【译】Go 语言内存模型：2022-06-06 版

The Go Memory Model: Version of June 6, 2022

孔俊

2022-10-20

Contents

1 译者序

原文 *The Go Memory Model* 描述 Go 语言内存模型，这里的内存模型实际上是“内存一致性模型”（memory consistency model）。笔者修改了格式错误并翻译全文。

The Go Memory Model 在内存一致性模型层面，描述 Go 语言实现（Go implementation）提供的一致性保证，并进一步指出该保证对 Go 语言实现的限制；在工程实践层面，描述使用同步原语确保 goroutine 间可见性的方法，并进一步指出常见的错误同步手法。

Go 语言内存模型在语言层面，保证无数据竞争程序 (data-race-free program) 符合顺序一致模型（sequential consistency model），通过 channel 和`sync`包中的同步原语（synchronizing primitive) 提供内存一致性保证。

处理器层面上，不同处理器的不同的内存一致性模型，例如 x86 的 TSO(Total Store Order)。语言层面的内存一致模型是建立在处理器之上的抽象，使程序员可以忽略处理器的差异，依赖语言提供的内存一致性模型编程。Go 语言为无数据竞争程序提供顺序一致模型的保证，程序员只要确保自己编写的程序没有数据竞争，就可以认为“不论是什么处理器，程序执行都是顺序一致的”。

处理器顺序一致模型可以参考 lamport 的著名论文 *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs* 和我的博客 【译】如何设计正确运行多进程程序的多核计算机。

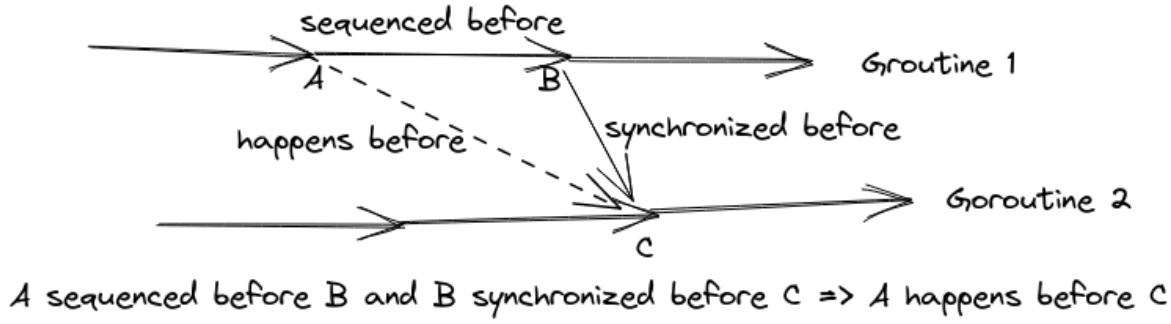
注意，要区分同步线程执行次序的“同步原语”和保证内存操作可见性的“同步”。前者的目的是限制线程的执行次序，比如使用条件变量让线程 A 等待线程 B 完成某操作；后者的目的是保证内存位置上内存操作的可见性，比如线程 A 可以观察到线程 B 对内存位置 *x* 的写，但不保证线程 A 和 B 的执行次序。

一般来说，编程语言提供的同步原语（互斥锁、读写锁等）实现通过处理器提供的内存屏障（memory barrier）指令等机制保证了内存操作的可见性，即编程语言提供的同步原语附带内存一致性保证。Go 程序员使用`sync`包提供的互斥锁等同步原语和语言内置的 channel 等保证内存可见性。C++ 内存模型不仅提供了符合程序员视角的顺序一致模型，还提供了更宽松的内存一致性模型以便聪明程序员用脑力换算力。原文建议：别太聪明，直接使用`sync`包提供的同步原语！

2 TL;DR

- 无数据竞争程序的执行符合顺序一致模型。
- goroutine 内的内存操作满足 sequenced before 关系。
- goroutine 间的同步操作满足 synchronized before 关系。
- happens before 和 sequenced before 关系是偏序。
- happens before 关系定义为 synchronized before 和 sequenced before 的传递闭包。

