Go语言 标准库 Template包

html/template包实现了数据驱动的模板，用于生成可对抗代码注入的安全HTML输出。它提供了和text/template包相同的接口，Go语言中输出HTML的场景都应使用text/template包。

模板与渲染

在一些前后端不分离的Web架构中，我们通常需要在后端将一些数据渲染到HTML文档中，从而实现动态的网页（网页的布局和样式大致一样，但展示的内容并不一样）效果。

我们这里说的模板可以理解为事先定义好的HTML文档文件，模板渲染的作用机制可以简单理解为文本替换操作（使用相应的数据去替换HTML文档中事先准备好的标记）。

Go语言的模板引擎

Go语言内置了文本模板引擎text/template和用于HTML文档的html/template。它们的作用机制可以简单归纳如下：

1. 模板文件通常定义为.tmpl和.tpl为后缀（也可以使用其他的后缀），必须使用UTF8编码。
2. 模板文件中使用{{和}}包裹和标识需要传入的数据。
3. 传给模板这样的数据就可以通过点号（.）来访问，如果数据是复杂类型的数据，可以通过{{ .FieldName }}来访问它的字段。
4. 除{{和}}包裹的内容外，其他内容均不做修改原样输出。

Go语言模板引擎的使用可以分为三部分：定义模板文件、解析模板文件和模板渲染。

定义模板文件

我们按照Go模板语法定义一个`hello.tmpl`的模板文件，内容如下

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>Hello {{.}}</p>
</body>
</html>
```

解析和渲染模板文件

模板文件解析

上面定义好了模板文件之后，可以使用下面的常用方法去解析模板文件，得到模板对象：

```
func (t *Template) Parse(src string) (*Template, error)
func ParseFiles(filenames ...string) (*Template, error)
func ParseGlob(pattern string) (*Template, error)
```

三种方式都可以实现。

ParseFiles函数

当我们调用ParseFiles函数解析模板文件时，Go会创建一个新的模板，并将给定的模板文件的名字作为新模板的名字，如果该函数中传入了多个文件名，那么也只会返回一个模板，而且以第一个文件的文件名作为模板的名字，至于其他文件对应的模板则会被放到一个map中。让我们再来看一下HelloWorld中的代码：

```
t, _ := template.ParseFiles("hello.html")
```

以上代码相当于调用New函数创建一个新模板，然后再调用template的ParseFiles方法：

```
t := template.New("hello.html")
t, _ = t.ParseFiles("hello.html")
```

以上方法在解析模板时都没有对错误进行处理，Go提供了一个Must函数专门用来处理这个错误。Must函数可以包裹起一个函数，被包裹的函数会返回一个指向模板的指针和一个错误，如果错误不是nil，那么Must函数将产生一个panic。

Must函数代码：

```
t := template.Must(template.ParseFiles("hello.html"))
```

ParseGlob函数

通过该函数可以通过指定一个规则一次性传入多个模板文件，如：

```
t, _ := template.ParseGlob("*.html")
```

模板渲染

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{})
error
```

Execute方法

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

如果只有一个模板文件，调用这个方法总是可行的；但是如果有多个模板文件，调用这个方法只能得到第一个模板。

ExecuteTemplate方法（模板之间的嵌套 包含和定义）

```
t, _ := template.ParseFiles("hello.html", "hello2.html")
```

变量t就是一个包含了两个模板的模板集合，第一个模板的名字是hello.html,第二个模板的名字是hello2.html,如果直接调用Execute方法，则只有模板hello.html会被执行，如何想要执行模板hello2.html，则需要调用ExecuteTemplate方法

```
t.ExecuteTemplate(w, “hello2.html”, “我要在hello2.html中显示”)
```

接下来我们通过execute函数把数据放入到模板文件里面去。

然后我们创建一个`main.go`文件，在其中写下HTTP server端代码如下：

```
package main

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

func sayHello(w http.ResponseWriter, r *http.Request) {
    // 解析指定文件生成模板对象
    tmpl, err := template.ParseFiles("template/hello.tmpl")
    if err != nil {
        fmt.Println("加载模板文件失败! err:", err)
        return
    }
    // 利用给定数据渲染模板，并将结果写入w
    tmpl.Execute(w, "包子是个大帅哥! ")
}

func main() {
    http.HandleFunc("/", sayHello)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        fmt.Println("HTTP 服务启动失败! err:", err)
        return
    }
}
```

