Отчет по выполнению HW3. Проффилировщик pprof

Профиллировщик

Постановка задачи

Изначальная функция, которую необходимо оптимизировать имеет сигнатуру

```
func SlowSearch(out io.Writer)
```

Она выполняет задачу поиска пользователей и подсчета кол-ва уникальных имен браузера из JSON Файла. От нас необходимо реализовать функцию `func FastSearch(out io.Writer)`, которая значительно улучшит производительность исходной реализации.

Для начала запустим бенчмарки данных функций: будем использовать команду `go test -bench . -benchmem`

```
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$ go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: hw3
cpu: AMD Ryzen 7 5700X 8-Core Processor
BenchmarkSlow-16               50          26392146 ns/op        20410681 B/op     182840 allocs/op
BenchmarkFast-16               46          26406538 ns/op        20406793 B/op     182841 allocs/op
PASS
ok      hw3     2.655s
```

Пока что `func FastSearch(out io.Writer)` имеет такую же реализацию как и `SlowSearch()`.

Референсное улучшение выглядит как:

```
BenchmarkSolution-8 500 2782432 ns/op 559910 B/op 10422 allocs/op
```

По условию задания необходимо, чтобы хотя бы один из параметров (ns/op, B/op, allocs/op) был быстрее чем в *BenchmarkSolution* и еще один лучше чем в *BenchmarkSolution* + 20% (fast < solution * 1.2)

