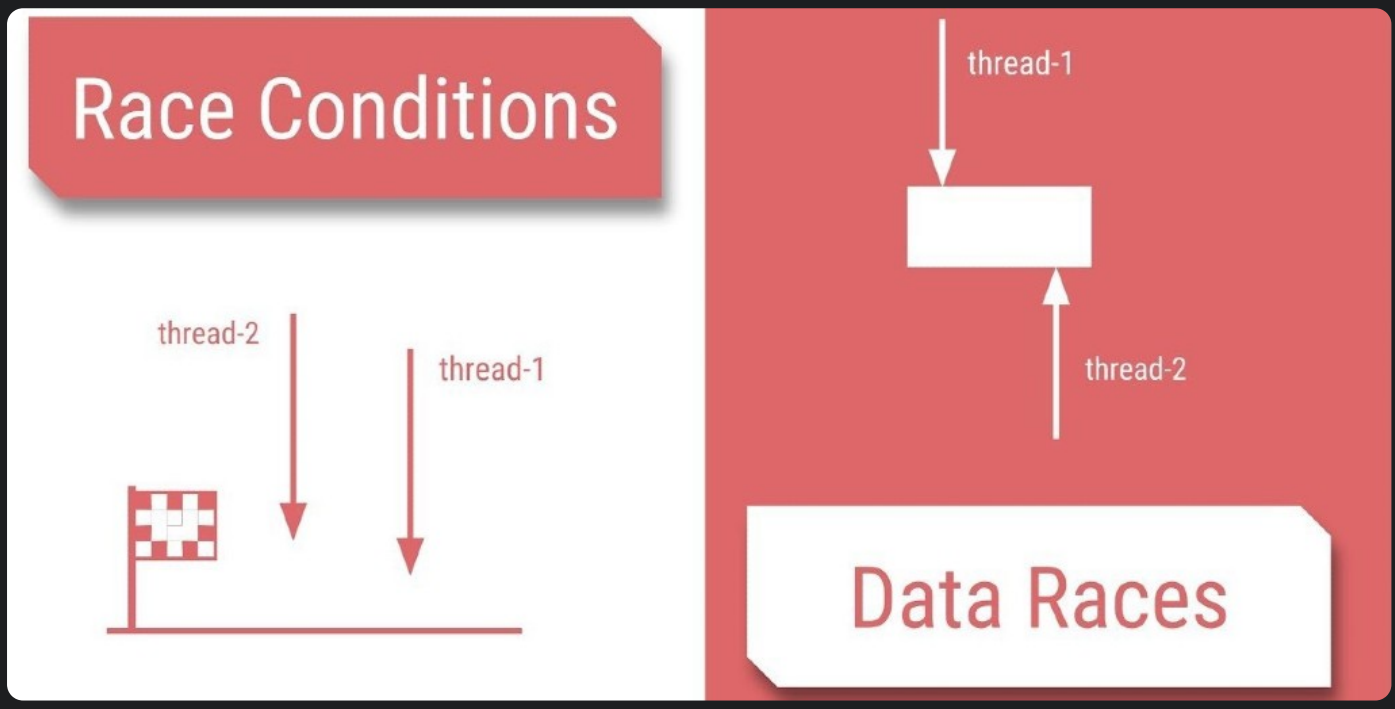
Race Condition и Data Race

июля 10, 2019 · 5 минут · German Gorelkin



Начинаем серию статей о проблемах многопоточности, параллелизме, concurrency и других интересных штуках.

1. **Race condition и Data Race**
2. Deadlocks, Livelocks и Starvation
3. Примитивы синхронизации в Go
4. Безопасная работа с каналами в Go
5. Goroutine Leaks

Race condition и **data race** — две разные проблемы многопоточности, которые часто путают. Попробуем разобраться.

