

Gorm

xbZhong

2025-12-10

[本页PDF](#)

Gorm

是Go的**ORM框架**，支持：MySQL、PostgreSQL等，但所有方法的参数都是空接口类型，不看文档很难知道要传递什么参数

导入模块

- 以 MySQL 为例，除了导入 Gorm，还要导入数据库驱动

```
1 import (
2     "gorm.io/gorm"
3     "gorm.io/driver/mysql"
4 )
```

连接数据库

- 显式连接
 - 驱动会自动将 dsn 解析为对应的配置，即 mysql.Config
 - dsn 是数据库连接信息的字符串，里面包含：
 - 用户名
 - 密码
 - 服务器地址
 - 端口
 - 数据库名称
 - 参数：字符集、时区、是否自动把 DATETIME 转成 Go 的 time.Time

```
1 dsn := "root:123456@tcp(192.168.48.138:3306)/hello?
2 charset=utf8mb4&parseTime=True&loc=Local"
3 db, err := gorm.Open(mysql.Open(dsn))
```

- 手动传入配置，使用 mysql.Config 创建配置类

```
1 db, err := gorm.Open(mysql.New(mysql.Config{
2     DSN:                 "root:123456@tcp(127.0.0.1:3306)/hello?
3     charset:             "utf8mb4&parseTime=True&loc=Local",
4 })()
```

- gorm.Open() 方法用于提供GORM封装的**数据库操作对象**，它可以进行链式操作、事务管理、迁移等功能

- `mysql.New(mysql.Config{})` : 灵活方式，可通过 `mysql.Config{}` 设置更多参数
- `mysql.Open(dsn)` : 简单方式，直接传数据库连接字符串

模型

Gorm同样也有校验器 `validator`，只不过他的校验器规则和Gin的不太一样

指定列名

- 使用结构体标签
- 规则： `gorm:"column:column_name"`

```

1 type Person struct {
2     Id      uint    `gorm:"column:ID;"` 
3     Name    string   `gorm:"column:Name;"` 
4     Address string
5     Mom     string
6     Dad     string
7 }
```

指定表名

- 通过实现 `Table` 接口就可以指定表名

```

1 type Tabler interface {
2     TableName() string
3 }
```

- 让实体类结构体实现这个接口，返回表名（字符串格式）

```

1 type Person struct {
2     Id      uint    `gorm:"column:ID;"` 
3     Name    string   `gorm:"column:Name;"` 
4     Address string
5     Mom     string
6     Dad     string
7 }
8
9 func (p Person) TableName() string {
10     return "person"
11 }
```

时间追踪

当实体类包含 `CreatedAt` 或者 `UpdatedAt` 时，在创建或更新记录时，如果其为零值，那么gorm会自动使用 `time.Now()` 来设置时间

- 自动时间填充会根据字段的**实际类型**来决定填充的**值类型**

字段声明的类型	填充类型
<code>time.Time</code>	<code>time.Now()</code> 的 <code>time.Time</code> 值
<code>int64</code>	Unix 时间戳（秒）
<code>int</code>	Unix 时间戳（秒）

```
1 type Person struct {
2     Id      uint
3     Name    string
4     Address string
5     Mom    string
6     Dad    string
7
8     CreatedAt sql.NullTime
9     UpdatedAt sql.NullTime
10 }
11
12 func (p Person) TableName() string {
13     return "person"
14 }
```

也支持时间戳追踪

- 可以自己声明 `tag`，觉得要存储什么级别的时间戳

tag	填充内容
<code>autoCreateTime</code>	创建时填充 Unix 秒级时间戳
<code>autoUpdateTime</code>	更新时填充 Unix 秒级时间戳
<code>autoCreateTime:milli</code>	创建时填充毫秒级时间戳
<code>autoUpdateTime:nano</code>	更新时填充纳秒级时间戳

```
1 type User struct {
2     ID      uint
3     CreatedAt int64 `gorm:"autoCreateTime"`        // 秒级时间戳
4     UpdatedAt int64 `gorm:"autoUpdateTime:milli"` // 毫秒级时间戳
5 }
```

