Русский

Урок SQLite

Начинаем программировать

Подключим драйвер для sql lite _ «github.com/mattn/go-sqlite3».

Подключимся к СУБД и проверим успешность подключения

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }
}
```

после выполнения этой программы в папке проекта появится файл **store.db**

Обратите внимание, если вы не вызовете метод **PingContext**, то **store.db** не создастся.

Создаем 2 таблицы: **users** и **expressions**, в котором будем хранить пользователей и выражения, которые они отправляют на вычисления.

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/mattn/go-sqlite3"
)

func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        expression TEXT NOT NULL,
        user_id INTEGER NOT NULL,
```

```
        FOREIGN KEY (user_id)  REFERENCES expressions (id)
    );`
    )

    if _, err := db.ExecContext(ctx, usersTable); err != nil {
        return err
    }

    if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
        return err
    }

    return nil
}

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }

    if err = createTables(ctx, db); err != nil {
        panic(err)
    }

}
```

Поскольку строки мы не вычитываем, то мы используем `ExecContext` в функции `createTables`

Посмотрим, как вставлять данные в функцию:

```
package main

import (
    "context"
    "database/sql"

    _ "github.com/mattn/go-sqlite3"
)

type (
    User struct {
        ID      int64
        Name    string
        Balance int64
    }

    Expression struct {
        ID         int64
        Expression string
        UserID     int64
    }
)

func insertUser(ctx context.Context, db *sql.DB, user *User) (int64, error) {
    var q = `
    INSERT INTO users (name, balance) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, user.Name, user.Balance)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
```

```
        return 0, err
    }

    return id, nil
}

func insertExpression(ctx context.Context, db *sql.DB, expression *Expression) (int64, error) {
    var q = `
    INSERT INTO expressions (expression, user_id) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, expression.Expression, expression.UserID)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        expression TEXT NOT NULL,
        user_id INTEGER NOT NULL,

        FOREIGN KEY (user_id)  REFERENCES expressions (id)
    );`
    )

    if _, err := db.ExecContext(ctx, usersTable); err != nil {
        return err
    }

    if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
        return err
    }

    return nil
}

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }

    if err = createTables(ctx, db); err != nil {
        panic(err)
    }

    user := &User{
        Name:    "Petr",
        Balance: 200,
```

```
    }
    userID, err := insertUser(ctx, db, user)
    if err != nil {
        panic(err)
    }

    expression := &Expression{
        Expression: "2+2",
        UserID:     userID,
    }
    expressionID, err := insertExpression(ctx, db, expression)
    if err != nil {
        panic(err)
    }
    expression.ID = expressionID

}
```

Мы завели две структуры: **User** и **Expression**. Это наши модели для представления в коде данных из таблиц СУБД. Попробуйте вставить в в таблицу с помощью этого кода пользователя с отрицательным балансом. Вы получите ошибку `panic: CHECK constraint failed: balance >= 0`. Таким образом, как бы сильно мы не старались ошибиться на уровне кода, ограничения в СУБД не дадут нам модифицировать данные до состояния, которое не имеет смысла. Для этого нужны наши ограничения.

Обратите внимание, на строку `id, err := result.LastInsertId()` здесь мы получаем идентификатором последней вставленной строки.

