

---

## 【译】Go 语言数据竞争检测器

Data Race Detector

孔俊

2022-10-25

## Contents

<b>1</b>	<b>简介</b>	<b>2</b>
<b>2</b>	<b>使用</b>	<b>2</b>
<b>3</b>	<b>报告格式</b>	<b>2</b>
<b>4</b>	<b>选项</b>	<b>3</b>
<b>5</b>	<b>排除测试</b>	<b>4</b>
<b>6</b>	<b>怎样使用</b>	<b>4</b>
<b>7</b>	<b>典型的数据竞争</b>	<b>4</b>
7.1	循环计数器上的竞争 . . . . .	5
7.2	意外的共享变量 . . . . .	6
7.3	不受保护的全局变量 . . . . .	6
7.4	不受保护的原始类型变量。 . . . .	7
7.5	未同步的发送和关闭操作 . . . . .	8
<b>8</b>	<b>要求</b>	<b>9</b>
<b>9</b>	<b>运行时开销</b>	<b>9</b>

## 1 简介

数据竞争是并发程序中最普遍和最难调试的 **bug**。当两个 **goroutine** 并发访问同一变量且至少一个访问是写时发生数据竞争。更多细节参考 [The Go Memory Model](#)。

译者注

[The Go Memory Model](#) 可以参考我的博客 [【译】Go 语言内存模型：2022-06-06 版](#)。

这有一个可以导致程序崩溃（**crashes**）和内存损坏（**memory corruption**）的数据竞争的例子：

```
1 xian sxian sfunc main() {
2     c := make(chan bool)
3     m := make(map[string]string)
4     go func() {
5         m["1"] = "a" // First conflicting access.
6         c <- true
7     }()
8     m["2"] = "b" // Second conflicting access.
9     <-c
10    for k, v := range m {
11        fmt.Println(k, v)
12    }
13 }
```

## 2 使用

为了帮助调试这些 **bug**，Go 内置了数据竞争检测器（**data race detector**）。给 **go** 命令加 **-race** 标志来使用它：

```
1 $ go test -race mypkg // to test the package
2 $ go run -race mysrc.go // to run the source file
3 $ go build -race mycmd // to build the command
4 $ go install -race mypkg // to install the package
```

## 3 报告格式

当数据竞争检测器发现程序中的数据竞争时，它会打印一份报告。报告包含冲突访问（**conflicting accesses**）的 **goroutine** 和创建它的 **goroutine** 的堆栈跟踪（**stack traces**）。这是一个例子：

```
1 WARNING: DATA RACE
2 Read by goroutine 185:
3   net.(*pollServer).AddFD()
4     src/net/fd_unix.go:89 +0x398
5   net.(*pollServer).WaitWrite()
6     src/net/fd_unix.go:247 +0x45
7   net.(*netFD).Write()
8     src/net/fd_unix.go:540 +0x4d4
9   net.(*conn).Write()
10    src/net/net.go:129 +0x101
11   net.func·060()
12    src/net/timeout_test.go:603 +0xaf
13
14 Previous write by goroutine 184:
15   net.setWriteDeadline()
16     src/net/sockopt_posix.go:135 +0xdf
17   net.setDeadline()
18     src/net/sockopt_posix.go:144 +0x9c
19   net.(*conn).SetDeadline()
20     src/net/net.go:161 +0xe3
21   net.func·061()
22     src/net/timeout_test.go:616 +0x3ed
23
24 Goroutine 185 (running) created at:
25   net.func·061()
26     src/net/timeout_test.go:609 +0x288
27
28 Goroutine 184 (running) created at:
29   net.TestProlongTimeout()
30     src/net/timeout_test.go:618 +0x298
31   testing.tRunner()
32     src/testing/testing.go:301 +0xe8
```

