Go语言 数据操作 MySQL

MySQL是一个基于结构化查询语言（SQL）的开源关系数据库管理系统。它是一种关系数据库，可将数据组织到一个或多个表中，其中数据相互关联。MySQL 是行业领先的开源数据库管理系统。它是一个多用户、多线程的数据库管理系统。

数据库驱动程序：数据库驱动程序实现了用于数据库连接的协议。驱动程序就像一个适配器，连接到特定数据库的通用接口。

Go 有 sql 包，它提供了一个围绕 SQL（或类似 SQL）数据库的通用接口。 sql 包必须与数据库驱动程序一起使用。该软件包提供自动连接池。每次查询数据库时，我们都在使用应用程序启动时设置的连接池中的连接。连接被重用。

准备工作

下载安装MySQL：

https://dev.mysql.com/downloads/mysql/

Go安装MySQL驱动：

```
go get github.com/go-sql-driver/mysql
```

创建一个go_db数据库

```
create database go_db
```

创建表：

```
CREATE TABLE `user`  (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
DEFAULT NULL,
  `age` int(3) NULL DEFAULT NULL,
  `email` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
DEFAULT NULL,
  `sex` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT
NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARACTER SET = utf8mb4 COLLATE =
utf8mb4_general_ci ROW_FORMAT = Dynamic;

SET FOREIGN_KEY_CHECKS = 1;
```

```
CREATE TABLE `address`  (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `province` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
DEFAULT NULL,
  `city` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
DEFAULT NULL,
  `uid` int(11) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARACTER SET = utf8mb4 COLLATE =
utf8mb4_general_ci ROW_FORMAT = Dynamic;

SET FOREIGN_KEY_CHECKS = 1;
```

导入包:

```
package main

import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

当导入带有空白标识符前缀 `_` 的包时，将调用包的 init 函数。该函数注册驱动程序。

MySQL使用

链接mysql

使用 `sql.Open`，我们打开一个由其数据库驱动程序名称和驱动程序特定数据源名称指定的数据库，通常至少由数据库名称和连接信息组成。它不与数据库建立任何连接，也不验证驱动程序连接参数。相反，它只是为以后使用准备数据库抽象。当第一次需要时，将延迟建立与底层数据存储的第一个实际连接。

```
db, err := sql.Open("mysql", "<user>:<password>@tcp(127.0.0.1:3306)/<database-
name>")
db, err := sql.Open("数据库类型", "用户名:密码@tcp(地址:端口)/数据库名")
```

```
db, err := sql.Open("mysql", "root:123456@tcp(127.0.0.1:3306)/go_db")
defer.Close()
```

Close 将连接返回到连接池。如果 sql.DB 的生命周期不应超出函数范围，则 defer db.Close 是常用的。

创建表结构体

```
type User struct {
    Id    int    `db:"id"`
    Name  string `db:"name"`
    Age   int    `db:"age"`
    Sex   string `db:"sex"`
    Email string `db:"email"`
}

type Address struct {
    Province string `db:"province"`
    City     string `db:"city"`
    Uid      int    `db:"uid"`
}
```

Ping测试MySQL 及 查询 MySQL 版本号

```
package main

import (
    "database/sql"
    "fmt"
    "log"

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

func main() {
    db, err := sql.Open("mysql", "root:123456@tcp(127.0.0.1:3306)/go_db") // 链接
Mysql
    if err != nil {
        fmt.Println("数据库连接失败！")
        log.Fatalln(err)
    }

    defer db.Close() // 关闭mysql连接

    err2 := db.Ping() // Ping测试Mysql
    if err2 != nil {
        fmt.Println("无法访问数据库！")
        log.Fatalln(err)
    }

    var version string
    err3 := db.QueryRow("SELECT VERSION()").Scan(&version)
    if err3 != nil {
        fmt.Println("无法查询版本号！")
        log.Fatalln(err3)
    }
    fmt.Println(version)
}
#
5.7.30
```

DB（数据库对象）

sql.DB 类型代表了数据库，其它语言操作数据库的时候，需要创建一个连接，对于 Go 而言则是需要创建一个数据库类型，它不是数据库连接，Go 中的连接来自内部实现的连接池，连接的建立是惰性的，连接将会在操作的时候，由连接池创建并维护

> 使用 `sql.Open` 函数创建数据库类型，第一个是数据库驱动名，第二个是连接信息的字符串

