模板设计者文档

这份文档描述了模板引擎中的语法和语义结构，对于创建 Jinja 模板是一份相当有用 的参考。因为模板引擎非常灵活，应用中的配置会在分隔符和未定义值的行为方面与 这里的配置有细微差异。

概要

模板仅仅是文本文件。它可以生成任何基于文本的格式（HTML、XML、CSV、LaTex 等等）。 它并没有特定的扩展名， `.html` 或 `.xml` 都是可以的。

模板包含 变量 或 表达式 ，这两者在模板求值的时候会被替换为值。模板中 还有标签，控制模板的逻辑。模板语法的大量灵感来自于 Django 和 Python 。

下面是一个最小的模板，它阐明了一些基础。我们会在文档中后面的部分解释细节:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}
</body>
</html>
```

这包含了默认的设定。应用开发者也会把语法从 `{% foo %}` 改成 `<% foo %>` 或类似的东西。

这里有两种分隔符: `{% ... %}` 和 `{{ ... }}` 。前者用于执行诸如 for 循环 或赋值的语句，后者把表达式的结果打印到模板上。

变量

应用把变量传递到模板，你可能在模板中弄混。变量上面也可以有你能访问的属性或元 素。变量看起来是什么，完全取决于应用提供了什么。

你可以使用点（ `.` ）来访问变量的属性，作为替代，也可以使用所谓的“下标”语 法（ `[]` ）。下面的几行效果是一样的:

```
{{ foo.bar }}
{{ foo['bar'] }}
```

知晓花括号 不是 变量的一部分，而是打印语句的一部分是重要的。如果你访问标签 里的不带括号的变量。

如果变量或属性不存在，会返回一个未定义值。你可以对这类值做什么取决于应用的配 置，默认的行为是它如果被打印，其求值为一个空字符串，并且你可以迭代它，但其它 操作会失败。

实现:

为方便起见，Jinja2 中 `foo.bar` 在 Python 层中做下面的事情:

- 检查 *foo* 上是否有一个名为 *bar* 的属性。
- 如果没有，检查 *foo* 中是否有一个 `'bar'` 项 。
- 如果没有，返回一个未定义对象。

`foo['bar']` 的方式相反，只在顺序上有细小差异:

- 检查在 *foo* 中是否有一个 `'bar'` 项。

- 如果没有，检查 *foo* 上是否有一个名为 *bar* 的属性。
- 如果没有，返回一个未定义对象。

如果一个对象有同名的项和属性，这很重要。此外，有一个 **attr()** 过滤 器，它只查找属性。

过滤器

变量可以通过 过滤器 修改。过滤器与变量用管道符号（ | ）分割，并且也 可以用圆括号传递可选参数。多个过滤器可以链式调用，前一个过滤器的输出会被作为 后一个过滤器的输入。

例如 `{{ name|striptags|title }}` 会移除 *name* 中的所有 HTML 标签并且改写 为标题样式的大小写格式。过滤器接受带圆括号的参数，如同函数调用。这个例子会 把一个列表用逗号连接起来: `{{ list|join(', ') }}` 。

下面的 内置过滤器清单 节介绍了所有的内置过滤器。

测试

除了过滤器，所谓的“测试”也是可用的。测试可以用于对照普通表达式测试一个变量。 要测试一个变量或表达式，你要在变量后加上一个 *is* 以及测试的名称。例如，要得出 一个值是否定义过，你可以用 `name is defined` ，这会根据 *name* 是否定义返回 true 或 false 。

测试也可以接受参数。如果测试只接受一个参数，你可以省去括号来分组它们。例如， 下面的两个表达式做同样的事情:

```
{% if loop.index is divisibleby 3 %}
{% if loop.index is divisibleby(3) %}
```

下面的 内置测试清单 章节介绍了所有的内置测试。

注释

要把模板中一行的部分注释掉，默认使用 `{# ... #}` 注释语法。这在调试或 添加给你自己或其它模板设计者的信息时是有用的:

```
{# note: disabled template because we no longer use this
    {% for user in users %}
        ...
    {% endfor %}
#}
```

空白控制

默认配置中，模板引擎不会对空白做进一步修改，所以每个空白（空格、制表符、换行符 等等）都会原封不动返回。如果应用配置了 Jinja 的 *trim_blocks* ，模板标签后的 第一个换行符会被自动移除（像 PHP 中一样）。

此外，你也可以手动剥离模板中的空白。当你在块（比如一个 for 标签、一段注释或变 量表达式）的开始或结束放置一个减号（ - ），可以移除块前或块后的空白:

```
{% for item in seq -%}
    {{ item }}
{%- endfor %}
```

这会产出中间不带空白的所有元素。如果 *seq* 是 `1` 到 `9` 的数字的列表， 输出会是 `123456789` 。

如果开启了 行语句 ，它们会自动去除行首的空白。

提示:

标签和减号之间不能有空白。

有效的:

```
{%- if foo -%}...{% endif %}
```

无效的:

```
{% - if foo - %}...{% endif %}
```

转义

有时想要或甚至必要让 Jinja 忽略部分，不会把它作为变量或块来处理。例如，如果 使用默认语法，你想在在使用把 {{ 作为原始字符串使用，并且不会开始一个变量 的语法结构，你需要使用一个技巧。

最简单的方法是在变量分隔符中（ {{ ）使用变量表达式输出:

```
{{ '{{' }}
```

对于较大的段落，标记一个块为 *raw* 是有意义的。例如展示 Jinja 语法的实例， 你可以在模板中用这个片段:

```
{% raw %}
    <ul>
    {% for item in seq %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endraw %}
```

行语句

如果应用启用了行语句，就可以把一个行标记为一个语句。例如如果行语句前缀配置为 # ，下面的两个例子是等价的:

```
<ul>
# for item in seq
    <li>{{ item }}</li>
# endfor
</ul>
```

```
<ul>
{% for item in seq %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

行语句前缀可以出现在一行的任意位置，只要它前面没有文本。为了语句有更好的可读 性，在块的开始（比如 *for* 、 *if* 、 *elif* 等等）以冒号结尾:

```
# for item in seq:
    ...
# endfor
```

