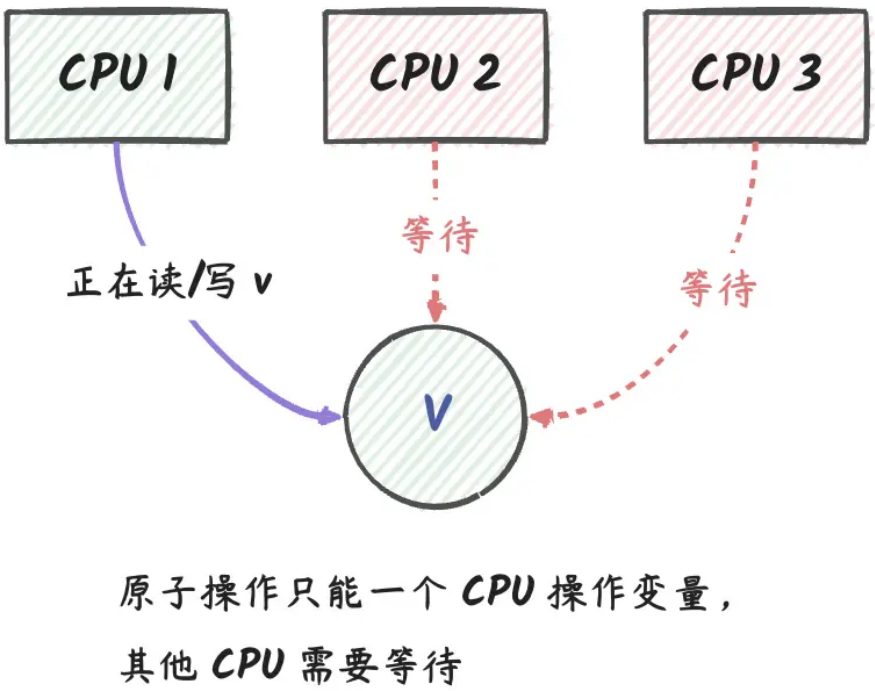
在我们前面的一些介绍 `sync` 包相关的文章中，我们应该也发现了，其中有不少地方使用了原子操作。比如 `sync.WaitGroup`、`sync.Map` 再到 `sync.Pool`，这些结构体的实现中都有原子操作的身影。原子操作在并发编程中是一种非常重要的操作，它可以保证并发安全，而且效率也很高。本文将会深入探讨一下 go 中原子操作的原理、使用场景、用法等内容。

什么是原子操作?

> 原子操作是变量级别的互斥锁。

如果让我用一句话来说明什么是原子操作，那就是：**原子操作是变量级别的互斥锁。**简单来说，就是同一时刻，只能有一个 CPU 对变量进行读或写。当我们想要对某个**变量**做并发安全的修改，除了使用官方提供的 `Mutex`，还可以使用 `sync/atomic` 包的原子操作，它能够保证对变量的读取或修改期间不被其他的协程所影响。

我们可以用下图来表示：



说明：在上图中，我们有三个 CPU 逻辑核，其中 CPU 1 正在对变量 `v` 做原子操作，这个时候 CPU 2 和 CPU 3 不能对 `v` 做任何操作，在 CPU 1 操作完成后，CPU 2 和 CPU 3 可以获取到 `v` 的最新值。

> 从这个角度看，我们可以把 `sync/atomic` 包中的原子操作看成是变量级别的互斥锁。就是说，在 go 中，当一个协程对变量做原子操作时，其他协程不能对这个变量做任何操作，直到这个协程操作完成。

原子操作的使用场景是什么?

拿一个简单的例子来说明一下原子操作的使用场景：

```
func TestAtomic(t *testing.T) {
var sum = 0
```

```
var wg sync.WaitGroup
wg.Add(1000)

// 启动 1000 个协程，每个协程对 sum 做加法操作
for i := 0; i < 1000; i++ {
go func() {
defer wg.Done()
sum++
}()
}

// 等待所有的协程都执行完毕
wg.Wait()
fmt.Println(sum) // 这里输出多少呢?
}
```

我们可以在自己的电脑上运行一下这段代码，看看输出的结果是多少。不出意外的话，应该每次可能都不一样，而且应该也不是 1000，这是为什么呢?

