一 选择器

1. 缩排

Stylus蛮“玄幻”的(如基于缩进)，空格有重要的意义，所以，我们使用缩排和凹排代替花括号{以及}

|  |
| --- |
| body  color white |
| 对应  body {  color: #fff;  } |
| 如果你喜欢，你可以把冒号加上，用做分隔，便于阅读：  body  color: white |

2. 规则集

Stylus就跟CSS一样，允许你使用逗号为多个选择器同时定义属性。

|  |
| --- |
| textarea, input  border 1px solid #eee |
| 换新行是一样的效果  textarea  input  border 1px solid #eee |
| 等同于  textarea,  input {  border: 1px solid #eee;  } |
| 该规则唯一的例外就是长得像属性的选择器。例如，下面的foo bar baz可能是个属性或者是选择器。  foo bar baz  > input  border 1px solid  为解决这个原因，我们可以在尾部加个逗号：  foo bar baz,  form input,  > a  border 1px solid |

3. 父级引用

字符&指向父选择器。下面这个例子，我们两个选择器(textarea和input)在:hover伪类选择器上都改变了color值

|  |
| --- |
| textarea  input  color #A7A7A7  &:hover  color #000 |
| 等同于  textarea,  input {  color: #a7a7a7;  }  textarea:hover,  input:hover {  color: #000;  } |
| 理解:父级指上一缩排距离 |

4. 消除歧义

类似padding - n的表达式可能既被解释成减法运算，也可能被释义成一元负号属性。为了避免这种歧义，用括号包裹表达式：

|  |
| --- |
| pad(n)  padding (- n)  body  pad(5px) |
| 等同于  body {  padding: -5px;  }  理解:不想进行运算。就使用括号进行包裹 |

然而，只有在函数中才会这样（因为函数同时用返回值扮演混合或回调）。

例如，下面这个就是OK的（产生与上面相同的结果）：

|  |
| --- |
| body  padding -5px |

有Stylus无法处理的属性值？unquote()可以帮你：

|  |
| --- |
| filter  unquote('progid:DXImageTransform.Microsoft.BasicImage(rotation=1)') |
| 生成为：  filter progid:DXImageTransform.Microsoft.BasicImage(rotation=1) |

二 变量

1. 变量

可以指定表达式为变量，然后在我们的样式中贯穿使用：

|  |
| --- |
| font-size = 14px  body  font : font-size Arial, sans-seri |
| body {  font: 14px Arial, sans-serif;  } |

变量甚至可以组成一个表达式列表：

|  |
| --- |
| font-size = 14px  font = font-size "Lucida Grande", Arial  body  font : font sans-serif |
| body {  font: 14px "Lucida Grande", Arial sans-serif;  } |

标识符（变量名，函数等），也可能包括$字符。例如：

|  |
| --- |
| $font-size = 14px  body {  font: $font-size sans-serif;  } |

2. 属性查找

Stylus有另外一个很酷的独特功能，不需要分配值给变量就可以定义引用属性。下面是个很好的例子，元素水平垂直居中对齐（典型的方法是使用百分比和margin负值），如下：

|  |
| --- |
| #logo  position: absolute  top: 50%  left: 50%  width: w = 150px  height: h = 80px  margin-left: -(w / 2)  margin-top: -(h / 2) |
| 我们不使用这里的变量w和h, 而是简单地前置@字符在属性名前来访问该属性名对应的值： |
| #logo  position: absolute  top: 50%  left: 50%  width: 150px  height: 80px  margin-left: -(@width / 2)  margin-top: -(@height / 2)  疑问：是否需要定高定宽？ |

另外使用案例是基于其他属性有条件地定义属性。在下面这个例子中，我们默认指定z-index值为1，但是，只有在z-index之前未指定的时候才这样：

|  |
| --- |
| position()  position: arguments  z-index: 1 unless @z-index  #logo  z-index: 20  position: absolute  #logo2  position: absolute |

属性会“向上冒泡”查找堆栈直到被发现，或者返回null（如果属性搞不定）。下面这个例子，@color被弄成了blue.

|  |
| --- |
| body  color: red  ul  li  color: blue  a  background-color: @color  // 因为a本身未定义color属性 |

三 插值

1. 插值

Stylus支持通过使用{}字符包围表达式来插入值，其会变成标识符的一部分。例如，-webkit-{'border' + '-radius'}等同于-webkit-border-radius.

比较好的例子就是私有前缀属性扩展：

|  |
| --- |
| vendor(prop, args)  -webkit-{prop} args  -moz-{prop} args  {prop} args  border-radius()  vendor('border-radius', arguments)  box-shadow()  vendor('box-shadow', arguments)  button  border-radius 1px 2px / 3px 4px |
| 等同于  button {  -webkit-border-radius: 1px 2px / 3px 4px;  -moz-border-radius: 1px 2px / 3px 4px;  border-radius: 1px 2px / 3px 4px;  } |

2. 选择器插值

插值也可以在选择器上起作用。例如，我们可以指定表格前5行的高度，如下：

|  |
| --- |
| table  for row in 1 2 3 4 5  tr:nth-child({row})  height: 10px \* row |
| 等同于:  table tr:nth-child(1) {  height: 10px;  }  table tr:nth-child(2) {  height: 20px;  }  table tr:nth-child(3) {  height: 30px;  }  table tr:nth-child(4) {  height: 40px;  }  table tr:nth-child(5) {  height: 50px;  } |

四 运算符

1. 运算优先级

|  |
| --- |
| []  ! ~ + -  is defined  \*\* \* / %  + -  ... ..  <= >= < >  in  == is != is not isnt  is a  && and || or  ?:  = := ?= += -= \*= /= %=  not  if unless |

2. 一元运算符

以下一元运算符可用，!, not, -, +, 以及~.

|  |
| --- |
| !0  // => true  !!0  // => false  !1  // => false  !!5px  // => true  -5px  // => -5px  --5px  // => 5px  not true  // => false  not not true  // => true |

逻辑运算符not的优先级较低，因此，下面这个例子可以替换：

|  |
| --- |
| a = 0  b = 1  !a and !b  // => false  // 解析为: (!a) and (!b)  // !a => true !b=>false; true and false =>false |
| 用:  not a or b  // => false  // 解析为: not (a or b)  // a or b =>true !true => false |

