Pug.js模板引擎中文文档

Pug 是一款健壮、灵活、功能丰富的模板引擎，专门为 Node.js 平台开发。Pug 是由 Jade 改名而来。

目　录

致谢

当前文档 《Pug.js模板引擎中文文档》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-03-09。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：pugjs https://github.com/pugjs/pug-zh-cn

文档地址：http://www.bookstack.cn/books/pug-zh-cn

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

简体中文版 Pug 文档

本版本库存有简体中文版的 Pug 文档，可访问 https://pugjs.org/zh-cn/ 查看。

Simplified Chinese documentation of Pug

This repository contains the documentation of Pug in Simplified Chinese. It will be live at https://pugjs.org/zh-cn/.

入门指南 ~~ Getting Started

安装 ~~ Installation

Pug 可以通过 npm 获得:

```
$ npm install pug
```

概要 ~~ Overview

Pug 的渲染操作一般来说是相当简单的。 `pug.compile()` 会把 Pug 代码编译成一个 JavaScript 函数，并且这个函数有一个参数可用于传入数据（局部变量， `locals` ）。调用这个编译出来的函数，并且传入您的数据，很好！ 这时返回的就是用您提供的数据渲染的 HTML 字符串了。

这个编译出来的函数可以被重复使用，也可以传入不同的数据。

```
//- template.pug
p #{name}的 Pug 代码！
```

```
const pug = require('pug');

// 编译这份代码
const compiledFunction = pug.compileFile('template.pug');

// 渲染一组数据
console.log(compiledFunction({
  name: '李莉'
}));
// "<p>李莉的 Pug 代码！</p>"

// 渲染另外一组数据
console.log(compiledFunction({
  name: '张伟'
}));
// "<p>张伟的 Pug 代码！</p>"
```

Pug 也提供了 `pug.render()` 系列的函数，它们把编译和渲染两个步骤合二为一。当然，在每次

执行 `render` 的时候，这样一个模板函数都需要被重新编译一遍，这会在一定程度上影响性能。但同时，您也可以在执行 `render` 的时候配合使用 `cache` 选项，它将会把编译出来的函数自动存储到内部缓存中。

```
1. const pug = require('pug');
2.
3. // 编译并使用一组数据渲染 template.pug
4. console.log(pug.renderFile('template.pug', {
5.   name: 'Timothy'
6. }));
7. // "<p>Timothy 的 Pug 代码！</p>"
```

与 Express 集成 ~~ Express Integration

Express 良好地集成了 Pug。这是一个流行的 Node.js 网站框架，Pug 在其中扮演一个视图引擎的角色。您可以阅读 Express 优秀的文档来了解 Express 是如何与 Pug 集成的。

生产环境下的默认配置 ~~ Production Defaults

在 Express 框架里，环境变量 `NODE_ENV` 用来告知网站应用程序：它执行的环境是开发环境还是生产环境。Express 和 Pug 都会在生产环境下调整一些默认配置，以给用户提供更好的开箱即用的体验。特别是当 `process.env.NODE_ENV` 设置为 `'production'` 、Pug 配合 Express 使用的时候，默认 `compileDebug` 为 `false` ， `cache` 为 `true` 。

如果需要覆盖 `compileDebug` 和 `cache` 的默认配置，您可以在 `app.locals` 或者 `res.locals` 对象里设置各自对应的选项。 `cache` 选项也可以通过 Express 的 `app.disable` / `enable('view cache')` 来设定。更多的细节可以阅读 Express 的 API 参考文档。

API 参考文档 ~~ API Reference

此页面将详细说明如何 JavaScript API 来对 Pug 代码进行渲染。

::: float info 提示 现在 Pug 可以在您的浏览器控制台上使用！想测试各种 Pug 的 API，您可以尝试在控制台输入：

```
pug.render('p Hello world!');
```

:::

选项 ~~ Options

所有的 API 方法都可以使用以下的选项：

```
filename
~ string
~ 要编译的代码的文件名。用于异常信息以及（使用相对路径的）包含（`include`）和扩展
（`extends`）操作。默认值是 `'Pug'`。
basedir
~ string
~ 模板里所有绝对定位的根目录。
doctype
~ string
~ 如果 doctype 没有出现模板里（比如模板只渲染一个 HTML 片段），那么您可以在这里指定它。在一些需要控制自闭合标签和布尔值属性的代码样式的时候非常有用。您可以阅读 [doctype]
(../language/doctype.html#doctype-option) 的说明来了解更多细节。
pretty
~ boolean | string
~ （不赞成使用）在输出的 HTML 里添加 `'  '` 这样的空格缩进来获得更好的代码可读性。如果这里指定了一个字符串（比如 `'\t'`），那么将会使用它作为控制缩进的字符。我们强烈建议您不要使用这个选项，它改变解释器和空格渲染工作的方式会极其频繁地导致一些错误。我们将会在未来移除这个功能。默认为 `false`。
filters
~ object
~ 存放[自定义过滤器]($src-language-filters.html#custom-filters)的哈希表。默认为
`undefined`。
self
~ boolean
```

~ 是否使用一个叫做 `self` 的命名空间来存放局部变量。这可以加速编译的过程，但是，相对于原来书写比如 `variable` 来访问局部变量，您将需要改为 `self.variable` 来访问它们。默认为 `false`。
debug
~ boolean
~ 当设置为 `true` 时，编译产生的函数代码会被记录到标准输出。
compileDebug
~ boolean
~ 当设置为 `true` 时，源代码会被包含在编译出来的模板函数中，用于提供更详实的错误信息（这在开发时会有用）。它默认是启用的，除非是在 [Express](https://expressjs.com/) 的生产环境模式中。
globals
~ Array<string>
~ 该数组用于向模板中添加全局对象的名字，编译器将保证它们不被局部作用域影响。
cache
~ boolean
~ 当设置为 `true` 时，编译出来的函数会被缓存下来。此时必须填写 `filename` 选项作为缓存的索引字段。该选项仅用于 `render` 函数。默认为 `false`。
inlineRuntimeFunctions
~ boolean
~ 相对于使用 `require` 来获得公用的运行时函数，是否需要直接嵌入这些运行时函数。在 `compileClient` 函数里默认是 `true`，因此就不需要再手动包含那些函数（从而让其能在浏览器上运行）。在其他的 `compile` / `render` 系列函数中，默认是 `false`。
name
~ string
~ 模板函数的名称。仅用于 `compileClient` 函数。默认值是 `'template'`。

方法 ~~ Methods

pug.compile(source, ?options)

