Go sync.WaitGroup и Проблема выравнивания

октября 17, 2024 · 12 минут · German Gorelkin

Перевод/заметки Go sync.WaitGroup and The Alignment Problem

WaitGroup - это способ подождать, пока несколько горутин закончат свою работу. Мы начнем с основ, а затем, как обычно, разберемся, как это устроено под капотом.

Что такое `sync.WaitGroup`?

Давайте сначала разберемся с проблемой: представьте, что у вас на руках большая задача, и вы решили разбить ее на более мелкие подзадачи, которые могут выполняться одновременно, не завися друг от друга.

Чтобы справиться с этим, мы используем горутины, поскольку они позволяют выполнять эти подзадачи параллельно:

```
func main() {
    for i := 0; i < 10; i++ {
        go func(i int) {
            fmt.Println("Task", i)
        }(i)
    }

    fmt.Println("Done")
}

// Output:
// Done
```

Но вот в чем дело: велика вероятность того, что главная горутина завершит работу раньше, чем остальные запущенные задачи.

Когда мы запускаем множество горутин, чтобы они выполняли свои задачи, мы хотим отслеживать их, чтобы главная горутина не закончила работу и не вышла до того, как все остальные закончат. Вот тут-то и приходит на помощь WaitGroup. Каждый раз, когда одна из наших горутин завершает свою задачу, она сообщает об этом WaitGroup.

Когда все горутины зарегистрировались как «done», главная горутина понимает, что можно заканчивать работу, и все аккуратно завершается.

```
func main() {
```

```
    var wg sync.WaitGroup

    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func(i int) {
            defer wg.Done()
            fmt.Println("Task", i)
        }(i)
    }

    wg.Wait()
    fmt.Println("Done")
}

// Output:
// Task 0
// Task 1
// Task 2
// Task 3
// Task 4
// Task 5
// Task 6
// Task 7
// Task 8
// Task 9
// Done
```

Итак, вот как это обычно происходит:

- *Добавление горутин*: Перед запуском горутин вы сообщаете WaitGroup, сколько их ожидать. Для этого используется функция `WaitGroup.Add(n)`, где n - количество горутин, которые вы планируете запустить.
- *Горутины запущены*: Каждая горутина запускается и делает свое дело. Когда она закончит, она должна сообщить об этом WaitGroup, вызвав `WaitGroup.Done()`, чтобы уменьшить счетчик на единицу.
- *Ожидание всех горутин*: В главной горутине вы вызываете `WaitGroup.Wait()`. Это приостанавливает работу главной горутины до тех пор, пока счетчик в WaitGroup не достигнет нуля. Проще говоря, она ждет, пока все остальные горутины не завершат свою работу и не подадут сигнал об окончании.

Обычно `WaitGroup.Add(1)` используется при запуске горутины:

```
for i := 0; i < 10; i++ {
```

```
    wg.Add(1)
    go func() {
        defer wg.Done()
        ...
    }()
}
```

Технически оба способа хороши, но использование `wg.Add(1)` немного снижает производительность. Тем не менее, он менее подвержен ошибкам по сравнению с использованием `wg.Add(n)` .

Как выглядит `sync.WaitGroup` ?

Для начала давайте посмотрим исходный код `sync.WaitGroup` .

```
type WaitGroup struct {
    noCopy noCopy

    state atomic.Uint64
    sema  uint32
}

type noCopy struct{}

func (*noCopy) Lock()   {}
func (*noCopy) Unlock() {}
```

В Go легко скопировать структуру, просто присвоив ее другой переменной. Но некоторые структуры, например WaitGroup, копировать не стоит. Потому что внутреннее состояние может рассинхронизироваться между копиями.

noCopy

Структура `noCopy` включена в WaitGroup, чтобы помочь предотвратить ошибки копирования, но не путем выброса ошибок, а в качестве предупреждения.

Структура `noCopy` на самом деле не влияет на работу вашей программы. Вместо этого она служит маркером, по которому такие инструменты, как `go vet` , могут определить, когда структура была скопирована.

```
type noCopy struct{}
```

```
func (*noCopy) Lock()   {}
func (*noCopy) Unlock() {}
```

- У нее нет полей, поэтому он не занимает никакого значимого места в памяти.
- У нее есть два метода, `Lock` и `Unlock`, которые ничего не делают (no-op). Эти методы предназначены для работы с программой проверки `-copylocks` в утилите `go vet`.

Когда вы запускаете `go vet` на своем коде, он проверяет, не были ли скопированы структуры с полем `noCopy`.

