2022
11
1
06:49 PM

对 Go 内存模型的一些理解

分类：Go　标签：Go

Russ Cox 曾经写过一个《Memory Models》系列，今年 6 月 Go 官网也更新了他写的 《The Go Memory Model》。这两篇都是非常值得一读的文章，网上也有不少翻译。因为这些文章的信息量较大，我还是尝试解读一下吧。

首先说下硬件内存模型。

来看看这个例子（本文所有例子都假设所有变量的初始值是 0）。

```
// Thread 1            // Thread 2
x = 1                  y = 1
r1 = y                 r2 = x
```

最终是否可能得到 `r1 == 0` 且 `r2 == 0`?

从代码本身来看，`r1 = y` 是在 `x = 1` 之后执行的，而 `r2 = x` 又是在 `y = 1` 之后执行的，那么 `r1` 和 `r2` 应该至少有一个是 1 才对。
然而我们对代码本身的观察，只是基于顺序一致性（sequential consistency）模型，实际的机器模型为了更快地执行程序，并不会采用这种对程序员而言最直观的模型。例如在 x86 机器上，线程 1 可能读取到 `y == 0`，从而设置 `r1 = 0`，此时对两个变量的写入只是放入了这个处理器的写队列里，并不一定刷新到了内存中，因此线程 2 仍然可能读到 `x == 0`，从而设置 `r2 = 0`。

那么，x86 的这种写队列有什么特点呢?
最大的特点就是每个处理器都有一个写队列，且写入是有顺序的：假如线程 1 先后执行了 `x = 1` 和 `y = 1`，则线程 2 如果观测到了 `y == 1`，那么肯定也能观测到 `x == 1`。这个模型也被称为 x86 Total Store Order (x86-TSO)。

如何解决因为写队列，导致线程无法及时观测到其他线程的写入呢?
方法是引入内存屏障（memory barrier），这样线程在开始读取前，会先刷新其写入队列到内存：

```
// Thread 1            // Thread 2
x = 1                  y = 1
barrier                barrier
r1 = y                 r2 = x
```

加上屏障之后，假设线程 1 执行了 `x = 1` 之后，刷新写队列到内存，然后读取 `y` 时得到 0，那么线程 2 一定还没执行 `y = 1` 并刷新其写队列，最终 `y` 不可能是 0，也就解决了上述的问题。

再来看这个例子：

```
// Thread 1            // Thread 2
x = 1                  r1 = y
y = 1                  r2 = x
```

最终是否可能得到 `r1 == 1` 且 `r2 == 0`？
以 x86-TSO 模型来看，若要得到 `r1 == 1`，那么线程 2 一定观察到了 `y == 1`，则更早之前执行的 `x = 1` 也是能观察到的，因此 `r2` 不可能是 0。
可是在 ARM 架构上，这是可能的。ARM 架构的每个处理器都直接从自己的完整内存副本中读取和写入，而传播到其他处理器时，允许重新排序。也就是说，虽然线程 1 先后执行了 `x = 1` 和 `y = 1`，但线程 2 可能观测到 `y == 1` 且 `x == 1`。

但即便 ARM 架构允许传播到其他处理器时，进行重新排序，这些其他的处理器观察到的顺序仍然是一样的，例如：

```
// Thread 1    // Thread 2    // Thread 3    // Thread 4
x = 1          y = 1          r1 = x         r3 = y
                              r2 = y         r4 = x
```

是否可能观测到 `r1 == 1`、`r2 == 2`、`r3 == 2` 且 `r4 == 1`？也就是说线程 3 先后观察到 `x == 1` 和 `x == 2`，而线程 4 相反。
答案是不会，ARM 会保证线程 3 和 4 看到的顺序是一样的。

Adve 和 Hill 还提出了一种 data-race-free (DRF) 模型，它假设硬件有普通内存读写操作和同步内存读写操作，普通内存读写可以在同步操作之间进行重排序，但不允许跨越它们（即同步操作也可作为内存屏障）。
如果一个程序的不同线程对同一内存地址要么都是读，要么用同步操作串行执行，那么这个程序被称为无数据竞争的。
他们认为，执行无数据竞争的程序，要使其像是按照顺序一致的顺序一样。这种保证无数据竞争程序的顺序一致性的观点，则被缩写为 DRF-SC。x86 和 ARM 都是能满足 DRF-SC 的。

接着说说编程语言的内存模型。

除了硬件之外，编译器也可能对生成的指令进行重排，甚至比 ARM 的内存模型更宽松，但它们也需要满足 DRF-SC。

那么各种编程语言各有怎样的模型呢?

Java 早期（1996 年）的内存模型尝试用 `volatile` 关键字来同步，但它仅要求读写需要访问内存，并不能禁止重排，因此是非常弱的内存模型。

新的 Java 内存模型（2004 年）引入了同步操作，可以建立 happens-before 关

系。除 DRF-SC 以外，Java 还定义了有数据竞争时的程序行为：对于 word (32 bit) 或更小的变量 x，读取 x 必须观察到某次对它的写入；如果读取 r 观察到对 x 的写入 w，那么 r 不发生在 w 之前。

C++ (2011 年)的内存模型没有给有数据竞争时的程序任何保证，这些程序是未定义行为（C++ 本身就有很多未定义行为，因此不差这个）。C、Rust 和 Swift 也采用了相同的内存模型。

JavaScript 的内存模型（2017 年）与 C++ 有 3 点不同：

1. 仅采用顺序一致性的原子。
2. 定义了竞争访问的结果，允许竞争读返回任何值，甚至是不相关的数据，这可能导致私有数据的泄露。
3. 定义了当原子和非原子操作在同一个内存地址使用时，以及当使用不同大小访问同一个内存地址时会发生什么。

