ozontech

ozon tech

Лекция 7. Работа в Go с PostgreSQL (лекция 1)

Юсипов Гаяз

Разработчик информационных систем в команде пользовательского контента

OZON

Москва, 2021

Содержание

- Общие сведения о PostgreSQL: индексы, транзакционность
- Подключение к PostgreSQL: драйверы, DSN, пул соединений
- Миграции
- Фикстуры
- Выполнение запросов
- NULL значения
- SQL инъекции
- Библиотека jmoiron/sqlx (расширение database/sql)
- Библиотека Masterminds/squirrel (query builder)
- Транзакции и блокировки
- Тестирование

PostgreSQL

> *PostgreSQL или Postgres — свободная объектно-реляционная система управления базами данных (СУБД).*

Актуальная версия — 14.0, от 30 сентября 2021

Максимальный размер базы данных	Нет ограничений
Максимальный размер таблицы	32 TB
Максимальный размер поля	1 GB
Максимум записей в таблице	Ограничено размерами таблицы
Максимум полей в записи	250—1600, в зависимости от типов полей
Максимум индексов в таблице	Нет ограничений

Индексы

- **B-Tree (B-дерево)**
- Hash
- GiST
- SP-GiST
- GIN
- BRIN

По умолчанию команда CREATE INDEX создаёт индексы типа B-Tree, эффективные в большинстве случаев.

Примеры создания индексов

- Создание индекса (B-дерево) по столбцу title в таблице films:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

- Создание индекса по выражению lower(title) для эффективного регистронезависимого поиска:

```
CREATE INDEX ON films (lower(title));
```

(имя индекса выберет система `films_lower_idx`)

Создание индекса без блокировки записи в таблицу:

С версии 8.2

```
CREATE INDEX CONCURRENTLY my_table_index ON sales_table (quantity);
```

- без блокировок на добавление, изменение или удаление записей в таблице
- можно без проблем работать с базой пока индекс строится
- приведет к дополнительному сканированию таблицы
- выполняется значительно дольше чем обычное построение индекса
- нагружает систему, это может аффектить другие операции

```
REINDEX INDEX CONCURRENTLY my_table_index -- с PG 12+
```

```
DROP INDEX CONCURRENTLY my_table_index
```

Hash

Хеш-индексы хранят 32-битный хеш-код, полученный из значения индексированного столбца, поэтому хеш-индексы работают только с простыми условиями равенства. Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором =. Создать такой индекс можно следующей командой:

```
CREATE INDEX имя ON таблица USING HASH (столбец);
```

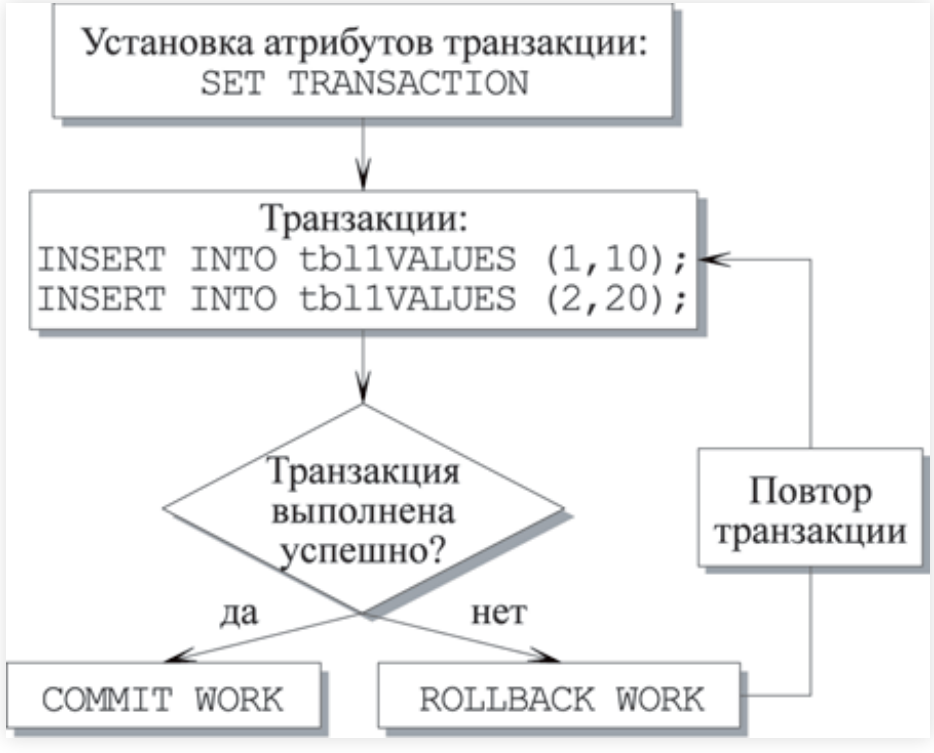
Hash работают быстрее, чем B-Tree, в тех случаях, когда у вас есть известное значение ключа, а лучше известное уникальное значение.