```
var Db *sqlx.DB
db, err := sqlx.Open("mysql","username:password@tcp(ip:port)/database?
charset=utf8")
Db = db
```

Results 和 Result（结果集）

新增、更新、删除；和查询所用的方法不一样，所有返回的类型也不同

- Result 是新增、更新、删除时返回的结果集
- Results 是查询数据库时的结果集，sql.Rows 类型表示查询返回多行数据的结果集，sql.Row 则表示单行查询的结果集

Statements（语句）

sql.Stmt 类型表示 sql 语句，例如 DDL,DML 等类似的 sql 语句，可以当成 prepare 语句构造查询，也可以直接使用 sql.DB 的函数对其操作

- DML（data manipulation language）是数据操纵语言：它们是SELECT、UPDATE、INSERT、DELETE，就象它的名字一样，这4条命令是用来对数据库里的数据进行操作的语言。
- DDL（data definition language）是数据定义语言：DDL比DML要多，主要的命令有CREATE、ALTER、DROP等，DDL主要是用在定义或改变表（TABLE）的结构，数据类型，表之间的链接和约束等初始化工作上，他们大多在建立表时使用。
- DCL（DataControlLanguage）是数据库控制语言：是用来设置或更改数据库用户或角色权限的语句，包括（grant,deny,revoke等）语句。

MySQL 连接池相关参数配置

为了方便测试后续CURD各功能函数，本着`高内聚、低耦合`的原则，我们定义一个全局变量`db`，用来保存连接数据库的句柄。将上面的示例代码拆分为独立的函数`initDB`，在程序启动时完成初始化，其他功能函数就可以直接使用全局变量了。
初始化连接函数

```
func init() {
    dsn := "root:123456@tcp(127.0.0.1:3306)/go_db?
charset=utf8mb4&parseTime=True"
    db, err := sql.Open("mysql", dsn) //注意使用全局对象进行赋值
    if err != nil {
        log.Fatalf("数据库连接失败: %v", err)
    }
    if err := db.Ping(); err != nil {
        log.Fatalf("无法访问数据库: %v", err)
    }
    // Maximum Idle Connections 最大空闲连接数
```

```
    db.SetMaxIdleConns(5)
    // Maximum Open Connections 最大打开的连接数，默认值为0表示不限制
    db.SetMaxOpenConns(10)
    // Idle Connection Timeout 连接最大空闲时长
    db.SetConnMaxIdleTime(1 * time.Second)
    // Connection Lifetime 连接最大存活时长
    db.SetConnMaxLifetime(30 * time.Second)
}
```

- 最好让一小部分连接处于空闲状态，以应对查询吞吐量的突然峰值
- 应该根据我们的数据库服务器和应用程序服务器的网络容量来设置打开连接的最大数量，其中较小的将是限制因素。

增删改 操作

插入、更新和删除操作都使用`Exec`方法。

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

新增数据

```
// 插入数据
func insertDemo() {
    newUser := User{
        Name:  "王五",
        Age:   25,
        Sex:   "女",
        Email: "wangwu@163.com",
    }
    sqlStr := "insert into user(name,age,sex,email) values (?,?,?,?)"

