

# تمرين عملی چهارم

نویسندها:

کسری مجلل ۹۶۳۱۴۱۹

آرمین ذوالفقاری داریانی ۹۷۳۱۰۸۲

دانیال حمدی ۹۷۳۱۱۱۱

امیرمهدی زرین نژاد ۹۷۳۱۰۸۷

## پیش زمینه:

- تست بار (Load Test): با دانستن میزان بار روی سیستم خود، در شرایط نرمال و اوج، رفتار سیستم را زیر این مقادیر بار بررسی می‌کنیم، تا از درستی و کارایی مطلوب در سیستم اطمینان حاصل کنیم.
- تست استرس (Stress Test): در این نوع تست، به میزان بسیار بالاتری از بار نرمال سیستم (یا حتی بار در شرایط پیک) روی سیستم بار می‌گذاریم، تا محدودیت‌های سیستم را سنجیده، تا از پایداری سیستم در این شرایط اطمینان حاصل کنیم. باید توجه کنیم که میزان بار در این تست، از بار در شرایط اوج (پیک) سیستم هم با اختلاف بیشتر است. برای مثال، میزان باری که یک فروشگاه در «جمعه سیاه» دریافت می‌کند، می‌تواند میزانی برای تست استرس باشد، که این میزان چندین برابر میزان بار پیک سیستم در ایام دیگر سال است. مثال دیگری از تست استرس، پایداری در برابر حملات DDoS است.

- مفهوم Check: در هر Check، یک یا چندین درستی یک یا چندین شرط بررسی می‌کنیم. این شرط‌ها عموماً روی ابعاد مختلف ریسپانس گرفته شده از سرور هستند. برای مثال بررسی می‌کنیم که آیا status ریسپانس گرفته شده ۲۰۰ است یا خیر. تفاوت Check با مفهوم پرکاربرد assert در زبان‌های برنامه‌نویسی، این است که پاس نشدن چک برخلاف assert، منجر به توقف اجرای آن تست نمی‌شود، به عبارت دیگر Fatal نیست.

- مفهوم Threshold: معیارهای به صورت pass/fail هستند که انتظارات ما از کارایی سیستم را بیان می‌کنند. برای مثال، ممکن است یکی از انتظارات ما از کارایی سیستم این باشد که بیشتر از ۱٪ خطای تولید نشود. پاس شدن یا نشدن این مورد را می‌توان به صورت یک Threshold بیان کرد. برای هر (برخلاف Check) به کمک فیلد abortOnFailure می‌توان بیان کرد که با برآورده نشدن Threshold برنامه متوقف شود (یا نشود).

## تست بار واحد چکها

- در response‌های لیست کروکودیل‌های پابلیک، (API شماره‌ی ۵)، HTTP status code ۲۰۰ برابر است.

```
export const options = {
  stages: [
    {duration: '15s', target: 150},
    {duration: '30s', target: 150},
    {duration: '15s', target: 0},
  ],
};

export default function () {
  const res = http.get('https://test-api.k6.io/public/crocodiles/');
  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}
```

خروجی:

همان طور که در تصویر بالا مشخص است، تست با vu ۱۵۰ که به صورت تدریجی ساخته می شوند، انجام شده است. همچنین بنا بر تصویر، چک استاتوس کد ریسپانس برای ۱۰۰٪ درخواستها با موفقیت پاس شده است.

## تست بار سناریو چکهای سناریوی اول

- در response های لیست کروکودیل های پابلیک، API شماره ۵، HTTP status code برابر ۲۰۰ است.

