GORM介绍与展望

GORM - THE FANTASTIC ORM LIBRARY FOR GOLANG

ABOUT ME

Jinzhu / 张金柱

- Github：github.com/jinzhu/gorm
- GORM 官网: https://gorm.io

GORM 基本介绍

- 链式API
- 关联处理（一对一，一对多，多对多，多态）
- 钩子（创建，更新，删除，查找）
- Preload/事务/复合主键/SQL Builder...
- 可扩展性
- 简单易用，开发者友好

GORM 基本介绍

```go
// 连接数据库
db, err := gorm.Open("sqlite3", "test.db") // gorm v1

// 创建数据库表
db.AutoMigrate(&Product{})

// 创建
db.Create(&Product{Code: "L1212", Price: 1000})
db.Save(&product) // 如果主键有值更新，否则创建

// 读取
var product Product
db.First(&product, 1)                   // 查询id为1的product
db.First(&product, "code = ?", "L1212") // 查询code为l1212的product

// 更新product的price为2000
db.Model(&product).Update("Price", 2000)

db.Updates(&product) // 只更新非零值字段
db.Select("*").Updates(&product) // 全部字段，v2 支持
db.Select("Code", "Price").Updates(&product) // 只更新 Code, Price
db.Omit("Code").Updates(&product) // 除 Code 外的非零值字段
db.Select("*").Omit("Code").Updates(&product) // 除 Code 外的全部字段
db.Select("Price").Updates(map[string]interface{"Price": 200, "Name": "T-Shirt"})

// 删除 - 删除product
db.Delete(&product)
```

GORM 关联 - 定义

```go
// User 拥有一个 `Account` (has one), 拥有多个 `Pets` (has many), 多个 `Toys` (has many 多态)
// 属于某 Company (belongs to), 属于某 Manager (belongs to 单表), 管理 Team (has many 单表)
// 会多种 languages (many to many) 拥有很多 friends (many to many 单表)
// 他的 `Pet` 也有一个玩具 Toy (has one 多态)

type User struct {
  gorm.Model
  Name       string
  Account    Account
  Pets       []*Pet
  Toys       []Toy `gorm:"polymorphic:Owner"`
  CompanyID *int
  Company    Company
  ManagerID *uint
  Manager    *User
  Team       []User     `gorm:"foreignkey:ManagerID"`
  Languages []Language `gorm:"many2many:UserSpeak"`
  Friends    []*User    `gorm:"many2many:user_friends"`
}

type Pet struct {
  gorm.Model
  UserID int
  Toy    Toy `gorm:"polymorphic:Owner;"`
}

type Toy struct {
  gorm.Model
  OwnerID   string
  OwnerType string
}
```

GORM 关联 - CRUD

```go
// 保存用户及其关联
db.Save(&User{
  Name:      "jinzhu",
  // Language 主键为空，创建并建立关联，主键有值，只建立关联，可设置 Name 为主键
  Languages: []Language{{Name: "zh-CN"}, {Name: "en-US"}},
})

langAssociation := db.Model(&user).Association("Languages")

// 查询用户掌握的语言
langAssociation.Find(&languages)

// 将汉语，德语添加到用户掌握的语言中
langAssociation.Append([]Language{languageZH, languageEN})

// 把用户掌握的语言替换为汉语，德语
langAssociation.Replace([]Language{languageZH, languageDE})

// 删除用户掌握的两个语言
langAssociation.Delete(languageZH, languageEN)

// 删除用户所有掌握的语言
langAssociation.Clear()

// 返回用户所掌握的语言的数量
langAssociation.Count()
```

GORM 关联 - Preload / Joins 预加载

```go
type User struct {
  gorm.Model
  Profile Profile
  Orders  []Order
}

type Order struct {
  gorm.Model
  State string
}

// 查询用户的时候并找出其订单
db.Preload("Orders").Find(&users)

// 查询用户的时候找出其未取消的订单
db.Preload("Orders", "state NOT IN (?)", "cancelled").Find(&users)

// 查询用户的时候找出其订单和其信用卡
db.Preload("Orders.CreditCard").Find(&users)

// 查询用户的时候找出其订单和其信用卡
db.Joins("Profile").Find(&users)
```