3. 二元运算符

下标运算符[]允许我们通过索引获取表达式内部值。括号表达式可以充当元组（如(15px 5px), (1, 2, 3)）。

下面这个例子使用错误处理的元组（并展示了该结构的多功能性）：

|  |
| --- |
| add(a, b)  if a is a 'unit' and b is a 'unit' //必须是计量单位  a + b  else  (error 'a 和 b 必须是 units!')  body  padding add(1,'5')  // => padding: error "a 和 b 必须是 units";  padding add(1,'5')[0]  // => padding: error;  padding add(1,'5')[0] == error  // => padding: true;  padding add(1,'5')[1]  // => padding: "a 和 b 必须是 units"; |
| 这儿有个更复杂的例子。现在，我们调用内置的error()函数，当标识符（第一个值）等于error的时候返回错误信息。  if (val = add(1,'5'))[0] == error  error(val[1]) |

4. 范围

同时提供包含界线操作符(..)和范围操作符(...)，见下表达式：

|  |
| --- |
| 1..5  // => 1 2 3 4 5  1...5  // => 1 2 3 4 |

5. 加减 + -

二元加乘运算其单位会转化，或使用默认字面量值。例如，5s - 2px结果是3s.

|  |
| --- |
| 15px - 5px  // => 10px  5 - 2  // => 3  5in - 50mm  // => 3.031in  5s - 1000ms  // => 4s  20mm + 4in  // => 121.6mm  "foo " + "bar"  // => "foo bar"  "num " + 15  // => "num 15" |

6. 乘除 / \* %

|  |
| --- |
| 2000ms + (1s \* 2)  // => 4000ms  5s / 2  // => 2.5s  4 % 2  // => 0 |

当在属性值内使用/时候，你必须用括号包住。否则/会根据其字面意思处理（支持CSS的line-height）。

|  |
| --- |
| font: 14px/1.5;  但是，下面这个却等同于14px ÷ 1.5:  font: (14px/1.5);  只有/操作符的时候需要这样。 |

7. 指数

|  |
| --- |
| 2 \*\* 8  // => 256  //2 的8次方 |

8. 相等与关系运算符

相等运算符可以被用来等同单位、颜色、字符串甚至标识符。这是个强大的概念，甚至任意的标识符（例如wahoo）可以作为原子般使用。函数可以返回yes和no代替true和false（虽然不建议）。

|  |
| --- |
| 5 == 5  // => true  10 > 5  // => true  #fff == #fff  // => true  true == false  // => false  wahoo == yay  // => false  wahoo == wahoo  // => true  "test" == "test"  // => true  true is true  // => true  'hey' is not 'bye'  // => true  'hey' isnt 'bye'  // => true  (foo bar) == (foo bar)  // => true  (1 2 3) == (1 2 3)  // => true  (1 2 3) == (1 1 3)  // => false |

只有精确值才匹配，例如，0 == false和null == false均返回false.

别名：

|  |
| --- |
| == is  != is not  != isnt |

9. 真与假

Stylus近乎一切都是true, 包括有后缀的单位，甚至0%, 0px等都被认作true.

不过，0在算术上本身是false.

表达式（或“列表”）长度大于1被认为是真。

|  |
| --- |
| true：  0%  0px  1px  -1  -1px  hey  'hey'  (0 0 0)  ('' '') |
| false  0  null  false  '' |

10. 逻辑操作符

逻辑操作符&&和||别名是and / or。它们优先级相同。

|  |
| --- |
| 5 && 3  // => 3  0 || 5  // => 5  0 && 5  // => 0  #fff is a 'rgba' and 15 is a 'unit'  // => true |

1. 存在操作符 in

检查左边内容是否在右边的表达式中。

简单的例子：

|  |
| --- |
| nums = 1 2 3  1 in nums  // => true  5 in nums  // => false |

一些未定义的标识符

|  |
| --- |
| words = foo bar baz  bar in words  // => true  HEY in words  // => false |

元祖同样适用

|  |
| --- |
| vals = (error 'one') (error 'two')  error in vals  // => false  (error 'one') in vals  // => true  (error 'two') in vals  // => true  (error 'something') in vals  // => false |

混合书写适用例子

|  |
| --- |
| pad(types = padding, n = 5px)  if padding in types  padding n  if margin in types  margin n  body  pad()  body  pad(margin)  body  pad(padding margin, 10px)  对应于：  body {  padding: 5px;  }  body {  margin: 5px;  }  body {  padding: 10px;  margin: 10px;  } |

12. 条件赋值 ?= :=

条件赋值操作符?=（别名?:）让我们无需破坏旧值（如果存在）定义变量。该操作符可以扩展成三元内is defined的二元操作。

例如，下面这些都是平起平坐的：

|  |
| --- |
| color := white  color ?= white  color = color is defined ? color : white  //如果color属性未赋值，就进行赋值 |

如果我们使用等号=, 就只是简单地赋值。

|  |
| --- |
| color = white  color = black  color  // => black |

但当使用?=，第二个相当就嗝屁了（因为变量已经定义了）：

|  |
| --- |
| color = white  color ?= black  color  // => white |

1. 实例检查 is a

Stylus提供一个二元运算符叫做is a, 用做类型检查。

|  |
| --- |
| 15 is a 'unit'  // => true  #fff is a 'rgba'  // => true  15 is a 'rgba'  // => false |
| 疑问；类型有多少？ |
| 另外，我们可以使用type()这个内置函数。  type(#fff) == 'rgba'  // => true  注意：color是唯一的特殊情况，当左边是RGBA或者HSLA节点时，都为true. |

14. 变量定义

此伪二元运算符右边空荡荡，左边无计算。用来检查变量是否已经分配了值。

|  |
| --- |
| foo is defined  // => false  foo = 15px  foo is defined  // => true  #fff is defined  // => 'invalid "is defined" check on non-variable #fff' |
| 另外，我们可以使用内置lookup(name)方法做这个活动态查找。  name = 'blue'  lookup('light-' + name)  // => null  light-blue = #80e2e9  lookup('light-' + name)  // => #80e2e9 |
| 该操作符必不可少，因为一个未定义的标识符仍是真值。如：  body  if ohnoes  padding 5px  当未定义的时候，产生的是下面的CSS：  body {  padding: 5px;  }  显然，这不是我们想要的，如下书写就安全了：  body  if ohnoes is defined  padding 5px |

15. 三元运算符

三元运算符的运作正如大部分语言里面的那样。三个操作对象的操作符（条件表达式、真表达式以及假表达式）。

|  |
| --- |
| num = 15  num ? unit(num, 'px') : 20px  // => 15px |

16. 铸造

作为替代简洁的内置unit()函数，语法(expr) unit可用来强制后缀。

|  |
| --- |
| body  n = 5  foo: (n)em  foo: (n)%  foo: (n + 5)%  foo: (n \* 5)px  foo: unit(n + 5, '%')  foo: unit(5 + 180 / 2, deg) |

17. 颜色操作

颜色操作提供了一个简洁，富有表现力的方式来改变其组成。例如，我们可以对每个RGB：

|  |
| --- |
| #0e0 + #0e0  // => #0f0 |

另外一个例子是通过增加或减少百分值调整颜色亮度。颜色亮，加；暗，则减。

|  |
| --- |
| #888 + 50%  // => #c3c3c3  #888 - 50%  // => #444 |

我们也可以通过增加或减去色度调整色调。例如，红色增加65deg就变成了黄色。

|  |
| --- |
| #f00 + 50deg  // => #ffd500 |

值适当固定。例如，我们可以"旋转"180度的色调，如果目前的值是320deg, 将变成140deg.

