GORM V2 中文文档

SliverHorn

目　录

入门指南

目前GormV2最佳实践后台管理系统 gin-vue-admin

- 基于gin+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

gf-vue-admin

- 基于goframe+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

导航

概述

GORM 指南

用于Golang的出色的ORM库旨在对开发人员友好。

特性

- 全功能 ORM
- 关联 (Has One、Has Many、Belongs To、Many To Many、多态、单表继承)
- Create、Save、Update、Delete、Find 前/后的勾子
- 基于 `Preload` 、 `Joins` 的预加载
- 事务、嵌套事务、保存点、回滚至保存点
- Context、Prepared Statment 模式、DryRun 模式
- 批量插入、FindInBatches、查询至 Map
- SQL Builder, Upsert, Locking, Optimizer/Index/Comment Hints
- 复合主键
- 自动迁移
- 自定义 Logger
- 灵活的可扩展插件 API：Database Resolver（读写分离）、Prometheus...
- 所有特性都通过了测试
- 开发者友好

安装

```
go get -u gorm.io/gorm
go get -u gorm.io/driver/sqlite
```

快速入门

```
package main

import (
  "gorm.io/gorm"
  "gorm.io/driver/sqlite"
)

type Product struct {
  gorm.Model
  Code  string
```

```
  Price uint
}

func main() {
  db, err := gorm.Open(sqlite.Open("test.db"), &amp;gorm.Config{})
  if err != nil {
    panic("failed to connect database")
  }

  // 迁移 schema
  db.AutoMigrate(&amp;Product{})

  // Create
  db.Create(&amp;Product{Code: "D42", Price: 100})

  // Read
  var product Product
  db.First(&amp;product, 1) // 根据整形主键查找
  db.First(&amp;product, "code = ?", "D42") // 查找 code 字段值为 D42 的记录

  // Update - 将 product 的 price 更新为 200
  db.Model(&amp;product).Update("Price", 200)
  // Update - 更新多个字段
  db.Model(&amp;product).Updates(Product{Price: 200, Code: "F42"}) // 仅更新非零
值字段
  db.Model(&amp;product).Updates(map[string]interface{}{"Price": 200, "Code": "
F42"})

  // Delete - 删除 product
  db.Delete(&amp;product, 1)
}
```

声明模型

模型定义

模型定义

模型一般基于 Go 的基本数据类型、实现了 Scanner 和 Valuer 接口的自定义类型以及它们的指针/别名

例如：

```
type User struct {
  ID           uint
  Name         string
  Email        *string
  Age          uint8
  Birthday     *time.Time
  MemberNumber sql.NullString
  ActivedAt    sql.NullTime
  CreatedAt    time.Time
  UpdatedAt    time.Time
}
```

约定

GORM 倾向于约定，而不是配置。默认情况下，GORM 使用 `ID` 作为主键，使用结构体名的 `蛇形复数` 作为表名，字段名的 `蛇形` 作为列名，并使用 `CreatedAt` 、 `UpdatedAt` 字段追踪创建、更新时间

遵循 GORM 已有的约定，可以减少您的配置和代码量。如果约定不符合您的需求，GORM 允许您自定义配置它们

gorm.Model

GORM 定义一个 `gorm.Model` 结构体，其包括字段 `ID` 、 `CreatedAt` 、 `UpdatedAt` 、 `DeletedAt`

```
// gorm.Model 的定义
type Model struct {
  ID        uint           `gorm:"primaryKey"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt gorm.DeletedAt `gorm:"index"`
}
```

您可以将它嵌入到您的结构体中，以包含这几个字段，详情请参考 嵌入结构体

```
type User struct {
  gorm.Model
  Name string
}
// 等效于
type User struct {
  ID        uint           `gorm:"primaryKey"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt gorm.DeletedAt `gorm:"index"`
  Name string
}
```

高级选项

字段级权限控制

可导出的字段在使用 GORM 进行 CRUD 时拥有全部的权限，此外，GORM 允许您用标签控制字段级别的权限。这样您就可以让一个字段的权限是只读、只写、只创建、只更新或者被忽略

```
type User struct {
  Name string `gorm:"<-:create"` // 允许读和创建
  Name string `gorm:"<-:update"` // 允许读和更新
  Name string `gorm:"<-"`        // 允许读和写（创建和更新）
  Name string `gorm:"<-:false"`  // 允许读，禁止写
  Name string `gorm:"->"`        // 只读（除非有自定义配置，否则禁止写）
  Name string `gorm:"->;<-:create"` // 允许读和写
  Name string `gorm:"->:false;<-:create"` // 仅创建（禁止从 db 读）
  Name string `gorm:"-"`  // 读写操作均会忽略该字段
}
```

创建/更新时间追踪（纳秒、毫秒、秒、Time）

GORM 约定使用 `CreatedAt` 、 `UpdatedAt` 追踪创建/更新时间。如果您定义了他们，GORM 在创建/更新时会自动填充 当前时间 至这些字段

要使用不同名称的字段，您可以配置 `autoCreateTim` 、 `autoUpdateTim` 标签

如果您想要保存纳秒、毫秒、秒级 UNIX 时间戳，而不是 time，您只需简单地将 `time.Time` 修改为 `int` 即可

```
type User struct {
  CreatedAt time.Time // 在创建时，如果该字段值为零值，则使用当前时间填充
  UpdatedAt int       // 在创建时该字段值为零值或者在更新时，使用当前秒级时间戳填充
  Updated   int64 `gorm:"autoUpdateTime:nano"` // 使用纳秒级时间戳填充更新时间
  Updated   int64 `gorm:"autoUpdateTime:milli"` // 使用毫秒级时间戳填充更新时间
  Created   int64 `gorm:"autoCreateTime"`      // 使用秒级时间戳填充创建时间
```

```
}
```

嵌入结构体

对于匿名字段，GORM 会将其字段包含在父结构体中，例如：

```
type User struct {
  gorm.Model
  Name string
}
// 等效于
type User struct {
  ID        uint           `gorm:"primaryKey"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt gorm.DeletedAt `gorm:"index"`
  Name string
}
```

对于正常的结构体字段，你也可以通过标签 `embedded` 将其嵌入，例如：

```
type Author struct {
  Name  string
  Email string
}

type Blog struct {
  ID      int
  Author  Author `gorm:"embedded"`
  Upvotes int32
}
// 等效于
type Blog struct {
  ID    int64
  Name  string
  Email string
  Upvotes  int32
}
```

并且，您可以使用标签 `embeddrefix` 来为 db 中的字段名添加前缀，例如：

```
type Blog struct {
  ID      int
  Author  Author `gorm:"embedded;embeddedPrefix:author_"`
  Upvotes int32
}
```

```
// 等效于
type Blog struct {
  ID          int64
  AuthorName  string
  AuthorEmail string
  Upvotes     int32
}
```

字段标签

在声明模型时，标签是可选的，GORM 支持以下标签：

标签名对大小写不敏感，但建议使用 `小驼峰（camelCase）` 的命名方式。

标签名	说明
column	指定 db 列名
type	列数据类型，推荐使用兼容性好的通用类型，例如：bool、int、uint、float、string、time、bytes。并可与其他标签一起使用，例如：`not null` 、 `size` ， `autoIncrement` ... 像 `varbinary(8)` 这样指定数据库数据类型也是支持的。在使用指定数据库数据类型时，它需要是完整的数据库数据类型，如：`MEDIUMINT UNSINED not NULL AU`
size	指定列大小，例如： `size:256`
primaryKey	指定列为主键
unique	指定列为唯一
default	指定列的默认值
precision	指定列的精度
not null	指定列为 NOT NULL
autoIncrement	指定列为自动增长
embedded	嵌套字段
embeddedPrefix	嵌套字段的前缀
autoCreateTime	创建时追踪当前时间，对于 `int` 字段，它会追踪秒级时间戳，您可以使用 `nano` / `milli` 来追踪纳秒、毫秒时间戳，例如：`autoCreateTime:nano`

autoUpdateTime	创建/更新时追踪当前时间，对于 `int` 字段，它会追踪秒级时间戳，您可以使用 `nano` / `milli` 来追踪纳秒、毫秒时间戳，例如：`autoUpdateTime:milli`
index	根据参数创建索引，多个字段拥有相同的名称则创建复合索引，参考 索引 获取详情
uniqueIndex	与 `index` 相同，但创建的是唯一索引
check	创建检查约束，例如 `check:(age > 13)` ，查看 约束 获取详情
<-	设置字段写入的权限， `<-:create` 只创建、 `<-:update` 只更新、 `<-:false` 无权限
->	设置字段读取权限
-	忽略此字段（禁止读写）

关联标签

GORM 允许通过标签为关联配置外键、约束、many2many 表，详情请参考 关联部分

连接到数据库

连接到数据库

GORM 官方支持的数据库类型有: MySQL, PostgreSQL, SQlite, SQL Server

MySQL

```
import (
  "gorm.io/driver/mysql"
  "gorm.io/gorm"
)

func main() {
  // 参考 https://github.com/go-sql-driver/mysql#dsn-data-source-name 获取详情
  dsn := "user:pass@tcp(127.0.0.1:3306)/dbname?charset=utf8mb4&parseTime=True&loc=Local"
  db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
}
```

注意：

想要 Gorm 正确的处理 `time.Time` ，您需要带上 `parseTime` 参数。 (查看更多参数)

想要支持完整的 UTF-8 编码，您需要将 `charset=utf8` 更改为 `charset=utf8mb4` 。 查看 此文章 获取详情

MySQl 驱动程序提供了 一些高级配置 可以在初始化过程中使用，例如：

```
db, err := gorm.Open(mysql.New(mysql.Config{
  DSN: "gorm:gorm@tcp(127.0.0.1:3306)/gorm?charset=utf8&parseTime=True&loc=Local", // DSN data source name
  DefaultStringSize: 256, // string 类型字段的默认长度
  DisableDatetimePrecision: true, // 禁用 datetime 精度, MySQL 5.6 之前的数据库不支持

  DontSupportRenameIndex: true, // 重命名索引时采用删除并新建的方式, MySQL 5.7 之前的数据库和 MariaDB 不支持重命名索引
  DontSupportRenameColumn: true, // 用 `change` 重命名列, MySQL 8 之前的数据库和 MariaDB 不支持重命名列
  SkipInitializeWithVersion: false, // 根据版本自动配置
}), &gorm.Config{})
```

PostgreSQL

```
import (
```

```
  "gorm.io/driver/postgres"
  "gorm.io/gorm"
)

dsn := "user=gorm password=gorm dbname=gorm port=9920 sslmode=disable TimeZone=
Asia/Shanghai"
db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
```

我们使用 pgx 作为 postgres 的 database/sql 驱动器，默认情况下，它会启用 prepared statement 缓存，你可以这样禁用它：

```
// https://github.com/go-gorm/postgres
db, err := gorm.Open(postgres.New(postgres.Config{
  DSN: "user=gorm password=gorm dbname=gorm port=9920 sslmode=disable TimeZone=
Asia/Shanghai",
  PreferSimpleProtocol: true, // 禁用隐式 prepared statement
}), &gorm.Config{})
```

SQLite

```
import (
  "gorm.io/driver/sqlite"
  "gorm.io/gorm"
)

// github.com/mattn/go-sqlite3
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{})
```

注意： 您也可以使用 `file::memory:?cache=shared` 替代文件路径。 这会告诉 SQLite 在系统内存中使用一个临时数据库。 (查看 SQLite 文档 获取详情)

SQL Server

```
import (
  "gorm.io/driver/sqlserver"
  "gorm.io/gorm"
)

// github.com/denisenkom/go-mssqldb
dsn := "sqlserver://gorm:LoremIpsum86@localhost:9930?database=gorm"
db, err := gorm.Open(sqlserver.Open(dsn), &gorm.Config{})
```

Microsoft 为 GO (GORM) 使用 SQL Server 提供了 一份指南

连接池

GORM 使用 database/sql 来维护连接池

```
sqlDB, err := db.DB()

// SetMaxIdleConns 设置空闲连接池中连接的最大数量
sqlDB.SetMaxIgleConns(10)

// SetMaxOpenConns 设置打开数据库连接的最大数量
sqlDB.SetMaxOpenConns(100)

// SetConnMaxLifetime 设置了连接可复用的最大时间
sqlDB.SetConnMaxLifetime(time.Hour)
```

查看 通用接口 获取详情。

不支持的数据库

有些数据库可能兼容 `mysql` 、 `postgres` 的方言，在这种情况下，你可以直接使用这些数据库的方言。

对于其它不支持的数据，我们鼓励且欢迎大家伙开发更多数据库类型的驱动！

CRUD 接口

目前GormV2最佳实践后台管理系统 gin-vue-admin

- 基于gin+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

gf-vue-admin

- 基于goframe+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

导航

- 更新
- 删除
- 原生SQL和SQL生成器

创建

创建

创建记录

```
user := User{Name: "Jinzhu", Age: 18, Birthday: time.Now()}

result := db.Create(&user) // 通过数据的指针来创建

user.ID             // 返回插入数据的主键
result.Error        // 返回 error
result.RowsAffected // 返回插入记录的条数
```

选定字段创建

用选定字段的来创建

```
db.Select("Name", "Age", "CreatedAt").Create(&user)
// INSERT INTO `users` (`name`,`age`,`created_at`) VALUES ("jinzhu", 18, "2020-07-04 11:05:21.775")
```

创建时排除选定字段

```
db.Omit("Name", "Age", "CreatedAt").Create(&user)
// INSERT INTO `users` (`birthday`,`updated_at`) VALUES ("2020-01-01 00:00:00.000", "2020-07-04 11:05:21.775")
```

创建钩子

GORM 允许 `BeforeSave` , `BeforeCreate` , `AfterSave` , `AfterCreate` 等钩子，创建记录时会调用这些方法, 详情请参阅 钩子

```
func (u *User) BeforeCreate(tx *gorm.DB) (err error) {
  u.UUID = uuid.New()

    if u.Role == "admin" {
        return errors.New("invalid role")
    }
    return
}
```

批量插入

将切片数据传递给 `Create` 方法，GORM 将生成一个单一的 SQL 语句来插入所有数据，并回填主键的值，钩子方法也会被调用。

```
var users = []User{{Name: "jinzhu1"}, {Name: "jinzhu2"}, {Name: "jinzhu3"}}
DB.Create(&users)

for _, user := range users {
  user.ID // 1,2,3
}
```

Upsert、关联创建 同样支持批量插入

高级

关联创建

如果您的模型定义了任何关系（relation），并且它有非零关系，那么在创建时这些数据也会被保存

```
type CreditCard struct {
  gorm.Model
  Number   string
  UserID   uint
}

type User struct {
  gorm.Model
  Name       string
  CreditCard CreditCard
}

db.Create(&User{
  Name: "jinzhu",
  CreditCard: CreditCard{Number: "411111111111"}
})
// INSERT INTO `users` ...
// INSERT INTO `credit_cards` ...
```

您也可以通过 `Select` 、 `Omit` 跳过关联保存

```
db.Omit("CreditCard").Create(&user)

// 跳过所有关联
db.Omit(clause.Associations).Create(&user)
```

默认值

您可以通过标签 `default` 为字段定义默认值，如：

```
type User struct {
  ID         int64
  Name       string `gorm:"default:'galeone'"`
  Age        int64  `gorm:"default:18"`
    uuid.UUID  UUID   `gorm:"type:uuid;default:gen_random_uuid()"` // db 函数
}
```

插入记录到数据库时，零值 字段将使用默认值

注意 像 `0` 、 `''` 、 `false` 等零值，不会将这些字段定义的默认值保存到数据库。您需要使用指针类型或 Scanner/Valuer 来避免这个问题，例如：

```
type User struct {
  gorm.Model
  Name string
  Age  *int           `gorm:"default:18"`
  Active sql.NullBool `gorm:"default:true"`
}
```

注意 对于在数据库中有默认值的字段，你必须为其 struct 设置 `default` 标签，否则 GORM 将在创建时使用该字段的零值，例如：

```
type User struct {
    ID   string `gorm:"default:uuid_generate_v3()"`
    Name string
    Age  uint8
}
```

Upsert 及冲突

GORM 为不同数据库提供了兼容的 Upsert 支持

```
import "gorm.io/gorm/clause"

// 不处理冲突
DB.Clauses(clause.OnConflict{DoNothing: true}).Create(&user)

// `id` 冲突时，将字段值更新为默认值
DB.Clauses(clause.OnConflict{
  Columns:   []clause.Column{{Name: "id"}},
  DoUpdates: clause.Assignments(map[string]interface{}{"role": "user"}),
```

```
}).Create(&users)
// MERGE INTO "users" USING *** WHEN NOT MATCHED THEN INSERT *** WHEN MATCHED T
HEN UPDATE SET ***; SQL Server
// INSERT INTO `users` *** ON DUPLICATE KEY UPDATE ***; MySQL

// Update columns to new value on `id` conflict
DB.Clauses(clause.OnConflict{
  Columns:   []clause.Column{{Name: "id"}},
  DoUpdates: clause.AssignmentColumns([]string{"name", "age"}),
}).Create(&users)
// MERGE INTO "users" USING *** WHEN NOT MATCHED THEN INSERT *** WHEN MATCHED T
HEN UPDATE SET "name"="excluded"."name"; SQL Server
// INSERT INTO "users" *** ON CONFLICT ("id") DO UPDATE SET "name"="excluded"."
name", "age"="excluded"."age"; PostgreSQL
// INSERT INTO `users` *** ON DUPLICATE KEY UPDATE `name`=VALUES(name),`age=VAL
UES(age); MySQL
```

