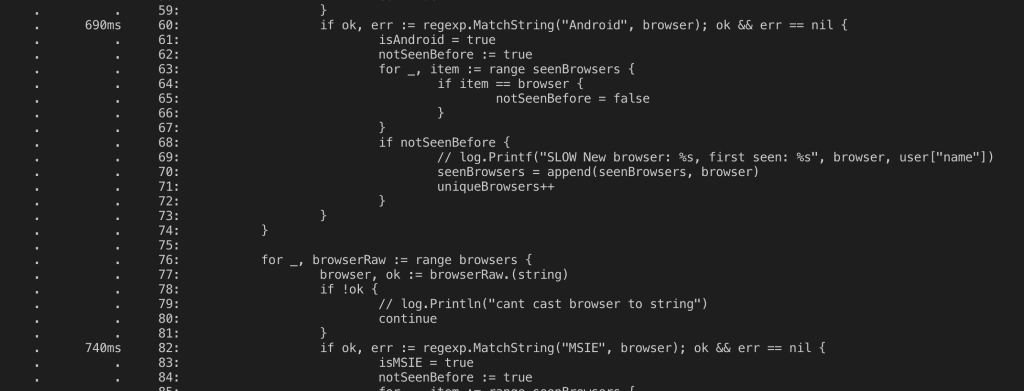
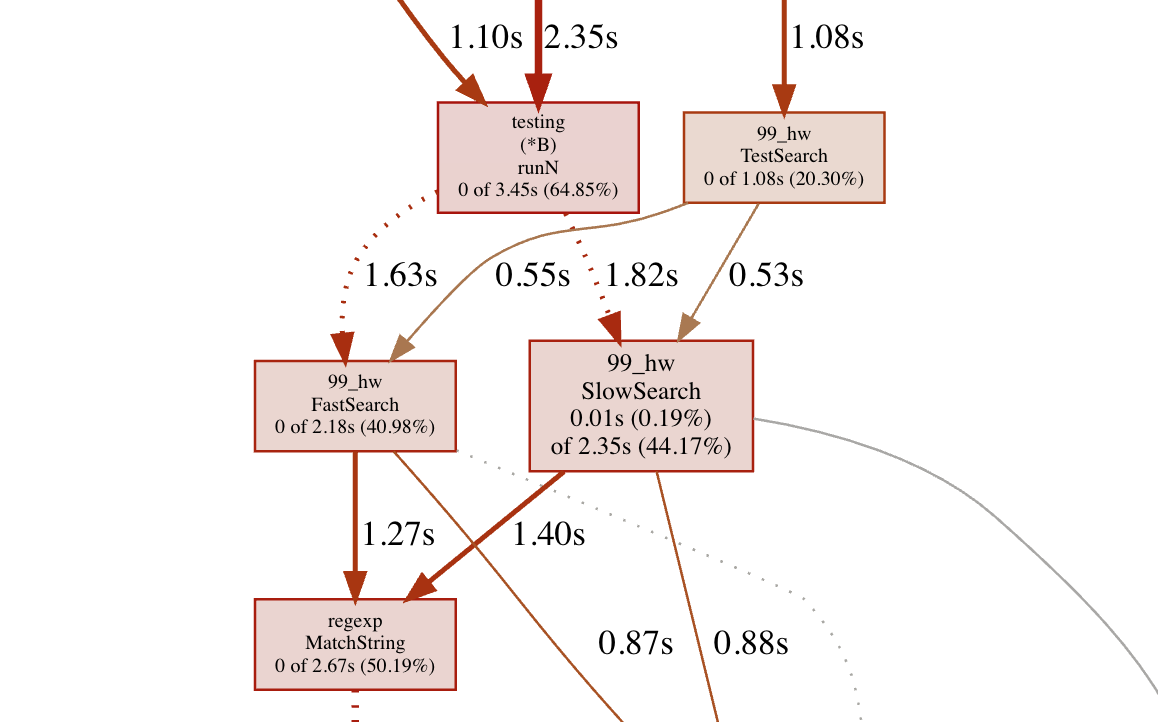
1. **Замена регулярным выражениям**

Сгенерируем файлы cpu.out и mem.out. Начнём с процессорного времени. Команда list FastSearch даёт следующий результат:



Взглянем на график:



Делаем вывод, что с помощью регулярных выражений делать поиск в строке неэффективно. В данный момент 50% вычислений занимает поиск подстроки.

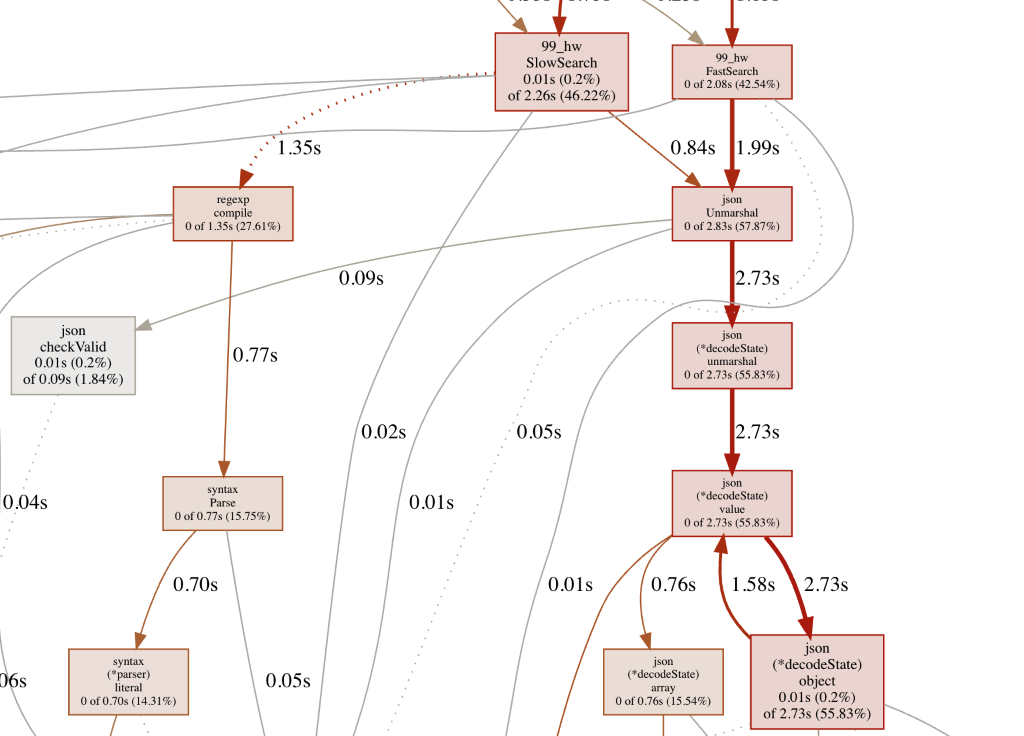
Заменим на strings.Contains и посмотрим результат:

**BenchmarkSlow-8                2         687629454 ns/op        19052184 B/op     195812 allocs/op**

**BenchmarkFast-8                4         265832168 ns/op         5427118 B/op      59734 allocs/op**

Как мы видим, теперь на одну операцию уходит в 2,6 раза меньше времени. Также получили уменьшение используемой памяти в 3,5 раза.

