目　录

概述

入门指南

- 模型定义
- 惯例
- 连接数据库

CRUD 接口

- 创建
- 查询
- 更新
- 删除

关联

- Belongs To
- Has One
- Has Many
- Many To Many
- 关联
- 预加载

教程

- 链式操作
- 错误处理
- 钩子
- 事务
- 数据库迁移
- 原生 SQL 和 SQL 生成器
- 通用数据库接口

高级主题

- 复合主键
- 创建插件
- GORM Dialects
- 自定义 Logger
- 更新日志

概述

一个神奇的，对开发人员友好的 Golang ORM 库

概览

- 全特性 ORM (几乎包含所有特性)
- 模型关联 (一对一， 一对多，一对多（反向）， 多对多， 多态关联)
- 钩子 (Before/After Create/Save/Update/Delete/Find)
- 预加载
- 事务
- 复合主键
- SQL 构造器
- 自动迁移
- 日志
- 基于GORM回调编写可扩展插件
- 全特性测试覆盖
- 开发者友好

安装

```
go get -u github.com/jinzhu/gorm
```

快速开始

```
package main

import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/sqlite"
)

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

```
}

func main() {
  db, err := gorm.Open("sqlite3", "test.db")
  if err != nil {
    panic("failed to connect database")
  }
  defer db.Close()

  //自动检查 Product 结构是否变化，变化则进行迁移
  db.AutoMigrate(&Product{})

  // 增
  db.Create(&Product{Code: "L1212", Price: 1000})

  // 查
  var product Product
  db.First(&product, 1) // 找到id为1的产品
  db.First(&product, "code = ?", "L1212") // 找出 code 为 11212 的产品

  // 改 - 更新产品的价格为 2000
  db.Model(&product).Update("Price", 2000)

  // 删 - 删除产品
  db.Delete(&product)
}
```

入门指南

模型定义

惯例

连接数据库

模型定义

模型一般都是普通的 Golang 的结构体，Go的基本数据类型，或者指针。 `sql.Scanner` 和 `driver.Valuer` ，同时也支持接口。

例子：

```
type User struct {
  gorm.Model
  Name         string
  Age          sql.NullInt64
  Birthday     *time.Time
  Email        string  `gorm:"type:varchar(100);unique_index"`
  Role         string  `gorm:"size:255"` //设置字段的大小为255个字节
  MemberNumber *string `gorm:"unique;not null"` // 设置 memberNumber 字段唯一且不为空
  Num          int     `gorm:"AUTO_INCREMENT"` // 设置 Num字段自增
  Address      string  `gorm:"index:addr"` // 给Address 创建一个名字是 `addr`的索引
  IgnoreMe     int     `gorm:"-"` //忽略这个字段
}
```

结构标签

标签是声明模型时可选的标记。GORM 支持以下标记：

支持的结构标签

标签	说明
Column	指定列的名称
Type	指定列的类型
Size	指定列的大小，默认是 255
PRIMARY_KEY	指定一个列作为主键
UNIQUE	指定一个唯一的列
DEFAULT	指定一个列的默认值
PRECISION	指定列的数据的精度

标签	说明
NOT NULL	指定列的数据不为空
AUTO_INCREMENT	指定一个列的数据是否自增
INDEX	创建带或不带名称的索引，同名创建复合索引
UNIQUE_INDEX	类似 `索引` ，创建一个唯一的索引
EMBEDDED	将 struct 设置为 embedded
EMBEDDED_PREFIX	设置嵌入式结构的前缀名称
-	忽略这些字段

关联的结构标签

有关详细信息，请查看「关联」部分

标签	说明
MANY2MANY	指定连接表名称
FOREIGNKEY	指定外键
ASSOCIATION_FOREIGNKEY	指定关联外键
POLYMORPHIC	指定多态类型
POLYMORPHIC_VALUE	指定多态的值
JOINTABLE_FOREIGNKEY	指定连接表的外键
ASSOCIATION_JOINTABLE_FOREIGNKEY	指定连接表的关联外键
SAVE_ASSOCIATIONS	是否自动保存关联
ASSOCIATION_AUTOUPDATE	是否自动更新关联
ASSOCIATION_AUTOCREATE	是否自动创建关联
ASSOCIATION_SAVE_REFERENCE	是否引用自动保存的关联
PRELOAD	是否自动预加载关联

惯例

gorm.Model

`gorm.Model` 是一个包含一些基本字段的结构体，包含的字段有 `ID`，`CreatedAt`，`UpdatedAt`，`DeletedAt`。

你可以用它来嵌入到你的模型中，或者也可以用它来建立自己的模型。

```
// gorm.Model 定义
type Model struct {
  ID        uint `gorm:"primary_key"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt *time.Time
}

// 将字段 `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt` 注入到 `User` 模型中
type User struct {
  gorm.Model
  Name string
}

// 声明 gorm.Model 模型
type User struct {
  ID   int
  Name string
}
```

`ID` 作为主键

GORM 默认使用 `ID` 作为主键名。

```
type User struct {
  ID   string // 字段名 `ID` 将被作为默认的主键名
}

// 设置字段 `AnimalID` 为默认主键
type Animal struct {
  AnimalID int64 `gorm:"primary_key"`
  Name     string
```

```
  Age      int64
}
```

复数表名

表名是结构体名称的复数形式

```
type User struct {} // 默认的表名是 `users`

// 设置 `User` 的表名为 `profiles`
func (User) TableName() string {
  return "profiles"
}

func (u User) TableName() string {
    if u.Role == "admin" {
        return "admin_users"
    } else {
        return "users"
    }
}

// 如果设置禁用表名复数形式属性为 true，`User` 的表名将是 `user`
db.SingularTable(true)
```

指定表名

```
// 用 `User` 结构体创建 `delete_users` 表
db.Table("deleted_users").CreateTable(&User{})

var deleted_users []User
db.Table("deleted_users").Find(&deleted_users)
//// SELECT * FROM deleted_users;

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

修改默认表名

你可以通过定义 `DefaultTableNameHandler` 字段来对表名使用任何规则。

```
gorm.DefaultTableNameHandler = func (db *gorm.DB, defaultTableName string) strin
g  {
    return "prefix_" + defaultTableName;
}
```

蛇形列名

列名是字段名的蛇形小写形式

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

// 重写列名
type Animal struct {
    AnimalId    int64     `gorm:"column:beast_id"`         // 设置列名为 `beast_
id`
    Birthday    time.Time `gorm:"column:day_of_the_beast"` // 设置列名为 `day_of
_the_beast`
    Age         int64     `gorm:"column:age_of_the_beast"` // 设置列名为 `age_of
_the_beast`
}
```

时间戳跟踪

CreatedAt

对于有 `CreatedAt` 字段的模型，它将被设置为首次创建记录的当前时间。