Давайте научимся получать значения из СУБД.

```
package main

import (
    "context"
    "database/sql"
    "log"
    "strconv"

    _ "github.com/mattn/go-sqlite3"
)

type (
    User struct {
        ID      int64
        Name    string
        Balance int64
    }

    Expression struct {
        ID         int64
        Expression string
        UserID     int64
    }
)

func (u User) Print() string {
    id := strconv.FormatInt(u.ID, 10)
    balance := strconv.FormatInt(u.Balance, 10)
    return "ID: " + id + " Name: " + u.Name + " Balance: " + balance
}
func (e Expression) Print() string {
    id := strconv.FormatInt(e.ID, 10)
    userID := strconv.FormatInt(e.UserID, 10)
    return "ID: " + id + " Expression" + e.Expression + " UserID:" + userID
}

func insertUser(ctx context.Context, db *sql.DB, user *User) (int64, error) {
    var q = `
    INSERT INTO users (name, balance) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, user.Name, user.Balance)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
```

```
    if err != nil {
        return 0, err
    }

    return id, nil
}

func insertExpression(ctx context.Context, db *sql.DB, expression *Expression) (int64, error) {
    var q = `
    INSERT INTO expressions (expression, user_id) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, expression.Expression, expression.UserID)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func selectUsers(ctx context.Context, db *sql.DB) ([]User, error) {
    var users []User
    var q = "SELECT id, name, balance FROM users"
    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        u := User{}
        err := rows.Scan(&u.ID, &u.Name, &u.Balance)
        if err != nil {
            return nil, err
        }
        users = append(users, u)
    }

    return users, nil
}

func selectExpressions(ctx context.Context, db *sql.DB) ([]Expression, error) {
    var expressions []Expression
    var q = "SELECT id, expression, user_id FROM expressions"

    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        e := Expression{}
        err := rows.Scan(&e.ID, &e.Expression, &e.UserID)
        if err != nil {
            return nil, err
        }
        expressions = append(expressions, e)
    }

    return expressions, nil
}

func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
```

```
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        expression TEXT NOT NULL,
        user_id INTEGER NOT NULL,

        FOREIGN KEY (user_id)  REFERENCES expressions (id)
    );`
    )

    if _, err := db.ExecContext(ctx, usersTable); err != nil {
        return err
    }

    if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
        return err
    }

    return nil
}

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }

    if err = createTables(ctx, db); err != nil {
        panic(err)
    }

    user := &User{
        Name:    "Petr",
        Balance: 200,
    }
    userID, err := insertUser(ctx, db, user)
    if err != nil {
        panic(err)
    }

    expression := &Expression{
        Expression: "2+2",
        UserID:     userID,
    }
    expressionID, err := insertExpression(ctx, db, expression)
    if err != nil {
        panic(err)
    }
    expression.ID = expressionID

    users, err := selectUsers(ctx, db)
    if err != nil {
        panic(err)
    }

    for i := range users {
        log.Println(users[i].Print())
    }

    expressions, err := selectExpressions(ctx, db)
    if err != nil {
        panic(err)
    }
```

```
    for i := range expressions {
        log.Println(expressions[i].Print())
    }
}
```

Здесь мы воспользовались функцией `QueryContext` в функциях **selectUsers** и **selectExpressions**, которую мы используем, чтобы получать значения строк из СУБД.

