模板层

> Django中的HTML文件并不是一个简单的前端页面，他支持多种渲染方式；
>
> 比如`Smart`或是`Jinja`这样出名的模板语言引擎，默认django使用的是templates引擎来进行模板页面的渲染，这也被称为Django模板语言（DTL）

- 模板语言主要有**模板变量**和**模板标签**

```
<div> {{ account }} </div>
```

- 模板变量通过视图函数传递字典变量，字典的`key`值为对应模板变量名，对应`value`是模板变量实际被渲染的值

```
def index(request):
    content = {'account':'test'}
    return render(request,html,content)
```

- 最终渲染结果

```
<div> test </div>
```

> 模板引擎遇到这个变量，将会计算他，并且将结果覆盖；
>
> 如果视图函数中，并没有给这个模板变量赋值，也没有关系，模板会自动将这个`{{ account }}`处理为一个空

> **注意**：模板变量名，只能以字母数字下划线构成，下划线不可以打头，并且也不可以使用表单符号组成模板变量

变量获取方式

> 当我们返回的是一个字符串数据，这应该是最简单的处理方式，前端模板会直接将它展示到页面中。
>
> 如果返回的是一个列表序列数据或者字典键值对数据，我们可以使用模板语言中的`.`符号来进行其中值的获取，当模板变量中有符号`.`的存在时，比如`content.key`他会按照如下顺序来进行查找：

1. 字典查找：`content[key]`
2. 对应属性和方法查找：`content.key`
    - > 注意：因为对应key值的查找优先级要低于属性方法，所以要尽量避免使用数据内置方法作为key值
3. 序列索引方式查找：`content[index]`

Python数据在模板

> Python常用数据有int，float，str，list，tuple，dict，set等
>
> 把他们都通过模板变量传递到模板页面试试

```python
def index(request):
    int_ = 123
    float_ = 0.123
    str_ = 'str'
    list_ = ['l','i','s','t']
    tuple_ = ('t','u','p','l','e')
    dict_ = {"key":"value"}
    set_ = {'s','e','t'}
    return render(request,"index.html",locals())
```

- `locals()`：该函数将当前作用域下的所有变量名和对应变量值组成字典，免去了我们构造存储字典的麻烦

```
<p>{{ int_ }}</p>
<p>{{ float_ }}</p>
<p>{{ str_ }}</p>
<p>{{ list_ }}</p>
<p>{{ tuple_ }}</p>
<p>{{ dict_ }}</p> <p>{{ dict_.key }}</p>
<p>{{ set_ }}</p>
```

> 到了模板页面上，这些变量值其实也都像变成了字符串一样，直接展示

```
123

0.123

str

['l', 'i', 's', 't']

('t', 'u', 'p', 'l', 'e')

{'key': 'value'}

{'e', 's', 't'}
```

for 标签

> 使用模板中使用标签语言`{% for %}`和`{% endfor%}`，对视图函数传递的数据集进行遍历访问，比如上面传递的字符串，列表，元祖，字典，集合这样的数据
>
> 和普通模板变量不同，模板标签使用大括号百分号的组合`{% lag %}`，具有有一些特殊的功能性

> 模板中的标签`{% for %}`与Python中的for循环类似，要记得有闭合模板标签`{{ endfor }}`

```
{% for var in sequence %}
    {{ var }}
{% endfor %}
```

- 来把上面的数据进行访问

```
{% for var in str_ %}
    <p>{{ var }}</p>
{% endfor %}
---------------------
{% for var in list_ %}
    <p>{{ var }}</p>
{% endfor %}
---------------------
{% for var in tuple_ %}
    <p>{{ var }}</p>
{% endfor %}
---------------------
{% for var in set_ %}
    <p>{{ var }}</p>
{% endfor %}
```

> 看到的效果和在Python中迭代访问的结果是差不多的，并且模板循环还会使对应的标签也进行循环

> 接下来来看字典，通过模板循环从字典中取出来的是字典的key值

```
{% for var in dict_ %}
    <p>{{ var }} {{ dict_.var}}</p>
{% endfor %}
```

> 在for循环遍历访问字典的时候，不能再像Python语法里一样，直接通过迭代获取`key`之后通过
>
> `dict[key]`或是`dict.key`，拿到对应value；
>
> 模板变量不会把`var`解释成取到的对应`key`值，var只是作为了一个单纯的`var`字符串，除非在字典中，有`var`字符串做为字典的键值，否则是取不到的

- 正确的对字典中键值对进行获取的方式是通过内置字典`items`属性：

```
{% for key,value in dict_.items %}
    <p>{{ key }} {{ value}}</p>
{% endfor %}
```