也可以查看 高级查询 中的 `FirstOrInit` , `FirstOrCreate`

查看 原生 SQL 及构造器 获取更多细节

查询

查询

检索单个对象

GORM 提供 `First`，`Take`，`Last` 方法，以便从数据库中检索单个对象。当查询数据库时它添加了 `LIMIT 1` 条件。当没有找到记录时，它会返回错误 `ErrRecordNotFound`

```
// 获取第一条记录（主键升序）
db.First(&user)
// SELECT * FROM users ORDER BY id LIMIT 1;

// 获取一条记录，没有指定排序字段
db.Take(&user)
// SELECT * FROM users LIMIT 1;

// 获取最后一条记录（主键降序）
db.Last(&user)
// SELECT * FROM users ORDER BY id DESC LIMIT 1;

result := db.First(&user)
result.RowsAffected // 返回找到的记录数
result.Error        // returns error

// 检查 ErrRecordNotFound 错误
errors.Is(result.Error, gorm.ErrRecordNotFound)
```

检索对象

```
// 获取全部记录
result := db.Find(&users)
// SELECT * FROM users;

result.RowsAffected // 返回找到的记录数，相当于 `len(users)`
result.Error        // returns error
```

条件

String 条件

```
// 获取第一条匹配的记录
db.Where("name = ?", "jinzhu").First(&user)
// SELECT * FROM users WHERE name = 'jinzhu' ORDER BY id LIMIT 1;
```

```
// 获取全部匹配的记录
db.Where("name <> ?", "jinzhu").Find(&users)
// SELECT * FROM users WHERE name <> 'jinzhu';

// IN
db.Where("name IN ?", []string{"jinzhu", "jinzhu 2"}).Find(&users)
// SELECT * FROM users WHERE name in ('jinzhu','jinzhu 2');

// LIKE
db.Where("name LIKE ?", "%jin%").Find(&users)
// SELECT * FROM users WHERE name LIKE '%jin%';

// AND
db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)
// SELECT * FROM users WHERE name = 'jinzhu' AND age >= 22;

// Time
db.Where("updated_at > ?", lastWeek).Find(&users)
// SELECT * FROM users WHERE updated_at > '2000-01-01 00:00:00';

// BETWEEN
db.Where("created_at BETWEEN ? AND ?", lastWeek, today).Find(&users)
// SELECT * FROM users WHERE created_at BETWEEN '2000-01-01 00:00:00' AND '2000
-01-08 00:00:00';
```

Struct & Map 条件

```
// Struct
db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)
// SELECT * FROM users WHERE name = "jinzhu" AND age = 20 ORDER BY id LIMIT 1;

// Map
db.Where(map[string]interface{}{"name": "jinzhu", "age": 20}).Find(&users)
// SELECT * FROM users WHERE name = "jinzhu" AND age = 20;

// 主键切片条件
db.Where([]int64{20, 21, 22}).Find(&users)
// SELECT * FROM users WHERE id IN (20, 21, 22);
```

注意 当使用结构作为条件查询时，GORM 只会查询非零值字段。这意味着如果您的字段值为 `0` 、 `''` 、 `false` 或其他 零值，该字段不会被用于构建查询条件，例如：

```
db.Where(&User{Name: "jinzhu", Age: 0}).Find(&users)
// SELECT * FROM users WHERE name = "jinzhu";
```

您可以使用 map 来构建查询条件，例如：

```
db.Where(map[string]interface{}{"Name": "jinzhu", "Age": 0}).Find(&users)
// SELECT * FROM users WHERE name = "jinzhu" AND age = 0;
```

内联条件

用法与 `Where` 类似

```
// 根据主键获取记录（仅适用于整型主键）
db.First(&user, 23)
// SELECT * FROM users WHERE id = 23;
// 根据主键获取记录，如果是非整型主键
db.First(&user, "id = ?", "string_primary_key")
// SELECT * FROM users WHERE id = 'string_primary_key';

// Plain SQL
db.Find(&user, "name = ?", "jinzhu")
// SELECT * FROM users WHERE name = "jinzhu";

db.Find(&users, "name <> ? AND age > ?", "jinzhu", 20)
// SELECT * FROM users WHERE name <> "jinzhu" AND age > 20;

// Struct
db.Find(&users, User{Age: 20})
// SELECT * FROM users WHERE age = 20;

// Map
db.Find(&users, map[string]interface{}{"age": 20})
// SELECT * FROM users WHERE age = 20;
```

Not 条件

构建 NOT 条件，用法与 `Where` 类似

```
db.Not("name = ?", "jinzhu").First(&user)
// SELECT * FROM users WHERE NOT name = "jinzhu" ORDER BY id LIMIT 1;

// Not In
db.Not(map[string]interface{}{"name": []string{"jinzhu", "jinzhu 2"}}).Find(&us
ers)
// SELECT * FROM users WHERE name NOT IN ("jinzhu", "jinzhu 2");

// Struct
db.Not(User{Name: "jinzhu", Age: 18}).First(&user)
// SELECT * FROM users WHERE name <> "jinzhu" AND age <> 18 ORDER BY id LIMIT 1;
```

```
// 不在主键切片中的记录
db.Not([]int64{1,2,3}).First(&user)
// SELECT * FROM users WHERE id NOT IN (1,2,3) ORDER BY id LIMIT 1;
```

Or 条件

```
db.Where("role = ?", "admin").Or("role = ?", "super_admin").Find(&users)
// SELECT * FROM users WHERE role = 'admin' OR role = 'super_admin';

// Struct
db.Where("name = 'jinzhu'").Or(User{Name: "jinzhu 2", Age: 18}).Find(&users)
// SELECT * FROM users WHERE name = 'jinzhu' OR (name = 'jinzhu 2' AND age = 18
);

// Map
db.Where("name = 'jinzhu'").Or(map[string]interface{}{"name": "jinzhu 2", "age":
 18}).Find(&users)
// SELECT * FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2';
```

您还可以在高级查询中查看 Group 条件，更轻松地编写复杂 SQL

选择特定字段

选择您想从数据库中检索的字段，默认情况下会选择全部字段

```
db.Select("name", "age").Find(&users)
// SELECT name, age FROM users;

db.Select([]string{"name", "age"}).Find(&users)
// SELECT name, age FROM users;

db.Table("users").Select("COALESCE(age,?)", 42).Rows()
// SELECT COALESCE(age,'42') FROM users;
```

Order

指定从数据库检索记录时的排序方式

```
db.Order("age desc, name").Find(&users)
// SELECT * FROM users ORDER BY age desc, name;

// Multiple orders
db.Order("age desc").Order("name").Find(&users)
```

```
// SELECT * FROM users ORDER BY age desc, name;
```

Limit & Offset

`Limit` 指定获取记录的最大数量 `Offset` 指定在开始返回记录之前要跳过的记录数量

```
db.Limit(3).Find(&users)
// SELECT * FROM users LIMIT 3;

// 通过 -1 消除 Limit 条件
db.Limit(10).Find(&users1).Limit(-1).Find(&users2)
// SELECT * FROM users LIMIT 10; (users1)
// SELECT * FROM users; (users2)

db.Offset(3).Find(&users)
// SELECT * FROM users OFFSET 3;

db.Limit(10).Offset(5).Find(&users)
// SELECT * FROM users OFFSET 5 LIMIT 10;

// 通过 -1 消除 Offset 条件
db.Offset(10).Find(&users1).Offset(-1).Find(&users2)
// SELECT * FROM users OFFSET 10; (users1)
// SELECT * FROM users; (users2)
```

Group & Having

```
type result struct {
  Date  time.Time
  Total int
}

db.Model(&User{}).Select("name, sum(age) as total").Where("name LIKE ?", "group%").Group("name").First(&result)
// SELECT name, sum(age) as total FROM `users` WHERE name LIKE "group%" GROUP BY `name`


db.Model(&User{}).Select("name, sum(age) as total").Group("name").Having("name = ?", "group").Find(&result)
// SELECT name, sum(age) as total FROM `users` GROUP BY `name` HAVING name = "group"

rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) as total").Group("date(created_at)").Rows()
for rows.Next() {
  ...
```

```
}

rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) a
s total").Group("date(created_at)").Having("sum(amount) > ?", 100).Rows()
for rows.Next() {
  ...
}

type Result struct {
  Date  time.Time
  Total int64
}
db.Table("orders").Select("date(created_at) as date, sum(amount) as total").Gro
up("date(created_at)").Having("sum(amount) > ?", 100).Scan(&results)
```

Distinct

从模型中选择不相同的值

```
db.Distinct("name", "age").Order("name, age desc").Find(&results)
```

`Distinct` 也可以配合 `Pluck` , `Count` 使用

Joins

指定 Joins 条件

```
type result struct {
  Name  string
  Email string
}
db.Model(&User{}).Select("users.name, emails.email").Joins("left join emails on
 emails.user_id = users.id").Scan(&result{})
// SELECT users.name, emails.email FROM `users` left join emails on emails.user
_id = users.id

rows, err := db.Table("users").Select("users.name, emails.email").Joins("left j
oin emails on emails.user_id = users.id").Rows()
for rows.Next() {
  ...
}

db.Table("users").Select("users.name, emails.email").Joins("left join emails on
 emails.user_id = users.id").Scan(&results)

// 带参数的多表连接
db.Joins("JOIN emails ON emails.user_id = users.id AND emails.email = ?", "jinz
```

```
hu@example.org").Joins("JOIN credit_cards ON credit_cards.user_id = users.id").
Where("credit_cards.number = ?", "411111111111").Find(&user)
```

Joins 预加载

您可以使用 `Joins` 实现单条 SQL 预加载关联记录，例如：

```
db.Joins("Company").Find(&users)
// SELECT `users`.`id`,`users`.`name`,`users`.`age`,`Company`.`id` AS `Company_
_id`,`Company`.`name` AS `Company__name` FROM `users` LEFT JOIN `companies` AS
`Company` ON `users`.`company_id` = `Company`.`id`;
```

参考 预加载 了解详情

Scan

Scan 结果至 struct，用法与 `Find` 类似

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

var result Result
db.Table("users").Select("name", "age").Where("name = ?", "Antonio").Scan(&resu
lt)

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

高级查询

高级查询

智能选择字段

GORM 允许通过 `Select` 方法选择特定的字段，如果您在应用程序中经常使用此功能，你可以定义一个较小的 API 结构体，以实现自动选择特定的字段。

```
type User struct {
  ID     uint
  Name   string
  Age    int
  Gender string
  // 假设后面还有几百个字段...
}

type APIUser struct {
  ID   uint
  Name string
}

// 查询时会自动选择 `id`, `name` 字段
db.Model(&User{}).Limit(10).Find(&APIUser{})
// SELECT `id`, `name` FROM `users` LIMIT 10
```

Locking (FOR UPDATE)

GORM 支持多种类型的锁，例如：

```
DB.Clauses(clause.Locking{Strength: "UPDATE"}).Find(&users)
// SELECT * FROM `users` FOR UPDATE

DB.Clauses(clause.Locking{
  Strength: "SHARE",
  Table: clause.Table{Name: clause.CurrentTable},
}).Find(&users)
// SELECT * FROM `users` FOR SHARE OF `users`
```

参考 原生 SQL 及构造器 获取详情

子查询

子查询可以嵌套在查询中，GORM 允许在使用 `*gorm.DB` 对象作为参数时生成子查询

```
db.Where("amount > ?", db.Table("orders").Select("AVG(amount)")).Find(&orders)
// SELECT * FROM "orders" WHERE amount > (SELECT AVG(amount) FROM "orders");

subQuery := db.Select("AVG(age)").Where("name LIKE ?", "name%").Table("users")
db.Select("AVG(age) as avgage").Group("name").Having("AVG(age) > (?)", subQuery)
.Find(&results)
// SELECT AVG(age) as avgage FROM `users` GROUP BY `name` HAVING AVG(age) > (SE
LECT AVG(age) FROM `users` WHERE name LIKE "name%")
```

From 子查询

GORM 允许您在 `Table` 方法中通过 FROM 子句使用子查询，例如：

```
db.Table("(?) as u", DB.Model(&User{}).Select("name", "age")).Where("age = ?",
18}).Find(&User{})
// SELECT * FROM (SELECT `name`,`age` FROM `users`) as u WHERE `age` = 18

subQuery1 := DB.Model(&User{}).Select("name")
subQuery2 := DB.Model(&Pet{}).Select("name")
db.Table("(?) as u, (?) as p", subQuery1, subQuery2).Find(&User{})
// SELECT * FROM (SELECT `name` FROM `users`) as u, (SELECT `name` FROM `pets`)
 as p
```

Group 条件

使用 Group 条件可以更轻松的编写复杂 SQL

```
db.Where(
    DB.Where("pizza = ?", "pepperoni").Where(DB.Where("size = ?", "small").Or("
size = ?", "medium")),
).Or(
    DB.Where("pizza = ?", "hawaiian").Where("size = ?", "xlarge"),
).Find(&Pizza{}).Statement

// SELECT * FROM `pizzas` WHERE (pizza = "pepperoni" AND (size = "small" OR siz
e = "medium")) OR (pizza = "hawaiian" AND size = "xlarge")
```

命名参数

GORM 支持 `sql.NamedArg` 和 `map[string]interface{}{}` 形式的命名参数，例如：

```
DB.Where("name1 = @name OR name2 = @name", sql.Named("name", "jinzhu")).Find(&u
ser)
// SELECT * FROM `users` WHERE name1 = "jinzhu" OR name2 = "jinzhu"
```

```
DB.Where("name1 = @name OR name2 = @name", map[string]interface{}{"name": "jinz
hu"}).First(&user)
// SELECT * FROM `users` WHERE name1 = "jinzhu" OR name2 = "jinzhu" ORDER BY `u
sers`.`id` LIMIT 1
```

查看 原生 SQL 及构造器 获取详情

Find 至 map

GORM 允许扫描结果至 `map[string]interface{}` 或 `[]map[string]interface{}` ，此时别忘了指定 `Model` 或 `Table` ，例如：

```
var result map[string]interface{}
DB.Model(&User{}).First(&result, "id = ?", 1)

var results []map[string]interface{}
DB.Table("users").Find(&results)
```

FirstOrInit

获取第一条匹配的记录，或者根据给定的条件初始化一个 struct（仅支持 sturct 和 map 条件）

```
// 未找到 user，根据给定的条件初始化 struct
db.FirstOrInit(&user, User{Name: "non_existing"})
// user -> User{Name: "non_existing"}

// 找到了 `name` = `jinzhu` 的 user
db.Where(User{Name: "jinzhu"}).FirstOrInit(&user)
// user -> User{ID: 111, Name: "Jinzhu", Age: 18}

// 找到了 `name` = `jinzhu` 的 user
db.FirstOrInit(&user, map[string]interface{}{"name": "jinzhu"})
// user -> User{ID: 111, Name: "Jinzhu", Age: 18}
```

如果没有找到记录，可以使用包含更多的属性的结构体初始化 user， `Attrs` 不会被用于生成查询 SQL

```
// 未找到 user，则根据给定的条件以及 Attrs 初始化 user
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOrInit(&user)
// SELECT * FROM USERS WHERE name = 'non_existing' ORDER BY id LIMIT 1;
// user -> User{Name: "non_existing", Age: 20}

// 未找到 user，则根据给定的条件以及 Attrs 初始化 user
db.Where(User{Name: "non_existing"}).Attrs("age", 20).FirstOrInit(&user)
// SELECT * FROM USERS WHERE name = 'non_existing' ORDER BY id LIMIT 1;
```

```
// user -> User{Name: "non_existing", Age: 20}

// 找到了 `name` = `jinzhu` 的 user，则忽略 Attrs
db.Where(User{Name: "Jinzhu"}).Attrs(User{Age: 20}).FirstOrInit(&user)
// SELECT * FROM USERS WHERE name = jinzhu' ORDER BY id LIMIT 1;
// user -> User{ID: 111, Name: "Jinzhu", Age: 18}
```

不管是否找到记录， `Assign` 都会将属性赋值给 struct，但这些属性不会被用于生成查询 SQL

```
// 未找到 user，根据条件和 Assign 属性初始化 struct
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrInit(&user)
// user -> User{Name: "non_existing", Age: 20}

// 找到 `name` = `jinzhu` 的记录，依然会更新 Assign 相关的属性
db.Where(User{Name: "Jinzhu"}).Assign(User{Age: 20}).FirstOrInit(&user)
// SELECT * FROM USERS WHERE name = jinzhu' ORDER BY id LIMIT 1;
// user -> User{ID: 111, Name: "Jinzhu", Age: 20}
```

FirstOrCreate

获取第一条匹配的记录，或者根据给定的条件创建一条新纪录（仅支持 sturct 和 map 条件）

```
// 未找到 user，则根据给定条件创建一条新纪录
db.FirstOrCreate(&user, User{Name: "non_existing"})
// INSERT INTO "users" (name) VALUES ("non_existing");
// user -> User{ID: 112, Name: "non_existing"}

// 找到了 `name` = `jinzhu` 的 user
db.Where(User{Name: "jinzhu"}).FirstOrCreate(&user)
// user -> User{ID: 111, Name: "jinzhu", "Age}: 18
```

如果没有找到记录，可以使用包含更多的属性的结构体创建记录， `Attrs` 不会被用于生成查询 SQL 。

```
// 未找到 user，根据条件和 Assign 属性创建记录
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOrCreate(&user)
// SELECT * FROM users WHERE name = 'non_existing' ORDER BY id LIMIT 1;
// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
// user -> User{ID: 112, Name: "non_existing", Age: 20}

// 找到了 `name` = `jinzhu` 的 user，则忽略 Attrs
db.Where(User{Name: "jinzhu"}).Attrs(User{Age: 20}).FirstOrCreate(&user)
// SELECT * FROM users WHERE name = 'jinzhu' ORDER BY id LIMIT 1;
// user -> User{ID: 111, Name: "jinzhu", Age: 18}
```

不管是否找到记录， `Assign` 都会将属性赋值给 struct，并将结果写回数据库

```
// 未找到 user，根据条件和 Assign 属性创建记录
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrCreate(&user)
// SELECT * FROM users WHERE name = 'non_existing' ORDER BY id LIMIT 1;
// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
// user -> User{ID: 112, Name: "non_existing", Age: 20}

// 找到了 `name` = `jinzhu` 的 user，依然会根据 Assign 更新记录
db.Where(User{Name: "jinzhu"}).Assign(User{Age: 20}).FirstOrCreate(&user)
// SELECT * FROM users WHERE name = 'jinzhu' ORDER BY id LIMIT 1;
// UPDATE users SET age=20 WHERE id = 111;
// user -> User{ID: 111, Name: "jinzhu", Age: 20}
```

