Go Template Basics Explained In Color

REVISION 1

Hawthorne-Press.com

Go Template Basics Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

Table of Contents

GO TEMPLATE BASICS EXPLAINED IN COLOR

INTRODUCTION

This paper will introduce the reader to the basic concepts of using ***text/templates.*** The outputs of example programs are to Standard Output. One of the major uses of templates to create text strings that contain variable information extracted from maps, arrays, structures, and slices. With a little of creative thought, you will be able to simplify many aspects of your work as a programmer. Template code can only extract data from structures that contain *Exportable Names* (i.e. the first letter capitalized).

The text structure that encapsulates template code is double curly brackets, also referred to as braces.

The first thing to understand about templates is they require three steps to create and use.

1. Create template and give it a name using *template.New().*
2. Parse the template using *template.Parse().*
3. Execute the template using *template.Execute()*

As a convenience, template creation and parsing can be accomplished in one-step by chaining the calls:

```
t, err := template.New("test").Parse("The contents of blank are \"{{.Data}}\"")
```

The creation and parsing phases builds a *parsing tree* in the background that is “walked” by the *template.Execute()* to output the results of the template. While this information is available via functions in ***text/template/parse***, the normal user should have no reason to delve that deeply into the somewhat hidden workings to the template package.

Template constructors come in two basic types: {{ }} and {{- -}}. The only difference is with the second form it will trim white space around the template result. The following code trims the white space after the close of the first template and the white space preceding the second template. This is a common template pattern!

```
tmpl, err = template.New("numb").Parse("value = {{23 -}} < {{- 45}}\n")
 err = tmpl.Execute(os.Stdout, nil)
```

Produces the following output:

value = 23<45

Notice that the second parameter of the **Execute** function is **nil**. It normally contains a map, array, slice, or structure that is supplying the data values used by the template. However, in this case, the contents of the template structures are all **constants,** and thus no data source was required by the Execute function.

In the following sections, we will discuss the various items that make up the *text/template* package. Since this a basic introduction, some aspects will be simplified and more complex interactions will be left to a later discussion.

Much of the informational text in this document is pulled unapologetically from the *text/template* package documentation and rewriting the package prose would only obscure to original author's intent.

Basic Template Elements

Actions

Within Action Statements, there may Pipelines. These represent evaluations of data. Action Statements provide various controls that conditionally select how and when data is processed. The following descriptions are from the template package documentation.

The following color-coding is present to help identify the various elements: Green for comments and text, blue for actions, and red for pipelines.

{{/*comment */}}
A comment; discarded. It May contain newlines. Comments do not nest and must start and end at the delimiters, as shown above.

{{pipeline}}
The default textual representation 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, zero, 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 ("comparable"), 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.

{{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" }}

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.

Pipelines

Within template Actions is a **Pipeline,** which is a collection Commands possibly "chained" together with the vertical bar character ("|"). This is an analog to the Unix/Linux pipelines. Processing is left to right and each command's output becomes the next command's *last argument*! Commands are **arguments, methods**, or **functions.** Let us discuss arguments first

Arguments

Arguments can be any of the following:

An argument can be a *boolean, string, character, integer, floating-point, imaginary* or *complex constant* in Go syntax. These behave like Go's untyped constants. The keyword **nil**, represents an untyped **Go nil** or a period.

A *variable name* that is alphanumeric string, possibly empty, proceeded by a dollar sign such as **$mymoney**.

An argument may be the value of a *field in a data structure*. The name of the field must be exportable, that is starting with a capital letter, such as **.Field**.

For *maps*, specifying the key returns the element value indexed by the key, such as **.Key**. Note that keys unlike fields do not have to start with a capital letter.

For niladic *methods*, i.e. methods without parameters, prefixing the method name with a period returns either one return value or two return values with the second value being an error. If the error is non-nil execution terminates and the error value is returned to the caller of the Execute Method.

The name of a niladic *functions*, such as **slant**, produces the value of invoking **slant().** The return types and values behave as methods.

The arguments described above generally describe all the available arguments. However, please read the package description of combinations that are more complex and argument chaining of methods, fields and keys to any depth.

METHODS

A Method can only take arguments if they are alone or the last element in a chain. Methods in the middle of a chain may not have arguments. **.Methods** in templates executes exportable methods defined on Go variables and structures.