把一个 Pug 模板编译成一个可多次使用、可传入不同局部变量渲染的函数。

source
~ string
~ 需要编译的 Pug 代码
options
~ ?options
~ 存放选项的对象

```parameter-list (returns) returns ~ function ~ 一个可以从包含局部变量的对象渲染生成出 HTML 的函数
```

```
1.
2.  ```js
3.  var pug = require('pug');
4.
5.  // Compile a function
6.  var fn = pug.compile('string of pug', options);
7.
8.  // Render the function
9.  var html = fn(locals);
10. // => '<string>of pug</string>'
```

pug.compileFile(path, ?options)

从文件中读取一个 Pug 模板，并编译成一个可多次使用、可传入不同局部变量渲染的函数。

```
1.  path
2.  ~ string
3.  ~ 需要编译的 Pug 代码文件的位置
4.  options
5.  ~ ?options
6.  ~ 存放选项的对象
```

```parameter-list (returns) returns ~ function ~ 一个可以从包含局部变量的对象渲染生成出 HTML 的函数

```
1.
2.  ```js
3. var pug = require('pug');
4.
5. // 编译出一个函数
6. var fn = pug.compileFile('到 Pug 代码文件的路径', options);
7.
8. // 渲染它
9. var html = fn(locals);
10. // => '从 Pug 生成的 HTML 代码'
```

## pug.compileClient(source, ?options)

把一个 Pug 模板编译成一份 JavaScript 代码字符串，它可以直接用在浏览器上而不需要 Pug 的运行时库。
```

```
source
~ string
~ 需要编译的 Pug 代码
options
~ ?options
~ 存放选项的对象
```

```parameter-list (returns) returns ~ string ~ 一份 JavaScript 代码字符串```

```

```js
var pug = require('pug');

// 编译出一个函数
var fn = pug.compileClient('string of pug', options);

// 渲染它
var html = fn(locals);
// => 'function template(locals) { return "从 Pug 生成的 HTML 代码"; }'
```

pug.compileClientWithDependenciesTracked(source, ?options)

与 `compileClient` 相似，但这个函数返回的是这样一个结构：

```
{
  'body': 'function (locals) {...}',
  'dependencies': ['filename.pug']
}
```

您应该在需要 `dependencies` 来实现一些诸如“监视 Pug 文件变动”的功能的时候，才使用该函数。

pug.compileFileClient(path, ?options)

从文件中读取一个 Pug 模板并编译成一份 JavaScript 代码字符串，它可以直接用在浏览器上而不需要 Pug 的运行时库。

```
path
~ string
~ 需要编译的 Pug 代码文件的位置
```

```
options
~ ?options
~ 存放选项的对象
options.name
~ string
~ 如果您传递了 `.name` 属性给选项对象，那么编译出来的函数名称就会被换成这个属性指定的名称。
```

```parameter-list (returns) returns ~ string ~ 一份 JavaScript 代码字符串```

```

首先，写下我们的模板……

```pug
h1 这是一个 Pug 模板
h2 作者：#{author}
```

然后将 Pug 编译为函数代码字符串。

```
var fs = require('fs');
var pug = require('pug');

// 将模板编译为一个函数的代码字符串
var jsFunctionString = pug.compileFileClient('/path/to/pugFile.pug', {name:
"fancyTemplateFun"});

// 或许您还想要把模板编译到一个叫做 templates.js 的文件里，让浏览器直接使用。
fs.writeFileSync("templates.js", jsFunctionString);
```

您获得的输出的结果可能类似下面的内容（即被写入 `templates.js` 中的代码）：

```
function fancyTemplateFun(locals) {
  var buf = [];
  var pug_mixins = {};
  var pug_interp;

  var locals_for_with = (locals || {});

  (function (author) {
    buf.push("<h1>这是一个 Pug 模板</h1><h2>作者："
      + (pug.escape((pug_interp = author) == null ? '' : pug_interp))
      + "</h2>");
  }.call(this, "author" in locals_for_with ?
```

```
    locals_for_with.author : typeof author !== "undefined" ?
      author : undefined)
  );

  return buf.join("");
}
```

请确保您已经把 Pug 的运行时库（ `node_modules/pug/runtime.js` ）传给了浏览器，从而让浏览器能够顺利执行您刚才编译出来的函数。

```
<!DOCTYPE html>
<html>
  <head>
    <script src="/runtime.js"></script>
    <script src="/templates.js"></script>
  </head>

  <body>
    <h1>这是一个神奇的模板。</h1>

    <script type="text/javascript">
      var html = window.fancyTemplateFun({author: "enlore"});
      var div = document.createElement("div");
      div.innerHTML = html;
      document.body.appendChild(div);
    </script>
  </body>
</html>
```

pug.render(source, ?options, ?callback)

```
source
~ string
~ 需要渲染的 Pug 代码
options
~ ?options
~ 存放选项的对象，同时也直接用作局部变量的对象
callback
~ ?function
~ Node.js 风格的回调函数，用于接收渲染结果。**注意：这个回调是同步执行的。**
```

```parameter-list (returns) returns ~ string ~ 渲染出来的 HTML 字符串
```

```
1.
2.  ```js
3.  var pug = require('pug');
4.
5.  var html = pug.render('string of pug', options);
6.  // => '<string>of pug</string>'
```

pug.renderFile(path, ?options, ?callback)

```
1.  path
2.  ~ string
3.  ~ 需要渲染的 Pug 代码文件的位置
4.  options
5.  ~ ?options
6.  ~ 存放选项的对象，同时也直接用作局部变量的对象
7.  callback
8.  ~ ?function
9.  ~ Node.js 风格的回调函数，用于接收渲染结果。**注意：这个回调是同步执行的。**
```

```parameter-list (returns) returns ~ string ~ 渲染出来的 HTML 字符串
```

```
1.
2.  ```js
3.  var pug = require('pug');
4.
5.  var html = pug.renderFile('path/to/file.pug', options);
6.  // ...
```

属性 ~~ Properties

pug.filters

这是一个存放自定义过滤器的哈希表。

这个对象和 `filters` 选项有着同样的语义，但它是全局地应用在所有 Pug 编译过程的。当一个过滤器名称同时出现在了 `pug.filters` 和 `options.filters` 里，本 `filters` 选项将被优先选择。

::: float warning 不赞成使用 不赞成使用这个属性，应该使用 `filters` 选项取而代之。
:::

迁移到 Pug v2 ~~ Migrating to Pug 2

Pug v2 已经在 2016 年 8 月发布。为了尽可能改善新版本的体验，我们不得不作出决定，移除、或者不赞成使用一些 API 和未归档的特性。我们努力让这些变更尽可能不具破坏性。当前，这些变更大多数会以控制台输出的方式进行警告。

