Go template高级用法、深入详解、手册、指南、剖析

发布于2020-08-20 17:44:04　　阅读 9.6K

以下为test.html文件的内容，里面使用了一个template语法 `{{.}}` 。

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>Go Web</title>
    </head>
    <body>
        {{ . }}
    </body>
</html>
```

以下是test.html同目录下的一个go web程序：

```
package main

import (
    "html/template"
    "net/http"
)

func tmpl(w http.ResponseWriter, r *http.Request) {
    t1, err := template.ParseFiles("test.html")
    if err != nil {
        panic(err)
    }
    t1.Execute(w, "hello world")
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/tmpl", tmpl)
    server.ListenAndServe()
}
```

前面的html文件中使用了一个template的语法 `{{.}}` ，这部分是需要通过go的template引擎进行解析，然后替换成对应的内容。

在go程序中，handler函数中使用 `template.ParseFiles("test.html")` ，它会自动创建一个模板(关联到变量t1上)，并解析一个或多个文本文件(不仅仅是html文件)，解析之后就可以使用 `Execute(w,"hello world")` 去执行解析后的模板对象，执行过程是合并、替换的过程。例如上面的 `{{.}}` 中的 `.` 会替换成当前对象"hello world"，并和其它纯字符串内容进行合并，最后写入w中，也就是发送到浏览器"hello world"。

本文不解释这些template包的函数、方法以及更底层的理论知识，本文只解释template的语法，如果觉得这些无法理解，或者看不懂官方手册，请看深入剖析Go template。

关于点"."和作用域

在写template的时候，会经常用到"."。比如 `{{.}}` 、 `{{len .}}` 、 `{{.Name}}` 、 `{{$x.Name}}` 等等。

在template中，点"."代表**当前作用域的当前对象**。它类似于java/c++的this关键字，类似于perl/python的self。如果了解perl，它更可以简单地理解为默认变量 `$_` 。

例如，前面示例test.html中 `{{.}}` ，这个点是顶级作用域范围内的，它代表 `Execute(w,"hello worold")` 的第二个参数"hello world"。也就是说它代表这个字符串对象。

再例如，有一个Person struct。

```
type Person struct {
    Name string
    Age  int
}

func main(){
    p := Person{"longshuai",}
    tmpl, _ := template.New("test").Parse("Name: {{.Name}}, Age: {{.Age}}")
    _ = tmpl.Execute(os.Stdout, p)
}
```

这里 `{{.Name}}` 和 `{{.Age}}` 中的点"."代表的是顶级作用域的对象p，所以Execute()方法执行的时候，会将 `{{.Name}}` 替换成 `p.Name` ，同理 `{{.Age}}` 替换成 `{{p.Age}}` 。

但是并非只有一个顶级作用域，range、with、if等内置action都有自己的本地作用域。它们的用法后文解释，这里仅引入它们的作用域来解释"."。

例如下面的例子，如果看不懂也没关系，只要从中理解"."即可。

```
package main

import (
    "os"
    "text/template"
)

type Friend struct {
    Fname string
}
```

```
type Person struct {
    UserName string
    Emails   []string
    Friends  []*Friend
}

func main() {
    f1 := Friend{Fname: "xiaofang"}
    f2 := Friend{Fname: "wugui"}
    t := template.New("test")
    t = template.Must(t.Parse(
`hello {{.UserName}}!
{{ range .Emails }}
an email {{ . }}
{{- end }}
{{ with .Friends }}
{{- range . }}
my friend name is {{.Fname}}
{{- end }}
{{ end }}`))
    p := Person{UserName: "longshuai",
        Emails:  []string{"a1@qq.com", "a2@gmail.com"},
        Friends: []*Friend{&f1, &f2}}
    t.Execute(os.Stdout, p)
}
```

输出结果：

```
hello longshuai!