1. **Распаковка JSON в структуру**

Теперь большая часть процессорного времени(2.83s и 57.87%) уходит на распаковку JSON:

Попробуем создать структуру User

**type User struct {**

**Name string `json:name`**

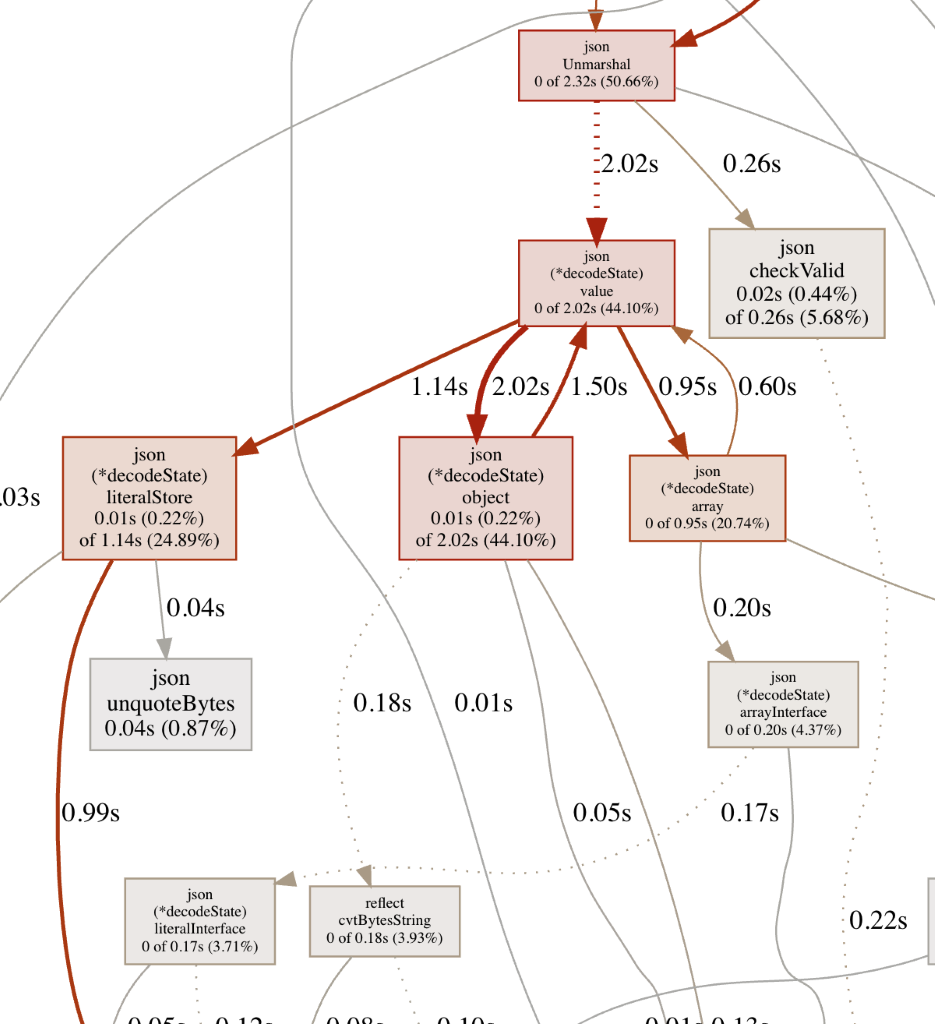
**Email string `json:email`**

**Browsers []string `json:browsers`**

**}**

И анмаршалить в неё, вместо map[string]interface{}. Таким образом мы явно пропишем типы и при распаковке будем иметь больше информации и типах. Также это поможем нам избежать приведений типов.

Получили следующую картину:



BenchmarkSlow-8                2         706696124 ns/op        19033352 B/op     195806 allocs/op

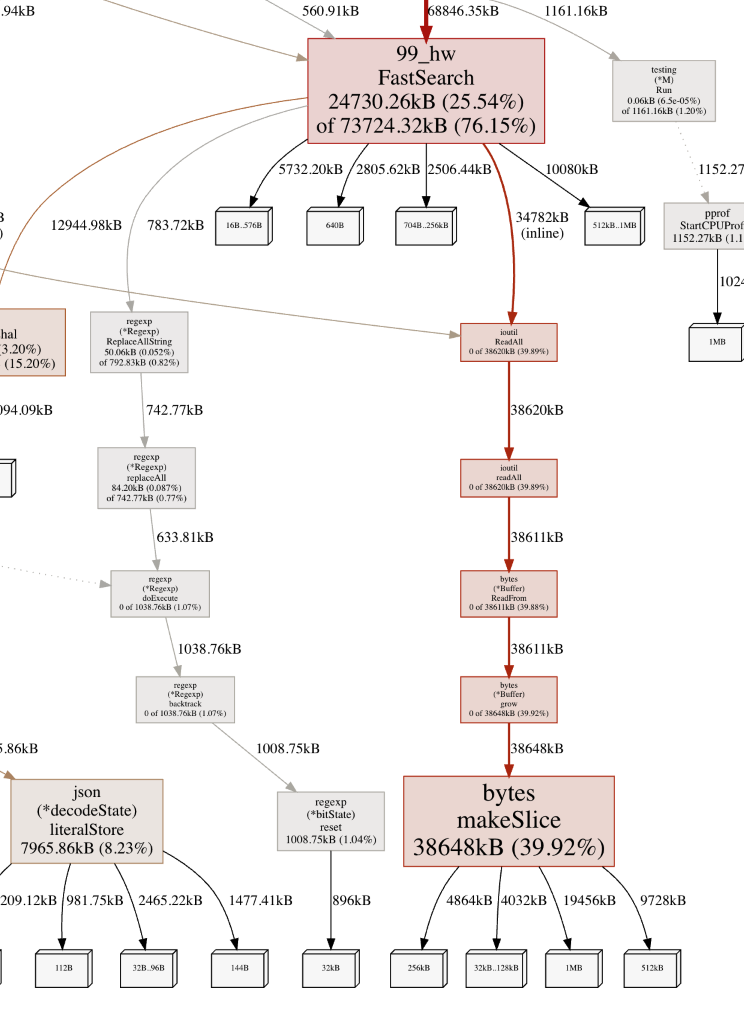
BenchmarkFast-8               15          74255451 ns/op         4401761 B/op      15813 allocs/op

Как видим, эта оптимизация позволила ещё ускорить программу в 3,6 раз. А также потреблять ещё меньше памяти.

