Package json go1.15.2 Latest

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

Overview

Package json implements encoding and decoding of JSON as defined in RFC 7159. The mapping between JSON and Go values is described in the documentation for the Marshal and Unmarshal functions.

See "JSON and Go" for an introduction to this package:
https://golang.org/doc/articles/json_and_go.html

func Compact

```
func Compact(dst *bytes.Buffer, src []byte) error
```

Compact appends to dst the JSON-encoded src with insignificant space characters elided.

func HTMLEscape

```
func HTMLEscape(dst *bytes.Buffer, src []byte)
```

HTMLEscape appends to dst the JSON-encoded src with <, >, &, U+2028 and U+2029 characters inside string literals changed to \u003c, \u003e, \u0026, \u2028, \u2029 so that the JSON will be safe to embed inside HTML <script> tags. For historical reasons, web browsers don't honor standard HTML escaping within <script> tags, so an alternative JSON encoding must be used.

func Indent

```
func Indent(dst *bytes.Buffer, src []byte, prefix, indent string) error
```

Indent appends to dst an indented form of the JSON-encoded src. Each element in a JSON object or array begins on a new, indented line beginning with prefix followed by one or more copies of indent according to the indentation nesting. The data appended to dst does not begin with the prefix nor any indentation, to make it easier to embed inside other formatted JSON data. Although leading space characters (space, tab, carriage return, newline) at the beginning of src are dropped, trailing space characters at the end of src are preserved and copied to dst. For example, if src has no trailing spaces, neither will dst; if src ends in a trailing newline, so will dst.

func Marshal

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

Marshal returns the JSON encoding of v.

Marshal traverses the value v recursively. If an encountered value implements the Marshaler interface and is not a nil pointer, Marshal calls its MarshalJSON method to produce JSON. If no MarshalJSON method is present but the value implements encoding.TextMarshaler instead, Marshal calls its MarshalText method and encodes the result as a JSON string. The nil pointer exception is not strictly necessary but mimics a similar, necessary exception in the behavior of UnmarshalJSON.

Otherwise, Marshal uses the following type-dependent default encodings:

Boolean values encode as JSON booleans.

Floating point, integer, and Number values encode as JSON numbers.

String values encode as JSON strings coerced to valid UTF-8, replacing invalid bytes with the Unicode replacement rune. So that the JSON will be safe to embed inside HTML <script> tags, the string is encoded using HTMLEscape, which replaces "<", ">", "&", U+2028, and U+2029 are escaped to "\u003c","\u003e", "\u0026", "\u2028", and "\u2029". This replacement can be disabled when using an Encoder, by calling SetEscapeHTML(false).

Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string, and a nil slice encodes as the null JSON value.

Struct values encode as JSON objects. Each exported struct field becomes a member of the object, using the field name as the object key, unless the field is omitted for one of the reasons given below.

The encoding of each struct field can be customized by the format string stored under the "json" key in the struct field's tag. The format string gives the name of the field, possibly followed by a comma-separated list of options. The name may be empty in order to specify options without overriding the default field name.

The "omitempty" option specifies that the field should be omitted from the encoding if the field has an empty value, defined as false, 0, a nil pointer, a nil interface value, and any empty array, slice, map, or string.

As a special case, if the field tag is "-", the field is always omitted. Note that a field with name "-" can still be generated using the tag "-,".

Examples of struct field tags and their meanings:

```
// Field appears in JSON as key "myName".
Field int `json:"myName"`

// Field appears in JSON as key "myName" and
// the field is omitted from the object if its value is empty,
// as defined above.
Field int `json:"myName,omitempty"`
```

```
// Field appears in JSON as key "Field" (the default), but
// the field is skipped if empty.
// Note the leading comma.
Field int `json:",omitempty"`

// Field is ignored by this package.
Field int `json:"-"`

// Field appears in JSON as key "-".
Field int `json:"-,"`
```

The "string" option signals that a field is stored as JSON inside a JSON-encoded string. It applies only to fields of string, floating point, integer, or boolean types. This extra level of encoding is sometimes used when communicating with JavaScript programs:

```
Int64String int64 `json:",string"`
```

The key name will be used if it's a non-empty string consisting of only Unicode letters, digits, and ASCII punctuation except quotation marks, backslash, and comma.