an email a1@qq.com
an email a2@gmail.com

my friend name is xiaofang
my friend name is wugui
```

这里定义了一个Person结构，它有两个slice结构的字段。在Parse()方法中：

顶级作用域的 `{{.UserName}}` 、 `{{.Emails}}` 、 `{{.Friends}}` 中的点都代表Execute()的第二个参数，也就是Person对象p，它们在执行的时候会分别被替换成p.UserName、p.Emails、p.Friends。

因为Emails和Friend字段都是可迭代的，在 `{{range .Emails}}...{{end}}` 这一段结构内部 `an email {{.}}` ，这个"."代表的是range迭代时的每个元素对象，也就是p.Emails这个slice中的每个元素。

同理，with结构内部 `{{range .}}` 的"."代表的是p.Friends，也就是各个，再此range中又有一层迭代，此内层 `{{.Fname}}` 的点代表Friend结构的实例，分别是 `&f1` 和 `&f2` ，所以 `{{.Fname}}` 代表实例对象的Fname字段。

去除空白

template引擎在进行替换的时候，是完全按照文本格式进行替换的。除了需要评估和替换的地方，所有的行分隔符、空格等等空白都原样保留。所以，**对于要解析的内容，不要随意缩进、随意换行。**

可以在 `{{` 符号的后面加上短横线并保留一个或多个空格"- "来去除它前面的空白(包括换行符、制表符、空格等)，即 `{{- xxxx` 。

在 `}}` 的前面加上一个或多个空格以及一个短横线"-"来去除它后面的空白，即 `xxxx -}}` 。

例如：

```
{{23}} < {{45}}        -> 23 < {{23}} < {{- 45}}      ->  23 <{{23 -}} < {{45}}
```

其中 `{{23 -}}` 中的短横线去除了这个替换结构后面的空格，即 `}} <` 中间的空白。同理 `{{- 45}}` 的短横线去除了 `< {{` 中间的空白。

再看上一节的例子中：

```
t.Parse(
`hello {{.UserName}}!
{{ range .Emails }}
an email {{ . }}
{{- end }}
{{ with .Friends }}
{{- range . }}
my friend name is {{.Fname}}
{{- end }}
{{ end }}`)
```

注意，上面没有进行缩进。因为缩进的制表符或空格在替换的时候会保留。

第一行和第二行之间输出时会换行输出，不仅如此， `range {{.Emails}}` 自身也占一行，在替换的时候它会被保留为空行。除非range前面没加 `{{-` 。由于range的 `{{- end` 加上了去除前缀空白，所以每次迭代的时候，每个元素之间都换行输出但却不多一空行，如果这里的end去掉 `{{-` ，则每个迭代的元素之间输出的时候都会有空行。同理后面的with和range。

注释

注释方式： `{{/* a comment */}}` 。

注释后的内容不会被引擎进行替换。但需要注意，注释行在替换的时候也会占用行，所以应该去除前缀和后缀空白，否则会多一空行。

```
{{- /* a comment without prefix/suffix space */}}
{{/* a comment without prefix/suffix space */ -}}
{{- /* a comment without prefix/suffix space */ -}}
```

注意，应该只去除前缀或后缀空白，不要同时都去除，否则会破坏原有的格式。例如：

```
t.Parse(
`hello {{.UserName}}!
```

```
{{- /* this line is a comment */}}
{{ range .Emails }}
an email {{ . }}
{{- end }}
```

管道pipeline

pipeline是指产生数据的操作。比如 `{{.}}` 、 `{{.Name}}` 、 `funcname args` 等。

可以使用管道符号 `|` 链接多个命令，用法和unix下的管道类似： `|` 前面的命令将运算结果(或返回值)传递给后一个命令的最后一个位置。

例如：

```
{{.}} | printf "%s\n" "abcd"
```

`{{.}}` 的结果将传递给printf，且传递的参数位置是"abcd"之后。

命令可以有超过1个的返回值，这时第二个返回值必须为err类型。

需要注意的是，并非只有使用了 `|` 才是pipeline。Go template中，pipeline的概念是传递数据，只要能产生数据的，都是pipeline。这使得某些操作可以作为另一些操作内部的表达式先运行得到结果，就像是Unix下的命令替换一样。

例如，下面的 `(len "output")` 是pipeline，它整体先运行。