- 读操作 *r* 读取到的值 *W(r)* 一定是最近一次 happens before *r* 的写操作 *w* 的值。
- 不满足 happens before 关系，即使读到了写 *w* 的值，也不能保证能读到更早的写 *w*’ 的值。
- 读小于或等于一个字的数据一定能读到其他 goroutine 最后写入的值。
- goroutine 的退出不同步于任何事件。
- 读、原子读、获取 mutex、channel 接收是 read-like 操作。
- 写、原子写、释放 mutex、channel 发送和 channel 关闭是 write-like 操作。
- compare-and-swap 同时是 read-like 和 write-like 的。



happens-before-relationship

3 简介

Go 语言内存模型指定何种条件下保证一个 goroutine 对某变量的读操作可以观察到其他 goroutine 写此变量的值。

3.1 建议

修改多个 goroutine 同时访问的数据的程序必须序列化（serialize）这些访问。

为了序列化访问，使用 channel 操作或其他同步原语，例如`sync`和`sync/atomic`包中的那些。

如果你必须阅读此文档来理解你程序的行为，你就聪明过头了。

别太聪明。

4 非正式概述

Go 以与其他语言几乎相同的方式处理其内存模型，旨在保持语义简单、可理解和有用。本节给出了给出一般概述，对于大多数程序员来说应该足够了。下一节将更正式地描述内存模型。

数据竞争（*data race*）定义为对一个内存地址的写操作和其他对该内存地址的读写操作并发进行，除非所有涉及的访问都是`sync`/`atomic`包提供的原子数据访问。就像之前提到的，强烈建议程序员使用合适的同步手段避免数据竞争。没有数据竞争时，Go 程序看起来就像所有 goroutine 被复用在一个多处理器上。这一性质有时称为 DRF-SC：无数据竞争的程序顺序一致执行。

虽然程序员应当编写无数据竞争的程序，但仍然有对 Go 实现可以如何响应数据竞争的限制。实现可能总是报告数据竞争并终止程序。否则，单字大小或小于单字大小的内存读取必须观察到实际写入到该位置的（可能由一个并发执行的 goroutine）、仍未被覆盖的值。这些实现约束让 Go 更像 Java 或 JavaScript，这些语言中大部分竞争不会有严重后果；更不像 C 和 C++，这些语言中存在数据竞争的程序完全是未定义的，编译器可以做任何事。Go 的方式旨在让错误的程序更加可靠和更易于调试的同时，仍然坚持数据竞争是错误，工具应该诊断并报告它们。

5 内存模型

以下 Go 语言内存模型的正式定义紧跟 Hans-J. Boehm 和 Sarita V. Adve 在 PLDI 2008 发表的 *Foundations of the C++ Concurrency Memory Model* 中的定义。无数据竞争程序及对无数据竞争程序的顺序一致模型的定义和这篇论文中的相同。

内存模型描述程序执行的要求，程序执行由 goroutine 执行组成，goroutine 执行由内存操作组成。

一个内存操作（*memory operation*）建模为以下四点：

- 内存操作的类型，指明它是普通的数据读取，普通的数据写入，或者同步操作（*synchronizing operation*）例如原子数据访问、互斥锁操作或 channel 操作；
- 内存操作在程序中的位置；
- 内存位置或访问的变量，和
- 内存操作读或写的值。

一些内存操作是 *read-like* 的，包括读、原子读、获取 mutex、channel 接收。其他内存操作是 *write-like* 的，包括写、原子写、释放 mutex、channel 发送和 channel 关闭。另一些，例如原子的 compare-and-swap 同时是 read-like 和 write-like 的。

goroutine 执行（goroutine execution）建模为单个 goroutine 执行的内存操作集合。

要求 **1**: 给定从内存读取或写入的值，每个 goroutine 内的内存操作必须对应该 goroutine 的一次正确的顺序执行。执行必须和 *sequenced before* 关系一致。sequenced before 关系是Go 语言规范为控制流结构和语句的执行次序定义的一个偏序。

Go 程序执行（*program execution*）建模为 goroutine 执行的集合，映射 M 指定 read-like 操作读取的 write-like 操作。（同一程序的多个执行可以有不同的程序执行）。

> 译者注
>
> 同一程序的多个执行可以有不同的程序执行（Multiple executions of the same program can have different program executions.）的实例是多 goroutine 程序，Go scheduler 每次执行调度 goroutine 的结果未必相同，第一次可能 goroutine 1 先运行，第二次执行可能 goroutine 1 后运行。

