Jinja2
中文文档

介绍

> 原文出处：http://docs.jinkan.org/docs/jinja2/

Jinja2 是一个现代的，设计者友好的，仿照 Django 模板的 Python 模板语言。 它速度快，被广泛使用，并且提供了可选的沙箱模板执行环境保证安全:

```
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

特性:

- 沙箱中执行
- 强大的 HTML 自动转义系统保护系统免受 XSS
- 模板继承
- 及时编译最优的 python 代码
- 可选提前编译模板的时间
- 易于调试。异常的行数直接指向模板中的对应行。
- 可配置的语法

如果你接触过其它的基于文本的模板语言，比如 Smarty 或 Django ，那么 Jinja2 会让你有 宾至如归的感觉。Jinja2 通过坚持 Python 原则来保证对设计者和开发者友好，为模板环境添加有帮助的功能。

预备知识

Jinja2 需要至少 **Python 2.4** 版本来运行。此外，如果你使用 Python 2.4 ，一个可 以创建 python 扩展的可用的 C 编译器会为调试器安装。

如果你没有一个可用的 C 编译器，并且你视图安装带调试支持的源码版本，你会得到一个 编译器错误。 If you don’ t have a working C-compiler and you are trying to install the source

安装

条条大路通 Jinja2 。如果你不确定怎么做，用 Python egg 或 tarball 吧。

作为一个 Python egg（通过 easy_install）

你可以用 easy_install 或 pip 安装最新的版本的 Jinja2:

```
sudo easy_install Jinja2
sudo pip install Jinja2
```

这会在你的 Python 安装中的 site-packages 目录安装一个 Jinja2 egg 。

（如果你在 Windows 的命令行中安装，省略 sudo 并且确保你用管理员权限运行 命令行）

从 tarball 版本安装

1. 从 download page 下载最新的 tarball
2. 解包 tarball
3. sudo python setup.py install

注意这需要你已经安装了 setuptools 或 distribute ，首选后者。

这会在你 Python 安装的 site-packages 目录安装 Jinja2 。

安装开发版本

1. 安装 git
2. git clone git://github.com/mitsuhiko/jinja2.git
3. cd jinja2
4. ln -s jinja2 /usr/lib/python2.X/site-packages

作为第四步的替代选择，你也可以执行 python setup.py develop ，这会通过 disbribute 在开发模式下安装包。这样也有编译 C 扩展的优势。

加速 MarkupSafe

从 2.5.1 开始， Jinja2 会检查是否安装 MarkupSafe 模块。如果它找到了， 它会用这个模块的 Markup 类来代替自带的。 MarkupSafe 替换 Jinja2 中附带的 老的加速模块，其优势在于更好的安装脚本，自动试图安装 C 的版本并在不可行时 漂亮地退化到纯 Python 实现的版本。

MarkupSafe 的 C 实现要快得多，并推荐用于 Jinja2 自动转义。

启用调试支持模块

默认 Jinja2 不会编译调试支持模块。如果你没有 Python 头文件或可用的编译器， 启用它会失败。这当你在 Windows 上安装 Jinja2 是很常见的情况。

由于调试模式只对 Python 2.4 是必要的，所以你不需要这么做，除非你在运行 2.4:

```
sudo python setup.py --with-debugsupport install
```

基本 API 使用

本节简要介绍 Jinja2 模板的 Python API 。

最基本的方式就是通过 Template 创建一个模板并渲染它。 如果你的模板不是从字符串加载，而是文件系统或别的数据源，无论如何这都不 是推荐的方式:

```
>>> from jinja2 import Template
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
u'Hello John Doe!'
```

通过创建一个 Template 的实例，你会得到一个新的模板对象，提供一 个名为 render() 的方法，该方法在有字典或关键字参数时调用 扩充模板。字典或关键字参数会被传递到模板，即模板“上下文”。

如你所见， Jinja2 内部使用 unicode 并且返回值也是 unicode 字符串。所以确 保你的应用里也确实使用 unicode 。

实验性的 Python 3 支持

Jinja 2.3 带来 Python 3 的实验性支持。这意味着在新版本上，所有的单元测试 都会通过，但是仍有一些小 bug 和不一致的行为。如果你发现任何 bug ，请向 Jinja bug tracker提供反馈。

也请记住本文档是为 Python 2 编撰的，你会需要手动把示例代码转换为 Python 3 的语法。

API

本文档描述 Jinja2 的 API 而不是模板语言。这对实现模板接口，而非创建 Jinja2 模板，是最有用的参考

基础

Jinja2 使用一个名为 Environment 的中心对象。这个类的实例用于存储配 置、全局对象，并用于从文件系统或其它位置加载模板。即使你通过:class:Template 类的构造函数用字符串创建模板，也会为你自动创建一个环境，尽管是共享的。

大多数应用在应用初始化时创建一个 Environment 对象，并用它加载模板。 在某些情况下，如果使用多份配置，使用并列的多个环境无论如何是有用的。

配置 Jinja2 为你的应用加载文档的最简单方式看起来大概是这样:

```
from jinja2 import Environment, PackageLoader
env = Environment(loader=PackageLoader('yourapplication', 'templates'))
```

这会创建一个默认设定下的模板环境和一个在 yourapplication python 包中的 templates文件夹中寻找模板的加载器。多个加载器是可用的，如果你需要从 数据库或其它资源加载模板，你也可以自己写一个。

你只需要调用 get_template() 方法从这个环境中加载模板，并会返回已加载的 Template:

```
template = env.get_template('mytemplate.html')
```

用若干变量来渲染它，调用 render() 方法:

```
print template.render(the='variables', go='here')
```

使用一个模板加载器，而不是向 Template 或 Environment.from_string() 传递字符串，有许多好处。除了使用上便利， 也使得模板继承成为可能。

Unicode

Jinja2 内部使用 Unicode ，这意味着你需要向渲染函数传递 Unicode 对象或只包含 ASCII 字符的字符串。此外，换行符按照默认 UNIX 风格规定行序列结束（ \n ）。

Python 2.x 支持两种表示字符串对象的方法。一种是 str 类型，另一种是 unicode 类型，它们都继承

于 basestring 类型。不幸的是，默认的 str 不 应该用于存储基于文本的信息，除非只用到 ASCII 字符。在 Python 2.6 中，可以 在模块层指定 unicode 为默认值，而在 Python 3 中会是默认值。

要显式使用一个 Unicode 字符串，你需要给字符串字面量加上 u 前缀： u'Hänsel undGretel sagen Hallo' 。这样 Python 会用当前模块的字符编码来 解码字符串，来把字符串存储为 Unicode 。如果没有指定编码，默认是 ASCII ， 这意味着你不能使用任何非 ASCII 的标识符。

在使用 Unicode 字面量的 Python 模块的首行或第二行添加下面的注释，来妥善设 置模块编码:

```
# -*- coding: utf-8 -*-
```

我们推荐为 Python 模块和模板使用 utf-8 编码，因为在 utf-8 中，可以表示 Unicode 中的每个字符，并且向后兼容 ASCII 。对于 Jinja2 ，模板的默认编码 假定为 utf-8 。

用 Jinja2 来处理非 Unicode 数据是不可能的。这是因为 Jinja2 已经在语言层 使用了 Unicode 。例如 Jinja2 在表达式中把不间断空格视为有效的空格，这需要 获悉编码或操作一个 Unicode 字符串。

关于 Python 中 Unicode 的更多细节，请阅读完善的 Unicode documentation 。

另一件重要的事情是 Jinja2 如何处理模板中的字符串字面量。原生实现会对所有 字符串字面量使用 Unicode ，但在过去这是有问题的，因为一些库显式地检查它 们的类型是否为str 。例如 datetime.strftime 不接受 Unicode 参数。 为了不彻底破坏它， Jinja2 对只有 ASCII 的字符串返回 str，而对其它返回 unicode:

```
>>> m = Template(u"{% set a, b = 'foo', 'föö' %}").module
>>> m.a
'foo'
>>> m.b
u'f\xf6\xf6'
```

高层 API

高层 API 即是你会在应用中用于加载并渲染模板的 API 。 *低层 API* 相反，只在你想深入挖掘 Jinja2 或 *开发扩展* 时有用。

class jinja2.Environment([*options*])

The core component of Jinja is the Environment. It contains important shared variables like configuration, filters, tests, globals and others. Instances of this class may be modified if they are not shared and if no template was loaded so far. Modifications on environments after the first template was loaded will lead to surprising effects and undefined behavior.

Here the possible initialization parameters:

> block_start_string
>
> The string marking the begin of a block. Defaults to '{%'.
>
> block_end_string
>
> The string marking the end of a block. Defaults to '%}'.
>
> variable_start_string
>
> The string marking the begin of a print statement. Defaults to '{{'.
>
> variable_end_string
>
> The string marking the end of a print statement. Defaults to '}}'.
>
> comment_start_string
>
> The string marking the begin of a comment. Defaults to '{#'.
>
> comment_end_string
>
> The string marking the end of a comment. Defaults to '#}'.
>
> line_statement_prefix
>
> If given and a string, this will be used as prefix for line based statements. See also *行语句*.
>
> line_comment_prefix
>
> If given and a string, this will be used as prefix for line based based comments. See also *行语句*.
>
> New in version 2.2.
>
> trim_blocks
>
> If this is set to True the first newline after a block is removed (block, not variable tag!). Defaults to False.
>
> lstrip_blocks
>
> If this is set to True leading spaces and tabs are stripped from the start of a line to a block. Defaults to False.
>
> newline_sequence
>
> The sequence that starts a newline. Must be one of '\r', '\n' or '\r\n'. The default is '\n' which is a useful default for Linux and OS X systems as well as web applications.
>
> keep_trailing_newline
>
> Preserve the trailing newline when rendering templates. The default is False, which causes a single newline, if present, to be stripped from the end of the template.
>
> New in version 2.7.
>
> extensions

> List of Jinja extensions to use. This can either be import paths as strings or extension classes. For more information have a look at *the extensions documentation*.
>
> optimized
>
> should the optimizer be enabled? Default is True.
>
> undefined
>
> Undefined or a subclass of it that is used to represent undefined values in the template.
>
> finalize
>
> A callable that can be used to process the result of a variable expression before it is output. For example one can convert None implicitly into an empty string here.
>
> autoescape
>
> If set to true the XML/HTML autoescaping feature is enabled by default. For more details about auto escaping see Markup. As of Jinja 2.4 this can also be a callable that is passed the template name and has to return True or Falsedepending on autoescape should be enabled by default.
>
> Changed in version 2.4: autoescape can now be a function
>
> loader
>
> The template loader for this environment.
>
> cache_size
>
> The size of the cache. Per default this is 50 which means that if more than 50 templates are loaded the loader will clean out the least recently used template. If the cache size is set to 0 templates are recompiled all the time, if the cache size is -1 the cache will not be cleaned.
>
> auto_reload
>
> Some loaders load templates from locations where the template sources may change (ie: file system or database). If auto_reload is set to True (default) every time a template is requested the loader checks if the source changed and if yes, it will reload the template. For higher performance it’ s possible to disable that.
>
> bytecode_cache
>
> If set to a bytecode cache object, this object will provide a cache for the internal Jinja bytecode so that templates don’ t have to be parsed if they were not changed.
>
> See *字节码缓存* for more information.

shared

如果模板通过 Template 构造函数创建，会自动创建一个环境。这 些环境被创建为共享的环境，这意味着多个模板拥有相同的匿名环境。对所有 模板共享环境，这个属性为 True ，反之为 False 。

sandboxed

如果环境在沙箱中，这个属性为 True 。沙箱模式见文档中的SandboxedEnvironment 。

filters

该环境的过滤器字典。只要没有加载过模板，添加新过滤器或删除旧的都是 安全的。自定义过滤器见 *自定义过滤器* 。有效的过滤器名称见 *标识符的说明* 。

tests

该环境的测试函数字典。只要没有加载过模板，修改这个字典都是安全的。 自定义测试见 see *自定义测试* 。有效的测试名见 *标识符的说明* 。

globals

一个全局变量字典。这些变量在模板中总是可用。只要没有加载过模板，修 改这个字典都是安全的。更多细节见 *全局命名空间* 。有效的 对象名见 *标识符的说明*。

overlay([*options*])

Create a new overlay environment that shares all the data with the current environment except of cache and the overridden attributes. Extensions cannot be removed for an overlayed environment. An overlayed environment automatically gets all the extensions of the environment it is linked to plus optional extra extensions.

Creating overlays should happen after the initial environment was set up completely. Not all attributes are truly linked, some are just copied over so modifications on the original environment may not shine through.

undefined([*hint*, *obj*, *name*, *exc*])

为 name 创建一个新 Undefined 对象。这对可能为某些操作返回 未定义对象过滤器和函数有用。除了 hint ，为了良好的可读性，所有参数 应该作为关键字参数传入。如果提供了 hint ，它被用作异常的错误消息， 否则错误信息会由 obj 和name 自动生成。 exc 为生成未定义对象而 不允许未定义的对象时抛出的异常。默认的异常是 UndefinedError 。 如果提供了 hint ， name 会被发送。

创建一个未定义对象的最常用方法是只提供名称:

```
return environment.undefined(name='some_name')
```

这意味着名称 some_name 未被定义。如果名称来自一个对象的属性，把 持有它的对象告知未定义对象对丰富错误消息很有意义:

```
if not hasattr(obj, 'attr'):
    return environment.undefined(obj=obj, name='attr')
```

更复杂的例子中，你可以提供一个 hint 。例如 first() 过滤器 用这种方法创建一个未定义对象:

```
return environment.undefined('no first item, sequence was empty')
```

如果 name 或 obj 是已知的（比如访问了了一个属性），它应该传递给 未定义对象，即使提供了自定义的 hint 。这让未定义对象有可能增强错误 消息。

add_extension(*extension*)

Adds an extension after the environment was created.

New in version 2.5.

compile_expression(*source*, *undefined_to_none=True*)

A handy helper method that returns a callable that accepts keyword arguments that appear as variables in the expression. If called it returns the result of the expression.

This is useful if applications want to use the same rules as Jinja in template “configuration files” or similar situations.

Example usage:

```
>>> env = Environment()
>>> expr = env.compile_expression('foo == 42')
>>> expr(foo=23)
False
>>> expr(foo=42)
True
```

Per default the return value is converted to None if the expression returns an undefined value. This can be changed by setting undefined_to_none to False.

```
>>> env.compile_expression('var')() is None
True
>>> env.compile_expression('var', undefined_to_none=False)()
Undefined
```

New in version 2.1.

compile_templates(*target*, *extensions=None*, *filter_func=None*, *zip='deflated'*,*log_function=None*, *ignore_errors=True*, *py_compile=False*)

Finds all the templates the loader can find, compiles them and stores them intarget.
If zip is None, instead of in a zipfile, the templates will be will be stored in a directory. By default a deflate zip algorithm is used, to switch to the stored algorithm, zip can be set to 'stored'.

extensions and filter_func are passed to list_templates(). Each template returned will be compiled to the target folder or zipfile.

By default template compilation errors are ignored. In case a log function is provided, errors are logged. If you want template syntax errors to abort the compilation you can set ignore_errors to False and you will get an exception on syntax errors.

If py_compile is set to True .pyc files will be written to the target instead of standard .py files. This flag does not do anything on pypy and Python 3 where pyc files are not picked up by itself and don’ t give much benefit.

New in version 2.4.

extend(*attributes*)

Add the items to the instance of the environment if they do not exist yet. This is used by *extensions* to register callbacks and configuration values without breaking inheritance.

from_string(*source*, *globals=None*, *template_class=None*)

Load a template from a string. This parses the source given and returns a Template object.

get_or_select_template(*template_name_or_list*, *parent=None*, *globals=None*)

Does a typecheck and dispatches to select_template() if an iterable of template names is given, otherwise to get_template().

New in version 2.3.

get_template(*name*, *parent=None*, *globals=None*)

Load a template from the loader. If a loader is configured this method ask the loader for the template and returns a Template. If the parent parameter is notNone, join_path() is called to get the real template name before loading.

The globals parameter can be used to provide template wide globals. These variables are available in the context at render time.

If the template does not exist a TemplateNotFound exception is raised.

Changed in version 2.4: If name is a Template object it is returned from the function unchanged.

join_path(*template*, *parent*)

Join a template with the parent. By default all the lookups are relative to the loader root so this method returns the template parameter unchanged, but if the paths should be relative to the parent template, this function can be used to calculate the real template name.

Subclasses may override this method and implement template path joining here.

list_templates(*extensions=None*, *filter_func=None*)

Returns a list of templates for this environment. This requires that the loader supports the loader’ s list_templates() method.

If there are other files in the template folder besides the actual templates, the returned list can be filtered. There are two ways: either extensions is set to a list of file extensions for templates, or a filter_func can be provided which is a callable that is passed a template name and should return True if it should end up in the result list.

If the loader does not support that, a TypeError is raised.

New in version 2.4.

select_template(*names*, *parent=None*, *globals=None*)

Works like get_template() but tries a number of templates before it fails. If it cannot find any of the templates, it will raise a TemplatesNotFound exception.

New in version 2.3.

Changed in version 2.4: If names contains a Template object it is returned from the function unchanged.

class jinja2.Template

The central template object. This class represents a compiled template and is used to evaluate it.

Normally the template object is generated from an Environment but it also has a constructor tha makes it possible to create a template instance directly using the constructor. It takes the same arguments as the environment constructor but it’ s not possible to specify a loader.

Every template object has a few methods and members that are guaranteed to exist. However it’ s important that a template object should be considered immutable. Modifications on the object are not supported.

Template objects created from the constructor rather than an environment do have an environment attribute that points to a temporary environment that is probably shared with other templates created with the constructor and compatible settings.

```
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
u'Hello John Doe!'
```

```
>>> stream = template.stream(name='John Doe')
>>> stream.next()
u'Hello John Doe!'
>>> stream.next()
Traceback (most recent call last):
  ...
StopIteration
```

globals

该模板的全局变量字典。修改这个字典是不安全的，因为它可能与其它模板或 加载这个模板的环境共享。

name

模板的加载名。如果模板从字符串加载，这个值为 None 。

filename

模板在文件系统上的文件名，如果没有从文件系统加载，这个值为 None 。

render([*context*])

This method accepts the same arguments as the dict constructor: A dict, a dict subclass or some keyword arguments. If no arguments are given the context will be empty. These two calls do the same:

```
template.render(knights='that say nih')
template.render({'knights': 'that say nih'})
```

This will return the rendered template as unicode string.

generate([*context*])

For very large templates it can be useful to not render the whole template at once but evaluate each statement after another and yield piece for piece. This method basically does exactly that and returns a generator that yields one item after another as unicode strings.

It accepts the same arguments as render().

stream([*context*])

Works exactly like generate() but returns a TemplateStream.

make_module(*vars=None*, *shared=False*, *locals=None*)

This method works like the module attribute when called without arguments but it will evaluate the template on every call rather than caching it. It’ s also possible to provide a dict which is then used as context. The arguments are the same as for the new_context() method.

module

The template as module. This is used for imports in the template runtime but is also useful if one wants to access exported template variables from the Python layer:

```
>>> t = Template('{% macro foo() %}42{% endmacro %}23')
>>> unicode(t.module)
u'23'
>>> t.module.foo()
u'42'
```

class jinja2.environment.TemplateStream

A template stream works pretty much like an ordinary python generator but it can buffer multiple items to reduce the number of total iterations. Per default the output is unbuffered which means that for every unbuffered instruction in the template one unicode string is yielded.

If buffering is enabled with a buffer size of 5, five items are combined into a new unicode string. This is mainly useful if you are streaming big templates to a client via WSGI which flushes after each iteration.

disable_buffering()

Disable the output buffering.

dump(*fp*, *encoding=None*, *errors='strict'*)

Dump the complete stream into a file or file-like object. Per default unicode strings are written, if you want to encode before writing specify an encoding.

Example usage:

```
Template('Hello {{ name }}!').stream(name='foo').dump('hello.html')
```

enable_buffering(*size=5*)¶

Enable buffering. Buffer size items before yielding them.

自动转义

New in version 2.4.

从 Jinja 2.4 开始，自动转义的首选途径就是启用 *自动转义扩展* 并为自动转义配置一个合适的默认值。这使得在单个模板基础上开关自动转义成为 可能（比如 HTML 对 文本）

这里推荐为以 .html 、 .htm 、 .xml 以及 .xhtml 的模板开启 自动转义 ，并对所有其它扩展名禁用:

```
def guess_autoescape(template_name):
    if template_name is None or '.' not in template_name:
        return False
    ext = template_name.rsplit('.', 1)[1]
    return ext in ('html', 'htm', 'xml')

env = Environment(autoescape=guess_autoescape,
                  loader=PackageLoader('mypackage'),
                  extensions=['jinja2.ext.autoescape'])
```

假设实现一个自动转义函数，确保你也视 None 为有效模板名接受。这会在从字符 串生成模板时传递。

可以用 autoescape 块在模板内临时地更改这种行为。（见 *自动转义扩展* ）。

标识符的说明

Jinja2 使用正规的 Python 2.x 命名规则。有效的标识符必须匹配 [a-zA-Z_][a-zA-Z0-9_]*。事实上，当前不允许非 ASCII 字符。这个限制可能 会在 Python 3 充分规定 unicode 标识符后消失。

过滤器和测试会在独立的命名空间中查找，与标识符语法有细微区别。过滤器和测 试可以包含点，用于按主题给过滤器和测试分组。例如，把一个名为 to.unicode 的函数添加到过滤器字典是完全有效的。过滤器和测试标识符的正则表达式是 [a-zA-Z_][a-zA-Z0-9_]*(.[a-zA-Z_][a-zA-Z0-9_]*)* 。

未定义类型

这些类可以用作未定义类型。 Environment 的构造函数接受一个可以是 那些类或一个Undefined 的自定义子类的 undefined 参数。无论何时， 这些对象创建或返回时，模板引擎都不能查出其名称或访问其属性。未定义值上的 某些操作之后是允许的，而其它的会失败。

最接近常规 Python 行为的是 StrictUndefined ，如果它是一个未定义对象， 它不允许除了测试之外的一切操作。

class jinja2.Undefined

The default undefined type. This undefined type can be printed and iterated over, but every other access will raise an UndefinedError:

```
>>> foo = Undefined(name='foo')
>>> str(foo)
''
>>> not foo
True
>>> foo + 42
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
```

_undefined_hint

None 或给未定义对象的错误消息 unicode 字符串。

_undefined_obj

None 或引起未定义对象创建的对象（例如一个属性不存在）。

_undefined_name

未定义变量/属性的名称，如果没有此类信息，留为 None 。

_undefined_exception

未定义对象想要抛出的异常。这通常是 UndefinedError 或 SecurityError 之一。

_fail_with_undefined_error(***args**, kwargs*)

参数任意，调用这个方法时会抛出带有由未定义对象上存储的未定义 hint 生成的错误信息的 _undefined_exception 异常。

class jinja2.DebugUndefined

An undefined that returns the debug info when printed.