此页面将详细介绍您应该如何将代码从旧版本迁移到最新版本的 Pug 上。

新的名称 ~~ Project Rename

因为商标方面的问题，这个项目的名称已经在 Pug v2 发布之际从“Jade”变更为“Pug”。这也意味着官方支持的文件扩展名从 `.jade` 变为 `.pug` 。尽管依然支持 `.jade` ，但这是不赞成的，我们建议所有的用户都立刻将其改为 `.pug` 。

已经移除的语言特性 ~~ Removed Language Features

绝大多数被移除的内容都可以被 pug-lint 自动检测出来，这是我们官方提供的代码规范器。

传统 Mixin 调用 ~~ Legacy Mixin Call

```
\\\\\\\\\\ old.pug <
//- 旧版本

mixin foo('whatever')
\\\\\\\\\\ new.pug >
//- 新版本

+foo('whatever')
```

我们移除了传统的调用 mixin 的语句，这样可以更容易区分 Mixin 的声明和调用。所有这类旧语句在 Jade v1 里都已经警告。

属性嵌入 ~~ Attribute Interpolation

```
\\\\\\\\\\ old.pug <
//- 旧版本

a(href='#{link}')
```

```
5.
6.  a(href='before#{link}after')
7.  \\\\\\\\\\ new.pug >
8.  //- 新版本
9.
10. a(href=link)
11.
12. //- (在 Node.js/io.js ≥ 1.0.0)
13. a(href=`before${link}after`)
14. //- (任何场合)
15. a(href='before' + link + 'after')
```

我们移除了在标签属性里的嵌入支持。它给实现平添了不必要的复杂度，而且也让用户不易注意到属性的赋值其实可以是任意的 JavaScript 表达式。阅读属性的文档来了解更多关于标签属性的语法。

带有前缀的 `each` 语法 ~~ Prefixed `each` Syntax

```
1.  \\\\\\\\\\ old.pug <
2.  //- 旧版本
3.
4.  - each a in b
5.    = a
6.
7.  - for a in b
8.    = a
9.  \\\\\\\\\\ new.pug >
10. //- 新版本
11.
12. each a in b
13.   = a
14.
15. for a in b
16.   = a
```

这里的 `each` 并非 JavaScript 的语法，在 JavaScript 代码行中使用 `each` “关键字”会让人感到非常困惑。没有括号的 `for` 关键字也是同样的情况。

只需要简单地删去 `-` ，您的代码应该就能重新工作。

已删除的 API ~~ Removed API

以下导出属性和编译选项已经被移除。请确保您的代码没有使用它们。

属性 ~~ Properties

`doctype`

此前，未归档的对象 `jade.doctype` 是一个存放 `doctype 缩写`的哈希表。用户可以通过扩充这个对象来自定义额外的 doctype 缩写，或者修改已有的 doctype 缩写。

在 Pug v2 中，这个对象已经从 Pug 分离出来，单独成为一个叫做 `doctypes` 的包。如果您要扩充 doctype 缩写，可以写成一个 `codeGen` 的插件。

`nodes`

此前，未归档的对象 `jade.nodes` 是一个存放（未归档的）Jade 抽象语法树节点的构造函数的哈希表。

在 Pug v2 中，我们弃用了这种做法，取而代之的是使用抽象语法树节点的 `type` 属性实现鸭子类型。

`selfClosing`

此前，未归档的数组 `jade.selfClosing` 可以用于添加或者修改自闭合标签的条目。

在 Pug v2 中，这个数组已经从 Pug 分离出来，单独成为一个叫做 `void-elements` 的包。如果您要修改这个数组，可以写成一个 `codeGen` 的插件。

`utils`

此前，`jade.utils` 对象包含了三个模板引擎内部比较有用的函数：

`utils.merge` 已经从 Pug 中移除并不再使用。其功能大体上可以用 ECMAScript 2015 的 `Object.assign` 方法及其他变种来实现。

`utils.stringify` 已经从 Pug 分离出来到叫做 `js-stringify` 的包，同时有额外的跨站脚本攻击保护。建议所有用户都改用此代码包。

`utils.walkAST` 已经分离到一个叫做 `pug-walk` 的包。

`Compiler`, `Lexer`, `Parser`

此前，未归档的 Jade 的编译器（Compiler）、词法分析器（Lexer）和解析器（Parser）通过这三个属性导出。用户可以从这些类创建自己的编译器、词法分析器和解析器，从而自定义编译的行为。

Pug v2 允许通过插件自定义编译的行为，同时移除这些导出属性。

对应到 Pug v2 中，这几个类现在已经分为 `pug-code-gen`，`pug-lexer` 和 `pug-parser` 这几

个包，并且有各种与旧版本不兼容的变更。

选项 ~~ Options

`compiler`, `lexer`, `parser`

这些选项曾被用于已经被移除的 `Compiler`, `Lexer` 和 `Parser` 类。

`client`

该选项曾用于模板函数的编译。大约从 2013 年起不推荐使用，并用 `compileClient` 函数代替，从那时起已经进行了警告。

- 属性
- 分支条件
- 代码
- 注释
- 条件
- Doctype
- 过滤器
- 包含
- 继承与扩展
- 嵌入
- 迭代
- Mixin
- 纯文本
- 标签

属性 Attribute ~~ Attributes

标签属性和 HTML 语法非常相似，但它们的值就是普通的 JavaScript 表达式。您可以用逗号作为属性分隔符，不过不加逗号也是允许的。

（注意：本页的示例中，包含管道符号 `|` 的行是用于控制空格的。）

```
a(href='baidu.com') 百度
|
|
a(class='button' href='baidu.com') 百度
|
|
a(class='button', href='baidu.com') 百度
```

所有 JavaScript 表达式都能用：

```
//- 已登录
- var authenticated = true
body(class=authenticated ? 'authed' : 'anon')
```

多行属性 ~~ Multiline Attributes

如果您有很多属性，您可以把它们分几行写：

```
input(
  type='checkbox'
  name='agreement'
  checked
)
```

如果您有一个很长属性，并且您的 JavaScript 运行时引擎支持 ECMAScript 2015 模板字符串（包括 Node.js 和 io.js v1.0.0 和更新的版本），您可以使用它来写属性值：

``pug-preview (features=['templatestrings']) input(data-json= { “非常”: “长的”, “数据”: true } `)

```

## 用引号括起来的属性 ~~ Quoted Attributes