将上面的`main.go`文件编译执行，然后使用浏览器访问`http://127.0.0.1:9090`就能看到页面上显示了“Hello 包子是个大帅哥！”。 这就是一个最简单的模板渲染的示例。

模板语法

```
模板语法都包含在 {{和}} 中间，其中 {{.}} 中的点表示当前对象。
```

{{.}}，里面的这个点，就是传入的对象。例如：

```
name := "包子"
err = t.Execute(w,name)
```

这样子， 括号里面的点就表示张三。可以直接放入到模板文件里面去动态的显示。

当然，除了基本类型，还可以设置结构体，或者map。来看看结构体的

```
user := User{
    Name:"包子",
    Gender:"男",
    Age:18,
}
err = t.Execute(w,u1)
```

这样也可以，取数据的时候通过 `.Name` `.Age` `.Gender` 的形式进行读取数据即可。

注意要首字母大写哈，因为结构体要对外开放权限。

完整示例：

模板文件代码如下：

index.tmpl

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>包子信息:</p>
    <p>姓名： {{.user1.Name}}</p>
    <p>性别： {{.user1.Gender}}</p>
    <p>年龄： {{.user1.Age}}</p>
    <hr>
    <p>馒头信息:</p>
    <p>姓名： {{.user2.name}}</p>
    <p>性别： {{.user2.gender}}</p>
    <p>年龄： {{.user2.age}}</p>
</body>
</html>
```

HTTP server端代码如下：

main.go

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
```

```
)

type User struct {
    Name   string
    Gender string
    Age    int
}

func index(w http.ResponseWriter, r *http.Request) {
    // 解析模板
    t, err := template.ParseFiles("template/index.tmpl")
    if err != nil {
        fmt.Println("加载模板文件失败!err:", err)
        return
    }
    // 渲染模板
    user1 := User{
        Name:   "包子",
        Gender: "男",
        Age:    20,
    }
    user2 := map[string]interface{}{
        "name":   "馒头",
        "gender": "女",
        "age":    18,
    }
    err = t.Execute(w, map[string]interface{}{
        "user1": user1,
        "user2": user2,
    })
    if err != nil {
        fmt.Println("渲染模板失败! err:", err)
        return
    }
}

func main() {
    http.HandleFunc("/index", index)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        fmt.Println("HTTP 服务启动失败! err:", err)
        return
    }
}
```

我们可以看到，模板的引用也是非常的简单。值得一说的是要注意大小写，map里面的因为直接就是key/value的形式，所以不用首字母大写。

模板的注释

- {{/* 需要注释的内容 */}}
- 注释，执行时会忽略。可以多行。注释不能嵌套，并且必须紧贴分界符始止。

pipeline（传递）

pipeline是指产生数据的操作。比如{{.}}、{{.Name}}等。Go的模板语法中支持使用管道符号|链接多个命令，用法和unix下的管道类似：|前面的命令会将运算结果(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的模板语法中，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

我们还可以在模板中声明变量，用来保存传入模板的数据或其他语句生成的结果。具体语法如下：

```
// 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
```

移除空格

跟trim函数差不多，去除前后的空格

```
{{- .Name -}}
```

注意：- 要紧挨{{和}}，同时与模板值之间需要使用空格分隔。

条件判断

Go模板语法中的条件判断有以下几种:

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

range（遍历）

Go的模板语法中使用range关键字进行遍历，有以下两种写法，其中pipeline的值必须是数组、切片、字典或者通道。

```
{{range pipeline}} T1 {{end}}
如果pipeline的值其长度为0，不会有任何输出

{{range pipeline}} T1 {{else}} T0 {{end}}
如果pipeline的值其长度为0，则会执行T0。
```

示例:

```
示例1: .在end范围内表示每次迭代内的元素
{{range .}}
    <a href="#">{{.}}</a>
{{else}}
    没有遍历到任何内容
{{end}}
```

示例:

```
hobbyList := []string{
    "篮球",
    "足球",
    "双色球",
}

示例2：为range的遍历取别名
{{ range $idx, $hobby := .hobbyList}}
    <p>{{$idx}} - {{$hobby}}</p>
{{end}}
```

with

主体变量，我们可以通过with来省略。

```
{{with pipeline}} T1 {{end}}
如果pipeline为empty不产生输出，否则将dot设为pipeline的值并执行T1。不修改外面的dot。

{{with pipeline}} T1 {{else}} T0 {{end}}
如果pipeline为empty，不改变dot并执行T0，否则dot设为pipeline的值并执行T1。
```