```
{{println (len "output")}}
```

下面是Pipeline的几种示例，它们都输出 `"output"` ：

```
{{`"output"`}}
{{printf "%q" "output"}}
{{"output" | printf "%q"}}
{{printf "%q" (print "out" "put")}}
{{"put" | printf "%s%s" "out" | printf "%q"}}
{{"output" | printf "%s" | printf "%q"}}
```

变量

可以在template中定义变量：

```
// 未定义过的变量
$var := pipeline

// 已定义过的变量
$var = pipeline
```

例如：

```
{{- $how_long :=(len "output")}}
{{- println $how_long}}   // 输出6
```

再例如：

```
tx := template.Must(template.New("hh").Parse(
`{{range $x := . -}}
{{$y := }}
{{- if (gt $x )}}{{println $x $y ($z := )}}{{- end}}
{{- end}}
`))
s := []int{, , , , }
_ = tx.Execute(os.Stdout, s)
```

输出结果：

上面的示例中，使用range迭代slice，每个元素都被赋值给变量 `$x` ，每次迭代过程中，都新设置一个变量 `$y` ，在内层嵌套的if结构中，可以使用这个两个外层的变量。在if的条件表达式中，使用了一个内置的比较函数gt，如果 `$x` 大于33，则为true。在println的参数中还定义了一个 `$z` ，之所以能定义，是因为 `($z := 444)` 的过程是一个Pipeline，可以先运行。

需要注意三点：

1. **变量有作用域，只要出现end，则当前层次的作用域结束。内层可以访问外层变量，但外层不能访问内层变量。**

2. **有一个特殊变量 `$` ，它代表模板的最顶级作用域对象(通俗地理解，是以模板为全局作用域的全局变量)，在Execute()执行的时候进行赋值，且一直不变。**例如上面的示例中， `$ = [11 22 33 44 55]` 。再例如，define定义了一个模板t1，则t1中的 `$` 作用域只属于这个t1。

3. **变量不可在模板之间继承。**普通变量可能比较容易理解，但对于特殊变量"."和"$"，比较容易搞混。见下面的例子。

例如：

```
func main() {
    t1 := template.New("test1")
    tmpl, _ := t1.Parse(
`
{{- define "T1"}}ONE {{println .}}{{end}}
{{- define "T2"}}{{template "T1" $}}{{end}}
{{- template "T2" . -}}
`)
    _ = tmpl.Execute(os.Stdout, "hello world")
}
```

上面使用define额外定义了T1和T2两个模板，T2中嵌套了T1。 `{{template "T2" .}}` 的点代表顶级作用域的"hello world"对象。在T2中使用了特殊变量 `$` ，这个 `$` 的范围是T2的，不会继承顶级作用域"hello world"。但因为执行T2的时候，传递的是"."，所以这里的 `$` 的值仍然是"hello world"。

不仅 `$` 不会在模板之间继承， `.` 也不会在模板之间继承(其它所有变量都不会继承)。实际上，template可以看作是一个函数，它的执行过程是 `template("T2",.)` 。如果把上面的 `$` 换成".", 结果是一样的。如果换成 `{{template "T2"}}` ，则 `$=nil`

如果看不懂这些，后文有解释。

条件判断

有以下几种if条件判断语句，其中第三和第四是等价的。

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

需要注意的是，pipeline为false的情况是各种数据对象的0值：数值0，指针或接口是nil，数组、slice、map或string则是len为0。

range...end迭代

有两种迭代表达式类型：

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

range可以迭代slice、数组、map或channel。迭代的时候，会设置"."为当前正在迭代的元素。

对于第一个表达式，当迭代对象的值为0值时，则range直接跳过，就像if一样。对于第二个表达式，则在迭代到0值时执行else语句。