要求 **2**：对于一个给定的程序执行，映射 M 当仅限于同步操作时，必须可以通过某种和序列（sequencing）一致的同步操作的隐式全序和这些操作读取、写入的值解释。

synchronized before 关系是源自 W 的同步内存操作上的偏序。如果一个同步 read-like 内存操作 *r* 观察（observes）到一个同步 write-like 内存操作 *w*（也就是说，如果 *W(r) = w*），那么 *w synchronized before r*。非正式地，synchronized before 关系是前面提到的隐含的全序的一个子集，仅限于 *W* 直接观察到的信息。

> 译者注
>
> 这里的意思应该是，当 M 是一个同步操作时，程序执行的结果可以通过一个符合顺序一致模型的执行次序解释。

要求 **3**: 对一个平凡的（非同步的）在内存位置 *x* 上的数据读取 *r*，*W(r)* 必须是对 *r* 可见的（*visible*）写 *w*，这里的可见意味着同时满足以下两点：

1. *w* happens before *r*。
2. *w* 不 happen before 任何其他 happens before r 的向 *x* 的写 *w'*。

> 译者注
>
> 这个 *W(r)* 读到的是最近一次 happens before *r* 的 *w* 的结果。

内存位置 *x* 上的读-写数据竞争（*read-write data race*）包含一个 *x* 上的 read-like 内存操作 *r* 和一个 *x* 上的 write-like 内存操作 *w*，其中至少一个是非同步（的 non-synchronizing），这样的读写会乱序 happens before（也就是说，既不 *r* happens before *w*，也不 *w* happens before *r*）。

内存位置 *x* 上的写-写数据竞争（*write-write data race*）包含两个 *x* 上的 write-like 内存操作 *w* 和 *w'*，其中至少一个是非同步的，这样的写写操作会乱序 happens before。

注意，如果内存位置 *x* 上没有读-写或写-写数据竞争，那么所有 *x* 上的读 *r* 只有一个可能的 *W(r)*：这个唯一的、按 happens before 次序立刻先于它的 *w*。

更一般地说，可以证明任何无数据竞争的 Go 程序（这意味着它没有具有读写数据竞争的程序执行）的执行结果，只能由一些顺序一致的、交错的 goroutine 执行解释。（证明与上面引用的 Boehm 和 Adve 论文的第 7 节相同。）这个性质称为 DRF-SC。

正式定义的目的是匹配其他语言（包括 C、C++、Java、JavaScript、Rust 和 Swift）为无竞争程序提供的 DRF-SC 保证。

某些 Go 语言操作（例如 goroutine 创建和内存分配）充当同步操作。这些操作对 synchronized-before 偏序的的影响记录在下面的“同步“一节中。各个包负责为其已的操作提供类似的文档。

6 对包含数据竞争的程序的实现限制

前面的章节给了无数据的竞争程序执行的定义。这一节非正式地描述实现必须为存在竞争的程序提供的语义。

首先，任何实现都可以检测数据竞争，报告竞争和停止程序的执行。实现使用 ThreadSanitizer（使用“`go build -race`”访问）做这件事。

否则，一个不大于一个机器字的内存位置 *x* 上的读 *r* 必须观察到一些写 *w*，使得 *r* 不 happens before *w*，且没有一个写 *w*’使得 *w* happens before *w*’且 *w*’ happens before *r*。

> 译者注
>
> 上面的讲法，简单地说就是：小于一个机器字的 *W(r)* 读到的是最近一次 happens before *r* 的 *w* 的结果。

此外，不允许观察到非因果的和凭空产生的写。

鼓励大于一个机器字的内存位置上的读观察到单个允许的写 *w*，但不要求满足机器字大小内存位置上的语义。出于性能原因，实现可以视更大的操作为一系列不限定次序的独立的机器字大小的操作。这意味着多机器字大小数据结构上的竞争可以导致和单一写不一致的值。当值依赖于内部的 (pointer, length) 或 (pointer, type) 对的一致性时，就像大部分 Go 实现中的 interface、map、slice 和 string 一样，这些竞争会反过来导致任意的内存损坏（memory corruption）。