示例：

```
<body>
    <p>包子信息:</p>
    {{with .user1}}
    <p>姓名： {{.Name}}</p>
    <p>性别： {{.Gender}}</p>
    <p>年龄： {{.Age}}</p>
    {{end}}
    <hr>
    <p>馒头信息:</p>
    {{with .user2}}
    <p>姓名： {{.name}}</p>
    <p>性别： {{.gender}}</p>
    <p>年龄： {{.age}}</p>
    {{end}}
</body>
```

嵌套template和define

我们可以在template中嵌套其他的template。这个template可以是单独的文件，也可以是通过`define`定义的template。

举个例子： `t.tmpl`文件内容如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>tmpl test</title>
</head>
<body>

    <h1>测试嵌套template语法</h1>
    <hr>
    {{template "ul.tmpl"}}
    <hr>
    {{template "ol.tmpl"}}
</body>
</html>

{{ define "ol.tmpl"}}
<ol>
    <li>吃饭</li>
    <li>睡觉</li>
    <li>打豆豆</li>
</ol>
{{end}}
```

`ul.tmpl`文件内容如下：

```
<ul>
    <li>注释</li>
    <li>日志</li>
    <li>测试</li>
</ul>
```

我们注册一个`templDemo`路由处理函数

```
http.HandleFunc("/tmpl", tmplDemo)
```

`tmplDemo`函数的具体内容如下：

```
func tmplDemo(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles("./t.tmpl", "./ul.tmpl")
    if err != nil {
        fmt.Println("加载模板文件失败！ err:", err)
        return
    }
    user := UserInfo{
        Name:   "包子",
        Gender: "男",
        Age:    18,
    }
    tmpl.Execute(w, user)
}
```

注意：在解析模板时，被嵌套的模板一定要在后面解析，例如上面的示例中`t.tmpl`模板中嵌套了`ul.tmpl`，所以`ul.tmpl`要在`t.tmpl`后进行解析。

block

作用： 定义一组模板，在自己的页面中，继承者一组模板，然后再重新定义其中的block块的内容。

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

`block`是定义模板`{{define "name"}} T1 {{end}}`和执行`{{template "name" pipeline}}`缩写，典型的用法是定义一组根模板，然后通过在其中重新定义块模板进行自定义。

定义一个根模板`templates/base.tmpl`，内容如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>Go Templates</title>
</head>
<body>
<div class="container-fluid">
    {{block "content" . }}{{end}}
</div>
</body>
</html>
```

然后定义一个`templates/index.tmpl`，”继承”`base.tmpl`：

```
{{template "base.tmpl"}}

{{define "content"}}
    <div>Hello world!</div>
{{end}}
```

然后使用`template.ParseGlob`按照正则匹配规则解析模板文件，然后通过`ExecuteTemplate`渲染指定的模板：

```
func index(w http.ResponseWriter, r *http.Request){
    tmpl, err := template.ParseGlob("templates/*.tmpl")
    if err != nil {
        fmt.Println("加载模板文件失败！ err:", err)
        return
    }
    err = tmpl.ExecuteTemplate(w, "index.tmpl", nil)
    if err != nil {
        fmt.Println("渲染模板失败！ err:", err)
        return
    }
}
```

如果我们的模板名称冲突了，例如不同业务线下都定义了一个index.tmpl模板，我们可以通过下面两种方法来解决。

1. 在模板文件开头使用{{define 模板名}}语句显式的为模板命名。
2. 可以把模板文件存放在templates文件夹下面的不同目录中，然后使用template.ParseGlob("templates/*/.tmpl")解析模板。

修改默认的标识符

Go标准库的模板引擎使用的花括号{{和}}作为标识，而许多前端框架（如Vue和 AngularJS）也使用{{和}}作为标识符，所以当我们同时使用Go语言模板引擎和以上前端框架时就会出现冲突，这个时候我们需要修改标识符，修改前端的或者修改Go语言的。这里演示如何修改Go语言模板引擎默认的标识符：

```
template.New("test").Delims("{[", "]}").ParseFiles("./t.tmpl")
```