```
tx := template.Must(template.New("hh").Parse(
`{{range $x := . -}}
{{println $x}}
{{- end}}
`))
s := []int{11, 22, 33, 44, 55}
_ = tx.Execute(os.Stdout, s)
```

需注意的是，range的参数部分是pipeline，所以在迭代的过程中是可以进行赋值的。但有两种赋值情况：

```
{{range $value := .}}
{{range $key,$value := .}}
```

如果range中只赋值给一个变量，则这个变量是当前正在迭代元素的值。如果赋值给两个变量，则第一个变量是索引值(map/slice是数值，map是key)，第二个变量是当前正在迭代元素的值。

下面是在html中使用range的一个示例。test.html文件内容如下：

```
<html>
    <head>
```

```
            <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
            <title>Go Web</title>
        </head>
        <body>
            <ul>
                {{ range . }}
                    <li>{{ . }}</li>
                {{ else }}
                    <li> Nothing to show </li>
                {{ end}}
            </ul>
        </body>
    </html>
```

以下是test.html同目录下的go程序文件：

```
    package main

    import (
        "html/template"
        "net/http"
    )

    func main() {
        server := http.Server{
            Addr: "127.0.0.1:8080",
        }
        http.HandleFunc("/process", process)
        server.ListenAndServe()
    }

    func process(w http.ResponseWriter, r *http.Request) {
        t1 := template.Must(template.ParseFiles("test.html"))
        s := []string{
            "星期一",
            "星期二",
            "星期三",
            "星期四",
            "星期五",
            "星期六",
            "星期日",}
        t1.Execute(w, s)
    }
```

with...end

with用来设置"."的值。两种格式：

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

对于第一种格式，当pipeline不为0值的时候，点"."设置为pipeline运算的值，否则跳过。对于第二种格式，当pipeline为0值时，执行else语句块，否则"."设置为pipeline运算的值，并执行T1。

例如：

```
{{with "xx"}}{{println .}}{{end}}
```

上面将输出 `xx` ，因为"."已经设置为"xx"。

内置函数和自定义函数

template定义了一些内置函数，也支持自定义函数。关于如何自定义函数，见深入剖析Go template。

以下是内置的函数列表：

```
and
    返回第一个为空的参数或最后一个参数。可以有任意多个参数。
    and x y等价于if x then y else x

not
    布尔取反。只能一个参数。

or
    返回第一个不为空的参数或最后一个参数。可以有任意多个参数。
    "or x y"等价于"if x then x else y"。

print
printf
println
    分别等价于fmt包中的Sprint、Sprintf、Sprintln

len
    返回参数的length。

index
    对可索引对象进行索引取值。第一个参数是索引对象，后面的参数是索引位。
    "index x 1 2 3"代表的是x[][][]。
    可索引对象包括map、slice、array。

call
    显式调用函数。第一个参数必须是函数类型，且不是template中的函数，而是外部函数。
    例如一个struct中的某个字段是func类型的。
    "call .X.Y 1 2"表示调用dot.X.Y(, )，Y必须是func类型，函数参数是和。
    函数必须只能有一个或个返回值，如果有第二个返回值，则必须为error类型。
```

除此之外，还内置一些用于比较的函数：

```
eq arg1 arg2:
    arg1 == arg2时为true
ne arg1 arg2:
    arg1 != arg2时为true
lt arg1 arg2:
    arg1 < arg2时为true
le arg1 arg2:
    arg1 <= arg2时为true
gt arg1 arg2:
    arg1 > arg2时为true
ge arg1 arg2:
    arg1 >= arg2时为true
```

对于eq函数，支持多个参数：

```
eq arg1 arg2 arg3 arg4...
```

它们都和第一个参数arg1进行比较。它等价于：

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

示例：

