XML

XML is a significant markup language mainly intended as a means of serialising data structures as a text document. Go has basic support for XML document processing.

Introduction

XML is now a widespread way of representing complex data structures serialised into text format. It is used to describe documents such as DocBook and XHTML. It is used in specialised markup languages such as MathML and CML (Chemistry Markup Language). It is used to encode data as SOAP messages for Web Services, and the Web Service can be specified using WSDL (Web Services Description Language).

At the simplest level, XML allows you to define your own tags for use in text documents. Tags can be nested and can be interspersed with text. Each tag can also contain attributes with values. For example,

```
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>
```

The structure of any XML document can be described in a number of ways:

- A document type definition DTD is good for describing structure
- XML schema are good for describing the data types used by an XML document
- RELAX NG is proposed as an alternative to both

There is argument over the relative value of each way of defining the structure of an XML document. We won't buy into that, as Go does not suport any of them. Go cannot check for validity of any document against a schema, but only for well-formedness.

Four topics are discussed in this chapter: parsing an XML stream, marshalling and unmarshalling Go data into XML, and XHTML.

Parsing XML

Go has an XML parser which is created using `NewParser`. This takes an `io.Reader` as parameter and returns a pointer to `Parser`. The main method of this type is `Token` which returns the next token in the input stream. The token is one of the types `StartElement`, `EndElement`, `CharData`, `Comment`, `ProcInst` or `Directive`.

The types are

`StartElement`

The type `StartElement` is a structure with two field types:

```
type StartElement struct {
    Name Name
    Attr []Attr
}

type Name struct {
    Space, Local string
}

type Attr struct {
    Name  Name
    Value string
}
```

`EndElement`

This is also a structure

```
type EndElement struct {
    Name Name
}
```

`CharData`

This type represents the text content enclosed by a tag and is a simple type

```
type CharData []byte
```

`Comment`

Similarly for this type

```
type Comment []byte
```

`ProcInst`

A ProcInst represents an XML processing instruction of the form <?target inst?>

```
type ProcInst struct {
    Target string
    Inst   []byte
}
```

`Directive`

A Directive represents an XML directive of the form <!text>. The bytes do not include the <! and > markers.

```
type Directive []byte
```

A program to print out the tree structure of an XML document is

```
/* Parse XML
 */

package main

import (
        "encoding/xml"
        "fmt"
        "io/ioutil"
        "os"
        "strings"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "file")
                os.Exit(1)
        }
        file := os.Args[1]
        bytes, err := ioutil.ReadFile(file)
        checkError(err)
        r := strings.NewReader(string(bytes))

        parser := xml.NewDecoder(r)
        depth := 0
        for {
                token, err := parser.Token()
                if err != nil {
                        break
                }
                switch t := token.(type) {
                case xml.StartElement:
                        elmt := xml.StartElement(t)
                        name := elmt.Name.Local
                        printElmt(name, depth)
                        depth++
                case xml.EndElement:
                        depth--
                        elmt := xml.EndElement(t)
                        name := elmt.Name.Local
                        printElmt(name, depth)
                case xml.CharData:
                        bytes := xml.CharData(t)
                        printElmt("\""+string([]byte(bytes))+"\"", depth)
                case xml.Comment:
                        printElmt("Comment", depth)
                case xml.ProcInst:
                        printElmt("ProcInst", depth)
                case xml.Directive:
                        printElmt("Directive", depth)
```

```
                default:
                        fmt.Println("Unknown")
                }
        }
}

func printElmt(s string, depth int) {
        for n := 0; n < depth; n++ {
                fmt.Print("  ")
        }
        fmt.Println(s)
}

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

Note that the parser includes all CharData, including the whitespace between tags.

If we run this program against the `person` data structure given earlier, it produces

```
person
  "
  "
  name
    "
    "
    family
      " Newmarch "
    family
    "
    "
    personal
      " Jan "
    personal
    "
  "
  name
  "
  "
  email
    "
    jan@newmarch.name
  "
  email
  "
  "
  email
    "
    j.newmarch@boxhill.edu.au
  "
  email
  "
```

