Peter the Great
Saint-Petersburg Polytechnic University

Параллельное программирование в Go

Исполнитель: Квентин Тарантино

06.07.1995

POLYTECH

Горутины

- Горутины (goroutines) — это легковесные потоки выполнения, создаваемые в Go для работы с параллельными и конкурентными операциями.

```go
func main() {
    go goroutine()                  // Начало горутины
    time.Sleep(4 * time.Second) // Ожидание выполнения функций
} // Здесь все горутины останавливаются
func goroutine() {
    time.Sleep(3 * time.Second)
    fmt.Println("Hello")
}
```

Реализация

- Горутины имеют небольшой вес и стоят немногим больше, чем выделение места в стеке. Место в куче выделяется и освобождается по мере необходимости.
- Если горутина блокируется, планировщик Go переключает контекст на другую горутину, продолжая выполнение.

Создание нескольких горутин

- При каждом использовании ключевого слова go начинается новая горутина. Все появления горутин должны запускаться одновременно.

```
func main() {
    for i := 0; i < 5; i++ {
        go goroutine(i)
    }
    time.Sleep(4 * time.Second)
}

func goroutine(id int) {
    time.Sleep(3 * time.Second)
    fmt.Println("Hello ", id)
}
```

Каналы

- Каналы (channels) — это механизм обмена данными между горутинами, который обеспечивает безопасность и предсказуемость. Они предотвращают состояние гонки (race condition) и помогают в координации параллельных задач.

```
// небуферизованный канал int-ов
ic := make(chan int)
// буферизованный канал на 10 строк
sc := make(chan string, 10)
```

Операции с каналами

- Отправка данных
- Получение данных
- Закрытие канала

```
value := <-ic
ic <- value
close(ic)
```

Буферизованные и небуферизованные каналы

- Если пропускная способность канала равна нулю или отсутствует, канал не буферизуется и отправитель блокируется до тех пор, пока получатель не получит значение.
- Если канал имеет буфер, отправитель блокируется только до тех пор, пока значение не будет скопировано в буфер. Если буфер заполнен, ждем пока какой-либо получатель не получит значение.

Пример использования канала

```go
func main() {
    c := make(chan int) // Делает канал для связи
    for i := 0; i < 5; i++ {
        go goroutine(i, c)
    }
    for i := 0; i < 5; i++ {
        goroutineID := <-c // Получает значение от канала
        fmt.Println("goroutine ", goroutineID, " has finished sleeping")
    }
}

func goroutine(id int, c chan int) { // Объявляет канал как аргумент
    time.Sleep(3 * time.Second)
    fmt.Println("Hello ", id)
    c <- id // Отправляет значение обратно к main
}
```

Производитель-потребитель

```go
func producer(ch chan<- int) {
    for i := 0; i < 10; i++ {
        ch <- i
        time.Sleep(100 * time.Millisecond)
    }
    close(ch)
}
func consumer(ch <-chan int) {
    for num := range ch {
        fmt.Println("Получено число:", num)
    }
}
func main() {
    ch := make(chan int)
    go producer(ch)
    consumer(ch)
}
```

Гонка данных

- Гонка данных происходит, когда две горутины одновременно обращаются к одной и той же переменной и хотя бы одно из обращение является записью.

```go
func race() {
    wait := make(chan struct{})
    n := 0
    go func() {
        n++ // чтение, увеличение, запись
        close(wait)
    }()
    n++ // конфликтующий доступ
    <-wait
    fmt.Println(n)
}
```

гонка данных. пример.

Гонка данных

- Предпочтительный способ обработки одновременного доступа к данным в Go – использовать канал для передачи данных от одной горутины к следующей.

```
func sharingIsCaring() {
    ch := make(chan int)
    go func() {
        n := 0 // Локальная переменная видна только для первой горутины
        n++
        ch <- n // Данные отправляются из первой горутины
    }()
    n := <-ch // ...и благополучно прибывают во вторую
    n++
    fmt.Println(n) // Вывод: 2
}
```

```go
func main() {
    i := 0
    go func() {
        i++ // запись
    }()
    fmt.Println(i)
}
```

```
$ go run -race main.go
0
==================
WARNING: DATA RACE
Write by goroutine 6:
  main.main.func1()
      /tmp/main.go:7 +0x44

Previous read by main goroutine:
  main.main()
      /tmp/main.go:9 +0x7e
```

Как отлаживать deadlock-и

- Дэдлоки возникают, когда горутины ждут друг друга и ни одна из них не может завершиться.

```
func main() {
        ch := make(chan int)
        ch <- 1
        fmt.Println(<-ch)
}
```

Как отлаживать deadlock-и

- Дэдлоки возникают, когда горутины ждут друг друга и ни одна из них не может завершиться.

Как отлаживать deadlock-и

- Программа застрянет на операции отправки, ожидая вечно, пока кто-то прочитает значение. Go способен обнаруживать подобные ситуации во время выполнения. Вот результат нашей программы:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        .../deadlock.go:7 +0x6c
```

Блокировка взаимного исключения (мьютекс)

- Иногда удобнее синхронизировать доступ к данным с помощью явной блокировки, а не с помощью каналов. Стандартная библиотека Go предлагает для этой цели блокировку взаимного исключения sync.Mutex.
- В следующем примере мы создаем безопасную и простую в использовании конкурентную структуру данных AtomicInt, в которой хранится integer. Любое количество горутин может безопасно получить доступ к этому числу с помощью методов Add и Value.

Блокировка взаимного исключения (мьютекс)

```
type AtomicInt struct {
    mu sync.Mutex
    n  int
}
```

```
func (a *AtomicInt) Add(n int) {
    a.mu.Lock()
    a.n += n
    a.mu.Unlock()
}

func (a *AtomicInt) Value() int {
    a.mu.Lock()
    n := a.n
    a.mu.Unlock()
    return n
}
```