Anonymous struct fields are usually marshaled as if their inner exported fields were fields in the outer struct, subject to the usual Go visibility rules amended as described in the next paragraph. An anonymous struct field with a name given in its JSON tag is treated as having that name, rather than being anonymous. An anonymous struct field of interface type is treated the same as having that type as its name, rather than being anonymous.

The Go visibility rules for struct fields are amended for JSON when deciding which field to marshal or unmarshal. If there are multiple fields at the same level, and that level is the least nested (and would therefore be the nesting level selected by the usual Go rules), the following extra rules apply:

1) Of those fields, if any are JSON-tagged, only tagged fields are considered, even if there are multiple untagged fields that would otherwise conflict.

2) If there is exactly one field (tagged or not according to the first rule), that is selected.

3) Otherwise there are multiple fields, and all are ignored; no error occurs.

Handling of anonymous struct fields is new in Go 1.1. Prior to Go 1.1, anonymous struct fields were ignored. To force ignoring of an anonymous struct field in both current and earlier versions, give the field a JSON tag of "-".

Map values encode as JSON objects. The map's key type must either be a string, an integer type, or implement encoding.TextMarshaler. The map keys are sorted and used as JSON object keys by applying the following rules, subject to the UTF-8 coercion described for string values above:

```
- keys of any string type are used directly
- encoding.TextMarshalers are marshaled
- integer keys are converted to strings
```

Pointer values encode as the value pointed to. A nil pointer encodes as the null JSON value.

Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON value.

Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value causes Marshal to return an UnsupportedTypeError.

JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an error.

func MarshalIndent

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

MarshalIndent is like Marshal but applies Indent to format the output. Each JSON element in the output will begin on a new line beginning with prefix followed by one or more copies of indent according to the indentation nesting.

func Unmarshal

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

Unmarshal parses the JSON-encoded data and stores the result in the value pointed to by v. If v is nil or not a pointer, Unmarshal returns an InvalidUnmarshalError.

Unmarshal uses the inverse of the encodings that Marshal uses, allocating maps, slices, and pointers as necessary, with the following additional rules:

To unmarshal JSON into a pointer, Unmarshal first handles the case of the JSON being the JSON literal null. In that case, Unmarshal sets the pointer to nil. Otherwise, Unmarshal unmarshals the JSON into the value pointed at by the pointer. If the pointer is nil, Unmarshal allocates a new value for it to point to.

To unmarshal JSON into a value implementing the Unmarshaler interface, Unmarshal calls that value's UnmarshalJSON method, including when the input is a JSON null. Otherwise, if the value implements encoding.TextUnmarshaler and the input is a JSON quoted string, Unmarshal calls that value's UnmarshalText method with the unquoted form of the string.

To unmarshal JSON into a struct, Unmarshal matches incoming object keys to the keys used by Marshal (either the struct field name or its tag), preferring an exact match but also accepting a case-insensitive match. By default, object keys which don't have a corresponding struct field are ignored (see Decoder.DisallowUnknownFields for an alternative).

To unmarshal JSON into an interface value, Unmarshal stores one of these in the interface value:

```
bool, for JSON booleans
float64, for JSON numbers
```

```
string, for JSON strings
[]interface{}, for JSON arrays
map[string]interface{}, for JSON objects
nil for JSON null
```

To unmarshal a JSON array into a slice, Unmarshal resets the slice length to zero and then appends each element to the slice. As a special case, to unmarshal an empty JSON array into a slice, Unmarshal replaces the slice with a new empty slice.

To unmarshal a JSON array into a Go array, Unmarshal decodes JSON array elements into corresponding Go array elements. If the Go array is smaller than the JSON array, the additional JSON array elements are discarded. If the JSON array is smaller than the Go array, the additional Go array elements are set to zero values.

