Oracle 数据库 GO 编程

http://bbs.gocn.im/forum.php 内分享

晨笛　　整理

2013/3/27

目　　录

1. Oracle 的 OCI 接口编程介绍

3.1 OCI 环境初始化

这部分描述如何初始化 OCI 环境，设立一个服务器的连接，并且授权一个用户对数据库执行操作。

首先，初始化 OCI 环境时有三个主要步骤：

- **创建 OCI 环境**
- **分配句柄和描述符**
- **应用程序初始化、连接和创建会话**

3.1.1 创建 OCI 环境

每一个 OCI 函数调用都是在 OCIEnvCreate()函数所创建的环境中执行的。这个函数必须在其他 OCI 函数执行前被调用。唯一的例外是设置 OCI 共享模式的进程级属性。

OCIEnvCreate()函数的模式(mode)参数指定应用程序是否能够：

- 运行在多线程环境中(mode=OCI_THREADED).
- 使用对象(mode=OCI_OBJECT).Use the AQ subscription registration

程序也可以选择不使用这些模式而使用默认模式(mode=OCI_DEFAULT)或者它们的联合，使用垂直条(|)将它们分开。例如，如果 mode=(OCI_THREADED|OCI_OBJECT)，然后应用程序运行在多线程环境中并且使用对象。

我们可以为每个 OCI 环境指明用户定义的内存管理函数。

3.1.2 分配句柄和描述符

Oracle 数据库提供了用于分配和释放句柄及描述符的 OCI 函数。在向 OCI 函数传递句柄之前我们必须使用 OCIHandleAlloc()函数分配句柄,除非 OCI 函数,比如 OCIBindByPos(),自动为我们分配句柄。

我们使用 OCIHandleAlloc()函数来分配表 2-1 列出的句柄。

3.1.3 应用程序初始化、连接和创建会话

一个应用程序必须调用 OCIEnvCreate()函数来初始化 OCI 环境句柄。

沿着这一步,应用程序有两个选项来设置一个服务器连接和开始一个用户会话:单用户、单连接或者多会话多连接。

注意：应该使用 OCIEnvCreate()函数而不是 OCIInitialize()函数和 OCIEnvInit()函数。OCIInitialize()和 OCIEnvInit()函数是向后兼容的函数。

3.1.3.1 单用户、单连接

如果应用程序在任何时刻仅为各个数据库连接维持一个单用户会话,这个选项就是简化的登录函数。

当一个应用程序调用 OCILogon()函数时，OCI 库初始化传递给它的服务上下午句柄，并且创建一个到用户指定的数据库的连接。

下面的例子显示了 OCILogon()函数的用法：

```
OCILogon(envhp,  errhp,  &svchp,  (text*) “hr”,  nameLen, (text*) “hr”,
            passwdLen,  (text*) “oracledb”,  dbnameLen);
```

这个函数的参数包括服务上下午句柄(它将被初始化)，用户名，用户的密码，还有用来设置连接的数据库名。这个函数会隐式地分配服务器句柄和用户会话句柄。

如果应用程序使用这个登录函数，则服务上下文句柄、服务器句柄和用户会话句柄将成为只读的。应用程序不能通过改变服务上下午句柄的相关属性来转换会话或事务。

使用 OCILogon()函数初始化它的会话和认证的应用程序必须调用 OCILogoff()函数来终止它们。

3.1.3.2 多会话或者多连接

这个选项使用显式的服务器连接和开始会话函数来维持一个数据库连接之上的多用户会话和连接。服务器连接和开始会话的函数分别是：

- OCIServerAttach()—创建一个到数据源执行 OCI 操作的访问路径。
- OCISessionBegin()—为一个用户设置一个针对某一特定服务器的会话。为了在数据库服务器上执行数据库操作，必须执行这个函数。

这两个函数设置一个能够使我们执行 SQL 或者 PL/SQL 语句的执行环境。

3.1.4 在 OCI 中处理 SQL 语句

在 OCI 中处理 SQL 语句的细节在后面描述。

3.2 提交或者回滚事务

应用程序通过调用 OCITransCommit()函数来提交对数据库的修改。这个函数使用一个服务上下文句柄作为它的一个参数。与此服务上下文句柄相连的事务将被提交。应用程序可以显式地创建事务或者当应用程序修改数据库时隐式地创建事务。

注意：如果使用 OCIExecute()函数的 OCI_COMMIT_ON_SUCCESS 模式，应用程序可以在每个语句执行后选择性地提交事务，这样可以节省一个额外的网络开销。

使用 OCITransRollback()函数来回滚事务。

如果一个应用程序以非正常方式同 Oracle 断开连接，比如网络连接断开，并且 OCITransCommit()函数没有被调用，则所有活动的事务都会被自动回滚。

3.3 终止应用程序

一个 OCI 程序需要在它终止前执行如下步骤：

1. 调用 OCISessionEnd()函数删除每一个会话的用户会话。
2. 调用 OCIServerDetach()函数删除每一个数据源的访问。
3. 调用 OCIHandleFree()函数来释放每一个句柄。
4. 删除环境句柄，环境句柄会释放所有与之相连的句柄。