错误的同步的例子在下面的”错误的同步“一节。

实现的限制的例子在下面的”错误的编译“一节。

7 同步

7.1 初始化

程序初始化运行在单独的 goroutine 中，但是这个线程可以创建其他并发运行的 goroutine。

如果一个包 *p* 导入包 *q*，*q* 的 `init`函数的完成 *happens before* 任何 *p* 的函数的开始。

所有`init`函数的完成 *synchronized before* 函数`main.main`的开始。

7.2 创建 **goroutine**

启动一个新协程的*go*语句 *synchronized before* 该协程执行的开始。

例如，在这个程序中：

```
var a string

func f() {
    print(a)
}

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

调用hello将在未来的某个点（可能在hello返回后）打印"hello, world"。

> 译者注
>
> a = "hello, world" sequenced before go f()
>
> go f() synchronized before f()

7.3 销毁 **goroutine**

goroutine 的退出不 synchronized before 程序的任何事件。比如，在这个程序：

```
var a string

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

没有任何同步事件跟随在对a赋值后，所以不保证它可以被其他任何 goroutine 观察到。事实上，激进的编译器可以删除这整个go语句。

> 译者注
>
> Go 内存模型规定”goroutine 的退出不 synchronized before 程序的任何事件“，这里创建的 gouroutine 的任何操作不被任何其他 goroutine 观察到是合法的。因此编译器可以认为这个go语句不产生任何影响，是多余的，直接删掉。

如果一个 goroutine 的副作用必须被另一个 goroutine 观察到，请使用锁或 channel 通信之类的同步机制建立一个相对次序（relative ordering）。

7.4 channel 通信

channel 通信是 goroutine 间通信的主要同步方法。一个 channel 上的每个发送都匹配该 channel 上对应的接收（通常在不同 goroutine 上）。

给定 *channel* 上的发送操作 *synchronized before* 该 *channel* 上接收操作的完成。

这个程序：

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

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

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

保证打印`"hello, world"`。对`a`的写 sequenced before `c`上的发送，`c`上的发送 synchronized before `c`上对应的接收完成，接收操作的完成 sequenced before 调用`print`。

给定 *channel* 的关闭 *synchronized before* 因 *channel* 关闭而返回空值的接收。

在之前的例子中，用`close(c)`替代`c <- 0`得到确保同样行为的程序。

给定 *unbuffered channel* 上的接收 *synchronized before* 对应 *channel* 上发送操作的完成。

这个程序（和上面的一样，但是交换了发送和接收语句，并使用 unbufferd channel）：

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

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

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

也保证打印`"hello, world"`。对`a`的写 sequenced before `c`上的接收，`c`上的接收 synchronized before `c`上对应发送的完成，`c`上对应发送的完成 sequenced before `print`。

如果 channel 是有缓冲的（例如`c = make(chan int, 1)`），那就不能保证这个程序打印`"hello, world"`。（它可能打印空字符串，崩溃或做别的什么事。）

> 译者注
>
> `c`上的第一次接收 synchronized before 其上第 1+1=2 次发送，因此程序中`<- c`和`c <- 0`不构成 synchronized before 关系，不能保证`print(a)`观察到`a = "hello, world"`。

容量为 *C* 的 *channel* 上的第 *k* 次接收 *synchronized before* 其上第 *k+C* 次发送的完成。

这个规则泛化了前面有关 bufferd channel 的规则。它允许使用 bufferd channel 建模计数信号量：channel 中的元素数量对应活跃使用的数量，channel 的容量对应最大并发使用量，发送数据相当于获取信号量，接收数据相当于释放信号量。这是限制并发的常用手法。

这个程序为工作列表的每一项启动一个 goroutine，但是 goroutine 使用`limit` channel 协调，确保任意时刻最多只有 3 个运行的工作。

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

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

7.5 锁

`sync`包实现了`sync.Mutex`和`sync.RWMutex`两种锁数据类型。

对于任意*`sync.Mutex`*或*`sync.RWMutex`*变量*`l`*和 *n < m*，第 *n* 次调用*`l.Unlock()`* *synchronized before* 第 *m* 次调用*`l.Lock()`*返回。

> 译者注
>
> 注意！这里是 n < m，意味着第 N 次 unlock 和第 N 次 lock 不构造 synchronized before 关系。下面的程序就是例子。

这个程序：

```
var l sync.Mutex
var a string

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

func main() {
    l.Lock()
    go f()
    l.Lock()
    print(a)
}
```

保证打印"print, world"。第一次调用l.Unlock()（在f中）synchronized before 第二次调用l.Lock()（在main()中）的返回，它 sequenced before print。

