Go sync.Mutex. Normal и Starvation Mode

октября 13, 2024 · 8 минут · German Gorelkin

1. **Go sync.Mutex. Normal и Starvation Mode**
2. Go sync.WaitGroup и Проблема выравнивания
3. Go sync.Pool и механика, лежащая в его основе
4. Go sync.Cond, самый недооцененный механизм синхронизации

Перевод/заметки Go sync.Mutex: Normal and Starvation Mode

Mutex, или **MUT**ual **EX**clusion, в Go - это способ убедиться, что только одна горутина одновременно работает с общим ресурсом. Этим ресурсом может быть кусок кода, целое число, map, структура, канал или практически все, что угодно.

Зачем нам нужен `sync.Mutex`?

Если вы достаточно долго работали с `map` в Go, вы могли столкнуться с такой неприятной ошибкой:

```
fatal error: concurrent map read and map write
```

Это происходит потому, что мы не защищаем нашу `map` от нескольких горутин, которые одновременно пытаются получить к ней доступ.

Когда горутина блокирует мьютекс, она, по сути, говорит: «*Эй, я собираюсь немного попользоваться этим общим ресурсом*», а все остальные горутины должны ждать, пока мьютекс не будет разблокирован. Как только гарутина закончит работу, она должна разблокировать мьютекс.

Все просто. Давайте посмотрим, как это работает, на примере простого счетчика:

```
var counter = 0
var wg sync.WaitGroup

func incrementCounter() {
    counter++
    wg.Done()
}

func main() {
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go incrementCounter()
    }

    wg.Wait()
    fmt.Println(counter)
}
```

Итак, у нас есть переменная-счетчик, которую используют 1000 горутинами.

Новичку в Go покажется, что результат должен быть равен 1000, но это не так. Этот эффект называется «**race condition**».

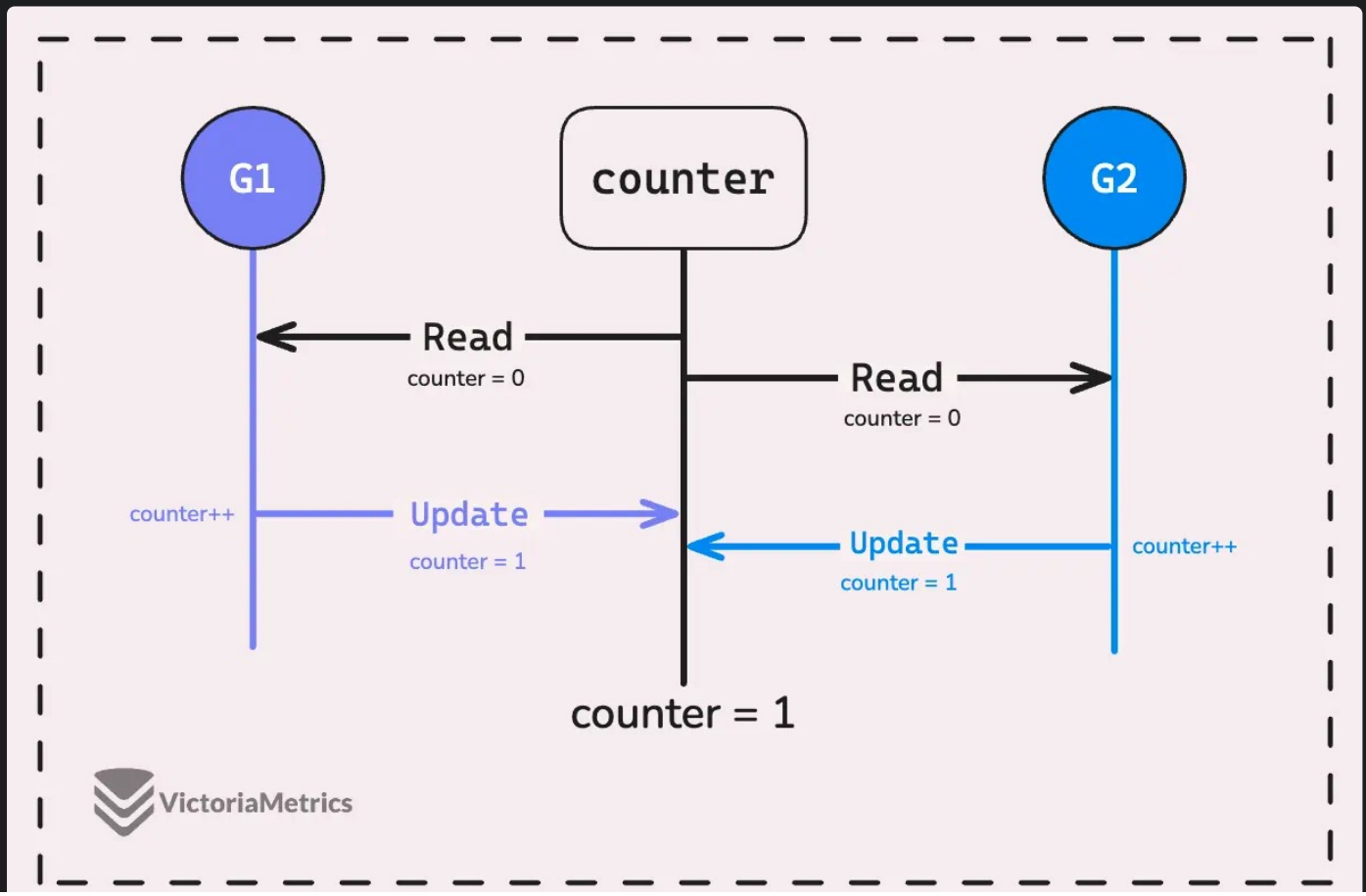
Состояние гонки возникает, когда несколько горутин пытаются одновременно получить доступ к общим данным и изменять их без синхронизации. В данном случае операция инкремента (`counter++`) не является атомарной.

