Package xml go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package xml implements a simple XML 1.0 parser that understands XML name spaces.

Constants

```
const (
    // Header is a generic XML header suitable for use with the output of Marshal.
    // This is not automatically added to any output of this package,
    // it is provided as a convenience.
    Header = `<?xml version="1.0" encoding="UTF-8"?>` + "\n"
)
```

Variables

```
var HTMLAutoClose []string = htmlAutoClose
```

HTMLAutoClose is the set of HTML elements that should be considered to close automatically.

See the Decoder.Strict and Decoder.Entity fields' documentation.

```
var HTMLEntity map[string]string = htmlEntity
```

HTMLEntity is an entity map containing translations for the standard HTML entity characters.

See the Decoder.Strict and Decoder.Entity fields' documentation.

func Escape

```
func Escape(w io.Writer, s []byte)
```

Escape is like EscapeText but omits the error return value. It is provided for backwards compatibility with Go 1.0. Code targeting Go 1.1 or later should use EscapeText.

func EscapeText

```
func EscapeText(w io.Writer, s []byte) error
```

EscapeText writes to w the properly escaped XML equivalent of the plain text data s.

func Marshal

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

Marshal returns the XML encoding of v.

Marshal handles an array or slice by marshaling each of the elements. Marshal handles a pointer by marshaling the value it points at or, if the pointer is nil, by writing nothing. Marshal handles an interface value by marshaling the value it contains or, if the interface value is nil, by writing nothing. Marshal handles all other data by writing one or more XML elements containing the data.

The name for the XML elements is taken from, in order of preference:

```
- the tag on the XMLName field, if the data is a struct
- the value of the XMLName field of type Name
- the tag of the struct field used to obtain the data
- the name of the struct field used to obtain the data
- the name of the marshaled type
```

The XML element for a struct contains marshaled elements for each of the exported fields of the struct, with these exceptions:

```
- the XMLName field, described above, is omitted.
- a field with tag "-" is omitted.
- a field with tag "name,attr" becomes an attribute with
  the given name in the XML element.
- a field with tag ",attr" becomes an attribute with the
  field name in the XML element.
- a field with tag ",chardata" is written as character data,
  not as an XML element.
- a field with tag ",cdata" is written as character data
  wrapped in one or more <![CDATA[ ... ]]> tags, not as an XML element.
- a field with tag ",innerxml" is written verbatim, not subject
  to the usual marshaling procedure.
- a field with tag ",comment" is written as an XML comment, not
  subject to the usual marshaling procedure. It must not contain
  the "--" string within it.
- a field with a tag including the "omitempty" option is omitted
  if the field value is empty. The empty values are false, 0, any
  nil pointer or interface value, and any array, slice, map, or
  string of length zero.
- an anonymous struct field is handled as if the fields of its
  value were part of the outer struct.
- a field implementing Marshaler is written by calling its MarshalXML
  method.
- a field implementing encoding.TextMarshaler is written by encoding the
  result of its MarshalText method as text.
```

If a field uses a tag "a>b>c", then the element c will be nested inside parent elements a and b. Fields that appear next to each other that name the same parent will be enclosed in one XML element.

If the XML name for a struct field is defined by both the field tag and the struct's XMLName field, the names must match.

See MarshalIndent for an example.

Marshal will return an error if asked to marshal a channel, function, or map.

func MarshalIndent

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

MarshalIndent works like Marshal, but each XML element begins on a new indented line that starts with prefix and is followed by one or more copies of indent according to the nesting depth.

func Unmarshal

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

Unmarshal parses the XML-encoded data and stores the result in the value pointed to by v, which must be an arbitrary struct, slice, or string. Well-formed data that does not fit into v is discarded.

Because Unmarshal uses the reflect package, it can only assign to exported (upper case) fields. Unmarshal uses a case-sensitive comparison to match XML element names to tag values and struct field names.

