第十章 Go语言数据库编程

本章重点为大家介绍如下的内容：

* 什么是数据库
* 关系型数据库
* 数据库范式
* MySQL的特点
* Go程序中实现MySQL数据库操作

10.1 数据库介绍

数据库（Database）是按照数据结构来组织、存储和管理数据的仓库，每个数据库都有不同的API用于创建、存储、管理数据。

为了方便数据的存储和管理，将数据按照特定的规律存储在磁盘上。通过数据库管理系统，有效地组织和管理存储在数据库中的数据。

数据库（database）是按照数据结构来组织、存储和管理数据的仓库（通常是一个文件或一组文件）。

理解数据库的一种最简单的办法就是将其想象为一个文件柜，此文件柜是一个存放数据的物理位置。

数据其实可以存储在文本文件中，但是只适用于数据量很少的时候，数据量很少的情况下，在文件中保存数据、读写数据，操作简单便利且速度快；但是当数据量稍微大的情况，普通文本文件保存数据的方式就显得捉襟见肘，甚至束手无策。比如数据结构、数据类型无法设计，数据读写方式复杂且速度慢，对于复杂的数据查询更是无法做到。所以需要专业的数据库软件；

存储数据也可以使用微软的 Excel，但是，随着数据越来越多，在 Excel这样的电子表格里查找数据就变得很慢，而且不可靠。所以需要更高效的数据库管理软件。

而在当今的互联网中，最常见的数据库模型主要是两种，即关系型数据库和非关系型数据库。

关系型数据库包括：Oracle、MySQL、SQLServer、Access、DB2等

非关系型数据库包括：

键值存储型数据库：Redis

文档存储型数据库：MongoDB、CouchDB

列表存储型数据库：HBase

图数据库：FlockDB

10.2 MySQL数据库介绍

MySQL 是一个关系型数据库管理系统，目前属于 Oracle （甲骨文）旗下产品；

MySQL 是最流行的关系型数据库管理系统之一。在 WEB 应用方面，MySQL是最好的 RDBMS (Relational Database Management System，关系型数据库管理系统) 应用软件。关系型数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性；

MySQL所使用的 SQL 语言是用于访问数据库的最标准的结构化查询语言；

MySQL支持大型的数据库。可以处理拥有上千万条记录的大型数据库。支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB；

MySQL支持运行于多个系统上，且支持多种语言。包括C、C++、Python、Java、Perl、PHP、Ruby等；

与其他的大型数据库，例如 Oracle、DB2、SQL Server等相比，MySQL 自有它的不足之处，但是这丝毫也没有减少它受欢迎的程度。对于一般的个人使用者和中小型企业来说，MySQL提供的功能已经绰绰有余。而且MySQL 采用了双授权政策，分为社区版和商业版，由于其体积小、速度快，尤其是开源这一特点，不需要支付额外的费用，总体拥有成本低，一般中小型网站的开发都选择 MySQL 作为网站数据库。

10.3 Go程序操作MySQL数据库

10.3.1 安装mysql模块

go官方提供了database包，database包下有sql/driver。该包用来定义操作数据库的接口，这保证了无论使用哪种数据库，他们的操作方式都是相同的。但go官方并没有提供连接数据库的driver，如果要操作数据库，还需要第三方的driver 包。最常用的有：https://github.com/go-sql-driver/mysql ，支持database/sql，全部采用go编写。获取该第三方库的方法是在代码中直接引用该库的地址：

import github.com/go-sql-driver/mysql

编译通过后，Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着连接MySQL服务器。

例：

import (

"database/sql"

"fmt"

\_ "github.com/go-sql-driver/mysql"

)

匿名导包：只导入包但是不使用包内的类型和数据。使用匿名导包的方式(在包路径前添加下划线 “\_” )导入MySQL驱动。匿名导入的包与其他方式导入包一样，会让导入包编译到可执行文件中。

Go 提供了database/sql包，用于对SQL数据库的访问。它提供了一系列接口方法，用于访问关系数据库。它并不会提供数据库特有的方法，那些特有的方法交给数据库驱动去实现。

