Урок Спринт 2.7

Потоковая обработка

Один из самых популярных шаблонов `concurrency` в `Go` это **pipeline** (или потоковая обработка), который предполагает соединение нескольких этапов обработки через каналы для создания конвейера обработки данных. Этот шаблон особенно полезен при работе с большими объемами данных, которые необходимо обрабатывать последовательно.

`Pipeline` состоит из нескольких этапов, где каждый выполняет определенную операцию с данными и передает их на следующий этап по каналам. Этап может иметь любое количество входящих и исходящих каналов, за исключением первого и последнего этапов, которые имеют только исходящие или входящие каналы соответственно. Первый этап обычно называют источником (`source` или `producer`); последний — потребитель (`sink` или `consumer`).

Давайте рассмотрим этот шаблон на простом примере:

```
func EvenNumbersGen(numbers ...int) <-chan int {
    out := make(chan int) // канал для записи выходных данных
    go func() { // запускаем в отдельной горутине
        defer close(out) // закроем канал, когда больше нет данных
        for _, num := range numbers {
            if num%2 == 0 {
                out <- num // запишем в канал
            }
        }
    }()
    return out // вернём канал
}
```

Первый этап, `EvenNumbersGen`, — это функция, которая получает на вход список целых чисел, а возвращает канал, который выдает только чётные числа из списка. `EvenNumbersGen` запускает горутину, которая отправляет целые числа в канал и закрывает канал, когда все значения отправлены. Этап `DoubleNumbers` удваивает каждое число, полученное на предыдущем этапе, и передаёт его в выходной канал.

```
func DoubleNumbers(in <-chan int) <-chan int {
    out := make(chan int) // выходной канал
```

```
    go func() {
        defer close(out) // закроем канал
        for num := range in {
            out <- num * 2
        }
    }()
    return out  // вернём канал
}
```

Функция `main` настраивает `pipeline` и запускает финальный этап: она получает значения со второго этапа и печатает каждое, пока канал не закроется:

```
func main() {
    // канал в четными числами
    evens := EvenNumbersGen(1, 2, 3, 4, 5, 6)
    // канал с удвоенными числами
    out := DoubleNumbers(evens)
    // печатаем каждое число
    for num := range out {
        fmt.Println(num) // 4, 8, 12
    }
}
```

`DoubleNumbers` имеет одинаковый тип входных и выходных данных, поэтому можно использовать его вот так:

```
func main() {
    // канал в четными числами
    evens := EvenNumbersGen(1, 2, 3, 4, 5, 6)
    // канал с дважды удвоенными числами
    out := DoubleNumbers(DoubleNumbers(evens))
    // печатаем каждое число
    for num := range out {
        fmt.Println(num) //8, 16, 24
    }
}
```

Несколько функций могут читать один и тот же канал, это называется `fan-out`.

```
func main() {
    // канал в четными числами
    evens := EvenNumbersGen(1, 2, 3, 4, 5, 6)
    // Распределим нагрузку. Обе функцию читают из одного канала
    doubled1 := DoubleNumbers(evens)
    doubled2 := DoubleNumbers(evens)
}
```

Это позволяет распределить нагрузку среди обработчиков. Например, операция чтения файлов с диска может занимать значительное время, но если диски независимы, то гораздо быстрее будет это сделать нескольким обработчикам, где каждый читает отдельный файл. После чтения, возможен более быстрый

этап обработки, который будет делать всё последовательно. Для этого нужно объединить каналы:

```
// Прочитаем значения с объединенного канала
for n := range merge(doubled1, doubled2) {
    fmt.Println(n) // 4, 8, 12 - могут быть в любом порядке
}
```

Функция `merge` может быть использована для объединения каналов `double1` и `double2` в один канал:

```
func merge(cs ...<-chan int) <-chan int {
    // для синхронизации
    var wg sync.WaitGroup
    // объединять будем в этот канал
    out := make(chan int)
    // output - функция, которая копирует данные из
    // входящего канала (пока он не закрыт) в out,
    output := func(c <-chan int) {
        defer wg.Done()
        for n := range c {
            out <- n
        }
    }
    // добавим по количеству каналов
    wg.Add(len(cs))
    for _, c := range cs {
        // запустим функцию output в отдельном канале
        go output(c)
    }
    // запустим горутину, ожидающую завершения чтения данных из всех входящих
    // каналов. Затем закроем выходящий канал
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```