预定义函数

执行模板时，函数从两个函数字典中查找：首先是模板函数字典，然后是全局函数字典。一般不在模板内定义函数，而是使用Funcs方法添加函数到模板里。

名称	含义
and	函数返回它的第一个empty参数或者最后一个参数； 就是说"and x y"等价于"if x then y else x"；所有参数都会执行；
or	返回第一个非empty参数或者最后一个参数； 亦即"or x y"等价于"if x then x else y"；所有参数都会执行；
not	返回它的单个参数的布尔值的否定
len	返回它的参数的整数类型长度
index	执行结果为第一个参数以剩下的参数为索引/键指向的值； 如"index x 1 2 3"返回x[1][2][3]的值；每个被索引的主体必须是数组、切片或者字典。
print	即fmt.Sprint , printf和println同理
html	返回与其参数的文本表示形式等效的转义HTML。 这个函数在html/template中不可用。
urlquery	以适合嵌入到网址查询中的形式返回其参数的文本表示的转义值。 这个函数在html/template中不可用。
js	返回与其参数的文本表示形式等效的转义JavaScript。
call	执行结果是调用第一个参数的返回值。 该参数必须是函数类型，其余参数作为调用该函数的参数； 如"call .X.Y 1 2"等价于go语言里的dot.X.Y(1, 2)； 其中Y是函数类型的字段或者字典的值，或者其他类似情况； call的第一个参数的执行结果必须是函数类型的值（和预定义函数如print明显不同）； 该函数类型值必须有1到2个返回值，如果有2个则后一个必须是error接口类型； 如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用模板执行者该错误；

比较函数

布尔函数会将任何类型的零值视为假，其余视为真。

下面是定义为函数的二元比较运算的集合：

名称	含义
eq	如果arg1 等于 arg2则返回真
ne	如果arg1 不等于 arg2则返回真
lt	如果arg1 小于 arg2则返回真
le	如果arg1 小于或等于 arg2则返回真
gt	如果arg1 大于 arg2则返回真
ge	如果arg1 大于或等于 arg2则返回真

为了简化多参数相等检测，eq（只有eq）可以接受2个或更多个参数，它会将第一个参数和其余参数依次比较，返回下式的结果：

```
{{eq arg1 arg2 arg3}}
```

比较函数只适用于基本类型（或重定义的基本类型，如"type Celsius float32"）。但是，整数和浮点数不能互相比较。

自定义函数

Go的模板支持自定义函数。

```
func sayHello(w http.ResponseWriter, r *http.Request) {
    htmlByte, err := ioutil.ReadFile("./hello.tmpl")
    if err != nil {
        fmt.Println("读取模板文件失败！ err:", err)
        return
    }
    // 自定义一个夸人的模板函数
    praise := func(arg string) (string, error) {
        return arg + "真帅", nil
    }
    // 采用链式操作在Parse之前调用Funcs添加自定义的kua函数
    tmpl, err := template.New("hello").Funcs(template.FuncMap{"praise":
praise}).Parse(string(htmlByte))
    if err != nil {
        fmt.Println("加载模板文件失败！ err:", err)
        return
    }

    user := UserInfo{
        Name:   "包子",
        Gender: "男",
        Age:    18,
    }
    // 使用user渲染模板，并将结果写入w
    tmpl.Execute(w, user)
}
```

我们可以在模板文件`hello.tmpl`中按照如下方式使用我们自定义的`praise`函数了。

```
{{praise .Name}}
```

text/template与html/template的区别

`text/template` 是将内容都已text文本格式返回。

`html/template` 针对的是需要返回HTML内容的场景，在模板渲染过程中会对一些有风险的内容进行转义，以此来防范跨站脚本攻击。

例如，我定义下面的模板文件：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    {{.}}
</body>
</html>
```

这个时候传入一段JS代码并使用 `html/template` 去渲染该文件，会在页面上显示出转义(转义成普通的字符串)后的JS内容。 `alert('包子你怎么这么帅！')` 这就是 `html/template` 为我们做的事。

但是在某些场景下，我们如果相信用户输入的内容，不想转义的话（即：原html的格式输出），可以自行编写一个safe函数，手动返回一个 `template.HTML` 类型的内容。示例如下：

```
func xss(w http.ResponseWriter, r *http.Request){
    // new一个template,再funcs功能里添加safe函数，再parseFiles文件
    tmpl,err := template.New("xss.tmpl").Funcs(template.FuncMap{
        "safe": func(s string)template.HTML {
            return template.HTML(s)
        },
    }).ParseFiles("./xss.tmpl")
    if err != nil {
        fmt.Println("加载模板文件失败！ err:", err)
        return
    }
    jsStr := `<script>alert('包子你怎么这么帅')</script>`
    err = tmpl.Execute(w, jsStr)
    if err != nil {
        fmt.Println(err)
    }
}
```

这样我们只需要在模板文件不需要转义的内容后面使用我们定义好的safe函数就可以了。

```
{{ . | safe }}
```