通常来说,导入包后就能调用该包中的数据和方法。但是对于数据库操作来说，我们不应该直接使用导入的驱动包所提供的方法，而是应该使用 sql.DB对象所提供的统一的方法。因此在导入 mysql 驱动时，使用了匿名导入包的方式。当导入一个数据库驱动后，该驱动会自行初始化并注册到Golang的database/sql上下文中，这样就可以通过 database/sql 包所提供的方法来访问数据库了。

10.3.2 连接数据库

使用sql包中的Open()函数连接数据库，sql.Open()函数原型如下:

func Open(driverName, dataSourceName string) (\*DB, error)

driverName:使用的驱动名，这个名字其实就是数据库驱动注册到 database/sql 时所使用的名字。

dataSourceName: 数据库连接信息，这个连接包含了数据库的用户名,密码,数据库主机以及需要连接的数据库名等信息。

db, err := sql.Open("mysql", "用户名:密码@tcp(IP:端口)/数据库?charset=utf8")

例：

package main

import (

"database/sql"

"fmt"

\_"github.com/go-sql-driver/mysql"

)

func main() {

db,err := sql.Open("mysql", "root:123456@tcp(127.0.0.1:3306)/testdb?charset=utf8")

if err != nil {

fmt.Println(err)

return

}

db.Close()

}

sql.Open()函数返回sql.DB对象，作为操作数据库的入口对象sql.DB,主要为我们提供了两个重要的功能。

1.sql.DB 通过数据库驱动为我们提供管理底层数据库连接的打开和关闭操作。

2.sql.DB 为我们管理数据库连接池。正在使用的连接被标记为繁忙，用完后回到连接池等待下次使用。所以，如果你没有把连接释放回连接池，会导致过多连接使系统资源耗尽。

sql.Open()返回的sql.DB对象是协程并发安全的。sql.DB的设计就是用来作为长连接使用的。不要频繁Open, Close。比较好的做法是，为每个不同的datastore建一个DB对象，保持这些对象Open。如果需要短连接，那么把DB作为参数传入function，而不要在function中Open, Close。

10.3.3 增删改数据

通过直接调用db对象的Exec()方法执行增删改的SQL语句。

func (db \*DB) Exec(query string, args ...interface{}) (Result, error)

通过db.Exec()插入数据，通过返回的err可知插入失败的原因，通过返回的结果可以进一步查询本次插入数据影响的行数RowsAffected和最后插入的id(如果数据库支持查询最后插入id)。

例：

result, err := db.Exec("INSERT INTO userinfo (username, departname, created) VALUES (?, ?, ?)","Steven","区块链教学部","2017-10-1")

预编译语句(PreparedStatement)提供了诸多好处, 因此我们在开发中尽量使用它。PreparedStatement 可以实现自定义参数的查询，PreparedStatement 通常来说,比手动拼接字符串 SQL 语句高效，PreparedStatement 可以防止SQL注入攻击。一般用Prepared Statements和Exec()完成INSERT, UPDATE, DELETE操作。

使用db对象的Prepare()方法获得预编译对象stmt，然后调用Exec()方法。

func (db \*DB) Prepare(query string) (\*Stmt, error)

例：

stmt, err := db.Prepare("INSERT userinfo SET username=?,departname=?,created=?")

result, err := stmt.Exec("Jackson", "研发部", "2017-10-1")

获取影响数据库的行数，可以根据该数值判断是否操作（插入、删除或修改）成功。

count, err := result.RowsAffected()

获得刚刚添加数据的自增ID

id, err := result.LastInsertId()

10.3.4 查询数据

数据库查询的一般步骤如下：

1.调用 db.Query()方法执行 SQL 语句, 此方法返回一个 Rows 作为查询结果；

func (db \*DB) Query(query string, args ...interface{}) (\*Rows, error)

2.将rows.Next()方法的返回值作为for循环的条件，迭代查询数据；

func (rs \*Rows) Next() bool

3.在循环中，通过 rows.Scan()方法读取每一行数据；

func (rs \*Rows) Scan(dest ...interface{}) error

4.调用db.Close()关闭查询。

也可以通过QueryRow()方法查询单条数据

func (db \*DB) QueryRow(query string, args ...interface{}) \*Row

例：

var username, departname, created string

err := db.QueryRow("SELECT username,departname,created FROM user\_info WHERE uid=?", 3).Scan(&username, &departname, &created)