Unmarshal maps an XML element to a struct using the following rules. In the rules, the tag of a field refers to the value associated with the key 'xml' in the struct field's tag (see the example above).

```
* If the struct has a field of type []byte or string with tag
   ",innerxml", Unmarshal accumulates the raw XML nested inside the
   element in that field. The rest of the rules still apply.

* If the struct has a field named XMLName of type Name,
   Unmarshal records the element name in that field.

* If the XMLName field has an associated tag of the form
   "name" or "namespace-URL name", the XML element must have
   the given name (and, optionally, name space) or else Unmarshal
   returns an error.

* If the XML element has an attribute whose name matches a
   struct field name with an associated tag containing ",attr" or
   the explicit name in a struct field tag of the form "name,attr",
   Unmarshal records the attribute value in that field.

* If the XML element has an attribute not handled by the previous
   rule and the struct has a field with an associated tag containing
   ",any,attr", Unmarshal records the attribute value in the first
   such field.
```

```
* If the XML element contains character data, that data is
   accumulated in the first struct field that has tag ",chardata".
   The struct field may have type []byte or string.
   If there is no such field, the character data is discarded.

* If the XML element contains comments, they are accumulated in
   the first struct field that has tag ",comment".  The struct
   field may have type []byte or string. If there is no such
   field, the comments are discarded.

* If the XML element contains a sub-element whose name matches
   the prefix of a tag formatted as "a" or "a>b>c", unmarshal
   will descend into the XML structure looking for elements with the
   given names, and will map the innermost elements to that struct
   field. A tag starting with ">" is equivalent to one starting
   with the field name followed by ">".

* If the XML element contains a sub-element whose name matches
   a struct field's XMLName tag and the struct field has no
   explicit name tag as per the previous rule, unmarshal maps
   the sub-element to that struct field.

* If the XML element contains a sub-element whose name matches a
   field without any mode flags (",attr", ",chardata", etc), Unmarshal
   maps the sub-element to that struct field.

* If the XML element contains a sub-element that hasn't matched any
   of the above rules and the struct has a field with tag ",any",
   unmarshal maps the sub-element to that struct field.

* An anonymous struct field is handled as if the fields of its
   value were part of the outer struct.

* A struct field with tag "-" is never unmarshaled into.
```

If Unmarshal encounters a field type that implements the Unmarshaler interface, Unmarshal calls its UnmarshalXML method to produce the value from the XML element. Otherwise, if the value implements encoding.TextUnmarshaler, Unmarshal calls that value's UnmarshalText method.

Unmarshal maps an XML element to a string or []byte by saving the concatenation of that element's character data in the string or []byte. The saved []byte is never nil.

Unmarshal maps an attribute value to a string or []byte by saving the value in the string or slice.

Unmarshal maps an attribute value to an Attr by saving the attribute, including its name, in the Attr.

Unmarshal maps an XML element or attribute value to a slice by extending the length of the slice and mapping the element or attribute to the newly created value.

Unmarshal maps an XML element or attribute value to a bool by setting it to the boolean value represented by the string. Whitespace is trimmed and ignored.

Unmarshal maps an XML element or attribute value to an integer or floating-point field by setting the field to the result of interpreting the string value in decimal. There is no check for overflow. Whitespace is trimmed and ignored.

Unmarshal maps an XML element to a Name by recording the element name.

Unmarshal maps an XML element to a pointer by setting the pointer to a freshly allocated value and then mapping the element to that value.