Race condition

Существует много формулировок определения:

> **Race condition** представляет собой класс проблем, в которых корректное поведение системы зависит от двух независимых событий, происходящих в правильном порядке, однако отсутствует механизм, для того чтобы гарантировать фактическое возникновение этих событий.
>
> **Race condition** — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.
>
> **Race condition** — это нежелательная ситуация, которая возникает, когда устройство или система пытается выполнить две или более операций одновременно, но из-за природы устройства или системы, операции должны выполняться в правильной последовательности, чтобы быть выполненными правильно.
>
> **Race condition** — это недостаток, связанный с синхронизацией или упорядочением событий, что приводит к ошибочному поведению программы.

Но мне нравиться наиболее короткое и простое:

> **Race condition** — это недостаток, возникающий, когда время или порядок событий влияют на правильность программы.

Важно, что Race condition — это семантическая ошибка.

В проектирование электронных схем есть похожая проблема:

> *Состязание сигналов* — явление в цифровых устройствах несоответствия работы данного устройства с заданным алгоритмом работы по причине возникновения переходных процессов в реальной аппаратуре.

Рассмотрим пример, где результат не определен:

```
go func() {
    fmt.Printf("A->")
}()

go func() {
    fmt.Printf("B")
}()
```

Если запустить такой код много раз, то можно увидеть примерно такое:

```
A->B
A->B
A->B
A->B
BA->
A->B
```

Результат выполнения кода зависит от порядка выполнения горутин. Это типичная ошибка race condition. Ситуации могут быть гораздо сложней и не очевидней.

Учитывая, что race condition семантическая ошибка, нет общего способа который может отличить правильное и неправильное поведение программы в общем случае.

Помочь могут хорошие практики и проверенные паттерны.

Еще один пример:

```
x := 0
for {
    go func() {
        x++
    }()
    go func() {
        if x%2 == 0 {
            time.Sleep(1 * time.Millisecond)
            fmt.Println(x)
        }
    }()
}
```

В результате на консоле получим четные и нечетные числа, а расчитывали увидеть только четные.

Проблемы с доступом к общим ресурсам проще обнаружить автоматически и решаются они обычно с помощью **синхронизации**:

```
var mu sync.Mutex
x := 0
for{
   go func() {
      mu.Lock()
      x++
      mu.Unlock()
   }()
   go func() {
      mu.Lock()
      if x%2 == 0 {
         time.Sleep(1 * time.Millisecond)
         fmt.Println(x)
      }
      mu.Unlock()
   }()
}
```

или **локальной копией**:

```
x := 0
for i := 0; i < 1000; i++ {
   go func() {
      x++
   }()
   go func() {
      y := x
      if y%2 == 0 {
         time.Sleep(1 * time.Millisecond)
         fmt.Println(y)
      }
   }()
}
```

Data Race

> **Data race** это состояние когда разные потоки обращаются к одной ячейке памяти без какой-либо синхронизации и как минимум один из потоков

> осуществляет запись.

Пример с балансом на счету:

```
type account struct {
   balance int
}
func deposit(acc *account, amount int) {
   acc.balance += amount
}
```

Запускаем в разных горутинах:

```
acc := account{balance: 0}
var wg sync.WaitGroup

for i := 0; i < 1000; i++ {
   wg.Add(1)
   go func(n int) {
      deposit(&acc, 1)
      wg.Done()
   }(i)
}
wg.Wait()

fmt.Printf("balance=%d\n", acc.balance)
```

Изначально баланс равен 0, депозитим 1000 раз по 1. Ожидаем баланс равный 1000, но результат другой:

```
balance=876
```

Потеряли много денег.

Причина в том, что операция acc.balance += amount не атомарная. Она может разложиться на 3:

```
tmp := acc.balance
tmp = tmp + amount
acc.balance = tmp
```

Пока мы меняем временную переменную в одном потоке, в других уже изменен

основной balance. Таким образом теряется часть изменений.

Например, у нас 2 параллельных потока выполнения, каждый должен прибавить к балансу по 1:

```
tmp := acc.balance // 100      ||  tmp := acc.balance // 100
tmp = tmp + amount // 101      ||  tmp = tmp + amount // 101
acc.balance = tmp  // 101      ||  acc.balance = tmp  // 101
```

Ожидали получить баланс=102, а получили = 101.