> **注意**：模板语言中，不会出现索引超出范围的`IndexError`或者Key值不存在的`KeyError`，取不出任何东西则只是一个空

- 在`{% for %}`循环中，我们还可以使用很多有用的模板变量，方便我们控制循环

```
{% for var in iterable %}
    {{ forloop.counter }} <!--当前循环次数，从1开始计数 -->
    {{ forloop.counter0 }} <!--当前循环次数，从0开始计数 -->
    {{ forloop.revcounter }} <!--当前循环次数，从最大长度开始 -->
    {{ forloop.revcounter0 }} <!--当前循环次数，从最大索引开始 -->
    {{ forloop.first }} <!-- 判断是否为第一次循环 -->
    {{ forloop.last }} <!-- 判断是否为第一次循环 -->
    {{ forloop.parentloop }} <!-- 当循环嵌套时，访问上层循环 -->
{% endfor %}
```

- 通过`{% empty %}`标签判断迭代对象是否为空

```
{% for var in test_list %}
    {{ var }}
{% empty %}
    空空如也
{% endfor %}
```

if 标签

> 可以通过`{% if %}`标签语法来进行模板变量的值判断；
>
> 语法如下

```
{% if test_list %}
    列表不为空
{% elif test_dict %}
    列表为空，字典不为空
{% else %}
    列表字典均为空
{% endif %}
```

- 并且`if`标签还支持`and`、`or`及`not`来进行变量的布尔判断

```
{% if test_list and test_dict %}
    列表、字典均不为空
{% endif %}
```

```
{% if not test_list %}
    列表为空时才能满足IF条件判断
{% endif %}
```

```
{% if test_list or test_dict %}
    列表、字典某一个不为空
{% endif %}
```

```
{% if not test_list or test_dict %}
    列表为空或字典不为空
{% endif %}
```

```
{% if test_list and not test_dict %}
    列表不为空并且字典为空
{% endif %}
```

- 也支持同时使用and及or语句，但是and的条件判断优先级要高于or语句

```
{% if a or b and c %}
等同于
{% if a or (b and c) %}
```

- `if`标签还支持`==`、`!=`、`>`、`<`、`>=`、`<=`的判断用法

```
{% if var == "1" %}
    这个值是"1"
{% endif %}
```

```
{% if var != "x" %}
    判断不相等成立
{% endif %}
```

```
{% if var < 100 %}
    var大于100
{% endif %}
```

```
{% if var >= 100 %}
    var大于100
{% endif %}
```

> 如果判断的数据类型在后台传递到模板变量时具体数值类型为整型或浮点型而不是字符串;
>
> **不需要在判断的时候加字符串的标识引号**

- 在模板语言中，不支持连续判断

```
{% if 100 > var > 50 %}
    var大于50小于100
{% endif %}
```

> 应该使用and语句写成这样

```
{% if 100 > var and var > 50 %}
    var大于50小于100
{% endif %}
```

- 除去运算符之外，`if`标签还支持`in`和`not in`的判断运算

```
{% if 1 in test_list %}
    列表中有数字1
{% endif %}
```

其他常用标签

> 其实`django`官方提供了不只`if`和`for`这样的模板标签，还提供了很多可以让我们在模板页面上实现之前只能在后台进行逻辑实现的功能标签，比如以下：

comment 标签

> `comment`标签常用来注释，在`{% comment %}`和`{% endcomment %}`中间的部分内容会被忽略；
>
> 这个标签不能嵌套使用

```
{% comment %}
    这里的内容会被忽略，相当于注释起来。
{% endcomment %}
```

autoescape 标签

> 默认情况下，为了安全起见，模板在接收到一个`HTML`标签或者`css`样式等具有实际意义的变量字符串时；
>
> 会对他进行转义，不会将这个字符串处理为HTML中实际的标签。一个`<h1>`标签到最后会被处理成：`<h1>`，这样浏览器就不会把他解释成一个标签的样式了

- 转义规则

符号	转义规则
<	`<`
>	`>`
'（单引号）	`&#39;`
"（双引号）	`&quot;`
&	`&`

> 那么有的时候，我们可能需要这样类似的HTML标签真正效果展示出来，
>
> 比如一个`<h1>`标签我们希望他真正展示出`h1`的样子，而不是一个朴素的`<h1>`字符串，那么就需要我们使用autoescape标签来进行防止转义处理

```
str_ = "<h1>这是H1标签</h1>"
```

```
{% autoescape on %}
    {{ str_ }}
{% endautoescape %}
{% autoescape off %}
    {{ str_ }}
{% endautoescape %}
```

cycle 标签

> `cycle`标签提供一些可迭代数据；
>
> 它的结构像是一个环，其中的数据通过空格分割，你可以使用任意数量的值，作为接下里每一次循环迭代的数据

> 其他包含在单引号 `'` 或者双引号 `"` 中的值被认为是可迭代字符串，如果没有被字符串引号包围的值被当作模版变量

```
list_ = ['l','i','s','t'] # 视图定义的模板变量
```

```
<style>
    .red{
        color:red;
    }
    .blue{
        color:blue;
    }
</style>

{% for var in list_ %}
    <p class="{% cycle 'red' 'blue' %}">
        {{ var }}
    </p>
{% endfor %}
```