注意：当一个父句柄被释放后，任何与其相连的句柄都会被自动释放。

对 OCIServerDetach()函数和 OCISessionEnd()函数的调用不是必须的，但是建议进行调用。如果应用程序终止并且没有调用 OCITransCommit()函数没有被调用，则任何之前的事务都会被自动回滚。

2.Go 语言中 database/sql 接口说明

Go 与 PHP 不同的地方是 Go 没有官方提供数据库驱动，而是为开发者开发数据库驱动定义了一些标准接口，开发者可以根据定义的接口来开发相应的数据库驱动，这样做有一个好处，只要按照标准接口开发的代码， 以后需要迁移数据库时，不需要任何修改。那么 Go 都定义了哪些标准接口呢？让我们来详细的分析一下。

sql.Register

这个存在于 database/sql 的函数是用来注册数据库驱动的，当第三方开发者开发数据库驱动时，都会实现 init 函数，在 init 里面会调用这个 Register(name string, driver driver.Driver)完成本驱动的注册。

我们来看一下 mymysql、sqlite3 的驱动里面都是怎么调用的：

```
//https://github.com/mattn/go-sqlite3 驱动
func init() {
  sql.Register("sqlite3", &SQLiteDriver{})
}

//https://github.com/mikespook/mymysql 驱动
// Driver automatically registered in database/sql
var d = Driver{proto: "tcp", raddr: "127.0.0.1:3306"}
func init() {
  Register("SET NAMES utf8")
```

```
    sql.Register("mymysql", &d)
}
```

我们看到第三方数据库驱动都是通过调用这个函数来注册自己的数据库驱动名称以及相应的 driver 实现。在 database/sql 内部通过一个 map 来存储用户定义的相应驱动。

```
var drivers = make(map[string]driver.Driver)
drivers[name] = driver
```

因此通过 database/sql 的注册函数可以同时注册多个数据库驱动，只要不重复。在我们使用 database/sql 接口和第三方库的时候经常看到如下:

```
import (
    "database/sql"
    _ "github.com/mattn/go-sqlite3"
)
```

新手都会被这个 _ 所迷惑，其实这个就是 Go 设计的巧妙之处，我们在变量赋值的时候经常看到这个符号，它是用来忽略变量赋值的占位符，那么包引入用到这个符号也是相似的作用，这儿使用 _ 的意思是引入后面的包名而不直接使用这个包中定义的函数，变量等资源，但包在引入的时候会自动调用包的 init 函数以完成对包的初始化。

driver.Driver

Driver 是一个数据库驱动的接口，他定义了一个 method： Open(name string)，这个方法返回一个数据库的 Conn 接口。

```
type Driver interface {
    Open(name string) (Conn, error)
}
```

返回的 Conn 只能用来进行一次 goroutine 的操作，也就是说不能把这个 Conn 应用于 Go 的多个 goroutine 里面。如下代码会出现错误

```
...
go goroutineA (Conn)  //执行查询操作
go goroutineB (Conn)  //执行插入操作
...
```

上面这样的代码可能会使 Go 不知道某个操作究竟是由哪个 goroutine 发起的,从而导致数据混乱，比如可能会把 goroutineA 里面执行的查询操作的结果返回给 goroutineB 从而使 B 错误地把此结果当成自己执行的插入数据。

第三方驱动都会定义这个函数，它会解析 name 参数来获取相关数据库的连接信息，解析完成后，它将使用此信息来初始化一个 Conn 并返回它。

driver.Conn

Conn 是一个数据库连接的接口定义，他定义了一系列方法，这个 Conn 只能应用在一个 goroutine 里面，不能使用在多个 goroutine 里面，详情请参考上面的说明。

```
type Conn interface {
    Prepare(query string) (Stmt, error)
    Close() error
```

```
    Begin() (Tx, error)
}
```

Prepare 函数返回与当前连接相关的执行 Sql 语句的准备状态，可以进行查询、删除等操作。

Close 函数关闭当前的连接，执行释放连接拥有的资源等清理工作。因为驱动实现了 database/sql 里面建议的 conn pool，所以你不用再去实现缓存 conn 之类的，这样会容易引起问题。

Begin 函数返回一个代表事务处理的 Tx，通过它你可以进行查询,更新等操作，或者对事务进行回滚、递交。

driver.Stmt

Stmt 是一种准备好的状态，和 Conn 相关联，而且只能应用于一个 goroutine 中，不能应用于多个 goroutine。

```
type  Stmt  interface {
    Close() error
    NumInput() int
    Exec(args []Value) (Result, error)
    Query(args []Value) (Rows, error)
}
```

