GO-08 01: Конфигурирование подключения к БД

Описание:

Как и во многих других языках программирования, в Go нам очень часто приходится работать с базами данных, особенно при написании микросервисов. В этом разделе мы научимся правильно конфигурировать подключения к базе данных PostgreSQL, делать запросы, создавать миграции и получать метрики. При этом будем в основном использовать нативные средства работы с БД - библиотеки database/sql и github.com/jackc/pgx, так как это очень распространенная практика. Но также попробуем использовать популярную ORM для Go - GORM github.com/go-gorm/gorm. И конечно, проанализируем разницу между этими подходами. Что ж, приступим!
Для работы с БД нам, собственно, понадобится сама БД :) Чтобы ее поднять быстро и просто, нам нужны docker и docker-compose, а также минимальные знания этих технологий. Кстати говоря, дальше мы будем строить микросервисную архитектуру и приложения на ней. Поэтому нам придется часто использовать docker.
Поднимаем Postgres в docker-compose
Создадим файл ~/rebrain/docker-compose.yaml со следующим содержанием:

```
# файл ~/rabrain/docker-compose.yaml
version: '3'
services:
 pgdb:
   image: "postgres:11"
   container_name: "pgdb"
   environment:
     - POSTGRES_USER=db_user
     - POSTGRES_PASSWORD=pwd123
   ports:
     - "54320:5432"
   volumes:
     - /Users/<user>/rebrain/db/data:/var/lib/postgresql/data    ### Mac Os
 #   - /home/<user>/rebrain/db/data:/var/lib/postgresql/data     ### Linux
 #   - //c/Users/<user>/rebrain/db/data:/var/lib/postgresql/data ### Windows
```

Мы добавили в docker-compose.yaml наше первое приложение - базу данных PostgreSQL pgdb - сразу создали пользователя db_user и задали ему пароль pwd123. Хотим, чтобы наша база крутилась на порту 54320, а также прокинем volume себе в систему. В примере есть volumes для всех систем, поэтому раскомментируйте тот вариант, который вам подходит. Только не забудьте поменять <user> на реальное имя пользователя в вашей системе.
Стартуем наш docker-compose.yaml:

```
$ cd ~/rebrain/
$ docker-compose up -d
Creating network "rebrain_default" with the default driver
Pulling db (postgres:11)...
11: Pulling from library/postgres
7d2977b12acb: Pull complete
0189767a99c6: Pull complete
2ac96eba8c9d: Pull complete
8b4f0db1ff6e: Pull complete
9e30cfe22768: Pull complete
8c90c3e75b96: Pull complete
5ddcc5e296f9: Pull complete
fd42372a1ee8: Pull complete
db53e89e9aa9: Pull complete
90d820846158: Pull complete
07f8ae023b87: Pull complete
66523f120c51: Pull complete
31944359dec5: Pull complete
c6c4e5d2f560: Pull complete
Digest:
sha256:495de69cd2f7be2e6363c980e2ddf99fb1bef997a51d800c1f72be2b35648b3
b
Status: Downloaded newer image for postgres:11
Creating pgdb ... done


$ docker-compose ps
Name              Command              State            Ports
-------------------------------------------------------------------------
pgdb   docker-entrypoint.sh postgres   Up      0.0.0.0:54320->5432/tcp

$ ls ~/rebrain/db/data/
```

```
PG_VERSION          pg_dynshmem/        pg_multixact/
pg_snapshots/       pg_tblspc/          postgresql.auto.conf
base/               pg_hba.conf         pg_notify/
pg_stat/            pg_twophase/        postgresql.conf
global/             pg_ident.conf       pg_replslot/
pg_stat_tmp/        pg_wal/             postmaster.opts
pg_commit_ts/       pg_logical/         pg_serial/
pg_subtrans/        pg_xact/            postmaster.pid
```

