**Overview （概览）**

作为一种 CSS 扩展, Less 不仅向后兼容 CSS, 它还使用现有的 CSS 语法新增了额外的特性. 这使得学习 Less 更轻松, 一旦有任何问题，可以随时退回使用标准的 CSS.

**Variables （变量）**

顾名思义:

@nice-blue: #5B83AD;

@light-blue: @nice-blue + #111;

#header {

color: @light-blue;

}

输出:

#header {

color: #6c94be;

}

注意，由于变量只能定义一次，实际上他们就是“常量”.

**Mixins （混合）**

混合就是一种将一系列属性从一个规则集引入(“混合”)到另一个规则集的方式。假设我们有以下样式:

.bordered {

border-top: dotted 1px black;

border-bottom: solid 2px black;

}

我们希望在另一个规则集内部使用上面这些属性。那么，我们就只需要访问我们想要的属性所在类的名称即可，就像下面这样：

#menu a {

color: #111;

.bordered;

}

.post a {

color: red;

.bordered;

}

类 .bordered 的属性现在就会同事呈现在 #menu a 和 .post a 中了（注意，同样可以将 #ids 作为 mixins）。

**Learn more**

* [More about mixins](http://www.css88.com/doc/less/features/#mixins-feature)
* [Parametric Mixins](http://www.css88.com/doc/less/features/#mixins-parametric-feature)

**Nested rules （嵌套规则）**

Less 为我们提供了嵌套的能力, 而不是合并在样式表中.假设我们有下面的 CSS:

#header {

color: black;

}

#header .navigation {

font-size: 12px;

}

#header .logo {

width: 300px;

}

在 Less 中，我们可以以下面这种方式编写:

#header {

color: black;

.navigation {

font-size: 12px;

}

.logo {

width: 300px;

}

}

这样的代码更简洁, 它模仿了 HTML 的结构.

使用这种方法照样可以在混合中包含伪类(pseudo-selectors)。下面是一个经典的 clearfix 代码，在这里使用 mixin 重写了（& 表示当前选择器的父选择器）:

.clearfix {

display: block;

zoom: 1;

&:after {

content: " ";

display: block;

font-size: 0;

height: 0;

clear: both;

visibility: hidden;

}

}

**参见**

* [Parent Selectors（父级选择器）](http://www.css88.com/doc/less/features/#parent-selectors-feature)

**Media query bubbling and nested media queries （媒体查询合嵌套媒体查询）**

媒体查询同样可以嵌套在选择器中。选择器将被复制到媒体查询体内：

.screencolor{

@media screen {

color: green;

@media (min-width:768px) {

color: red;

}

}

@media tv {

color: black;

}

}

输出:

@media screen {

.screencolor {

color: green;

}

}

@media screen and (min-width: 768px) {

.screencolor {

color: red;

}

}

@media tv {

.screencolor {

color: black;

}

}

**Operations （运算）**

任何数值，颜色和变量都可以进行运算。这里有一对示例：

@base: 5%;

@filler: @base \* 2;

@other: @base + @filler;

color: #888 / 4;

background-color: @base-color + #111;

height: 100% / 2 + @filler;

最后的输出结果与你预期的一样 -- Less 能够推断颜色和单位之间的区别。如果在一个运算中使用了单位，比如：

@var: 1px + 5;

在这个例子中 Less 会在最终输出结果中使用这个单位 -- 6px。

**Functions （函数）**

Less 提供了许多用于转换颜色，处理字符串和进行算术运算的函数。他们在函数参考一节有详细的的介绍。

这些函数使用起来非常简单。在下面的例子中我们使用 percentage 将 0.5 转换为 50%，然后将基础颜色值的饱和度增加了 5%，最后将背景颜色的亮度增加了 25% 之后又将色相值增加 8:

@base: #f04615;

@width: 0.5;

.class {

width: percentage(@width); // returns `50%`

color: saturate(@base, 5%);

background-color: spin(lighten(@base, 25%), 8);