```
>>> foo = DebugUndefined(name='foo')
>>> str(foo)
'{{ foo }}'
>>> not foo
True
>>> foo + 42
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
```

class jinja2.StrictUndefined

An undefined that barks on print and iteration as well as boolean tests and all kinds of comparisons. In other words: you can do nothing with it except checking if it' s defined using

the defined test.

```
>>> foo = StrictUndefined(name='foo')
>>> str(foo)
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
>>> not foo
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
>>> foo + 42
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
```

未定义对象由调用 undefined 创建。

实现

Undefined 对象通过重载特殊的 **underscore** 方法实现。例如 默认的 Undefined 类实现 **unicode** 为返回一个空字符串，但 **int** 和其它会始终抛出异常。你可以自己通过返回 0 实现转换为 int:

```
class NullUndefined(Undefined):
    def __int__(self):
        return 0
    def __float__(self):
        return 0.0
```

要禁用一个方法，重载它并抛出 _undefined_exception 。因 为这在未定义对象中非常常用，未定义对象有辅助方法 _fail_with_undefined_error() 自动抛出错误。这里的一个类 工作类似正规的 Undefined ，但它在迭代时阻塞:

```
class NonIterableUndefined(Undefined):
    __iter__ = Undefined._fail_with_undefined_error
```

上下文

class jinja2.runtime.Context

The template context holds the variables of a template. It stores the values passed to the template and also the names the template exports. Creating instances is neither supported nor useful as it’ s created automatically at various stages of the template evaluation and should not be created by hand.

The context is immutable. Modifications on parent **must not** happen and modifications on vars are allowed from generated template code only. Template filters and global functions marked as contextfunction()s get the active context passed as first argument and are allowed to access the context read-only.

The template context supports read only dict operations (get, keys, values, items,iterkeys, itervalues, iteritems, **getitem**, **contains**). Additionally there is aresolve() method that doesn’ t fail with a KeyError but returns an Undefined object for missing variables.

parent

一个模板查找的只读全局变量的词典。这些变量可能来自另一个 Context ，或是Environment.globals ，或是 Template.globals ，或指向一个由全局变量和传递到渲染函数的变 量联立的字典。它一定不能被修改。

vars

模板局域变量。这个列表包含环境和来自 parent 范围的上下文函数 以及局域修改和从模板中导出的变量。模板会在模板求值时修改这个字典， 但过滤器和上下文函数不允许修改它。

environment

加载该模板的环境

exported_vars

这设定了所有模板导出量的名称。名称对应的值在 vars 字典中。 可以用get_exported() 获取一份导出变量的拷贝字典。

name

拥有此上下文的模板的载入名。

blocks

模板中块当前映射的字典。字典中的键是块名称，值是注册的块的列表。每个 列表的最后一项是当前活动的块（继承链中最新的）。

eval_ctx

当前的 *求值上下文* 。

call(*callable*, **args,** kwargs*)

Call the callable with the arguments and keyword arguments provided but inject the active context or environment as first argument if the callable is

acontextfunction() or environmentfunction().

get_all()

Return a copy of the complete context as dict including the exported variables.

get_exported()

Get a new dict with the exported variables.

resolve(*key*)

Looks up a variable like **getitem** or get but returns an Undefined object with the name of the name looked up.

实现

Python frame 中的局域变量在函数中是不可变的，出于同样的原因，上下文是不可 变的。 Jinja2 和 Python 都不把上下文/ frame 作为变量的数据存储，而只作为 主要的数据源。

当模板访问一个模板中没有定义的变量时， Jinja2 在上下文中查找变量，此后， 这个变量被视为其是在模板中定义得一样。

加载器

加载器负责从诸如文件系统的资源加载模板。环境会把编译的模块像 Python 的sys.modules 一样保持在内存中。与 sys.models 不同，无论如何这个 缓存默认有大小限制，且模板会自动重新加载。 所有的加载器都是 BaseLoader 的子类。如果你想要创建自己的加载器，继 承 BaseLoader 并重载 get_source 。

class jinja2.BaseLoader

Baseclass for all loaders. Subclass this and override get_source to implement a custom loading mechanism. The environment provides a get_template method that calls the loader’ s load method to get the Template object.

A very basic example for a loader that looks up templates on the file system could look like this:

```
from jinja2 import BaseLoader, TemplateNotFound
from os.path import join, exists, getmtime

class MyLoader(BaseLoader):

    def __init__(self, path):
        self.path = path

    def get_source(self, environment, template):
        path = join(self.path, template)
        if not exists(path):
            raise TemplateNotFound(template)
        mtime = getmtime(path)
        with file(path) as f:
            source = f.read().decode('utf-8')
        return source, path, lambda: mtime == getmtime(path)
```

get_source(*environment*, *template*)

Get the template source, filename and reload helper for a template. It’ s passed the environment and template name and has to return a tuple in the form(source, filename, uptodate) or raise a TemplateNotFound error if it can’ t locate the template.

The source part of the returned tuple must be the source of the template as unicode string or a ASCII bytestring. The filename should be the name of the file on the filesystem if it was loaded from there, otherwise None. The filename is used by python for the tracebacks if no loader extension is used.

The last item in the tuple is the uptodate function. If auto reloading is enabled it’ s always called to check if the template changed. No arguments are passed so the function must store the old state somewhere (for example in a closure). If it returns False the template will be reloaded.

load(*environment*, *name*, *globals=None*)

Loads a template. This method looks up the template in the cache or loads one by calling get_source(). Subclasses should not override this method as loaders working on collections of other loaders (such as PrefixLoader or ChoiceLoader) will not call this method but get_source directly.

这里有一个 Jinja2 提供的内置加载器的列表:

class jinja2.FileSystemLoader(*searchpath*, *encoding='utf-8'*)

Loads templates from the file system. This loader can find templates in folders on the file system and is the preferred way to load them.

The loader takes the path to the templates as string, or if multiple locations are wanted a list of

them which is then looked up in the given order:

```
>>> loader = FileSystemLoader('/path/to/templates')
>>> loader = FileSystemLoader(['/path/to/templates', '/other/path'])
```

Per default the template encoding is 'utf-8' which can be changed by setting theencoding parameter to something else.

class jinja2.PackageLoader(*package_name*, *package_path='templates'*, *encoding='utf-8'*)

Load templates from python eggs or packages. It is constructed with the name of the python package and the path to the templates in that package:

```
loader = PackageLoader('mypackage', 'views')
```

If the package path is not given, 'templates' is assumed.

Per default the template encoding is 'utf-8' which can be changed by setting theencoding parameter to something else. Due to the nature of eggs it’ s only possible to reload templates if the package was loaded from the file system and not a zip file.

class jinja2.DictLoader(*mapping*)

Loads a template from a python dict. It’ s passed a dict of unicode strings bound to template names. This loader is useful for unittesting:

```
>>> loader = DictLoader({'index.html': 'source here'})
```

Because auto reloading is rarely useful this is disabled per default.

class jinja2.FunctionLoader(*load_func*)

A loader that is passed a function which does the loading. The function becomes the name of the template passed and has to return either an unicode string with the template source, a tuple in the form (source, filename, uptodatefunc) or None if the template does not exist.

```
>>> def load_template(name):
...     if name == 'index.html':
...         return '...'
...
>>> loader = FunctionLoader(load_template)
```

The uptodatefunc is a function that is called if autoreload is enabled and has to returnTrue if the template is still up to date. For more details have a look atBaseLoader.get_source() which has the same return value.

class jinja2.PrefixLoader(*mapping*, *delimiter='/'*)

A loader that is passed a dict of loaders where each loader is bound to a prefix. The prefix is delimited from the template by a slash per default, which can be changed by setting the delimiter argument to something else:

```
loader = PrefixLoader({
    'app1':     PackageLoader('mypackage.app1'),
    'app2':     PackageLoader('mypackage.app2')
})
```

By loading 'app1/index.html' the file from the app1 package is loaded, by loading'app2/index.html' the file from the second.

class jinja2.ChoiceLoader(*loaders*)

This loader works like the PrefixLoader just that no prefix is specified. If a template could not be found by one loader the next one is tried.

```
>>> loader = ChoiceLoader([
...     FileSystemLoader('/path/to/user/templates'),
...     FileSystemLoader('/path/to/system/templates')
... ])
```

This is useful if you want to allow users to override builtin templates from a different location.

class jinja2.ModuleLoader(*path*)

This loader loads templates from precompiled templates.

Example usage:

```
>>> loader = ChoiceLoader([
...     ModuleLoader('/path/to/compiled/templates'),
...     FileSystemLoader('/path/to/templates')
... ])
```

Templates can be precompiled with Environment.compile_templates().

字节码缓存

Jinja 2.1 和更高的版本支持外部字节码缓存。字节码缓存使得在首次使用时把生成的字节码 存储到文件系统或其它位置来避免处理模板。

这在当你有一个在首个应用初始化的 web 应用， Jinja 一次性编译大量模板拖慢应用时尤其 有用。

要使用字节码缓存，把它实例化并传给 Environment 。

class jinja2.BytecodeCache

To implement your own bytecode cache you have to subclass this class and overrideload_bytecode() and dump_bytecode(). Both of these methods are passed a Bucket.

A very basic bytecode cache that saves the bytecode on the file system:

```
from os import path

class MyCache(BytecodeCache):

    def __init__(self, directory):
        self.directory = directory

    def load_bytecode(self, bucket):
        filename = path.join(self.directory, bucket.key)
        if path.exists(filename):
            with open(filename, 'rb') as f:
                bucket.load_bytecode(f)

    def dump_bytecode(self, bucket):
        filename = path.join(self.directory, bucket.key)
        with open(filename, 'wb') as f:
            bucket.write_bytecode(f)
```

A more advanced version of a filesystem based bytecode cache is part of Jinja2.

clear()

Clears the cache. This method is not used by Jinja2 but should be implemented to allow applications to clear the bytecode cache used by a particular environment.

dump_bytecode(*bucket*)

Subclasses have to override this method to write the bytecode from a bucket back to the cache. If it unable to do so it must not fail silently but raise an exception.

load_bytecode(*bucket*)

Subclasses have to override this method to load bytecode into a bucket. If they are not able to find code in the cache for the bucket, it must not do anything.

class jinja2.bccache.Bucket(*environment*, *key*, *checksum*)

Buckets are used to store the bytecode for one template. It’ s created and initialized by the bytecode cache and passed to the loading functions.

The buckets get an internal checksum from the cache assigned and use this to automatically reject outdated cache material. Individual bytecode cache subclasses don’ t have to care about cache invalidation.

environment

创建 bucket 的 Environment

key

该 bucket 的唯一键

code

如果已加载，则为字节码，否则为 None 。

bytecode_from_string(*string*)

Load bytecode from a string.

bytecode_to_string()

Return the bytecode as string.

load_bytecode(*f*)

Loads bytecode from a file or file like object.

reset()

Resets the bucket (unloads the bytecode).

write_bytecode(*f*)

Dump the bytecode into the file or file like object passed.

内建的字节码缓存:

class jinja2.FileSystemBytecodeCache(*directory=None*,*pattern='_jinja2%s.cache'*)

A bytecode cache that stores bytecode on the filesystem. It accepts two arguments: The directory where the cache items are stored and a pattern string that is used to build the filename.

If no directory is specified the system temporary items folder is used.

The pattern can be used to have multiple separate caches operate on the same directory. The default pattern is '*_jinja2*%s.cache'. %s is replaced with the cache key.

