

[Discover Packages](#) > [Standard library](#) > [text](#) > [template](#) 


template

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [16](#) |Imported by: [58,082](#)

Details

 Valid [go.mod](#) file  Redistributable license  Tagged version  Stable version [Learn more](#)

Repository

cs.opensource.google/go/go

Links

 [Report a Vulnerability](#) Documentation 

<> Documentation

Overview

[Text and spaces](#)[Actions](#)[Arguments](#)[Pipelines](#)[Variables](#)[Examples](#)[Functions](#)[Associated templates](#)[Nested template definitions](#)

Package template implements data-driven templates for generating textual output.

To generate HTML output, see package [html/template](#), which has the same interface as this package but automatically secures HTML output against certain attacks.

Templates are executed by applying them to a data structure. Annotations in the template refer to elements of the data structure (typically a field of a struct or a key in a map) to control execution and derive values to be displayed. Execution of the template walks the structure and sets the cursor, represented by a period '.' and called "dot", to the value at the current location in the structure as execution proceeds.

The input text for a template is UTF-8-encoded text in any format. "Actions"--data evaluations or control structures--are delimited by "{{" and "}}"; all text outside actions is copied to the output unchanged.

Once parsed, a template may be executed safely in parallel, although if parallel executions share a `Writer` the output may be interleaved.

Here is a trivial example that prints "17 items are made of wool".

```
type Inventory struct {
    Material string
    Count     uint
}
sweaters := Inventory{"wool", 17}
tmpl, err := template.New("test").Parse("{{.Count}} items are made of {{.Material}}")
if err != nil { panic(err) }
err = tmpl.Execute(os.Stdout, sweaters)
if err != nil { panic(err) }
```

More intricate examples appear below.

Text and spaces

By default, all text between actions is copied verbatim when the template is executed. For example, the string " items are made of " in the example above appears on standard output when the program is run.

However, to aid in formatting template source code, if an action's left delimiter (by default "{{") is followed immediately by a minus sign and white space, all trailing white space is trimmed from the immediately preceding text. Similarly, if the right delimiter ("}") is preceded by white space and a minus sign, all leading white space is trimmed from the immediately following text. In these trim markers, the white space must be present: "{{- 3}}" is like "{{3}}" but trims the immediately preceding text, while "{{-3}}" parses as an action containing the number -3.

For instance, when executing the template whose source is

```
"{{23 -}} < {{- 45}}"
```

the generated output would be

```
"23<45"
```

For this trimming, the definition of white space characters is the same as in Go: space, horizontal tab, carriage return, and newline.

Actions

Here is the list of actions. "Arguments" and "pipelines" are evaluations of data, defined in detail in the corresponding sections that follow.

```
{{/* a comment */}}
{{- /* a comment with white space trimmed from preceding and following text */ -}}
A comment; discarded. May contain newlines.
Comments do not nest and must start and end at the
```

delimiters, as shown here.

`{{pipeline}}`

The default textual representation (the same as would be printed by `fmt.Print`) of the value of the pipeline is copied to the output.

`{{if pipeline}} T1 {{end}}`

If the value of the pipeline is empty, no output is generated; otherwise, `T1` is executed. The empty values are `false`, `0`, any `nil` pointer or interface value, and any array, slice, map, or string of length zero. `Dot` is unaffected.

`{{if pipeline}} T1 {{else}} T0 {{end}}`

If the value of the pipeline is empty, `T0` is executed; otherwise, `T1` is executed. `Dot` is unaffected.

`{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}`

To simplify the appearance of if-else chains, the else action of an if may include another if directly; the effect is exactly the same as writing

`{{if pipeline}} T1 {{else}}{{if pipeline}} T0 {{end}}{{end}}`

`{{range pipeline}} T1 {{end}}`

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, nothing is output; otherwise, `dot` is set to the successive elements of the array, slice, or map and `T1` is executed. If the value is a map and the keys are of basic type with a defined order, the elements will be visited in sorted key order.