Она состоит из нескольких шагов, ниже приведен ассемблерный код Go для `counter++` в архитектуре ARM64:

```
MOVD    main.counter(SB), R0
ADD $1, R0, R0
MOVD    R0, main.counter(SB)
```

`counter++` - это операция **read-modify-write**, и эти шаги не являются атомарными, то есть они не выполняются как единое, непрерывное действие.

Например, горутина *G1* считывает значение счетчика, а перед тем как записать обновленное значение, горутина *G2* считывает то же самое значение. Затем обе горутины записывают свои обновленные значения, но поскольку они считывают одно и то же исходное значение, один инкремент практически теряется.



Использование пакета `atomic` - хороший способ справиться с этой проблемой, но сегодня мы сосредоточимся на том, как мьютекс решает эту проблему:

```
var counter = 0
var wg sync.WaitGroup
var mutex sync.Mutex

func incrementCounter() {
    mutex.Lock()
    counter++
    mutex.Unlock()
    wg.Done()
}

func main() {
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go incrementCounter()
    }

    wg.Wait()
    fmt.Println(counter)
}
```

Теперь результат равен 1000, как мы и ожидали. Использование мьютекса здесь очень простое: оберните критическую секцию с помощью `Lock` и `Unlock`. Но будьте внимательны, если вы вызовете `Unlock` на уже разблокированном мьютексе, это приведет к фатальной ошибке `sync: unlock of unlocked mutex`.

Кроме того, вы можете установить `GOMAXPROCS` на 1, выполнив `runtime.GOMAXPROCS(1)`, и результат тогда будет корректным. Это происходит потому, что наши горутины не будут выполняться параллельно, а функция достаточно проста, чтобы не быть вытесненной во время выполнения.

Mutex Structure: The Anatomy

Прежде чем мы рассмотрим, как работает поток блокировки и разблокировки `sync.Mutex`, давайте разберем структуру самого мьютекса:

```
package sync
```

```
type Mutex struct {
    state int32
    sema  uint32
}
```

По своей сути мьютекс в Go состоит из двух полей: `state` и `sema`. Они могут выглядеть как простые числа, но на самом деле в них заложено нечто большее, чем кажется на первый взгляд.

