Data Transport through gob

This lesson discusses specifically the binary type of data and how Go uses the binary type to transfer data using the gob package.

What is gob?

The *gob* is Go's format for *serializing* and *deserializing* program data in *binary* format. It is found in the `encoding` package. Data in this format is called a *gob* (short for Go binary format). It is similar to **Python's pickle** or **Java's Serialization**. It is typically used in transporting arguments and results of *remote procedure calls* (RPCs) (see the `rpc` package Chapter 13). More generally, it used for data transport between applications and machines. In what way is it different from JSON or XML? The gob was tailored explicitly for working in an environment that is entirely in Go, for example, communicating between two servers written in Go.

This package works with the language in a way that an externally-defined, language-independent encoding cannot. That's why the format is binary in the first place, not a text-format like JSON or XML. Gobs are not meant to be used in other languages than Go because, in the encoding and decoding process, Go's reflection capability is used.

Explanation

Gob files or streams are entirely self-describing. For every type, they contain a description of that type, and they can always be decoded in Go without any knowledge of the file's contents. Only exported fields are encoded; zero values are not taken into account. When decoding structs, fields are matched by name and compatible type, and only fields that exist in both are affected. In this way, a gob decoder client will still function when in the source datatype fields have been added. The client will continue to recognize the previously existing fields. Also, there is excellent flexibility provided, e.g., integers are encoded as unsized, and variable-length, regardless of the concrete Go type at the sender side.

So if we have at the sender side a struct `T`:

```
type T struct { X, Y, Z int }
var t = T{X: 7, Y: 0, Z: 8}
```

This can be captured at the receiver side in a variable `u` of type struct `U`:

```
type U struct { X, Y *int8 }
var u U
```

At the receiver, `X` gets the value 7, and `Y` the value 0 (which was not transmitted). In the same way as `json`, `gob` works by creating an encoder object with a `NewEncoder()` function and calling `Encode()`, again completely generalized by using `io.Writer`. The inverse is done with a decoder object with a `NewDecoder()` function and calling `Decode()`, generalized by using `io.Reader`.

In the following program, you'll find a simple example of decoding and encoding, simulating transmission over a network with a buffer of bytes:

```
package main
import (
"bytes"
"fmt"
"encoding/gob"
"log"
)

type P struct {
  X, Y, Z int
  Name string
}

type Q struct {
  X, Y *int32
  Name string
}

func main() {
  // Initialize the encoder and decoder. Normally enc and dec would be
  // bound to network connections and the encoder and decoder would
  // run in different processes.
  var network bytes.Buffer // Stand-in for a network connection
  enc := gob.NewEncoder(&network) // Will write to network.
  dec := gob.NewDecoder(&network)// Will read from network.
  // Encode (send) the value.
  err := enc.Encode(P{3, 4, 5, "Pythagoras"})
  if err != nil {
    log.Fatal("encode error:", err)
  }
  // Decode (receive) the value.
  var q Q
  err = dec.Decode(&q)
  if err != nil {
    log.Fatal("decode error:", err)
  }
  fmt.Printf("%q: {%d,%d}\n", q.Name, *q.X, *q.Y)
}
```

Encoding and Decoding with gob

In the code above, a network connection is simulated by a buffer called `network` at **line 23**. At **line 24**, we initialize the encoder `enc` on the buffer; this will write to the network. At **line 25**, we initialize the decoder `dec` on the buffer; this will read from the network.

On our side, we have the data of the struct type `P`, defined at **line 9**. On the other side, we will receive the data in the struct type `Q`, defined at **line 14**.

At **line 27**, we encode a `P` struct. At **line 32** a variable `q` of struct type `Q` is declared, and the data transmitted is decoded in `q`. Apart from the normal error-handling, we print out this `q` value: **"Pythagoras": {3,4}**.

Note that `Z` value **5** was not received because it wasn't defined in `Q`.

The following is an example of a mixture of readable and binary data, as you will see when you try to read it in a text editor:

```go
package main
import (
"encoding/gob"
"log"
"os"
)

type Address struct {
  Type string
  City string
  Country string
}

type VCard struct {
  FirstName string
  LastName string
  Addresses []*Address
  Remark string
}

func main() {
  pa := &Address{"private", "Aartselaar","Belgium"}
  wa := &Address{"work", "Boom", "Belgium"}
  vc := VCard{"Jan", "Kersschot", []*Address{pa,wa}, "none"}

  // fmt.Printf("%v: \n", vc) // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
  // using an encoder:
  file, _ := os.OpenFile("output/vcard.gob", os.O_CREATE|os.O_WRONLY, 0)
  defer file.Close()
  enc := gob.NewEncoder(file)
  err := enc.Encode(vc)
  if err != nil {
    log.Println("Error in encoding gob")
  }
}
```

Reading and Encoding Binary Data

In the code above, we define our previous struct types `Address` (from **line 8** to **line 12**) and `Vcard` (from **line 14** to **line 19**).

Then, in `main()` (from **line 22** to **line 24**), we make instances of these structs. We will write the `vc` value to the file **output/vcard.gob**. This file is opened for writing at **line 28** because the *second* argument is `_` as error-handling is left out. We make sure the file gets closed at **line 29**. Then, at **line 30**, we define an encoder `enc` on this file. Next, we use `enc` at **line 31** to encode the `vc` data in *gob* format and write it to the file. Lastly, from **line 32** to **line 34**, we check for a possible error.

Now that you're familiar with the `gob` package and how it deals with binary data, the next lesson brings you a challenge to solve.