- در response های API شماره ۶، HTTP status code برابر ۲۰۰، و کروکودیل فیلد های مورد نظر را داراست.

```
export function scenario_1() {
  let publicCrocossRes = http.get(PUBLIC_CROCOS_URL)
  check(publicCrocossRes, {
    "public crocos response status is 200": (r) => r.status === 200, // Unauthorized request will receive HTTP 401
  });
  let numCrocodiles = publicCrocossRes.json().length

  const selectedCrocRes = http.get(PUBLIC_CROCOS_URL + Math.round(randomIntBetween(1, numCrocodiles)));
  const crocoFields = ['id', 'name', 'sex', 'date_of_birth', 'age']
  check(selectedCrocRes, {
    "selected croco response status is 200": (r) => r.status === 200, // Unauthorized request will receive HTTP 401
    "selected croco has all croco fields": (r) => hasAllFields(r, crocoFields), // k6 doesn't have schema validation
  });
}
```

## چکهای سناریوی دوم

- برای دریافت توکن احراز هویت (لاگین)، استاتوس کد، و وجود فیلد های access token و refresh token بررسی شده اند.

```

function login() {
    let credentials = loginData.users[0];

    let loginRes = http.post(LOGIN_URL, {
        username: credentials.username,
        password: credentials.password
    });
    check(loginRes, {
        "login response status is 200": (r) => r.status === 200,
        "refresh token is present": (r) => 'refresh' in r.json(),
        "access token is present": (r) => 'access' in r.json()
    });

    return loginRes
}

```

- برای دریافت لیست کروکودیل‌های فروشگاه، استاتوس کد بررسی شده است.

- برای ساخت ۱۰ کروکودیل در فروشگاه، چک می‌کنیم که استاتوس کد ریسپانس برابر ۲۰۱

(CREATED) بوده، و همچنین آی‌دی آبجکت ساخته شده در ریسپانس موجود باشد.

```

function create10CocosForStore(params) {
    let storeNumCocos = getStoreNumCocos(params)

    while (storeNumCocos < 10) {
        let newCroco = crocosData.crocodiles[storeNumCocos]

        let createCrocoRes = http.post(MY_CROCOS_URL, newCroco, params)
        check(createCrocoRes, {
            "create status code is 201": (r) => r.status === 201,
            "new croco assigned id is present": (r) => 'id' in r.json(),
        });

        storeNumCocos = getStoreNumCocos(params)
    }
}

```

## Threshold های تعریف شده

اولین Threshold را روی درصد خطای درخواست‌ها به این صورت تعریف می‌کنیم که «نرخ میزان خطاهای

باید کمتر از ۱٪ باشد.»

دومین Threshold را روی زمان دریافت رسپانس به این صورت تعریف می‌کنیم که ۹۵٪ درخواست‌ها

باید کمتر از ۱ ثانیه جواب بگیرند.

```
export const options = {
  thresholds: {
    http_req_failed: ['rate<0.01'], // http errors should be less than 1%
    http_req_duration: ['p(95)<1000'], // 95% of requests should be below 200ms
  },
  scenarios: {
    scenario_1: {executor: 'ramping-arrival-rate'...},
    scenario_2: {executor: 'shared-iterations'...},
  },
};
```

## خروجی تست سناریوها: کانفیگ بار پایین

- منظور از vu تعداد کاربران مجازی‌ای است که درخواست ارسال می‌کنند، iterations هم مجموع تعداد درخواستی است که این کاربران ارسال خواهند کرد. start time زمان شروع ارسال درخواست برای هر سناریو را مشخص می‌کند. graceful stop نشان می‌دهد که با بروز خطأ، و گرفتن سیگنال توقف، فرآیند آن سناریو، چند ثانیه زمان برای clean up و خروج دارد.
- کانفیگ بار پایین به صورت زیر است.

```
scenario_1: {  
    // name of the executor to use  
    executor: 'ramping-arrival-rate',  
  
    // common scenario configuration  
    startTime: '0s',  
    gracefulStop: '3s',  
  
    // executor-specific configuration  
    stages: [  
        {duration: '15s', target: 5},  
        {duration: '30s', target: 10},  
        {duration: '15s', target: 0},  
    ],  
    preAllocatedVUs: 100, // how large the initial pool of VUs would be  
  
    // what to execute  
    exec: 'scenario_1'  
},  
scenario_2: {  
    // name of the executor to use  
    executor: 'shared-iterations',  
  
    // common scenario configuration  
    startTime: '5s',  
    gracefulStop: '3s',  
  
    // executor-specific configuration  
    vus: 5,  
    iterations: 10,  
    maxDuration: '5s',  
  
    // what to execute  
    exec: 'scenario_2'  
},
```