Индекс	Сложность
B-Tree	O(log n)
Hash	O(1)

Транзакционность

Транзакции — это фундаментальное понятие во всех СУБД. Транзакция объединяет последовательность действий в одну операцию «всё или ничего».

Реализация транзакций в СУБД PostgreSQL основана на мультиверсионной модели (Multiversion Concurrency Control, MVCC).



Уровни изоляции транзакций

Стандарт SQL описывает условия (феномены), недопустимые для различных уровней изоляции:

- dirty read, «грязное» чтение: транзакция читает данные, записанные параллельной незавершённой транзакцией.
- nonrepeatable read, неповторяемое чтение: транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).
- phantom read, фантомное чтение: транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.
- serialization anomaly, аномалия сериализации: результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

1. **Read uncommited**: самый низкий уровень изоляции. По стандарту SQL допускается чтение «грязных» (незафиксированных) данных. В PostgreSQL требования более строгие, чтение «грязных» данных не допускается.
2. **Read committed**: уровень транзакционности по-умолчанию. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.
3. **Repeatable read**: все то же, что и не предыдущем уровне, также не допускается фантомное чтение. То есть, реализация этого уровня более строгая, чем в стандарте SQL (это не противоречит стандарту).
4. **Serializable**: не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Уровень изоляции	**«Грязное» чтение**	**Неповторяемое чтение**	**Фантомное чтение**	**Аномал сериал**
Read uncommited (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возмож
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возмож
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возмож
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозм

* Read Uncommitted в PostgreSQL действует как Read Committed

postgrespro.ru/docs/postgresql/14/transaction-iso

Транзакции в Postgres

- существует 4 уровня изоляций транзакций
- в Postgres чтение «грязных» данных недопустимо, поэтому в Postgres фактически 3 уровня
- read uncommited существует формально, работает как алиас для read committed
- read committed используется по-умолчанию

Вопросы?

Партицирование

Партицирование — разбиение большой таблицы на несколько таблиц по определённому условию (ключу партицирования)

Шардирование

Шардирование — разбиение большой базы данных на несколько выделенных баз данных, расположенных на разных серверах

Установка Postgres локально и через Docker

- раздел скачивания с инструкциями postgresql.org/download/
- официальные Docker образы hub.docker.com/_/postgres

- Локально для Debian и производных (Ubuntu, Linux Mint etc)

```
sudo apt update
sudo apt install postgresql
sudo systemctl start postgresql
sudo -u postgres psql
```

- Локально для macOS, используя brew

```
brew install postgresql
brew services start postgresql
psql postgres
```

- Используя Docker

```
docker run --rm -d \
  --name postgres \
  -e POSTGRES_PASSWORD=password \
  -p 5432:5432 \
  postgres

docker exec -it postgres psql -Upostgres
```

Драйверы в Go для подключения к Postgres

```
import "database/sql"
// ...
sql.Open(driver, url string) (*DB, error)
db.ExecContext(...) (Result, error)
db.QueryRowContext(...) *Row
db.QueryContext(...) (*Rows, error)
// ... это лишь часть методов
```

