gorm文档(英文)

书栈(BookStack.CN)

目　录

致谢

当前文档 《gorm文档(英文)》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-01-21。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN)，为知识的传承献上您的一份力量!

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/gorm_en

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

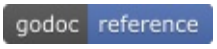
0. Summary

Summary

1. Getting Started with GORM

GORM

The fantastic ORM library for Golang, aims to be developer friendly.



Overview

- Full-Featured ORM (almost)
- Associations (Has One, Has Many, Belongs To, Many To Many, Polymorphism)
- Callbacks (Before/After Create/Save/Update/Delete/Find)
- Preloading (eager loading)
- Transactions
- Composite Primary Key
- SQL Builder
- Auto Migrations
- Logger
- Extendable, write Plugins based on GORM callbacks
- Every feature comes with tests
- Developer Friendly

Install

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

Upgrading To V1.0

- CHANGELOG

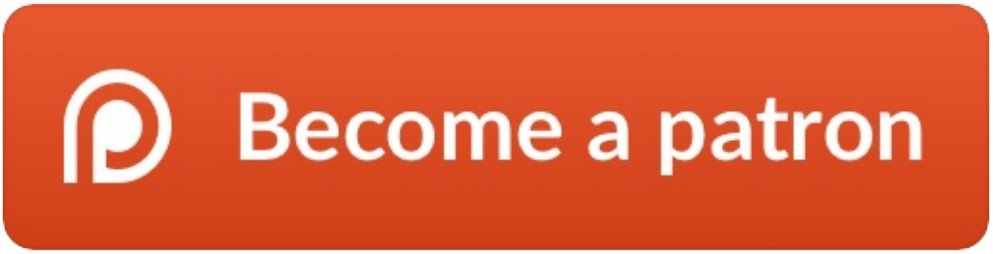
Quick Start

```go
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()

  // Migrate the schema
  db.AutoMigrate(&Product{})

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

  // Read
  var product Product
  db.First(&product, 1) // find product with id 1
  db.First(&product, "code = ?", "L1212") // find product with code l1212

  // Update - update product's price to 2000
  db.Model(&product).Update("Price", 2000)

  // Delete - delete product
  db.Delete(&product)
}
```

Contributors

https://github.com/jinzhu/gorm/graphs/contributors

Supporting the project



Author

jinzhu

- http://github.com/jinzhu
- wosmvp@gmail.com
- http://twitter.com/zhangjinzhu

License

Released under the MIT License.

2. Database

Database

Connecting to a database

In order to connect to a database, you need to import the database's driver first. For example:

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

GORM has wrapped some drivers, for easier to remember the import path, so you could import the mysql driver with

```
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"
```

MySQL

NOTE: In order to handle `time.Time` , you need to include `parseTime` as a parameter. (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 user=gorm dbname=gorm sslmode=disable
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, it can running on your Mac, Linux with Docker

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

```
5.
6.  func main() {
7.    db, err = gorm.Open("mssql", "sqlserver://username:password@localhost:1433?
    database=dbname")
8.    defer db.Close()
9.  }
```

Write Dialect for unsupported databases

GORM officially supports the above databases, but you could write a dialect for unsupported databases.

To write your own dialect, refer to: https://github.com/jinzhu/gorm/blob/master/dialect.go

Migration

Auto Migration

Automatically migrate your schema, to keep your schema update to date.

WARNING: AutoMigrate will **ONLY** create tables, missing columns and missing indexes, and **WON'T** change existing column's type or delete unused columns to protect your data.

```
1.  db.AutoMigrate(&User{})
2.
3.  db.AutoMigrate(&User{}, &Product{}, &Order{})
4.
5.  // Add table suffix when create tables
6.  db.Set("gorm:table_options", "ENGINE=InnoDB").AutoMigrate(&User{})
```

Has Table

```
1.  // Check model `User`'s table exists or not
2.  db.HasTable(&User{})
3.
4.  // Check table `users` exists or not
5.  db.HasTable("users")
```

Create Table

```
1.  // Create table for model `User`
2.  db.CreateTable(&User{})
3.
```

```
// will append "ENGINE=InnoDB" to the SQL statement when creating table `users`
db.Set("gorm:table_options", "ENGINE=InnoDB").CreateTable(&User{})
```

Drop table

```
// Drop model `User`'s table
db.DropTable(&User{})

// Drop table `users`
db.DropTable("users")

// Drop model's `User`'s table and table `products`
db.DropTableIfExists(&User{}, "products")
```

ModifyColumn

Modify column's type to given value

```
// change column description's data type to `text` for model `User`
db.Model(&User{}).ModifyColumn("description", "text")
```