Поле `state` - это 32-битное целое число, которое показывает текущее состояние мьютекса. Оно разделено на несколько битов, которые кодируют различные части информации о мьютексе.

Давайте составим список состояний по картинке:

- **Locked (бит 0)**: Заблокирован ли мьютекс в данный момент. Если значение равно 1, мьютекс заблокирован и никакая другая горутина не может его захватить.
- **Woken (бит 1)**: Устанавливается в 1, если какая-либо горутина была разбужена и пытается получить мьютекс. Другие горутины не должны быть разбужены без необходимости.
- **Starvation (бит 2)**: Этот бит показывает, находится ли мьютекс в режиме голодания (установлен в 1). О том, что означает этот режим, мы расскажем чуть позже.
- **Официант (бит 3-31)**: Остальные биты отслеживают, сколько горутин ожидают мьютекс.

Другое поле, `sema`, - это uint32, которое действует как семафор для управления и подачи сигналов ожидающим горутинам. Когда мьютекс разблокируется, одна из ожидающих горутин пробуждается, чтобы получить блокировку.

В отличие от поля `state`, `sema` не имеет определенной схемы расположения битов и полагается на внутренний код во время выполнения для обработки логики семафора.

Mutex Lock Flow

В функции `mutex.Lock` есть два пути: быстрый путь для обычного случая и медленный путь для обработки нестандартного случая.

```
func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    // Slow path (outlined so that the fast path can be inlined)
    m.lockSlow()
}
```

Быстрый путь предназначен для очень быстрой работы и, как ожидается, справится с большинством случаев получения блокировки, когда мьютекс еще не используется. Этот путь также инлайнится, то есть встраивается непосредственно в вызывающую функцию:

```
$ go build -gcflags="-m"

./main.go:13:12: inlining call to sync.(*Mutex).Lock
./main.go:15:14: inlining call to sync.(*Mutex).Unlock
```

FYI, этот встраиваемый быстрый путь - изящный трюк, использующий оптимизацию в Go, и он часто встречается в исходном коде Go.

Когда операция **CAS(Compare And Swap)** в быстром пути завершается неудачей, это означает, что поле `state` не было равно 0, поэтому мьютекс в данный момент заблокирован.

На самом деле здесь важен медленный путь `m.lockSlow`, который выполняет большую часть тяжелой работы. При медленном пути горутина продолжает активно **spinning**, пытаясь получить блокировку, а не просто переходит в очередь ожидания.

> Что значит *spinning*?

Спиннинг(spinning) означает, что горутина входит в замкнутый цикл, многократно проверяя состояние мьютекса и не передавая CPU.

В данном случае это не простой цикл, а низкоуровневые инструкции ассемблера для выполнения **spin-wait**. Давайте посмотрим на этот код на архитектуре ARM64:

```
TEXT runtime·procyield(SB),NOSPLIT,$0-0
    MOVWU   cycles+0(FP), R0
again:
    YIELD
    SUBW    $1, R0
    CBNZ    R0, again
    RET
```

Ассемблерный код выполняет замкнутый цикл в течение 30 раз (`runtime.procyield(30)`), многократно освобождая CPU и уменьшая счетчик спинов.

После спиннинга он снова пытается завладеть блокировкой. Если это не удается, у него есть еще три шанса, прежде чем он остановится. Таким образом, в общей сложности он пытается сделать 120 циклов. Если он все еще не может получить блокировку, он увеличивает количество ожидающих, ставит себя в очередь ожидания, уходит в сон, ждет сигнала, чтобы проснуться и попробовать снова.

> Зачем нам нужен спиннинг?

Идея спиннинга заключается в том, чтобы подождать некоторое время в надежде, что мьютекс скоро освободится, позволяя горутине захватить мьютекс без накладных расходов на **sleep-wake** цикл.

Если наш компьютер не имеет нескольких ядер, спиннинг не включается, потому что он просто тратит процессорное время.

> Но что, если другая горутина уже ждет мьютекс? Кажется несправедливым, если эта горутина получит блокировку первой.

