Урок Спринт 2.9

Параллельная обработка

Аннотация

Освежим в памяти предыдущие уроки, в которых мы строили конвейеры обработки данных: вычисления производились в несколько этапов, которые преобразуют и обогащают результат. Обычно существует начальный этап, на котором формируется исходная последовательность элементов данных. На этом этапе эти элементы данных передаются один за другим на последующие этапы, каждый из которых обрабатывает данные, выдает результат и передаёт дальше. Хорошим примером являются конвейеры обработки изображений, в которых изображение декодируется, преобразуется, фильтруется, обрезается и кодируется в другое изображение.

Параллельная обработка

Рассмотрим подробнее еще один пример: напишем утилиту, которая сканирует файлы в директории и ищет в данных файлах шаблон, заданный регулярным выражением. Для начала создадим структуру `Work`:

```
type Work struct {
    file    string         // путь к файлу
    pattern *regexp.Regexp // регулярное выражение для поиска в файле
}
```

Данные о файлах в директории будут передаваться в канал `jobs` на первом этапе обработки:

```
func FileNameGen(dir string, pattern *regexp.Regexp) <-chan Work {
    jobs := make(chan Work) // канал для записи информации о файлах
    go func() {
        defer close(jobs) // закроем канал после обхода всех файлов
        // функция для перебора файлов в директории
        filepath.Walk(dir, func(path string, d fs.FileInfo, err error) error {
            if err != nil {
                return err
            }
            // пропускаем вложенные директории
```

```
            if !d.IsDir() {
                // запишем в канал файл, которые нужно обработать на следующем этапе
                jobs <- Work{file: path, pattern: pattern}
            }
            return nil
        })
    }()
    return jobs
}
```

Сама функция поиска будет получать файлы из канала `jobs`:

```
func worker(jobs <-chan Work) {
    // получаем информацию об очередном файле из канала
    for work := range jobs {
        // открываем файл
        f, err := os.Open(work.file)
        if err != nil {
            fmt.Println(err)
            continue // пропустим ошибки для упрощения
        }
        // scanner для чтения построчно
        scn := bufio.NewScanner(f)
        lineNumber := 1
        // читаем файл
        for scn.Scan() {
            // поиск в каждой строке
            result := work.pattern.Find(scn.Bytes())
            // если нашли - выведем результат на экран
            if len(result) > 0 {
                fmt.Printf("%s#%d: %s\n", work.file,
                    lineNumber, string(result))
            }
            lineNumber++
        }
        f.Close()
    }
}
```

Функция `main` может выглядеть так:

```
func main(){
    // регулярное выражение для поиска
    pattern := regexp.MustCompile(os.Args[2])
    // сканирование директории
    jobs := FileNameGen(os.Args[1], pattern)
    // поиск внутри каждого файла
    worker(jobs)
}
```

Можно попробовать запустить утилиту — найдём все вхождения `func` внутри файлов в текущей

директории:

```
go run main.go "./" ".*func.*"
```

Отлично! Однако, зачастую некоторые этапы обработки требуют большее время и ресурсы для выполнения своей задачи. Если бы в нашей директории были большие файлы, основное время выполнения заняла бы функция `worker`. Как мы можем ускорить выполнение? Было бы неплохо, если несколько экземпляров `worker` выполняли работу одновременно, то есть параллельно обрабатывали файлы, поступающие с предыдущего этапа. Для этого создадим пул обработчиков (`worker pool`):

```
wg := sync.WaitGroup{} //  для ожидания всех обработчиков
for i := 0; i < 3; i++ { // ограничим размер пула до трёх
    wg.Add(1)
    go func() {
        defer wg.Done() // отметим, что обработчик завершил работу
        worker(jobs)
    }()
}
wg.Wait() // дождёмся окончания работы всех обработчиков
```

Теперь, на этапе обработке файла мы создали пул из трёх обработчиков, которые будут забирать файлы из канала `jobs`. Каждый обработчик будет выполнять свою работу в своей горутине, при этом максимальное число одновременно обрабатываемых файлов мы ограничили тремя.