DEV4 : profiling et plus si affinités

loig.jezequel@univ-nantes.fr

Constat de départ

Ce qu'on a fait jusqu'à présent

1. Écriture d'un programme
2. Validité :
   - écriture d'un jeu de tests,
   - correction du programme
3. Efficacité :
   - utilisation de benchmarks,
   - ???

Que faire si un programme n'est pas efficace ?

- Comment savoir d'où vient cette inefficacité ?
- Comment savoir quoi modifier ?

Profiling (profilage ?)

Une solution : le profiling

- Lien entre les fonctions (qui appelle qui)
- temps de calcul utilisé pour chaque fonction
- visualisation des points de contention d'un programme.

Intérêt

Permet de savoir les points du programmes qui sont coûteux, ceux sur lesquels il est intéressant de consacrer du temps de développement pour optimiser le code.

Profiling en Go, principe

Principe général pour construire un profile

- Arrêt du programme toutes les quelques ms,
- lecture du contenu de la pile d'exécution,
- enregistrement des fonctions qui sont en cours,
- enregistrement des fonctions qui sont en attente (de retours d'autres fonctions).

Le profile obtenu

Un ensemble de points (*samples*) indiquant les fonctions utilisées à différents moments (très nombreux), permet d'obtenir :

- une approximation du temps pendant lequel chaque fonction est active,
- une approximation du temps pendant lequel chaque fonction attend le retour d'une autre fonction.

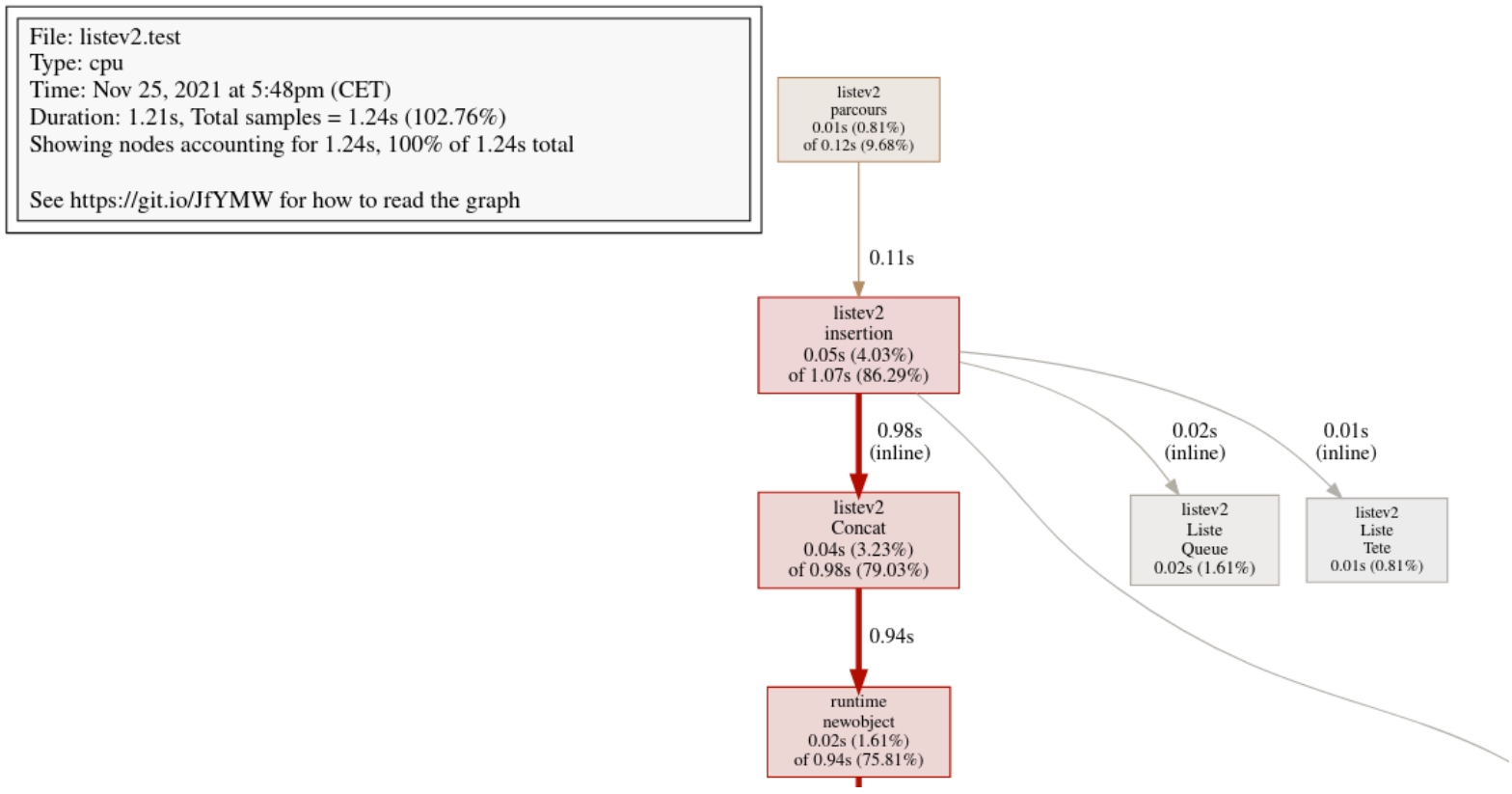
Résultat d'un profiling en Go

```
File: listev2.test
Type: cpu
Time: Nov 25, 2021 at 5:48pm (CET)
Duration: 1.21s, Total samples = 1.24s (102.76%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top10
Showing nodes accounting for 960ms, 77.42% of 1240ms total
Showing top 10 nodes out of 80
      flat  flat%   sum%        cum   cum%
     520ms 41.94% 41.94%      920ms 74.19%  runtime.mallocgc
     150ms 12.10% 54.03%      210ms 16.94%  runtime.heapBitsSetType
      60ms  4.84% 58.87%       60ms  4.84%  runtime.nextFreeFast (inline)
      50ms  4.03% 62.90%     1070ms 86.29%  listev2.insertion
      40ms  3.23% 66.13%      980ms 79.03%  listev2.Concat (inline)
      40ms  3.23% 69.35%       60ms  4.84%  runtime.heapBitsForAddr
      30ms  2.42% 71.77%       30ms  2.42%  runtime.acquirem (inline)
      30ms  2.42% 74.19%       30ms  2.42%  runtime.procyield
      20ms  1.61% 75.81%       20ms  1.61%  listev2.Liste.Queue (inline)
      20ms  1.61% 77.42%       20ms  1.61%  runtime.arenaIndex (inline)
(pprof)
```