This will be more obvious with an example program. Defined methods shown in red.

```
package main

import (                            // Import necessary packages
    "text/template"
    "log"
    "os"
)

type Myage struct {                 // A structure supplying an exportable value
    Age int
}

type Person string                  // A variable type to be used for creating methods

func (p Person) Label() string {    // Method Label: Returns a string from Person
    return "This is " + string(p)
}

func (a Myage) Ager(age int) int {  // Method Ager: Returns the Age from Myage
    return age
}

// Create and execute two types of templates:
//
func main() {
    tmpl, err := template.New("").Parse("{{.Label}}\n")
    if err != nil {
        log.Fatalf("Parse: %v", err)
    }

    tmpl.Execute(os.Stdout, Person("Bob"))

    t, err := template.New("").Parse("His age is now {{.Ager 69}}\n")
    err = t.Execute(os.Stdout, Myage{})
}
```

In the program we create two methods, **Label()** and **Ager(age int)** on different types on data. The Label method is defined on a string variable. While methods do not have to be defined on structures, they do have to have exportable names, i.e. their alphanumeric name must start with a capital letter. The Ager method is defined on a structure containing one int field, *Age*, and notice that it is an exportable field.

As you can see, arguments are space separated, not delimited with commas. This is the primary reason they must be either alone or the last method in a pipeline.

In the function **main()** we create two templates and execute each separately. Creating and using methods is straight forward, however as we will see Functions are a little different.

FUNCTIONS

Functions come in two flavors, *built-in* and *custom*. The template package, *text/template,* provide eighteen predefined functions. We will discuss custom templates later in this section.

Built-in Functions

The following descriptions are from the template package documentation.

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". All the arguments are evaluated.

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 do not 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.

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.

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". All the arguments are evaluated.

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.

The boolean function will take any zero value as false and a non-zero as true. The following set of binary comparison operators are functions, and since they are functions, their arguments are evaluated!

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

Multi-way Equality Comparison

or simpler multi-way equality tests, **eq** only accepts two or more arguments and compares the second and subsequent to the first, return in effect:

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

Custom Functions

Custom functions are implemented as standard Go Functions. However, how they are executed in the template environment is different. For example to be called from a template they have to installed with a function map using the template method *Funcs(funcmap).* The parameter has the type of *FuncMap*.

```
type FuncMap map[string]interface{}
```

The FuncMap description from the package document:

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.

In following sample program the function name “title” is mapped the standard string function **strings.Title**.

```
funcMap := template.FuncMap{
   "title": strings.Title,
}
```

Notice that unlike template methods, functions do not have to start with a capital letter.

Next, function arguments are listed after the function name is space separated fashion without parenthesis.

.

In the following sample program, the “title” function transforms any string argument string into camel case with the stings function Title. Thus, the text of the parameter *.Title* is chained to the “title” function.

```
package main

import (                              // Import necessary packages
    "os"
    "strings"
    "text/template"
)

func main() {

    tplVars := map[string]string{     // Parameter Map
        "Title":   "hello world",
        "Content": "hi there",
    }

    funcMap := template.FuncMap {     // Function Map
        "title": strings.Title,
    }

    // Create templates
    tpl := template.Must(template.New("main").Funcs(funcMap).Parse("{{.Title | title }} produces
{{.Content | title}}\n"))

    tpl.Execute(os.Stdout, tplVars)    // Execute Template with parameter map
}
```

The program produces:

Hello World produces Hi There

Compare the output with the parameter map where the lower case text is converted to Title Case, sometimes called “camel case”.

Template Operational Functions And Methods

Operational Methods are methods defined in the *text/Template* package itself. These control all aspects of template processing. These are not to be confused with custom or built-in methods.

Additionally, we will only discuss the primary methods and functions that a normal user will need. Please look at *text/package* documentation for other functions that may be used under certain circumstances.

Template Operational Functions

The following are the primary functions used to create and process templates.

Function: New(name string) *Template

This function creates a new undefined template with the name parameter string. This function must be used to create a new top-level template.

Function: 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 primarily intended to check for template parsing errors. The *Must()* code is shown in Green and parameter code is shown in red.

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

Function: ParseFiles(filenames ...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.

The succeeding template files must be included the first, primary, template to be functional. The text from the first template file is in blue, the code in red is also in the first file but it loads a subsequent template file.

```
        Title is {{.Title}}
        {{ template “footer.html” .}}
```

The dot after “footer.html” passes the data from *Execute()* through the footer template and the value becomes the dot operator in the included template.

Function: ParseGlob(pattern string) (*Template, error)

Uses a pattern, such as “*.tmpl”, to load files which is the equivalent to specifying all filenames in a *ParseFiles()* call. The first file found in the pattern becomes the primary template. The function is shown in red below.

```
        t = template.Must(template.New("name").ParseGlob("*.tmpl"))
```

Template Operational Methods

The following are the methods that are used to control template processing. The difference between operational functions described above and these methods is that they are associated with the specified template.

Method: (*Template) Execute(wr io.Writer, data interface{}) (err 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. Templates may be executed safely in parallel.

