Go sync.Cond - самый недооцененный механизм синхронизации

декабря 15, 2024 · 15 минут · German Gorelkin

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

Перевод/заметки Go sync.Pool and the Mechanics Behind It

`sync.Cond` — это примитив синхронизации, который, однако, не так часто используется, как его аналоги, такие как `sync.Mutex` или `sync.WaitGroup`. В большинстве проектов и даже в стандартных библиотеках его место занимают другие механизмы синхронизации.

Что такое `sync.Cond`?

Когда горутина ожидает определённого события, например, изменения общих данных, она может «блокироваться», то есть приостанавливать свою работу до тех пор, пока не получит разрешение на продолжение.

Проще всего это реализовать с помощью цикла, возможно, с добавлением `time.Sleep` для предотвращения перегрузке процессора от ожидания.

```
// wait until condition is true
for !condition {
}

// or
for !condition {
    time.Sleep(100 * time.Millisecond)
}
```

Это не очень эффективно, так как цикл продолжает работать в фоновом режиме, потребляя ресурсы процессора, даже если ничего не изменилось.

В этой ситуации на помощь приходит `sync.Cond` — более эффективный способ взаимодействия между горутинами. Если быть более точным, это так называемая *condition variable*.

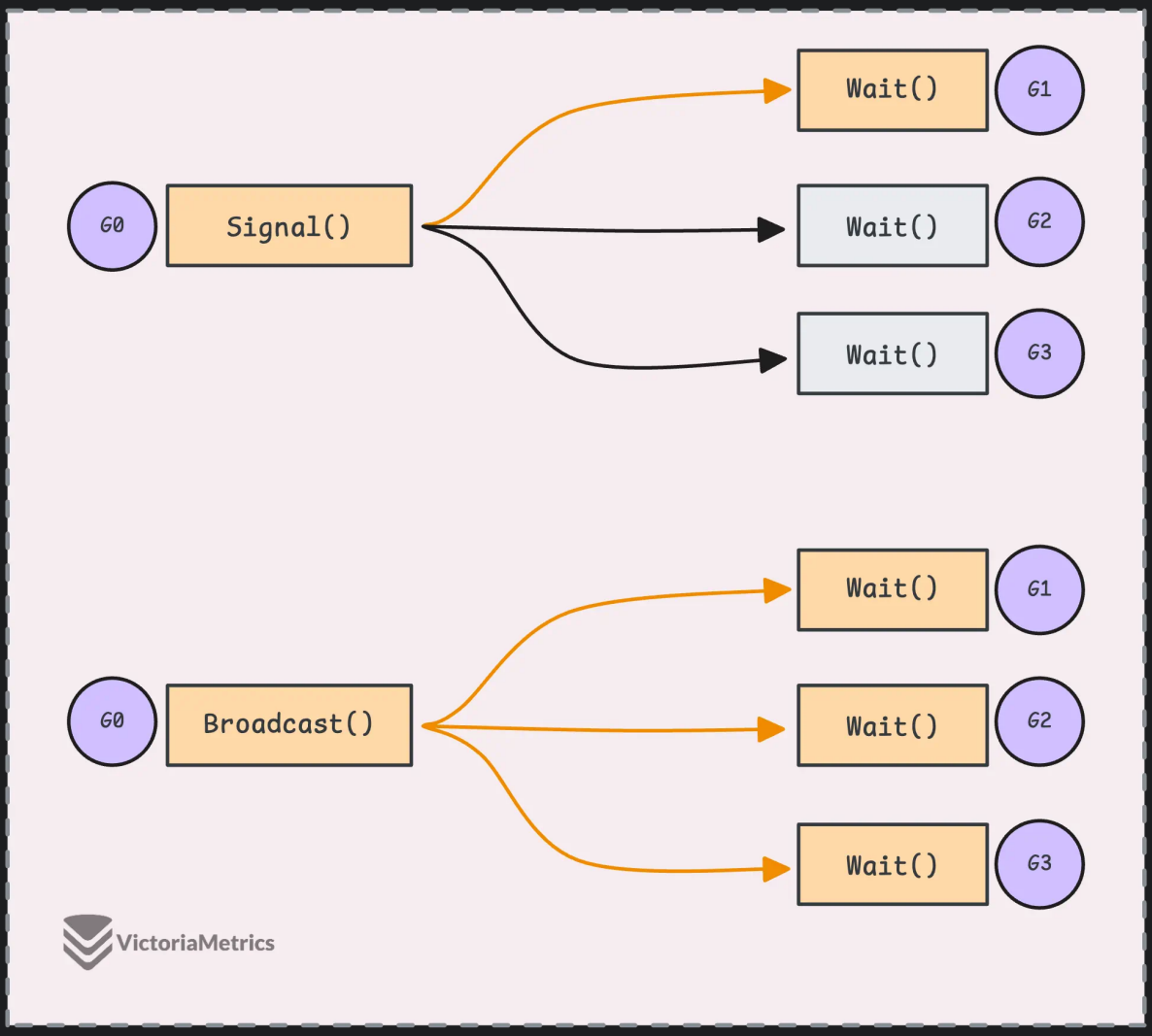
- Когда одна горутина ожидает определенного события (когда некоторое условие становится истинным), она может вызвать метод `Wait()`.
- Другая горутина, обнаружив, что условие может быть выполнено, может воспользоваться методами `Signal()` или `Broadcast()` для того, чтобы разбудить ожидающие горутины и дать им сигнал о необходимости