```
>>> bcc = FileSystemBytecodeCache('/tmp/jinja_cache', '%s.cache')
```

This bytecode cache supports clearing of the cache using the clear method.

class jinja2.MemcachedBytecodeCache(*client*, *prefix='jinja2/bytecode/'*, *timeout=None*, *ignore_memcache_errors=True*)

This class implements a bytecode cache that uses a memcache cache for storing the information. It does not enforce a specific memcache library (tummy’ s memcache or cmemcache) but will accept any class that provides the minimal interface required.

Libraries compatible with this class:

- werkzeug.contrib.cache
- python-memcached
- cmemcache

(Unfortunately the django cache interface is not compatible because it does not support storing binary data, only unicode. You can however pass the underlying cache client to the bytecode cache which is available asdjango.core.cache.cache._client.)

The minimal interface for the client passed to the constructor is this:

class MinimalClientInterface

set(*key*, *value*[, *timeout*])

Stores the bytecode in the cache. value is a string and timeout the timeout of the key. If timeout is not provided a default timeout or no timeout should be assumed, if it’ s provided it’ s an integer with the number of seconds the cache item should exist.

get(*key*)

Returns the value for the cache key. If the item does not exist in the cache the return value must be None.

The other arguments to the constructor are the prefix for all keys that is added before the actual cache key and the timeout for the bytecode in the cache system. We recommend a high (or no) timeout.

This bytecode cache does not support clearing of used items in the cache. The clear method is a no-operation function.

New in version 2.7: Added support for ignoring memcache errors through theignore_memcache_errors parameter.

实用工具

这些辅助函数和类在你向 Jinja2 环境中添加自定义过滤器或函数时很有用。

jinja2.environmentfilter(*f*)

Decorator for marking evironment dependent filters. The current Environment is passed to the filter as first argument.

jinja2.contextfilter(*f*)

Decorator for marking context dependent filters. The current Context will be passed as first argument.

jinja2.evalcontextfilter(*f*)

Decorator for marking eval-context dependent filters. An eval context object is passed as first argument. For more information about the eval context, see *求值上下文*.

New in version 2.4.

jinja2.environmentfunction(*f*)

This decorator can be used to mark a function or method as environment callable. This decorator works exactly like the contextfunction() decorator just that the first argument is the active Environment and not context.

jinja2.contextfunction(*f*)

This decorator can be used to mark a function or method context callable. A context callable is passed the active Context as first argument when called from the template. This is useful if a function wants to get access to the context or functions provided on the context object. For example a function that returns a sorted list of template variables the current template exports could look like this:

```
@contextfunction
def get_exported_names(context):
    return sorted(context.exported_vars)
```

jinja2.evalcontextfunction(*f*)

This decorator can be used to mark a function or method as an eval context callable. This is similar to the contextfunction() but instead of passing the context, an evaluation context object is passed. For more information about the eval context, see*求值上下文*.

New in version 2.4.

jinja2.escape(*s*)

把字符串 s 中 &、 、 >、 ' 和 " 转换为 HTML 安 全的序列。如果你需要在 HTML 中显示可能包含这些字符的文本，可以使用它。这 个函数不会转义对象。这个函数不会转义含有 HTML 表达式比如已转义数据的对象。

返回值是一个 Markup 字符串。

jinja2.clear_caches()

Jinja2 keeps internal caches for environments and lexers. These are used so that Jinja2 doesn’ t have to recreate environments and lexers all the time. Normally you don’ t have to care about that but if you are messuring memory consumption you may want to clean the caches.

jinja2.is_undefined(*obj*)

Check if the object passed is undefined. This does nothing more than performing an instance check against Undefined but looks nicer. This can be used for custom filters or tests that want to react to undefined variables. For example a custom default filter can look like this:

```
def default(var, default=''):
    if is_undefined(var):
        return default
    return var
```

class jinja2.Markup([*string*])

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the **html** interface a couple of frameworks and web applications use. Markup is a direct subclass of unicode and provides all the methods of unicode just that it escapes arguments passed and always returnsMarkup.

The escape function returns markup objects so that double escaping can’ t happen.

The constructor of the Markup class can be used for three different things: When passed an unicode object it’ s assumed to be safe, when passed an object with an HTML representation (has an **html** method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
...  def __html__(self):
...   return '<a href="#">foo</a>'
...
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

If you want object passed being always treated as unsafe you can use the escape()classmethod to create a Markup object:

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

Operations on a markup string are markup aware which means that all arguments are passed through the escape() function:

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp;amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

classmethod escape(*s*)

Escape the string. Works like escape() with the difference that for subclasses of Markup this function would return the correct subclass.

striptags()

Unescape markup into an text_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

unescape()

Unescape markup again into an text_type string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

Note

Jinja2 的 Markup 类至少与 Pylons 和 Genshi 兼容。预计不久更多模板 引擎和框架会采用**html** 的概念。

异常

exception jinja2.TemplateError(*message=None*)

Baseclass for all template errors.

exception jinja2.UndefinedError(*message=None*)

Raised if a template tries to operate on Undefined.

exception jinja2.TemplateNotFound(*name*, *message=None*)

Raised if a template does not exist.

exception jinja2.TemplatesNotFound(*names=()*, *message=None*)

Like TemplateNotFound but raised if multiple templates are selected. This is a subclass of TemplateNotFound exception, so just catching the base exception will catch both.

New in version 2.2.

exception jinja2.TemplateSyntaxError(*message*, *lineno*, *name=None*,*filename=None*)

Raised to tell the user that there is a problem with the template.

message

错误信息的 utf-8 字节串。

lineno

发生错误的行号。

name

模板的加载名的 unicode 字符串。

filename

加载的模板的文件名字节串，以文件系统的编码（多是 utf-8 ， Windows 是 mbcs ）。

文件名和错误消息是字节串而不是 unicode 字符串的原因是，在 Python 2.x 中，不对异常和回溯使用 unicode ，编译器同样。这会在 Python 3 改变。

exception jinja2.TemplateAssertionError(*message*, *lineno*, *name=None*,*filename=None*)

Like a template syntax error, but covers cases where something in the template caused an error at compile time that wasn’ t necessarily caused by a syntax error. However it’ s a direct subclass

of TemplateSyntaxError and has the same attributes.

自定义过滤器

自定义过滤器只是常规的 Python 函数，过滤器左边作为第一个参数，其余的参数作 为额外的参数或关键字参数传递到过滤器。

例如在过滤器 {{ 42|myfilter(23) }} 中，函数被以 myfilter(42, 23) 调 用。这里给出一个简单的过滤器示例，可以应用到 datetime 对象来格式化它们:

```
def datetimeformat(value, format='%H:%M / %d-%m-%Y'):
    return value.strftime(format)
```

你可以更新环境上的 filters 字典来把它注册到模板环境上:

```
environment.filters['datetimeformat'] = datetimeformat
```

在模板中使用如下:

```
written on: {{ article.pub_date|datetimeformat }}
publication date: {{ article.pub_date|datetimeformat('%d-%m-%Y') }}
```

也可以传给过滤器当前模板上下文或环境。当过滤器要返回一个未定义值或检查当前的autoescape 设置时很有用。为此，有三个装饰器： environmentfilter() 、 contextfilter()和 evalcontextfilter() 。

这里是一个小例子，过滤器把一个文本在 HTML 中换行或分段，并标记返回值为安全 的 HTML 字符串，因为自动转义是启用的:

```
import re
from jinja2 import evalcontextfilter, Markup, escape

_paragraph_re = re.compile(r'(?:\r\n|\r|\n){2,}')

@evalcontextfilter
def nl2br(eval_ctx, value):
    result = u'\n\n'.join(u'<p>%s</p>' % p.replace('\n', '<br>\n')
                          for p in _paragraph_re.split(escape(value)))
    if eval_ctx.autoescape:
        result = Markup(result)
    return result
```

上下文过滤器工作方式相同，只是第一个参数是当前活动的 Context 而 不是环境。

求值上下文

求值上下文（缩写为 eval context 或 eval ctx ）是 Jinja 2.4 中引入的新对象， 并可以在运行时激活/停用已编译的特性。

当前它只用于启用和禁用自动转义，但也可以用于扩展。

在之前的 Jinja 版本中，过滤器和函数被标记为环境可调用的来从环境中检查自动 转义的状态。在新版本中鼓励通过求值上下文来检查这个设定。

之前的版本:

```
@environmentfilter
def filter(env, value):
    result = do_something(value)
    if env.autoescape:
        result = Markup(result)
    return result
```

在新版本中，你可以用 contextfilter() 从实际的上下文中访问求值上下 文，或用evalcontextfilter() 直接把求值上下文传递给函数:

```
@contextfilter
def filter(context, value):
    result = do_something(value)
    if context.eval_ctx.autoescape:
        result = Markup(result)
    return result

@evalcontextfilter
def filter(eval_ctx, value):
    result = do_something(value)
    if eval_ctx.autoescape:
        result = Markup(result)
    return result
```

求值上下文一定不能在运行时修改。修改只能在扩展中的用 nodes.EvalContextModifier 和nodes.ScopedEvalContextModifier 发生，而不是通过求值上下文对象本身。

class jinja2.nodes.EvalContext(*environment*, *template_name=None*)

Holds evaluation time information. Custom attributes can be attached to it in extensions.

autoescape

True 或 False 取决于自动转义是否激活。

volatile

如果编译器不能在编译期求出某些表达式的值，为 True 。在运行时应该 始终为False 。

自定义测试

测试像过滤器一样工作，只是测试不能访问环境或上下文，并且它们不能链式使用。 测试的返回值应该是 True 或 False 。测试的用途是让模板设计者运行类型和 一致性检查。

这里是一个简单的测试，检验一个变量是否是素数:

```
import math

def is_prime(n):
    if n == 2:
        return True
    for i in xrange(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

你可以通过更新环境上的 tests 字典来注册它:

```
environment.tests['prime'] = is_prime
```

模板设计者可以在之后这样使用测试:

```
{% if 42 is prime %}
    42 is a prime number
{% else %}
    42 is not a prime number
{% endif %}
```

全局命名空间

Environment.globals 字典中的变量是特殊的，它们对导入的模板也是可用的， 即使它们不通过上下文导入。这是你可以放置始终可访问的变量和函数的地方。此外，Template.globals 是那些对特定模板可用的变量，即对所有的 render() 调用可用。``

低层 API

低层 API 暴露的功能对理解一些实现细节、调试目的或高级 *扩展* 技巧是有用的。除非你准确地了解你在做什么，否则 不推荐使用这些 API 。

Environment.lex(*source*, *name=None*, *filename=None*)

Lex the given sourcecode and return a generator that yields tokens as tuples in the form (lineno, token_type, value). This can be useful for *extension development* and debugging templates.

This does not perform preprocessing. If you want the preprocessing of the extensions to be applied you have to filter source through the preprocess() method.

Environment.parse(*source*, *name=None*, *filename=None*)

Parse the sourcecode and return the abstract syntax tree. This tree of nodes is used by the compiler to convert the template into executable source- or bytecode. This is useful for debugging or to extract information from templates.

If you are *developing Jinja2 extensions* this gives you a good overview of the node tree generated.

Environment.preprocess(*source*, *name=None*, *filename=None*)

Preprocesses the source with all extensions. This is automatically called for all parsing and compiling methods but *not* for lex() because there you usually only want the actual source tokenized.

Template.new_context(*vars=None*, *shared=False*, *locals=None*)

Create a new Context for this template. The vars provided will be passed to the template. Per default the globals are added to the context. If shared is set to True the data is passed as it to the context without adding the globals.

locals can be a dict of local variables for internal usage.