然后说下 Go 的内存模型。

Go 的内存模型很像 Java / JavaScript，它也定义了竞争访问的结果：Go 的实现允许在数据竞争时报告并终结程序（例如并发访问 map 可能会 panic）；读取 word (32 bit) 或更小的内存地址，必须观察到对这个地址的实际写入，且还未被覆写。
Go 使用了一个基于 happens-before 对读写竞争的定义。最主要的有 2 点：

1. 对于 `sync.Mutex` 或 `sync.RWMutex` 变量 `l` 和 `n < m`，第 n 次调用 `l.Unlock()` 发生于第 m 次调用 `l.Lock()` 的返回前。（也就是如果未被 `Unlock()` 的话，后续的 `Lock()` 是不会返回的。）
2. 如果原子（atomic）操作 A 的效果被原子操作 B 观测到了，那么 A 发生于 B 之前。（也就是 `atomic.Load()` 观测到了 `atomic.Store()` 的结果，那么 `atomic.Store()` 发生于 `atomic.Load()` 之前。）

这里虽然看上去像废话，但我举一个 `sync.Once` 的例子就能知道为啥要这样定义了：

```
type Once struct {
    done uint32
    m    Mutex
}

func (o *Once) Do(f func()) {
    if atomic.LoadUint32(&o.done) == 0 {
        o.doSlow(f)
    }
}
```

```
func (o *Once) doSlow(f func()) {
    o.m.Lock()
    defer o.m.Unlock()
    if o.done == 0 {
        defer atomic.StoreUint32(&o.done, 1)
        f()
    }
}
```

有这样几个问题：

1. 为什么要用 `if atomic.LoadUint32(&o.done) == 0` 来判断，而不能直接用 `if o.done == 0`？
   因为不用原子操作或加锁的话，访问 `o.done` 可能获取到寄存器或缓存中的值，而不会观测到其他处理器对 `o.done` 的修改。
   不过以我的理解，如果对 `o.done` 的读未做同步，影响顶多是进入 `doSlow()` 中进行二次检查，然后这个处理器就会观测到最新的 `o.done`，以后不会再获取到旧值了。
   而且在实际的测试中，x86 的机器是可以立刻（小于 100 纳秒）观测到其他处理器对 `o.done` 的修改。
2. 为什么 `doSlow()` 中要用 `Mutex` 来加锁?
   首先，如果不加锁，初始化时如果有多个线程同时进入 `doSlow()`，则 `f()` 可能会被调用多次。其次，初始化后根本不会再进入 `doSlow()`，所以在这里面加锁并不会影响后续调用的性能。
3. 为什么 `doSlow()` 中的 `if o.done == 0` 不用原子操作?
   因为第一个获取到锁的线程执行完 `atomic.StoreUint32(&o.done, 1)` 才释放了锁，这个行为 happens-before 后续获取到锁的线程，这些线程一定会观测到 `o.done == 1`。
4. 为什么 `doSlow()` 中的 `atomic.StoreUint32(&o.done, 1)` 需要用原子操作?
   因为外面的 `if atomic.LoadUint32(&o.done) == 0` 使用了原子操作，里面用 `atomic.StoreUint32(&o.done, 1)` 才能与之形成 happens-before 的关系，即保证 `doSlow()` 返回后，后续对 `atomic.LoadUint32(&o.done)` 的调用都会返回 1。而如果不用原子操作，外面的 `atomic.LoadUint32(&o.done) == 0`（或更激进的 `o.done == 0`）并不能与 `o.m.Lock()` 或 `o.m.Unlock()` 构成 happens-before 关系，也就没法保证外面能读取到被修改后的值。
   同样，因为第一条的原因，在 x86 上，外面是可以立刻观测到值的变化的。即使没有观测到，`doSlow()` 中还有二次检查，并不会导致 `f()` 被执行多次。但出于兼容性考虑，不使用原子操作时，可能会造成观测不到 `o.done` 的写入而一直进入 `doSlow()`，反而更加影响性能。而且这个原子操作只需要执行一次，并不

会影响后续调用的性能。

Go 内存模型还限制了一些编译器优化：

- 不允许引入源代码中不存在的写入。
- 不允许将无数据竞争的程序引入数据竞争：
  - 不能将写操作移出它所出现的条件语句。例如下列程序：

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

不能改写成：

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

  - 不假设循环终止。例如下列程序，不能把对 `*p` 或 `*q` 的访问提到循环之前：

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

  - 不假设被调用的函数总是返回或者没有同步操作。例如下列程序，不能把对 `*p` 或 `*q` 的访问提到函数调用之前：

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

- 不允许单次读操作观测到多个值。这意味着不从共享内存中重新加载局部变量，例如：

```
i := *p
if i < 0 || i >= len(funcs) {
    panic("invalid function index")
}
... complex code ...
// 此处编译器不能重新加载 i = *p
```

```
funcs[i]()
```

- 不允许单次写操作写入多个值。这意味着不能将变量将要被写入内存作为临时存储，例如下列程序：

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

  不能被改写成：

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

以上的优化是 C/C++ 编译器所允许的，这是 Go 与它们不同的地方。

最后，LeoYang90 还写了一篇《Go 内存一致性模型》，这个则偏向于实现细节，且是中文的，我就不解读了。这里面解释了为啥硬件或编译器要对内存模型做优化，宁愿舍弃顺序一致性：相对于寄存器或 L1 缓存而言，内存的访问延迟要慢 100 倍，所以尽量避免内存访问成了不得已的选择。