продолжить выполнение своих задач.

```
// Suspends the calling goroutine until the condition is met
func (c *Cond) Wait() {}

// Wakes up one waiting goroutine, if there is one
func (c *Cond) Signal() {}

// Wakes up all waiting goroutines
func (c *Cond) Broadcast() {}
```



Хорошо, давайте рассмотрим небольшой гипотетический пример. На этот раз мы обратимся к теме покемонов и представим, что ожидаем появления определённого существа и хотим сообщить об этом другим игрокам.

```
var pokemonList = []string{"Pikachu", "Charmander", "Squirtle", "Bulbasaur", "Jig
var cond = sync.NewCond(&sync.Mutex{})
var pokemon = ""
```

```
func main() {
    // Consumer
    go func() {
        cond.L.Lock()
        defer cond.L.Unlock()

        // waits until Pikachu appears
        for pokemon != "Pikachu" {
            cond.Wait()
        }
        println("Caught" + pokemon)
        pokemon = ""
    }()

    // Producer
    go func() {
        // Every 1ms, a random Pokémon appears
        for i := 0; i < 100; i++ {
            time.Sleep(time.Millisecond)

            cond.L.Lock()
            pokemon = pokemonList[rand.Intn(len(pokemonList))]
            cond.L.Unlock()

            cond.Signal()
        }
    }()

    time.Sleep(100 * time.Millisecond) // lazy wait
}

// Output:
// Caught Pikachu
```

В этом примере одна из горутин ожидает появления Пикачу, а другая, выступающая в роли производителя, случайным образом выбирает покемона из списка и оповещает первую о появлении нового.

Когда производитель отправляет сигнал, потребитель просыпается и проверяет, появился ли нужный покемон. Если да, то мы успешно ловим его, а если нет — засыпаем и ожидаем следующего.

Однако есть одна проблема: между отправкой сигнала производителем и реальным пробуждением потребителя может пройти некоторое время. За это

время покемон может измениться, поскольку горутина потребителя может проснуться с задержкой, превышающей 1 миллисекунду (что случается редко), или другая горутина может внести изменения в общий список покемонов.

Таким образом, `sync.Cond` как бы говорит: «Эй, что-то изменилось! Проснись и проверь это, но если ты опоздаешь, это может произойти снова».

В языке Go каналы обычно используются чаще, чем `sync.Cond`, поскольку они проще, более естественны и привычны для большинства разработчиков.

В этом примере вы можете легко отправить имя покемона через канал или использовать пустую структуру `struct{}` для передачи сигнала без отправки каких-либо данных. Однако наша проблема заключается не только в передаче сообщений по каналам, но и в работе с общим состоянием.

