【译】Go 语言数据竞争检测器

Data Race Detector

孔俊

2022-10-25

Contents

1 简介 **2**

2 使用 **2**

3 报告格式 **2**

4 选项 **3**

5 排除测试 **4**

6 怎样使用 **4**

7 典型的数据竞争 **4**

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

8 要求 **9**

9 运行时开销 **9**

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：

```
xian sxian sfunc main() {
    c := make(chan bool)
    m := make(map[string]string)
    go func() {
        m["1"] = "a" // First conflicting access.
        c <- true
    }()
    m["2"] = "b" // Second conflicting access.
    <-c
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

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：

```
$ go test -race mypkg    // to test the package
$ go run -race mysrc.go  // to run the source file
$ go build -race mycmd   // to build the command
$ 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：

```
WARNING: DATA RACE
Read by goroutine 185:
  net.(*pollServer).AddFD()
      src/net/fd_unix.go:89 +0x398
  net.(*pollServer).WaitWrite()
      src/net/fd_unix.go:247 +0x45
  net.(*netFD).Write()
      src/net/fd_unix.go:540 +0x4d4
  net.(*conn).Write()
      src/net/net.go:129 +0x101
  net.func·060()
      src/net/timeout_test.go:603 +0xaf

Previous write by goroutine 184:
  net.setWriteDeadline()
      src/net/sockopt_posix.go:135 +0xdf
  net.setDeadline()
      src/net/sockopt_posix.go:144 +0x9c
  net.(*conn).SetDeadline()
      src/net/net.go:161 +0xe3
  net.func·061()
      src/net/timeout_test.go:616 +0x3ed

Goroutine 185 (running) created at:
  net.func·061()
      src/net/timeout_test.go:609 +0x288

Goroutine 184 (running) created at:
  net.TestProlongTimeout()
      src/net/timeout_test.go:618 +0x298
  testing.tRunner()
      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 退出前的总休眠（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：

```
// +build !race

package foo

// The test contains a data race. See issue 123.
func TestFoo(t *testing.T) {
    // ...
}

// The test fails under the race detector due to timeouts.
func TestBar(t *testing.T) {
    // ...
}

// The test takes too long under the race detector.
func TestBaz(t *testing.T) {
    // ...
}
```

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

```
func main() {
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go func() {
            fmt.Println(i) // Not the 'i' you are looking for.
            wg.Done()
        }()
    }
    wg.Wait()
}
```

函数字面量中的变量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：

```
func main() {
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go func(j int) {
            fmt.Println(j) // Good. Read local copy of the loop counter
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

> 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

```
// ParallelWrite writes data to file1 and file2, returns the errors.
func ParallelWrite(data []byte) chan error {
    res := make(chan error, 2)
    f1, err := os.Create("file1")
    if err != nil {
        res <- err
    } else {
        go func() {
            // This err is shared with the main goroutine,
            // so the write races with the write below.
            _, err = f1.Write(data)
            res <- err
            f1.Close()
        }()
    }
    f2, err := os.Create("file2") // The second conflicting write to
       err.
    if err != nil {
        res <- err
    } else {
        go func() {
            _, err = f2.Write(data)
            res <- err
            f2.Close()
        }()
    }
    return res
}
```

解决办法是在 goroutine 中引入新变量（注意:=的使用）。

```
            ...
            _, err := f1.Write(data)
            ...
            _, err := f2.Write(data)
            ...
```

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service map 上竞争。对同一 map 的并发读写是是安全的：

```
var service map[string]net.Addr

func RegisterService(name string, addr net.Addr) {
    service[name] = addr
}

func LookupService(name string) net.Addr {
    return service[name]
}
```

为了让这份代码安全，使用互斥锁保护访问。

```
var (
    service   map[string]net.Addr
    serviceMu sync.Mutex
)

func RegisterService(name string, addr net.Addr) {
    serviceMu.Lock()
    defer serviceMu.Unlock()
    service[name] = addr
}

func LookupService(name string) net.Addr {
    serviceMu.Lock()
    defer serviceMu.Unlock()
    return service[name]
}
```

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：

```
type Watchdog struct{ last int64 }

func (w *Watchdog) KeepAlive() {
    w.last = time.Now().UnixNano() // First conflicting access.
}

func (w *Watchdog) Start() {
    go func() {
        for {
            time.Sleep(time.Second)
            // Second conflicting access.
            if w.last < time.Now().Add(-10*time.Second).UnixNano() {
                fmt.Println("No keepalives for 10 seconds. Dying.")
                os.Exit(1)
            }
        }
    }()
```

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。

```
type Watchdog struct{ last int64 }

func (w *Watchdog) KeepAlive() {
    atomic.StoreInt64(&w.last, time.Now().UnixNano())
}

func (w *Watchdog) Start() {
    go func() {
        for {
            time.Sleep(time.Second)
            if atomic.LoadInt64(&w.last) < time.Now().Add(-10*time.
                Second).UnixNano() {
                fmt.Println("No keepalives for 10 seconds. Dying.")
                os.Exit(1)
            }
        }
    }()
}
```

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。

```
c := make(chan struct{}) // or buffered channel

// The race detector cannot derive the happens before relation
// for the following send and close operations. These two operations
// are unsynchronized and happen concurrently.
go func() { c <- struct{}{} }()
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。

```
c := make(chan struct{}) // or buffered channel

go func() { c <- struct{}{} }()
<-c
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。