我们也可能一次调整几个值（包括alpha），通过使用rgb(), rgba(), hsl(), 或 hsla():

|  |
| --- |
| #f00 - rgba(100,0,0,0.5)  // => rgba(155,0,0,0.5) |

18. 格式化

格式化字符串模样的字符串%可以用来生成字面量值，通过传参给内置s()方法。

|  |
| --- |
| 单一值：  'X::Microsoft::Crap(%s)' % #fc0  // => X::Microsoft::Crap(#fc0) |
| 多个值：  '-webkit-gradient(%s, %s, %s)' % (linear (0 0) (0 100%))  // => -webkit-gradient(linear, 0 0, 0 100%) |

五 混合书写(mixins)

1. 混入

混入和函数定义方法一致，但是应用却大相径庭。

例如，下面有定义的border-radius(n)方法，其却作为一个mixin（如，作为状态调用，而非表达式）调用。

当border-radius()选择器中调用时候，属性会被扩展并复制在选择器中。

|  |
| --- |
| border-radius(n)  -webkit-border-radius n  -moz-border-radius n  border-radius n  form input[type=button]  border-radius(5px) |
| 编译成：  form input[type=button] {  -webkit-border-radius: 5px;  -moz-border-radius: 5px;  border-radius: 5px;  } |

使用混入书写，你可以完全忽略括号，提供梦幻般私有属性的支持。

|  |
| --- |
| border-radius(n)  -webkit-border-radius n  -moz-border-radius n  border-radius n  form input[type=button]  border-radius 5px |

注意到我们混合书写中的border-radius当作了属性，而不是一个递归函数调用。

更进一步，我们可以利用arguments这个局部变量，传递可以包含多值的表达式。

|  |
| --- |
| border-radius()  -webkit-border-radius arguments  -moz-border-radius arguments  border-radius arguments |

现在，我们可以像这样子传值：border-radius 1px 2px / 3px 4px!

另外一个很赞的应用是特定的私有前缀支持——例如IE浏览器的透明度：

|  |
| --- |
| support-for-ie ?= true  opacity(n)  opacity n  if support-for-ie  filter unquote('progid:DXImageTransform.Microsoft.Alpha(Opacity=' + round(n \* 100) + ')')  #logo  &:hover  opacity 0.5 |
| 渲染为：  #logo:hover {  opacity: 0.5;  filter: progid:DXImageTransform.Microsoft.Alpha(Opacity=50);  } |

2. 父级引用

混合书写可以利用父级引用字符&, 继承父业而不是自己筑巢。

例如，我们要用stripe(even, odd)创建一个条纹表格。even和odd均提供了默认颜色值，每行也指定了background-color属性。我们可以在tr嵌套中使用&来引用tr，以提供even颜色。

|  |
| --- |
| stripe(even = #fff, odd = #eee)  tr  background-color odd  &.even  &:nth-child(even)  background-color even |
| 然后，利用混合书写，如下：  table  stripe()  td  padding 4px 10px  table#users  stripe(#303030, #494848)  td  color white |

另外，stripe()的定义无需父引用：

|  |
| --- |
| stripe(even = #fff, odd = #eee)  tr  background-color odd  tr.even  tr:nth-child(even)  background-color even |
| 如果你愿意，你可以把stripe()当作属性调用。  stripe #fff #000 |

3. 混合书写中的混合书写

然，混合书写可以利用其它混合书写，建立在它们自己的属性和选择器上。

例如，下面我们创建内联comma-list()（通过inline-list()）以及逗号分隔的无序列表。

|  |
| --- |
| inline-list()  li  display inline  comma-list()  inline-list()  li  &:after  content ', '  &:last-child:after  content ''  ul  comma-list() |
| 渲染：  ul li:after {  content: ", ";  }  ul li:last-child:after {  content: "";  }  ul li {  display: inline;  } |

六 方法

1. 函数

Stylus强大之处就在于其内置的语言函数定义。其定义与混入(mixins)一致；却可以返回值。

1. 返回值

很简单的例子，两数值相加的方法：

|  |
| --- |
| add(a, b)  a + b |
| 我们可以在特定条件下使用该方法，如在属性值中：  body  padding add(10px, 5)  渲染为：  body {  padding: 15px;  } |

3. 默认参数

可选参数往往有个默认的给定表达。在Stylus中，我们甚至可以超越默认参数。

|  |
| --- |
| 我们可以在特定条件下使用该方法，如在属性值中：  add(a, b = a)  a + b  add(10, 5)  // => 15  add(10)  // => 20 |

注意：因为参数默认是赋值，我们可可以使用函数调用作为默认值。

|  |
| --- |
| add(a, b = unit(a, px))  a + b |

4. 函数体

我们可以把简单的add()方法更进一步。通过内置unit()把单位都变成px, 因为赋值在每个参数上，因此，我们可以无视单位换算。

|  |
| --- |
| add(a, b = a)  a = unit(a, px)  b = unit(b, px)  a + b  add(15%, 10deg)  // => 25 |

5. 多个返回值

Stylus的函数可以返回多个值，就像你给变量赋多个值一样。

例如，下面就是一个有效赋值：

|  |
| --- |
| sizes = 15px 10px  sizes[0]  // => 15px |

类似的，我们可以在函数中返回多个值：

|  |
| --- |
| sizes()  15px 10px  sizes()[0]  // => 15px |

有个小小的例外就是返回值是标识符。例如，下面看上去像一个属性赋值给Stylus（因为没有操作符）。

|  |
| --- |
| swap(a, b)  b a |
| 为避免歧义，我们可以使用括号，或是return关键字。  swap(a, b)  (b a)  swap(a, b)  return b a |

6. 条件

比方说，我们想要创建一个名为stringish()的函数，用来决定参数是否是字符串。我们检查val是否是字符串或缩进（类似字符）。如下，使用yes和no代替true和false.

|  |
| --- |
| stringish(val)  if val is a 'string' or val is a 'ident'  yes  else  no |
| 使用：  stringish('yay') == yes  // => true  stringish(yay) == yes  // => true  stringish(0) == no  // => true |