Наш пример довольно прост, но если к общей переменной pokemon обращаются несколько горутин, давайте посмотрим, что произойдет, если мы используем канал:

- Если мы используем канал для передачи имени покемона, нам все равно потребуется мьютекс для защиты общей переменной pokemon.
- Если мы используем канал только для передачи сигнала, мьютекс все равно необходим для управления доступом к общему состоянию.
- Если мы проверим наличие Пикачу в производителе, а затем отправим его по каналу, нам также понадобится мьютекс. Кроме того, мы нарушим принцип разделения задач, когда производитель берет на себя логику, которая на самом деле принадлежит потребителю.

Хорошо, но как быть с широковещательными сигналами?

Вы можете имитировать широковещательный сигнал для всех ожидающих горутин, использующих канал, просто закрыв его с помощью функции `close(ch)`. Когда вы закрываете канал, все горутины, получающие сигнал из этого канала, будут уведомлены. Однако обратите внимание, что закрытый канал не может быть использован повторно.

В чём же тогда преимущество `sync.Cond` ?

Теоретически, чтобы *condition variable*, например `sync.Cond` , могла передавать сигналы, его не обязательно привязывать к блокировке.

Вы можете предоставить пользователям возможность управлять своими собственными блокировками вне *cond*, что может показаться более гибким решением. Однако на самом деле это не является техническим ограничением, а скорее служит защитой от возможных ошибок.

Управление вручную может легко привести к ошибкам, потому что паттерн не совсем интуитивно понятен: вы должны разблокировать мьютекс перед вызовом `Wait()` , а затем снова заблокировать его, когда горутина проснется. Этот процесс может показаться неудобным и чреват ошибками, например, забыванием заблокировать или разблокировать в нужное время.

Как правило, горутины, вызывающие `cond.Wait()` , должны проверять некоторое общее состояние в цикле, например, так:

```
for !checkSomeSharedState() {
    cond.Wait()
}
```

Блокировка, встроенная в `sync.Cond` , помогает нам справиться с процессом блокировки/разблокировки, делая код чище и менее подверженным ошибкам.

Как этим пользоваться?

Если вы внимательно посмотрите на предыдущий пример, то заметите последовательный паттерн в потребителе: мы всегда блокируем мьютекс перед ожиданием (`.Wait()`) выполнения условия и разблокируем его после выполнения условия.

Кроме того, мы оборачиваем условие ожидания внутри цикла, вот краткое пояснение:

```
// Consumer
go func() {
```

```
    cond.L.Lock()
    defer cond.L.Unlock()

    // waits until Pikachu appears
    for pokemon != "Pikachu" {
        cond.Wait()
    }
    println("Caught" + pokemon)
}()
```

`Cond.Wait()`

Когда мы вызываем `Wait()` в `sync.Cond`, мы говорим текущей горутине, что нужно подождать, пока не будет выполнено некоторое условие.

Вот что происходит за кулисами:

1. Когда горутина ожидает выполнения некоторого условия, она добавляется в список других горутин, ожидающих того же условия. Все эти горутины блокируются и не могут продолжить свою работу, пока их не «разбудит» вызов `Signal()` или `Broadcast()`.
2. Важно отметить, что перед вызовом `Wait()` необходимо заблокировать мьютекс. Это связано с тем, что `Wait()` выполняет одну важную операцию — автоматически снимает блокировку (вызывает `Unlock()`) перед тем, как перевести горутину в спящий режим. Это позволяет другим горутинам захватить блокировку и выполнять свою работу, пока исходная горутина ожидает.
3. Когда ожидающая горутина просыпается (по сигналу `Signal()` или `Broadcast()`), она не сразу возобновляет свою работу. Сначала ей необходимо снова получить блокировку (`Lock()`).

