Build a Geo locating for zip code

Let's say we want to convert zip code to lat/long. We probably can use Google, but Google limit 3000 API call for free. Same rule applied to other services. Even a commercial solution can cost ~500/year http://greatdata.com/zip-codes-lat-long

Nathan released a great library for this purpose. However, it's written in Python. With the mess of Python2 and Python3, it can be a litte bit tricky for non Python dev to deploy and understanding it.

A better way to solve this is to use Go Lang with an embedded database. Therefore, we can easily deploy the application with a single binary file. LevelDB is an embedded database, high-performance, and include a very good compression library using Snappy.

Design our application

Our application will include 2 components:

- Importer: it imports data from external source into our LevelDB database.
- API: it returns to HTTP request in a RESTful way. Such as: yourdomain.com/api/95136 to return Geo coding for 95136 zip code

Ideally, we will build a command line utility to do two things by passing parameter into it. To parse command line parameter, we will use Codegangsta's cli package.

Implement main package

Let's create a package in your GOPATH. We will call it go-ziplocate, and we will put it inside `kureikain` namespace. I'm doing this because my code will be published in Github later and you can easily follow it.

```
$ cd $GOPATH/src
$ mkdir -p github.com/kureikain/go-ziplocate
```

```
$ cd github.com/kureikain/go-ziplocate
```

Now, let's create a file call `gozip.go` with this content

```
package main
import (
  "os"
  //"runtime"
  "github.com/codegangsta/cli"
  "github.com/kureikain/go-ziplocate/cmd"
)
const APP_VER = "0.0.1"
func main() {
  app := cli.NewApp()
  app.Name = "gozip"
  app.Usage = "Geocoder for code ZIP code"
  app.Version = APP_VER

  //Register command
  app.Commands = []cli.Command{
    cmd.CmdWeb,
    cmd.CmdImport,
  }
  app.Flags = append(app.Flags, []cli.Flag{}...)

  app.Run(os.Args)
}
```

Ideally, we will follow cli package. We create an `App` struct of cli package and assign `Name`, `Usage`, `Version` respectively.

Notice the `Commands` field. We are importing another sub package call `cmd` inside our `go-ziplocate` folder. We will go into its detail later. The idea here is: instead of defining the command handlers direcly into `main` package and in main file, we are moving them into separae package for easily testing, better code organization, and keep our main file small and lean enough.

What is in the `Commands` field? Imagine you are using `go` command line utility, you can run a couple of command

```
$ go build
$ go run
$ go test
```

We want to do the same with our package, assume its name is `gozip`, we want to have:

```
$ gozip import
$ gozip web
```

to run importer and web server respectively. cli package allows us to do that with sub command. The syntax looks like this:

```
app := cli.NewApp()
app.Name = "greet"
app.Usage = "fight the loneliness!"
app.Commands = []cli.Command{
  {
    Name:      "add",
    ShortName: "a",
    Usage:     "add a task to the list",
    Action: func(c *cli.Context) {
      println("added task: ", c.Args().First())
    },
  },
}
```

The `Commands` field is a slice of `cli.Command` struct. `Command` struct has field `Action` which is a function to be run when the command is invoked. The only different here in our main file is, we are moving the definition into separate packages.

Part 1: Building importer

Now, we got our main file which acts like a router. It know what to invoked when a paramter is passed into it.

Let's create our `import.go` file in `cmd` directory.

```
$ mkdir cmd
$ touch cmd/import.go
$ vim cmd/import.go (or use any of your favourite editor)
```

then paste this content or

```
package cmd

import (
  "github.com/codegangsta/cli"
)

var CmdImport = cli.Command{
  Name: "import",
  Usage: "import -file /absolute/path/to/shapefile.shp",
  Description: `Import shapefile into GoZip database which is ba
  Action: runImport,
  Flags: []cli.Flag{
    cli.StringFlag{
      Name: "file",
      Value: "",
      Usage: "shp file path to import",
    },
  },
}

func runImport(c *cli.Context) {
}
```

Still nothing fancy here. As we promised in previous main file, we will define the `Command` struct here. Notice the `Action` field is assigned to `runImport` function. This function will be executed when importer are invoked.

Now, we will have to import data of zip code, geo location into LevelDB from a data source. Luckily, that data is publicly available at https://www.census.gov/geo/maps-data/data/tiger-line.html

The data is stored in a format call shapefile. We can work with it using a library

call go-shp. This library enables us to iterator *shapefile*, read all its element, or even writing to a *shapefile*. At this moment, we just read it.

Let's change our `runImport` to reflect our changes:

```
func runImport(c *cli.Context) {
  var file = c.String("file")
  log.Printf("Importing local repositories...%s", file)

  //Open LevelDB database
  db, err := leveldb.OpenFile("zipdata", nil)
  if err != nil {
    log.Fatal("Cannot create database")
  }

  defer db.Close()

  openWithHandler(file, func (zip string, geocoder []byte) bool
    db.Put([]byte(zip), geocoder, nil)
    return true
  })
  log.Println("Finish importing!")
}
```

and update `import` to include LevelDB library:

```
import (
  "github.com/codegangsta/cli"
  "github.com/syndtr/goleveldb/leveldb"
)
```

We will parse `-file` parameter on command line to get the path to shapefile.

Second. we will open a database call *zipdata* with leveldb package using function `OpenFile`. If the database is existed, an connection to it is established. And we can read/write into the database. Otherwise, an empty database is created, and ready for data reading/writing. Since LevelDb is a key-value database, we will use zip code as the key, and to keep thing simple, its

value is the combination of latitude,longitude in a simple format separate by ":". For example, the value of "08861" key will be "40.519553:–74.273578". Data of LevelDB is a byte array. We can easily convert string into an byte array in Go and store it.

We also use `defer` for `db.Close`. What is `defer`? `defer` is like a destructor in PHP, Ruby,…It will push the function call into a special list. This list will be run after the surrounding function is returned. By doing that, we will make sure `db.Close()` will be called after `runImport` is finished and return. `defer` is like a nice way to clean up the data: close database connection, remove temporary file, close file handler, flush log,…

Then we call function `openWithHandler`, passing a file, and another anonymous function to it. The anonymous function receive 2 parameter:

- string as zip code
- geocoder byte array of geo code

Inside this function, we write into LevelDB by using its `Put` function of package.

The idea behinds `openWithHandler` is we open the file, and iterator over it, then we pass the data at each item into the function handle. This way, we can easily pass any function handler to do different thing. Eg: in this implementation we can grad the data, and use handler function to write data into a LevelDB storage, but we can later on, change it so we only change handler function, leaving the iterator code on *shapefile* unchange. Writing in Go, we have:

```
import (
  "github.com/codegangsta/cli"
  "github.com/jonas-p/go-shp"
  "github.com/syndtr/goleveldb/leveldb"
)

func openWithHandler(file string, handler func(zip string, geocc
  shape, err := shp.Open(file)
  if err != nil { log.Fatal(err) }
  defer shape.Close()
```

```
  // loop through all features in the shapefile
  var centroid zip.Point
  var boundary shp.Box
  for shape.Next() {
      n, p := shape.Shape()
      boundary = p.BBox()
      centroid.X = (boundary.MinX + boundary.MaxX) / 2
      centroid.Y = (boundary.MinY + boundary.MaxY) / 2
      //This is a naive way to convert struct to string to byte.
      handler(shape.ReadAttribute(n, 0), []byte(fmt.Sprintf("%f:
  }
}
```

Notice We updated `import` to include new package `go-shp`. You can read more about this package by following its document. `go-shp` exposes its function via `shp` package name. For now, we know that we use these main function of `shp` package to interact with the shape file:

- Open(file): to open shapefile for reading data. Open return a `Reader` struct of `shp` package.
- Close(): close the file after finishing
- Reader.Next(): iterator over element of shapefile
- Reader.Shape() function: returns the most recent feature that was read by a call to Next. It returns 2 value: the index of current element in the shape file, and the `Shape` object. The latitude, longitude are holded in this object.
- Reader.ReadAttribute(n, f): will read the value of field f-th of element n-th in shape file. Both are zero-based index. Those value can be consider meta data of the `Shape` object. Those meta data can include: zip code, city,… In our shape file which we download before, the first field is the ZIP code.

`Shape` object is a `interface`. Mostly, we will used its `BBox` function which return a `Box` struct includes:

- MinX
- MinY

- MaxX
- MaxY

All are `float64` to denote the value of geo code of boundary of the region. To make thing simple, we can imagine the region of a zip code is a square shape, with (MinX, MinY) and (MaxX, MaxY) is the longitude and latitude of boundary respectively.

Our `openWithHandler` just opens the shape file. Then we iterator over it, grab the GEO data with `Shape` and `BBox` method. Then we calculated central point in a naive way, use Euclid formula:

```
for shape.Next() {
      n, p := shape.Shape()
      boundary = p.BBox()
      centroid.X = (boundary.MinX + boundary.MaxX) / 2
      centroid.Y = (boundary.MinY + boundary.MaxY) / 2
}
```

Then we read zip code at first element

```
handler(shape.ReadAttribute(n, 0), []byte(fmt.Sprintf("%f:%f", c
```

We store geo data into a string with latitude and longitude separate by *:*.

Part 2: Building API

We will use build in `http` package with its `Mux` to handle API request.

Part 3: Writing Test

Code without test is legacy. Let do some simple test.