}

**Namespaces and Accessors （命名空间和访问器）**

(不要将它与 [CSS @namespace](http://www.w3.org/TR/css3-namespace/) 或 [namespace 选择器](http://www.w3.org/TR/css3-selectors/#typenmsp)混为一谈)。

有时候，出于组织的目的，或者为了提供一些封装，你会希望将你的mixins 组合在一起。在 Less 中做到这一点非常直观，假设你想在 #bundle 下捆绑一些 mixins 和变量，以便稍候复用或者分发：

#bundle {

.button {

display: block;

border: 1px solid black;

background-color: grey;

&:hover {

background-color: white

}

}

.tab { ... }

.citation { ... }

}

现在如果我们想在 #header a 中混合 .button 类，那么我们可以这样做：

#header a {

color: orange;

#bundle > .button;

}

需要注意的是命名空间内声明的变量将只作用于该命名空间，并且在作用域外通过相同的语法是无效的，你会用它来引用一个mixin (#Namespace > .mixin-name)。因此，举例来说，你不能这么做： (#Namespace > @this-will-not-work)。

**Scope （作用域）**

Less 中的作用域与编程语言中的作用域概念非常相似。首先会在局部查找变量和混合，如果没找到，编译器就会在父作用域中查找，依次类推。

@var: red;

#page {

@var: white;

#header {

color: @var; // white

}

}

变量和混合不必在使用前声明，因此下面的代码与前面的例子等价：

@var: red;

#page {

#header {

color: @var; // white

}

@var: white;

}

**参见**

* [Lazy Loading(延迟加载)](http://www.css88.com/doc/less/features/#variables-feature-lazy-loading)

**Comments （注释）**

可以使用块注释和行注释:

/\* One hell of a block

style comment! \*/

@var: red;

// Get in line!

@var: white;

**Importing （导入）**

导入工作与你预期的一样。你可以导入一个 .less 文件，然后这个文件中的所有变量都可以使用了。对于 .less 文件而言，其扩展名是可选的。

@import "library"; // library.less

@import "typo.css";

**Variables（变量）**

在一个地方管理常用的值。

**Overview （概述）**

在你的样式表中相同的值重复几十次 *甚至上百次* 并不少见：

a,

.link {

color: #428bca;

}

.widget {

color: #fff;

background: #428bca;

}

变量通过为你提供一种在一个地方管理这些值的方法让你的代码变得更容易维护：

// 变量

@link-color: #428bca; // sea blue

@link-color-hover: darken(@link-color, 10%);

// 用法

a,

.link {

color: @link-color;

}

a:hover {

color: @link-color-hover;

}

.widget {

color: #fff;

background: @link-color;

}

**Variable Interpolation （变量插值）**

在上面的例子主要集中于*在CSS规则中使用变量管理值*，实际上它们还可以用在其他地方，比如选择器名称，属性名，URLs以及@import语句中。

**Selectors （选择器）**

版本: 1.4.0

// 变量

@mySelector: banner;

// 用法

.@{mySelector} {

font-weight: bold;

line-height: 40px;

margin: 0 auto;

}

编译为：

.banner {

font-weight: bold;

line-height: 40px;

margin: 0 auto;

}

**URLs**

// Variables

@images: "../img";

// Usage

body {

color: #444;

background: url("@{images}/white-sand.png");

}

**Import statements （导入语句）**

版本: 1.4.0

语法: @import "@{themes}/tidal-wave.less";

注意，目前都只有将变量声明在根作用域或者是当前作用域中，然后只有当前文件以及使用这个文件时才会考虑什么时候查找一个变量。这意味着这种用法通常在你注入变量到编译过程中或者在根文件的开始部分声明变量的做法是有限的。

当你引入一个CSS文件，同时不使用内联选项（比如，import语句保持不变）时让面的规则就不会应用。

示例：

// 变量

@themes: "../../src/themes";

// 用法

@import "@{themes}/tidal-wave.less";

**Properties （属性）**

版本: 1.6.0

@property: color;

.widget {

@{property}: #0ee;

background-@{property}: #999;

}

编译为:

.widget {

color: #0ee;

background-color: #999;