Начальные результаты

CPU

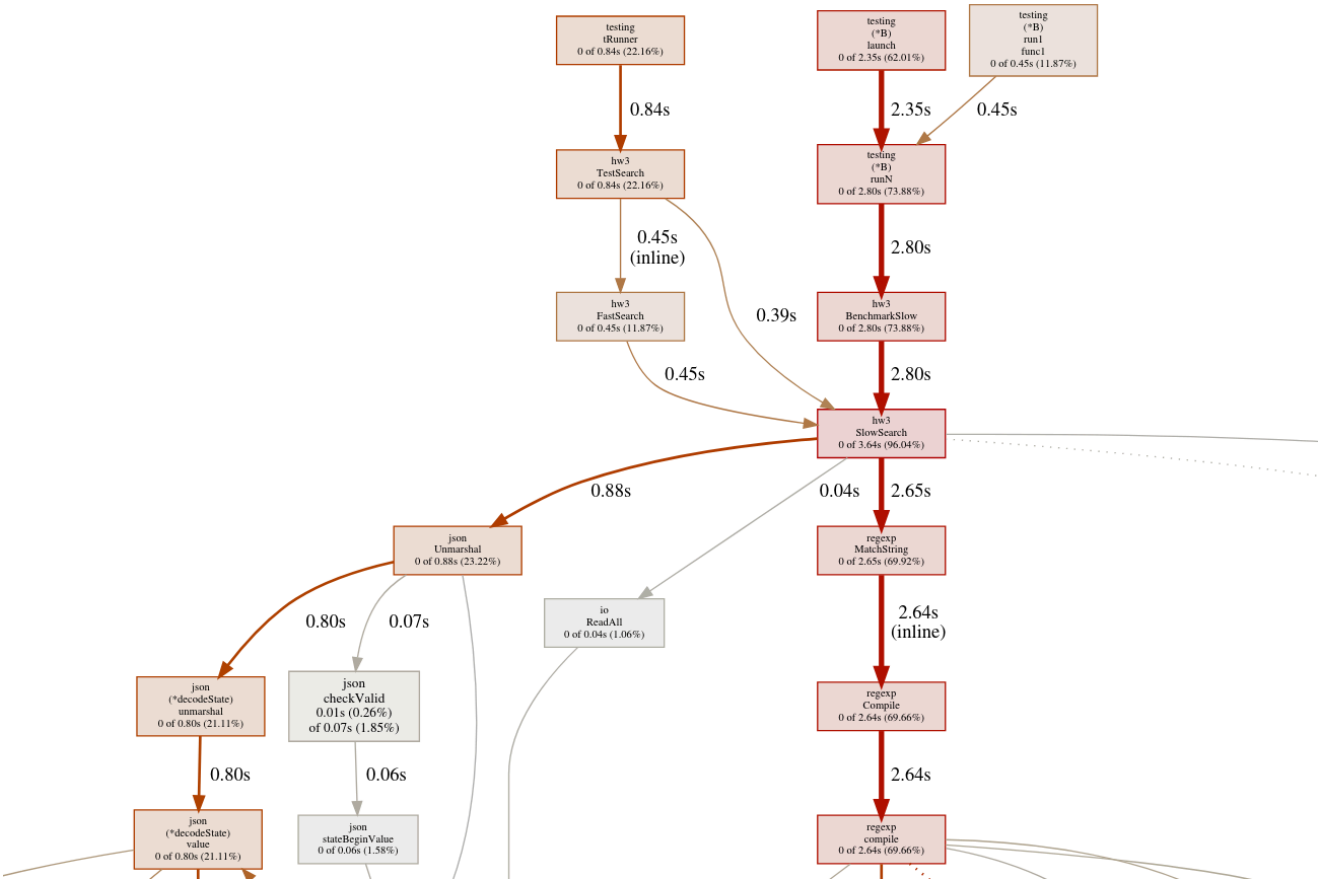
Запустим проффилировщик и получим в текстом виде снимок по исходнику. Для начала будем работать с профилем с CPU.

```
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$ go tool pprof hw3.test pprof_output/cpu.out
File: hw3.test
Type: cpu
Time: Jan 19, 2025 at 4:51pm (MSK)
Duration: 3.81s, Total samples = 3.79s (99.38%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) list SlowSearch
Total: 3.79s
ROUTINE ======================== hw3.SlowSearch in /home/mefja/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw/common.go
         0      3.64s (flat, cum) 96.04% of Total
         .          .     15:func SlowSearch(out io.Writer) {
         .          .     16:   file, err := os.Open(filePath)
         .          .     17:   if err != nil {
         .          .     18:           panic(err)
         .          .     19:   }
         .          .     20:
         .       40ms     21:   fileContents, err := io.ReadAll(file)
         .          .     22:   if err != nil {
         .          .     23:           panic(err)
         .          .     24:   }
         .          .     25:
         .          .     26:   r := regexp.MustCompile("@")
         .          .     27:   seenBrowsers := []string{}
         .          .     28:   uniqueBrowsers := 0
         .          .     29:   foundUsers := ""
         .          .     30:
         .          .     31:   lines := strings.Split(string(fileContents), "\n")
         .          .     32:
         .          .     33:   users := make([]map[string]interface{}, 0)
         .          .     34:   for _, line := range lines {
```

Отметим сразу горячие точки, которые мы можем наблюдать на поверхности

```
40ms      21:   fileContents, err := io.ReadAll(file)
40ms      35:           user := make(map[string]interface{})
910ms     37:           err := json.Unmarshal([]byte(line), &user)
1.43s     61:                   if ok, err := regexp.MatchString("Android",...
1.22s     83:                   if ok, err := regexp.MatchString("MSIE",...
```

Построим граф вызовов, полученный с проффилировщика. Львиную долю времени CPU занимает regexp, и также много времени занимает `json.Unmarshal`

