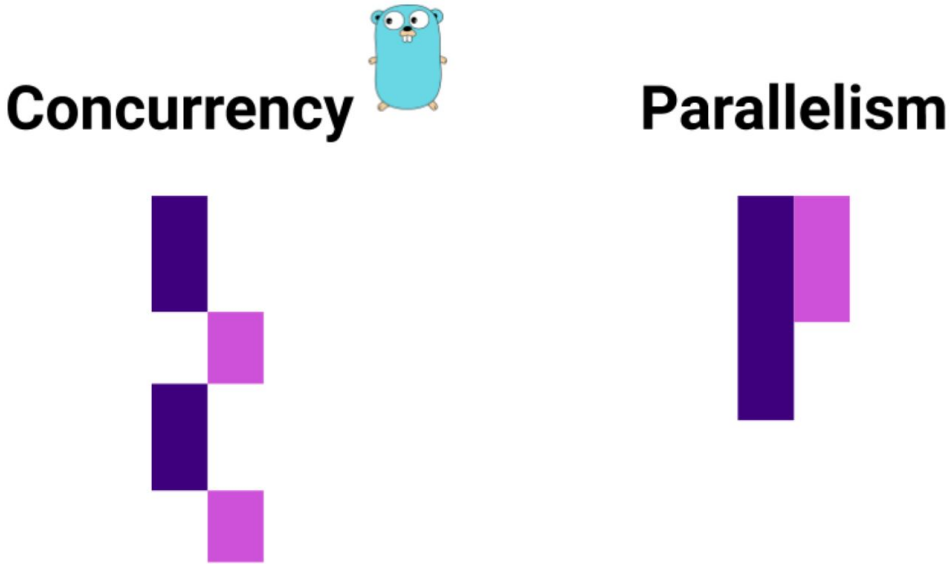
Программный Регион

Golang
GoRoutine, sync

Конкурентность и параллелизм



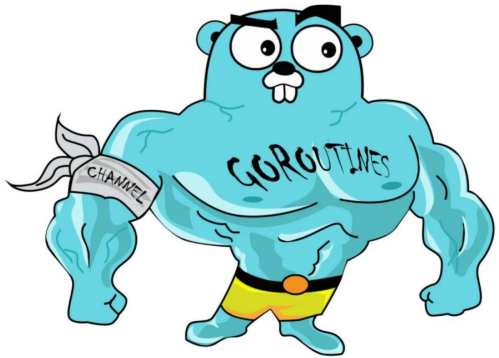
Конкурентность и параллелизм

Конкурентность - это возможность разбивать алгоритм или компьютерную программу на отдельные блоки, которые могут выполняться независимо и коммуницировать между собой.

Конкурентность - это в первую очередь дизайн подход к проектированию ПО, в то время как параллелизм - это способ его выполнения.

GoRoutine

Горутины реализуют в Golang обёрточный функционал потоков, а управляются они скорее из среды выполнения Go, нежели из операционной системы.



GoRoutine

В Go мы можем увеличить количество ядер простой строчкой кода. Приложению будет дана команда перейти на несколько ядер:

runtime.GOMAXPROCS(4)

GoRoutine

GoRoutine - функция, выполняющаяся конкурентно с другими горутинами в том же адресном пространстве.

Приравнивают к легковесным потокам.

Маленькие накладные расходы на запуск относительно потока.

Главная GoRoutine - func main().

Объявление - go anyFunc(args).

GoRoutine

Когда использовать ?

- Если нужна асинхронность. Например когда мы работаем с сетью, диском, базой данных и т.п.
- Если время выполнения функции достаточно велико и можно получить выигрыш, нагрузив другие ядра.