Отлично! Видим, что у нас успешно поднят инстанс базы данных Postgres на порту 54320 с пользователем db_user и паролем pwd123. А также прокинулись volumes в директорию ~/rebrain/db/data/.
Теперь создадим базу данных под названием db_test:

```
$ docker-compose exec pgdb psql -U db_user -c 'CREATE DATABASE
db_test'
CREATE DATABASE
```

База готова, и к ней можно пробовать подключаться.
Подключение к Postgres из Go
Для работы с SQL базами в Go существует библиотека database/sql. Эта библиотека предоставляет базовые возможности для работы с любой БД (по ссылке go-database-sql.org можно подробно ознакомиться с тем, как она работает). Но для каждой конкретной базы, будь то MySQL, Postgres, Oracle и т.д., лучше использовать конкретный драйвер. Для Postgres долгое время основным драйвером был github.com/lib/pq, но сегодня он потерял свою актуальность. И ему на смену пришел github.com/jackc/pgx. Поэтому мы будем работать с БД, используя актуальную и поддерживаемую библиотеку. Для начала нам нужно создать небольшой проект для заданий по базам данных. Назовем его godb:

```
~/mygo
└── godb
    └── cmd
        └── app
            └── main.go
```

Теперь добавим в файл main.go следующий код и подробно его рассмотрим:

```
//файл main.go
package main

import (
```

```
  "context"
  "fmt"
  "net/url"
  "os"

  //Импортируем библиотеку для работы с posgtres
  "github.com/jackc/pgx/v4"
)

func main()  {
  //Создание строки для подключения
  connStr :=
fmt.Sprintf("%s://%s:%s@%s:%s/%s?sslmode=disable&connect_timeout=%d",
     "postgres",
     url.QueryEscape("db_user"),
     url.QueryEscape("pwd123"),
     "localhost",
     "54320",
     "db_test",
     5)

  //Контекст с отменой
  ctx, _ := context.WithCancel(context.Background())

  //Подключение к базе данных. В случае неуспешного подключения
выводим ошибку
  conn, err := pgx.Connect(ctx, connStr)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Connection OK!")

  //Ping делает пустой запрос к базе данных (";") для проверки наличия
фактического соединения
  err = conn.Ping(ctx)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
     os.Exit(1)
  }
```

```
  fmt.Println("Query OK!")

//Закрываем соединение с базой
conn.Close(ctx)

}
```

После добавления кода в файл main.go нужно запустить команду:

```
$ cd ~/mygo/godb/
$ gp mod init godb
$ go mod download
```

Мы ее запустили, чтобы скачать недостающую библиотеку github.com/jackc/pgx/v4в локальное пространство. Также обратите внимание на то, что мы используем именно четвертую версию этой библиотеки. Можно вспомнить, как работает версионирование мажорных версий в go mod.
Дальше мы создаем переменную connStr, в которой определяем URI для подключения к базе данных.

```
 connStr :=
fmt.Sprintf("%s://%s:%s@%s:%s/%s?sslmode=disable&connect_timeout=%d",
     "postgres",
     url.QueryEscape("db_user"),
     url.QueryEscape("pwd123"),
     "localhost",
     "54320",
     "db_test",
     5)
```

Postgres поддерживает строку подключения двух видов:
https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING
ключ/значение и RFC3986. Мы будем использовать последнюю.
Собираем URI с помощью fmt.Sprintf. Первый параметр - это название протокола - postgres. Дальше идет авторизация: передаем имя пользователя и пароль, который мы создавали во время поднятия базы, - db_user и pwd123. Передавать их следует через функцию url.QueryEscape, так как в пароле и логине могут быть символы, не поддерживаемые URL. Их нужно декодировать, и они должны быть url safe. После @ идут хост и порт. Мы запускали контейнер в локально установленном docker и пробрасывали порт 54320, соответственно указываем localhost и 54320. После протокола, авторизации и адреса с портом указывается через слеш /название базы, к которой мы хотим подключиться. Базу мы создали db_test, ее и передаем. Основной URL мы собрали, но для подключения мы можем использовать еще ряд параметров, все они описаны в

