Hướng dẫn sử dụng thư viện `vdb` trong Go

Thư viện `vdb` cung cấp một cách tiện lợi để làm việc với cơ sở dữ liệu trong Go, hỗ trợ quản lý tenant database, thực hiện các thao tác CRUD (Create, Read, Update, Delete), và xử lý các truy vấn phức tạp như join. Tài liệu này hướng dẫn cách sử dụng `vdb` dựa trên các ví dụ test được cung cấp.

1. Thiết lập cơ bản

1.1. Khởi tạo database quản lý tenant
Để sử dụng `vdb`, trước tiên cần thiết lập database quản lý tenant. Database này không tự động migrate các model, mà chỉ dùng để quản lý thông tin tenant.

```go
vdb.SetManagerDb("mysql", "tenant_manager")
```

- **Hàm `SetManagerDb`**:
  - **Tham số**:
    - `driver`: Loại driver database (`mysql`, `sqlserver`, `postgres`).
    - `dbName`: Tên database quản lý tenant.
  - **Ý nghĩa**: Chỉ định database quản lý tenant để `vdb` thực hiện các thao tác liên quan.

1.2. Khởi tạo kết nối database
Sử dụng hàm `initDb` để khởi tạo kết nối với database quản lý tenant.

```go
func initDb(driver string, conn string) {
    var err error
    db, err = vdb.Open(driver, conn)
    if err != nil {
        panic(err)
    }
    err = db.Ping()
    if err != nil {
        panic(err)
    }
}
```

- **Hàm `vdb.Open`**:
  - **Tham số**:
    - `driver`: Tên driver (`mysql`, `sqlserver`, `postgres`).
    - `conn`: Chuỗi kết nối (connection string).
  - **Kết quả**: Trả về đối tượng `*vdb.TenantDB` để quản lý tenant.
- **Kiểm tra kết nối**: Sử dụng `db.Ping()` để đảm bảo kết nối thành công.

1.3. Tạo database tenant
Tạo một database tenant mới bằng hàm `CreateDB`.

```go
testDb, err = db.CreateDB("test001")
assert.NoError(t, err)
assert.Equal(t, "test001", testDb.GetDBName())
```

- **Hàm `CreateDB`**:
  - **Tham số**: Tên database tenant (ví dụ: `test001`).

  - **Kết quả**: Trả về đối tượng `*vdb.TenantDB` đại diện cho tenant database mới.
- **Lưu ý**: Đảm bảo database quản lý tenant đã được thiết lập trước khi gọi `CreateDB`.

2. Thao tác CRUD với `vdb`

2.1. Thêm bản ghi (Insert)
Sử dụng hàm `Insert` để thêm một hoặc nhiều bản ghi vào database.

```go
user := &models.User{
    UserId:       vdb.Ptr(uuid.NewString()),
    Email:        "test@test.com",
    Phone:        "0987654321",
    Username:     vdb.Ptr("test001"),
    HashPassword: vdb.Ptr("123456"),
    BaseModel: models.BaseModel{
        Description: vdb.Ptr("test"),
        CreatedAt:   vdb.Ptr(time.Now()),
    },
}
err := testDb.Insert(user)
```