> 很简单的就可以通过 `cycle` 标签进行循环中的样式切换啦
>
> 循环遍历出来的列表中每一个字都是换着颜色展示

> 某些时候，我们可能希望在使用一次 `cycle` 之后，接下来使用不是继续向后迭代取值，而是继续沿用这一次取到的值；
>
> 那么我们可以通过 `as` 语法给 `cycle` 标签取别名，在接下来需要沿用的地方直接使用别名作为模板变量即可，比如这样

```
{% for var in list_ %}
    <p>
        {% cycle 'red' 'blue' as style %}
    </p>
    <span>{{ style }}</span>
    <span>{{ style }}</span>
{% endfor %}
```

- 每一次循环取到的cycle其中的值，都可以通过别名style在这次循环区域中重复利用而不递进

> 但是我们发现个问题，`as` 语句本身在使用时也会造成对 `cycle` 中的数据进行一次取值，那有什么办法可以在第一次 `as` 语句出现时，我只做声明，而不是为了取值，`django` 模板中提供了一个叫 `silent` 的属性，可以用来 `as` 命名时不进行取值动作

```
{% cycle 'red' 'blue' as style silent %}
```

```
{% cycle 'red' 'blue' as style silent %}
<h1>{{ style }}</h1>
<h1>{{ style }}</h1>
{% cycle style %}
<h1>{{ style }}</h1>
{% cycle style %}
<h1>{{ style }}</h1>
<h1>{{ style }}</h1>
```

> 通过`silent`可以在初次定义时不进行取值，
>
> 接下来使用`cycle`所创建的迭代器，每次访问得到当前值，不会继续向后迭代；
>
> 如果希望取到下一个值，可以使用迭代器标签`{% style %}`，这次访问不会生产数据，但是会让迭代访问位置向后推进一次，在接下来`{{ style }}`使用将得到下一个值

```
red
red
blue
red
red
```

ifchanged 标签

> `ifchanged`标签用在`for`标签中；
>
> 检测这一次迭代的值和上一次迭代的值是否有改变，可以搭配`else`标签使用，用来确定是没有改变；
>
> 检查标签`{% ifchanged %}`和`{% endifchanged %}`之间的数据在每一次迭代过程中是否发生改变

```
list_ = [1,1,1,2,3]
```

```
{% for var in list_ %}
    {% ifchanged %}
        {{ var }}
    {% else %}
        数据未发生变化
    {% endifchanged %}
    <br>
{% endfor %}
```

```
1
数据未发生变化
数据未发生变化
2
3
```

firstof 标签

> `firstof`标签用来查找到标签内变量中第一个为`True`的参数并输出，如果标签内变量均为`False`，那么输出空

```
{% firstof 0 0 0 0 0 "哈哈哈" %}
```

```
哈哈哈
```

ifequal 标签

> `ifequal`标签接收两个变量，用来判断这两个值是否相等，如果相等，展示对应内容

```
{% ifequal 1 "1" %}
    <h3>两个值相等</h3>
{% else %}
    <h3>两个值不等</h3>
{% endifequal %}
<br>
```

- 这都不用想的，俩类型都不一样

ifnotequal 标签

> 上面的反义

now 标签

> 显示日期或时间，标签必须一个参数，用来指定当前时间日期的描述方式；
>
> 输出最终格式与项目时区及语言设置有关
>
> 参数是一个描述字符串，比较多，记着常用的就行

秒	
u	微妙，000000-999999
s	秒，00-59
Z	时区偏移量（UTC），单位为秒。-43200到43200
U	自Unix 时间以来的秒数。1970年1月1日00:00:00 UTC

分钟	
i	分钟，00-59

小时	
g	12小时格式，1-12'
G	24小时格式，00-23
h	12小时格式，00-12
H	24小时格式，00-23
O	时区差值，单位小时，如：Asia/Shanghai时区：+0800
P	当前几时几分，如：5:30 pm
f	当前几时几分，不包含上下午标示，如：5:30
a	小写字母：a.m.、p.m.
A	大写字母：AM、PM

