开发

SQLBuilder

SQLBuilder是一个用于生成SQL语句的库。

项目：

- https://gitee.com/iRainIoT/go-sqlbuilder、https://github.com/parkingwang/go-sqlbuilder
  - 已支持MySQL基本Select/Update/Insert/Delete/Where等语法
  - 目前只支持MySQL语法
  - 未支持多表查询
  - ```
    $ go get -u github.com/parkingwang/go-sqlbuilder
    ```
- https://github.com/huandu/go-sqlbuilder，功能更强
  - ```
    $ go get github.com/huandu/go-sqlbuilder
    ```



```go
package main

import (
    "database/sql"
    "fmt"
    "log"

    _ "github.com/go-sql-driver/mysql" // mysql驱动
    "github.com/huandu/go-sqlbuilder"
)

type Emp struct {
    emp_no     int
    first_name string
    last_name  string
    gender     byte
    birth_date string
}

func main() {
    db, err := sql.Open("mysql", "wayne:wayne@/test") // 打开数据库
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    query := sqlbuilder.Select("emp_no", "first_name", "last_name",
"gender", "birth_date").
        From("employees").
        Where("emp_no > 10015").  // 试一试Where("emp_no > ?")
```

```
        Offset(2).Limit(2).
        OrderBy("emp_no").Desc(). // 按照什么字段排序，降序
        String()                  // 输出为字符串，底层调用Build()
    fmt.Println(query)
    rows, err := db.Query(query)
    if err != nil {
        log.Fatalln(err)
    }

    for rows.Next() {
        var e Emp
        // 要和上面select的字段顺序对应
        rows.Scan(&e.emp_no, &e.first_name, &e.last_name, &e.gender,
&e.birth_date)
        fmt.Println(e)
    }
}

SELECT emp_no, first_name, last_name, gender, birth_date FROM employees
WHERE emp_no > 10015 ORDER BY emp_no DESC LIMIT 2 OFFSET 2
```

本质上sqlbuilder就是在生成SQL语句字符串。

args参数化

```
builder := sqlbuilder.Select("emp_no", "first_name", "last_name", "gender",
"birth_date").
From("employees")
builder.Where(
    builder.In("emp_no", 10008, 10010, 10020), // 参数化
)
query, args := builder.Build()
fmt.Printf("%s\n%v\n", query, args) // args是参数

SELECT emp_no, first_name, last_name, gender, birth_date FROM employees
WHERE emp_no IN (?, ?, ?)
[10008 10010 10020]
```

ORM

对象关系映射（Object Relational Mapping，ORM）。指的是对象和关系之间的映射，使用面向对象的方式操作数据库。

```
关系模型和Go对象之间的映射
table  => struct    ，表映射为结构体
row    => object    ，行映射为实例
column => property  ，字段映射为属性
```

举例，有表student，字段为id int，name varchar，age int

```go
type Student struct {
    id   int
    name string
    age  int
}

Student{100, "Tom", 20}
Student{101, "Jerry", 18}
```

可以认为ORM是一种高级抽象，对象的操作最终还是会转换成对应关系数据库操作的SQL语句，数据库操作的结构会被封装成对象。

GORM

GORM是一个友好的、功能全面的、性能不错的基于Go语言实现的ORM库。

安装

gorm.io/driver/mysql依赖github.com/go-sql-driver/mysql，可以认为它是对驱动的再封装。

```shell
$ go get -u github.com/go-sql-driver/mysql

$ go get -u gorm.io/gorm
$ go get -u gorm.io/driver/mysql
```

文档

- 英文 https://gorm.io/docs/
- 中文 https://gorm.io/zh_CN/docs/index.html

连接

https://gorm.io/zh_CN/docs/connecting_to_the_database.html#MySQL

```go
package main

import (
    "fmt"
    "log"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

var db *gorm.DB

func init() {
    // dsn := "wayne:wayne@/test"
    dsn := "wayne:wayne@tcp(localhost:3306)/test?charset=utf8mb4"
    var err error
    db, err = gorm.Open(mysql.Open(dsn), &gorm.Config{}) // 不要用:=
    if err != nil {
        log.Fatalln(err)
    }
```

```
    fmt.Println(db)
}
```