GORM 钩子

```go
// Before/After Save 保存前/后 (在更新及创建的时候调用)
func (u *User) BeforeSave(tx *gorm.DB) (err error) {
  // xxx
}

// Before/After Create 创建前/后调用
func (u *User) AfterCreate(tx *gorm.DB) (err error) {
  // xxx
}

// Before/After Update 更新数据前/后调用
func (u *User) BeforeUpdate(tx *gorm.DB) (err error) {
  // xxx
}

// Before/After Delete 更新数据前后调用
func (u *User) AfterDelete(tx *gorm.DB) (err error) {
  // xxx
  tx.New().Model(&Address{}).Where("user_id = ?", u.ID).Update("invalid", false)
}

// AfterFind 查询数据后调用
func (u *User) AfterFind(tx *gorm.DB) (err error) {
  // xxx
}
```

GORM 与 SQL/DATABASE

```go
// Raw SQL
db.Raw("SELECT name, age FROM users WHERE name = ?", 3).Scan(&result)

// sql Row
row := db.Table("users").Where("name = ?", "jinzhu").Select("name, age").Row()
row.Scan(&name, &age)

// Rows 迭代
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Rows()
defer rows.Close()
for rows.Next() {
  // ...
  rows.Scan(&name, &age, &email)

  // ...
  var user User
  db.ScanRows(rows, &user)
}

// SetMaxIdleConns sets the maximum number of connections in the idle connection pool.
db.DB().SetMaxIdleConns(10)

// SetMaxOpenConns sets the maximum number of open connections to the database.
db.DB().SetMaxOpenConns(100)

// SetConnMaxLifetime sets the maximum amount of time a connection may be reused.
db.DB().SetConnMaxLifetime(time.Hour)
```

GORM 与 事务

```go
// begin a transaction
tx := db.Begin()

// do some database operations in the transaction (use 'tx', not 'db')
tx.Create(...)

// rollback the transaction in case of error
tx.Rollback()

// Or commit the transaction
tx.Commit()
```

```go
db.Transaction(func(tx *gorm.DB) error {
  if err := tx.Create(&Animal{Name: "Giraffe"}).Error; err != nil {
    // return any error will rollback
    return err
  }

  if err := tx.Create(&Animal{Name: "Lion"}).Error; err != nil {
    return err
  }

  // return nil will commit
  return nil
})
```

GORM 与 代码复用

```go
func Paginate(r *http.Request) func(*gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    page, _ := strconv.Atoi(r.URL.Query().Get("page"))
    if page == 0 {
      page = 1
    }

    perPage, _ := strconv.Atoi(r.URL.Query().Get("size"))
    if perPage > 20 {
      perPage = 20
    }

    offset := (page - 1) * perPage

    return db.Offset(offset).Limit(perPage)
  }
}

// 多个对象复用同一个代码块
db.Scopes(Paginate(r)).Find(&users)
db.Scopes(Paginate(r)).Find(&articles)
db.Scopes(Paginate(r)).Find(&products)
```

GORM 与 代码复用

```go
func AmountGreaterThan1000(db *gorm.DB) *gorm.DB {
  return db.Where("amount > ?", 1000)
}

func PaidWithCreditCard(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode = ?", "card")
}

func PaidWithCod(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode = ?", "cod")
}
```

```go
func OrderStatus(status []string) func (db *gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    return db.Where("status IN (?)", status)
  }
}

// 一个对象使用多个代码块
db.Scopes(AmountGreaterThan1000, PaidWithCreditCard).Find(&orders)
// Find all credit card orders and amount greater than 1000

db.Scopes(AmountGreaterThan1000, PaidWithCod).Find(&orders)
// Find all COD orders and amount greater than 1000

db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid", "shipped"})).Find(&orders)
// Find all paid, shipped orders that amount greater than 1000
```