这是因为，CPU 在对 `sum` 做加法的时候，需要先将 `sum` 目前的值读取到 CPU 的寄存器中，然后再进行加法操作，最后再写回到内存中。如果有两个 CPU 同时取了 `sum` 的值，然后都进行了加法操作，然后都再写回到内存中，那么就会导致 `sum` 的值被覆盖，从而导致结果不正确。

举个例子，目前内存中的 `sum` 为 1，然后两个 CPU 同时取了这个 1 来做加法，然后都得到了结果 2，然后这两个 CPU 将各自的计算结果写回到内存中，那么内存中的 `sum` 就变成了 2，而不是 3。

在这种场景下，我们可以使用原子操作来实现并发安全的加法操作：

```
func TestAtomic1(t *testing.T) {
// 将 sum 的类型改成 int32，因为原子操作只能针对 int32、int64、uint32、uint64、uintptr 这几种类型
var sum int32 = 0
var wg sync.WaitGroup
wg.Add(1000)

// 启动 1000 个协程，每个协程对 sum 做加法操作
for i := 0; i < 1000; i++ {
go func() {
defer wg.Done()
// 将 sum++ 改成下面这样
atomic.AddInt32(&sum, 1)
}()
```

```
}


wg.Wait()
fmt.Println(sum) // 输出 1000
}
```

在上面这个例子中，我们每次执行都能得到 1000 这个结果。

> 因为使用原子操作的时候，同一时刻只能有一个 CPU 对变量进行读或写，所以就不会出现上面的问题了。

所以很多需要对变量做并发读写的地方，我们都可以考虑一下，是否可以使用原子操作来实现并发安全的操作（而不是使用互斥锁，互斥锁效率相比原子操作要低一些）。

原子操作是怎么实现的?

看完上面原子操作的介绍，有没有觉得原子操作很神奇，居然有这么好用的东西。那它到底是怎么实现的呢?

一般情况下，原子操作的实现需要特殊的 CPU 指令或者系统调用。这些指令或者系统调用可以保证在执行期间不会被其他操作或事件中断，从而保证操作的原子性。

例如，在 x86 架构的 CPU 中，可以使用 `LOCK` 前缀来实现原子操作。`LOCK` 前缀可以与其他指令一起使用，用于锁定内存总线，防止其他 CPU 访问同一内存地址，从而实现原子操作。在使用 `LOCK` 前缀的指令执行期间，CPU 会将当前处理器缓存中的数据写回到内存中，并锁定该内存地址，防止其他 CPU 修改该地址的数据（所以原子操作总是可以读取到最新的数据）。一旦当前 CPU 对该地址的操作完成，CPU 会释放该内存地址的锁定，其他 CPU 才能继续对该地址进行访问。

x86 LOCK 的时候发生了什么

我们再来捋一下上面的内容，看看 `LOCK` 前缀是如何实现原子操作的：

1. CPU 会将当前处理器缓存中的数据写回到内存中。（因此我们总能读取到最新的数据）
2. 然后锁定该内存地址，防止其他 CPU 修改该地址的数据。
3. 一旦当前 CPU 对该地址的操作完成，CPU 会释放该内存地址的锁定，其他 CPU 才能继续对该地址进行访问。

> 其他架构的 CPU 可能会略有不同，但是原理是一样的。

原子操作有什么特征?

1. 不会被中断：原子操作是一个不可分割的操作，要么全部执行，要么全部不执行，不会出现中间状态。这是保证原子性的基本前提。同时，**原子操作过程中不会有上下文切换的过程。**
2. 操作对象是共享变量：原子操作通常是对共享变量进行的，也就是说，多个协程可以同时访问这个变量，因此需要采用原子操作来保证数据的一致性和正确性。
3. 并发安全：原子操作是并发安全的，可以保证多个协程同时进行操作时不会出现数据竞争问题（虽然说是同时，但是实际上在操作那个变量的时候是互斥的）。
4. 无需加锁：原子操作不需要使用互斥锁来保证数据的一致性和正确性，因此可以避免互斥锁的使用带来的性能损失。

5. 适用场景比较局限：原子操作适用于操作单个变量，如果需要同时并发读写多个变量，可能需要考虑使用互斥锁。

go 里面有哪些原子操作?