模型定义

https://gorm.io/zh_CN/docs/models.html

GORM 倾向于约定优于配置，如果不遵从约定就要写自定义配置

- 使用名为ID的属性会作为主键
- 使用snake_cases作为表名
  - 结构体命名为employee，那么数据库表名就是employees
- 使用snake_case作为字段名，字段首字母大写

```
// 不符合约定的定义，很多都需要配置，直接用不行
type Emp struct { // 默认表名emps
    emp_no     int    // 不是ID为主键，需要配置
    first_name string // 首字母未大写，也需要配置
    last_name  string
    gender     byte
    birth_date string
}

// 符合约定的定义如下
type student struct { // 默认表名students
    ID   int    // Id也可以
    Name string // 字段首字母要大写
    Age  int
}
```

表名配置

```
// 表名并没有遵守约定
func (Emp) TableName() string {
    return "employees"
}
```

字段配置

```
package main

import (
    "fmt"
    "log"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"
    "gorm.io/gorm/logger"
)

```

```go
var db *gorm.DB

func init() {
    dsn := "wayne:wayne@tcp(localhost:3306)/test?charset=utf8mb4"
    var err error
    db, err = gorm.Open(mysql.Open(dsn), &gorm.Config{
        Logger: logger.Default.LogMode(logger.Info),
    }) // 不要用:=
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Println(db)
}

type Emp struct { // 默认表名emps
    EmpNo     int    `gorm:"primaryKey"` // 不是ID为主键
    FirstName string // 首字母大写，对应字段first_name
    LastName  string
    Gender    byte
    BirthDate string
}

// 表名并没有遵守约定
func (Emp) TableName() string {
    return "employees"
}

func main() {
    var e Emp
    row := db.Take(&e) // 等价于Limit 1，取1条
    fmt.Println(row)
    fmt.Println(row.Error)
    fmt.Println(e)
}
```

使用`gorm:"primaryKey"`来指定字段为主键，默认使用名为ID的属性作为主键。primaryKey是tag名，大小写不敏感，但建议小驼峰。

列名

如果未按照约定定义字段，需要定义结构体属性时指定数据库字段名称是什么。

```go
BirthDate string `gorm:"column:birth_date"` // 字段名可以不符合约定，但字段名首字母一定要大写
Xyz string `gorm:"column:birth_date"` // 可以
```

迁移

https://gorm.io/zh_CN/docs/migration.html#%E8%A1%A8

下面，新建一个students表，来看看结构体中属性类型和数据库表中字段类型的对应关系

```go
// 迁移后，主键默认不为空，其他字段默认都是能为空的
```

```
type Student struct {
    ID       int       // 缺省主键bigint AUTO_INCREMENT
    Name     string    `gorm:"not null;type:varchar(48);comment:姓名"`
    Age      byte      // byte=>tinyint unsigned
    Birthday time.Time // datetime
    Gender   byte      `gorm:"type:tinyint"`
}

db.Migrator().CreateTable(&Student{})

CREATE TABLE `students` (
    `id` bigint AUTO_INCREMENT,
    `name` varchar(48) NOT NULL COMMENT '姓名',
    `age` tinyint unsigned,
    `birthday` datetime(3) NULL,
    `gender` tinyint,
    PRIMARY KEY (`id`)
)
```

由于int => bigint、string => longtext，这些默认转换不符合我们的要求，所以，在tag中使用type指定字段类型。

属性是用来构建结构体实例的，生成的SQL语句也要使用这些数据。而tag是用来生成迁移

```
Name     string     `gorm:"size:48"` 定义为varchar(48)
Age      int        `gorm:"size:32"` 定义为4字节的int
Age      int        `gorm:"size:64"` 定义为8字节的bigint
```

