Технічні науки

УДК 004.428.4

Крупельницький Леонід Віталійович
кандидат технічних наук,
доцент кафедри обчислювальної техніки
Вінницький національний технічний університет
Ткаченко Олександр Миколайович
кандидат технічних наук,
доцент кафедри обчислювальної техніки
Вінницький національний технічний університет
Комаров Володимир Леонідович
аспірант кафедри обчислювальної техніки
Вінницького національного технічного університету

ПРОФІЛЮВАННЯ КОДУ GOLANG-ПРОЕКТУ ТА ВИРІШЕННЯ ПРОБЛЕМ З ВИДІЛЕННЯМ ПАМ'ЯТІ

Анотація. *У даній статті розглядаються програмі засоби та методики пошуку проблем у програмних застосунках, написаних мовою програмування Golang. Зокрема розглядається методика пошуку проблем з неефективним виділенням пам'яті. Також описується спосіб вирішення та попередження виникнення проблем з виділенням пам'яті у застосунках.*

Ключові слова: *виділення пам'яті, golang, рефакторинг, профілювання, аналіз, vegeta, pprof.*

Вступ. Будь-які програмні продукти мають проблеми - як очевидні, так і приховані. Зокрема, приховані проблеми важко локалізувати простим налагодженням програми.

На допомогу приходить тестування під навантаженням та профілювання. Такі методики дозволяють виявити найчастіші проблеми, що можуть виникнути лише під достатньою кількістю навантаження [1,2].

В даній статті описано процес профілювання, аналізу вихідного коду, вирішення знайдених проблем, та перевірки результату вирішення проблем. Для прикладу профілювання можна використати будь-який HTTP сервіс, написаний мовою Golang. Тому, було обрано проект з відкритим кодом Flipt [3]. Він дозволяє запускати A/B-тести за допомомогою простого WEB-інтерфейсу.

Генерування великого об'єму трафіку

Перед тим, як розпочати профілювання, необхідно згенерувати великий об'єм трафіку, що навантажить додаток і допоможе побачити деяку його шаблонну поведінку. Не кожен проект має достатній об'єм трафіку в "бойовому" оточенні, що дозволив би оцінити роботу проекту під навантаженням. В такому разі можна застосувати різноманітні інтрументи для тестування проектів навантаженням. Одним з таких є Vegeta [4]. Автори проекту запевняють, що Vegeta - це універсальний HTTP-інструмент для тестування навантаженням. Ідея проекту утворилась за необхідності навантажувати HTTP-сервіси великою кількістю запитів, що надсилаються до сервісів з заданою частотою.

Проект Vegeta дозволяє створювати безперервний потік запитів у додаток, тому підходить для генерації достатнього об'єму трафіку. Такими запитами можна тестувати додаток стільки, скільки потрібно для того, щоб отримати такі показнити (метрики) як виділення / використання heap-пам'яті, особливості роботи горутин, час, затрачений на збір непотребу.

Оптимальна конфігурація запуску Vegeta для тестування проекту:

echo 'POST http://localhost:8080/api/v1/evaluate' | vegeta attack -rate 1000 -duration 1m -body evaluate.json

Команда запускає Vegeta в режимі attack, підправляючи HTTP POST-запити зі швидкістю 1000 запитів в секунду (що є доволі серйозним навантаженням) на протязі хвилини. JSON-дані, що відправляються в тілі кожного запиту не важливі. Вони потрібні для правильного формування тіла запиту.

Вимірювання

Наступним кроком, маючи інструмент для генерування достатньо великого об'єму трафіку, необхідно знайти спосіб вимірювання тієї взаємодії, яке спричиняв трафік на працюючий сервіс. На щастя, в пакеті Golang є хороший стандартний набір інструментів, що дозволяють вимірювати швидкодію додатків. Одним з таких є пакет pprof [5].

HTTP-маршрутизатор проекту, що тестується, реалізований за допомогою проекту go-chi/chi [6], тому не складно додати pprof, скориставшись відповідним проміжним обробником [7].

Для зручності, в одному вікні командного рядка необхідно запустити тестований проект з вбудованим проміжним обробником, а в іншому - генерування трафіку за допомогою Vegeta. Ще одне вікно необхідне для збору і аналізу даних профілювання heap-пам'яті:

pprof -http=localhost:9090 localhost:8080/debug/pprof/heap

Інструмент Google - pprof здатен візуалізувати дані профілювання в браузері (Рис. 1).