GORM 插件系统

- 扩展API
- 所有的功能都基于插件系统
- 可完全定制GORM的功能

GORM 插件系统

```
// create callbacks
db.callbacks.creates

// update callbacks
db.callbacks.updates

// delete callbacks
db.callbacks.deletes

// query callbacks
db.callbacks.queries

// row / rows callbacks
db.callbacks.row

// raw raw sql callbacks
db.callbacks.raw
```

GORM 插件系统 - CREATE

```go
// 创建对象
db.Create(&Product{Code: "L1212", Price: 1000})
```



```go
// GORM 找出 Create 所注册的所有方法，并一一调用
func Create(data interface{}) error {
  // ... 伪代码
  for _, f := range db.callbacks.creates {
    f()
  }
}
```

GORM 插件系统 - CREATE 过程

```
// GORM callbacks for creating
DefaultCallback.Create().Register("gorm:begin_transaction",
                                  beginTransactionCallback)
DefaultCallback.Create().Register("gorm:before_create",
                                  beforeCreateCallback)
DefaultCallback.Create().Register("gorm:save_before_associations",
                                  saveBeforeAssociationsCallback)
// 将数据插入数据库
DefaultCallback.Create().Register("gorm:create",
                                  createCallback)

DefaultCallback.Create().Register("gorm:save_after_associations",
                                  saveAfterAssociationsCallback)
DefaultCallback.Create().Register("gorm:after_create",
                                  afterCreateCallback)
DefaultCallback.Create().Register("gorm:commit_or_rollback_transaction",
                                  commitOrRollbackTransactionCallback)
```

GORM 插件系统 - 定制 CREATE 过程

```
// 删除Callback
DB.Callback().Create().Remove("gorm:begin_transaction")
DB.Callback().Create().Remove("gorm:commit_or_rollback_transaction")

// 替换Callback
DB.Callback().Create().Replace("gorm:update_time_stamp", func(*Scope) {
  scope.SetColumn("Created", now)
  scope.SetColumn("Updated", now)
})

// 注册新Callback
DB.Callback().Create().Register("my_plugin:new_callback", func(*Scope) {
  // xxx
})

// 指定Callback顺序
DB.Callback().Create().
  Before("gorm:before_create").Register("my_plugin:new_callback", func(*Scope) {
  // xxx
})

// 指定Callback顺序
DB.Callback().Create().
  Before("gorm:before_create").After("my_plugin:new_callback").
  Register("my_plugin:new_callback2", func(*Scope) {
    // xxx
  })
```

GORM 插件系统 - 图片处理


```go
type User struct {
  gorm.Model
  Name   string
  Avatar aliyun.OSS
}

avatar, err := os.Open("me.png")
user.Avatar.Scan(avatar)

// 保存用户的同时，将用户的图片上传至阿里云的OSS
db.Save(&user)

// 返回上传到阿里云图片的URL
user.Avatar.URL()

// 上传到其它的云
type User struct {
  // 上传至七牛云
  Avatar qiniu.Qiniu

  // 上传至 AWS S3
  Avatar aws.S3
}
```

GORM 插件系统 - 图片处理


```go
// 注册处理图片的方法创建Callback中
DB.Callback().Update().Before("gorm:before_update").
  Register("media_library:save_and_crop", saveAndCropImage)

// 注册处理图片的方法更新Callback中
DB.Callback().Create().After("gorm:after_create").
  Register("media_library:save_and_crop", saveAndCropImage)

func saveAndCropImage(tx *gorm.DB) {
  // 找出修改的对象所有的字段
  for _, field := range tx.Statement.Schema.Fields {
    // 判断如果该字段的类型是我们能处理的类型
    // 如果是的话，对数据进行处理并上传到云
  }
}
```

GORM 插件系统 - 数据校验


```go
type User struct {
 gorm.Model
 Name           string `valid:"required"`
 Password       string `valid:"length(6|20)"`
 SecurePassword string `valid:"numeric"`
 Email          string `valid:"email"`
 IP             string `valid:"ip"`
}
```

```go
// 注册校验数据的方法到创建Callback中
DB.Callback().Create().Before("gorm:before_create").
  Register("validations:validate", validate)
// 注册校验数据的方法到更新Callback中
DB.Callback().Update().Before("gorm:before_update").
  Register("validations:validate", validate)

func validate(tx *gorm.DB) {
 if _, err := govalidator.ValidateStruct(tx.Statement.Dest); err != nil {
   tx.AddError(err)
 }
}
```