DropColumn

```
// Drop column description from model `User`
db.Model(&User{}).DropColumn("description")
```

Add Foreign Key

```
// Add foreign key
// 1st param : foreignkey field
// 2nd param : destination table(id)
// 3rd param : ONDELETE
// 4th param : ONUPDATE
db.Model(&User{}).AddForeignKey("city_id", "cities(id)", "RESTRICT", "RESTRICT")
```

Indexes

```
// Add index for columns `name` with given name `idx_user_name`
db.Model(&User{}).AddIndex("idx_user_name", "name")

// Add index for columns `name`, `age` with given name `idx_user_name_age`
```

```
db.Model(&User{}).AddIndex("idx_user_name_age", "name", "age")

// Add unique index
db.Model(&User{}).AddUniqueIndex("idx_user_name", "name")

// Add unique index for multiple columns
db.Model(&User{}).AddUniqueIndex("idx_user_name_age", "name", "age")

// Remove index
db.Model(&User{}).RemoveIndex("idx_user_name")
```

3. Models

Models

Model Definition

```
type User struct {
    gorm.Model
    Birthday     time.Time
    Age          int
    Name         string  `gorm:"size:255"` // Default size for string is 255, reset it with this tag
    Num          int     `gorm:"AUTO_INCREMENT"`

    CreditCard        CreditCard      // One-To-One relationship (has one - use CreditCard's UserID as foreign key)
    Emails            []Email         // One-To-Many relationship (has many - use Email's UserID as foreign key)

    BillingAddress    Address         // One-To-One relationship (belongs to - use BillingAddressID as foreign key)
    BillingAddressID  sql.NullInt64

    ShippingAddress   Address         // One-To-One relationship (belongs to - use ShippingAddressID as foreign key)
    ShippingAddressID int

    IgnoreMe          int `gorm:"-"`   // Ignore this field
    Languages         []Language `gorm:"many2many:user_languages;"` // Many-To-Many relationship, 'user_languages' is join table
```

```
}

type Email struct {
    ID      int
    UserID  int     `gorm:"index"` // Foreign key (belongs to), tag `index` will create index for this column
    Email   string  `gorm:"type:varchar(100);unique_index"` // `type` set sql type, `unique_index` will create unique index for this column
    Subscribed bool
}

type Address struct {
    ID       int
    Address1 string         `gorm:"not null;unique"` // Set field as not nullable and unique
    Address2 string         `gorm:"type:varchar(100);unique"`
    Post     sql.NullString `gorm:"not null"`
}

type Language struct {
    ID   int
    Name string `gorm:"index:idx_name_code"` // Create index with name, and will create combined index if find other fields defined same name
    Code string `gorm:"index:idx_name_code"` // `unique_index` also works
}

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

Conventions

`gorm.Model` struct

Base model definition `gorm.Model`, including fields `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`, you could embed it in your model, or only write those fields you want

```
// Base Model's definition
type Model struct {
  ID        uint `gorm:"primary_key"`
  CreatedAt time.Time
  UpdatedAt time.Time
```

```
  DeletedAt *time.Time
}

// Add fields `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`
type User struct {
  gorm.Model
  Name string
}

// Only need field `ID`, `CreatedAt`
type User struct {
  ID        uint
  CreatedAt time.Time
  Name      string
}
```

Table name is the pluralized version of struct name

```
type User struct {} // default table name is `users`

// set User's table name to be `profiles`
func (User) TableName() string {
  return "profiles"
}

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

// Disable table name's pluralization globally
db.SingularTable(true) // if set this to true, `User`'s default table name will be `user`, table name setted with `TableName` won't be affected
```

Change default tablenames

You can apply any rules on the default table name by defining the `DefaultTableNameHandler`

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

```
}
```

Column name is the snake case of field's name

```
type User struct {
  ID uint             // column name will be `id`
  Name string         // column name will be `name`
  Birthday time.Time  // column name will be `birthday`
  CreatedAt time.Time // column name will be `created_at`
}

// Overriding Column Name
type Animal struct {
    AnimalId    int64     `gorm:"column:beast_id"`         // set column name to `beast_id`
    Birthday    time.Time `gorm:"column:day_of_the_beast"` // set column name to `day_of_the_beast`
    Age         int64     `gorm:"column:age_of_the_beast"` // set column name to `age_of_the_beast`
}
```

Field `ID` as primary key

```
type User struct {
  ID   uint  // field named `ID` will be the default primary field
  Name string
}

// Set a field to be primary field with tag `primary_key`
type Animal struct {
  AnimalId int64 `gorm:"primary_key"` // set AnimalId to be primary key
  Name     string
  Age      int64
}
```

Field `CreatedAt` used to store record's created time

Create records having `CreatedAt` field will set it to current time.