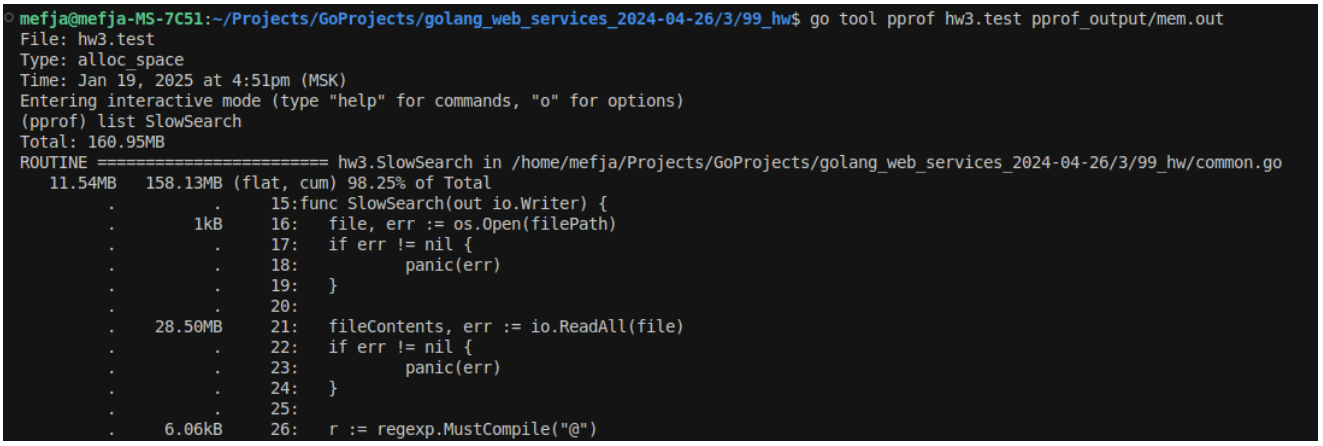


Также сделаем тоже самое, только для снимка памяти

[Профиллировщик]

Memory

Выпишем в текстом виде результат выполнения функции SlowSearch. Тип для отчетности выбран по умолчанию **alloc_space** (количество аллоцированных байт).



Также как и для CPU отметим подозрительные места, которые вызывают черезмерную аллокацию памяти, здесь список получился гораздо объемнее:

```
(flat)    (cum)
1.92MB    21.51MB

1kB                  16:   file, err := os.Open(filePath)
28.50MB              21:   fileContents, err := io.ReadAll(file)
6.06kB               26:   r := regexp.MustCompile("@")
4.92MB    5.05MB     31:   lines := strings.Split(string(fileContents), "\n")
437.50kB  437.50kB   35:   user := make(map[string]interface{})
4.53MB    15.55MB    37:   err := json.Unmarshal([]byte(line), &user)
136.94kB  136.94kB   41:   users = append(users, user)
64.73MB              61:   if ok, err := regexp.MatchString("Android", ...
22.38kB   22.38kB    71:   seenBrowsers = append(seenBrowsers, browser)
41.96MB              83:   if ok, err := regexp.MatchString("MSIE", ...
12.50kB   12.50kB    93:   seenBrowsers = append(seenBrowsers, browser)
68.66kB              104:  email := r.ReplaceAllString(user["email"].(string), )
1.46MB    1.60MB     105:  foundUsers += fmt.Sprintf("
38.12kB   85.62kB    108:  fmt.Fprintln(out, "found
```

В данном списке используются два столбца, *flat* и *cum*.

- **flat** - это объем памяти, который непосрдественно используется в данной функции, без учета вызовов других функций
- **cum (cumulative)** - это объем памяти, который используется в данной фукнции, включая все вызовы других функций, которые она делает.

Например:

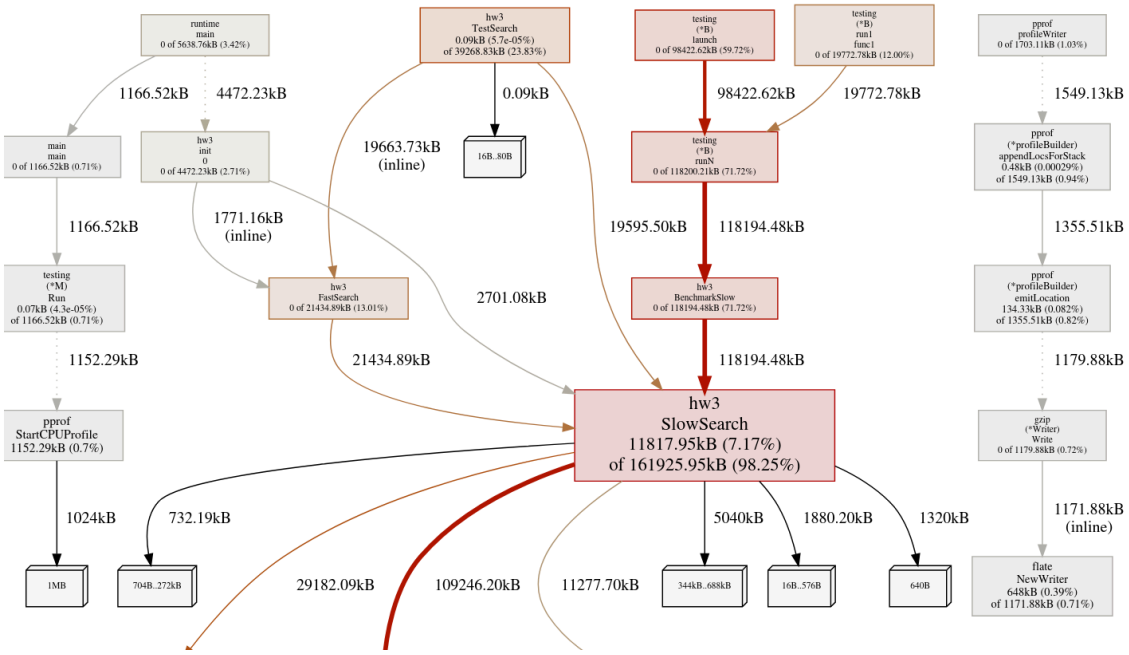
```
func foo() {
    x := make([]byte, 10) // 10 байт выделяются в foo()
    bar()                 // bar() выделяет ещё 20 байт
}

func bar() {
```

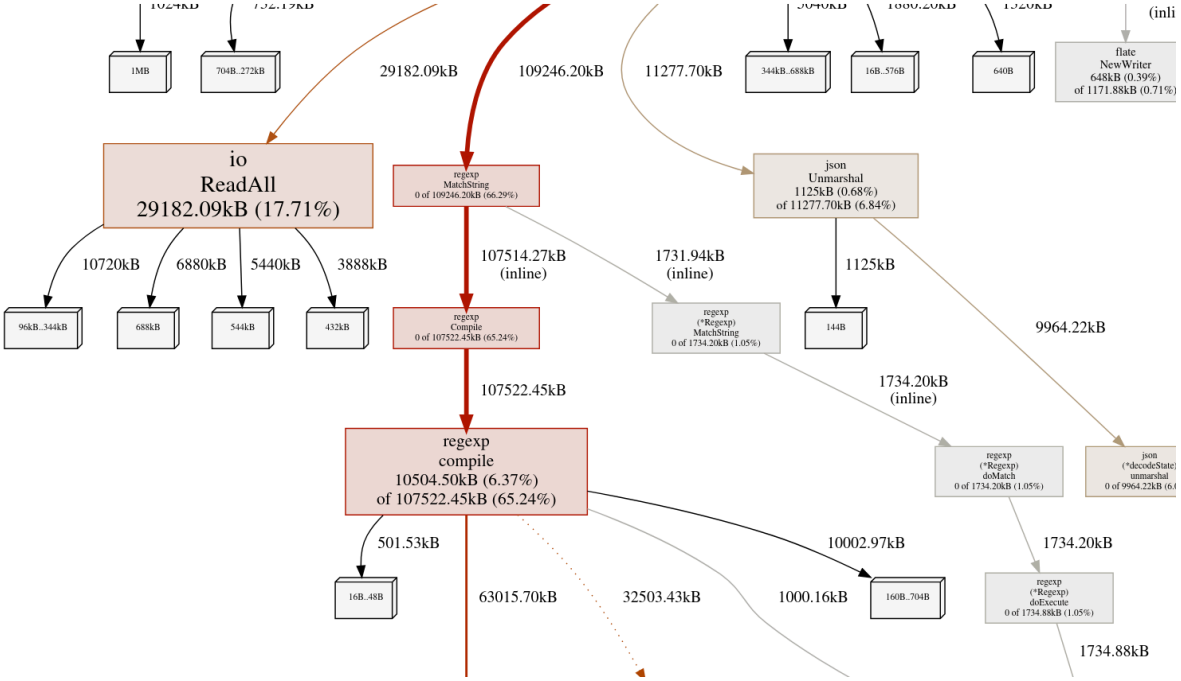
```
    y := make([]byte, 20) // 20 байт выделяются в bar()
}
```

То есть в столбце cum для `foo()` будет написано 30байт

Построим граф вызовов, полученный с проффилировщика.