月份	
b	月份英文字母的前三个表示，均小写。如："aug"
d	这个月的第几天，01-31
j	这个月的第几天，1-31
E	当前时区月份英文单词全拼
F	当前月份英文单词全拼
m	第几月，01-12
M	月份英文字母的前三个表示，首字母大写。如："Aug"
n	第几月，1-12
N	美联社月份缩写，如：'Jan.'，'Feb.'，'March'，'May'

星期	
D	星期几的英文单词前三个表示，首字母大写，如：'Fri'
l	星期几的英文单词全拼，如：'Friday'
w	星期几的数字，0（星期日）-6（星期六）
W	ISO-8601周数，今年第几周

年	
L	Bool值判断是否为闰年
y	第几年两位数字，如：2018年，返回18
Y	第几年，目前是：2018
Z	今天是今年的第几天

其他	
c	ISO 8601时间格式
e	当前时区名称：CST（美国-6:00，澳大利亚+9:30，中国+8:00，古巴-4:00）
r	RFC 5322格式化日期
S	一个月的第几天的英文序数后缀：'st'，'nd'，'rd'或'th'
t	当前月份的天数：28-31
T	当前时区

- 选出你想表达的时间，比如年月日，那么可以这样

```
{% now "Y-m-d" %}
```

```
2019-03-26
```

- 除了以上自己组合的格式化字符，还有一些已经预定义好的字符串

字符串	对应格式 时间
DATE_FORMAT	'N j, Y' Feb. 5, 2018
DATETIME_FORMAT	'N j, Y, P' Feb. 4, 2013, 4 p.m.
SHORT_DATE_FORMAT	'm/d/Y' 12/31/2015
SHORT_DATETIME_FORMAT	'm/d/Y P' 12/31/2019 11:59 p.m.

```
<ul>
    <li>DATE_FORMAT: {% now "DATE_FORMAT" %}</li>
    <li>DATETIME_FORMAT: {% now "DATETIME_FORMAT" %}</li>
    <li>SHORT_DATE_FORMAT: {% now "SHORT_DATE_FORMAT" %}</li>
    <li>SHORT_DATETIME_FORMAT: {% now "SHORT_DATETIME_FORMAT" %}</li>
</ul>
```

```
DATE_FORMAT: March 26, 2019
DATETIME_FORMAT: March 26, 2019, 3:45 a.m.
SHORT_DATE_FORMAT: 03/26/2019
SHORT_DATETIME_FORMAT: 03/26/2019 3:45 a.m.
```

> 最后`now`标签也支持`as`的用法，可以方便我们在模板中使用一个已经格式化好的输出

```
{% now "Y-m-d H:i:s" as show_time %}
<p>{{ show_time }}</p>
```

- 发现时间输出不正确，记得查看`settings`文件下的`TIME_ZONE`配置

过滤器

> 除了模板标签可以帮助我们对数据或者进行逻辑处理
>
> `django`中还提供了一款工具叫做过滤器，过滤器也可以实现一些模板变量的运算，判断或是其他逻辑处理

add

> 语法：`{{ var1|add:var2 }}`

> `add`过滤器可以实现`var1`与`var2`的相加，并且在遇到其他相同数据类型的，比如列表时，加号还可以重载为拼接功能

> 过滤器首先会将数据转换成Int类型，进行相加，如果转换失败，则会尝试使用`Python`中的数据类型
>
> 列表、元祖等这样的数据类型来进行转换，并且执行对应类型的加法
>
> 如果都转换失败，那么结果为一个空字符串

```
<p>add :{{ value|add:10 }}</p>
<p>add :{{ list_1|add:list_2 }}</p>
```

capfirst

> 语法：`{{ var|capfirst }}`

> 将变量第一个字母变为大写，如果第一个字符不是字母，过滤器不生效

```
<p>capfirst:{{ "Abc"|capfirst }}</p>
<p>capfirst:{{ "1abc"|capfirst }}</p>
```

center

> 语法：`{{ value|center:"length" }}`

> 使`value`在给定的`length`范围内居中

```
<p>center: {{ "abc"|center:"10" }}</p>
```

cut

> 语法：`{{ value|cut:"str" }}`

> 在`value`中移除所有`str`

```
<p>cut: {{ "a*b*c"|cut:"*" }}</p>
```

date

> 语法：`{{ value|date:SHORT_DATE_FORMAT" }}`

> 与`{% now %}`标签所使用格式字符一致；`value`为一个`datetime`对象
>
> 输出最终格式与项目时区及语言设置有关

```
import datetime
datetime = datetime.datetime.now()
```

```
<p>date: {{ datetime|date:"H:i" }}</p>
<p>date: {{ datetime|date:"Y/m/d" }}</p>
```

default

> 语法：`{{ value|default:"默认值" }}`

> 如果`value`值为假，则取"默认值"，反之返回`value`

```
<p>default: {{ 0|default:"这是展示的默认值" }}</p>
```