```
db.Create(&user) // will set `CreatedAt` to current time

// To change its value, you could use `Update`
db.Model(&user).Update("CreatedAt", time.Now())
```

Use `UpdatedAt` used to store record's updated time

Save records having `UpdatedAt` field will set it to current time.

```
// Whenever one or more `user` fields are edited using Save() or Update(), `UpdatedAt` will be set to current time
db.Save(&user) // will set `UpdatedAt` to current time
db.Model(&user).Update("name", "jinzhu") // will set `UpdatedAt` to current time
```

Use `DeletedAt` to store record's deleted time if field exists

Delete records having `DeletedAt` field, it won't be deleted from database, but only set field `DeletedAt` 's value to current time, and the record is not findable when querying, refer Soft Delete

3.1 Associations

Associations

Belongs To

```
// `User` belongs to `Profile`, `ProfileID` is the foreign key
type User struct {
  gorm.Model
  Profile   Profile
  ProfileID int
}

type Profile struct {
  gorm.Model
  Name string
}

db.Model(&user).Related(&profile)
//// SELECT * FROM profiles WHERE id = 111; // 111 is user's foreign key ProfileID
```

Specify Foreign Key

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

type User struct {
    gorm.Model
    Profile      Profile `gorm:"ForeignKey:ProfileRefer"` // use ProfileRefer as foreign key
    ProfileRefer uint
}
```

Specify Foreign Key & Association Key

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

type User struct {
    gorm.Model
    Profile   Profile `gorm:"ForeignKey:ProfileID;AssociationForeignKey:Refer"`
    ProfileID int
}
```

Has One

```
// User has one CreditCard, UserID is the foreign key
type User struct {
    gorm.Model
    CreditCard   CreditCard
}

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

var card CreditCard
db.Model(&user).Related(&card, "CreditCard")
//// SELECT * FROM credit_cards WHERE user_id = 123; // 123 is user's primary key
// CreditCard is user's field name, it means get user's CreditCard relations and fill it into variable card
// If the field name is same as the variable's type name, like above example, it could be omitted, like:
db.Model(&user).Related(&card)
```

Specify Foreign Key

```
type Profile struct {
  gorm.Model
  Name      string
```

```
  UserRefer uint
}

type User struct {
  gorm.Model
  Profile Profile `gorm:"ForeignKey:UserRefer"`
}
```

Specify Foreign Key & Association Key

```
type Profile struct {
  gorm.Model
  Name   string
  UserID uint
}

type User struct {
  gorm.Model
  Refer   uint
  Profile Profile `gorm:"ForeignKey:UserID;AssociationForeignKey:Refer"`
}
```

Has Many

```
// User has many emails, UserID is the foreign key
type User struct {
    gorm.Model
    Emails   []Email
}

type Email struct {
    gorm.Model
    Email   string
    UserID  uint
}

db.Model(&user).Related(&emails)
//// SELECT * FROM emails WHERE user_id = 111; // 111 is user's primary key
```

Specify Foreign Key

```
type Profile struct {
```

```
  gorm.Model
  Name      string
  UserRefer uint
}

type User struct {
  gorm.Model
  Profiles []Profile `gorm:"ForeignKey:UserRefer"`
}
```

Specify Foreign Key & Association Key

```
type Profile struct {
  gorm.Model
  Name   string
  UserID uint
}

type User struct {
  gorm.Model
  Refer   uint
  Profiles []Profile `gorm:"ForeignKey:UserID;AssociationForeignKey:Refer"`
}
```

Many To Many

```
// User has and belongs to many languages, use `user_languages` as join table
type User struct {
    gorm.Model
    Languages         []Language `gorm:"many2many:user_languages;"`
}

type Language struct {
    gorm.Model
    Name string
}

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
```

*With back-reference :

```
// User has and belongs to many languages, use `user_languages` as join table
// Make sure the two models are in different files
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'))
```

Specify Foreign Key & Association Key

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

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

Polymorphism

Supports polymorphic has-many and has-one associations.

```
  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
  }
```

Note: polymorphic belongs-to and many-to-many are explicitly NOT supported, and will throw errors.

Association Mode

Association Mode contains some helper methods to handle relationship things easily.