документации Postgres. Для нашей задачи мы используем 2 из них. Это sslmode=disable, которым отключаем шифрование данных по причине того, что, во-первых, это обучающие задания, а во-вторых, потому что мы используем приложение и базу внутри одной сети. Второй используемый нами параметр - это connect_timeout, который задает максимальное время ожидания для подключения. Очень важно задавать этот параметр (задается в секундах), так как, если он не определен, то по умолчанию попытка подключения будет бесконечной, и наше приложение в случае недоступности базы «залипнет» навсегда.
Определяем context:

```
 //Контекст с отменой
  ctx, _ := context.WithCancel(context.Background())
```

Библиотека github.com/jackc/pgx/v4 вынуждает нас производить все взаимодействия с базой, используя context, и это хорошо. Потому что контекст очень полезен, с помощью него логирование запросов и ответов из базы становится более информативным, а также можно управлять целостностью запросов в базу на уровне приложения с помощью context cancel. Преимущества использования контекста при работе с базой мы рассмотрим в следующем задании. А пока его просто создаем и используем, так как это обязательный аргумент.
Тут инициируется подключение к базе:

```
 //Подключение к базе данных. В случае неуспешного подключения
выводим ошибку
  conn, err := pgx.Connect(ctx, connStr)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Connection OK!")
```

И вот мы используем библиотеку pgx для непосредственного подключения к базе conn, err := pgx.Connect(ctx, connStr) передаем на вход контекст с отменой и URI базы данных. Функция pgx.Connect вернет нам объект с подключением или ошибку. В случае ошибки выведем ее в stdout и завершим выполнение программы кодом 1 os.Exit(1). В случае успешного подключения выведем сообщение Connection OK!
Далее, используя полученное подключение, мы вызываем у него функцию conn.Ping():

```
 //Ping делает пустой запрос к базе данных (";") для проверки наличия
фактического соединения
  err = conn.Ping(ctx)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
     os.Exit(1)
  }
```

```
  fmt.Println("Query OK!")
```

Эта функция выполняет пустой запрос к базе данных “;”, используя текущее подключение. Нужно это для того, чтобы сделать запрос в базу и тем самым проверить фактическое состояние подключения. Но в то же время нам не нужно нагружать базу, поэтому запрос пустой. Один из примеров использования этой функции - это healthcheck в микросервисах, когда система оркестрации стучится на определенный эндпоинт, в котором вызывается pgx.Ping(), и если функция вернет ошибку, то оркестратор будет знать, что данный микросервис не может работать полноценно из-за отсутствия подключения к БД и т.д. Мы же с вами используем эту функцию, для того чтобы фактически использовать подключение к базе и делать через него хоть какой-то, самый простой запрос в базу. Так как другие более сложные запросы мы пока что делать не умеем (но обязательно научимся в следующем задании). Так вот, если запрос к базе прошел успешно, мы выведем Ping OK!, а если нет, то остановим приложение с ошибкой.
После чего в конце функции main закрываем соединение с базой.

```
//Закрываем соединение с базой
conn.Close(ctx)
```

Итак, в данном случае мы поступили очень просто - создали соединение, выполнили запрос, закрыли соединение. Для первого знакомства такой подход нам подойдет, но в дальнейшем для полноценной работы с базой его нам будет недостаточно.
Давайте запустим приложение и посмотрим на вывод:

```
$ go run cmd/app/main.go
Connection OK!
Query OK!
```

Отлично! Видим, что прошли успешно и создание соединения, и сам запрос!
Забегая немного вперед, можно утверждать, что создание и управление соединениями в Go - это важная часть, которую нужно понимать. Поэтому давайте разберемся в этом и для начала посмотрим на наш совсем простой запрос поподробнее.
Для этого не будем закрывать «руками» соединение с базой и закомментируем эту строчку:

```
//Закрываем соединение с базой
//conn.Close(ctx)

select {}
```