```

如果您的属性名称中含有某些奇怪的字符，并且可能会与 JavaScript 语法产生冲突的话，请您将它们使用 `""` 或者 `''` 括起来。您还可以使用逗号来分割不同的属性。这种属性的例子有 Angular 2 中经常用到的 `[]` 和 `()`。

```pug-preview
//- 在这种情况下，`(click)` 会被当作函数调用而不是
//- 属性名称，这会导致一些稀奇古怪的错误。
div(class='div-class' (click)='play()')
```

```
div(class='div-class', (click)='play()')
div(class='div-class' '(click)'='play()')
```

属性嵌入 ~~ Attribute Interpolation

::: float danger 注意 在 Pug / Jade 的旧版本中支持一种嵌入语法:

```
a(href="/#{url}") Link
```

这种语法 已经不再被支持！ 它的相关替代可以在本文档的下面部分找到。另外请您参考我们的迁移指南以获取更多 Pug v2 与之前版本的不兼容情形。 :::

如果您想要在您的属性当中使用变量的话，您可以这样做:

1. 直接使用 JavaScript 写那个属性:

   ```
   - var url = 'pug-test.html';
   a(href='/' + url) 链接
   |
   |
   - url = 'https://example.com/'
   a(href=url) 另一个链接
   ```

2. 如果您的 JavaScript 运行时支持 ECMAScript 2015 模板字符串(包含在 Node.js/io.js 1.0.0 以及更新的版本当中)，那么您还可以使用这种方式来简化您的属性值:

   ```pug-preview (features=[‘templatestrings’])
   ```

   - var btnType = ‘info’
   - var btnSize = ‘lg’ button(type=’button’ class=’btn btn-‘ + btnType + ‘ btn-‘ + btnSize) | | button(type=’button’ class= btn btn-

`${btnType} btn-${btnSize}`) ```

不转义的属性 ~~ Unescaped Attributes

在默认情况下，所有的属性都经过转义（即把特殊字符转换成转义序列）来防止诸如跨站脚本攻击之类的攻击方式。如果您非要使用特殊符号，您需要使用 `!=` 而不是 `=` 。

```
1. div(escaped="<code>")
2. div(unescaped!="<code>")
```

::: float danger 危险 未经转义的缓存代码十分危险。您必须正确处理和过滤用户的输入来避免跨站脚本攻击。 :::

布尔值属性 ~~ Boolean Attributes

在 Pug 中，布尔值属性是经过映射的，这样布尔值（ `true` 和 `false` ）就能接受了。当没有指定值的时候，默认是 `true` 。

```
1. input(type='checkbox' checked)
2. |
3. |
4. input(type='checkbox' checked=true)
5. |
6. |
7. input(type='checkbox' checked=false)
8. |
9. |
10. input(type='checkbox' checked=true.toString())
```

如果 doctype 是 `html` 的话，Pug 就不会去映射属性，而是使用缩写样式（terse style）（所有浏览器都应该支持）。

```
1. doctype html
2. |
3. |
4. input(type='checkbox' checked)
5. |
6. |
7. input(type='checkbox' checked=true)
8. |
```

```
|
input(type='checkbox' checked=false)
|
|
input(type='checkbox' checked=true && 'checked')
```

样式属性 ~~ Style Attributes

`style` （样式）属性可以是一个字符串（就像其他普通的属性一样）还可以是一个对象，这在部分样式是由 JavaScript 生成的情况下非常方便。

```
a(style={color: 'red', background: 'green'})
```

类属性 ~~ Class Attributes

`class` （类）属性可以是一个字符串（就像其他普通的属性一样）还可以是一个包含多个类名的数组，这在类是由 JavaScript 生成的情况下非常方便。

```
- var classes = ['foo', 'bar', 'baz']
a(class=classes)
|
|
//- the class attribute may also be repeated to merge arrays
a.bang(class=classes class=['bing'])
```

它还可以是一个将类名映射为 `true` 或 `false` 的对象，这在使用条件性的类的时候非常有用。

```
- var currentUrl = '/about'
a(class={active: currentUrl === '/'} href='/') Home
|
|
a(class={active: currentUrl === '/about'} href='/about') About
```

类的字面值 ~~ Class Literal

类可以使用 `.classname` 语法来定义：

```
a.button
```

考虑到使用 `div` 作为标签名这种行为实在是太常见了，所以如果您省略掉标签名称的话，它就是默认值：

```
.content
```

ID 的字面值 ~~ ID Literal

ID 可以使用 `#idname` 语法来定义：

```
a#main-link
```

考虑到使用 `div` 作为标签名这种行为实在是太常见了，所以如果您省略掉标签名称的话，它就是默认值：

```
#content
```

&attributes

读作 “and attributes”， `&attributes` 语法可以将一个对象转化为一个元素的属性列表。

```
div#foo(data-bar="foo")&attributes({'data-foo': 'bar'})
```

这个对象不一定必须是一个字面值，它同样也可以是一个包含值的对象（参见 Mixin 属性）。

```
- var attributes = {};
- attributes.class = 'baz';
div#foo(data-bar="foo")&attributes(attributes)
```

::: float danger 危险 使用 `&attributes` 赋值的属性并不会经过自动转义过程。您必须正确处理和过滤用户的输入来避免跨站脚本攻击(XSS)。 但是如果您是从一个 mixin 调用中使用 `attributes` 的话，这个过程就是自动的。 :::

分支条件 Case ~~ Case

`case` 是 JavaScript 的 `switch` 指令的缩写，并且它接受如下的形式：

```
- var friends = 10
case friends
  when 0
    p 您没有朋友
  when 1
    p 您有一个朋友
  default
    p 您有 #{friends} 个朋友
```

分支传递 (Case Fall Through) ~~ Case Fall Through

您可以像 JavaScript 中的 `switch` 语句那样使用传递（fall through）。

```
- var friends = 0
case friends
  when 0
  when 1
    p 您的朋友很少
  default
    p 您有 #{friends} 个朋友
```

不同之处在于，在 JavaScript 中，传递会在明确地使用 `break` 语句之前一直进行。而在 Pug 中则是，传递会在遇到非空的语法块前一直进行下去。

在某些情况下，如果您不想输出任何东西的话，您可以明确地加上一个原生的 `break` 语句：

```
- var friends = 0
case friends
  when 0
    - break
  when 1
    p 您的朋友很少
  default
    p 您有 #{friends} 个朋友
```

块展开 ~~ Block Expansion

您也可以使用块展开的语法：

```
- var friends = 1
case friends
  when 0: p 您没有朋友
  when 1: p 您有一个朋友
  default: p 您有 #{friends} 个朋友
```

代码 Code ~~ Code

Pug 为您在模板中嵌入 JavaScript 提供了可能。这里有三种类型的代码。

不输出的代码 ~~ Unbuffered Code

用 `-` 开始一段不直接进行输出的代码，比如：

```
- for (var x = 0; x < 3; x++)
  li item
```

Pug 也支持把它们写成一个块的形式：

```
-
  var list = ["Uno", "Dos", "Tres",
          "Cuatro", "Cinco", "Seis"]
each item in list
  li= item
```