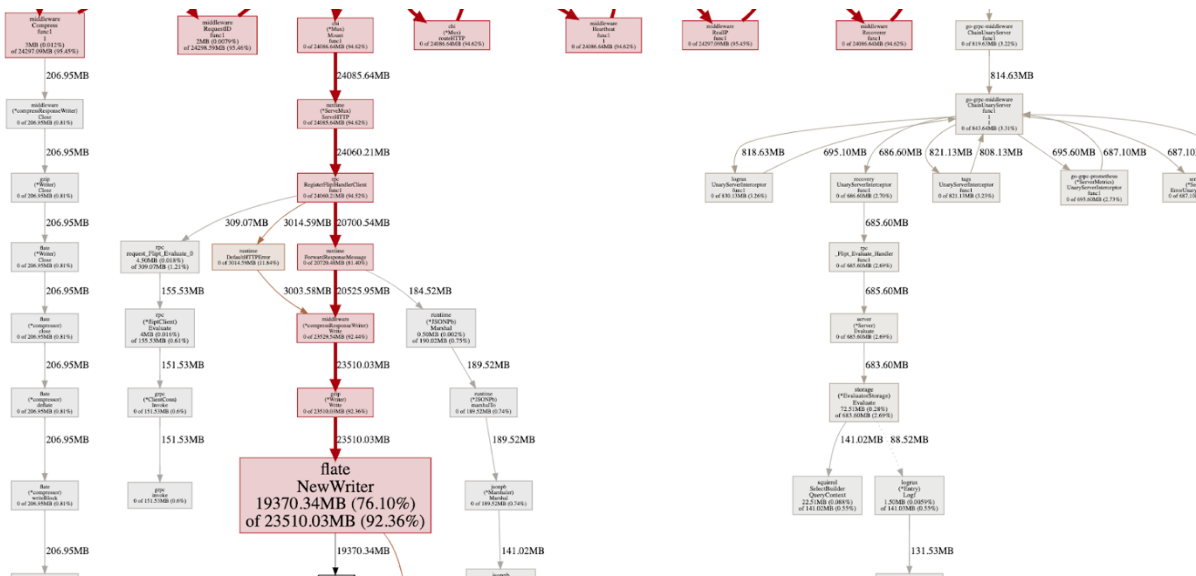

Рис. 1. Аналіз результатів профілювання пам'яті

Спочатку необхідно перевірити показники inuse_objects та inuse_space, щоб зрозуміти, що відбувається в heap-пам'яті. Проте, з даними показниками все в нормі. Далі необхідно перевірити показними allow_objects та alloc_space. А саме - на блок flate.NewWriter. Згідно з результатом профілювання, даний блок, на протязі хвилини сумарно виділив 19 370 МБ пам'яті. Отже, починаючи з даного блоку, пам'ять виділялась занадто часто, що призвело до таких об'ємів її виділення. Наступним кроком необхідно виявити блок, що відповідає за некоректну роботу з пам'яттю. Згідно зі схемою flate.NewWriter викликається з блоку gzip.(*Writer).Write, що, в свою чергу, викликається з middleware.(*compressResponseWriter).Write. Проблема знаходиться в одному з проміжних обробників інструменту, що використовується для реалізації HTTP-маршрутизації, ймовірно для стиснення відповідей HTTP, про що свідчить пакет gzip в ланцюжку виконання. Після аналізу вихідного коду тестованого проекту знайдено стрічку, що використовує обробник для стиснення HTTP-відповідей:

r.Use(middleware.Compress(gzip.DefaultCompression))

Щоб перевірити вірність знайденого вихідного коду, необхідно закоментувати стрічку, та повторити процес профілювання. Повторивши процес профілювання, об'єми сумарного виділення пам'яті значно зменшились.

Вирішення проблеми

Щоб дізнатись причину значного виділення пам'яті, необхідно проаналізувати вихідний код пакету, що використовується для реалізації HTTP-маршрутизації - Chi [6]. Але, спочатку, необхідно дізнатись версію пакету, що була використана під час профілювання:

```
go list -m all | grep chi
github.com/go-chi/chi v3.3.4+incompatible
```

Проаналізувавши вихідний код пакету за знайденою версією, знайдено метод, що відповідав за стиснення HTTP-відповідей:

```
func encoderDeflate(w http.ResponseWriter, level int) io.Writer {
  dw, err := flate.NewWriter(w, level)
  if err != nil {
    return nil
  }
  return dw
}
```

Подальший аналіз вихідного коду виявив, що метод flate.NewWriter, за допомогою проміжного обробника, викликався для кожної відповіді HTTP, що тягнуло за собою велику кількість операцій з виділення пам'яті, що було знайдено під час профілювання.