To unmarshal a JSON object into a map, Unmarshal first establishes a map to use. If the map is nil, Unmarshal allocates a new map. Otherwise Unmarshal reuses the existing map, keeping existing entries. Unmarshal then stores key-value pairs from the JSON object into the map. The map's key type must either be any string type, an integer, implement json.Unmarshaler, or implement encoding.TextUnmarshaler.

If a JSON value is not appropriate for a given target type, or if a JSON number overflows the target type, Unmarshal skips that field and completes the unmarshaling as best it can. If no more serious errors are encountered, Unmarshal returns an UnmarshalTypeError describing the earliest such error. In any case, it's not guaranteed that all the remaining fields following the problematic one will be unmarshaled into the target object.

The JSON null value unmarshals into an interface, map, pointer, or slice by setting that Go value to nil. Because null is often used in JSON to mean ``not present,'' unmarshaling a JSON null into any other Go type has no effect on the value and produces no error.

When unmarshaling quoted strings, invalid UTF-8 or invalid UTF-16 surrogate pairs are not treated as an error. Instead, they are replaced by the Unicode replacement character U+FFFD.

func Valid

```
func Valid(data []byte) bool
```

Valid reports whether data is a valid JSON encoding.

type Decoder

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

A Decoder reads and decodes JSON values from an input stream.

func NewDecoder

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

NewDecoder returns a new decoder that reads from r.

The decoder introduces its own buffering and may read data from r beyond the JSON values requested.

func (*Decoder) Buffered

```
func (dec *Decoder) Buffered() io.Reader
```

Buffered returns a reader of the data remaining in the Decoder's buffer. The reader is valid until the next call to Decode.

func (*Decoder) Decode

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

Decode reads the next JSON-encoded value from its input and stores it in the value pointed to by v.

See the documentation for Unmarshal for details about the conversion of JSON into a Go value.

func (*Decoder) DisallowUnknownFields

```
func (dec *Decoder) DisallowUnknownFields()
```

DisallowUnknownFields causes the Decoder to return an error when the destination is a struct and the input contains object keys which do not match any non-ignored, exported fields in the destination.

func (*Decoder) InputOffset

```
func (dec *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) More

```
func (dec *Decoder) More() bool
```

More reports whether there is another element in the current array or object being parsed.

func (*Decoder) Token

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

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

Token guarantees that the delimiters [ ] { } it returns are properly nested and matched: if Token encounters an unexpected delimiter in the input, it will return an error.

The input stream consists of basic JSON values—bool, string, number, and null—along with delimiters [ ] { } of type Delim to mark the start and end of arrays and objects. Commas and colons are elided.

func (*Decoder) UseNumber

```
func (dec *Decoder) UseNumber()
```

UseNumber causes the Decoder to unmarshal a number into an interface{} as a Number instead of as a float64.

type Delim

```
type Delim rune
```

A Delim is a JSON array or object delimiter, one of [ ] { or }.

func (Delim) String

```
func (d Delim) String() string
```

type Encoder

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

An Encoder writes JSON values 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 JSON encoding of v to the stream, followed by a newline character.

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

func (*Encoder) SetEscapeHTML

```
func (enc *Encoder) SetEscapeHTML(on bool)
```

SetEscapeHTML specifies whether problematic HTML characters should be escaped inside JSON quoted strings. The default behavior is to escape &, <, and > to \u0026, \u003c, and \u003e to avoid certain safety problems that can arise when embedding JSON in HTML.

In non-HTML settings where the escaping interferes with the readability of the output, SetEscapeHTML(false) disables this behavior.

func (*Encoder) SetIndent

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

SetIndent instructs the encoder to format each subsequent encoded value as if indented by the package-level function Indent(dst, src, prefix, indent). Calling SetIndent("", "") disables indentation.

type InvalidUTF8Error

```
type InvalidUTF8Error struct {
    S string // the whole string value that caused the error
}
```

Before Go 1.2, an InvalidUTF8Error was returned by Marshal when attempting to encode a string value with invalid UTF-8 sequences. As of Go 1.2, Marshal instead coerces the string to valid UTF-8 by replacing invalid bytes with the Unicode replacement rune U+FFFD.

Deprecated: No longer used; kept for compatibility.

func (*InvalidUTF8Error) Error