}

**Variable Names （变量名）**

还可以使用变量来定义变量名：

@fnord: "I am fnord.";

@var: "fnord";

content: @@var;

将会编译为：

content: "I am fnord.";

**Lazy Loading （延迟加载）**

变量是延迟加载的，在使用前不一定要预先声明。

有效的Less片段：

.lazy-eval {

width: @var;

}

@var: @a;

@a: 9%;

下面这个也是有效的Less片段：

.lazy-eval-scope {

width: @var;

@a: 9%;

}

@var: @a;

@a: 100%;

最终都会编译为：

.lazy-eval-scope {

width: 9%;

}

在定义一个变量两次时，只会使用最后定义的变量，Less会从当前作用域中向上搜索。这个行为类似于CSS的定义中始终使用最后定义的属性值。

比如：

@var: 0;

.class {

@var: 1;

.brass {

@var: 2;

three: @var;

@var: 3;

}

one: @var;

}

编译为：

.class {

one: 1;

}

.class .brass {

three: 3;

}

**default variables （默认变量）**

有时候你会用到默认变量-让你能够在没有设置某些变量的情况下设置指定的变量。这一特性并不强制要求你这么做，因为你可以很容易通过插入后定义同名变量的方式覆盖默认变量。

比如：

// library

@base-color: green;

@dark-color: darken(@base-color, 10%);

// use of library

@import "library.less";

@base-color: red;

因为[延迟加载](http://www.css88.com/doc/less/features/#variables-feature-lazy-loading)，这上面的代码能很好的工作 - 其中base-color会被重写，而dark-color依然是暗红色。

**Extend （扩展）**

extend是一个Less伪类，它会合并它所在的选择其和它所匹配的引用。

发布于 [v1.4.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

nav ul {

&:extend(.inline);

background: blue;

}

在上面设置的规则中，:extend选择器会在*.inline类出现的地方*在.inline上应用"扩展选择器"(也就是nav ul)。声明块保持原样，不会带有任何引用扩展(因为扩展并不是CSS)。

因此下面的代码：

nav ul {

&:extend(.inline);

background: blue;

}

.inline {

color: red;

}

输出：

nav ul {

background: blue;

}

.inline,

nav ul {

color: red;

}

注意nav ul:extend(.inline)选择器是如何输出得到nav ul的 - 输出之前移除了扩展，然后选择器块保持不变。如果代码块中没有放入属性，则从输入中移除它（但是扩展仍然会影响其他选择器）。

**Extend syntax （扩展语法）**

extend可以附加给一个选择器，也可以放入一个规则集中。它看起来像是一个带选择器参数伪类，也可以使用关键字all选择相邻的选择器。

示例:

.a:extend(.b) {}

// 上面的代码块与下面这个做一样的事情

.a {

&:extend(.b);

}

.c:extend(.d all) {

// 扩展".d"的所有实例，比如".x.d"或者".d.x"

}

.c:extend(.d) {

// 扩展选择器输出为".d"的唯一实例

}

它可以包含多个要扩展的类，使用逗号分割即可。

示例：

.e:extend(.f) {}

.e:extend(.g) {}

// 上面的代码与下面的做一样的事情

.e:extend(.f, .g) {}

**Extend attached to selector （为选择器附加扩展）**

给选择器附加扩展看起来就像一个普通的带参数的伪类选择器。一个选择器可以包含多个扩展分支，但是所有的扩展都必须在选择器的尾部。

* 选择器之后的扩展：pre:hover:extend(div pre)。
* 在选择器和扩展之间有空格是允许的：pre:hover :extend(div pre).
* 也允许有多个扩展: pre:hover:extend(div pre):extend(.bucket tr) - 注意这与 pre:hover:extend(div pre, .bucket tr)一样。
* 这是不允许的: pre:hover:extend(div pre).nth-child(odd)。因为扩展必须在最后。