在 go 中，主要有以下几种原子操作：`Add`、`CompareAndSwap`、`Load`、`Store`、`Swap`。

增减（Add）

1. 用于进行增加或减少的原子操作，函数名以 `Add` 为前缀，后缀针对特定类型的名称。
2. 原子增被操作的类型只能是数值类型，即 `int32`、`int64`、`uint32`、`uint64`、`uintptr`
3. 原子增减函数的第一个参数为原值，第二个参数是要增减多少。
4. 方法：

```
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)
```

比较并交换（CompareAndSwap）

也就是我们常见的 `CAS`，在 `CAS` 操作中，会需要拿旧的值跟 `old` 比较，如果相等，就将 `new` 赋值给 `addr`。如果不相等，则不做任何操作。最后返回一个 `bool` 值，表示是否成功 `swap`。

也就是说，这个操作可能是不成功的。这很正常，在并发环境下，多个协程对同一个变量进行操作，肯定会存在竞争的情况。在这种情况下，偶尔的失败是正常的，我们只需要在失败的时候，重新尝试即可。因为原子操作需要的时间往往是比较短的，因此在失败的时候，我们可以通过自旋的方式来再次进行尝试。

> 在这种情况下，如果不自旋，那就需要将这个协程挂起，等待其他协程完成操作，然后再次尝试。这个过程相比自旋可能会更加耗时。因为很有可能这次原子操作不成功，下一次就成功了。如果我们每次都将协程挂起，那么效率就会大大降低。

`for` + 原子操作的方式，在 go 的 `sync` 包中很多地方都有使用，比如 `sync.Map`，`sync.Pool` 等。这也是使用原子操作时一个非常常见的使用模式。

`CompareAndSwap` 的功能：

1. 用于比较并交换的原子操作，函数名以 `CompareAndSwap` 为前缀，后缀针对特定类型的名称。
2. 原子比较并交换被操作的类型可以是数值类型或指针类型，即 `int32`、`int64`、`uint32`、`uint64`、`uintptr`、`unsafe.Pointer`
3. 原子比较并交换函数的第一个参数为原值指针，第二个参数是要比较的值，第三个参数是要交换的值。
4. 方法：

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(addr *int64, old, new int64) (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)
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
```

载入（Load）

原子性的读取操作接受一个对应类型的指针值，返回该指针指向的值。原子性读取意味着读取值的同时，当前计算机的任何 CPU 都不会进行针对值的读写操作。

如果不使用原子 `Load`，当使用 `v := value` 这种赋值方式为变量 `v` 赋值时，读取到的 `value` 可能不是最新的，因为在读取操作时其他协程对它的读写操作可能会同时发生。

Load 操作有下面这些：

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

存储（Store）

`Store` 可以将 `val` 值保存到 `*addr` 中，`Store` 操作是原子性的，因此在执行 `Store` 操作时，当前计算机的任何 CPU 都不会进行针对 `*addr` 的读写操作。

1. 原子性存储会将 `val` 值保存到 `*addr` 中。

2. 与读操作对应的写入操作，`sync/atomic` 提供了与原子值载入 `Load` 函数相对应的原子值存储 `Store` 函数，原子性存储函数均以 `Store` 为前缀。

`Store` 操作有下面这些：

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

交换（Swap）

`Swap` 跟 `Store` 有点类似，但是它会返回 `*addr` 的旧值。

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

原子操作的使用场景是什么?

文章开头的地方，我们已经说了，原子操作本质上是一种**变量级别的互斥锁**。因此，原子操作的使用场景也是和互斥锁类似的，但是不一样的是，我们的锁粒度只是一个变量而已。也就是说，当我们不允许多个 CPU 同时对变量进行读写的时候（保证变量同一时刻只能一个 CPU 操作），就可以使用原子操作。

原子操作任意类型的值 – atomic.Value

从上一节中，我们知道了在 go 中原子操作可以操作 `int32`、`int64`、`uint32`、`uint64`、`uintptr`、`unsafe.Pointer` 这些类型的值。但是在实际开发中，我们的类型还有很多，比如 `string`、`struct` 等等，那这些类型的值如何进行原子操作呢？答案是使用 `atomic.Value`。

`atomic.Value` 是一个结构体，它的内部有一个 `any` 类型的字段，存储了我们要原子操作的值，也就是一个任意类型的值。

`atomic.Value` 支持以下操作：

- `Load`：原子性的读取 `Value` 中的值。
- `Store`：原子性的存储一个值到 `Value` 中。
- `Swap`：原子性的交换 `Value` 中的值，返回旧值。
- `CompareAndSwap`：原子性的比较并交换 `Value` 中的值，如果旧值和 `old` 相等，则将 `new` 存入 `Value` 中，返回 `true`，否则返回 `false`。

`atomic.Value` 的这些操作跟上面讲到的那些操作其实差不多，只不过 `atomic.Value` 可以操作任意类型的值。那 `atomic.Value` 是如何实现的呢?

atomic.Value 源码分析

`atomic.Value` 是一个结构体，这个结构体只有一个字段：

