

 TimothyGu Add old Readme_zh-cn.md

a1e6197 on 22 Aug 2016

1 contributor

1286 lines (968 sloc) | 28.4 KB

Pug - 模板引擎

Pug 是一个高性能的模板引擎，它深受 [Hamli](#) 影响，它是用 JavaScript 实现的, 并且可以供 [Node](#) 使用.

翻译: [草依山](#) 等

声明

本文档已经非常过时了，所以很多例子都有可能无法使用。最新的范例可以在[官方英文文档](#)找到。

从 Pug `v0.31.0` 开始放弃了对于 `<script>` 和 `<style>` 标签的平文本支持. 这个问题你可以在 `<script>` `<style>` 标签后加上 `.` 来解决.

希望这一点能让 Pug 对新手更友好, 同时也不影响到 Pug 本身的能力或者导致过度冗长.

如果你有大量的文件需要转换你可以用下 [fix-pug](#) 尝试自动完成这个过程.

Test drive

你可以在网上[试玩 Pug](#).

README 目录

- [特性](#)
- [其它实现](#)
- [安装](#)
- [浏览器支持](#)
- [公开 API](#)
- [语法](#)
 - [行结束标志](#)
 - [标签](#)
 - [标签文本](#)
 - [注释](#)
 - [块注释](#)
 - [内联](#)
 - [块展开](#)
 - [Case](#)
 - [属性](#)
 - [HTML](#)
 - [Doctypes](#)
- [过滤器](#)
- [代码](#)
- [循环](#)
- [条件语句](#)
- [模板继承](#)
- [Block append/prepend](#)

- [包含](#)
- [Mixins](#)
- [产生输出](#)
- [Makefile 的一个例子](#)
- [命令行的 Pug](#)
- [教程](#)
- [License](#)

特性

- 客户端支持
- 代码高可读
- 灵活的缩进
- 块展开
- Mixins
- 静态包含
- 属性改写
- 安全，默认代码是转义的
- 运行时和编译时上下文错误报告
- 命令行下编译pug模板
- HTML5 模式 (使用 `!!!>` `doctype html` 文档类型)
- 在内存中缓存(可选)
- 合并动态和静态标签类
- 可以通过 `filters` 修改树
- 模板继承
- 原生支持 [Express JS](#)
- 通过 `each` 枚举对象、数组甚至是不能枚举的对象
- 块注释
- 没有前缀的标签
- AST Filters
- 过滤器
 - `:stylus` 必须已经安装 [stylus](#)
 - `:less` 必须已经安装 [less.js](#)
 - `:markdown` 必须已经安装 [markdown-js](#) 或者 [node-discount](#)
 - `:cdata`
 - `:coffeescript` 必须已经安装[coffee-script](#)
- [Emacs Mode](#)
- [Vim Syntax](#)
- [TextMate Bundle](#)
- [Coda/SubEtha syntax Mode](#)
- [Screencasts](#)
- [html2pug converter](#)

其它实现

- [php](#)
- [scala](#)
- [ruby](#)
- [python](#)
- [java](#)

安装

通过 NPM:

```
npm install pug
```

浏览器支持

把 Pug 编译为一个可供浏览器使用的单文件，只需要简单的执行：

```
$ make pug.js
```

如果你已经安装了 uglifyjs (`npm install uglify-js`), 你可以执行下面的命令它会生成所有的文件。其实每一个正式版本里都帮你做了这事。

```
make pug.min.js
```

默认情况下，为了方便调试Pug会把模板组织成带有形如 `___.lineno = 3` 的行号的形式。在浏览器里使用的时候，你可以通过传递一个选项 `{ compileDebug: false }` 来去掉这个。下面的模板

```
p Hello #{name}
```

会被翻译成下面的函数：

```
function anonymous(locals, attrs, escape, rethrow) {
  var buf = [];
  with (locals || {}) {
    var interp;
    buf.push('\n<p>Hello ' + escape((interp = name) == null ? '' : interp) + '\n</p>');
  }
  return buf.join("");
}
```

