docs.jinkan.org

API — Jinja2 2.7 documentation

153-192 分钟

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 und Gretel 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:

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 *False* depending 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.

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 in *target*. 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 not *None*, `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 that 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 a `resolve()` 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 a `contextfunction()` 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 the *encoding* 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 the *encoding* 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 return *True* if the template is still up to date. For more details have a look at `BaseLoader.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 override `load_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 as *django.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 the *ignore_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 returns *Markup*.

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; 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 %}
```

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.