- реализует базовый интерфейс над SQL-совместимыми СУБД
- для работы нужен драйвер
- не привязана к какой-либо конкретной СУБД

SQLInterface github.com/golang/go/wiki/SQLInterface

Список драйверов golang.org/s/sqldrivers

Туториал по `database/sql` go-database-sql.org

lib/pq

```
go get github.com/lib/pq
```

- работает только с Postgres
- все еще часто встречается
- не использует cgo (pure go)
- maintenance mode, советуют переходить на pgx

go-pg/pg

```
go get github.com/go-pg/pg
```

- клиент и ORM для Postgres
- не использует cgo
- maintenance mode, советуют переходить на uptrace/bun

uptrace/bun

```
go get github.com/uptrace/bun
```

- клиент для PostgreSQL, MySQL, MariaDB и SQLite
- не использует cgo
- миграции и фикстуры «из коробки»
- первый коммит весной 2021, пока что мало популярен

bun.uptrace.dev

pgx - PostgreSQL Driver and Toolkit

```
go get github.com/jackc/pgx/v4
```

- работает только с Postgres
- популярная и активно развивается
- не использует cgo (pure go)
- поддержка фичей, специфичных для Postgres
- можно использовать свой интерфейс, отличный от `database/sql`

Интерфейс pgx можно выбрать если:

- используется только Postgres
- никакая другая библиотека в приложении не использует `database/sql`

Драйверы в Go для подключения к Postgres

- для подключения нужен драйвер
- большинство использует pgx (и мы тоже)

Подключение к Postgres

Open Database Connectivity (ODBC) — открытый стандарт API для доступа к БД.

DSN (data source name) — структура данных, содержащая информацию о конкретной БД, с которой ODBC драйвер может подключиться.

Пример DSN:

```
postgres://user:password@localhost:5432/db
```

или

```
user=user password=password host=localhost port=5432 database=db sslmode
```

```go
import (
    "database/sql"
    _ "github.com/jackc/pgx/v4/stdlib"
)

func main() {
    // Open лишь валидирует аргументы, но не создает подключения
    db, err := sql.Open("pgx", "postgres://...")
    // Ping создает подключение, если его еще нет
    err = db.PingContext(ctx)
}
```