Он выдаст сообщение об ошибке, чтобы сообщить вам о возможной проблеме. Это даст вам возможность исправить ее до того, как она превратится в баг:

```
func main() {
    var a sync.WaitGroup
    b := a

    fmt.Println(a, b)
}

// go vet:
// assignment copies lock value to b: sync.WaitGroup contains sync.noCopy
// call of fmt.Println copies lock value: sync.WaitGroup contains sync.noCopy
// call of fmt.Println copies lock value: sync.WaitGroup contains sync.noCopy
```

Обратите внимание, что это всего лишь мера предосторожности: когда мы пишем и тестируем наш код, мы все равно можем запускать его как обычно.

Внутреннее состояние

Состояние WaitGroup хранится в переменной `atomic.Uint64`. В этом единственном значении упаковано несколько вещей.

Вот как это происходит:

- *Счетчик (старшие 32 бита)*: Эта часть отслеживает количество горутин, которых ожидает WaitGroup. Когда вы вызываете `wg.Add()` с положительным значением, он увеличивает этот счетчик, а когда вы вызываете `wg.Done()`, он

уменьшает счетчик на единицу.
- *Ожидание (младшие 32 бита)*: Отслеживает количество горутин, ожидающих, пока этот счетчик (старшие 32 бита) достигнет нуля. Каждый раз, когда вы вызываете `wg.Wait()`, он увеличивает этот счетчик «ожидания». Как только счетчик достигнет нуля, он освободит все ожидающие горутины.

И последнее поле, `sema uint32`, которое является внутренним семафором, управляемым рантаймом Go.

Если горутина вызывает `wg.Wait()` и счетчик не равен нулю, она увеличивает счетчик ожидающих, а затем блокирует его, вызывая `runtime_Semacquire(&wg.sema)`. Этот вызов функции переводит горутину в спящий режим, пока она не будет разбужена соответствующим вызовом `runtime_Semrelease(&wg.sema)`.

Проблема выравнивания

Давайте посмотрим, как WaitGroup развивалась на протяжении нескольких версий Go:

Могу сказать, что ядро WaitGroup (counter, waiter, и semaphore) практически не менялось в разных версиях Go. Но их структуру много раз изменяли.

Когда мы говорим о *(выравнивании(alignment))*, мы имеем в виду необходимость хранения типов данных по определенным адресам памяти для обеспечения эффективного доступа.

Например, в 64-битной системе 64-битное значение, такое как `uint64`, в идеале должно храниться по адресу памяти, кратному 8 байтам. Причина в том, что процессор может получить выровненные данные за один раз, но если данные не выровнены, то для доступа к ним может потребоваться несколько операций.

Вот тут-то и возникают сложности:

На 32-битных архитектурах компилятор не гарантирует, что 64-битные значения

будут выровнены по границе 8 байт. Вместо этого они могут быть выровнены только по 4-байтовой границе.

Это становится проблемой, когда мы используем пакет `atomic` для выполнения операций над переменной `state`. В пакете `atomic` специально отмечено:

> На ARM, 386 и 32-битных MIPS ответственность за выравнивание 64-битных слов, доступ к которым осуществляется атомарно с помощью примитивных атомарных функций, лежит на вызывающей стороне.

Это означает, что если мы не выровняем переменную `state uint64` по границе 8 байт на этих 32-битных архитектурах, то это может привести к аварийному завершению программы.

Давайте посмотрим, как эта проблема решалась в разных версиях.

Go 1.5: `state1 [12]byte`

```
type WaitGroup struct {
    state1 [12]byte
    sema   uint32
}

func (wg *WaitGroup) state() *uint64 {
    if uintptr(unsafe.Pointer(&wg.state1))%8 == 0 {
        return (*uint64)(unsafe.Pointer(&wg.state1))
    } else {
        return (*uint64)(unsafe.Pointer(&wg.state1[4]))
    }
}
```

Вместо того чтобы напрямую использовать `uint64` для состояния, WaitGroup отводит 12 байт в массиве (`state1 [12]byte`).

Цель использования 12 байт заключается в том, чтобы обеспечить достаточно места для поиска 8-байтового сегмента, который правильно выровнен.

Выравнивание структуры зависит от типа ее поля, поэтому в зависимости от архитектуры оно может быть 8- или 4-байтовым. Если начальный адрес `state1` не выровнен, код может просто сдвинуться на несколько байт (в данном случае

на 4 байта), чтобы найти секцию в этих 12 байтах, которая будет выровнена.

Go 1.11: `state1 [3]uint32`

4 байта из 12 по сути ничего не делали. В Go 1.11 было принято решение упростить работу, объединив состояние (включающее counter и waiter) и sema всего в 3 поля `uint32` .

```
type WaitGroup struct {
    noCopy noCopy

    state1 [3]uint32
}

// state returns pointers to the state and sema fields stored within wg.state1.
func (wg *WaitGroup) state() (statep *uint64, semap *uint32) {
    if uintptr(unsafe.Pointer(&wg.state1))%8 == 0 {
        return (*uint64)(unsafe.Pointer(&wg.state1)), &wg.state1[2]
    } else {
        return (*uint64)(unsafe.Pointer(&wg.state1[1])), &wg.state1[0]
    }
}
```

Итак, как же нам теперь избежать проблемы выравнивания состояний? Ответ кроется в методе `state()` , но если вы не знакомы с пакетом `unsafe` , он может показаться немного сложным.