А вместо нее заблокируем основной поток, добавив оператор select{}. Теперь наша программа не будет заканчивать свою работу после запроса в базу и не будет закрывать

соединение. Если бы мы не заблокировали основной поток, то программа закончила бы свою работу и соединение с базой закрылось бы автоматически.
Запустим программу:

```
$ go run cmd/app/main.go
Connection OK!
Query OK!
```

Мы успешно подключились и выполнили запрос, но приложение не завершилось и продолжает оставаться запущенным. Это как раз то, что нам нужно. Давайте теперь посмотрим на подключения со стороны базы.
Выполним следующую команду, не останавливая процесс нашей программы:

```
$ cd ~/rebrain/
$ docker-compose exec pgdb psql -U db_user -c 'SELECT pid, usename,
state, query FROM pg_stat_activity WHERE state IS NOT NULL;'

  pid  | usename | state  |                                        query
-------+---------+--------+-----------------------------------------
-------------------------------------
 10836 | db_user | idle   | ;
 10926 | db_user | active | SELECT pid, usename, state, query FROM
pg_stat_activity WHERE state IS NOT NULL;
(2 rows)
```

Данной командой мы вывели все актуальные подключения к базе:

- `pid` - это ID подключения,
- `usename` - пользователь, из-под которого выполенено подключение,
- `state` - состояние подключения, `query` - последний запрос, который выполнен с помощью этого подключения.

Подключение с pid 10926 (в моем случае) - это то подключение и тот запрос, который мы выполнили из консоли и, собственно, получили этот вывод, поэтому это для нас не очень интересно. На него не обращаем внимания.
Подключение 10836 - это подключение, которое выполнили с помощью нашего Go приложения. В колонке query мы видим наш минимальный запрос “;”. В данный момент подключение находится в состоянии idle, это означает, что подключение установлено, находится в ожидании и готово исполнять запросы. В нашем случае произошло успешное подключение, при выполнении запроса статус изменялся на active (мы пока что этого не увидим). А затем после выполнения запроса “;” перешло в состояние idle и находится в

нем на данный момент, потому что мы его никак не закрывали, процесс программы находится в состоянии выполнения и держит этот коннект.
Далее, если мы попытаемся еще раз сделать запрос, это подключение на время запроса перейдет в состояние active, выполнит запрос и, если мы его не закроем, вернется в состояние idle. Тут все выглядит прозрачно и понятно.
Но если мы подумаем наперед, то вспомним, что одним из главных преимуществ Go является возможность параллельно выполнять код во множестве потоков, что позволяет сильно оптимизировать время на обработку данных. Давайте представим, что нам нужно выполнить параллельно 5 неких вычислений, результат каждого записать в базу данных и сделать это максимально быстро. Первое, что необходимо, - это запустить каждый процесс в своей горутине. В таком случае все вычисления произойдут практически одновременно. А вот на этапе записи в базу у нас образуется «бутылочное горлышко», все 5 горутин начнут пытаться писать в базу через установленное соединение, в итоге, встанут в очередь и будут записаны последовательно. А это в значительной степени лишает смысла распараллеливание кода с помощью горутин.
Применим к нашему примеру. Следующий код в обоих случаях выполняется примерно за одно и то же время:

```
for i := 0; i < 5; i++ {
  err = conn.Ping(ctx)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println(i, "Query OK!")
}
```

```
for i := 0; i < 5; i++ {

  go func(i int) {
     err = conn.Ping(ctx)
     if err != nil {
        fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
        os.Exit(1)
     }
     fmt.Println(i, "Query OK!")
  }(i)
}
```