Close 函数关闭当前的链接状态，但是如果当前正在执行 query，query 还是有效返回 rows 数据。

NumInput 函数返回当前预留参数的个数，当返回>=0 时数据库驱动就会智能检查调用者的参数。当数据库驱动包不知道预留参数的时候，返回-1。

Exec 函数执行 Prepare 准备好的 sql，传入参数执行 update/insert 等操作，返回 Result 数据

Query 函数执行 Prepare 准备好的 sql，传入需要的参数执行 select 操作，返回 Rows 结果集

driver.Tx

事务处理一般就两个过程，递交或者回滚。数据库驱动里面也只需要实现这两个函数就可以

```
type Tx interface {
    Commit() error
    Rollback() error
}
```

这两个函数一个用来递交一个事务，一个用来回滚事务。

driver.Execer

这是一个 Conn 可选择实现的接口

```
type Execer interface {
    Exec(query string, args []Value) (Result, error)
}
```

如果这个接口没有定义，那么在调用 DB.Exec,就会首先调用 Prepare 返回 Stmt，然后执行 Stmt 的 Exec，然后关闭 Stmt。

driver.Result

这个是执行 Update/Insert 等操作返回的结果接口定义

```
type Result interface {
    LastInsertId() (int64, error)
    RowsAffected() (int64, error)
}
```

LastInsertId 函数返回由数据库执行插入操作得到的自增 ID 号。

RowsAffected 函数返回 query 操作影响的数据条目数。

driver.Rows

Rows 是执行查询返回的结果集接口定义

```
type Rows interface {
    Columns() []string
    Close() error
    Next(dest []Value) error
}
```

Columns 函数返回查询数据库表的字段信息，这个返回的 slice 和 sql 查询的字段一一对应，而不是返回整个表的所有字段。

Close 函数用来关闭 Rows 迭代器。

Next 函数用来返回下一条数据，把数据赋值给 dest。dest 里面的元素必须是 driver.Value 的值除了 string，返回的数据里面所有的 string 都必须要转换成 []byte。如果最后没数据了，Next 函数最后返回 io.EOF。

driver.RowsAffected

RowsAffested 其实就是一个 int64 的别名，但是他实现了 Result 接口，用来底层实现 Result 的表示方式

```
type RowsAffected int64

func (RowsAffected) LastInsertId() (int64, error)

func (v RowsAffected) RowsAffected() (int64, error)
```

driver.Value

Value 其实就是一个空接口，他可以容纳任何的数据

type Value interface{}

drive 的 Value 是驱动必须能够操作的 Value，Value 要么是 nil，要么是下面的任意一种

int64

float64

bool

[]byte

string　[*]除了 Rows.Next 返回的不能是 string.

time.Time

driver.ValueConverter

ValueConverter 接口定义了如何把一个普通的值转化成 driver.Value 的接口

```
type ValueConverter interface {
    ConvertValue(v interface{}) (Value, error)
}
```

在开发的数据库驱动包里面实现这个接口的函数在很多地方会使用到，这个 ValueConverter 有很多好处：

- 转化 driver.value 到数据库表相应的字段，例如 int64 的数据如何转化成数据库表 uint16 字段
- 把数据库查询结果转化成 driver.Value 值
- 在 scan 函数里面如何把 dirve.Value 值转化成用户定义的值

driver.Valuer

Valuer 接口定义了返回一个 driver.Value 的方式

```
type Valuer interface {
    Value() (Value, error)
}
```

很多类型都实现了这个 Value 方法，用来自身与 driver.Value 的转化。

通过上面的讲解，你应该对于驱动的开发有了一个基本的了解，一个驱动只要实现了这些接口就能完成增删查改等基本操作了，剩下的就是与相应的数据库进行数据交互等细节问题了，在此不再赘述。

database/sql

database/sql 在 database/sql/driver 提供的接口基础上定义了一些更高阶的方法，用以简化数据库操作,同时内部还建议性地实现一个 conn pool。

```
type DB struct {
    driver   driver.Driver
    dsn      string
    mu       sync.Mutex // protects freeConn and closed
    freeConn []driver.Conn
    closed   bool
}
```

我们可以看到 Open 函数返回的是 DB 对象，里面有一个 freeConn，它就是那个简易的连接池。它的实现相当简单或者说简陋，就是当执行 Db.prepare 的时候会 defer db.putConn(ci, err),也就是把这个连接放入连接池，每次调用 conn 的时候会先判断 freeConn 的长度是否大于 0，大于 0 说明有可以复用的 conn，直接拿出来用就是了，如果不大于 0，则创建一个 conn,然后再返回之。

3.利用 Oracle Oci 定义 database/sql 接口

在 https://github.com/liudch?tab=repositories 上我搜集了五种 go 连接 oracle 数据库的 oci 实现，比较下来，https://github.com/liudch/goci-1 实现的比较好，这里以此实现来说明接口的设计。

