true。这就相当于“判断相等才替换”。

如果使用伪代码来表示这个原子操作，代码如下：

 复制代码


```
1 if *addr == old {  
2     *addr = new  
3     return true  
4 }  
5 return false
```

它支持的类型和方法如图所示：

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)  
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)  
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)  
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)  
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)  
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)
```

Swap

如果不需要比较旧值，只是比较粗暴地替换的话，就可以使用 Swap 方法，它替换后还可以返回旧值，伪代码如下：

 复制代码

```
1 old = *addr  
2 *addr = new  
3 return old
```

它支持的数据类型和方法如图所示：

```
func SwapInt32(addr *int32, new int32) (old int32)  
func SwapInt64(addr *int64, new int64) (old int64)  
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)  
func SwapUint32(addr *uint32, new uint32) (old uint32)  
func SwapUint64(addr *uint64, new uint64) (old uint64)  
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

Load

Load 方法会取出 addr 地址中的值，即使在多处理器、多核、有 CPU cache 的情况下，这个操作也能保证 Load 是一个原子操作。

它支持的数据类型和方法如图所示：

```
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
```

Store

Store 方法会把一个值存入到指定的 addr 地址中，即使在多处理器、多核、有 CPU cache 的情况下，这个操作也能保证 Store 是一个原子操作。别的 goroutine 通过 Load 读取出来，不会看到存取了一半的值。

它支持的数据类型和方法如图所示：

```
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
```

Value 类型

刚刚说的都是一些比较常见的类型，其实，atomic 还提供了一个特殊的类型：Value。它可以原子地存取对象类型，但也只能存取，不能 CAS 和 Swap，常常用在配置变更等场景中。


```
type Value
    func (v *Value) Load() (x interface{})
    func (v *Value) Store(x interface{})
```


接下来，我以一个配置变更的例子，来演示 Value 类型的使用。这里定义了一个 Value 类型的变量 config，用来存储配置信息。

首先，我们启动一个 goroutine，然后让它随机 sleep 一段时间，之后就变更一下配置，并通过我们前面学到的 Cond 并发原语，通知其它的 reader 去加载新的配置。

接下来，我们启动一个 goroutine 等待配置变更的信号，一旦有变更，它就会加载最新的配置。

通过这个例子，你可以了解到 Value 的 Store/Load 方法的使用，因为它只有这两个方法，只要掌握了它们的使用，你就完全掌握了 Value 类型。

 复制代码

```
1 type Config struct {
2     NodeName string
3     Addr      string
4     Count     int32
5 }
6
7 func loadNewConfig() Config {
8     return Config{
9         NodeName: "北京",
10        Addr:      "10.77.95.27",
11        Count:     rand.Int31(),
12    }
13 }
14 func main() {
15     var config atomic.Value
16     config.Store(loadNewConfig())
17     var cond = sync.NewCond(&sync.Mutex{})
18
19     // 设置新的config
20     go func() {
21         for {
22             time.Sleep(time.Duration(5+rand.Int63n(5)) * time.Second)
23             config.Store(loadNewConfig())
24             cond.Broadcast() // 通知等待着配置已变更
25         }
26     }()
27
28     go func() {
29         for {
30             cond.L.Lock()
31             cond.Wait() // 等待变更信号
32             c := config.Load().(Config) // 读取新的配置
```

```
33         fmt.Printf("new config: %+v\n", c)
34         cond.L.Unlock()
35     }
36 }()
37
38 select {}
39 }
```

好了，关于标准库的 atomic 提供的方法，到这里我们就学完了。事实上，atomic 包提供了非常好的支持各种平台的一致性的 API，绝大部分项目都是直接使用它。接下来，我再给你介绍一下第三方库，帮助你稍微开拓一下思维。

第三方库的扩展

其实，atomic 的 API 已经算是很简单的了，它提供了包一级的函数，可以对几种类型的数据执行原子操作。

不过有一点让人觉得不爽的是，或者是让熟悉面向对象编程的程序员不爽的是，函数调用有一点点麻烦。所以，有些人就对这些函数做了进一步的包装，跟 atomic 中的 Value 类型类似，这些类型也提供了面向对象的使用方式，比如关注度比较高的 [@uber-go/atomic](#)，它定义和封装了几种与常见类型相对应的原子操作类型，这些类型提供了原子操作的方法。这些类型包括 Bool、Duration、Error、Float64、Int32、Int64、String、Uint32、Uint64 等。