优化器、索引提示

优化器提示允许我们控制查询优化器选择某个查询执行计划。

```
import "gorm.io/hints"

DB.Clauses(hints.New("MAX_EXECUTION_TIME(10000)")).Find(&User{})
// SELECT * /*+ MAX_EXECUTION_TIME(10000) */ FROM `users`
```

索引提示允许传递索引提示到数据库，以防查询计划器出现混乱。

```
import "gorm.io/hints"

DB.Clauses(hints.UseIndex("idx_user_name")).Find(&User{})
// SELECT * FROM `users` USE INDEX (`idx_user_name`)

DB.Clauses(hints.ForceIndex("idx_user_name", "idx_user_id").ForJoin()).Find(&Us
er{})
// SELECT * FROM `users` FORCE INDEX FOR JOIN (`idx_user_name`,`idx_user_id`)"
```

参考 优化器提示、索引、备注 获取详情

迭代

GORM 支持通过行进行迭代

```
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Rows()
defer rows.Close()

for rows.Next() {
  var user User
  // ScanRows 将一行记录扫描至 user
  db.ScanRows(rows, &user)
```

```
  // 业务逻辑...
}
```

FindInBatches

用于批量查询并处理记录

```
// 每次批量处理 100 条
result := DB.Where("processed = ?", false).FindInBatches(&results, 100, func(tx
*gorm.DB, batch int) error {
  for _, result := range results {
    // 批量处理找到的记录
  }

  tx.Save(&results)

  tx.RowsAffected // 本次批量操作影响的记录数

  batch // Batch 1, 2, 3

  // 如果返回错误会终止后续批量操作
  return nil
})

result.Error // returned error
result.RowsAffected // 整个批量操作影响的记录数
```

查询钩子

对于查询操作，GORM 支持 `AfterFind` 钩子，查询记录后会调用它，详情请参考 钩子

```
func (u *User) AfterFind(tx *gorm.DB) (err error) {
  if u.Role == "" {
    u.Role = "user"
  }
  return
}
```

Pluck

Pluck 用于从数据库查询单个列，并将结果扫描到切片。如果您想要查询多列，您应该使用 `Scan`

```
var ages []int64
db.Find(&users).Pluck("age", &ages)
```

```
var names []string
db.Model(&User{}).Pluck("name", &names)

db.Table("deleted_users").Pluck("name", &names)

// Distinct Pluck
DB.Model(&User{}).Distinct().Pluck("Name", &names)
// SELECT DISTINCT `name` FROM `users`

// 超过一列的查询，应该使用 `Scan` 或者 `Find`，例如：
db.Select("name", "age").Scan(&users)
db.Select("name", "age").Find(&users)
```

Scopes

`Scopes` 允许你指定常用的查询，可以在调用方法时引用这些查询

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

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

func PaidWithCod(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode_sign = ?", "C")
}

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

db.Scopes(AmountGreaterThan1000, PaidWithCreditCard).Find(&orders)
// 查找所有金额大于 1000 的信用卡订单

db.Scopes(AmountGreaterThan1000, PaidWithCod).Find(&orders)
// 查找所有金额大于 1000 的货到付款订单

db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid", "shipped"})).Find(
&orders)
// 查找所有金额大于 1000 且已付款或已发货的订单
```

Count

Count 用于获取匹配的记录数

```
var count int64
db.Model(&User{}).Where("name = ?", "jinzhu").Or("name = ?", "jinzhu 2").Count(&
count)
// SELECT count(*) FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2'

db.Model(&User{}).Where("name = ?", "jinzhu").Count(&count)
// SELECT count(*) FROM users WHERE name = 'jinzhu'; (count)

db.Table("deleted_users").Count(&count)
// SELECT count(*) FROM deleted_users;

// 去重计数
DB.Model(&User{}).Distinct("name").Count(&count)
// SELECT COUNT(DISTINCT(`name`)) FROM `users`

db.Table("deleted_users").Select("count(distinct(name))").Count(&count)
// SELECT count(distinct(name)) FROM deleted_users

// 分组计数
users := []User{
  {Name: "name1"},
  {Name: "name2"},
  {Name: "name3"},
  {Name: "name3"},
}

DB.Model(&User{}).Group("name").Count(&count)
count // => 3
```

更新

更新

保存所有字段

`Save` 会保存所有的字段，即使字段是零值

```
db.First(&user)

user.Name = "jinzhu 2"
user.Age = 100
db.Save(&user)
// UPDATE users SET name='jinzhu 2', age=100, birthday='2016-01-01', updated_at
 = '2013-11-17 21:34:10' WHERE id=111;
```

Update/Updates

使用 `Update` 、 `Updates` 可以更新选定的字段

```
// 更新单个字段
// the user of `Model(&user)` needs to have primary key value, it is `111` in t
his example
db.Model(&user).Update("name", "hello")
// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111;


// 根据条件更新单个字段
db.Model(&user).Where("active = ?", true).Update("name", "hello")
// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111
 AND active=true;

// 通过 `struct` 更新多个字段，不会更新零值字段
db.Model(&user).Updates(User{Name: "hello", Age: 18, Active: false})
// UPDATE users SET name='hello', age=18, updated_at = '2013-11-17 21:34:10' WH
ERE id = 111;

// 通过 `map` 更新多个字段，零值字段也会更新
db.Model(&user).Updates(map[string]interface{}{"name": "hello", "age": 18, "act
ived": false})
// UPDATE users SET name='hello', age=18, actived=false, updated_at='2013-11-17
 21:34:10' WHERE id=111;
```

注意 当通过 struct 更新时，GORM 只会更新非零字段。 如果您想确保指定字段被更新，你应该使用

`Select` 更新选定字段，或使用 `map` 来完成更新操作

更新选定字段

如果您想要在更新时选定、忽略某些字段，您可以使用 `Select` 、 `Omit`

```
// Select 与 Map
// the user of `Model(&user)` needs to have primary key value, it is `111` in t
his example
db.Model(&user).Select("name").Updates(map[string]interface{}{"name": "hello",
"age": 18, "actived": false})
// UPDATE users SET name='hello' WHERE id=111;

db.Model(&user).Omit("name").Updates(map[string]interface{}{"name": "hello", "a
ge": 18, "actived": false})
// UPDATE users SET age=18, actived=false, updated_at='2013-11-17 21:34:10' WHE
RE id=111;

// Select 与 Struct
DB.Model(&result).Select("Name", "Age").Updates(User{Name: "new_name"})
// UPDATE users SET name='new_name', age=0 WHERE id=111;
```

更新钩子

对于更新操作，GORM 支持 `BeforeSave` 、 `BeforeUpdate` 、 `AfterSave` 、 `AfterUpdate` 钩子，这些方法将在更新记录时被调用，详情请参阅 钩子

```
func (u *User) BeforeUpdate(tx *gorm.DB) (err error) {
    if u.Role == "admin" {
        return errors.New("admin user not allowed to update")
    }
    return
}
```

批量更新

如果您尚未通过 `Model` 指定记录的主键，则 GORM 会执行批量更新

```
// 通过 struct 只能更新非零值，若要更新零值，可以使用 map[string]interface{}
db.Model(User{}).Where("role = ?", "admin").Updates(User{Name: "hello", Age: 18}
)
// UPDATE users SET name='hello', age=18 WHERE role = 'admin;

db.Table("users").Where("id IN (?)", []int{10, 11}).Updates(map[string]interface
{}{"name": "hello", "age": 18})
// UPDATE users SET name='hello', age=18 WHERE id IN (10, 11);
```

阻止全局更新

如果在没有任何条件的情况下执行批量更新，GORM 不会执行该操作，并返回 `ErrMissingWhereClause` 错误

您可以使用 `=` 之类的条件来强制全局更新

```
db.Model(&User{}).Update("name", "jinzhu").Error // gorm.ErrMissingWhereClause

db.Model(&User{}).Where("1 = 1").Update("name", "jinzhu")
// UPDATE users SET `name` = "jinzhu" WHERE 1=1
```

更新的记录数

```
// 通过 `RowsAffected` 得到更新的记录数
result := db.Model(User{}).Where("role = ?", "admin").Updates(User{Name: "hello"
, Age: 18})
// UPDATE users SET name='hello', age=18 WHERE role = 'admin;

result.RowsAffected // 更新的记录数
result.Error        // 更新的错误
```

高级用法

通过 SQL 表达式更新

GORM 允许通过 SQL 表达式更新列

```
DB.Model(&product).Update("price", gorm.Expr("price * ? + ?", 2, 100))
// UPDATE "products" SET "price" = price * '2' + '100', "updated_at" = '2013-11
-17 21:34:10' WHERE "id" = '2';

DB.Model(&product).Updates(map[string]interface{}{"price": gorm.Expr("price * ?
 + ?", 2, 100)})
// UPDATE "products" SET "price" = price * '2' + '100', "updated_at" = '2013-11
-17 21:34:10' WHERE "id" = '2';

DB.Model(&product).UpdateColumn("quantity", gorm.Expr("quantity - ?", 1))
// UPDATE "products" SET "quantity" = quantity - 1 WHERE "id" = '2';

DB.Model(&product).Where("quantity > 1").UpdateColumn("quantity", gorm.Expr("qu
antity - ?", 1))
// UPDATE "products" SET "quantity" = quantity - 1 WHERE "id" = '2' AND quantit
y > 1;
```

不使用钩子和时间追踪

如果您想在更新时跳过 `钩子` 方法和自动更新时间追踪，您可以使用 `UpdateColumn` 、`UpdateColumns`

```
// 更新单列，用法类似于 `Update`
db.Model(&user).UpdateColumn("name", "hello")
// UPDATE users SET name='hello' WHERE id = 111;

// 更新多列，用法类似于 `Updates`
db.Model(&user).UpdateColumns(User{Name: "hello", Age: 18})
// UPDATE users SET name='hello', age=18 WHERE id = 111;

// 配合 Select 更新多列，用法类似于 `Updates`
db.Model(&user).Select("name", "age").UpdateColumns(User{Name: "hello"})
// UPDATE users SET name='hello', age=0 WHERE id = 111;
```

检查字段是否有变更？

GORM 提供的 `Changed` 方法可以在 Before 钩子中检查字段是否有变更

`Changed` 方法只能与 `Update` 、`Updates` 方法一起使用，它只是检查 Model 对象字段的值与 `Update` 、`Updates` 的值是否相等，以及该字段是否会被更新（例如，可以通过 Select、Omit 排除某些字段），如果不相等，则返回 true，并更新记录

```
func (u *User) BeforeUpdate(tx *gorm.DB) (err error) {
  // 如果 role 字段有变更
    if tx.Statement.Changed("Role") {
    return errors.New("role not allowed to change")
    }

  if tx.Statement.Changed("Name", "Admin") { // 如果 Name 或 Role 字段有变更
    tx.Statement.SetColumn("Age", 18)
  }

  // 如果任意字段有变更
    if tx.Statement.Changed() {
        tx.Statement.SetColumn("RefreshedAt", time.Now())
    }
    return nil
}

db.Model(&User{ID: 1, Name: "jinzhu"}).Updates(map[string]interface{"name": "ji
nzhu2"})
// Changed("Name") => true
db.Model(&User{ID: 1, Name: "jinzhu"}).Updates(map[string]interface{"name": "ji
nzhu"})
```

```
// Changed("Name") => false, 因为 `Name` 没有变更
db.Model(&User{ID: 1, Name: "jinzhu"}).Select("Admin").Updates(map[string]inter
face{
  "name": "jinzhu2", "admin": false,
})
// Changed("Name") => false, 因为 `Name` 没有被 Select 选中并更新

db.Model(&User{ID: 1, Name: "jinzhu"}).Updates(User{Name: "jinzhu2"})
// Changed("Name") => true
db.Model(&User{ID: 1, Name: "jinzhu"}).Updates(User{Name: "jinzhu"})
// Changed("Name") => false, 因为 `Name` 没有变更
db.Model(&User{ID: 1, Name: "jinzhu"}).Select("Admin").Updates(User{Name: "jinz
hu2"})
// Changed("Name") => false, 因为 `Name` 没有被 Select 选中并更新
```

在更新时修改值

若要在 Before 钩子中改变要更新的值，如果它是一个完整的更新，可以使用 `Save` ；否则，应该使用 `scope.SetColumn` ，例如：

```
func (user *User) BeforeSave(scope *gorm.Scope) (err error) {
  if pw, err := bcrypt.GenerateFromPassword(user.Password, 0); err == nil {
    scope.SetColumn("EncryptedPassword", pw)
  }
}

db.Model(&user).Update("Name", "jinzhu")
```

删除

删除

删除记录

删除一条记录

```
// 删除一条已有的记录（email 的主键值为 10）
db.Delete(&email)
// DELETE from emails where id=10;

// 通过内联条件删除记录
db.Delete(&Email{}, 20)
// DELETE from emails where id=20;

// 带上其它条件
db.Where("name = ?", "jinzhu").Delete(&email)
// DELETE FROM emails WHERE id=10 AND name = 'jinzhu'
```

删除钩子

对于删除操作，GORM 支持 `BeforeDelete` 、 `AfterDelete` 钩子，在删除记录时会调用这些方法，详情请参考 钩子

```
func (u *User) BeforeDelete(tx *gorm.DB) (err error) {
    if u.Role == "admin" {
        return errors.New("admin user not allowed to delete")
    }
    return
}
```

批量删除

如果没有指定带有主键值的记录，GORM 将执行批量删除，删除所有匹配的记录

```
db.Where("email LIKE ?", "%jinzhu%").Delete(Email{})
// DELETE from emails where email LIKE "%jinzhu%";

db.Delete(Email{}, "email LIKE ?", "%jinzhu%")
// DELETE from emails where email LIKE "%jinzhu%";
```

阻止全局删除

如果在没有任何条件的情况下执行批量删除，GORM 不会执行该操作，并返回 `ErrMissingWhereClause` 错误

您可以使用 `=` 之类的条件来强制全局删除

```
db.Delete(&User{}).Error // gorm.ErrMissingWhereClause

db.Where("1 = 1").Delete(&User{})
// DELETE `users` WHERE 1=1
```

软删除

如果您的模型包含了一个 `gorm.deletedat` 字段（ `gorm.Model` 已经包含了该字段），它将自动获得软删除的能力！

拥有软删除能力的模型调用 `Delete` 时，记录不会被数据库。但 GORM 会将 `DeletedAt` 置为当前时间， 并且你不能再通过普通的查询方法找到该记录。

```
db.Delete(&user)
// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE id = 111;

// 批量删除
db.Where("age = ?", 20).Delete(&User{})
// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE age = 20;

// 在查询时会忽略被软删除的记录
db.Where("age = 20").Find(&user)
// SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;
```

如果您不想引入 `gorm.Model` ，您也可以这样启用软删除特性：

```
type User struct {
  ID      int
  Deleted gorm.DeletedAt
  Name    string
}
```

查找被软删除的记录

您可以使用 `Unscoped` 找到被软删除的记录

```
db.Unscoped().Where("age = 20").Find(&users)
// SELECT * FROM users WHERE age = 20;
```

永久删除

您也可以使用 `Unscoped` 永久删除匹配的记录

```
db.Unscoped().Delete(&order)
// DELETE FROM orders WHERE id=10;
```

原生SQL和SQL生成器

SQL 构建器

原生 SQL

原生 SQL 查询

```
type Result struct {
  ID   int
  Name string
  Age  int
}

var result Result
db.Raw("SELECT id, name, age FROM users WHERE name = ?", 3).Scan(&result)

db.Raw("SELECT id, name, age FROM users WHERE name = ?", 3).Scan(&result)

var age int
DB.Raw("select sum(age) from users where role = ?", "admin").Scan(&age)
```

执行原生 SQL

```
db.Exec("DROP TABLE users")
db.Exec("UPDATE orders SET shipped_at=? WHERE id IN ?", time.Now(), []int64{1,2,
3})

// SQL 表达式
DB.Exec("update users set money=? where name = ?", gorm.Expr("money * ? + ?", 1
0000, 1), "jinzhu")
```

注意 GORM 允许缓存准备好的语句来提高性能，详情请参考 性能

Row & Rows

获取 `*sql.Row` 结果

```
// 使用 GORM API 构建 SQL
row := db.Table("users").Where("name = ?", "jinzhu").Select("name", "age").Row()

row.Scan(&name, &age)
```

```
// 使用原生 SQL
row := db.Raw("select name, age, email from users where name = ?", "jinzhu").Row()
row.Scan(&name, &age, &email)
```

获取 `*sql.Rows` 结果

```
// 使用 GORM API 构建 SQL
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows()
defer rows.Close()
for rows.Next() {
  rows.Scan(&name, &age, &email)

  // 业务逻辑...
}

// 原生 SQL
rows, err := db.Raw("select name, age, email from users where name = ?", "jinzhu").Rows()
defer rows.Close()
for rows.Next() {
  rows.Scan(&name, &age, &email)

  // 业务逻辑...
}
```

转到 FindInBatches 获取如何在批量中查询和处理记录的信息， 转到 Group 条件 获取如何构建复杂 SQL 查询的信息

命名参数

GORM 支持 `sql.NamedArg` 或 `map[string]interface{}{}` 命名参数，例如：

```
DB.Where("name1 = @name OR name2 = @name", sql.Named("name", "jinzhu")).Find(&user)
// SELECT * FROM `users` WHERE name1 = "jinzhu" OR name2 = "jinzhu"

DB.Where("name1 = @name OR name2 = @name", map[string]interface{}{"name": "jinzhu2"}).First(&result3)
// SELECT * FROM `users` WHERE name1 = "jinzhu2" OR name2 = "jinzhu2" ORDER BY `users`.`id` LIMIT 1