Вот как работает `Wait()` под капотом:

```
func (c *Cond) Wait() {
    // Check if Cond has been copied
    c.checker.check()

    // Get the ticket number
    t := runtime_notifyListAdd(&c.notify)
```

```
    // Unlock the mutex
    c.L.Unlock()

    // Suspend the goroutine until being woken up
    runtime_notifyListWait(&c.notify, t)

    // Re-lock the mutex
    c.L.Lock()
}
```

Несмотря на простоту, в использовании `sync.Cond` есть несколько важных моментов:

1. Чтобы избежать копирования экземпляра `Cond`, предусмотрен специальный чекер. Если вы попытаетесь это сделать, то возникнет паника.
2. При вызове `cond.Wait()` мьютекс немедленно разблокируется. Поэтому перед вызовом `cond.Wait()` необходимо убедиться, что мьютекс уже заблокирован, иначе произойдет паника.
3. После пробуждения `cond.Wait()` снова блокирует мьютекс, что означает, что после завершения работы с общими данными вам нужно будет снова разблокировать его.
4. Большая часть функциональности `sync.Cond` реализована в runtime Go с помощью внутренней структуры данных `notifyList`, которая использует систему уведомлений на основе тикетов.

Из-за особенностей блокировки и разблокировки существует типичный паттерн, которому вы будете следовать при использовании `sync.Cond.Wait()`. Этот подход поможет избежать распространенных ошибок:

```
c.L.Lock()
for !condition() {
    c.Wait()
}
// ... make use of condition ...
c.L.Unlock()
```

Когда функция `Wait()` возвращается, мы не можем быть уверены, что ожидаемое условие сразу станет истинным. Пока наша горутина просыпается,

другие горутины могут изменить общее состояние, и условие, на которое мы ориентируемся, может перестать быть верным. Чтобы правильно учесть этот аспект, мы рекомендуем всегда использовать `Wait()` внутри цикла.

`Cond.Signal()` и `Cond.Broadcast()`

Метод `Signal()` используется для того, чтобы разбудить одну горутину, которая в данный момент ожидает cond.

- Если в данный момент нет ожидающих горутин, `Signal()` ничего не делает, в этом случае он практически не работает.
- Если есть ожидающие горутины, `Signal()` будит первую в очереди. Таким образом, если вы запустили кучу горутин, например от 0 до n, то первой будет разбужена 0-я горутина вызовом `Signal()` .

```
func main() {
    cond := sync.NewCond(&sync.Mutex{})
    for i := range 10 {
        go func(i int) {
            cond.L.Lock()
            defer cond.L.Unlock()
            cond.Wait()

            fmt.Println(i)
        }(i)

        time.Sleep(time.Millisecond)
    }

    time.Sleep(100 * time.Millisecond) // wait for goroutines to be ready
    cond.Signal()
    time.Sleep(100 * time.Millisecond) // wait for goroutines to be woken up
}

// Output:
// 0
```

`Signal()` используется для того, чтобы разбудить одну горутину и сообщить ей, что условие **возможно** выполнено. Вот как выглядит реализация `Signal()` :

```
func (c *Cond) Signal() {
```

```
    c.checker.check()
    runtime_notifyListNotifyOne(&c.notify)
}
```

Вы не обязаны блокировать мьютекс перед вызовом `Signal()`, но в целом это хорошая идея, особенно если вы изменяете общие данные, и доступ к ним осуществляется параллельно.

Как насчет `cond.Broadcast()`?

```
func (c *Cond) Broadcast() {
    c.checker.check()
    runtime_notifyListNotifyAll(&c.notify)
}
```

Когда вы вызываете функцию `Broadcast()`, она пробуждает все ожидающие горутины и удаляет их из очереди. Внутренняя логика здесь довольно проста и скрыта за функцией `runtime_notifyListNotifyAll()`.

```
func main() {
    cond := sync.NewCond(&sync.Mutex{})
    for i := range 10 {
        go func(i int) {
            cond.L.Lock()
            defer cond.L.Unlock()

            cond.Wait()
            fmt.Println(i)
        }(i)
    }

    time.Sleep(100 * time.Millisecond) // wait for goroutines to be ready
    cond.Broadcast()
    time.Sleep(100 * time.Millisecond) // wait for goroutines to be woken up
}

// Output:
// 8
// 6
// 3
// 2
// 4
// 5
// 1
// 0
```

```
// 9
// 7
```

На этот раз все горутины пробуждаются в течение 100 миллисекунд, но нет определенного порядка их пробуждения.