主键

默认情况下，名为 `Id` 的字段就是主键，当然也可以使用结构体标签指定主键字段，也支持联合主键

- 语法：`gorm:"primaryKey;"`

```
1 type Person struct {
2     Id      uint `gorm:"primaryKey;"`
3     Name    string
4     Address string
5     Mom    string
6     Dad    string
7
8     CreatedAt sql.NullTime
9     UpdatedAt sql.NullTime
10 }
```

索引

可以加快数据查找速度，使用了B+树

- `index`：创建普通索引
- `uniqueIndex`：创建唯一索引
- `index:idx_name`：指定索引名为 `idx_name`
- `uniqueIndex:idx_name`：指定唯一索引名为 `idx_name`
- `sort:desc`：设置索引顺序为降序
- 有多个选项的时候用逗号分开

```
1 type Person struct {
2     ID      uint   `gorm:"primaryKey;"`
3     Name    string `gorm:"index"`           // 普通索引，名字由 GORM 自动生成
4     Email   string `gorm:"uniqueIndex"`      // 唯一索引
5     Address string `gorm:"index:idx_addr,unique"` // 指定索引名，并设置唯一
6 }
```

外键

定义外键关系，是通过嵌入结构体的方式来进行的，不是数据库字段，是 **GORM** 的关联关系

语法

- `foreignKey`：本表外键是谁
- `reference`：关联对方表的哪个字段
- `constraint`：外键约束

不同外键写法

- 基本写法

```

1 type User struct {
2     ID      uint
3     Name    string
4     Orders  []Order
5 }
6
7 type Order struct {
8     ID      uint
9     UserID  uint   // 外键字段
10    User    User   `gorm:"foreignKey:UserID"`
11 }

```

- 反向写法，Gorm可以自动推断外键

```

1 type Order struct {
2     ID      uint
3     UserID  uint
4     User    User
5 }

```

外键约束

- 使用 `constrained:数据库操作:约束条件`

约束关键字	意思
RESTRICT / NO ACTION	禁止删除或更新父表记录（默认）
CASCADE	父表删除/更新，子表自动跟着删/改
SET NULL	父表删除/更新，子表外键字段变成 NULL
SET DEFAULT	父表删除/更新，子表外键设为默认值

标签

字段映射与类型定义

标签名	说明	示例 (<code>gorm="..."</code>)
<code>column</code>	指定数据库中的列名。	<code>column:user_name</code>
<code>type</code>	指定列的数据类型（Go通用类型或完整原生类型）。	<code>type:string</code> 或 <code>type:varchar(255)</code>
<code>size</code>	定义数据类型的大小或长度。	<code>size:100</code>
<code>default</code>	定义列的默认值。	<code>default:0</code>

标签名	说明	示例 (gorm="...")
comment	迁移时为字段添加注释。	comment:用户昵称
serializer	指定序列化/反序列化方法 (如 JSON)。	serializer:json
precision	指定列的精度 (用于浮点数或 decimal)。	
scale	指定列的大小/小数位数 (用于浮点数或 decimal)。	
embedded	嵌套结构体字段，作为当前表中的列。	embedded
embeddedPrefix	嵌入字段的列名前缀。	embeddedPrefix:addr_

约束、索引与自增

标签名	说明	示例 (gorm="...")
primaryKey	将字段定义为主键。	primaryKey
unique	将字段定义为唯一键/约束。	unique
not null	指定列为 NOT NULL (不可为空)。	not null
autoIncrement	指定列为自动增长。	autoIncrement
autoIncrementIncrement	自动步长，控制连续记录之间的间隔。	
index	创建普通索引 (同名创建复合索引)。	index:idx_status
uniqueIndex	创建唯一索引 (同名创建复合唯一索引)。	uniqueIndex:uni_email
check	创建检查约束。	check:age > 18

权限控制与忽略

标签名	说明	示例 (gorm="...")
<-	设置字段的写入权限 (create / update / false)。	<-:create (只创建时写入)
->	设置字段的读取权限 (false)。	->:false (无读权限)
-	忽略该字段，不进行读写和迁移。	-
-:	忽略迁移 (-: 后跟 migration 或 all)。	-:migration