driver.go

```
package goci

/*
```

```
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib

*/
import "C"
import (
    "database/sql"
    "database/sql/driver"
    "errors"
    "unsafe"
)

type drv struct{}

func init() {
    sql.Register("goci", &drv{})
}

func (d *drv) Open(dsn string) (driver.Conn, error) {
    conn := &connection{}

    // initialize the oci environment used for all other oci calls
    result := C.OCIEnvCreate((**C.OCIEnv)(unsafe.Pointer(&conn.env)), C.
OCI_DEFAULT, nil, nil, nil, nil, 0, nil)
    if result != C.OCI_SUCCESS {
        return nil, errors.New("Failed: OCIEnvCreate()")
    }

    // error handle
    result = C.OCIHandleAlloc(conn.env, &conn.err, C.OCI_HTYPE_ERROR, 0,
 nil)
    if result != C.OCI_SUCCESS {
        return nil, errors.New("Failed: OCIHandleAlloc() - creating err
or handle")
    }
```

```
    // Log in the user
    err := conn.performLogon(dsn)
    if err != nil {
        return nil, err
    }
    return conn, nil
}

func ociGetError(err unsafe.Pointer) error {
    var errcode C.sb4
    var errbuff [512]C.char
    C.OCIErrorGet(err, 1, nil, &errcode, (*C.OraText)(unsafe.Pointer(&e
rrbuff[0])), 512, C.OCI_HTYPE_ERROR)
    s := C.GoString(&errbuff[0])
    return errors.New(s)
}
```

conn.go

```
package goci

/*
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib

*/
import "C"
import (
    "database/sql/driver"
    "strings"
    "unsafe"
)

type connection struct {
    env unsafe.Pointer
```

```
    err unsafe.Pointer
    svr unsafe.Pointer
}

func (conn *connection) performLogon(dsn string) error {

    user, pwd, host := parseDsn(dsn)
    puser := C.CString(user)
    defer C.free(unsafe.Pointer(puser))
    ppwd := C.CString(pwd)
    defer C.free(unsafe.Pointer(ppwd))
    phost := C.CString(host)
    defer C.free(unsafe.Pointer(phost))

    result := C.OCILogon2((*C.OCIEnv)(unsafe.Pointer(conn.env)),
        (*C.OCIError)(conn.err),
        (**C.OCIServer)(unsafe.Pointer(&conn.svr)),
        (*C.OraText)(unsafe.Pointer(puser)),
        C.ub4(C.strlen(puser)),
        (*C.OraText)(unsafe.Pointer(ppwd)),
        C.ub4(C.strlen(ppwd)),
        (*C.OraText)(unsafe.Pointer(phost)),
        C.ub4(C.strlen(phost)),
        C.OCI_LOGON2_STMTCACHE)
    if result != C.OCI_SUCCESS {
        return ociGetError(conn.err)
    }
    return nil
}

func (conn *connection) exec(cmd string) error {
  stmt, err := conn.Prepare(cmd)
  if err == nil {
    defer stmt.Close()
    _, err = stmt.Exec(nil)
  }
  return err
}

func (conn *connection) Begin() (driver.Tx, error) {
  if err := conn.exec("BEGIN"); err != nil {
```

```
    return nil, err
  }
  return &transaction{conn}, nil
}

func (conn *connection) Prepare(query string) (driver.Stmt, error) {
    pquery := C.CString(query)
    defer C.free(unsafe.Pointer(pquery))
    var stmt unsafe.Pointer

    if C.OCIHandleAlloc(conn.env, &stmt, C.OCI_HTYPE_STMT, 0, nil) != C.
OCI_SUCCESS {
        return nil, ociGetError(conn.err)
    }
    result := C.OCIStmtPrepare((*C.OCIStmt)(stmt), (*C.OCIError)(conn.e
rr),
        (*C.OraText)(unsafe.Pointer(pquery)), C.ub4(C.strlen(pquery)),
        C.ub4(C.OCI_NTV_SYNTAX), C.ub4(C.OCI_DEFAULT))
    if result != C.OCI_SUCCESS {
        return nil, ociGetError(conn.err)
    }
    return &statement{handle: stmt, conn: conn}, nil
}

func (conn *connection) Close() error {
    return nil
}

// Makes a lightweight call to the server. A successful result indicate
s the server is active.  A block indicates the connection may be in use
 by
// another thread. A failure indicates a communication error.
func (conn *connection) ping() error {
    if C.OCIPing((*C.OCIServer)(conn.svr), (*C.OCIError)(conn.err), C.O
CI_DEFAULT) != C.OCI_SUCCESS {
        return ociGetError(conn.err)
    }
    return nil
}

// expect the dsn in the format of: user/pwd@host:port/SID
```

```
func parseDsn(dsn string) (user, pwd, host string) {
    tokens := strings.SplitN(dsn, "@", 2)
    if len(tokens) > 1 {
        host = tokens[1]
    }
    userpass := strings.SplitN(tokens[0], "/", 2)
    if len(userpass) > 1 {
        pwd = userpass[1]
    }
    user = userpass[0]
    return
}
```