The supporting code is shown in blue, the *Execute()* method in red, and as usual comments and text are in green.

```
    tplVars := map[string]string{     // Parameter Map
        "Title":   "hello world",
    }
…
    // Create templates
    tpl := template.Must(template.New("main").Funcs(funcMap).Parse("The title is {{.Title }}\n"))

    tpl.Execute(os.Stdout, tplVars)    // Execute Template with parameter map
}
```

Method: (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) 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.

Example of this method is shown in *text/template* under "Examples(Helpers)". The excerpt below shows this method in red.

```
    err = templates.ExecuteTemplate(os.Stdout, "driver1", nil)
    if err != nil {
            log.Fatalf("driver1 execution: %s", err)
    }
```

Method: (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.

Method: (t *Template) Name() string

Name returns the name of the template.

Method: (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.

The following example shows two different ways of creating and using additional templates. It also demonstrates the usage of Lookup and Name Methods. The creation and loading of a secondary template is shown red. The text content associated with the secondary template is shown in orange.

```
package main

import (
    "text/template"
    "log"
    "os"
)

func main() {
    tplVars := map[string]string{ // Parameter Map
        "Title":   "hello world",
        "Content": "hi there",
    }
// Demonstration of creating master and a secondary templates
    tp := template.Must(template.New("Prime").Parse("Title = {{.Title}}\n{{template `next` .}}"))
    // Create a new template and parse the letter into it.
    _, err := tp.New("next").Parse("Content = {{.Content}}\n")    //Secondary Template "next"

// Demonstration of Lookup and Name Methods
    tlk := tp.Lookup("next")
    fmt.Println("Name from template lookup is ", tlk.Name(),"\n")
```

```
// Execute Master Template
    err = tp.Execute(os.Stdout, tplVars)
    if err != nil {
        log.Fatalf("Execute: %v", err)
    }
//Create the same result using Block Action
    cp := template.Must(template.New("Combo").Parse("Title = {{.Title}}\n{{block `content` . }} Content =
{{.Content}} {{end}}"))
// Execute
    err = cp.Execute(os.Stdout, tplVars)
    if err != nil {
        log.Fatalf("Excute: %v", err)
    }
}
```

Method: (*Template) Parse

Parse defines the template by parsing the text. Nested template definitions will be associated with the top-level template t. Parse may be called multiple times to parse definitions of templates to associate with **t**.

Method: (*Template) ParseFiles(filenames ...string) (*Template, error)

ParseFiles parses the named files and associates the resulting templates with **t.**

Method: (*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**.

Method: (t *Template) Templates() []*Template

Templates returns a slice of defined templates associated with **t**.

Template Examples

Now that we have looked at all the basic elements, we will put them together in a number of examples. The package documentation, while complete, is better understood with a concrete example.

This example is one of numerous examples shown in package documentation. This particular program shows a common situation, a form letter sent a number of people. The letter is merged with data from a structure slice containing the appropriate information on each recipient.

Form Letter Example

```
package main

import (
    "log"
    "os"
    "text/template"
)

func main() {
    const letter = `       // Define a template for Form Letter.
Dear {{.Name}},
{{if .Attended}}
It was a pleasure to see you at the wedding.
{{- else}}
```

```
It is a shame you couldn't make it to the wedding.
{{- end}}
{{with .Gift -}}
Thank you for the lovely {{.}}.
{{end}}
Best wishes,
Josie
`
    // Prepare some data to insert into the template.
    type Recipient struct {
        Name, Gift string
        Attended   bool
    }
    var recipients = []Recipient{
        {"Aunt Mildred", "bone china tea set", true},
        {"Uncle John", "moleskin pants", false},
        {"Cousin Rodney", "", false},
    }

    // Create a new template and parse the letter into it.
    t := template.Must(template.New("letter").Parse(letter))

    // Execute the template for each recipient.
    for _, r := range recipients {
        err := t.Execute(os.Stdout, r)
        if err != nil {
            log.Println("executing template:", err)
        }
    }

}
```

Letter Template Components

This template uses three different template components with both types of template bracketing:

1. Variable Name Insertions: **{{ .Name}}**
2. If – Else –End construction: **{{if pipeline}} T1 {{else}} To {{end}}**
3. The With –End construction: **{{with pipeline}} T1 {{end}}**

Also, notice that both types of template brackets are used to control white space in the parsed output.

Variable Name Insertion - {{.Name}}

One of the simplest template constructions is to insert variable names. In this case, the name of a user is inserted into the parsed output with:

```
        Dear {{.Name}},
```

The value of the exported field *Name* in the recipient structure replaces **.Name** in the template.