DB.Raw("SELECT * FROM users WHERE name1 = @name OR name2 = @name2 OR name3 = @name", sql.Named("name", "jinzhu1"), sql.Named("name2", "jinzhu2")).Find(&user)
// SELECT * FROM users WHERE name1 = "jinzhu1" OR name2 = "jinzhu2" OR name3 =
```

```
"jinzhu1"

DB.Exec("UPDATE users SET name1 = @name, name2 = @name2, name3 = @name", sql.Named("name", "jinzhunew"), sql.Named("name2", "jinzhunew2"))
// UPDATE users SET name1 = "jinzhunew", name2 = "jinzhunew2", name3 = "jinzhunew"

DB.Raw("SELECT * FROM users WHERE (name1 = @name AND name3 = @name) AND name2 = @name2", map[string]interface{}{"name": "jinzhu", "name2": "jinzhu2"}).Find(&user)
// SELECT * FROM users WHERE (name1 = "jinzhu" AND name3 = "jinzhu") AND name2 = "jinzhu2"
```

将 `sql.Rows` 扫描至 struct

```
rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows() // (*sql.Rows, error)
defer rows.Close()

for rows.Next() {
  var user User
  // ScanRows 将一行扫描至 user
  db.ScanRows(rows, &user)

  // 业务逻辑...
}
```

DryRun 模式

在不执行的情况下生成 `SQL` ，可以用于准备或测试生成的 SQL，详情请参考 Session

```
stmt := DB.Session(&Session{DryRun: true}).First(&user, 1).Statement
stmt.SQL.String() //=> SELECT * FROM `users` WHERE `id` = $1 ORDER BY `id`
stmt.Vars         //=> []interface{}{1}
```

高级

Clauses

GORM 内部使用 SQL builder 生成 SQL。对于每个操作，GORM 都会创建一个 `*gorm.Statement` 对象，所有的 GORM API 都是在为 `statement` 添加/修改 `Clause` ，最后，GORM 会根据这些 Clause 生成 SQL

例如，当通过 `First` 进行查询时，它会在 `Statement` 中添加以下 Clause

```
clause.Select{Columns: "*"}
clause.From{Tables: clause.CurrentTable}
clause.Limit{Limit: 1}
clause.OrderByColumn{
  Column: clause.Column{Table: clause.CurrentTable, Name: clause.PrimaryKey},
}
```

然后 GORM 在回调中构建最终的查询 SQL，像这样：

```
Statement.Build("SELECT", "FROM", "WHERE", "GROUP BY", "ORDER BY", "LIMIT", "FO
R")
```

生成 SQL：

```
SELECT * FROM `users` ORDER BY `users`.`id` LIMIT 1
```

您可以自定义 `Clause` 并与 GORM 一起使用，这需要实现 Interface 接口

可以参考 示例

Clause 构建器

不同的数据库, Clause 可能会生成不同的 SQL，例如：

```
db.Offset(10).Limit(5).Find(&users)
// SQL Server 会生成
// SELECT * FROM "users" OFFSET 10 ROW FETCH NEXT 5 ROWS ONLY
// MySQL 会生成
// SELECT * FROM `users` LIMIT 5 OFFSET 10
```

之所以支持 Clause，是因为 GORM 允许数据库驱动程序通过注册 Clause Builder 来取代默认值，这儿有一个 Limit 的示例

Clause 选项

GORM 定义了很多 Clause，其中一些 Clause 提供了你可能会用到的选项

尽管很少会用到它们，但如果你发现 GORM API 与你的预期不符合。这可能可以很好地检查它们，例如：

```
DB.Clauses(clause.Insert{Modifier: "IGNORE"}).Create(&user)
// INSERT IGNORE INTO users (name,age...) VALUES ("jinzhu",18...);
```

StatementModifier

GORM 提供了 StatementModifier 接口，允许您修改语句，使其符合您的要求，这儿有一个 Hint 示例

```
import "gorm.io/hints"

DB.Clauses(hints.New("hint")).Find(&User{})
// SELECT * /*+ hint */ FROM `users`
```

关联

目前GormV2最佳实践后台管理系统 gin-vue-admin

- 基于gin+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

gf-vue-admin

- 基于goframe+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

导航

- Belongs To
- HasOne
- Has Many

- Many To Many
- 关联模式
- 预加载

Belongs To

Belongs To

Belongs To

`belongs to` 会与另一个模型建立了一对一的连接。 这种模型的每一个实例都“属于”另一个模型的一个实例。

例如，您的应用包含 user 和 company，并且每个 user 都可以分配给一个 company

```
// `User` 属于 `Company`, `CompanyID` 是外键
type User struct {
  gorm.Model
  Name      string
  CompanyID int
  Company   Company
}

type Company struct {
  ID   int
  Name string
}
```

重写外键

要定义一个 belongs to 关系，必须存在外键，默认的外键使用拥有者的类型名加上主字段名

对于上面例子，定义属于 `Company` 的 `User` ，其外键一般是 `CompanyID`

此外，GORM 还提供了一种自定义外键的方法，例如：

```
type User struct {
  gorm.Model
  Name         string
  CompanyRefer int
  Company      Company `gorm:"foreignKey:CompanyRefer"`
  // 使用 CompanyRefer 作为外键
}

type Company struct {
  ID   int
  Name string
}
```

重写引用

对于 belongs to 关系，GORM 通常使用拥有者的主字段作为外键的值。 对于上面的例子，它是 `Company` 的 `ID` 字段
当您将 user 分配给某个 company 时，GORM 会将 company 的 `ID` 保存到用户的 `CompanyID` 字段
此外，您也可以使用标签 `references` 手动更改它，例如：

```
type User struct {
  gorm.Model
  Name      string
  CompanyID string
  Company   Company `gorm:"references:Code"` // 使用 Code 作为引用
}

type Company struct {
  ID   int
  Code string
  Name string
}
```

Belongs to 的 CRUD

查看 关联模式 获取 belongs to 相关的用法

预加载

GORM 可以通过 `Preload` 、 `Joins` 预加载 belongs to 关联的记录，查看 预加载 获取详情

外键约束

你可以通过标签 `constraint` 并带上 `OnUpdate` 、 `OnDelete` 实现外键约束，例如：

```
type User struct {
  gorm.Model
  Name      string
  CompanyID int
  Company   Company `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
}

type Company struct {
  ID   int
  Name string
}
```

HasOne

Has One

Has One

`has one` 与另一个模型建立一对一的关联，但它和一对一关系有些许不同。 这种关联表明一个模型的每个实例都包含或拥有另一个模型的一个实例。

例如，您的应用包含 user 和 credit card 模型，且每个 user 只能有一张 credit card。

```
// User 有一张 CreditCard, CreditCardID 是外键
type User struct {
  gorm.Model
  CreditCard CreditCard
}

type CreditCard struct {
  gorm.Model
  Number string
  UserID uint
}
```

重写外键

对于 `has one` 关系，同样必须存在外键字段。拥有者将把属于它的模型的主键保存到这个字段。

这个字段的名称通常由 `has one` 模型的类型加上其 `主键` 生成，对于上面的例子，它是 `UserID` 。

为 user 添加 credit card 时，它会将 user 的 `ID` 保存到自己的 `UserID` 字段。

如果你想要使用另一个字段来保存该关系，你同样可以使用标签 `foreignKey` 来更改它，例如：

```
type User struct {
  gorm.Model
  CreditCard CreditCard `gorm:"foreignKey:UserName"`
  // 使用 UserName 作为外键
}

type CreditCard struct {
  gorm.Model
  Number   string
  UserName string
}
```

重写引用

默认情况下，拥有者实体会将 `has one` 对应模型的主键保存为外键，您也可以修改它，用另一个字段来保存，例如下个这个使用 `Name` 来保存的例子。
您可以使用标签 `references` 来更改它，例如：

```
type User struct {
  gorm.Model
  Name       string     `sql:"index"`
  CreditCard CreditCard `gorm:"foreignkey:UserName;references:name"`
}

type CreditCard struct {
  gorm.Model
  Number   string
  UserName string
}
```

多态关联

GORM 为 `has one` 和 `has many` 提供了多态关联支持，它会将拥有者实体的表名、主键都保存到多态类型的字段中。

```
type Cat struct {
  ID    int
  Name  string
  Toy   Toy `gorm:"polymorphic:Owner;"`
}

type Dog struct {
  ID   int
  Name string
  Toy  Toy `gorm:"polymorphic:Owner;"`
}

type Toy struct {
  ID        int
  Name      string
  OwnerID   int
  OwnerType string
}

db.Create(&Dog{Name: "dog1", Toy: Toy{Name: "toy1"}})
// INSERT INTO `dogs` (`name`) VALUES ("dog1")
// INSERT INTO `toys` (`name`,`owner_id`,`owner_type`) VALUES ("toy1","1","dogs
")
```

您可以使用标签 `polymorphicValue` 来更改多态类型的值，例如：

```
type Dog struct {
  ID   int
  Name string
  Toy  Toy `gorm:"polymorphic:Owner;polymorphicValue:master"`
}

type Toy struct {
  ID        int
  Name      string
  OwnerID   int
  OwnerType string
}

db.Create(&Dog{Name: "dog1", Toy: Toy{Name: "toy1"}})
// INSERT INTO `dogs` (`name`) VALUES ("dog1")
// INSERT INTO `toys` (`name`,`owner_id`,`owner_type`) VALUES ("toy1","1","master")
```

Has One 的 CURD

查看 关联模式 获取 `has one` 相关的用法

预加载

GORM 可以通过 `Preload` 、 `Joins` 预加载 `has one` 关联的记录，查看 预加载 获取详情

自引用 Has One

```
type User struct {
  gorm.Model
  Name      string
  ManagerID *uint
  Manager   *User
}
```

外键约束

你可以通过标签 `constraint` 并带上 `OnUpdate` 、 `OnDelete` 实现外键约束，例如：

```
type User struct {
  gorm.Model
  CreditCard CreditCard `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
}

type CreditCard struct {
  gorm.Model
```

```
    Number string
    UserID uint
}
```

Has Many

Has Many

Has Many

`has many` 与另一个模型建立了一对多的连接。 不同于 `has one` ，拥有者可以有零或多个关联模型。例如，您的应用包含 user 和 credit card 模型，且每个 user 可以有多张 credit card。

```
// User 有多张 CreditCard, UserID 是外键
type User struct {
  gorm.Model
  CreditCards []CreditCard
}

type CreditCard struct {
  gorm.Model
  Number string
  UserID uint
}
```

```
type User struct {
  gorm.Model
  CreditCards []CreditCard `gorm:"foreignKey:UserRefer"`
}

type CreditCard struct {
  gorm.Model
  Number    string
  UserRefer uint
}
```

```
type User struct {
  gorm.Model
  MemberNumber string
  CreditCards  []CreditCard `gorm:"foreignKey:UserNumber;references:MemberNumb
er"`
}

type CreditCard struct {
  gorm.Model
  Number     string
  UserNumber string
}
```

```
type Dog struct {
  ID   int
  Name string
  Toys []Toy `gorm:"polymorphic:Owner;"`
}

type Toy struct {
  ID        int
  Name      string
  OwnerID   int
  OwnerType string
}

db.Create(&Dog{Name: "dog1", Toy: []Toy{{Name: "toy1"}, {Name: "toy2"}}})
// INSERT INTO `dogs` (`name`) VALUES ("dog1")
// INSERT INTO `toys` (`name`,`owner_id`,`owner_type`) VALUES ("toy1","1","dog
s"), ("toy2","1","dogs")
```

```
type Dog struct {
  ID   int
  Name string
  Toys []Toy `gorm:"polymorphic:Owner;polymorphicValue:master"`
}

type Toy struct {
  ID        int
  Name      string
  OwnerID   int
  OwnerType string
}

db.Create(&Dog{Name: "dog1", Toy: []Toy{{Name: "toy1"}, {Name: "toy2"}}})
// INSERT INTO `dogs` (`name`) VALUES ("dog1")
// INSERT INTO `toys` (`name`,`owner_id`,`owner_type`) VALUES ("toy1","1","mas
ter"), ("toy2","1","master")
```

```
type User struct {
  gorm.Model
  Name      string
  ManagerID *uint
  Team      []User `gorm:"foreignkey:ManagerID"`
}
```

```
type User struct {
  gorm.Model
```

```
    CreditCards []CreditCard `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NUL
L;"`
  }

  type CreditCard struct {
    gorm.Model
    Number string
    UserID uint
  }
```

你可以通过标签 `constraint` 并带上 `OnUpdate` 、 `OnDelete` 实现外键约束，例如：

外键约束

自引用 Has Many

GORM 可以通过 `Preload` 预加载 has many 关联的记录，查看 预加载 获取详情

预加载

查看 关联模式 获取 has many 相关的用法

Has Many 的 CURD

您可以使用标签 `polymorphicValue` 来更改多态类型的值，例如：

GORM 为 `has one` 和 `has many` 提供了多态关联支持，它会将拥有者实体的表名、主键都保存到多态类型的字段中。

多态关联

同样的，您也可以使用标签 `references` 来更改它，例如：

为 user 添加 credit card 时，GORM 会将 user 的 `ID` 字段保存到 credit card 的 `UserID` 字段。

GORM 通常使用拥有者的主键作为外键的值。 对于上面的例子，它是 `User` 的 `ID` 字段。

重写引用

此外，想要使用另一个字段作为外键，您可以使用 `foreignKey` 标签自定义它：

例如，要定义一个属于 `User` 的模型，则其外键应该是 `UserID` 。

要定义 `has many` 关系，同样必须存在外键。 默认的外键名是拥有者的类型名加上其主键字段名

重写外键

Many To Many

Many To Many

Many To Many

Many to Many 会在两个 model 中添加一张连接表。

例如，您的应用包含了 user 和 language，且一个 user 可以说多种 language，多个 user 也可以说一种 language。

```
// User 拥有并属于多种 language, `user_languages` 是连接表
type User struct {
  gorm.Model
  Languages []Language `gorm:"many2many:user_languages;"`
}

type Language struct {
  gorm.Model
  Name string
}
```

当使用 GORM 的 `AutoMigrate` 为 `User` 创建表时，GORM 会自动创建连接表

反向引用

```
// User 拥有并属于多种 language, `user_languages` 是连接表
type User struct {
  gorm.Model
  Languages []*Language `gorm:"many2many:user_languages;"`
}

type Language struct {
  gorm.Model
  Name string
  Users []*User `gorm:"many2many:user_languages;"`
}
```

重写外键

对于 `many2many` 关系，连接表会同时拥有两个模型的外键，例如：

```
type User struct {
  gorm.Model
```

```
  Languages []Language `gorm:"many2many:user_languages;"`
}

type Language struct {
  gorm.Model
  Name string
}

// Join Table: user_languages
//   foreign key: user_id, reference: users.id
//   foreign key: language_id, reference: languages.id
```

若要重写它们，可以使用标签 `foreignKey` 、 `reference` 、 `joinforeignKey` 、 `joinReferences` 。当然，您不需要使用全部的标签，你可以仅使用其中的一个重写部分的外键、引用。

```
type User struct {
    gorm.Model
    Profiles []Profile `gorm:"many2many:user_profiles;foreignKey:Refer;joinFore
ignKey:UserReferID;References:UserRefer;JoinReferences:ProfileRefer"`
    Refer    uint
}

type Profile struct {
    gorm.Model
    Name      string
    UserRefer uint
}

// Which creates join table: user_profiles
//   foreign key: user_refer_id, reference: users.refer
//   foreign key: profile_refer, reference: profiles.user_refer
```

自引用 Many2Many

自引用 many2many 关系

```
type User struct {
  gorm.Model
    Friends []*User `gorm:"many2many:user_friends"`
}

// Which creates join table: user_friends
//   foreign key: user_id, reference: users.id
//   foreign key: friend_id, reference: users.id
```

预加载

GORM 允许通过 `Preload` 预加载 has many 关联的记录，查看 预加载 获取详情

Many2Many 的 CURD

查看 关联模式 获取 many2many 相关的用法

自定义连接表

`连接表` 可以是一个全功能的模型，支持 `Soft Delete` 、 `钩子` 、定义更多的字段，就跟其它模型一样。您可以通过 `SetupJoinTable` 指定它，例如：

```
type Person struct {
  ID        int
  Name      string
  Addresses []Address `gorm:"many2many:person_addresses;"`
}

type Address struct {
  ID   uint
  Name string
}

type PersonAddress struct {
  PersonID  int
  AddressID int
  CreatedAt time.Time
  DeletedAt gorm.DeletedAt
}

func (PersonAddress) BeforeCreate(db *gorm.DB) error {
  // ...
}

// PersonAddress 必须定义相关的的外键字段，否则会报错
db.SetupJoinTable(&Person{}, "Addresses", &PersonAddress{})
```

外键约束

你可以通过标签 `constraint` 并带上 `OnUpdate` 、 `OnDelete` 实现外键约束，例如：

```
type User struct {
  gorm.Model
  Languages []Language `gorm:"many2many:user_speaks;"`
}

type Language struct {
  Code string `gorm:"primarykey"`
```

```
  Name string
}

// CREATE TABLE `user_speaks` (`user_id` integer,`language_code` text,PRIMARY K
EY (`user_id`,`language_code`),CONSTRAINT `fk_user_speaks_user` FOREIGN KEY (`u
ser_id`) REFERENCES `users`(`id`) ON DELETE SET NULL ON UPDATE CASCADE,CONSTRAI
NT `fk_user_speaks_language` FOREIGN KEY (`language_code`) REFERENCES `language
s`(`code`) ON DELETE SET NULL ON UPDATE CASCADE);
```

复合外键

如果您的模型使用了 复合主键，GORM 会默认启用复合外键。

您也可以覆盖默认的外键、指定多个外键，只需用逗号分隔那些键名，例如：

