Примитивы Синхронизации в Go

февраля 9, 2020 · 5 минут · German Gorelkin

Продолжаем 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

Пакет `sync` содержит примитивы, которые наиболее полезны для низкоуровневой синхронизации доступа к памяти.

WaitGroup

> **WaitGroup** — это отличный способ дождаться завершения набора одновременных операций.

Запустим несколько goroutine и дождемся завершения их работы:

```
var wg sync.WaitGroup

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

   fmt.Println("1st goroutine sleeping...")
   time.Sleep(100 * time.Millisecond)
}()

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

   fmt.Println("2nd goroutine sleeping...")
   time.Sleep(200 * time.Millisecond)
}()

wg.Wait()
fmt.Println("All goroutines complete.")
```

У нас нет гарантий когда будут запущены наши goroutine. Возможна ситуация когда при вызове `Wait` еще не будет ни одной запущенной goroutine. По этому важно вызвать `Add` за пределами процедур, которые они помогают отслеживать.

Пример *неопределенного* поведения:

```
var wg sync.WaitGroup
go func() {
  wg.Add(1)
  defer wg.Done()
  fmt.Println("1st goroutine sleeping...")
  time.Sleep(1)
 }()
wg.Wait()
```

```
fmt.Println("All goroutines complete.")
```

О WaitGroup можно думать как о concurrent-safe счетчике.

Вызовы `Add` увеличивает счетчик на переданное число, а вызовы `Done` уменьшают счетчик на единицу. `Wait` блокируется пока счетчик не станет равным нулю.

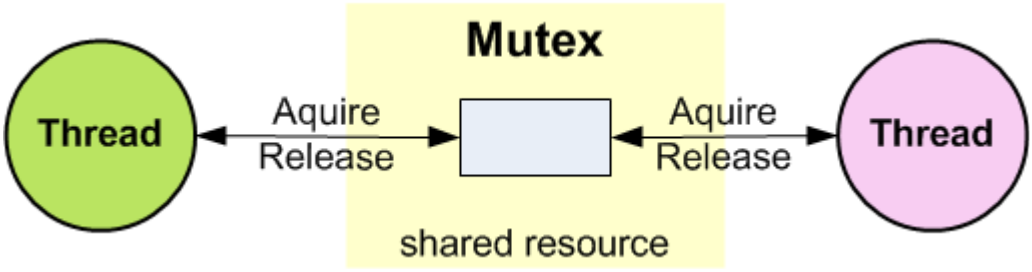
Обычно `Add` вызывают как можно ближе к goroutine. Но иногда удобно используют `Add` для отслеживания группы goroutine одновременно.

Например в таких циклах:

```
hello := func(wg *sync.WaitGroup, id int) {
   defer wg.Done()
   fmt.Printf("Hello from %v!\n", id)
}

const numGreeters = 5
var wg sync.WaitGroup

wg.Add(numGreeters)
for i := 0; i < numGreeters; i++ {
   go hello(&wg, i+1)
}

wg.Wait()
```

go-patterns

Mutex and RWMutex



> ***Mutex** означает **mutual exclusion(взаимное исключение)** и является способом защиты **critical section(критическая секция)** вашей программы.*

> **Критическая секция** — это область вашей программы, которая требует эксклюзивного доступа к общему ресурсу. При нахождении в критической секции двух (или более) потоков возникает состояние **race(гонки)**. Так же возможны проблемы **взаимной блокировки(deadlock)**.

Mutex обеспечивает безопасный доступ к общим ресурсам.

Простой пример счетчика:

```
type counter struct{
   count int
}
func (c *counter) Increment() {
   c.count++
}
func (c *counter) Decrement() {
   c.count--
}
```

Напишем тест, который будет в разных goroutine увеличивать или уменьшать общее значение:

```
c := new(counter)

var wg sync.WaitGroup
numLoop := 1000

wg.Add(numLoop)
for i := 0; i < numLoop; i++ {
   go func() {
      defer wg.Done()
      c.Increment()
   }()
}

wg.Add(numLoop)
for i := 0; i < numLoop; i++ {
   go func() {
      defer wg.Done()
      c.Decrement()
   }()
}

wg.Wait()

expected := 0
```

```
assert.Equal(t, expected, c.count)
```

Результат:

```
expected: 0
actual:   52
```

Используем **Mutex** для синхронизации доступа:

```
type counter struct{
   sync.Mutex
   count int
}
func (c *counter) Increment() {
   c.Lock()
   defer c.Unlock()
   c.count++
}
func (c *counter) Decrement() {
   c.Lock()
   defer c.Unlock()
   c.count--
}
```

Мы вызываем `Unlock` в `defer` . Это очень распространенная идиома при использовании **Mutex**, чтобы гарантировать, что вызов всегда происходит, даже при панике. Несоблюдение этого требования может привести к **deadlock** вашей программы. Хотя `defer` и несет небольшие затраты.

Критическая секция названа так, потому что она отражает узкое место в вашей программе. Вход в критическую секцию и выход из нее обходится довольно дорого, поэтому обычно люди пытаются минимизировать время, проведенное в критических секциях.

Возможно не все процессы будут читать и записывать в общую память. В этом случае вы можете воспользоваться мьютексом другого типа.

RWMutex

> **RWMutex** концептуально то же самое, что и **Mutex**: он защищает доступ к памяти. Тем не менее, RWMutex дает вам немного больше контроля над

> памятью. Вы можете запросить блокировку для чтения, и в этом случае вам будет предоставлен доступ, если блокировка не удерживается для записи.

Это означает, что произвольное число читателей может удерживать блокировку читателя, пока ничто другое не удерживает блокировку писателя.