`{{range pipeline}} T1 {{else}} T0 {{end}}`

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, `dot` is unaffected and `T0` is executed; otherwise, `dot` is set to the successive elements of the array, slice, or map and `T1` is executed.

`{{break}}`

The innermost `{{range pipeline}}` loop is ended early, stopping the current iteration and bypassing all remaining iterations.

`{{continue}}`

The current iteration of the innermost `{{range pipeline}}` loop is stopped, and the loop starts the next iteration.

`{{template "name"}}`

The template with the specified name is executed with `nil` data.

`{{template "name" pipeline}}`

The template with the specified name is executed with `dot` set to the value of the pipeline.

```
{{block "name" pipeline}} T1 {{end}}
```

A block is shorthand for defining a template

```
{{define "name"}} T1 {{end}}
```

and then executing it in place

```
{{template "name" pipeline}}
```

The typical use is to define a set of root templates that are then customized by redefining the block templates within.

```
{{with pipeline}} T1 {{end}}
```

If the value of the pipeline is empty, no output is generated; otherwise, dot is set to the value of the pipeline and T1 is executed.

```
{{with pipeline}} T1 {{else}} T0 {{end}}
```

If the value of the pipeline is empty, dot is unaffected and T0 is executed; otherwise, dot is set to the value of the pipeline and T1 is executed.

Arguments

An argument is a simple value, denoted by one of the following.

- A boolean, string, character, integer, floating-point, imaginary or complex constant in Go syntax. These behave like Go's untyped constants. Note that, as in Go, whether a large integer constant overflows when assigned or passed to a function can depend on whether the host machine's ints are 32 or 64 bits.
- The keyword nil, representing an untyped Go nil.
- The character '.' (period): . The result is the value of dot.
- A variable name, which is a (possibly empty) alphanumeric string preceded by a dollar sign, such as \$piOver2 or \$ The result is the value of the variable. Variables are described below.
- The name of a field of the data, which must be a struct, preceded by a period, such as .Field The result is the value of the field. Field invocations may be chained: .Field1.Field2 Fields can also be evaluated on variables, including chaining: \$x.Field1.Field2
- The name of a key of the data, which must be a map, preceded by a period, such as .Key The result is the map element value indexed by the key. Key invocations may be chained and combined with fields to any depth: .Field1.Key1.Field2.Key2 Although the key must be an alphanumeric identifier, unlike with field names they do not need to start with an upper case letter. Keys can also be evaluated on variables, including chaining: \$x.key1.key2
- The name of a niladic method of the data, preceded by a period, such as .Method The result is the value of invoking the method with dot as the receiver, dot.Method(). Such a method must have one return value (of any type) or two return values, the second of which is an error. If it has two and the returned error is non-nil, execution terminates and an error is returned to the caller as the value of Execute. Method invocations may be chained and combined with fields and keys to any depth: .Field1.Key1.Method1.Field2.Key2.Method2 Methods can also be evaluated on variables, including chaining: \$x.Method1.Field
- The name of a niladic function, such as fun The result is the value of invoking the function, fun(). The return types and values behave as in methods. Functions and function names are described below.
- A parenthesized instance of one the above, for grouping. The result may be accessed by a field or map key invocation. print (.F1 arg1) (.F2 arg2) (.StructValuedMethod "arg").Field

Arguments may evaluate to any type; if they are pointers the implementation automatically indirections to the base type when required. If an evaluation yields a function value, such as a function-valued field of a struct, the function is not invoked automatically, but it can be used as a truth value for an if action and the like. To invoke it, use the call function, defined below.

Pipelines

A pipeline is a possibly chained sequence of "commands". A command is a simple value (argument) or a function or method call, possibly with multiple arguments:

Argument

The result is the value of evaluating the argument.

.Method [Argument...]

The method can be alone or the last element of a chain but, unlike methods in the middle of a chain, it can take arguments.

The result is the value of calling the method with the arguments:

dot.Method(Argument1, etc.)

functionName [Argument...]