带输出的代码 ~~ Buffered Code

用 `=` 开始一段带有输出的代码，它应该是可以被求值的一个 JavaScript 表达式。为安全起见，它将被 HTML 转义：

```
p
  = '这个代码被 <转义> 了！'
```

也可以写成行内形式，同样也支持所有的 JavaScript 表达式：

```
p= '这个代码被 <转义> 了！'
```

不转义的、带输出的代码 ~~ Unescaped Buffered Code

用 `!=` 开始一段不转义的，带有输出的代码。这将不会做任何转义，所以用于执行用户的输入将会不安全：

```
1.  p
2.    != '这段文字 <strong>没有</strong> 被转义！'
```

同样也可以写成行内形式，支持所有的 JavaScript 表达式:

```
1.  p!= '这段文字' + ' <strong>没有</strong> 被转义！'
```

::: float danger 危险 不转义的输出可能是危险的，您必须确保任何来自用户的输入都是安全可靠的，以防止发生跨站脚本攻击（XSS）。 :::

注释 Comment ~~ Comments

带输出的注释和 JavaScript 的单行注释类似，它们像标签，能生成 *HTML* 注释，而且必须独立一行。

```
// 一些内容
p foo
p bar
```

Pug 也同样提供了不输出的注释，只需要加上一个横杠。这些内容仅仅会出现在 Pug 模板之中，不会出现在渲染后的 HTML 中。

```
//- 这行不会出现在结果里
p foo
p bar
```

块注释 ~~ Block Comments

一个格式正确的块注释应该像这样:

```
body
  //-
    给模板写的注释
    随便写多少字
    都没关系。
  //
    给生成的 HTML 写的注释
    随便写多少字
    都没关系。
```

条件注释 ~~ Conditional Comments

Pug 没有特殊的语法来表示条件注释（conditional comments）。条件注释是一种用于分辨 Internet Explorer 新老版本的特殊标记。不过因为所有以 `<` 开头的行都会被当作纯文本，因此直接写一个 HTML 风格的条件注释也是没问题的。

```
doctype html

```

```
<!--[if IE 8]>
<html lang="en" class="lt-ie9">
<![endif]-->
<!--[if gt IE 8]><!-->
<html lang="en">
<!--<![endif]-->

body
  p Supporting old web browsers is a pain.

</html>
```

条件 Conditional ~~ Conditionals

Pug 的条件判断的一般形式的括号是可选的，所以您可以省略掉开头的 `-` ，效果是完全相同的。类似一个常规的 JavaScript 语法形式。

```
- var user = { description: 'foo bar baz' }
- var authorised = false
#user
  if user.description
    h2.green 描述
    p.description= user.description
  else if authorised
    h2.blue 描述
    p.description.
      用户没有添加描述。
      不写点什么吗……
  else
    h2.red 描述
    p.description 用户没有描述
```

Pug 同样也提供了它的反义版本 `unless` （下面示例的效果是等价的）：

```
\\\\\\\\\\ a.pug <
unless user.isAnonymous
  p 您已经以 #{user.name} 的身份登录。
\\\\\\\\\\ b.pug >
if !user.isAnonymous
  p 您已经以 #{user.name} 的身份登录。
```

Doctype

```
doctype html
```

Doctype 缩写 ~~ Doctype Shortcuts

以下是一些常用的 doctype 的缩写:

```
```

自定义 doctype ~~ Custom Doctypes

您也可以自定义一个 doctype 字面值：

```pug-preview
doctype html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN"
```

Doctype 选项 ~~ Doctype Option

Doctype 会影响 Pug 的编译结果。比如自闭合的标签是以 `/>` 还是以 `>` 结束，这取决于指定了是 HTML 还是 XML。布尔值属性也同样会受到影响。

如果因为某些原因，不能在模板里使用 `doctype` 关键字（比如需要渲染的是 HTML 的一个片段），但您依然需要指定 doctype 的时候，您就可以通过 `doctype` 选项来设置了。

```
var pug = require('./');

var source = 'img(src="foo.png")';

pug.render(source);
// => '<img src="foo.png"/>'

pug.render(source, {doctype: 'xml'});
// => '<img src="foo.png"></img>'

pug.render(source, {doctype: 'html'});
// => '<img src="foo.png">'
```

过滤器 Filter ~~ Filters

过滤器可以让您在 Pug 中使用其他的语言。它们接受传入一个文本块的内容。向过滤器传递参数，只需要将参数如同标签属性一样，放在括号里即可，如： `:less(ieCompat=false)`

所有的 JSTransformer 模块都可以被用作 Pug 的过滤器。有名的过滤器比如 `:babel` 、 `:uglify-js` 、 `:scss` 和 `:markdown-it` 。阅读这些 JSTransformer 的文档来了解它们具体支持什么选项。如果您找不到一个您所期望的过滤器，您也可以自己写自定义过滤器。

举一个例子，如果您想要能在您的 Pug 模板中使用 CoffeeScript 语言和 Markdown 语言（使用 Markdown-it 渲染），那么您首先要确定这些东西已经安装好：

```
$ npm install --save jstransformer-coffee-script
$ npm install --save jstransformer-markdown-it
```

现在可以渲染下面这个模板了。

```
:markdown-it(linkify langPrefix='highlight-')
  # Markdown

  Markdown document with http://links.com and

  ```js
  var codeBlocks;
  ```
script
  :coffee-script
    console.log 'This is coffee script'
```

::: float warning 警告 过滤器在 Pug 编译的时候被渲染，这意味着它们可以很快呈现出来，但是同时也意味着它们不支持动态的内容和选项。

默认情况下，基于 JSTransformer 的过滤器在浏览器上编译的时候也是不可用的，除非那个 JSTransformer 模块被明确地封装、引入了，并且通过一个 CommonJS 平台，比如 Browserify 或者 Webpack，使之可以在浏览器上执行。事实上，您现在正在阅读的这个页面就使用了 Browserify 使得过滤器能够在浏览器上执行。

把模板放在服务器上编译渲染则不存在这个限制。 :::

行内语法 ~~ Inline Syntax

如果过滤器的内容很短，我们甚至可以像一个 HTML 标签一样去使用它：

```
p
  :markdown-it(inline) **加粗文字**

p.
  在漫无边际的无聊纯文本构成的垃圾文字的海洋上，
  突然一只野生的 #[:markdown-it(inline) *Markdown*]
  出现在了我的视野。
```

