Data serialisation

Communication between a client and a service requires the exchange of data. This data may be highly structured, but has to be serialised for transport. This chapter looks at the basics of serialisation and then considers several techniques supported by Go APIs.

Introduction

A client and server need to exchange information via messages. TCP and UDP provide the transport mechanisms to do this. The two processes also have to have a protocol in place so that message exchange can take place meaningfully.

Messages are sent across the network as a sequence of bytes, which has no structure except for a linear stream of bytes. We shall address the various possibilities for messages and the protocols that define them in the next chapter. In this chapter we concentrate on a component of messages - the data that is transferred.

A program will typically build complex data structures to hold the current program state. In conversing with a remote client or service, the program will be attempting to transfer such data structures across the network - that is, outside of the application's own address space.

Programming languages use structured data such as

- records/structures
- variant records
- array - fixed size or varying
- string - fixed size or varying
- tables - e.g. arrays of records
- non-linear structures such as
  - circular linked list
  - binary tree
  - objects with references to other objects

None of IP, TCP or UDP packets know the meaning of any of these data types. All that they can contain is a sequence of bytes. Thus an application has to *serialise* any data into a stream of bytes in order to write it, anddeserialise the stream of bytes back into suitable data structures on reading it. These two operations are known as *marshalling* and *unmarshalling* respectively.

For example, consider sending the following variable length table of two columns of variable length strings:

fred	programmer
liping	analyst
sureerat	manager

This could be done by in various ways. For example, suppose that it is known that the data will be an unknown number of rows in a two-column table. Then a marshalled form could be

```
    3                   // 3 rows, 2 columns assumed
    4 fred              // 4 char string,col 1
    10 programmer       // 10 char string,col 2
    6 liping            // 6 char string, col 1
    7 analyst           // 7 char string, col 2
    8 sureerat          // 8 char string, col 1
    7 manager           // 7 char string, col 2
```

Variable length things can alternatively have their length indicated by terminating them with an "illegal" value, such as '\0' for strings:

```
    3
    fred\0
    programmer\0
    liping\0
    analyst\0
    sureerat\0
    manager\0
```

Alternatively, it may be known that the data is a 3-row fixed table of two columns of strings of length 8 and 10 respectively. Then a serialisation could be

```
    fred\0\0\0\0
    programmer
    liping\0\0
    analyst\0\0\0
    sureerat
    manager\0\0\0
```

Any of these formats is okay - but the message exchange protocol must specify which one is used, or allow it to be determined at runtime.

Mutual agreement

The previous section gave an overview of the issue of data serialisation. In practise, the details can be considerably more complex. For example, consider the first possibility, marshalling a table into the stream

```
    3
    4 fred
    10 programmer
    6 liping
    7 analyst
    8 sureerat
    7 manager
```

Many questions arise. For example, how many rows are possible for the table - that is, how big an integer do we need to describe the row size? If it is 255 or less, then a single byte will do, but if it is more, then a short, integer or long may be needed. A similar problem occurs for the length of each string. With the characters themselves, to which character set do they belong? 7 bit ASCII? 16 bit Unicode? The question of character sets is discussed at length in a later chapter.

The above serialisation is *opaque* or *implicit*. If data is marshalled using the above format, then there is nothing in the serialised data to say how it should be unmarshalled. The unmarshalling side has to know exactly how the data is serialised in order to unmarshal it correctly. For example, if the number of rows is marshalled as an eight-bit integer, but unmarshalled as a sixteen-bit integer, then an incorrect result will occur as the receiver tries to unmarshall 3 and 4 as a sixteen-bit integer, and the receiving program will almost certainly fail later.

An early well-known serialisation method is XDR (external data representation) used by Sun's RPC, later known as ONC (Open Network Computing). XDR is defined by RFC 1832 and it is instructive to see how precise this specification is. Even so, XDR is inherently type-unsafe as serialised data contains no type information. The correctness of its use in ONC is ensured primarily by compilers generating code for both marshalling and unmarshalling.

Go contains no explicit support for marshalling or unmarshalling opaque serialised data. The RPC package in Go does not use XDR, but instead uses "gob" serialisation, described later in this chapter.

