Programming Assignment: Конвейер функций

You have not submitted. You must earn 1/1 points to pass.

Deadline Pass this assignment by January 13, 11:59 PM PST

Instructions My submission Discussions

hw2_signer.zip

В этом задании мы пишем аналог unix pipeline, что-то вроде:

```
grep 127.0.0.1 | awk '{print $2}' | sort | uniq -c | sort -nr
```

Когда STDOUT одной программы передаётся как STDIN в другую программу

Но в нашем случае эти роли выполняют каналы, которые мы передаём из одной функции в другую.

Само задание по сути состоит из двух частей

1. Написание функции ExecutePipeline которая обеспечивает нам конвейерную обработку функций-воркеров, которые что-то делают.
2. Написание нескольких функций, которые считают нам какую-то условную хеш-сумму от входных данных

Расчет хеш-суммы реализован следующей цепочкой:

- SingleHash считает значение crc32(data)+"~"+crc32(md5(data)) (конкатенация двух строк через ~), где data - то что пришло на вход (по сути - числа из первой функции)
- MultiHash считает значение crc32(th+data)) (конкатенация цифры, приведённой к строке и строки), где th=0..5 (т.е. 6 хешей на каждое входящее значение), потом берёт конкатенацию результатов в порядке расчета (0..5), где data - то что пришло на вход (и ушло на выход из SingleHash)
- CombineResults получает все результаты, сортирует (https://golang.org/pkg/sort/), объединяет отсортированный результат через _

How to submit

When you're ready to submit, you can upload files for each part of the assignment on the "My submission" tab.

(символ подчеркивания) в одну строку

- crc32 считается через функцию DataSignerCrc32
- md5 считается через DataSignerMd5

В чем подвох:

- DataSignerMd5 может одновременно вызываться только 1 раз, считается 10 мс. Если одновременно запустится несколько - будет перегрев на 1 сек
- DataSignerCrc32, считается 1 сек
- На все расчеты у нас 3 сек.
- Если делать в лоб, линейно - для 7 элементов это займёт почти 57 секунд, следовательно надо это как-то распараллелить

Результаты, которые выводятся если отправить 2 значения (закомментировано в тесте):

```
0 SingleHash data 0
0 SingleHash md5(data) cfcd208495d565ef66e7dff9f98764da
0 SingleHash crc32(md5(data)) 502633748
0 SingleHash crc32(data) 4108050209
0 SingleHash result 4108050209~502633748
4108050209~502633748 MultiHash: crc32(th+step1)) 0 2956866606
4108050209~502633748 MultiHash: crc32(th+step1)) 1 803518384
4108050209~502633748 MultiHash: crc32(th+step1)) 2 1425683795
4108050209~502633748 MultiHash: crc32(th+step1)) 3 3407918797
4108050209~502633748 MultiHash: crc32(th+step1)) 4 2730963093
4108050209~502633748 MultiHash: crc32(th+step1)) 5 1025356555
4108050209~502633748 MultiHash result:
29568666068035183841425683795340791879727309630931025356555

1 SingleHash data 1
1 SingleHash md5(data) c4ca4238a0b923820dcc509a6f75849b
1 SingleHash crc32(md5(data)) 709660146
1 SingleHash crc32(data) 2212294583
1 SingleHash result 2212294583~709660146
2212294583~709660146 MultiHash: crc32(th+step1)) 0 495804419
2212294583~709660146 MultiHash: crc32(th+step1)) 1 2186797981
2212294583~709660146 MultiHash: crc32(th+step1)) 2 4182335870
2212294583~709660146 MultiHash: crc32(th+step1)) 3 1720967904
2212294583~709660146 MultiHash: crc32(th+step1)) 4 259286200
2212294583~709660146 MultiHash: crc32(th+step1)) 5 2427381542
2212294583~709660146 MultiHash result:
4958044192186797981418233587017209679042592862002427381542

CombineResults 29568666068035183841425683795340791879727309630931025356555_4958044192186797981418233587017209679042592862002427381542
```

Код писать в **signer.go**. В этот файл не надо добавлять ничего из common.go, он уже будет на сервере.

Запускать как `go test -v -race`

Подсказки:

- Задание построено так чтобы хорошо разобраться со всем материалом лекции, т.е. вдумчиво посмотреть примеры и применить их на практике. Искать по гуглу или стек оферфлоу ничего не надо

- Вам не надо накапливать данные - сразу передаём их дальше (например awk из кода выше - на это есть отдельный тест. Разве что функция сама не решает накопить - у нас это CombineResults или sort из кода выше
- Подумайте, как будет организовано завершение функции если данные конечны. Что для этого надо сделать?
- Если вам встретился рейс (опция -race) - исследуйте его вывод - когда читаем, когда пишем, из каких строк кода. Там как правило содержится достаточно информации для нахождения источника проблемы.
- Прежде чем приступать к распараллеливанию функций, чтобы уложиться в отведённый таймаут - сначала напишите линейный код, который будет выдавать правильный результат, лучше даже начать с меньшего количества значений чтобы совпадало с тем что в задании
- Вы можете ожидать, что у вас никогда не будет более 100 элементов во входных данных
- Ответ на вопрос "когда закрывается цикл по каналу" помогает в реализации ExecutePipeline
- Ответ на вопрос "мне нужны результаты предыдущих вычислений?" помогают распараллелить SingleHash и MultiHash
- Хорошо помогает нарисовать схему рассчетов

Естественно нельзя самим считать хеш-суммы в обход предоставляемых функций - их вызов будет проверяться

Эталонное решение занимает 130 строк с учетом дебага который вы видите выше