У **Data Race** есть точное определение, которое не обязательно связано с корректностью, и поэтому их можно обнаружить. Существует множество разновидностей детекторов гонки данных (статическое/динамическое обнаружение, обнаружение на основе блокировок, обнаружение основанное на предшествующих событий, обнаружение гибридного data race).

У Go есть хороший **Data Race Detector** с помощью которого такие ошибки можно обнаружить.

Решается проблема с помощью синхронизации:

```
var mu sync.Mutex

func deposit(acc *account, amount int) {
   mu.Lock()
   acc.balance += amount
   mu.Unlock()
}
```

Иногда более эффективным решением будет использовать пакет atomic.

```
func deposit(acc *account, amount int64) {
   atomic.AddInt64(&acc.balance, amount)
}
```

Race Condition и Data Race

Функция для перевода средств с одного счета на другой:

```
func transfer1(accFrom, accTo *account, amount int) error {
   if accFrom.balance < amount {
      return fmt.Errorf("accFrom.balance<amount")
   }
   accTo.balance += amount
   accFrom.balance -= amount
   return nil
}
```

На одном счету у нас будет 1000, а на другом 0. Переводим по 1 в 1000 горутинах и ожидаем, что все деньги из одного счета перетекут в другой:

```
accFrom := account{balance: 1000}
accTo := account{balance: 0}
var wg sync.WaitGroup

for i := 0; i < 1000; i++ {
   wg.Add(1)
   go func(n int) {
      err := transfer1(&accFrom, &accTo, 1)
      if err != nil {
         fmt.Printf("error for n=%d\n", n)
      }
      wg.Done()
   }(i)
}
wg.Wait()

fmt.Printf("accFrom.balance=%d\naccTo.balance=%d\n", accFrom.balance, accTo.balan
```

Но результат может быть таким:

```
accFrom.balance=84
accTo.balance=915
```

Если запустить цикл на большее кол-во операций, то можно получить еще интересней:

```
accFrom.balance=0
accTo.balance=997
```

При вызове из нескольких потоков без внешней синхронизации эта функция допускает как **data race** (несколько потоков могут одновременно пытаться обновить баланс счета), так и **race condition** (в параллельном контексте это приведет к потере денег).

Для решения можно применить синхронизацию и локальную копию. Общая логика может быть не такой линейной и в итоге код может выглядит например так:

```
func transfer2(accFrom, accTo *account, amount int) error {
   mu.Lock()
   bal := accFrom.balance
   mu.Unlock()

   if bal < amount {
      return fmt.Errorf("accFrom.balance<amount")
   }
   mu.Lock()
   accTo.balance += amount
   mu.Unlock()

   mu.Lock()
   accFrom.balance -= amount
   mu.Unlock()

   return nil
}
```

У нас синхронизированы все участки с записью и чтением, у нас есть локальная копия, **Race Detector больше не ругается на код**. Запускаем 1000 операций и получаем верный результат:

```
accFrom.balance=0
accTo.balance=1000
```

Но что если горутин будет 10к:

```
accFrom.balance=-15
accTo.balance=1015
```

Мы решили проблему data race, но race condition остался. В данном случае можно

сделать блокировку на всю логику перевода средств, но это не всегда возможно.

Решив Data Race через синхронизацию доступа к памяти (блокировки) не всегда решается race condition и logical correctness.

Код примеров **github**.

Дополнительная информация

- https://blog.regehr.org/archives/490
- https://dzone.com/articles/race-condition-vs-data-race
- https://golang.org/doc/articles/race_detector.html
- https://en.wikipedia.org/wiki/Race_condition

go concurrency

« ПРЕДЫДУЩАЯ
Распределенные Системы. Брендан Бёрнс. Введение

СЛЕДУЮЩАЯ »
Выравнивание И Заполнение Структур