```
// Start Association Mode
var user User
db.Model(&user).Association("Languages")
// `user` is the source, it need to be a valid record (contains primary key)
// `Languages` is source's field name for a relationship.
// If those conditions not matched, will return an error, check it with:
// db.Model(&user).Association("Languages").Error


// Query - Find out all related associations
db.Model(&user).Association("Languages").Find(&languages)


// Append - Append new associations for many2many, has_many, will replace current association for has_one, belongs_to
db.Model(&user).Association("Languages").Append([]Language{languageZH, languageEN})
db.Model(&user).Association("Languages").Append(Language{Name: "DE"})


// Delete - Remove relationship between source & passed arguments, won't delete those arguments
db.Model(&user).Association("Languages").Delete([]Language{languageZH, languageEN})
db.Model(&user).Association("Languages").Delete(languageZH, languageEN)


// Replace - Replace current associations with new one
db.Model(&user).Association("Languages").Replace([]Language{languageZH, languageEN})
```

```
26. db.Model(&user).Association("Languages").Replace(Language{Name: "DE"}, languageEN)
27.
28.
29. // Count - Return the count of current associations
30. db.Model(&user).Association("Languages").Count()
31.
32.
33. // Clear - Remove relationship between source & current associations, won't delete those
    associations
34. db.Model(&user).Association("Languages").Clear()
```

4. CRUD: Reading and Writing Data

CRUD: Reading and Writing Data

Create

Create Record

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

db.NewRecord(user) // => returns `true` as primary key is blank

db.Create(&user)

db.NewRecord(user) // => return `false` after `user` created
```

Default Values

You could define default value in the `gorm` tag, then the inserting SQL will ignore these fields that has default value and its value is blank, and after insert the record into database, gorm will load those fields's value from database.

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

var animal = Animal{Age: 99, Name: ""}
db.Create(&animal)
// INSERT INTO animals("age") values('99');
// SELECT name from animals WHERE ID=111; // the returning primary key is 111
```

```
// animal.Name => 'galeone'
```

Setting Primary Key In Callbacks

If you want to set primary field's value in `BeforeCreate` callback, you could use `scope.SetColumn`, for example:

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

Extra Creating option

```
// Add extra SQL option for inserting SQL
db.Set("gorm:insert_option", "ON CONFLICT").Create(&product)
// INSERT INTO products (name, code) VALUES ("name", "code") ON CONFLICT;
```

Query

```
// Get first record, order by primary key
db.First(&user)
//// SELECT * FROM users ORDER BY id LIMIT 1;

// Get last record, order by primary key
db.Last(&user)
//// SELECT * FROM users ORDER BY id DESC LIMIT 1;

// Get all records
db.Find(&users)
//// SELECT * FROM users;

// Get record with primary key (only works for integer primary key)
db.First(&user, 10)
//// SELECT * FROM users WHERE id = 10;
```

Query With Where (Plain SQL)

```
// Get first matched record
db.Where("name = ?", "jinzhu").First(&user)
//// SELECT * FROM users WHERE name = 'jinzhu' limit 1;
```

```
4.
5.  // Get all matched records
6.  db.Where("name = ?", "jinzhu").Find(&users)
7.  //// SELECT * FROM users WHERE name = 'jinzhu';
8.
9.  db.Where("name <> ?", "jinzhu").Find(&users)
10.
11. // IN
12. db.Where("name in (?)", []string{"jinzhu", "jinzhu 2"}).Find(&users)
13.
14. // LIKE
15. db.Where("name LIKE ?", "%jin%").Find(&users)
16.
17. // AND
18. db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)
19.
20. // Time
21. db.Where("updated_at > ?", lastWeek).Find(&users)
22.
23. db.Where("created_at BETWEEN ? AND ?", lastWeek, today).Find(&users)
```

Query With Where (Struct & Map)

NOTE When query with struct, GORM will only query with those fields has value

```
1.  // Struct
2.  db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)
3.  //// SELECT * FROM users WHERE name = "jinzhu" AND age = 20 LIMIT 1;
4.
5.  // Map
6.  db.Where(map[string]interface{}{"name": "jinzhu", "age": 20}).Find(&users)
7.  //// SELECT * FROM users WHERE name = "jinzhu" AND age = 20;
8.
9.  // Slice of primary keys
10. db.Where([]int64{20, 21, 22}).Find(&users)
11. //// SELECT * FROM users WHERE id IN (20, 21, 22);
```

Query With Not

```
1.  db.Not("name", "jinzhu").First(&user)
2.  //// SELECT * FROM users WHERE name <> "jinzhu" LIMIT 1;
3.
4.  // Not In
```

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

// Not In slice of primary keys
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;

// Plain 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";
```