```
// Value 提供一致类型值的原子加载和存储。type Value struct { v any}复制代码
```

Load – 读取

`Load` 返回由最近的 `Store` 设置的值。如果还没有 `Store` 过任何值，则返回 `nil`。

```
// Load 返回由最近的 Store 设置的值。
func (v *Value) Load() (val any) {
// atomic.Value 转换为 efaceWords
vp := (*efaceWords)(unsafe.Pointer(v))

// 判断 atomic.Value 的类型
typ := LoadPointer(&vp.typ)
// 第一次 Store 还没有完成，直接返回 nil
if typ == nil || typ == unsafe.Pointer(&firstStoreInProgress) {
// firstStoreInProgress 是一个特殊的变量，存储到 typ 中用来表示第一次 Store 还没有完成
return nil
}
```

```
// 获取 atomic.Value 的值
data := LoadPointer(&vp.data)
// 将 val 转换为 efaceWords 类型
vlp := (*efaceWords)(unsafe.Pointer(&val))
// 分别赋值给 val 的 typ 和 data
vlp.typ = typ
vlp.data = data
return
}
```

在 `atomic.Value` 的源码中，我们都可以看到 `efaceWords` 的身影，它实际上代表的是 `interface{}/any` 类型：

```
// 表示一个 interface{}/any 类型type efaceWords struct { typ unsafe.Pointer data unsafe.Pointer}复制代码
```

看到这里我们会不会觉得很困惑，直接返回 `val` 不就可以了吗？为什么要将 `val` 转换为 `efaceWords` 类型呢？

这是因为 go 中的原子操作只能操作 `int32`、`int64`、`uint32`、`uint64`、`uintptr`、`unsafe.Pointer` 这些类型的值，不支持 `interface{}` 类型，但是如果了解 `interface{}` 底层结构的话，我们就知道 `interface{}` 底层其实就是一个结构体，它有两个字段，一个是 `type`，一个是 `data`，`type` 用来存储 `interface{}` 的类型，`data` 用来存储 `interface{}` 的值。而且这两个字段都是 `unsafe.Pointer` 类型的，所以其实我们可以对 `interface{}` 的 `type` 和 `data` 分别进行原子操作，这样最终其实也可以达到了原子操作 `interface{}` 的目的了，是不是非常地巧妙呢？

Store – 存储

`Store` 将 `Value` 的值设置为 `val`。对给定值的所有存储调用必须使用相同具体类型的值。不一致类型的存储会发生恐慌，`Store(nil)` 也会 `panic`。

```
// Store 将 Value 的值设置为 val。
func (v *Value) Store(val any) {
// 不能存储 nil 值
if val == nil {
panic("sync/atomic: store of nil value into Value")
}
// atomic.Value 转换为 efaceWords
vp := (*efaceWords)(unsafe.Pointer(v))
// val 转换为 efaceWords
vlp := (*efaceWords)(unsafe.Pointer(&val))
// 自旋进行原子操作，这个过程不会很久，开销相比互斥锁小
for {
```

```
// LoadPointer 可以保证获取到的是最新的
typ := LoadPointer(&vp.typ)
// 第一次 store 的时候 typ 还是 nil，说明是第一次 store
if typ == nil {
// 尝试开始第一次 Store。
// 禁用抢占，以便其他 goroutines 可以自旋等待完成。
// （如果允许抢占，那么其他 goroutine 自旋等待的时间可能会比较长，因为可能会需要进行协程调度。）
runtime_procPin()
// 抢占失败，意味着有其他 goroutine 成功 store 了，允许抢占，再次尝试 Store
// 这也是一个原子操作。
if !CompareAndSwapPointer(&vp.typ, nil, unsafe.Pointer(&firstStoreInProgress)) {
runtime_procUnpin()
continue
}
// 完成第一次 store
// 因为有 firstStoreInProgress 标识的保护，所以下面的两个原子操作是安全的。
StorePointer(&vp.data, vlp.data) // 存储值（原子操作）
StorePointer(&vp.typ, vlp.typ) // 存储类型（原子操作）
runtime_procUnpin() // 允许抢占
return
}