```
db.Create(&user) // 将设置 `CreatedAt` 为当前时间

// 你可以使用 `Update` 方法来更改默认时间
db.Model(&user).Update("CreatedAt", time.Now())
```

UpdatedAt

对于有 `UpdatedAt` 字段的模型，它将被设置为记录更新时的当前时间。

```
db.Save(&user) // 将设置 `UpdatedAt` 为当前时间

db.Model(&user).Update("name", "jinzhu") // 将设置 `UpdatedAt` 为当前时间
```

DeletedAt

对于有 `DeletedAt` 字段的模型，当删除它们的实例时，它们并没有被从数据库中删除，只是将 `DeletedAt` 字段设置为当前时间。参考 Soft Delete

连接数据库

连接数据库

为了连接数据库，你首先要导入数据库驱动程序。例如：

```
import _ "github.com/go-sql-driver/mysql"
```

GORM 已经包含了一些驱动程序，为了方便的去记住它们的导入路径，你可以像下面这样导入 mysql 驱动程序

```
import _ "github.com/jinzhu/gorm/dialects/mysql"
// import _ "github.com/jinzhu/gorm/dialects/postgres"
// import _ "github.com/jinzhu/gorm/dialects/sqlite"
// import _ "github.com/jinzhu/gorm/dialects/mssql"
```

支持的数据库

MySQL

注意： 为了正确的处理 `time.Time` ，你需要包含 `parseTime` 作为参数。 (More supported parameters)

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mysql"
)

func main() {
  db, err := gorm.Open("mysql", "user:password@/dbname?charset=utf8&parseTime=True&loc=Local")
  defer db.Close()
}
```

PostgreSQL

```
import (
  "github.com/jinzhu/gorm"
```

```
  _ "github.com/jinzhu/gorm/dialects/postgres"
)

func main() {
  db, err := gorm.Open("postgres", "host=myhost port=myport user=gorm dbname=gorm password=mypassword")
  defer db.Close()
}
```

Sqlite3

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/sqlite"
)

func main() {
  db, err := gorm.Open("sqlite3", "/tmp/gorm.db")
  defer db.Close()
}
```

SQL Server

Get started with SQL Server，它可以通过 Docker 运行在你的 Mac， Linux 上。

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/mssql"
)

func main() {
  db, err := gorm.Open("mssql", "sqlserver://username:password@localhost:1433?database=dbname")
  defer db.Close()
}
```

不支持的数据库

GORM 官方支持以上四种数据库， 你可以为不支持的数据库编写支持，参考 GORM Dialects

CRUD 接口

创建

查询

更新

删除

创建

创建记录

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

db.NewRecord(user) // => 返回 `true` ，因为主键为空

db.Create(&user)

db.NewRecord(user) // => 在 `user` 之后创建返回 `false`
```

默认值

你可以通过标签定义字段的默认值，例如：

```
type Animal struct {
    ID   int64
    Name string `gorm:"default:'galeone'"`
    Age  int64
}
```

然后 SQL 会排除那些没有值或者有 零值 的字段，在记录插入数据库之后，gorm将从数据库中加载这些字段的值。

```
var animal = Animal{Age: 99, Name: ""}
db.Create(&animal)
// INSERT INTO animals("age") values('99');
// SELECT name from animals WHERE ID=111; // 返回的主键是 111
// animal.Name => 'galeone'
```

注意 所有包含零值的字段，像 `0` ， `''` ， `false` 或者其他的 零值 不会被保存到数据库中，但会使用这个字段的默认值。你应该考虑使用指针类型或者其他的值来避免这种情况:

```
// Use pointer value
type User struct {
  gorm.Model
  Name string
```

```
  Age  *int `gorm:"default:18"`
}

// Use scanner/valuer
type User struct {
  gorm.Model
  Name string
  Age  sql.NullInt64 `gorm:"default:18"`
}
```

在钩子中设置字段值

如果你想在 `BeforeCreate` 函数中更新字段的值，应该使用 `scope.SetColumn` ，例如：

```
func (user *User) BeforeCreate(scope *gorm.Scope) error {
  scope.SetColumn("ID", uuid.New())
  return nil
}
```

创建额外选项

```
// 为插入 SQL 语句添加额外选项
db.Set("gorm:insert_option", "ON CONFLICT").Create(&product)
// INSERT INTO products (name, code) VALUES ("name", "code") ON CONFLICT;
```

查询

查询

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

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

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

// 获取所有的记录
db.Find(&users)
//// SELECT * FROM users;

// 通过主键进行查询 (仅适用于主键是数字类型)
db.First(&user, 10)
//// SELECT * FROM users WHERE id = 10;
```

Where

原生 SQL

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

// 获取所有匹配的记录
db.Where("name = ?", "jinzhu").Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu';

// <>
db.Where("name <> ?", "jinzhu").Find(&users)

// IN
db.Where("name in (?)", []string{"jinzhu", "jinzhu 2"}).Find(&users)
```

```
// LIKE
db.Where("name LIKE ?", "%jin%").Find(&users)

// AND
db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)

// Time
db.Where("updated_at > ?", lastWeek).Find(&users)

// BETWEEN
db.Where("created_at BETWEEN ? AND ?", lastWeek, today).Find(&users)
```

Struct & Map

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

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

// 多主键 slice 查询
db.Where([]int64{20, 21, 22}).Find(&users)
//// SELECT * FROM users WHERE id IN (20, 21, 22);
```

NOTE 当通过struct进行查询的时候，GORM 将会查询这些字段的非零值， 意味着你的字段包含 `0` ， `''` ， `false` 或者其他 零值, 将不会出现在查询语句中， 例如:

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

你可以考虑适用指针类型或者 scanner/valuer 来避免这种情况。

```
// 使用指针类型
type User struct {
  gorm.Model
  Name string
  Age  *int
}

// 使用 scanner/valuer
type User struct {
```

```
  gorm.Model
  Name string
  Age  sql.NullInt64
}
```

Not

和 `Where` 查询类似

```
db.Not("name", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu" LIMIT 1;

// 不包含
db.Not("name", []string{"jinzhu", "jinzhu 2"}).Find(&users)
//// SELECT * FROM users WHERE name NOT IN ("jinzhu", "jinzhu 2");

//不在主键 slice 中
db.Not([]int64{1,2,3}).First(&user)
//// SELECT * FROM users WHERE id NOT IN (1,2,3);

db.Not([]int64{}).First(&user)
//// SELECT * FROM users;

// 原生 SQL
db.Not("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE NOT(name = "jinzhu");

// Struct
db.Not(User{Name: "jinzhu"}).First(&user)
//// SELECT * FROM users WHERE name <> "jinzhu";
```

Or

```
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"}).Find(&users)
//// SELECT * FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2';

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

行内条件查询

和 `Where` 查询类似。

需要注意的是，当使用链式调用传入行内条件查询时，这些查询不会被传参给后续的中间方法。