Попробуем получить конкретного пользователя по его идентификатору.

```
package main

import (
    "context"
    "database/sql"
    "log"
    "strconv"

    _ "github.com/mattn/go-sqlite3"
)

type (
    User struct {
        ID      int64
        Name    string
        Balance int64
    }

    Expression struct {
        ID         int64
        Expression string
        UserID     int64
    }
)

func (u User) Print() string {
    id := strconv.FormatInt(u.ID, 10)
    balance := strconv.FormatInt(u.Balance, 10)
    return "ID: " + id + " Name: " + u.Name + " Balance: " + balance
}
func (e Expression) Print() string {
    id := strconv.FormatInt(e.ID, 10)
    userID := strconv.FormatInt(e.UserID, 10)
    return "ID: " + id + " Expression" + e.Expression + " UserID:" + userID
}

func insertUser(ctx context.Context, db *sql.DB, user *User) (int64, error) {
    var q = `
    INSERT INTO users (name, balance) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, user.Name, user.Balance)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func insertExpression(ctx context.Context, db *sql.DB, expression *Expression) (int64, error) {
    var q = `
    INSERT INTO expressions (expression, user_id) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, expression.Expression, expression.UserID)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
```

```
    if err != nil {
        return 0, err
    }

    return id, nil
}

func selectUsers(ctx context.Context, db *sql.DB) ([]User, error) {
    var users []User
    var q = "SELECT id, name, balance FROM users"
    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        u := User{}
        err := rows.Scan(&u.ID, &u.Name, &u.Balance)
        if err != nil {
            return nil, err
        }
        users = append(users, u)
    }

    return users, nil
}

func selectExpressions(ctx context.Context, db *sql.DB) ([]Expression, error) {
    var expressions []Expression
    var q = "SELECT id, expression, user_id FROM expressions"

    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        e := Expression{}
        err := rows.Scan(&e.ID, &e.Expression, &e.UserID)
        if err != nil {
            return nil, err
        }
        expressions = append(expressions, e)
    }

    return expressions, nil
}

func selectUserByID(ctx context.Context, db *sql.DB, id int64) (User, error) {
    u := User{}
    var q = "SELECT id, name, balance FROM users WHERE id = $1"
    err := db.QueryRowContext(ctx, q, id).Scan(&u.ID, &u.Name, &u.Balance)
    if err != nil {
        return u, err
    }

    return u, nil
}

func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
            expression TEXT NOT NULL,
            user_id INTEGER NOT NULL,

            FOREIGN KEY (user_id)  REFERENCES expressions (id)
        );`
        )

        if _, err := db.ExecContext(ctx, usersTable); err != nil {
            return err
        }

        if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
            return err
        }

        return nil
    }

    func main() {
        ctx := context.TODO()

        db, err := sql.Open("sqlite3", "store.db")
        if err != nil {
            panic(err)
        }
        defer db.Close()

        err = db.PingContext(ctx)
        if err != nil {
            panic(err)
        }

        if err = createTables(ctx, db); err != nil {
            panic(err)
        }

        user := &User{
            Name:    "Petr",
            Balance: 200,
        }
        userID, err := insertUser(ctx, db, user)
        if err != nil {
            panic(err)
        }

        expression := &Expression{
            Expression: "2+2",
            UserID:     userID,
        }
        expressionID, err := insertExpression(ctx, db, expression)
        if err != nil {
            panic(err)
        }
        expression.ID = expressionID

        users, err := selectUsers(ctx, db)
        if err != nil {
            panic(err)
        }

        for i := range users {
            log.Println(users[i].Print())
        }

        expressions, err := selectExpressions(ctx, db)
        if err != nil {
            panic(err)
        }

        for i := range expressions {
            log.Println(expressions[i].Print())
        }
```

```
    u, err := selectUserByID(ctx, db, 1)
    if err != nil {
        panic(err)
    }
    log.Println(u.Print())
}
```

Здесь мы воспользовались функцией `QueryRowContext` в функции **selectUserByID**, которую мы используем, чтобы получать значение одной строки из СУБД. Обратите внимание, что если подставить несуществующий в СУБД идентификатор (например 1000), то мы получим ошибку `panic: sql: no rows in result set`