    // “Exec”方法返回“Result”类型，而不是“Row”类型`
    // 遵循相同的参数模式来添加参数 上面问号对应下边参数
    result, err := Db.Exec(sqlStr, newUser.Name, newUser.Age, newUser.Sex,
newUser.Email)
    if err != nil {
        log.Fatalf("无法插入数据: %v", err)
    }

    theID, err := result.LastInsertId() //新插入的id
    if err != nil {
        log.Fatalf("获取插入的ID失败:%v", err)
    }
    fmt.Printf("插入成功。插入ID是： %d\n", theID)
}
```

修改数据

```
// 更新数据
func updateDemo() {
    newUser := User{
        Name: "王五",
        Age:  28,
```

```
    }
    sqlStr := "update user set age=? where name = ?" // 修改语句
    result, err := Db.Exec(sqlStr, newUser.Age, newUser.Name)
    if err != nil {
        log.Fatalf("修改失败:%v", err)
    }

    // “Result”类型具有特殊方法，如“RowsAffected”，RowsAffected()的方法获取更新的条数 ，数据库报告的受影响行总数
    rows, err := result.RowsAffected()
    if err != nil {
        log.Fatalf("获取影响条数失败: %v", err)
    }
    // 更新了多少行
    fmt.Printf("修改成功, 修改条数:%v", rows)

}
```

删除数据

```
// 删除数据
func deleteDemo() {
    newUser := User{
        Name: "张三",
    }
    sqlStr := "delete from user where name = ?"
    result, err := Db.Exec(sqlStr, newUser.Name)
    if err != nil {
        log.Fatalf("删除失败:%v", err)
    }

    // RowsAffected()的方法获取更新的条数 ，数据库报告的受影响行总数
    rows, err := result.RowsAffected()
    if err != nil {
        log.Fatalf("获取受影响条数: %v", err)
    }
    // 删除了多少行
    fmt.Printf("删除成功，删除条数:%v", rows)
}
```

查询数据

单条查询使用QueryRow方法。

```
// 查询单条数据
func queryDemo(id int) {
    sqlStr := "select id,name,age,email,sex from user where id = ? limit 1"
    // `QueryRow`始终从数据库中返回一行数据
    // `QueryRow`第二个参数可以接收多个参数，同理，sqlStr可以有多个 ?占位符 进行匹配
    // `QueryRow`之后必须调用Scan方法进行数据绑定，进行数据库链接释放
    row := Db.QueryRow(sqlStr, id)

    var u User
    if err := row.Scan(&u.Id, &u.Name, &u.Age, &u.Email, &u.Sex); err != nil {
```

```
        log.Fatalf("无法扫描数据: %v", err)
    }
    fmt.Printf("User:%+v\n", u)
}
```

多行查询使用Query。

```
// 查询多条数据
func queryMultiDemo() {
    sqlStr := "select id,name,age,email,sex from user where id > ?"
    // 多行查询使用Query
    rows, err := Db.Query(sqlStr, 1)
    if err != nil {
        log.Fatalf("无法执行查询: %v", err)
    }
    //循环读取结果集数据
    users := []User{}
    for rows.Next() {
        var u User
        // 创建“user”实例并将当前行的结果写入其中
        err := rows.Scan(&u.Id, &u.Name, &u.Age, &u.Email, &u.Sex)
        if err != nil {
            log.Fatalf("无法扫描数据: %v", err)
        }
        users = append(users, u)
    }
    // 打印长度和所有元素
    fmt.Printf("len:%v,users:%+v\n", len(users), users)
}
```

超时相关功能

sql的慢查询也是一个需要考虑的问题，如果某个请求的查询比较耗时，就会阻塞其他请求拿到连接，因此我们需要在连接sql时考虑到如果超时则取消功能。这里我们使用context进行测试：