自动时间追踪

标签名	说明	示例 (gorm="...")
autoCreateTime	创建时自动追踪时间 (支持 nano / milli)。	autoCreateTime:milli
autoUpdateTime	创建/更新时自动追踪时间 (支持 nano / milli)。	autoUpdateTime

关系配置

标签名	说明	示例 (<code>gorm="..."</code>)
<code>foreignKey</code>	指定当前模型的外键列名。	
<code>references</code>	指定外键引用表的列名。	
<code>polymorphic</code>	指定多态类型 (如模型名)。	
<code>polymorphicValue</code>	指定多态值。	
<code>many2many</code>	指定连接表表名 (用于多对多)。	
<code>joinForeignKey</code>	指定连接表的外键列名, 映射到当前表。	
<code>joinReferences</code>	指定连接表的外键列名, 映射到引用表。	
<code>constraint</code>	指定关系上的 OnUpdate/OnDelete 约束行为。	<code>constraint:OnDelete:CASCADE</code>

钩子

钩子就是在对数据库进行操作时, 自动触发的函数, 可以在操作前或者操作后执行一些逻辑

- 常用的钩子

钩子名称	触发时机
<code>BeforeCreate</code>	插入记录前
<code>AfterCreate</code>	插入记录后
<code>BeforeUpdate</code>	更新记录前
<code>AfterUpdate</code>	更新记录后
<code>BeforeSave</code>	保存记录前 (Create 或 Update 都会触发)
<code>AfterSave</code>	保存记录后
<code>BeforeDelete</code>	删除记录前
<code>AfterDelete</code>	删除记录后
<code>AfterFind</code>	查询记录后

- 方法签名
 - 若执行钩子函数的时候返回 `err`, 会停止数据库操作
 - 钩子函数是在结构体上定义的方法

```
func (obj *StructName) HookName(tx *gorm.DB) (err error)
```

迁移

迁移就是根据Go结构体自动创建或更新数据库表结构

主要方法

- 自动迁移: `AutoMigrate()`
 - 会根据结构体生成表和列
 - 若表存在, 自动新增缺失的列和索引, 若表不存在, 自动建表

```
db.AutoMigrate(&User{})
```

- 手动迁移: `Migrator API`, 如下所示

```
1 type Migrator interface {
2     // AutoMigrate
3     AutoMigrate(dst ...interface{}) error
4
5     // Database
6     CurrentDatabase() string
7     FullDataTypeOf(*schema.Field) clause.Expr
8     GetTypeAliases(databaseTypeName string) []string
9
10    // Tables
11    CreateTable(dst ...interface{}) error
12    DropTable(dst ...interface{}) error
13    HasTable(dst interface{}) bool
14    RenameTable(oldName, newName interface{}) error
15    GetTables() (tableList []string, err error)
16    TableType(dst interface{}) (TableType, error)
17
18    // Columns
19    AddColumn(dst interface{}, field string) error
20    DropColumn(dst interface{}, field string) error
21    AlterColumn(dst interface{}, field string) error
22    MigrateColumn(dst interface{}, field *schema.Field, columnType ColumnType)
23    error
24    HasColumn(dst interface{}, field string) bool
25    RenameColumn(dst interface{}, oldName, field string) error
26    ColumnTypes(dst interface{}) ([]ColumnType, error)
27
28    // Views
29    CreateView(name string, option ViewOption) error
30    DropView(name string) error
31
32    // Constraints
33    CreateConstraint(dst interface{}, name string) error
34    DropConstraint(dst interface{}, name string) error
35    HasConstraint(dst interface{}, name string) bool
36
37    // Indexes
38    CreateIndex(dst interface{}, name string) error
39    DropIndex(dst interface{}, name string) error
40    HasIndex(dst interface{}, name string) bool
41    RenameIndex(dst interface{}, oldName, newName string) error
42    GetIndexes(dst interface{}) ([]Index, error)
43 }
```

