存储 - GeeORM

#golang #学习 #ORM #reflect

ORM（Object Rational Mapping，对象关系映射）

> Go 语言中使用比较广泛 ORM 框架是 gorm 和 xorm。

- 将面向对象语言程序中的对象自动持久化到关系数据库中；
- **对象和数据库之间的桥梁**，避免写繁琐的 SQL 语言；
- 通过操作具体的对象，完成对**关系型数据库**的操作；

数据库	面向对象的编程语言
表（Table）	类（Class / Struct）
记录（Record / Row）	对象（Object）
字段（Field / Column）	对象属性（Attribute）

- SQL 语句

```
CREATE TABLE `User` (`Name` text, `Age`, integer);
INSERT INTO `User` (`Name`, `Age`) VALUES ("Tom", 18);
SELECT * FROM `User`
```

- 对应的 ORM 框架语句；

```
type User struct {
        Name string
        Age int
}

// 从参数 &User{} 中得到对应结构体的名称 User 作为表名，属性名作为列名，属性类型为列类型；
orm.CreateTable(&User{})
// 知道每个属性的值；
orm.Save(&User{"Tom", 18})
var users []User
// 从传入的空切片 &[]User 得到表名 User，并从数据库中取得所有的记录转换成 User 对象；
orm.Find(&users)
```

ORM 的通用性的实现 —— 反射机制

> ❓ 如何根据任意类型的指针，得到对应结构体的名称?
> ❗ GoLang 的反射机制（Reflect）

反射（Reflect）

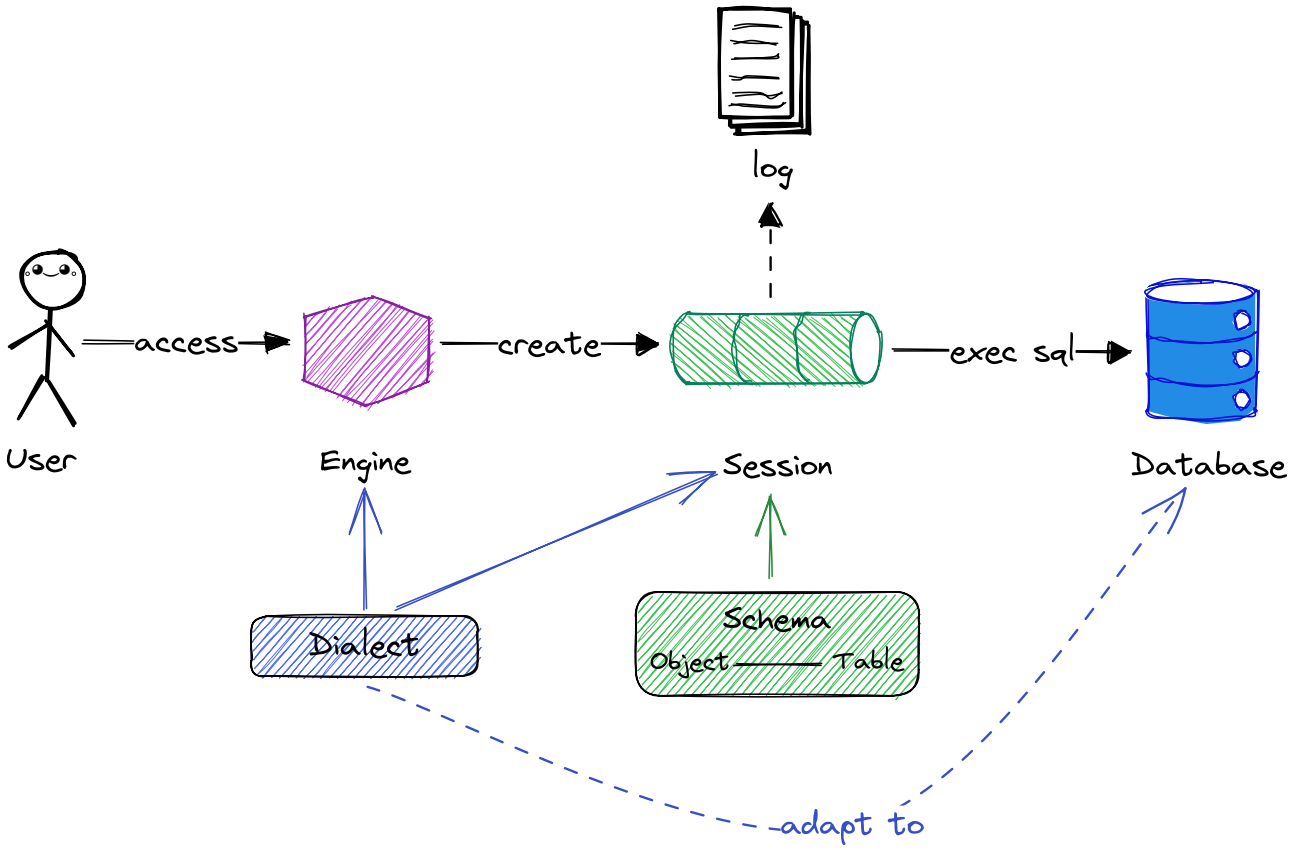
- `ValueOf(i interface{}) Value {…}`
  - 获取输入参数接口中的数据的值，如果接口为空则返回 `0`；
- `TypeOf(i interface{}) Value {…}`