Informations importantes (une ligne, une fonction)

- **flat** temps d'activité de la fonction
- **sum** somme des temps d'activité des fonctions
- **cum** temps d'activité + temps d'attente de retours

Résultat d'un profiling en Go, version graphique



Informations importantes (une boîte, une fonction)

- taille de la boîte (flat)
- couleur de la boîte (cum)
- flèches (appels de fonctions et temps d'attente)

Obtenir et visualiser un profile

Obtenir un profile (benchmark XXX, module YYY)

go test -cpuprofile=monprofile.prof -bench=XXX

Remarque

On peut aussi obtenir un profile sans benchmarks, mais ça demande d'outiller le programme. Une bonne bibliothèque pour ça : `https://github.com/pkg/profile`

Visualiser ce profile

go tool pprof YYY.test monprofile.prof

Résultat

Ouvre une interface dans laquelle on peut entrer des commandes, comme par exemple :

- top10 (10 fonctions qui ont la plus grande valeur pour flat)
- web (version graphique du profile dans un navigateur)

Et maintenant, un exemple :
différentes implantations des listes

Documentation

Pourquoi documenter son code ?

- savoir le maintenir même longtemps après l'avoir écrit
- permettre à d'autres de le comprendre
- permettre à d'autres de l'utiliser (sinon il ne sert à rien)

Comment documenter son code ?

- commentaires
  - \+ code et documentation sont liés (plus simple à maintenir)
  - \- nécessité de parcourir le code pour accéder à la documentation
- documentation externe
  - \+ documentation accessible sans lire le code
  - \- plus compliqué à maintenir (deux endroits différents)

La documentation en Go

Principe

- sous formes de commentaires
- qui permettent d'extraire une documentation externe

Mise en œuvre

On commente en général :

- paquets, fonctions, structures, variables, constantes
- en priorité s'ils sont visibles à l'extérieur du paquet
- en mettant un commentaire juste au dessus de ce qu'on commente

Les autres commentaires, qui ne font pas partie de la documentation (explication d'implantations particulières, d'algorithmes, etc), sont placés ailleurs (dans le code).

Extraire la documentation d'un module

En ligne de commande

go doc -all

Dans un navigateur

godoc -http=:6060 -goroot=/usr/share/go

- attention, ici c'est godoc sans espace (pas comme avant)
- il faut avoir préalablement installé godoc
  (utiliser go get golang.org/x/tools/cmd/godoc)
- démarre un serveur web accessible sur le port 6060
- `http://localhost:8080/pkg/YYY/` pour le paquet YYY

Plus d'informations

- go help doc
- `https://pkg.go.dev/golang.org/x/tools/cmd/godoc`

Et maintenant, un exemple :
documentation de notre paquet sur les listes

Quelques autres commandes pratiques

Indentation avec gofmt

- Faire du code lisible est important,
- l'indentation aide beaucoup à lire du code,
- gofmt permet une indentation propre de manière automatique

Style avec golint

- Le style (nommage des variables, organisation des commentaires) aide aussi beaucoup à lire du code
- golint permet de vérifier qu'un code vérifie un certain nombre de règles de style (il existe d'autres commandes)

Constructions suspectes avec go vet

La commande go vet permet de trouver un certain nombre d'erreurs potentielles dans le code (qui ne sont pas du code incorrect, mais qui en général font que, à l'exécution, il y aura des problèmes)