注意：yes和no并不是布尔值。本例中，它们只是简单的未定义标识符。

另外一个例子：

|  |
| --- |
| compare(a, b)  if a > b  higher  else if a < b  lower  else  equal |
| 使用：  compare(5, 2)  // => higher  compare(1, 5)  // => lower  compare(10, 10)  // => equal |

7. 别名

给函数起个别名，和简单，直接等于就可以了。例如上面的add()弄个别名plus(), 如下：

|  |
| --- |
| 使用：  plus = add  plus(1, 2)  // => 3 |

8. 变量函数

我们可以把函数当作变量传递到新的函数中。例如，invoke()接受函数作为参数，因此，我们可以传递add()以及sub().

|  |
| --- |
| invoke(a, b, fn)  fn(a, b)  add(a, b)  a + b  body  padding invoke(5, 10, add)  padding invoke(5, 10, sub) |
| 结果：  body {  padding: 15;  padding: -5;  } |

1. 参数

arguments 是所有函数体都有的局部变量，包含传递所有的参数

|  |
| --- |
| sum()  n = 0  for num in arguments  n = n + num  sum(1,2,3,4,5)  // => 15 |

10. 哈希示例

下面，我们定义get(hash, key)方法，用来返回key值或null. 我们遍历每个键值对，如果键值匹配，返回对应的值。

|  |
| --- |
| get(hash, key)  return pair[1] if pair[0] == key for pair in hash |
| 下面例子可以证明，语言函数模样的Stylus表达式具有更大的灵活性。 |
| hash = (one 1) (two 2) (three 3)  get(hash, two)  // => 2  get(hash, three)  // => 3  get(hash, something)  // => null |

七 关键字参数——keywords arguments

1. 关键字参数

Stylus支持关键字参数，或"kwargs". 允许你根据相关参数名引用参数。

下面这些例子功能上都是一样的。但是，我们可以在列表中的任何地方放置关键字参数。其余不键入参数将适用于尚未得到满足的参数。

|  |
| --- |
| body {  color: rgba(255, 200, 100, 0.5);  color: rgba(red: 255, green: 200, blue: 100, alpha: 0.5);  color: rgba(alpha: 0.5, blue: 100, red: 255, 200);  color: rgba(alpha: 0.5, blue: 100, 255, 200);  } |
| 等同于  body {  color: rgba(255,200,100,0.5);  color: rgba(255,200,100,0.5);  color: rgba(255,200,100,0.5);  color: rgba(255,200,100,0.5);  } |

查看函数或混合书写中接受的参数，可以使用p()方法。

|  |
| --- |
| p(rgba)  生成  inspect: rgba(red, green, blue, alpha) |

八 内置方法

1. red(color)

返回color中的红色比重。

|  |
| --- |
| red(#c00)  // => 204 |

2. green(color)

返回color中的绿色比重。

|  |
| --- |
| green(#0c0)  // => 204 |

1. blue(color)

返回color中的蓝色比重。

|  |
| --- |
| blue(#00c)  // => 204 |

4. alpha(color)

返回color中的透明度比重。

|  |
| --- |
| alpha(#fff)  // => 1  alpha(rgba(0,0,0,0.3))  // => 0.3 |

5. dark(color)

检查color是否是暗色。

|  |
| --- |
| dark(black)  // => true  dark(#005716)  // => true  dark(white)  // => false |

6. light(color)

检查color是否是亮色。

|  |
| --- |
| light(black)  // => false  light(white)  // => true  light(#00FF40)  // => true |

7. hue(color)

返回给定color的色调。

|  |
| --- |
| hue(hsla(50deg, 100%, 80%))  // => 50deg |

8. saturation(color)

返回给定color的饱和度。

|  |
| --- |
| saturation(hsla(50deg, 100%, 80%))  // => 100% |

9. lightness(color)

返回给定color的亮度。

|  |
| --- |
| lightness(hsla(50deg, 100%, 80%))  // => 80% |

10. push(expr, args…)

后面推送给定的args给expr.

|  |
| --- |
| nums = 1 2  push(nums, 3, 4, 5)  nums  // => 1 2 3 4 5 |

别名为append().

11. unshift(expr, args…)

起始位置插入给定的args给expr.

|  |
| --- |
| nums = 4 5  unshift(nums, 3, 2, 1)  nums  // => 1 2 3 4 5 |

别名为prepend().

12. keys(pairs)

返回给定pairs中的键。

|  |
| --- |
| pairs = (one 1) (two 2) (three 3)  keys(pairs)  // => one two |

1. values(pairs)

返回给定pairs中的值。

|  |
| --- |
| pairs = (one 1) (two 2) (three 3)  values(pairs)  // => 1 2 3 |

14. typeof(node)

字符串形式返回node类型。

|  |
| --- |
| type(12)  // => 'unit'  typeof(12)  // => 'unit'  typeof(#fff)  // => 'rgba'  type-of(#fff)  // => 'rgba' |

别名有type-of和type.

15. unit(unit[, type])

返回unit类型的字符串或空字符串，或者赋予type值而无需单位转换。

|  |
| --- |
| unit(10)  // => ''  unit(15in)  // => 'in'  unit(15%, 'px')  // => 15px  unit(15%, px)  // => 15px |

16. match(pattern, string)

检测string是否匹配给定的pattern.

|  |
| --- |
| match('^foo(bar)?', foo)  match('^foo(bar)?', foobar)  // => true  match('^foo(bar)?', 'foo')  match('^foo(bar)?', 'foobar')  // => true  match('^foo(bar)?', 'bar')  // => false |

17. abs(unit)

绝对值

|  |
| --- |
| abs(-5px)  // => 5px  abs(5px)  // => 5px |

18. cell(unit)

向上取整

|  |
| --- |
| ceil(5.5in)  // => 6in |

19. floor(unit)

向下取整

|  |
| --- |
| floor(5.6px)  // => 5px |

20. round(unit)

四舍五入取整

|  |
| --- |
| round(5.5px)  // => 6px  round(5.4px)  // => 5px |

21. min(a,b)

取最小值

|  |
| --- |
| min(1, 5)  // => 1 |

22. max(a,b)

取最大值

|  |
| --- |
| max(1, 5)  // => 5 |

23. even(unit)

是否为偶数。

|  |
| --- |
| even(6px)  // => true |

24. add(unit)

是否为奇数。

|  |
| --- |
| odd(5mm)  // => true |

25. sum(nums)

求和.

|  |
| --- |
| sum(1 2 3)  // => 6 |

26. avg(nums)