    - 动态获取输入参数接口中的值的类型，如果接口为空则返回 `nil`；
- `Indirect(v Value) Value`
    - 如果 `v` 不是 `nil` 指针，返回 `v` 指向的实例；
    - 如果 `v` 是 `nil` 指针，返回 `0` 值；
    - 如果不是指针，返回 `v` 本身；
- `New(typ Type) Value`
    - 返回一个值，该值表示指向指定类型的新零值的指针；
- `(reflect.Type).Name()`
    - 返回类名（字符串）；
- `(reflect.Type).Field(i)`
    - 获取第 `i` 个成员变量；

SQLite

数据库访问

```
.
├── gee.db              // SQLite 数据库
├── geeorm.go           // 用户交互
├── log                 // 日志
│   └── log.go
└── session             // 数据库交互
    └── raw.go
```



对象表结构映射

Dialect

- ORM 需要兼容多种数据库，使用 dialect 隔离不同数据库之间的差异，便于扩展；
- 实现最大程度上的复用和解耦；

```
type Dialect interface {
        // 用于将 Go 语言的类型转换为数据库对应的数据类型
        DataTypeOf(typ reflect.Value) string
        // 返回某个表是否存在的SQL语句,参数为表名
        TableExistSQL(tableName string) (string, []interface{})
}
```

Schema

- 实现对象（Object）和表（Table）的转换；
- 给定一个任意的对象，转换为关系数据库中的表结构；

```
表名 Table Name → 结构体名 Struct Name
字段名 Column Name → 属性名 Attribute Name
字段类型 Column Type → 属性类型 Attribute Type
额外的约束 Constrain → 属性的 Tag
```

```
type Field struct {
        Name string // 字段名
        Type string // 字段类型
        Tag  string // 约束条件
}

type Schema struct {
        Model     interface{}        // 被映射的对象
        Name      string             // 表名
        Fields    []*Field           // 字段
        FieldName []string           // 字段名(列名)
        fieldMap  map[string]*Field // 字段名和 Field 的映射关系
}
```

Session

```
type Session struct {
        ...
        dialect  dialect.Dialect // SQL 数据库适配
        refTable *schema.Schema  // 表与对象的映射
}
```

Engine

```
type Engine struct {
        ...
        dialect dialect.Dialect
}
```