Пул соединений

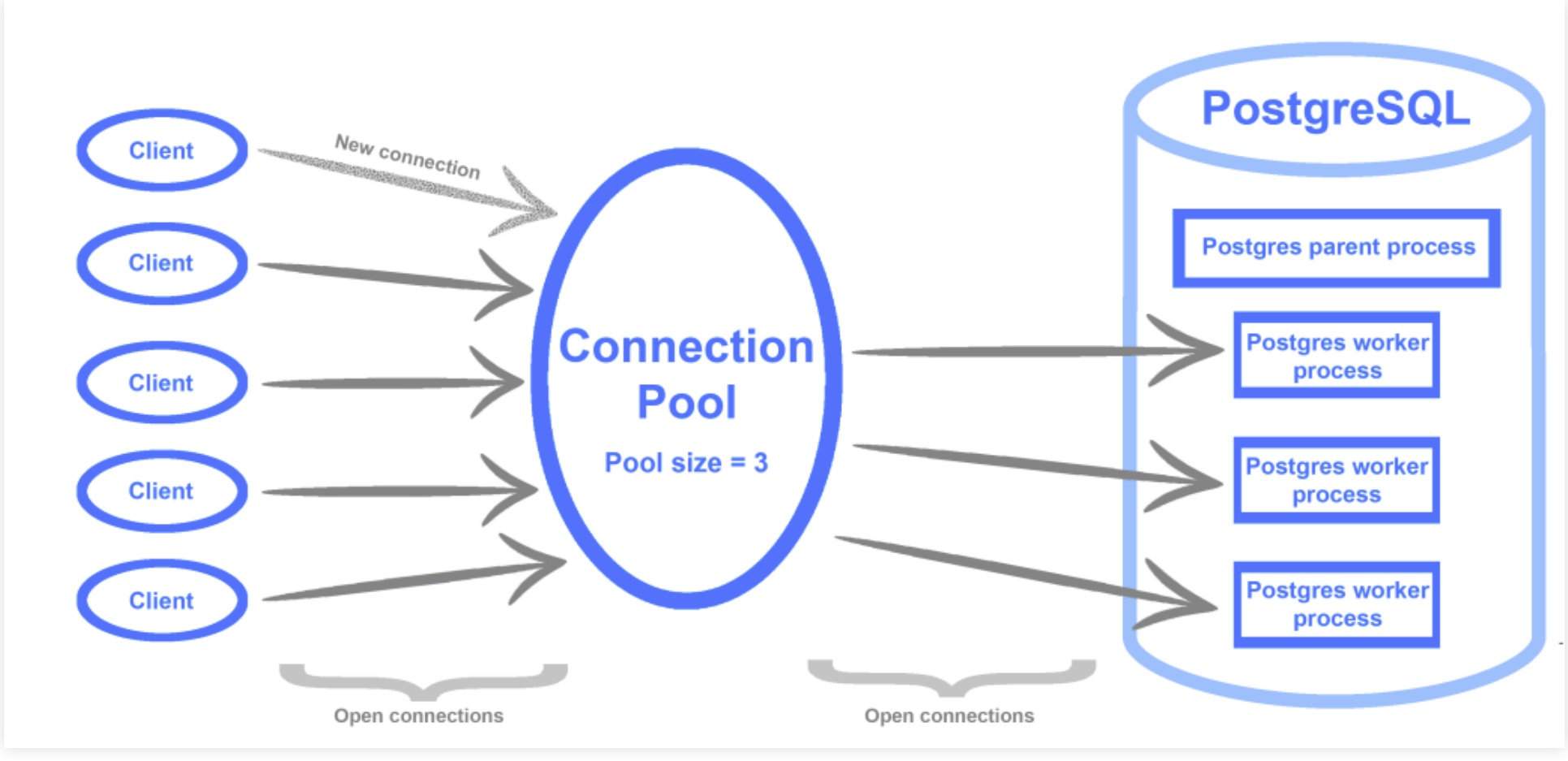
`sql.DB` - это пул соединений с базой данных. Соединения будут открываться по мере необходимости.

`sql.DB` - безопасен для конкурентного использования (так же как `http.Client`)

Настройки пула:

```
// Макс. число открытых соединений от этого процесса
db.SetMaxOpenConns(n int)
// Макс. число открытых неиспользуемых соединений
db.SetMaxIdleConns(n int)
// Макс. время жизни одного подключения
db.SetConnMaxLifetime(d time.Duration)
```

go-database-sql.org/connection-pool.html



pgbouncer.org — Lightweight connection pooler for PostgreSQL

Миграции

Для миграций используем `goose`

```
go install github.com/pressly/goose/v3/cmd/goose@latest
```

или через brew:

```
brew install goose
```

- можно использовать как CLI утилиту (поддерживаются только SQL файлы)
- можно подключить как библиотеку и запускать из кода (поддерживаются SQL и go файлы)