statement.go

```
package goci

/*
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib

*/
import "C"
import (
    "database/sql/driver"
    "fmt"
    "log"
    "unsafe"
)

type statement struct {
    handle unsafe.Pointer
    conn   *connection
    closed bool
}
```

```
func (stmt *statement) Close() error {
    if stmt.closed {
        return nil
    }
    stmt.closed = true
    C.OCIHandleFree(stmt.handle, C.OCI_HTYPE_STMT)
    stmt.handle = nil

    return nil
}

func (stmt *statement) NumInput() int {
    var num C.int
    if r := C.OCIAttrGet(stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer
(&num), nil, C.OCI_ATTR_BIND_COUNT, (*C.OCIError)(stmt.conn.err)); r !=
 C.OCI_SUCCESS {
        log.Println(ociGetError(stmt.conn.err))
    }
    return int(num)
}

// Exec executes a query that doesn't return rows, such
// as an INSERT or UPDATE.
func (stmt *statement) Exec(args []driver.Value) (driver.Result, error)
 {
    if err := stmt.bind(args); err != nil {
        return nil, err
    }

    if C.OCIStmtExecute((*C.OCIServer)(stmt.conn.svr), (*C.OCIStmt)(stm
t.handle), (*C.OCIError)(stmt.conn.err), 1, 0, nil, nil, C.OCI_DEFAULT)
 != C.OCI_SUCCESS {
        return nil, ociGetError(stmt.conn.err)
    }
    return &result{stmt}, nil
}

// Exec executes a query that may return rows, such as SELECT.
func (stmt *statement) Query(v []driver.Value) (driver.Rows, error) {
    if err := stmt.bind(v); err != nil {
        return nil, err
```

```
    }

    // determine the type of statement.  For select statements iter is
set to zero, for other statements, only execute once.
    // * NOTE * Should an error be returned if the statement is a non-s
elect type?
    var stmt_type C.int
    if C.OCIAttrGet(stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer(&stmt
_type), nil, C.OCI_ATTR_STMT_TYPE, (*C.OCIError)(stmt.conn.err)) != C.O
CI_SUCCESS {
        log.Println(ociGetError(stmt.conn.err))
    }
    iter := C.ub4(1)
    if stmt_type == C.OCI_STMT_SELECT {
        iter = 0
    }
    // set the row prefetch.  Only one extra row per fetch will be retu
rned unless this is set.
    prefetchSize := C.ub4(100)
    if C.OCIAttrSet(stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer(&pref
etchSize), 0, C.OCI_ATTR_PREFETCH_ROWS, (*C.OCIError)(stmt.conn.err)) !
= C.OCI_SUCCESS {
        log.Println(ociGetError(stmt.conn.err))
    }

    // execute the statement
    if C.OCIStmtExecute((*C.OCIServer)(stmt.conn.svr), (*C.OCIStmt)(stm
t.handle), (*C.OCIError)(stmt.conn.err), iter, 0, nil, nil, C.OCI_DEFAU
LT) != C.OCI_SUCCESS {
        err := ociGetError(stmt.conn.err)
        log.Println(err)
        return nil, err
    }

    // find out how many output columns there are
    var cols C.ub2
    if C.OCIAttrGet(stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer(&col
s), nil, C.OCI_ATTR_PARAM_COUNT, (*C.OCIError)(stmt.conn.err)) != C.OCI
_SUCCESS {
        err := ociGetError(stmt.conn.err)
        log.Println(err)
```

```
        return nil, err
    }

    // build column meta-data
    columns := make([]column, int(cols))
    for pos := 0; pos < int(cols); pos++ {
        col := &columns[pos]
        var param unsafe.Pointer
        var colType C.ub2
        var colSize C.ub4
        var colName *C.char
        var nameSize C.ub4

        if C.OCIParamGet(stmt.handle, C.OCI_HTYPE_STMT, (*C.OCIError)(s
tmt.conn.err), (*unsafe.Pointer)(unsafe.Pointer(&param)), C.ub4(pos+1))
 != C.OCI_SUCCESS {
            err := ociGetError(stmt.conn.err)
            log.Println(err)
            return nil, err
        }

        C.OCIAttrGet(param, C.OCI_DTYPE_PARAM, unsafe.Pointer(&colType),
 nil, C.OCI_ATTR_DATA_TYPE, (*C.OCIError)(stmt.conn.err))
        C.OCIAttrGet(param, C.OCI_DTYPE_PARAM, unsafe.Pointer(&colName),
 &nameSize, C.OCI_ATTR_NAME, (*C.OCIError)(stmt.conn.err))
        C.OCIAttrGet(param, C.OCI_DTYPE_PARAM, unsafe.Pointer(&colSize),
 nil, C.OCI_ATTR_DATA_SIZE, (*C.OCIError)(stmt.conn.err))

        col.kind = int(colType)
        col.size = int(colSize)
        col.name = C.GoStringN(colName, (C.int)(nameSize))
        col.raw = make([]byte, int(colSize+1))

        var def *C.OCIDefine
        result := C.OCIDefineByPos((*C.OCIStmt)(stmt.handle), &def, (*C.
OCIError)(stmt.conn.err),
            C.ub4(pos+1), unsafe.Pointer(&col.raw[0]), C.sb4(colSize+1),
 C.SQLT_CHR, nil, nil, nil, C.OCI_DEFAULT)
        if result != C.OCI_SUCCESS {
            return nil, ociGetError(stmt.conn.err)
        }
```

```
    }

    return &rows{stmt, columns}, nil
}

type column struct {
    name string
    kind int
    size int
    raw  []byte
}

func (stmt *statement) bind(args []driver.Value) error {
    var binding *C.OCIBind
    for pos, value := range args {
        buffer := []byte(fmt.Sprint("%v", value))
        buffer = append(buffer, 0)
        result := C.OCIBindByPos((*C.OCIStmt)(stmt.handle), &binding,
(*C.OCIError)(stmt.conn.err), C.ub4(pos+1),
            unsafe.Pointer(&buffer[0]), C.sb4(len(buffer)), C.SQLT_STR,
 nil, nil, nil, 0, nil, C.OCI_DEFAULT)
        if result != C.OCI_SUCCESS {
            return ociGetError(stmt.conn.err)
        }
    }
    return nil
}
```