GORM 2.0

- 性能优化
- Context 支持
- Batch Insert, Preload with Join, Auto Migrate 外键处理
- 部分API调整，插件API调整
- 更健壮，让用户少踩坑
- 提高可扩展性

GORM 2.0 — 初始化

```go
dsn := "gorm:gorm@tcp(localhost:9910)/gorm?charset=utf8&parseTime=True&loc=Local"
// import _ "github.com/jinzhu/gorm/dialects/mysql"
// gorm.Open("mysql", dsn)

import "github.com/jinzhu/gorm/dialects/mysql"
gorm.Open(mysql.Open(dsn) /* dialector interface */, &gorm.Config{
  Logger:          logger.Default, // 默认值
  NowFunc:         func() time.Time {
    return time.Now().Local()
  }, // 默认值
  NamingStrategy: schema.NamingStrategy{
    TablePrefix: "t_",
    SingularTable: true,
  },
})
```

GORM 2.0 — Dialector

```go
// Dialector GORM database dialector
type Dialector interface {
  Initialize(*DB) error
  Migrator(db *DB) Migrator
  DataTypeOf(*schema.Field) string
  BindVarTo(writer clause.Writer, stmt *Statement, v interface{})
  QuoteTo(clause.Writer, string)
  Explain(sql string, vars ...interface{}) string
}
```

GORM 2.0 — Dialector

```go
package postgres

type Dialector struct {
  DSN string
}

func Open(dsn string) gorm.Dialector {
  return &Dialector{DSN: dsn}
}

func (dialector Dialector) Initialize(db *gorm.DB) (err error) {
  // register callbacks
  callbacks.RegisterDefaultCallbacks(db, &callbacks.Config{
    WithReturning: true,
  })
  db.ConnPool, err = sql.Open("postgres", dialector.DSN)
  return
}

func (dialector Dialector) Migrator(db *gorm.DB) gorm.Migrator {
  return Migrator{migrator.Migrator{Config: migrator.Config{
    DB:                          db,
    Dialector:                   dialector,
    CreateIndexAfterCreateTable: true,
  }}}
}

func (dialector Dialector) BindVarTo(writer clause.Writer, stmt *gorm.Statement, v interface{}) {
  writer.WriteByte('$')
  writer.WriteString(strconv.Itoa(len(stmt.Vars)))
}

func (dialector Dialector) QuoteTo(writer clause.Writer, str string) {
  writer.WriteByte('"')
  writer.WriteString(str)
```

```go
package callbacks

type Config struct {
  LastInsertIDReversed bool
  WithReturning        bool
}

func RegisterDefaultCallbacks(db *gorm.DB, config *Config) {
  enableTransaction := func(db *gorm.DB) bool {
    return !db.SkipDefaultTransaction
  }

  createCallback := db.Callback().Create()
  createCallback.Match(enableTransaction).Register("gorm:begin_transaction", BeginTransaction)
  createCallback.Register("gorm:before_create", BeforeCreate)
  createCallback.Register("gorm:save_before_associations", SaveBeforeAssociations)
  createCallback.Register("gorm:create", Create(config))
  createCallback.Register("gorm:save_after_associations", SaveAfterAssociations)
  createCallback.Register("gorm:after_create", AfterCreate)
  createCallback.Match(enableTransaction).Register("gorm:commit_or_rollback_transaction", Comm

  queryCallback := db.Callback().Query()
  queryCallback.Register("gorm:query", Query)
  queryCallback.Register("gorm:preload", Preload)
  queryCallback.Register("gorm:after_query", AfterQuery)

  deleteCallback := db.Callback().Delete()
  deleteCallback.Match(enableTransaction).Register("gorm:begin_transaction", BeginTransaction)
  deleteCallback.Register("gorm:before_delete", BeforeDelete)
  deleteCallback.Register("gorm:delete", Delete)
  deleteCallback.Register("gorm:after_delete", AfterDelete)
  deleteCallback.Match(enableTransaction).Register("gorm:commit_or_rollback_transaction", Comm

  updateCallback := db.Callback().Update()
  updateCallback.Match(enableTransaction).Register("gorm:begin_transaction", BeginTransaction)
  updateCallback.Register("gorm:before_update", BeforeUpdate)
  updateCallback.Register("gorm:save_before_associations", SaveBeforeAssociations)
  updateCallback.Register("gorm:update", Update)
  updateCallback.Register("gorm:save_after_associations", SaveAfterAssociations)
  updateCallback.Register("gorm:after_update", AfterUpdate)
  updateCallback.Match(enableTransaction).Register("gorm:commit_or_rollback_transaction", Comm

  db.Callback().Row().Register("gorm:raw", RowQuery)
  db.Callback().Raw().Register("gorm:raw", RawExec)
```

