Урок Спринт 2.6

Каналы

Вы уже знакомились с каналами, как со средством синхронизации горутин. Давайте вспомним и применим подход с ожиданием блокировки на канале, чтобы избавиться от `time.Sleep(time.Second)` в предыдущем примере.

Рассмотрим простой пример:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan struct{})
    // горутина, которая асинхронно производит вычисления
    go func() {
        fmt.Println("начинаем вычисления...")
        // эмитируем длинные вычисления
        time.Sleep(time.Second)
        fmt.Println("заканчиваем вычисления ...")
        // закрываем канал, чтобы получить сообщения
        close(ch)
    }()

    // программа блокируется
    <-ch
    fmt.Println("завершаем программу")
}
```

В этом примере программа основная горутина будет заблокирована на строке `<-ch`. Пока из канала не будет прочитано сообщение, строка `fmt.Println("завершаем программу")` не будет выполнена. Давайте вспомним, что при чтении из закрытого канала мы получаем дефолтое значение типа данных. В нашем случае дефолтным значением будет пустая структура.

Можно представить алгоритм работы следующим образом:

- инициализируем канал

- что-то асинхронно делаем
- там где, нужно подождать читаем из канала
- после завершения действия закрываем канал

Но в нашем исходном примере надо было дождаться завершения ни одной горутины, а сразу нескольких. Давайте решим задачу "в лоб". Заведём каналов по числу задач, завершения которых хотим дождаться.

Создадим слайс из каналов:

```
const size int = 10
channels := make([]chan struct{}, size)
```

Не забудем, что каналы надо создать функцией `make`

```
for i := range channels {
    channels[i] = make(chan struct{})
}
```

Ожидаем сообщения из всех каналов

```
for i := 0; i < size; i++ {
    <-channels[i]
}
```

В результате получим следующий код:

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
)

type SafeSlice struct {
    results []int
    mx      *sync.Mutex
}

func New() *SafeSlice {
    return &SafeSlice{
        mx:      &sync.Mutex{},
        results: []int{},
    }
}

// Добавляем к слайсу элемент item
func (s *SafeSlice) Append(item int) {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    s.mx.Lock()
```

```
    defer s.mx.Unlock()
    s.results = append(s.results, random())
}

// Получаем элемент слайсу по индексу
func (s *SafeSlice) Get(index int) int {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    s.mx.Lock()
    defer s.mx.Unlock()
    return s.results[index]
}

// Функция генерирует случайное число в интервале [0, 100)
func random() int {
    const max int = 100
    return rand.Intn(max)
}

func main() {
    safeSlice := New()
    const size int = 10
    // объявляем слайс каналов
    channels := make([]chan struct{}, size)
    // создаем каналы функцией make
    for i := range channels {
        channels[i] = make(chan struct{})
    }
    // Заполняем слайс случайными числами
    for i := 0; i < size; i++ {
        go func(i int) {
            safeSlice.Append(random())
            // закрываем канал после выполнения задачи
            close(channels[i])
        }(i)
    }
    // ждем, пока не получим сообщения из всех каналов
    for i := 0; i < size; i++ {
        <-channels[i]
    }

    // Поэлементно печатаем слайс на экран
    for i := 0; i < size; i++ {
        fmt.Println(safeSlice.Get(i))
    }
}
```

Согласитесь, это довольно громоздкая реализация. При этом для конкурентного программирования довольно естественным выглядит желание запустить несколько конкурентных вычислений и по их завершению выполнить какое-то действие над полученным результатом. Давайте попробуем упростить решение.

Справка