Урок Спринт 2.10

Реализация

В предыдущих уроках мы рассматривали типичную реализацию `fan-out — worker pool`. Теперь давайте представим, что нам нужно выполнить обратную задачу, то есть объединить результаты работы из нескольких каналов в один (`fan-in`). Пусть у нас есть набор каналов `channels ...<-chan int` и мы хотим объединить всё в один канал.

```
// FanIn - объединяет данные из нескольких каналов в один.
func FanIn(ctx context.Context, channels ...<-chan int) <-chan int{
    outputCh := make(chan int)      // выходной канал
    wg := sync.WaitGroup{}        // для ожидания завершения
    for _, ch := range channels { // переберём все каналы
        wg.Add(1)
        // для каждого канала вызовем функцию в отдельной горутине
        go func(input <-chan int) {
            defer wg.Done() // отметим, что функция завершилась
            for { // цикл для получения данных из входных каналов
                select {
                case data, ok := <-input: // получим данные из канала
                    if !ok {
                        return // данных больше нет - выходим
                    }
                    outputCh <- data // запишем данные в выходной канал
                case <-ctx.Done(): // если нужно завершить - выходим
                    return
                }
            }
        }(ch) // передадим входной канал в функцию
    }
    go func() {
        wg.Wait()       // дождёмся завершения обработки всех каналов
        close(outputCh) // закроем выходной канал
    }()

    return outputCh // вернём канал
}
```

Теперь давайте вспомним генератор чисел из наших уроков:

```
func EvenNumbersGen(ctx context.Context, 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 <-ctx.Done():
                return // выходим
            default:
                if num%2 == 0 {
                    out <- num // запишем в канал
                }
            }
        }
    }()
    return out // вернём канал
}
```

Также нам понадобится `OddNumbersGen` — вы можете реализовать её по аналогии с `EvenNumbersGen`. А теперь решим задачу объединения чётных и нечётных чисел в один канал:

```
// исходный слайс
nums := []int{0, 1, 2, 3, 4, 5, 6}
// канал с чётными числами
inputCh1 := EvenNumbersGen(ctx, nums...)
// канал с нечётными числами
inputCh2 := OddNumbersGen(ctx, nums...)
// выходной канал, где должны быть все числа из nums
outCh := FanIn(ctx, inputCh1, inputCh2)
for num := range outCh {
    fmt.Println(num) // 2, 0, 1, 4, 3, 6, 5 (у вас может быть другой порядок)
}
```

Задачу объединения мы решили, но, скорее всего, порядок чисел в канала `outCh` будет нарушен. Для некоторых задач порядок объединения является важным критерием, например, если вы разделили сжатый файл на несколько частей, передали их по каналу и хотите снова собрать все части. Таким образом, каждая часть сообщения должна иметь информацию о своём порядковом номере. Для этого нам подойдёт интерфейс `sequenced`:

```
type sequenced interface {
    getSequence() int // порядковый номер части сообщения
}
```

В примере выше мы использовали элементы `int`, как части сообщения, теперь создадим отдельный тип для реализации интерфейса `sequenced`:

```
type Num struct {
    int
}
// getSequence реализует интерфейс sequenced.
func (n Num) getSequence() int {
```

```
    // для упрощения примера будем считать
    // само число как порядковый номер части
    return n.int
}
```

Теперь нужно немного изменить функцию `EvenNumbersGen`(и `OddNumbersGen`):

```
func EvenNumbersGen[T sequenced](
    ctx context.Context,
    numbers ...T,
) <-chan T{
    // тело функции (останется как домашняя работа)
}
```