## 4 选项

环境变量GORACE设置竞争检测器选项，格式为GORACE="option1=val1 option2=val2"。

有以下选项：

- **log\_path**（默认值为stderr）：竞争检测器把报告写入名为log\_path.pid的文件。专用文件名stdout和stderr分别将报告写到标准输出和标准错误。
- **exitcode**（默认值为66）：检测到数据竞争后退出时的退出码（exit status）。
- **strip\_path\_prefix**（默认值为""）：去除所有报告中的路径的前缀，让报告更简洁。
- **history\_size**（默认值为1）：每个goroutine的内存访问历史是‘32K \* 2\*\* history\_size个元素。增大这个值会增大内存开销，但可以避免报告报“failed to restore the stack”错误。
- **atexit\_sleep\_ms**（默认值为1000）：主goroutine退出前的总体眠（sleep）毫秒数。

## 5 排除测试

当你使用 `-race` 标志构建（build）时，`go` 命令定义了构建标签 `race`。你可以使用这个标签在运行竞争检测器时排除一些代码和测试。一些例子：

```
1 // +build !race
2
3 package foo
4
5 // The test contains a data race. See issue 123.
6 func TestFoo(t *testing.T) {
7     // ...
8 }
9
10 // The test fails under the race detector due to timeouts.
11 func TestBar(t *testing.T) {
12     // ...
13 }
14
15 // The test takes too long under the race detector.
16 func TestBaz(t *testing.T) {
17     // ...
18 }
```

## 6 怎样使用

使用竞争检测器（`go test -race`）运行你的测试。竞争检测器只检测到发生在运行时的竞争，所以它不能发现未执行代码路径中的竞争。如果你的测试覆盖率不足，你运行真实负载下使用 `-race` 构建的可执行文件时可能会发现更多竞争。

## 7 典型的数据竞争

这里有一些典型的数据竞争。竞争检测器可以检测到它们。

## 7.1 循环计数器上的竞争

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(5)
4     for i := 0; i < 5; i++ {
5         go func() {
6             fmt.Println(i) // Not the 'i' you are looking for.
7             wg.Done()
8         }()
9     }
10    wg.Wait()
11 }
```

函数数字量中的变量*i*与循环使用的变量相同，因此 *goroutine* 的读取与递增循环变量竞争。（该程序通常打印 55555，而不是 01234。）通过拷贝变量来修复这个程序：

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(5)
4     for i := 0; i < 5; i++ {
5         go func(j int) {
6             fmt.Println(j) // Good. Read local copy of the loop counter
7             wg.Done()
8         }(i)
9     }
10    wg.Wait()
11 }
```

### 译者注

在 Go 语言中，for 语句中定义的循环变量，存在于整个循环期间，而非一次循环。例如上面的 **for i := 0; i < 5; i++**，整个循环期间的*i*是同一个变量，而非每次循环创建一个新的局部变量*i*。

有人提议修改循环变量的语义，见 [redefining for loop variable semantics #56010](#)。

## 7.2 意外的共享变量

```
1 // ParallelWrite writes data to file1 and file2, returns the errors.
2 func ParallelWrite(data []byte) chan error {
3     res := make(chan error, 2)
4     f1, err := os.Create("file1")
5     if err != nil {
6         res <- err
7     } else {
8         go func() {
9             // This err is shared with the main goroutine,
10             // so the write races with the write below.
11             _, err = f1.Write(data)
12             res <- err
13             f1.Close()
14         }()
15     }
16     f2, err := os.Create("file2") // The second conflicting write to
17     err.
18     if err != nil {
19         res <- err
20     } else {
21         go func() {
22             _, err = f2.Write(data)
23             res <- err
24             f2.Close()
25         }()
26     }
27     return res
28 }
```

解决办法是在 goroutine 中引入新变量（注意:=的使用）。

```
1         ...
2         _, err := f1.Write(data)
3         ...
4         _, err := f2.Write(data)
5         ...
```

