Go 内存模型

版本：2012年3月6日 || 译者：Oling Cat，Ants Arks，特别感谢 Fall Ark 的帮助

引言

Go内存模型阐明了一个Go程对某变量的写入，如何才能确保被另一个读取该变量的Go程监测到。

忠告

程序在修改被多个Go程同时访问的数据时必须序列化该访问。

要序列化访问，需通过信道操作，或其它像 sync 和 sync/atomic 包中的同步原语来保护数据。

若您的程序行为必须通过阅读本文档才能理解，那...想必您一定十分聪明咯？

别自作聪明。

事件的发生次序

在单个Go程中，读取和写入的表现必须与程序指定的执行顺序相一致。换言之， 仅在不会改变语言规范对Go程行为的定义时，编译器和处理器才会对读取和写入的执行重新排序。由于存在重新排序，一个Go程监测到的执行顺序可能与另一个Go程监到的不同。例如，若一个Go程执行 a = 1; b = 2;，另一个Go程可能监测到 b 的值先于 a 更新。

为了详细论述读取和写入的必要条件，我们定义了事件发生顺序，它表示Go程序中内存操作执行的 偏序关系。 若事件 e_1 发生在 e_2 之前， 那么我们就说 e_2 发生在 e_1 之后。 换言之，若 e_1 既未发生在 e_2 之前， 又未发生在 e_2 之后，那么我们就说 e_1 与 e_2 是并发的。

在单一Go程中，事件发生的顺序即为程序所表达的顺序。

若以下条件均成立，则对变量 v 的读取操作 *r* 就允许对 v 的写入操作 *w* 进行监测：

1. *r* 不发生在 *w* 之前。
2. 在 *w* 之后 *r* 之前，不存在其它对 v 进行的写入操作 *w'*。

为确保对变量 v 的读取操作 *r* 能够监测到特定的对 v 进行写入的操作 *w*，需确保 *w* 是唯一允许被 *r* 监测的写入操作。也就是说，若以下条件均成立，则 *r* 能保证监测到 *w*：

1. *w* 发生在 *r* 之前。
2. 对共享变量 v 的其它任何写入操作都只能发生在 *w* 之前或 *r* 之后。

这对条件的要求""比第一对更强，它需要确保没有其它写入操作与 *w* 或 *r* 并发。

在单个Go程中并不存在并发，因此这两条定义是等价的：读取操作 *r* 可监测最近的写入操作 *w* 对 v 写入的值。当多个Go程访问共享变量 v 时，它们必须通过同步事件来建立发生顺序的条件，以此确保读取操作能监测到预期的写入。

以变量 v 所属类型的零值来对 v 进行初始化，其表现如同在内存模型中进行的写入操作。

对大于单个机器字的值进行读取和写入，其表现如同以不确定的顺序对多个机器字大小的值进行操作。

译注**(Ants Arks)**：
a 不在 b 之前，并不意味着 a 就在 b 之后，它们可以并发。这样的话，第一种说法， 即对于两个并发的Go程来说，一个Go程能否读到另一个Go程写入的数据，可能有，也可能没有。 第二种说法，由于 r 发生在 w 之后，r 之前并没有其它的 w'，也没有 w" 和 r 并列，因此 r 读到的值必然是 w 写入的值。下面结合图形进行说明，其中 r 为 read，w 为 write，它们都对值进行操作.

单Go程的情形：
```
-- w0 ---- r1 -- w1 ---- w2 ----  r2 ---- r3 ------>
```

这里不仅是个偏序关系，还是一个良序关系：所有 r/w 的先后顺序都是可比较的。

双Go程的情形：
```
-- w0 -- r1 -- r2 ---- w3 ----  w4 ---- r5 -------->
-- w1 ----- w2 -- r3 ----  r4 ---- w5 -------->
```

单Go程上的事件都有先后顺序；而对于两条Go程，情况又有所不同。即便在时间上 r1 先于 w2 发生，但由于每条Go程的执行时长都像皮筋一样伸缩不定，因此二者在逻辑上并无先后次序。换言之，即二者并发。
对于并发的 r/w，r3 读取的结果可能是前面的 w2，也可能是上面的 w3，甚至 w4 的值；
而 r5 读取的结果，可能是 w4 的值，也能是 w1、w2、w5 的值，但不可能是 w3 的值。