```
type Tag struct {
  ID     uint   `gorm:"primaryKey"`
  Locale string `gorm:"primaryKey"`
  Value  string
}

type Blog struct {
  ID         uint   `gorm:"primaryKey"`
  Locale     string `gorm:"primaryKey"`
  Subject    string
  Body       string
  Tags       []Tag `gorm:"many2many:blog_tags;"`
  SharedTags []Tag `gorm:"many2many:shared_blog_tags;ForeignKey:id;References:i
d"`
  LocaleTags []Tag `gorm:"many2many:locale_blog_tags;ForeignKey:id,locale;Refer
ences:id"`
}

// Join Table: blog_tags
//   foreign key: blog_id, reference: blogs.id
//   foreign key: blog_locale, reference: blogs.locale
//   foreign key: tag_id, reference: tags.id
//   foreign key: tag_locale, reference: tags.locale

// Join Table: shared_blog_tags
//   foreign key: blog_id, reference: blogs.id
//   foreign key: tag_id, reference: tags.id

// Join Table: locale_blog_tags
//   foreign key: blog_id, reference: blogs.id
//   foreign key: blog_locale, reference: blogs.locale
//   foreign key: tag_id, reference: tags.id
```

还可以查看 复合主键

关联模式

实体关联

自动创建、更新

在创建、更新记录时，GORM 会通过 Upsert 自动保存关联及其引用记录。

```
user := User{
  Name:            "jinzhu",
  BillingAddress:  Address{Address1: "Billing Address - Address 1"},
  ShippingAddress: Address{Address1: "Shipping Address - Address 1"},
  Emails:          []Email{
    {Email: "jinzhu@example.com"},
    {Email: "jinzhu-2@example.com"},
  },
  Languages:       []Language{
    {Name: "ZH"},
    {Name: "EN"},
  },
}

db.Create(&user)
// BEGIN TRANSACTION;
// INSERT INTO "addresses" (address1) VALUES ("Billing Address - Address 1") ON
 DUPLICATE KEY DO NOTHING;
// INSERT INTO "addresses" (address1) VALUES ("Shipping Address - Address 1") O
N DUPLICATE KEY DO NOTHING;
// INSERT INTO "users" (name,billing_address_id,shipping_address_id) VALUES ("j
inzhu", 1, 2);
// INSERT INTO "emails" (user_id,email) VALUES (111, "jinzhu@example.com") ON D
UPLICATE KEY DO NOTHING;
// INSERT INTO "emails" (user_id,email) VALUES (111, "jinzhu-2@example.com") ON
 DUPLICATE KEY DO NOTHING;
// INSERT INTO "languages" ("name") VALUES ('ZH') ON DUPLICATE KEY DO NOTHING;
// INSERT INTO "user_languages" ("user_id","language_id") VALUES (111, 1) ON DU
PLICATE KEY DO NOTHING;
// INSERT INTO "languages" ("name") VALUES ('EN') ON DUPLICATE KEY DO NOTHING;
// INSERT INTO user_languages ("user_id","language_id") VALUES (111, 2) ON DUPL
ICATE KEY DO NOTHING;
// COMMIT;

db.Save(&user)
```

跳过自动创建、更新

若要在创建、更新时跳过自动保存，您可以使用 `Select` 或 `Omit` ，例如：

```
user := User{
  Name:            "jinzhu",
  BillingAddress:  Address{Address1: "Billing Address - Address 1"},
  ShippingAddress: Address{Address1: "Shipping Address - Address 1"},
  Emails:          []Email{
    {Email: "jinzhu@example.com"},
    {Email: "jinzhu-2@example.com"},
  },
  Languages:       []Language{
    {Name: "ZH"},
    {Name: "EN"},
  },
}

db.Select("Name").Create(&user)
// INSERT INTO "users" (name) VALUES ("jinzhu", 1, 2);

db.Omit("BillingAddress").Create(&user)
// 创建 user 时，跳过自动创建 BillingAddress

db.Omit(clause.Associations).Create(&user)
// 创建 user 时，跳过自动创建所有关联记录
```

关联模式

关联模式包含一些在处理关系时有用的方法

```
// 开始关联模式
var user User
db.Model(&user).Association("Languages")
// `user` 是源模型，它的主键不能为空
// 关系的字段名是 `Languages`
// 如果上面两个条件匹配，会开始关联模式，否则会返回错误
db.Model(&user).Association("Languages").Error
```

查找关联

查找所有匹配的关联记录

```
db.Model(&user).Association("Languages").Find(&languages)

// 带条件的查找
codes := []string{"zh-CN", "en-US", "ja-JP"}
db.Model(&user).Where("code IN ?", codes).Association("Languages").Find(&langua
ges)
```

添加关联

为 `to` 、 `has many` 添加新的关联；为 `has one` , `belongs to` 替换当前的关联

```
db.Model(&user).Association("Languages").Append([]Language{languageZH, languageEN})

db.Model(&user).Association("Languages").Append(Language{Name: "DE"})

db.Model(&user).Association("CreditCard").Append(CreditCard{Number: "411111111111"})
```

替换关联

用一个新的关联替换当前的关联

```
db.Model(&user).Association("Languages").Replace([]Language{languageZH, languageEN})

db.Model(&user).Association("Languages").Replace(Language{Name: "DE"}, languageEN)
```

删除关联

如果存在，则删除源模型与参数之间的关系，只会删除引用，不会从数据库中删除这些对象。

```
db.Model(&user).Association("Languages").Delete([]Language{languageZH, languageEN})
db.Model(&user).Association("Languages").Delete(languageZH, languageEN)
```

清空关联

删除源模型与关联之间的所有引用，但不会删除这些关联

```
db.Model(&user).Association("Languages").Clear()
```

关联计数

返回当前关联的计数

```
db.Model(&user).Association("Languages").Count()
```

批量处理数据

关联模式支持批量处理数据，例如：

```
// 查询所有用户的所有角色
db.Model(&users).Association("Role").Find(&roles)

// 将 userA 移出所有的 Team
db.Model(&users).Association("Team").Delete(&userA)

// 获取所有 Team 成员的不重复计数
db.Model(&users).Association("Team").Count()

// 对于 `Append`、`Replace` 的批量处理，参数与数据的长度必须相等，否则会返回错误
var users = []User{user1, user2, user3}
// 例如：我们有 3 个 user，将 userA 添加到 user1 的 Team，将 userB 添加到 user2 的 Team，将 userA、userB、userC 添加到 user3 的 Team
db.Model(&users).Association("Team").Append(&userA, &userB, &[]User{userA, userB, userC})
// 将 user1 的 Team 重置为 userA，将 user2的 team 重置为 userB，将 user3 的 team 重置为 userA、userB 和 userC
db.Model(&users).Association("Team").Replace(&userA, &userB, &[]User{userA, userB, userC})
```

关联标签

标签	描述
foreignKey	指定外键
references	指定引用
polymorphic	指定多态类型
polymorphicValue	指定多态值、默认表名
many2many	指定连接表表名
jointForeignKey	指定连接表的外键
joinReferences	指定连接表的引用外键
constraint	关系约束

预加载

预加载

预加载

GORM 允许在 `Preload` 的其它 SQL 中直接加载关系，例如：

```
type User struct {
  gorm.Model
  Username string
  Orders   []Order
}

type Order struct {
  gorm.Model
  UserID uint
  Price  float64
}

// 查找 user 时预加载相关 Order
db.Preload("Orders").Find(&users)
// SELECT * FROM users;
// SELECT * FROM orders WHERE user_id IN (1,2,3,4);

db.Preload("Orders").Preload("Profile").Preload("Role").Find(&users)
// SELECT * FROM users;
// SELECT * FROM orders WHERE user_id IN (1,2,3,4); // has many
// SELECT * FROM profiles WHERE user_id IN (1,2,3,4); // has one
// SELECT * FROM roles WHERE id IN (4,5,6); // belongs to
```

Joins 预加载

`Preload` 在一个单独查询中加载关联数据。而 `Join Preload` 会使用 inner join 加载关联数据，例如：

```
db.Joins("Company").Joins("Manager").Joins("Account").First(&user, 1)
db.Joins("Company").Joins("Manager").Joins("Account").First(&user, "users.name = ?", "jinzhu")
db.Joins("Company").Joins("Manager").Joins("Account").Find(&users, "users.id IN ?", []int{1,2,3,4,5})
```

注意 `Join Preload` 适用于一对一的关系，例如： `has one` , `belongs to`

预加载全部

与创建、更新时使用 `Select` 类似，`clause.Associations` 也可以和 `Preload` 一起使用，它可以用来 `预加载` 全部关联，例如：

```
type User struct {
  gorm.Model
  Name       string
  CompanyID  uint
  Company    Company
  Role       Role
}

db.Preload(clause.Associations).Find(&users)
```

带条件的预加载

GORM 允许带条件的 Preload 关联，类似于内联条件

```
// 带条件的预加载 Order
db.Preload("Orders", "state NOT IN (?)", "cancelled").Find(&users)
// SELECT * FROM users;
// SELECT * FROM orders WHERE user_id IN (1,2,3,4) AND state NOT IN ('cancelled
');

db.Where("state = ?", "active").Preload("Orders", "state NOT IN (?)", "cancelle
d").Find(&users)
// SELECT * FROM users WHERE state = 'active';
// SELECT * FROM orders WHERE user_id IN (1,2) AND state NOT IN ('cancelled');
```

自定义预加载 SQL

您可以通过 `func(db *gorm.DB) *gorm.DB` 实现自定义预加载 SQL，例如：

```
db.Preload("Orders", func(db *gorm.DB) *gorm.DB {
  return db.Order("orders.amount DESC")
}).Find(&users)
// SELECT * FROM users;
// SELECT * FROM orders WHERE user_id IN (1,2,3,4) order by orders.amount DESC;
```

嵌套预加载

GORM 支持嵌套预加载，例如：

```
db.Preload("Orders.OrderItems.Product").Preload("CreditCard").Find(&users)

// 自定义预加载 `Orders` 的条件
// 这样, GORM 就不会加载不匹配的 order 记录
db.Preload("Orders", "state = ?", "paid").Preload("Orders.OrderItems").Find(&us
ers)
```

教程

目前GormV2最佳实践后台管理系统 gin-vue-admin

- 基于gin+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

gf-vue-admin

- 基于goframe+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

导航

- 会话
- 钩子
- 事务
- 迁移
- Logger
- 常规数据库接口
- 性能
- 数据类型
- Scopes
- 约定
- 设置

Context

Context

GORM 通过 `WithContext` 方法提供了 Context 支持

单会话模式

单会话模式通常被用于执行单次操作

```
db.WithContext(ctx).Find(&users)
```

持续会话模式

持续会话模式通常被用于执行一系列操作，例如：

```
tx := db.WithContext(ctx)
tx.First(&user, 1)
tx.Model(&user).Update("role", "admin")
```

Chi 中间件示例

在处理 API 请求时持续会话模式会比较有用。例如，您可以在中间件中为 `*gorm.DB` 设置超时 Context，然后使用 `*gorm.DB` 处理所有请求

下面是一个 Chi 中间件的示例：

```
func SetDBMiddleware(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    timeoutContext, _ := context.WithTimeout(context.Background(), time.Second)
    ctx := context.WithValue(r.Context(), "DB", db.WithContext(timeoutContext))
    next.ServeHTTP(w, r.WithContext(ctx))
  })
}

r := chi.NewRouter()
r.Use(SetDBMiddleware)

r.Get("/", func(w http.ResponseWriter, r *http.Request) {
  db, ok := ctx.Value("DB").(*gorm.DB)

  var users []User
  db.Find(&users)
```

```
  // 以下省略 32 个 DB 操作...
})

r.Get("/user", func(w http.ResponseWriter, r *http.Request) {
  db, ok := ctx.Value("DB").(*gorm.DB)

  var user User
  db.First(&user)

  // 以下省略 32 个 DB 操作...
})
```

注意 通过 `WithContext` 设置的 `Context` 是线程安全的，参考会话获取详情

Logger

Logger 也可以接受 `Context` ，可用于日志追踪，请参考 Logger获取详情

处理错误

处理错误

在 Go 中，处理错误是很重要的。

我们鼓励您在调用任何 Finisher 方法 后，都进行错误检查

处理错误

GORM 的错误处理与常见的 Go 代码不同，因为 GORM 提供的是链式 API。

如果遇到任何错误，GORM 会设置 `*gorm.DB` 的 `Error` 字段，您需要像这样检查它：

```
if err := db.Where("name = ?", "jinzhu").First(&user).Error; err != nil {
  // 处理错误...
}
```

```
if result := db.Where("name = ?", "jinzhu").First(&user); result.Error != nil {

  // 处理错误...
}
```

```
// 检查错误是否为 RecordNotFound
err := db.First(&user, 100).Error
errors.Is(tx.Error, ErrRecordNotFound)
```

错误列表参考

Errors

当 `First` 、 `Last` 、 `Take` 方法找不到记录时，GORM 会返回 `ErrRecordNotFound` 错误。如果发生了多个错误，你可以通过 `errors.Is` 判断错误是否为 `ErrRecordNotFound` ，例如：

ErrRecordNotFound

或者

链式方法

链式方法

GORM 允许进行链式操作，所以您可以像这样写代码：

```
db.Where("name = ?", "jinzhu").Where("age = ?", 18).First(&user)
```

GORM 中有三种类型的方法： `链式方法` 、 `Finisher 方法` 、 `新建会话方法`

链式方法

链式方法是将 `Clauses` 修改或添加到当前 `Statement` 的方法，例如：

`Where` , `Select` , `Omit` , `Joins` , `Scopes` , `Preload` , `Raw` ...

这是 完整方法列表，也可以查看 SQL 构建器 获取更多关于 `Clauses` 的信息

Finisher Method

Finishers 是会立即执行注册回调的方法，然后生成并执行 SQL，比如这些方法：

`Create` , `First` , `Find` , `Take` , `Save` , `Update` , `Delete` , `Scan` , `Row` , `Rows` ...

查看完整方法列表

新建会话模式

在初始化了 `*gorm.DB` 或 `新建会话方法` 后， 调用下面的方法会创建一个新的 `Statement` 实例而不是使用当前的

GROM 定义了 `Session` 、 `WithContext` 、 `Debug` 方法做为 `新建会话方法` ，参考会话 获得详细

让我们用一些例子来解释它：

示例 1：

```
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})
// db 是一个刚完成初始化的 *gorm.DB 实例，这是一个 `新建会话`
db.Where("name = ?", "jinzhu").Where("age = ?", 18).Find(&users)
// `Where("name = ?", "jinzhu")` 是调用的第一个方法，它会创建一个新 `Statement`
// `Where("age = ?", 18)` 会复用 `Statement`，并将条件添加至这个 `Statement`
// `Find(&users)` 是一个 finisher 方法，它运行注册的查询回调，生成并运行下面这条 SQL：
// SELECT * FROM users WHERE name = 'jinzhu' AND age = 18;

db.Where("name = ?", "jinzhu2").Where("age = ?", 20).Find(&users)
// `Where("name = ?", "jinzhu2")` 也是调用的第一个方法，也会创建一个新 `Statement`
// `Where("age = ?", 20)` 会复用 `Statement`，并将条件添加至这个 `Statement`
// `Find(&users)` 是一个 finisher 方法，它运行注册的查询回调，生成并运行下面这条 SQL：
```

```
// SELECT * FROM users WHERE name = 'jinzhu2' AND age = 20;

db.Find(&users)
// 对于这个 `新建会话模式` 的 `*gorm.DB` 实例来说，`Find(&users)` 是一个 finisher 方法
也是第一个调用的方法。
// 它创建了一个新的 `Statement` 运行注册的查询回调，生成并运行下面这条 SQL：
// SELECT * FROM users;
```

示例 2：

```
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})
// db 是一个刚完成初始化的 *gorm.DB 实例，这是一个 `新建会话`
tx := db.Where("name = ?", "jinzhu")
// `Where("name = ?", "jinzhu")` 是第一个被调用的方法，它创建了一个新的 `Statement`
并添加条件

tx.Where("age = ?", 18).Find(&users)
// `tx.Where("age = ?", 18)` 会复用上面的那个 `Statement`，并向其添加条件
// `Find(&users)` 是一个 finisher 方法，它运行注册的查询回调，生成并运行下面这条 SQL：
// SELECT * FROM users WHERE name = 'jinzhu' AND age = 18

tx.Where("age = ?", 28).Find(&users)
// `tx.Where("age = ?", 18)` 同样会复用上面的那个 `Statement`，并向其添加条件
// `Find(&users)` 是一个 finisher 方法，它运行注册的查询回调，生成并运行下面这条 SQL：
// SELECT * FROM users WHERE name = 'jinzhu' AND age = 18 AND age = 20;
```

注意 在示例 2 中，第一个查询会影响第二个查询生成的 SQL ，因为 GORM 复用 `Statement` 这可能会引发预期之外的问题，请参考 线程安全 了解如何避免该问题。

线程安全

新初始化的 `*gorm.DB` 或调用 `新建会话方法` 后，GORM 会创建新的 `Statement` 实例。因此想要复用 `*gorm.DB` ，您需要确保它们处于 `新建会话模式` ，例如：

```
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})

// 安全的使用新初始化的 *gorm.DB
for i := 0; i < 100; i++ {
  go db.Where(...).First(&user)
}

tx := db.Where("name = ?", "jinzhu")
// 不安全的复用 Statement
for i := 0; i < 100; i++ {
  go tx.Where(...).First(&user)
}
```