pressly.github.io/goose

Фикстуры

Для фикстур используем github.com/go-testfixtures/testfixtures

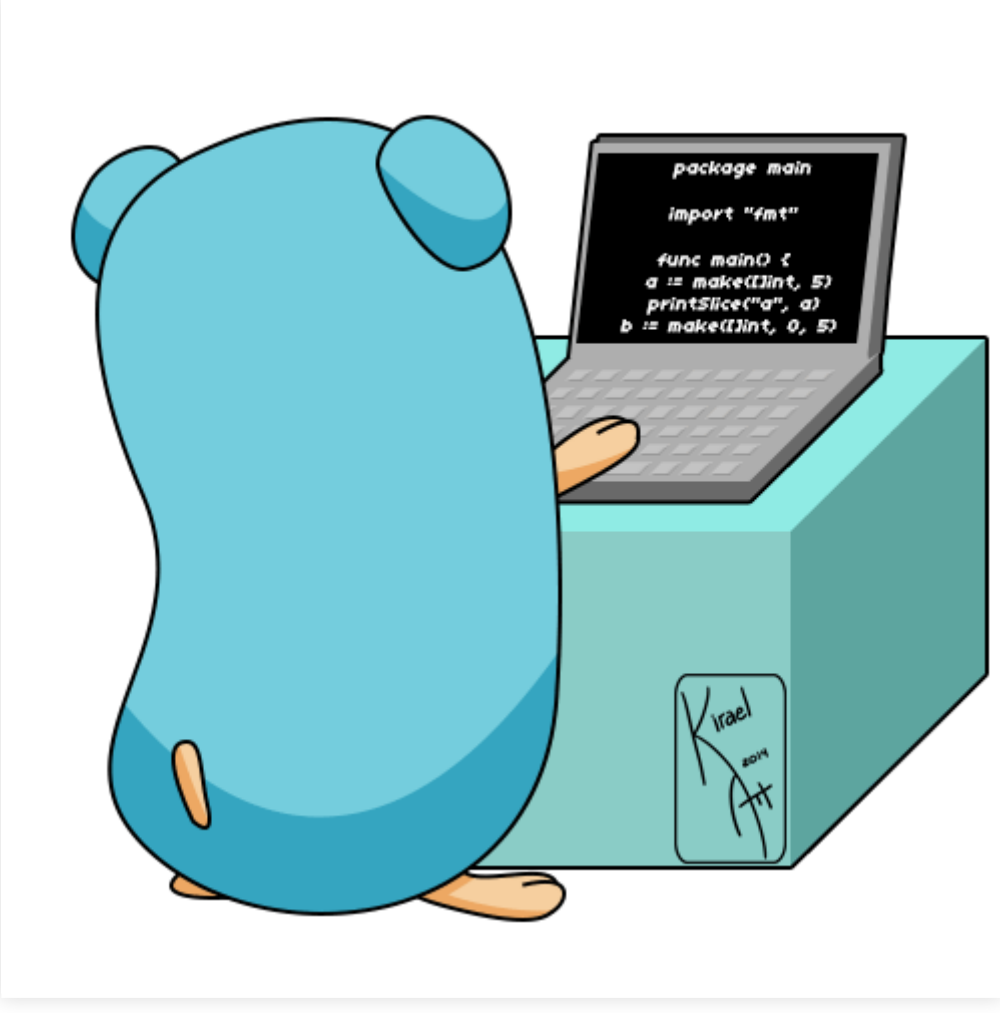
```
go get github.com/go-testfixtures/testfixtures/v3
```

```
fixtures, err := testfixtures.New(
  testfixtures.Database(db),
  testfixtures.Dialect("postgres"),
  testfixtures.Paths(
    "fixtures/products.yml"
  ),
)
```

- можно использовать как CLI утилиту
- можно подключить как библиотеку
- есть встроенный шаблонизатор

Не используйте для тестов отличную СУБД от той, что у вас на prod!

Выполнение запросов



Объект sql.Rows

```
// возвращает имена колонок в выборке
rows.Columns() ([]string, error)
// возвращает типы колонок в выборке
rows.ColumnTypes() ([]*ColumnType, error)
// переходит к следующей строке или возвращает false
rows.Next() bool
// заполняет переменные из текущей строки
rows.Scan(dest ...interface{}) error
// закрывает объект Rows
rows.Close()
// возвращает ошибку, встреченную при итерации
rows.Err() error
```

PreparedStatements

PreparedStatement - это заранее разобранный запрос, который можно выполнять повторно.

PreparedStatement - временный объект, который создается в СУБД и живет в рамках сессии, или пока не будет закрыт.

```
// создаем подготовленный запрос
stmt, err := db.Prepare("delete from events where id = $1") // *sql.Stmt
if err != nil {
 log.Fatal(err)
}
// освобождаем ресурсы в СУБД
defer stmt.Close()
// многократно выполняем запрос
for _, id := range ids {
  _, err = stmt.Exec(id)
  if err != nil {
    log.Fatal(err)
  }
}
```

http://go-database-sql.org/prepared.html

Работа с соединением

`*sql.DB` - это пул соединений. Даже последовательные запросы могут использовать разные соединения с базой.