```
// 测试超时
func timeOutDemo() {
    // 创建上下文
    ctx := context.Background()
    // 超时为300ms
    ctx, _ = context.WithTimeout(ctx, 300*time.Millisecond)
    // 慢查询
    _, err := Db.QueryContext(ctx, "select sleep(1)")
    if err != nil {
        log.Fatalf("无法执行查询: %v", err)
    }
}
```

我们设置超时时间为300ms，但是这条sql语句需要1s才能执行完成。所以报错。

MySQL事务操作

MySQL事务特性：

- 原子性
- 一致性
- 隔离性
- 持久性

Go 关于事物有三个方法

- Begin()开始事物。
- Commit()提交事物。
- Rollback()失败回滚。

事务的操作要么一起成功、要么一起失败。

```
// 事务操作示例
func transactionDemo() {
    tx, err := Db.Begin() //开启事务
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        fmt.Printf("事务开启失败:%v\n", err)
        return
    }

    sqlStr := "Update user set age = 34 where id = ?"
    result, err := tx.Exec(sqlStr, 2)
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        fmt.Printf("修改失败:%v\n", err)
        return
    }
    rows, err := result.RowsAffected()
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        log.Fatalf("获取不到受影响行数: %v", err)
        return
    }
    fmt.Println(rows)

    if rows == 1 {
        fmt.Println("只有一个用户影响，提交事务")
        tx.Commit() //提交事务
    } else {
        fmt.Println("影响条数错误，回滚")
        tx.Rollback()
    }