```
// 通过主键进行查询 (仅适用于主键是数字类型)
db.First(&user, 23)
//// SELECT * FROM users WHERE id = 23 LIMIT 1;
// 非数字类型的主键查询
db.First(&user, "id = ?", "string_primary_key")
//// SELECT * FROM users WHERE id = 'string_primary_key' LIMIT 1;

// 原生 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;
```

额外的查询选项

```
// 为查询 SQL 添加额外的选项
db.Set("gorm:query_option", "FOR UPDATE").First(&user, 10)
//// SELECT * FROM users WHERE id = 10 FOR UPDATE;
```

FirstOrInit

获取第一条匹配的记录，或者通过给定的条件下初始一条新的记录（仅适用与于 struct 和 map 条件）。

```
// 未查询到
db.FirstOrInit(&user, User{Name: "non_existing"})
//// user -> User{Name: "non_existing"}

// 查询到
db.Where(User{Name: "Jinzhu"}).FirstOrInit(&user)
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
db.FirstOrInit(&user, map[string]interface{}{"name": "jinzhu"})
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Attrs

如果未找到记录，则使用参数初始化 struct

```
// 未查询到
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}

db.Where(User{Name: "non_existing"}).Attrs("age", 20).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = 'non_existing';
//// user -> User{Name: "non_existing", Age: 20}

// 查询到
db.Where(User{Name: "Jinzhu"}).Attrs(User{Age: 30}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user -> User{Id: 111, Name: "Jinzhu", Age: 20}
```

Assign

无论是否查询到数据，都将参数赋值给 struct

```
// 未查询到
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrInit(&user)
//// user -> User{Name: "non_existing", Age: 20}

// 查询到
db.Where(User{Name: "Jinzhu"}).Assign(User{Age: 30}).FirstOrInit(&user)
//// SELECT * FROM USERS WHERE name = jinzhu';
//// user -> User{Id: 111, Name: "Jinzhu", Age: 30}
```

FirstOrCreate

获取第一条匹配的记录，或者通过给定的条件创建一条记录 （仅适用与于 struct 和 map 条件）。

```
// 未查询到
db.FirstOrCreate(&user, User{Name: "non_existing"})
//// INSERT INTO "users" (name) VALUES ("non_existing");
//// user -> User{Id: 112, Name: "non_existing"}

// 查询到
db.Where(User{Name: "Jinzhu"}).FirstOrCreate(&user)
//// user -> User{Id: 111, Name: "Jinzhu"}
```

Attrs

如果未查询到记录，通过给定的参数赋值给 struct ， 然后使用这些值添加一条记录。

```
// 未查询到
db.Where(User{Name: "non_existing"}).Attrs(User{Age: 20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
//// user -> User{Id: 112, Name: "non_existing", Age: 20}

// 查询到
db.Where(User{Name: "jinzhu"}).Attrs(User{Age: 30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// user -> User{Id: 111, Name: "jinzhu", Age: 20}
```

Assign

无论是否查询到，都将其分配给记录，并保存到数据库中。

```
// 未查询到
db.Where(User{Name: "non_existing"}).Assign(User{Age: 20}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'non_existing';
//// INSERT INTO "users" (name, age) VALUES ("non_existing", 20);
//// user -> User{Id: 112, Name: "non_existing", Age: 20}

// 查询到
db.Where(User{Name: "jinzhu"}).Assign(User{Age: 30}).FirstOrCreate(&user)
//// SELECT * FROM users WHERE name = 'jinzhu';
//// UPDATE users SET age=30 WHERE id = 111;
//// user -> User{Id: 111, Name: "jinzhu", Age: 30}
```

高级查询

子查询

使用 `*gorm.expr` 进行子查询

```
db.Where("amount > ?", DB.Table("orders").Select("AVG(amount)").Where("state = ?", "paid").QueryExpr()).Find(&orders)
// SELECT * FROM "orders"  WHERE "orders"."deleted_at" IS NULL AND (amount > (SELECT AVG(amount) FROM "orders"  WHERE (state = 'paid')));
```

查询

指定要从数据库检索的字段，默认情况下，将选择所有字段。

```
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

使用 Order 从数据库查询记录时，当第二个参数设置为 `true` 时，将会覆盖之前的定义条件。

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

// 多个排序条件
db.Order("age desc").Order("name").Find(&users)
//// SELECT * FROM users ORDER BY age desc, name;

// 重新排序
db.Order("age desc").Find(&users1).Order("age", true).Find(&users2)
//// SELECT * FROM users ORDER BY age desc; (users1)
//// SELECT * FROM users ORDER BY age; (users2)
```

Limit

指定要查询的最大记录数

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

// 用 -1 取消 LIMIT 限制条件
db.Limit(10).Find(&users1).Limit(-1).Find(&users2)
//// SELECT * FROM users LIMIT 10; (users1)
//// SELECT * FROM users; (users2)
```

Offset

指定在开始返回记录之前要跳过的记录数。

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

// 用 -1 取消 OFFSET 限制条件
db.Offset(10).Find(&users1).Offset(-1).Find(&users2)
//// SELECT * FROM users OFFSET 10; (users1)
//// SELECT * FROM users; (users2)
```

Count

获取模型记录数

```
db.Where("name = ?", "jinzhu").Or("name = ?", "jinzhu 2").Find(&users).Count(&co
unt)
//// SELECT * from USERS WHERE name = 'jinzhu' OR name = 'jinzhu 2'; (users)
//// SELECT count(*) FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2'; (cou
nt)

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;
```

注意： 在查询链中使用 `Count` 时，必须放在最后一个位置，因为它会覆盖 `SELECT` 查询条件。

Group 和 Having

```
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) as 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").Group("date(created_at)").Having("sum(amount) > ?", 100).Scan(&results)
```

Joins

指定关联条件

```
rows, err := db.Table("users").Select("users.name, emails.email").Joins("left join 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)

// 多个关联查询
db.Joins("JOIN emails ON emails.user_id = users.id AND emails.email = ?", "jinzhu@example.org").Joins("JOIN credit_cards ON credit_cards.user_id = users.id").Where("credit_cards.number = ?", "411111111111").Find(&user)
```

Pluck

使用 Pluck 从模型中查询单个列作为集合。如果想查询多个列，应该使用 `Scan` 代替。

```
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)

// Requesting more than one column? Do it like this:
db.Select("name, age").Find(&users)
```

Scan

将 Scan 查询结果放入另一个结构体中。

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

var result Result
db.Table("users").Select("name, age").Where("name = ?", 3).Scan(&result)

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

更新

更新所有字段

`Save` 方法在执行 SQL 更新操作时将包含所有字段，即使这些字段没有被修改。

```
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;
```