比如 Bool 类型，提供了 CAS、Store、Swap、Toggle 等原子方法，还提供 String、MarshalJSON、UnmarshalJSON 等辅助方法，确实是一个精心设计的 atomic 扩展库。关于这些方法，你一看名字就能猜出来它们的功能，我就不多说了。

其它的数据类型也和 Bool 类型相似，使用起来就像面向对象的编程一样，你可以看下下面的这段代码。

```
1  var running atomic.Bool
2  running.Store(true)
3  running.Toggle()
4  fmt.Println(running.Load()) // false
```

[复制代码](#)

使用 atomic 实现 Lock-Free queue

atomic 常常用来实现 Lock-Free 的数据结构，这次我会给你展示一个 Lock-Free queue 的实现。

Lock-Free queue 最出名的就是 Maged M. Michael 和 Michael L. Scott 1996 年发表的 [论文](#) 中的算法，算法比较简单，容易实现，伪代码的每一行都提供了注释，我就不在这里贴出伪代码了，因为我们使用 Go 实现这个数据结构的代码几乎和伪代码一样：

[复制代码](#)

```
1 package queue
2 import (
3     "sync/atomic"
4     "unsafe"
5 )
6 // lock-free的queue
7 type LKQueue struct {
8     head unsafe.Pointer
9     tail unsafe.Pointer
10 }
11 // 通过链表实现，这个数据结构代表链表中的节点
12 type node struct {
13     value interface{}
14     next  unsafe.Pointer
15 }
16 func NewLKQueue() *LKQueue {
17     n := unsafe.Pointer(&node{})
18     return &LKQueue{head: n, tail: n}
19 }
20 // 入队
21 func (q *LKQueue) Enqueue(v interface{}) {
22     n := &node{value: v}
23     for {
24         tail := load(&q.tail)
25         next := load(&tail.next)
26         if tail == load(&q.tail) { // 尾还是尾
27             if next == nil { // 还没有新数据入队
28                 if cas(&tail.next, next, n) { //增加到队尾
29                     cas(&q.tail, tail, n) //入队成功，移动尾巴指针
30                     return
31                 }
32             } else { // 已有新数据加到队列后面，需要移动尾指针
33                 cas(&q.tail, tail, next)
34             }
35         }
36     }
37 }
```

```
38 // 出队, 没有元素则返回nil
39 func (q *LKQueue) Dequeue() interface{} {
40     for {
41         head := load(&q.head)
42         tail := load(&q.tail)
43         next := load(&head.next)
44         if head == load(&q.head) { // head还是那个head
45             if head == tail { // head和tail一样
46                 if next == nil { // 说明是空队列
47                     return nil
48                 }
49                 // 只是尾指针还没有调整, 尝试调整它指向下一个
50                 cas(&q.tail, tail, next)
51             } else {
52                 // 读取出队的数据
53                 v := next.value
54                 // 既然要出队了, 头指针移动到下一个
55                 if cas(&q.head, head, next) {
56                     return v // Dequeue is done. return
57                 }
58             }
59         }
60     }
61 }
62
63 // 将unsafe.Pointer原子加载转换成node
64 func load(p *unsafe.Pointer) (n *node) {
65     return (*node)(atomic.LoadPointer(p))
66 }
67
68 // 封装CAS,避免直接将*node转换成unsafe.Pointer
69 func cas(p *unsafe.Pointer, old, new *node) (ok bool) {
70     return atomic.CompareAndSwapPointer(
71         p, unsafe.Pointer(old), unsafe.Pointer(new))
72 }
```

我来给你介绍下这里的主要逻辑。

这个 lock-free 的实现使用了一个辅助头指针 (head)，头指针不包含有意义的数据，只是一个辅助的节点，这样的话，出队入队中的节点会更简单。

入队的时候，通过 CAS 操作将一个元素添加到队尾，并且移动尾指针。

出队的时候移除一个节点，并通过 CAS 操作移动 head 指针，同时在必要的时候移动尾指针。

总结

好了，我们来小结一下。这节课，我们学习了 atomic 的基本使用方法，以及它提供的几种方法，包括 Add、CAS、Swap、Load、Store、Value 类型。除此之外，我还介绍了一些第三方库，并且带你实现了 Lock-free queue。到这里，相信你已经掌握了 atomic 提供的各种方法，并且能够应用到实践中了。

最后，我还想和你讨论一个额外的问题：对一个地址的赋值是原子操作吗？

这是一个很有趣的问题，如果是原子操作，还要 atomic 包干什么？官方的文档中并没有特意的介绍，不过，在一些 issue 或者论坛中，每当有人谈到这个问题时，总是会被建议用 atomic 包。