Посмотрим как это работает:

```
func (c *counter) CountV1() int {
   c.Lock()
   defer c.Unlock()
   return c.count
}
func (c *counter) CountV2() int {
   c.RLock()
   defer c.RUnlock()
   return c.count
}
```

`CountV2` не блокирует `count` если не было блокировок на запись.

Немного бенчмарков:

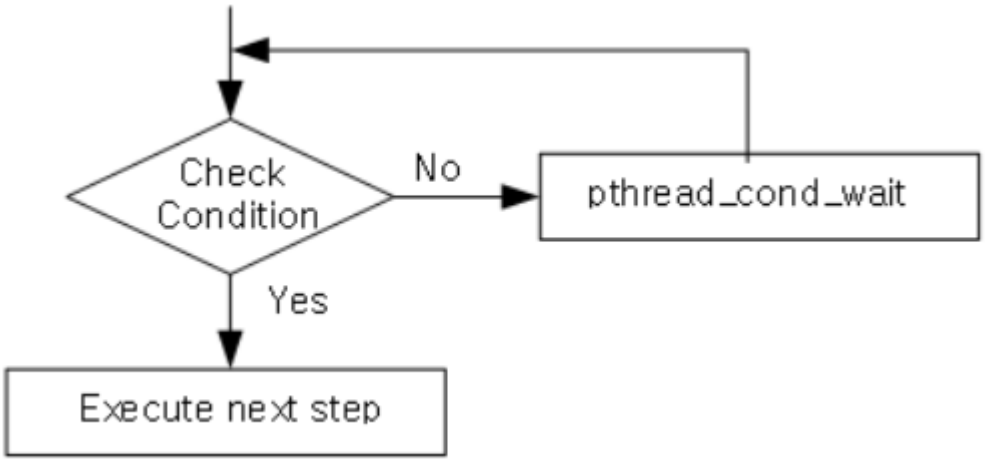
```
func BenchmarkCountV1(b *testing.B) {
   c := new(counter)
   var wg sync.WaitGroup
   for i := 0; i < b.N; i++ {
      for j := 0; j < 1000; j++ {
         wg.Add(1)
         go func() {
            defer wg.Done()
            c.CountV1()
         }()
      }
      wg.Wait()
   }
}

func BenchmarkCountV2(b *testing.B) {
   c := new(counter)
   var wg sync.WaitGroup
   for i := 0; i < b.N; i++ {
      for j := 0; j < 1000; j++ {
         wg.Add(1)
         go func() {
```

```
            defer wg.Done()
            c.CountV2()
         }()
      }
      wg.Wait()
   }
}
```

```
BenchmarkCountV1-8          2132            501896 ns/op
BenchmarkCountV2-8          3358            306254 ns/op
```

go-patterns

Cond



> **Условная переменная(condition variable)** — *примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания.*

Сигнал не несет никакой информации, кроме факта, что произошло какое-то событие. Очень часто мы хотим подождать один из этих сигналов, прежде чем продолжить выполнение. Один из наивных подходов состоит в использовании бесконечного цикла:

```
for conditionTrue() == false {
   time.Sleep(1 * time.Millisecond)
}
```

Но это довольно неэффективно, и вам нужно выяснить, как долго спать: слишком долго, и вы искусственно снижаете производительность; слишком мало, и вы отнимаете слишком много процессорного времени. Было бы лучше, если бы у процесса был какой-то способ эффективно спать, пока ему не будет дан сигнал проснуться и проверить его состояние.

Такие задачи могут решать каналы или вариации **паттерна PubSub(Publisher-Subscriber)**.

Но если у вас низкоуровневая библиотека, где необходим более производительный код, тогда можно использовать тип `sync.Cond` .

Пример

Предположим у нас есть некоторый общий ресурс в системе. Одна группа процессов может изменять его состояния, а другая группа должна реагировать на эти изменения.

```
type message struct {
   cond *sync.Cond
   msg  string
}

func main() {
   msg := message{
      cond: sync.NewCond(&sync.Mutex{}),
   }

   // 1
   for i := 1; i <= 3; i++ {
      go func(num int) {
         for {
            msg.cond.L.Lock()
            msg.cond.Wait()
            fmt.Printf("hello, i am worker%d. text:%s\n", num, msg.msg)
            msg.cond.L.Unlock()
         }
      }(i)
   }

   // 2
   scanner := bufio.NewScanner(os.Stdin)
   fmt.Print("Enter text: ")
```

```
    for scanner.Scan() {
       msg.cond.L.Lock()
       msg.msg = scanner.Text()
       msg.cond.L.Unlock()

       msg.cond.Broadcast()
    }

 }
```

Мы запустили 3 goroutine которые ждут сигнала. Обратите внимание, что вызов `Wait` не просто блокирует, он приостанавливает текущую процедуру, позволяя другим процедурам запускаться.

При входе `Wait` вызывается Unlock в Locker переменной Cond, а при выходе из `Wait` вызывается Lock в Locker переменной Cond. К этому нужно немного привыкнуть.

Во второй части мы читаем ввод из консоли и отправляем сигнал об изменении состояния.

`Broadcast` отправляет сигнал всем ожидающим goroutine. А метод `Signal` находит goroutine, которая ждала дольше всего и будет ее.

Подробней можно почитать в **Go sync.Cond, самый недооцененный механизм синхронизации**

proposal: Go 2: sync: remove the Cond type — дискуссия о необходимости Cond в sync.

concurrency/examples/sync/cond

Дополнительная информация

1. Concurrency in Go. by Katherine Cox-Buday
2. https://golang.org/pkg/sync

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Безопасная Работа с Каналами в Go

СЛЕДУЮЩАЯ »
Deadlocks, Livelocks и Starvation