GO-13 01: in-memory хранение

Описание:

В этом задании мы рассмотрим такой механизм хранения данных для наших программ, как in-memory.
In-Memory означает хранение данных непосредственно в памяти программы, без помощи таких инструментов, как базы данных. В Go есть очень удобный механизм для реализации такого внутреннего хранилища - это map. Какие же удобства нам предоставляет map для организации in-memory хранения:

1. Наличие ключа - когда мы рассматриваем хранение данных, нам всегда нужно задумываться о поиске этих данных, ведь если мы будем хранить все данные, например, в массиве, то наш поиск превратится в еще одну часть логики нашей программы. Другое дело - мапа, так как мапа построена на такой структуре данных, как хеш-таблица. В ней есть ключи, которые и выступают удобным механизмом поиска данных в нашем хранилище, а также зачастую - и самым быстрым O(1).
2. Наличие возможности проверки присутствия данных по определенному ключу - как мы запомнили ранее, когда мы пытаемся взять значение из мапы по ключу, то, кроме этого значения, она нам отдает bool значение, которое и отвечает, есть данные по этому ключу в мапе или нет.
3. Хранение абсолютно любых типов данных - от строк до сложных вложенных структур.

Когда нам стоит использовать in-memory хранение
Чаще всего такой вид хранения используют как простую реализацию кеша. Например, у нас есть сервис, который отдает информацию по заказам клиента. Заказ - это такая сущность, которая, во-первых, имеет какую-то отличительную особенность (например, id), а также заказ чаще всего статичен и не подвержен постоянному изменению (если брать самый простой случай). Поэтому такие данные хорошо бы хранить в памяти и не ходить на каждый запрос от пользователя в базу, тем самым мы снижаем нагрузку на базу и не теряем в функциональности нашего сервиса. Вот простенький пример реализации такого кеша через мапу.

```
package main

import (
    "fmt"
    "time"
    "net/http"
)

type Order struct {
    ID          string
```

```
    Name        string
    Description string
    Price       int64
    CreatedAt   time.Time
}

func getOrderFromDB() Order {
     time.Sleep(3 * time.Second)
     return Order {
          ID:          "1",
          Name:        "Заказ номер 1",
          Description: "Заказ кросовок nike",
          Price:       3500,
          CreatedAt:   time.Now(),
     }
}

func main() {
    cache := make(map[string]Order)
    http.HandleFunc("/order", func(w http.ResponseWriter, r
*http.Request) {
        id := r.URL.Query().Get("id")
        if id == "" {
            fmt.Fprintln(w, "fail get order")
            return
        }

        if order, ok := cache[id]; ok {
            fmt.Fprintln(w, order)
            return
        }

        order := getOrderFromDB()
        cache[order.ID] = order
        fmt.Fprintln(w, order)
    })

    _ = http.ListenAndServe(":8080", nil)
}
```

Таким образом, мы имеем 1 заказ в нашем учебном сервисе с ID = 1 и, когда мы в первый раз его запросим, то ответ получим спустя целых 3 секунды (так мы имитируем сетевые задержки и обработку базы). В следующий раз мы получим наш заказ намного быстрее, в нашем случае - почти мгновенно.
Когда нам не стоит использовать in-memory хранение
Ну а теперь давайте немножко поговорим о минусах такого подхода.

1. Мы теряем данные, когда сервис выключается. Если данные хранятся в базе, то это хорошо, но бывают и другие случаи, где данные хранятся только в in-memory, и тогда надо задуматься о сохранности наших данных.
2. Потребление ресурсов памяти. RAM не бесконечен и рано или поздно, если не следить за своим in-memory хранилищем, то сервис скорее всего упадет, а если еще и данные не сохранились в базу, то мы потеряем и данные.
3. Нужно заботиться о параллельном исполнении вашего кода. Например, если вы пишете сервис и он отдает информацию по http, используя при этом in-memory кеш, то нужно позаботиться, чтобы в вашу мапу в один момент времени писал только кто-то один, иначе мы можем уронить наш сервис.
4. Коэффициент масштабируемости вашего сервиса падает. Нам становится сложно горизонтально масштабировать сервис, ведь мы не можем нормально и без костылей сделать репликацию данных из такого хранилища, а если мы все-таки как-то это сделаем, то снова взорвемся по памяти, потому что RAM не бесконечный. Также дело обстоит и с вертикальным масштабированием, без конца покупать RAM очень дорого и нецелесообразно.

Таким образом, если вы хотите хранить большое количество данных, то скорее всего вам не нужно in-memory хранение, а лучше воспользоваться базой данных, подходящей под ваши задачи. Если вы боитесь потерять данные, которые вы храните в in-memory, тогда вам также лучше рассмотреть другие способы хранения или позаботиться о надежном хранилище помимо in-memory на случай падения сервиса или других аномалий.
Ну а если же, например, ваши данные - это какая-то реалтайм информация, которую можно легко получить или сгенерировать, или данные надежно хранятся в базе и вы хотите ускорить процесс их доставки до пользователя, тогда in-memory - это неплохой вариант. Особенно если вы не хотите заморачиватся с базами, которые как раз заточены под хранение такой информации (например, redis).

Полезные ссылки:

- Maps by example

Задание:

Давайте попробуем реализовать кеш в нашем учебном сервисе, который хранит в себе информацию о пользователях системы.
Условия

1. Кеш должен быть реализован средствами языка Go (без redis или подобных ему баз, которые используются для реализации кеша в приложениях).
2. Если в кеше нет данных по конкретному пользователю, мы должны брать этого пользователя из базы.
3. Нужно позаботиться о потокобезопасности нашего кеша, чтобы он мог выдерживать параллельные запросы.
4. Кеш должен держать данные не бесконечно, иначе о изменениях мы узнаем, только перезапустив сервис. Кеш должен держать данные 5 секунд, затем снова идти в базу и обновлять информацию в кеше.

Support

Make команды

- make goose-install - команда, которая установит вам утилиту goose, она нужна вам, для того чтобы накатывать миграции в базу.
- make bench-install - команда, которая установит вам утилиту gobench для стресс-теста сервиса.
- make psql-run - команда, которая поднимет нам Postgres базу в докере.
- make migrate - команда, которая накатит нам миграции из папки ./migrations при помощи утилиты goose.
- make run - команда, которая поднимает сервис.
- make strees - команда, которая запускает стресс-тест сервиса при помощи утилиты gobench.

Http handlers

- POST /user - ручка для добавления нового пользователя (в миграциях добавляются 3 пользователя, но если вы захотите добавить еще, то это можно сделать при помощи данной ручки).
- GET /user/:email - ручка для получения пользователя по его email адресу - это ручка, с которой мы непосредственно будем работать.

Misc

Все http ручки работают через gin router, регистрация ручек происходит в пакете module13/internal/handlers.

Порядок действий

1. Форкните репозиторий module13 с кодом данного задания в группу с вашими репозиториями - golang_users_repos/<your_gitlab_id>.
2. В вашем проекте module13 сделайте новую ветку 01_task.
3. Устанавливаем goose при помощи команды make goose-install.
4. Устанавливаем gobench при помощи команды make bench-install.
5. Запускаем контейнер с базой при помощи команды make psql-run.
6. Накатываем миграции в базу при помощи команды make migrate.
7. Запускаем сервис при помощи команды make run.
8. Пробуем провести стресс-тест при помощи команды make stress.
9. В пакете module13/internal/handlers/user пробуем реализовать для ручки GetUser кеш.

10. Снова запускаем стресс-тест и сравниваем результаты до и после.
11. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 01_task.