rows.go

```
package goci

/*
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib
```

```
*/
import "C"
import (
    "database/sql/driver"
    "io"
  "fmt"
)

type rows struct {
    stmt    *statement
    columns []column
}

// Columns returns the names of the columns. The number of
// columns of the result is inferred from the length of the
// slice.  If a particular column name isn't known, an empty
// string should be returned for that entry.
func (r *rows) Columns() []string {
    names := make([]string, len(r.columns))
    for pos, column := range r.columns {
        names[pos] = column.name
    }
    return names

}

// Close closes the rows iterator.
func (r *rows) Close() error {
  return r.stmt.Close()
}

// Next is called to populate the next row of data into
// the provided slice. The provided slice will be the same
// size as the Columns() are wide.
//
// The dest slice may be populated only with
// a driver Value type, but excluding string.
// All string values must be converted to []byte.
//
// Next should return io.EOF when there are no more rows.
func (r *rows) Next(dest []driver.Value) error {
```

```
    rv := C.OCIStmtFetch2((*C.OCIStmt)(r.stmt.handle), (*C.OCIError)(r.
stmt.conn.err), 1, C.OCI_FETCH_NEXT, 1, C.OCI_DEFAULT)
  if rv == C.OCI_ERROR {
    err :=  ociGetError(r.stmt.conn.err)
    fmt.Println(err)
    return ociGetError(r.stmt.conn.err)
  }
  if rv == C.OCI_NO_DATA {
    fmt.Println("no data")
    return io.EOF
  }

  for i := range dest {
    dest[i] = string(r.columns[i].raw)
  }
    return nil
}
```

result.go

```
package goci

/*
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib

*/
import "C"
import (
    "unsafe"
)

type result struct {
    stmt *statement
}
```

```
func (r *result) LastInsertId() (int64, error) {
    var t C.ub4
    if C.OCIAttrGet(r.stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer(&t),
 nil, C.OCI_ATTR_ROWID, (*C.OCIError)(r.stmt.conn.err)) != C.OCI_SUCCES
S {
        return 0, ociGetError(r.stmt.conn.err)
    }
    return int64(t), nil
}

func (r *result) RowsAffected() (int64, error) {
    var t C.ub4
    if C.OCIAttrGet(r.stmt.handle, C.OCI_HTYPE_STMT, unsafe.Pointer(&t),
 nil, C.OCI_ATTR_ROW_COUNT, (*C.OCIError)(r.stmt.conn.err)) != C.OCI_SU
CCESS {
        return 0, ociGetError(r.stmt.conn.err)
    }
    return int64(t), nil
}
```

transaction.go

```
package goci

/*
#include <oci.h>
#include <stdlib.h>
#include <string.h>

#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/pu
blic
#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_
1/lib

*/
import "C"

type transaction struct {
    conn *connection
}

func (tx *transaction) Commit() error {
```

```
    if err := tx.conn.exec("COMMIT"); err != nil {
        return err
    }
    return nil
}

func (tx *transaction) Rollback() error {
    if err := tx.conn.exec("ROLLBACK"); err != nil {
        return err
    }
    return nil
}
```

4.编译说明

编译注意事项：

在相关程序中加了 cgo 编译时需加的头文件和连接库

```
#cgo CFLAGS: -I/home/oracle/app/oracle/product/11.2.0/client_1/rdbms/public

#cgo LDFLAGS: -lclntsh -L/home/oracle/app/oracle/product/11.2.0/client_1/lib
```

您在使用时，请将 /home/oracle/app/oracle/product/11.2.0/client_1 修改为你的系统中 $ORACLE_HOME 的字串值

如果你使用简易的安装包，

请在 http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html

下载：

oracle-instantclient11.2-basic-11.2.0.3.0-1.x86_64.rpm

oracle-instantclient11.2-devel-11.2.0.3.0-1.x86_64.rpm

并安装和设置 oracle 环境变量。

同时需设置

```
export TNS_ADMIN=/home/oracle/app/oracle/product/11.2.0/client_1/network/admin
```