Давайте научимся обновлять значение в СУБД. Обновим баланс конкретного пользователя.

```
package main

import (
    "context"
    "database/sql"
    "log"
    "strconv"

    _ "github.com/mattn/go-sqlite3"
)

type (
    User struct {
        ID      int64
        Name    string
        Balance int64
    }

    Expression struct {
        ID         int64
        Expression string
        UserID     int64
    }
)

func (u User) Print() string {
    id := strconv.FormatInt(u.ID, 10)
    balance := strconv.FormatInt(u.Balance, 10)
    return "ID: " + id + " Name: " + u.Name + " Balance: " + balance
}
func (e Expression) Print() string {
    id := strconv.FormatInt(e.ID, 10)
    userID := strconv.FormatInt(e.UserID, 10)
    return "ID: " + id + " Expression" + e.Expression + " UserID:" + userID
}

func insertUser(ctx context.Context, db *sql.DB, user *User) (int64, error) {
    var q = `
    INSERT INTO users (name, balance) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, user.Name, user.Balance)
    if err != nil {
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func insertExpression(ctx context.Context, db *sql.DB, expression *Expression) (int64, error) {
    var q = `
    INSERT INTO expressions (expression, user_id) values ($1, $2)
    `
    result, err := db.ExecContext(ctx, q, expression.Expression, expression.UserID)
    if err != nil {
        return 0, err
```

```
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func selectUsers(ctx context.Context, db *sql.DB) ([]User, error) {
    var users []User
    var q = "SELECT id, name, balance FROM users"
    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        u := User{}
        err := rows.Scan(&u.ID, &u.Name, &u.Balance)
        if err != nil {
            return nil, err
        }
        users = append(users, u)
    }

    return users, nil
}

func selectExpressions(ctx context.Context, db *sql.DB) ([]Expression, error) {
    var expressions []Expression
    var q = "SELECT id, expression, user_id FROM expressions"

    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        e := Expression{}
        err := rows.Scan(&e.ID, &e.Expression, &e.UserID)
        if err != nil {
            return nil, err
        }
        expressions = append(expressions, e)
    }

    return expressions, nil
}

func selectUserByID(ctx context.Context, db *sql.DB, id int64) (User, error) {
    u := User{}
    var q = "SELECT id, name, balance FROM users WHERE id = $1"
    err := db.QueryRowContext(ctx, q, id).Scan(&u.ID, &u.Name, &u.Balance)
    if err != nil {
        return u, err
    }

    return u, nil
}

func updateUser(ctx context.Context, db *sql.DB, id int64, diff int64) error {
    var q = "UPDATE users SET balance = balance+$1 WHERE id = $2"
    _, err := db.ExecContext(ctx, q, diff, id)
    if err != nil {
        return err
    }

    return nil
}
```

```
func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        expression TEXT NOT NULL,
        user_id INTEGER NOT NULL,

        FOREIGN KEY (user_id)  REFERENCES expressions (id)
    );`
    )

    if _, err := db.ExecContext(ctx, usersTable); err != nil {
        return err
    }

    if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
        return err
    }

    return nil
}

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }

    if err = createTables(ctx, db); err != nil {
        panic(err)
    }

    user := &User{
        Name:    "Petr",
        Balance: 200,
    }
    userID, err := insertUser(ctx, db, user)
    if err != nil {
        panic(err)
    }

    expression := &Expression{
        Expression: "2+2",
        UserID:     userID,
    }
    expressionID, err := insertExpression(ctx, db, expression)
    if err != nil {
        panic(err)
    }
    expression.ID = expressionID

    users, err := selectUsers(ctx, db)
    if err != nil {
        panic(err)
    }

    for i := range users {
```

```
        log.Println(users[i].Print())
    }

    expressions, err := selectExpressions(ctx, db)
    if err != nil {
        panic(err)
    }

    for i := range expressions {
        log.Println(expressions[i].Print())
    }

    err = updateUser(ctx, db, 1, -20)
    if err != nil {
        panic(err)
    }

    u, err := selectUserByID(ctx, db, 1)
    if err != nil {
        panic(err)
    }
    log.Println(u.Print())
}
```

Функция **updateUser** изменяет баланс пользователя с заданным **id** на **diff**. Здесь мы не увидели ничего нового — всё то же самое. Просто используем **UPDATE** вместо **INSERT**. Попробуйте сделать баланс пользователя отрицательным, чтобы убедиться, что наше ограничение работает.