که در سناریوی اول، طی ۱۵ ثانیه از ۰ vu به ۵ vu رسیده، ۳۰ ثانیه آنها را نگه داشته، و سپس طی ۱۵ ثانیه vu ها را به صفر می‌رساند. در سناریوی دوم به طور ثابت از ۵ vu و ۱۰ ایتریشن، با بیشینه duration ۵ ثانیه استفاده شده است.

```

execution: local
script: 2-scenario.js
output: InfluxDBv1 (http://localhost:8086)

scenarios: (100.00%) 2 scenarios, 105 max VUs, 1m3s max duration (incl. graceful stop):
  * scenario_1: Up to 50.00 iterations/s for 1m0s over 3 stages (maxVUs: 100, exec: scenario_1, gracefulStop: 3s)
  * scenario_2: 10 iterations shared among 5 VUs (maxDuration: 5s, exec: scenario_2, startTime: 5s, gracefulStop: 3s)

WARN[0020] Insufficient VUs, reached 100 active VUs and cannot initialize more executor=ramping-arrival-rate scenario=scenario_1

running (1m00.1s), 000/105 VUs, 2156 complete and 0 interrupted iterations
scenario_1 ✓ [=====] 000/100 VUs 1m0s 01 iters/s
scenario_2 ✓ [=====] 5 VUs 3.3s/5s 10/10 shared iters

✓ public crocos response status is 200
✓ selected croco response status is 200
✓ selected croco has all croco fields
✓ login response status is 200
✓ refresh token is present
✓ access token is present
✓ login successful

checks.....: 100.00% ✓ 6478 x 0
data_received.....: 3.6 MB 60 kB/s
data_sent.....: 990 kB 17 kB/s
dropped_iterations.....: 103 1.715185/s
http_req_blocked.....: avg=12.97ms min=1μs med=6μs max=923.92ms p(90)=11μs p(95)=21μs
http_req_connecting.....: avg=4.26ms min=0s med=0s max=301.84ms p(90)=0s p(95)=0s
✓ http_req_duration.....: avg=583.92ms min=242.09ms med=568.53ms max=1.52s p(90)=881.23ms p(95)=968.12ms
  { expected_response:true }.....: avg=583.92ms min=242.09ms med=568.53ms max=1.52s p(90)=881.23ms p(95)=968.12ms
✓ http_req_failed.....: 0.00% ✓ 0 x 6468
http_req_receiving.....: avg=137.08μs min=16μs med=111μs max=5.54ms p(90)=247μs p(95)=320μs
http_req_sending.....: avg=40.72μs min=4μs med=31μs max=1.02ms p(90)=68μs p(95)=101.64μs
http_req_tls_handshaking.....: avg=8.67ms min=0s med=0s max=581.32ms p(90)=0s p(95)=0s
http_req_waiting.....: avg=583.74ms min=241.81ms med=568.34ms max=1.52s p(90)=881.07ms p(95)=967.9ms
http_reqs.....: 6468 107.706986/s
iteration_duration.....: avg=1.79s min=778.9ms med=1.83s max=3.31s p(90)=2.37s p(95)=2.53s
iterations.....: 2156 35.902329/s
vus.....: 100 min=100 max=105
vus_max.....: 105 min=105 max=105

```

در میزان بار کم، تمامی Check‌ها با موفقیت گذرانده شده‌اند، ۳.۶ مگابایت داده دریافت شده، به طور میانگین

۱۲ میلی‌ثانیه

## کانفیگ بار زیاد

باید توجه کنیم که در این قسمت، با محدودیت منابع شبکه‌ی سیستم درخواست‌زننده طرف هستیم. سیستم عامل‌های

Unix محدودیتی برای تعداد سوکت باز دارند، که به عنوان Bottleneck هنگام درخواست زیاد عمل کرده، و منجر

به Fail شدن ریکوئست‌ها می‌شود. در این [لينك](#) از داکیومنتیشن k6 این موضوع به تفصیل مطرح شده است.