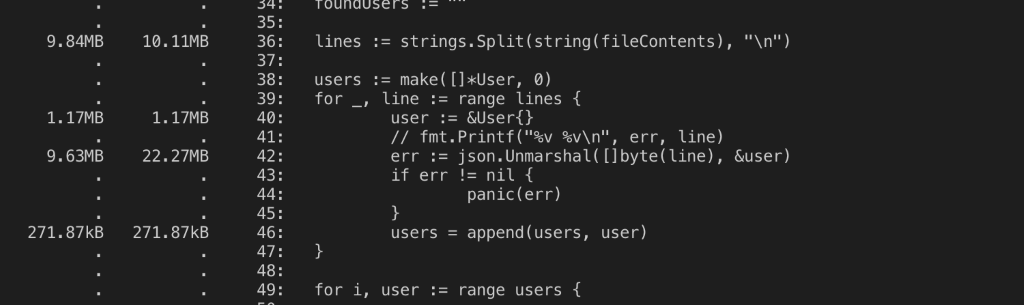
1. **Использование bufio.Scanner**

Перейдём к профилированию памяти.

Посмотрим, какая операция наиболее затратная по памяти:



Согласно графику мы видим, что много памяти уходит на материализацию всего файла в памяти(ioutil.readAll), при этом мы не указали изначально размер буфера(по умолчанию начальный размер составляет 4096 байт), и много ресурсов ушло на постоянное перевыделение памяти под бОльший буфер(он же удваивается каждый раз когда доходит до предела). Также существенно уходит на создание массива из строки путём strings.Split:



Для оптимизации мы можем считывать файл построчно и размер буфера должен быть при этом меньше, чем при сканировании целого файла. Используем для этого bufio.Scanner:

**s := bufio.NewScanner(file)**

**for s.Scan() {**

**user := &User{}**

**// fmt.Printf("%v %v\n", err, line)**

**err := json.Unmarshal([]byte(s.Text()), &user)**

**if err != nil {**

**panic(err)**

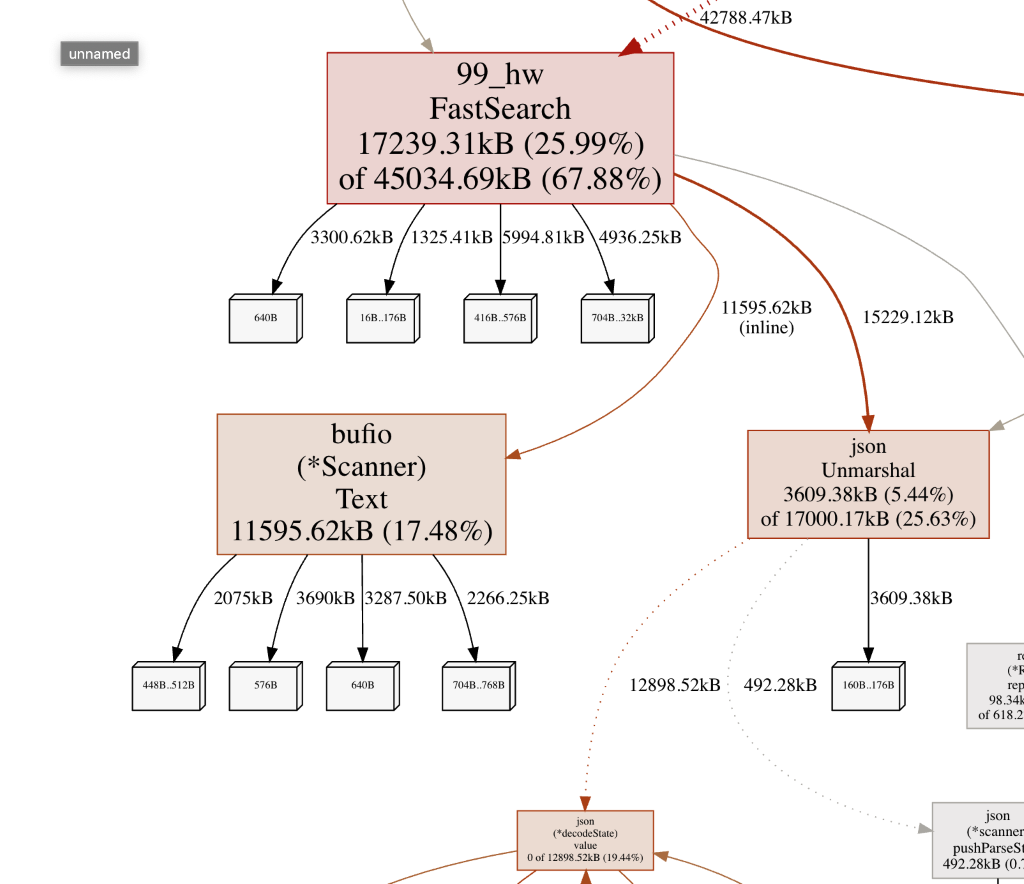
**}**

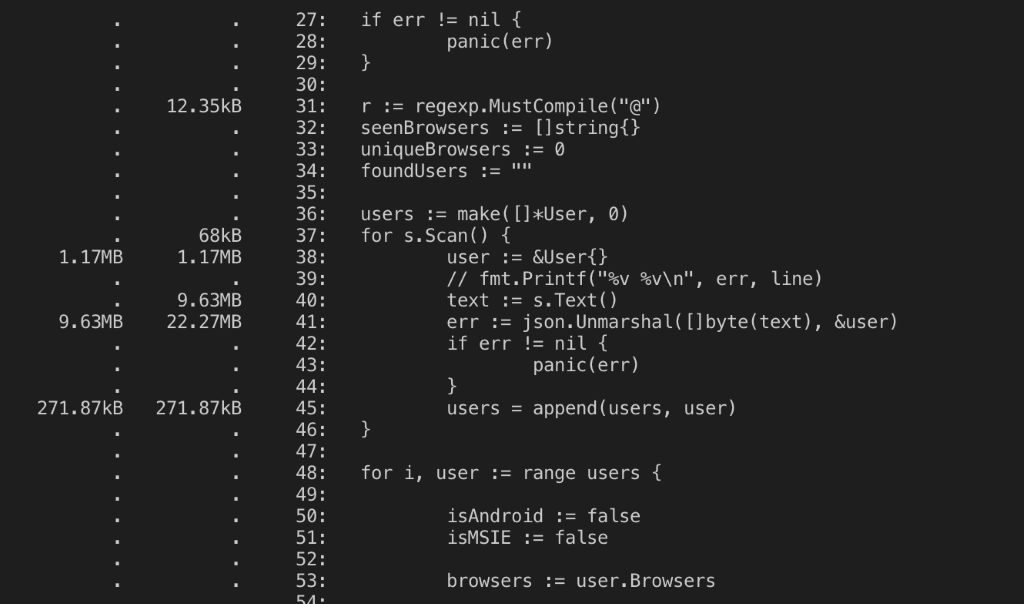
**users = append(users, user)**

**}**

Результат:

**BenchmarkSlow-8                2         675732498 ns/op        18978032 B/op     195801 allocs/op**

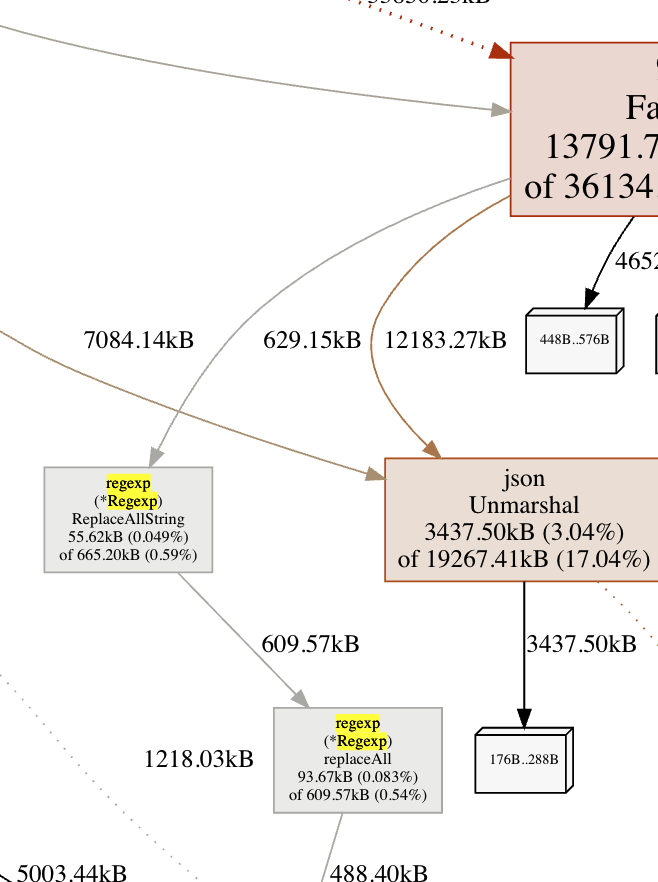
**BenchmarkFast-8               14          77110428 ns/op         2317124 B/op      16802 allocs/op** 



Тут мы видим, что s.Text() в цикле тратит намного меньше памяти за счёт переиспользования небольшой области памяти. В целом после этого шага используется в 2 раза меньше памяти(2317124 против 4401761 B/op)

1. **Использование strings.Replace**

В начале я забыл убрать Replace с помощью регулярного выражения:



Попробуем убрать и посмотрим результат:

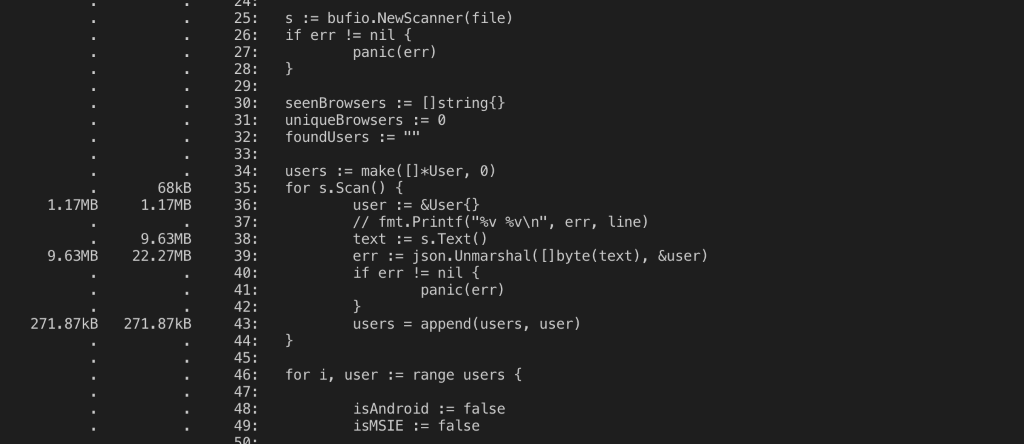
**BenchmarkSlow-8                2         683603812 ns/op        19107748 B/op     195816 allocs/op**

**BenchmarkFast-8               15          76141030 ns/op         2277159 B/op      16584 allocs/op**

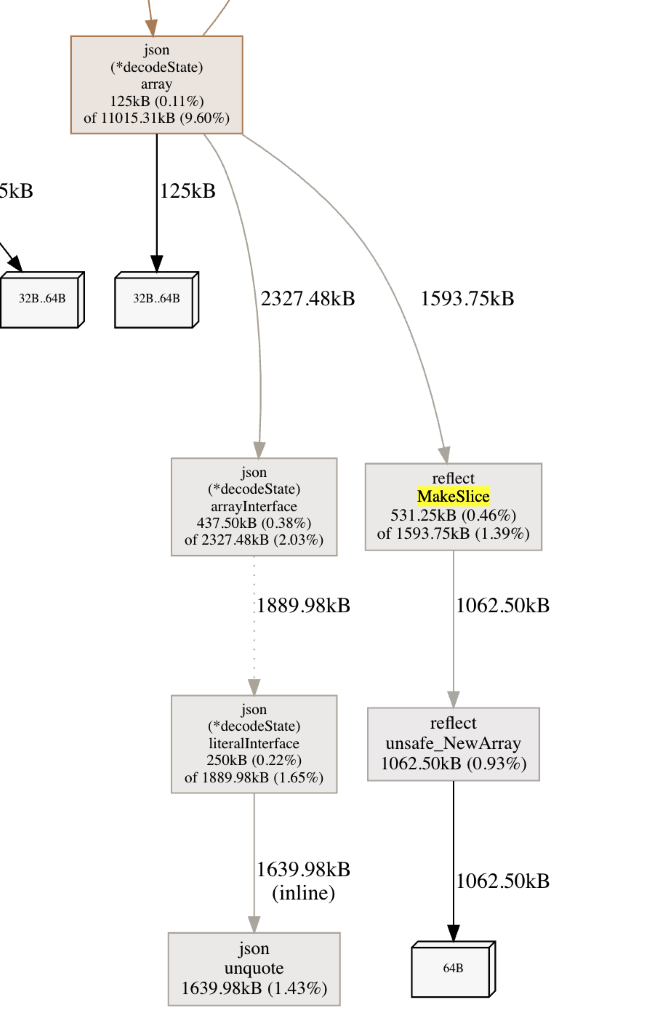
Ещё немного выиграли по памяти и времени.