Если нужно получить одно конкретное соединение, то

```
conn, err := db.Conn(ctx) // *sql.Conn
// вернуть соединение в pool
defer conn.Close()
// далее - обычная работа как с *sql.DB
err := conn.ExecContext(ctx, query1, arg1, arg2)
rows, err := conn.QueryContext(ctx, query2, arg1, arg2)
```

Транзакции

Транзакция - группа запросов, которые либо выполняются, либо не выполняются вместе. Внутри транзакции все запросы видят “согласованное” состояние данных.

На уровне SQL для транзакций используются отдельные запросы: `BEGIN`, `COMMIT` и `ROLLBACK`.

```
tx, err := db.BeginTx(ctx, nil) // *sql.Tx
if err != nil {
  log.Fatal(err)
}
// далее - обычная работа как с *sql.DB
err := tx.ExecContext(ctx, query1, arg1, arg2)
rows, err := tx.QueryContext(ctx, query2, arg1, arg2)
err := tx.Commit() // или tx.Rollback()
if err != nil {
  // commit не прошел, данные не изменились
}
  // далее объект tx не пригоден для использования
```

http://go-database-sql.org/modifying.html

NULL

В SQL базах любая колонка может быть объявлена к NULL / NOT NULL. NULL - это не 0 и не пустая строка, это отсутствие значения.

```
create table users (
  id serial primary key,
  name text not null,
  age int null
);
```

Для обработки `NULL` в Go предлагается использовать специальные типы:

```
var id, realAge int64
var name string
var age sql.NullInt64
err := db.
    QueryRowContext(ctx, "select * from users where id = 1").
    Scan(&id, &name, &age)

if age.Valid {
  realAge = age.Int64
} else {
  // обработка на ваше усмотрение
}
```

SQL Injection

Опасно:

```
query := "select * from users where name = '" + name + "'"
query := fmt.Sprintf("select * from users where name = '%s'", name)
```

Потому что в name может оказаться что-то типа:

```
"jack'; truncate users; select 'pawned"
```

SQL Injection

Правильный подход - использовать placeholders для подстановки значений в SQL:

```
row := db.QueryRowContext(ctx, "select * from users where name = $1", na
```

Однако это не всегда возможно. Так работать не будет:

```
db.QueryRowContext(ctx, "select * from $1 where name = $2", table, name)
db.QueryRowContext(ctx, "select * from user order by $1 limit 3", order)
```

Проверить код на инъекции (и другие проблемы безопасности):
github.com/securego/gosec

Проблемы database/sql

- placeholder зависят от базы: `$1` в Postgres, `?` в MySQL и `:name` в Oracle
- есть только базовые типы, но нет, например `sql.NullDate`
- `rows.Scan(arg1, arg2, arg3)` - неудобен, легко ошибиться
- нет возможности `rows.StructScan(&event)`

Расширение sqlx

```
go get github.com/jmoiron/sqlx
```

`jmoiron/sqlx` - обертка, прозрачно расширяющая стандартную библиотеку `database/sql`:

- `sqlx.DB` - обертка над `*sql.DB`
- `sqlx.Tx` - обертка над `*sql.Tx`
- `sqlx.Stmt` - обертка над `*sql.Stmt`
- `sqlx.NamedStmt` - PreparedStatement с поддержкой именованых параметров

Расширение sqlx

Подключение `jmoiron/sqlx`:

```
import "github.com/jmoiron/sqlx"
// ...
db, err := sqlx.Open("pgx", dsn) // *sqlx.DB
rows, err := db.QueryContext("select * from events") // *sqlx.Rows
...
```

sqlx: именованные placeholder'ы

Можно передавать параметры запроса в виде словаря:

```
sql := "select * from events where owner = :owner and start_date = :star
rows, err := db.NamedQueryContext(ctx, sql, map[string]interface{}{
  "owner": 42,
  "start": "2019-12-31",
})
```