Поскольку `state1` теперь представляет собой массив `uint32` , он начинается с адреса, кратного 4 байтам.

Для переменной типа массив ее выравнивание соответствует выравниванию типа элемента.

Вот как работает метод `state()` , если адрес `wg.state1` не выровнен по 8 байт:

- Первый элемент (`state1[0]`) используется для *sema*.
- Второй элемент (`state1[1]`) используется для отслеживания *waiter*.
- Третий элемент (`state1[2]`) используется для *counter*.

А если адрес `wg.state1` выровнен по 8 байт, то элементы в `state1`

переставляются между *state* и *sema*, как показано на рисунке выше.

Go 1.18: `state1 uint64; state2 uint32`

Проблема предыдущего подхода заключалась в том, что независимо от того, имеем ли мы дело с 64- или 32-битным выравниванием, нам все равно приходилось жонглировать *state* и *sema* в 12-байтном массиве, верно? Но поскольку большинство систем сегодня 64-битные, имело смысл оптимизировать их под более распространенный сценарий.

Идея в Go 1.18 проста: на 64-битной системе нам не нужно делать ничего особенного, `state1` хранит состояние, а `state2` служит для семафора:

```
type WaitGroup struct {
    noCopy noCopy

    state1 uint64
    state2 uint32
}

// state returns pointers to the state and sema fields stored within wg.state*.
func (wg *WaitGroup) state() (statep *uint64, semap *uint32) {
    if unsafe.Alignof(wg.state1) == 8 || uintptr(unsafe.Pointer(&wg.state1))%8 ==
        // state1 is 64-bit aligned: nothing to do.
        return &wg.state1, &wg.state2
    } else {
        state := (*[3]uint32)(unsafe.Pointer(&wg.state1))
        return (*uint64)(unsafe.Pointer(&state[1])), &state[0]
    }
}
```

Однако на системе с 4-байтовым выравниванием код возвращается к предыдущему решению Go 1.11. Метод `state()` преобразует `state1` и `state2` в трехэлементный массив `uint32` и переставляет элементы, чтобы сохранить правильное выравнивание `state` и `sema`.

Самое интересное происходит в Go 1.20, где они решили использовать `atomic.Uint64` для обработки состояния, полностью устранив необходимость в методе `state()`.

Go 1.20: `state atomic.Uint64`

В Go 1.19 была введена ключевая оптимизация, гарантирующая, что значения `uint64`, используемые в атомарных операциях, всегда выравниваются по 8-байтовым границам, даже на 32-битных архитектурах.

Чтобы добиться этого, Russ Cox ввел специальную структуру `atomic.Uint64`, которая, по сути, является оберткой для `uint64`:

```
type Uint64 struct {
    _ noCopy
    _ align64
    v uint64
}

// align64 may be added to structs that must be 64-bit aligned.
// This struct is recognized by a special case in the compiler
// and will not work if copied to any other package.
type align64 struct{}
```

Когда вы включаете `align64` в структуру, это сигнализирует компилятору Go, что вся структура должна быть выровнена по границе 8 байт в памяти.

`align64` - это обычная пустая структура. У нее нет никаких методов или особого поведения.

Само по себе поле `align64` не занимает места, но оно служит «маркером», который указывает компилятору Go, что со структурой нужно работать по-другому. Компилятор Go распознает `align64` и автоматически корректирует расположение памяти структуры, добавляя необходимые дополнения(padding), чтобы вся структура начиналась на границе 8 байт.

После этого структура WaitGroup становится намного проще:

```
type WaitGroup struct {
    noCopy noCopy

    state atomic.Uint64
    sema  uint32
}
```

Благодаря `atomic.Uint64` состояние гарантированно выравнивается на 8 байт, так что вам не придется беспокоиться о проблемах выравнивания, которые могут испортить атомарные операции над 64-битными переменными.

Как работает `sync.WaitGroup` внутри

Почему бы нам просто не разделить `counter` и `waiter` на две отдельные переменные `uint32` ?

Если бы мы использовали мьютекс для управления параллелизмом, нам не пришлось бы беспокоиться о том, на какой системе мы находимся - 32-битной или 64-битной, не было бы проблем с выравниванием.

Однако здесь есть компромисс. Использование блокировок, как и мьютексов, увеличивает накладные расходы, особенно если операции выполняются часто. Каждый раз, когда вы блокируете и разблокируете, вы добавляете немного задержки, которая может накапливаться, если речь идет о высокочастотных операциях.