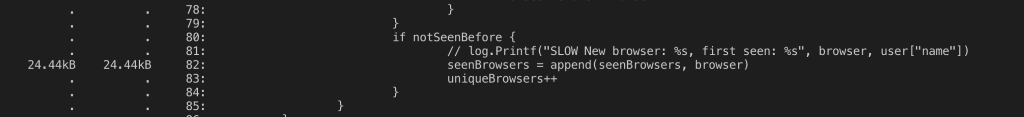
1. **Убираем лишний слайс данных, т.к. все значения не нужны одновременно**

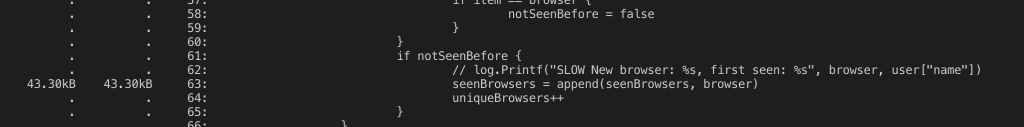
Посмотрим, что ещё можно оптимизировать:



Тут мы видим, что выделяется память на слайс. Это же место на графике:







В действительности мы можем обойтись без этого слайса, просто проходя по файлу и добавляя запись в результат.

А также можем проходить по массиву Browsers один раз, вместо двух проходов

**BenchmarkSlow-8                2         683382046 ns/op        18962464 B/op     195785 allocs/op**

**BenchmarkFast-8               15          75207016 ns/op         2259027 B/op      16571 allocs/op**

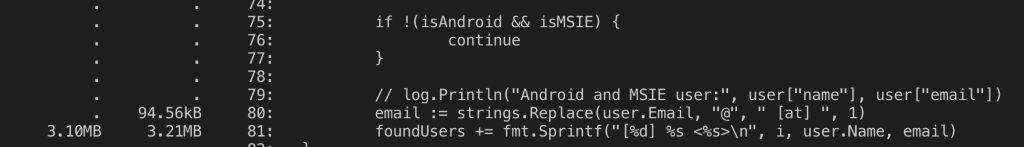
Если переделать алгоритм поиска уникальных браузеров на использование мапы, то по памяти становится хуже, а по процессору то же самое:

**BenchmarkSlow-8                2         743590432 ns/op        19021692 B/op     195805 allocs/op**

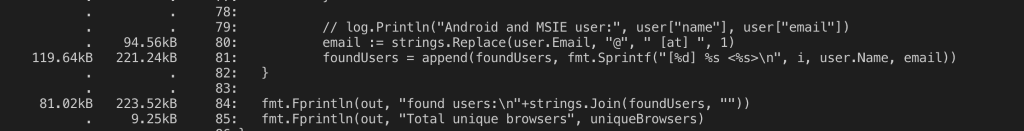
**BenchmarkFast-8               14          80813412 ns/op         2265683 B/op      16572 allocs/op**

Конечно, если мы рассчитываем на большее количество данных, использовать мапу было бы необходимо, чтобы избежать квадратичной сложности.(Позже поменял на мапу)

Ещё можно оптимизировать конкатенацию строк, на которую уходит избыток памяти, т.к. строки неиммутабельны:



Создадим массив и будем его аппендить, для вывода результата используем strings.Join



(Позже поменял конечно на strings.Builder)