创建

Create

在创建新的记录时，大多数情况都会用到 `Create` 方法

```
func (db *DB) Create(value interface{}) (tx *DB)
```

例子

- 传入的必须是指针，因为创建完成后，gorm 会将主键写入 user 结构体中
- 如果传入的是一个切片，就会进行**批量创建**
- 执行之后可以获取 `err`，查看执行数据库操作过程中是否发生错误

```
1 user := Person{  
2     Name: "jack",  
3 }  
4  
5 // 必须传入引用  
6 db = db.Create(&user)  
7  
8 // 执行过程中发生的错误  
9 err = db.Error  
10 // 创建的数目  
11 affected := db.RowsAffected
```

- 当传入的数据量太大的时候，可以使用 `CreateInBatches` 进行批次创建

```
db = db.CreateInBatches(&user, 50)
```

- `Save` 方法也可以创建记录，它的作用是当主键匹配时就更新记录，否则就插入

```
1 // 函数签名  
2 func (db *DB) Save(value interface{}) (tx *DB)  
3 // 例子  
4 user := []Person{  
5     {Name: "jack"},  
6     {Name: "mike"},  
7     {Name: "lili"},  
8 }  
9  
10 db = db.Save(&user)
```

Upsert

`Save` 方法只是对主键进行匹配，可以用 `Clause` 来完成更加自定义的 `upsert`

- 使用 `clause.OnConflict` 进行自定义条件能够以插入和更新

- 下面是 `clause.OnConflict` 结构体的完整字段

```
1 type OnConflict struct {
2     Columns      []Column
3     Where        Where
4     TargetWhere  Where
5     OnConstraint string
6     DoNothing    bool
7     DoUpdates   AssignmentSet
8     UpdateAll   bool
9 }
```

- `Columns`：使用 `[]clause.Column` 结构体指定冲突检测字段，并且可以指定排序
- `OnConstraint`：当有多个约束名的时候可以指定使用哪个约束
- `TargetWhere`：使用 `clause.Where{}` 结构体指定表达式列表，常用在冲突检测阶段
 - 使用 `[]clause.Expression` 指定查询条件
- `Where`：与 `TargetWhere` 语法类似，常用在更新阶段
- `DoUpdates`：使用 `clause.Assignments` 接收一个 `clause.Assignment` 的切片
 - 在切片中有三种不同的写法：匿名结构体，显式创建，`map` 写法
 - 匿名结构体

```
1 DoUpdates: clause.Assignments([]clause.Assignment{
2     {Column: clause.Column{Name: "name"}, Value: "new_name"},
3     {Column: clause.Column{Name: "login_count"}, Value:
4         gorm.Expr("login_count + 1")},
5     {Column: clause.Column{Name: "updated_at"}, Value:
6         gorm.Expr("NOW()")},
7 }),
```

- 显式创建

```
1 DoUpdates: clause.Assignments([]clause.Assignment{
2     clause.Assignment{
3         Column: clause.Column{Name: "name"},
4         Value:  "new_name",
5     },
6     clause.Assignment{
7         Column: clause.Column{Name: "login_count"},
8         Value:  gorm.Expr("login_count + 1"),
9     },
10    )),
11 },
```

- map 写法

```
1 DoUpdates: clause.Assignments(map[string]interface{}{
2     "name":          "new_name",
3     "login_count":   gorm.Expr("login_count + 1"),
4     "updated_at":    gorm.Expr("NOW()"),
5 }),
```

- 记得给冲突字段加索引，这样可以保证性能和约束条件

- 要在最后执行 Create 方法

```
1 db.Clauses(clause.OnConflict{
2     Columns:  []clause.Column{{Name: "name"}}, // 冲突检测字段
3     DoNothing: false,
4     DoUpdates: clause.AssignmentColumns([]string{"address"}), // 只更新 address
5     UpdateAll: false,
6 }).Create(&p)
```

增加

First

方法签名

```
func (db *DB) First(dest interface{}, cnds ...interface{}) (tx *DB)
```