求平均数

|  |
| --- |
| avg(1 2 3)  // => 2 |

27. join(delim, vals…)

给定vals使用delim连接。

|  |
| --- |
| join(' ', 1 2 3)  // => "1 2 3"  join(',', 1 2 3)  // => "1,2,3"  join(', ', foo bar baz)  // => "foo, bar, baz"  join(', ', foo, bar, baz)  // => "foo, bar, baz"  join(', ', 1 2, 3 4, 5 6)  // => "1 2, 3 4, 5 6" |

1. hsla(color | h,s,l,a)

转换给定color为HSLA节点，或h,s,l,a比重值。

|  |
| --- |
| hslaa(10deg, 50%, 30%, 0.5)  // => HSLA  hslaa(#ffcc00)  // => HSLA |

29. hsla(color | h,s,l)

转换给定color为HSLA节点，或h,s,l比重值。

|  |
| --- |
| hsla(10, 50, 30)  // => HSLA  hsla(#ffcc00)  // => HSLA |

30. rgba(color | r,g,b,a)

从r,g,b,a通道返回RGBA, 或提供color来调整透明度。

|  |
| --- |
| rgba(255,0,0,0.5)  // => rgba(255,0,0,0.5)  rgba(255,0,0,1)  // => #ff0000  rgba(#ffcc00, 0.5)  // rgba(255,204,0,0.5) |

另外，stylus支持#rgba以及#rrggbbaa符号。

|  |
| --- |
| #fc08  // => rgba(255,204,0,0.5)  #ffcc00ee  // => rgba(255,204,0,0.9) |

31. rgb(color | r,g,b)

从r,g,b通道返回RGBA或生成一个RGBA节点。

|  |
| --- |
| rgb(255,204,0)  // => #ffcc00  rgb(#fff)  // => #fff |

32. lighten(color, amount)

给定color增亮amount值。该方法单位敏感，例如，支持百分比，如下：

|  |
| --- |
| lighten(#2c2c2c, 30)  // => #787878  lighten(#2c2c2c, 30%)  // => #393939 |

33. darken(color, amount)

给定color变暗amount值。该方法单位敏感，例如，支持百分比，如下：

|  |
| --- |
| darken(#D62828, 30)  // => #551010  darken(#D62828, 30%)  // => #961c1c |

34. desaturate(color, amount)

给定color饱和度减小amount.

|  |
| --- |
| desaturate(#f00, 40%)  // => #c33 |

35. saturate(color, amount)

给定color饱和度增加amount.

|  |
| --- |
| saturate(#c33, 40%)  // => #f00 |

36. invert(color)

颜色反相。红绿蓝颜色反转，透明度不变。

|  |
| --- |
| invert(#d62828)  // => #29d7d7 |

37. unquote(str | ident)

给定str引号去除，返回Literal节点。

|  |
| --- |
| unquote("sans-serif")  // => sans-serif  unquote(sans-serif)  // => sans-serif  unquote('1px / 2px')  // => 1px / 2px |

38. s(fmt, …)

s()方法类似于unquote()，不过后者返回的是Literal节点，而这里起接受一个格式化的字符串，非常像C语言的sprintf(). 目前，唯一标识符是%s.

|  |
| --- |
| s('bar()');  // => bar()  s('bar(%s)', 'baz');  // => bar("baz")  s('bar(%s)', baz);  // => bar(baz)  s('bar(%s)', 15px);  // => bar(15px)  s('rgba(%s, %s, %s, 0.5)', 255, 100, 50);  // => rgba(255, 100, 50, 0.5)  s('bar(%Z)', 15px);  // => bar(%Z)  s('bar(%s, %s)', 15px);  // => bar(15px, null) |

为表现一致检测这个%字符串操作符。

39. operate(op, left, right)

在left和right操作对象上执行给定的op.

|  |
| --- |
| op = '+'  operate(op, 15, 5)  // => 20 |

40. length([expr])

括号表达式扮演元组，length()方法返回该表达式的长度。

|  |
| --- |
| length((1 2 3 4))  // => 4  length((1 2))  // => 2  length((1))  // => 1  length(())  // => 0  length(1 2 3)  // => 3  length(1)  // => 1  length()  // => 0 |

41. warn(msg)

使用给定的error警告，并不退出。

|  |
| --- |
| warn("oh noes!") |

1. error(msg)

伴随着给定的错误msg退出。

|  |
| --- |
| add(a, b)  unless a is a 'unit' and b is a 'unit'  error('add() expects units')  a + b |

43. last(expr)

返回给定expr的最后一个值。

|  |
| --- |
| nums = 1 2 3  last(nums)  last(1 2 3)  // => 3  list = (one 1) (two 2) (three 3)  last(list)  // => (three 3) |

44. p(expr)

检查给定的expr.

|  |
| --- |
| fonts = Arial, sans-serif  p('test')  p(123)  p((1 2 3))  p(fonts)  p(#fff)  p(rgba(0,0,0,0.2))  add(a, b)  a + b  p(add) |
| 标准输出：  inspect: "test"  inspect: 123  inspect: 1 2 3  inspect: Arial, sans-serif  inspect: #fff  inspect: rgba(0,0,0,0.2)  inspect: add(a, b) |

45. opposite-position(positions)

返回给定positions相反内容。

|  |
| --- |
| opposite-position(right)  // => left  opposite-position(top left)  // => bottom right  opposite-position('top' 'left')  // => bottom right |

46. image-size(path)

返回指定path图片的width和height. 向上查找路径的方法和@import一样，paths设置的时候改变。

|  |
| --- |
| width(img)  return image-size(img)[0]  height(img)  return image-size(img)[1]  image-size('tux.png')  // => 405px 250px  image-size('tux.png')[0] == width('tux.png')  // => true |

47. add-property(name, expr)

使用给定的expr为最近的块域添加属性name。

|  |
| --- |
| something()  add-property('bar', 1 2 3)  s('bar')  body  foo: something() |
| 编译为  body {  bar: 1 2 3;  foo: bar;  } |

48. 未定义方法

未定义方法一字面量形式输出。例如，我们可以在CSS中调用rgba-stop(50%, #fff), 其会按照你所期望的显示，我们也可以使用这些内部助手。

下面这个例子中我们简单定义了方法stop(), 其返回了字面上rgba-stop()调用。

|  |
| --- |
| stop(pos, rgba)  rgba-stop(pos, rgba)  stop(50%, orange)  // => rgba-stop(50%, #ffa500) |