Self-describing data

Self-describing data carries type information along with the data. For example, the previous data might get encoded as

```
table
   uint8 3
   uint 2
string
   uint8 4
   []byte fred
string
```

```
   uint8 10
   []byte programmer
string
   uint8 6
   []byte liping
string
   uint8 7
   []byte analyst
string
   uint8 8
   []byte sureerat
string
   uint8 7
   []byte manager
```

Of course, a real encoding would not normally be as cumbersome and verbose as in the example: small integers would be used as type markers and the whole data would be packed in as small a byte array as possible. (XML provides a counter-example, though.). However, the principle is that the marshaller will generate such type information in the serialised data. The unmarshaller will know the type-generation rules and will be able to use this to reconstruct the correct data structure.

ASN.1

Abstract Syntax Notation One (ASN.1) was originally designed in 1984 for the telecommunications industry. ASN.1 is a complex standard, and a subset of it is supported by Go in the package "asn1". It builds self-describing serialised data from complex data structures. Its primary use in current networking systems is as the encoding for X.509 certificates which are heavily used in authentication systems. The support in Go is based on what is needed to read and write X.509 certificates.

Two functions allow us to marshal and unmarshal data

```
func Marshal(val interface{}) ([]byte, os.Error)
func Unmarshal(val interface{}, b []byte) (rest []byte, err os.Error)
```

The first marshals a data value into a serialised byte array, and the second unmarshals it. However, the first argument of type `interface` deserves further examination. Given a variable of a type, we can marshal it by just passing its value. To unmarshal it, we need a variable of a named type that will match the serialised data. The precise details of this are discussed later. But we also need to make sure that the variable is allocated to memory for that type, so that there is actually existing memory for the unmarshalling to write values into.

We illustrate with an almost trivial example, of marshalling and unmarshalling an integer. We can pass an integer value to `Marshal` to return a byte array, and unmarshal the array into an integer variable as in this program:

```
/* ASN.1
 */

package main

import (
	"encoding/asn1"
	"fmt"
	"os"
)

func main() {
	mdata, err := asn1.Marshal(13)
	checkError(err)

	var n int
	_, err1 := asn1.Unmarshal(mdata, &n)
	checkError(err1)

	fmt.Println("After marshal/unmarshal: ", n)
}

func checkError(err error) {
	if err != nil {
		fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
		os.Exit(1)
	}
}
```

The unmarshalled value, is of course, 13.

Once we move beyond this, things get harder. In order to manage more complex data types, we have to look more closely at the data structures supported by ASN.1, and how ASN.1 support is done in Go.

Any serialisation method will be able to handle certain data types and not handle some others. So in order to determine the suitability of any serialisation such as ASN.1, you have to look at the possible data types supported versus those you wish to use in your application. The following ASN.1 types are taken from http://www.obj-sys.com/asn1tutorial/node4.html

The simple types are

- BOOLEAN: two-state variable values
- INTEGER: Model integer variable values
- BIT STRING: Model binary data of arbitrary length

- OCTET STRING: Model binary data whose length is a multiple of eight
- NULL: Indicate effective absence of a sequence element
- OBJECT IDENTIFIER: Name information objects
- REAL: Model real variable values
- ENUMERATED: Model values of variables with at least three states
- CHARACTER STRING: Models values that are strings of characters fro

Character strings can be from certain character sets

- NumericString: 0,1,2,3,4,5,6,7,8,9, and space
- PrintableString: Upper and lower case letters, digits, space, apostrophe, left/right parenthesis, plus sign, comma, hyphen, full stop, solidus, colon, equal sign, question mark
- TeletexString (T61String): The Teletex character set in CCITT's T61, space, and delete
- VideotexString: The Videotex character set in CCITT's T.100 and T.101, space, and delete
- VisibleString (ISO646String): Printing character sets of international ASCII, and space
- IA5String: International Alphabet 5 (International ASCII)
- GraphicString 25 All registered G sets, and space GraphicString

And finally, there are the structured types:

- SEQUENCE: Models an ordered collection of variables of different type
- SEQUENCE OF: Models an ordered collection of variables of the same type
- SET: Model an unordered collection of variables of different types
- SET OF: Model an unordered collection of variables of the same type
- CHOICE: Specify a collection of distinct types from which to choose one type
- SELECTION: Select a component type from a specified CHOICE type
- ANY: Enable an application to specify the type Note: ANY is a deprecated ASN.1 Structured Type. It has been replaced with X.680 Open Type.