```
{{ if (gt $x ) }}{{println $x}}{{ end }}
```

嵌套template：define和template

define可以直接在待解析内容中定义一个模板，这个模板会加入到common结构组中，并关联到关联名称上。如果不理解，还是建议阅读深入剖析Go template。

定义了模板之后，可以使用template这个action来执行模板。template有两种格式：

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

第一种是直接执行名为name的template，点设置为nil。第二种是点"."设置为pipeline的值，并执行名为name的template。可以将template看作是函数：

```
template("name)
template("name",pipeline)
```

例如：

```
func main() {
    t1 := template.New("test1")
    tmpl, _ := t1.Parse(
`{{- define "T1"}}ONE {{println .}}{{end}}
{{- define "T2"}}TWO {{println .}}{{end}}
```

```
    {{- define "T3"}}{{template "T1"}}{{template "T2" "haha"}}{{end}}
    {{- template "T3" -}}
    `)
        _ = tmpl.Execute(os.Stdout, "hello world")
    }
```

输出结果：

```
ONE <nil>
TWO haha
```

上面定义了4个模板，一个是test1，另外三个是使用define来定义的T1、T2、T3，其中t1是test1模板的关联名称。T1、T2、T3和test1共享一个common结构。其中T3中包含了执行T1和T2的语句。最后只要 `{{template T3}}` 就可以执行T3，执行T3又会执行T1和T2。也就是实现了嵌套。此外，执行 `{{template "T1"}}` 时，点设置为nil，而 `{{temlate "T2" "haha"}}` 的点设置为了"haha"。

注意，**模板之间的变量是不会继承的。**

下面是html文件中嵌套模板的几个示例。

t1.html文件内容如下：

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
    <title>Go Web Programming</title>
</head>

<body>
    <div> This is t1.html before</div>
    <div>This is the value of the dot in t1.html - [{{ . }}]</div>
    <hr />
    {{ template "t2.html" }}
    <hr />
    <div> This is t1.html after</div>
</body>

</html>
```

因为内部有 `{{template "t2.html"}}` ，且此处没有使用define去定义名为"t2.html"的模板，所以需要加载解析名为t2.html的文件。t2.html文件内容如下：

```
<div style="">
    This is t2.html<br/>
    This is the value of the dot in t2.html - [{{ . }}]
</div>
```

处理这两个文件的handler函数如下：

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("t1.html", "t2.html")
    t.Execute(w, "Hello World!")
}
```

上面也可以不额外定义t2.html文件，而是直接在t1.html文件中使用define定义一个模板。修改t1.html文件如下：

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
    <title>Go Web Programming</title>
</head>

<body>
    <div> This is t1.html before</div>
    <div>This is the value of the dot in t1.html - [{{ . }}]</div>
    <hr />
    {{ template "t2.html" }}
    <hr />
    <div> This is t1.html after</div>
</body>

</html>

{{define "t2.html"}}
<div style="">
    This is t2.html<br/>
    This is the value of the dot in t2.html - [{{ . }}]
</div>
{{end}}
```

然后在handler中，只需解析t1.html一个文件即可。

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("t1.html")
    t.Execute(w, "Hello World!")
}
```

block块

```
{{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.
```

根据官方文档的解释：block等价于define定义一个名为name的模板，并在"有需要"的地方执行这个模板，执行时将"."设置为pipeline的值。

但应该注意，**block的第一个动作是执行名为name的模板，如果不存在，则在此处自动定义这个模板，并执行这个临时定义的模板。换句话说，block可以认为是设置一个默认模板。**

例如：

```
{{block "T1" .}} one {{end}}
```

它首先表示 `{{template "T1" .}}` ，也就是说先找到T1模板，如果T1存在，则执行找到的T1，如果没找到T1，则临时定义一个 `{{define "T1"}} one {{end}}` ，并执行它。

下面是正常情况下不使用block的示例。

home.html文件内容如下：

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>Go Web Programming</title>
    </head>
    <body>
        {{ template "content" }}
    </body>
</html>
```

