Введение в нагрузочное тестирование на K6

Автор - Кирилл Грищук (https://kirya522.tech/)

🔧 Шаг 1: Подготовка окружения

```
go mod init loadtest-demo
go mod tidy
```

Добавить код приложения main.go

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
)

// Имитация тяжёлой операции
func heavyComputation(userID string) string {
    sum := 0
    for i := 0; i < 1e8; i++ {
        sum += i + rand.Intn(100)
    }
    return fmt.Sprintf("User %s - Сумма: %d", userID, sum)
}
```

```
// Эндпоинт без кеша
func noCacheHandler(w http.ResponseWriter, r *http.Request) {
    userID := r.URL.Query().Get("user")
    if userID == "" {
        userID = "guest"
    }
    result := heavyComputation(userID)

    fmt.Fprintf(w, "Источник: без кеша | Результат: %s", result)
}

func main() {
    http.HandleFunc("/no-cache", noCacheHandler)

    fmt.Println("Сервер запущен на http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

Запустить сервер

```
go run main.go
```

Установи k6:

Подробнее об инструменте

```
brew install k6    # macOS
choco install k6   # Windows
winget install k6   # Windows
apt install k6     # Linux
```

🚀 Шаг 2: Базовый тест

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  vus: 20,
  duration: '10s',
};

export default function () {
  http.get('http://localhost:8080/no-cache');
  sleep(1);
}
```

Теория

Чем VUs отличаются от RPS?

- **VUs (виртуальные пользователи)** контролируют количество одновременно выполняемых потоков.
- **RPS (запросы в секунду)** — это фактическая метрика нагрузки, зависящая от времени выполнения запросов.

Пример зависимости:

- Если 10 VUs выполняют запросы, каждый из которых занимает 2 секунды, RPS будет равен:

$$RPS = \frac{\text{Количество VUs}}{\text{Время выполнения запроса}} = \frac{10}{2} = 5$$

Выполняем тест

```
k6 run load.js
```

🚨 Шаг 3: Анализируем результат

```
data_received..................: 17 kB  1.5 kB/s
data_sent......................: 7.1 kB 601 B/s
http_req_blocked...............: avg=392.48µs min=1µs      med=
http_req_connecting............: avg=189.46µs min=0s       med=
http_req_duration..............: avg=1.76s    min=999.55ms med=
  { expected_response:true }...: avg=1.76s    min=999.55ms med=
http_req_failed................: 0.00%  0 out of 81
http_req_receiving.............: avg=80.9µs   min=7µs      med=
http_req_sending...............: avg=40.39µs  min=2µs      med=
http_req_tls_handshaking.......: avg=0s       min=0s       med=
http_req_waiting...............: avg=1.76s    min=999.49ms med=
http_reqs......................: 81     6.824519/s
iteration_duration.............: avg=2.76s    min=2s       med=
iterations.....................: 81     6.824519/s
vus............................: 17     min=17      max=20
vus_max........................: 20     min=20      max=20
```

✅ Шаг 4: Оптимизируем код за счет кеша

Добавляем обработчик с кешом

```
package main

import (
```

```
    "fmt"
    "math/rand"
    "net/http"
    "sync"
)

var (
    cache      = make(map[string]string)
    cacheMutex = sync.RWMutex{}
)

// Имитация тяжёлой операции
func heavyComputation(userID string) string {
    sum := 0
    for i := 0; i < 1e8; i++ {
        sum += i + rand.Intn(100)
    }
    return fmt.Sprintf("User %s - Сумма: %d", userID, sum)
}

// Эндпоинт без кеша
func noCacheHandler(w http.ResponseWriter, r *http.Request) {
    userID := "guest"
    result := heavyComputation(userID)

    fmt.Fprintf(w, "Источник: без кеша | Результат: %s", result)
}

func withCacheHandler(w http.ResponseWriter, r *http.Request) {
    cacheMutex.RLock()
    item, found := cache["heavyResult"]
    cacheMutex.RUnlock()

    if found {
        fmt.Fprintf(w, "Источник: кеш | Результат: %s", item)
        return
```

```
    }

    result := heavyComputation("guest")
    cacheMutex.Lock()
    cache["heavyResult"] = result
    cacheMutex.Unlock()

    fmt.Fprintf(w, "Источник: расчёт | Результат: %s", result)
}

func main() {
    http.HandleFunc("/no-cache", noCacheHandler)
    http.HandleFunc("/cache", withCacheHandler)

    fmt.Println("Сервер запущен на http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

Добавляем еще один тест load-cached.js

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
    scenarios: {
      cached_users: {
        executor: 'constant-vus',
        exec: 'cached',
        vus: 300,
        duration: '10s',
      },
    },
};
```

```
export function cached() {
  http.get('http://localhost:8080/cache');
}
```

Запускаем

```
k6 run load-cached.js
```

Сравниваем результат

```
data_received..................: 173 MB 17 MB/s
data_sent......................: 72 MB  7.2 MB/s
http_req_blocked...............: avg=5.54µs   min=0s med=0s
http_req_connecting............: avg=290ns    min=0s med=0s
http_req_duration..............: avg=3.4ms    min=0s med=2.97ms
 { expected_response:true }...: avg=3.4ms    min=0s med=2.97ms m
http_req_failed................: 0.01%  98 out of 842707
http_req_receiving.............: avg=237.27µs min=0s med=0s
http_req_sending...............: avg=23.17µs  min=0s med=0s
http_req_tls_handshaking.......: avg=0s       min=0s med=0s
http_req_waiting...............: avg=3.14ms   min=0s med=2.9ms
http_reqs......................: 842707 84257.205366/s
iteration_duration.............: avg=3.53ms   min=0s med=2.99ms
iterations.....................: 842707 84257.205366/s
vus............................: 300    min=300          max=300
vus_max........................: 300    min=300          max=300
```

🔥Сервер обрабатывает в 12000 раз больше запросов

```
// Было
http_reqs.....................: 6.824519/s

// Стало
http_reqs.....................: 84257.205366/s
```