Query With Inline Condition

NOTE When query with primary key, you should carefully check the value you passed is a valid primary key, to avoid SQL injection

```
// Get by primary key (only works for integer primary key)
db.First(&user, 23)
//// SELECT * FROM users WHERE id = 23 LIMIT 1;
// Get by primary key if it were a non-integer type
db.First(&user, "id = ?", "string_primary_key")
//// SELECT * FROM users WHERE id = 'string_primary_key' LIMIT 1;

// 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;
```

Query With 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)
```

Query Chains

Gorm has a chainable API, you could use it like this

```
db.Where("name <> ?","jinzhu").Where("age >= ? and role <> ?",20,"admin").Find(&users)
//// SELECT * FROM users WHERE name <> 'jinzhu' AND age >= 20 AND role <> 'admin';

db.Where("role = ?", "admin").Or("role = ?", "super_admin").Not("name = ?",
"jinzhu").Find(&users)
```

SubQuery

```
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')));
```

Extra Querying option

```
// Add extra SQL option for selecting SQL
db.Set("gorm:query_option", "FOR UPDATE").First(&user, 10)
//// SELECT * FROM users WHERE id = 10 FOR UPDATE;
```

FirstOrInit

Get first matched record, or initalize a new one with given conditions (only works with struct, map conditions)

```
// Unfound
```

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

// Found
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

Initalize struct with argument if record haven't been found

```
// Unfound
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}

// Found
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

Assign argument to results regardless it is found or not

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

// Found
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

Get first matched record, or create a new one with given conditions (only works with struct, map conditions)

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

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

Attrs

Assgin struct with argument if record haven't been found

```
// Unfound
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}

// Found
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

Assign it to the record regardless it is found or not, and save back to database.

```
// Unfound
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}

// Found
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}
```

Select

Specify fields that you want to retrieve from database, by default, will select all fields;

```
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

Specify order when retrieve records from database, set reorder to `true` to overwrite defined conditions

```
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;

// ReOrder
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

Specify the number of records to be retrieved

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

// Cancel limit condition with -1
db.Limit(10).Find(&users1).Limit(-1).Find(&users2)
//// SELECT * FROM users LIMIT 10; (users1)
//// SELECT * FROM users; (users2)
```

Offset

Specify the number of records to skip before starting to return the records

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

```
3.
4. // Cancel offset condition with -1
5. db.Offset(10).Find(&users1).Offset(-1).Find(&users2)
6. //// SELECT * FROM users OFFSET 10; (users1)
7. //// SELECT * FROM users; (users2)
```

Count

Get how many records for a model

```
1. db.Where("name = ?", "jinzhu").Or("name = ?", "jinzhu 2").Find(&users).Count(&count)
2. //// SELECT * from USERS WHERE name = 'jinzhu' OR name = 'jinzhu 2'; (users)
3. //// SELECT count(*) FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2'; (count)
4.
5. db.Model(&User{}).Where("name = ?", "jinzhu").Count(&count)
6. //// SELECT count(*) FROM users WHERE name = 'jinzhu'; (count)
7.
8. db.Table("deleted_users").Count(&count)
9. //// SELECT count(*) FROM deleted_users;
```

Group & Having

```
1. rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) as
   total").Group("date(created_at)").Rows()
2. for rows.Next() {
3.     ...
4. }
5.
6. rows, err := db.Table("orders").Select("date(created_at) as date, sum(amount) as
   total").Group("date(created_at)").Having("sum(amount) > ?", 100).Rows()
7. for rows.Next() {
8.     ...
9. }
10.
11. type Result struct {
12.     Date  time.Time
13.     Total int64
14. }
15. db.Table("orders").Select("date(created_at) as date, sum(amount) as
    total").Group("date(created_at)").Having("sum(amount) > ?", 100).Scan(&results)
```

Joins

Specify Joins conditions

```
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)

// multiple joins with parameter
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

Query single column from a model as a map, if you want to query multiple columns, you could use `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)

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

Scan

Scan results into another struct.

```
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)
```

Scopes

Pass current database connection to `func(*DB) *DB`, which could be used to add conditions dynamically

```
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)
// Find all credit card orders and amount greater than 1000

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

db.Scopes(OrderStatus([]string{"paid", "shipped"})).Find(&orders)
// Find all paid, shipped orders
```

Specifying The Table Name

```
// Create `deleted_users` table with struct User's definition
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';
```

Preloading (Eager loading)

```
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 ('cancelled');

db.Where("state = ?", "active").Preload("Orders", "state NOT IN (?)",
"cancelled").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
```

Custom Preloading SQL

You could custom preloading SQL by passing in `func(db *gorm.DB) *gorm.DB` (same type as the one used for Scopes), for example:

```
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;
```

Nested Preloading

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

Update

Update All Fields

`Save` will include all fields when perform the Updating SQL, even it is not changed

```
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 Changed Fields

If you only want to update changed Fields, you could use `Update` , `Updates`

```
// Update single attribute if it is changed
db.Model(&user).Update("name", "hello")
//// UPDATE users SET name='hello', updated_at='2013-11-17 21:34:10' WHERE id=111;

// Update single attribute with combined conditions
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;

// Update multiple attributes with `map`, will only update those changed fields
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;

// Update multiple attributes with `struct`, will only update those changed & non blank fields
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;

// WARNING when update with struct, GORM will only update those fields that with non blank value
// For below Update, nothing will be updated as "", 0, false are blank values of their types
db.Model(&user).Updates(User{Name: "", Age: 0, Actived: false})
```

Update Selected Fields

If you only want to update or ignore some fields when updating, you could use `Select` , `Omit`

```
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;
```

Update Changed Fields Without Callbacks

Above updating operations will perform the model's `BeforeUpdate` , `AfterUpdate` method, update its `UpdatedAt` timestamp, save its `Associations` when updaing, if you don't want to call them, you could use `UpdateColumn` , `UpdateColumns`

```
// 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;
```

Batch Updates

Callbacks won't run when do batch updates

```
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);

// Update with struct only works with none zero values, or use map[string]interface{}
db.Model(User{}).Updates(User{Name: "hello", Age: 18})
//// UPDATE users SET name='hello', age=18;

// Get updated records count with `RowsAffected`
db.Model(User{}).Updates(User{Name: "hello", Age: 18}).RowsAffected
```

Update with SQL Expression

```
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;
```

Change Updating Values In Callbacks

If you want to change updating values in callbacks using `BeforeUpdate`, `BeforeSave`, you could use `scope.SetColumn`, for example:

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

Extra Updating option

```
// Add extra SQL option for updating SQL
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);
```

Delete

WARNING When delete a record, you need to ensure it's primary field has value, and GORM will use the primary key to delete the record, if primary field's blank, GORM will delete all records for the model

```
// Delete an existing record
db.Delete(&email)
//// DELETE from emails where id=10;
```

```
4.
5.  // Add extra SQL option for deleting SQL
6.  db.Set("gorm:delete_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Delete(&email)
7.  //// DELETE from emails where id=10 OPTION (OPTIMIZE FOR UNKNOWN);
```

Batch Delete

Delete all matched records

```
1.  db.Where("email LIKE ?", "%jinzhu%").Delete(Email{})
2.  //// DELETE from emails where email LIKE "%jinzhu%";
3.
4.  db.Delete(Email{}, "email LIKE ?", "%jinzhu%")
5.  //// DELETE from emails where email LIKE "%jinzhu%";
```

Soft Delete

If model has `DeletedAt` field, it will get soft delete ability automatically! then it won't be deleted from database permanently when call `Delete` , but only set field `DeletedAt` 's value to current time

```
1.  db.Delete(&user)
2.  //// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE id = 111;
3.
4.  // Batch Delete
5.  db.Where("age = ?", 20).Delete(&User{})
6.  //// UPDATE users SET deleted_at="2013-10-29 10:23" WHERE age = 20;
7.
8.  // Soft deleted records will be ignored when query them
9.  db.Where("age = 20").Find(&user)
10. //// SELECT * FROM users WHERE age = 20 AND deleted_at IS NULL;
11.
12. // Find soft deleted records with Unscoped
13. db.Unscoped().Where("age = 20").Find(&users)
14. //// SELECT * FROM users WHERE age = 20;
15.
16. // Delete record permanently with Unscoped
17. db.Unscoped().Delete(&order)
18. //// DELETE FROM orders WHERE id=10;
```

Associations

By default when creating/updating a record, GORM will save its associations, if the association has primary

key, GORM will call `Update` to save it, otherwise it will be created.

```
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)
```

Refer Associations for more details

Skip Save Associations when creating/updating

By default when saving an record, GORM will save its associations also, you could skip it by set `gorm:save_associations` to `false`

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

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

Skip Save Associations by Tag

You could use Tag to config your struct to never save an association when creating/updating

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

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

5. Callbacks

Callbacks

You could define callback methods to pointer of model struct, it will be called when creating, updating, querying, deleting, if any callback returns an error, gorm will stop future operations and rollback all changes.