在此文件中指定了要执行一个名为"content"的模板，但此文件中没有使用define定义该模板，所以需要在其它文件中定义名为content的模板。现在分别在两个文件中定义两个content模板：

red.html文件内容如下：

```
{{ define "content" }}
    <h1 style="color: red;">Hello World!</h1>
{{ end }}
```

blue.html文件内容如下：

```
{{ define "content" }}
    <h1 style="color: blue;">Hello World!</h1>
{{ end }}
```

在handler中，除了解析home.html，还根据需要解析red.html或blue.html：

```
func process(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    t := template.New("test")
    if rand.Intn() >  {
        t, _ = template.ParseFiles("home.html", "red.html")
    } else {
        t, _ = template.ParseFiles("home.html", "blue.html")
    }
    t.Execute(w,"")
}
```

如果使用block，那么可以设置默认的content模板。例如将原本定义在blue.html中的content设置为默认模板。

修改home.html：

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>Go Web Programming</title>
    </head>
    <body>
        {{ block "content" . }}
            <h1 style="color: blue;">Hello World!</h1>
        {{ end }}
    </body>
</html>
```

然后修改handler:

```
func process(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    t := template.New("test")
    if rand.Intn() >  {
        t, _ = template.ParseFiles("home.html", "red.html")
    } else {
        t, _ = template.ParseFiles("home.html")
    }
    t.Execute(w,"")
}
```

当执行else语句块的时候，发现home.html中要执行名为content的模板，但在ParseFiles()中并没有解析包含content模板的文件。于是执行block定义的content模板。而执行非else语句的时候，因为red.html中定义了content，会直接执行red.html中的content。

block通常设置在顶级的根文件中，例如上面的home.html中。

html/template的上下文感知

对于html/template包，有一个很好用的功能：上下文感知。text/template没有该功能。

上下文感知具体指的是根据所处环境css、js、html、url的path、url的query，自动进行不同格式的转义。

例如，一个handler函数的代码如下：

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("test.html")
    content := `I asked: <i>"What's up?"</i>`
    t.Execute(w, content)
}
```

上面content是Execute的第二个参数，它的内容是包含了特殊符号的字符串。

下面是test.html文件的内容：

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>Go Web Programming</title>
    </head>
    <body>
        <div>{{ . }}</div>
        <div><a href="/{{ . }}">Path</a></div>
        <div><a href="/?q={{ . }}">Query</a></div>
        <div><a onclick="f('{{ . }}')">Onclick</a></div>
    </body>
</html>
```

上面test.html中有4个不同的环境，分别是html环境、url的path环境、url的query环境以及js环境。虽然对象都是 `{{.}}` ，但解析执行后的值是不一样的。如果使用curl获取源代码，结果将如下：

```
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
</head>

<body>
    <div>I asked: &lt;i&gt;&#34;What&#39;s up?&#34;&lt;/i&gt;</div>
    <div>
        <a href="/I%20asked:%20%3ci%3e%22What%27s%20up?%22%3c/i%3e">
            Path
        </a>
    </div>
    <div>
        <a href="/?q=I%20asked%3a%20%3ci%3e%22What%27s%20up%3f%22%3c%2fi%3e">
            Query
        </a>
    </div>
    <div>
```

```
            <a onclick="f('I asked: \x3ci\x3e\x22What\x27s up?\x22\x3c\/i\x3e')">
                Onclick
            </a>
        </div>
    </body>

    </html>
```

不转义

上下文感知的自动转义能让程序更加安全，比如防止XSS攻击(例如在表单中输入带有 `<script>...</script>` 的内容并提交，会使得用户提交的这部分script被执行)。

如果确实不想转义，可以进行类型转换。

```
    type CSS
    type HTML
    type JS
    type URL
```

转换成指定个时候，字符都将是字面意义。

例如：

```
    func process(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("tmpl.html")
        t.Execute(w, template.HTML(r.FormValue("comment")))
    }
```

本文只介绍template的语法和用法，关于template包的函数、方法、template的结构和原理，见：深入剖析Go template。

高级用法示例，请参考：https://www.calhoun.io/intro-to-templates-p3-functions/