提示:

若有未闭合的圆括号、花括号或方括号，行语句可以跨越多行:

```
<ul>
# for href, caption in [('index.html', 'Index'),
                        ('about.html', 'About')]:
    <li><a href="{{ href }}">{{ caption }}</a></li>
# endfor
</ul>
```

从 Jinja 2.2 开始，行注释也可以使用了。例如如果配置 ## 为行注释前缀， 行中所有 ## 之后的内容（不包括换行符）会被忽略:

```
# for item in seq:
    <li>{{ item }}</li>     ## this comment is ignored
# endfor
```

模板继承

Jinja 中最强大的部分就是模板继承。模板继承允许你构建一个包含你站点共同元素的基 本模板“骨架”，并定义子模板可以覆盖的 块 。

听起来复杂，实际上很简单。从例子上手是最易于理解的。

基本模板

这个模板，我们会把它叫做 base.html ，定义了一个简单的 HTML 骨架文档，你可 能使用一个简单的两栏页面。用内容填充空的块是子模板的工作:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
```

在本例中， {% block %} 标签定义了四个字幕版可以填充的块。所有的 *block* 标签 告诉模板引擎子模板可以覆盖模板中的这些部分。

子模板

一个子模板看起来是这样:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome on my awesome homepage.
    </p>
{% endblock %}
```

{% extend %} 标签是这里的关键。它告诉模板引擎这个模板“继承”另一个模板。 当模板系统对这个模板求值时，首先定位父模板。 extends 标签应该是模板中的第一个 标签。它前面的所有东西都会按照普通情况打印出来，而且可能会导致一些困惑。更多 该行为的细节以及如何利用它，见 *Null-Master* 退回 。

模板的文件名依赖于模板加载器。例如 **FileSystemLoader** 允许你用文件名访 问其它模板。你可以使用斜线访问子目录中的模板:

```
{% extends "layout/default.html" %}
```

这种行为也可能依赖于应用内嵌的 Jinja 。注意子模板没有定义 `footer` 块，会 使用父模板中的值。

你不能在同一个模板中定义多个同名的 `{% block %}` 标签。因为块标签以两种 方向工作，所以存在这种限制。即一个块标签不仅提供一个可以填充的部分，也在父级 定义填充的内容。如果同一个模板中有两个同名的 `{% blok %}` 标签，父模板 无法获知要使用哪一个块的内容。

如果你想要多次打印一个块，无论如何你可以使用特殊的 *self* 变量并调用与块同名 的函数:

```
<title>{% block title %}{% endblock %}</title>
<h1>{{ self.title() }}</h1>
{% block body %}{% endblock %}
```

Super 块

可以调用 *super* 来渲染父级块的内容。这会返回父级块的结果:

```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ super() }}
{% endblock %}
```

命名块结束标签

Jinja2 允许你在块的结束标签中加入的名称来改善可读性:

```
{% block sidebar %}
    {% block inner_sidebar %}
        ...
    {% endblock inner_sidebar %}
{% endblock sidebar %}
```

无论如何， *endblock* 后面的名称一定与块名匹配。

嵌套块和作用域

嵌套块可以胜任更复杂的布局。而默认的块不允许访问块外作用域中的变量:

```
{% for item in seq %}
    <li>{% block loop_item %}{{ item }}{% endblock %}</li>
{% endfor %}
```

这个例子会输出空的 `<li>` 项，因为 *item* 在块中是不可用的。其原因是，如果 块被子模板替换，变量在其块中可能是未定义的或未被传递到上下文。

从 Jinja 2.2 开始，你可以显式地指定在块中可用的变量，只需在块声明中添加 *scoped* 修饰，就把块设定到作用域中:

```
{% for item in seq %}
    <li>{% block loop_item scoped %}{{ item }}{% endblock %}</li>
{% endfor %}
```

当覆盖一个块时，不需要提供 *scoped* 修饰。

模板对象

Changed in version 2.4.

当一个模板对象被传递到模板上下文，你也可以从那个对象继承。假设调用 代码传递 *layout_template* 布局模板到环境，这段代码会工作:

```
{% extends layout_template %}
```

之前 *layout_template* 变量一定是布局模板文件名的字符串才能工作。

HTML 转义

当从模板生成 HTML 时，始终有这样的风险:变量包含影响已生成 HTML 的字符。有两种 解决方法:手动转义每个字符或默认自动转义所有的东西。

Jinja 两者都支持，使用哪个取决于应用的配置。默认的配置未开启自动转义有这样几个 原因:

- 转义所有非安全值的东西也意味着 Jijna 转义已知不包含 HTML 的值，比如数字，对 性能有巨大影响。
- 关于变量安全性的信息是易碎的。可能会发生强制标记一个值为安全或非安全的情况， 而返回值会被作为 HTML 转义两次。

使用手动转义

如果启用了手动转义，按需转义变量就是 你的 责任。要转义什么？如果你有 一个 可能 包含 `>`、 `<`、 `&` 或 `"` 字符的变量，你必须转义 它，除非变量中的 HTML 有可信的良好格式。转义通过用管道传递到过滤器 `|e` 来实现: `{{ user.username|e }}` 。

使用自动转义

当启用了自动转移，默认会转移一切，除非值被显式地标记为安全的。可以在应用中 标记，也可以在模板中使用 *|safe* 过滤器标记。这种方法的主要问题是 Python 本 身没有被污染的值的概念，所以一个值是否安全的信息会丢失。如果这个信息丢失， 会继续转义，你最后会得到一个转义了两次的内容。

但双重转义很容易避免，只需要依赖 Jinja2 提供的工具而不使用诸如字符串模运算符 这样的 Python 内置结构。

返回模板数据（宏、 *super* 、 *self.BLOCKNAME* ）的函数，其返回值总是被标记 为安全的。

模板中的字符串字面量在自动转义中被也被视为是不安全的。这是因为安全的字符串是 一个对 Python 的扩展，而不是每个库都能妥善地使用它。

控制结构清单

控制结构指的是所有的那些可以控制程序流的东西 —— 条件（比如 if/elif/ekse ）、 for 循环、以及宏和块之类的东西。控制结构在默认语法中以 `{% .. %}` 块的形式 出现。