Not all of these are supported by Go. Not all possible values are supported by Go. The rules as given in the Go "asn1" package documentation are

- An ASN.1 INTEGER can be written to an int or int64. If the encoded value does not fit in the Go type, Unmarshal returns a parse error.
- An ASN.1 BIT STRING can be written to a BitString.
- An ASN.1 OCTET STRING can be written to a []byte.
- An ASN.1 OBJECT IDENTIFIER can be written to an ObjectIdentifier.
- An ASN.1 ENUMERATED can be written to an Enumerated.

- An ASN.1 UTCTIME or GENERALIZEDTIME can be written to a *time.Time.
- An ASN.1 PrintableString or IA5String can be written to a string.
- Any of the above ASN.1 values can be written to an interface{}. The value stored in the interface has the corresponding Go type. For integers, that type is int64.
- An ASN.1 SEQUENCE OF x or SET OF x can be written to a slice if an x can be written to the slice's element type.
- An ASN.1 SEQUENCE or SET can be written to a struct if each of the elements in the sequence can be written to the corresponding element in the struct.

Go places real restrictions on ASN.1. For example, ASN.1 allows integers of any size, while the Go implementation will only allow upto signed 64-bit integers. On the other hand, Go distinguishes between signed and unsigned types, while ASN.1 doesn't. So for example, transmitting a value of `uint64` may fail if it is too large for `int64`,

In a similar vein, ASN.1 allows several different character sets. Go only supports PrintableString and IA5String (ASCII). ASN.1 does not support Unicode characters (which require the BMPString ASN.1 extension). The basic Unicode character set of Go is not supported, and if an application requires transport of Unicode characters, then an encoding such as UTF-7 will be needed. Such encodings are discussed in a later chapter on character sets.

We have seen that a value such as an integer can be easily marshalled and unmarshalled. Other basic types such as booleans and reals can be similarly dealt with. Strings which are composed entirely of ASCII characters can be marshalled and unmarshalled. However, if the string is, for example, "hello \u00bc" which contains the non-ASCII character '¼' then an error will occur: "ASN.1 structure error: PrintableString contains invalid character". This code works, as long as the string is only composed of printable characters:

```
        s := "hello"
        mdata, _ := asn1.Marshal(s)

        var newstr string
        asn1.Unmarshal(mdata, &newstr)
```

ASN.1 also includes some "useful types" not in the above list, such as UTC time. Go supports this UTC time type. This means that you can pass time values in a way that is not possible for other data values. ASN.1 does not support pointers, but Go has special code to manage pointers to time values. The function GetLocalTime returns *time.Time. The special code marshals this, and it can be unmarshalled into a pointer variable to atime.Time object. Thus this code works

```
        t := time.LocalTime()
        mdata, err := asn1.Marshal(t)

        var newtime = new(time.Time)
        _, err1 := asn1.Unmarshal(&newtime, mdata)
```

Both LocalTime and new handle pointers to a *time.Time, and Go looks after this special case.

In general, you will probably want to marshal and unmarshal structures. Apart from the special case of time, Go will happily deal with structures, but not with pointers to structures. Operations such as new create pointers, so you have to dereference them before marshalling/unmarshalling them. Go normally dereferences pointers for you when needed, but not in this case. These both work for a type T:

```
// using variables
var t1 T
t1 = ...
mdata1, _ := asn1.Marshal(t)

var newT1 T
asn1.Unmarshal(&newT1, mdata1)

/// using pointers
var t2 = new(T)
*t2 = ...
mdata2, _ := asn1.Marshal(*t2)

var newT2 = new(T)
asn1.Unmarshal(newT2, mdata2)
```

Any suitable mix of pointers and variables will work as well.

