如何消除程序中的数据竞争

周光远 华为

从一些问题说起

1

```go
func main() {
    s := 0
    var wg sync.WaitGroup
    wg.Add(2)
    for j:=0;j<2;j++{
        go func() {
            for i := 0; i < 1000000; i++ {
                s++
            }
            wg.Done()
        }()
    }
    wg.Wait()
    log.Println(s)
}
```

```
2021/09/10 16:27:34 1005594
```

2

```
fatal error: concurrent map writes

goroutine 5 [running]:
runtime.throw({0x463362, 0x0})
        runtime/panic.go:1198 +0x71 fp=0xc000040740 sp=0xc000040710 pc=0x42c1f1
runtime.mapassign_faststr(0x0, 0x0, {0x4617c5, 0x2})
        runtime/map_faststr.go:211 +0x39c fp=0xc0000407a8 sp=0xc000040740 pc=0x40dbfc
main.main.func1()
        ./map.go:7 +0x3c fp=0xc0000407e0 sp=0xc0000407a8 pc=0x455e3c
runtime.goexit()
        runtime/asm_amd64.s:1581 +0x1 fp=0xc0000407e8 sp=0xc0000407e0 pc=0x452e41
created by main.main
        ./map.go:5 +0x65

goroutine 1 [runnable]:
main.main()
        ./map.go:11 +0x85
exit status 2
```

3

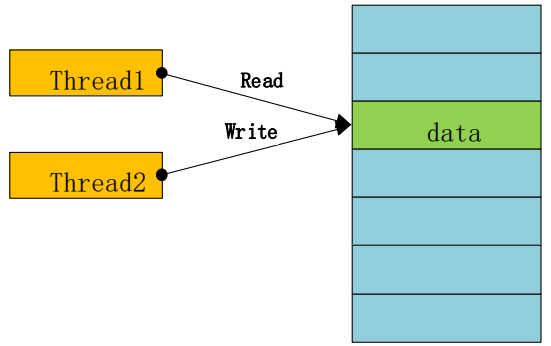
```go
var a string
func main(){
    go modify("Gopher meetup")
    go modify("Hello")
    for {
      log.Println(a)
    }
}

func modify(s string)  {
    for {
        a = s
    }
}
```

```
2021/09/10 16:18:38 Gopher meetup
2021/09/10 16:18:38 Gophe
2021/09/10 16:18:38 Gophe
2021/09/10 16:18:38 Gopher meetup
2021/09/10 16:18:38 Gopher meetup
2021/09/10 16:18:38 Hello
2021/09/10 16:18:38 HelloKhmerLat
2021/09/10 16:18:38 HelloKhmerLat
2021/09/10 16:18:38 Gophe
2021/09/10 16:18:38 Gopher meetup
2021/09/10 16:18:38 Hello
2021/09/10 16:18:38 HelloKhmerLat
2021/09/10 16:18:38 Gophe
2021/09/10 16:18:38 Hello
```

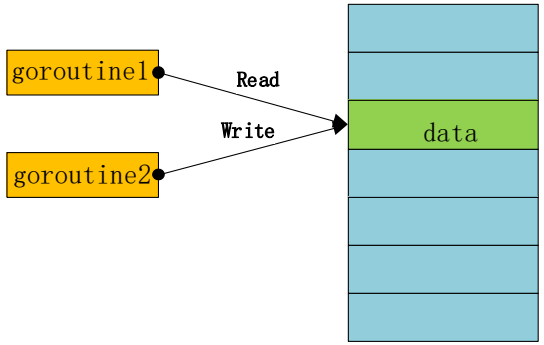
数据竞争

什么是数据竞争

数据竞争（data race）：在程序中，多线程（至少两个线程）并发访问同一个内存地址，且至少其中一次访问是写操作。



Go语言中的数据竞争（data race）： data race occurs when two **goroutines** access the same variable concurrently and at least one of the accesses is a write.