- 按照主键升序查找第一条记录
- 也可以使用 Table 和 Model 指定查询表，前者接收字符串表名，后者接收实体模型

```
1 db.Table("person").Find(&p)
2 db.Model(Person{}).Find(&p)
```

Take

与 `First` 一样，但是不会根据主键排序

```
func (db *DB) Take(dest interface{}, cnds ...interface{}) (tx *DB)
```

Plunk

用于批量查询一个表的单列，查询的结果可以收集到一个指定类型的切片中，**不一定非得是实体类型的切片**

```
func (db *DB) Pluck(column string, dest interface{}) (tx *DB)
```

将所有人的地址搜集到一个字符串切片中

```
1 var adds []string
2
3 // SELECT `address` FROM `person` WHERE name IN ('jack','lili')
4 db.Model(Person{}).Where("name IN ?", []string{"jack", "lili"}).Pluck("address",
&adds)
```

Count

用于统计实体记录的数量

- 需要传入变量的指针

```
func (db *DB) Count(count *int64) (tx *DB)
```

例子

```
1 var count int64
2
3 // SELECT count(*) FROM `person`
4 db.Model(Person{}).Count(&count)
```

Find

批量查询

```
func (db *DB) Find(dest interface{}, args ...interface{}) (tx *DB)
```

Select

通过 `Select` 方法指定查询字段

```
func (db *DB) Select(query interface{}, args ...interface{}) (tx *DB)
```

例子

```
1 // SELECT `address`, `name` FROM `person` ORDER BY `person`.`id` LIMIT 1
2 db.Select("address", "name").First(&p)
```

同时可以使用 `Omit` 方法忽略字段

```
func (db *DB) Omit(columns ...string) (tx *DB)
```

例子

```
1 // SELECT `person`.`id`, `person`.`name` FROM `person` WHERE id IN (1,2,3,4)
2 db.Omit("address").Where("id IN ?", []int{1, 2, 3, 4}).Find(&ps)`
```

Where

条件查询

```
func (db *DB) Where(query interface{}, args ...interface{}) (tx *DB)
```

使用 `?` 进行占位，在后面的 `args` 参数列表填入具体数值

```
1 var p Person
2
3 db.Where("id = ?", 1).First(&p)
```

或者使用 `Or` 方法来构建 OR 语句

```
func (db *DB) Or(query interface{}, args ...interface{}) (tx *DB)
```

Or 方法也可以用 ? 占位

```
1 // SELECT * FROM `person` WHERE id = 1 OR name = 'jack' AND address = 'usa' ORDER
2 BY `person`.`id` LIMIT 1
3 db.Where("id = ?", 1).
4     Or("name = ?", "jack").
5     Where("address = ?", "usa").
6     First(&p)
```

Not 方法也是一样

```
func (db *DB) Not(query interface{}, args ...interface{}) (tx *DB)
```

对于 IN 条件，可以直接在 Where 方法里面传入切片

```
db.Where("address IN ?", []string{"cn", "us"}).Find(&ps)
```

对于多列 IN 条件，需要用 [][]any 类型来承载参数

```
1 // SELECT * FROM `person` WHERE (id, name, address) IN ((1,'jack','uk'),
2 ('mike','usa'))
3 db.Where("(id, name, address) IN ?", [][]any{{1, "jack", "uk"}, {2, "mike",
4 "usa"}}).Find(&ps)
```

Order

排序使用的方法

```
func (db *DB) Order(value interface{}) (tx *DB)
```

支持多次调用

```
1 // SELECT * FROM `person` ORDER BY name ASC, id DESC,address
2 db.Order("name ASC, id DESC").Order("address").Find(&ps)
```

Limit

用于分页查询，有 Limit 和 Offset 方法

```
1 func (db *DB) Limit(limit int) (tx *DB)
2
3 func (db *DB) Offset(offset int) (tx *DB)
```

例子

```
1 var (
2     ps []Person
3     page = 2
4     size = 10
5 )
6
7 // SELECT * FROM `person` LIMIT 10 OFFSET 10
8 db.Offset((page - 1) * size).Limit(size).Find(&ps)
```

