GO语言内存模型

姓名 王康

时间 2022.06.17

背景

处理器从单核到多核的变化，带来的思考：

- 硬件和编译器对于程序执行的优化是否还有效？

  在单线程中编译器按照最优的逻辑对普通的内存读写做重新排序，这可能会影响对线程对于变量个观察结果。

  https://goplay.tools/snippet/X1itMxRGNZ5

- 程序自身是否依然是有效的？

  多线程同时读写同一块内存时，可能出现出乎意料的结果。

  https://goplay.tools/snippet/9iylVyipaWR

背景

DRF-SC 无数据竞争的顺序一致性

- 顺序一致性（ sequential consistency）：多处理器系统的任何执行结果都是相同的，就好像所有处理器的操作都是按某种顺序执行的，每个处理器的操作都是按程序指定的顺序出现的。
- 无数据竞争（Data Race Free）：当对一个线程对某块内存执行写操作时候，不存在其他线程同时对这块内存执行写或者读的操作。

非正式结论：如果软件避免了数据竞争，那么硬件就好像是顺序一致的。

背景

GO 语言并发编程的便利性以及频繁性

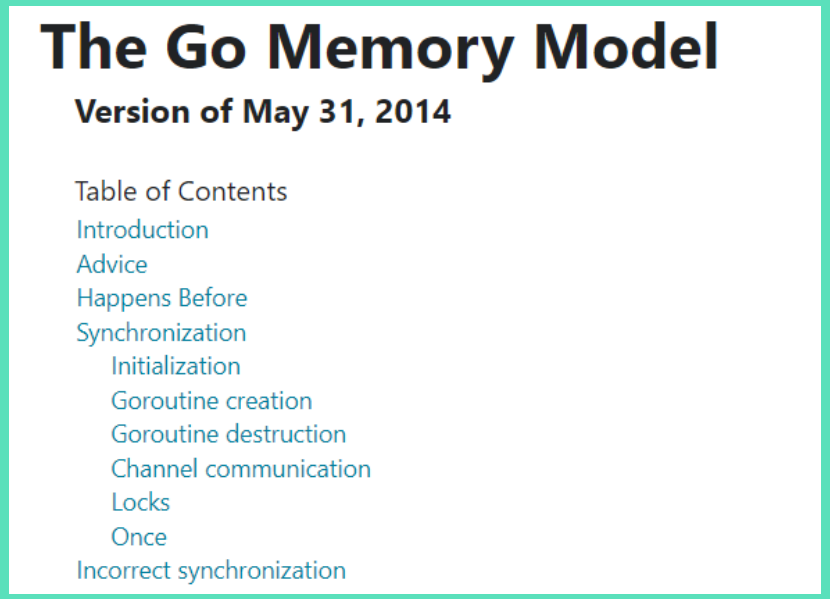
Workload	Goroutines/Threads		Ave. Execution Time	
	client	server	client-Go	server-Go
g_sync_ping_pong	7.33	2.67	63.65%	76.97%
sync_ping_pong	7.33	4	63.23%	76.57%
qps_unconstrained	201.46	6.36	91.05%	92.73%

Table 3. Dynamic information when executing RPC benchmarks. *The ratio of goroutine number divided by thread number and the average goroutine execution time normalized by the whole application's execution time.*

Tu T, Liu X, Song L, et al. Understanding real-world concurrency bugs in go[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 865-878.

