### German Gorelkin Blog 🌣

Dashboard & Categories Tags Archives Donate!

Главная » Posts

## **Goroutine Leaks**

апреля 23, 2020 · 3 минуты · 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

**Горутины(goroutines) легко и быстро создавать.** К тому же они достаточно легковесные.

Рантайм мультиплексирует горутины на потоки операционной системы,

занимается их запуском и переключением. Поэтому нам не приходится беспокоиться об этом уровне абстракции. Это одно из главных преимуществ Golang.

А вот управлять горутинами не так легко. Нет механизма завершения горутин из вне. Она или закончит свою работу или в ней произойдет ошибка.

Программа может создать огромное кол-во горутин которые не смогут завершить свою работу. У *garbage collector* не будет возможности освободить занятую ими память. И это может станет проблемой.

# Paccмотрим несколько примером Goroutine Leaks

Особенность nil канала в том, что он блокирует отправителя. Подробней можно прочитать в Безопасная работа с каналами в Go

```
doWork := func(strings <-chan string) <-chan interface{} {
   completed := make(chan interface{})

   go func() {
      defer fmt.Println("doWork exited.")
      defer close(completed)

      for s := range strings {
         fmt.Println(s)
      }
    }()
    return completed
}

doWork(nil)

fmt.Println("Done.")</pre>
```

Горутина останется запущенной и будет занимать память до завершения всего процесса.

Таких горутин может быть очень много:

```
chans :=make([]chan string, 1_000_000)
for _, ch := range chans{
    // doWork(nil)
    doWork(ch)
}
```

#### Возможна обратная ситуация

Горутина блокируется при записи в канал:

```
newZeroStream := func() <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)

        for {
            out <- 0
        }
      }()
      return out
}

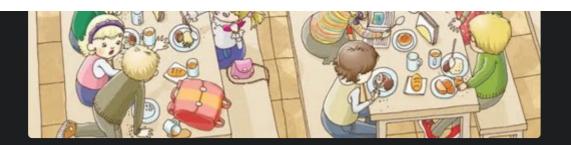
ch := newZeroStream()
fmt.Println("3 zeros:")
for i := 1; i <= 3; i++ {
      fmt.Printf("%d: %d\n", i, <-ch)
}</pre>
```

Мы прочитали нужные нам три нолика и пошли дальше. A newZeroStream осталась заблокирована навсегда.

Точно так же горутины могут утекать в ожидании данных из сети, от пользователя, от других подсистем и прочие.

# Родительский контроль





Для решения таких проблем нам нужна связь между родительской программой и дочерними горутинами.

Родительская горутина должна иметь возможность сигнализировать об отмене своим подопечным. А дочерние элементы должны обрабатывать такие сигналы и корректно завершаться.

По соглашению, *сигналом* обычно является канал только для чтения с именем done. Родительская программа передает этот канал дочерней программе, а затем закрывает канал, когда нужно отменить дочернюю программу.

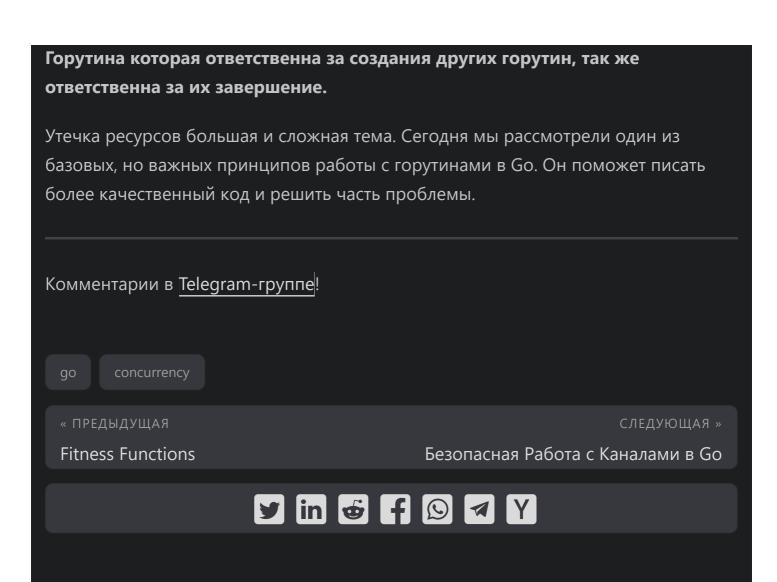
```
terminared := make(chan interface{})
   defer close(terminared)
      case s := <-strings:</pre>
         fmt.Println(s)
return terminared
close(done)
```

```
<-terminated //4
fmt.Println("Done.")</pre>
```

- 1. Передаем первым параметром сигнальный канал done
- 2. Используем for-select с проверкой на поступления сигнала от родительской горутины. Если сигнал поступил, то завершаем горутину.
- 3. Таймаут. Отдельная горутина которая отправит сигнал о завершении через 1 сек, если dowork еще будет работать.
- 4. Синхронизация с dowork

## **Context**

В Go 1.7 появился пакет солтехт . **Context** решает туже задачу что и паттерн done и даже больше. На данный момент является стандартным решение.



© 2025 German Gorelkin Blog Powered by Hugo & PaperMod