```
func (e *InvalidUTF8Error) Error() string
```

type InvalidUnmarshalError

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

An InvalidUnmarshalError describes an invalid argument passed to Unmarshal. (The argument to Unmarshal must be a non-nil pointer.)

func (*InvalidUnmarshalError) Error

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

type Marshaler

```
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}
```

Marshaler is the interface implemented by types that can marshal themselves into valid JSON.

type MarshalerError

```
type MarshalerError struct {
    Type reflect.Type
    Err  error
    // contains filtered or unexported fields
}
```

A MarshalerError represents an error from calling a MarshalJSON or MarshalText method.

func (*MarshalerError) Error

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

func (*MarshalerError) Unwrap

```
func (e *MarshalerError) Unwrap() error
```

Unwrap returns the underlying error.

type Number

```
type Number string
```

A Number represents a JSON number literal.

func (Number) Float64

```
func (n Number) Float64() (float64, error)
```

Float64 returns the number as a float64.

func (Number) Int64

```
func (n Number) Int64() (int64, error)
```

Int64 returns the number as an int64.

func (Number) String

```
func (n Number) String() string
```

String returns the literal text of the number.

type RawMessage

```
type RawMessage []byte
```

RawMessage is a raw encoded JSON value. It implements Marshaler and Unmarshaler and can be used to delay JSON decoding or precompute a JSON encoding.

func (RawMessage) MarshalJSON

```
func (m RawMessage) MarshalJSON() ([]byte, error)
```

MarshalJSON returns m as the JSON encoding of m.

func (*RawMessage) UnmarshalJSON

```
func (m *RawMessage) UnmarshalJSON(data []byte) error
```

UnmarshalJSON sets *m to a copy of data.

type SyntaxError

```
type SyntaxError struct {
    Offset int64 // error occurred after reading Offset bytes
    // contains filtered or unexported fields
}
```

A SyntaxError is a description of a JSON syntax error.

func (*SyntaxError) Error

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

type Token

```
type Token interface{}
```

A Token holds a value of one of these types:

```
Delim, for the four JSON delimiters [ ] { }
bool, for JSON booleans
float64, for JSON numbers
Number, for JSON numbers
string, for JSON string literals
nil, for JSON null
```

type UnmarshalFieldError

```
type UnmarshalFieldError struct {
    Key   string
    Type  reflect.Type
    Field reflect.StructField
}
```

An UnmarshalFieldError describes a JSON object key that led to an unexported (and therefore unwritable) struct field.

Deprecated: No longer used; kept for compatibility.

func (*UnmarshalFieldError) Error

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

type UnmarshalTypeError

```
type UnmarshalTypeError struct {
    Value  string       // description of JSON value - "bool", "array", "number -5"
    Type   reflect.Type // type of Go value it could not be assigned to
    Offset int64        // error occurred after reading Offset bytes
    Struct string       // name of the struct type containing the field
    Field  string       // the full path from root node to the field
}
```

An UnmarshalTypeError describes a JSON value that was not appropriate for a value of a specific Go type.

func (*UnmarshalTypeError) Error

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

type Unmarshaler

```
type Unmarshaler interface {
    UnmarshalJSON([]byte) error
```

```
}
```

Unmarshaler is the interface implemented by types that can unmarshal a JSON description of themselves. The input can be assumed to be a valid encoding of a JSON value. UnmarshalJSON must copy the JSON data if it wishes to retain the data after returning.

By convention, to approximate the behavior of Unmarshal itself, Unmarshalers implement UnmarshalJSON([]byte("null")) as a no-op.

type UnsupportedTypeError

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

An UnsupportedTypeError is returned by Marshal when attempting to encode an unsupported value type.

func (*UnsupportedTypeError) Error

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

type UnsupportedValueError

```
type UnsupportedValueError struct {
    Value reflect.Value
    Str   string
}
```

func (*UnsupportedValueError) Error

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