GORM 2.0 — Dialector

```go
// sqlite
func (dialector Dialector) Initialize(db *gorm.DB) (err error) {
  callbacks.RegisterDefaultCallbacks(db, &callbacks.Config{
    LastInsertIDReversed: true,
  })
  db.ConnPool, err = sql.Open("sqlite3", dialector.DSN)
  return
}

// redis
type Dialector struct {
  Client *redis.Client
}

func Open(config *redis.Config) gorm.Dialector {
  return &Dialector{Client: redis.New(config)}
}

func (dialector Dialector) Initialize(db *gorm.DB) (err error) {
  db.Callback().Create().Register("gorm:create", dialector.Create)
  // ...
}

func (dialector Dialector) Create(db *gorm.DB) {
  primaryField := db.Statement.Schema.PrioritizedPrimaryField
  primaryValue := primaryField.ValueOf(db.Statement.ReflectValue)
  objectKey := path.Join(db.Statement.Table, fmt.Sprint(primaryValue))
  dialector.Client.HMSet(objectKey, convertObjectToMap(db.Statement.Dest))
}
```

GORM 2.0 — SQL Builder

```go
stmt := gorm.Statement{DB: db, Schema: userSchema}

stmt.AddClause(clause.Select{})
stmt.AddClause(clause.From{})
stmt.AddClause(clause.Where{
  Exprs: clause.Add(clause.Eq{"name": "jinzhu"}, clause.Gt{"age": 18}),
})
stmt.AddClause(clause.Limit{Limit: 10, Offset: 10})
stmt.Build("SELECT", "FROM", "WHERE", "LIMIT")

// SELECT * FROM `users` WHERE `name`=? AND `age`>? LIMIT 10 OFFSET 10;
stmt.SQL.String()
// []interface{}{"jinzhu", 18}
stmt.Vars
```

GORM 2.0 — SQL Builder

```
// MicroSoft SQL Server
// SELECT * FROM `users` WHERE `name`=? AND `age`>? OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY
stmt.SQL.String()
// []interface{}{"jinzhu", 18}
stmt.Vars
```

```
// Dialector
func (dialector Dialector) Initialize(db *gorm.DB) (err error) {
  db.ClauseBuilders["LIMIT"] = func(c clause.Clause, b clause.Builder) {
    limit, _ := c.Expression.(Limit)
    b.WriteString(
      fmt.Sprintf("OFFSET %d ROWS FETCH NEXT %d ROWS ONLY", limit.Offset, limit.Limit),
    )
  }
}
```

GORM 2.0 — SQL Builder

```go
type Sharding struct {
  Node string
}

func (sharding Sharding) ModifyStatement(stmt *gorm.Statement) {
  if sharding.Node != "" {
    clause := stmt.Clauses["SELECT"]
    clause.BeforeExpressions = append(clause.BeforeExpressions, clause.Expr{
      SQL: "/*"+sharding.Node+"*/",
    })
    stmt.Clauses["SELECT"] = clause
  }
}

db.Clauses(Sharding{Node: "node2"}).First(&user)
// => /* node2 */ SELECT * FROM `users` ORDER BY `id` LIMIT 1;

// clause.AfterNameExpressions
// => SELECT /* master */ * FROM `users` ORDER BY `id` LIMIT 1;

// clause.AfterExpressions
// => SELECT * /* master */ FROM `users` ORDER BY `id` LIMIT 1;
```