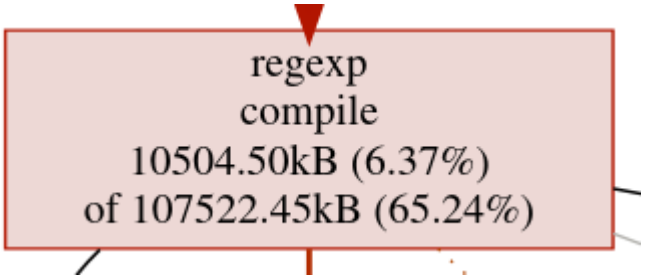


Пойдем ниже по графу вдоль самой жирной красной стрелке



Здесь встречаем вызов `regexp compile`. Также много памяти аллоцирует вызов `io.ReadAll`. Если рассмотрим отдельный блок, то увидим надпись 10504kb of 107522kb. Это означает что сам regexp compile выделяет напрямую память равную 10504kb, что как раз и является flat памятью. А вот кумулитативная память в данном случае равна 107522kb, эти аллокации будут видны ниже по

графу и происходят далее по стеку вызовов.



Более подробное описание того, что делает заданная функция

Исходное состояние

Напишем здесь основные моменты, которые происходят в функции, для того чтобы понять, какую логику мы должны сохранить и соптимизировать. Первое что происходит это открытие файла и чтение всего его содержимого с помощью вызова `io.ReadAll(file)`.

Далее мы разбиваем полученное содержимое на строки и итерируемся по ним. Перед циклом создаем мапу `users`, ключом которой будет строка, а значеним `interface{}` - это мапа для хранения всех юзеров, извлеченных из файла.

Итерируясь по строкам мы каждый раз аллоцируем нового юзера через

```
user := make(map[string]interface{})
```

и делаем десериализацию json'a

Затем итерируемся по массиву пользователей и в отдельных циклах проверяем на регулярные выражения строки "Android" и "MSIE". Если есть соответствие то проверяем, встречали ли мы такой бразуер раньше. Если нет, то записываем его в отдельный массив уникальных браузеров.

В самом конце выводим число уникальных браузеров и список пользователей, которых мы добавили на основе регулярных выражений и уникальных браузеров.

Места в коде, которые впервую очередь необходимо исправить

1. Попробовать читать файл не весь целеком, а построчно - что должно уменьшить аллокацию памяти.
2. Попробовать эффективно узнать кол-во строк в файле, чтобы во время аллокации массива `users` указать капасити для наиболее лучшей аллокации. В таком случае даже должна отпасть необходимость каждый раз аллоцировать нового user для того, чтобы добавить его в слайс, вместо этого уже в проинициализированной памяти мы будем класть десириализованный json объект.
3. Избавиться от двух повторяющихся циклов, в которых используется проверка на регулярные выражения, а в идеале вообще делать все действия в том цикле, которые идет по строчкам файлов.
4. Избавиться от регулярных выражений и заменить на `string.contains` поскольку это более легковесная операция.

5. Использовать мапы вместо массивов для более быстрого поиска по ключу вместо линейного поиска, например слайс `browsers`
6. Использовать Strings.Builder для конкотанации строк вместо +=

Полученные результаты после первых улучшений

```
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$ go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: hw3
cpu: AMD Ryzen 7 5700X 8-Core Processor
BenchmarkSlow-16              54          25811420 ns/op        20358485 B/op     182834 allocs/op
BenchmarkFast-16             154           7229028 ns/op         2763041 B/op      47631 allocs/op
PASS
ok      hw3     3.367s
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$
```

После проведенных улучшений видим заметную разницу в производительности, кол-во повторений выполняемого кода увеличилось в три раза, кол-во аллокаций, кол-во выделяемой памяти уменьшилось также в несколько раз. Однако, код все еще выполняется недостаточно производительно. Воспользуемся проффилировщиком для просмотра узких мест.