> 非空非0为真，0或空为假

default_if_none

> 语法：`{{ value|default_if_none:"默认值" }}`

> 如果`value`值为`None`，则取"默认值"，反之返回`value`

```
<p>default_if_none: {{ None|default_if_none:"value值为None" }}</p>
<p>default_if_none: {{ 0|default_if_none:"aaaa" }}</p>
```

dictsort

> 语法：`{{ value|dictsort:"attr" }}`

> `value`为字典列表数据，列表中数据均为类字典数据：`[ {1:'a'}, {2:'b'}, ]`
>
> 根据给定`attr`值进行排序，一般是**从小到大**的顺序

```
sort_list_dict = [
    {'name': '小绿', 'department': 'Development', 'age': 32},
    {'name': '小红', 'department': 'Leader', 'age': 21},
    {'name': '小飞', 'department': 'Test', 'age': 18},
    {'name': '小落', 'department': 'Development', 'age': 15},
    {'name': '大胖', 'department': 'Leader', 'age': 43}
]
```

```
<p>dictsort: </p>
{% for var in sort_list_dict|dictsort:"age" %}
    <p>
        {{ var.name }}
    </p>
{% endfor %}
```

dictsortreversed

> 语法：`{{ value|dictsortreversed:"attr" }}`

> 与`dictsort`功能相同，但是排序方式与`dictsort`相反，从大到小

divisibleby

> 语法：`{{ value|divisibleby:num }}`

> 如果给定的`value`可以被`num`整除，返回`True`；反之，返回`False`
>
> 常用来做整除判断

```
<p>divisibleby: {{ 8|divisibleby:2 }}</p>
```

escape

> 语法：`{{ value|escape }}`

> 将value值转义输出；
>
> 可以在取消转义autoescape标签下，选择性的打开某些需要转义的数据

```
{% autoescape off %}
    {{ str_|escape }}
    {{ str_ }}
{% endautoescape %}
```

safe

> 语法：`{{ value|safe }}`

> 取消转义，与`{% autoescape off %}`标签意义相同

```
<p>{{ str_|safe }}</p>
```

safeseq

> 语法：`{{ value|safeseq }}`

> 处理一个包含标签字符串的列表数据，简单的`safe`是不行的，因为`safe`过滤器会把内容先整体处理为字符串；而不是依次过滤序列中的数据，而`safeseq`过滤器则会依次处理序列中的每一个数据

```
list_ = [
    "<h1>第一个</h1>",
    "<h2>第二个</h2>",
    "<h3>第三个</h3>",
]
```

```
{{ list_|safe|join:"" }}
<br>
------------------
<br>
{{ list_|safeseq|join:"" }}
```

filesizeformat

> 语法：`{{ value|filesizeformat }}`

> 格式化value值为人类可读的计算机存储单位。如：1 bytes、1.2 MB；
>
> 如果不是一个可以处理的数值类型，返回0。
>
> 最小单位为byte

```
<p>filesizeformat : {{ "1"|filesizeformat }}</p>
<p>filesizeformat : {{ "3758331"|filesizeformat }}</p>
```

first

> 语法：`{{ value|first }}`

> 返回序列数据`value`中的第一项

```
<p>first : {{ "abc"|first }}</p>
```

last

> 语法：`{{ value|last }}`

> 返回序列数据`value`中的最后一项

```
<p>last : {{ "abc"|last }}</p>
```

floatformat

> 语法：`{{ value|floatformat:"精度" }}`

> 设置浮点数`value`的精度，没有参数时，默认四舍五入**保留小数点后一位**

```
<p>floatformat : {{ "2.2332"|floatformat:"2" }}</p>
<p>floatformat : {{ "2.2550"|floatformat:"2" }}</p>
<p>floatformat : {{ "2.0000"|floatformat:"2" }}</p>
```

join

> 语法：`{{ value|join:"str" }}`

> 将序列数据`value`通过`str`进行拼接

```
<p>join : {{ "abc"|join:"*" }}</p>
```

length_is

> 语法：`{{ value|length_is:"num" }}`

> 判断序列`value`的长度是否为`num`，如果是，返回`True`，反之返回`False`

```
<p>length_is : {{ "abc"|length_is:4 }}</p>
<p>length_is : {{ "abcd"|length_is:4 }}</p>
```