The fields of a structure must all be exportable, that is, field names must begin with an uppercase letter. Go uses the reflect package to marshal/unmarshal structures, so it must be able to examine all fields. This type cannot be marshalled:

```
      type T struct {
          Field1 int
          field2 int // not exportable
      }
```

ASN.1 only deals with the data types. It does not consider the names of structure fields. So the following type T1 can be marshalled/unmarshalled into type T2 as the corresponding fields are the same types:

```
type T1 struct {
    F1 int
    F2 string
}

type T2 struct {
    FF1 int
    FF2 string
}
```

Not only the types of each field must match, but the number must match as well. These two types don't work:

```
type T1 struct {
    F1 int
}

type T2 struct {
    F1 int
    F2 string // too many fields
}
```

ASN.1 daytime client and server

Now (finally) let us turn to using ASN.1 to transport data across the network.

We can write a TCP server that delivers the current time as an ASN.1 Time type, using the techniques of the last chapter. A server is

```
/* ASN1 DaytimeServer
 */
package main

import (
        "encoding/asn1"
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        service := ":1200"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)
        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                daytime := time.Now()
                // Ignore return network errors.
                mdata, _ := asn1.Marshal(daytime)
                conn.Write(mdata)
                conn.Close() // we're finished
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

which can be compiled to an executable such as `ASN1DaytimeServer` and run with no arguments. It will wait for connections and then send the time as an ASN.1 string to the client.

A client is

```

/* ASN.1 DaytimeClient
 */
package main

import (
        "bytes"
        "encoding/asn1"
        "fmt"
        "io"
        "net"
        "os"
        "time"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        result, err := readFully(conn)
        checkError(err)

        var newtime time.Time
        _, err1 := asn1.Unmarshal(result, &newtime)
        checkError(err1)

        fmt.Println("After marshal/unmarshal: ", newtime.String())

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

This connects to the service given in a form such as `localhost:1200`, reads the TCP packet and decodes the ASN.1 content back into a string, which it prints.

We should note that neither of these two - the client or the server - are compatable with the text-based clients and servers of the last chapter. This client and server are exchanging ASN.1 encoded data values, not textual strings.

JSON

JSON stands for JavaScript Object Notation. It was designed to be a lighweight means of passing data between JavaScript systems. It uses a text-based format and is sufficiently general that it has become used as a general purpose serialisation method for many programming languages.

JSON serialises objects, arrays and basic values. The basic values include string, number, boolean values and the null value. Arrays are a comma-separated list of values that can represent arrays, vectors, lists or sequences of various programming languages. They are delimited by square brackets "[...]". Objects are represented by a list of "field: value" pairs enclosed in curly braces "{ ... }".

For example, the table of employees given earlier could be written as an array of employee objects:

```
[
   {Name: fred, Occupation: programmer},
   {Name: liping, Occupation: analyst},
   {Name: sureerat, Occupation: manager}
]
```

There is no special support for complex data types such as dates, no distinction between number types, no recursive types, etc. JSON is a very simple language, but nevertheless can be quite useful. Its text-based format makes it easy for people to use, even though it has the overheads of string handling.

From the Go JSON package specification, marshalling uses the following type-dependent default encodings:

- Boolean values encode as JSON booleans.
- Floating point and integer values encode as JSON numbers.
- String values encode as JSON strings, with each invalid UTF-8 sequence replaced by the encoding of the Unicode replacement character U+FFFD.
- Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string.
- Struct values encode as JSON objects. Each struct field becomes a member of the object. By default the object's key name is the struct field name converted to lower case. If the struct field has a tag, that tag will be used as the name instead.
- Map values encode as JSON objects. The map's key type must be string; the object keys are used directly as map keys.
- Pointer values encode as the value pointed to. (Note: this allows trees, but not graphs!). A nil pointer encodes as the null JSON object.
- Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON object.
- Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value causes Marshal to return an InvalidTypeError.
- JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an infinite recursion.