For

遍历序列中的每项。例如，要显示一个由 *users* ` 变量提供的用户列表:

```
<h1>Members</h1>
<ul>
{% for user in users %}
  <li>{{ user.username|e }}</li>
{% endfor %}
</ul>
```

因为模板中的变量保留它们的对象属性，可以迭代像 *dict* 的容器:

```
<dl>
{% for key, value in my_dict.iteritems() %}
    <dt>{{ key|e }}</dt>
    <dd>{{ value|e }}</dd>
{% endfor %}
</dl>
```

注意无论如何字典通常是无序的，所以你可能需要把它作为一个已排序的列表传入 到模板或使用 *dictsort* 过滤器。

在一个 for 循环块中你可以访问这些特殊的变量:

变量	描述
loop.index	当前循环迭代的次数（从 1 开始）
loop.index0	当前循环迭代的次数（从 0 开始）
loop.revindex	到循环结束需要迭代的次数（从 1 开始）
loop.revindex0	到循环结束需要迭代的次数（从 0 开始）
loop.first	如果是第一次迭代，为 True 。
loop.last	如果是最后一次迭代，为 True 。
loop.length	序列中的项目数。
loop.cycle	在一串序列间期取值的辅助函数。见下面的解释。

在 for 循环中，可以使用特殊的 *loop.cycle* 辅助函数，伴随循环在一个字符串/变 量列表中周期取值:

```
{% for row in rows %}
    <li class="{{ loop.cycle('odd', 'even') }}">{{ row }}</li>
{% endfor %}
```

从 Jinja 2.1 开始，一个额外的 *cycle* 辅助函数允许循环限定外的周期取值。 更多信息请阅读 全局函数清单 。

与 Python 中不同，模板中的循环内不能 *break* 或 *continue* 。但你可以在迭代 中过滤序列来跳过项目。下面的例子中跳过了所有隐藏的用户:

```
{% for user in users if not user.hidden %}
    <li>{{ user.username|e }}</li>
{% endfor %}
```

好处是特殊的 *loop* 可以正确地计数，从而不计入未迭代过的用户。

如果因序列是空或者过滤移除了序列中的所有项目而没有执行循环，你可以使用 *else* 渲染一个用于替换的块:

```
<ul>
{% for user in users %}
    <li>{{ user.username|e }}</li>
{% else %}
    <li><em>no users found</em></li>
{% endfor %}
</ul>
```

也可以递归地使用循环。当你处理诸如站点地图之类的递归数据时很有用。要递归地 使用循环，你只需要在循环定义中加上 *recursive* 修饰，并在你想使用递归的地 方，对可迭代量调用 *loop* 变量。

下面的例子用递归循环实现了站点地图:

```
<ul class="sitemap">
{%- for item in sitemap recursive %}
    <li><a href="{{ item.href|e }}">{{ item.title }}</a>
    {%- if item.children -%}
        <ul class="submenu">{{ loop(item.children) }}</ul>
    {%- endif %}</li>
{%- endfor %}
</ul>
```

If

Jinja 中的 *if* 语句可比 Python 中的 if 语句。在最简单的形式中，你可以测试 一个变量是否未定义，为空或 false:

```
{% if users %}
<ul>
{% for user in users %}
    <li>{{ user.username|e }}</li>
{% endfor %}
```

```
</ul>
{% endif %}
```

像在 Python 中一样，用 *elif* 和 *else* 来构建多个分支。你也可以用更复杂的 表达式:

```
{% if kenny.sick %}
    Kenny is sick.
{% elif kenny.dead %}
    You killed Kenny!  You bastard!!!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

If 也可以被用作 内联表达式 并作为 循环过滤 。

宏

宏类似常规编程语言中的函数。它们用于把常用行为作为可重用的函数，取代 手动重复的工作。

这里是一个宏渲染表单元素的小例子:

```
{% macro input(name, value='', type='text', size=20) -%}
    <input type="{{ type }}" name="{{ name }}" value="{{
        value|e }}" size="{{ size }}">
{%- endmacro %}
```

在命名空间中，宏之后可以像函数一样调用:

```
<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```

如果宏在不同的模板中定义，你需要首先使用 *import* 。

在宏内部，你可以访问三个特殊的变量:

varargs
: 如果有多于宏接受的参数个数的位置参数被传入，它们会作为列表的值保存在 *varargs* 变量上。

kwargs
: 同 *varargs* ，但只针对关键字参数。所有未使用的关键字参数会存储在 这个特殊变量中。

caller
: 如果宏通过 *call* 标签调用，调用者会作为可调用的宏被存储在这个 变量中。

宏也可以暴露某些内部细节。下面的宏对象属性是可用的:

name
: 宏的名称。`{{ input.name }}` 会打印 `input` 。

arguments
: 一个宏接受的参数名的元组。

defaults
: 默认值的元组。

catch_kwargs
: 如果宏接受额外的关键字参数（也就是访问特殊的 *kwargs* 变量），为 *true* 。

catch_varargs
: 如果宏接受额外的位置参数（也就是访问特殊的 *varargs* 变量），为 *true* 。

caller
: 如果宏访问特殊的 *caller* 变量且由 *call* 标签调用，为 *true* 。

如果一个宏的名称以下划线开始，它不是导出的且不能被导入。

调用

在某些情况下，需要把一个宏传递到另一个宏。为此，可以使用特殊的 *call* 块。 下面的例子展示了如何让宏利用调用功能:

```
{% macro render_dialog(title, class='dialog') -%}
    <div class="{{ class }}">
        <h2>{{ title }}</h2>
        <div class="contents">
            {{ caller() }}
        </div>
    </div>
{%- endmacro %}

{% call render_dialog('Hello World') %}
    This is a simple dialog rendered by using a macro and
    a call block.
{% endcall %}
```