Поэтому у нашего мьютекса есть два режима: **Normal** и **Starvation**. *В режиме Starvation спиннинг не работает*.

В обычном режиме горутины, ожидающие мьютекса, выстраиваются в очередь по принципу «первый пришел - первый ушел» (FIFO). Когда горутина просыпается, чтобы попытаться захватить мьютекс, она не получает управление сразу. Вместо этого она вынуждена конкурировать с любыми новыми горутинами, которые также хотят получить мьютекс в это время.

Эта конкуренция складывается в пользу новых горутин, потому что они уже работают на CPU и могут быстро попытаться захватить мьютекс, в то время как горутина в очереди еще только просыпается.

В результате только что проснувшаяся горутина может часто проигрывать гонку новым претендентам и возвращаться в начало очереди.

> Что, если этой горутине не повезло, и она всегда просыпается при появлении новой горутины?

Хороший вопрос. Если это произойдет, то блокировка так и не будет получена. Вот почему нам нужно переключить мьютекс в режим голодания. Существует issue #13086, в котором обсуждается проблема несправедливости предыдущего дизайна.

Режим голодания(starvation) включается, если горутине не удается получить блокировку в течение более 1 миллисекунды. Он предназначен для того, чтобы ожидающие горутины в конце концов получили справедливый шанс на мьютекс.

В этом режиме, когда горутина освобождает мьютекс, она напрямую передает управление горутине, находящейся в начале очереди. Это означает отсутствие конкуренции и гонки со стороны новых горутин. Они даже не пытаются получить его и просто присоединяются к концу очереди ожидания.

На изображении выше мьютекс продолжает передавать доступ G1, G2 и так далее. Каждая ожидавшая горутина получает управление и проверяет два условия:

- Если это последняя горутина в очереди ожидания.
- Если ей пришлось ждать менее одной миллисекунды.

Если одно из этих условий истинно, мьютекс переключается обратно в обычный режим.

Mutex Unlock Flow

Поток разблокировки проще, чем поток блокировки. У нас все еще есть два пути: быстрый путь, который инлайнится, и медленный путь, который обрабатывает нестандартные случаи.

Быстрый путь сбрасывает заблокированный бит в состояние мьютекса. Если вы помните анатомию мьютекса, это первый бит `mutex.state` . Если сброс этого бита делает состояние нулевым, это означает, что никакие другие флаги не установлены (например, ожидающие горутины), и наш мьютекс теперь полностью свободен.

Но что делать, если состояние не нулевое?

Вот тут-то и приходит на помощь медленный путь, который должен знать, находится ли наш мьютекс в нормальном режиме или в режиме голодания:

```
func (m *Mutex) unlockSlow(new int32) {
    // 1. Attempting to unlock an already unlocked mutex
    // will cause a fatal error.
    if (new+mutexLocked)&mutexLocked == 0 {
        fatal("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
        old := new
        for {
            // 2. If there are no waiters, or if the mutex is already locked,
            // or woken, or in starvation mode, return.
            if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexSta
                return
            }
            // Grab the right to wake someone.
            new = (old - 1<<mutexWaiterShift) | mutexWoken
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
                runtime_Semrelease(&m.sema, false, 1)
                return
            }
            old = m.state
        }
```

```
    } else {
        // 3. If the mutex is in starvation mode, hand off the ownership
        // to the first waiting goroutine in the queue.
        runtime_Semrelease(&m.sema, true, 1)
    }
}
```

В обычном режиме, если есть ожидающие и ни одна из горутин не была разбужена или не получила блокировку, мьютекс пытается атомарно уменьшить счетчик ожидающих и включить флаг `mutexWoken` . Если это удалось, он отпускает семафор, чтобы разбудить одну из ожидающих горутин для получения мьютекса.

В режиме голодания он атомарно увеличивает семафор (`mutex.sem`) и передает право владения мьютексом непосредственно первой ожидающей горутине в очереди. Второй аргумент `runtime_Semrelease` определяет, будет ли передача истинной.

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Go sync.WaitGroup и Проблема выравнивания

СЛЕДУЮЩАЯ »
Go Профилировщики