linebreaksbr

> 语法：`{{ value|linebreaksbr }}`

> 将字符串`value`中的所有换行符`\n`转换为`HTML`换行符`<br>`

```
str_ = "abc\nbbb"
```

```
<p>linebreaksbr : {{ str_|linebreaksbr }}</p>
```

linenumbers

> 语法：`{{ value|linenumbers }}`

> 显示`value`数据的行号，一般来说，是根据`value`字符串中的`\n`换行来确定每一行

```
str_ = "abc\nbbb"
```

```
<p>linenumbers :<br> {{ str_|linenumbers }}</p>
<p>linenumbers :<br> {{ str_|linenumbers|linebreaksbr }}</p>
```

ljust

> 语法：`{{ value|ljust:"num" }}`

> 将字符串`value`按照给定宽度`num`左对齐

```
<p>ljust : {{"test"|ljust:"10" }}</p>
```

> HTML中空格是被忽略的，所以直观的我们并看不到这个过滤器的对齐效果
>
> 需要使用 才可以在HTML中展示真正的空格效果，这个操作会在之后的自定义过滤器中为大家介绍

rjust

> 语法：{{ value|rjust:"num" }}

> 将字符串value按照给定宽度num右对齐

```
<p>rjust : {{ "test"|rjust:"10" }}</p>
```

lower

> 语法：{{ value|lower }}

> 将字符串value中的全部字符串小写

```
<p>rjust : {{ "Aa123Bb"|lower }}</p>
```

upper

> 语法：{{ value|upper }}

> 将字符串value中的全部字符串大写

```
<p>upper : {{ "Aa123Bb"|upper }}</p>
```

title

> 语法：{{ value|title }}

> 将value字符串中每一个单词首字母大写，其余字符小写

```
<p>title : {{ "heLLO a12b world"|title }}</p>
```

make_list

> 语法：{{ value|make_list }}

> 将value转换为列表

```
<p>make_list : {{ "a1好a2a"|make_list }}</p>
```

```
['a', '1', '好', 'a', '2', 'a']
```

random

> 语法：`{{ value|random }}`

> 返回`value`序列中的一个随机值

```
<p>random : {{ "12345"|random }}</p>
```

slice

> 语法：`{{ value|slice:"start:stop:step" }}`

> 与`Python`中序列切片用法类似，取出一定范围内的数据

```
<p>slice : {{ "abcdef"|slice:"0:5" }}</p>
<p>slice : {{ "abcdef"|slice:"0:6" }}</p>
<p>slice : {{ "abcdef"|slice:"0:6:2" }}</p>
```

time

> 语法：`{{ value|time:"time_format" }}`

> 与`date`过滤器类似，但该过滤器只处理时、分、秒；
>
> 根据时间格式化字符输出时间，输出最终格式与项目时区及语言设置有关

```
import datetime
datetime = datetime.datetime.now()
```

```
<p>time: {{ datetime|time:"H:i" }}</p>
<p>time: {{ datetime|time:"Y/m/d" }}</p>
```

timesince

> 语法：`{{ start_time|timesince:end_time }}`

> 计算从`start_time`一直到`end_time`的时间间隔，`end_time`为可选，没有该值，截至从当前时间开始
>
> 分钟为返回最小单位

```
start_time = datetime.datetime(2019, 3, 3, 15)
end_time = datetime.datetime(2019, 3, 5, 17)
```

```
<p>time : {{ start_time|timesince:end_time }}</p
<p>time : {{ start_time|timesince }}</p>
```

```
time : 2 days, 2 hours
time : 3 weeks, 2 days
```

urlencode

> 语法：`{{ value|urlencode }}`

> 使用连接编码格式处理`value`

```
<p>urlencode : {{ "http://example.com"|urlencode }}</p>
```

```
urlencode : http%3A//example.com
```

urlize

> 语法：`{{ value|urlize }}`

> 使连接字符串`value`变为可点击的a标签连接

```
<p>urlize : {{ "http://example.com"|urlize }}</p>
<p>urlize : {{ "http://example.com" }}</p>
```