如果一个规则集包含多个选择器，所有选择器都可以使用extend关键字。下面演示了一个规则集中多个带extend的选择器：

.big-division,

.big-bag:extend(.bag),

.big-bucket:extend(.bucket) {

// body

}

**Extend inside ruleset （规则集内的扩展）**

也可以使用&:extend(selector)语法在规则集内置入extend。将extend放入规则集内是一种将它放入单个规则选择器的快捷方式。

规则内的extend：

pre:hover,

.some-class {

&:extend(div pre);

}

与给每个选择器添加一个extend完全相同：

pre:hover:extend(div pre),

.some-class:extend(div pre) {}

**Extending nested Selectors （嵌套选择器中的扩展）**

extend还可以匹配嵌套选择器，比如有下面的Less：

示例：

.bucket {

tr { // nested ruleset with target selector

color: blue;

}

}

.some-class:extend(.bucket tr) {} // nested ruleset is recognized

输出：

.bucket tr,

.some-class {

color: blue;

}

从本质上将extend会查找编译后的CSS，而不是原始的less。

示例：

.bucket {

tr & { // nested ruleset with target selector

color: blue;

}

}

.some-class:extend(tr .bucket) {} // nested ruleset is recognized

输出：

tr .bucket,

.some-class {

color: blue;

}

**Exact Matching with Extend （扩展中的精确匹配）**

Extend默认会在选择器之间寻找精确匹配。它不管选择器是以星号开始还是不是。它也不管两个nth表达式是否具有相同的意义，它们必须以相同的形式匹配。唯一例外的是属性选择器中的引号，less会知道它们是相同的，然后匹配它。

示例：

.a.class,

.class.a,

.class > .a {

color: blue;

}

.test:extend(.class) {} // 不会匹配上面的任何选择器的值

号开头也是有关系的。选择器\*.class和.class是等价的，而extend不会匹配它们：

\*.class {

color: blue;

}

.noStar:extend(.class) {} // 不会匹配\*.class选择器

输出：

\*.class {

color: blue;

}

伪类的顺序是有关系的。选择器link:hover:visited和link:visited:hover匹配相同的元素集合，但是extend会区别对待它们：

link:hover:visited {

color: blue;

}

.selector:extend(link:visited:hover) {}

输出：

link:hover:visited {

color: blue;

}

**nth expression （nth表达式）**

Nth形式的表达式也是有关系的。Nth表达式1n+3和n+3是等价的，但是extend并不能匹配它们：

:nth-child(1n+3) {

color: blue;

}

.child:extend(:nth-child(n+3)) {}

输出：

:nth-child(1n+3) {

color: blue;

}

属性选择器中的引号类型也是有关系的。以下所有都是等价的：

[title=identifier] {

color: blue;

}

[title='identifier'] {

color: blue;

}

[title="identifier"] {

color: blue;

}

.noQuote:extend([title=identifier]) {}

.singleQuote:extend([title='identifier']) {}

.doubleQuote:extend([title="identifier"]) {}

输出：

[title=identifier],

.noQuote,

.singleQuote,

.doubleQuote {

color: blue;

}

[title='identifier'],

.noQuote,

.singleQuote,

.doubleQuote {

color: blue;

}

[title="identifier"],

.noQuote,

.singleQuote,

.doubleQuote {

color: blue;

}

**Extend "all"**

当你在extend参数的最后面指定all关键字时，它会告诉告诉匹配作为其他选择器一部分的选择器。这个选择器会被复制，然后匹配的选择器部分会使用扩展替换，创建一个新的选择器。

示例：

.a.b.test,

.test.c {

color: orange;

}

.test {

&:hover {

color: green;

}

}

.replacement:extend(.test all) {}

输出：

.a.b.test,

.test.c,

.a.b.replacement,

.replacement.c {

color: orange;

}

.test:hover,

.replacement:hover {

color: green;

}

*你可以认为这种操作模式就是无损搜索和替换。*

**Selector Interpolation with Extend （扩展中的选择器插值）**

Extend不能匹配变量选择器。如果选择器包含变量，extend会忽略它。

这是一个悬而未决的特性，改变它并不容易。然而，extend可以附加给插值选择器。

带变量的选择器不会匹配：

@variable: .bucket;