С другой стороны, мы используем атомарные операции для безопасного изменения этой 64-битной переменной, без необходимости блокировать и разблокировать мьютекс.

Теперь давайте разберем, как работает каждый метод в WaitGroup, и поймем, как Go реализует алгоритм без блокировки, используя `state` .

`wg.Add(delta int)`

Когда мы передаем значение в метод `wg.Add(?)` , он соответствующим образом корректирует счетчик.

Если вы передаете положительную дельту, она прибавляется к счетчику. Интересно, что можно передать и отрицательную дельту, которая будет вычитаться из счетчика.

Как вы уже догадались, `wg.Done()` - это просто сокращение для `wg.Add(-1)` :

```
// Done decrements the [WaitGroup] counter by one.
func (wg *WaitGroup) Done() {
    wg.Add(-1)
}
```

Однако если отрицательная дельта приведет к тому, что счетчик упадет ниже нуля, программа запаникует. WaitGroup не проверяет счетчик перед обновлением, она сначала обновляет его, а затем проверяет. Это означает, что если счетчик станет отрицательным после вызова `wg.Add(?)`, он останется отрицательным до тех пор, пока не произойдет паника.

Так что, если вы поймаете эту панику и планируете повторно использовать WaitGroup, будьте осторожны.

```
func (wg *WaitGroup) Add(delta int) {
    ... // we excludes the race stuffs

    // Add the delta to the counter
    state := wg.state.Add(uint64(delta) << 32)

    // Extract the counter and waiter count from the state
    v := int32(state >> 32)
    w := uint32(state)
    ...

    if v < 0 {
        panic("sync: negative WaitGroup counter")
    }
    if w != 0 && delta > 0 && v == int32(delta) {
        panic("sync: WaitGroup misuse: Add called concurrently with Wait")
    }
    if v > 0 || w == 0 {
        return
    }
    if wg.state.Load() != state {
        panic("sync: WaitGroup misuse: Add called concurrently with Wait")
    }

    // Reset waiters count to 0.
    wg.state.Store(0)
    for ; w != 0; w-- {
        runtime_Semrelease(&wg.sema, false, 0)
    }
}
```

Обратите внимание, что вы можете увеличивать счетчик атомарно, вызывая `wg.state.Add(uint64(delta) << 32)`.

Вот что, возможно, не очень известно: **когда вы добавляете положительную дельту для обозначения начала новых задач, это должно происходить до вызова `wg.Wait()`**.

Таким образом, если вы повторно используете группу WaitGroup или хотите дождаться одной партии задач, сбросить ее, а затем дождаться другой партии, вам нужно убедиться, что все вызовы `wg.Wait` завершены, прежде чем начинать новые вызовы `wg.Add(positive)` для следующей партии задач.

Это позволяет избежать путаницы в том, какими задачами в данный момент управляет WaitGroup.

С другой стороны, вы можете вызвать `wg.Add(?)` с отрицательной дельтой в любое время.

`Wait()`

О `wg.Wait()` можно сказать не так уж много. По сути, она в цикле пытается увеличить количество waiter, используя атомарную операцию **CAS (Compare-And-Swap)**.

```
// Wait blocks until the [WaitGroup] counter is zero.
func (wg *WaitGroup) Wait() {
    ... // we excludes the race stuffs

    for {
        // Load the state of the WaitGroup
        state := wg.state.Load()
        v := int32(state >> 32)
        w := uint32(state)
        if v == 0 {
            // Counter is 0, no need to wait.
            ...

            return
        }
        // Increment waiters count.
        if wg.state.CompareAndSwap(state, state+1) {
            ...
```

```
            runtime_Semacquire(&wg.sema)
            if wg.state.Load() != 0 {
                panic("sync: WaitGroup is reused before previous Wait has returne
            }
            ...

            return
        }
    }
}
```

Если операция CAS завершилась неудачно, это означает, что другая горутина изменила состояние, возможно, счетчик достиг нуля или был увеличен/уменьшен. В этом случае `wg.Wait()` не может просто считать, что все осталось как было, поэтому она делает еще одну попытку.

Когда CAS завершается успешно, `wg.Wait()` увеличивает счетчик ожидания, а затем переводит горутину в спящий режим с помощью семафора.

В конце `wg.Add()` проверяет два условия: равен ли счетчик 0 и больше ли waiter 0. Если оба условия верны, она будит все ожидающие горутины и сбрасывает состояние в 0.

Вот, собственно, и вся история с sync.WaitGroup.

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Использование benchstat проекций в анализе Go бенчмарков

СЛЕДУЮЩАЯ »
Go sync.Mutex. Normal и Starvation Mode