Group

Group 和 Having 方法多用于分组操作

```
1 func (db *DB) Group(name string) (tx *DB)
2
3 func (db *DB) Having(query interface{}, args ...interface{}) (tx *DB)
```

例子

```
1 var (
2     ps []Person
3 )
4
5 // SELECT `address` FROM `person` GROUP BY `address` HAVING address IN
6 // ('cn','us')
7 db.Select("address").Group("address").Having("address IN ?", []string{"cn",
8 "us"}).Find(&ps)
```

Distinct

多用于去重

```
func (db *DB) Distinct(args ...interface{}) (tx *DB)
```

例子

```
1 // SELECT DISTINCT `name` FROM `person` WHERE address IN ('cn', 'us')
2 db.Where("address IN ?", []string{"cn", "us"}).Distinct("name").Find(&ps)
```

子查询

嵌套 `Select` 即可

- `Where` 中使用子查询

```
1 // SELECT * FROM `person` WHERE id > (SELECT AVG(id) FROM `person`
2 db.Where("id > (?)", db.Model(Person{}).Select("AVG(id)").Find(&ps))
```

- `from` 中使用子查询

```
1 sub := db.Model(&Order{}).Select("user_id, SUM(amount) as
2   total").Group("user_id")
3 db.Table("(?) as t", sub).Where("t.total > ?", 100).Find(&result)
```

通用格式

```
1 // 创建子查询
2 sub := db.Model(&A{}).Select("xxx").Where("...")
3
4 // 使用
5 db.Where("column IN (?)", sub).Find(&list)
6
7 // 或者作为表
8 db.Table("(?) as t", sub).Find(&list)
```

锁

gorm 使用 `clause.Locking` 子句来提供锁的支持

```
1 // SELECT * FROM `person` FOR UPDATE
2 db.Clauses(clause.Locking{Strength: "UPDATE"}).Find(&ps)
3
4 // SELECT * FROM `person` FOR SHARE NOWAIT
5 db.Clauses(clause.Locking{Strength: "SHARE", Options: "NOWAIT"}).Find(&ps)
```

悲观锁

在SQL语句里面使用 `FOR UPDATE` 启用悲观锁，在Gorm里面需要在事务里面启用

- `Strength` : 用来指定锁的类型

<code>Strength</code>	对应 SQL	作用
<code>"UPDATE"</code>	<code>FOR UPDATE</code>	写锁
<code>"SHARE"</code>	<code>FOR SHARE</code>	读锁
<code>"NO KEY UPDATE"</code>	PostgreSQL 专用	不允许更新主键
<code>"KEY SHARE"</code>	PostgreSQL 专用	更弱的读锁

```
1 import "gorm.io/gorm/clause"
2
3 tx := db.Begin() // 必须用事务
4 tx.Clauses(clause.Locking{Strength: "UPDATE"}).First(&user, 1)
5 tx.Commit()
```

乐观锁

在定义结构体的时候加上 `gorm:"version"` 结构体标签即可

```
1 type User struct {
2     ID      uint
3     Balance int
4     Version int `gorm:"version"`
5 }
```

修改

Save

Update

主要用来更新单列字段

```
func (db *DB) Update(column string, value interface{}) (tx *DB)
```

例子

```
1 var p Person
2
3 db.First(&p)
4
5 // UPDATE `person` SET `address`='poland' WHERE id = 2
6 db.Model(Person{}).Where("id = ?", p.Id).Update("address", "poland")
```

Updates

`Updates` 方法用于更新多列，接收结构体和 map 作为参数，并且当结构体字段为零值时，会忽略该字段，但在 map 中不会

```
func (db *DB) Updates(values interface{}) (tx *DB)
```

例子

```
1 var p Person
2
3 db.First(&p)
4
5 // UPDATE `person` SET `name`='jojo', `address`='poland' WHERE `id` = 2
6 db.Model(p).Updates(Person{Name: "jojo", Address: "poland"})
7
8 // UPDATE `person` SET `address`='poland', `name`='jojo' WHERE `id` = 2
9 db.Model(p).Updates(map[string]any{"name": "jojo", "address": "poland"})
```