Template.root_render_func(*context*)

这是低层的渲染函数。它接受一个必须由相同模板或兼容的模板的 new_context() 创建的 Context 。这个渲染函数由编译器从 模板代码产生，并返回一个生产 unicode 字符串的生成器。

如果模板代码中发生了异常，模板引擎不会重写异常而是直接传递原始的异常。 事实上，这个函数只在 render() / generate() / stream() 的调用里被调用。

Template.blocks

一个块渲染函数的字典。其中的每个函数与 root_render_func() 的工作 相同，并且有相同的限制。

Template.is_up_to_date

如果有可用的新版本模板，这个属性是 False ，否则是 True 。

注意

低层 API 是易碎的。未来的 Jinja2 的版本将不会试图以不向后兼容的方式修改它， 而是在 Jinja2 核心的修改中表现出来。比如如果 Jinja2 在之后的版本中引入一 个新的 AST 节点，它会由 parse() 返回。

元 API

New in version 2.2.

元 API 返回一些关于抽象语法树的信息，这些信息能帮助应用实现更多的高级模板概 念。所有的元 API 函数操作一个 Environment.parse() 方法返回的抽象语法 树。

jinja2.meta.find_undeclared_variables(*ast*)

Returns a set of all variables in the AST that will be looked up from the context at runtime. Because at compile time it’ s not known which variables will be used depending on the path the execution takes at runtime, all variables are returned.

```
>>> from jinja2 import Environment, meta
>>> env = Environment()
>>> ast = env.parse('{% set foo = 42 %}{{ bar + foo }}')
>>> meta.find_undeclared_variables(ast)
set(['bar'])
```

Implementation

Internally the code generator is used for finding undeclared variables. This is good to know because the code generator might raise a TemplateAssertionError during compilation and as a matter of fact this function can currently raise that exception as well.

jinja2.meta.find_referenced_templates(*ast*)

Finds all the referenced templates from the AST. This will return an iterator over all the hardcoded template extensions, inclusions and imports. If dynamic inheritance or inclusion is used, None will be yielded.

```
>>> from jinja2 import Environment, meta
>>> env = Environment()
>>> ast = env.parse('{% extends "layout.html" %}{% include helper %}')
>>> list(meta.find_referenced_templates(ast))
['layout.html', None]
```

This function is useful for dependency tracking. For example if you want to rebuild parts of the website after a layout template has changed.

沙箱

Jinja2 沙箱用于为不信任的代码求值。访问不安全的属性和方法是被禁止的。

假定在默认配置中 env 是一个 SandboxedEnvironment 实例，下面的代码展示 了它如何工作:

```
>>> env.from_string("{{ func.func_code }}").render(func=lambda:None)
u''
>>> env.from_string("{{ func.func_code.do_something }}").render(func=lambda:None)
Traceback (most recent call last):
  ...
SecurityError: access to attribute 'func_code' of 'function' object is unsafe.
```

API

class jinja2.sandbox.SandboxedEnvironment([*options*])

The sandboxed environment. It works like the regular environment but tells the compiler to generate sandboxed code. Additionally subclasses of this environment may override the methods that tell the runtime what attributes or functions are safe to access.

If the template tries to access insecure code a SecurityError is raised. However also other exceptions may occour during the rendering so the caller has to ensure that all exceptions are catched.

call_binop(*context*, *operator*, *left*, *right*)

For intercepted binary operator calls (intercepted_binops()) this function is executed instead of the builtin operator. This can be used to fine tune the behavior of certain operators.

New in version 2.6.

call_unop(*context*, *operator*, *arg*)

For intercepted unary operator calls (intercepted_unops()) this function is executed instead of the builtin operator. This can be used to fine tune the behavior of certain operators.

New in version 2.6.

default_binop_table* = {'//': , '%': , '+': , ' ': , '-': , '/': , ' ': }

default callback table for the binary operators. A copy of this is available on each instance of a sandboxed environment as binop_table

default_unop_table* = {'+': , '-': }*

default callback table for the unary operators. A copy of this is available on each instance of a sandboxed environment as unop_table

intercepted_binops* = frozenset([])*

a set of binary operators that should be intercepted. Each operator that is added to this set (empty by default) is delegated to the call_binop() method that will perform the operator. The default operator callback is specified by binop_table.

The following binary operators are interceptable: //, %, +, , -, /, *and* *

The default operation form the operator table corresponds to the builtin function. Intercepted calls are always slower than the native operator call, so make sure only to intercept the ones you are interested in.

New in version 2.6.

intercepted_unops* = frozenset([])*

a set of unary operators that should be intercepted. Each operator that is added to this set (empty by default) is delegated to the call_unop() method that will perform the operator. The default operator callback is specified by unop_table.

The following unary operators are interceptable: +, -

The default operation form the operator table corresponds to the builtin function. Intercepted calls are always slower than the native operator call, so make sure only to intercept the ones you are interested in.

New in version 2.6.

is_safe_attribute(*obj*, *attr*, *value*)

The sandboxed environment will call this method to check if the attribute of an object is safe to access. Per default all attributes starting with an underscore are considered private as well as the special attributes of internal python objects as returned by the is_internal_attribute() function.

is_safe_callable(*obj*)

Check if an object is safely callable. Per default a function is considered safe unless the unsafe_callable attribute exists and is True. Override this method to alter the behavior, but this won’ t affect the unsafe decorator from this module.

class jinja2.sandbox.ImmutableSandboxedEnvironment([*options*])

Works exactly like the regular SandboxedEnvironment but does not permit modifications on the builtin mutable objects list, set, and dict by using themodifies_known_mutable() function.

exception jinja2.sandbox.SecurityError(*message=None*)

Raised if a template tries to do something insecure if the sandbox is enabled.

jinja2.sandbox.unsafe(*f*)

Marks a function or method as unsafe.

```
@unsafe
def delete(self):
    pass
```

jinja2.sandbox.is_internal_attribute(*obj*, *attr*)

Test if the attribute given is an internal python attribute. For example this function returns True for the func_code attribute of python objects. This is useful if the environment method is_safe_attribute() is overridden.

```
>>> from jinja2.sandbox import is_internal_attribute
>>> is_internal_attribute(lambda: None, "func_code")
True
>>> is_internal_attribute((lambda x:x).func_code, 'co_code')
True
>>> is_internal_attribute(str, "upper")
False
```

jinja2.sandbox.modifies_known_mutable(*obj*, *attr*)

This function checks if an attribute on a builtin mutable object (list, dict, set or deque) would modify it if called. It also supports the “user” -versions of the objects (sets.Set,UserDict.* etc.) and with Python 2.6 onwards the abstract base classes MutableSet,MutableMapping, and MutableSequence.

```
>>> modifies_known_mutable({}, "clear")
True
>>> modifies_known_mutable({}, "keys")
False
>>> modifies_known_mutable([], "append")
True
>>> modifies_known_mutable([], "index")
False
```

If called with an unsupported object (such as unicode) False is returned.

```
>>> modifies_known_mutable("foo", "upper")
False
```

提示

Jinja2 沙箱自己并没有彻底解决安全问题。特别是对 web 应用，你必须晓得用户 可能用任意 HTML 来创建模板，所以保证他们不通过注入 JavaScript 或其它更多 方法来互相损害至关重要（如果你在同一个服务 器上运行多用户）。

同样，沙箱的好处取决于配置。我们强烈建议只向模板传递非共享资源，并 且使用某种属性白名单。

也请记住，模板会抛出运行时或编译期错误，确保捕获它们。

运算符拦截

New in version 2.6.

为了性能最大化， Jinja2 会让运算符直接条用类型特定的回调方法。这意味着， 通过重载 Environment.call() 来拦截是不可能的。此外，由于运算符的工作 方式，把运算符转换为特殊方法不总是直接可行的。比如为了分类，至少一个特殊 方法存在。

在 Jinja 2.6 中，开始支持显式的运算符拦截。必要时也可以用于自定义的特定 运算符。为了拦截运算符，需要覆写 SandboxedEnvironment.intercepted_binops 属性。当需要拦截的运算符 被添加到这个集合， Jinja2 会生成调用 SandboxedEnvironment.call_binop() 函数的字节码。对于一元运算符， 必须替代地使用 unary 属性和方法。

SandboxedEnvironment.call_binop 的默认实现会使用 SandboxedEnvironment.binop_table 来把运算符标号翻译成执行默认 运算符行为的回调。

这个例子展示了幂（ ** ）操作符可以在 Jinja2 中禁用:

```
from jinja2.sandbox import SandboxedEnvironment

class MyEnvironment(SandboxedEnvironment):
    intercepted_binops = frozenset(['**'])

    def call_binop(self, context, operator, left, right):
        if operator == '**':
            return self.undefined('the power operator is unavailable')
        return SandboxedEnvironment.call_binop(self, context,
                                               operator, left, right)
```

确保始终调入 super 方法，即使你不拦截这个调用。 Jinja2 内部会调用 这个方法来对表达式求值。

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 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 foois 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 something> if is true> else 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 thelist 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 countoccurrences 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 tofalse 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。

扩展

Jinja2 支持扩展来添加过滤器、测试、全局变量或者甚至是处理器。扩展的主要动力是 把诸如添加国际化支持的常用代码迁移到一个可重用的类。

添加扩展

扩展在 Jinja2 环境创建时被添加。一旦环境被创建，就不能添加额外的扩展。要添加 一个扩展，传递一个扩展类或导入路径的列表到 Environment 构造函数的 environment 参数。下面的例子创建了一个加载了 i18n 扩展的 Jinja2 环境:

```
jinja_env = Environment(extensions=['jinja2.ext.i18n'])
```

i18n 扩展

Import name: jinja2.ext.i18n

Jinja2 当前只附带一个扩展，就是 i18n 扩展。它可以与 gettext 或 babel 联合使用。如果启用了 i18n 扩展， Jinja2 提供了 trans 语句来标记被其包裹的 字符串为可翻译的，并调用 gettext 。

在启用虚拟的 _ 函数后，之后的 gettext 调用会被添加到环境的全局变量。那么 一个国际化的应用应该不仅在全局，以及在每次渲染中在命名空间中提供至少一个 gettext 或可选的ngettext 函数。

环境方法

在启用这个扩展后，环境提供下面的额外方法:

jinja2.Environment.install_gettext_translations(*translations*, *newstyle=False*)

在该环境中全局安装翻译。提供的翻译对象要至少实现 uggettext 和 ungettext 。 gettext.NullTranslations 和 gettext.GNUTranslations 类和 Babel 's 的 Translations 类也被支持。

Changed in version 2.5: 添加了新样式的 gettext

jinja2.Environment.install_null_translations(*newstyle=False*)

安装虚拟的 gettext 函数。这在你想使应用为国际化做准备但还不想实现完整的 国际化系统时很有用。

Changed in version 2.5: 添加了新样式的 gettext

jinja2.Environment.install_gettext_callables(*gettext*, *ngettext*,*newstyle=False*)

在环境中把给出的 gettext 和 ngettext 可调用量安装为全局变量。它们 应该表现得几乎与标准库中的 gettext.ugettext() 和 gettext.ungettext() 函数相同。

如果激活了 新样式 ，可调用量被包装为新样式的可调用量一样工作。更多 信息见 *新样式 Gettext* 。

New in version 2.5.

jinja2.Environment.uninstall_gettext_translations()

再次卸载翻译。

jinja2.Environment.extract_translations(*source*)

从给定的模板或源中提取本地化字符串。

对找到的每一个字符串，这个函数生产一个 (lineno, function, message) 元组，在这里:

- lineno 是这个字符串所在行的行号。
- function 是 gettext 函数使用的名称（如果字符串是从内嵌的 Python 代码中抽取的）。
- message 是字符串本身（一个 unicode 对象，在函数有多个字符串参数 时是一个unicode 对象的元组）。

如果安装了 Babel ， *Babel 集成* 可以用来为 babel 抽取字符串。

对于一个对多种语言可用而对所有用户给出同一种的语言的 web 应用（例如一个法国社 区安全了一个多种语言的论坛软件）可能会一次性加载翻译并且在环境生成时把翻译方 法添加到环境上:

```
translations = get_gettext_translations()
env = Environment(extensions=['jinja2.ext.i18n'])
env.install_gettext_translations(translations)
```

get_get_translations 函数会返回当前配置的翻译器。（比如使用 gettext.find ）

模板设计者的 i18n 扩展使用在 *模板文档* 中有描述。

新样式 Gettext

New in version 2.5.