迁移用的较少，主要让大家理解其作用。

结构体属性类型用来封装实例的数据，Tag中类型指定迁移到数据库表中字段的类型

新增

参考 https://gorm.io/zh_CN/docs/create.html#%E5%88%9B%E5%BB%BA%E8%AE%B0%E5%BD%95

```
type Student struct {
    ID       int       // 缺省主键bigint AUTO_INCREMENT
    Name     string    `gorm:"size:48"` //`gorm:"not null;type:varchar(48);comment:姓名"`
    Age      byte      // byte=>tinyint unsigned
    Birthday time.Time // datetime
    Gender   byte      //`gorm:"type:tinyint"`
}

func (s *Student) String() string {
    return fmt.Sprintf("%d", s.ID)
}
```

```go
// 新增一条
n := time.Now()
s := Student{Name: "Tom", Age: 20, Birthday: &n}
fmt.Println(s)
result := db.Create(&s) // 新增，传入指针
fmt.Println(s) // 注意前后ID的变化
fmt.Println(result.Error)
fmt.Println(result.RowsAffected)
```

```go
// 新增多条
n := time.Now()
s := Student{Name: "Tom", Age: 20, Birthday: &n}
fmt.Println(s)
result := db.Create([]*Student{&s, &s, &s}) // 传入指针的切片
fmt.Println(s)
fmt.Println(result.Error)
fmt.Println(result.RowsAffected)
```

查询一条

Take被转换为Limit 1。

```go
var s Student
fmt.Println(s) // 零值
row := db.Take(&s) // LIMIT 1
fmt.Println(s) // 被填充
fmt.Println(row)
fmt.Println(row.Error)
```

```go
row := db.First(&s) // ORDER BY `students`.`id` LIMIT 1
```

```go
row := db.Last(&s) // ORDER BY `students`.`id` DESC LIMIT 1
```

根据id查，可以使用下面的方式

```go
row := db.First(&s, 15)
```

```go
var s = Student{ID: 16}
row := db.First(&s)
```

时间相关错误

1、时间类型字段

上例错误如下，主要是使用了*time.Time，而不是string。

```
sql: Scan error on column index 3, name "birthday": unsupported Scan, storing driver.Value type []uint8 into type *time.Time
[]byte 转 *time.Time失败了
```

解决方案

- 在连接字符串中增加parseTime=true，这样时间类型就会自动转化为time.Time类型
  - ```
    dsn := "wayne:wayne@tcp(localhost:3306)/test?charset=utf8mb4&parseTime=true"
    ```
- 也可以`Birthday string`，拿到Birthday字符串后，必要时，自行转换成时间类型

2、UTC时间

Create写入的时间，也就是说time.Now()取当前时区时间，但是存入数据库的时间是UTC时间。

Take拿回的时间也是UTC时间，可以通过s.Birthday.Local()转成当前时区时间。

如果想存入的时间或读取的时间直接是当前时区时间，可以使用loc参数loc=Local。

如果loc=Local

- 存入时，数据库字段中的时间就是当前时区的时间值
- 读取时，数据库字段中的时间就被解读为当前时区

```
dsn := "wayne:wayne@tcp(localhost:3306)/test?charset=utf8mb4&parseTime=true&loc=Local"

// time/zoneinfo.go
func LoadLocation(name string) (*Location, error) {
    if name == "" || name == "UTC" {
        return UTC, nil
    }
    if name == "Local" {
        return Local, nil
    }
    ...省略
}
```

千万不要存入数据库时采用Local存入，却使用UTC解读时间，会造成时间时区的混乱。应该保证时间存入、读取时区一致。

一定要统一项目中数据库中时间类型字段的时区。可以考虑统一采用UTC，为了本地化显示转换为当前时区即可。

查询所有