通过prepare批量数据查询如下：

例：

stmt, err := db.Prepare("SELECT \* FROM user\_info WHERE uid<?")

rows, err := stmt.Query(10)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

fmt.Println(\*user)

}

查询数据时需要注意以下的细节，rows.Scan()方法的参数顺序很重要，必须和查询结果的column相对应（数量和顺序都需要一致），例如 “SELECT \* From user\_info where age >=20 AND age < 30” 查询的column 顺序是 “id, name, age” 和插入操作顺序相同, 因此 rows.Scan() 也需要按照此顺序 rows.Scan(&id, &name, &age), 不然会造成数据读取的错位。

因为Go是强类型语言，所以查询数据时先定义数据类型。查询数据库中的数据存在三种可能：存在值、存在零值、未赋值NULL 三种状态，因此可以将待查询的数据类型定义为sql.NullString 、 sql.NullInt64类型等。可以通过判断Valid值来判断查询到的值是否为赋值状态还是未赋值NULL状态。

每次db.Query操作后, 都建议调用rows.Close()。因为 db.Query() 会从数据库连接池中获取一个连接, 这个底层连接在结果集(rows)未关闭前会被标记为处于繁忙状态。当遍历读到最后一条记录时，会发生一个内部EOF错误，自动调用rows.Close()。但如果出现异常，提前退出循环，rows不会关闭，连接不会回到连接池中，连接也不会关闭, 则此连接会一直被占用。因此通常使用 defer rows.Close() 来确保数据库连接可以正确放回到连接池中。

阅读源码发现rows.Close()操作是幂等操作，而一个幂等操作的特点是：其任意多次执行所产生的影响与一次执行的影响相同。所以即便对已关闭的rows再执行close()也没关系。

10.3.5 示例代码

1.案例中的表结构

CREATE TABLE user\_info (

uid int(10) NOT NULL AUTO\_INCREMENT,

username varchar(64) DEFAULT NULL,

departname varchar(64) DEFAULT NULL,

created date DEFAULT NULL,

PRIMARY KEY (uid)

) ;