嵌套过滤器 ~~ Nested Filters

可以在一行里同时指定多个过滤器来对一个文本块进行处理。文本首先被传递到最后的过滤器，然后它的结果会被传到倒数第二个过滤器作为输入，以此类推。

在下面的例子里，内容将首先被 `babel` 过滤器处理，然后是 `cdata-js` 。

```
script
  :cdata-js:babel(presets=['es2015'])
    const myFunc = () => `这是一行在 CD${'ATA'} 里的 ECMAScript 2015 代码`;
```

自定义过滤器 ~~ Custom Filters

您可以通过 `filters` 选项给 Pug 提供您自定义的过滤器。

```pug-preview-readonly demo \\\\\ options.js < options.filters = { 'my-own-filter': function (text, options) { if (options.addStart) text = '始\n' + text; if (options.addEnd) text = text + '\n终'; return text; } }; \\\\\ index.pug < p :my-own-filter(addStart addEnd) 过滤 正文 \\\\\ output.html >

始 过滤 正文 终
```

包含 Include ~~ Includes

包含（include）功能允许您把另外的文件内容插入进来。

```
\\\\\\\\\\ index.pug
//- index.pug
doctype html
html
  include includes/head.pug
  body
    h1 我的网站
    p 欢迎来到我这简陋得不能再简陋的网站。
    include includes/foot.pug
\\\\\\\\\\ includes/head.pug
//- includes/head.pug
head
  title 我的网站
  script(src='/javascripts/jquery.js')
  script(src='/javascripts/app.js')
\\\\\\\\\\ includes/foot.pug
//- includes/foot.pug
footer#footer
  p Copyright (c) foobar
```

被包含的文件的路径，如果是一个绝对路径（如 `include /root.pug`），那么前面会加上 `options.basedir` 选项属性来进行解析。否则，路径应该相对于正在被编译的当前文件。

在 Pug v1 里，如果没有给出文件扩展名，会自动加上 `.pug`。但是这个特性在 Pug v2 中这是不赞成使用的。

包含纯文本 ~~ Including Plain Text

被包含的如果不是 Pug 文件，那么就只会当作文本内容来引入。

```
\\\\\\\\\\ index.pug
//- index.pug
doctype html
html
  head
    style
```

```
      include style.css
  body
    h1 我的网站
    p 欢迎来到我这简陋得不能再简陋的网站。
    script
      include script.js
\\\\\\\\\\ style.css
/* style.css */
h1 {
  color: red;
}
\\\\\\\\\\ script.js
// script.js
console.log('真了不起！');
```

使用过滤器包含文本 ~~ Including Filtered Text

您可以合并过滤器和包含语句，从而做到引入文件内容并直接用过滤器处理它们。

```
\\\\\\\\\\ index.pug <
//- index.pug
doctype html
html
  head
    title 一篇文章
  body
    include:markdown-it article.md
\\\\\\\\\\ article.md <
# article.md

这是一篇用 Markdown 写的文章。
\\\\\\\\\\ output.html >
<!DOCTYPE html>
<html>
  <head>
    <title>一篇文章</title>
  </head>
  <body>
    <h1>article.md</h1>
    <p>这是一篇用 Markdown 写的文章。</p>
  </body>
```

```
23.  </html>
```

模板继承 Inheritance ~~ Template Inheritance

Pug 支持使用 `block` 和 `extends` 关键字进行模板的继承。一个称之为“块”（block）的代码块，可以被子模板覆盖、替换。这个过程是递归的。

Pug 的块可以提供一份默认内容，当然这是可选的，见下述 `block scripts`、`block content` 和 `block foot`。

```
1.  //- layout.pug
2.  html
3.    head
4.      title 我的站点 - #{title}
5.      block scripts
6.        script(src='/jquery.js')
7.    body
8.      block content
9.      block foot
10.       #footer
11.         p 一些页脚的内容
```

现在我们来扩展这个布局：只需要简单地创建一个新文件，并如下所示用一句 `extends` 来指出这个被继承的模板的路径。您现在可以定义若干个新的块来覆盖父模板里对应的“父块”。值得注意的是，因为这里的 `foot` 块 没有 被重定义，所以会依然输出“一些页脚的内容”。

在 Pug v1 里，如果没有给出文件扩展名，会自动加上 `.pug`。但是这个特性在 Pug v2 中 这是不赞成使用的。

```
1.  //- page-a.pug
2.  extends layout.pug
3.
4.  block scripts
5.    script(src='/jquery.js')
6.    script(src='/pets.js')
7.
8.  block content
9.    h1= title
10.   - var pets = ['猫', '狗']
11.   each petName in pets
12.     include pet.pug
```

```
//- pet.pug
p= petName
```

同样，也可以覆盖一个块并在其中提供一些新的块。如下面的例子所展示的那样，`content` 块现在暴露出两个新的块 `sidebar` 和 `primary` 用来被扩展。当然，它的子模板也可以把整个 `content` 给覆盖掉。

```
//- sub-layout.pug
extends layout.pug

block content
  .sidebar
    block sidebar
      p 什么都没有
  .primary
    block primary
      p 什么都没有
```

```
//- page-b.pug
extends sub-layout.pug

block content
  .sidebar
    block sidebar
      p 什么都没有
  .primary
    block primary
      p 什么都没有
```

块内容的添补 append / prepend ~~ Block append / prepend

Pug 允许您去替换（默认的行为）、`prepend`（向头部添加内容），或者 `append`（向尾部添加内容）一个块。 假设您有一份默认的脚本要放在 `head` 块中，而且希望将它应用到 每一个页面，那么您可以这样做：

```
//- layout.pug
html
  head
```

```
    block head
      script(src='/vendor/jquery.js')
      script(src='/vendor/caustic.js')
  body
    block content
```

现在假设您有一个页面，那是一个 JavaScript 编写的游戏。您希望把一些游戏相关的脚本也像默认的那些脚本一样放进去，那么您只要简单地 `append` 这个块：

```
//- page.pug
extends layout.pug

block append head
  script(src='/vendor/three.js')
  script(src='/game.js')
```

当使用 `block append` 或者 `block prepend` 时， `block` 关键字是可省略的：

```
//- page.pug
extends layout

append head
  script(src='/vendor/three.js')
  script(src='/game.js')
```

常见错误 ~~ Common mistakes

Pug 模板继承是一个非常强大的功能，您可以借助它将复杂的页面模板拆分成若干个小而简洁的文件。然而，如果您将大量的模板继承、链接在一起，那么有可能会反而把事情弄得更加复杂。