通过使用 Pug 的 `./runtime.js` 你可以在浏览器使用这些预编译的模板而不需要使用 Pug, 你只需要使用 `runtime.js` 里的工具函数, 它们会放在 `pug.attrs`, `pug.escape` 这些里。把选项 `{ client: true }` 传递给 `pug.compile()`, Pug 会把这些帮助函数的引用放在 `pug.attrs`, `pug.escape` .

```
function anonymous(locals, attrs, escape, rethrow) {
  var attrs = pug.attrs, escape = pug.escape, rethrow = pug.rethrow;
  var buf = [];
  with (locals || {}) {
    var interp;
    buf.push('\n<p>Hello ' + escape((interp = name) == null ? '' : interp) + '\n</p>');
  }
  return buf.join("");
}
```

公开 API

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

// Compile a function
var fn = pug.compile('string of pug', options);
fn(locals);
```

选项

- `self` 使用 `self` 命名空间来持有本地变量. (默认为 `false`)
- `locals` 本地变量对象
- `filename` 异常发生时使用, `includes` 时必需
- `debug` 输出 token 和翻译后的函数体
- `compiler` 替换掉 pug 默认的编译器
- `compileDebug` `false` 的时候调试的结构不会被输出
- `pretty` 为输出加上了漂亮的空格缩进 (默认为 `false`)

语法 ### 行结束标志

CRLF 和 **CR** 会在编译之前被转换为 **LF**

标签

标签就是一个简单的单词:

```
html
```

它会被转换为 `<html></html>`

标签也是可以有 id 的:

```
div#container
```

它会被转换为 `<div id="container"></div>`

怎么加 class 呢?

```
div.user-details
```

转换为 `<div class="user-details"></div>`

多个 class 和 id? 也是可以搞定的:

```
div#foo.bar.baz
```

转换为 `<div id="foo" class="bar baz"></div>`

不停的 `div div div` 很讨厌啊, 可以这样:

```
#foo  
.bar
```

这个算是我们的语法糖, 它已经被很好的支持了, 上面的会输出:

```
<div id="foo"></div><div class="bar"></div>
```

标签文本

只需要简单的把内容放在标签之后:

```
p wahoo!
```

它会被渲染为 `<p>wahoo!</p>` .

很帅吧, 但是大段的文本怎么办呢:

```
p  
| foo bar baz  
| rawr rawr  
| super cool  
| go pug go
```

渲染为 `<p>foo bar baz rawr.....</p>`

怎么和数据结合起来? 所有类型的文本展示都可以和数据结合起来, 如果我们把 `{ name: 'tj', email: 'tj@vision-media.ca' }` 传给编译函数, 下面是模板上的写法:

```
#user #{name} &lt;#{email}&gt;
```

它会被渲染为 `<div id="user">tj <tj@vision-media.ca></div>`

当就是要输出 `#{}` 的时候怎么办? 转义一下!

```
p \#{something}
```

它会输出 `<p>\#{something}</p>`

同样可以使用非转义的变量 `!{html}`，下面的模板将直接输出一个 `<script>` 标签：

```
- var html = "<script></script>"
| !{html}
```

内联标签同样可以使用文本块来包含文本：

```
label
| Username:
input(name='user[name]')
```

或者直接使用标签文本：

```
label Username:
input(name='user[name]')
```

只包含文本的标签，比如 `<script>`，`<style>`，和 `<textarea>` 不需要前缀 `|` 字符，比如：

```
html
head
  title Example
  script
    if (foo) {
      bar();
    } else {
      baz();
    }
}
```

这里还有一种选择，可以使用 `.` 来开始一段文本块，比如：

```
p.
  foo asdf
  asdf
  asdfasdfaf
  asdf
  asd.
```

会被渲染为：

```
<p>foo asdf
asdf
  asdfasdfaf
  asdf
asd
.
</p>
```

这和带一个空格的 `p .` 是不一样的，带空格的会被 Pug 的解析器忽略，当作一个普通的文字：