背景

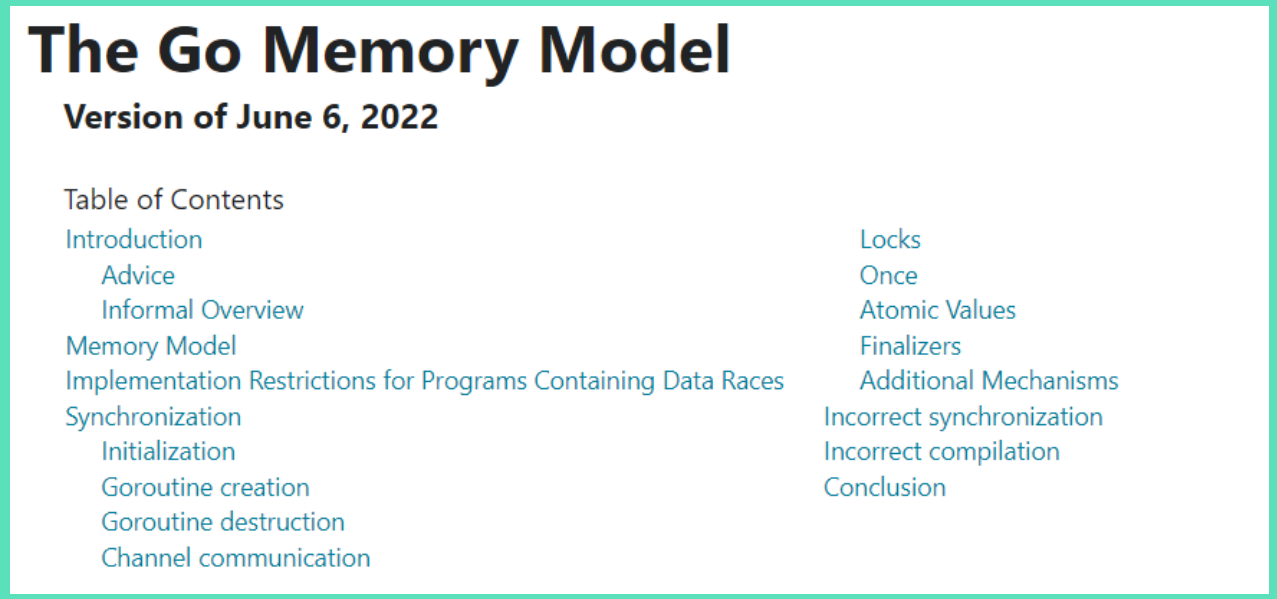
GO 语言内存模型更新



The Go Memory Model

Version of May 31, 2014

Table of Contents



The Go Memory Model

Version of June 6, 2022

Table of Contents

GO语言内存模型

- 编程语言的内存模型描述了线程通过内存的交互，以及对数据的共享使用。
- GO语言内存模型：
  - 定义了当一个 goroutine 读取一个变量的时候，怎样可以确保观察到其他 goroutine 中对同一变量的写入所产生的值。
  - 建议：
    - 修改一个可以同时被多个 goroutine 访问的变量时需要串行化访问；
    - 使用 channel 或者其他同步原语（sync和sync/atomic）来实现串行化；
    - 如果你必须阅读本文的其余部分才能理解你的程序的行为，那你太聪明了；
    - 别自作聪明。

    “synchronize to eliminate data races, and then programs will behave as if sequentially consistent, leaving no need to understand the remainder of the memory model.” —— rsc

GO语言内存模型

- 正式的定义：

  The memory model describes the requirements on program executions, which are made up of goroutine executions, which in turn are made up of memory operations.

  - 内存操作的四个要素：
    - 类型：普通内存操作以及同步内存操作
    - 位置：在程序中申明的位置
    - 内存：操作的内存
    - 值：操作读取自内存或写入到内存的值

GO语言内存模型

- The requirements：

1. 对于单个 Goroutine 来说，组成他的所有内存操作必须满足 *sequenced before* 的关系。（语言规范中定义程序语句执行的顺序的子集）
   - y[f()], ok = g(h(), i()+x[j()], <-c), k()：执行顺序是 f(), h(), i(), j(), <-c, g(), k()
   - a, f := 1, func() int { a++; return a } \n x := []int{a, f()}： x 可能是 [1, 2] or [2, 2]
2. 对于整个程序的来说，所有同步内存操作满足 *synchronized before* 的关系。（一个同步读操作 r 观察到了一个同步写操作 w 的结果，那么 w synchronized before r）

happened before 是 sequenced before 和 synchronized before 并集的传递闭包。

- 父子关系不是传递的：A 是 B 的父亲，B 是 C 的父亲，但是 A 不是 C 的父亲。
- 祖先关系是传递的： A 是 B 的祖先，B 是 C 的祖先，同时 A 是 C 的祖先。

GO语言内存模型

- The requirements：

  3. 在程序执行中，对于内存位置 x 任何一个普通的读操作 r 来说，存在一个对应的写操作 w 对其可见。
     - w happens before r；
     - 对 x 的任何其他写入都发生在 w 之前或 r 之后。