A missing element or empty attribute value will be unmarshaled as a zero value. If the field is a slice, a zero value will be appended to the field. Otherwise, the field will be set to its zero value.

type Attr

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

An Attr represents an attribute in an XML element (Name=Value).

type CharData

```
type CharData []byte
```

A CharData represents XML character data (raw text), in which XML escape sequences have been replaced by the characters they represent.

func (CharData) Copy

```
func (c CharData) Copy() CharData
```

Copy creates a new copy of CharData.

type Comment

```
type Comment []byte
```

A Comment represents an XML comment of the form <!--comment-->. The bytes do not include the <!-- and --> comment markers.

func (Comment) Copy

```
func (c Comment) Copy() Comment
```

Copy creates a new copy of Comment.

type Decoder

```
type Decoder struct {
    // Strict defaults to true, enforcing the requirements
    // of the XML specification.
    // If set to false, the parser allows input containing common
    // mistakes:
    //  * If an element is missing an end tag, the parser invents
    //    end tags as necessary to keep the return values from Token
    //    properly balanced.
    //  * In attribute values and character data, unknown or malformed
    //    character entities (sequences beginning with &) are left alone.
    //
    // Setting:
    //
    //  d.Strict = false
    //  d.AutoClose = xml.HTMLAutoClose
    //  d.Entity = xml.HTMLEntity
    //
    // creates a parser that can handle typical HTML.
    //
    // Strict mode does not enforce the requirements of the XML name spaces TR.
    // In particular it does not reject name space tags using undefined prefixes.
    // Such tags are recorded with the unknown prefix as the name space URL.
    Strict bool

    // When Strict == false, AutoClose indicates a set of elements to
    // consider closed immediately after they are opened, regardless
    // of whether an end element is present.
    AutoClose []string

    // Entity can be used to map non-standard entity names to string replacements.
    // The parser behaves as if these standard mappings are present in the map,
    // regardless of the actual map content:
    //
    //  "lt": "<",
    //  "gt": ">",
    //  "amp": "&",
    //  "apos": "'",
    //  "quot": `"`,
    Entity map[string]string

    // CharsetReader, if non-nil, defines a function to generate
    // charset-conversion readers, converting from the provided
    // non-UTF-8 charset into UTF-8. If CharsetReader is nil or
    // returns an error, parsing stops with an error. One of the
    // CharsetReader's result values must be non-nil.
    CharsetReader func(charset string, input io.Reader) (io.Reader, error)

    // DefaultSpace sets the default name space used for unadorned tags,
    // as if the entire XML stream were wrapped in an element containing
    // the attribute xmlns="DefaultSpace".
    DefaultSpace string
```

```
    // contains filtered or unexported fields
}
```

A Decoder represents an XML parser reading a particular input stream. The parser assumes that its input is encoded in UTF-8.

func NewDecoder

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder creates a new XML parser reading from r. If r does not implement io.ByteReader, NewDecoder will do its own buffering.

func NewTokenDecoder

```
func NewTokenDecoder(t TokenReader) *Decoder
```

NewTokenDecoder creates a new XML parser using an underlying token stream.

func (*Decoder) Decode

```
func (d *Decoder) Decode(v interface{}) error
```

Decode works like Unmarshal, except it reads the decoder stream to find the start element.

func (*Decoder) DecodeElement

```
func (d *Decoder) DecodeElement(v interface{}, start *StartElement) error
```

DecodeElement works like Unmarshal except that it takes a pointer to the start XML element to decode into v. It is useful when a client reads some raw XML tokens itself but also wants to defer to Unmarshal for some elements.

func (*Decoder) InputOffset

```
func (d *Decoder) InputOffset() int64
```

InputOffset returns the input stream byte offset of the current decoder position. The offset gives the location of the end of the most recently returned token and the beginning of the next token.

func (*Decoder) RawToken

```
func (d *Decoder) RawToken() (Token, error)
```

RawToken is like Token but does not verify that start and end elements match and does not translate name space prefixes to their corresponding URLs.

func (*Decoder) Skip

```
func (d *Decoder) Skip() error
```

Skip reads tokens until it has consumed the end element matching the most recent start element already consumed. It recurs if it encounters a start element, so it can be used to skip nested structures. It returns nil if it finds an end element matching the start element; otherwise it returns an error describing the problem.

func (*Decoder) Token

```
func (d *Decoder) Token() (Token, error)
```

Token returns the next XML token in the input stream. At the end of the input stream, Token returns nil, io.EOF.