```
"
person
"
"
```

Note that as no DTD or other XML specification has been used, the tokenizer correctly prints out all the white space (a DTD may specify that the whitespace can be ignored, but without it that assumption cannot be made.)

There is a potential trap in using this parser. It re-uses space for strings, so that once you see a token you need to copy its value if you want to refer to it later. Go has methods such as `func (c CharData) Copy() CharData` to make a copy of data.

Unmarshalling XML

Go provides a function `Unmarshal` and a method `func (*Parser) Unmarshal` to unmarshal XML into Go data structures. The unmarshalling is not perfect: Go and XML are different languages.

We consider a simple example before looking at the details. We take the XML document given earlier of

```
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>
```

We would like to map this onto the Go structures

```
type Person struct {
        Name Name
        Email []Email
}

type Name struct {
        Family string
        Personal string
}

type Email struct {
        Type string
        Address string
```

```
}
```

This requires several comments:

1. Unmarshalling uses the Go reflection package. This requires that all fields by public i.e. start with a capital letter. Earlier versions of Go used case-insensitive matching to match fields such as the XML string "name" to the field `Name`. Now, though, *case-sensitive* matching is used. To perform a match, the structure fields must be tagged to show the XML string that will be matched against. This changes `Person` to

```
type Person struct {
  Name Name `xml:"name"`
  Email []Email `xml:"email"`
}
```

6. While tagging of fields can attach XML strings to fields, it can't do so with the names of the structures. An additional field is required, with field name "XMLName". This only affects the top-level struct, `Person`

```
type Person struct {
        XMLName Name `xml:"person"`
  Name Name `xml:"name"`
         Email []Email `xml:"email"`
 }
```

12. Repeated tags in the map to a slice in Go
13. Attributes within tags will match to fields in a structure only if the Go field has the tag ",attr". This occurs with the field `Type` of `Email`, where matching the attribute "type" of the "email" tag requires `` `xml:"type,attr"` ``
14. If an XML tag has no attributes and only has character data, then it matches a `string` field by the same name (case-sensitive, though). So the tag `` `xml:"family"` `` with character data "Newmarch" maps to the string field `Family`
15. But if the tag has attributes, then it must map to a structure. Go assigns the character data to the field with tag `,chardata`. This occurs with the "email" data and the field `Address` with tag `,chardata`

A program to unmarshal the document above is

```
/* Unmarshal
 */

package main

import (
```

```
        "encoding/xml"
        "fmt"
        "os"
        //"strings"
)

type Person struct {
        XMLName Name     `xml:"person"`
        Name    Name     `xml:"name"`
        Email   []Email  `xml:"email"`
}

type Name struct {
        Family   string `xml:"family"`
        Personal string `xml:"personal"`
}

type Email struct {
        Type    string `xml:"type,attr"`
        Address string `xml:",chardata"`
}

func main() {
        str := `<?xml version="1.0" encoding="utf-8"?>
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>`

        var person Person

        err := xml.Unmarshal([]byte(str), &person)
        checkError(err)

        // now use the person structure e.g.
        fmt.Println("Family name: \"" + person.Name.Family + "\"")
        fmt.Println("Second email address: \"" + person.Email[1].Address +
"\"")
}

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

(Note the spaces are correct.). The strict rules are given in the package specification.

Marshalling XML

Go 1 also has support for marshalling data structures into an XML document. The function is

```
func Marshal(v interface}{) ([]byte, error)
```

This was used as a check in the last two lines of the previous program.

XHTML

HTML does not conform to XML syntax. It has unterminated tags such as '
'. XHTML is a cleanup of HTML to make it compliant to XML. Documents in XHTML can be managed using the techniques above for XML.

HTML

There is some support in the XML package to handle HTML documents even though they are not XML-compliant. The XML parser discussed earlier can handle many HTML documents if it is modified by

```
        parser := xml.NewDecoder(r)
        parser.Strict = false
        parser.AutoClose = xml.HTMLAutoClose
        parser.Entity = xml.HTMLEntity
```

Conclusion

Go has basic support for dealing with XML strings. It does not as yet have mechanisms for dealing with XML specification languages such as XML Schema or Relax NG.

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