GORM 2.0 — SQL Builder

```go
package dba

var sqls sync.Map

func Approve(name, sql string) {
  sqls.Store(name, sql)
}

func Get(name string, vars ...interface{}) clause.Interface {
  sql, _ := sqls.Load(name)
  return Clause{SQL: sql.(string), Vars: vars}
 }

 func (clause Clause) ModifyStatement(stmt *gorm.Statement) {
   stmt.SQL.WriteString(clause.SQL)
   stmt.SQL.Vars = clause.Vars
 }

 // https://www.quora.com/Whats-the-most-complex-SQL-query-you-ever-wrote
 dba.Approve("complexsql1", `SELECT DATEADD(day,-1,pxd_upd_date), pbd_prod_code, psd_
FNULL(CONCAT(CONCAT(TRIM(TRAILING ' ' FROM P.pbd_prod_name), CASE LENGTH(CONVERT(VARC
 ....................
WHERE psd_stk_qty+psd_booked_qty+psd_po_qty>0 ORDER BY pbd_maj_grp ASC, pbd_min_grp A

 db.Clauses(dba.Get("complexsql1", arg1, arg2, arg3)).Scan(&users)
```

GORM 2.0 — Migrator

```go
type Migrator interface {
  // AutoMigrate
  AutoMigrate(dst ...interface{}) error

  // Database
  CurrentDatabase() string

  // Tables
  CreateTable(dst ...interface{}) error
  DropTable(dst ...interface{}) error
  HasTable(dst interface{}) bool
  RenameTable(oldName, newName string) error

  // Columns
  AddColumn(dst interface{}, field string) error
  DropColumn(dst interface{}, field string) error
  AlterColumn(dst interface{}, field string) error
  HasColumn(dst interface{}, field string) bool
  RenameColumn(dst interface{}, oldName, field string) error
  ColumnTypes(dst interface{}) ([]*sql.ColumnType, error)

  // Views
  CreateView(name string, option ViewOption) error
  DropView(name string) error

  // Constraints
  CreateConstraint(dst interface{}, name string) error
  DropConstraint(dst interface{}, name string) error
  HasConstraint(dst interface{}, name string) bool

  // Indexes
  CreateIndex(dst interface{}, name string) error
  DropIndex(dst interface{}, name string) error
  HasIndex(dst interface{}, name string) bool
  RenameIndex(dst interface{}, oldName, newName string) error
}
```

GORM 2.0 — Migrator

```go
// sqlite
func (dialector Dialector) Migrator(db *gorm.DB) gorm.Migrator {
  return Migrator{
    migrator.Migrator{Config: migrator.Config{
      DB:                          db,
      Dialector:                   dialector,
      CreateIndexAfterCreateTable: true,
    }},
  }
}
```

```go
type Migrator struct {
  migrator.Migrator
}

func (m Migrator) HasTable(value interface{}) bool {
  var count int
  m.Migrator.RunWithValue(value, func(stmt *gorm.Statement) error {
    return m.DB.Raw(
      "SELECT count(*) FROM sqlite_master WHERE type='table' AND name=?", stmt.Table,
    ).Row().Scan(&count)
  })
  return count > 0
}
```

GORM 2.0 — Migrator

```go
// https://github.com/go-gormigrate/gormigrate
gormigrate.New(db, gormigrate.DefaultOptions, []*gormigrate.Migration{
  {
    ID: "202003181400",
    Migrate: func(tx *gorm.DB) error {
      type Person struct {
        gorm.Model
        Name string
      }
      return tx.AutoMigrate(&Person{}).Error
    },
    Rollback: func(tx *gorm.DB) error {
      return tx.DropTable(&Person{}).Error
    },
  },
  {
    ID: "202003191400",
    Migrate: func(tx *gorm.DB) error {
      type Person struct {
        Age int `gorm:"size:3"`
      }
      return tx.Migrator().AlterColumn(&Person{}, "Age").Error
    },
    Rollback: func(tx *gorm.DB) error {
      type Person struct {
        Age int
      }
      return tx.Migrator().AlterColumn(&Person{}, "Age").Error
    },
  },
})
```