The result is the value of calling the function associated with the name:

function(Argument1, etc.)

Functions and function names are described below.

A pipeline may be "chained" by separating a sequence of commands with pipeline characters '|'. In a chained pipeline, the result of each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline.

The output of a command will be either one value or two values, the second of which has type error. If that second value is present and evaluates to non-nil, execution terminates and the error is returned to the caller of Execute.

Variables

A pipeline inside an action may initialize a variable to capture the result. The initialization has syntax

```
$variable := pipeline
```

where \$variable is the name of the variable. An action that declares a variable produces no output.

Variables previously declared can also be assigned, using the syntax

```
$variable = pipeline
```

If a "range" action initializes a variable, the variable is set to the successive elements of the iteration. Also, a "range" may declare two variables, separated by a comma:

```
range $index, $element := pipeline
```

in which case `$index` and `$element` are set to the successive values of the array/slice index or map key and element, respectively. Note that if there is only one variable, it is assigned the element; this is opposite to the convention in Go range clauses.

A variable's scope extends to the "end" action of the control structure ("if", "with", or "range") in which it is declared, or to the end of the template if there is no such control structure. A template invocation does not inherit variables from the point of its invocation.

When execution begins, `$` is set to the data argument passed to `Execute`, that is, to the starting value of `dot`.

Examples

Here are some example one-line templates demonstrating pipelines and variables. All produce the quoted word "output":

```
{{"\\"output\\""}}
    A string constant.
{{`"output"`}}
    A raw string constant.
{{printf "%q" "output"}}
    A function call.
{"output" | printf "%q"}
    A function call whose final argument comes from the previous
    command.
{{printf "%q" (print "out" "put")}}
    A parenthesized argument.
{"put" | printf "%s%s" "out" | printf "%q"}
    A more elaborate call.
{"output" | printf "%s" | printf "%q"}
    A longer chain.
{{with "output"}}{{printf "%q" .}}{{end}}
    A with action using dot.
{{with $x := "output" | printf "%q"}}{{$x}}{{end}}
    A with action that creates and uses a variable.
{{with $x := "output"}}{{printf "%q" $x}}{{end}}
    A with action that uses the variable in another action.
{{with $x := "output"}}{{$x | printf "%q"}}{{end}}
    The same, but pipelined.
```

Functions

During execution functions are found in two function maps: first in the template, then in the global function map. By default, no functions are defined in the template but the `Funcs` method can be used to add them.

Predefined global functions are named as follows.

and

Returns the boolean AND of its arguments by returning the first empty argument or the last argument. That is,

"and x y" behaves as "if x then y else x."

Evaluation proceeds through the arguments left to right and returns when the result is determined.

call

Returns the result of calling the first argument, which must be a function, with the remaining arguments as parameters. Thus "call .X.Y 1 2" is, in Go notation, dot.X.Y(1, 2) where Y is a func-valued field, map entry, or the like.

The first argument must be the result of an evaluation that yields a value of function type (as distinct from a predefined function such as print). The function must return either one or two result values, the second of which is of type error. If the arguments don't match the function or the returned error value is non-nil, execution stops.

html

Returns the escaped HTML equivalent of the textual representation of its arguments. This function is unavailable in html/template, with a few exceptions.

index

Returns the result of indexing its first argument by the following arguments. Thus "index x 1 2 3" is, in Go syntax, x[1][2][3]. Each indexed item must be a map, slice, or array.

slice

slice returns the result of slicing its first argument by the remaining arguments. Thus "slice x 1 2" is, in Go syntax, x[1:2], while "slice x" is x[:], "slice x 1" is x[1:], and "slice x 1 2 3" is x[1:2:3]. The first argument must be a string, slice, or array.

js

Returns the escaped JavaScript equivalent of the textual representation of its arguments.

len

Returns the integer length of its argument.

not

Returns the boolean negation of its single argument.

or

Returns the boolean OR of its arguments by returning the first non-empty argument or the last argument, that is, "or x y" behaves as "if x then x else y".