七 其余参数

Stylus支持name...形式的其余参数。这些参数可以消化传递给混写或函数的参数们。这在处理浏览器私有属性，如-moz或-webkit的时候很管用。

下面这个例子中，所有的参数们(1px, 2px, ...)都被一个args参数给简单消化了：

|  |
| --- |
| box-shadow(args...)  -webkit-box-shadow args  -moz-box-shadow args  box-shadow args  #login  box-shadow 1px 2px 5px #eee |
| 生成为：  #login {  -webkit-box-shadow: 1px 2px 5px #eee;  -moz-box-shadow: 1px 2px 5px #eee;  box-shadow: 1px 2px 5px #eee;  } |

我们想指定特定的参数，如x-offset，我们可以使用args[0], 或者，我们可能希望重新定义混入。

|  |
| --- |
| box-shadow(offset-x, args...)  got-offset-x offset-x  -webkit-box-shadow offset-x args  -moz-box-shadow offset-x args  box-shadow offset-x args  #login  box-shadow 1px 2px 5px #eee |
| 生成为：  #login {  got-offset-x: 1px;  -webkit-box-shadow: 1px 2px 5px #eee;  -moz-box-shadow: 1px 2px 5px #eee;  box-shadow: 1px 2px 5px #eee;  } |

1. 参数们

arguments变量可以实现表达式的精确传递，包括逗号等等。这可以弥补args...参数的一些不足，见下面的例子：

|  |
| --- |
| box-shadow(args...)  -webkit-box-shadow args  -moz-box-shadow args  box-shadow args  #login  box-shadow #ddd 1px 1px, #eee 2px 2px |
| 结果并非称心如意：  #login {  -webkit-box-shadow: #ddd 1px 1px #eee 2px 2px;  -moz-box-shadow: #ddd 1px 1px #eee 2px 2px;  box-shadow: #ddd 1px 1px #eee 2px 2px;  } |

逗号给忽略了。我们现在使用arguments重新定义这个混合书写。

|  |
| --- |
| box-shadow()  -webkit-box-shadow arguments  -moz-box-shadow arguments  box-shadow arguments  body  box-shadow #ddd 1px 1px, #eee 2px 2pxx |
| 满意  body {  -webkit-box-shadow: #ddd 1px 1px, #eee 2px 2px;  -moz-box-shadow: #ddd 1px 1px, #eee 2px 2px;  box-shadow: #ddd 1px 1px, #eee 2px 2px;  } |

八 注释

1. 注释

Stylus支持三种注释，单行注释，多行注释，以及多行缓冲注释。

2. 单行注释

跟JavaScript一样，双斜杠，CSS中不输出。

|  |
| --- |
| // 我是注释!  body  padding 5px // 蛋疼的padding |

3. 多行注释

多行注释看起来有点像CSS的常规注释。然而，它们只有在compress选项未启用的时候才会被输出。

|  |
| --- |
| /\*  \* 给定数值合体  \*/  add(a, b)  a + b |

4. 多行缓冲注释

跟多行注释类似，不同之处在于开始的时候，这里是/\*!. 这个相当于告诉Stylus压缩的时候这段无视直接输出。

|  |
| --- |
| /\*!  \* 给定数值合体  \*/  add(a, b)  a + b |

九 条件

1. 条件

条件提供了语言的控制，否则就是纯粹的静态语言。提供的条件有导入、混入、函数以及更多。下面的例子纯粹示例，并不是使用建议。

1. if / else if / else

这没什么好说的，跟一般的语言一致，if表达式满足(true)的时候执行后面语句块，否则，继续后面的else if或else.

下面这个例子，根据overload的条件，决定是使用padding还是margin.

|  |
| --- |
| 例子 1  overload-padding = true  if overload-padding  padding(y, x)  margin y x  body  padding 5px 10px |
| 例子 2  box(x, y, margin = false)  padding y x  if margin  margin y x  body  box(5px, 10px, true) |
| 另外的box()帮手：  box(x, y, margin-only = false)  if margin-only  margin y x  else  padding y x |

2. 除非(unless)

熟悉Ruby程序语言的用户应该都知道unless条件，其基本上与if相反，本质上是(!(expr)).

下面这个例子中，如果disable-padding-override是undefined或false, padding将被干掉，显示margin代替之。但是，如果是true, padding将会如期继续输出padding 5px 10px.

|  |
| --- |
| disable-padding-override = true  unless disable-padding-override is defined and disable-padding-override  padding(x, y)  margin y x  body  padding 5px 10px |
| 理解：当条件为false执行 |

3. 后缀条件

Stylus支持后缀条件，这就意味着if和unless可以当作操作符；当右边表达式为真的时候执行左边的操作对象。

例如，我们定义negative()来执行一些基本的检查。下面我们使用块式条件：

|  |
| --- |
| negative(n)  unless n is a 'unit'  error('无效数值')  if n < 0  yes  else  no |

接下来，我们利用后缀条件让我们的方法简洁。

|  |
| --- |
| negative(n)  error('无效数值') unless n is a 'unit'  return yes if n < 0  no |

当然，我们可以更进一步。如这个n < 0 ? yes : no可以用布尔代替：n < 0.

后缀条件适用于大多数的单行语句。如，@import, @charset, 混合书写等。当然，下面所示的属性也是可以的：

|  |
| --- |
| pad(types = margin padding, n = 5px)  padding unit(n, px) if padding in types  margin unit(n, px) if margin in types  body  pad()  body  pad(margin)  body  apply-mixins = true  pad(padding, 10) if apply-mixins |
| 生成为：  body {  padding: 5px;  margin: 5px;  }  body {  margin: 5px;  }  body {  padding: 10px;  } |

十 迭代

1. 迭代

Stylus允许你通过for/in对表达式进行迭代形式如下：

|  |
| --- |
| for <val-name> [, <key-name>] in <expression> |
| 例如：  body  for num in 1 2 3  foo num |
| 生成：  body {  foo: 1;  foo: 2;  foo: 3;  } |
| 例子：  body  fonts = Impact Arial sans-serif  for font, i in fonts  foo i font |
| 生成：  body {  foo: 0 Impact;  foo: 1 Arial;  foo: 2 sans-serif;  } |

2. 混合书写

我们可以在混写中使用循环实现更强大的功能，例如，我们可以把表达式对作为使用插值和循环的属性。

下面，我们定义apply(), 利用所有的arguments，这样逗号分隔以及表达式列表都会支持。

|  |
| --- |
| apply(props)  props = arguments if length(arguments) > 1  for prop in props  {prop[0]} prop[1]  body  apply(one 1, two 2, three 3)  body  list = (one 1) (two 2) (three 3)  apply(list) |

3. 函数书写

Stylus函数同样可以包含for循环。下面就是简单使用示例：

|  |
| --- |
| sum(nums)  sum = 0  for n in nums  sum += n  sum(1 2 3)  // => 6 |