从微观看数据竞争

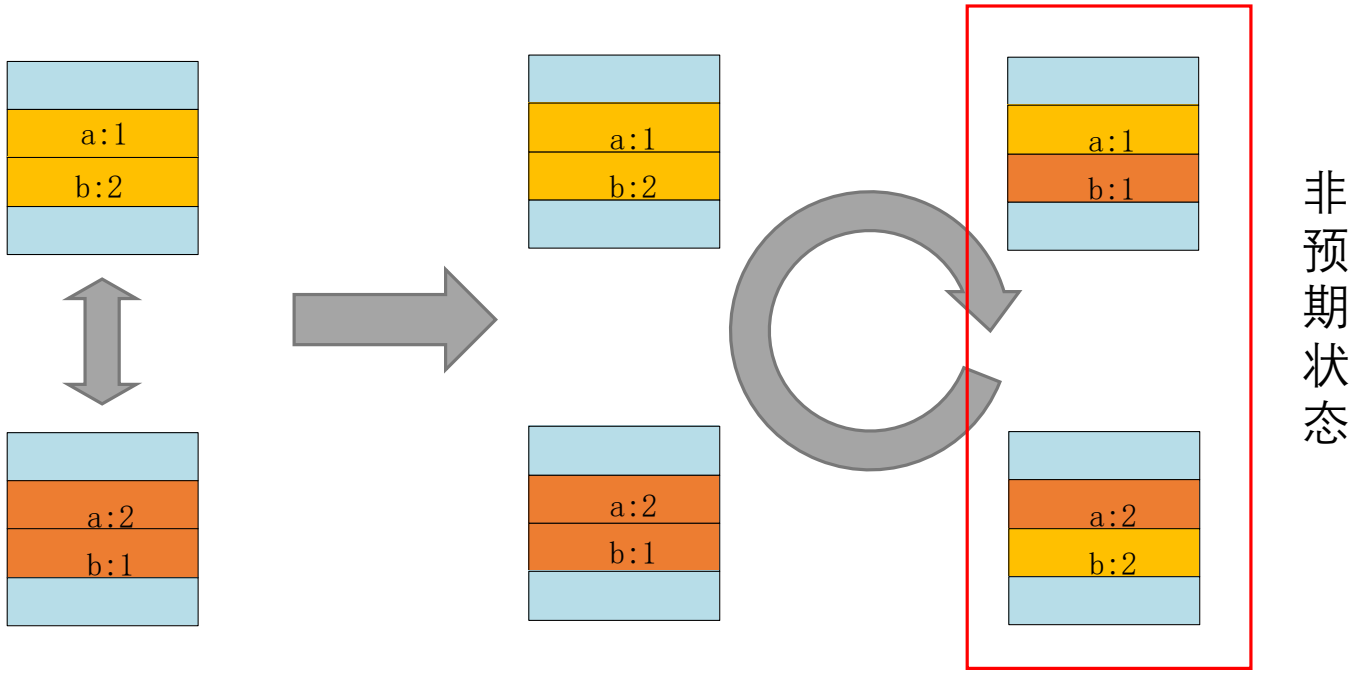
时间上：多个并发的读写操作被观察到的顺序无法预知。

空间上：并发读写时观察到非预期的数据。

```
type data struct {
    a uint64
    b uint64
}

var d = data{a: 1, b: 2}

go func() {
    for {
        d = data{a: 1, b: 2}
    }
}()

go func() {
    for {
        d = data{a: 2, b: 1}
    }
}()
```

goroutine 1

goroutine 2

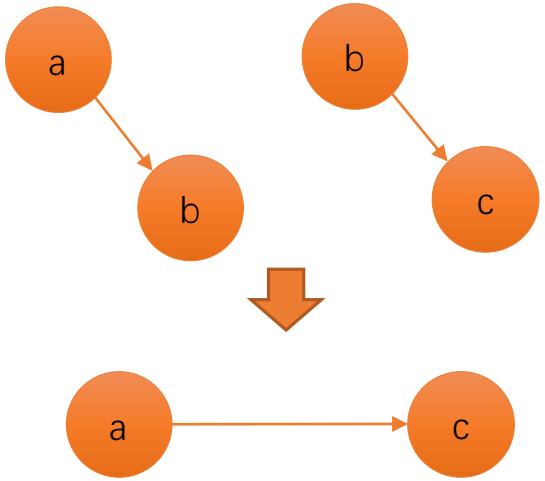


Happens Before

In computer science, the happened-before is a relation between the result of two events, such that if one event should happen before another event, the result must reflect that, even if those events are in reality executed out of order.