从版本 2.5 开始你可以使用新样式的 gettext 调用。这些的启发源于 trac 的内部 gettext 函数并且完全被 babel 抽取工具支持。如果你不使用 Babel 的抽取工具， 它可能不会像其它抽取工具预期的那样工作。

标准 gettext 调用和新样式的 gettext 调用有什么区别？通常，它们要输入的东西 更少，出错率更低。并且如果在自动转义环境中使用它们，它们也能更好地支持自动 转义。这里是一些新老样式调用的差异:

标准 gettext:

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

新样式看起来是这样:

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

新样式 gettext 的优势是你需要输入的更少，并且命名占位符是强制的。后者看起 来似乎是缺陷，但解决了当翻译者不能切换两个占位符的位置时经常勉励的一大堆 麻烦。使用新样式的 gettext ，所有的格式化字符串看起来都一样。

除此之外，在新样式 gettext 中，如果没有使用占位符，字符串格式化也会被使用， 这使得所有的字符串表现一致。最后，不仅是新样式的 gettext 调用可以妥善地为 解决了许多转义相关问题的自动转义标记字符串，许多模板也在使用自动转义时体验 了多次。

表达式语句

Import name: jinja2.ext.do

“do”又叫做表达式语句扩展，向模板引擎添加了一个简单的 do 标签，其工作如同 一个变量表达式，只是忽略返回值。

循环控制

Import name: jinja2.ext.loopcontrols

这个扩展添加了循环中的 break 和 continue 支持。在启用它之后， Jinja2 提供的这两个关键字如同 Python 中那样工作。

With 语句

Import name: jinja2.ext.with_

New in version 2.3.

这个扩展添加了 with 关键字支持。使用这个关键字可以在模板中强制一块嵌套的 作用域。变量可以在 with 语句的块头中直接声明，或直接在里面使用标准的 set 语句。

自动转义扩展

Import name: jinja2.ext.autoescape

New in version 2.4.

自动转义扩展允许你在模板内开关自动转义特性。如果环境的 autoescape 设定为 False ，它可以被激活。如果是 True 可以被关闭。这个设定的覆盖是有作用域的。

编写扩展

你可以编写扩展来向 Jinja2 中添加自定义标签。这是一个不平凡的任务，而且通常不需 要，因为默认的标签和表达式涵盖了所有常用情况。如 i18n 扩展是一个扩展有用的好例 子，而另一个会是碎片缓存。

当你编写扩展时，你需要记住你在与 Jinja2 模板编译器一同工作，而它并不验证你传递 到它的节点树。如果 AST 是畸形的，你会得到各种各样的编译器或运行时错误，这调试起 来极其可怕。始终确保你在使用创建正确的节点。下面的 API 文档展示了有什么节点和如 何使用它们。

示例扩展

下面的例子用 Werkzeug 的缓存 contrib 模块为 Jinja2 实现了一个 cache 标签:

```
from jinja2 import nodes
from jinja2.ext import Extension

class FragmentCacheExtension(Extension):
    # a set of names that trigger the extension.
    tags = set(['cache'])

    def __init__(self, environment):
        super(FragmentCacheExtension, self).__init__(environment)

        # add the defaults to the environment
        environment.extend(
            fragment_cache_prefix='',
            fragment_cache=None
        )

    def parse(self, parser):
```

```
        # the first token is the token that started the tag.  In our case
        # we only listen to ``'cache'`` so this will be a name token with
        # `cache` as value.  We get the line number so that we can give
        # that line number to the nodes we create by hand.
        lineno = parser.stream.next().lineno

        # now we parse a single expression that is used as cache key.
        args = [parser.parse_expression()]

        # if there is a comma, the user provided a timeout.  If not use
        # None as second parameter.
        if parser.stream.skip_if('comma'):
            args.append(parser.parse_expression())
        else:
            args.append(nodes.Const(None))

        # now we parse the body of the cache block up to `endcache` and
        # drop the needle (which would always be `endcache` in that case)
        body = parser.parse_statements(['name:endcache'], drop_needle=True)

        # now return a `CallBlock` node that calls our _cache_support
        # helper method on this extension.
        return nodes.CallBlock(self.call_method('_cache_support', args),
                               [], [], body).set_lineno(lineno)

    def _cache_support(self, name, timeout, caller):
        """Helper callback."""
        key = self.environment.fragment_cache_prefix + name

        # try to load the block from the cache
        # if there is no fragment in the cache, render it and store
        # it in the cache.
        rv = self.environment.fragment_cache.get(key)
        if rv is not None:
            return rv
        rv = caller()
        self.environment.fragment_cache.add(key, rv, timeout)
        return rv
```

而这是你在环境中使用它的方式:

```
from jinja2 import Environment
from werkzeug.contrib.cache import SimpleCache

env = Environment(extensions=[FragmentCacheExtension])
env.fragment_cache = SimpleCache()
```

之后，在模板中可以标记块为可缓存的。下面的例子缓存一个边栏 300 秒:

```
{% cache 'sidebar', 300 %}
<div class="sidebar">
    ...
</div>
{% endcache %}
```

扩展 API

扩展总是继承 jinja2.ext.Extension 类:

class jinja2.ext.Extension(*environment*)

Extensions can be used to add extra functionality to the Jinja template system at the parser level. Custom extensions are bound to an environment but may not store environment specific data on self. The reason for this is that an extension can be bound to another environment (for overlays) by creating a copy and reassigning theenvironment attribute.

As extensions are created by the environment they cannot accept any arguments for configuration. One may want to work around that by using a factory function, but that is not possible as extensions are identified by their import name. The correct way to configure the extension is storing the configuration values on the environment. Because this way the environment ends up acting as central configuration storage the attributes may clash which is why extensions have to ensure that the names they choose for configuration are not too generic. prefix for example is a terrible name,fragment_cache_prefix on the other hand is a good name as includes the name of the extension (fragment cache).

identifier

扩展的标识符。这始终是扩展类的真实导入名，不能被修改。

tags

如果扩展实现自定义标签，这是扩展监听的标签名的集合。

attr(*name*, *lineno=None*)

Return an attribute node for the current extension. This is useful to pass constants on extensions to generated template code.

```
self.attr('_my_attribute', lineno=lineno)
```

call_method(*name*, *args=None*, *kwargs=None*, *dyn_args=None*,*dyn_kwargs=None*, *lineno=None*)

Call a method of the extension. This is a shortcut for attr() + jinja2.nodes.Call.

filter_stream(*stream*)

It’ s passed a TokenStream that can be used to filter tokens returned. This method has to return an iterable of Tokens, but it doesn’ t have to return a TokenStream.

In the ext folder of the Jinja2 source distribution there is a file calledinlinegettext.py which implements a filter that utilizes this method.

parse(*parser*)

If any of the tags matched this method is called with the parser as first argument. The token the parser stream is pointing at is the name token that matched. This method has to return one or a list of multiple nodes.

preprocess(*source*, *name*, *filename=None*)

This method is called before the actual lexing and can be used to preprocess the source.
The filename is optional. The return value must be the preprocessed source.

解析器 API

传递到 Extension.parse() 的解析器提供解析不同类型表达式的方式。下 面的方法可能会在扩展中使用:

class jinja2.parser.Parser(*environment*, *source*, *name=None*, *filename=None*,*state=None*)

This is the central parsing class Jinja2 uses. It’ s passed to extensions and can be used to parse expressions or statements.

filename

解析器处理的模板文件名。这 **不是** 模板的加载名。加载名见 name 。对于不是从文件系统中加载的模板，这个值为 None 。

name

模板的加载名。

stream

当前的 TokenStream 。

fail(*msg*, *lineno=None*, *exc=*)

Convenience method that raises exc with the message, passed line number or last line number as well as the current name and filename.

free_identifier(*lineno=None*)

Return a new free identifier as InternalName.

parse_assign_target(*with_tuple=True*, *name_only=False*,*extra_end_rules=None*)

Parse an assignment target. As Jinja2 allows assignments to tuples, this function can parse all allowed assignment targets. Per default assignments to tuples are parsed, that can be disable however by setting with_tuple to False. If only assignments to names are wanted name_only can be set to True. Theextra_end_rules parameter is forwarded to the tuple parsing function.

parse_expression(*with_condexpr=True*)

Parse an expression. Per default all expressions are parsed, if the optionalwith_condexpr parameter is set to False conditional expressions are not parsed.

parse_statements(*end_tokens*, *drop_needle=False*)

Parse multiple statements into a list until one of the end tokens is reached. This is used to parse the body of statements as it also parses template data if appropriate. The parser checks first if the current token is a colon and skips it if there is one. Then it checks for the block end and parses until if one of the end_tokens is reached. Per default the active token in the stream at the end of the call is the matched end token. If this is not wanted drop_needle can be set to True and the end token is removed.

parse_tuple(*simplified=False*, *with_condexpr=True*, *extra_end_rules=None*,*explicit_parentheses=False*)

Works like parse_expression but if multiple expressions are delimited by a comma a Tuple node is created. This method could also return a regular expression instead of a tuple if no commas where found.

The default parsing mode is a full tuple. If simplified is True only names and literals are parsed. The no_condexpr parameter is forwarded toparse_expression().

Because tuples do not require delimiters and may end in a bogus comma an extra hint is needed that marks the end of a tuple. For example for loops support tuples between for and in. In that case the extra_end_rules is set to ['name:in'].

explicit_parentheses is true if the parsing was triggered by an expression in parentheses. This is used to figure out if an empty tuple is a valid expression or not.

class jinja2.lexer.TokenStream(*generator*, *name*, *filename*)

A token stream is an iterable that yields Tokens. The parser however does not iterate over it but calls next() to go one token ahead. The current active token is stored ascurrent.

current

当前的 Token 。

eos

Are we at the end of the stream?

expect(*expr*)

Expect a given token type and return it. This accepts the same argument asjinja2.lexer.Token.test().

look()

Look at the next token.

next()

Go one token ahead and return the old one

next_if(*expr*)

Perform the token test and return the token if it matched. Otherwise the return value is None.

push(*token*)

Push a token back to the stream.

skip(*n=1*)

Got n tokens ahead.

skip_if(*expr*)

Like next_if() but only returns True or False.

class jinja2.lexer.Token

Token class.

lineno

token 的行号。

type

token 的类型。这个值是被禁锢的，所以你可以用 is 运算符同任意字符 串比较。

value

token 的值。

test(*expr*)

Test a token against a token expression. This can either be a token type

or'token_type:token_value'. This can only test against string values and types.

test_any(**iterable*)

Test against multiple token expressions.