更新已更改的字段

如果你只想更新已经修改了的字段，可以使用 `Update` ， `Updates` 方法。

```
// 如果单个属性被更改了，更新它
db.Model(&user).Update("name", "hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111;

// 使用组合条件更新单个属性
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;

// 使用 `map` 更新多个属性，只会更新那些被更改了的字段
db.Model(&user).Updates(map[string]interface{}{"name": "hello", "age": 18, "actived": false})
//// UPDATE users SET name='hello', age=18, actived=false, updated_at='2013-11-17 21:34:10' WHERE id=111;

// 使用 `struct` 更新多个属性，只会更新那些被修改了的和非空的字段
db.Model(&user).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18, updated_at = '2013-11-17 21:34:10' WHERE id = 111;

// 警告： 当使用结构体更新的时候，GORM 只会更新那些非空的字段
```

```
// 例如下面的更新，没有东西会被更新，因为像 "", 0, false 是这些字段类型的空值
db.Model(&user).Updates(User{Name: "", Age: 0, Actived: false})
```

更新选中的字段

如果你在执行更新操作时只想更新或者忽略某些字段，可以使用 `Select` ， `Omit` 方法。

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

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

更新列钩子方法

上面的更新操作更新时会执行模型的 `BeforeUpdate` 和 `AfterUpdate` 方法，来更新 `UpdatedAt` 时间戳，并且保存他的 `关联` 。如果你不想执行这些操作，可以使用 `UpdateColumn` ， `UpdateColumns` 方法。

```
// Update single attribute, similar with `Update`
db.Model(&user).UpdateColumn("name", "hello")
//// UPDATE users SET name='hello' WHERE id = 111;

// Update multiple attributes, similar with `Updates`
db.Model(&user).UpdateColumns(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18 WHERE id = 111;
```

批量更新

批量更新时，钩子函数不会执行

```
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);

// 使用结构体更新将只适用于非零值，或者使用 map[string]interface{}
```

```
db.Model(User{}).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18;

// 使用 `RowsAffected` 获取更新影响的记录数
db.Model(User{}).Updates(User{Name: "hello", Age: 18}).RowsAffected
```

带有表达式的 SQL 更新

```
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("quantity - ?", 1))
//// UPDATE "products" SET "quantity" = quantity - 1 WHERE "id" = '2' AND quantity > 1;
```

在钩子函数中更新值

如果你想使用 `BeforeUpdate` 、 `BeforeSave` 钩子函数修改更新的值，可以使用 `scope.SetColumn` 方法，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)
  }
}
```

额外的更新选项

```
// 在更新 SQL 语句中添加额外的 SQL 选项
db.Model(&user).Set("gorm:update_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Update("name", "hello")
```

```
//// UPDATE users SET name='hello', updated_at = '2013-11-17 21:34:10' WHERE id=
111 OPTION (OPTIMIZE FOR UNKNOWN);
```

删除

删除记录

警告：当删除一条记录的时候，你需要确定这条记录的主键有值，GORM会使用主键来删除这条记录。如果主键字段为空，GORM会删除模型中所有的记录。

```
// 删除一条存在的记录
db.Delete(&email)
//// DELETE from emails where id=10;

// 为删除 SQL 语句添加额外选项
db.Set("gorm:delete_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Delete(&email)
//// DELETE from emails where id=10 OPTION (OPTIMIZE FOR UNKNOWN);
```

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%";
```

软删除

如果模型中有 `DeletedAt` 字段，它将自动拥有软删除的能力！当执行删除操作时，数据并不会永久的从数据库中删除，而是将 `DeletedAt` 的值更新为当前时间。

```
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;

// 在查询记录时，软删除记录会被忽略
db.Where("age = 20").Find(&user)
```

```
//// SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;

// 使用 Unscoped 方法查找软删除记录
db.Unscoped().Where("age = 20").Find(&users)
//// SELECT * FROM users WHERE age = 20;

// 使用 Unscoped 方法永久删除记录
db.Unscoped().Delete(&order)
//// DELETE FROM orders WHERE id=10;
```

关联

Belongs To

Has One

Has Many

Many To Many

关联

预加载

Belongs To

属于

`belongs to` 关联建立一个和另一个模型的一对一连接，使得模型声明每个实例都「属于」另一个模型的一个实例 。

例如，如果你的应用包含了用户和用户资料， 并且每一个用户资料只分配给一个用户

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

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

外键

为了定义从属关系， 外键是必须存在的， 默认的外键使用所有者类型名称加上其主键。

像上面的例子，为了声明一个模型属于 `User` ，它的外键应该为 `UserID` 。

GORM 提供了一个定制外键的方法，例如:

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

type Profile struct {
    gorm.Model
  Name      string
  User      User `gorm:"foreignkey:UserRefer"` // 使用 UserRefer 作为外键
  UserRefer string
}
```

关联外键

对于从属关系， GORM 通常使用所有者的主键作为外键值，在上面的例子中，就是 `User` 的 `ID` 。

当你分配一个资料给一个用户， GORM 将保存用户表的 `ID` 值 到 用户资料表的 `UserID` 字段里。

你可以通过改变标签 `association_foreignkey` 来改变它， 例如：

```
type User struct {
    gorm.Model
  Refer int
    Name string
}

type Profile struct {
    gorm.Model
  Name      string
  User      User `gorm:"association_foreignkey:Refer"` // use Refer 作为关联外键
  UserRefer string
}
```

使用属于

你能找到 `belongs to` 和 `Related` 的关联

```
db.Model(&user).Related(&profile)
//// SELECT * FROM profiles WHERE user_id = 111; // 111 is user's ID
```

更多高级用法，请参考 Association Mode

Has One

Has One

`has one` 关联也是与另一个模型建立一对一的连接，但语义（和结果）有些不同。 此关联表示模型的每个实例包含或拥有另一个模型的一个实例。

例如，如果你的应用程序包含用户和信用卡，并且每个用户只能有一张信用卡。

```
// 用户有一个信用卡，CredtCardID 外键
type User struct {
    gorm.Model
    CreditCard   CreditCard
}

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

外键

对于一对一关系，一个外键字段也必须存在，所有者将保存主键到模型关联的字段里。

这个字段的名字通常由 `belongs to model` 的类型加上它的 `primary key` 产生的，就上面的例子而言，它就是 `CreditCardID`

当你给用户一个信用卡， 它将保存一个信用卡的 `ID` 到 `CreditCardID` 字段中。

如果你想使用另一个字段来保存这个关系，你可以通过使用标签 `foreignkey` 来改变它，例如：

```
type User struct {
  gorm.Model
  CreditCard CreditCard `gorm:"foreignkey:CardRefer"`
}

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

```
    UserName string
}
```

关联外键

通常，所有者会保存 `belogns to model` 的主键到外键，你可以改为保存其他字段， 就像下面的例子使用 `Number` 。

```
type User struct {
  gorm.Model
  CreditCard CreditCard `gorm:"association_foreignkey:Number"`
}

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

多态关联

支持多态的一对多和一对一关联。

```
  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
  }
```