Happens Before关系是一种保证，如果a事件Happens Before b事件，那么b事件一定能观察到a事件的结果；

- 观察意味着b事件对a事件的结果存在依赖；
- Happens Before关系不代表代码真实执行的时间；
- 真实执行的时间不影响Happens Before关系；



传递性：
对于任意的事件a,b,c，如果a → b, 同时，b → c，则有： a → c.
（后续用→ 代表happens before）

Go语言的Happens Before

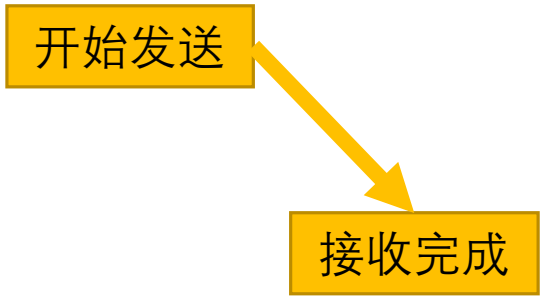
Go 中的 happens before 有以下保证（https://golang.org/ref/mem）：

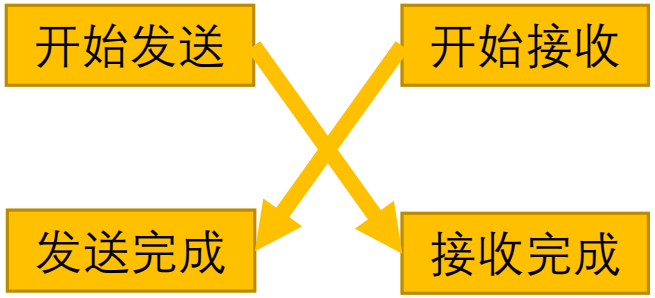
```
1   a := 1
2   b := a
3   c := b
4   d := c
5   print(d)
```

goroutine：

a := 1 → b := a → c := b → d := c → print(d)

channel：





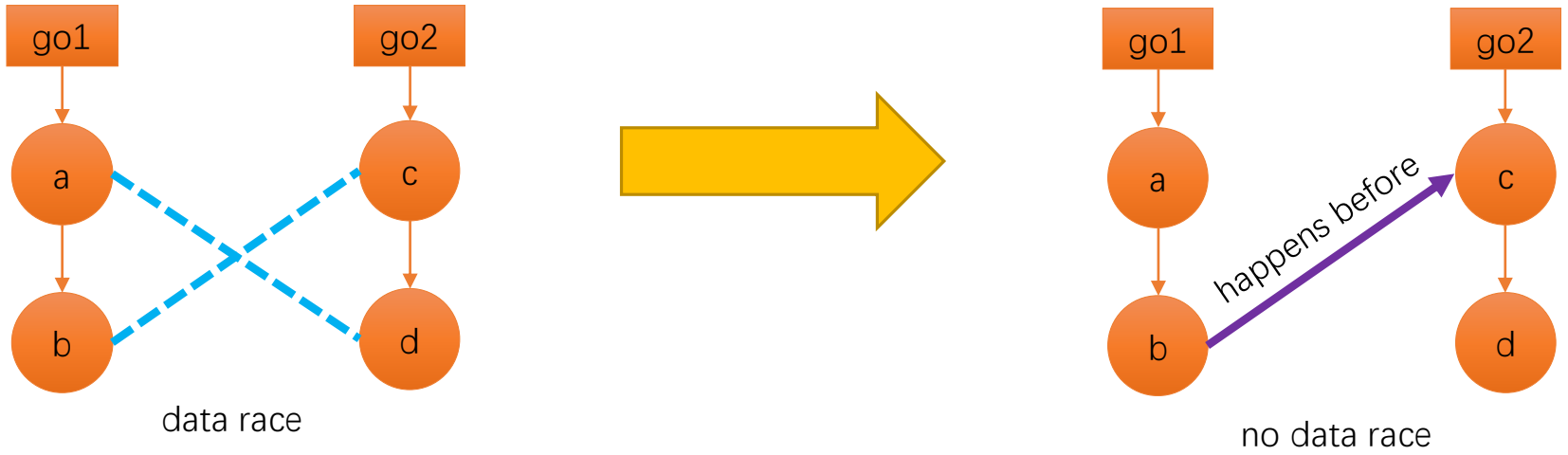
无缓冲通道

- 所有通道：开始发送 → 接收完成（同一个数据）；
- 对于无缓冲channel：开始接收 → 发送完成（同一个数据）；

其他的对于init函数，锁，协程，原子操作，sync包里的功能，还有许多保证，更详细可以看：
https://golang.org/ref/mem
https://go101.org/article/memory-model.html

消除数据竞争的原理

消除数据竞争，实质就是利用go提供的保证和传递性来建立事件之间的happens before关系。



实战分析

```
1     a := make(map[int]int)
2
3     go func() {
4         a[1] = 1 1
5     }()
6
7     a[2] = 2 2
```

1处的代码和2处的代码没有建立任何happens before关系，所以存在数据竞争。

```
1     a := make(map[int]int)
2     c := make(chan struct{}, 1)
3
4     go func() {
5         a[1] = 1          1
6         c <- struct{}{}   2
7     }()
8
9     <-c               3
10    a[2] = 2          4
```

利用通道的happens before保证:建立了2 → 3 。

再结合传递性可以得到：1 → 2 → 3 → 4.
所以消除了数据竞争

怎么检测数据竞争?

检测数据竞争

go语言提供了一个数据竞争的检测功能，除过检测一些简单的数据竞争场景，还结合go语言自身的内存模型，提供了更丰富和精确的检测报告。

```
$ go test -race mypkg
$ go run -race mysrc.go
$ go build -race mycmd
$ go install -race mypkg
```

```
4   var a int64
5   go func() {
6       a = 1
7   }()
8   a = 2
```

```
==================
WARNING: DATA RACE
Write at 0x00c000098008 by goroutine 6:
  main.main.func1()
      app.go:6 +0x30

Previous write at 0x00c000098008 by main goroutine:
  main.main()
      app.go:8 +0xba

Goroutine 6 (running) created at:
  main.main()
      app.go:5 +0xb0
==================
Found 1 data race(s)
```

race检测数据竞争的原理

通过-race这个tag控制raceenabled的值为true，运行时调用race相关的函数记录内存访问的时间和方式，同时检测这些访问之间是否有happens before保证，race实际基于ThreadSanitizer。

```
4 func mapaccess1_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer {
5     if raceenabled && h != nil {
6         callerpc := getcallerpc()
7         racereadpc(unsafe.Pointer(h), callerpc, funcPC(mapaccess1_fast32))
8     }
9     ..................
10 }
```

1. **race是运行时检测，非静态代码检查。**
2. **只有真正执行到的代码才能被检查出来数据竞争。**
3. **代码只要执行到，无论本次是否同时访问，都可以检测出是否有竞争。**
4. **严重影响性能，可能会占用数倍的cpu和内存。**

除过基础的用法，还支持使用GORACE环境变量控制race检测的一些行为，如输出报告的路径，检测到竞争是否退出等。详细见：https://golang.org/doc/articles/race_detector