```
var students []*Student
rows := db.Find(&students)
fmt.Println(students)
fmt.Println(rows)
fmt.Println(rows.Error)
```
```

distinct

```
rows := db.Distinct("name").Find(&students)
```

投影

投影是关系模型的操作，就是选择哪些字段。

```
rows := db.Select("id", "name", "age").Find(&students)
rows := db.Select([]string{"id", "name", "age"}).Find(&students)
```

Limit和Offset

```
var students []*Student
rows := db.Limit(2).Offset(2).Find(&students)
```

条件查询

1、字符串条件

```
var students []*Student
rows := db.Where("name = ?", "Tom").Find(&students)
rows := db.Where("name <> ?", "Tom").Find(&students)
rows := db.Where("name in ?", []string{"jerry", "tom"}).Find(&students)
rows := db.Where("name like ?", "t%").Find(&students)
rows := db.Where("name like binary ?", "T%").Find(&students)
rows := db.Where("name like ? and age > ?", "t%", 20).Find(&students)
rows := db.Where("id between ? and ?", 15, 17).Find(&students) // id范围[15, 17]
rows := db.Where("id = ? or id = ?", 15, 17).Find(&students) // or
```

2、struct或map条件

```
rows := db.Where([]int{14, 16, 17}).Find(&students) // WHERE `students`.`id` IN (14,16,17)
rows := db.Where(&Student{}).Find(&students) // find all
rows := db.Where(&Student{ID: 15}).Find(&students)
rows := db.Where(&Student{ID: 15, Name: "Tom"}).Find(&students) // and
rows := db.Where(map[string]interface{}{"name": "Tom", "id": 16}).Find(&students) // and
```

struct条件中出现了零值，例如`db.Where(&Student{Name: "Tom", Age: 0})`，Age是零值，就不会出现在条件中。

```
rows := db.Where(&Student{Name: "Tom", Age: 20}, "name", "age").Find(&students) // 指定使用结构体里面的name和age字段作为条件，and
```

3、Not

将Where换成Not即可，表示条件取反。

```
rows := db.Not("id = ? or id = ?", 15, 17).Find(&students)
rows := db.Not("name = ?", "Tom").Find(&students)
```

4、Or

Or的用法和Where一样。

Where().Where()是And关系，Where().Or()是Or关系。

```
rows := db.Where("name = ?", "Tom").Or("name=?", "Jerry").Find(&students)
rows := db.Where("name = ?", "Tom").Or(&Student{Name:
"Jerry"}).Find(&students)
```

排序

```
rows := db.Order("id desc").Find(&students) // ORDER BY id desc

rows := db.Order("name, id desc").Find(&students)         // ORDER BY name,id
desc
rows := db.Order("name").Order("id desc").Find(&students) // ORDER BY name,id
desc
```

分组

```
rows := db.Group("id").Find(&students) // GROUP BY `id`
rows := db.Group("name").Find(&students) // GROUP BY `name`
rows := db.Group("id").Group("name").Find(&students) // GROUP BY `id`,`name`
```

```
// SELECT name, count(id) as c FROM `students` GROUP BY `name`
rows := db.Select("name, count(id) as c").Group("name").Find(&students)
// 但是students中没有属性来保存count的值
```

```
// 使用Rows()返回所有行，自行获取字段值，但是要用Table指定表名
type Result struct {
    name  string
    count int
}
var r = Result{}
rows, err := db.Table("students").Select("name, count(id) as
c").Group("name").Rows()
fmt.Println(err)
// 遍历每一行，填充2个属性的结构体实例
for rows.Next() {
    rows.Scan(&r.name, &r.count)
    fmt.Println(r, "@@@")
}
```

```
type Result struct { // 和Select的投影字段对应
    Name  string
    Count int
```

```
}
var r = Result{}
rows, err := db.Table("students").Select("name, count(id) as c").Group("name").Having("c > 3").Rows()
fmt.Println(err)
// 遍历每一行，填充2个属性的结构体实例
for rows.Next() {
    rows.Scan(&r.Name, &r.Count)
    fmt.Println(r, "@@@")
}

// 使用Scan填充容器
type Result struct {
    Name string
    C    int
}
var rows = []*Result{}
db.Table("students").Select("name, count(id) as c").Group("name").Having("c > 3").Scan(&rows)
for i, r := range rows {
    fmt.Printf("%d, %T %#[2]v\n", i, r)
}
```