برای مثال زمانی که تعداد vus‌ها را برای سناریوی اول به ۱۰۰، و برای سناریوی دوم به ۸۰ می‌رسانیم، با خروجی

زیر رو به رو می‌شویم.

```

cktrace
WARN[0054] Request Failed error="Get \"https://test-api.k6.io/public/crocodiles/\": dial tcp 107.22.175.151:443: socket: too many open files"
ERROR[0054] the body is null so we can't transform it to JSON - this likely was because of a request error getting the response
running at reflect.methodValueCall (native)
scenariot scenario_1 (file:///Users/Danial/WebstormProjects/k6-testing/2-scenario.js:98:28(19)) executor=ramping-arrival-rate scenario=scenario_1 source=static cktrace
WARN[0058] Request Failed error="Get \"https://test-api.k6.io/public/crocodiles/\": dial tcp 107.22.175.151:443: socket: too many open files"
ERROR[0058] the body is null so we can't transform it to JSON - this likely was because of a request error getting the response
running at reflect.methodValueCall (native)
scenariot scenario_1 (file:///Users/Danial/WebstormProjects/k6-testing/2-scenario.js:98:28(19)) executor=ramping-arrival-rate scenario=scenario_1 source=static cktrace

running (1m00.4s), 000/244 VUs, 4985 complete and 0 interrupted iterations
scenario_1 ✓ [=====] 000/164 VUs 1m0s 002 iters/s
scenario_2 ✓ [=====] 80 VUs 0m40.6s/1m0s 500/500 shared iters

  x public crocos response status is 200
    ✓ 48% - ✘ 21.70 / ✘ 2315
    ✓ selected croco response status is 200
    ✓ selected croco has all croco fields
    ✓ login response status is 200
    ✓ refresh token is present
    ✓ access token is present
    ✓ login successful

  checks .....: 78.61% ✘ 8510 x 2315
  data_received ..: 5.8 MB 96 kB/s
  data_sent .....: 1.5 MB 25 kB/s
  dropped_iterations ..: 14 0.231884/s
  http_req_blocked ..: avg=19.99ms min=0s med=5µs max=1.88s p(90)=11µs p(95)=21µs
  http_req_connecting ..: avg=6.4ms min=0s med=0s max=364.21ms p(90)=0s p(95)=0s
  x http_req_duration ..: avg=947.01ms min=0s med=276.86ms max=8.01s p(90)=3.63s p(95)=4.85s
    { expected:response:true } ..: avg=1.22s min=235.83ms med=290.61ms max=8.01s p(90)=4.01s p(95)=5.05s
  x http_req_failed ..: 22.42% ✘ 2315 ✘ 8010
  http_req_receiving ..: avg=110.92µs min=0s med=107µs max=1.09ms p(90)=225.6µs p(95)=262µs
  http_req_sending ..: avg=33.17µs min=0s med=29µs max=842µs p(90)=61µs p(95)=88µs
  http_req_tls_handshaking ..: avg=13.56ms min=0s med=0s max=1.6s p(90)=0s p(95)=0s
  http_req_waiting ..: avg=946.87ms min=0s med=276.71ms max=8.01s p(90)=3.63s p(95)=4.85s
  http_reqs .....: 10325 171.014595/s
  iteration_duration ..: avg=2s min=185.12µs med=803.38ms max=17.09s p(90)=6.76s p(95)=8.79s
  iterations .....: 4985 82.567337/s
  vus .....: 164 min=150 max=244
  vus_max .....: 244 min=230 max=244

ERROR[0061] some thresholds have failed

```

خطای socket: too many open files نشان از عدم امکان باز کردن سوکت جدید دارد، که باعث می‌شود امکان

ریکوئستها Fail شده و در نتیجه همان‌طور که در شکل مشخص است درصد پاس شدن چک اول، به ۴۸٪

برسد.