Перш ніж розпочинати пошук вирішення проблем необхідно перевірити її наявність в найсвіжішому вихідному коді пакету маршрутизації:

```
go get -u -v "github.com/go-chi/chi@master"
```

Встановивши останню версію вихідного коду, необхідно повторити профілювання (Рис. 2).

Flat	Flat%	Sum%	Cum	Cum%	Name
154.25MB	6.54%	6.54%	194.78MB	8.26%	compress/flate.NewWriter
114.51MB	4.86%	11.40%	114.51MB	4.86%	github.com/lann/ps.(*tree).clone
87.01MB	3.69%	15.09%	854.61MB	36.24%	github.com/markphelps/flipt/storage.(*EvaluatorStorage).Evaluate
86.03MB	3.65%	18.73%	86.03MB	3.65%	bytes.makeSlice
81.02MB	3.44%	22.17%	81.02MB	3.44%	github.com/sirupsen/logrus.(*Entry).WithFields
76.01MB	3.22%	25.39%	76.01MB	3.22%	strings.genSplit
60.52MB	2.57%	27.96%	60.52MB	2.57%	golang.org/x/net/http2.(*Framer).readMetaFrame.func1
59.02MB	2.50%	30.46%	59.02MB	2.50%	reflect.mapassign
59.01MB	2.50%	32.96%	101.02MB	4.28%	github.com/golang/protobuf/jsonpb.jsonProperties
51.02MB	2.16%	35.13%	205.54MB	8.72%	github.com/sirupsen/logrus.(*TextFormatter).Format
40.03MB	1.70%	36.83%	40.03MB	1.70%	compress/flate.(*compressor).initDeflate
40.01MB	1.70%	38.52%	40.01MB	1.70%	reflect.New
35.51MB	1.51%	40.03%	35.51MB	1.51%	reflect.packEface
35.51MB	1.51%	41.53%	50.01MB	2.12%	github.com/golang/protobuf/proto.(*Properties).setFieldProps
32.01MB	1.36%	42.89%	32.01MB	1.36%	google.golang.org/grpc/internal/transport.(*decodeState).addMetadata
30.50MB	1.29%	44.18%	92.01MB	3.90%	fmt.Sprint

Рис. 2. Повторний аналіз результатів профілювання пам'яті

Згідно з результатами профілювання, блок, що відповідав за велику кількість операцій виділення пам'яті відпрацьовує зі значно меншою кількістю таких операцій, та сумарно виділяє 194 МБ.

Висновок. За результатом аналізу проекту з відкритим кодом, знайдено проблеми зі швидкодією, зокрема в використаних пакетах з відкритим кодом. Це говорить про те, що навіть популярні проекти з відкритим кодом мають проблеми. Проте, активне суспільство, що займається розвитком таких проектів, сприяє досить швидкому вирішенню таких проблем.

Не обов'язкова наявність реальних навантажень на сервіси, для перевірки їх на здатність підтримувати роботу навіть під час величезної кількості запитів. Завдяки спеціалізованим інструментам, на прикладі Vegeta, цілком можливо генерувати достатню кількість запитів, для тестування сервісів під навантаженням.

Інструменти мови Golang дозволяють протестувати та проаналізувати в повній мірі проекти, знайти слабкі місця та швидко усунути їх.

Література

1. Tsoukalos M. Mastering Go: Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures, 2nd Edition. / M. Tsoukalos // Packt Publishing Ltd. 2019 [529-542].
2. Profiling Go Programs. URL: https://blog.golang.org/pprof
3. Flipt – a modern feature flag solution. URL: https://github.com/markphelps/flipt
4. Vegeta – a versatile HTTP load testing tool. URL: https://github.com/tsenart/vegeta
5. Pprof – a tool for visualization and analysis of profiling data. URL: https://github.com/google/pprof
6. Chi - a lightweight, idiomatic and composable router for building Go HTTP services. URL: https://github.com/go-chi/chi
7. Profiler – a convenient subrouter used for mounting net/http/pprof. URL: https://github.com/go-chi/chi/blob/master/middleware/profiler.go