同样，在词法分析模块中也有一个实用函数可以计算字符串中的换行符数目:

```
.. autofunction:: jinja2.lexer.count_newlines
```

AST

AST（抽象语法树: Abstract Syntax Tree）用于表示解析后的模板。它有编译器之后 转换到可执行的 Python 代码对象的节点构建。提供自定义语句的扩展可以返回执行自 定义 Python 代码的节点。

下面的清单展示了所有当前可用的节点。 AST 在 Jinja2 的各个版本中有差异，但会向 后兼容。

更多信息请见 jinja2.Environment.parse() 。

class jinja2.nodes.Node

Baseclass for all Jinja2 nodes. There are a number of nodes available of different types. There are four major types:

- Stmt: statements
- Expr: expressions
- Helper: helper nodes
- Template: the outermost wrapper node

All nodes have fields and attributes. Fields may be other nodes, lists, or arbitrary values. Fields are passed to the constructor as regular positional arguments, attributes as keyword arguments. Each node has two attributes: lineno (the line number of the node) and environment.
The environment attribute is set at the end of the parsing process for all nodes automatically.

find(*node_type*)

Find the first node of a given type. If no such node exists the return value isNone.

find_all(*node_type*)

Find all the nodes of a given type. If the type is a tuple, the check is performed for any of the tuple items.

iter_child_nodes(*exclude=None*, *only=None*)

Iterates over all direct child nodes of the node. This iterates over all fields and yields the values of they are nodes. If the value of a field is a list all the nodes in that list are returned.

iter_fields(*exclude=None*, *only=None*)

This method iterates over all fields that are defined and yields (key, value) tuples. Per default all fields are returned, but it’ s possible to limit that to some fields by providing the only parameter or to exclude some using the exclude parameter. Both should be sets or tuples of field names.

set_ctx(*ctx*)

Reset the context of a node and all child nodes. Per default the parser will all generate nodes that have a ‘load’ context as it’ s the most common one. This method is used in the parser to set assignment targets and other nodes to a store context.

set_environment(*environment*)

Set the environment for all nodes.

set_lineno(*lineno*, *override=False*)

Set the line numbers of the node and children.

class jinja2.nodes.Expr

Baseclass for all expressions.

Node type: Node

as_const(*eval_ctx=None*)

Return the value of the expression as constant or raise Impossible if this was not possible.

An EvalContext can be provided, if none is given a default context is created which requires the nodes to have an attached environment.

Changed in version 2.4: the eval_ctx parameter was added.

can_assign()

Check if it’ s possible to assign something to this node.

class jinja2.nodes.BinExpr(*left*, *right*)

Baseclass for all binary expressions.

Node type: Expr

class jinja2.nodes.Add(*left*, *right*)

Add the left to the right node.

Node type: BinExpr

class jinja2.nodes.And(*left*, *right*)

Short circuited AND.

Node type: BinExpr

class jinja2.nodes.Div(*left*, *right*)

Divides the left by the right node.

Node type: BinExpr

class jinja2.nodes.FloorDiv(*left*, *right*)

Divides the left by the right node and truncates conver the result into an integer by truncating.

Node type: BinExpr

class jinja2.nodes.Mod(*left*, *right*)

Left modulo right.

Node type: BinExpr

class jinja2.nodes.Mul(*left*, *right*)

Multiplies the left with the right node.

Node type: BinExpr

class jinja2.nodes.Or(*left*, *right*)

Short circuited OR.

Node type: BinExpr

class jinja2.nodes.Pow(*left*, *right*)

Left to the power of right.

Node type: BinExpr

class jinja2.nodes.Sub(*left*, *right*)

Substract the right from the left node.

Node type: BinExpr

class jinja2.nodes.Call(*node*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)

Calls an expression. args is a list of arguments, kwargs a list of keyword arguments (list of Keyword nodes), and dyn_args and dyn_kwargs has to be either None or a node that is used as

node for dynamic positional (*args) or keyword (**kwargs) arguments.

Node type: Expr

class jinja2.nodes.Compare(*expr*, *ops*)

Compares an expression with some other expressions. ops must be a list of Operands.

Node type: Expr

class jinja2.nodes.Concat(*nodes*)

Concatenates the list of expressions provided after converting them to unicode.

Node type: Expr

class jinja2.nodes.CondExpr(*test*, *expr1*, *expr2*)

A conditional expression (inline if expression). ({{ foo if bar else baz }})

Node type: Expr

class jinja2.nodes.ContextReference

Returns the current template context. It can be used like a Name node, with a 'load'ctx and will return the current Context object.

Here an example that assigns the current template name to a variable named foo:

```
Assign(Name('foo', ctx='store'),
       Getattr(ContextReference(), 'name'))
```

Node type: Expr

class jinja2.nodes.EnvironmentAttribute(*name*)

Loads an attribute from the environment object. This is useful for extensions that want to call a callback stored on the environment.

Node type: Expr

class jinja2.nodes.ExtensionAttribute(*identifier*, *name*)

Returns the attribute of an extension bound to the environment. The identifier is the identifier of the Extension.

This node is usually constructed by calling the attr() method on an extension.

Node type: Expr

class jinja2.nodes.Filter(*node*, *name*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)

This node applies a filter on an expression. name is the name of the filter, the rest of the fields are the same as for Call.

If the node of a filter is None the contents of the last buffer are filtered. Buffers are created by macros and filter blocks.

Node type: Expr

class jinja2.nodes.Getattr(*node*, *attr*, *ctx*)

Get an attribute or item from an expression that is a ascii-only bytestring and prefer the attribute.

Node type: Expr

class jinja2.nodes.Getitem(*node*, *arg*, *ctx*)

Get an attribute or item from an expression and prefer the item.
Node type: Expr

class jinja2.nodes.ImportedName(*importname*)

If created with an import name the import name is returned on node access. For example ImportedName('cgi.escape') returns the escape function from the cgi module on evaluation. Imports are optimized by the compiler so there is no need to assign them to local variables.

Node type: Expr

class jinja2.nodes.InternalName(*name*)

An internal name in the compiler. You cannot create these nodes yourself but the parser provides a free_identifier() method that creates a new identifier for you. This identifier is not available from the template and is not threated specially by the compiler.

Node type: Expr

class jinja2.nodes.Literal

Baseclass for literals.

Node type: Expr

class jinja2.nodes.Const(*value*)

All constant values. The parser will return this node for simple constants such as 42 or"foo" but it can be used to store more complex values such as lists too. Only constants with a safe representation (objects where eval(repr(x)) == x is true).

Node type: Literal

class jinja2.nodes.Dict(*items*)

Any dict literal such as {1: 2, 3: 4}. The items must be a list of Pair nodes.

Node type: Literal

class jinja2.nodes.List(*items*)

Any list literal such as [1, 2, 3]

Node type: Literal

class jinja2.nodes.TemplateData(*data*)

A constant template string.

Node type: Literal

class jinja2.nodes.Tuple(*items*, *ctx*)

For loop unpacking and some other things like multiple arguments for subscripts. Like for Name ctx specifies if the tuple is used for loading the names or storing.

Node type: Literal

class jinja2.nodes.MarkSafe(*expr*)

Mark the wrapped expression as safe (wrap it as Markup).

Node type: Expr

class jinja2.nodes.MarkSafeIfAutoescape(*expr*)

Mark the wrapped expression as safe (wrap it as Markup) but only if autoescaping is active.

New in version 2.5.

Node type: Expr

class jinja2.nodes.Name(*name*, *ctx*)

Looks up a name or stores a value in a name. The ctx of the node can be one of the following values:

- store: store a value in the name
- load: load that name
- param: like store but if the name was defined as function parameter.

Node type: Expr

class jinja2.nodes.Slice(*start*, *stop*, *step*)

Represents a slice object. This must only be used as argument for Subscript.

Node type: Expr

class jinja2.nodes.Test(*node*, *name*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)

Applies a test on an expression. name is the name of the test, the rest of the fields are the same as for Call.

Node type: Expr

class jinja2.nodes.UnaryExpr(*node*)

Baseclass for all unary expressions.

Node type: Expr

class jinja2.nodes.Neg(*node*)

Make the expression negative.

Node type: UnaryExpr

class jinja2.nodes.Not(*node*)

Negate the expression.

Node type: UnaryExpr

class jinja2.nodes.Pos(*node*)

Make the expression positive (noop for most expressions)

Node type: UnaryExpr

class jinja2.nodes.Helper

Nodes that exist in a specific context only.

Node type: Node

class jinja2.nodes.Keyword(*key*, *value*)

A key, value pair for keyword arguments where key is a string.

Node type: Helper

class jinja2.nodes.Operand(*op*, *expr*)

Holds an operator and an expression. The following operators are available: %, *, +,-, //, /, eq, gt, gteq, in, lt, lteq, ne, not, notin

Node type: Helper

class jinja2.nodes.Pair(*key*, *value*)

A key, value pair for dicts.

Node type: Helper

class jinja2.nodes.Stmt

Base node for all statements.

Node type: Node

class jinja2.nodes.Assign(*target*, *node*)

Assigns an expression to a target.

Node type: Stmt

class jinja2.nodes.Block(*name*, *body*, *scoped*)

A node that represents a block.

Node type: Stmt

class jinja2.nodes.Break

Break a loop.

Node type: Stmt

class jinja2.nodes.CallBlock(*call*, *args*, *defaults*, *body*)

Like a macro without a name but a call instead. call is called with the unnamed macro as caller argument this node holds.

Node type: Stmt

class jinja2.nodes.Continue

Continue a loop.

Node type: Stmt

class jinja2.nodes.EvalContextModifier(*options*)

Modifies the eval context. For each option that should be modified, a Keyword has to be added to the options list.

Example to change the autoescape setting:

```
EvalContextModifier(options=[Keyword('autoescape', Const(True))])
```

Node type: Stmt

class jinja2.nodes.ScopedEvalContextModifier(*options*, *body*)

Modifies the eval context and reverts it later. Works exactly like EvalContextModifierbut will only modify the EvalContext for nodes in the body.

Node type: EvalContextModifier

class jinja2.nodes.ExprStmt(*node*)

A statement that evaluates an expression and discards the result.

Node type: Stmt

class jinja2.nodes.Extends(*template*)

Represents an extends statement.

Node type: Stmt

class jinja2.nodes.FilterBlock(*body*, *filter*)

Node for filter sections.

Node type: Stmt

class jinja2.nodes.For(*target*, *iter*, *body*, *else_*, *test*, *recursive*)

The for loop. target is the target for the iteration (usually a Name or Tuple), iter the iterable. body is a list of nodes that are used as loop-body, and else_ a list of nodes for the else block. If no else node exists it has to be an empty list.

For filtered nodes an expression can be stored as test, otherwise None.

Node type: Stmt

class jinja2.nodes.FromImport(*template*, *names*, *with_context*)

A node that represents the from import tag. It's important to not pass unsafe names to the name attribute. The compiler translates the attribute lookups directly into getattr calls and does *not* use the subscript callback of the interface. As exported variables may not start with

double underscores (which the parser asserts) this is not a problem for regular Jinja code, but if this node is used in an extension extra care must be taken.

The list of names may contain tuples if aliases are wanted.

Node type: Stmt

class jinja2.nodes.If(*test*, *body*, *else_*)

If test is true, body is rendered, else else_.

Node type: Stmt

class jinja2.nodes.Import(*template*, *target*, *with_context*)

A node that represents the import tag.

Node type: Stmt

class jinja2.nodes.Include(*template*, *with_context*, *ignore_missing*)

A node that represents the include tag.

Node type: Stmt

class jinja2.nodes.Macro(*name*, *args*, *defaults*, *body*)

A macro definition. name is the name of the macro, args a list of arguments anddefaults a list of defaults if there are any. body is a list of nodes for the macro body.

Node type: Stmt

class jinja2.nodes.Output(*nodes*)

A node that holds multiple expressions which are then printed out. This is used both for the print statement and the regular template data.

Node type: Stmt

class jinja2.nodes.Scope(*body*)

An artificial scope.

Node type: Stmt

class jinja2.nodes.Template(*body*)

Node that represents a template. This must be the outermost node that is passed to the compiler.

Node type: Node

exception jinja2.nodes.Impossible

Raised if the node could not perform a requested action.