参考：https://go.dev/blog/race-detector，https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm

race检测数据竞争

```
3 func main() {
4     c := make(chan struct{})
5     go func() { c <- struct{}{} }()
6     close(c)
7 }
```

```
11 func main() {
12     c := make(chan struct{})
13     go func() { c <- struct{}{} }()
14     <-c
15     close(c)
16 }
```

```
==================
WARNING: DATA RACE
Read at 0x00c0000240d0 by goroutine 6:
  runtime.chansend()
      runtime/chan.go:158 +0x0
  main.main.func1()
      app.go:5 +0x35

Previous write at 0x00c0000240d0 by main goroutine:
  runtime.closechan()
      runtime/chan.go:355 +0x0
  main.main()
      app.go:6 +0xae

Goroutine 6 (running) created at:
  main.main()
      app.go:5 +0xa4
==================
Found 1 data race(s)
```

race检测数据竞争

```
 8      var a uint64
 9      go func() {
10          atomic.StoreUint64(&a, 1)
11      }()
12      println(a)
```

```
==================
WARNING: DATA RACE
Write at 0x00c0000160a8 by goroutine 6:
  sync/atomic.StoreInt64()
      runtime/race_amd64.s:248 +0xb
  main.main.func1()
      app.go:10 +0x44

Previous read at 0x00c0000160a8 by main goroutine:
  main.main()
      app.go:12 +0x92

Goroutine 6 (running) created at:
  main.main()
     app.go:9 +0x7a
==================
Found 1 data race(s)
```

race检测数据竞争

```go
 9 func ParallelWrite(data []byte) chan error {
10     res := make(chan error, 2)
11     f1, err := os.Create("file1")
12     go func() {
13         _, err = f1.Write(data)
14         res <- err
15     }()
16
17     f2, err := os.Create("file2")
18     go func() {
19         _, err = f2.Write(data)
20         res <- err
21     }()
22
23     return res
24 }
```

```
==================
WARNING: DATA RACE
Write at 0x00c00011a1d0 by main goroutine:
  main.ParallelWrite()
      demo4/app.go:17 +0x2e6
  main.main()
      demo4/app.go:6 +0x53

Previous write at 0x00c00011a1d0 by goroutine 7:
  main.ParallelWrite.func1()
      demo4/app.go:13 +0x94

Goroutine 7 (finished) created at:
  main.ParallelWrite()
      demo4/app.go:12 +0x290
  main.main()
      demo4/app.go:6 +0x53
==================
Found 1 data race(s)
```

race只记录最后一次的写访问，因此可能不能一次检测到所有的数据竞争，消除当前已知问题后，需要再次检测。

race检测数据竞争

```
11      for i := 0; i < 5; i++ {
12          go func() {
13              fmt.Println(i)
14          }()
15      }
```

```
==================
WARNING: DATA RACE
Read at 0x00c0000ba018 by goroutine 7:
  main.main.func1()
      demo3/app.go:13 +0x3d

Previous write at 0x00c0000ba018 by main goroutine:
  main.main()
      demo3/app.go:11 +0xab

Goroutine 7 (running) created at:
  main.main()
      demo3/app.go:12 +0x8f
==================
```

```
11      for i := 0; i < 5; i++ {
12          go func(j int) {
13              fmt.Println(j)
14          }(i)
15      }
```

消除数据竞争实践

消除数据竞争

可用的happens：

- 互斥锁/读写锁
- 原子操作
- 通道

- Sync包中的其他能力(sync.Map, sync.WaitGroup, sync.Cond,sync.Once)
- golang.org/x/sync中的能力(errgroup, semaphore, singleflight)

以map为例：

```
var mtx sync.Mutex
var m map[int]int

mtx.Lock()
m[1] = 1
mtx.Unlock()
```

```
var mtx sync.RWMutex
var m map[int]int

mtx.RLock()
println(m[1])
mtx.RUnlock()
```

```
var m sync.Map

m.Store("key", "value")
m.Load("key")
m.LoadOrStore("key", "value")
m.Delete("key")
m.LoadAndDelete("key")
```

sync.Map

```
var m map[int]int

newm := make(map[int]int)
for k, v := range m {
    newm[k] = v
}
newm[100] = 100
m = newm
```