值得注意的是，在扩展模板中，顶级元素（没有缩进的一级）只能是带名字的块，或者是混入的定义。理解这一点非常重要，因为父模板定义了整个页面的总体布局和结构，而扩展的模板只能为其特定的块添补或者替换内容。不妨假设我们创建了一个子模板，尝试在已有的块之外再添加内容。很明显，Pug 将没法知道这个内容最终应该在页面的何处出现。

这也包括不输出的代码，因为它们有可能包含生成标记的代码。如果您需要在子模板中定义变量，有这样一些方法：

- 向 Pug 选项添加变量，或者在父模板中用不输出的代码定义。子模板将会继承这些变量。
- 在子模板的一个块中定义变量。扩展模板至少要有一个块，否则它将会是空的。所以在这里面定义变量就好了。

基于同样的原因，Pug 的带输出的注释也不能出现在扩展模板的顶层：在最终的 HTML 中，它们输出的 HTML 注释找不到地方放。

嵌入 Interpolation ~~ Interpolation

Pug 提供了若干种操作符以满足您不同的嵌入需求。

字符串嵌入，转义 ~~ String Interpolation, Escaped

观察下面例子中的局部变量 `title` 、 `author` 和 `theGreat` 是如何被嵌入模板的。

```
- var title = "On Dogs: Man's Best Friend";
- var author = "enlore";
- var theGreat = "<span>转义!</span>";

h1= title
p #{author} 笔下源于真情的创作。
p 这是安全的：#{theGreat}
```

`title` 被简单地求值；但在 `#{` 和 `}` 里面的代码也会被求值，转义，并最终嵌入到模板的渲染输出中。

里面可以是任何的正确的 JavaScript 表达式，您可以自由发挥。

```
- var msg = "not my inside voice";
p This is #{msg.toUpperCase()}
```

Pug 足够聪明来分辨到底哪里才是嵌入表达式的结束，所以您不用担心表达式中有 `}` ，也不需要额外的转义。

```
p 不要转义 #{'}'}!
```

如果您需要表示一个 `#{` 文本，您可以转义它，也可以用嵌入功能来生成（可以，这很元编程）。

```
p Escaping works with \#{interpolation}
p Interpolation works with #{'#{interpolation}'} too!
```

字符串嵌入，不转义 ~~ String Interpolation, Unescaped

您当然也 并不是必须 要用安全的转义来构造内容。您可以嵌入没有转义的文本进入模板中。

```
- var riskyBusiness = "<em>我希望通过外籍教师 Peter 找一位英语笔友。</em>";
.quote
  p 李华：!{riskyBusiness}
```

::: float danger 危险 请您务必清醒地意识到，如果直接使用您的用户提供的数据，未进行转义的内容可能会带来安全风险。不要相信用户的输入！ :::

标签嵌入 ~~ Tag Interpolation

嵌入功能不仅可以嵌入 JavaScript 表达式的值，也可以嵌入用 Pug 书写的标签。它看起来应该像这样：

```
p.
  这是一个很长很长而且还很无聊的段落，还没有结束，是的，非常非常地长。
  突然出现了一个 #[strong 充满力量感的单词]，这确实让人难以 #[em 忽视]。
p.
  使用带属性的嵌入标签的例子：
  #[q(lang="es") ¡Hola Mundo!]
```

您确实可以在 Pug 中嵌入一行原始 HTML 代码来做同样的事情。但问题是，如何只用 Pug 来做这件事情？将 Pug 的标签语句用 `#[` 和 `]` 括起来就可以了。它会被求值并嵌入到它原来位置的内容中。

空格的调整 ~~ Whitespace Control

标签嵌入功能，在需要嵌入的位置上前后的空格非常关键的时候，就变得非常有用了。因为 Pug 默认会去除一个标签前后的所有空格。请观察下面一个例子：

```
p
  | 如果我不用嵌入功能来书写，一些标签比如
  strong strong
  | 和
  em em
  | 可能会产生意外的结果。
p.
  如果用了嵌入，在 #[strong strong] 和 #[em em] 旁的空格就会让我舒服多了。
```

阅读文档“纯文本”页中关于空格控制的章节更深入地探讨本话题。

迭代 Iteration ~~ Iteration

Pug 目前支持两种主要的迭代方式： `each` 和 `while` 。

each

这是 Pug 的头等迭代方式，让您在模板中迭代数组和对象更为简便：

```
ul
  each val in [1, 2, 3, 4, 5]
    li= val
```

您还可以在迭代时获得索引值：

```
ul
  each val, index in ['〇', '一', '二']
    li= index + ': ' + val
```

Pug 还让您能够迭代对象中的键值：

```
ul
  each val, index in {1:'一',2:'二',3:'三'}
    li= index + ': ' + val
```

用于迭代的对象或数组仅仅是个 JavaScript 表达式，因此它可以是变量、函数调用的结果，又或者其他的什么东西。

```
- var values = [];
ul
  each val in values.length ? values : ['没有内容']
    li= val
```

您还能添加一个 `else` 块，这个语句块将会在数组与对象没有可供迭代的值时被执行。下面这个例子和上面的例子的作用是等价的：

```
- var values = [];
ul
  each val in values
    li= val
```

```
5.   else
6.     li 没有内容
```

您也可以使用 `for` 作为 `each` 的别称。

while

您也可以使用 `while` 来创建一个循环：

```
1. - var n = 0;
2. ul
3.   while n < 4
4.     li= n++
```

混入 Mixin ~~ Mixins

混入是一种允许您在 Pug 中重复使用一整个代码块的方法。

```
//- 定义
mixin list
  ul
    li foo
    li bar
    li baz
//- 使用
+list
+list
```

它们会被编译成函数形式，您可以传递一些参数：

```
mixin pet(name)
  li.pet= name
ul
  +pet('猫')
  +pet('狗')
  +pet('猪')
```

混入的块 ~~ Mixin Blocks

混入也可以把一整个代码块像内容一样传递进来：

```
mixin article(title)
  .article
    .article-wrapper
      h1= title
      if block
        block
      else
        p 没有提供任何内容。

+article('Hello world')

+article('Hello world')
```

```
13.   p 这是我
14.   p 随便写的文章
```

混入的属性 ~~ Mixin Attributes

混入也可以隐式地，从“标签属性”得到一个参数 `attributes` ：

```
1.  mixin link(href, name)
2.    //- attributes == {class: "btn"}
3.    a(class!=attributes.class href=href)= name
4.
5.  +link('/foo', 'foo')(class="btn")
```

::: float info 提示 `attributes` 里的值已经被（作为标签属性）转义了，所以您可能需要用 `!=` 的方式赋值以避免发生二次转义（详细解释可以查阅不转义的属性）。 :::