// 另外一个 goroutine 正在进行第一次 Store。自旋等待。
if typ == unsafe.Pointer(&firstStoreInProgress) {
continue
}

// 第一次 Store 已经完成了，下面不是第一次 Store 了。
// 需要检查当前 Store 的类型跟第一次 Store 的类型是否一致，不一致就 panic。
if typ != vlp.typ {
panic("sync/atomic: store of inconsistently typed value into Value")
}

// 后续的 Store 只需要 Store 值部分就可以了。
// 因为 atomic.Value 只能保存一种类型的值。
StorePointer(&vp.data, vlp.data)
```

```
return
}
}
```

在 `Store` 中，有以下几个注意的点：

1. 使用 `firstStoreInProgress` 来确保第一次 `Store` 的时候，只有一个 `goroutine` 可以进行 `Store` 操作，其他的 `goroutine`需要自旋等待。如果没有这个保护，那么存储 `typ` 和 `data` 的时候就会出现竞争（因为需要两个原子操作），导致数据不一致。在这里其实可以将 `firstStoreInProgress` 看作是一个互斥锁。
2. 在进行第一次 `Store` 的时候，会将当前的 goroutine 和 `P` 绑定，这样拿到 `firstStoreInProgress` 锁的协程就可以尽快地完成第一次 `Store`操作，这样一来，其他的协程也不用等待太久。
3. 在第一次 `Store` 的时候，会有两个原子操作，分别存储类型和值，但是因为有 `firstStoreInProgress` 的保护，所以这两个原子操作本质上是对 `interface{}` 的一个原子存储操作。
4. 其他协程在看到有 `firstStoreInProgress` 标识的时候，就会自旋等待，直到第一次 `Store` 完成。
5. 在后续的 `Store` 操作中，只需要存储值就可以了，因为 `atomic.Value` 只能保存一种类型的值。

Swap – 交换

`Swap` 将 `Value` 的值设置为 `new` 并返回旧值。对给定值的所有交换调用必须使用相同具体类型的值。同时，不一致类型的交换会发生恐慌，`Swap(nil)` 也会 `panic`。

```
// Swap 将 Value 的值设置为 new 并返回旧值。
func (v *Value) Swap(new any) (old any) {
// 不能存储 nil 值
if new == nil {
panic("sync/atomic: swap of nil value into Value")
}

// atomic.Value 转换为 efaceWords
vp := (*efaceWords)(unsafe.Pointer(v))
// new 转换为 efaceWords
np := (*efaceWords)(unsafe.Pointer(&new))
// 自旋进行原子操作，这个过程不会很久，开销相比互斥锁小
for {
// 下面这部分代码跟 Store 一样，不细说了。
// 这部分代码是进行第一次存储的代码。
typ := LoadPointer(&vp.typ)
if typ == nil {
runtime_procPin()
if !CompareAndSwapPointer(&vp.typ, nil, unsafe.Pointer(&firstStoreInProgress)) {
runtime_procUnpin()
```

```
continue
}
StorePointer(&vp.data, np.data)
StorePointer(&vp.typ, np.typ)
runtime_procUnpin()
return nil
}
if typ == unsafe.Pointer(&firstStoreInProgress) {
continue
}
if typ != np.typ {
panic("sync/atomic: swap of inconsistently typed value into Value")
}

// ---- 下面是 Swap 的特有逻辑 ----
// op 是返回值
op := (*efaceWords)(unsafe.Pointer(&old))
// 返回旧的值
op.typ, op.data = np.typ, SwapPointer(&vp.data, np.data)
return old
}
}
```

CompareAndSwap – 比较并交换

`CompareAndSwap` 将 `Value` 的值与 `old` 比较，如果相等则设置为 `new` 并返回 `true`，否则返回 `false`。对给定值的所有比较和交换调用必须使用相同具体类型的值。同时，不一致类型的比较和交换会发生恐慌，`CompareAndSwap(nil, nil)` 也会 `panic`。

```
// CompareAndSwap 比较并交换。
func (v *Value) CompareAndSwap(old, new any) (swapped bool) {
// 注意：old 是可以为 nil 的，new 不能为 nil。
// old 是 nil 表示是第一次进行 Store 操作。
if new == nil {
panic("sync/atomic: compare and swap of nil value into Value")
}