整体替换：拷贝->修改->替换

消除数据竞争：将并发访问串行化

```
 8 var m map[int]int
 9 var c chan *request
10
11 func main() {
12     go worker()
13 }
14
15 func worker() {
16     for req := range c {
17         switch req.operation {
18         case "write":
19             m[req.key] = req.value
20             close(req.result)
21         case "read":
22             req.result <- m[req.key]
23             close(req.result)
24         }
25     }
26 }
27
28 func write() {
29     req := &request{"write", 1, 1, make(chan int, 1)}
30     c <- req
31     <-req.result
32 }
33
34 type request struct {
35     operation string
36     key       int
37     value     int
38     result    chan int
39 }
```

不支持并发访问的某种资源

启动一个工作协程

从通道持续接收各种操作“请求”

对于需要结果的请求将结果传回去

访问时，构造“请求”，并用通道发送，之后根据需要等待结果或继续其他工作。

消除数据竞争：运用原子操作

```go
if atomic.CompareAndSwapInt64(&i, 0, 1) {
    // do race operation
    atomic.StoreInt64(&i, 0)
}
```

```go
if atomic.SwapInt64(&i, 0) != 0 {
    // do race operation
    atomic.StoreInt64(&i, 1)
}
```

```go
for {
    old := atomic.LoadInt64(&i)
    newVal := old + 10
    if atomic.CompareAndSwapInt64(&i, old, newVal) {
        break
    }
}
```

可保证只有一个协程操作竞争资源

CAS的经典用法，保证当前协程基于最新数据做修改

消除数据竞争的误区

消除数据竞争的误区：标志变量

```
var a string
var done bool

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

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

As before, there is no guarantee that, in `main`, observing the write to `done` implies observing the write to `a`, so this program could print an empty string too. Worse, there is no guarantee that the write to `done` will ever be observed by `main`, since there are no synchronization events between the two threads. The loop in `main` is not guaranteed to finish.

1. 在main中，观察到done=true时，不保证能观察到a也被写入。
2. 在main中，不保证一定能观察到done被写入。

来源：https://golang.org/ref/mem

消除数据竞争的误区：基础数据类型的数据竞争

对于int，float，bool，指针系列的数据类型，这些数据类型上出现的数据竞争意味着什么?

```
var (
    last   int64
    pi     int64
    modify int64
)
go func() {
    pi = 1
    modify = nanotime()
}()
for {
    temp := nanotime()
    if pi == 0 {
        last = temp
    }
    break
}

if modify < last {
    println("modify", modify)
    println("last  ", last)
}
```

这些数据类型的大小小于等于寄存器的宽度，并且go语言能够保证按相应字节对齐，读写时使用一条指令即可完成，所以能够保证不出现部分修改的问题;

但不能保证修改后立即能被其他协程观察到。

是否需要消除取决于应用代码是否能够容忍读到旧数据

```
modify 939834946790400
last   939834946790500
```

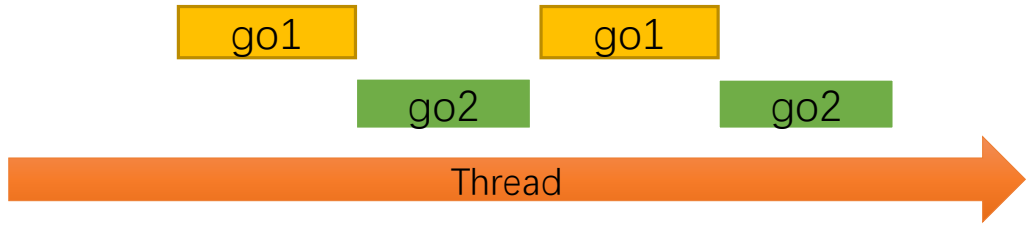
示意代码，复现需要增加额外处理。

硬件对存在数据竞争的程序的行为不做任何保证。

消除数据竞争的误区：GOMAXPROCS=1?

