Sample Program

The sample program connects to a public MongoDB database I have hosted with MongoLab. If you have Go and Bazaar installed on your machine, you can run the program

```
        Location  BuoyLocation   bson: location
    }
)

// main is the entry point for the application.
func main() {
    // We need this object to establish a session to our MongoDB.
    mongoDBDialInfo := &mgo.DialInfo{
        Addrs:    []string{MongoDBHosts},
        Timeout:  60 * time.Second,
        Database: AuthDatabase,
        Username: AuthUserName,
        Password: AuthPassword,
    }

    // Create a session which maintains a pool of socket connections
    // to our MongoDB.
    mongoSession, err := mgo.DialWithInfo(mongoDBDialInfo)
    if err != nil {
        log.Fatalf("CreateSession: %s\n", err)
    }

    // Reads may not be entirely up-to-date, but they will always see the
    // history of changes moving forward, the data read will be consistent
    // across sequential queries in the same session, and modifications made
    // within the session will be observed in following queries (read-your-writes).
    // http://godoc.org/labix.org/v2/mgo#Session.SetMode
    mongoSession.SetMode(mgo.Monotonic, true)

    // Create a wait group to manage the goroutines.
    var waitGroup sync.WaitGroup

    // Perform 10 concurrent queries against the database.
    waitGroup.Add(10)
    for query := 0; query < 10; query++ {
        go RunQuery(query, &waitGroup, mongoSession)
    }

    // Wait for all the queries to complete.
    waitGroup.Wait()
```

```
    log.Println("All Queries Completed")
}

// RunQuery is a function that is launched as a goroutine to perform
// the MongoDB work.
func RunQuery(query int, waitGroup *sync.WaitGroup, mongoSession *mgo.Session) {
    // Decrement the wait group count so the program knows this
    // has been completed once the goroutine exits.
    defer waitGroup.Done()

    // Request a socket connection from the session to process our query.
    // Close the session when the goroutine exits and put the connection back
    // into the pool.
    sessionCopy := mongoSession.Copy()
    defer sessionCopy.Close()

    // Get a collection to execute the query against.
    collection := sessionCopy.DB(TestDatabase).C("buoy_stations")

    log.Printf("RunQuery : %d : Executing\n", query)

    // Retrieve the list of stations.
    var buoyStations []BuoyStation
    err := collection.Find(nil).All(&buoyStations)
    if err != nil {
        log.Printf("RunQuery : ERROR : %s\n", err)
        return
    }

    log.Printf("RunQuery : %d : Count[%d]\n", query, len(buoyStations))
}
```

Now that you have seen the entire program, we can break it down. Let's start with the type structures that are defined in the beginning:

```
type (
    // BuoyCondition contains information for an individual station.
    BuoyCondition struct {
        WindSpeed     float64 `bson:"wind_speed_milehour"`
        WindDirection int     `bson:"wind_direction_degnorth"`
        WindGust      float64 `bson:"gust_wind_speed_milehour"`
    }

    // BuoyLocation contains the buoy's location.
    BuoyLocation struct {
        Type        string    `bson:"type"`
        Coordinates []float64 `bson:"coordinates"`
    }

    // BuoyStation contains information for an individual station.
    BuoyStation struct {
```

```
    BuoyStation struct {
        ID        bson.ObjectId `bson:"_id,omitempty"`
        StationId string        `bson:"station_id"`
        Name      string        `bson:"name"`
        LocDesc   string        `bson:"location_desc"`
        Condition BuoyCondition `bson:"condition"`
        Location  BuoyLocation  `bson:"location"`
    }
)
```

The structures represent the data that we are going to retrieve and unmarshal from our query. BuoyStation represents the main document and BuoyCondition and BuoyLocation are embedded documents. The mgo driver makes it easy to use native types that represent the documents stored in our collections by using tags. With the tags, we can control how the mgo driver unmarshals the returned documents into our native Go structures.

Now let's look at how we connect to a MongoDB database using mgo:

```
// We need this object to establish a session to our MongoDB.
mongoDBDialInfo := &mgo.DialInfo{
    Addrs:    []string{MongoDBHosts},
    Timeout:  60 * time.Second,
    Database: AuthDatabase,
    Username: AuthUserName,
    Password: AuthPassword,
}

// Create a session which maintains a pool of socket connections
// to our MongoDB.
mongoSession, err := mgo.DialWithInfo(mongoDBDialInfo)
if err != nil {
    log.Fatalf("CreateSession: %s\n", err)
}
```

We start with creating a mgo.DialInfo object. Connecting to a replica set can be accomplished by providing multiple addresses in the Addrs field or with a single address. If we are using a single host address to connect to a replica set, the mgo driver will learn about any remaining hosts from the replica set member we connect to. In our case we are connecting to a single host.

After providing the host, we specify the database, username and password we need for authentication. One thing to note is that the database we authenticate against may not necessarily be the database our application needs to access. Some applications authenticate against the admin database and then use other databases depending on their configuration. The mgo driver supports these types of configurations very well.