```
p .
```

渲染为：

```
<p>.</p>
```

需要注意的是文本块需要两次转义。比如想要输出下面的文本：

```
<p>foo\bar</p>
```

使用：

```
p.  
  foo\\bar
```

注释

单行注释和 JavaScript 里是一样的，通过 `//` 来开始，并且必须单独一行：

```
// just some paragraphs  
p foo  
p bar
```

渲染为：

```
<!-- just some paragraphs -->  
<p>foo</p>  
<p>bar</p>
```

Pug 同样支持不输出的注释，加一个短横线就行了：

```
//- will not output within markup  
p foo  
p bar
```

渲染为：

```
<p>foo</p>  
<p>bar</p>
```

块注释

块注释也是支持的：

```
body  
  //  
  注释
```

渲染为：

```
<body>  
  <!--注释-->  
</body>
```

条件注释

对于条件注释，Pug 没有任何特殊的语法。因为 Pug 会把所有以 `<` 开始的代码处理为纯文本，您可以直接使用 HTML：

```
body  
  <!--[if IE]>  
  a(href='http://www.mozilla.com/en-US/firefox/') Get Firefox  
  <![endif]-->
```

渲染为：

```
<body>  
  <!--[if IE]>  
  <a href="http://www.mozilla.com/en-US/firefox/">Get Firefox</a>  
  <![endif]-->  
</body>
```

内联

Pug 支持以自然的方式定义标签嵌套：

```
ul
  li.first
    a(href='#') foo
  li
    a(href='#') bar
  li.last
    a(href='#') baz
```

块展开

块展开可以帮助你在一行内创建嵌套的标签，下面的例子和上面的是一样的：

```
ul
  li.first: a(href='#') foo
  li: a(href='#') bar
  li.last: a(href='#') baz
```

Case

`case` 表达式按下面这样的形式写：

```
html
  body
    friends = 10
    case friends
      when 0
        p you have no friends
      when 1
        p you have a friend
      default
        p you have #{friends} friends
```

块展开在这里也可以使用：

```
friends = 5

html
  body
    case friends
      when 0: p you have no friends
      when 1: p you have a friend
      default: p you have #{friends} friends
```

属性

Pug 现在支持使用 `(` 和 `)` 作为属性分隔符

```
a(href='/login', title='View login page') Login
```

当一个值是 `undefined` 或者 `null` 属性 不会被加上，所以呢，它不会编译出 `'something="null"'`。

```
div(something=null)
```

Boolean 属性也是支持的：

```
input(type="checkbox", checked)
```

使用代码的 Boolean 属性只有当属性为 `true` 时才会输出：

```
input(type="checkbox", checked=someValue)
```

多行同样也是可用的：

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

多行的时候可以不加逗号：

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

加点空格，格式好看一点？同样支持

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

冒号也是支持的：

```
rss(xmlns:atom="atom")
```

假如我有一个 `user` 对象 `{ id: 12, name: 'tobi' }` 我们希望创建一个指向 `/user/12` 的链接 `href`，我们可以使用普通的 JavaScript 字符串连接，如下：

```
a(href='/user/' + user.id)= user.name
```

或者我们使用 Pug 的修改方式，这个我想很多使用 Ruby 或者 CoffeeScript 的人会看起来像普通的 JS..：

```
a(href='/user/#{user.id}')= user.name
```

`class` 属性是一个特殊的属性，你可以直接传递一个数组，比如 `bodyClasses = ['user', 'authenticated']`：

```
body(class=bodyClasses)
```

HTML

内联的 HTML 是可以的，我们可以使用管道定义一段文本：

```
html
  body
    | <h1>Title</h1>
    | <p>foo bar baz</p>
```

或者我们可以使用 `.` 来告诉 Pug 我们需要一段文本：