注意：多态属于和多对多是明确的不支持并将会抛出错误。

使用一对一

你可以通过 `Related` 找到 `has one` 关联。

```
var card CreditCard
db.Model(&user).Related(&card, "CreditCard")
//// SELECT * FROM credit_cards WHERE user_id = 123; // 123 是用户表的主键
// CreditCard  是用户表的字段名，这意味着获取用户的信用卡关系并写入变量 card。
// 像上面的例子，如果字段名和变量类型名一样，它就可以省略， 像：
db.Model(&user).Related(&card)
```

更多高级用法，请参考 Association Mode

Has Many

一对多

`has many` 关联就是创建和另一个模型的一对多关系， 不像 `has one` ，所有者可以拥有0个或多个模型实例。

例如，如果你的应用包含用户和信用卡， 并且每一个用户都拥有多张信用卡。

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

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

外键

为了定义一对多关系， 外键是必须存在的，默认外键的名字是所有者类型的名字加上它的主键。

就像上面的例子，为了定义一个属于 `User` 的模型，外键就应该为 `UserID` 。

使用其他的字段名作为外键， 你可以通过 `foreignkey` 来定制它， 例如:

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

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

外键关联

GORM 通常使用所有者的主键作为外键的值， 在上面的例子中，它就是 `User` 的 `ID` 。

当你分配信用卡给一个用户， GORM 将保存用户 `ID` 到信用卡表的 `UserID` 字段中。

你能通过 `association_foreignkey` 来改变它， 例如:

```
type User struct {
    gorm.Model
  MemberNumber string
    CreditCards  []CreditCard `gorm:"foreignkey:UserMemberNumber;association_for
eignkey:MemberNumber"`
}

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

多态关联

支持多态的一对多和一对一关联。

```
  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
  }
```

注意：多态属于和多对多是明确不支持并会抛出错误的。

使用一对多

你可以通过 `Related` 找到 `has many` 关联关系。

```
db.Model(&user).Related(&emails)
//// SELECT * FROM emails WHERE user_id = 111; // 111 是用户表的主键
```

更多高级用法， 请参考 Association Mode

Many To Many

多对多

多对多为两个模型增加了一个中间表。

例如，如果你的应用包含用户和语言， 一个用户会说多种语言，并且很多用户会说一种特定的语言。

```
// 用户拥有并属于多种语言， 使用 `user_languages` 作为中间表
type User struct {
    gorm.Model
    Languages         []Language `gorm:"many2many:user_languages;"`
}

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

反向关联

```
// 用户拥有并且属于多种语言，使用 `user_languages` 作为中间表
type User struct {
    gorm.Model
    Languages         []*Language `gorm:"many2many:user_languages;"`
}

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

db.Model(&language).Related(&users)
//// SELECT * FROM "users" INNER JOIN "user_languages" ON "user_languages"."user_id" = "users"."id" WHERE  ("user_languages"."language_id" IN ('111'))
```

外键

```
type CustomizePerson struct {
  IdPerson string             `gorm:"primary_key:true"`
  Accounts []CustomizeAccount `gorm:"many2many:PersonAccount;association_foreign
key:idAccount;foreignkey:idPerson"`
}

type CustomizeAccount struct {
  IdAccount string `gorm:"primary_key:true"`
  Name      string
}
```

外键会为两个结构体创建一个多对多的关系，并且这个关系将通过外键 `customize_person_id_person` 和 `customize_account_id_account` 保存到中间表 `PersonAccount` 。

中间表外键

如果你想改变中间表的外键，你可以用标签 `association_jointable_foreignkey` , `jointable_foreignkey`

```
type CustomizePerson struct {
  IdPerson string             `gorm:"primary_key:true"`
  Accounts []CustomizeAccount `gorm:"many2many:PersonAccount;foreignkey:idPerso
n;association_foreignkey:idAccount;association_jointable_foreignkey:account_id;j
ointable_foreignkey:person_id;"`
}

type CustomizeAccount struct {
  IdAccount string `gorm:"primary_key:true"`
  Name      string
}
```

自引用

为了定义一个自引用的多对多关系，你不得不改变中间表的关联外键。

和来源表外键不同的是它是通过结构体的名字和主键生成的，例如：

```
type User struct {
  gorm.Model
  Friends []*User `gorm:"many2many:friendships;association_jointable_foreignkey:
```

```
friend_id"`
}
```

GORM 将创建一个带外键 `user_id` 和 `friend_id` 的中间表， 并且使用它去保存用户表的自引用关系。

然后你可以像普通关系一样操作它， 例如：

```
DB.Preload("Friends").First(&user, "id = ?", 1)

DB.Model(&user).Association("Friends").Append(&User{Name: "friend1"}, &User{Name: "friend2"})

DB.Model(&user).Association("Friends").Delete(&User{Name: "friend2"})

DB.Model(&user).Association("Friends").Replace(&User{Name: "new friend"})

DB.Model(&user).Association("Friends").Clear()

DB.Model(&user).Association("Friends").Count()
```

使用多对多

```
db.Model(&user).Related(&languages, "Languages")
//// SELECT * FROM "languages" INNER JOIN "user_languages" ON "user_languages"."language_id" = "languages"."id" WHERE "user_languages"."user_id" = 111

//  当查询用户时预加载 Language
db.Preload("Languages").First(&user)
```

更多高级用法， 请参考 Association Mode

关联

自动创建/更新

GORM 将在创建或保存一条记录的时候自动保存关联和它的引用，如果关联有一个主键，GORM 将调用 `Update` 来更新它， 不然，它将会被创建。

```
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@example.com"},
    },
    Languages:       []Language{
        {Name: "ZH"},
        {Name: "EN"},
    },
}

db.Create(&user)
//// BEGIN TRANSACTION;
//// INSERT INTO "addresses" (address1) VALUES ("Billing Address - Address 1");
//// INSERT INTO "addresses" (address1) VALUES ("Shipping Address - Address 1");
//// INSERT INTO "users" (name,billing_address_id,shipping_address_id) VALUES (
"jinzhu", 1, 2);
//// INSERT INTO "emails" (user_id,email) VALUES (111, "jinzhu@example.com");
//// INSERT INTO "emails" (user_id,email) VALUES (111, "jinzhu-2@example.com");
//// INSERT INTO "languages" ("name") VALUES ('ZH');
//// INSERT INTO user_languages ("user_id","language_id") VALUES (111, 1);
//// INSERT INTO "languages" ("name") VALUES ('EN');
//// INSERT INTO user_languages ("user_id","language_id") VALUES (111, 2);
//// COMMIT;

db.Save(&user)
```

关闭自动更新

如果你的关联记录已经存在在数据库中， 你可能会不想去更新它。

你可以设置 `gorm:association_autoupdate` 为 `false`

```
// 不更新有主键的关联，但会更新引用
db.Set("gorm:association_autoupdate", false).Create(&user)
db.Set("gorm:association_autoupdate", false).Save(&user)
```