Next we use the mgo.DialWithInfo method to create a mgo.Session object. Each sess specifies a Strong or Monotonic mode, and other settings such as write concern and preference. The mgo.Session object maintains a pool of connections to MongoDB. We can create multiple sessions with different modes and settings to support different aspects of our applications.

```
// Reads may not be entirely up-to-date, but they will always see the
// history of changes moving forward, the data read will be consistent
// across sequential queries in the same session, and modifications made
// within the session will be observed in following queries (read-your-writes).
// http://godoc.org/labix.org/v2/mgo#Session.SetMode.
mongoSession.SetMode(mgo.Monotonic, true)
```

The next line of code sets the mode for the session. There are three modes that can be set, Strong, Monotonic and Eventual. Each mode sets a specific consistency for how reads and writes are performed. For more information on the differences between each mode, check out the documentation for the mgo driver.

We are using Monotonic mode which provides reads that may not entirely be up to date, but the reads will always see the history of changes moving forward. In this mode reads occur against secondary members of our replica sets until a write happens. Once a write happens, the primary member is used. The benefit is some distribution of the reading load can take place against the secondaries when possible.

With the session set and ready to go, next we execute multiple queries concurrently:

```
// Create a wait group to manage the goroutines.
var waitGroup sync.WaitGroup

// Perform 10 concurrent queries against the database.
waitGroup.Add(10)
for query := 0; query < 10; query++ {
    go RunQuery(query, &waitGroup, mongoSession)
}

// Wait for all the queries to complete.
waitGroup.Wait()
log.Println("All Queries Completed")
```

This code is classic Go concurrency in action. First we create a sync.WaitGroup object so we can keep track of all the goroutines we are going to launch as they complete their work. Then we immediately set the count of the sync.WaitGroup object to ten and use a for loop to launch ten goroutines using the RunQuery function. The keyword go is used to launch a

function or method to run concurrently. The final line of code calls the Wait method o sync.WaitGroup object which locks the main goroutine until everything is done proces

To learn more about Go concurrency and better understand how this particular piece of code works, check out these posts on concurrency and channels.

Now let's look at the RunQuery function and see how to properly use the mgo.Session object to acquire a connection and execute a query:

```
// Decrement the wait group count so the program knows this
// has been completed once the goroutine exits.
defer waitGroup.Done()

// Request a socket connection from the session to process our query.
// Close the session when the goroutine exits and put the connection back

// into the pool.
sessionCopy := mongoSession.Copy()
defer sessionCopy.Close()
```

The very first thing we do inside of the RunQuery function is to defer the execution of the Done method on the sync.WaitGroup object. The defer keyword will postpone the execution of the Done method, to take place once the RunQuery function returns. This will guarantee that the sync.WaitGroup objects count will decrement even if an unhandled exception occurs.

Next we make a copy of the session we created in the main goroutine. Each goroutine needs to create a copy of the session so they each obtain their own socket without serializing their calls with the other goroutines. Again, we use the defer keyword to postpone and guarantee the execution of the Close method on the session once the RunQuery function returns. Closing the session returns the socket back to the main pool, so this is very important.

```
// Get a collection to execute the query against.
collection := sessionCopy.DB(TestDatabase).C("buoy_stations")


log.Printf("RunQuery : %d : Executing\n", query)

// Retrieve the list of stations.
var buoyStations []BuoyStation
err := collection.Find(nil).All(&buoyStations)
if err != nil {
    log.Printf("RunQuery : ERROR : %s\n", err)
    return
}
```

```
log.Printf("RunQuery : %d : Count[%d]\n", query, len(buoyStations))
```

To execute a query we need a mgo.Collection object. We can get a mgo.Collection object through the mgo.Session object by specifying the name of the database and then the collection. Using the mgo.Collection object, we can perform a Find and retrieve all the documents from the collection. The All function will unmarshal the response into our slice of BuoyStation objects. A slice is a dynamic array in Go. Be aware that the All method will load all the data in memory at once. For large collections it is better to use the Iter method instead. Finally, we just log the number of BuoyStation objects that are returned.

Conclusion

The example shows how to use Go concurrency to launch multiple goroutines that can execute queries against a MongoDB database independently. Once a session is established, the mgo driver exposes all of the MongoDB functionality and handles the unmarshaling of BSON documents into Go native types.

MongoDB can handle a large number of concurrent requests when you architect your MongoDB databases and collections with concurrency in mind. Go and the mgo driver are perfectly aligned to push MongoDB to its limits and build software that can take advantage of all the computing power that is available.

The mgo driver provides a safe way to leverage Go's concurrency support and you have the flexibility to execute queries concurrently and in parallel. It is best to take the time to learn a bit about MongoDB replica sets and load balancer configuration. Then make sure the load balancer is behaving as expected under the different types of load your application can produce.

Now is a great time to see what MongoDB and Go can do for your software applications, web services and service platforms. Both technologies are being battle tested everyday by all types of companies, solving all types of business and computing problems.

Inanc Gumus • a year ago

Actually, sending too many concurrent requests to *Mongo* fails if you didn't set socket pool limit. See this example and if you have any suggestions about it, please write a comment: https://groups.google.com/f...

12 • Reply • Share ›

Youtube

Twitter

LinkedIn

Slack

StackOverflow

Get MongoDB Email Updates

Email Address →