    fmt.Println("事务结束")
```

```
}
```

完整示例：

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"
    "time"

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

var Db *sql.DB

// 定义初始化数据库函数
func init() {
    dsn := "root:123456@tcp(127.0.0.1:3306)/go_db?
charset=utf8mb4&parseTime=True"
    db, err := sql.Open("mysql", dsn) //注意使用全局对象进行赋值
    if err != nil {
        log.Fatalf("数据库连接失败: %v", err)
    }

    if err := db.Ping(); err != nil {
        log.Fatalf("无法访问数据库: %v", err)
    }
    // Maximum Idle Connections 最大空闲连接数
    db.SetMaxIdleConns(5)
    // Maximum Open Connections 最大打开的连接数，默认值为0表示不限制
    db.SetMaxOpenConns(10)
    // Idle Connection Timeout 连接最大空闲时长
    db.SetConnMaxIdleTime(1 * time.Second)
    // Connection Lifetime 连接最大存活时长
    db.SetConnMaxLifetime(30 * time.Second)

    Db = db
}

type User struct {
    Id    int    `db:"id"`
    Name  string `db:"name"`
    Age   int    `db:"age"`
    Sex   string `db:"sex"`
    Email string `db:"email"`
}

type Address struct {
    Province string `db:"province"`
    City     string `db:"city"`
    Uid      int    `db:"uid"`
}
```

```
// 插入数据
func insertDemo() {
    newUser := User{
        Name:  "王五",
        Age:   25,
        Sex:   "女",
        Email: "wangwu@163.com",
    }
    sqlStr := "insert into user(name,age,sex,email) values (?,?,?,?)"

    // “Exec”方法返回“Result”类型，而不是“Row”类型`
    // 遵循相同的参数模式来添加参数 上面问号对应下边参数
    result, err := Db.Exec(sqlStr, newUser.Name, newUser.Age, newUser.Sex,
newUser.Email)
    if err != nil {
        log.Fatalf("无法插入数据: %v", err)
    }

    theID, err := result.LastInsertId() //新插入的id
    if err != nil {
        log.Fatalf("获取插入的ID失败:%v", err)
    }
    fmt.Printf("插入成功。插入ID是： %d\n", theID)
}

// 更新数据
func updateDemo() {
    newUser := User{
        Name: "王五",
        Age:  28,
    }
    sqlStr := "update user set age=? where name = ?" // 修改语句
    result, err := Db.Exec(sqlStr, newUser.Age, newUser.Name)
    if err != nil {
        log.Fatalf("修改失败:%v", err)
    }

    // “Result”类型具有特殊方法，如“RowsAffected”，RowsAffected()的方法获取更新的条数 ，
数据库报告的受影响行总数
    rows, err := result.RowsAffected()
    if err != nil {
        log.Fatalf("获取影响条数失败: %v", err)
    }
    // 更新了多少行
    fmt.Printf("修改成功, 修改条数:%v", rows)

}

// 删除数据
func deleteDemo() {
    newUser := User{
        Name: "张三",
    }
    sqlStr := "delete from user where name = ?"
```

```
    result, err := Db.Exec(sqlStr, newUser.Name)
    if err != nil {
        log.Fatalf("删除失败:%v", err)
    }

    // RowsAffected()的方法获取更新的条数 ， 数据库报告的受影响行总数
    rows, err := result.RowsAffected()
    if err != nil {
        log.Fatalf("获取受影响条数: %v", err)
    }
    // 删除了多少行
    fmt.Printf("删除成功，删除条数:%v", rows)
}

// 查询单条数据
func queryDemo(id int) {
    sqlStr := "select id,name,age,email,sex from user where id = ? limit 1"
    // `QueryRow`始终从数据库中返回一行数据
    // `QueryRow`第二个参数可以接收多个参数，同理，sqlStr可以有多个 ?占位符 进行匹配
    // `QueryRow`之后必须调用Scan方法进行数据绑定，进行数据库链接释放
    row := Db.QueryRow(sqlStr, id)

    var u User
    if err := row.Scan(&u.Id, &u.Name, &u.Age, &u.Email, &u.Sex); err != nil {
        log.Fatalf("无法扫描数据: %v", err)
    }
    fmt.Printf("User:%+v\n", u)
}

// 查询多条数据
func queryMultiDemo() {
    sqlStr := "select id,name,age,email,sex from user where id > ?"
    // 多行查询使用Query
    rows, err := Db.Query(sqlStr, 1)
    if err != nil {
        log.Fatalf("无法执行查询: %v", err)
    }
    //循环读取结果集数据
    users := []User{}
    for rows.Next() {
        var u User
        // 创建“user”实例并将当前行的结果写入其中
        err := rows.Scan(&u.Id, &u.Name, &u.Age, &u.Email, &u.Sex)
        if err != nil {
            log.Fatalf("无法扫描数据: %v", err)
        }
        users = append(users, u)
    }
    // 打印长度和所有元素
    fmt.Printf("len:%v,users:%+v\n", len(users), users)
}

// 测试超时
func timeOutDemo() {
    // 创建上下文
```

```
    ctx := context.Background()
    // 超时为300ms
    ctx, _ = context.WithTimeout(ctx, 300*time.Millisecond)
    // 慢查询
    _, err := Db.QueryContext(ctx, "select sleep(1)")
    if err != nil {
        log.Fatalf("无法执行查询: %v", err)
    }
}

// 事务操作示例
func transactionDemo() {
    tx, err := Db.Begin() //开启事务
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        fmt.Printf("事务开启失败:%v\n", err)
        return
    }

    sqlStr := "Update user set age = 34 where id = ?"
    result, err := tx.Exec(sqlStr, 2)
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        fmt.Printf("修改失败:%v\n", err)
        return
    }
    rows, err := result.RowsAffected()
    if err != nil {
        if tx != nil {
            tx.Rollback() //回滚
        }
        log.Fatalf("获取不到受影响行数: %v", err)
        return
    }
    fmt.Println(rows)

    if rows == 1 {
        fmt.Println("只有一个用户影响，提交事务")
        tx.Commit() //提交事务
    } else {
        fmt.Println("影响条数错误，回滚")
        tx.Rollback()
    }

    fmt.Println("事务结束")
}

func main() {
    // insertDemo()
    // updateDemo()
    // deleteDemo()
```

```
    // queryDemo(1)
    // queryMultiDemo()
    // timeOutDemo()
    transactionDemo()
    defer Db.Close() // 注意这行代码要写在下面
}
```