Давайте теперь поговорим о транзакции. Представьте себе, что каждая вычисляемая операция стоит денег. Таким образом, нам одновременно нужно вставить операцию в СУБД и обновить баланс пользователя.

```
package main

import (
    "context"
    "database/sql"
    "log"
    "strconv"

    _ "github.com/mattn/go-sqlite3"
)

type (
    User struct {
        ID      int64
        Name    string
        Balance int64
    }

    Expression struct {
        ID         int64
        Expression string
        UserID     int64
    }
)

func (u User) Print() string {
    id := strconv.FormatInt(u.ID, 10)
    balance := strconv.FormatInt(u.Balance, 10)
    return "ID: " + id + " Name: " + u.Name + " Balance: " + balance
}
func (e Expression) Print() string {
    id := strconv.FormatInt(e.ID, 10)
    userID := strconv.FormatInt(e.UserID, 10)
    return "ID: " + id + " Expression" + e.Expression + " UserID:" + userID
}

func insertExpression(ctx context.Context, tx *sql.Tx, expression *Expression) (int64, error) {
    var q = `
    INSERT INTO expressions (expression, user_id) values ($1, $2)
    `
    result, err := tx.ExecContext(ctx, q, expression.Expression, expression.UserID)
    if err != nil {
```

```
        return 0, err
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, err
    }

    return id, nil
}

func selectUsers(ctx context.Context, db *sql.DB) ([]User, error) {
    var users []User
    var q = "SELECT id, name, balance FROM users"
    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        u := User{}
        err := rows.Scan(&u.ID, &u.Name, &u.Balance)
        if err != nil {
            return nil, err
        }
        users = append(users, u)
    }

    return users, nil
}

func selectExpressions(ctx context.Context, db *sql.DB) ([]Expression, error) {
    var expressions []Expression
    var q = "SELECT id, expression, user_id FROM expressions"

    rows, err := db.QueryContext(ctx, q)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        e := Expression{}
        err := rows.Scan(&e.ID, &e.Expression, &e.UserID)
        if err != nil {
            return nil, err
        }
        expressions = append(expressions, e)
    }

    return expressions, nil
}

func selectUserByID(ctx context.Context, db *sql.DB, id int64) (User, error) {
    u := User{}
    var q = "SELECT id, name, balance FROM users WHERE id = $1"
    err := db.QueryRowContext(ctx, q, id).Scan(&u.ID, &u.Name, &u.Balance)
    if err != nil {
        return u, err
    }

    return u, nil
}

func updateUser(ctx context.Context, tx *sql.Tx, id int64, diff int64) error {
    var q = "UPDATE users SET balance = balance+$1 WHERE id = $2"
    _, err := tx.ExecContext(ctx, q, diff, id)
    if err != nil {
        return err
    }

    return nil
}
```

```
func createTables(ctx context.Context, db *sql.DB) error {
    const (
        usersTable = `
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        balance INTEGER NOT NULL CHECK(balance >= 0)
    );`

        expressionsTable = `
    CREATE TABLE IF NOT EXISTS expressions(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        expression TEXT NOT NULL,
        user_id INTEGER NOT NULL,

        FOREIGN KEY (user_id)  REFERENCES expressions (id)
    );`
    )

    if _, err := db.ExecContext(ctx, usersTable); err != nil {
        return err
    }

    if _, err := db.ExecContext(ctx, expressionsTable); err != nil {
        return err
    }

    return nil
}

func main() {
    ctx := context.TODO()

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    err = db.PingContext(ctx)
    if err != nil {
        panic(err)
    }

    if err = createTables(ctx, db); err != nil {
        panic(err)
    }

    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        panic(err)
    }

    err = updateUser(ctx, tx, 1, -20)
    if err != nil {
        tx.Rollback()
        panic(err)
    }

    expression := &Expression{
        Expression: "2-2",
        UserID:     1,
    }
    expressionID, err := insertExpression(ctx, tx, expression)
    if err != nil {
        panic(err)
    }
    expression.ID = expressionID

    tx.Commit()

    users, err := selectUsers(ctx, db)
```

```
    if err != nil {
        panic(err)
    }

    for i := range users {
        log.Println(users[i].Print())
    }

    expressions, err := selectExpressions(ctx, db)
    if err != nil {
        panic(err)
    }

    for i := range expressions {
        log.Println(expressions[i].Print())
    }
}
```

Обратите внимание, что функции вставки выражения и обновления пользователя практически не поменялись. Мы теперь вызываем эти методы у экземпляра структуры транзакии. **db.BeginTx** — стартует транзакцию. Все перечисленные функции доступны у транзакции так же, как и у подключения к базе данных.

tx.Rollback() - откатывает транзакцию. Все выполненные до этого момента операции откатываются до исходного состояния.

Справка