// atomic.Value 转换为 efaceWords
vp := (*efaceWords)(unsafe.Pointer(v))
// new 转换为 efaceWords
```

```
np := (*efaceWords)(unsafe.Pointer(&new))
// old 转换为 efaceWords
op := (*efaceWords)(unsafe.Pointer(&old))

// old 和 new 类型必须一致，且不能为 nil
if op.typ != nil && np.typ != op.typ {
panic("sync/atomic: compare and swap of inconsistently typed values")
}

// 自旋进行原子操作，这个过程不会很久，开销相比互斥锁小
for {
// LoadPointer 可以保证获取到的 typ 是最新的
typ := LoadPointer(&vp.typ)
if typ == nil { // atomic.Value 是 nil，还没 Store 过
// 准备进行第一次 Store，但是传递进来的 old 不是 nil，compare 这一步就失败了。直接返回 false
if old != nil {
return false
}

// 下面这部分代码跟 Store 一样，不细说了。
// 这部分代码是进行第一次存储的代码。
runtime_procPin()
if !CompareAndSwapPointer(&vp.typ, nil, unsafe.Pointer(&firstStoreInProgress)) {
runtime_procUnpin()
continue
}
StorePointer(&vp.data, np.data)
StorePointer(&vp.typ, np.typ)
runtime_procUnpin()
return true
}
if typ == unsafe.Pointer(&firstStoreInProgress) {
continue
}
if typ != np.typ {
panic("sync/atomic: compare and swap of inconsistently typed value into Value")
}
```

```
// 通过运行时相等性检查比较旧版本和当前版本。
// 这允许对值类型进行比较，这是包函数所没有的。
// 下面的 CompareAndSwapPointer 仅确保 vp.data 自 LoadPointer 以来没有更改。
data := LoadPointer(&vp.data)
var i any
(*efaceWords)(unsafe.Pointer(&i)).typ = typ
(*efaceWords)(unsafe.Pointer(&i)).data = data
if i != old { // atomic.Value 跟 old 不相等
return false
}
// 只做 val 部分的 cas 操作
return CompareAndSwapPointer(&vp.data, data, np.data)
}
}
```

这里需要特别说明的只有最后那个比较相等的判断，也就是 `data := LoadPointer(&vp.data)` 以及往后的几行代码。在开发 `atomic.Value` 第一版的时候，那个开发者其实是将这几行写成 `CompareAndSwapPointer(&vp.data, old.data, np.data)` 这种形式的。但是在旧的写法中，会存在一个问题，如果我们做 `CAS` 操作的时候，如果传递的参数 `old` 是一个结构体的值这种类型，那么这个结构体的值是会被拷贝一份的，同时再会被转换为 `interface{}/any` 类型，这个过程中，其实参数的 `old` 的 `data` 部分指针指向的内存跟 `vp.data` 指向的内存是不一样的。这样的话，`CAS` 操作就会失败，这个时候就会返回 `false`，但是我们本意是要比较它的值，出现这种结果显然不是我们想要的。

> 将值作为 `interface{}` 参数使用的时候，会存在一个将值转换为 `interface{}` 的过程。具体我们可以看看 `interface{}` 的实现原理。

所以，在上面的实现中，会将旧值的 `typ` 和 `data` 赋值给一个 `any` 类型的变量，然后使用 `i != old` 这种方式进行判断，这样就可以实现在比较的时候，比较的是值，而不是由值转换为 `interface{}` 后的指针。

其他原子类型

我们现在知道了，`atomic.Value` 可以对任意类型做原子操作。而对于其他的原子类型，比如 `int32`、`int64`、`uint32`、`uint64`、`uintptr`、`unsafe.Pointer` 等，其实在 go 中也提供了包装的类型，让我们可以以对象的方式来操作这些类型。

对应的类型如下：

- `atomic.Bool`：这个比较特别，但底层实际上是一个 `uint32` 类型的值。我们对 `atomic.Bool` 做原子操作的时候，实际上是对 `uint32` 做原子操作。
- `atomic.Int32`：`int32` 类型的包装类型
- `atomic.Int64`：`int64` 类型的包装类型

- `atomic.Uint32`：`uint32` 类型的包装类型
- `atomic.Uint64`：`uint64` 类型的包装类型
- `atomic.Uintptr`：`uintptr` 类型的包装类型
- `atomic.Pointer`：`unsafe.Pointer` 类型的包装类型

这几种类型的实现的代码基本一样，除了类型不一样，我们可以看看 `atomic.Int32` 的实现：