Когда вызывается `Broadcast()`, она помечает все ожидающие горутины как готовые к запуску, но они не запускаются сразу, а выбираются на основе базового алгоритма планировщика Go, который может быть немного непредсказуемым.

Как это работает внутри

Всегда полезно понять, чем обусловлен выбор дизайна и какие проблемы он пытается решить.

Copy checker

В пакете `sync` есть функция проверки копирования (`copyChecker`), которая определяет, был ли объект `Cond` скопирован после его первого использования. Это может произойти с помощью любого из общедоступных методов, таких как `Wait()`, `Signal()` или `Broadcast()`.

Если после первого использования `Cond` будет скопирован, программа выдаст ошибку: "*sync.Cond is copied*".

Вы могли видеть нечто подобное в `sync.WaitGroup` или `sync.Pool`, где для предотвращения копирования используется поле `noCopy`.

`copyChecker` - это `uintptr` который содержит адрес памяти:

- После первого использования `sync.Cond`, `copyChecker` сохраняет адрес памяти самого себя, указывая на объект `cond.copyChecker`.
- Если этот объект копируется, то адрес `&cond.copyChecker` изменится (поскольку новая копия находится в другом месте памяти), а `uintptr` останется прежним.

Проверка проста: сравнить адреса памяти. Если они различаются, бум, паника.

```
// copyChecker holds back pointer to itself to detect object copying.
type copyChecker uintptr

func (c *copyChecker) check() {
    if uintptr(*c) != uintptr(unsafe.Pointer(c)) &&
        !atomic.CompareAndSwapUintptr((*uintptr)(c), 0, uintptr(unsafe.Pointer(c)
        uintptr(*c) != uintptr(unsafe.Pointer(c)) {
        panic("sync.Cond is copied")
    }
}
```

Давайте разделим этот процесс на две основные проверки, поскольку первая и последняя проверки делают практически одно и то же.

Первая проверка, `uintptr(*c)!= uintptr(unsafe.Pointer(c))`, позволяет узнать, не изменился ли адрес в памяти. Если адрес изменился, то объект был скопирован. Однако есть один момент: если `copyChecker` используется впервые, его значение будет равно 0, так как он ещё не был инициализирован.

Вторая проверка, `!atomic.CompareAndSwapUintptr((*uintptr)(c), 0, uintptr(unsafe.Pointer(c)))`, осуществляется с помощью операции *Compare-And-Swap (CAS)*. Она служит одновременно для инициализации и проверки:

- Если CAS завершается успешно, это означает, что `copyChecker` был только что инициализирован, а значит, объект еще не был скопирован, и мы можем продолжать работу.
- Если же CAS не срабатывает, это свидетельствует о том, что `copyChecker` уже был инициализирован, и нам необходимо выполнить последнюю проверку (`uintptr(*c)!= uintptr(unsafe.Pointer(c)`), чтобы удостовериться, что объект не был скопирован.

Последняя проверка, `uintptr(*c)!= uintptr(unsafe.Pointer(c))`, аналогична первой и служит для подтверждения того, что объект не был скопирован после выполнения всех предыдущих операций.

Необходимость третьей проверки обусловлена тем, что первые две проверки не являются атомарными.

Если функция `copyChecker` вызывается впервые, она ещё не была инициализирована, и её значение равно нулю. В этом случае проверка не сможет обнаружить некорректное копирование, даже если объект не был скопирован, а просто не был инициализирован.

`notifyList` - Tick-based Notification List

Помимо механизмов блокировки и проверки копирования, одной из важных частей `sync.Cond` является `(notifyList)` .

```
type Cond struct {
    noCopy noCopy
    L Locker

    notify  notifyList

    checker copyChecker
}

type notifyList struct {
    wait   uint32
    notify uint32
    lock   uintptr
    head   unsafe.Pointer
    tail   unsafe.Pointer
}
```

Итак, `notifyList` в пакете `sync` и в пакете `runtime` отличаются друг от друга, но имеют одинаковое имя и размещение в памяти (по назначению). Чтобы действительно понять, как это работает, нам нужно посмотреть на версию в пакете `runtime` :

```
type notifyList struct {
    wait atomic.Uint32
    notify uint32

    lock mutex

    head *sudog
    tail *sudog
}
```

Если вы посмотрите на `head` и `tail`, то, вероятно, догадаетесь, что это какой-то связанный список, и будете правы. Это связный список `sudog` (сокращение от *pseudo-goroutine*), который представляет собой горутину, ожидающую события синхронизации, например, ожидание получения или отправки данных по каналу или ожидание cond.