Или структуры:

```
type QueryArgs{
 Owner int64
 Start string
}

sql := "select * from events where owner = :owner and start_date = :star

rows, err := db.NamedQueryContext(ctx, sql, QueryArgs{
 Owner: 42,
 Start: "2019-12-31",
})
```

sqlx: сканирование

Можно сканировать результаты в словарь:

```
sql := "select * from events where start_date > $1"
rows, err := db.QueryContext(ctx, sql, "2020-01-01") // *sqlx.Rows
for rows.Next() {
 results := make(map[string]interface{})
 err := rows.MapScan(results)
 if err != nil {
  log.Fatal(err)
 }
 // обрабатываем result
}
```

sqlx: сканирование

Можно сканировать результаты в структуру:

```
type Event {
  Id int64
  Title string
  Description string `db:"descr"`
}
sql := "select * from events where start_date > $1"
rows, err := db.NamedQueryContext(ctx, sql, "2020-01-01") // *sqlx.Rows
events := make([]Event)
for rows.Next() {
    var event Event
    err := rows.StructScan(&event)
    if err != nil {
      log.Fatal(err)
    }
    events = append(events, event)
}
```

Squirrel - fluent SQL generator for Go

```
go get github.com/Masterminds/squirrel
```

```
import sq "github.com/Masterminds/squirrel"
// ...
sql, args, err := sq.
 Insert("users").Columns("name", "age").
 Values("moe", 13).Values("larry", sq.Expr("? + 5", 12)).
 ToSql()
```

```
sql == "INSERT INTO users (name,age) VALUES (?,?),(?,? + 5)"
```

```
query := sq.Insert(tableName).
  Columns("description").
  Values(task.Description).
  Suffix(`RETURNING "id"`).
  RunWith(r.db).
  PlaceholderFormat(sq.Dollar)

query.QueryRowContext(ctx).Scan(&task.Id)
```

```
  query := sq.Update(tableName).
    SetMap(map[string]interface{}{
      "id": task.Description,
    }).
    Where(sq.Eq{"id": task.Id}).
    RunWith(r.db).
    PlaceholderFormat(sq.Dollar)

  _, err := query.ExecContext(ctx)
  return err
```

Тестирование

https://github.com/DATA-DOG/go-txdb – Single transaction based sql.Driver for GO

Позволяет писать изолированные тесты.

```
func init() {
  txdb.Register("txdb", "mysql", "root@/txdb_test")
}

func main() {
  db, err := sql.Open("txdb", "identifier")
  if err != nil {
    log.Fatal(err)
  }
  defer db.Close()
  if _, err := db.Exec(`INSERT INTO users(username) VALUES("gopher")`);
    log.Fatal(err)
    }
 }
```

Моки

https://github.com/DATA-DOG/go-sqlmock

```
func TestShouldUpdateStats(t *testing.T) {
  db, mock, err := sqlmock.New()
  if err != nil {
    t.Fatalf("an error '%s' was not expected when opening a stub databas
  }
  defer db.Close()

  mock.ExpectBegin()
  mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2, 3)).WillRet
  mock.ExpectCommit()
  if err = recordStats(db, 2, 3); err != nil {
    t.Errorf("error was not expected while updating stats: %s", err)
  }
  if err := mock.ExpectationsWereMet(); err != nil {
    t.Errorf("there were unfulfilled expectations: %s", err)
  }
}
```

Explain

```
explain (analyse, verbose, costs, buffers, format json) -- ...
```

Visualizing and understanding PostgreSQL EXPLAIN plans

Спасибо за внимание

- Go database/sql tutorial go-database-sql.org
- Типы индексов postgrespro.ru/docs/postgresql/14/indexes-types
- Why upgrade PostgreSQL? why-upgrade.depesz.com
- Хорошие новости для тех, кто всё ещё использует row-level локи в PostgreSQLhttps://habr.com/ru/company/ozontech/blog/555358/
- SQL миграции в Postgres. Часть 1 habr.com/en/company/miro/blog/540500/
- Топ ошибок со стороны разработки при работе с PostgreSQL youtube.com/watch?v=HjLnY0aPQZo

Gaiaz Iusipov g.iusipov@gmail.com 2021