```go
func main() {
    runtime.GOMAXPROCS(1)
    for {
        test()
    }
}

func test() {
    a := make(map[int]int)
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        a[1] = 1
        wg.Done()
    }()
    a[2] = 2
    wg.Wait()
}
```

```
$ export GOMAXPROCS=1
$ go run -trimpath app.go
fatal error: concurrent map writes

goroutine 18877 [running]:
runtime.throw(0x47d682, 0x15)
        runtime/panic.go:1117 +0x72 fp=0xc000036760 sp=0xc000036730 pc=0x430832
runtime.mapassign_fast64(0x46eb60, 0xc000424060, 0x1, 0xc000423f40)
        runtime/map_fast64.go:101 +0x33e fp=0xc0000367a0 sp=0xc000036760 pc=0x40f53e
main.test.func1(0xc000424060, 0xc00040fc40)
        app.go:20 +0x45 fp=0xc0000367d0 sp=0xc0000367a0 pc=0x4647a5
runtime.goexit()
        runtime/asm_amd64.s:1371 +0x1 fp=0xc0000367d8 sp=0xc0000367d0 pc=0x460da1
created by main.test
        app.go:19 +0x87

goroutine 1 [runnable]:
main.test()
        app.go:23 +0xaa
main.main()
        app.go:11 +0x2f
exit status 2
```



Go语言运行时对存在数据竞争的程序的行为不做任何保证
开发者需要假设调度程序可能会在代码中的每个点在 Goroutine 之间切换

来源：https://stackoverflow.com/questions/19066478/data-race-when-gomaxprocs-1

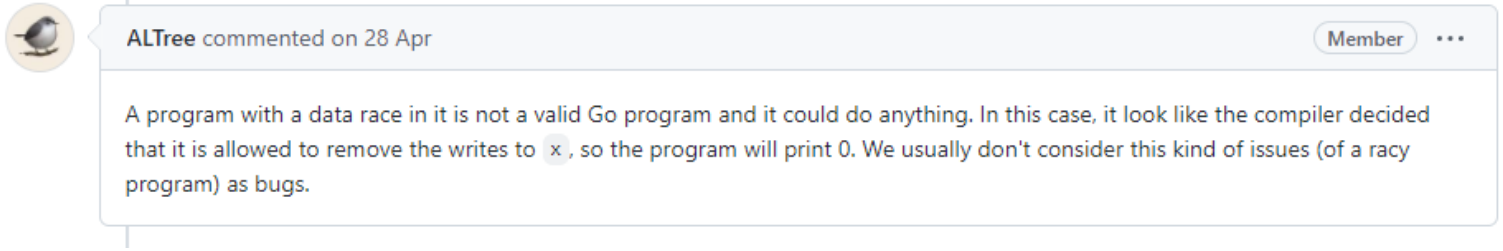
消除数据竞争的误区：Sleep?

依靠sleep延时能不能消除数据竞争?

```go
func main() {
    a := 0
    go func() {
        for {
            a = 1
        }
    }()
    time.Sleep(time.Second)
    log.Println(a)
    //打印结果永远是0
}
```

```
TEXT main.main.func1(SB) ./app.go
  app.go:11       0x486280        90              NOPL
  app.go:1        0x486281        ebfd            JMP main.main.func1(SB)
```

在编译结果中，a=1这个赋值语句已经被优化掉了，只剩下空的循环。



编译器对存在数据竞争的程序的行为不做任何保证。

来源：https://github.com/golang/go/issues/45826;https://blog.leonardwang.cn/archives/print0

消除数据竞争的误区：内置数据结构的数据竞争

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}

type stringStruct struct {
    str unsafe.Pointer
    len int
}

type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

Go语言的string，slice，interface的底层实现都是结构体，对于它们本身的读写访问需要考虑数据竞争

```
type hmap struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32

    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr

    extra *mapextra
}
```

```
type hchan struct {
    qcount   uint
    dataqsiz uint
    buf      unsafe.Pointer
    elemsize uint16
    closed   uint32
    elemtype *_type
    sendx    uint
    recvx    uint
    recvq    waitq
    sendq    waitq
    lock mutex
}
```

```
func makechan(t *chantype, size int) *hchan
func makemap(t *maptype, hint int, h *hmap) *hmap
```

而channel和map底层虽然也是结构体，使用者实际使用的是其指针，其自身的读写只需考虑指针的数据竞争

消除数据竞争的误区：锁保护的是谁?