对于任何*sync.RWMutex*变量*l*上的*l.RLock*调用，都存在一个 *n*，使得第 *n* 次调用*l.Unlock* *synchronized before* *l.RLock*的返回，并且匹配的*l.RUnlock*调用 *synchronized before* 第 *n+1* 次*l.Lock*调用返回。

对*l.TryLock*（或*l.TryRLock*）的成功调用等价于调用*l.Lock*（或*l.RLock*）。不成功的调用完全没有同步效果。就内存模型而言，可以认为l.TryLock（或l.TryRlock）即使在互斥锁l被解锁时也能够返回 false。

> 译者注
>
> 原文没有详细解释“就内存模型而言，可以认为l.TryLock（或l.TryRlock）即使在互斥锁l被解锁时也能够返回 false。”
>
> 我认为这样讲是因为，l.Lock（或l.RLock）执行成功也未必会有同步效果（见上面 unlock 和 lock 的规则），如果l.Lock（或l.RLock) 执行成功但没有同步效果，等价于对应的l.TryLock（或 (l.TryRLock) 没有同步效果，相当于l.TryLock（或l.TryRLock）调用失败的情形。

7.6 Once

sync包通过Once类型提供了一种多 goroutine 情形下安全的初始化机制。多个线程可以为特定f执行once.Do(f)，但只有一个会运行f()，其他的调用阻塞直到f()返回。

> 译者注
>
> 这里和下文的线程都指 goroutine。

once.Do(f) 中单个*f()* 调用的完成 *synchronized before* 任何*once.Do(f)* 调用。

在这个程序中：

```
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只会调用setup一次。setup函数将在调用print前完成。结果是"hello, world"被打印两次。

7.7 原子值

sync/atomic包中的 API 统称为“原子操作”，可用于同步不同 goroutine 的执行。如果原子操作 *A* 的副作用被原子操作 *B* 观察到，那么 *A* synchronized before *B*。程序中执行的所有原子操作看起来就像以某种顺序一致次序执行。

前面的定义和 C++ 顺序一致的原子类型以及 Java **volatile**遍历有相同语义。

> 译者注
>
> 这里的同步不是指锁等同步原语协调不同线程的执行次序，而是指可见性，不能混为一谈。
>
> 锁等同步原语已经实现了这种可见性上的同步，详见上面的同步规则。

7.8 Finalizer

runtime包提供一个SetFinalizer函数，添加一个当特定对象不可达后被调用的 finalizer。调用SetFinalizer(x, f) synchronized before 调用f(x)。

> 译者注
>
> Finalizer 即对象不可从程序访问后调用的回调函数，对象不可从程序访问意味着该对象该被垃圾回收器回收了。

7.9 其他机制

sync包提供了额外的同步抽象，包括 condition variables, lock-free maps, allocation pools 和 wait groups。它们的文档指定了它们对同步所做的保证。

其他提供了同步抽象的包应该也记录了它们所做的保证。

8 错误的同步

带数据竞争的程序是错误的，可以表现出非顺序一致的执行。特别要注意读 *r* 可以观察到任何和 *r* 并发的写 *w* 写入的值。即使发生这种情况，也不意味发生在 *r* 之后的读可以观察到发生在 *w* 之前的写。

> 译者注
>
> 满足 happens before 关系才可以保证 *r* 观察到 happens before 关系上最近的 *w*。

在这个程序中：

```
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先打印2后打印0的情况。

这一事实让一些常见手法失效。

双重检查锁（double-checked locking）是一种避免同步开销的尝试。例如，twoprint 程序可能被错误地写为：

```
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()
}
```

但是这里不能保证在doprint中观察到写入done隐含观察到写入a。这个版本（错误地）打印一个空字符串而非"hello, world"。

另一个错误手法是忙等待（busy waiting）一个值，例如：

```
var a string
var done bool

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

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

和之前一样，无法保证在main中观察到写入done隐含观察到写入a，所以这个程序也可以打印一个空白字符串。更糟糕的是，无法保证写入done被main观察到，因为两个线程间没有同步事件。不保证main中的loop可以结束。

关于这个主题还有一个更难以察觉的变形，例如这个程序：

