Using SQL in Go Applications

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

Many applications work on some set of local data. Even some command line applications need to keep data across invocations. Flat files in a columnar or JSON format work for many cases. However, it can get to the point where a more structured approach can make life easier. SQL databases are the typical next stopping point for a structured approach to data.

Go has a generic interface around SQL with `database/sql` in the standard library. The interface supports drivers which provide the backing to common database technologies. A list of common drivers is available on the Go wiki: https://github.com/golang/go/wiki/SQLDrivers. While most of these are dependent on an external data service, one, SQLite, is not.

SQLite is a self-contained SQL database engine. It stores its data in a file, which makes it easy to embed in local applications. The underlying implementation is in C and has many common language bindings, including several for Go. In Go, this does require cgo support which should, in general, work. However, be aware that it may require additional C compiler binaries to be installed, and cross compilation will require even more.

In this article, we're going to work with the Go SQL interface, specifically the github.com/mattn/go-sqlite3 driver.

The code for these examples can be found at https://github.com/cmceniry/login in the `sql` directory. This code is using dep for dependency management, but this should work with Go modules as well. After downloading the code, you can run each example directly from the main package's directory (`login/sql`) with `go run EXAMPLE/main.go`. The examples use the same example database which will get created in the main package's directory. If you change directories out of that, it may get confused.

Note: As mentioned, you may also need to install SQLite development packages in your environment to complete these examples.

The SQL Interface

The SQL interface provides a simple way to perform the most common SQL methods: open and close a database, execute a Data Definition Language (DDL) or Data Manipulation Language (DML) statement, and perform a query. SQL abstracts away much of the overhead such as connecting to the database, handling connection pooling, and performing connection cleanup.

Since it is an interface, the expectation is to interact with **all** databases the same way, regardless of back-end driver.

Import

The `database/sql` driver mechanism relies on the blank identifier, _, import. All of the examples use this import format.

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

As normal, the blank identifier indicates to ignore an item. In this case, it's ignoring all of the exported identifiers from the `go-sqlite3` package. Our code will not be using any of the possible functions or variables from `go-sqlite3` directly.

However, the normal import actions still happen. This includes the variable definitions and initialization mechanisms. Inside the `go-sqlite` module is an `init` function. On the first import of a package, it runs this `init` function. In this case, it registers itself with the `database/sql` drivers available and makes it available as a back end.

github.com/mattn/go-sqlite/sqlite3.go.

```
func init() {
  sql.Register("sqlite3", &SQLiteDriver{})
  }
```

Other libraries enhance the Go runtime using the blank identifier. The standard HTTP profiling library, `net/http/pprof`, is another example of a library that you do not call directly. This is a practice that you can use for your code, but use it with caution.

Note: There is a common order to how the `init` functions (and package-level variables) are run: imported packages and then alphabetical by package file within a package. However, it is still very easy to put yourself in a situation where you are attempting to use them in a different order.

Creating a DB

In our first example, we will create a simple database. The database will be defined with a simple schema:

create/main.go: schema.

```
var schema = CREATE TABLE sample (
  i INTEGER,
  s TEXT,
  t DATETIME DEFAULT CURRENT_TIMESTAMP
  )
```

With this schema in hand, we can start initializing our database. We begin our `main` function with a call to open the database. The arguments to `Open` tell the SQL interface which driver to use with which options. The options are specific to the driver—in this case, "read," "write," and "create." We then rely on Go's `defer` mechanism to ensure that we close the database when we're done.

create/main.go: opencreate,close.

```
func main() {
    db, err := sql.Open("sqlite3", "file:testdb?mode=rwc")
    ...
    defer db.Close()
```

With the open database, we now create our schema in it. We can call the `Exec` function on the database and pass in the schema string as the argument. `Exec` returns two values—a result and an error. The result is meaningless for DDL statements, so the main concern here is to receive the error. For the example, handling the error is a simple `panic`.

create/main.go: exec.

```
    _, err = db.Exec(schema)
    if err != nil {
        panic(err)
    }
```

We will see this same `Exec` function in the next example and will examine the result.