或者使用 GORM 的标签， `gorm:"association_autoupdate:false"`

```
type User struct {
  gorm.Model
  Name       string
  CompanyID  uint
  // 不更新有主键的关联，但会更新引用
  Company    Company `gorm:"association_autoupdate:false"`
}
```

关闭自动创建

即使你禁用了 `AutoUpdating` ，仍然会创建没有主键的关联，并保存它的引用。

你可以通过把 `gorm:association_autocreate` 设置为 `false` 来禁用这个行为。

```
// 不创建没有主键的关联，不保存它的引用。
db.Set("gorm:association_autocreate", false).Create(&user)
db.Set("gorm:association_autocreate", false).Save(&user)
```

或者使用 GORM 标签， `gorm:"association_autocreate:false"`

```
type User struct {
  gorm.Model
  Name       string
  // 不创建没有主键的关联，不保存它的引用。
  Company1   Company `gorm:"association_autocreate:false"`
}
```

关闭自动创建/更新

禁用 `AutoCreate` 和 `AutoUpdate` ，你可以一起使用它们两个的设置。

```
db.Set("gorm:association_autoupdate", false).Set("gorm:association_autocreate",
false).Create(&user)
```

```
type User struct {
  gorm.Model
  Name    string
  Company Company `gorm:"association_autoupdate:false;association_autocreate:false"`
}
```

或者使用 `gorm:save_associations`

```
db.Set("gorm:save_associations", false).Create(&user)
db.Set("gorm:save_associations", false).Save(&user)

type User struct {
  gorm.Model
  Name    string
  Company Company `gorm:"association_autoupdate:false"`
}
```

关闭保存引用

如果你不想当更新或保存数据的时候保存关联的引用， 你可以使用下面的技巧

```
db.Set("gorm:association_save_reference", false).Save(&user)
db.Set("gorm:association_save_reference", false).Create(&user)
```

或者使用标签

```
type User struct {
  gorm.Model
  Name       string
  CompanyID  uint
  Company    Company `gorm:"association_save_reference:false"`
}
```

关联模式

关联模式包含一些可以轻松处理与关系相关的事情的辅助方法。

```
// 开启关联模式
var user User
```

```
db.Model(&user).Association("Languages")
// `user` 是源表，必须包含主键
// `Languages` 是源表关系字段名称。
// 只有上面两个条件都能匹配，关联模式才会生效， 检查是否正常：
// db.Model(&user).Association("Languages").Error
```

查找关联

查找匹配的关联

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

增加关联

为 `many to many` ， `has many` 新增关联， 为 `has one` ， `belongs to` 替换当前关联

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

替换关联

用一个新的关联替换当前关联

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

删除关联

删除源和参数对象之间的关系， 只会删除引用，不会删除他们在数据库中的对象。

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

清理关联

删除源和当前关联之间的引用，不会删除他们的关联

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

统计关联

返回当前关联的统计数

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

预加载

预加载

```
db.Preload("Orders").Find(&users)
//// SELECT * FROM users;
//// SELECT * FROM orders WHERE user_id IN (1,2,3,4);

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 ('cancelle
d');

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');

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
```

自动预加载

始终自动预加载的关联

```
type User struct {
  gorm.Model
  Name       string
  CompanyID  uint
  Company    Company `gorm:"PRELOAD:false"` //没有预加载
  Role       Role                           // 已经预加载
}

db.Set("gorm:auto_preload", true).Find(&users)
```

嵌套预加载

```
db.Preload("Orders.OrderItems").Find(&users)
db.Preload("Orders", "state = ?", "paid").Preload("Orders.OrderItems").Find(&users)
```

自定义预加载 SQL

您可以通过传入 `func（db * gorm.DB）* gorm.DB` 来自定义预加载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;
```

教程

链式操作

错误处理

钩子

事务

数据库迁移

原生 SQL 和 SQL 生成器

通用数据库接口

链式操作

链式操作

Gorm 继承了链式操作接口， 所以你可以写像下面一样的代码：

```
db, err := gorm.Open("postgres", "user=gorm dbname=gorm sslmode=disable")

// 创建一个新的关系
tx := db.Where("name = ?", "jinzhu")

// 新增更多的筛选条件
if someCondition {
    tx = tx.Where("age = ?", 20)
} else {
    tx = tx.Where("age = ?", 30)
}

if yetAnotherCondition {
    tx = tx.Where("active = ?", 1)
}
```

直到调用立即方法之前都不会产生查询，在某些场景中会很有用。

就像你可以封装一个包来处理一些常见的逻辑

创建方法

创建方法就是那些会产生 SQL 查询并且发送到数据库，通常它就是一些 CRUD 方法， 就像:

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

下面是一个创建方法的例子:

```
tx.Find(&user)
```

生成

```
SELECT * FROM users where name = 'jinzhu' AND age = 30 AND active = 1;
```

Scopes 方法

Scope 方法基于链式操作理论创建的。

使用它，你可以提取一些通用逻辑，写一些更可用的库。

```
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.Scopes(AmountGreaterThan1000).Where("status in (?)", status)
    }
}

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

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

db.Scopes(AmountGreaterThan1000, OrderStatus([]string{"paid", "shipped"})).Find
(&orders)
// 查找大于1000的所有付费和运单
```

多个创建方法

当使用 GORM 的创建方法，后面的创建方法将复用前面的创建方法的搜索条件（不包含内联条件）

```
db.Where("name LIKE ?", "jinzhu%").Find(&users, "id IN (?)", []int{1, 2, 3}).Cou
nt(&count)
```

生成

```
SELECT * FROM users WHERE name LIKE 'jinzhu%' AND id IN (1, 2, 3)

SELECT count(*) FROM users WHERE name LIKE 'jinzhu%'
```

线程安全

所有的链式操作都将会克隆并创建一个新的数据库对象（共享一个连接池），GORM 对于多个 goroutines 的并发使用是安全的。

错误处理

在 Go 语言中，错误处理是很重要的。

Go 语言中鼓励人们在任何 创建方法之后去检查错误。

错误处理

由于 GORM 的 链式 API，GORM 中的错误处理与惯用的 Go 代码不同，但它仍然相当容易。

如果发生任何错误，GORM 会将其设置为 `*gorm.DB` 的 `Error` 字段，你可以这样检查：

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

或者

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

错误

在处理数据期间，发生几个错误很普遍，GORM 提供了一个 API 来将所有发生的错误作为切片返回

```
// 如果有多个错误产生，`GetErrors` 返回一个 `[]error`的切片
db.First(&user).Limit(10).Find(&users).GetErrors()

fmt.Println(len(errors))

for _, err := range errors {
  fmt.Println(err)
}
```

RecordNotFound 错误

GORM 提供了一个处理 `RecordNotFound` 错误的快捷方式，如果发生了多个错误，它将检查每个错误，如果它们中的任何一个是 `RecordNotFound` 错误。