```
html
  body.
    <h1>Title</h1>
    <p>foo bar baz</p>
```

上面的两个例子都会渲染成相同的结果：

```
<html><body><h1>Title</h1>
<p>foo bar baz</p>
</body></html>
```

这条规则适应于在 Pug 里的任何文本：

```
html
  body
    h1 User <em>#{name}</em>
```

Doctypes

添加文档类型只需要简单的使用 `!!!`，或者 `doctype` 跟上下面的可选项：

```
!!!
```

会渲染出 *transitional* 文档类型, 或者:

```
!!! 5
```

或

```
!!! html
```

或

```
doctype html
```

Doctype 是大小写不敏感的, 所以下面两个是一样的:

```
doctype Basic
doctype basic
```

当然也是可以直接传递一段文档类型的文本:

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

渲染后:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN">
```

会输出 *HTML5* 文档类型. 下面的默认文档类型, 可以很简单的扩展:

```
var doctypes = exports.doctypes = {
  '5': '<!DOCTYPE html>',
  'xml': '<?xml version="1.0" encoding="utf-8" ?>',
  'default': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">',
  'transitional': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">',
  'strict': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">',
  'frameset': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">',
  '1.1': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">',
  'basic': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN" "http://www.w3.org/TR/xhtml1-basic/xhtml1-basic.dtd">',
  'mobile': '<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.2//EN" "http://www.openmobilealliance.org/DTD/xhtml1-basic/xhtml1-basic.dtd">'
};
```

通过下面的代码可以很简单的改变默认文档类型:

```
pug.doctypes.default = 'whatever you want';
```

过滤器

过滤器前缀 `:`, 比如 `:markdown` 会把下面块里的文本交给专门的函数进行处理。查看顶部 *特性* 里有哪些可用的过滤器。

```
body
:markdown
  Woah! pug _and_ markdown, very cool
  we can even link to [stuff](http://google.com)
```

渲染为:

```
<body><p>Woah! pug and markdown, very cool we can even link to \[stuff\]</p></body>
```

代码

Pug 目前支持三种类型的可执行代码。第一种是前缀 `-`，这是不会被输出的：

```
- var foo = 'bar';
```

这可以用在条件语句或者循环中：

```
- for (var key in obj)
  p= obj[key]
```

由于 Pug 的缓存技术，下面的代码也是可以的：

```
- if (foo)
  ul
    li yay
    li foo
    li worked
- else
  p oh no! didnt work
```

哈哈，甚至是很长的循环也是可以的：

```
- if (items.length)
  ul
    - items.forEach(function(item){
      li= item
    - })
```

所以你想要的！

下一步我们要 转义 输出的代码，比如我们返回一个值，只要前缀一个 `=`：

```
- var foo = 'bar'
= foo
h1= foo
```

它会渲染为 `bar<h1>bar</h1>`。为了安全起见，使用 `=` 输出的代码默认是转义的，如果想直接输出不转义的值可以使用 `!=`：

```
p!= aVarContainingMoreHTML
```

Pug 同样是设计师友好的，它可以使 JavaScript 更直接更富表现力。比如下面的赋值语句是相等的，同时表达式还是通常的 JavaScript：

```
- var foo = 'foo ' + 'bar'
foo= 'foo ' + 'bar'
```

Pug 会把 `if`，`else if`，`else`，`until`，`while`，`unless` 同别的优先对待，但是你得记住它们还是普通的 JavaScript：

```
if foo == 'bar'
  ul
    li yay
    li foo
    li worked
else
  p oh no! didnt work
```

循环

尽管已经支持 JavaScript 原生代码，Pug 还是支持了一些特殊的标签，它们可以让模板更加易于理解，其中之一就是 `each`，这种形式：

```
each VAL[, KEY] in OBJ
```

一个遍历数组的例子：

```
- var items = ["one", "two", "three"]
each item in items
  li= item
```