Insert

Once the database is initialized, we can start feeding data into it. Since this is a new process, we need to reopen the database. In this case, we don't want to create it, so we will leave off the "create" option to open.

insert/main.go: open,close.

```
func main() {
    db, err := sql.Open("sqlite3", "file:testdb?mode=rw")
    ...
    defer db.Close()
```

With the open database, we can add data to it like any other SQL data addition—`INSERT`. As with the previous example, we use the `Exec` function to perform the insert. The first argument to `Exec` is the SQL statement to execute—in this case, a simple insert into the `sample` table of an integer and a string. While SQLite uses dynamic typing, we're still using parameterized bind variables, ?, instead of combining our values directly with our SQL statement. This provides two large benefits: First, we do not have to handle the type conversion into the statement. (This type handling will show up again in the next example, `query`.) Second, this form is much less susceptible to SQL injection attacks. The remaining arguments to `Exec` are bound to the respective positional ?. `Exec` is variadic in that the number of arguments is dependent on the SQL statement.

insert/main.go: query.

```
    res, err := db.Exec(
        "INSERT INTO sample (i, s) VALUES (?, ?)",
        2,
        "2",
    )
```

If there is a syntax or back-end issue, an error will be returned. After checking the error, we also want to confirm how many rows were inserted. For inserts, this may not matter as much, but in other cases (`SET`) it can indicate an issue in data or logic. We obtain the numbers of rows inserted with the `RowsAffected` method of our result.

insert/main.go: rows.

```
        affected, err := res.RowsAffected()
```

With the value in hand, we can print it out and visually inspect it.

insert/main.go: print.

```
        fmt.Printf("%d row(s) inserted\n", affected)
```

The output of this should be fairly simple:

```
    $ go run insert/main.go
    1 row(s) inserted
```

`RowsAffected` is really the only indicator of the impact of your SQL statement, and may or may not be interesting depending on your situation. If you alter the insert statement to include additional `VALUES` pairs, it will increase accordingly. It can also be more than one for `SET` statements which affect multiple lines. It can even be zero in the cases where no rows match, indicating a logic or data error.

Query

In our final example, we're going to pull previously inserted data back out of the database. As in the previous insert example, we will see inferred type conversion.

As before, we start the `main` function by opening the database.

query/main.go: open,close.

```
    func main() {
        db, err := sql.Open("sqlite3", "file:testdb?mode=rw")
        ...
        defer db.Close()
```

Next we use the `Query` function to submit our SQL statement. `Query` behaves very similarly to `Exec`. It is variadic. The first argument is our SQL query statement, which may contain bind variables, `?`, in the `WHERE` clause. Any additional arguments are bound to their positionally respective bind variables. Yes, the `DATE(t) <= DATE(?)` is a bit superfluous but is included for demonstrative purposes.

query/main.go: query.

```
        rows, err := db.Query(
            SELECT i, s, t FROM sample WHERE DATE(t)
<= DATE(?), time.NOW(),
               )
```

If the query is successful, a result set is returned. Behind the scenes, the SQLite package creates a cursor which holds the location of the data—relative to both the query result processing and its location in the database file. To avoid consistency issues, this also locks the database until this query is complete. The indicator that the query is complete is with a `Close` on the result set. For this simple example, we can release the statement when we finish the function, so we use Go's `defer` mechanism.

query/main.go: stmtclose.

```
        defer rows.Close()
```

Now we can process the returned rows by iterating through the rows. To move through the cursor, we call the `Next` function. The `Next` function updates the underlying cursor information for the next unprocessed row. The `Query` does not do this initially, so a first call to `Next` is required to even begin to access data. This also allows us to wrap it all in a `for` loop.

query/main.go: next.

```
        for rows.Next() {
```

With the cursor properly in place for our next row, we can pull all of the values out of the row. We need a place to store the data local to our code, so we start by defining some variables. We then pass pointers for those variables into the `Scan` function, which will set them as appropriate. In addition to providing a place for the data, using pointers to our variables allows for `Scan` to cast the row values into the appropriate type. `Scan` is also variadic, and the position of arguments to it are the respective positions for the fields in the `SELECT` statement.

query/main.go: scan.

```
            var i int64
            var s string
            var t time.Time
            err := rows.Scan(&i, &s, &t)
```

Now we can print the results out.

query/main.go: printout.

```
            fmt.Printf("%s: %d %s\n", t, i, s)
```

An example output of this looks like:

```
    2019-06-15 13:21:06 +0000 UTC: 1 1
    2019-06-15 13:21:11 +0000 UTC: 1 1
    2019-06-16 18:03:38 +0000 UTC: 1 1
    2019-06-17 04:44:27 +0000 UTC: 1 1
```

Conclusion

In these examples, we've explored the `database/sql` package and an accompanying driver for it, the `github.com/mattn/go-sqlite3` for SQLite. In addition to what has been demonstrated here, the `database/sql` package and the various back ends provide other features—interrogating the columns and arbitrary results, handling timeouts with `Context`, direct creation of prepared SQL statements, and many more. You can dig into the Go SQL interface at http://go-database-sql.org.

Sometimes data gets complex enough that writing flat file parsers becomes tedious. Sometimes you have to interact with an existing application database. Go's SQL interface provides a simple way to interact with many different types of SQL databases. I hope this has given you a good basis for using SQL when needed. Good luck, and Happy Going.