您也可以直接用 `&attributes` 方法来传递 `attributes` 参数：

```
1.  mixin link(href, name)
2.    a(href=href)&attributes(attributes)= name
3.
4.  +link('/foo', 'foo')(class="btn")
```

::: float info 提示 `+link(class="btn")` 这种写法也是允许的，且等价于 `+link()(class="btn")` ，因为 Pug 会判断括号内的内容是属性还是参数。但我们鼓励您使用后者的写法，明确地传递空的参数，确保第一对括号内始终都是参数列表。 :::

剩余参数 ~~ Rest Arguments

您可以用剩余参数（rest arguments）语法来表示参数列表最后传入若干个长度不定的参数，比如：

```
1.  mixin list(id, ...items)
2.    ul(id=id)
3.      each item in items
4.        li= item
5.
6.  +list('my-list', 1, 2, 3, 4)
```

纯文本 Plain Text ~~ Plain Text

Pug 提供了四种方法来放置纯文本，换句话说，任何的代码或者文字都将几乎不经过任何处理，直接输出到 HTML 中。它们在不同的情况下会派上不同的用途。纯文本中依然可以使用标签和字符串的嵌入操作，不过每行开头就不再是 Pug 标签。同样，因为纯文本不经处理，因此您也可以在这里面包含原始 HTML 代码。

一个很常见的坑就是控制渲染出的 HTML 中那些空格。我们将会在本页最后讨论这个话题。

标签中的纯文本 ~~ Inline in a Tag

要添加一段行内的纯文本，这是最简单的方法。这行内容的第一项就是标签本身。标签与一个空格后面接着的任何东西，都是这个标签的文本内容。

```
1.  p 这是一段纯洁的<em>文本</em>内容.
```

原始 HTML ~~ Literal HTML

如果一行的开头是左尖括号（ `<` ），那么整行都会被当作纯文本。这在一些书写 HTML 反而更方便的场合下会很有用，一个经典的例子就是条件注释。因为原始 HTML 标签不会被处理，因此它们不像 Pug 的标签，不会自动闭合。

```
1.  <html>
2.
3.  body
4.    p 缩进 body 标签没有意义，
5.    p 因为 HTML 本身对空格不敏感。
6.
7.  </html>
```

管道文本 ~~ Piped Text

另外一种向模板添加纯文本的方法就是在一行前面加一个管道符号（ `|` ），这个字符在类 Unix 系统下常用作“管道”功能，因此得名。该方法在混合纯文本和行内标签时会很有用，我们稍后在空格控制章节中将会深入探讨。

```
1.  p
2.    | 管道符号总是在最开头，
```

```
3.    | 不算前面的缩进。
```

标签中的纯文本块 ~~ Block in a Tag

有的时候您可能想要写一个非常大的纯文本块。一个典型的例子就是用 `script` 和 `style` 内嵌 JavaScript 和 CSS 代码。为此，只需要在标签后面紧接一个点 `.` ，在标签有属性的时候，则是紧接在闭括号后面。在标签和点之间不要有空格。块内的纯文本内容必须缩进一层：

```
1.  script.
2.    if (usingPug)
3.      console.log('这是明智的选择。')
4.    else
5.      console.log('请用 Pug。')
```

您也可以在父块内，创建一个“点”块，跟在某个标签的后面。

```
1.  div
2.    p This text belongs to the paragraph tag.
3.    br
4.    .
5.      This text belongs to the div tag.
```

空格控制 ~~ Whitespace Control

控制输出 HTML 中的空格，可能是学习 Pug 的过程中最为艰难的部分之一，但别担心，您很快就能掌握它。

关于空格，只需记住两个要点。当编译渲染 HTML 的时候：

1. Pug 删掉缩进，以及所有元素间的空格。
    - 因此，一个闭标签会紧挨着下一个开标签。这对于块级元素，比如段落，通常不是个问题，因为它们被浏览器分别作为一个一个段落在页面上渲染（除非您改变了它们的 CSS `display` 属性）。而当您需要在两个元素间插入空格时，请看下面的方法。
2. Pug 保留符合以下条件的元素内的空格：
    - 一行文本之中所有中间的空格；
    - 在块的缩进后的开头的空格；
    - 一行末尾的空格；
    - 纯文本块、或者连续的管道文本行之间的换行。

因此，Pug 会丢掉标签之间的空格，但保留内部的空格。这将有助于完全掌握应该如何操作标签和纯文

本，甚至可以让您在一个单词中间插入一个标签。

```
a ……用一个链接结束的句子
| 。
```

如果您需要添加空格，有几个选择：

推荐方案 ~~ Recommended Solutions

您可以添加一个甚至更多的管道文本行——只有空格或者什么都没有的管道文本。这将会在渲染出来的 HTML 中插入空格。

```
| 千万别
|
button#self-destruct 按
|
| 我！
```

如果您需要插入的标签并不需要太多的属性，那么您或许可以试试在一个纯文本块内使用更简单的标签嵌入或者原始 HTML。

```
p.
  使用常规的标签可以让您的代码行短小精悍，
  但使用嵌入标签会使代码变得更 #[em 清晰易读]。
  ——如果您的标签和文本之间是用空格隔开的。
```

不推荐方案 ~~ Not recommended

您可以在文本的开头（在块的缩进、管道符号、或者标签后）或者末尾，按照您的需求直接添加空格。

注意此处的行尾和行前空格：

```
| 嘻嘻，快看
a(href="http://example.biz/kitteh.png") 这张照片
|  这是我的橘猫
```

上述方案工作正常，但不得不承认有些危险：很多代码编辑器默认都会移除行尾的多余空格。您和其他所有的代码贡献者，可能都必须要配置编辑器，使得不会自动移除行尾空格。

标签 Tag ~~ Tags

在默认情况下，在每行文本的开头（或者紧跟白字符的部分）书写这个 HTML 标签的名称。使用缩进来表示标签间的嵌套关系，这样可以构建一个 HTML 代码的树状结构。

```
ul
  li Item A
  li Item B
  li Item C
```

Pug 还知道哪个元素是自闭合的:

```
img
```

块展开 ~~ Block Expansion

为了节省空间， Pug 为嵌套标签提供了一种内联式语法。

```
a: img
```

自闭合标签 ~~ Self Closing Tags

诸如 `img` , `meta` , 和 `link` 之类的标签都是自动闭合的（除非您使用 XML 类型的 doctype）。 您也可以通过在标签后加上 `/` 来明确声明此标签是自闭合的，但请您仅在明确清楚这是在做什么的情况下使用。

```
foo/
foo(bar='baz')/
```

输出空格 ~~ Rendered Whitespace

标签前后的空格都会被移除，因此您要控制这些标签是否需要处理。空格的控制在纯文本中会详细介绍。