渲染为:

```
<li>one</li>
<li>two</li>
<li>three</li>
```

遍历一个数组同时带上索引:

```
items = ["one", "two", "three"]
each item, i in items
  li #{item}: #{i}
```

渲染为:

```
<li>one: 0</li>
<li>two: 1</li>
<li>three: 2</li>
```

遍历一个对象的键值:

```
obj = { foo: 'bar' }
each val, key in obj
  li #{key}: #{val}
```

将会渲染为: `<li>foo: bar</li>`

Pug 在内部会把这些语句转换成原生的 JavaScript 语句, 就像使用 `users.forEach(function(user){`, 词法作用域和嵌套会像在普通的 JavaScript 中一样:

```
each user in users
  each role in user.roles
    li= role
```

如果你喜欢, 也可以使用 `for` :

```
for user in users
  for role in user.roles
    li= role
```

条件语句

Pug 条件语句和使用了 `(-)` 前缀的 JavaScript 语句是一致的, 然后它允许你不使用圆括号, 这样会看上去对设计师更友好一点, 同时要在心里记住这个表达式渲染出的是 常规 JavaScript:

```
for user in users
  if user.role == 'admin'
    p #{user.name} is an admin
  else
    p= user.name
```

和下面的使用了常规 JavaScript 的代码是相等的:

```
for user in users
  - if (user.role == 'admin')
    p #{user.name} is an admin
  - else
    p= user.name
```

Pug 同时支持 `unless`, 这和 `if (!(expr))` 是等价的:

```
for user in users
  unless user.isAnonymous
    p
      | Click to view
      a(href='/users/' + user.id)= user.name
```

模板继承

Pug 支持通过 `block` 和 `extends` 关键字来实现模板继承。一个块就是一个 Pug 的 `block`，它将在子模板中实现，同时是支持递归的。

Pug 块如果没有内容，Pug 会添加默认内容，下面的代码默认会输出 `block scripts`，`block content`，和 `block foot`。

```
html
  head
    h1 My Site - #{title}
    block scripts
      script(src='/jquery.js')
  body
    block content
    block foot
    #footer
      p some footer content
```

现在我们来继承这个布局，简单创建一个新文件，像下面那样直接使用 `extends`，给定路径（可以选择带 `.pug` 扩展名或者不带）。你可以定义一个或者更多的块来覆盖父级块内容，注意到这里的 `foot` 块没有定义，所以它还会输出父级的 "some footer content"。

```
extends extend-layout

block scripts
  script(src='/jquery.js')
  script(src='/pets.js')

block content
  h1= title
  each pet in pets
    include pet
```

同样可以在一个子块里添加块，就像下面实现的块 `content` 里又定义了两个可以被实现的块 `sidebar` 和 `primary`，或者子模板直接实现 `content`。

```
extends regular-layout

block content
  .sidebar
    block sidebar
      p nothing
  .primary
    block primary
      p nothing
```

前置、追加代码块

Pug 允许你 替换（默认）、前置和追加 blocks。比如，假设你希望在 所有 页面的头部都加上默认脚本，你可以这么做：

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

现在假设你有一个 Javascript 游戏的页面，你希望在默认脚本之外添加一些游戏相关的脚本，你可以直接 `append` 上代码块：

```
extends layout

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

使用 `block append` 或 `block prepend` 时 `block` 是可选的:

```
extends layout

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

包含

Includes 允许你静态包含一段 Pug, 或者别的存放在单个文件中的东西比如 CSS, HTML 非常常见的例子是包含头部和页脚。假设我们有一个下面目录结构的文件夹:

```
./layout.pug
./includes/
  ./head.pug
  ./tail.pug
```

下面是 `layout.pug` 的内容:

```
html
  include includes/head
  body
    h1 My Site
    p Welcome to my super amazing site.
  include includes/foot
```