ORACLE_HOME 以您系统实际情况进行修改。

5.应用举例

```
package main

  import (
        "database/sql"
        _ "goci"  // 根据实际部署情况修改
        "os"
        "log"
  )


  func main() {
        // 为 log 添加短文件名,方便查看行数
        log.SetFlags(log.Lshortfile | log.LstdFlags)


        log.Println("Oracle Driver example")
```

```
        os.Setenv("NLS_LANG", "")
        dsn := os.Getenv("ORACLE_DSN") // 把用户名/口令@SID 定义到此环境变量中
        if dsn == "" {
        os.Exit(2) // 出错退出
        }
        db, _ := sql.Open("goci",dsn)

        rows, err := db.Query("select 3.14, 'foo' from dual")
        if err != nil {
                log.Fatal(err)
        }
        defer db.Close()

        for rows.Next() {
                var f1 float64
                var f2 string
                rows.Scan(&f1, &f2)
                log.Println(f1, f2) // 3.14 foo
        }
        rows.Close()
```

```
        // 先删表,再建表
        db.Exec("drop table sdata")
        db.Exec("create table sdata(name varchar2(256))")

        db.Exec("insert into sdata values('中文')")
        db.Exec("insert into sdata values('1234567890ABCabc!@#$%^&*()_+')")

        rows, err = db.Query("select * from sdata")
        if err != nil {
                log.Fatal(err)
        }

        for rows.Next() {
                var name string
                rows.Scan(&name)
                log.Printf("Name = %s, len=%d", name, len(name))
        }
        rows.Close()
  }
```

此程序在 redhat server 6 的 64 位机器上编译测试正常通过。

6.Go 中有关时间的操作

24 时区，GMT，UTC，DST，CST 时间介绍

全球 24 个时区的划分

相较于两地时间表，可以显示世界各时区时间和地名的世界时区表（World Time），就显得精密与复杂多了，通常世界时区表的表盘上会标示着全球 24 个时区的城市名称，但究竟这 24 个时区是如何产生的？过去世界各地原本各自订定当地时间，但随着交通和电讯的发达，各地交流日益频繁，不同的地方时间，造成许多困扰，于是在西元 1884 年的国际会议上制定了全球性的标准时，明定以英国伦敦格林威治这个地方为零度经线的起点（亦称为本初子午线），并以地球由西向东每 24 小时自转一周 360°，订定每隔经度 15°，时差 1 小时。而每 15°的经线则称为该时区的中央经线，将全球划分为 24 个时区，其中包含 23 个整时区及 180°经线左右两侧的 2 个半时区。就全球的时间来看，东经的时间比西经要早，也就是如果格林威治时间是中午 12 时，则中央经线 15°E 的时区为下午 1 时，中央经线 30°E 时区的时间为下午 2 时；反之，中央经线 15°W 的时区时间为上午 11 时，中央经线 30°W 时区的时间为上午 10 时。以台湾为例，台湾位于东经 121°，换算后与格林威治就有 8 小时的时差。如果两人同时从格林威治的 0°各往东、西方前进，当他们在经线 180°时，就会相差 24 小时，所以经线 180°被定为国际换日线，由西向东通过此线时日期要减去一日，反之，若由东向西则要增加一日。

格林威治标准时间 GMT

十七世纪，格林威治皇家天文台为了海上霸权的扩张计画而进行天体观测。1675 年旧皇家观测所(Old Royal Observatory) 正式成立，到了 1884 年决定以通过格林威治的子午线作为划分地球东西两半球的经度零度。观测所门口墙上有一个标志 24 小时的时钟，显示当下的时间，对全球而言，这里所设定的时间是世界时间参考点，全球都以格林威治的时间作为标准来设定时间，这就是我们耳熟能详的「格林威治标准时间」(Greenwich Mean Time，简称 G.M.T.)的由来，标示在手表上，则代表此表具有两地时间功能，也就是同时可以显示原居地和另一个国度的时间。

世界协调时间 UTC

多数的两地时间表都以 GMT 来表示，但也有些两地时间表上看不到 GMT 字样，出现的反而是 UTC 这 3 个英文字母，究竟何谓 UTC？事实上，UTC 指的是 Coordinated Universal Time - 世界协调时间（又称世界标准时间、世界统一时间），是经过平均太阳时(以格林威治时间 GMT 为准)、地轴运动修正后的新时标以及以「秒」为单位的国际原子时所综合精算而成的时间，计算过程相当严谨精密，因此若以「世界标准时间」的角度来说，UTC 比 GMT 来得更加精准。其误差值必须保持在 0.9 秒以内，若大于 0.9 秒则由位于巴黎的国际地球自转事务中央局发布闰秒，使 UTC 与地球自转周期一致。所以基本上 UTC 的本质强调的是比 GMT 更为精确的世界时间标准，不过对于现行表款来说，GMT 与 UTC 的功能与精确度是没有差别的。