A program to store JSON serialised data into a file is

```
/* SaveJSON
 */

package main

import (
	"encoding/json"
	"fmt"
	"os"
)

type Person struct {
	Name  Name
	Email []Email
}

type Name struct {
	Family   string
	Personal string
}

type Email struct {
	Kind    string
	Address string
}

func main() {
	person := Person{
		Name: Name{Family: "Newmarch", Personal: "Jan"},
		Email:        []Email{Email{Kind:        "home",        Address:
"jan@newmarch.name"},
			Email{Kind:              "work",              Address:
"j.newmarch@boxhill.edu.au"}}}

	saveJSON("person.json", person)
}

func saveJSON(fileName string, key interface{}) {
	outFile, err := os.Create(fileName)
	checkError(err)
	encoder := json.NewEncoder(outFile)
	err = encoder.Encode(key)
	checkError(err)
	outFile.Close()
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

and to load it back into memory is

```go
/* LoadJSON
 */

package main

import (
        "encoding/json"
        "fmt"
        "os"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}
func main() {
        var person Person
        loadJSON("person.json", &person)

        fmt.Println("Person", person.String())
}

func loadJSON(fileName string, key interface{}) {
        inFile, err := os.Open(fileName)
        checkError(err)
        decoder := json.NewDecoder(inFile)
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The serialised form is (formatted nicely)

```
{"Name":{"Family":"Newmarch",
         "Personal":"Jan"},
 "Email":[{"Kind":"home","Address":"jan@newmarch.name"},
          {"Kind":"work","Address":"j.newmarch@boxhill.edu.au"}
         ]
}
```

A client and server

A client to send a person's data and read it back ten times is

```
/* JSON EchoClient
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/json"
        "bytes"
        "io"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {
        person := Person{
                Name: Name{Family: "Newmarch", Personal: "Jan"},
                Email:        []Email{Email{Kind:        "home",        Address:
"jan@newmarch.name"},
                        Email{Kind:                 "work",              Address:
"j.newmarch@boxhill.edu.au"}}}

        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        encoder := json.NewEncoder(conn)
        decoder := json.NewDecoder(conn)

        for n := 0; n < 10; n++ {
                encoder.Encode(person)
                var newPerson Person
                decoder.Decode(&newPerson)
                fmt.Println(newPerson.String())
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

and the corrsponding server is

```
/* JSON EchoServer
 */
package main
```

```
import (
        "fmt"
        "net"
        "os"
        "encoding/json"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {

        service := "0.0.0.0:1200"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                encoder := json.NewEncoder(conn)
                decoder := json.NewDecoder(conn)

                for n := 0; n < 10; n++ {
                        var person Person
                        decoder.Decode(&person)
                        fmt.Println(person.String())
                        encoder.Encode(person)
                }
                conn.Close() // we're finished
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The gob package

Gob is a serialisation technique specific to Go. It is designed to encode Go data types specifically and does not at present have support for or by any other languages. It supports all Go data types except for channels, functions and interfaces. It supports integers of all types and sizes, strings and booleans, structs, arrays and slices. At present it has some problems with circular structures such as rings, but that will improve over time.

Gob encodes type information into its serialised forms. This is far more extensive than the type information in say an X.509 serialisation, but far more efficient than the type information contained in an XML document. Type information is only included once for each piece of data, but includes, for example, the names of struct fields.

This inclusion of type information makes Gob marshalling and unmarshalling fairly robust to changes or differences between the marshaller and unmarshaller. For example, a struct

```
 struct T {
     a int
     b int
}
```

can be marshalled and then unmarshalled into a different struct

```
 struct T {
     b int
     a int
}
```

where the order of fields has changed. It can also cope with missing fields (the values are ignored) or extra fields (the fields are left unchanged). It can cope with pointer types, so that the above struct could be unmarshalled into

```
 struct T {
     *a int
     **b int
}
```

To some extent it can cope with type coercions so that an `int` field can be broadened into an `int64`, but not with incompatable types such as `int` and `uint`.

To use Gob to marshall a data value, you first need to create an `Encoder`. This takes a `Writer` as parameter and marshalling will be done to this write stream. The encoder has a method `Encode` which marshalls the value to the stream. This method can be called multiple times on multiple pieces of data. Type information for each data type is only written once, though.

You use a `Decoder` to unmarshall the serialised data stream. This takes a `Reader` and each read returns an unmarshalled data value.