## 7.3 不受保护的全局变量

从多个 goroutine 调用以下代码会导致在 `servicemap` 上竞争。对同一 `map` 的并发读写是安全的：

```
1 var service map[string]net.Addr
2
3 func RegisterService(name string, addr net.Addr) {
4     service[name] = addr
5 }
6
7 func LookupService(name string) net.Addr {
8     return service[name]
9 }
```

为了让这份代码安全，使用互斥锁保护访问。

```
1 var (
2     service map[string]net.Addr
3     serviceMu sync.Mutex
4 )
5
6 func RegisterService(name string, addr net.Addr) {
7     serviceMu.Lock()
8     defer serviceMu.Unlock()
9     service[name] = addr
10 }
11
12 func LookupService(name string) net.Addr {
13     serviceMu.Lock()
14     defer serviceMu.Unlock()
15     return service[name]
16 }
```

#### 7.4 不受保护的原始类型变量。

数据竞争也会发生在原始类型变量（`bool`、`int`、`int64`等等）上，如下例所示：

```
1 type Watchdog struct{ last int64 }
2
3 func (w *Watchdog) KeepAlive() {
4     w.last = time.Now().UnixNano() // First conflicting access.
5 }
6
7 func (w *Watchdog) Start() {
8     go func() {
9         for {
10             time.Sleep(time.Second)
11             // Second conflicting access.
12             if w.last < time.Now().Add(-10*time.Second).UnixNano() {
13                 fmt.Println("No keepalives for 10 seconds. Dying.")
14                 os.Exit(1)
15             }
16         }
17     }()
```



即使是这种”无辜的“的数据竞争，由于编译器优化或处理器的内存乱序，也会导致难以调试的问题。

解决这种竞争的经典方法是使用 `channel` 或 `mutex`。为了保持无锁行为，也可以使用 `sync/atomic` 包。

```
1 type Watchdog struct{ last int64 }
2
3 func (w *Watchdog) KeepAlive() {
4     atomic.StoreInt64(&w.last, time.Now().UnixNano())
5 }
6
7 func (w *Watchdog) Start() {
8     go func() {
9         for {
10             time.Sleep(time.Second)
11             if atomic.LoadInt64(&w.last) < time.Now().Add(-10*time.
12                 Second).UnixNano() {
13                 fmt.Println("No keepalives for 10 seconds. Dying.")
14                 os.Exit(1)
15             }
16         }
17     }()
```

## 7.5 未同步的发送和关闭操作

像这个例子展示的那样，同一 `channel` 上未同步的发送和关闭操作也可能是竞争条件。

```
1 c := make(chan struct{}) // or buffered channel
2
3 // The race detector cannot derive the happens before relation
4 // for the following send and close operations. These two operations
5 // are unsynchronized and happen concurrently.
6 go func() { c <- struct{}{} }()
7 close(c)
```

根据 Go 语言内存模型，`channel` 上的发送 `happens before` 其上对应的接收完成。为了同步发送和关闭操作，使用接收操作确保发送在关闭前完成。

```
1 c := make(chan struct{}) // or buffered channel
2
3 go func() { c <- struct{}{} }()
4 <-c
5 close(c)
```

## 8 要求

数据竞争检测器需要启用 `cgo`, 支持 `linux/amd64`、`linux/ppc64le`、`linux/arm64`、`freebsd/amd64`、`netbsd/amd64`、`darwin/amd64`、`darwin/arm64` 和 `windows/amd64`。

## 9 运行时开销

竞争检测的开销因程序而异。对于典型的程序，内存使用量可能增加 5 到 10 倍，执行时间增加 2 到 20 倍。

目前竞争检测器额外为每个 `defer` 和 `recover` 语句分配 8 字节。这些额外分配的内存直到 `goroutine` 退出才释放。这意味着如果你有一个长时间运行的、定期执行 `defer` 和 `recover` 调用的 `goroutine`，程序的内存使用量可能无限量增加。这些内存分配不会显示在 `runtime.ReadMemStats` 或 `runtime/pprof` 的输出中。