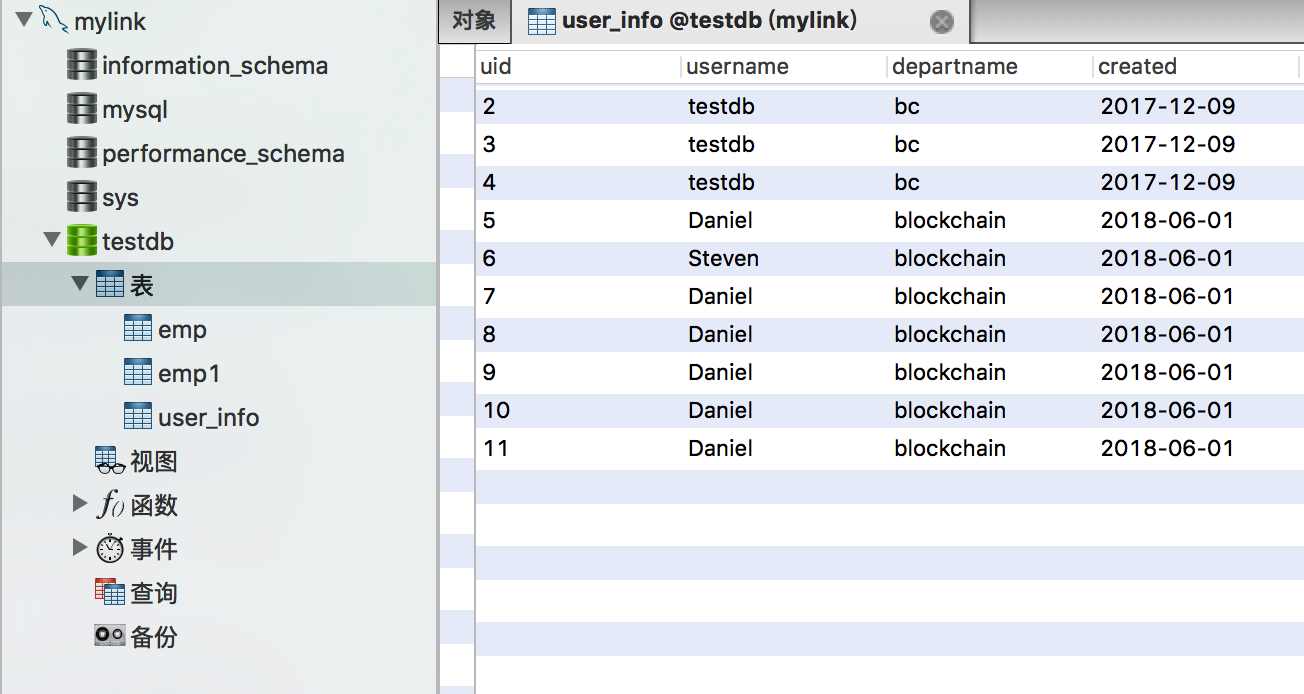


图 10.1

2.完整示例代码

package main

import (

"database/sql"

"fmt"

\_ "github.com/go-sql-driver/mysql"

)

//定义数据库连接信息

type DbConn struct {

Dsn string //数据库驱动连接字符串

Db \*sql.DB

}

//user\_info表的映射对象

type UserTable struct {

Uid int

Username string

Department string

Created string

}

func main() {

var err error

dbConn := DbConn{

Dsn: "root:root@tcp(127.0.0.1:3306)/testdb?charset=utf8",

}

dbConn.Db, err = sql.Open("mysql", dbConn.Dsn)

if err != nil {

panic(err)

return

}

defer dbConn.Db.Close()

//1.测试封装的ExecData()方法

//dbConn.execData(&dbConn)

//2.测试封装的PreExecData()方法

//dbConn.preExecData(&dbConn)

//3.查询单行数据

//查询最后一条数据的信息

//result := dbConn.QueryRowData("select \* from user\_info where uid=(select max(uid) from user\_info)")

//fmt.Println(result)

//4.查询多行数据

//result := dbConn.QueryData("select \* from user\_info where uid<10")

//fmt.Println(len(result))

////遍历查询的结果集

//for k, v := range result {

// fmt.Println("uid：", k, v)

//}

//5.查询多行数据

result := dbConn.PreQueryData("select \* from user\_info where uid<? order by uid desc" , 10)

fmt.Println(len(result))

//遍历查询的结果集

for k, v := range result {

fmt.Println("uid：", k, v)

}

}

//测试封装的ExecData()函数

func execData(dbConn \*DbConn) {

count, id, err := dbConn.ExecData("INSERT user\_info(username , departname , created) VALUES ('Josh','business group','2018-07-3')")

//count , err := execData("UPDATE user\_info SET created='2018-06-30' WHERE uid=14")

//count , err := execData("DELETE FROM user\_info WHERE uid=10")

if err != nil {

fmt.Println(err.Error())

} else {

fmt.Println("受影响行数：", count)

fmt.Println("新添加数据的id：", id)

}

}

//测试封装的PreExecData()函数

func preExecData(dbConn \*DbConn) {

count, id, err := dbConn.PreExecData("INSERT user\_info(username , departname , created) VALUES (?,?,?)", "Jackson", "Education Department", "2017-10-8")

//count, id, err := PreExecData("Delete from user\_info WHERE uid<?", 4)

//count, id, err := PreExecData("UPDATE user\_info set departname=? WHERE departname = ?", "BC Group", "blockchain")

if err != nil {

fmt.Println(err.Error())

} else {

fmt.Println("受影响行数：", count)

fmt.Println("新添加数据的id：", id)

}

}

//1.封装增删改数据的函数，该函数直接使用DB的Exec()方法实现数据操作

func (dbConn \*DbConn) ExecData(sqlString string) (count, id int64, err error) {

result, err := dbConn.Db.Exec(sqlString)

if err != nil {

panic(err)

return

}

if id, err = result.LastInsertId(); err != nil {

panic(err)

return

}

if count, err = result.RowsAffected(); err != nil {

panic(err)

return

}

return count, id, nil

}

//2.封装增删改数据的函数，该函数使用预编译语句加Exec()方法实现增删改数据

func (dbConn \*DbConn) PreExecData(sqlString string, args ...interface{}) (count, id int64, err error) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

result, err := stmt.Exec(args ...)

