# Go 模板渲染

*24 Aug 2018* **@Morven's Life**

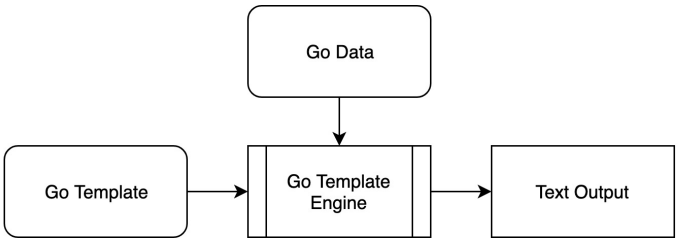
随着近几年 **Restful** 架构的盛行，前后端分离大行其道，模板渲染也由后端转移到了前端，后端只需要提供资源数据即可，这样导致类似于 **JSP**、**PHP** 等传统服务端模板脚本语言几乎问人问津了。但是在 **Go** 语言中，模板渲染技术不只局限于服务端标记语言（如 **HTML**）的渲染，**GO** 语言经常使用模版语言来处理譬如插入特定数据的文本转化等，虽然没有正则表达式那么灵活，但是渲染效率远优于正则表达式，而且使用起来也更简单。对于某些云计算的场景十分友好。今天，我们就来详细聊一聊 **Go** 语言模板渲染的技术细节。

## 运行机制

模板的渲染技术本质上都是一样的，一句话来说明就是**字符串模板和结构化数据**的结合，再详细地讲就是将定义好的模板应用于结构化的数据，使用注解语法引用数据结构中的元素（例如 **Struct** 中的特定字段，**Map** 中的键值）并显示它们的值。模板在执行过程中遍历数据结构并且设置当前光标（`.` 往往表示当前的作用域）标识当前位置的元素。

类似于 **Python** 语言的 **jinja**与 **NodeJS** 语言中的 **jade**等模版引擎，**Go** 语言模板引擎的运行机制也是类似：

1. 创建模板对象
2. 解析模板字串
3. 加载数据渲染模板




## Go 模板渲染核心包

**Go**语言提供了两个标准库用来处理模板渲染 `text/template` 和 `html/template` ，它们的接口几乎一摸一样，但处理的模板数据不同。其中 `text/template` 用来处理普通文本的模板渲染，而 `html/template` 专门用来渲染格式化 **HTML** 字符串。

下面的例子我们使用 `text/template` 来处理普通文本模板的渲染：

```
package main

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

```
type Student struct {
    ID      uint
    Name    string
}

func main() {
    stu := Student{0, "jason"}
    tmpl, err := template.New("test").Parse("The name for student {{.ID}} is {{.Name}}")
    if err != nil { panic(err) }
    err = tmpl.Execute(os.Stdout, stu)
    if err != nil { panic(err) }
}
```

上述代码第4行引入 `text/template` 来处理普通文本模板渲染，第14行定义一个模板对象 `test` 来解析变量 `"The name for student {{.ID}} is {{.Name}}"` 模板字符串，第16行使用定义好的结构化数据来渲染模版到标准输出。

> **Note:** 要引用的模板数据一定是 **export** 出来的，也就是说对应的字段必须以大写字母开头，比如例子中的 `Student` 结构体中的 `ID` 与 `Name` 字段。

我们再来看一个 **HTML** 字符串默板渲染的例子：

```
func templateHandler(w http.ResponseWriter, r *http.Request){
    tmpl := `<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"> <title>Go Template Dem
    </head>
    <body>
        {{ . }}
    </body>
</html>`

    t := template.New("hello.html")
    t, _ = t.Parse(tmpl)
    t.Execute(w, "Hello, Go Template!")
}
```

> **Note:** 在 **Go** 语言中不倾向于使用单引号来表示字符串，请根据需要使用双引号或反引号。另外，**Go** 语言的字符串类型在本质上就与其他语言的字符串类型不同。**Java** 中的 `String` 、**C++** 中的 `std::string` 以及 **python3** 中的 `str` 类型都只是定宽字符序列，而 **Go** 语言的字符串是一个用 **UTF-8** 编码的变宽字符序列，也就是说，它的每一个字符都用一个或多个字节表示。 **Go** 语言中的字符串字面量使用双引号或反引号(**"`"**)来创建：
>
> - 双引号用来创建可解析的字符串字面量 **(**支持转义，但不能用来引用多行**)**；

> - 反引号用来创建原生(raw)字符串字面量，这些字符串可能由多行组成(不支持任何转义序列)，原生的字符串字面量多用于书写多行消息、HTML 以及正则表达式。

本地部署执行：

```
$ curl -i http://127.0.0.1:8080/
HTTP/1.1 200 OK
Date: Fri, 09 Dec 2016 09:04:36 GMT
Content-Length: 223
Content-Type: text/html; charset=utf-8

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"> <title>Go Template Dem
    </head>
    <body>
        Hello, Go Template!
    </body>
</html>
```

Go 语言不仅可以直接解析模板字符串，也可以使用 **ParseFile** 解析模板文件，还是标准的处理流程：**创建-加载-渲染。**

# 模板命名

之前的例子模板对象是有名字的，可以在创建模板对象的时候显示命名，也可以让 Go 模版自动命名。但是如果涉及到嵌套模板的时候，该如何命名模板呢，这种情况下，模板文件有好多个！

Go 模版渲染包提供了 **ExecuteTemplate() 方法**，用于渲染执行指定名字的 Go 模板。比如加载 `hello` 模板的时候，可以指定 `layout.html` ：