双Go程交叉同步的情形：
```
-- r0 -- r1 ---|------ r2 ------------|-- w5 ------>
-- w1 --- w2 --|-- r3 ---- r4 -- w4 --|------->
```

现在上面添加了两个同步点，即 | 处。这样的话，r3 就是后于 r1 ，先于 w5 发生的。
r2 之前的写入为 w2，但与其并发的有 w4，因此 r2 的值是不确定的：可以是 w2，也可以是 w4。
而 r4 之前的写入的是 w2，与它并发的并没有写入，因此 r4 读取的值为 w2。

到这里，Go程间的关系就很清楚了。若不加同步控制，那么所有的Go程都是“平行”并发的。换句话说， 若不进行同步，那么 main 函数以外的Go程都是无意义的，因为这样可以认为 main 跟它们没有关系。 只有加上同步控制，例如锁或信道，Go程间才有了相同的“节点”，在它们的两边也就有了执行的先后顺序， 不过两个“节点”之间的部分，同样还是可以自由伸缩，没有先后顺序的。如此推广，多条Go程的同步就成了有向的网。

同步

初始化

程序的初始化运行在单个Go程中，但该Go程可能会创建其它并发运行的Go程。

若包 p 导入了包 q，则 q 的 init 函数会在 p 的任何函数启动前完成。

函数 main.main 会在所有的 init 函数结束后启动。

Go程的创建

go 语句会在当前Go程开始执行前启动新的Go程。

例如，在此程序中，

```
var a string

func f() {
        print(a)
}

func hello() {
        a = "hello, world"
        go f()
}
```

调用 hello 或许会在将来的某一时刻打印 "hello, world" （在 hello 返回之后则会打印零值）。

Go程的销毁

Go程无法确保在程序中的任何事件发生之前退出。例如，在此程序中：

```
var a string

func hello() {
        go func() { a = "hello" }()
        print(a)
}
```

对 a 进行赋值后并没有任何同步事件，因此它无法保证被其它任何Go程检测到。 实际上，一个积极的编译器可能会删除整条 go 语句。

若一个Go程的作用必须被另一个Go程监测到，需使用锁或信道通信之类的同步机制来建立顺序关系。

信道通信

信道通信是在Go程之间进行同步的主要方法。在特定信道上的每一次发送操作都有与其对应的接收操作相匹配， 这通常发生在不同的信道上。

信道上的发送操作总在对应的接收操作完成前发生。

此程序：

```
var c = make(chan int, 10)
var a string

func f() {
        a = "hello, world"
        c <- 0
}

func main() {
        go f()
        <-c
        print(a)
}
```

可保证打印出 "hello, world"。该程序首先对 a 进行写入， 然后在 c 上发送信号，随后从 c 接收对应的信号，最后执行 print 函数。

若在信道关闭后从中接收数据，接收者就会收到该信道返回的零值。

在上一个例子中，用 close(c) 代替 c <- 0 仍能保证该程序产生相同的行为。

从无缓冲信道进行的接收，要发生在对该信道进行的发送完成之前。

此程序（与上面的相同，但交换了发送和接收语句的位置，且使用无缓冲信道）：

```
var c = make(chan int)
var a string

func f() {
        a = "hello, world"
        <-c
}


func main() {
        go f()
        c <- 0
        print(a)
}
```

也可保证打印出 "hello, world"。该程序首先对 a 进行写入， 然后从 c 中接收信号，随后向

c 发送对应的信号，最后执行 print 函数。

若该信道为带缓冲的（例如，c = make(chan int, 1)），则该程序将无法保证打印出 "hello, world"。（它可能会打印出空字符串，崩溃，或做些别的事情。）

TODO: 优化语句 在某信道上进行的的第 ***k*** 次容量为 ***C*** 的发送发生在第 ***k+C*** 次从该信道进行的接收完成之前。

The *k*th receive on a channel with capacity *C* happens before the *k*+*C*th send from that channel completes.