插入 Insert 与查询 Select

Clause

- 复杂的 SQL 语句 -> 多个子句；

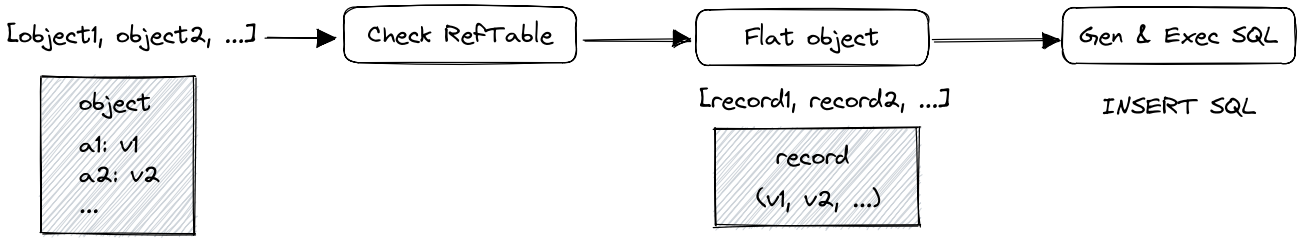
```
SELECT col1, col2, ...
FROM table_name
WHERE [ conditions ]
GROUP BY col1
HAVING [ conditions ]
```

```
type Session struct {
        ...
        clause   clause.Clause   // 用于构造 SQL 语句
}
```

Insert

```
INSERT INTO table_name (col1, col2, col3, ...) VALUES
        (A1, A2, A3, ...),
        (B1, B2, B3, ...),
```

- Schema 将 Object 展开
  - pre：`User{Name: "Tom", Age: 18}`
  - after：`("Tom", 18)`

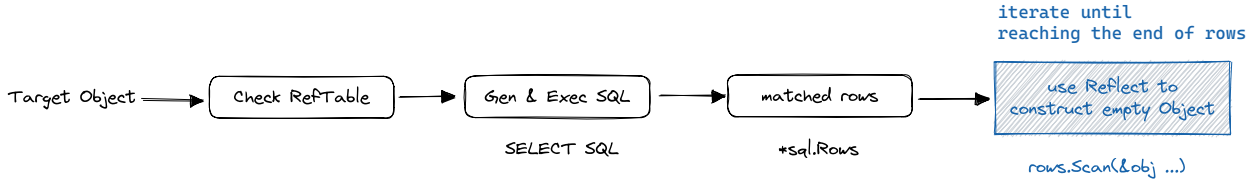



```
// schema.go
func (s *Schema) RecordValues(dest interface{}) []interface{} {
        destValue := reflect.Indirect(reflect.ValueOf(dest))
        var fieldValues []interface{}
        for _, field := range s.Fields {
                fieldValues = append(fieldValues,
destValue.FieldByName(field.Name).Interface())
        }
        return fieldValues
}

// record.go
func (s *Session) Insert(values ...interface{}) (int64, error) {
        // values: 要插入的 object
        recordValues := make([]interface{}, 0)
        for _, value := range values {
                table := s.Model(value).RefTable()
                // 其实只需要设置一次，这里是为了方便
                s.clause.Set(clause.INSERT, table.Name, table.FieldNames)
                // object -> 展开成 record
                recordValues = append(recordValues, table.RecordValues(value))
        }
```

```
        s.clause.Set(clause.VALUES, recordValues... )
        sql, vars := s.clause.Build(clause.INSERT, clause.VALUES)

        result, err := s.Raw(sql, vars... ).Exec()
        if err ≠ nil {
                return 0, err
        }
        return result.RowsAffected()
}
```

Find

- 需要根据平铺展开的字段值构造出对象 Object




```
func (s *Session) Find(values interface{}) error {
        destSlice := reflect.Indirect(reflect.ValueOf(values))

        // 获取 object 的类型
        destType := destSlice.Type().Elem()
        // 找到 object 对应的表
        table := s.Model(reflect.New(destType).Elem().Interface()).RefTable()

        // 根据表结构构造对应的 SELECT 语句，查找所有符合条件的 record → rows
        s.clause.Set(clause.SELECT, table.Name, table.FieldNames)
        sql, vars := s.clause.Build(clause.SELECT, clause.WHERE, clause.ORDERBY,
clause.LIMIT)
        rows, err := s.Raw(sql, vars... ).QueryRows()
        if err ≠ nil {
                return err
        }

        // 遍历每一行，利用 reflect 创建 object 实例 → dest
        for rows.Next() {
                dest := reflect.New(destType).Elem()
                var values []interface{}

                // 将 dest 的所有字段平铺，构造切片 values
                for _, name := range table.FieldNames {
                        values = append(values,
dest.FieldByName(name).Addr().Interface())
                }

                // 将每一列的值依次赋值给 values 的每一个字段
                if err := rows.Scan(values... ); err ≠ nil {
                        return err
                }
                destSlice.Set(reflect.Append(destSlice, dest))
        }
```

```
        return rows.Close()
}
```