满足上述要求的 Go 程序被认为是 Data Race Free 的。（由 DRF—SC，所以……）

GO语言内存模型

有数据竞争的程序的语义：

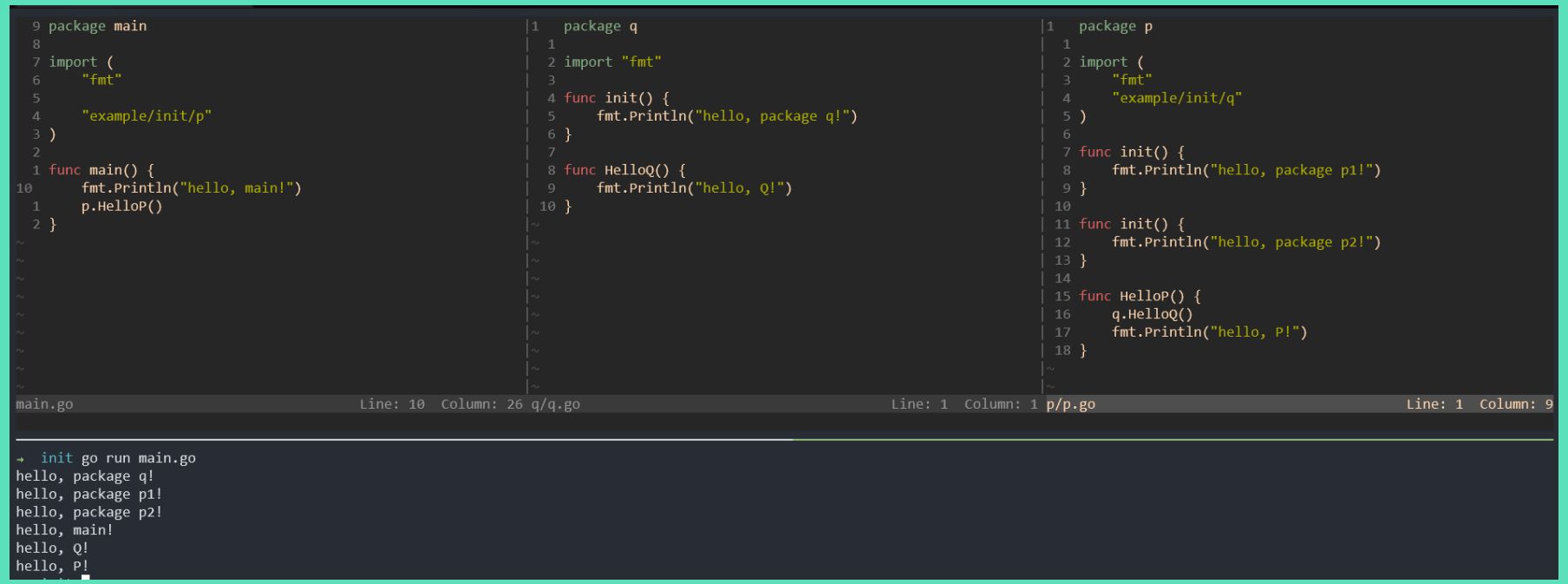
- 其他语言：
  - Java：做了定义。对于CPU字长大小或更小的变量，对变量(或字段)x的读取必须看到对x的某一次写入所存储的值。如果读取r观察到对x的写入w，那么r不发生在w之前。这意味着r可以观察发生在r之前的所有写入，并且它可以观察与r竞争的写入。
  - C++：没有给有竞争的程序任何保证。不定义竞争语义允许实现检测和诊断竞争并停止执行。
- GO语言：

  Go的方法介于这两者之间。提供 “go build –race” 来检测竞争；对于CPU字长大小或更小的变量和Java一样；多CPU字长数据结构上的竞争(multiword竞态)可能导致单次写入产生不一致值，也可能会导致内存损坏。

GO语言同步原语

Init 函数

- 如果package p引入了package q, 那么q的init函数的执行完成一定happen-before p的所有init函数(之前)
- main.main 函数一定 happen after 所有的init函数完成(之后)



GO语言同步原语

Goroutine 的创建和销毁

- Go 语句创建一个 goroutine 一定happen before goroutine执行(之前)。
- goroutine 的退出不能保证在程序中的任何事件之前同步。

```
var a string

func f() {
        print(a)
}

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

```
var a string

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

GO语言同步原语

channel

- 一个 channel 上的发送是在该 channel 的相应接收完成之前发生的。
- channel 的关闭发生在接收之前，因为通道被关闭而返回一个零值。
- 一个无缓冲 channel 的接收发生在该 channel 的发送完成之前。
- 一个容量为 C 的 channel 上，第 k 次接收发生在该 channel 的第 k+C 次发送完成之前。

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

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

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

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

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

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

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

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

GO语言同步原语

Lock

- 对于任意的 sync.Mutex 或者 sync.RWMutex 类型的变量 l 以及 n < m, 调用第n次 l.UnLock() 一定 happen before 第 m 次的 l.Lock() 返回 (之前)

Once

- once.Do(f)中的对f的单次调用一定happen before 任意次的对once.Do(f)调用返回(之前)

GO语言同步原语

原子操作

- 如果原子操作 B 观察到原子操作 A 的效果，则 A happen before B 。

  https://goplay.tools/snippet/Woylx8bDwhz

Finalizers

- 对 SetFinalizer(x, f) 的调用 happen before finalization 调用 f(x) 。

Additional Mechanisms

- sync 提供了其他的同步机制，包括: condition variables, lock-free maps, allocation pools, 和 wait groups.

参考资料

- The Go Memory Model - The Go Programming Language(2014)
  - The Go Memory Model - The Go Programming Language (2022)
- research!rsc: Memory Models (swtch.com)
  - [译]硬件内存模型 (colobu.com)
  - [译]编程语言内存模型 (colobu.com)
  - [译]更新Go内存模型 (colobu.com)
- Memory Order Guarantees in Go -Go 101
- Understanding Real-World Concurrency Bugs in Go (songlh.github.io)
- A Study of Real-World Data Races in Golang (arxiv.org)
- Order of evaluation - The Go Programming Language (golang.org)
- Go 1.19 Release Notes - The Go Programming Language (golang.org)