This rule generalizes the previous rule to buffered channels. It allows a counting semaphore to be modeled by a buffered channel: the number of items in the channel corresponds to the number of active uses, the capacity of the channel corresponds to the maximum number of simultaneous uses, sending an item acquires the semaphore, and receiving an item releases the semaphore. This is a common idiom for limiting concurrency.

This program starts a goroutine for every entry in the work list, but the goroutines coordinate using the limit channel to ensure that at most three are running work functions at a time.

```
var limit = make(chan int, 3)

func main() {
        for _, w := range work {
                go func(w func()) {
                        limit <- 1
                        w()
                        <-limit
                }(w)
        }
        select{}
}
```

锁

sync 包实现了两种锁的数据类型：sync.Mutex 和 sync.RWMutex。

对于任何 sync.Mutex 或 sync.RWMutex 类型的变量 l 以及 ***n < m***，对 l.Unlock() 的第 ***n*** 次调用在对 l.Lock() 的第 ***m*** 次调用返回前发生。

此程序：

```
var l sync.Mutex
var a string

func f() {
        a = "hello, world"
        l.Unlock()
}

func main() {
        l.Lock()
        go f()
```

```
        l.Lock()
        print(a)
}
```

可保证打印出 "hello, world"。该程序首先（在 f 中）对 l.Unlock() 进行第一次调用，然后（在 main 中）对 l.Lock() 进行第二次调用，最后执行 print 函数。

对于任何 sync.RWMutex 类型的变量 l 对 l.RLock 的调用，存在一个这样的 ***n***，使得 l.RLock 在对 l.Unlock 的第 ***n*** 次调用之后发生（返回），且与其相匹配的 l.RUnlock 在对 l.Lock的第 ***n+1*** 次调用之前发生。

Once 类型

sync 包通过 Once 类型为存在多个Go程的初始化提供了安全的机制。 多个线程可为特定的 f 执行 once.Do(f)，但只有一个会运行 f()，而其它调用会一直阻塞，直到 f() 返回。

通过 once.Do(f) 对 f() 的单次调用在对任何其它的 once.Do(f) 调用返回之前发生（返回）。

在此程序中：

```
var a string
var once sync.Once

func setup() {
        a = "hello, world"
}

func doprint() {
        once.Do(setup)
        print(a)
}

func twoprint() {
        go doprint()
        go doprint()
}
```

调用 twoprint 会打印两次 "hello, world"。 第一次对 twoprint 的调用会运行一次 setup。

错误的同步

请注意，读取操作 *r* 可能监测到与其并发的写入操作 *w* 写入的值。即便如此，也并不意味着发生在 *r* 之后的读取操作会监测到发生在 *w* 之前的写入操作。

在此程序中：

```
var a, b int

func f() {
        a = 1
        b = 2
```

```
}

func g() {
        print(b)
        print(a)
}

func main() {
        go f()
        g()
}
```

可能会发生 g 打印出 2 之后再打印出 0。

这个事实会使很多习惯变得无效。

双重检测锁是种避免同步开销的尝试。例如，twoprint 程序可能会错误地写成：

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}

func doprint() {
        if !done {
                once.Do(setup)
        }
        print(a)
}

func twoprint() {
        go doprint()
        go doprint()
}
```

但这里并不保证在 doprint 中对 done 的写入进行监测蕴含对 a 的写入进行监测。这个版本可能会（错误地）打印出一个空字符串而非 "hello, world"。

另一种错误的习惯就是忙于等待一个值，就像这样：

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}

func main() {
        go setup()
        for !done {
        }
```

```
        print(a)
}
```

和前面一样，这里不保证在 main 中对 done 的写入的监测， 蕴含对 a 的写入也进行监测，因此该程序也可能会打印出一个空字符串。 更糟的是，由于在两个线程之间没有同步事件，因此无法保证对 done 的写入总能被 main 监测到。main 中的循环不保证一定能结束。

这个主题有种微妙的变体，例如此程序：

```
type T struct {
        msg string
}

var g *T

func setup() {
        t := new(T)
        t.msg = "hello, world"
        g = t
}

func main() {
        go setup()
        for g == nil {
        }
        print(g.msg)
}
```

即便 main 能够监测到 g != nil 并退出循环， 它也无法保证能监测到 g.msg 的初始化值。

这里所有例子的解决方案都是相同的：使用显式的同步。