Creating An Object

Available Callbacks for creating

```
// begin transaction
BeforeSave
BeforeCreate
// save before associations
// update timestamp `CreatedAt`, `UpdatedAt`
// save self
// reload fields that have default value and its value is blank
// save after associations
AfterCreate
AfterSave
// commit or rollback transaction
```

Updating An Object

Available Callbacks for updating

```
// begin transaction
BeforeSave
BeforeUpdate
// save before associations
// update timestamp `UpdatedAt`
// save self
// save after associations
AfterUpdate
```

```
9.  AfterSave
10. // commit or rollback transaction
```

Deleting An Object

Available Callbacks for deleting

```
1.  // begin transaction
2.  BeforeDelete
3.  // delete self
4.  AfterDelete
5.  // commit or rollback transaction
```

Querying An Object

Available Callbacks for querying

```
1.  // load data from database
2.  // Preloading (edger loading)
3.  AfterFind
```

Callback Examples

```
1.  func (u *User) BeforeUpdate() (err error) {
2.      if u.readonly() {
3.          err = errors.New("read only user")
4.      }
5.      return
6.  }
7.
8.  // Rollback the insertion if user's id greater than 1000
9.  func (u *User) AfterCreate() (err error) {
10.     if (u.Id > 1000) {
11.         err = errors.New("user id is already greater than 1000")
12.     }
13.     return
14. }
```

Save/Delete operations in gorm are running in transactions, so changes made in that transaction are not visible unless it is commited.
If you want to use those changes in your callbacks, you need to run your SQL in the same transaction. So you need to pass current transaction to callbacks like this:

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

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

6. Advanced Usage

Advanced Usage

Error Handling

After perform any operations, if there are any error happened, GORM will set it to `*DB` 's `Error` field

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

// If there are more than one error happened, get all of them with `GetErrors`, it returns `[]error`
db.First(&user).Limit(10).Find(&users).GetErrors()

// Check if returns RecordNotFound error
db.Where("name = ?", "hello world").First(&user).RecordNotFound()

if db.Model(&user).Related(&credit_card).RecordNotFound() {
    // no credit card found handling
}
```

Transactions

To perform a set of operations within a transaction, the general flow is as below.

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

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

// ...

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

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

A Specific Example

```
func CreateAnimals(db *gorm.DB) err {
  tx := db.Begin()
  // Note the use of tx as the database handle once you are within a transaction

  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
  }

  tx.Commit()
  return nil
}
```

SQL Builder

Run Raw SQL

Run Raw SQL

```
db.Exec("DROP TABLE users;")
db.Exec("UPDATE orders SET shipped_at=? WHERE id IN (?)", time.Now(), []int64{11,22,33})
```

```
3.
4.  // Scan
5.  type Result struct {
6.      Name string
7.      Age  int
8.  }
9.
10. var result Result
11. db.Raw("SELECT name, age FROM users WHERE name = ?", 3).Scan(&result)
```

sql.Row & sql.Rows

Get query result as `*sql.Row` or `*sql.Rows`

```
1.  row := db.Table("users").Where("name = ?", "jinzhu").Select("name, age").Row() // (*sql.Row)
2.  row.Scan(&name, &age)
3.
4.  rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows()
    // (*sql.Rows, error)
5.  defer rows.Close()
6.  for rows.Next() {
7.      ...
8.      rows.Scan(&name, &age, &email)
9.      ...
10. }
11.
12. // Raw SQL
13. rows, err := db.Raw("select name, age, email from users where name = ?", "jinzhu").Rows() //
    (*sql.Rows, error)
14. defer rows.Close()
15. for rows.Next() {
16.     ...
17.     rows.Scan(&name, &age, &email)
18.     ...
19. }
```

Scan sql.Rows In Iteration

```
1.  rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows()
    // (*sql.Rows, error)
2.  defer rows.Close()
3.
4.  for rows.Next() {
5.    var user User
```

```
    db.ScanRows(rows, &user)
    // do something
}
```

Generic database interface sql.DB

Get generic database interface *sql.DB from `*gorm.DB` connection

```
// Get generic database object `*sql.DB` to use its functions
db.DB()

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

Connection Pool

```
db.DB().SetMaxIdleConns(10)
db.DB().SetMaxOpenConns(100)
```

Composite Primary Key

Set multiple fields as primary key to enable composite primary key

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

Logger

Gorm has built-in logger support, by default, it will print happened errors

```
// Enable Logger, show detailed log
db.LogMode(true)

// Disable Logger, don't show any log
db.LogMode(false)

// Debug a single operation, show detailed log for this operation
db.Debug().Where("name = ?", "jinzhu").First(&User{})
```

Customize Logger

Refer GORM's default logger for how to customize it https://github.com/jinzhu/gorm/blob/master/logger.go

```
db.SetLogger(gorm.Logger{revel.TRACE})
db.SetLogger(log.New(os.Stdout, "\r\n", 0))
```

7. Development

Development

Architecture

Gorm use chainable API, `*gorm.DB` is the bridge of chains, for each chain API, it will create a new relation.