这两个包含 `includes/head` 和 `includes/foot` 都会读取相对于给 `layout.pug` 参数 `filename` 的路径的文件, 这是一个绝对路径, 不用担心 Express 帮你搞定这些了。Include 会解析这些文件, 并且插入到已经生成的语法树中, 然后渲染为你期待的内容:

```
<html>
  <head>
    <title>My Site</title>
    <script src="/javascripts/jquery.js">
    </script><script src="/javascripts/app.js"></script>
  </head>
  <body>
    <h1>My Site</h1>
    <p>Welcome to my super lame site.</p>
    <div id="footer">
      <p>Copyright>(c) foobar</p>
    </div>
  </body>
</html>
```

前面已经提到, `include` 可以包含比如 HTML 或者 CSS 这样的内容。给定一个扩展名后, Pug 不会把这个文件当作一个 Pug 源代码, 并且会把它当作一个普通文本包含进来:

```
html
  head
    //- css and js have simple filters that wrap them in
    <style> and <script> tags, respectively
    include stylesheet.css
    include script.js
  body
    //- "markdown" files will use the "markdown" filter
    to convert Markdown to HTML
    include introduction.markdown
    //- html files have no filter and are included verbatim
    include content.html
```

Include 也可以接受块内容，给定的块将会附加到包含文件 最后 的块里。举个例子，`head.pug` 包含下面的内容：

```
head
  script(src='/jquery.js')
```

我们可以像下面给 `include head` 添加内容, 这里是添加两个脚本.

```
html
  include head
    script(src='/foo.js')
    script(src='/bar.js')
  body
    h1 test
```

在被包含的模板中，你也可以使用 `yield` 语句。`yield` 语句允许你明确地标明 `include` 的代码块的放置位置。比如，假设你希望把代码块放在 `scripts` 之前，而不是附加在 `scripts` 之后：

```
head
  yield
  script(src='/jquery.js')
  script(src='/jquery.ui.js')
```

由于被包含的Pug会按字面解析并合并到AST中，词法范围的变量的效果和直接写在同一个文件中的相同。这就意味着 `include` 可以用作 `partial` 的替代，例如，假设我们有一个引用了 `user` 变量的 `user.pug` 文件：

```
h1= user.name
p= user.occupation
```

接着，当我们迭代 `users` 的时候，只需简单地加上 `include user`。因为在循环中 `user` 变量已经被定义了，被包含的模板可以访问它。

```
users = [{ name: 'Tobi', occupation: 'Ferret' }]

each user in users
  .user
    include user
```

以上代码会生成：

```
<div class="user">
  <h1>Tobi</h1>
  <p>Ferret</p>
</div>
```

`user.pug` 引用了 `user` 变量，如果我们希望使用一个不同的变量 `user`，那么我们可以直接定义一个新变量 `user = person`，如下所示：

```
each person in users
  .user
    user = person
    include user
```

Mixins

Mixins 在编译的模板里会被 Pug 转换为普通的 JavaScript 函数。Mixins 可以还参数，但不是必需的：

```
mixin list
  ul
    li foo
    li bar
    li baz
```

使用不带参数的 `mixin` 看上去非常简单，在一个块外：

```
h2 Groceries
mixin list
```

Mixins 也可以带一个或者多个参数，参数就是普通的 JavaScript 表达式，比如下面的例子：

```
mixin pets(pets)
  ul.pets
    - each pet in pets
      li= pet

mixin profile(user)
  .user
    h2= user.name
    mixin pets(user.pets)
```

会输出像下面的 HTML：

```
<div class="user">
  <h2>tj</h2>
  <ul class="pets">
    <li>tobi</li>
    <li>loki</li>
    <li>jane</li>
    <li>manny</li>
  </ul>
</div>
```

产生输出

假设我们有下面的 Pug 源码：

```
- var title = 'yay'
h1.title #{title}
p Just an example
```