```go
ctx, _ := context.WithTimeout(context.Background(), time.Second)
ctxDB := db.WithContext(ctx)
// 在 `新建会话方法` 之后是安全的
for i := 0; i < 100; i++ {
  go ctxDB.Where(...).First(&user)
}

ctx, _ := context.WithTimeout(context.Background(), time.Second)
ctxDB := db.Where("name = ?", "jinzhu").WithContext(ctx)
// 在 `新建会话方法` 之后是安全的
for i := 0; i < 100; i++ {
  go ctxDB.Where(...).First(&user) // `name = 'jinzhu'` 会应用到每次循环中
}

tx := db.Where("name = ?", "jinzhu").Session(&gorm.Session{WithConditions: true}
)
// 在 `新建会话方法` 之后是安全的
for i := 0; i < 100; i++ {
  go tx.Where(...).First(&user) // `name = 'jinzhu'` 会应用到每次循环中
}
```

会话

会话

GORM 提供了 `Session` 方法，这是一个 新建会话方法 ，它允许创建带配置的新建会话模式：

```
// 会话配置
type Session struct {
  DryRun         bool
  PrepareStmt    bool
  WithConditions bool
  Context        context.Context
  Logger         logger.Interface
  NowFunc        func() time.Time
}
```

DryRun

DarRun 模式会生成但不执行 `SQL` ，可以用于准备或测试生成的 SQL，详情请参考 Session：

```
// 新建会话模式
stmt := db.Session(&Session{DryRun: true}).First(&user, 1).Statement
stmt.SQL.String() //=> SELECT * FROM `users` WHERE `id` = $1 ORDER BY `id`
stmt.Vars         //=> []interface{}{1}

// 全局 DryRun 模式
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{DryRun: true})

// 不同的数据库生成不同的 SQL
stmt := db.Find(&user, 1).Statement
stmt.SQL.String() //=> SELECT * FROM `users` WHERE `id` = $1 // PostgreSQL
stmt.SQL.String() //=> SELECT * FROM `users` WHERE `id` = ?  // MySQL
stmt.Vars         //=> []interface{}{1}
```

PrepareStmt

`PreparedStmt` 在执行任何 SQL 时都会创建一个 prepared statement 并将其缓存，以提高后续的效率，例如：

```
// 全局模式，所有 DB 操作都会 创建并缓存 prepared stmt
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  PrepareStmt: true,
})
```

```
// 会话模式
tx := db.Session(&Session{PrepareStmt: true})
tx.First(&user, 1)
tx.Find(&users)
tx.Model(&user).Update("Age", 18)

// returns prepared statements manager
stmtManger, ok := tx.ConnPool.(*PreparedStmtDB)

// 关闭 *当前会话* 的 prepared statements
stmtManger.Close()

// 为 *当前会话* prepared SQL
stmtManger.PreparedSQL

// 开启当前数据库连接池（所有会话）的 prepared statements
stmtManger.Stmts // map[string]*sql.Stmt

for sql, stmt := range stmtManger.Stmts {
  sql  // prepared SQL
  stmt // prepared statement
  stmt.Close() // 关闭 prepared statement
}
```

WithConditions

`WithCondition` 会共享 `*gorm.DB` 的条件，例如：

```
tx := db.Where("name = ?", "jinzhu").Session(&gorm.Session{WithConditions: true}
)

tx.First(&user)
// SELECT * FROM users WHERE name = "jinzhu" ORDER BY id

tx.First(&user, "id = ?", 10)
// SELECT * FROM users WHERE name = "jinzhu" AND id = 10 ORDER BY id

// 不共享 `WithConditions`
tx2 := db.Where("name = ?", "jinzhu").Session(&gorm.Session{WithConditions: fal
se})
tx2.First(&user)
// SELECT * FROM users ORDER BY id
```

Context

`Context`，您可以通过 `Context` 来追踪 SQL 操作，例如：

```
timeoutCtx, _ := context.WithTimeout(context.Background(), time.Second)
tx := db.Session(&Session{Context: timeoutCtx})

tx.First(&user) // 带 timeoutCtx 的查询
tx.Model(&user).Update("role", "admin") // 带 timeoutCtx 的更新
```

GORM 也提供快捷调用方法 `WithContext` ，其实现如下：

```
func (db *DB) WithContext(ctx context.Context) *DB {
  return db.Session(&Session{WithConditions: true, Context: ctx})
}
```

Logger

Gorm 允许使用 `Logger` 选项自定义内建 Logger，例如：

```
newLogger := logger.New(log.New(os.Stdout, "\r\n", log.LstdFlags),
              logger.Config{
                SlowThreshold: time.Second,
                LogLevel:      logger.Silent,
                Colorful:      false,
              })
db.Session(&Session{Logger: newLogger})

db.Session(&Session{Logger: logger.Default.LogMode(logger.Silent)})
```

查看 Logger 获取详情

NowFunc

`NowFunc` 允许改变 GORM 获取当前时间的实现，例如：

```
db.Session(&Session{
  NowFunc: func() time.Time {
    return time.Now().Local()
  },
})
```

Debug

`Debug` 只是将会话的 `Logger` 修改为调试模式的快捷方法，其实现如下：

```
func (db *DB) Debug() (tx *DB) {
  return db.Session(&Session{
```

```
    WithConditions: true,
    Logger:         db.Logger.LogMode(logger.Info),
  })
}
```

钩子

钩子

对象生命周期

钩子是在创建、查询、更新、删除等操作之前、之后调用的函数。

如果您已经为模型定义了指定的方法，它会在创建、更新、查询、删除时自动被调用。如果任何回调返回错误，GORM 将停止后续的操作并回滚事务。

钩子方法的函数签名应该是 `func(*gorm.DB) error`

钩子

创建对象

创建时可用的钩子

```
// 开始事务
BeforeSave
BeforeCreate
// 关联前的 save
// 插入记录至 db
// 关联后的 save
AfterCreate
AfterSave
// 提交或回滚事务
```

代码示例：

```
func (u *User) BeforeCreate(tx *gorm.DB) (err error) {
  u.UUID = uuid.New()

  if !u.IsValid() {
    err = errors.New("can't save invalid data")
  }
  return
}

func (u *User) AfterCreate(tx *gorm.DB) (err error) {
  if u.ID == 1 {
    tx.Model(u).Update("role", "admin")
  }
  return
}
```

注意 在 GORM 中保存、删除操作会默认运行在事务上， 因此在事务完成之前该事务中所作的更改是不可见的，如果您的钩子返回了任何错误，则修改将被回滚。

```
func (u *User) AfterCreate(tx *gorm.DB) (err error) {
  if !u.IsValid() {
    return errors.New("rollback invalid user")
  }
  return nil
}
```

更新对象

更新时可用的钩子

```
// 开始事务
BeforeSave
BeforeUpdate
// 关联前的 save
// 更新 db
// 关联后的 save
AfterUpdate
AfterSave
// 提交或回滚事务
```

代码示例：

```
func (u *User) BeforeUpdate(tx *gorm.DB) (err error) {
  if u.readonly() {
    err = errors.New("read only user")
  }
  return
}

// 在同一个事务中更新数据
func (u *User) AfterUpdate(tx *gorm.DB) (err error) {
  if u.Confirmed {
    tx.Model(&Address{}).Where("user_id = ?", u.ID).Update("verfied", true)
  }
  return
}
```

删除对象

更新时可用的钩子

```
// 开始事务
BeforeDelete
// 删除 db 中的数据
AfterDelete
// 提交或回滚事务
```

代码示例：

```
// 在同一个事务中更新数据
func (u *User) AfterDelete(tx *gorm.DB) (err error) {
  if u.Confirmed {
    tx.Model(&Address{}).Where("user_id = ?", u.ID).Update("invalid", false)
  }
  return
}
```

查询对象

更新时可用的钩子

```
// 从 db 中加载数据
// Preloading (eager loading)
AfterFind
```

代码示例：

```
func (u *User) AfterFind(tx *gorm.DB) (err error) {
  if u.MemberShip == "" {
    u.MemberShip = "user"
  }
  return
}
```

修改当前操作

```
func (u *User) BeforeCreate(tx *gorm.DB) error {
  // 通过 tx.Statement 修改当前操作，例如：
  tx.Statement.Select("Name", "Age")
  tx.Statement.AddClause(clause.OnConflict{DoNothing: true})

  // 在没有 `WithConditions` 参数的清空下，tx 是一个新建会话模式
  // 基于 tx 的操作会在同一个事务中，但不会带上任何当前的条件
  var role Role
  err := tx.First(&role, "name = ?", user.Role).Error
  // SELECT * FROM roles WHERE name = "admin"
```

```
    // ...
    return err
}
```

事务

事务

禁用默认事务

为了确保数据一致性，GORM 会在事务里执行写入操作（创建、更新、删除）。如果没有这方面的要求，您可以在初始化时禁用它，这将获得大约 30%+ 性能提升。

```
// 全局禁用
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  SkipDefaultTransaction: true,
})

// 持续会话模式
tx := db.Session(&Session{SkipDefaultTransaction: true})
tx.First(&user, 1)
tx.Find(&users)
tx.Model(&user).Update("Age", 18)
```

事务

要在事务中执行一系列操作，一般流程如下：

```
db.Transaction(func(tx *gorm.DB) error {
  // 在事务中执行一些 db 操作（从这里开始，您应该使用 'tx' 而不是 'db'）
  if err := tx.Create(&Animal{Name: "Giraffe"}).Error; err != nil {
    // 返回任何错误都会回滚事务
    return err
  }

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

  // 返回 nil 提交事务
  return nil
})
```

嵌套事务

GORM 支持嵌套事务，您可以回滚较大事务内执行的一部分操作，例如：

```
DB.Transaction(func(tx *gorm.DB) error {
```

```
  tx.Create(&user1)

  tx.Transaction(func(tx2 *gorm.DB) error {
    tx2.Create(&user2)
    return errors.New("rollback user2") // 回滚 user2
  })

  tx.Transaction(func(tx2 *gorm.DB) error {
    tx2.Create(&user3)
    return nil
  })

  return nil
})

// 仅提交 user1, user3
```

手动事务

```
// 开始事务
tx := db.Begin()

// 在事务中执行一些 db 操作（从这里开始，您应该使用 'tx' 而不是 'db'）
tx.Create(...)

// ...

// 遇到错误时回滚事务
tx.Rollback()

// 否则，提交事务
tx.Commit()
```

一个特殊的示例

```
func CreateAnimals(db *gorm.DB) error {
  // 再唠叨一下，事务一旦开始，你就应该使用 tx 处理数据
  tx := db.Begin()
  defer func() {
    if r := recover(); r != nil {
      tx.Rollback()
    }
  }()

  if err := tx.Error; err != nil {
    return err
  }
```

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

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

  return tx.Commit().Error
}
```

SavePoint、RollbackTo

GORM 提供了 `SavePoint` 、 `Rollbackto` 来提供保存点以及回滚至保存点，例如：

```
tx := DB.Begin()
tx.Create(&user1)

tx.SavePoint("sp1")
tx.Create(&user2)
tx.RollbackTo("sp1") // 回滚 user2

tx.Commit() // 最终仅提交 user1
```

迁移

迁移

AutoMigrate

AutoMigrate 用于自动迁移您的 schema，保持您的 schema 是最新的。

注意：AutoMigrate* 只会创建表，它会忽略外键、约束、列和索引。为了保护您的数据，它不会*更改现有列的类型或删除未使用的列。

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

db.AutoMigrate(&User{}, &Product{}, &Order{})

// 创建表时添加后缀
db.Set("gorm:table_options", "ENGINE=InnoDB").AutoMigrate(&User{})
```

注意 AutoMigrate 会自动创建数据库外键约束，您可以在初始化时禁用此功能，例如：

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  DisableForeignKeyConstraintWhenMigrating: true,
})
```

Migrator 接口

GORM 提供了 Migrator 接口，该接口为每个数据库提供了统一的 API 接口，可用来为您的数据库构建独立迁移，例如：

SQLite 不支持 `ALTER COLUMN` 、 `DROP COLUMN` ，当你试图修改表结构，GORM 将创建一个新表、复制所有数据、删除旧表、重命名新表。

一些版本的 MySQL 不支持 rename 列，索引。GORM 将基于您使用 MySQL 的版本执行不同 SQL

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

  // Database
  CurrentDatabase() string
  FullDataTypeOf(*schema.Field) clause.Expr

  // Tables
  CreateTable(dst ...interface{}) error
  DropTable(dst ...interface{}) error
```

```
  HasTable(dst interface{}) bool
  RenameTable(oldName, newName interface{}) 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)

  // 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
}
```

当前数据库

返回当前使用的数据库名

```
db.Migrator().CurrentDatabase()
```

表

```
// 为 `User` 创建表
db.Migrator().CreateTable(&User{})

// 将 "ENGINE=InnoDB" 添加到创建 `User` 的 SQL 里去
db.Set("gorm:table_options", "ENGINE=InnoDB").CreateTable(&User{})

// 检查 `User` 对应的表是否存在
db.Migrator().HasTable(&User{})
db.Migrator().HasTable("users")

// 如果存在表则删除（删除时会忽略、删除外键约束）
db.Migrator().DropTable(&User{})
db.Migrator().DropTable("users")

// 重命名表
db.Migrator().RenameTable(&User{}, &UserInfo{})
db.Migrator().RenameTable("users", "user_infos")
```

列

```
type User struct {
  Name string
}

// 添加 name 字段
db.Migrator().AddColumn(&User{}, "Name")
// 删除 name 字段
db.Migrator().DropColumn(&User{}, "Name")
// 修改 name 字段
db.Migrator().AlterColumn(&User{}, "Name")
// 检查字段是否存在
db.Migrator().HasColumn(&User{}, "Name")

type User struct {
  Name    string
  NewName string
}

// 重命名字段
db.Migrator().RenameColumn(&User{}, "Name", "NewName")
db.Migrator().RenameColumn(&User{}, "name", "new_name")

// 获取字段类型
db.Migrator().ColumnTypes(&User{}) ([]*sql.ColumnType, error)
```

约束

```
type UserIndex struct {
  Name  string `gorm:"check:name_checker,name <> 'jinzhu'"`
}

// 创建约束
db.Migrator().CreateConstraint(&User{}, "name_checker")

// 删除约束
db.Migrator().DropConstraint(&User{}, "name_checker")

// 检查约束是否存在
db.Migrator().HasConstraint(&User{}, "name_checker")
```

索引

```
type User struct {
  gorm.Model
```

```
  Name string `gorm:"size:255;index:idx_name,unique"`
}

// 为 Name 字段创建索引
db.Migrator().CreateIndex(&User{}, "Name")
db.Migrator().CreateIndex(&User{}, "idx_name")

// 为 Name 字段删除索引
db.Migrator().DropIndex(&User{}, "Name")
db.Migrator().DropIndex(&User{}, "idx_name")

// 检查索引是否存在
db.Migrator().HasIndex(&User{}, "Name")
db.Migrator().HasIndex(&User{}, "idx_name")

type User struct {
  gorm.Model
  Name  string `gorm:"size:255;index:idx_name,unique"`
  Name2 string `gorm:"size:255;index:idx_name_2,unique"`
}
// 修改索引名
db.Migrator().RenameIndex(&User{}, "Name", "Name2")
db.Migrator().RenameIndex(&User{}, "idx_name", "idx_name_2")
```

约束

GORM 会在自动迁移和创建表时创建约束，查看 约束 或 数据库索引 获取详情

其他迁移工具

GORM 的 AutoMigrate 在大多数情况下都工作得很好，但如果您正在寻找更严格的迁移工具，GORM 提供一个通用数据库接口，可能对您有帮助。

```
// returns `*sql.DB`
db.DB()
```

查看 通用接口 获取详情。

Logger

Logger

Logger

Gorm 有一个 默认 logger 实现，默认情况下，它会打印慢 SQL 和错误

Logger 接受的选项不多，您可以在初始化时自定义它，例如：

```
newLogger := logger.New(
  log.New(os.Stdout, "\r\n", log.LstdFlags), // io writer
  logger.Config{
    SlowThreshold: time.Second,   // 慢 SQL 阈值
    LogLevel:      logger.Silent, // Log level
    Colorful:      false,         // 禁用彩色打印
  },
)

// 全局模式
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{
  Logger: newLogger,
})

// 新建会话模式
tx := db.Session(&Session{Logger: newLogger})
tx.First(&user)
tx.Model(&user).Update("Age", 18)

// Debug 单个操作，会将该会话的日志级别调整为 logger.Info
db.Debug().Where("name = ?", "jinzhu").First(&User{})
```

日志级别

GORM 定义了这些日志级别： `Silent` 、 `Error` 、 `Warn` 、 `Info`

```
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{
  Logger: logger.Default.LogMode(logger.Silent),
})
```

自定义 Logger

您可用参考 GORM 的 默认 logger 来定义您自己的 logger

Logger 需要实现以下接口，它接受 `context` ，所以你可以用它来追踪日志

```
type Interface interface {
    LogMode(LogLevel) Interface
    Info(context.Context, string, ...interface{})
    Warn(context.Context, string, ...interface{})
    Error(context.Context, string, ...interface{})
    Trace(ctx context.Context, begin time.Time, fc func() (string, int64), err
error)
}
```

常规数据库接口

常规数据库接口 sql.DB

GORM 提供了 `DB` 方法，可用于从当前 `*gorm.DB` 返回一个通用的数据库接口 *sql.DB

```
// 获取通用数据库对象 sql.DB, 然后使用其提供的功能
sqlDB, err := db.DB()

// Ping
sqlDB.Ping()

// Close
sqlDB.Close()

// 返回数据库统计信息
sqlDB.Stats()
```

注意 如果底层连接的数据库不是 `*sql.DB` ，它会返回错误

连接池

```
// 获取通用数据库对象 sql.DB , 然后使用其提供的功能
sqlDB, err := db.DB()

// SetMaxIdleConns 用于设置连接池中空闲连接的最大数量。
sqlDB.SetMaxIgleConns(10)

// SetMaxOpenConns 设置打开数据库连接的最大数量。
sqlDB.SetMaxOpenConns(100)

// SetConnMaxLifetime 设置了连接可复用的最大时间。
sqlDB.SetConnMaxLifetime(time.Hour)
```

性能

性能

GORM 已经优化了许多东西来提高性能，其默认性能对大多数应用来说都够用了。但这里还是有一些关于如何为您的应用改进性能的方法。

禁用默认事务

对于写操作（创建、更新、删除），为了确保数据的完整性，GORM 会将它们封装在事务内运行。但这会降低性能，你可以在初始化时禁用这种方式

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  SkipDefaultTransaction: true,
})
```