CPU

Выполнив консольную команду `go tool pprof hw3.test pprof_output/cpu.out`, а затем `list FastSearch()` получим следующие результататы

```
40ms      36:           user := make(map[string]interface{}, 0)
1.18s     37:           err := json.Unmarshal([]byte(scanner.Text()), &user)
20ms      76:           email := r.ReplaceAllString(user["email"].(string), " [at] ")
```

Очень много времени занимает анмаршалинг json сущностей. В таком случае следующей итерацией подключим библиотеку easyjson к проекту, для более оптимальной десириализации данных.

Memory

Для памяти выполним комману go tool `pprof hw3.test pprof_output/mem.out`. Далее получим список мест, где тратиться памяти больше всего.

```
.          1.25kB     23: file, err := os.Open(filePath)
.          7.58kB     30: r := regexp.MustCompile("@")
.            40kB     35: for i := 0; scanner.Scan(); i++ {
546.88kB 546.88kB     36: user := make(map[string]interface{}, 0)
5.66MB     25.09MB    37: err := json.Unmarshal([]byte(scanner.Text()), &user)
17.81kB    17.81kB    59: seenBrowsers[browser] = true
96.58kB    96.58kB    67: seenBrowsers[browser] = true
.          477.91kB   76: email := r.ReplaceAllString(user["email"].(string), " [at] "
...
18.27kB    275.55kB   77: foundUsersBuilder.WriteString(fmt.Sprintf("[%d] %s <%s>\n", i
..
47.66kB    76.16kB    80: fmt.Fprintln(out, "found
users:\n"+foundUsersBuilder.String())
```

Добавление easyjson к проекту

Для использования easyjson необходимо создать структуру, в которую мы будем анмаршалить наш JSON. Структура User будет выглядить следующим образом:

```
package user

//easyjson:json
type User struct {
        Browsers []string `json:"browsers"`
        Company  string   `json:"company"`
        Country  string   `json:"country"`
        Email    string   `json:"email"`
        Job      string   `json:"job"`
        Name     string   `json:"name"`
        Phone    string   `json:"phone"`
}
```

Запустим бенчмарки и заметим значительные улучшения.

```
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$ go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: hw3
cpu: AMD Ryzen 7 5700X 8-Core Processor
BenchmarkSlow-16              63          26029034 ns/op        20397836 B/op     182841 allocs/op
BenchmarkFast-16             561           2081398 ns/op         1841690 B/op      13712 allocs/op
PASS
ok      hw3     3.084s
```

Однако этого все еще недостаточно для выполнения задания, поскольку кол-во байтов на операцию и аллокаций на операций превосходит референсные значения.

Откажемся от регулярных выражений при замене `@` на `[at]`, вероятно они отнимают довольно много памяти и аллокации, скорее всего `strings.Replace` будет более легким в исполнении. Также значительно много производительности анмаршалинг не всех байтов JSON сущности, а для начала только массива браузеров, поскольку в цикле есть условие, что если браузеры не подходят по условию то мы двигаемся дальше.
Напишем вспомогательную функцию, которая делает анмаршалинг JSON только для браузеров, дадим ей название `getBrowserBytes`. Проведем тесты:

```
mefja@mefja-MS-7C51:~/Projects/GoProjects/golang_web_services_2024-04-26/3/99_hw$ go test -bench . -benchmem
goos: linux
goarch: amd64
pkg: hw3
cpu: AMD Ryzen 7 5700X 8-Core Processor
BenchmarkSlow-16              62          25048985 ns/op        20391032 B/op     182838 allocs/op
BenchmarkFast-16             883           1334051 ns/op         564387 B/op       6076 allocs/op
PASS
ok      hw3     2.928s
```

После данных изменений задание можно считать выполненым.