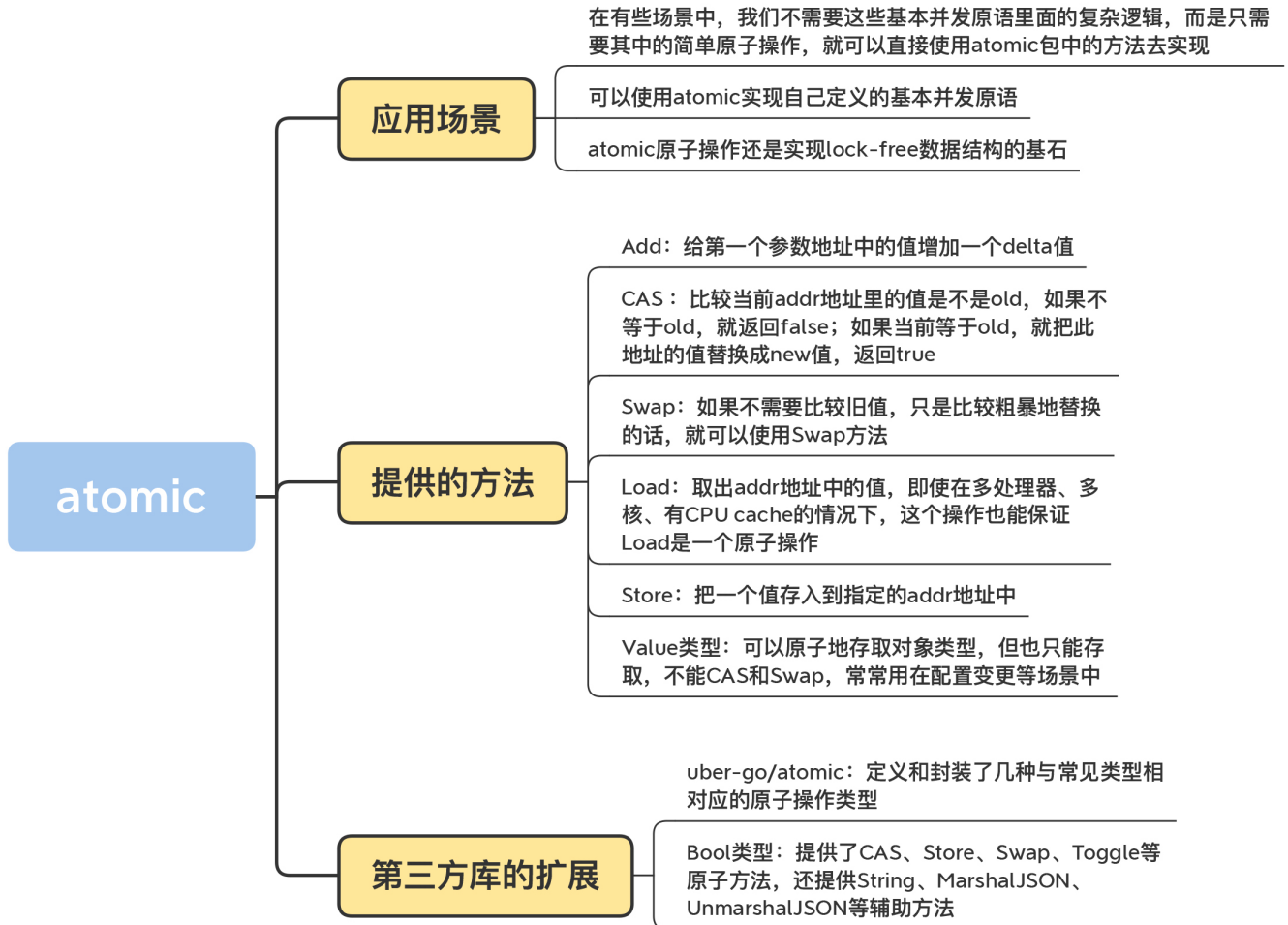
🔗 [Dave Cheney](#)就谈到过这个问题，讲得非常好。我来给你总结一下他讲的知识点，这样你就比较容易理解使用 atomic 和直接内存操作的区别了。