Slices of bytes in the returned token data refer to the parser's internal buffer and remain valid only until the next call to Token. To acquire a copy of the bytes, call CopyToken or the token's Copy method.

Token expands self-closing elements such as
 into separate start and end elements returned by successive calls.

Token guarantees that the StartElement and EndElement tokens it returns are properly nested and matched: if Token encounters an unexpected end element or EOF before all expected end elements, it will return an error.

Token implements XML name spaces as described by https://www.w3.org/TR/REC-xml-names/. Each of the Name structures contained in the Token has the Space set to the URL identifying its name space when known. If Token encounters an unrecognized name space prefix, it uses the prefix as the Space rather than report an error.

type Directive

```
type Directive []byte
```

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

func (Directive) Copy

```
func (d Directive) Copy() Directive
```

Copy creates a new copy of Directive.

type Encoder

```
type Encoder struct {
    // contains filtered or unexported fields
}
```

An Encoder writes XML data to an output stream.

func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that writes to w.

func (*Encoder) Encode

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode writes the XML encoding of v to the stream.

See the documentation for Marshal for details about the conversion of Go values to XML.

Encode calls Flush before returning.

func (*Encoder) EncodeElement

```
func (enc *Encoder) EncodeElement(v interface{}, start StartElement) error
```

EncodeElement writes the XML encoding of v to the stream, using start as the outermost tag in the encoding.

See the documentation for Marshal for details about the conversion of Go values to XML.

EncodeElement calls Flush before returning.

func (*Encoder) EncodeToken

```
func (enc *Encoder) EncodeToken(t Token) error
```

EncodeToken writes the given XML token to the stream. It returns an error if StartElement and EndElement tokens are not properly matched.

EncodeToken does not call Flush, because usually it is part of a larger operation such as Encode or EncodeElement (or a custom Marshaler's MarshalXML invoked during those), and those will call Flush when finished. Callers that create an Encoder and then invoke EncodeToken directly, without using Encode or EncodeElement, need to call Flush when finished to ensure that the XML is written to the underlying writer.

EncodeToken allows writing a ProcInst with Target set to "xml" only as the first token in the stream.

func (*Encoder) Flush

```
func (enc *Encoder) Flush() error
```

Flush flushes any buffered XML to the underlying writer. See the EncodeToken documentation for details about when it is necessary.

func (*Encoder) Indent

```
func (enc *Encoder) Indent(prefix, indent string)
```

Indent sets the encoder to generate XML in which each element begins on a new indented line that starts with prefix and is followed by one or more copies of indent according to the nesting depth.

type EndElement

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

An EndElement represents an XML end element.

type Marshaler

```
type Marshaler interface {
    MarshalXML(e *Encoder, start StartElement) error
}
```

Marshaler is the interface implemented by objects that can marshal themselves into valid XML elements.

MarshalXML encodes the receiver as zero or more XML elements. By convention, arrays or slices are typically encoded as a sequence of elements, one per entry. Using start as the element tag is not required, but doing so will enable Unmarshal to match the XML elements to the correct struct field. One common implementation strategy is to construct a separate value with a layout corresponding to the desired XML and then to encode it using e.EncodeElement. Another common strategy is to use repeated calls to e.EncodeToken to generate the XML output one token at a time. The sequence of encoded tokens must make up zero or more valid XML elements.

type MarshalerAttr

```
type MarshalerAttr interface {
    MarshalXMLAttr(name Name) (Attr, error)
}
```

MarshalerAttr is the interface implemented by objects that can marshal themselves into valid XML attributes.