GORM 2.0 — Schema

```go
type Schema struct {
  Name                   string
  ModelType              reflect.Type
  Table                  string
  PrioritizedPrimaryField *Field
  PrimaryFields          []*Field
  Fields                 []*Field
  Relationships          Relationships
  //...
}

type Field struct {
  Name                  string
  DBName                string
  ReflectValueOf        func(reflect.Value) reflect.Value
  ValueOf               func(reflect.Value) (value interface{}, zero bool)
  Set                   func(reflect.Value, interface{}) error
  //...
}

type Relationships struct {
  HasOne    []*Relationship
  BelongsTo []*Relationship
  HasMany   []*Relationship
  Many2Many []*Relationship
  Relations map[string]*Relationship
}

type Relationship struct {
  Name                    string
  Type                    RelationshipType
  References              []Reference
  // ...
}
```

GORM 2.0 — Schema

```
switch {
case len(field.StructField.Index) == 1:
  if field.FieldType.Kind() == reflect.Ptr {
    field.ReflectValueOf = func(value reflect.Value) reflect.Value {
      fieldValue := reflect.Indirect(value).Field(field.StructField.Index[0])
      if fieldValue.IsNil() {
        fieldValue.Set(
      }
      return fieldValue
    }
  } else {
    field.ReflectValueO
      return reflect.In
```

```
switch {
case len(field.StructField.Index) == 1:
  field.ValueOf = func(value reflect.Value) (interface{}, bool) {
    fieldValue := reflect.Indirect(value).Field(field.StructField.Index[0])
    return fieldValue.Interface(), fieldValue.IsZero()
  }
case len(field.StructField.Index) == 2 && field.StructField.Index[0] >= 0:
  field.ValueOf = func(value reflect.Value) (interface{}, bool) {
    fieldValue := reflect.Indirect(value).Field(field.StructField.Index[0]).
    return fieldValue.Interface(), fieldValue.IsZero()
  }
default:
  field.ValueOf = func(value reflect.Value) (interface{}, bool) {
    v := reflect.Indirect(value)

    for _, idx := range field.StructField.Index {
      if idx >= 0 {
```

```
switch field.FieldType.Kind()
case reflect.Bool:
  field.Set = func(value refl
    switch data := v.(type) {
    case bool:
      field.ReflectValueOf(va
    case *bool:
      field.ReflectValueOf(va
    default:
      return recove
    }
    return nil
  }
case reflect.Int, r
  field.Set = func(
    switch data :=
    case int64:
      field.Reflect
    case int:
      field.Reflect
```

```
func (schema *Schema) parseRelation(field *Field) {
  var (
    err        error
    fieldValue = reflect.New(field.IndirectFieldType).Interface()
    relation   = &Relationship{
      Name:        field.Name,
      Field:       field,
      Schema:      schema,
      foreignKeys: toColumns(field.TagSettings["FOREIGNKEY"]),
      primaryKeys: toColumns(field.TagSettings["REFERENCES"]),
```

GORM 2.0 — 执行过程

```
// => db.First(&user)

func (db *DB) First(out interface{}, conds ...interface{}) (tx *DB) {
  tx = db.getInstance().Limit(1).Order(clause.OrderByColumn{
    Column: clause.Column{Table: clause.CurrentTable, Name: clause.PrimaryKey},
  })
  // ...
  tx.Statement.Dest = out
  tx.callbacks.Query().Execute(tx)
  return
}
```

```
func Execute() {
  curTime := time.Now()

  tx.Statement.Schema, err = schema.Parse(tx.Statement.Dest, tx.cache, tx.NamingStrategy)

  for _, f := range db.callbacks.query {
    f(db)
  }

  db.Logger.Trace(curTime, func() (string, int64) {
    return db.Dialector.Explain(stmt.SQL.String(), stmt.Vars...), db.RowsAffected
  }, db.Error)
}
```