在现在的系统中，write 的地址基本上都是对齐的 (aligned)。比如，32 位的操作系统、CPU 以及编译器，write 的地址总是 4 的倍数，64 位的系统总是 8 的倍数（还记得 WaitGroup 针对 64 位系统和 32 位系统对 state1 的字段不同的处理吗）。对齐地址的写，不会导致其他人看到只写了一半的数据，因为它通过一个指令就可以实现对地址的操作。如果地址不是对齐的话，那么，处理器就需要分成两个指令去处理，如果执行了一个指令，其它人就会看到更新了一半的错误的数据，这被称做撕裂写 (torn write)。所以，你可以认为赋值操作是一个原子操作，这个“原子操作”可以认为是保证数据的完整性。

但是，对于现代的多处理多核的系统来说，由于 cache、指令重排，可见性等问题，我们对原子操作的意义有了更多的追求。在多核系统中，一个核对地址的值的更改，在更新到主内存中之前，是在多级缓存中存放的。这时，多个核看到的数据可能是不一样的，其它的核可能还没有看到更新的数据，还在使用旧的数据。

多处理器多核心系统为了处理这类问题，使用了一种叫做内存屏障 (memory fence 或 memory barrier) 的方式。一个写内存屏障会告诉处理器，必须要等到它管道中的未完成的写操作（特别是写操作）都被刷新到内存中，再进行操作。此操作还会让相关的处理器的 CPU 缓存失效，以便让它们从主存中拉取最新的值。

atomic 包提供的方法会提供内存屏障的功能，所以，atomic 不仅仅可以保证赋值的数据完整性，还能保证数据的可见性，一旦一个核更新了该地址的值，其它处理器总是能读取到它的最新值。但是，需要注意的是，因为需要处理器之间保证数据的一致性，atomic 的操作也是会降低性能的。



思考题

atomic.Value 只有 Load/Store 方法，你是不是感觉意犹未尽？你可以尝试为 Value 类型增加 Swap 和 CompareAndSwap 方法（可以参考一下 [这份资料](#)）。

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，也欢迎你今天的分享内容分享给你的朋友或同事。

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 11 | Context: 信息穿透上下文

下一篇 13 | Channel: 另辟蹊径，解决并发问题

精选留言 (9)

 写留言



myrfy

2020-11-06

恰好老婆大人是做芯片MMU相关工作的，咨询了一下她，她告诉我现代的CPU基本上都在硬件层面保证了多核之间数据视图的一致性，也就是说普通的LOAD/STORE命令在硬件层面处理器就可以保证cache的一致性。如果是这样的话，那是不是可以理解为atomic包对指针的作用，主要是防止编译器做指令重排呢？因为编译器在这些现代架构上没必要使用特殊的指令了。...

展开

作者回复: atomic主要是对这几种cpu架构的封装。你老婆是对的，你可以好好请教一下你老婆

 3

 8

**蜉蝣**

2020-11-14

这个 lock-free queue 是能看懂，但要自己写出来就感觉有点难了。就譬如 `tail == load(&q.tail)` 和 `head == load(&q.head)` 的检查，我就想不到还要再做一次检查。前面章节看源码的时候也有这种感觉，能看懂，但自己写肯定想不到哪里要多检查一次。

展开 ∨

**端贺**

2020-11-14

晁老师的内功真是深厚，整个系列读下来还是有点吃力的，尤其是文中推荐的外链，需要多花点时间好好消化，感谢晁老师。

展开 ∨

作者回复: 加油!!! 赞你认真的态度

**SuperDai**

2020-11-12

老师，无锁队列对消费者数量和生产者数量是不是有要求？是不是要求消费者数量为1还是生产者数量为1？

作者回复: 没有要求，但我个人觉得数量不易过大

**SuperDai**

2020-11-12

老师，无锁队列对消费者数量和生产者数量是不是有要求？是不是要求消费者数量为1还是生产者数量为1？

作者回复: 没有要求

**李金狗**

2020-11-11

`cas(&q.tail, tail, n)` //入队成功，移动尾巴指针 :这一步有失败的风险吧。



橙子888
2020-11-06

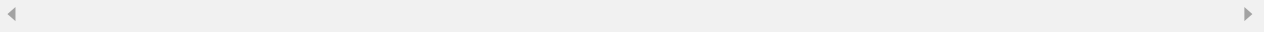
打卡。
展开 ▾



末班车
2020-11-06

老师您好，之前在用atomic的时候，疑惑为啥没有提供it int16的相关方法，这是不是也跟内存对齐有关系啊？
展开 ▾

作者回复: Go官方运行时只支持32bit/64bit系统，最小支持单位就是32bit



青生先森
2020-11-06

每周一三五早上，打卡，最后的总结涨知识了。
展开 ▾