GoRoutine

```go
func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    f("direct")

    go f("goroutine1")
    go f("goroutine2")

    time.Sleep(time.Second)
    fmt.Println("done")
}
```

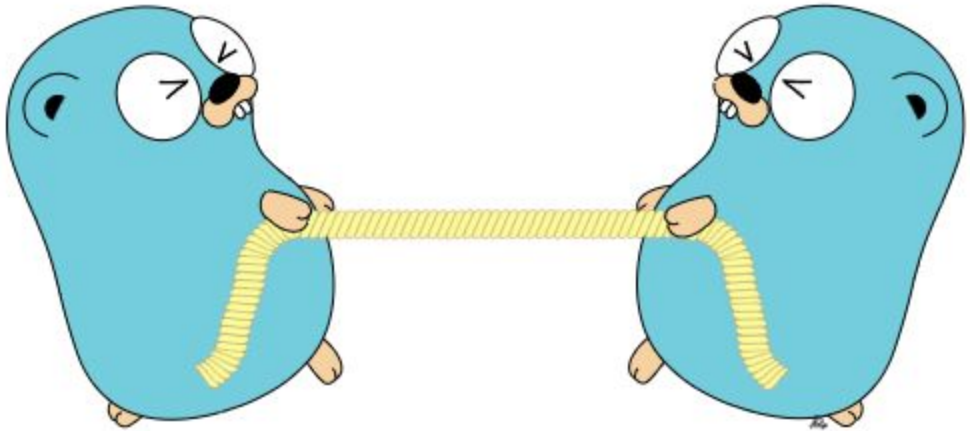
WaitGroup

WaitGroup ожидает завершения коллекции goroutine. Основная goroutine вызывает Add, чтобы установить количество goroutine, которых необходимо ожидать. Затем каждая из goroutine запускается и вызывает Done, когда завершается. В то же время, Wait может быть использован, чтобы блокировать, пока все goroutine не завершились.

WaitGroup

```go
func worker(id int, wg *sync.WaitGroup) {
    fmt.Printf("Worker %d starting\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)

    wg.Done()
}

func main() {

    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go worker(i, &wg)
    }

    wg.Wait()
}
```

Data race



Data race

Гонка данных происходит, когда две goroutines одновременно обращаются к одной и той же переменной, и, по крайней мере, одно из обращений является записью.

Единственный способ избежать гонки данных - это синхронизировать доступ ко всем изменяемым данным, которые совместно используются потоками.

Mutex

Мьютексы позволяют разграничить доступ к некоторым общим ресурсам, гарантируя, что только одна горутина имеет к ним доступ в определенный момент времени. И пока одна горутина не освободит общий ресурс, другая горутина не может с ним работать.

Mutex

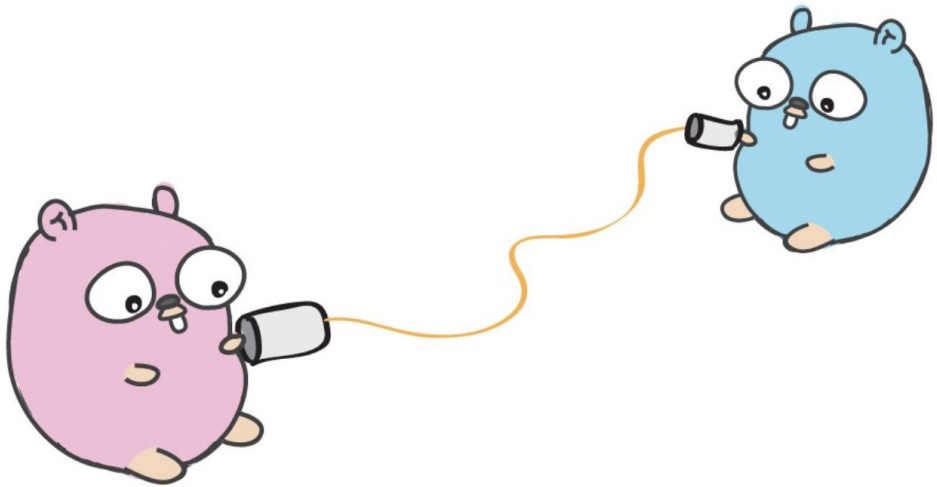
Мьютекс представляет тип sync.Mutex.

- Lock()
- Unlock()
- Не всегда выгодно использование Lock() и deffer Unlock().