@{variable} { // 插值选择器

color: blue;

}

.some-class:extend(.bucket) {} // 找不到匹配

同时在extend中使用目标选择器变量也什么都不匹配：

.bucket {

color: blue;

}

.some-class:extend(@{variable}) {} // 插值选择器什么也不匹配

@variable: .bucket;

上面两个例子都会编译为：

.bucket {

color: blue;

}

然而, :extend 附加给插值选择器是能够工作的：

.bucket {

color: blue;

}

@{variable}:extend(.bucket) {}

@variable: .selector;

上面的例子会编译为：

.bucket, .selector {

color: blue;

}

**Scoping / Extend Inside @media （作用域/@media内的扩展）**

编写在media声明内的extend也应该只匹配同一media声明内的选择器：

@media print {

.screenClass:extend(.selector) {} // media内的extend

.selector { // 这个会匹配到-因为在同一的media内

color: black;

}

}

.selector { // 定义样式表中的规则 - extend会忽略它

color: red;

}

@media screen {

.selector { // 另一个media声明内的规则 - extend也会忽略它

color: blue;

}

}

最终编译为：

@media print {

.selector,

.screenClass { /\* 同一media内的规则扩展成功 \*/

color: black;

}

}

.selector { /\* 定义样式表中的规则被忽略 \*/

color: red;

}

@media screen {

.selector { /\* 其他media中的规则也被忽略 \*/

color: blue;

}

}

编写在media声明内的extend不会匹配嵌套声明内的选择器：

@media screen {

.screenClass:extend(.selector) {} // media内的extend

@media (min-width: 1023px) {

.selector { // 嵌套media内的规则 - extend会忽略它

color: blue;

}

}

}

编译为：

@media screen and (min-width: 1023px) {

.selector { /\* 其他嵌套media内的规则被忽略 \*/

color: blue;

}

}

顶级extend匹配一切，包括media嵌套内的选择器：

@media screen {

.selector { /\* media嵌套内的规则 - 顶级extend正常工作 \*/

color: blue;

}

@media (min-width: 1023px) {

.selector { /\* media嵌套内的规则 - 顶级extend正常工作 \*/

color: blue;

}

}

}

.topLevel:extend(.selector) {} /\* 顶级extend匹配一切 \*/

编译为：

@media screen {

.selector,

.topLevel { /\* media嵌套内的规则被扩展了 \*/

color: blue;

}

}

@media screen and (min-width: 1023px) {

.selector,

.topLevel { /\* media嵌套内的规则被扩展了 \*/

color: blue;

}

}

**Duplication Detection （检测重复）**

现在，这里还没有检测重复。

示例：

.alert-info,

.widget {

/\* declarations \*/

}

.alert:extend(.alert-info, .widget) {}

输出：

.alert-info,

.widget,

.alert,

.alert {

/\* declarations \*/

}

**Use Cases for Extend （扩展用例）**

**Classic Use Case （经典用例）**

经典用于就是避免添加基础类。比如，如果你有：

.animal {

background-color: black;

color: white;

}

如果你想有一个animal子类型，并且要重写背景颜色。那么你有两个选择，首先改变你的HTML

<a class="animal bear">Bear</a>

.animal {

background-color: black;

color: white;

}

.bear {

background-color: brown;

}

或者简化HTML，然后在你的less中使用extend，比如：

<a class="bear">Bear</a>

.animal {

background-color: black;

color: white;

}

.bear {

&:extend(.animal);

background-color: brown;

}

**Reducing CSS Size （CSS尺寸归并）**

Mixins会复制所有的属性到选择器中，这可能导致不必要的重复。因此你可以使用extend来代替mixin将你要用的属性移过去，这样就会生成更少的CSS。

mixin示例：

.my-inline-block() {

display: inline-block;

font-size: 0;

}

.thing1 {

.my-inline-block;

}

.thing2 {

.my-inline-block;

}

输出：

.thing1 {

display: inline-block;

font-size: 0;

}

.thing2 {

display: inline-block;

font-size: 0;