```
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)
}
```

即使main观察到g != nil并且退出循环，也不能保证它会观察到g.msg的初始值。

在所有这些例子中，解决办法是相同的：使用显式的同步机制。

9 错误的编译

Go 内存模型对编译器优化的限制和对 Go 程序的限制一样多。一些单线程中靠谱的优化放到所有 Go 程序中是不可靠的。尤其是编译器不得引入原程序中不存在的写操作，不得让单个读操作观察到多个值，以及不得允许单个写操作写多个值。

以下所有例子假设*p和*q指向可被多个 goroutine 访问的内存位置。

不得给无竞争程序引入数据竞争意味着不得将可能出现在条件语句中的写操作移到条件语句之外。例如，编译器不得反转此程序中的条件：

```
*p = 1
if cond {
    *p = 2
}
```

也就是说，编译器不能将程序改写成这个：

```
*p = 2
if !cond {
    *p = 1
}
```

如果cond为 false 并且另一 goroutine 正在读*p，那么在原始程序中，其他 goroutine 只能观察所有*p的旧值和1。在重写的程序中，其他 goroutine 可以观察到2，这在原始程序中是不可能的。

不引入数据竞争也意味着不能假设循环会终止。例如，编译器通常不得将对*p或*q的访问移动到此程序中的循环之前：

```
n := 0
for e := list; e != nil; e = e.next {
    n++
}
i := *p
*q = 1
```

如果list指向一个循环链表，那么原始程序将不会访问*p和*q但是重写的程序会。（如果编译器可以证明*p不会 panic，那么前移*p是安全的；前移*q还需要编译器证明没有其他 goroutine 可以访问*q。）

不引入数据竞争也意味着不得假设被调用的函数总能返回或和同步操作无关。例如，编译器不得将此程序中访问*p和*q移动到函数调用之前（至少在不直接了解 f的准确行为的情况下）：

```
f()
i := *p
*q = 1
```

如果调用永不返回，那么原始程序将永远不会访问*p和*q，但重写的程序会。如果调用包含同步操作，那么原始程序可以在访问*p和 *q之前建立 happens before 关系，但重写的程序不会。

不允许单个写操作写入多个值也意味着，在写入局部变量之前不得使用该局部变量作为临时存储。例如，编译器不得在此程序中使用*p作为临时存储：

```
*p = i + *p/2
```

也就是说，编译器不能将程序改写成这个：

```
*p /= 2
*p += i
```

如果i和*p一开始等于 2，原始代码执行*p = 3，所以竞争的线程只能从*p读取到 2 或 3。重写的代码先执行*p = 1后执行p =3，让竞争线程可能读取到 1。

注意，所有这些优化在 C/C++ 编译器中都是允许的：和 C/C++ 编译器共用后端的 Go 编译器必须小心地禁用对 Go 无效的优化。

注意，禁止引入数据竞争不适用于编译器可以证明在目标平台上竞争不影响正确执行的情形。例如，在几乎所有 CPU 上，都可以重写

```
n := 0
for i := 0; i < m; i++ {
    n += *shared
}
```

为：

```
n := 0
local := *shared
for i := 0; i < m; i++ {
    n += local
}
```

前提是可以证明`*shared`在访问时不会出错，因为潜在添加的读取不会影响任何现有的并发读取或写入。另一方面，这种重写在源到源的翻译器中是无效的。

> 译者注
>
> 禁止引入数据竞争是为了确保多线程程序的正确执行（程序执行的副作用和原始程序的相同），如果引入了数据竞争但不影响多线程程序的执行，那么提高性能却引入数据竞争的编译器优化是合理的。
>
> 在这个改写中，程序在 for 语句之前读取一次`*shared`，循环中不再读取。从 Go 语言内存模型的视角看，原始程序中对`*shared`的访问没有任何同步操作，不保证循环中能观察到其他 goroutine 的修改。因此循环中`*shared`只观察到 for 语句开始时的值是合法的。
>
> 从处理器内存一致性模型的角度看，绝大多数现代处理器（例如 AMD64、ARM）都允许 store-load reroder。因此不保证循环中能观察到其他线程对`*shared`的修改。
>
> 在这些允许 store-load reorder 的处理器上，编译器的改写是合理的。但在不允许 store-load reorder 的处理器上，循环中一定能观察到其他线程对`*shared`的修改，改写不等价，因此原文说“基本上在所有 CPU 上，都可以重写”、“这种重写在源到源的翻译器中是无效的”。

10 结论

编写无数据竞争程序的 Go 程序员可以依赖这些程序的顺序一致执行，就像在几乎所有其他现代编程语言中一样。

当涉及到带有竞争的程序时，程序员和编译器都应该记住这个建议：别太聪明。