Evaluation proceeds through the arguments left to right and returns when the result is determined.

print

An alias for fmt.Sprint

printf

An alias for fmt.Sprintf

println

An alias for fmt.Sprintln

urlquery

Returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query. This function is unavailable in html/template, with a few exceptions.

The boolean functions take any zero value to be false and a non-zero value to be true.

There is also a set of binary comparison operators defined as functions:

```
eq
    Returns the boolean truth of arg1 == arg2
ne
    Returns the boolean truth of arg1 != arg2
lt
    Returns the boolean truth of arg1 < arg2
le
    Returns the boolean truth of arg1 <= arg2
gt
    Returns the boolean truth of arg1 > arg2
ge
    Returns the boolean truth of arg1 >= arg2
```

For simpler multi-way equality tests, `eq` (only) accepts two or more arguments and compares the second and subsequent to the first, returning in effect

```
arg1==arg2 || arg1==arg3 || arg1==arg4 ...
```

(Unlike with `||` in Go, however, `eq` is a function call and all the arguments will be evaluated.)

The comparison functions work on any values whose type Go defines as comparable. For basic types such as integers, the rules are relaxed: size and exact type are ignored, so any integer value, signed or unsigned, may be compared with any other integer value. (The arithmetic value is compared, not the bit pattern, so all negative integers are less than all unsigned integers.) However, as usual, one may not compare an `int` with a `float32` and so on.

Associated templates

Each template is named by a string specified when it is created. Also, each template is associated with zero or more other templates that it may invoke by name; such associations are transitive and form a name space of templates.

A template may use a template invocation to instantiate another associated template; see the explanation of the "template" action above. The name must be that of a template associated with the template that contains the invocation.

Nested template definitions

When parsing a template, another template may be defined and associated with the template being parsed. Template definitions must appear at the top level of the template, much like global variables in a Go program.

The syntax of such definitions is to surround each template declaration with a "define" and "end" action.

The define action names the template being created by providing a string constant. Here is a simple example:


```
{{define "T1"}}ONE{{end}}
{{define "T2"}}TWO{{end}}
{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}
{{template "T3"}}
```

This defines two templates, T1 and T2, and a third T3 that invokes the other two when it is executed. Finally it invokes T3. If executed this template will produce the text

```
ONE TWO
```

By construction, a template may reside in only one association. If it's necessary to have a template addressable from multiple associations, the template definition must be parsed multiple times to create distinct **Template* values, or must be copied with the `Clone` or `AddParseTree` method.

`Parse` may be called multiple times to assemble the various associated templates; see the `ParseFiles` and `ParseGlob` functions and methods for simple ways to parse related templates stored in files.

A template may be executed directly or through `ExecuteTemplate`, which executes an associated template identified by name. To invoke our example above, we might write,

```
err := tpl.Execute(os.Stdout, "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

or to invoke a particular template explicitly by name,

```
err := tpl.ExecuteTemplate(os.Stdout, "T2", "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

Index

```
func HTMLEscape(w io.Writer, b []byte)
func HTMLEscapeString(s string) string
func HTMLEscaper(args ...any) string
func IsTrue(val any) (truth, ok bool)
func JSEscape(w io.Writer, b []byte)
func JSEscapeString(s string) string
func JSEscaper(args ...any) string
func URLQueryEscaper(args ...any) string
type ExecError
    func (e ExecError) Error() string
    func (e ExecError) Unwrap() error
type FuncMap
type Template
    func Must(t *Template, err error) *Template
```

```
func New(name string) *Template
func ParseFS(fsys fs.FS, patterns ...string) (*Template, error)
func ParseFiles(filenames ...string) (*Template, error)
func ParseGlob(pattern string) (*Template, error)
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
func (t *Template) Clone() (*Template, error)
func (t *Template) DefinedTemplates() string
func (t *Template) Delims(left, right string) *Template
func (t *Template) Execute(wr io.Writer, data any) error
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data any) error
func (t *Template) Funcs(funcMap FuncMap) *Template
func (t *Template) Lookup(name string) *Template
func (t *Template) Name() string
func (t *Template) New(name string) *Template
func (t *Template) Option(opt ...string) *Template
func (t *Template) Parse(text string) (*Template, error)
func (t *Template) ParseFS(fsys fs.FS, patterns ...string) (*Template, error)
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
func (t *Template) ParseGlob(pattern string) (*Template, error)
func (t *Template) Templates() []*Template
```