连接：

|  |
| --- |
| join(delim, args)  buf = ''  for arg, index in args  if index  buf += delim + arg  else  buf += arg  join(', ', foo bar baz)  // => "foo, bar, baz" |

4. 后缀

就跟if/unless可以利用后面语句一样，for也可以。如下后缀解析的例子：

|  |
| --- |
| sum(nums)  sum = 0  sum += n for n in nums  join(delim, args)  buf = ''  buf += i ? delim + arg : arg for arg, i in args |

我们也可以从循环返回，下例子就是n % 2 == 0为true的时候返回数值。

|  |
| --- |
| first-even(nums)  return n if n % 2 == 0 for n in nums  first-even(1 3 5 5 6 3 2)  // => 6 |

十一 导入 —— @import

1. 导入

Stylus支持字面@import CSS, 也支持其他Stylus样式的动态导入。

2. 字面CSS

任何.css扩展的文件名将作为字面量。例如：

|  |
| --- |
| @import "reset.css" |
| 渲染为：  @import "reset.css" |

3. stylus 导入

当使用@import没有.css扩展，会被认为是Stylus片段（如：@import "mixins/border-radius"）。

@import工作原理为：遍历目录队列，并检查任意目录中是否有该文件（类似node的require.paths）。该队列默认为单一路径，从filename选项的dirname衍生而来。 因此，如果你的文件名是/tmp/testing/stylus/main.styl，导入将显现为/tmp/testing/stylus/。

@import也支持索引形式。这意味着当你@import blueprint, 则会理解成blueprint.styl或blueprint/index.styl. 对于库而言，这很有用，既可以展示所有特征与功能，同时又能导入特征子集。

如下很常见的库结构：

./tablet

|-- index.styl

|-- vendor.styl

|-- buttons.styl

|-- images.styl

下面这个例子中，我们设置paths选项用来为Stylus提供额外路径。在./test.styl中，我们可以@import "mixins/border-radius"或@import "border-radius"（因为./mixins 暴露给了Stylus）。

|  |
| --- |
| /\*\*  \* 依赖模块  \*/  var stylus = require('../')  , str = require('fs').readFileSync(\_\_dirname + '/test.styl', 'utf8');  var paths = [  \_\_dirname  , \_\_dirname + '/mixins'  ];  stylus(str)  .set('filename', \_\_dirname + '/test.styl')  .set('paths', paths)  .render(function(err, css){  if (err) throw err;  console.log(css);  }); |

3. JavaScript导入API

当使用.import(path)方法，这些导入是被推迟的，直到赋值。

|  |
| --- |
| var stylus = require('../')  , str = require('fs').readFileSync(\_\_dirname + '/test.styl', 'utf8');  stylus(str)  .set('filename', \_\_dirname + '/test.styl')  .import('mixins/vendor')  .render(function(err, css){  if (err) throw err;  console.log(css);  }); |

|  |
| --- |
| 下面语句：  @import 'mixins/vendor'  等同于  .import('mixins/vendor') |

十二 媒体查询—— @media

@media工作原理和在常规CSS中一样，但是，要使用Stylus的块状符号。

|  |
| --- |
| @media print  #header  #footer  display none |
| 生成为：  @media print {  #header,  #footer {  display: none;  }  } |

十三 自定义字体——@font-face

@font-face跟其在CSS中作用表现一样，在后面简单地添加个块状属性即可，类似下面：

|  |
| --- |
| @media print  #header  #footer  display none |
| 生成为：  @font-face  font-family Geo  font-style normal  src url(fonts/geo\_sans\_light/GensansLight.ttf)  .ingeo  font-family Geo |
| 生成为：  @font-face {  font-family: Geo;  font-style: normal;  src: url("fonts/geo\_sans\_light/GensansLight.ttf");  }  .ingeo {  font-family: Geo;  } |

十四 关键帧——@keyframes

1. @keyframes

Stylus支持@keyframes规则，当编译的时候转换成@-webkit-keyframes：

|  |
| --- |
| @keyframes pulse  0%  background-color red  opacity 1.0  -webkit-transform scale(1.0) rotate(0deg)  33%  background-color blue  opacity 0.75  -webkit-transform scale(1.1) rotate(-5deg)  67%  background-color green  opacity 0.5  -webkit-transform scale(1.1) rotate(5deg)  100%  background-color red  opacity 1.0  -webkit-transform scale(1.0) rotate(0deg) |
| 生成为：  @-webkit-keyframes pulse {  0% {  background-color: red;  opacity: 1;  -webkit-transform: scale(1) rotate(0deg);  }  33% {  background-color: blue;  opacity: 0.75;  -webkit-transform: scale(1.1) rotate(-5deg);  }  67% {  background-color: green;  opacity: 0.5;  -webkit-transform: scale(1.1) rotate(5deg);  }  100% {  background-color: red;  opacity: 1;  -webkit-transform: scale(1) rotate(0deg);  }  } |

2. 扩展

使用@keyframes，通过vendors变量，会自动添加私有前缀(webkit moz official)。这意味着你可以子啊任意时候立即高效地做修改。

考虑下面的例子：

|  |
| --- |
| @keyframes foo {  from {  color: black  }  to {  color: white  }  } |
| 扩增两个默认前缀，官方解析：  @-moz-keyframes foo {  0% {  color: #000;  }  100% {  color: #fff;  }  }  @-webkit-keyframes foo {  0% {  color: #000;  }  100% {  color: #fff;  }  }  @keyframes foo {  0% {  color: #000;  }  100% {  color: #fff;  }  } |
| 如果我们只想有标准解析，很简单，修改vendors： |
| vendors = official  @keyframes foo {  from {  color: black  }  to {  color: white  }  } |
| 生成为：  @keyframes foo {  0% {  color: #000;  }  100% {  color: #fff;  }  } |

十五 继承——@extend

1. 继承

Stylus的@extend指令受SASS实现的启发，基本一致，除了些轻微差异。此功能大大简化了继承其他语义规则集的语义规则集的维护。

2. 混合书写下的“继承”

尽管你可以使用混写实现类似效果，但会导致重复的CSS. 典型的模式式定义如下的几个类名，然后归结到一个元素中，例如"warning message".