夏日节约时间 DST

所谓「夏日节约时间」Daylight Saving Time（简称 D.S.T.），是指在夏天太阳升起的比较早时，将时钟拨快一小时，以提早日光的使用，在英国则称为夏令时间(Summer Time)。这个构想于 1784 年由美国班杰明·富兰克林提出来，1915 年德国成为第一个正式实施夏令日光节约时间的国家，以削减灯光照明和耗电开支。自此以后，全球以欧洲和北美为主的约 70 个国家都引用这个做法。目前被划分成两个时区的印度也正在商讨是否全国该统一实行夏令日光节约时间。欧洲手机上也有很多 GSM 系统的基地台，除了会传送当地时间外也包括夏令日光节约时间，做为手机的时间标准，使用者可以自行决定要开启或关闭。值得注意的是，某些国家有实施「夏日节约时间」的制度，出国时别忘了跟随当地习惯在表上调整一下，这可是机械表没有的功能设计哦！

CST 时间

CST 却同时可以代表如下 4 个不同的时区：

Central Standard Time (USA) UT-6:00

Central Standard Time (Australia) UT+9:30

China Standard Time UT+8:00

Cuba Standard Time UT-4:00

可见，CST 可以同时表示美国，澳大利亚，中国，古巴四个国家的标准时间。

【摘自：http://www.douban.com/note/147740972/ 】

Oracle 编程中 go 语言中的时间处理

我们以 goracle 包（https://github.com/bradfitz/go-sql-test/tree/master/src/github.com/tgulacsi）为例，用一个简单的例子说明。

```
1  package main
2  import (
3          "database/sql"
4          _ "engine/goracle"
5          "log"
6          "time"
7  )
8  func main() {
9          log.Println("Oracle Driver example")
10         db, _ := sql.Open("goracle", "dbname/passwd@query")
11         rows, err := db.Query("SELECT contno,cvalidate,appntname FROM lccont WHERE contno like '8210100%'")
12         if err != nil {
13                 log.Fatal(err)
```

```
14          }
15          defer db.Close()
16          for rows.Next() {
17                  var f1 string
18                  //var f2 []byte
19                  var f2 time.Time
20                  var f3 string
21                  rows.Scan(&f1, &f2, &f3)
22                  log.Println(f1, f2, f3)
23          }
24          rows.Close()
25  }
```

第 4 行 引入 goracle 包

第 17 行定义字符串变量来保存单证号码

第 18 行调试过 BLOB 等类可用 [] byte 读取

第 19 行为时间类型变量，用 time.Time

编译后的运行结果片段：

```
2013/03/27 11:46:09 82101000000000000000 2013-03-26 00:00:00 +0800 CST 王某

2013/03/27 11:46:09 82101000000000000001 2013-03-26 00:00:00 +0800 CST 赵某
```

其中的时间、中文均正常。

如果将第 17 行 var f1 string 修改为 var f1 []byte，编译并运行的结果片段如下，其中的点点为替换原数据值。

```
2013/03/27 11:51:39 [56 50 ..........] 2013-03-26 00:00:00 +0800 CST 王某

2013/03/27 11:51:39 [56 50..........] 2013-03-26 00:00:00 +0800 CST 赵某
```

后 记

本文绝大多数资料来源于互联网，本人经过整理，提供大家参考。下面给出go 资源链接，仅供参考。

1.官网

http://golang.org/

http://tour.golang.org/#1

http://golang.org/doc/

2.国内几个博客：

http://www.cnblogs.com/zitsing/tag/go/

http://www.cnblogs.com/yjf512/category/385369.html

http://www.mikespook.com/learning-go/

3.Golang 中文社区 分享交流

http://bbs.gocn.im/forum.php

4.golang-china - Go 语言中文翻译项目

http://code.google.com/p/golang-china/

5.go 语言指南

http://tour.golang.tc/#1

6.Go web 编程

https://github.com/astaxie/build-web-application-with-golang/blob/master/preface.md

7.go 代码片段分享

http://www.sharejs.com/codes/go/?start=20

8.让我们一起 GO

http://my.oschina.net/615stu/blog/102395

9.Go Web 编程框架

http://robfig.github.com/revel/

https://github.com/astaxie/beego

https://github.com/golangers/framework

https://github.com/ungerik/go-start

https://github.com/hoisie/web

http://www.getwebgo.com/

http://go-start.org/

https://github.com/paulbellamy/mango

http://labix.org/mgo

https://github.com/eaigner/hood

https://github.com/paulbellamy/mango

https://github.com/hoisie/web.go

https://github.com/qleelulu/goku

http://code.google.com/p/go-fastweb/source/checkout

http://code.google.com/p/goweb/

10.GO 库列表

http://go-lang.cat-v.org/pure-go-libs

11.EXCEL 2007 操作库

https://github.com/tealeg

12.GBK 和 UTF-8 字符转换。

https://github.com/xushiwei/go-iconv

http://bbs.gocn.im/thread-136-1-1.html

13.go 语言在 linux 下如何连接 oracle 数据库

http://www.oschina.net/question/110132_91394

14.许式伟老师 杰作汇集

https://github.com/xushiwei/

15.QQ 群：

Go Web 编程交流群 259315004

Go 语言精英群　　29994666

Golang online　　259718627

16.本示范参考自：

https://github.com/astaxie/build-web-application-with-golang/blob/master/ebook/05.1.mdy

https://github.com/egravert/goci

https://github.com/liudch