MarshalXMLAttr returns an XML attribute with the encoded value of the receiver. Using name as the attribute name is not required, but doing so will enable Unmarshal to match the attribute to the correct struct field. If MarshalXMLAttr returns the zero attribute Attr{}, no attribute will be generated in the output. MarshalXMLAttr is used only for struct fields with the "attr" option in the field tag.

type Name

```
type Name struct {
    Space, Local string
}
```

A Name represents an XML name (Local) annotated with a name space identifier (Space). In tokens returned by Decoder.Token, the Space identifier is given as a canonical URL, not the short prefix used in the document being parsed.

type ProcInst

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

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

func (ProcInst) Copy

```
func (p ProcInst) Copy() ProcInst
```

Copy creates a new copy of ProcInst.

type StartElement

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

A StartElement represents an XML start element.

func (StartElement) Copy

```
func (e StartElement) Copy() StartElement
```

Copy creates a new copy of StartElement.

func (StartElement) End

```
func (e StartElement) End() EndElement
```

End returns the corresponding XML end element.

type SyntaxError

```
type SyntaxError struct {
    Msg  string
    Line int
}
```

A SyntaxError represents a syntax error in the XML input stream.

func (*SyntaxError) Error

```
func (e *SyntaxError) Error() string
```

type TagPathError

```
type TagPathError struct {
    Struct       reflect.Type
    Field1, Tag1 string
    Field2, Tag2 string
}
```

A TagPathError represents an error in the unmarshaling process caused by the use of field tags with conflicting paths.

func (*TagPathError) Error

```
func (e *TagPathError) Error() string
```

type Token

```
type Token interface{}
```

A Token is an interface holding one of the token types: StartElement, EndElement, CharData, Comment, ProcInst, or Directive.

func CopyToken

```
func CopyToken(t Token) Token
```

CopyToken returns a copy of a Token.

type TokenReader

```
type TokenReader interface {
    Token() (Token, error)
}
```

A TokenReader is anything that can decode a stream of XML tokens, including a Decoder.

When Token encounters an error or end-of-file condition after successfully reading a token, it returns the token. It may return the (non-nil) error from the same call or return the error (and a nil token) from a subsequent call. An instance of this general case is that a TokenReader returning a non-nil token at the end of the token stream may return either io.EOF or a nil error. The next Read should return nil, io.EOF.

Implementations of Token are discouraged from returning a nil token with a nil error. Callers should treat a return of nil, nil as indicating that nothing happened; in particular it does not indicate EOF.

type UnmarshalError

```
type UnmarshalError string
```

An UnmarshalError represents an error in the unmarshaling process.

func (UnmarshalError) Error

```
func (e UnmarshalError) Error() string
```

type Unmarshaler

```
type Unmarshaler interface {
    UnmarshalXML(d *Decoder, start StartElement) error
}
```

Unmarshaler is the interface implemented by objects that can unmarshal an XML element description of themselves.

UnmarshalXML decodes a single XML element beginning with the given start element. If it returns an error, the outer call to Unmarshal stops and returns that error. UnmarshalXML must consume exactly one XML element. One common implementation strategy is to unmarshal into a separate value with a layout matching the expected XML using d.DecodeElement, and then to copy the data from that value into the receiver. Another common strategy is to use d.Token to process the XML object one token at a time. UnmarshalXML may not use d.RawToken.

type UnmarshalerAttr

```
type UnmarshalerAttr interface {
    UnmarshalXMLAttr(attr Attr) error
```

```
}
```

UnmarshalerAttr is the interface implemented by objects that can unmarshal an XML attribute description of themselves.

UnmarshalXMLAttr decodes a single XML attribute. If it returns an error, the outer call to Unmarshal stops and returns that error. UnmarshalXMLAttr is used only for struct fields with the "attr" option in the field tag.

type UnsupportedTypeError

```
type UnsupportedTypeError struct {
    Type reflect.Type
}
```

UnsupportedTypeError is returned when Marshal encounters a type that cannot be converted into XML.

func (*UnsupportedTypeError) Error

```
func (e *UnsupportedTypeError) Error() string
```

BUGs

- Mapping between XML elements and data structures is inherently flawed: an XML element is an order-dependent collection of anonymous values, while a data structure is an order-independent collection of named values. See package json for a textual representation more suitable to data structures.