Как же выйти из этой ситуации? Правильно! Нам нужно больше соединений. Теоретически мы можем для каждого запроса создать отдельное соединение. Но это, конечно, не лучшая идея. Потому что запросов может быть слишком много, а лимит на количество соединений, например, всего 100 (по умолчанию в postgres). А если нам нужно сделать 101 запрос или больше, может получиться ситуация, при которой все соединения будут заняты и наше приложение просто сработает с ошибкой, так как не сможет получить соединение для запроса. Тогда мы можем закрывать соединения после выполненного запроса. То есть так:

```
for i := 0; i < 5; i++ {

  go func(i int) {
  //Каждую итерацию цикла устанавливаем новое соединение
  conn, err := pgx.Connect(ctx, connStr)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Connection OK!")

  //Делаем запрос, используя вновь созданное соединение
  err = conn.Ping(ctx)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println(i, "Query OK!")


  //Закрываем соединение
  conn.Close(ctx)
  }(i)
}
```

И теоретически это сработает. Но тут есть другая проблема: установка соединения тоже занимает какие-то ресурсы и время. И чаще всего это даже сложнее и дольше, чем сам запрос. Каждый раз открывать и закрывать соединение - не самый оптимальный вариант. Но в итоге выход, конечно же, есть. Им будет пул открытых соединений, находящихся в состоянии idle, то есть в состоянии ожидания, и распределение между ними всех запросов. Это самый оптимальный и правильный вариант. И дальше мы рассмотрим, как

работать с базой с помощью пула соединений. К счастью, библиотека pgx имеет в своем наборе отличный пакет github.com/jackc/pgx/v4/pgxpool, который позволяет удобно и, особенно не задумываясь о внутренней реализации, использовать пул соединений и распределять запросы между соединениями внутри него!
Давайте начнем использовать пул вместо обычного соединения. Для этого вместо github.com/jackc/pgx/v4/ импортируем пакет github.com/jackc/pgx/v4/pgxpool.
Затем создадим конфиг для пула, передавая туда строку подключения connStr, и зададим максимальное количество соединений:

```
poolConfig, _ := pgxpool.ParseConfig(connStr)
poolConfig.MaxConns = 5
```

Вместо использования conn, err := pgx.Connect(ctx, connStr) создадим пул, передавая в него контекст и конфиг dbConfig:

```
pool, err := pgxpool.ConnectConfig(ctx, poolConfig)
if err != nil {
  fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
  os.Exit(1)
}
fmt.Println("Connection OK!")
```

Структура Conn, возвращаемая pgxpool, не имплементирует функцию Ping. Поэтому минимальный запрос к БД ”;” на этот раз мы напишем руками через функцию Exec. А также сделаем несколько запросов асинхронно, используя горутины и цикл:

```
for i := 0; i < 5; i++ {
  go func(count int) {
     //Функция Exec выполняет запрос к БД.
     _, err = pool.Exec(ctx, ";")
     if err != nil {
        fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
        os.Exit(1)
     }
     fmt.Println(count, "Query OK!")
  }(i)
}
```

Отлично! Теперь давайте посмотрим, что получилось:

```
//файл main.go
```

```go
package main

import (
  "context"
  "fmt"
  "net/url"
  "os"

  //Импортируем пакет для работы с пулом соединений
  "github.com/jackc/pgx/v4/pgxpool"
)

func main()  {
  connStr :=
fmt.Sprintf("%s://%s:%s@%s:%s/%s?sslmode=disable&connect_timeout=%d",
     "postgres",
     url.QueryEscape("db_user"),
     url.QueryEscape("pwd123"),
     "localhost",
     "54320",
     "db_test",
     5)

  ctx, _ := context.WithCancel(context.Background())

  //Сконфигурируем пул, задав для него максимальное количество
соединений
  poolConfig, _ := pgxpool.ParseConfig(connStr)
  poolConfig.MaxConns = 5

  //Получаем пул соединений, используя контекст и конфиг
  pool, err := pgxpool.ConnectConfig(ctx, poolConfig)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Connection OK!")

  for i := 0; i < 5; i++ {
     go func(count int) {
```

```
            //Функция Exec выполняет запрос к БД.
            _, err = pool.Exec(ctx, ";")
            if err != nil {
               fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
               os.Exit(1)
            }
            fmt.Println(count, "Query OK!")
         }(i)
      }

      select {}

}
```