Как мы может теперь упорядочить элементы? Для каждого входного канала мы будем использовать горутину, которая получает элемент данных и отправляет его в некий временный канал для ожидания. Временный канал будет хранить элементы следующего типа:

```
type fanInRecord[T sequenced] struct {
    index int // порядковый номер горутины, из которой получено сообщение
    data  T // непосредственно данные
    pause chan struct{} // канал для синхронизации
}


// inTemp - записывает данные из каналов в один выходной с ожиданием.
func inTemp[T sequenced](
    ctx context.Context,
    channels ...<-chan T,
) <-chan fanInRecord[T] {
    // канал для ожидания
    fanInCh := make(chan fanInRecord[T])
    // для синхронизации
    wg := sync.WaitGroup{}
    // перебор всех входных каналов
    for i := range channels {
        wg.Add(1)
        // запустим горутину для получения данных из канала
        go func(index int) {
            defer wg.Done()
            // канал для синхронизации
            pauseCh := make(chan struct{})
            // цикл для получения данных из канала
            for {
                select {
                // получим данные из канала
                case data, ok := <-channels[index]:
                    if !ok {
                        return // канал закрыт - выходим
                    }
                    // положим во временный канал вместе с индексом
                    fanInCh <- fanInRecord[T]{
```

```
                        // индекс канала, откуда пришли данные
                        index: index,
                        // данные из канала
                        data:  data,
                        // канал для синхронизации
                        pause: pauseCh,
                    }
                case <-ctx.Done():
                    return
                }
                // ждём, пока в канал pause не будет передан сигнал
                // о получении очередного элемента из канала
                select {
                case <-pauseCh:
                // сняли с паузы
                // продолжим обработку данных из входного канала
                case <-ctx.Done():
                    return
                }
            }
        }(i)
    }
    go func() {
        // ожидаем завершения
        wg.Wait()
        close(fanInCh)
    }()
    // вернём канал с неотсортированными элементами
    return fanInCh
}
```

Далее нужно реализовать функционал получения данных из временного канала и, непосредственно, синхронизации для упорядочивания элементов:

```
func processTempCh[T sequenced](
    ctx context.Context,
    inputChannelsNum int, // количество входных каналов
    fanInCh <-chan fanInRecord[T], // временный канал с данными
) <-chan T {
    // выходной канал с упорядоченными данными
    outputCh := make(chan T)
    go func() {
        defer close(outputCh)
        // порядковый номер очередного элемента
        expected := 0
        // буфер для ожидания элементов по количеству входных каналов
        queuedData := make([]*fanInRecord[T], inputChannelsNum)
        for in := range fanInCh {
            // если получили элемент с номером, который ожидаем
            if in.data.getSequence() == expected {
                select {
                // запишем элемент в выходной канал
```

```
                case outputCh <- in.data:
                    // снимем с паузы исходный канал
                    // для продолжения обработки из входного канала
                    in.pause <- struct{}{}
                    // инкремент номера очередного элемента
                    expected++
                    // здесь нужно реализовать запись в выходной канал
                    // из буфера queuedData (задача для домашней работы)
                case <-ctx.Done():
                    return
                }
            } else {
                // если НЕ получили элемент с номером, который ожидаем
                // запишем элемент в буфер
                in := in
                queuedData[in.index] = &in
            }
        }
    }()
    return outputCh
}
```

В функции выше мы получаем элементы из временного канала, и если номер полученного элемента не соответствует ожидаемому, то записываем его в буфер. Горутина, из которой пришёл данный элемент будет заблокирована каналом `pause`. Если приходит элемент с нужным номером, он напрямую отправляется в выходной канал, а горутина, отправившая его, разблокируется. Далее нужно просканировать элементы в буфере и, если очередные элементы там, также отправить их в выходной канал. Реализации этой части будет одной из задач домашней работы. Теперь нам осталось запустить код для проверки:

```
nums := []Num{
    {0}, {1}, {2}, {3}, {4}, {5}, {6},
}
inputCh1 := EvenNumbersGen(ctx, nums...)
inputCh2 := OddNumbersGen(ctx, nums...)
// запись во временную очередь
inCh := inTemp(ctx, inputCh1, inputCh2)
// обработка из временной очереди и упорядочивание
outCh := processTempCh(ctx, 2, inCh)

for num := range outCh {
    fmt.Println(num) // {0} {1} {2} {3} {4} {5} {6}
}
```

В этом уроке мы разобрали задачу `ordered fan-in`. Обратите внимание, что мы не использовали критические секции, мьютексы, а каждая горутина выполняет свою работу, как только становятся доступными новые данные. Однако, вы всегда можете задействовать и другие средства для задач синхронизации.

Справка