缓存 Prepared Statement

执行任何 SQL 时都创建 prepared statement 并缓存，可以提高后续的调用速度

```
// 全局模式
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  PrepareStmt: true,
})

// 会话模式
tx := db.Session(&Session{PrepareStmt: true})
tx.First(&user, 1)
tx.Find(&users)
tx.Model(&user).Update("Age", 18)
```

带 PreparedStmt 的 SQL 生成器

Prepared Statement 也可以和原生 SQL 一起使用，例如：

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  PrepareStmt: true,
})

db.Raw("select sum(age) from users where role = ?", "admin").Scan(&age)
```

您也可以使用 GORM 的 API DryRun 模式 编写 SQL 并执行 prepared statement ，查看 会话模式 获取详情

选择字段

默认情况下，GORM 在查询时会选择所有的字段，您可以使用 `Select` 来指定您想要的字段

```
db.Select("Name", "Age").Find(&Users{})
```

或者定义一个较小的 API 结构体，使用 智能选择字段功能

```
type User struct {
  ID     uint
  Name   string
  Age    int
  Gender string
  // 假设后面还有几百个字段...
}

type APIUser struct {
  ID   uint
  Name string
}

// 查询时会自动选择 `id`、`name` 字段
db.Model(&User{}).Limit(10).Find(&APIUser{})
// SELECT `id`, `name` FROM `users` LIMIT 10
```

迭代、FindInBatches

用迭代或 in batches 查询并处理记录

索引提示

索引 用于提高数据检索和 SQL 查询性能。 `索引提示` 向优化器提供了在查询处理过程中如何选择索引的信息。与 optimizer 相比，它可以更灵活地选择更有效的执行计划

```
import "gorm.io/hints"

DB.Clauses(hints.UseIndex("idx_user_name")).Find(&User{})
// SELECT * FROM `users` USE INDEX (`idx_user_name`)

DB.Clauses(hints.ForceIndex("idx_user_name", "idx_user_id").ForJoin()).Find(&Us
er{})
// SELECT * FROM `users` FORCE INDEX FOR JOIN (`idx_user_name`,`idx_user_id`)"

DB.Clauses(
    hints.ForceIndex("idx_user_name", "idx_user_id").ForOrderBy(),
    hints.IgnoreIndex("idx_user_name").ForGroupBy(),
```

```
).Find(&User{})
// SELECT * FROM `users` FORCE INDEX FOR ORDER BY (`idx_user_name`,`idx_user_id
`) IGNORE INDEX FOR GROUP BY (`idx_user_name`)"
```

数据类型

数据类型

GORM 提供了少量接口，使用户能够为 GORM 定义支持的数据类型，这里以 json 为例

实现数据类型

Scanner / Valuer

自定义的数据类型必须实现 Scanner 和 Valuer 接口，以便让 GORM 知道如何将该类型接收、保存到数据库

例如:

```
type JSON json.RawMessage

// 实现 sql.Scanner 接口, Scan 将 value 扫描至 Jsonb
func (j *JSON) Scan(value interface{}) error {
  bytes, ok := value.([]byte)
  if !ok {
    return errors.New(fmt.Sprint("Failed to unmarshal JSONB value:", value))
  }

  result := json.RawMessage{}
  err := json.Unmarshal(bytes, &result)
  *j = JSON(result)
  return err
}

// 实现 driver.Valuer 接口, Value 返回 json value
func (j JSON) Value() (driver.Value, error) {
  if len(j) == 0 {
    return nil, nil
  }
  return json.RawMessage(j).MarshalJSON()
}
```

GormDataTypeInterface

自定义数据类型在不同的数据库中可能是不同数据类型，您可以实现 `GormDataTypeInterface` 来设置它们，例如：

```
type GormDataTypeInterface interface {
  GormDBDataType(*gorm.DB, *schema.Field) string
}
```

```
func (JSON) GormDBDataType(db *gorm.DB, field *schema.Field) string {
  // 使用 field.Tag、field.TagSettings 获取字段的 tag
  // 查看 https://github.com/go-gorm/gorm/blob/master/schema/field.go 获取全部的选项

  // 根据不同的数据库驱动返回不同的数据类型
  switch db.Dialector.Name() {
  case "mysql":
    return "JSON"
  case "postgres":
    return "JSONB"
  }
  return ""
}
```

Clause Expression

自定义数据类型可能需要特殊的 SQL，此时 GORM 提供的 API 不适用。这时候您可以定义一个 `Builder` 来实现 `clause.Expression` 接口

```
type Expression interface {
    Build(builder Builder)
}
```

查看 JSON 获取详情

```
// 根据 Clause Expression 生成 SQL
db.Find(&user, datatypes.JSONQuery("attributes").HasKey("role"))
db.Find(&user, datatypes.JSONQuery("attributes").HasKey("orgs", "orga"))

// MySQL
// SELECT * FROM `users` WHERE JSON_EXTRACT(`attributes`, '$.role') IS NOT NULL
// SELECT * FROM `users` WHERE JSON_EXTRACT(`attributes`, '$.orgs.orga') IS NOT NULL

// PostgreSQL
// SELECT * FROM "user" WHERE "attributes"::jsonb ? 'role'
// SELECT * FROM "user" WHERE "attributes"::jsonb -> 'orgs' ? 'orga'

db.Find(&user, datatypes.JSONQuery("attributes").Equals("jinzhu", "name"))
// MySQL
// SELECT * FROM `user` WHERE JSON_EXTRACT(`attributes`, '$.name') = "jinzhu"

// PostgreSQL
// SELECT * FROM "user" WHERE json_extract_path_text("attributes"::json,'name') = 'jinzhu'
```

自定义数据类型集合

我们创建了一个 Github 仓库，用于收集各种自定义数据类型https://github.com/go-gorm/datatype，非常欢迎同学们的 pull request ;)

Scopes

Scopes

Scopes 允许您轻松地复用常见的逻辑

查询

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

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

func PaidWithCod(db *gorm.DB) *gorm.DB {
  return db.Where("pay_mode_sign = ?", "C")
}

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

db.Scopes(AmountGreaterThan1000, PaidWithCreditCard).Find(&orders)
// 查找所有金额大于 1000 的信用卡订单

db.Scopes(AmountGreaterThan1000, PaidWithCod).Find(&orders)
// 查找所有金额大于 1000 的 COD 订单

db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid", "shipped"})).Find(
&orders)
// 查找所有金额大于1000 的已付款或已发货订单
```

分页

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

```
    pageSize, _ := strconv.Atoi(r.Query("page_size"))
    switch {
    case pageSize > 100:
      pageSize = 100
    case pageSize <= 0:
      pageSize = 10
    }

    offset := (page - 1) * perPasge
    return db.Offset(offset).Limit(pageSize)
  }
}

db.Scopes(Paginate(r)).Find(&users)
db.Scopes(Paginate(r)).Find(&articles)
```

更新

```
func CurOrganization(r *http.Request) func(db *gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    org := r.Query("org")

    if org != "" {
      var organization Organization
      if db.Session(&Session{}).First(&organization, "name = ?", org).Error ==
nil {
        return db.Where("organization_id = ?", org.ID)
      }
    }

    db.AddError("invalid organization")
    return db
  }
}
```

约定

约定

使用 `ID` 作为主键

默认情况下，GORM 会使用 `ID` 作为表的主键。

```
type User struct {
  ID   string // 默认情况下，名为 `ID` 的字段会作为表的主键
  Name string
}
```

你可以通过标签 `primaryKey` 将其它字段设为主键

```
// 将 `AnimalID` 设为主键
type Animal struct {
  ID     int64
  UUID   string `gorm:"primaryKey"`
  Name   string
  Age    int64
}
```

此外，您还可以看看 复合主键

复数表名

GORM 使用结构体名的 `蛇形命名` 作为表名。对于结构体 `User` ，根据约定，其表名为 `users`

TableName

您可以实现 `Tabler` 接口来更改默认表名，例如：

```
type Tabler interface {
    TableName() string
}

// TableName 会将 User 的表名重写为 `profiles`
func (User) TableName() string {
  return "profiles"
}
```

注意： `TableName` 不支持动态变化，它会被缓存下来以便后续使用。想要使用动态表名，你可以使用下面的

代码：

```
func UserTable(user User) func (db *gorm.DB) *gorm.DB {
  return func (db *gorm.DB) *gorm.DB {
    if user.Admin {
      return db.Table("admin_users")
    }

    return db.Table("users")
  }
}

DB.Scopes(UserTable(user)).Create(&user)
```

临时指定表明

您可以使用 `Table` 方法临时指定表名，例如：

```
// 根据 User 的字段创建 `deleted_users` 表
db.Table("deleted_users").AutoMigrate(&User{})

// 从另一张表查询数据
var deletedUsers []User
db.Table("deleted_users").Find(&deletedUsers)
// SELECT * FROM deleted_users;

db.Table("deleted_users").Where("name = ?", "jinzhu").Delete(&User{})
// DELETE FROM deleted_users WHERE name = 'jinzhu';
```

查看 from 子查询 了解如何在 FROM 子句中使用子查询

命名策略

GORM 允许用户通过覆盖默认的 `命名策略` 更改默认的命名约定，命名策略被用于构建： `TableName` 、 `ColumnName` 、 `JoinTableName` 、 `RelationshipFKName` 、 `CheckerName` 、 `IndexName` 。查看 GORM 配置 获取详情

列名

根据约定，数据表的列名使用的是 struct 字段名的 `蛇形命名`

```
type User struct {
  ID        uint      // 列名是 `id`
  Name      string    // 列名是 `name`
  Birthday  time.Time // 列名是 `birthday`
  CreatedAt time.Time // 列名是 `created_at`
```

```
}
```

您可以使用标签 `column` 或 命名策略 来覆盖列名

```
type Animal struct {
  AnimalID int64     `gorm:"column:beast_id"`          // 将列名设为 `beast_id`
  Birthday time.Time `gorm:"column:day_of_the_beast"` // 将列名设为 `day_of_the_
beast`
  Age      int64     `gorm:"column:age_of_the_beast"` // 将列名设为 `age_of_the_
beast`
}
```

时间戳追踪

CreatedAt

对于有 `CreatedAt` 字段的模型，创建记录时，如果该字段值为零值，则将该字段的值设为当前时间

```
db.Create(&user) // 将 `CreatedAt` 设为当前时间

// 想要修改该字段的值，你可以使用 `Update`
db.Model(&user).Update("CreatedAt", time.Now())
```

UpdatedAt

对于有 `UpdatedAt` 字段的模型，更新记录时，将该字段的值设为当前时间。创建记录时，如果该字段值为零值，则将该字段的值设为当前时间

```
db.Save(&user) // 将 `UpdatedAt` 设为当前时间

db.Model(&user).Update("name", "jinzhu") // 也会将 `UpdatedAt` 设为当前时间
```

注意 GORM 支持拥有多种类型的时间追踪字段。可以根据 UNIX 秒、纳秒、其它类型追踪时间，查看 模型 获取详情

设置

设置

GORM 提供了 `Set` ， `Get` ， `InstanceSet` ， `InstanceGet` 方法来允许用户传值给 钩子 或其他方法 Gorm 中有一些特性用到了这种机制，如迁移表格时传递表格选项。

```
// 创建表时添加表后缀
db.Set("gorm:table_options", "ENGINE=InnoDB").AutoMigrate(&User{})
```

Set / Get

使用 `Set` / `Get` 传递设置到钩子方法，例如：

```
type User struct {
  gorm.Model
  CreditCard CreditCard
  // ...
}

func (u *User) BeforeCreate(tx *gorm.DB) error {
  myValue, ok := tx.Get("my_value")
  // ok => true
  // myValue => 123
}

type CreditCard struct {
  gorm.Model
  // ...
}

func (card *CreditCard) BeforeCreate(tx *gorm.DB) error {
  myValue, ok := tx.Get("my_value")
  // ok => true
  // myValue => 123
}

myValue := 123
db.Set("my_value", myValue).Create(&User{})
```

InstanceSet / InstanceGet

使用 `InstanceSet` / `InstanceGet` 传递设置到 `*Statement` 的钩子方法，例如：

```
type User struct {
  gorm.Model
  CreditCard CreditCard
  // ...
}

func (u *User) BeforeCreate(tx *gorm.DB) error {
  myValue, ok := tx.InstanceGet("my_value")
  // ok => true
  // myValue => 123
}

type CreditCard struct {
  gorm.Model
  // ...
}

// 在创建关联时, GORM 创建了一个新 `*Statement`, 所以它不能读取到其它实例的设置
func (card *CreditCard) BeforeCreate(tx *gorm.DB) error {
  myValue, ok := tx.InstanceGet("my_value")
  // ok => false
  // myValue => nil
}

myValue := 123
db.InstanceSet("my_value", myValue).Create(&User{})
```

高级主题

目前GormV2最佳实践后台管理系统 gin-vue-admin

- 基于gin+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

gf-vue-admin

- 基于goframe+vue搭建的后台管理系统框架，集成:
  - jwt鉴权
  - 权限管理
  - 动态路由
  - 分页封装
  - 多点登录拦截
  - 资源权限
  - 上传下载
  - 代码生成器
  - 表单生成器等基础功能
  - 五分钟一套CURD前后端代码
- 你值得拥有，赶紧来体验吧

导航

- 数据库索引
- 约束
- 复合主键
- 安全
- GORM 配置
- 编写插件
- 编写驱动
- 更新日志
- 社区
- 贡献

DBResolver

DBResolver

DBResolver 为 GORM 提供了多个数据库支持，支持以下功能：

- 支持多个 sources、replicas
- 读写分离
- 根据工作表、struct 自动切换连接
- 手动切换连接
- Sources/Replicas 负载均衡
- 适用于原生 SQL

https://github.com/go-gorm/dbresolver

用法

```
import (
  "gorm.io/gorm"
  "gorm.io/plugin/dbresolver"
  "gorm.io/driver/mysql"
)

DB, err := gorm.Open(mysql.Open("db1_dsn"), &gorm.Config{})

DB.Use(dbresolver.Register(dbresolver.Config{
  // `db2` 作为 sources, `db3`、`db4` 作为 replicas
  Sources:  []gorm.Dialector{mysql.Open("db2_dsn")},
  Replicas: []gorm.Dialector{mysql.Open("db3_dsn"), mysql.Open("db4_dsn")},
  // sources/replicas 负载均衡策略
  Policy: dbresolver.RandomPolicy{},
}).Register(dbresolver.Config{
  // `db1` 作为 sources（DB 的默认连接），对于 `User`、`Address` 使用 `db5` 作为 rep
licas
  Replicas: []gorm.Dialector{mysql.Open("db5_dsn")},
}, &User{}, &Address{}).Register(dbresolver.Config{
  // `db6`、`db7` 作为 sources, 对于 `orders`、`Product` 使用 `db8` 作为 replicas
  Sources:  []gorm.Dialector{mysql.Open("db6_dsn"), mysql.Open("db7_dsn")},
  Replicas: []gorm.Dialector{mysql.Open("db8_dsn")},
}, "orders", &Product{}, "secondary"))
```

事务

使用 transaction 时，DBResolver 也会使用一个事务，且不会切换 sources/replicas 连接

自动切换连接

DBResolver 会根据工作表、struct 自动切换连接

对于原生 SQL，DBResolver 会从 SQL 中提取表名以匹配 Resolver，除非 SQL 开头为 `SELECT` ，否则 DBResolver 总是会使用 `sources` ，例如：

```
// `User` Resolver 示例
DB.Table("users").Rows() // replicas `db5`
DB.Model(&User{}).Find(&AdvancedUser{}) // replicas `db5`
DB.Exec("update users set name = ?", "jinzhu") // sources `db1`
DB.Raw("select name from users").Row().Scan(&name) // replicas `db5`
DB.Create(&user) // sources `db1`
DB.Delete(&User{}, "name = ?", "jinzhu") // sources `db1`
DB.Table("users").Update("name", "jinzhu") // sources `db1`

// 全局 Resolver 示例
DB.Find(&Pet{}) // replicas `db3`/`db4`
DB.Save(&Pet{}) // sources `db2`

// Orders Resolver 示例
DB.Find(&Order{}) // replicas `db8`
DB.Table("orders").Find(&Report{}) // replicas `db8`
```

读写分离

DBResolver 的读写分离目前是基于 GORM callback 实现的。

对于 `Query` 、 `Row` callback，如果手动指定为 `Write` 模式，此时会使用 `sources` ，否则使用 `replicas` 。对于 `Raw` callback，如果 SQL 是以 `SELECT` 开头，语句会被认为是只读的，会使用 `replicas` ，否则会使用 `sources` 。

手动切换连接

```
// 使用 Write 模式：从 sources db `db1` 读取 user
DB.Clauses(dbresolver.Write).First(&user)

// 指定 Resolver：从 `secondary` 的 replicas db `db8` 读取 user
DB.Clauses(dbresolver.Use("secondary")).First(&user)

// 指定 Resolver 和 Write 模式：从 `secondary` 的 sources db `db6` 或 `db7` 读取 u
ser
DB.Clauses(dbresolver.Use("secondary"), dbresolver.Write).First(&user)
```

负载均衡

GORM 支持基于策略的 sources/replicas 负载均衡，自定义策略需实现以下接口：

```
type Policy interface {
    Resolve([]gorm.ConnPool) gorm.ConnPool
}
```

当前只实现了一个 `RandomPolicy` 策略，如果没有指定策略，它就是默认策略。

连接池

```
DB.Use(
  dbresolver.Register(dbresolver.Config{ /* xxx */ }).
  SetConnMaxIdleTime(time.Hour).
  SetConnMaxLifetime(24 * time.Hour).
  SetMaxIdleConns(100).
  SetMaxOpenConns(200)
)
```