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

// create a new relation
db = db.Where("name = ?", "jinzhu")

// filter even more
if SomeCondition {
    db = db.Where("age = ?", 20)
} else {
    db = db.Where("age = ?", 30)
}
if YetAnotherCondition {
    db = db.Where("active = ?", 1)
}
```

When we start to perform any operations, GORM will create a new `*gorm.Scope` instance based on current `*gorm.DB`

```
// perform a querying operation
db.First(&user)
```

And based on current operation's type, it will call registered `creating`, `updating`, `querying`, `deleting` or `row_querying` callbacks to run the operation.

For above example, will call `querying` callbacks, refer Querying Callbacks

Write Plugins

GORM itself is powered by `Callbacks`, so you could fully customize GORM as you want

Register a new callback

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

db.Callback().Create().Register("update_created_at", updateCreated)
// register a callback for Create process
```

Delete an existing callback

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

Replace an existing callback

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

Register callback orders

```
db.Callback().Create().Before("gorm:create").Register("update_created_at", updateCreated)
db.Callback().Create().After("gorm:create").Register("update_created_at", updateCreated)
db.Callback().Query().After("gorm:query").Register("my_plugin:after_query", afterQuery)
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:be
 beforeCreate)
```

Pre-Defined Callbacks

GORM has defiend callbacks to perform its CRUD operations, check them out before start write your plugins

- Create callbacks
- Update callbacks
- Query callbacks
- Delete callbacks
- Row Query callbacks

Row Query callbacks will be called when run `Row` or `Rows`, by default there is no registered callbacks for it, you could register a new one like:

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

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

View https://godoc.org/github.com/jinzhu/gorm to view all available API

8. Change Log

Change Log

v1.0

Breaking Changes

- `gorm.Open` **return type** `*gorm.DB` **instead of** `gorm.DB`
- **Updating will only update changed fields**

  Most applications won't be affected, only when you are changing updating values in callbacks like `BeforeSave`, `BeforeUpdate`, you should use `scope.SetColumn` then, for example:

```
1. func (user *User) BeforeUpdate(scope *gorm.Scope) {
2.   if pw, err := bcrypt.GenerateFromPassword(user.Password, 0); err == nil {
3.     scope.SetColumn("EncryptedPassword", pw)
4.     // user.EncryptedPassword = pw  // doesn't work, won't including EncryptedPassword
   field when updating
5.   }
6. }
```

- **Soft Delete's default querying scope will only check** `deleted_at IS NULL`

  Before it will check `deleted_at` less than `0001-01-02` also to exclude blank time, like:

  `SELECT * FROM users WHERE deleted_at IS NULL OR deleted_at <= '0001-01-02'`

  But it is not necessary if you are using type `*time.Time` for your model's `DeletedAt`, which has been used by `gorm.Model`, so below SQl is enough

  `SELECT * FROM users WHERE deleted_at IS NULL`

  So if you are using `gorm.Model`, then you are good, nothing need to be change, just make sure all records having blank time for `deleted_at` set to `NULL`, sample migrate script:

```
1. import (
2.     "github.com/jinzhu/now"
3. )
```

```
4.
5.  func main() {
6.    var models = []interface{}{&User{}, &Image{}}
7.    for _, model := range models {
8.      db.Unscoped().Model(model).Where("deleted_at < ?", now.MustParse("0001-01-
    02")).Update("deleted_at", gorm.Expr("NULL"))
9.    }
10. }
```

- **New ToDBName logic**

  Before when GORM convert Struct, Field's name to db name, only those common initialisms from golint like `HTTP` , `URI` are special handled.

  So field `HTTP` 's db name will be `http` not `h_t_t_p` , but some other initialisms like `SKU` that not in golint, it's db name will be `s_k_u` , which looks ugly, this release fixed this, any upper case initialisms should be converted correctly.

  If your applications using some upper case initialisms which doesn't exist in golint, you need to overwrite default column name with tag, like `gorm:"column:s_k_u"` , or alert your database's column name according to new logic

- Error `RecordNotFound` has been renamed to `ErrRecordNotFound`
- `mssql` driver has been moved out from default drivers, import it with `import _ "github.com/jinzhu/gorm/dialects/mssql"`
- `Hstore` has been moved to package `github.com/jinzhu/gorm/dialects/postgres`