当 `compileDebug` 选项不是 `false`，Pug 会编译时会把函数里加上 `___.lineno = n;`，这个参数会在编译出错时传递给 `rethrow()`，而这个函数会在 Pug 初始输出时给出一个有用的错误信息。

```
function anonymous(locals) {
  var ___ = { lineno: 1, input: "- var title = 'yay'\nh1.title #{title}\np Just an example", filename: "test.pug" };
  var rethrow = pug.rethrow;
  try {
    var attrs = pug.attrs, escape = pug.escape;
    var buf = [];
    with (locals || {}) {
      var interp;
      ___.lineno = 1;
      var title = 'yay'
      ___.lineno = 2;
      buf.push('<h1');
      buf.push(attrs({ "class": ('title') }));
      buf.push('>');
      buf.push('' + escape((interp = title) == null ? '' : interp) + '');
      buf.push('</h1>');
      ___.lineno = 3;
      buf.push('<p>');
      buf.push('Just an example');
      buf.push('</p>');
    }
    return buf.join('');
  } catch (err) {
    rethrow(err, ___.input, ___.filename, ___.lineno);
  }
}
```

当 `compileDebug` 参数是 `false`，这个参数会被去掉，这样对于轻量级的浏览器端模板是非常有用的。结合 Pug 的参数和当前源码库里的 `./runtime.js` 文件，你可以通过 `toString()` 来编译模板而不需要在浏览器端运行整个 Pug 库，这样可以提高性能，也可以减少载入的 JavaScript 数量。

```
function anonymous(locals) {
  var attrs = pug.attrs, escape = pug.escape;
  var buf = [];
  with (locals || {}) {
    var interp;
    var title = 'yay'
    buf.push('<h1');
    buf.push(attrs({ "class": ('title') }));
    buf.push('>');
    buf.push('' + escape((interp = title) == null ? '' : interp) + '');
    buf.push('</h1>');
    buf.push('<p>');
    buf.push('Just an example');
    buf.push('</p>');
  }
  return buf.join('');
}
```

Makefile 的一个例子

通过执行 `make`，下面的 Makefile 例子可以把 `pages/*.pug` 编译为 `pages/*.html`。

```
PUG = $(shell find pages/*.pug)
HTML = $(PUG:.pug=.html)

all: $(HTML)

%.html: %.pug
    pug < $< --path $< > $@

clean:
    rm -f $(HTML)

.PHONY: clean
```

这个可以和 `watch(1)` 命令起来产生像下面的行为：

```
$ watch make
```

命令行的 Pug

使用: `pug [options] [dir|file ...]`

选项:

<code>-h, --help</code>	输出帮助信息
<code>-v, --version</code>	输出版本号
<code>-o, --out <dir></code>	输出编译后的 HTML 到 <dir>
<code>-O, --obj <str></code>	JavaScript 选项
<code>-p, --path <path></code>	在处理 stdio 时, 查找包含文件时的查找路径
<code>-P, --pretty</code>	格式化 HTML 输出
<code>-c, --client</code>	编译浏览器端可用的 runtime.js
<code>-D, --no-debug</code>	关闭编译的调试选项(函数会更小)
<code>-w, --watch</code>	监视文件改变自动刷新编译结果

Examples:

```
# 编译整个目录
$ pug templates

# 生成 {foo,bar}.html
$ pug {foo,bar}.pug

# 在标准IO下使用pug
$ pug < my.pug > my.html

# 在标准IO下使用pug, 同时指定用于查找包含的文件
$ pug < my.pug -p my.pug > my.html

# 在标准IO下使用pug
$ echo "h1 Pug!" | pug
```

```
# foo, bar 目录渲染到 /tmp
$ pug foo bar --out /tmp
```

注意: 从 `v0.31.0` 的 `-o` 选项已经指向 `--out`, `-o` 相应做了交换

教程

- cssdeck interactive [Pug syntax tutorial](#)
- cssdeck interactive [Pug logic tutorial](#)
- in [Japanese](#)

License

(The MIT License)

Copyright (c) 2009-2010 TJ Holowaychuk <tj@vision-media.ca>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.