```
//检查是否返回 RecordNotFound 错误
db.Where("name = ?", "hello world").First(&user).RecordNotFound()

if db.Model(&user).Related(&credit_card).RecordNotFound() {
  // 数据没有找到
}

if err := db.Where("name = ?", "jinzhu").First(&user).Error; gorm.IsRecordNotFou
ndError(err) {
  // 数据没有找到
}
```

钩子

对象的生命周期

钩子是一个在 插入/查询/更新/删除 之前或之后被调用的方法。

如果你在一个模型中定义了特殊的方法，它将会在插入，更新，查询，删除的时候被自动调用，如果任何的回调抛出错误，GORM 将会停止将要执行的操作并且回滚当前的改变。

钩子

创建一个对象

可用于创建的钩子

```
// 开启事务
BeforeSave
BeforeCreate
// 连表前的保存
// 更新时间戳 `CreatedAt`, `UpdatedAt`
// 保存自己
// 重载哪些有默认值和空的字段
// 链表后的保存
AfterCreate
AfterSave
// 提交或回滚事务
```

代码例子:

```
func (u *User) BeforeSave() (err error) {
    if u.IsValid() {
        err = errors.New("can't save invalid data")
    }
    return
}

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

```
    return
}
```

注意，在 GORM 中的保存/删除 操作会默认进行事务处理，所以在事物中，所有的改变都是无效的，直到它被提交为止:

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

更新一个对象

可用于更新的钩子

```
// 开启事务
BeforeSave
BeforeUpdate
// 链表前的保存
// 更新时间戳 `UpdatedAt`
// 保存自身
// 链表后的保存
AfterUpdate
AfterSave
// 提交或回滚的事务
```

代码示例:

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

// 在事务结束后，进行更新数据
func (u *User) AfterUpdate(tx *gorm.DB) (err error) {
  if u.Confirmed {
    tx.Model(&Address{}).Where("user_id = ?", u.ID).Update("verfied", true)
  }
    return
}
```

删除一个对象

可用于删除的钩子

```
// 开启事务
BeforeDelete
// 删除自身
AfterDelete
// 提交或回滚事务
```

代码示例:

```
// 在事务结束后进行更新数据
func (u *User) AfterDelete(tx *gorm.DB) (err error) {
  if u.Confirmed {
    tx.Model(&Address{}).Where("user_id = ?", u.ID).Update("invalid", false)
  }
    return
}
```

查询一个对象

可用于查询的钩子

```
// 从数据库中读取数据
// 加载之前（急于加载）
AfterFind
```

代码示例:

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

事务

GORM 默认在事务中执行单个 `create` ， `update` ， `delete` 操作，以确保数据库数据完整性。

如果你想将多个 `create` ， `update` ， `delete` 当成一个原子性操作， `Transaction` 就是为了这个而创造的。

事务

要在事务中执行一组操作，正常的流程如下所示。

```
// 开启事务
tx := db.Begin()

// 在事务中执行一些数据库操作 （从这里开始使用 'tx'，而不是 'db'）
tx.Create(...)

// ...

// 发生错误回滚事务
tx.Rollback()

// 或者提交这个事务
tx.Commit()
```

具体例子

```
func CreateAnimals(db *gorm.DB) err {
  // 注意在事务中要使用 tx 作为数据库句柄
  tx := db.Begin()
  defer func() {
    if r := recover(); r != nil {
      tx.Rollback()
    }
  }()

  if tx.Error != 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
}
```

数据库迁移

自动迁移

使用 migrate 来维持你的表结构一直处于最新状态。

警告：migrate 仅支持创建表、增加表中没有的字段和索引。为了保护你的数据，它并不支持改变已有的字段类型或删除未被使用的字段

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

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

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

其他数据库迁移工具

GORM 的数据库迁移工具能够支持主要的数据库，但是如果你要寻找更多的迁移工具， GORM 会提供的数据库接口，这可能可以给到你帮助。

```
// 返回 `*sql.DB`
db.DB()
```

参考 通用接口 以获得更多详细说明

表结构的方法

Has Table

```
// 检查模型中 User 表是否存在
db.HasTable(&User{})

// 检查 users 表是否存在
db.HasTable("users")
```

Create Table

```
// 通过模型 User 创建表
db.CreateTable(&User{})

// 在创建 users 表的时候，会在 SQL 语句中拼接上 `"ENGINE=InnoDB"`
db.Set("gorm:table_options", "ENGINE=InnoDB").CreateTable(&User{})
```

Drop table

```
// 删除模型 User 表
db.DropTable(&User{})

// 删除 users 表
db.DropTable("users")

// 删除模型 User 表和 products 表
db.DropTableIfExists(&User{}, "products")
```

ModifyColumn

以给定的值来定义字段类型

```
// User 模型，改变 description 字段的数据类型为 `text`
db.Model(&User{}).ModifyColumn("description", "text")
```

DropColumn

```
// User 模型，删除 description 字段
db.Model(&User{}).DropColumn("description")
```

Add Indexes

```
// 为 `name` 字段建立一个名叫 `idx_user_name` 的索引
db.Model(&User{}).AddIndex("idx_user_name", "name")

// 为 `name`, `age` 字段建立一个名叫 `idx_user_name_age` 的索引
db.Model(&User{}).AddIndex("idx_user_name_age", "name", "age")

// 添加一条唯一索引
db.Model(&User{}).AddUniqueIndex("idx_user_name", "name")
```

```
// 为多个字段添加唯一索引
db.Model(&User{}).AddUniqueIndex("idx_user_name_age", "name", "age")
```

Remove Index

```
// 移除索引
db.Model(&User{}).RemoveIndex("idx_user_name")
```

Add Foreign Key

```
// 添加主键
// 第一个参数 : 主键的字段
// 第二个参数 : 目标表的 ID
// 第三个参数 : ONDELETE
// 第四个参数 : ONUPDATE
db.Model(&User{}).AddForeignKey("city_id", "cities(id)", "RESTRICT", "RESTRICT")
```

Remove ForeignKey

```
db.Model(&User{}).RemoveForeignKey("city_id", "cities(id)")
```

原生 SQL 和 SQL 生成器

运行原生 SQL

执行原生 SQL时不能通过链式调用其他方法

```
db.Exec("DROP TABLE users;")
db.Exec("UPDATE orders SET shipped_at=? WHERE id IN (?)", time.Now(), []int64{1
1,22,33})

// Scan
type Result struct {
    Name string
    Age  int
}

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

`sql.Row` 和 `sql.Rows`

使用 `*sql.Row` 或者 `*sql.Rows` 获得查询结果

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

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

// 原生SQL
rows, err := db.Raw("select name, age, email from users where name = ?", "jinzh
u").Rows() // (*sql.Rows, error)
defer rows.Close()
for rows.Next() {
```

```
    ...
    rows.Scan(&name, &age, &email)
    ...
}
```

扫描 `sql.Rows` 数据到模型