链式操作与更新删除

- 实现 Update, Delete 和 Count 子句：`record.go`

Chain 链式调用

- 目的：简化代码的编程方式；
- 流程：某个对象调用某个方法后，将该对象的指针返回，之后可以继续调用该对象的其他方法；
- 场景：某个对象需要一次调用多个方法来设置其属性；
- 举例：`s.Where("Age > 18").Limit(3).Find(&users)`
- 关键：返回 `Session`

```
func (s *Session) Limit(num int) *Session {
        s.clause.Set(clause.LIMIT, num)
        return s
}

func (s *Session) Where(desc string, args ...interface{}) *Session {
        var vars []interface{}
        vars = append(vars, desc)
        vars = append(vars, args)
        s.clause.Set(clause.WHERE, vars...)
        return s
}

func (s *Session) OrderBy(desc string) *Session {
        s.clause.Set(clause.ORDERBY, desc)
        return s
}
```

Hook 钩子

- 提前在可能增加功能的地方预设一个 Hook，当我们需要修改/增加这个地方的逻辑时，把扩展的类/方法挂载到这个点即可；
- 钩子机制设计的好坏，取决于扩展点选择的是否合适；
- ORM → 记录的增删查改前后都是非常合适；
- 脱敏 desensitization ，比较敏感的信息，比如密码，在提交到服务器后，应该变为 "**"，从而保护隐私。此外，此类功能函数并未直接在包内定义，而是在调用时候定义，像 " 钩子 " 钓鱼一样，这里的鱼就是用户定义函数。

Transaction 事物

- 1）原子性 (Atomicity)：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么全部不执行。
- 2）一致性 (Consistency): 几个并行执行的事务，其执行结果必须与按某一顺序 串行执行的结果相一致。
- 3）隔离性 (Isolation)：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。

- 4）持久性 (Durability)：对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

> 数据库事务 (transaction) 是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行,要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

```
type Session struct {
        ...
        tx *sql.Tx  // 用于支持 SQL 事务
}

func (s *Session) DB() CommonDB {
        if s.tx != nil {
                return s.tx
        }
        return s.db
}
```

```
type TxFunc func(*session.Session) (interface{}, error)

func (e *Engine) Transaction(f TxFunc) (result interface{}, err error) {
        s := e.NewSession()
        if err := s.Begin(); err != nil {
                return nil, err
        }
        defer func() {
                if p := recover(); p != nil {
                        _ = s.Rollback()
                        panic(p) // 回滚不覆盖 err，是因为回滚就是因为有业务的报错，所以不应该被这条语句覆盖掉业务的 err，业务 err 比回滚失败的 err 更重要
                } else if err != nil {
                        _ = s.Rollback() // error 不为空，不需要改变
                } else {
                        defer func() {
                                if err != nil {
                                        _ = s.Rollback()
                                }
                        }()
                        err = s.Commit() // error 为空,更新 error
                }
        }()

        return f(s)
}
```

迁移 Migrate

- 结构体 (struct) 变更时，数据库表的字段 (field) 自动迁移 (migrate)。
- 支持字段的新增与删除，不支持字段类型变更；

1. 新增字段

```
ALTER TABLE table_name ADD COLUMN col_name, col_type;
```

1. 删除字段

```
CREATE TABLE new_table AS SELECT col1, col2, ... from old_table
DROP TABLE old_table
ALTER TABLE new_table RENAME TO old_table;
```

- 先在原表中增加字段，再把不要的 Column 删除；