- **Hàm `vdb.Ptr`**:
  - Chuyển đổi giá trị thành con trỏ (`*T`) để hỗ trợ các trường có thể là `NULL`.
  - Ví dụ: `vdb.Ptr("test001")` trả về `*string`.
- **Hàm `Insert`**:
  - **Tham số**: Một hoặc nhiều struct model (ví dụ: `user`, `position`, `dept`).
  - **Xử lý lỗi**:
    - Nếu xảy ra lỗi, kiểm tra kiểu lỗi bằng `*vdb.DialectError`:
      ```go
      if vdbErr, ok := err.(*vdb.DialectError); ok {
          if vdbErr.ErrorType == vdb.DIALECT_DB_ERROR_TYPE_DUPLICATE {
              assert.Equal(t, []string{"Email"}, vdbErr.Fields)
              assert.Equal(t, []string{"email"}, vdbErr.DbCols)
              assert.Equal(t, "users", vdbErr.Table)
          }
      }
      ```
    - Các loại lỗi:
      - `DIALECT_DB_ERROR_TYPE_DUPLICATE`: Vi phạm ràng buộc duy nhất.
      - `DIALECT_DB_ERROR_TYPE_REFERENCES`: Vi phạm ràng buộc khóa ngoại.
      - `DIALECT_DB_ERROR_TYPE_REQUIRED`: Thiếu trường bắt buộc.
      - `DIALECT_DB_ERROR_TYPE_LIMIT_SIZE`: Vượt quá kích thước trường.

- **Thêm nhiều bản ghi trong transaction**:
  ```go
  tx, err := testDb.Begin()
  err = testDb.Insert(position, dept, user)
  if err != nil {
      tx.Rollback()
  } else {
      tx.Commit()
  }
  ```
  - Sử dụng `Begin`, `Commit`, `Rollback` để quản lý transaction, đảm bảo tính toàn vẹn dữ liệu.

2.2. Truy vấn bản ghi (Read)
Sử dụng hàm `First` để lấy bản ghi đầu tiên khớp với điều kiện.

```go
user := &models.User{}
err := testDb.First(user, "id = ?", 1)
assert.NoError(t, err)
assert.Equal(t, "test001", *user.Username)
```

- **Hàm `First`**:
  - **Tham số**:
    - Struct model để lưu kết quả.
    - Điều kiện truy vấn (ví dụ: `"id = ?"`).
    - Các tham số thay thế cho `?`.
  - **Kết quả**: Điền dữ liệu vào struct model và trả về lỗi nếu có.

2.3. Cập nhật bản ghi (Update)
`vdb` hỗ trợ nhiều cách cập nhật bản ghi:

a. Cập nhật một trường
```go
result := testDb.Model(&models.User{}).Where("id = ?", 1).Update("Username",
"test002")
assert.NoError(t, result.Error)
assert.Equal(t, int64(1), result.RowsAffected)
```

- **Hàm `Update`**:
  - **Tham số**:
    - Tên trường (theo PascalCase).
    - Giá trị mới.
  - **Kết quả**: Trả về đối tượng `result` với `RowsAffected` (số dòng bị ảnh
hưởng).

b. Cập nhật nhiều trường bằng map
```go
result := testDb.Model(&models.User{}).Where("id = ?", 1).Update(
    map[string]interface{}{
        "Username": "test003",
        "Email":    "william.henry.harrison@example-pet-store.com",
    },
)
```

- **Sử dụng map**:
  - Key là tên field (không phân biệt hoa thường, `vdb` tự động ánh xạ).
  - Value là giá trị mới.

c. Cập nhật với hàm database
Sử dụng `vdb.DbFunCall` để gọi hàm database trong câu lệnh update.

```go
result := testDb.Model(&models.User{}).Where("id = ?", 1).Update(
    "Username", vdb.DbFunCall("CONCAT(UPPER(Username),?)", "test003"))
```

- **Hàm `DbFunCall`**:
  - **Tham số**:

    - `expr`: Tên hàm database (ví dụ: `CONCAT`, `UPPER`, `LOWER`).
    - `args`: Các tham số truyền vào hàm.
  - **Ví dụ**: `CONCAT(UPPER(Username), 'test003')` nối chuỗi `Username` (viết hoa) với `test003`.

d. Kết hợp map và hàm database
```go
result := testDb.Model(&models.User{}).Where("email like ?",
testDb.LikeValue("*.edu")).Update(
    map[string]interface{}{
        "Username":    vdb.DbFunCall("lower(Username)"),
        "Email":       vdb.DbFunCall("CONCAT(UPPER(Email),?)", ".com"),
        "phone":       vdb.DbFunCall("CONCAT(LEFT(Phone,3),?)", "-123456"),
        "description": "Hệ thống sẽ tự động sửa tên field đúng với tên field trong database",
    },
)
```

- **Hàm `LikeValue`**:
  - Chuyển đổi mẫu `LIKE` (ví dụ: `*.edu` → `%.edu` cho MySQL).
- **Lưu ý**: `vdb` tự động ánh xạ tên field không phân biệt hoa thường sang tên cột trong database.