`head` и `tail` — это указатели на первую и последнюю горутины в списке. В то же время поля `wait` и `notify` представляют собой своего рода «*ticket*», который постоянно увеличивается и отражает позицию в очереди ожидающих горутин.

- `wait`: Это число обозначает **следующий ticket**, который будет выдан ожидающей горутине.
- `notify`: Отслеживает номер следующего билета, который должен быть уведомлен или разбужен.

`notifyListAdd()`

Когда горутина собирается ждать уведомления, она вызывает `notifyListAdd()`, чтобы сначала получить свой «*ticket*».

```
func (c *Cond) Wait() {
    c.checker.check()
    // Get the ticket number
    t := runtime_notifyListAdd(&c.notify)
    c.L.Unlock()
    // Add the goroutine to the list and suspend it
    runtime_notifyListWait(&c.notify, t)
    c.L.Lock()
}

func notifyListAdd(l *notifyList) uint32 {
    return l.wait.Add(1) - 1
}
```

Процесс распределения тикетов осуществляется с помощью атомарного счетчика. Когда какая-либо горутина вызывает метод `notifyListAdd()`, этот счетчик увеличивается, и задаче передается следующий доступный тикет.

Каждая горутина получает свой уникальный номер тикета, и этот процесс происходит **без блокировок**. Это означает, что несколько горутин могут запрашивать тикеты одновременно, не мешая друг другу.

Например, если текущий счетчик равен 5, то горутина, которая вызовет `notifyListAdd()` следующей, получит тикет с номером 5, а счетчик ожидания увеличится до 6, готовый к приему следующей горутины в очереди. Поле `wait` всегда указывает на следующий доступный тикет, который будет выдан.

Но здесь возникает небольшая сложность.

Поскольку несколько горутин могут получить билет одновременно, существует небольшой временной промежуток между тем, как они вызывают метод `notifyListAdd()`, и тем, как они фактически начинают выполнять `notifyListWait()`. Их порядок не гарантируется, даже если номера тикетов выдаются последовательно. Например, вместо ожидаемого порядка `1, 2, 3`, это может быть `3, 2, 1` или `2, 1, 3`, в зависимости от времени.

Получив свой тикет, горутина должна «дождаться» уведомления о своей очереди. Это происходит, когда горутина вызывает команду `notifyListWait(t)`, где `t` - номер тикета, который она только что получила.

```
func notifyListWait(l *notifyList, t uint32) {
    lockWithRank(&l.lock, lockRankNotifyList)

    // Return right away if this ticket has already been notified.
    if less(t, l.notify) {
        unlock(&l.lock)
        return
    }

    // Enqueue itself.
    s := acquireSudog()
    ...

    if l.tail == nil {
        l.head = s
    } else {
        l.tail.next = s
    }
    l.tail = s
```

```
    goparkunlock(&l.lock, waitReasonSyncCondWait, traceBlockCondWait, 3)
    ...

    releaseSudog(s)
}
```

Перед тем как сделать что-либо еще, горутина проводит проверку, не было ли уже получено уведомление о её тикете.

Она сравнивает свой собственный тикет (обозначенный как `t`) с текущим номером уведомления. Если номер уведомления уже превысил тикет горутины, то ей не требуется ждать — она может сразу обратиться к общему ресурсу и приступить к выполнению своей работы.

Эта быстрая проверка оказывается критически важной, особенно когда мы углубляемся в использование функций `Signal()` и `Broadcast()`.

Однако, если тикет этой горутины ещё не был уведомлён, она добавляет себя в список ожидающих и переходит в состояние ожидания(«парковка»), до получения соответствующего уведомления.