Prometheus

Prometheus

GORM 提供了 Prometheus 插件来收集 DBStats 和用户自定义指标

https://github.com/go-gorm/prometheus

用法

```
import (
  "gorm.io/gorm"
  "gorm.io/driver/sqlite"
  "gorm.io/plugin/prometheus"
)

db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{})

db.Use(prometheus.New(prometheus.Config{
  DBName:          "db1", // 使用 `DBName` 作为指标 label
  RefreshInterval: 15,    // 指标刷新频率（默认为 15 秒）
  PushAddr:        "prometheus pusher address", // 如果配置了 `PushAddr`，则推送指标

  StartServer:     true,  // 启用一个 http 服务来暴露指标
  HTTPServerPort:  8080,  // 配置 http 服务监听端口，默认端口为 8080 （如果您配置了多
个，只有第一个 `HTTPServerPort` 会被使用）
  MetricsCollector: []prometheus.MetricsCollector {
    &prometheus.MySQL{
      VariableNames: []string{"Threads_running"},
    },
  },  // 用户自定义指标
}))
```

用户自定义指标

您可以通过 GORM Prometheus 插件定义并收集自定义的指标，这需要实现 `MetricCollector` 接口

```
type MetricsCollector interface {
  Metrics(*Prometheus) []prometheus.Collector
}
```

MySQL

GORM 提供了一个示例，说明如何收集 MySQL 状态指标，查看 prometheus.MySQL 获取详情

```
&prometheus.MySQL{
  // 指标名前缀，默认为 `gorm_status_`
  // 例如： Threads_running 的指标名就是 `gorm_status_Threads_running`
  Prefix: "gorm_status_",
  // 拉取频率，默认使用 Prometheus 的 RefreshInterval
  Interval: 100,
  // 从 SHOW STATUS 选择变量变量，如果不设置，则使用全部的状态变量
  VariableNames: []string{"Threads_running"},
}
```

提示

提示

GORM 提供了优化器、索引、备注提示支持

https://github.com/go-gorm/hints

优化器提示

```
import "gorm.io/hints"

DB.Clauses(hints.New("hint")).Find(&User{})
// SELECT * /*+ hint */ FROM `users`
```

索引提示

```
import "gorm.io/hints"

DB.Clauses(hints.UseIndex("idx_user_name")).Find(&User{})
// SELECT * FROM `users` USE INDEX (`idx_user_name`)

DB.Clauses(hints.ForceIndex("idx_user_name", "idx_user_id").ForJoin()).Find(&Us
er{})
// SELECT * FROM `users` FORCE INDEX FOR JOIN (`idx_user_name`,`idx_user_id`)"

DB.Clauses(
    hints.ForceIndex("idx_user_name", "idx_user_id").ForOrderBy(),
    hints.IgnoreIndex("idx_user_name").ForGroupBy(),
).Find(&User{})
// SELECT * FROM `users` FORCE INDEX FOR ORDER BY (`idx_user_name`,`idx_user_id
`) IGNORE INDEX FOR GROUP BY (`idx_user_name`)"
```

备注提示

```
import "gorm.io/hints"

DB.Clauses(hints.Comment("select", "master")).Find(&User{})
// SELECT /*master*/ * FROM `users`;

DB.Clauses(hints.CommentBefore("insert", "node2")).Create(&user)
// /*node2*/ INSERT INTO `users` ...;

DB.Clauses(hints.CommentAfter("select", "node2")).Create(&user)
// /*node2*/ INSERT INTO `users` ...;
```

```
DB.Clauses(hints.CommentAfter("where", "hint")).Find(&User{}, "id = ?", 1)
// SELECT * FROM `users` WHERE id = ? /* hint */
```

数据库索引

数据库索引

GORM 允许通过 `index` 、 `uniqueIndex` 标签创建索引，这些索引将在使用 GORM 进行AutoMigrate 或 Createtable 时创建

索引标签

GORM 可以接受很多的索引设置，例如： `class` 、 `type` 、 `where` 、 `comment` 、 `expression` 、 `sort` 、 `collate`

下面的示例演示了如何使用它：

```
type User struct {
    Name  string `gorm:"index"`
    Name2 string `gorm:"index:idx_name,unique"`
    Name3 string `gorm:"index:,sort:desc,collate:utf8,type:btree,length:10,wher
e:name3 != 'jinzhu'"`
    Name4 string `gorm:"uniqueIndex"`
    Age   int64  `gorm:"index:,class:FULLTEXT,comment:hello \\, world,where:age
 > 10"`
    Age2  int64  `gorm:"index:,expression:ABS(age)"`
}
```

唯一索引

```
uniqueIndex` 标签的作用与 `index` 类似，它等效于 `index:,unique
type User struct {
    Name1 string `gorm:"uniqueIndex"`
    Name2 string `gorm:"uniqueIndex:idx_name,sort:desc"`
}
```

复合索引

两个字段使用同一个索引名将创建复合索引，例如：

```
type User struct {
    Name   string `gorm:"index:idx_member"`
    Number string `gorm:"index:idx_member"`
}
```

字段优先级

复合索引列的顺序会影响其性能，因此必须仔细考虑

您可以使用 `priority` 指定顺序，默认优先级值是 `10` ，如果优先级值相同，则顺序取决于模型结构体字段的顺序

```
type User struct {
    Name   string `gorm:"index:idx_member"`
    Number string `gorm:"index:idx_member"`
}
// column order: name, number

type User struct {
    Name   string `gorm:"index:idx_member,priority:2"`
    Number string `gorm:"index:idx_member,priority:1"`
}
// column order: number, name

type User struct {
    Name   string `gorm:"index:idx_member,priority:12"`
    Number string `gorm:"index:idx_member"`
}
// column order: number, name
```

多索引

一个字段接受多个 `index` 、 `uniqueIndex` 标签，这会在一个字段上创建多个索引

```
type UserIndex struct {
    OID          int64  `gorm:"index:idx_id;index:idx_oid,unique"`
    MemberNumber string `gorm:"index:idx_id"`
}
```

约束

约束

GORM 允许通过标签创建数据库约束，约束会在通过 GORM 进行 AutoMigrate 或创建数据表时被创建。

检查约束

通过 `check` 标签创建检查约束

```
type UserIndex struct {
    Name  string `gorm:"check:name_checker,name <> 'jinzhu'"`
    Name2 string `gorm:"check:name <> 'jinzhu'"`
    Name3 string `gorm:"check:,name <> 'jinzhu'"`
}
```

索引约束

查看 数据库索引 获取详情

外键约束

GORM 会为关联创建外键约束，您可以在初始化过程中禁用此功能：

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  DisableForeignKeyConstraintWhenMigrating: true,
})
```

GORM 允许您通过 `constraint` 标签的 `Ondelete` 、 `Ondelete` 选项设置外键约束，例如：

```
type User struct {
  gorm.Model
  CompanyID  int
  Company    Company    `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
  CreditCard CreditCard `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;"`
}

type CreditCard struct {
  gorm.Model
  Number string
  UserID uint
}

type Company struct {
```

```
    ID   int
    Name string
}
```

复合主键

复合主键

通过将多个字段设为主键，以创建复合主键，例如：

```
type Product struct {
  ID           string `gorm:"primaryKey"`
  LanguageCode string `gorm:"primaryKey"`
  Code         string
  Name         string
}
```

注意默认情况下，整形 `PrioritizedPrimaryField` 启用了 `AutoIncrement` ，要禁用它，您需要为整形字段关闭 `autoIncrement` ：

```
type Product struct {
  CategoryID uint64 `gorm:"primaryKey;autoIncrement:false"`
  TypeID     uint64 `gorm:"primaryKey;autoIncrement:false"`
}
```

安全

安全

GORM 使用 `database/sql` 的参数占位符来构造 SQL 语句，这可以自动转义参数，避免 SQL 注入数据

注意 Logger 打印的 SQL 并不像最终执行的 SQL 那样已经转义，复制和运行这些 SQL 时应当注意。

查询条件

用户的输入只能作为参数，例如：

```
userInput := "jinzhu;drop table users;"

// 安全的，会被转义
db.Where("name = ?", userInput).First(&user)

// SQL 注入
db.Where(fmt.Sprintf("name = %v", userInput)).First(&user)
```

内联条件

```
// 会被转义
db.First(&user, "name = ?", userInput)

// SQL 注入
db..First(&user, fmt.Sprintf("name = %v", userInput))
```

SQL 注入方法

为了支持某些功能，一些输入不会被转义，调用方法时要小心用户输入的参数。

```
db.Select("name; drop table users;").First(&user)
db.Distinct("name; drop table users;").First(&user)

db.Model(&user).Pluck("name; drop table users;", &names)

db.Group("name; drop table users;").First(&user)

db.Group("name").Having("1 = 1;drop table users;").First(&user)

db.Raw("select name from users; drop table users;").First(&user)

db.Exec("select name from users; drop table users;")
```

避免 SQL 注入的一般原则是，不信任用户提交的数据。您可以进行白名单验证来测试用户的输入是否为已知安全的、已批准、已定义的输入，并且在使用用户的输入时，仅将它们作为参数。

GORM 配置

GORM 配置

GORM 提供的配置可以在初始化时使用

```
type Config struct {
    SkipDefaultTransaction bool
    NamingStrategy schema.Namer
    Logger logger.Interface
    NowFunc func() time.Time
    DryRun bool
    PrepareStmt bool
    DisableAutomaticPing bool
    DisableForeignKeyConstraintWhenMigrating bool
}
```

跳过默认事务

为了确保数据一致性，GORM 会在事务里执行写入操作（创建、更新、删除）。如果没有这方面的要求，您可以在初始化时禁用它。

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  SkipDefaultTransaction: true,
})
```

命名策略

GORM 允许用户通过覆盖默认的 `命名策略` 更改默认的命名约定，这需要实现接口 `Namer`

```
type Namer interface {
    TableName(table string) string
    ColumnName(table, column string) string
    JoinTableName(table string) string
    RelationshipFKName(Relationship) string
    CheckerName(table, column string) string
    IndexName(table, column string) string
}
```

默认 `NamingStrategy` 也提供了几个选项，如：

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
```

```
  NamingStrategy: schema.NamingStrategy{
    TablePrefix: "t_",   // 表名前缀，`User` 的表名应该是 `t_users`
    SingularTable: true, // 使用单数表名，启用该选项，此时，`User` 的表名应该是 `t_use
r`
  },
})
```

Logger

允许通过覆盖此选项更改 GORM 的默认 logger，参考 Logger 获取详情

NowFunc

更改创建时间使用的函数

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  NowFunc: func() time.Time {
    return time.Now().Local()
  },
})
```

DryRun

生成 `SQL` 但不执行，可以用于准备或测试生成的 SQL，参考 会话 获取详情

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  DryRun: false,
})
```

PrepareStmt

`PreparedStmt` 在执行任何 SQL 时都会创建一个 prepared statement 并将其缓存，以提高后续的效率，参考 会话 获取详情

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  PrepareStmt: false,
})
```

DisableAutomaticPing

在完成初始化后，GORM 会自动 ping 数据库以检查数据库的可用性，若要禁用该特性，可将其设置为 `true`

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  DisableAutomaticPing: true,
```

```
})
```

DisableForeignKeyConstraintWhenMigrating

在 `AutoMigrate` 或 `CreateTable` 时，GORM 会自动创建外键约束，若要禁用该特性，可将其设置为 `true` ，参考 迁移 获取详情。

```
db, err := gorm.Open(sqlite.Open("gorm.db"), &gorm.Config{
  DisableForeignKeyConstraintWhenMigrating: true,
})
```

编写插件

编写插件

Callbacks

GORM 自身也是基于 `Callbacks` 的，包括 `Create` 、 `Query` 、 `Update` 、 `Delete` 、 `Row` 、 `Raw` 。此外，您也完全可以根据自己的意愿自定义 GORM

回调会注册到全局 `*gorm.DB` ，而不是会话级别。如果您想要 `*gorm.DB` 具有不同的回调，您需要初始化另一个 `*gorm.DB`

注册回调

注册回调至 callbacks

```
func cropImage(db *gorm.DB) {
  if db.Statement.Schema != nil {
    // 伪代码：裁剪图片字段并将其上传至 CDN
    for _, field := range db.Statement.Schema.Fields {
      switch db.Statement.ReflectValue.Kind() {
      case reflect.Slice, reflect.Array:
        for i := 0; i < db.Statement.ReflectValue.Len(); i++ {
          // 从字段获取 value
          if fieldValue, isZero := field.ValueOf(db.Statement.ReflectValue.Index
(i)); !isZero {
            if crop, ok := fieldValue.(CropInterface); ok {
              crop.Crop()
            }
          }
        }
      case reflect.Struct:
        // 从字段获取 value
        if fieldValue, isZero := field.ValueOf(db.Statement.ReflectValue); isZe
ro {
          if crop, ok := fieldValue.(CropInterface); ok {
            crop.Crop()
          }
        }

        // 设置字段的 value
        err := field.Set(db.Statement.ReflectValue, "newValue")
      }
    }

    // 当前 model 的所有字段
    db.Statement.Schema.Fields
```

```
    // 当前 model 的所有主键字段
    db.Statement.Schema.PrimaryFields

    // 优先的主键字段：DB 列名为 `id` 或定义的第一个主键
    db.Statement.Schema.PrioritizedPrimaryField

    // 当前 model 的所有关系
    db.Statement.Schema.Relationships

    field := db.Statement.Schema.LookUpField("Name")
    // 处理...
  }
}

db.Callback().Create().Register("crop_image", cropImage)
// 为 Crete 流程注册一个回调
```

删除回调

从 callbacks 中删除回调

```
db.Callback().Create().Remove("gorm:create")
// 从 Create 的 callbacks 中删除 `gorm:create`
```

替换回调

用一个新的回调替换已有的同名回调

```
db.Callback().Create().Replace("gorm:create", newCreateFunction)
// 用新函数 `newCreateFunction` 替换 Create 流程目前的 `gorm:create`
```

注册带顺序的回调

注册带顺序的回调

```
db.Callback().Create().Before("gorm:create").Register("update_created_at", upda
teCreated)
db.Callback().Create().After("gorm:create").Register("update_created_at", updat
eCreated)
db.Callback().Query().After("gorm:query").Register("my_plugin:after_query", aft
erQuery)
db.Callback().Delete().After("gorm:delete").Register("my_plugin:after_delete",
afterDelete)
db.Callback().Update().Before("gorm:update").Register("my_plugin:before_update",
 beforeUpdate)
```

```
db.Callback().Create().Before("gorm:create").After("gorm:before_create").Register("my_plugin:before_create", beforeCreate)
```

预定义回调

GORM 已经定义了 一些回调 来支持当前的 GORM 功能，在启动您的插件之前可以先看看这些回调

插件

GORM 提供了 `Use` 方法来注册插件，插件需要实现 `Plugin` 接口

```
type Plugin interface {
  Name() string
  Initialize(*gorm.DB) error
}
```

当插件首次注册到 GORM 时将调用 `Initialize` 方法，且 GORM 会保存已注册的插件，你可以这样访问访问：

```
db.Config.Plugins[pluginName]
```

查看 Prometheus 的例子

编写驱动

编写驱动

编写新驱动

GORM 官方支持 `sqlite` 、 `mysql` 、 `postgres` 、 `sqlserver` 。

有些数据库可能兼容 `mysql` 、 `postgres` 的方言，在这种情况下，你可以直接使用这些数据库的方言。

对于其它不兼容的情况，您可以自行编写一个新驱动，这需要实现方言接口。

```
type Dialector interface {
    Name() string
    Initialize(*DB) error
    Migrator(db *DB) Migrator
    DataTypeOf(*schema.Field) string
    DefaultValueOf(*schema.Field) clause.Expression
    BindVarTo(writer clause.Writer, stmt *Statement, v interface{})
    QuoteTo(clause.Writer, string)
    Explain(sql string, vars ...interface{}) string
}
```

查看MySQL 驱动的例子

更新日志

更新日志

v2.0 - 2020.07

GORM 2.0 是根据我们在过去几年里收到的反馈从零重写的，它引入了一些不兼容的 API 更改和许多改进

- 性能改进
- 模块化
- Context、批量插入、Prepared Statment、DryRun 模式、Join Preload、Find To Map、FindInBatches
- SavePoint、RollbackTo、嵌套事务
- 关联改进（删除、更新时），修改 Many2Many 的连接表，批量数据关联模式
- SQL 构建器、Upsert、Locking 和 Optimizer、Index、Comment 提示
- 支持多个字段的自动追踪创建、更新时间，且支持纳秒级、毫秒级、秒级时间戳
- 字段级权限控制：只读、只写、只创建、只更新、忽略
- 全新的 Migrator、Logger
- 命名策略(统一表名、字段名、连接表名、外键、检查器、索引名称规则)
- 更好的自定义数据类型支持（例如：JSON）
- 全新的插件系统、Hooks API

v1.0 - 2016.04

GORM V1 文档

破坏性变更

- `gorm.Open` 返回类型是 `*gorm.DB` 而不是 `gorm.DB`
- Update 只会更新有变更的字段
- 开启软删除后，默认只会检查 `deleted_at IS NULL` 的记录
- 新的 ToDBName 逻辑

  当 GORM 将 struct、字段转换为数据库名时，采用了类似于 golint 处理 `HTTP` 和 `URI` 缩写的方式。因此， `HTTP` 的数据库名是 `http` ，而不是 `h_t_t_p` 。

  但是对于列表中没有的其他缩写，例如但是对于列表中没有的其他缩写，例如 `SKU` ，db名是 `s_k_u` ，此次更新修复了该问题。
- `RecordNotFound` 错误已被重命名为 `ErrRecordNotFound`
- `mssql` 已被重命名为 ” github.com/jinzhu/gorm/dialects/mssql”
- `Hstore` 已移至 ” github.com/jinzhu/gorm/dialects/postgres”

社区

Community

v1社区

v2社区

贡献

Contribute

v1贡献

v2贡献