也可以向调用块传递参数。这在为循环做替换时很有用。总而言之，调用块的工作方 式几乎与宏相同，只是调用块没有名称。

这里是一个带参数的调用块的例子:

```
{% macro dump_users(users) -%}
    <ul>
    {%- for user in users %}
        <li><p>{{ user.username|e }}</p>{{ caller(user) }}</li>
    {%- endfor %}
    </ul>
{%- endmacro %}

{% call(user) dump_users(list_of_user) %}
    <dl>
        <dl>Realname</dl>
        <dd>{{ user.realname|e }}</dd>
        <dl>Description</dl>
        <dd>{{ user.description }}</dd>
    </dl>
{% endcall %}
```

过滤器

过滤器段允许你在一块模板数据上应用常规 Jinja2 过滤器。只需要把代码用 *filter* 节包裹起来:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

赋值

在代码块中，你也可以为变量赋值。在顶层的（块、宏、循环之外）赋值是可导出的，即 可以从别的模板中导入。

赋值使用 *set* 标签，并且可以为多个变量赋值:

```
{% set navigation = [('index.html', 'Index'), ('about.html', 'About')] %}
{% set key, value = call_something() %}
```

继承

extends 标签用于从另一个模板继承。你可以在一个文件中使用多次继承，但是 只会执行其中的一个。见上面的关于模板继承 的节。

块

块用于继承，同时作为占位符和用于替换的内容。 模板继承 节中详细地介绍了块。

包含

include 语句用于包含一个模板，并在当前命名空间中返回那个文件的内容渲 染结果:

```
{% include 'header.html' %}
    Body
{% include 'footer.html' %}
```

被包含的模板默认可以访问活动的上下文中的变量。更多关于导入和包含的上下文 行为见 导入上下文行为 。

从 Jinja 2.2 开始，你可以把一句 include 用 `ignore missing` 标记，这样 如果模板不存在，Jinja 会忽略这条语句。当与 `with` 或 `without context` 语句联合使用时，它必须被放在上下文可见性语句 之前 。这里是一些有效的例 子:

```
{% include "sidebar.html" ignore missing %}
{% include "sidebar.html" ignore missing with context %}
{% include "sidebar.html" ignore missing without context %}
```

New in version 2.2.

你也可以提供一个模板列表，它会在包含前被检查是否存在。第一个存在的模板会 被包含进来。如果给出了 *ignore missing* ，且所有这些模板都不存在，会退化 至不做任何渲染，否则将会抛出一个异常。

例子:

```
{% include ['page_detailed.html', 'page.html'] %}
{% include ['special_sidebar.html', 'sidebar.html'] ignore missing %}
```

Changed in version 2.4: 如果传递一个模板对象到模板上下文，你可以用 *include* 包含这个对 象。

导入

Jinja2 支持在宏中放置经常使用的代码。这些宏可以被导入，并不同的模板中使用。这 与 Python 中的 import 语句类似。要知道的是，导入量会被缓存，并且默认下导入的 模板不能访问当前模板中的非全局变量。更多关于导入和包含的上下文行为见 导入上下文行为 。

有两种方式来导入模板。你可以把整个模板导入到一个变量或从其中导入请求特定的宏 /导出量。

比如我们有一个渲染表单（名为 *forms.html* ）的助手模块:

```
{% macro input(name, value='', type='text') -%}
    <input type="{{ type }}" value="{{ value|e }}" name="{{ name }}">
{%- endmacro %}

{%- macro textarea(name, value='', rows=10, cols=40) -%}
    <textarea name="{{ name }}" rows="{{ rows }}" cols="{{ cols
        }}">{{ value|e }}</textarea>
{%- endmacro %}
```

最简单灵活的方式是把整个模块导入为一个变量。这样你可以访问属性:

```
{% import 'forms.html' as forms %}
<dl>
    <dt>Username</dt>
    <dd>{{ forms.input('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ forms.input('password', type='password') }}</dd>
</dl>
<p>{{ forms.textarea('comment') }}</p>
```

此外你也可以从模板中导入名称到当前的命名空间:

```
{% from 'forms.html' import input as input_field, textarea %}
<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
```

```
    <dd>{{ input_field('password', type='password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

名称以一个或更多下划线开始的宏和变量是私有的，不能被导入。

Changed in version 2.4: 如果传递一个模板对象到模板上下文，从那个对象中导入。

导入上下文行为

默认下，每个包含的模板会被传递到当前上下文，而导入的模板不会。这样做的原因 是导入量不会像包含量被缓存，因为导入量经常只作容纳宏的模块。

无论如何，这当然也可以显式地更改。通过在 import/include 声明中直接添加 *with context* 或 *without context* ，当前的上下文可以传递到模板，而且不会 自动禁用缓存。

这里有两个例子:

```
{% from 'forms.html' import input with context %}
{% include 'header.html' without context %}
```

提示:

在 Jinja 2.0 中，被传递到被包含模板的上下文不包含模板中定义的变量。 事实上，这不能工作:

```
{% for box in boxes %}
    {% include "render_box.html" %}
{% endfor %}
```

在 Jinja 2.0 中，被包含的模板 `render_box.html` 不能 访问 *box* 。从 Jinja 2.1 开始， `render_box.html` 可以 这么做。

表达式

Jinja 中到处都允许使用基本表达式。这像常规的 Python 一样工作，即使你不用 Python 工作，你也会感受到其带来的便利。

字面量

表达式最简单的形式就是字面量。字面量表示诸如字符串和数值的 Python 对象。下面 的字面量是可用的:

“Hello World”:
: 双引号或单引号中间的一切都是字符串。无论何时你需要在模板中使用一个字 符串（比如函数调用、过滤器或只是包含或继承一个模板的参数），它们都是 有用的。

42 / 42.23:
: 直接写下数值就可以创建整数和浮点数。如果有小数点，则为浮点数，否则为 整数。记住在 Python 里， `42` 和 `42.0` 是不一样的。

['list', 'of', 'objects']:
: 一对中括号括起来的东西是一个列表。列表用于存储和迭代序列化的数据。例如 你可以容易地在 for 循环中用列表和元组创建一个链接的列表:

```
<ul>
{% for href, caption in [('index.html', 'Index'), ('about.html', 'About'),
                         ('downloads.html', 'Downloads')] %}
    <li><a href="{{ href }}">{{ caption }}</a></li>
{% endfor %}
</ul>
```

('tuple', 'of', 'values'):
: 元组与列表类似，只是你不能修改元组。如果元组中只有一个项，你需要以逗号 结尾它。元组通常用于表示两个或更多元素的项。更多细节见上面的例子。

{‘dict’: ‘of’, ‘key’: ‘and’, ‘value’: ‘pairs’}:
Python 中的字典是一种关联键和值的结构。键必须是唯一的，并且键必须只有一个 值。字典在模板中很少使用，罕用于诸如 `xmlattr()` 过滤器之类。

true / false:
true 永远是 true ，而 false 始终是 false 。

提示:

特殊常量 *true* 、 *false* 和 *none* 实际上是小写的。因为这在过去会导致 混淆，过去 *True* 扩展为一个被认为是 false 的未定义的变量。所有的这三个 常量也可以被写成首字母大写（ *True* 、 *False* 和 *None* ）。尽管如此， 为了一致性（所有的 Jinja 标识符是小写的），你应该使用小写的版本。

算术

Jinja 允许你用计算值。这在模板中很少用到，但是为了完整性允许其存在。支持下面的 运算符:

+
把两个对象加到一起。通常对象是素质，但是如果两者是字符串或列表，你可以用这 种方式来衔接它们。无论如何这不是首选的连接字符串的方式！连接字符串见 ~ 运算符。 `{{ 1 + 1 }}` 等于 `2` 。

-
用第一个数减去第二个数。 `{{ 3 - 2 }}` 等于 `1` 。

/
对两个数做除法。返回值会是一个浮点数。 `{{ 1 / 2 }}` 等于 `{{ 0.5 }}` 。

//
对两个数做除法，返回整数商。 `{{ 20 // 7 }}` 等于 `2` 。

%
计算整数除法的余数。 `{{ 11 % 7 }}` 等于 `4` 。

*
用右边的数乘左边的操作数。 `{{ 2 * 2 }}` 会返回 `4` 。也可以用于重 复一个字符串多次。 `{{ '=' * 80 }}` 会打印 80 个等号的横条。

**
取左操作数的右操作数次幂。 `{{ 2**3 }}` 会返回 `8` 。

比较

==
比较两个对象是否相等。

!=
比较两个对象是否不等。

>
如果左边大于右边，返回 *true* 。

>=
如果左边大于等于右边，返回 *true* 。

<
如果左边小于右边，返回 *true* 。

<=
如果左边小于等于右边，返回 *true* 。

逻辑

对于 *if* 语句，在 *for* 过滤或 *if* 表达式中，它可以用于联合多个表达式:

and

如果左操作数和右操作数同为真，返回 true 。

or
: 如果左操作数和右操作数有一个为真，返回 true 。

not
: 对一个表达式取反（见下）。

(expr)
: 表达式组。

提示:

`is` 和 `in` 运算符同样支持使用中缀记法: `foo is not bar` 和 `foo not in bar` 而不是 `not foo is bar` 和 `not foo in bar` 。所有的 其它表达式需要前缀记法 `not (foo and bar)` 。

其它运算符

下面的运算符非常有用，但不适用于其它的两个分类:

in
: 运行序列/映射包含检查。如果左操作数包含于右操作数，返回 true 。比如 `{{ 1 in [1,2,3] }}` 会返回 true 。

is
: 运行一个 测试 。

|
: 应用一个 过滤器 。

~
: 把所有的操作数转换为字符串，并且连接它们。 `{{ "Hello " ~ name ~ "!" }}` 会返回（假设 *name* 值为 `''John'` ） `Hello John!` 。

()
: 调用一个可调用量:`{{ post.render() }}` 。在圆括号中，你可以像在 python 中一样使用位置参数和关键字参数: `{{ post.render(user, full=true) }}` 。

. / []
: 获取一个对象的属性。（见 变量 ）

If 表达式

同样，也可以使用内联的 *if* 表达式。这在某些情况很有用。例如你可以用来在一个 变量定义的情况下才继承一个模板，否则继承默认的布局模板:

```
{% extends layout_template if layout_template is defined else 'master.html' %}
```

一般的语法是 `<do something> if <something is true> else <do something else>` 。

else 部分是可选的。如果没有显式地提供 else 块，会求值一个未定义对象:

```
{{ '[%s]' % page.title if page.title }}
```

内置过滤器清单

abs(*number*)
: Return the absolute value of the argument.

attr(*obj*, *name*)
: Get an attribute of an object. `foo|attr("bar")` works like `foo["bar"]` just that always an attribute is returned and items are not looked up.

See *Notes on subscriptions* for more details.

batch(*value*, *linecount*, *fill_with=None*)
A filter that batches items. It works pretty much like *slice* just the other way round. It returns a list of lists with the given number of items. If you provide a second parameter this is used to fill up missing items. See this example:

```
<table>
{%- for row in items|batch(3, ' ') %}
  <tr>
  {%- for column in row %}
    <td>{{ column }}</td>
  {%- endfor %}
  </tr>
{%- endfor %}
</table>
```

capitalize(*s*)
Capitalize a value. The first character will be uppercase, all others lowercase.

center(*value*, *width=80*)
Centers the value in a field of a given width.

default(*value*, *default_value=u''*, *boolean=False*)
If the value is undefined it will return the passed default value, otherwise the value of the variable:

```
{{ my_variable|default('my_variable is not defined') }}
```

This will output the value of `my_variable` if the variable was defined, otherwise `'my_variable is not defined'`. If you want to use default with variables that evaluate to false you have to set the second parameter to *true*:

```
{{ ''|default('the string was empty', true) }}
```

Aliases : `d`

dictsort(*value*, *case_sensitive=False*, *by='key'*)
Sort a dict and yield (key, value) pairs. Because python dicts are unsorted you may want to use this function to order them by either key or value:

```
{% for item in mydict|dictsort %}
    sort the dict by key, case insensitive

{% for item in mydict|dictsort(true) %}
    sort the dict by key, case sensitive

{% for item in mydict|dictsort(false, 'value') %}
    sort the dict by key, case insensitive, sorted
    normally and ordered by value.
```

escape(*s*)
Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

Aliases : `e`

filesizeformat(*value*, *binary=False*)
Format the value like a ‘human-readable’ file size (i.e. 13 kB, 4.1 MB, 102 Bytes, etc). Per default decimal prefixes are used (Mega, Giga, etc.), if the second parameter is set to *True* the binary prefixes are used (Mebi, Gibi).

first(*seq*)
Return the first item of a sequence.

float(*value*, *default=0.0*)
Convert the value into a floating point number. If the conversion doesn’t work it will return `0.0`. You can override this default using the first parameter.

forceescape(*value*)
Enforce HTML escaping. This will probably double escape variables.

format(*value*, **args*, ***kwargs*)
Apply python string formatting on an object:

```
{{ "%s - %s"|format("Hello?", "Foo!") }}
    -> Hello? - Foo!
```

groupby(*value*, *attribute*)
Group a sequence of objects by a common attribute.

If you for example have a list of dicts or objects that represent persons with *gender*, *first_name* and *last_name* attributes and you want to group all users by genders you can do something like the following snippet:

```
<ul>
{% for group in persons|groupby('gender') %}
    <li>{{ group.grouper }}<ul>
    {% for person in group.list %}
        <li>{{ person.first_name }} {{ person.last_name }}</li>
    {% endfor %}</ul></li>
{% endfor %}
</ul>
```

Additionally it's possible to use tuple unpacking for the grouper and list:

```
<ul>
{% for grouper, list in persons|groupby('gender') %}
    ...
{% endfor %}
</ul>
```

As you can see the item we're grouping by is stored in the *grouper* attribute and the *list* contains all the objects that have this grouper in common.

Changed in version 2.6: It's now possible to use dotted notation to group by the child attribute of another attribute.

indent(*s*, *width=4*, *indentfirst=False*)
Return a copy of the passed string, each line indented by 4 spaces. The first line is not indented. If you want to change the number of spaces or indent the first line too you can pass additional parameters to the filter:

```
{{ mytext|indent(2, true) }}
    indent by two spaces and indent the first line too.
```

int(*value*, *default=0*)
Convert the value into an integer. If the conversion doesn't work it will return `0`. You can override this default using the first parameter.

join(*value*, *d=u''*, *attribute=None*)
Return a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default, you can define it with the optional parameter:

```
{{ [1, 2, 3]|join('|') }}
    -> 1|2|3

{{ [1, 2, 3]|join }}
    -> 123
```

It is also possible to join certain attributes of an object:

```
{{ users|join(', ', attribute='username') }}
```

New in version 2.6: The *attribute* parameter was added.

last(*seq*)
Return the last item of a sequence.

length(*object*)
Return the number of items of a sequence or mapping.

Aliases : `count`

list(*value*)
Convert the value into a list. If it was a string the returned list will be a list of characters.

lower(*s*)
Convert a value to lowercase.

map()
Applies a filter on a sequence of objects or looks up an attribute. This is useful when dealing with lists of objects but you are really only interested in a certain value of it.

The basic usage is mapping on an attribute. Imagine you have a list of users but you are only interested in a list of usernames:

```
Users on this page: {{ users|map(attribute='username')|join(', ') }}
```

Alternatively you can let it invoke a filter by passing the name of the filter and the arguments afterwards. A good example would be applying a text conversion filter on a sequence:

```
Users on this page: {{ titles|map('lower')|join(', ') }}
```

New in version 2.7.

pprint(*value*, *verbose=False*)
Pretty print a variable. Useful for debugging.

With Jinja 1.2 onwards you can pass it a parameter. If this parameter is truthy the output will be more verbose (this requires *pretty*)

random(*seq*)
Return a random item from the sequence.

reject()
Filters a sequence of objects by appying a test to either the object or the attribute and rejecting the ones with the test succeeding.

Example usage:

```
{{ numbers|reject("odd") }}
```

New in version 2.7.

rejectattr()
Filters a sequence of objects by appying a test to either the object or the attribute and rejecting the ones with the test succeeding.

```
{{ users|rejectattr("is_active") }}
{{ users|rejectattr("email", "none") }}
```

New in version 2.7.

replace(*s*, *old*, *new*, *count=None*)
Return a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced, the second is the replacement string. If the optional third argument `count` is given, only the first `count` occurrences are replaced:

```
{{ "Hello World"|replace("Hello", "Goodbye") }}
    -> Goodbye World

{{ "aaaaargh"|replace("a", "d'oh, ", 2) }}
    -> d'oh, d'oh, aaargh
```

reverse(*value*)
Reverse the object or return an iterator the iterates over it the other way round.

round(*value*, *precision=0*, *method='common'*)
Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:

- `'common'` rounds either up or down
- `'ceil'` always rounds up
- `'floor'` always rounds down

If you don't specify a method `'common'` is used.

```
{{ 42.55|round }}
    -> 43.0
{{ 42.55|round(1, 'floor') }}
    -> 42.5
```

Note that even if rounded to 0 precision, a float is returned. If you need a real integer, pipe it through *int*:

```
{{ 42.55|round|int }}
    -> 43
```

safe(*value*)
Mark the value as safe which means that in an environment with automatic escaping enabled this variable will not be escaped.

select()
Filters a sequence of objects by appying a test to either the object or the attribute and only selecting the ones with the test succeeding.

Example usage:

```
{{ numbers|select("odd") }}
```

New in version 2.7.

selectattr()
Filters a sequence of objects by appying a test to either the object or the attribute and only selecting the ones with the test succeeding.

Example usage:

```
{{ users|selectattr("is_active") }}
{{ users|selectattr("email", "none") }}
```

New in version 2.7.

slice(*value*, *slices*, *fill_with=None*)
Slice an iterator and return a list of lists containing those items. Useful if you want to create a div containing three ul tags that represent columns:

```
<div class="columwrapper">
  {%- for column in items|slice(3) %}
    <ul class="column-{{ loop.index }}">
    {%- for item in column %}
      <li>{{ item }}</li>
    {%- endfor %}
    </ul>
```

```
  {%- endfor %}
</div>
```

If you pass it a second argument it's used to fill missing values on the last iteration.

sort(*value*, *reverse=False*, *case_sensitive=False*, *attribute=None*)
Sort an iterable. Per default it sorts ascending, if you pass it true as first argument it will reverse the sorting.

If the iterable is made of strings the third parameter can be used to control the case sensitiveness of the comparison which is disabled by default.