人性化过滤器

> 除去上面所介绍的过滤器，django还提供了一个专门人性化处理数据的过滤器组件；
>
> 使用时，需要将**'django.contrib.humanize'**添加到`settings.py`文件中的`INSTALLED_APPS`属性中
>
> 之后在模板页面加载`{% load humanize %}`就可以使用到`humanize`中的人性化过滤器

apnumber

> 语法：`{{ value|apnumber }}`

> 将整数转化为字符串，并按照语言设置返回对应的数字表示方式

```
<p>intcomma : {{ "3000"|intcomma }}</p>
<p>intcomma : {{ "23300"|intcomma }}</p>
```

intword

> 语法：`{{ value|intword }}`

> 将一个大型数字转换成友好的文字表达形式，适合超过`100万`的数字

```
<p>intword : {{ "310100100"|intword }}</p>
```

naturalday

> 语法：`{{ value|naturalday }}`

> 返回value时间相对于今天。返回"今天"，"明天"或者"昨天"

```
today = datetime.datetime.now()
```

```
<p>naturalday : {{ today|naturalday }}</p>
```

naturaltime

> 语法：`{{ value|naturaltime }}`

> 获得`value`与当前时间的时间间隔，并使用合适的文字来描述；
>
> 如果超过一天间隔，将会使用`timesice`过滤器格式

```
start_time = datetime.datetime(2019, 3, 3, 15)
```

```
<p>naturalday : {{ start_time|naturaltime }}</p>
```

自定义过滤器

> 虽然有了django给我们提供的这么多方便的标签和过滤器；
>
> 但是有些时候，还不能达成我们想要的功能，那么就需要我们自定义标签和过滤器
>
> django默认的过滤器及标签文件夹：
>
> `django/template/defaultfilters.py`
>
> `django/template/defaulttags.py`

1. 在**当前app下**创建保存自定义标签及过滤器的文件夹，这个文件夹常命名为`templatetags`
2. 为了支持该文件夹可以作为模块导入，`templatetags`文件夹下创建`__init__.py`文件
3. 创建过滤器`xxxx.py`文件，文件名自定义
4. 过滤器文件头部必须包含名为`register`的全局变量，该变量是`template.Library`对象的实例
5. 自定义过滤器为一个`Python`函数，参数可以是`1-2`个
    - 比如`{{ value|upper }}`，过滤器函数名为`upper`，参数为`value`

    > 注意：过滤器参数可以是一个字符串，也可以使类似列表的其他类型，参数可以设置默认值。另外需要注意的是模板中无法进行异常处理，过滤器一旦出现错误，将会引发服务器错误
6. 最重要的一步，所有编写完成的过滤器函数，都要记得：使用`register.filter()`函数将其注册为`Library`实例
    - `register.filter(name=None, filter_func=None)`：注册过滤函数

        > `name`：一个字符串，表示过滤器在模板的使用名称
        >
        > `filter_func`：编写好的过滤器函数

```
# app/templatetags/my_filter.py
from django import template
register = template.Library()

def return_length(value):
    #  返回变量长度
    return len(str(value))

register.filter("return_length",return_length)
```

```
<p>{{ 'abc'|return_length }}</p>
<!-- 返回3 -->
```

> 此外：除了我们使用`register.filter`函数来对过滤器函数进行注册；
>
> 还可以将`register.filter`作为装饰器`@register.filter`来使用，可以更加方便的进行过滤器函数注册

```
from django import template
register = template.Library()

@register.filter(name="delete_space")
def delete_space(value):
    # 去掉value数据中所有空格
    return value.replace(" ","")
```

> 但是，这里有个问题，我们的过滤器经常期望处理的数据类型是一个字符串，但是以上过滤器如果在对数字类型进行处理时，会引发`'int' object has no attribute 'replace'`，这样的错误，那么需要我们对传入过滤器的value参数进行字符串转变的处理
>
> 解决办法也很简单，大家可能想到了直接用字符串工厂函数去转换传入参数、但是这里有更加优雅安全的方式，通过`django.template.defaultfilter`模块下的`stringfilter`装饰器来对过滤器函数进行装饰
>
> `stringfilter`**这个装饰器可以帮助我们把传入过滤器函数的参数转换为它的字符串值**

```
from django.template.defaultfilters import stringfilter
@register.filter(name="delete_space")
@stringfilter
def delete_space(value):
    return value.replace(" ","")
```

> 现在过滤器函数的`value`参数将会先被装饰器**@stringfilter**处理成对应的字符串类型之后
>
> 才会被作为参数传递到过滤器函数`delete_space`中
>
> 接下来通过这个过滤器处理一个**非字符串类型**也就不会在报错了

自定义标签

> 标签要实现的功能可以比过滤器更加强大，可以支持接收更多参数！

> 基本使用语法`{% tag "arg1" "arg2" "arg3" ... %}`

> 很多模板标签可以接收多个参数，字符串或者模板变量；并且可以将这些变量经过一系列处理之后返回一个字符串这样的标签我们可以通过django为我们提供的`simple_tag()`注册函数来进行编写，该函数来自于`django.template.Library`