`notifyListNotifyOne()`

Когда приходит время оповещать ожидающие горутины, система начинает с самого низкого номера тикета, который ещё не был обработан. Этот процесс контролируется с помощью `l.notify`.

```
func notifyListNotifyOne(l *notifyList) {
    // Fast path: If there are no new waiters, do nothing.
    if l.wait.Load() == atomic.Load(&l.notify) {
        return
    }

    lockWithRank(&l.lock, lockRankNotifyList)

    // Re-check under the lock to make sure there's something to do.
    t := l.notify
    if t == l.wait.Load() {
        unlock(&l.lock)
        return
    }
```

```
    // Move to the next ticket to notify.
    atomic.Store(&l.notify, t+1)

    // Find the goroutine with the matching ticket in the list.
    for p, s := (*sudog)(nil), l.head; s != nil; p, s = s, s.next {
        if s.ticket == t {
            // Found the goroutine with the ticket.
            n := s.next
            if p != nil {
                p.next = n
            } else {
                l.head = n
            }
            if n == nil {
                l.tail = p
            }
            unlock(&l.lock)
            s.next = nil
            readyWithTime(s, 4) // Mark the goroutine as ready.
            return
        }
    }
    unlock(&l.lock)
}
```

Помните, мы говорили о том, что порядок билетов не гарантирован?

У вас могут быть горутины с номерами билетов `2, 1, 3`, но количество уведомлений всегда увеличивается последовательно. Поэтому, когда система готова запустить новую горутину, она просматривает список в поисках горутины со следующим по порядку билетом (первым в очереди). Как только она ее находит, она удаляет эту горутину из списка и помечает ее как готовую к запуску.

Но вот дальше становится интересно: иногда возникают проблемы со временем. Предположим, что горутина взяла билет, но к моменту выполнения этой функции она еще не была добавлена в список ожидающих горутин.

Что произойдет в таком случае? Например, последовательность действий может быть такой: `notifyListAdd()` -> `notifyListNotifyOne()` -> `notifyListWait()`.

В этом случае функция проверит список, но не обнаружит горутину с соответствующим тикетом. Не волнуйтесь, функция `notifyListWait()` исправит ситуацию, когда горутина в конце концов вызовет ее.

Помните ту важную проверку, о которой я говорил ранее? Ту, что в функции `notifyListWait()` : `if less(t, l.notify) { ... }` ?

Эта проверка важна, потому что она позволяет горутине, имеющей номер тикета меньше, чем текущее значение `l.notify` , понять: «Эй, моя очередь уже прошла, я могу уйти сейчас». В этом случае горутина пропускает ожидание и немедленно приступает к работе с общим ресурсом.

Таким образом, даже если горутина еще не вошла в список, она все равно может получить уведомление, если у нее есть действительный тикет. Именно это делает конструкцию такой удобной: каждая горутина может сразу же взять свой тикет, не дожидаясь других или своей очереди на добавление в список. Благодаря этому все движется без лишних блокировок.

`notifyListNotifyAll()`

Теперь поговорим о последней части, `Broadcast()` или `notifyListNotifyAll()` . Эта функция намного проще, чем `notifyListNotifyOne()` :

```
func notifyListNotifyAll(l *notifyList) {
    // Fast path: If there are no new waiters, do nothing.
    if l.wait.Load() == atomic.Load(&l.notify) {
        return
    }

    lockWithRank(&l.lock, lockRankNotifyList)
    s := l.head
    l.head = nil
    l.tail = nil

    atomic.Store(&l.notify, l.wait.Load())
    unlock(&l.lock)

    // Ready all waiters in the list.
    for s != nil {
        next := s.next
        s.next = nil
        readyWithTime(s, 4)
        s = next
    }
}
```

Код довольно прост, и я думаю, что вы уже поняли его суть. По сути, `Broadcast()` просматривает весь список ожидающих горутин, отмечает все из них как готовые и очищает список.

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Слабые Указатели в Go

СЛЕДУЮЩАЯ »
Go Production Performance Gotcha - GOMAXPROCS