SQL表达式

有时候要对字段进行一些自增自减的操作，然后进行更新，这就要用到SQL表达式

上面的 `DoUpdates` 中就用了 `claude.Expr`

```
func Expr(expr string, args ...interface{}) clause.Expr
```

例子

```
1 // UPDATE `person` SET `age`=age + age, `name`='jojo' WHERE `id` = 2
2 db.Model(p).Updates(map[string]any{"name": "jojo", "age": gorm.Expr("age +
age")})
3
4 // UPDATE `person` SET `age`=age * 2 + age, `name`='jojo' WHERE `id` = 2
5 db.Model(p).Updates(map[string]any{"name": "jojo", "age": gorm.Expr("age * 2 +
age")})
```

删除

Delete

可以传实体结构，也可以传条件，若要执行批量删除就是传入切片

```
func (db *DB) Delete(value interface{},conds ...interface{}) (tx *DB)
```

例子

```
1 // DELETE FROM `person` WHERE id = 2
2 db.Model(Person{}).Where("id = ?", p.Id).Delete(nil)
```

软删除

定义

不真正从数据库中删除记录，而是通过一个标志字段来标记已删除状态

假如实体模型使用了软删除，那么在删除时，默认进行更新操作，若要永久删除的话可以使用 `Unscope` 方法

```
db.Unscoped().Delete(&Person{}, []uint{1, 2, 3})
```

事务

自动

闭包事务，通过 `Transaction` 方法传入一个闭包函数，如果函数返回值不为 `nil`，那么就会自动回滚

```
1 var ps []Person
2
3 db.Transaction(func(tx *gorm.DB) error {
4     err := tx.Create(&ps).Error
5     // 出错 回滚
6     if err != nil {
7         return err
8     }
9
10    err = tx.Model(Person{}).Where("id = ?", 1).Update("name", "jack").Error
11    // 出错 回滚
12    if err != nil {
13        return err
14    }
15
16    return nil
17 })
```

手动

推荐使用手动事务，由我们自己来控制何时回滚，何时提交，有下面三个方法

```
1 // Begin方法用于开启事务
2 func (db *DB) Begin(opts ...*sql.TxOptions) *DB
3
4 // Rollback方法用于回滚事务
5 func (db *DB) Rollback() *DB
6
7 // Commit方法用于提交事务
8 func (db *DB) Commit() *DB
```

例子

```

1 var ps []Person
2
3 tx := db.Begin()
4
5 err := tx.Create(&ps).Error
6 if err != nil {
7     tx.Rollback()
8     return
9 }
10
11 err = tx.Create(&ps).Error
12 if err != nil {
13     tx.Rollback()
14     return
15 }
16
17 err = tx.Model(Person{}).Where("id = ?", 1).Update("name", "jack").Error
18 if err != nil {
19     tx.Rollback()
20     return
21 }
22
23 tx.Commit()

```

关联定义

通过嵌入结构体和字段的形式来定义结构体与结构体之间的关联，创建了关联之后要注意各表的先后创建顺序

不同关联方式

- `has_one` : 一对一，不是标签
- `has_many` : 一对多，不是标签
- `many2many` : 多对多，是标签，后面跟中间表的表名

一对多

```

1 type User struct {
2     ID      uint
3     Orders []Order
4 }
5
6 type Order struct {
7     UserID uint
8 }

```

一对一

```
1 type User struct {
2     ID      uint
3     Profile Profile
4 }
5
6 type Profile struct {
7     UserID uint `gorm:"unique"` // 1对1必须唯一
8 }
```

多对多

```
1 type User struct {
2     ID      uint
3     Roles  []Role `gorm:"many2many:user_roles;"`  

4 }
5
6 type Role struct {
7     ID uint
8 }
```

关联操作

下述操作都是在**数据层面**进行操作

使用 `Association` 方法来操作结构体之间的关联：

- `column`：对应**结构体（类）的字段名**（被引用的字段）
- 对某个实体的关联字段进行**增删改查**

```
func (db *DB) Association(column string) *Association
```