Join

```
SELECT
    employees.emp_no,
    employees.first_name,
    employees.last_name,
    salaries.salary
FROM
    employees
    INNER JOIN
    salaries
    ON
        employees.emp_no = salaries.emp_no
```

```
type Result struct {
    EmpNo     int
    FirstName string
    LastName  string
    Salary    int
}

var results = []*Result{}
rows := db.Table("employees as e").Select("e.emp_no, first_name, last_name, salary").
Joins("join salaries on e.emp_no = salaries.emp_no").Find(&results)
fmt.Println(rows)
fmt.Println(rows.Error)
fmt.Println(rows.RowsAffected)
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
```

```
for i, r := range results {
    fmt.Println(i, r)
}
```

```
type Result struct {
    EmpNo     int
    FirstName string
    LastName  string
    Salary    int
}

rows, err := db.Table("employees as e").Select("e.emp_no, first_name,
last_name, salary").
Joins("join salaries as s on e.emp_no = s.emp_no").Rows()
fmt.Println(err)
var r Result
for rows.Next() {
    rows.Scan(&r.EmpNo, &r.FirstName, &r.LastName, &r.Salary)
    fmt.Println(r, "###")
}
```

```
type Result struct {
    EmpNo     int
    FirstName string
    LastName  string
    Salary    int
}

var results = []*Result{}
db.Table("employees as e").Select("e.emp_no, first_name, last_name,
salary").
Joins("join salaries as s on e.emp_no = s.emp_no").Scan(&results)
for i, r := range results {
    fmt.Println(i, r)
}
```


更新

https://gorm.io/zh_CN/docs/update.html

先查后改：先查到一个实例，对这个实例属性进行修改，然后调用db.Save()方法保存。

db.Save()方法会保存所有字段，对于没有主键的实例相当于Insert into，有主键的实例相当于Update。

```go
// 先查
var student Student
db.First(&student)
fmt.Println(student)

student.Age += 10
student.Name = "Sam"
// 后修改
db.Save(&student)
fmt.Println(student)
```

Update单个字段

```go
db.Model(&Student{ID: 13}).Update("age", 11) // 更新符合条件的所有记录的一个字段
// UPDATE `students` SET `age`=11 WHERE `id` = 13

r := db.Model(&Student{}).Update("age", 11) // 没有指定ID或Where条件，是全表更新age字段，这是非常危险的
fmt.Println(r.Error) // 会报WHERE conditions required错误，更新失败，这是一种保护
```

Updates更新多列

多个键值对，使用map或结构体实例传参。

同样，没有指定ID或Where条件，是全表更新age字段，这是非常危险的，报WHERE conditions required错误

```go
r := db.Model(&Student{}).Where("age < ?", 20).Updates(map[string]interface{}{"name": "John", "age": 23})
fmt.Println(r.Error)
```

```go
r := db.Model(&Student{}).Where("age < ?", 24).Updates(Student{Name: "John", Age:18})
fmt.Println(r.Error)
```

删除

https://gorm.io/zh_CN/docs/delete.html

删除操作是危险的，慎重操作！

```go
result := db.Delete(&Student{})
fmt.Println(result.Error)
// 报WHERE conditions required错误，这是全表删除，危险
```

```go
result := db.Delete(&Student{}, 15) // 指定主键
fmt.Println(result.Error)

db.Delete(&Student{}, []int{15, 16, 18}) // DELETE FROM `students` WHERE `students`.`id` IN (15,16,18)
```

```go
result := db.Where("id > ?", 15).Delete(&Student{}) // 删除符合条件的一批
fmt.Println(result.Error)
```