GORM 2.0 — 执行过程

```go
db.Callback().Query().Register("gorm:query", Query)
db.Callback().Query().Register("gorm:preload", Preload)
db.Callback().Query().Register("gorm:after_query", AfterQuery)

func Query(db *gorm.DB) {
  if db.Statement.SQL.String() == "" {
    db.Statement.AddClauseIfNotExists(clause.Select{})
    db.Statement.AddClauseIfNotExists(clause.From{})
    db.Statement.Build(
      "SELECT", "FROM", "JOINS", "WHERE", "GROUP BY", "ORDER BY", "LIMIT", "FOR",
    )
  }

  rows, err := db.Statement.ConnPool.QueryContext(
    db.Statement.Context, db.Statement.SQL.String(), db.Statement.Vars...,
  )
  if err != nil {
    db.AddError(err)
    return
  }
  defer rows.Close()

  Scan(rows, db)
}
```

GORM 2.0 — 执行过程

```go
type Statement struct {
  *DB
  ConnPool ConnPool
  Context  context.Context
  Schema   *schema.Schema
  Clauses  map[string]clause.Clause
  // ...
}

type ConnPool interface {
  ExecContext(context.Context, string, ...interface{}) (sql.Result, error)
  PrepareContext(context.Context, string) (*sql.Stmt, error)
  QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
  QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

type TxBeginner interface {
  BeginTx(ctx context.Context, opts *sql.TxOptions) (*sql.Tx, error)
}

type TxCommiter interface {
  Commit() error
  Rollback() error
}
```

GORM 2.0 — 制作读写分离插件

```go
// db.Statement.ConnPool.QueryContext(stmt.Context, stmt.SQL.String(), stmt.Vars...)

type RWSplit struct {
  Read  *sql.DB
  Write *sql.DB
}

func (rwsplit RWSplit) ReadDB(tx *gorm.DB) {
  if _, ok := tx.Statement.ConnPool.(TxCommiter); !ok {
    tx.Statement.ConnPool = rwsplit.Read
  }
}

func (rwsplit RWSplit) WriteDB(tx *gorm.DB) {
  if _, ok := tx.Statement.ConnPool.(TxCommiter); !ok {
    tx.Statement.ConnPool = rwsplit.Write
  }
}

db.Query().Before("gorm:query").Register("rwsplit:setdb", rwsplit.ReadDB)
db.Create().Before("gorm:before_create").Register("rwsplit:setdb",rwsplit.WriteDB)
db.Update().Before("gorm:before_update").Register("rwsplit:setdb",rwsplit.WriteDB)
db.Delete().Before("gorm:before_delete").Register("rwsplit:setdb",rwsplit.WriteDB)
```

GORM 2.0 — 制作 Prepared Statement Cache 插件

```go
type StatementLFUCache struct {
}

func (cache StatementLFUCache) Get(sql string) (*sql.Stmt, bool) { /* ... */ }

func (cache StatementLFUCache) SetDB(tx *grom.DB) {
  tx.Statement.ConnPool = StmtPool{StatementLFUCache: cache, DB: tx}
}

db.Query().Before("gorm:query").Register("stmtpool:set", cacher.SetDB)
db.Create().Before("gorm:before_create").Register("rwsplit:setdb",cacher.SetDB)
db.Update().Before("gorm:before_update").Register("rwsplit:setdb",cacher.SetDB)
db.Delete().Before("gorm:before_delete").Register("rwsplit:setdb",cacher.SetDB)

type StmtPool struct {
  StatementLFUCache
  DB gorm.ConnPool
}

func (pool StmtPool) ExecContext(ctx context.Context, sql string, args ...interface{}) (sql.Result, error
  if stmt, ok := pool.StatementLFUCache.Get(sql); ok {
    if tx, ok := pool.DB.(TxCommiter); ok {
      return tx.StmtContext(ctx, stmt).Exec(args...)
    } else {
      return stmt.ExecContext(ctx, sql, args...)
    }
  } else {
    return pool.DB.ExecContext(ctx, sql, args...)
  }
}

func (pool StmtPool) QueryContext(ctx context.Context, sql string, args ...interface{}) (*sql.Rows, error
  // ...
}
```

Q&A&谢谢