Mutex

```
6   var x  = 0
7   func increment(wg *sync.WaitGroup, m *sync.Mutex) {
8       m.Lock()
9       x++
10      m.Unlock()
11      wg.Done()
12  }
13  func main() {
14      var w sync.WaitGroup
15      var m sync.Mutex
16      for i := 0; i < 1000; i++ {
17          w.Add(1)
18          go increment(&w, &m)
19      }
20      w.Wait()
21      fmt.Println("final value of x", x)
22  }
```

Channel



Channel

Канал — это объект связи, с помощью которого горутины обмениваются данными. Технически это конвейер (или труба), откуда можно считывать или помещать данные. То есть одна горутина может отправить данные в канал, а другая — считать помещенные в этот канал данные.

Channel

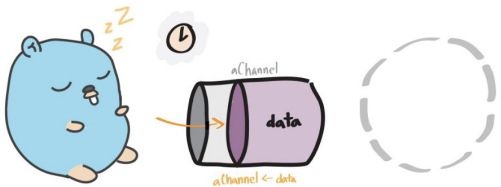
- Создание канала требует инициализации. c := make(chan int)
- Доступ по ссылке. При передаче аргументом не требует разыменования.
- Запись в канал. c <- data
- Чтение из канала. <- c
- Сохранение значения канала в переменную. data := <- c
- Буферизированные каналы

Channel

```
5   func greet(c chan string) {
6       fmt.Println("Hello " + <-c + "!")
7   }
8
9   func main() {
10      fmt.Println("main() started")
11      c := make(chan string)
12
13      go greet(c)
14
15      c <- "John"
16      fmt.Println("main() stopped")
17  }
```

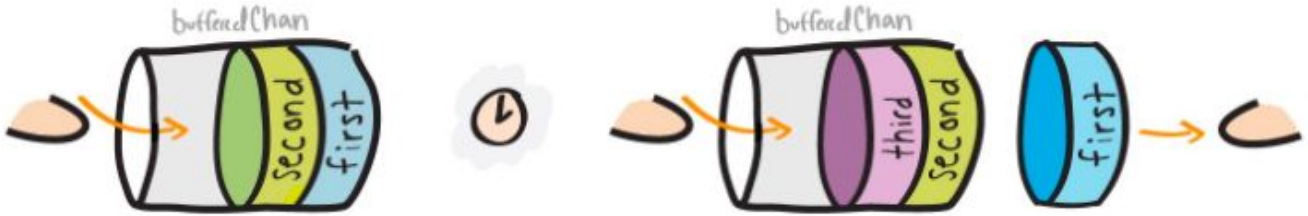
Channel

Запись и чтение данных являются блокируемыми операциями. Когда вы помещаете данные в канал, горутина блокируется до тех пор, пока данные не будут считаны другой горутиной из этого канала. По этой причине отпадает необходимость писать блокировки для взаимодействия горутин.



Channel

Отправка в буферизованный канал блокируется, только если буфер полон. Получение блокируется, когда буфер пуст.



Select

Select похож на switch без аргументов, но он может использоваться только для операций с каналами

Оператор select также является блокируемым, за исключением использования default

Если все блоки case являются блокируемыми, тогда select будет ждать до момента, пока один из блоков case разблокируется и будет выполнен

Неблокирующе чтение из канала

```
2   myChan := make(chan string)
3
4   go func(){
5    myChan <- "Message!"
6   }()
7
8   select {
9    case msg := <- myChan:
10    fmt.Println(msg)
11   default:
12    fmt.Println("No Msg")
13  }
14  time.Sleep(time.Second * 1)
15  select {
16   case msg := <- myChan:
17    fmt.Println(msg)
18   default:
19    fmt.Println("No Msg")
20  }
```

Deadlock

Deadlock возникает, когда группа goroutines ждет друг друга, и ни одна из них не может продолжить.

Причины этому:

- либо потому, что goroutine ждет канал (channel)
- либо потому, что goroutine ожидает одну из блокировок (lock) в пакете sync.

Расходы на синхронизацию GoRoutine