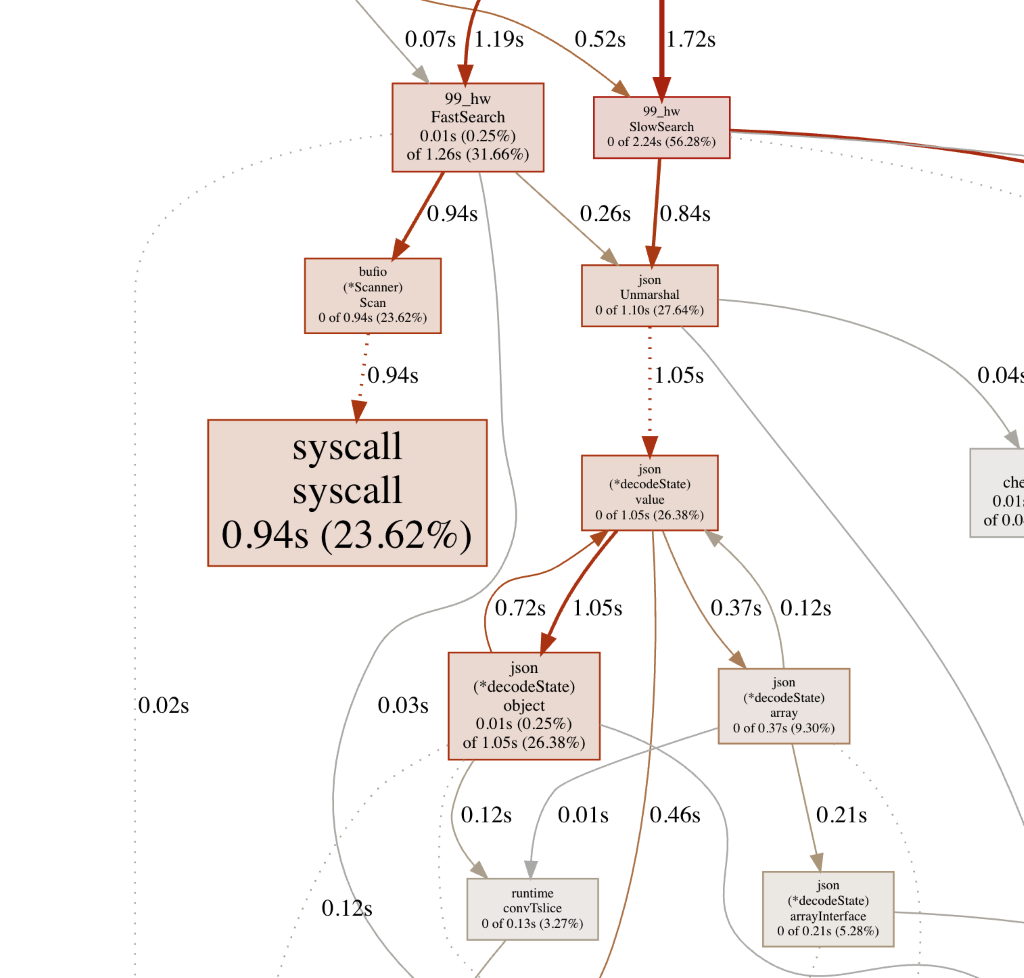
1. **Использование easyjson**

Вернёмся к анмаршаллингу и посмотрим что у нас есть сейчас(Результаты немного отличаются, перезапустил бенчмарки)

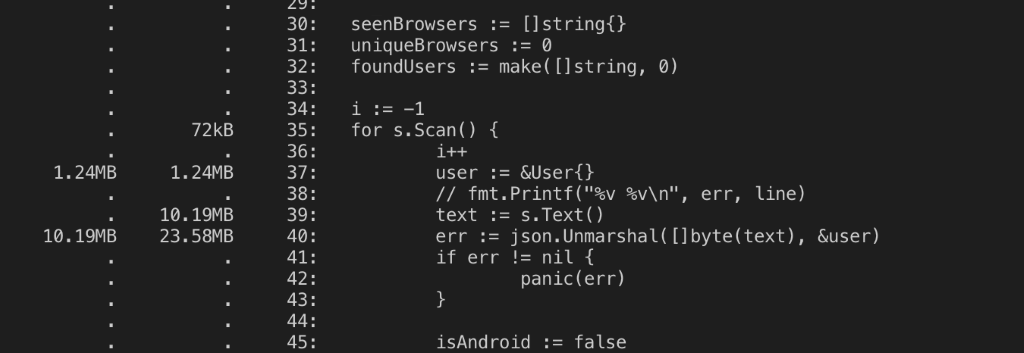
**BenchmarkSlow-8                2         670390244 ns/op        19037960 B/op     195810 allocs/op**

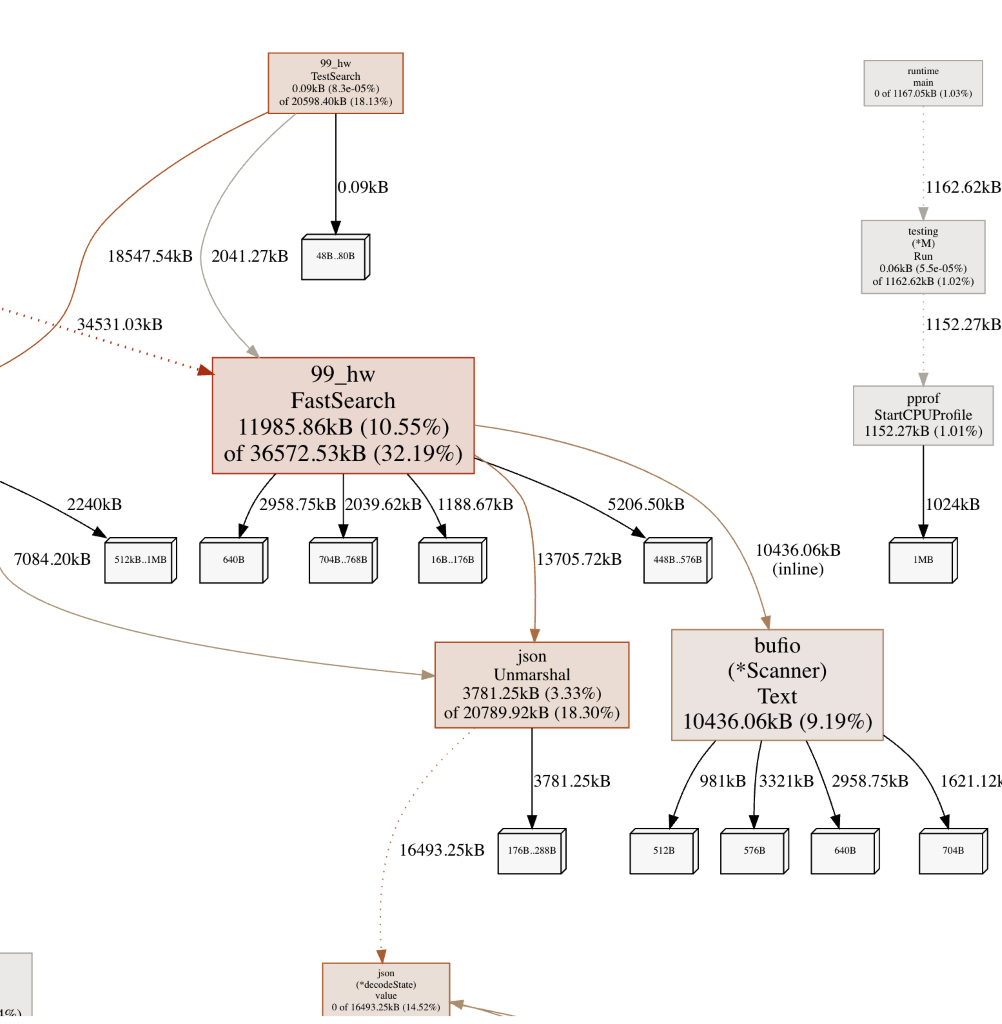
**BenchmarkFast-8               16          74644650 ns/op         2080427 B/op      16500 allocs/op**

Процессор:

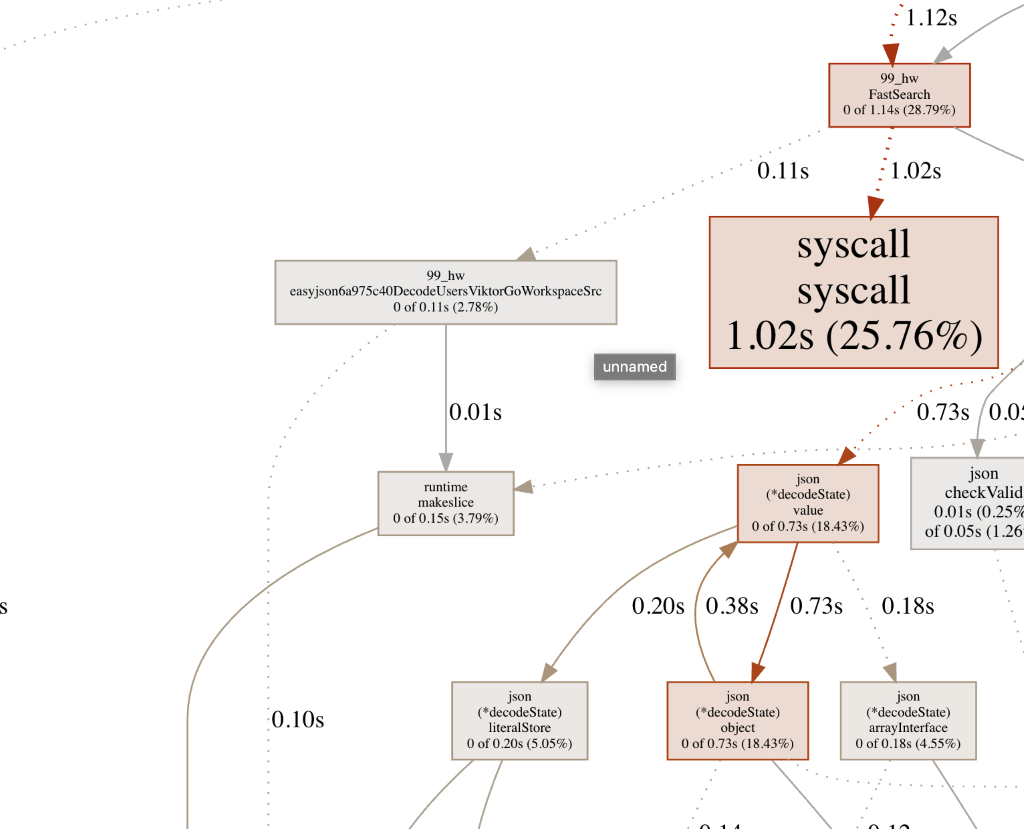


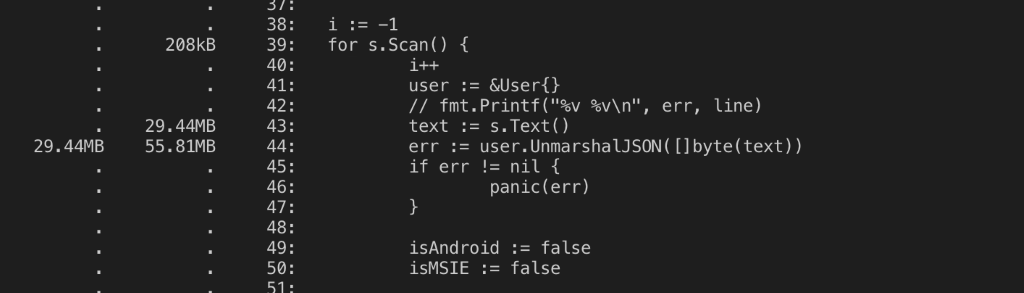
Память:

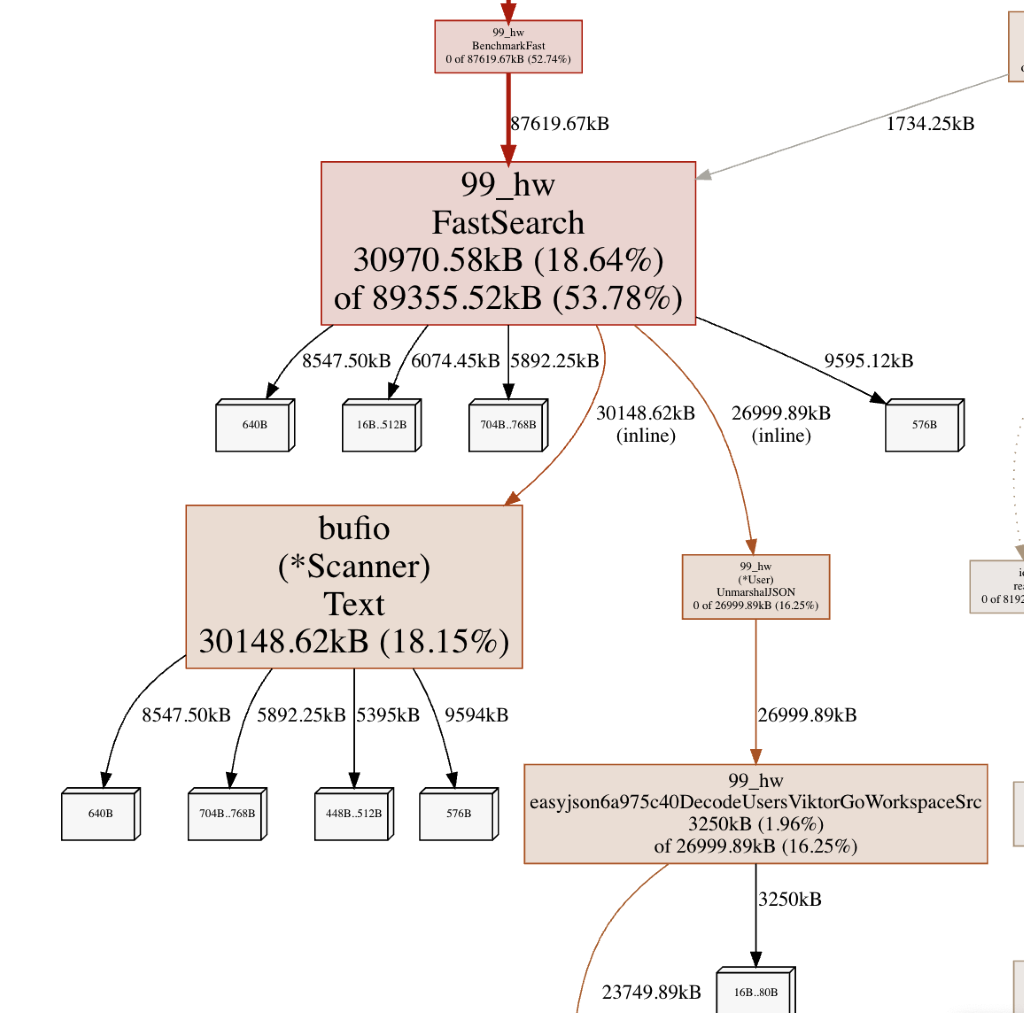




Сгенерируем код при помощи easyjson и посмотрим, что получилось:







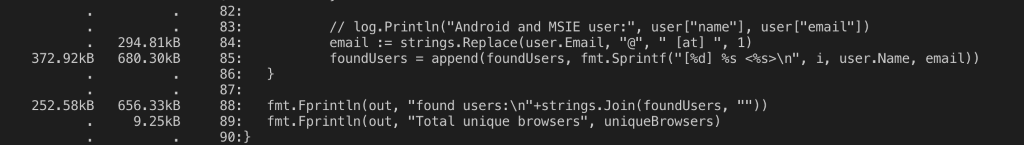
**BenchmarkSlow-8                2         674194085 ns/op        18903776 B/op     195791 allocs/op**

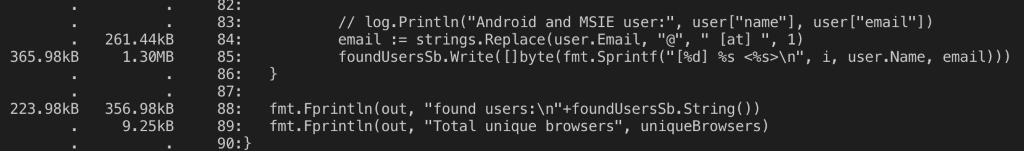
**BenchmarkFast-8               50          24549149 ns/op         1759399 B/op       9498 allocs/op**

По памяти прирост небольшой, а вот по скорости существенный: 0,11с вместо 1.13с! Тут мы наглядно видим, что кодогенерация намного эффективнее рефлексии.

1. **Использование Strings.Builder**

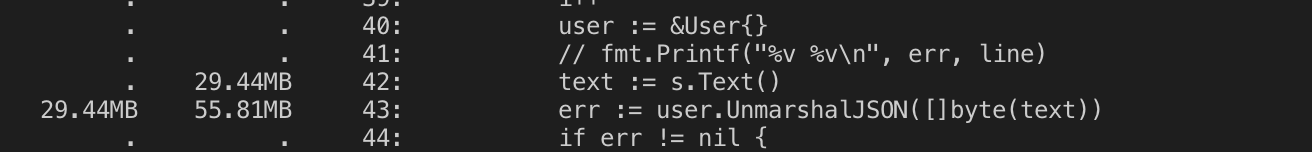
Также можно заменить слайс и append на strings.Builder, хотя в нашем случае разницы особо не будет(но будет разница между любым из этих двух подходов и конкатенацией строк):

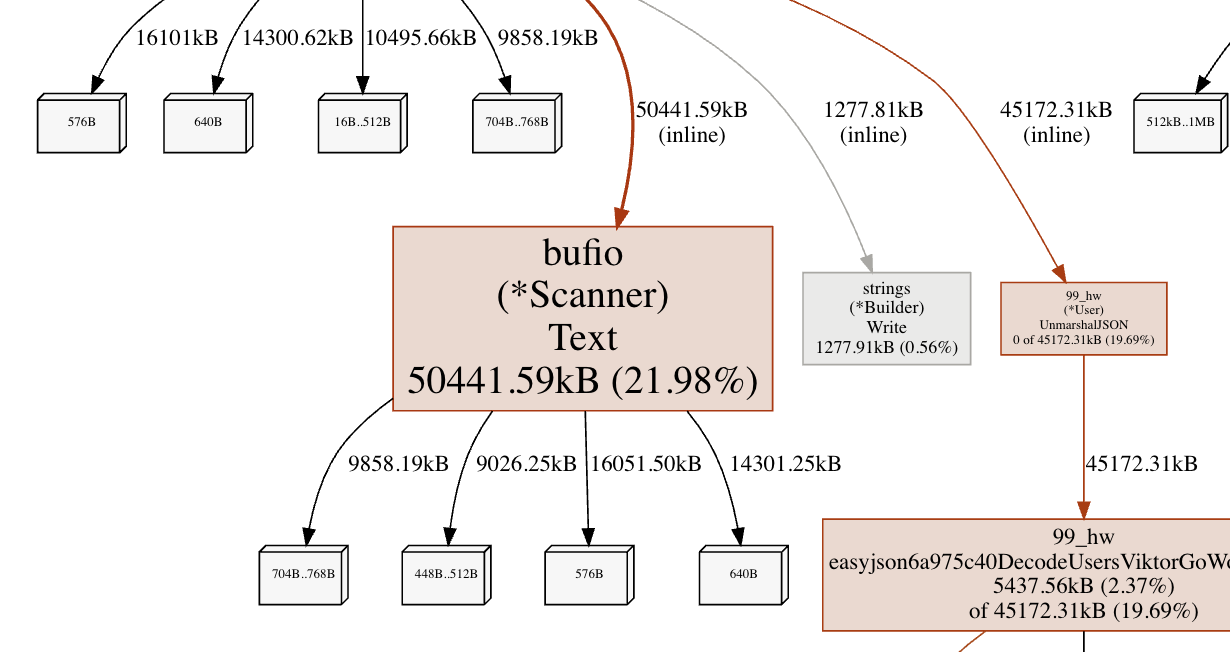




1. **Считывание байтов с помощью bufio.Scanner()**

Сейчас большую часть ресурсов уходит на Text(), что хорошо видно с помощью команды list FastSearch:

****

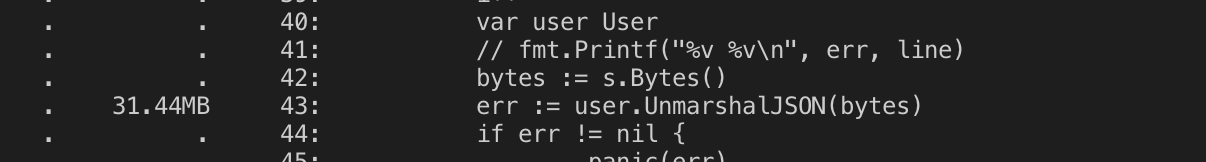


Сделаем так чтобы читать сразу в байтах без лишних конвертаций

**bytes := s.Bytes()**

**err := user.UnmarshalJSON(bytes)**

**Стало:**

****

**Результат:**

**BenchmarkSlow-8 2 652956442 ns/op 18993148 B/op 195791 allocs/op**

**BenchmarkFast-8 60 20565662 ns/op 586641 B/op 7579 allocs/op**

Всё! Полученный результат считаю приемлемым.