```
// An Int32 is an atomic int32. The zero value is zero.
type Int32 struct {
_ noCopy
v int32
}

// Load atomically loads and returns the value stored in x.
func (x *Int32) Load() int32 { return LoadInt32(&x.v) }

// Store atomically stores val into x.
func (x *Int32) Store(val int32) { StoreInt32(&x.v, val) }

// Swap atomically stores new into x and returns the previous value.
func (x *Int32) Swap(new int32) (old int32) { return SwapInt32(&x.v, new) }

// CompareAndSwap executes the compare-and-swap operation for x.
func (x *Int32) CompareAndSwap(old, new int32) (swapped bool) {
return CompareAndSwapInt32(&x.v, old, new)
}
```

可以看到，`atomic.Int32` 的实现都是基于 `atomic` 包中 `int32` 类型相关的原子操作函数来实现的。

原子操作与互斥锁比较

那我们有了互斥锁，为什么还要有原子操作呢？我们进行比较一下就知道了：

原子操作	互斥锁	
保护的范围	变量	代码块
保护的粒度	小	大
性能	高	低
如何实现的	硬件指令	软件层面实现，逻辑较多

> 如果我们只需要对某一个变量做并发读写，那么使用原子操作就可以了，因为原子操作的性能比互斥锁高很多。但是如果我们需要对多个变量做并发读写，那么就需要用到互斥锁了，这种场景往往是在一段代码中对不同变量做读写。

性能比较

我们前面这个表格提到了原子操作与互斥锁性能上有差异，我们写几行代码来进行比较一下：

```
// 系统信息 cpu: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
// 10.13 ns/op
func BenchmarkMutex(b *testing.B) {
var mu sync.Mutex

for i := 0; i < b.N; i++ {
mu.Lock()
mu.Unlock()
}
}

// 5.849 ns/op
func BenchmarkAtomic(b *testing.B) {
var sum atomic.Uint64

for i := 0; i < b.N; i++ {
sum.Add(uint64(1))
}
}
```

在对 `Mutex` 的性能测试中，我只是写了简单的 `Lock()` 和 `UnLock()` 操作，因为这种比较才算是对 `Mutex` 本身的测试，而在 `Atomic` 的性能测试中，对 `sum` 做原子累加的操作。最终结果是，使用 `Atomic` 的操作耗时大概比 `Mutex` 少了 `40%` 以上。

> 在实际开发中，`Mutex` 保护的临界区内往往有更多操作，也就意味着 `Mutex` 锁需要耗费更长的时间才能释放，也就是会需要耗费比上面这个 `40%` 还要多的时间另外一个协程才能获取到 `Mutex` 锁。

go 的 sync 包中的原子操作

在文章的开头，我们就说了，在 go 的 `sync.Map` 和 `sync.Pool` 中都有用到了原子操作，本节就来看一看这些操作。

sync.Map 中的原子操作

在 `sync.Map` 中使用到了一个 `entry` 结构体，这个结构体中大部分操作都是原子操作，我们可以看看它下面这两个方法的定义：

```
// 删除 entry
```

```
func (e *entry) delete() (value any, ok bool) {
for {
p := e.p.Load()
// 已经被删除了，不需要再删除
if p == nil || p == expunged {
return nil, false
}
// 删除成功
if e.p.CompareAndSwap(p, nil) {
return *p, true
}
}
}

// 如果条目尚未删除，trySwap 将交换一个值。
func (e *entry) trySwap(i *any) (*any, bool) {
for {
p := e.p.Load()
// 已经被删除了
if p == expunged {
return nil, false
}
// swap 成功
if e.p.CompareAndSwap(p, i) {
return p, true
}
}
}
```

我们可以看到一个非常典型的特征就是 `for` + `CompareAndSwap` 的组合，这个组合在 `entry` 中出现了很多次。

> 如果我们也需要对变量做并发读写，也可以尝试一下这种 for + CompareAndSwap 的组合。

sync.WaitGroup 中的原子操作

在 `sync.WaitGroup` 中有一个类型为 `atomic.Uint64` 的 `state` 字段，这个变量是用来记录 `WaitGroup` 的状态的。在实际使用中，它的高 32 位用来记录 `WaitGroup` 的计数器，低 32 位用来记录 `WaitGroup` 的 `Waiter` 的数量，也就是等待条件变量满足的协程数量。

如果不使用一个变量来记录这两个值，那么我们就需要使用两个变量来记录，这样就会导致我们需要对两个变量做并发读写，在这种情况下，我们就需要使用互斥锁来保护这两个变量，这样就会导致性

能的下降。
而使用一个变量来记录这两个值，我们就可以使用原子操作来保护这个变量，这样就可以保证并发读写的安全性，同时也能得到更好的性能：

```
// WaitGroup 的 Add 函数：高 32 位加上 delta state := wg.state.Add(uint64(delta) << 32) // WaitGroup 的 Wait 函数：低 32 位加 1 // 等待者的数量加 1 wg.state.CompareAndSwap(state, state+1)
```

CAS 操作有失败必然有成功

当然这里是指指向同一行 `CAS` 代码的时候（也就是有竞争的时候），如果是指向不同行 `CAS` 代码的时候，那么就不一定了。比如下面这个例子，我们把前面计算 `sum` 的例子改一改，改成用 `CAS` 操作来完成：

```
func TestCas(t *testing.T) {
var sum int32 = 0
var wg sync.WaitGroup
wg.Add(1000)

for i := 0; i < 1000; i++ {
go func() {
defer wg.Done()
// 这一行是有可能会失败的
atomic.CompareAndSwapInt32(&sum, sum, sum+1)
}()
}

wg.Wait()
fmt.Println(sum) // 不是 1000
}
```

在这个例子中，我们把 `atomic.AddInt32(&sum, 1)` 改成了 `atomic.CompareAndSwapInt32(&sum, sum, sum+1)`，这样就会导致有可能会有多个 goroutine 同时执行到 `atomic.CompareAndSwapInt32(&sum, sum, sum+1)` 这一行代码，这样肯定会有不同的 goroutine 同时拿到一个相同的 `sum` 的旧值，那么在这种情况下，就会导致 `CAS` 操作失败。也就是说，将 `sum` 替换为 `sum + 1` 的操作可能会失败。
失败意味着什么呢？意味着另外一个协程序先把 `sum` 的值加 1 了，这个时候其实我们不应该在旧的 `sum` 上加 1 了，而是应该在最新的 `sum` 上加上 1，那我们应该怎么做呢？我们可以在 `CAS` 操作失败的时候，重新获取 `sum` 的值，然后再次尝试 `CAS` 操作，直到成功为止：

```
func TestCas(t *testing.T) {
var sum int32 = 0
```

```
var wg sync.WaitGroup
wg.Add(1000)

for i := 0; i < 1000; i++ {
go func() {
defer wg.Done()
// cas 失败的时候，重新获取 sum 的值进行计算。
// cas 成功则返回。
for {
if atomic.CompareAndSwapInt32(&sum, sum, sum+1) {
return
}
}
}()
}

wg.Wait()
fmt.Println(sum)
}
```

总结

原子操作是并发编程中非常重要的一个概念，它可以保证并发读写的安全性，同时也能得到更好的性能。

最后，总结一下本文讲到的内容：

- 原子操作是更加底层的操作，它保护的是单个变量，而互斥锁可以保护一个代码片段，它们的使用场景是不一样的。
- 原子操作需要通过 CPU 指令来实现，而互斥锁是在软件层面实现的。
- go 里面的原子操作有以下这些：
  - `Add`：原子增减
  - `CompareAndSwap`：原子比较并交换
  - `Load`：原子读取
  - `Store`：原子写入
  - `Swap`：原子交换
- go 里面所有类型都能使用原子操作，只是不同类型的原子操作使用的函数不太一样。
- `atomic.Value` 可以用来原子操作任意类型的变量。
- go 里面有些底层实现也使用了原子操作，比如：
  - `sync.WaitGroup`：使用原子操作来保证计数器和等待者数量的并发读写安全性。

    - `sync.Map`：`entry` 结构体中基本所有操作都有原子操作的身影。
- 原子操作有失败必然有成功（说的是同一行 `CAS` 操作），如果 `CAS` 操作失败了，那么我们可以重新获取旧值，然后再次尝试 `CAS` 操作，直到成功为止。

总的来说，原子操作本身其实没有太复杂的逻辑，我们理解了它的原理之后，就可以很容易的使用它了。