Examples

[Template](#)

[Template \(Block\)](#)

[Template \(Func\)](#)

[Template \(Glob\)](#)

[Template \(Helpers\)](#)

[Template \(Share\)](#)

Constants

This section is empty.

Variables

This section is empty.

Functions

func [HTMLEscape](#)

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

func [HTMLEscapeString](#)

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

func **HTMLEscaper**

```
func HTMLEscaper(args ...any) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

func **IsTrue**

added in go1.6

```
func IsTrue(val any) (truth, ok bool)
```

IsTrue reports whether the value is 'true', in the sense of not the zero of its type, and whether the value has a meaningful truth value. This is the definition of truth used by if and other such actions.

func **JSEscape**

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

func **JSEscapeString**

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

func **JSEscaper**

```
func JSEscaper(args ...any) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

func **URLQueryEscaper**

```
func URLQueryEscaper(args ...any) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

Types

type **ExecError**

added in go1.6

```
type ExecError struct {  
    Name string // Name of template.
```

```
Err  error  // Pre-formatted error.
}
```

ExecError is the custom error type returned when Execute has an error evaluating its template. (If a write error occurs, the actual error is returned; it will not be of type ExecError.)

func (ExecError) Error

added in go1.6

```
func (e ExecError) Error() string
```

func (ExecError) Unwrap

added in go1.13

```
func (e ExecError) Unwrap() error
```

type FuncMap

```
type FuncMap map[string]any
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) return value evaluates to non-nil during execution, execution terminates and Execute returns that error.

Errors returned by Execute wrap the underlying error; call errors.As to uncover them.

When template execution invokes a function with an argument list, that list must be assignable to the function's parameter types. Functions meant to apply to arguments of arbitrary type can use parameters of type interface{} or of type reflect.Value. Similarly, functions meant to return a result of arbitrary type can return interface{} or reflect.Value.

type Template

```
type Template struct {
    *parse.Tree
    // contains filtered or unexported fields
}
```

Template is the representation of a parsed template. The *parse.Tree field is exported only for use by html/template and should be treated as unexported by all other clients.

► [Example](#)

► [Example \(Block\)](#)

► [Example \(Func\)](#)

► [Example \(Glob\)](#)

► [Example \(Helpers\)](#)

► [Example \(Share\)](#)

func Must

```
func Must(t *Template, err error) *Template
```

Must is a helper that wraps a call to a function returning (*Template, error) and panics if the error is non-nil. It is intended for use in variable initializations such as

```
var t = template.Must(template.New("name").Parse("text"))
```

func New

```
func New(name string) *Template
```

New allocates a new, undefined template with the given name.

func ParseFS

added in go1.16

```
func ParseFS(fsys fs.FS, patterns ...string) (*Template, error)
```

ParseFS is like ParseFiles or ParseGlob but reads from the file system fsys instead of the host operating system's file system. It accepts a list of glob patterns. (Note that most file names serve as glob patterns matching only themselves.)

func ParseFiles

```
func ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the base name and parsed contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results. For instance, ParseFiles("a/foo", "b/foo") stores "b/foo" as the template named "foo", while "a/foo" is unavailable.

func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

func (*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
```

AddParseTree associates the argument parse tree with the template t, giving it the specified name. If the template has not been defined, this tree becomes its definition. If it has been defined and already has that name, the existing definition is replaced; otherwise a new template is created, defined, and returned.