该技术实现是没什么问题，但是维护就比较麻烦了。

|  |
| --- |
| message,  .warning {  padding: 10px;  border: 1px solid #eee;  }  .warning {  color: #E2E21E;  } |

3. 使用\_\_@extend\_\_

使用\_\_@extend\_\_得到同样的输出，只要把对应的选择器传给@extend即可。然后.warning选择器就会继承已经存在的.message规则。

|  |
| --- |
| .message {  padding: 10px;  border: 1px solid #eee;  }  .warning {  @extend .message;  color: #E2E21E;  } |
| 这儿是个更复杂的例子，演示了\_\_@extend\_\_如何级联。  red = #E33E1E  yellow = #E2E21E  .message  padding: 10px  font: 14px Helvetica  border: 1px solid #eee  .warning  @extends .message  border-color: yellow  background: yellow + 70%  .error  @extends .message  border-color: red  background: red + 70%  .fatal  @extends .error  font-weight: bold  color: red |
| 生成为：  .message,  .warning,  .error,  .fatal {  padding: 10px;  font: 14px Helvetica;  border: 1px solid #eee;  }  .warning {  border-color: #e2e21e;  background: #f6f6bc;  }  .error,  .fatal {  border-color: #e33e1e;  background: #f7c5bc;  }  .fatal {  font-weight: bold;  color: #e33e1e;  } |

目前Stylus与SASS不同之处在于SASS不允许\_\_@extend\_\_嵌套选择器。

|  |
| --- |
| form  button  padding: 10px  a.button  @extend form button  Syntax error: Can't extend form button: can't extend nested selectors  // 解析错误： 无法继承自 button: 不能继承嵌套选择器  on line 6 of standard input  Use --trace for backtrace. |
| Stylus中，只要选择器匹配，就可以生效： |
| form  input[type=text]  padding: 5px  border: 1px solid #eee  color: #ddd  textarea  @extends form input[type=text]  padding: 10px |
| 生成:  form input[type=text],  form textarea {  padding: 5px;  border: 1px solid #eee;  color: #ddd;  }  textarea {  padding: 10px;  } |

十六 url()

1. 内联Data URI图像

Stylus捆绑了一个可选函数，名叫url()，其替换了字面上的url()调用（且使用base64 Data URIs有条件地内联它们）。

2. 示例

通过require('stylus').url该函数本身是可用的，其接受一个options对象，当看到url()时候，返回Stylus内部调用的函数。

.define(name, callback)方法指定了一个可被调用的JavaScript函数。在这种情况下，因为我们图片在./css/images中，我们可以忽视paths选项（默认情况下，会查找相关要呈现的图像文件）。如果愿意，该行为时可以改变的。

|  |
| --- |
| stylus(str)  .set('filename', \_\_dirname + '/css/test.styl')  .define('url', stylus.url())  .render(function(err, css){  }); |

例如，想象图片在./public/images, 我们想要使用url(images/tobi.png), 我们可以传递paths公共目录。这样，它就成为了向上查找进程的一部分。

同样，如果我们想替换为url(tobi.png), 我们可以传递paths: [\_\_dirname + '/public/images'].

|  |
| --- |
| stylus(str)  .set('filename', \_\_dirname + '/css/test.styl')  .define('url', stylus.url({ paths: [\_\_dirname + '/public'] }))  .render(function(err, css){  }); |

3. 选项-options

◎ limit 大小默认限制30Kb(30000)

◎ paths 图像解析路径

十七 CSS字面量

不管什么原因，如果遇到Stylus搞不定的特殊需求，你可以使用@css使其作为CSS字面量解决之。

|  |
| --- |
| @css {  body {  font: 14px;  }  } |
| 生成为：  body {  font: 14px;  } |

十八 字符转码

Stylus可以字符转码。这可以让字符变成标识符，或是渲染成字面量。

例如：

|  |
| --- |
| body  padding 1 \+ 2 |
| 编译成：  body {  padding: 1 + 2;  } |

注意Stylus中/当作为属性使用的时候需要用括号括起来：

|  |
| --- |
| body  font 14px/1.4  font (14px/1.4) |
| body {  font: 14px/1.4;  font: 10px;  } |

十九 javascript api

1. require模块

require模块，用给定的Stylus代码字符串调用render()，以及（可选的）optional对象。

传递filename参数可以利用Stylus框架提供更好的错误报告。

|  |
| --- |
| var stylus = require('stylus');  stylus.render(str, { filename: 'nesting.css' }, function(err, css){  if (err) throw err;  console.log(css);  }); |

我们可以用更渐进的方式实现做一样的事：

|  |
| --- |
| var stylus = require('stylus');  stylus(str)  .set('filename', 'nesting.css')  .render(function(err, css){  // logic  }); |

2. .set(setting, value)

应用诸如filename的设置，或导入paths:

|  |
| --- |
| .set('filename', \_\_dirname + '/test.styl')  .set('paths', [\_\_dirname, \_\_dirname + '/mixins']) |

3. include(path)

渐进替换.set('paths',...)的就是.include(). 当其他Stylus库（已暴露路径）暴露的时候，这个是理想的选择。

|  |
| --- |
| stylus(str)  .include(require('nib').path)  .include(process.env.HOME + '/mixins')  .render(...) |

4. import(path)

推迟给定path导入，直到计算被执行。下面这个例子基本上跟在Stylus片段中执行@import 'mixins/vendor'一样：

|  |
| --- |
| var stylus = require('../')  , str = require('fs').readFileSync(\_\_dirname + '/test.styl', 'utf8');  stylus(str)  .set('filename', \_\_dirname + '/test.styl')  .import('mixins/vendor')  .render(function(err, css){  if (err) throw err;  console.log(css);  }); |

5. define(name, node)

通过传递一个Node，我们可以定义一个全局变量。当库（该库依赖于其它库可用性）里面暴露某些条件特征的时候，这个就很有用。例如Nib扩展库条件支持node-canvas, 提供图片生成。

但这并不是一直可用的，因此Nib可以定义：

|  |
| --- |
| .define('has-canvas', stylus.nodes.false);  .define('some-setting', new stylus.nodes.String('some value')); |

如果可能，Stylus也会转换JavaScript值为Stylus值。

|  |
| --- |
| .define('string', 'some string')  .define('number', 15.5)  .define('some-bool', true)  .define('list', [1,2,3])  .define('list', [1,2,[3,4,[5,6]]])  .define('list', { foo: 'bar', bar: 'baz' })  .define('families', ['Helvetica Neue', 'Helvetica', 'sans-serif']) |

下面是一些规则应用在js函数返回值上：

|  |
| --- |
| .define('get-list', function(){  return ['foo', 'bar', 'baz'];  }) |

1. .define(name, fn)

7. .use(fn)

当被调用时，给定fn被渲染器调用，允许所有上面的方法被使用。这允许插件轻易地暴露自己，定义函数，路径等。

|  |
| --- |
| var mylib = function(style){  style.define('add', add);  style.define('sub', sub);  };  stylus(str)  .use(mylib)  .render(...) |