集成

Jinja2 提供了一些代码来继承到其它工具，诸如框架、 Babel 库或你偏好的编辑器 的奇特的代码高亮。这里是包含的这些的简要介绍。

帮助继承的文件在 这里 可 用。

Babel 集成

Jinja 提供了用 Babel 抽取器从模板中抽取 gettext 消息的支持，抽取器的接入点 名为 jinja2.ext.babel_extract 。 Babel 支持的被作为 *i18n 扩展* 的 一部分实现。

Gettext 消息从 trans 标签和代码表达式中抽取。

要从模板中抽取 gettext 消息，项目需要在它的 Babel 抽取方法 mapping file 中 有一个 Jinja2 节:

```
[jinja2: **/templates/**.html]
encoding = utf-8
```

Environment 的语法相关选项也可作为 mapping file 的配置值。例如告知 抽取器模板使用 %作为 line_statement_prefix 你可以这样写:

```
[jinja2: **/templates/**.html]
encoding = utf-8
line_statement_prefix = %
```

扩展 可能也被定义为传递一个逗号分割的导入路径列表作为 extensions 值。 i18n 扩展会被自动添加。

Changed in version 2.7: 直到 2.7 模板语法错误始终被忽略。因为许多人在模板文件夹中放置非模板的 html 文件，而这会随机报错，所以如此设定。假定是无论如何测试套件会捕获 模板中的语法错误。如果你不想要这个行为，你可以在设置中添加 slient=flase ，异常会被传播。

Pylons

从 Pylons 0.9.7 开始，集成 Jinja 到 Pylons 驱动的应用令人难以置信的简单。

模板引擎在 config/environment.py 中配置。为 Jinja2 的配置看起来是这样:

```
from jinja2 import Environment, PackageLoader
config['pylons.app_globals'].jinja_env = Environment(
    loader=PackageLoader('yourapplication', 'templates')
)
```

之后，你可以用 pylons.templating 模块中的 render_jinja 函数渲染 Jinja 模板。

此外，设置 Pylons 的 c 对象为严格模式是个好主意。按照默认，访问任何 c 对象 上不存在的属性会返回一个空字符串而不是一个未定义对象。更改这个，只需要使用这个 片段并添加到你的 config/environment.py 中:

```
config['pylons.strict_c'] = True
```

TextMate

在 Jinja2 项目根目录的 ext 文件夹中，有一个 TextMate 的 bundle 来提供 Jinja1 和 Jinja2 的基于文本的模板的语法高亮，同样也支持 HTML 。它也包含了一些常用的片 段。

Vim

同样，在 Jinja2 项目的根目录下的 ext 文件夹中的 Vim-scripts 目录有一个 Vim 的语法插件。 这个脚本 支持 Jinja1 和 Jinja2 。安装后， jinja 和 htmljinja 两种文件类型可用。前者 给基于文本的模板，后者给 HTML 模板。

把这些文件复制到你的 syntax 文件夹。

从其它的模板引擎切换

如果你过去使用一个不同的模板引擎，并且想要转换到 Jinja2 ，这里是一份简小的 指导展示了一些常见的、相似的 Python 文本模板引擎基本语法和语义差异

Jinja1

Jinja2 与 Jinja1 在 API 使用和模板语法上最为兼容。下面的列表解释了 Jinja1 和 Jinja2 的区别。

API

加载器

Jinja2 使用不同的加载器 API 。因为模板的内部表示更改，不再支持 memcached 这样的外部缓存系统。模板的内存开销与常规的 Python 模块相当，外部缓存不能 带来优势。如果你以前使用了一个自定义的加载器，请阅读 *loader API* 部分。

从字符串加载模板

在过去，在默认环境配置中使用 jinja.from_string 从字符串生成模板是可能 的。 Jinja2 提供了一个 Template 类来用于做同样的事情，但是需要 可选的额外配置。

自动 Unicode 转换

Jinja1 执行把字节串从一个给定编码到 unicode 对象的自动转换。这个转换不再 被实现，因为它与大多数使用常规 Python ASCII 字节串到 Unicode 转换的库不 一致。一个由 Jinja2 驱动的应用 *必须* 在内部的每个地方都使用 unicode 或 确保 Jinja2 只会被传递 unicode 字符串。

i18n

Jinja1 使用自定义的国际化翻译器。 i18n 现在作为 Jinja2 的一个扩展，并且 使用更简单、更 gettext 友好的接口，并且支持 babel 。更多细节见 *i18n 扩展* 。

内部方法

Jinja1 在环境对象上暴露了诸如 call_function 、 get_attribute 等内部 方法。当它们被标记为一个内部方法，则可以覆盖它们。 Jinja2 并没有等价的 方法。

沙箱

Jinja1 默认运行沙箱模式。实际上只有少数应用使用这一特性，所以这在 Jinja2 中是可选的。更多关于上下执行的细节见 SandboxedEnvironment 。

上下文

Jinja1 有一个上下文栈存储传递到环境的变量。在 Jinja2 中有一个类似的 对象，但它不允许修改也不是单例的。由于继承是动态的，现在当模板求值时 可能存在多个上下文对象。

过滤器和测试

过滤器和测试现在是常规的函数。不再允许使用工厂函数，且也没有必要。

模板

Jinja2 与 Jinja1 的语法几乎相同。区别是，现在宏需要用小括号包裹参数。

此外， Jinja2 允许动态继承和动态包含。老的辅助函数 rendertemplate 作古， 而使用include 。包含不再导入宏和变量声明，因为采用了新的 import 标签。 这个概念在 *导入*文档中做了解释。

另一个改变发生在 for 标签里。特殊的循环变量不再拥有 parent 属性，而 你需要自己给循环起别名。见 *访问父级循环* 了解更多细节。

Django

如果你之前使用 Django 模板，你应该会发现跟 Jinja2 非常相似。实际上， 很多的语法元素看起来相同，工作也相同。

尽管如此， Jinja2 提供了更多的在之前文档中描述的语法元素，并且某些 工作会有一点不一样。

本节介绍了模板差异。由于 API 是从根本上不同，我们不会再这里赘述。

方法调用

在 Django 中，方法调用是隐式的。在 Jinja2 中，你必须指定你要调用一个对象。如 此，这段 Django 代码:

```
{% for page in user.get_created_pages %}
    ...
{% endfor %}
```

在 Jinja 中应该是这样:

```
{% for page in user.get_created_pages() %}
    ...
{% endfor %}
```

这允许你给函数传递变量，且宏也使用这种方式，而这在 Django 中是不可能的。

条件

在 Django 中你可以使用下面的结构来判断是否相等:

```
{% ifequal foo "bar" %}
    ...
{% else %}
    ...
{% endifequal %}
```

在 Jinja2 中你可以像通常一样使用 if 语句和操作符来做比较:

```
{% if foo == 'bar' %}
    ...
{% else %}
    ...
{% endif %}
```

你也可以在模板中使用多个 elif 分支:

```
{% if something %}
    ...
{% elif otherthing %}
    ...
{% elif foothing %}
    ...
{% else %}
    ...
{% endif %}
```

过滤器参数

Jinja2 为过滤器提供不止一个参数。参数传递的语法也是不同的。一个这样的 Django 模板:

```
{{ items|join:", " }}
```

在 Jinja2 中是这样:

```
{{ items|join(', ') }}
```

实际上这有点冗赘，但它允许不同类型的参数——包括变量——且不仅是一种。

测试

除过滤器外，同样有用 is 操作符运行的测试。这里是一些例子:

```
{% if user.user_id is odd %}
    {{ user.username|e }} is odd
{% else %}
    hmm. {{ user.username|e }} looks pretty normal
{% endif %}
```

循环

因为循环与 Django 中的十分相似，仅有的不兼容是 Jinja2 中循环上下文的特殊变 量名为loop 而不是 Django 中的 forloop 。

周期计

Jinja 中没有 {% cycle %} 标签，因为它是隐式的性质。而你可以用循环对象 的 cycle 方法实现几乎相同的东西。

下面的 Django 模板:

```
{% for user in users %}
    <li class="{% cycle 'odd' 'even' %}">{{ user }}</li>
{% endfor %}
```

Jinja 中看起来是这样:

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

没有与 {% cycle ... as variable %} 等价的。

Mako

如果你迄今使用 Mako 并且想要转换到 Jinja2 ，你可以把 Jinja2 配置成 Mako 一 样:

```
env = Environment('<%', '%>', '${', '}', '%')
```

环境配置成这样后， Jinja2 应该可以解释一个 Mako 模板的小型子集。 Jinja2 不支持 嵌入 Python 代码，所以你可能需要把它们移出模板。 def 的语法（在 Jinja2 中 def 被叫做宏）并且模板继承也是不同的。下面的 Mako 模板:

```
<%inherit file="layout.html" />
<%def name="title()">Page Title</%def>
<ul>
% for item in list:
    <li>${item}</li>
% endfor
</ul>
```

在以上配置的 Jinja2 中看起来是这样:

```
<% extends "layout.html" %>
<% block title %>Page Title<% endblock %>
<% block body %>
<ul>
% for item in list:
    <li>${item}</li>
% endfor
</ul>
<% endblock %>
```

提示和技巧

这部分文档展示了一些 Jinja2 模板的提示和技巧。

Null-Master 退回

Jinja2 支持动态继承并且只要没有 extends 标签被访问过，就不分辨父模板和子模 板。而这会导致令人惊讶的行为：首个 extends 标签前的包括空白字符的所有东西 会被打印出来而不是被忽略，这也可以用作一个巧妙的方法。

通常，继承一个模板的子模板来添加基本的 HTML 骨架。而把 extends 标签放在 if 标签中，当 standalone 变量值为 false 时（按照默认未定义也为 false ）继 承布局模板是可行的。此外，一个非常基本的骨架会被添加到文件，这样如果确实带 置为 True 的standalone 渲染，一个非常基本的 HTML 骨架会被添加:

```
{% if not standalone %}{% extends 'master.html' %}{% endif -%}
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<title>{% block title %}The Page Title{% endblock %}</title>
<link rel="stylesheet" href="style.css" type="text/css">
{% block body %}
  <p>This is the page body.</p>
{% endblock %}
```

交替的行

如果你想要对一个表格或列表中的每行使用不同的样式，可以使用 loop 对象的 cycle 方法:

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

cycle 可接受无限数目的字符串。每次遭遇这个标签，列表中的下一项 就会被渲染。

高亮活动菜单项

你经常想要一个带有活动导航项的导航栏。这相当容易实现。因为在 block 外 的声明在子模板中是全局的，并且在布局模板求值前执行，在子模板中定义活动的 菜单项:

```
{% extends "layout.html" %}
{% set active_page = "index" %}
```

布局模板之后就可以访问 active_page 。此外，这意味着你可以为它定义默认 值:

```
{% set navigation_bar = [
    ('/', 'index', 'Index'),
    ('/downloads/', 'downloads', 'Downloads'),
    ('/about/', 'about', 'About')
] -%}
{% set active_page = active_page|default('index') -%}
...
<ul id="navigation">
{% for href, id, caption in navigation_bar %}
  <li{% if id == active_page %} class="active"{% endif
  %}><a href="{{ href|e }}">{{ caption|e }}</a>/li>
{% endfor %}
</ul>
...
```

访问父级循环

特殊的 loop 变量总是指向最里层的循环。如果想要访问外层的循环，可以给它 设置别名:

```
<table>
{% for row in table %}
  <tr>
  {% set rowloop = loop %}
  {% for cell in row %}
    <td id="cell-{{ rowloop.index }}-{{ loop.index }}>{{ cell }}</td>
  {% endfor %}
  </tr>
{% endfor %}
</table>
```