3. Truy vấn phức tạp (Join)

3.1. Inner Join, Left Join, Right Join, Full Join
Sử dụng các hàm join như `LeftJoin` để kết hợp dữ liệu từ nhiều bảng.

```go
type QueryResult struct {
    FullName     *string
    PositionID   *int64
    DepartmentID *int64
    Email        *string
    Phone        *string
}
items := []QueryResult{}
qr := testDb.From((&models.Employee{}).As("e")).LeftJoin(
    (&models.User{}).As("u"), "e.id = u.userId",
).Select(
    "concat(e.FirstName,' ', e.LastName) as fullName",
    "e.positionId",
    "e.departmentId",
    "u.email",
    "u.phone",
)
err := qr.ToArray(&items)
```

- **Hàm `From` và `As`**:
  - `From`: Chỉ định bảng chính (model).
  - `As`: Đặt bí danh (alias) cho bảng.
- **Hàm `LeftJoin`**:
  - **Tham số**:
    - Model và bí danh của bảng được join.
    - Điều kiện join (ví dụ: `e.id = u.userId`).
- **Hàm `Select`**:
  - Chỉ định các cột hoặc biểu thức (ví dụ: `concat(e.FirstName,' ', e.LastName) as

fullName`).
- **Hàm `ToArray`**:
  - Điền kết quả truy vấn vào slice của struct (ví dụ: `[]QueryResult`).

3.2. Self Join
Sử dụng self-join để truy vấn mối quan hệ trong cùng một bảng.

```go
type QueryResult struct {
    Name      *string
    ChildName *string
}
items := []QueryResult{}
qr := testDb.From(
    (&models.Department{}).As("d"),
).LeftJoin(
    (&models.Department{}).As("c"), "d.id = c.parentId",
).Select(
    "d.name",
    "c.name as childName",
)
err := qr.ToArray(&items)
```

- **Lưu ý**:
  - Các trường trong `QueryResult` phải là **PascalCase** vì Go chỉ điền giá trị vào các field exported.
  - `vdb` tự động chuyển tên cột sang PascalCase để ánh xạ vào struct.

4. Lưu ý quan trọng

- **Đăng ký model**: Đảm bảo các model được đăng ký trong `vdb.ModelRegistry` (thường trong hàm `init` của package `models`).
- **Transaction**: Sử dụng `Begin`, `Commit`, `Rollback` khi thực hiện nhiều thao tác để đảm bảo tính toàn vẹn dữ liệu.
- **Xử lý lỗi**: Kiểm tra lỗi `vdb.DialectError` để xác định nguyên nhân cụ thể (duplicate, references, required, v.v.).
- **Field không phân biệt hoa thường**: `vdb` tự động ánh xạ tên field (ví dụ: `phone` hoặc `Phone`) sang tên cột trong database.
- **Driver hỗ trợ**: Hiện tại, `vdb` hỗ trợ `mysql`, `sqlserver`, và `postgres`.

5. Kết luận

Thư viện `vdb` cung cấp một giao diện mạnh mẽ và linh hoạt để làm việc với cơ sở dữ liệu trong Go, đặc biệt phù hợp với mô hình multi-tenant. Với các tính năng như quản lý tenant, CRUD, join, và hỗ trợ hàm database, `vdb` giúp đơn giản hóa việc phát triển ứng dụng cơ sở dữ liệu. Hãy đảm bảo thiết lập đúng database quản lý tenant và đăng ký các model trước khi sử dụng.