```
{% for item in iterable|sort %}
    ...
{% endfor %}
```

It is also possible to sort by an attribute (for example to sort by the date of an object) by specifying the *attribute* parameter:

```
{% for item in iterable|sort(attribute='date') %}
    ...
{% endfor %}
```

Changed in version 2.6: The *attribute* parameter was added.

string(*object*)
Make a string unicode if it isn't already. That way a markup string is not converted back to unicode.

striptags(*value*)
Strip SGML/XML tags and replace adjacent whitespace by one space.

sum(*iterable*, *attribute=None*, *start=0*)
Returns the sum of a sequence of numbers plus the value of parameter 'start' (which defaults to 0). When the sequence is empty it returns start.

It is also possible to sum up only certain attributes:

```
Total: {{ items|sum(attribute='price') }}
```

Changed in version 2.6: The *attribute* parameter was added to allow suming up over attributes. Also the *start* parameter was moved on to the right.

title(*s*)
Return a titlecased version of the value. I.e. words will start with uppercase letters, all remaining characters are lowercase.

trim(*value*)
Strip leading and trailing whitespace.

truncate(*s*, *length=255*, *killwords=False*, *end='...'*)
Return a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true` the filter will cut the text at length. Otherwise it will discard the last word. If the text was in fact truncated it will append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter.

```
{{ "foo bar"|truncate(5) }}
    -> "foo ..."
{{ "foo bar"|truncate(5, True) }}
    -> "foo b..."
```

upper(*s*)
Convert a value to uppercase.

urlencode(*value*)

Escape strings for use in URLs (uses UTF-8 encoding). It accepts both dictionaries and regular strings as well as pairwise iterables.

New in version 2.7.

urlize(*value*, *trim_url_limit=None*, *nofollow=False*)
Converts URLs in plain text into clickable links.

If you pass the filter an additional integer it will shorten the urls to that number. Also a third argument exists that makes the urls "nofollow":

```
{{ mytext|urlize(40, true) }}
    links are shortened to 40 chars and defined with rel="nofollow"
```

wordcount(*s*)
Count the words in that string.

wordwrap(*s*, *width=79*, *break_long_words=True*, *wrapstring=None*)
Return a copy of the string passed to the filter wrapped after 79 characters. You can override this default using the first parameter. If you set the second parameter to *false* Jinja will not split words apart if they are longer than *width*. By default, the newlines will be the default newlines for the environment, but this can be changed using the wrapstring keyword argument.

New in version 2.7: Added support for the *wrapstring* parameter.

xmlattr(*d*, *autospace=True*)
Create an SGML/XML attribute string based on the items in a dict. All values that are neither *none* nor *undefined* are automatically escaped:

```
<ul{{ {'class': 'my_list', 'missing': none,
        'id': 'list-%d'|format(variable)}|xmlattr }}>
...
</ul>
```

Results in something like this:

```
<ul class="my_list" id="list-42">
...
</ul>
```

As you can see it automatically prepends a space in front of the item if the filter returned something unless the second parameter is false.

内置测试清单

callable(*object*)
Return whether the object is callable (i.e., some kind of function). Note that classes are callable, as are instances with a __call__() method.

defined(*value*)
Return true if the variable is defined:

```
{% if variable is defined %}
    value of variable: {{ variable }}
{% else %}
    variable is not defined
{% endif %}
```

See the **default()** filter for a simple way to set undefined variables.

divisibleby(*value*, *num*)
Check if a variable is divisible by a number.

escaped(*value*)
Check if the value is escaped.

even(*value*)
Return true if the variable is even.

iterable(*value*)
Check if it's possible to iterate over an object.

lower(*value*)
Return true if the variable is lowercased.

mapping(*value*)
Return true if the object is a mapping (dict etc.).

New in version 2.6.

none(*value*)
Return true if the variable is none.

number(*value*)
Return true if the variable is a number.

odd(*value*)
Return true if the variable is odd.

sameas(*value*, *other*)
Check if an object points to the same memory address than another object:

```
{% if foo.attribute is sameas false %}
    the foo attribute really is the `False` singleton
{% endif %}
```

sequence(*value*)
Return true if the variable is a sequence. Sequences are variables that are iterable.

string(*value*)
Return true if the object is a string.

undefined(*value*)
Like **defined()** but the other way round.

upper(*value*)
Return true if the variable is uppercased.

全局函数清单

默认下，下面的函数在全局作用域中可用:

range([*start*], *stop*[, *step*])
返回一个包含整等差级数的列表。 range(i, j) 返回 [i, i+1, i+2, ..., j-1] ； 起始值（！）默认为 0 。当给定了公差，它决定了增长（或减小）。 例如 range(4) 返回 [0, 1, 2, 3] 。末端的值被丢弃了。这些是一个 4 元素 数组的有效索引值。

例如重复一个模板块多次来填充一个列表是有用的。想向你有一个 7 个用户的 列表，但你想要渲染三个空项目来用 CSS 强制指定高度:

```
<ul>
{% for user in users %}
    <li>{{ user.username }}</li>
{% endfor %}
{% for number in range(10 - users|count) %}
```