Отменим с помощью Ctrl + C прошлое выполнение программы (если вы этого еще не сделали) и запустим программу.

```
$ go run cmd/app/main.go
Connection OK!
4 Query OK!
3 Query OK!
2 Query OK!
0 Query OK!
1 Query OK!
```

Отлично, пул работает. С нашей стороны никаких дополнительных действий (кроме настройки максимального количества подключений) не потребовалось. Все пять запросов к базе выполнились успешно! Посмотрим информацию о подключениях. Выполним команду:

```
$ docker-compose exec pgdb psql -U db_user -c 'SELECT pid, usename,
state, query FROM pg_stat_activity WHERE state IS NOT NULL;'
 pid  | usename | state  |                                          query
------+---------+--------+-----------------------------------------------
-------------------------------------
 2336 | db_user | idle   | ;
 2338 | db_user | idle   | ;
 2339 | db_user | idle   | ;
 2337 | db_user | idle   | ;
 2340 | db_user | idle   | ;
```

```
 2353 | db_user | active | SELECT pid, usename, state, query FROM
pg_stat_activity WHERE state IS NOT NULL;
(6 rows)
```

Видим, что сейчас у нас 5 соединений и для каждого была выполнена команда ";". После выполнения запроса все соединения перешли в состояние ожидания. И при следующих запросах pgxpool будет использовать только эти соединения, не создавая новых.
Мы получили возможность выполнения 5 запросов в базу параллельно. А значит, все наши горутины выполнились асинхронно и мы не потеряли ни во времени, ни в производительности, благодаря пулу соединений.
Еще мы можем на уровне приложения отслеживать состояние соединений в базе. Для этого в пакете "github.com/jackc/pgx/v4/pgxpool" в структуре Pool существует функция Stat, которая может возвращать максимальное количество соединений, текущее количество соединений и количество соединений в ожидании (idle).
После каждого запроса в базу отобразим информацию о подключениях. Добавим вывод метрик после fmt.Println(count, "Query OK!"):

```
for i := 0; i < 5; i++ {
  go func(count int) {
     //Функция Exec выполняет запрос к БД
     _, err = pool.Exec(ctx, ";")
     if err != nil {
        fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
        os.Exit(1)
     }
     fmt.Println(count, "Query OK!")
     //Выводим информацию о соединениях
     fmt.Printf("Conections - Max: %d, Iddle: %d, Total: %d \n",
pool.Stat().MaxConns(), pool.Stat().IdleConns(),
pool.Stat().TotalConns())

  }(i)
}
```

И перезапускаем приложение:

```
$ go run cmd/app/main.go
Connection OK!
4 Query OK!
Conections - Max: 5, Iddle: 1, Total: 5
1 Query OK!
Conections - Max: 5, Iddle: 3, Total: 5
```

```
3 Query OK!
Conections - Max: 5, Iddle: 3, Total: 5
2 Query OK!
Conections - Max: 5, Iddle: 4, Total: 5
0 Query OK!
Conections - Max: 5, Iddle: 5, Total: 5
```

Полезные ссылки:

- Go database/sql tutorial
- HTML - URL Encoding
- PGSQL - connection params

Задание

1. Повторите локально описанный в задании пример реализации подключения к БД или склонируйте репозиторий с примером module08_01, внеся в него такие изменения:
    - минимальное количество подключений в пуле (MinConns) - 20;
    - максимальное количество подключений в пуле (MaxConns) - 20;
    - количество горутин в цикле - 10.
2. В качестве ответа на задание прикрепите к ответу такие данные:
    - результат работы приложения;
    - результат запроса в базу (SELECT pid, usename, state, query FROM pg_stat_activity WHERE state IS NOT NULL;);
    - ВАЖНО: Просьба оформить ответ в кодовый блок для его читаемости, в противном случае задание будет отклонено.