IF – Else –End Construction - {{if pipeline}} T1 {{else}} T0 {{end}}

The color code Action statement above correlates with the following code fragment.

```
{{if .Attended}}
It was a pleasure to see you at the wedding.
{{- else}}
It is a shame you couldn't make it to the wedding.
{{- end}}
```

The pipeline text, **.Attended**, is used to determine if the recipient actually attended the event. If this Boolean variable is true, T1 is used. Otherwise, the T0 line is parsed.

The template brackets, **"{{- "**, are used two remove all white space between this construction and next line to be parsed. If left out there would be a blank line between these after either T1 or T0 text. For Example:

```
It is a shame you couldn't make it to the wedding.
                                                    // Removed by the {{- template bracket
Thank you for the lovely moleskin pants.
```

With – End Construction – {{with pipeline}} T1 {{end}}

This construction is another conditional construction. If the pipeline is empty, no output is produced. It also temporarily changes to focus of the *Dot Operator,* **{{.}}**, to the variable location specified by **{{with pipeline}}**. The Dot location is now a variable within the structure, rather than the recipient structure itself.

Now the Dot Operator "is" the value since there is no field to specify and it is shown in blue.

```
{{with .Gift -}}
Thank you for the lovely {{.}}.
{{end}}
```

If we had used an *if* statement instead, the dot operator focus would remain with the current recipient structure and we would have to specify to the field wanted, as shown in blue.

```
{{if .Gift -}}
Thank you for the lovely {{.Gift}}.
{{end}}
```

Block Example

This program is another example available in template package documentation. In this case, a slice of items is to be displayed in two different ways. We define a master template and an overlay template that modifies the action of the master template. Below is the result of running the block example. The program itself follows:

```
Names:
- Gamora
- Groot
- Nebula
- Rocket
- Star-Lord
Names: Gamora, Groot, Nebula, Rocket, Star-Lord
```

```
package main

import (
```

```
        "log"
        "os"
        "strings"
        "text/template"
)

func main() {
        const (
                master  = `Names:{{block "list" .}}{{"\n"}}{{range .}}{{println "-" .}}{{end}}{{end}}`
                overlay = `{{define "list"}} {{join . ", "}}{{end}} `
        )
        var (
                funcs     = template.FuncMap{"join": strings.Join}
                guardians = []string{"Gamora", "Groot", "Nebula", "Rocket", "Star-Lord"}
        )
        masterTmpl, err := template.New("master").Funcs(funcs).Parse(master)
        if err != nil {
                log.Fatal(err)
        }
        overlayTmpl, err := template.Must(masterTmpl.Clone()).Parse(overlay)
        if err != nil {
                log.Fatal(err)
        }
        if err := masterTmpl.Execute(os.Stdout, guardians); err != nil {
                log.Fatal(err)
        }
        if err := overlayTmpl.Execute(os.Stdout, guardians); err != nil {
                log.Fatal(err)
        }
}
```

First, the easiest way to understand a complex example is deconstruct it. The block action is of the form:

Block – End Construction – {{block "name" pipeline}} T1 {{end}}

Given:

```
master  = `Names:{{block "list" .}}{{"\n"}}{{range .}}{{println "-" .}}{{end}}{{end}}
```

What is T1? Let us look at it with T1 shown in red.

```
master  = `Names:{{block "list" .}}{{"\n"}}{{range .}}{{println "-" .}}{{end}}{{end}}`
```

The {{"\n"}} insures the text Names: is on its own line for the master template. The range action steps thru the provided slice, guardians, and prints each name on its own line.

Overlay Construction and Processing

Now for the tricky part, what is the overlay template doing. The overlay template is redefining the T1 portion of the block "list" in the master template. This technique comes in very handy when you must modify the processing part of a template. In this case, it changes how the same data is presented.

The result of the cloning the master template and then parsing with the overlay template effectively changes the master template to the following:

```
overlay  = `Names:{{block "list" .}}{{join . ", "}}{{end}}`
```

Obliviously, this is a contrived example since it would have been just as easy to write two templates and execute against the same slice value. However, you should be able to see how this would be very useful for a large complex template that has multiple blocks of specialized processing.

Advice For Examining Other Examples

When examining the other examples in the template package documentation, or elsewhere for that matter, if you have trouble understanding a section, deconstruct it into is basic pieces as we did in the previous section. I find that using color to differentiate the various components often makes the complex much simpler to understand. That is one of the main purposes of this series of documents, making complex code easier to understand.

A Final Word

As stated elsewhere in this document, this is only the beginning of what can be accomplished with Go Templates. For example, ***html/template*** uses the same interface to construct HTML interfaces that are more secure. Additionally, even basic ***text/templates*** provide for escaping HTML and JAVASCRIPT text strings'

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com
10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
05/06/2016	C.E. Thornton	All	Initial Document
05/27/2018	C.E. Thornton	Copyright	Update Address