if err != nil {

panic(err)

return

}

if id, err = result.LastInsertId(); err != nil {

panic(err)

return

}

if count, err = result.RowsAffected(); err != nil {

panic(err)

return

}

return count, id, nil

}

//3.查询当行数据

func (dbConn \*DbConn) QueryRowData(sqlString string) (data UserTable) {

user := new(UserTable)

err := dbConn.Db.QueryRow(sqlString).Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

return

}

return \*user

}

//4.未使用预编译，直接查询多行数据

func (dbConn \*DbConn) QueryData(sqlString string) (resultSet map[int]UserTable) {

rows, err := dbConn.Db.Query(sqlString)

defer rows.Close()

if err != nil {

panic(err)

return

}

resultSet = make(map[int]UserTable)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

resultSet[user.Uid] = \*user

}

return resultSet

}

//5.使用预编译语句进行查询多行数据

func (dbConn \*DbConn) PreQueryData(sqlString string , args ...interface{}) (resultSet map[int]UserTable) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

rows, err := stmt.Query(args ...)

defer rows.Close()

if err != nil {

panic(err)

return

}

resultSet = make(map[int]UserTable)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

resultSet[user.Uid] = \*user

}

return resultSet

}

//无返回值，只打印输出，用于测试

func (dbConn \*DbConn) PreQueryData2(sqlString string , args ...interface{}) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

rows, err := stmt.Query(args ...)

defer rows.Close()

if err != nil {

panic(err)

return

}

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

fmt.Println(\*user)

}

}

10.4 xorm库

xorm 是一个简单而强大的Go语言ORM库. 通过它可以使数据库操作非常简便。

xorm特性如下：

* 支持 Struct 和数据库表之间的灵活映射，并支持自动同步
* 事务支持
* 同时支持原始SQL语句和ORM操作的混合执行
* 使用连写来简化调用
* 支持使用ID, In, Where, Limit, Join, Having, Table, SQL, Cols等函数和结构体等方式作为条件
* 支持级联加载Struct
* Schema支持（仅Postgres）
* 支持缓存
* 通过 xorm.io/reverse 支持根据数据库自动生成 xorm 结构体
* 支持记录版本（即乐观锁）
* 通过 xorm.io/builder 内置 SQL Builder 支持
* 上下文缓存支持
* 支持日志上下文

目前支持的Go数据库驱动和对应的数据库如下：

* Mysql5.\* / Mysql8.\* / Mariadb / Tidb

github.com/go-sql-driver/mysql

github.com/ziutek/mymysql/godrv

* Postgres / Cockroach

github.com/lib/pq

* SQLite

github.com/mattn/go-sqlite3

* MsSql

github.com/denisenkom/go-mssqldb

* Oracle

github.com/mattn/go-oci8 (试验性支持)

10.4.1 获取xorm库

获取该xorm库的方法是在代码中直接引用该库的地址：

import github.com/xorm.io/xorm

编译通过后，Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着使用xorm操作MySQL服务。

xorm库手册地址：http://gobook.io/read/gitea.com/xorm/manual-en-US/

xorm库文档地址：https://pkg.go.dev/xorm.io/xorm?tab=doc

10.4.2 xorm连接数据库

创建引擎，driverName, dataSourceName和database/sql的写法一致。

例：

package main

import (

"fmt"

\_ "github.com/go-sql-driver/mysql"

"github.com/go-xorm/xorm"

)

func main() {

engine, err := xorm.NewEngine("mysql", "root:@tcp(127.0.0.1:3306)/testdb?charset=utf8")

if err != nil{

fmt.Println(err)

return

}

engine.ShowSQL(true)//设置在控制台输出SQL语句，默认为false

}

xorm允许创建Engine组。

dataSourceNameSlice := []string{masterDataSourceName, slave1DataSourceName, slave2DataSourceName}

engineGroup, err := xorm.NewEngineGroup(driverName, dataSourceNameSlice)

所有使用 engine 都可以简单的用 engineGroup 来替换。

10.4.3 xorm查询数据

xorm同时支持原始SQL语句和ORM操作的混合执行，原始SQL执行方式与之前讲过的方法一致。本节将对xorm特有的方式进行讲解。

xorm的查询分为查询一条数据与查询多条数据两种方式，查询一条数据使用Get方法，查询多条数据使用Find方法。

1.查询一条数据

例：

package main

import (

"fmt"