```
    <li class="empty"><span>...</span></li>
{% endfor %}
</ul>
```

lipsum(*n=5*, *html=True*, *min=20*, *max=100*)
在模板中生成 lorem ipsum 乱数假文。默认会生成 5 段 HTML，每段在 20 到 100 词之间。如果 HTML 被禁用，会返回常规文本。这在测试布局时生成简单内容时很有 用。

dict(***items*)
方便的字典字面量替代品。`{'foo' : 'bar'}` 与 `dict(foo=bar)` 等价。

class **cycler**(**items*)
周期计允许你在若干个值中循环，类似 *loop.cycle* 的工作方式。不同于 *loop.cycle* 的是，无论如何你都可以在循环外或在多重循环中使用它。

比如如果你想要显示一个文件夹和文件列表，且文件夹在上，它们在同一个列表中且 行颜色是交替的。

下面的例子展示了如何使用周期计:

```
{% set row_class = cycler('odd', 'even') %}
<ul class="browser">
{% for folder in folders %}
  <li class="folder {{ row_class.next() }}">{{ folder|e }}</li>
{% endfor %}
{% for filename in files %}
  <li class="file {{ row_class.next() }}">{{ filename|e }}</li>
{% endfor %}
</ul>
```

周期计有下面的属性和方法:

reset()
重置周期计到第一个项。

next()
返回当前项并跳转到下一个。

current
返回当前项。.

New in version 2.1.

class **joiner**(*sep=', '*)
一个小巧的辅助函数用于“连接”多个节。连接器接受一个字符串，每次被调用时返回 那个字符串，除了第一次调用时返回一个空字符串。你可以使用它来连接:

```
{% set pipe = joiner("|") %}
{% if categories %} {{ pipe() }}
    Categories: {{ categories|join(", ") }}
{% endif %}
{% if author %} {{ pipe() }}
    Author: {{ author() }}
{% endif %}
{% if can_edit %} {{ pipe() }}
    <a href="?action=edit">Edit</a>
{% endif %}
```

New in version 2.1.

扩展

下面的几节涵盖了可能被应用启用的 Jinja2 内置的扩展。应用也可以提供进一步 的扩展，但这不会在此描述。会有独立的文档来解释那种情况下的扩展。

i18n

如果启用来 i18n 扩展，可以把模板中的部分标记为可译的。标记一个段为可译的，可 以使用 *trans*:

```
<p>{% trans %}Hello {{ user }}!{% endtrans %}</p>
```

要翻译一个模板表达式——比如使用模板过滤器或访问对象的属性——你需要绑定表达式到 一个名称来在翻译块中使用:

```
<p>{% trans user=user.username %}Hello {{ user }}!{% endtrans %}</p>
```

如果你需要在 *trans* 标签中绑定一个以上的表达式，用逗号来分割（ , ）:

```
{% trans book_title=book.title, author=author.name %}
This is {{ book_title }} by {{ author }}
{% endtrans %}
```

在翻译块中不允许使用语句，只能使用变量标签。

为表示复数，在 *trans* 和 *endtrans* 之间用 *pluralize* 标签同时指定单数和复 数形式:

```
{% trans count=list|length %}
There is {{ count }} {{ name }} object.
{% pluralize %}
There are {{ count }} {{ name }} objects.
{% endtrans %}
```

默认情况下块中的第一个变量用于决定使用单数还是复数。如果这不奏效，你可以指定 用于复数的名称作为 *pluralize* 的参数:

```
{% trans ..., user_count=users|length %}...
{% pluralize user_count %}...{% endtrans %}
```

也可以翻译表达式中的字符串。为此，有三个函数:

_ *gettext*: 翻译一个单数字符串 - *ngettext*: 翻译一个复数字符串 - _: *gettext* 的别名

例如你可以容易地这样打印一个已翻译的字符串:

```
{{ _('Hello World!') }}
```

你可以使用 *format* 过滤器来使用占位符:

```
{{ _('Hello %(user)s!')|format(user=user.username) }}
```

因为其它语言可能不会用同样的顺序使用词汇，要使用多个占位符，应始终用字符 串参数传给 *format* 。

Changed in version 2.5.

如果激活了新样式的 gettext 调用（ 新样式 *Gettext* ），使用占位符 会更加简单:

```
{{ gettext('Hello World!') }}
{{ gettext('Hello %(name)s!', name='World') }}
{{ ngettext('%(num)d apple', '%(num)d apples', apples|count) }}
```

注意 *ngettext* 函数的格式化字符串自动接受 *num* 参数作为计数作为附加的 常规参数。

表达式语句

如果加载了表达式语句扩展，一个名为 *do* 的扩展即可用。它工作几乎如同常规的变量 表达式（ `{{ ... }}` ），只是它不打印任何东西。这可以用于修改列表:

```
{% do navigation.append('a string') %}
```

循环控制

如果应用启用来 循环控制，则可以在循环中使用 *break* 和 *continue* 。到达 *break* 时，循环终止。到达 *continue* 时，当前处理会终止并 从下一次迭代继续。

这个循环每两项跳过一次:

```
{% for user in users %}
    {%- if loop.index is even %}{% continue %}{% endif %}
    ...
{% endfor %}
```

同样，这个循环 10 次迭代之后会终止处理:

```
{% for user in users %}
    {%- if loop.index >= 10 %}{% break %}{% endif %}
{%- endfor %}
```

With 语句

New in version 2.3.

如果应用启用了 *With* 语句，将允许在模板中使用 *with* 关键 字。这使得创建一个新的内作用域。这个作用域中的变量在外部是不可见的。

With 用法简介:

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}           foo is 42 here
{% endwith %}
foo is not visible here any longer
```

因为在作用域的开始设置变量很常见，你可以在 with 语句里这么做。下面的两 个例子是等价的:

```
{% with foo = 42 %}
    {{ foo }}
{% endwith %}

{% with %}
    {% set foo = 42 %}
    {{ foo }}
{% endwith %}
```

自动转义扩展

New in version 2.4.

如果你的应用程序设置了 自动转义扩展，你就可以在模版中开启或者关闭自动转义。

例子:

```
{% autoescape true %}
自动转义在这块文本中是开启的。
{% endautoescape %}

{% autoescape false %}
自动转义在这块文本中是关闭的。
{% endautoescape %}
```

在 *endautoescape* 标签之后，自动转义的行为将回到与之前相同的状态。