```go
type item struct {
    data int
}

var m = make(map[string]*item)
var lock sync.RWMutex

func read(key string) *item {
    lock.RLock()
    defer lock.RUnlock()
    return m[key]
}
func write(key string, i *item) {
    lock.Lock()
    defer lock.Lock()
    m[key] = i
}
```

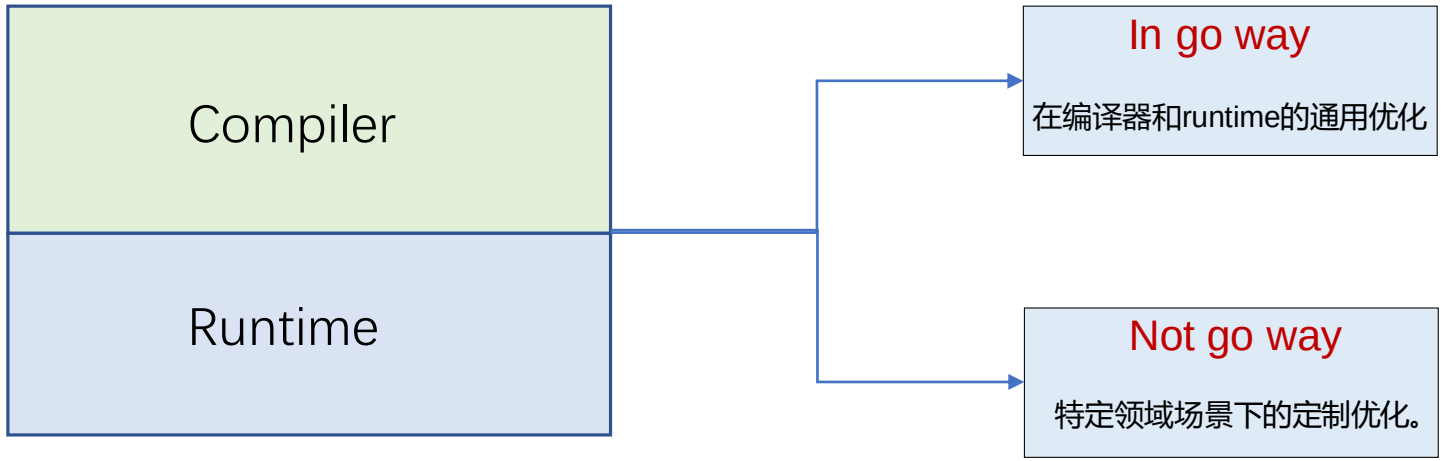
在使用读写锁保护map时，如果需要修改map里的元素，使用读锁还是写锁?

```go
func writeValue(key string, data int) {
    lock.RLock()
    defer lock.RUnlock()
    m[key].data = data
}

func writeValue2(key string, data int) {
    lock.Lock()
    defer lock.Unlock()
    m[key].data = data
}
```

总结

- 数据竞争的底层原理
- 消除数据竞争的原则
- 数据竞争的检测
- 数据竞争的常见消除方法
- 其他的一些误区

基于Golang的特性优化



社区贡献
（bug修复、特性优化、提案推送）

定制优化

对Go语言技术感兴趣或者想加入我们的Go语言优化团队，可以关注这个公众号。

这个公众号将会不定期发布一些技术文章和招聘信息。

华为德科精英研发项目： Golang软件研发

- 你对**Golang**有更深刻的理解了吗？！
- 你对我们的**项目**更加兴趣浓厚了吗？！
- 是否想要和我们的软件专家一起研讨，**成长自我**？！

- **加入我们吧！！**

联系人：陈女士

电话：18729056712（微信）

Thank You!