\_ "github.com/go-sql-driver/mysql"

"github.com/go-xorm/xorm"

"time"

)

//定义与表结构一致的结构体

type User struct {

Id int64

Name string

Salt string

Age int

Passwd string `xorm:"varchar(32)"`

Created time.Time `xorm:"created"`

Updated time.Time `xorm:"updated"`

}

func main() {

//连接MySQL数据库

engine, err := xorm.NewEngine("mysql", "root:@tcp(127.0.0.1:3306)/testdb?charset=utf8")

if err != nil{

fmt.Println(err)

return

}

engine.ShowSQL(true)//设置在控制台输出SQL语句，默认为false

var user User//声明结构体

has , err := engine.Get(&user)//查询一条数据

if has {

fmt.Println(user)

}

}

因为我们设置显示SQL语句，所以在控制输出的内容为：

[xorm] [info] 2020/04/12 22:16:40.344033 [SQL] SELECT `id`, `name`, `salt`, `age`, `passwd`, `created`, `updated` FROM `user` LIMIT 1

{1 admin 0 21232f297a57a5a743894a0e4a801fc3 2020-04-01 00:00:00 +0800 CST 2020-04-01 00:00:00 +0800 CST}

控制台将Get使用的SQL语句显示在控制台上并将查询到的结果也输出。Get方法返回的第一个参数has是布尔值，如果查询到结果返回true，否则返回false。

2.查询多条数据

例：

package main

import (

"fmt"

\_ "github.com/go-sql-driver/mysql"

"github.com/go-xorm/xorm"

"time"

)

//定义与表结构一致的结构体

type User struct {

Id int64

Name string

Salt string

Age int

Passwd string `xorm:"varchar(32)"`

Created time.Time `xorm:"created"`

Updated time.Time `xorm:"updated"`

}

func main() {

//连接MySQL数据库

engine, err := xorm.NewEngine("mysql", "root:@tcp(127.0.0.1:3306)/testdb?charset=utf8")

if err != nil{

fmt.Println(err)

return

}

engine.ShowSQL(true)//设置在控制台输出SQL语句，默认为false

var users []User//声明结构体切片

err = engine.Find(&users)//查询一条数据

fmt.Println(users)

}

因为我们设置显示SQL语句，所以在控制输出的内容为：

[xorm] [info] 2020/04/12 22:19:54.605289 [SQL] SELECT `id`, `name`, `salt`, `age`, `passwd`, `created`, `updated` FROM `user`

[{1 admin 0 21232f297a57a5a743894a0e4a801fc3 2020-04-01 00:00:00 +0800 CST 2020-04-01 00:00:00 +0800 CST} {2 user 0 ee11cbb19052e40b

07aac0ca060c23ee 2020-04-01 00:00:00 +0800 CST 2020-04-01 00:00:00 +0800 CST} {3 tt 0 accc9105df5383111407fd5b41255e23 2020-04-01 00:

00:00 +0800 CST 2020-04-01 00:00:00 +0800 CST}]

控制台将Find使用的SQL语句显示在控制台上并将查询到的结果也输出。Find方法只有一个返回值是error类型，如果查询到结果返回true，否则返回false。如果没有查询到数据返回的结果一个空的切片。

以下为表数据为空的情况控制控制台的输出内容：

[xorm] [info] 2020/04/12 22:21:58.525385 [SQL] SELECT `id`, `name`, `salt`, `age`, `passwd`, `created`, `updated` FROM `user`

[]

3.扩展条件查询

10.4.4

10.4.5

[第十章 Go语言数据库编程 1](#_Toc27583)

[10.1 数据库介绍 1](#_Toc22832)

[10.2 MySQL数据库介绍 1](#_Toc13437)

[10.3 Go程序操作MySQL数据库 1](#_Toc2217)

[10.3.1 安装mysql模块 2](#_Toc10015)

[10.3.2 连接数据库 2](#_Toc21307)

[10.3.3 增删改数据 3](#_Toc28446)

[10.3.4 查询数据 3](#_Toc22210)

[10.3.5 示例代码 4](#_Toc23069)

[10.4 xorm库 10](#_Toc7332)