> 同样的，编写自定义标签函数完成之后，也需要进行注册，也可以直接将`@simple_tag`作为装饰器使用注册

```
from django import template

register = template.Library()

@register.simple_tag(name="myUpper")
def myUpper(value):
    # 将模板变量处理为纯大写的模板标签
    return str(value).upper()
# register.simple_tag(name="myUpper",func=myUpper)
```

- `simple_tag()` 函数在这里帮助我们做了如下工作：
    1. 检查标签函数所需参数数量
    2. 截掉参数中的引号，确保函数接收到的是一个普通的字符串
    3. 截掉参数中的引号，确保函数接收到的是一个普通的字符串

> 如果我们希望标签函数可以访问到当前模板中其他全部的模板变量值；
>
> 那么可以使用simple_tag(takes_context=True)参数
>
> 比如通过视图函数向模板返回了

```
value = '哈哈哈哈哈哈'
return render(request, template, locals())
```

> 可以在自定义标签处通过`simple_tag(takes_context=True)`来进行视图函数中`content`值的获取；
>
> 但是还要注意的是，此时自定义标签函数参数位置第一个必须为`context`

```
@register.simple_tag(takes_context=True)
def get_context(context):
    value = context.get("value")
    return "获取到的模板变量:%s" % value
```

> 模板页面直接使用

```
<p>{% get_context %}</p>
```

模板继承

> 关于模板，经常重复的编写页面是一个非常痛苦的事情；
>
> 那么在`django`中也提供了一种非常舒服方便的方法，可以使新的模板页面来继承自一个已编写好的`html`页面实现复用，免去重复工作；这就是模板继承

block

> 页面的继承不能说全部都拿过来，有时候只需要已经编写好的页面某些部分
>
> 其他部分提前挖好一些坑，去填充不同内容
>
> 挖坑可以通过模板中的`{% block %}`标签

```
{% block name %}
    预留区域，可供未来继承的页面覆盖
{% endblock name %}
```

> 设计一个可以被继承的父模板，我们经常叫做base.html

```
<!DOCTYPE html>
<html>

<head>
    <title>
        {% block title %}
        父模板标题
        {% endblock title %}
    </title>
</head>

<body>
    {% block top %}
        <h3>父模板</h3>
    {% endblock top %}

    {% block content %}
        <div>这里是父模板页面内容</div>
    {% endblock content %} </body>
</html>
```

> 在这个页面中，我们设计了三个`block`标签块`title`、`content`以及`top`；
>
> 每一个块都可以被之后继承的页面所覆盖新的内容

- 继承页面使用`{% extends "base.html" %}`标签进行页面的继承，现在编写一个test.html

```
{% extends "base.html" %}

{% block title %}
    子模版
{% endblock title %}

{% block top %}
    <h3>子模板</h3>
{% endblock top %}

{% block content %}
    <p>我是子模版</p>
{% endblock content %} </body>

{% block other %}
    哈哈哈哈哈
    这里的内容不会显示
    父模板并没有这样的block块
{% endblock other%}
```

> 除了对应`block`标签内容被子模板修改，其余内容均默认使用父模板中的

- 注意：
  - 如果父模板内有模板变量或者其他上下文数据，不会被子模板继承，但是子模板可以为父模板内的模板数据赋值
  - 如果需模板中具有模板变量等上下文数据，只有放到`block`标签块内数据才会显示
  - 子模板中修改父模板中并不存在的block块，子模板不会显示

模板加载

> 除了`{% extends %}`与`{% block %}`结合的方式可以继承一个父模板
>
> 我们还可以使用`{% include %}`一个新的标签进行模板加载，`include`标签使用语法与`extends`类似

include

> 现在新建一个html文件，名为`li.html`，用来写一个简单的列表

```
<ul>
    <li>吃饭</li>
    <li>睡觉</li>
    <li>玩耍</li>
    <li>{{ var }}</li>
</ul>
```

> 在需要导入的页面中使用`{% include "li.html" %}`进行引入\

```
{% extends "base.html" %}

{% block title %}
    子模版
{% endblock title %}

{% block top %}
    <h3>子模板</h3>
{% endblock top %}

{% block content %}
    <p>我是子模版</p>
    {% include "li.html" %}
{% endblock content %} </body>
```

> 被`include`引入的新模板，会在渲染完成之后添加到父模板所给定的对应`block`块中

> 与`extends`不同，`extends`常用来控制整个模板的样式和效果；
>
> 而`include`更加细化，可以在一个模板内包含其他多个模板
>
> **如果`include`所包含的模板页面中有模板变量需要被填充，会在包含`include`的页面下进行渲染**
>
> 这种行为也好像是，把一个新的渲染好的`html`页面嵌入了进来一样