定义数据

```
1 // 定义好数据
2 type Person struct {
3     Id      uint
4     Name    string
5     Address string
6     Age     uint
7
8     MomId sql.NullInt64
9     Mom   Mom `gorm:"foreignKey:MomId;"` 
10
11    SchoolId sql.NullInt64
12    School  School `gorm:"foreignKey:SchoolId;"` 
13
14    Houses []House `gorm:"many2many:person_house;"` 
15 }
16
17 type Mom struct {
18     Id      uint
19     Name    string
20 }
21
22 type School struct {
23     Id      uint
24     Name    string
25
26     Persons []Person
27 }
28
29 type House struct {
30     Id      uint
31     Name    string
32
33     Persons []Person `gorm:"many2many:person_house;"` 
34 }
35
36 type PersonHouse struct {
37     PersonId sql.NullInt64
38     Person   Person `gorm:"foreignKey:PersonId;"` 
39     HouseId  sql.NullInt64
40     House    House `gorm:"foreignKey:HouseId;"` 
41 }
42 jenny := Mom{
43     Name: "jenny",
44 }
45
46 mit := School{
```

```

47     Name: "MIT",
48     Persons: nil,
49 }
50
51 h1 := House{
52     Id: 0,
53     Name: "h1",
54     Persons: nil,
55 }
56
57 h2 := House{
58     Name: "h2",
59     Persons: nil,
60 }
61
62 jack := Person{
63     Name: "jack",
64     Address: "usa",
65     Age: 18,
66 }
67
68 mike := Person{
69     Name: "mike",
70     Address: "uk",
71     Age: 20,
72 }

```

创建关联

使用 `Append()` 方法创建关联

- 使用 `[]any` 切片放置要插入的数据
- 多对多关联会在中间表添加对应关系

```

1 db.Create(&jack)           // 创建 Person
2 db.Create(&mit)            // 创建 School
3 db.Model(&jack).Association("Mom").Append(&jenny) // 一对一
4 db.Model(&mit).Association("Persons").Append([]Person{jack, mike}) // 一对多
5 db.Model(&jack).Association("Houses").Append([]House{h1, h2})      // 多对多

```

查找关联

使用 `Find()` 方法创建关联

- 多对多会去查询中间表
- 只查询单个主实体的关联，不返回主实体字段，也就是不会自动把结果填到主实体的字段里

```
1 db.Model(&person).Association("Mom").Find(&mom)           // 一对一
2 db.Model(&school).Association("Persons").Find(&persons) // 一对多
3 db.Model(&persons).Association("Houses").Find(&houses)  // 多对多
```

更新关联

使用 `Replace()` 方法创建关联

- 底层会删除旧关联，创建新关联

```
1 db.Model(&jack).Association("Mom").Replace(&lili)           // 一对一
2 db.Model(&mit).Association("Persons").Replace(newPerson)      // 一对多
3 db.Model(&jack).Association("Houses").Replace([]House{h3,h4,h5}) // 多对多
```

删除关联

使用 `Delete()` 方法创建关联，只是删除表记录之间的关系，也就是数据

- 只删除关联关系，不清除实体
- 多对多不支持删除实体

```
1 db.Model(&jack).Association("Mom").Delete(&lili)           // 一对一
2 db.Model(&mit).Association("Persons").Delete(&persons)        // 一对多
3 db.Model(&jack).Association("Houses").Delete(&houses)        // 多对多
```

预加载

使用 `Preload()` 方法一次性加载关联实体，查询时把关联的数据提前加载到主实体里

- 一次性查主表和关联表，每个主实体的关联字段都会自动填充
- 支持嵌套，一次查询完成多层关联

```
1 var users []Person
2 db.Preload("Mom").Find(&users)
```

其它方法

`Clear()`：清空指定实体的全部关联关系，但不删除关联的实体

- 对一对一、一对多、多对多都适用

```
db.Model(&jack).Association("Houses").Clear()
```

`Unscoped().Delete()` : 删除关联关系并且删除实体记录

- 支持一对一、一对多，**不支持多对多**

```
db.Model(&jack).Association("Houses").Unscoped().Delete(&houses)
```