Урок Спринт 2.8

Многошаговая обработка данных

В предыдущем уроке мы использовали цикл `range` для получения значений из канала. Это позволяет быть уверенными, что все горутины завершатся после закрытия канала. Однако, в реальных системах не всегда нужно обрабатывать все значения из входящих каналов. Например, если на одном из этапов возникла ошибка и мы хотим завершить работу программы — нужно проинформировать об этом все этапы пайплайна. Как мы можем это сделать? Попробуем для этого использовать еще один канал:

```
done := make(chan struct{})
```

Тип данных в этом канале — пустая структура, потому что нам не важен сам тип, а важен факт появления данных в канале.

Теперь внесём изменения в генератор чисел:

```
func EvenNumbersGen(done <-chan struct{}, 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 {
            select {
            case <-done:
                return // после закрытия канала done - выходим
            default:
                if num%2 == 0 {
                    out <- num // запишем в канал
                }
            }
        }
    }()
    return out // вернём канал
}
```

Обратите внимание на изменения в функции. В предыдущей версии запись в канал была реализована так:

```
for _, num := range numbers {
    if num%2 == 0 {
        out <- num
```

```
    }
}
```

Здесь же мы используем конструкцию `select`:

```
select {
    case <-done:
        return
    default:
        if num%2 == 0 {
            out <- num // запишем в канал
        }
    }
```

Это позволит выйти из функции, когда мы получим данные из канала `done`, либо когда он будет закрыт (подробнее можно почитать **здесь**).

Использовать всё это можно вот так:

```
func main() {
    done := make(chan struct{})
    defer close(done) // закроем канал при завершении main
    // канал с четными числами
    evens := EvenNumbersGen(done, 1, 2, 3, 4, 5, 6)
    fmt.Println(<-evens) // прочитаем первое значение
    // ...
}
```

Теперь, при получении сигнала о завершении, горутина, созданная на этапе генерации чисел тоже завершится. Зачем? Несмотря на то, что горутины в Go легковесны, они тратят ресурсы и не удаляются сборщиком мусора. Поэтому важно завершать их выполнение, когда они больше не нужны.

Теперь рассмотрим, какие генераторы могут быть полезны в разработке. Генераторы — это любая функция, которая преобразует набор дискретных значений в поток данных в канале. Один из простых генераторов — `repeat`:

```
func Repeat[T any](
    done <-chan struct{},
    values ...T, // используем дженерики для передачи любого значения
) <-chan T {
    out := make(chan T)
    go func() {
        defer close(out)
        for { // бесконечный цикл
            for _, v := range values {
                select {
                case <-done:
                    return
                case out <- v:
                }
```

```
            }
        }
    }()
    return out
}
```

Он будет записывать в выходной канал переданные значения, пока не закрыт канал done. Сам по себе такой генератор не выглядит полезным, попробуем использовать в комбинации с функцией Take:

```
func Take[T any](
    done <-chan struct{},
    valueStream <-chan T,
    num int,
) <-chan T {
    out := make(chan T)
    go func() {
        defer close(out)
        for i := 0; i < num; i++ { // ограниченное число значений
            select {
            case <-done:
                return
            case out <- <-valueStream:
            }
        }
    }()
    return out
}
```

Take записывает в выходной канал только заданное число значений из входного. В комбинации это выглядит интересно:

```
func main() {
    done := make(chan struct{})
    defer close(done)
    out := Take(done, Repeat(done, 2), 4)
    for v := range out {
        fmt.Println(v) // 2 2 2 2
    }
}
```

В коде выше Repeat генерирует бесконечное число двоек, а Take берёт только четыре из них. Давайте попробуем модифицировать функцию Repeat, чтобы она записывала в канал не заданную нами последовательность, а результат работы фунцкии:

```
func RepeatFunc(
    done <-chan struct{},
    fn func() int, // функция, которая будет вызываться бесконечно
) <-chan int {
    out := make(chan int)
    go func() {
```

```
        defer close(out)
        for {
            select {
            case <-done:
                return
            case out <- fn(): // запишем в канал результат работы функции
            }
        }
    }()
    return out
}
```

Давайте попробуем вызывать функцию генерации случайного числа пять раз:

```
func main() {
    done := make(chan struct{})
    defer close(done)
    // функция генерации радномного числа
    randFunc := func() int { return rand.Int() }
    out := Take(done, RepeatFunc(done, randFunc), 5)
    for v := range out {
        fmt.Println(v) // здесь будут пять случайных чисел
    }
}
```

Теперь у нас есть генератор случайных чисел, где мы можем легко указать нужное нам количество элементов.