func (*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

func (*Template) DefinedTemplates

added in go1.5

```
func (t *Template) DefinedTemplates() string
```

DefinedTemplates returns a string listing the defined templates, prefixed by the string "; defined templates are: ". If there are none, it returns the empty string. For generating an error message here and in html/template.

func (*Template) Delims

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data any) error
```

Execute applies a parsed template to the specified data object, and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have

been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

If data is a `reflect.Value`, the template applies to the concrete value that the `reflect.Value` holds, as in `fmt.Print`.

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data any) error
```

`ExecuteTemplate` applies the template associated with `t` that has the given name to the specified data object and writes the output to `wr`. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel, although if parallel executions share a Writer the output may be interleaved.

func (*Template) Funcs

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

`Funcs` adds the elements of the argument map to the template's function map. It must be called before the template is parsed. It panics if a value in the map is not a function with appropriate return type or if the name cannot be used syntactically as a function in a template. It is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

func (*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

`Lookup` returns the template with the given name that is associated with `t`. It returns `nil` if there is no such template or the template has no definition.

func (*Template) Name

```
func (t *Template) Name() string
```

`Name` returns the name of the template.

func (*Template) New

```
func (t *Template) New(name string) *Template
```

`New` allocates a new, undefined template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a `{{template}}` action.

Because associated templates share underlying data, template construction cannot be done safely in parallel. Once the templates are constructed, they can be executed in parallel.

func (*Template) Option

added in go1.5

```
func (t *Template) Option(opt ...string) *Template
```

Option sets options for the template. Options are described by strings, either a simple string or "key=value". There can be at most one equals sign in an option string. If the option string is unrecognized or otherwise invalid, Option panics.

Known options:

missingkey: Control the behavior during execution if a map is indexed with a key that is not present in the map.

```
"missingkey=default" or "missingkey=invalid"
    The default behavior: Do nothing and continue execution.
    If printed, the result of the index operation is the string
    "<no value>".
"missingkey=zero"
    The operation returns the zero value for the map type's element.
"missingkey=error"
    Execution stops immediately with an error.
```

func (*Template) Parse

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse parses text as a template body for t. Named template definitions ({{define ...}} or {{block ...}} statements) in text define additional templates associated with t and are removed from the definition of t itself.

Templates can be redefined in successive calls to Parse. A template definition with a body containing only white space and comments is considered empty and will not replace an existing template's body. This allows using Parse to add new named template definitions without overwriting the main template body.

func (*Template) ParseFS

added in go1.16

```
func (t *Template) ParseFS(fsys fs.FS, patterns ...string) (*Template, error)
```

ParseFS is like ParseFiles or ParseGlob but reads from the file system fsys instead of the host operating system's file system. It accepts a list of glob patterns. (Note that most file names serve as glob patterns matching only themselves.)

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file. Since the templates created by ParseFiles are named by the base names of the argument files, t should usually

have the name of one of the (base) names of the files. If it does not, depending on t's contents before calling ParseFiles, t.Execute may fail. In that case use t.ExecuteTemplate to execute a valid template.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

func (*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseGlob parses the template definitions in the files identified by the pattern and associates the resulting templates with t. The files are matched according to the semantics of filepath.Match, and the pattern must match at least one file. ParseGlob is equivalent to calling t.ParseFiles with the list of files matched by the pattern.

When parsing multiple files with the same name in different directories, the last one mentioned will be the one that results.

func (*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates returns a slice of defined templates associated with t.



Source Files

[View all](#)

[doc.go](#)
[exec.go](#)

[funcs.go](#)
[helper.go](#)

[option.go](#)
[template.go](#)



Directories

[parse](#)

Package parse builds parse trees for templates as defined by text/template and html/template.

Why Go

[Use Cases](#)

[Case Studies](#)

Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

Packages

[Standard Library](#)

[About Go Packages](#)

About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

[Copyright](#)

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



[Google](#)