```
tmplstr := `{{ define "stu_info" }}
The name for student {{.ID}} is {{.Name}}
{{ end }}
{{ define "stu_name" }}
Student name is {{.Name}}
{{ end }}
`
stu := Student{0, "jason"}
tmpl, err := template.New("test").Parse(tmplstr)
if err != nil { panic(err) }
err = tmpl.ExecuteTemplate(os.Stdout, "stu_info", stu)
if err != nil { panic(err) }
}
```

在模板字符串中，使用了 `define` 这个 action 定义了两个命名模版 `stu_info` 与 `stu_name` 。这时虽然 `Parse()` 方法返回的模板对象里面包含两个模板名，但是 `ExecuteTemplate()` 执行的模板还是 `stu_info` 。

不仅可以通过 `define` 定义模板，还可以通过 `template` 引入定义好的模板，类似 jinja 的 `include` 指令：

```
tmplstr := `{{ define "stu_name" }}
Student name is {{.Name}}
{{ end }}
{{ define "stu_info" }}
{{ template "stu_name" . }}
{{ end }}
`
stu := Student{0, "jason"}
tmpl, err := template.New("test").Parse(tmplstr)
if err != nil { panic(err) }
err = tmpl.ExecuteTemplate(os.Stdout, "stu_info", stu)
if err != nil { panic(err) }
}
```

上面的例子当中我们在 `stu_info` 模板中使用 `template` 引入了 `stu_name` 模板，同时传给 `stu_name` 模板当前作用域的数据( . )，第三个参数是可选的，如果为空，则表示传给嵌套模版的数据为 `nil` 。

总而言之，创建模板对象和加载多个模板文件，执行模板文件的时候需要指定基础模板（`stu_info`），在基础模板中可以引入其他命名的模板。这里的 `define` 和 `template` 等在双花括号中的关键字其实都是 **Go** 语言模版的指令。

# 模板指令

通过前面的介绍，我们知道模板其实是包含了一个或多个由双花括号包含的 `{{ }}` 模板字符串的文本或者文件。大部分模板字符串只是按字面值打印，但是如果模板字符串包含指令（**Action**）就会触发其它的行为。每个指令都包含了一个用模板语言书写的表达式，一个指令虽然简短但是可以输出复杂的打印值，模板语言包含通过选择结构体成员、调用函数或方法、表达式控制流 `if-else` 语句和 `range` 循环语句，还有其它实例化模板等诸多特性。

概括起来，**Go** 语言模版渲染的指令是用于动态执行一些逻辑和展示数据的形式，大致分为下面几类：

- 条件语句
- 迭代
- 封装
- 引用

我们在之前的例子中看到了 `define` 和 `template` 的用法，下面再看看其他的模版指令怎么使用：

## 条件判断

条件判断的语法很简单：

```
{{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}}
```

举个简单的例子：

```
func test1(a int) bool {
    if a == 3 {
        return true;
    }
    return false;
}

func main() {
  t := template.New("test");
  t.Funcs(template.FuncMap{"test1": test1});
  // {{if 表达式}}{{else if}}{{else}}{{end}}
  // if后面可以是一个条件表达式，可以是字符串或布尔值变量
  // 注意if后面不能直接使用==来判断
  t, _ = t.Parse(`
{{if 1}}
    it's true
{{else}}
  it's false
{{end}}

{{$a := 4}}
{{if $a|test1}}
  $a=3
{{else}}
  $a!=3
{{end}}
`);
  t.Execute(os.Stdout, nil);
  fmt.Println();
```

## 迭代

对于一些序列累的数据结构，如 **Slice**、**Map**，可以使用迭代指令来遍历各个值，与 **Go** 语言本身的迭代类似，使用 `range` 进行处理：

详细规范如下：

```
{{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.



{{ range . }}
    <li>{{ . }}</li>
{{ else }}
 empty
{{ end }}
```

当 `range` 的对象为空的时候，则会执行 `else` 分支中指定的逻辑。

## with 封装

`with` 语言在 **Python** 中可以开启一个上下文环境。对于 **Go** 模版来说， `with` 语句的功能类似，其含义就是创建一个封闭的作用域，在其范围内，可以使用 `.` 获取 `with` 指令指定的参数，而与外面的作用域无关，只与 `with` 的参数有关：

详细规范如下：

```
{{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.
```

举个例子来说：

```
{{ with arg }}
   .
{{ end }}
```

在上面 `with` 指令里面的 `.` 代表新开辟的作用域，而不是 `with` 指令外面的作用域。 `with` 指令中的 `.` 与其外面的 `.` 是两个不相关的对象。 `with` 指令也可以有 `else` ，其中 `else` 中的 `.` 则和 `with` 外面的 `.` 一样，毕竟只有 `with` 指令内才有封闭的上下文：

```
{{ with ""}}
 Now the dot is set to {{ . }}
{{ else }}
 {{ . }}
{{ end }}
```

## 嵌套模板

```
func test1() string {
  return "test1";
}

func main() {
  t := template.New("test");
  t.Funcs(template.FuncMap{"test1": test1});
  // {{ define "templateName" }}Template Content{{end}} --> define the template
  // {{ template "templateName" }} --> reference the template
  // {{ template "templateName" function }} --> Pass the return value of function to {{.}}
  t, _ = t.Parse(`
{{define "tp1"}} template one {{end}}
{{define "tp2"}} template two {{.}} {{end}}
{{define "tp3"}} {{template "tp1"}} {{template "tp2"}} {{end}}
{{template "tp1"}}
{{template "tp2" test1}}
{{template "tp3" test1}}
`);
  t.Execute(os.Stdout, nil);
  fmt.Println();
}
```