A program to store gob serialised data into a file is

```
/* SaveGob
 */

package main

import (
	"fmt"
	"os"
	"encoding/gob"
)

type Person struct {
	Name  Name
	Email []Email
}

type Name struct {
	Family   string
	Personal string
}

type Email struct {
	Kind    string
	Address string
}

func main() {
	person := Person{
		Name: Name{Family: "Newmarch", Personal: "Jan"},
		Email:        []Email{Email{Kind:        "home",        Address:
"jan@newmarch.name"},
			Email{Kind:              "work",              Address:
"j.newmarch@boxhill.edu.au"}}}

	saveGob("person.gob", person)
}

func saveGob(fileName string, key interface{}) {
	outFile, err := os.Create(fileName)
	checkError(err)
	encoder := gob.NewEncoder(outFile)
```

```
        err = encoder.Encode(key)
        checkError(err)
        outFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

and to load it back into memory is

```

/* LoadGob
 */

package main

import (
        "fmt"
        "os"
        "encoding/gob"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}
func main() {
        var person Person
        loadGob("person.gob", &person)

        fmt.Println("Person", person.String())
}

func loadGob(fileName string, key interface{}) {
        inFile, err := os.Open(fileName)
        checkError(err)
        decoder := gob.NewDecoder(inFile)
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

A client and server

A client to send a person's data and read it back ten times is

```
/* Gob EchoClient
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/gob"
        "bytes"
        "io"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {
        person := Person{
                Name: Name{Family: "Newmarch", Personal: "Jan"},
                Email:        []Email{Email{Kind:        "home",        Address:
"jan@newmarch.name"},
                        Email{Kind:               "work",               Address:
"j.newmarch@boxhill.edu.au"}}}

        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        encoder := gob.NewEncoder(conn)
        decoder := gob.NewDecoder(conn)

        for n := 0; n < 10; n++ {
                encoder.Encode(person)
                var newPerson Person
                decoder.Decode(&newPerson)
                fmt.Println(newPerson.String())
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

and the corrsponding server is

```
/* Gob EchoServer
```

```
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/gob"
)

type Person struct {
        Name  Name
        Email []Email
}

type Name struct {
        Family   string
        Personal string
}

type Email struct {
        Kind    string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {

        service := "0.0.0.0:1200"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                encoder := gob.NewEncoder(conn)
                decoder := gob.NewDecoder(conn)

                for n := 0; n < 10; n++ {
                        var person Person
                        decoder.Decode(&person)
                        fmt.Println(person.String())
                        encoder.Encode(person)
                }
                conn.Close() // we're finished
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Encoding binary data as strings

Once upon a time, transmtting 8-bit data was problematic. It was often transmitted over noisy serial lines and could easily become corrupted. 7-bit data on the other hand could be transmitted more reliably because the 8th bit could be used as check digit. For example, in an "even parity" scheme, the check digit would be set to one or zero to make an even number of 1's in a byte. This allows detection of errors of a single bit in each byte.

ASCII is a 7-bit character set. A number of schemes have been developed that are more sophisticated than simple parity checking, but which involve translating 8-bit binary data into 7-bit ASCII format. Essentially, the 8-bit data is stretched out in some way over the 7-bit bytes.

Binary data transmitted in HTTP responses and requests is often translated into an ASCII form. This makes it easy to inspect the HTTP messages with a simple text reader without worrying about what strange 8-bit bytes might do to your display!

One common format is Base64. Go has support for many binary-to-text formats, including base64.

There are two principal functions to use for Base64 encoding and decoding:

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

A simple program just to encode and decode a set of eight binary digits is

```
/**
 * Base64
 */

package main
```

```go
import (
        "bytes"
        "encoding/base64"
        "fmt"
)

func main() {

        eightBitData := []byte{1, 2, 3, 4, 5, 6, 7, 8}
        bb := &bytes.Buffer{}
        encoder := base64.NewEncoder(base64.StdEncoding, bb)
        encoder.Write(eightBitData)
        encoder.Close()
        fmt.Println(bb)

        dbuf := make([]byte, 12)
        decoder := base64.NewDecoder(base64.StdEncoding, bb)
        decoder.Read(dbuf)
        for _, ch := range dbuf {
                fmt.Print(ch)
        }
}
```

Source: http://jan.newmarch.name/go/serialisation/chapter-serialisation.html