با آزمایش و بررسی محدودیت‌های سیستم عامل خود، می‌توانیم تعداد بیشینه‌ی VU‌های ممکن برای سیستم عامل را پیدا کرده، و با این تعداد تست را بار دیگر تکرار کنیم. باید توجه کنیم که هدف از تست، بررسی کارایی (تست بار) یا پایداری (تست استرس) سرور است، نه دستگاهی که تست روی آنها اجرا می‌شود؛ پس نباید بگذاریم که تست، دستگاه ما باشد.

Bottleneck بیشینه‌ی توان سیستم عامل ما، برابر کانفیگ زیر بود.

```

scenario_1: {
    // name of the executor to use
    executor: 'ramping-arrival-rate',

    // common scenario configuration
    startTime: '0s',
    gracefulStop: '3s',

    // executor-specific configuration
    stages: [
        {duration: '15s', target: 30},
        {duration: '30s', target: 30},
        {duration: '15s', target: 0},
    ],
    preAllocatedVUs: 150, // how large the initial pool of VUs would be
    maxVUs: 300, // if the preAllocatedVUs are not enough, we can initialize more

    // what to execute
    exec: 'scenario_1'
},
scenario_2: {
    // name of the executor to use
    executor: 'shared-iterations',

    // common scenario configuration
    startTime: '5s',
    gracefulStop: '3s',

    // executor-specific configuration
    vus: 30,
    iterations: 500,
    maxDuration: '1m',

    // what to execute
    exec: 'scenario_2'
}

```

که در آن، سناریوی اول ۳۰ vus و سناریوی دوم از ۳۰ کاربر که مجموعاً ۵۰۰ بار سناریو رو اجرا می‌کنند، استفاده می‌کند. این کانفیگ منجر به خروجی زیر شده است.

همان طور که قابل مشاهده است، ۱۰۰٪ چک‌ها پاس شده‌اند، ۴۳ مگابایت داده دریافت شده، ۱.۱ مگابایت داده ارسال شده و زمان طی شده برای ریکوئست‌ها به طور میانگین ۴۱۷ میلی‌ثانیه است. در این تست threshold که روی زمان طی شده برای درخواست‌ها (http-req\_duration) گذاشته بودیم، fail شده است. ۹۵٪ درخواست‌ها در زمانی کمتر از ۱۳ ثانیه انجام شده‌اند، در صورتی که برای برآورده شدن Threshold ما، این میزان باید کمتر از 1s باشد. البته دلیل اصلی این کندی، محدودیت‌های شبکه‌ای کلاینت است، نه سرور.

## مصورسازی به وسیله Grafana و Influxdb

ابتدا Influxdb را نصب کرده و آن را روی یک پورت (به طور پیش فرض ۸۰۸۶) بالا می آوریم، بدین صورت می توانیم خروجی k6 را با دستور زیر روی یک دیتابیس Influx بریزیم.

```
k6 run --out influxdb=http://localhost:8086/TARGETDB SCRIPT.js
```

که در آن TARGETDB نام دیتابیس هدف است.

سپس می‌توانیم روی پنل Grafana، این دیتابیس را از مسیر Configuration/Data sources/ Add Data source به عنوان یک Data Source اضافه کنیم.

حال می‌توانیم یک Dashboard بسازیم که رکوردهای داخل Data source را به صورت مصور نشان دهد. برای این کار از [این بنل از بیش تعریف شده](#) استفاده می‌کنیم. از مسیر Create/ Import این Dashboard را ایمپورت کرده، و به عنوان دیتابورس هم دیتابورسی را که در قسمت قبل تعریف کردیم، قرار می‌دهیم.

حال می‌توانیم متريک‌های مورد نظر را به صورت مصور ببینیم.

مثال، از خروجی Grafan برای تست بار سناریو، برای بار سبک:

