# Go语言 原子操作 atomic 包

原子操作即是进行过程中不能被中断的操作。也就是说, 针对某个值的原子操作在被进行的过程当中, CPU 绝不会再去进行其它的针对该值的操作。为了实现这样的严谨性, 原子操由 CPU 提供芯片级别的支持, 所以绝对有效, 即使在拥有多 CPU 核心, 或者多 CPU 的计算机系统中, 原子操作的保证也是不可撼动的。这使得原子操作可以完全地消除竞态条件, 并能够绝对地保证并发安全性, 它的执行速度要比其他的同步工具快得多, 通常会高出好几个数量级。

不过它的缺点也很明显, 正因为原子操作不能被中断, 所以它需要足够简单, 并且要求快速。你可以想象一下, 如果原子操作迟迟不能完成, 而它又不会被中断, 那么将会给计算机执行指令的效率带来多么大的影响, 所以操作系统层面只对针对二进制位或整数的原子操作提供了支持。

因此, 我们可以结合实际情况, 来判断是否可以将锁替换成原子操作。

代码中的加锁操作因为涉及内核态的上下文切换会比较耗时、代价比较高。针对基本数据类型我们还可以使用原子操作来保证并发安全，因为原子操作是Go语言提供的方法它在用户态就可以完成，因此性能比加锁操作更好。Go语言中原子操作由内置的标准库sync/atomic提供，用于同步访问整数和指针。

竞争条件是由于异步的访问共享资源，并试图同时读写该资源而导致的，使用互斥锁和通道的思路都是在线程获得到访问权后阻塞其他线程对共享内存的访问，而使用原子操作解决数据竞争问题则是利用了其不可被打断的特性。

atomic提供的原子操作能够确保任一时刻只有一个goroutine对变量进行操作，善用atomic能够避免程序中出现大量的锁操作。

- Go语言提供的原子操作都是非入侵式的
- 原子操作支持的类型包括`int32、int64、uint32、uint64、uintptr、unsafe.Pointer`。

atomic常见操作有：

- 增减 Add
- 载入读取 Load
- 比较并交换 CompareAndSwap
- 交换 Swap
- 存储写入 Store

## 增减操作

以Add为前缀的增减操作：

```
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)
```

示例：

```
func main() {
    var a int32 = 13
    addValue := atomic.AddInt32(&a, 1)
    fmt.Println("增加之后:", addValue)
    delValue := atomic.AddInt32(&a, -4)
    fmt.Println("减少之后:", delValue)
}
#结果
增加之后: 14
减少之后: 10
```

# 载入操作 Load(原子读取)

当我们要读取一个变量的时候，很有可能这个变量正在被写入，这个时候，我们就很有可能读取到写到一半的数据。 所以读取操作是需要一个原子行为的。

在atomic包中就是Load开头的函数群。

```
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)
```

示例:

```
package main

import (
    "fmt"
    "sync/atomic"
)

func main() {
    var a, b int32 = 13, 12
    fmt.Printf("atomic.LoadInt32(&a): %v\n", atomic.LoadInt32(&a))
    fmt.Printf("atomic.LoadInt32(&b): %v\n", atomic.LoadInt32(&b))
}
#结果
atomic.LoadInt32(&a): 13
atomic.LoadInt32(&b): 12
```

# 存储写入 Store (原子写入)

读取是有原子性的操作的，同样写入atomic包也提供了相关的操作包。

```
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)
```

示例：

```
package main

import (
    "fmt"
    "sync/atomic"
)

func main() {
    var c int32
    atomic.StoreInt32(&c, 18)
    fmt.Printf("atomic.LoadInt32(&c): %v\n", atomic.LoadInt32(&c))
}
#结果
atomic.LoadInt32(&c): 18
```

# 比较交换 CompareAndSwap(CAS)

Go中的Cas操作，是借用了CPU提供的原子性指令来实现。CAS操作修改共享变量时候不需要对共享变量加锁，而是通过类似乐观锁的方式进行检查，本质还是不断的占用CPU 资源换取加锁带来的开销（比如上下文切换开销）。

原子操作中的CAS(Compare And Swap),在`sync/atomic`包中，这类原子操作由名称以`CompareAndSwap`为前缀的若干个函数提供。

```
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)
```

`CompareAndSwap`函数会先判断参数addr指向的操作值与参数old的值是否相等，仅当此判断得到的结果是true之后，才会用参数new代表的新值替换掉原先的旧值，否则操作就会被忽略。

```
package main

import (
    "fmt"
    "sync/atomic"
)

func main() {
    var a, b int32 = 13, 13
    var c int32 = 9
    res := atomic.CompareAndSwapInt32(&a, b, c)
    fmt.Println("swapped:", res)
    fmt.Println("替换的值:", c)
    fmt.Println("替换之后a的值:", a)
}
#结果
swapped: true
替换的值: 9
替换之后a的值: 9
```

# 交换 Swap

上面的`CompareAndSwap`系列的函数需要比较后再进行交换，也有不需要进行比较就进行交换的原子操作。

```
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)
```

示例:

```
package main

import (
    "fmt"
    "sync/atomic"
)

func main() {
    var a, b int32 = 13, 12
    old := atomic.SwapInt32(&a, b)
    fmt.Println("old的值:", old)
    fmt.Println("替换之后a的值", a)
}
#结果
```

```
old的值: 13
替换之后a的值 12
```

# 原子操作与互斥锁的区别

首先atomic操作的优势是更轻量，比如CAS可以在不形成临界区和创建互斥量的情况下完成并发安全的值替换操作。这可以大大的减少同步对程序性能的损耗。

原子操作也有劣势。还是以CAS操作为例，使用CAS操作的做法趋于乐观，总是假设被操作值未曾被改变（即与旧值相等），并一旦确认这个假设的真实性就立即进行值替换，那么在被操作值被频繁变更的情况下，CAS操作并不那么容易成功。而使用互斥锁的做法则趋于悲观，我们总假设会有并发的操作要修改被操作的值，并使用锁将相关操作放入临界区中加以保护。

下面是几点区别：

- 互斥锁是一种数据结构，用来让一个线程执行程序的关键部分，完成互斥的多个操作
- 原子操作是无锁的，常常直接通过CPU指令直接实现
- 原子操作中的cas趋于乐观锁，CAS操作并不那么容易成功，需要判断，然后尝试处理
- 可以把互斥锁理解为悲观锁，共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程

`atomic`包提供了底层的原子性内存原语，这对于同步算法的实现很有用。这些函数一定要非常小心地使用，使用不当反而会增加系统资源的开销，对于应用层来说，最好使用通道或sync包中提供的功能来完成同步操作。

# atomic.Value

此类型的值相当于一个容器，可以被用来"原子地"存储（Store）和加载（Load）任意类型的值。当然这个类型也是原子性的。

有了`atomic.Value`这个类型，这样用户就可以在不依赖`Go`内部类型`unsafe.Pointer`的情况下使用到atomic提供的原子操作。

值提供了一个自动加载和一个一致的类型值的存储。Value 的零值从 Load 返回 nil 。一旦 Store 被调用，Value 不能被复制。

首次使用后不得复制 Value 。

```
type Value struct {
    v interface{}
}
```

里面主要是包含了两个方法

- `v.Store(c)` - 写操作，将原始的变量c存放到一个`atomic.Value`类型的v里。
- `c = v.Load()` - 读操作，从线程安全的v中读取上一步存放的内容。

## func (*Value) Load

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

Load 返回最近的存储设置的值。如果此值没有存储调用，则返回 nil 。

## func (*Value) Store

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

Store 将 Value 的值设置为 x 。对于给定值的所有对 Store的调用都必须使用相同具体类型的值。存储不一致的类型恐慌，就像 Store（nil） 一样。

# 总结

1、atomic中的操作是原子性的；

2、原子操作由底层硬件支持，而锁则由操作系统的调度器实现。锁应当用来保护一段逻辑，对于一个变量更新的保护，原子操作通常会更有效率，并且更能利用计算机多核的优势，如果要更新的是一个复合对象，则应当使用`atomic.Value`封装好的实现。

3、atomic中的代码，主要还是依赖汇编来来实现的原子操作。

# 原子操作实例：

之前的写法，使用锁实现协程的同步：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var i = 100
var lock sync.Mutex

func add() {
    lock.Lock()
    i++
    lock.Unlock()
}

func sub() {
    lock.Lock()
    i--
    lock.Unlock()
}

func main() {
    for i := 0; i < 100; i++ {
        go add()
        go sub()
    }

    time.Sleep(time.Second * 3)
    fmt.Printf("i: %v\n", i)
}
#结果
```

```
i: 100
```

使用原子操作:

```
package main

import (
    "fmt"
    "sync/atomic"
    "time"
)

var i int32 = 100

func add() {
    atomic.AddInt32(&i, 1)
}

func sub() {
    atomic.AddInt32(&i, -1)
}

func main() {
    for i := 0; i < 100; i++ {
        go add()
        go sub()
    }

    time.Sleep(time.Second * 3)
    fmt.Printf("i: %v\n", i)

}
#结果
i: 100
```