```
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 扫描一行到 user 模型
  db.ScanRows(rows, &user)

  // do something
}
```

通用数据库接口

GORM 提供了从当前的 `*gorm.DB` 连接中返回通用的数据库接口的方法 `DB` *sql.DB 。

```
// 获取通用数据库对象 sql.DB 来使用他的 db.DB() 方法

// Ping
db.DB().Ping()
```

注意： 如果底层的数据库连接不是 `*sql.DB` 。就像在事务中，它将返回 nil。

连接池

```
// SetMaxIdleConns 设置空闲连接池中的最大连接数。
db.DB().SetMaxIdleConns(10)

// SetMaxOpenConns 设置数据库连接最大打开数。
db.DB().SetMaxOpenConns(100)

// SetConnMaxLifetime 设置可重用连接的最长时间
db.DB().SetConnMaxLifetime(time.Hour)
```

高级主题

复合主键

创建插件

GORM Dialects

自定义 Logger

更新日志

复合主键

可以设置多个字段为主键来开启复合主键功能：

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

创建插件

GORM 本身由 `Callbacks` 提供支持，因此你可以根据需要完全自定义GORM。

注册新的 callback

将 callback 注册进如 callbacks：

```
func updateCreated(scope *Scope) {
    if scope.HasColumn("Created") {
        scope.SetColumn("Created", NowFunc())
    }
}

db.Callback().Create().Register("update_created_at", updateCreated)
// 注册 Create 进程的回调
```

删除已有的 callback

从 callbacks 中删除一个 callback：

```
db.Callback().Create().Remove("gorm:create")
// delete callback `gorm:create` from Create callbacks
```

替换 callback

替换拥有相同名字的 callback ：

```
db.Callback().Create().Replace("gorm:create", newCreateFunction)
// replace callback `gorm:create` with new function `newCreateFunction` for Crea
te process
```

注册 callback 的顺序

在注册 callbacks 时设置顺序：

```
db.Callback().Create().Before("gorm:create").Register("update_created_at", updat
eCreated)
db.Callback().Create().After("gorm:create").Register("update_created_at", update
```

```
Created)
db.Callback().Query().After("gorm:query").Register("my_plugin:after_query", afte
rQuery)
db.Callback().Delete().After("gorm:delete").Register("my_plugin:after_delete", a
fterDelete)
db.Callback().Update().Before("gorm:update").Register("my_plugin:before_update",
beforeUpdate)
db.Callback().Create().Before("gorm:create").After("gorm:before_create").Registe
r("my_plugin:before_create", beforeCreate)
```

自带的 Callbacks

GORM 在处理 CRUD 操作时自带了一些 Callback，建议你在写插件前先熟悉这些 Callback：

- Create callbacks
- Update callbacks
- Query callbacks
- Delete callbacks
- Row Query callbacks - 默认没有注册的 Callbacks

你可以用以下的方法来注册你的 Callback：

```
func updateTableName(scope *gorm.Scope) {
  scope.Search.Table(scope.TableName() + "_draft") // append `_draft` to table n
ame
}

db.Callback().RowQuery().Register("publish:update_table_name", updateTableName)
```

请前往查看所有的 API —— https://godoc.org/github.com/jinzhu/gorm 。

GORM Dialects

编写一个新的 Dialect

GORM 原生支持 sqlite, mysql, postgres 和 mssql。

你可以通过实现 dialect interface 接口，来新增对某个新的数据库的支持。

有一些关系型数据库与 mysql 和 postgres 语法兼容，因此你可以直接使用这两个数据库的 dialect 。

Dialect 专属的数据类型

一些 SQL 语法的 dialects 支持他们自定义的，非标准的字段类型，如 PostgreSQL 中的 `jsonb` 字段类型。GORM 支持一些类似此种类型，下面是这些类型的简单实用范例。

PostgreSQL

GORM 支持 PostgreSQL 专有的字段类型：

- jsonb
- hstore

以下这是 Model 的定义：

```
import (
    "encoding/json"
    "github.com/jinzhu/gorm/dialects/postgres"
)

type Document struct {
    Metadata postgres.Jsonb
    Secrets  postgres.Hstore
    Body     string
    ID       int
}
```

你可以这样子使用 Model：

```
password := "0654857340"
metadata := json.RawMessage(`{"is_archived": 0}`)
```

```
sampleDoc := Document{
  Body: "This is a test document",
  Metadata: postgres.Jsonb{ metadata },
  Secrets: postgres.Hstore{"password": &password},
}

// 将范例数据写入数据库
db.Create(&sampleDoc)

// 读取数据，来检测是否正确写入
resultDoc := Document{}
db.Where("id = ?", sampleDoc.ID).First(&resultDoc)

metadataIsEqual := reflect.DeepEqual(resultDoc.Metadata, sampleDoc.Metadata)
secretsIsEqual := reflect.DeepEqual(resultDoc.Secrets, resultDoc.Secrets)

// 应该打印 "true"
fmt.Println("Inserted fields are as expected:", metadataIsEqual && secretsIsEqua
l)
```

自定义 Logger

Logger

Gorm 建立了对 Logger 的支持，默认模式只会在错误发生的时候打印日志。

```
// 开启 Logger，以展示详细的日志
db.LogMode(true)

// 关闭 Logger，不再展示任何日志，即使是错误日志
db.LogMode(false)

// 对某个操作展示详细的日志，用来排查该操作的问题
db.Debug().Where("name = ?", "jinzhu").First(&User{})
```

自定义 Logger

参考 GORM 的默认 logger 是怎么自定义的
https://github.com/jinzhu/gorm/blob/master/logger.go

例如，使用 Revel 的 Logger 作为 GORM 的输出

```
db.SetLogger(gorm.Logger{revel.TRACE})
```

使用 `os.Stdout` 作为输出

```
db.SetLogger(log.New(os.Stdout, "\r\n", 0))
```

更新日志

v2.0

WIP

v1.0 - 2016.04.27

破坏性变更

- `gorm.Open` 返回类型为 `*gorm.DB` 而不是 `gorm.DB`；
- 更新只会更新更改的字段
- 只会使用 `deleted_at IS NULL` 来检测软删除
- 新的 ToDBName 逻辑

在 GORM 将 struct，Field 的名称转换为 db 名称之前，只有那些来自 golint 的常见初始化（如 `HTTP`，`URI`）是特殊处理的，所以 `HTTP` 的数据库名称是 `http`，而不是 `h_t_t_p`。

但是像一些不在列表里的缩写，如 `SKU` db 名称为 `s_k_u`，这次升级修复了此问题。

- 错误 `RecordNotFound` 重命名为 `ErrRecordNotFound`。
- `mssql` dialect 被重命名为 “github.com/jinzhu/gorm/dialects/mssql”
- `Hstore` 字段类型被移到专属的包里 “github.com/jinzhu/gorm/dialects/postgres”