}

示例 (用扩展):

.my-inline-block {

display: inline-block;

font-size: 0;

}

.thing1 {

&:extend(.my-inline-block);

}

.thing2 {

&:extend(.my-inline-block);

}

输出：

.my-inline-block,

.thing1,

.thing2 {

display: inline-block;

font-size: 0;

}

**Combining Styles / a more advanced mixin （合并样式/更高级的mixin）**

另一个用例可以用作mixin的替代 - 因为mixin仅仅能用于简单的选择器，如果你的html中有两个不同的块，但是你需要为这两个块应用相同的样式，那么你可以使用extend来关联这两块。

示例：

li.list > a {

// list styles

}

button.list-style {

&:extend(li.list > a); // 使用相同的列表样式

}

**Mixins (混合)**

从现有的样式混合（mixin）属性

你可以混合“类”选择器或者“id”选择器，例如：

.a, #b {

color: red;

}

.mixin-class {

.a();

}

.mixin-id {

#b();

}

以上将得到:

.a, #b {

color: red;

}

.mixin-class {

color: red;

}

.mixin-id {

color: red;

}

小提示：当你调用混合集的时候，括号可加可不加。

.a(); //这两种调用方式效果是一样的

.a;

**Not outputting the mixin （不输出混合集）**

如果你想要创建一个混合集，但是却不想让它输出到你的样式中，你可以在混合集的名字后面加上一个括号。

.my-mixin {

color: black;

}

.my-other-mixin() {

background: white;

}

.class {

.my-mixin;

.my-other-mixin;

}

输出：

.my-mixin {

color: black;

}

.class {

color: black;

background: white;

}

**Selectors in mixins （带选择器的混合集）**

混合集不仅可以包含各种属性，而且可以包括各种选择器。

例如：

.my-hover-mixin() {

&:hover {

border: 1px solid red;

}

}

button {

.my-hover-mixin();

}

输出：

button:hover {

border: 1px solid red;

}

**Namespaces （命名空间）**

如果你想要将属性混合到比较复杂的选择器中，你可以通过嵌套多层id或者class。

#outer {

.inner {

color: red;

}

}

.c {

#outer > .inner;

}

同样 > 是可选的

// 下面四种写法效果是一样的

#outer > .inner;

#outer > .inner();

#outer.inner;

#outer.inner();

这种用法的效果相当于我们熟知的命名空间，你可以把混合集放到一个id选择器里面，这样可以确保它（这个混合集）不会跟其他的库冲突。

例如：

#my-library {

.my-mixin() {

color: black;

}

}

// 可以这样调用

.class {

#my-library > .my-mixin();

}

**The !important keyword （!important 关键字）**

在调用的混合集后面追加 !important 关键字，可以使混合集里面的所有属性都继承 !important：

例如：

.foo (@bg: #f5f5f5, @color: #900) {

background: @bg;

color: @color;

}

.unimportant {

.foo();

}

.important {

.foo() !important;

}

结果为:

.unimportant {

background: #f5f5f5;

color: #900;

}

.important {

background: #f5f5f5 !important;

color: #900 !important;

}

**Parametric Mixins （带参数的混合）**

如何给mixins传递参数

mixins也可以接受参数，在它进行mix in操作时会将变量传递给选择器代码块。

比如：

.border-radius(@radius) {

-webkit-border-radius: @radius;

-moz-border-radius: @radius;

border-radius: @radius;

}

接下来，你可以在系一些规则集中混入变量值：

#header {

.border-radius(4px);

}

.button {

.border-radius(6px);

}

对于这些进行mixin操作的参数也可以有默认值：

.border-radius(@radius: 5px) {

-webkit-border-radius: @radius;

-moz-border-radius: @radius;

border-radius: @radius;

}

然后你可以像这样调用它：

#header {

.border-radius;

}

这里仍然会包含一个5px的border-radius。

你也可以使用不接受参数的mixins。如果你想从输出的CSS中隐藏规则集，但是又想在其他规则集中包含它的属性，这个特性是很有用的：

.wrap() {

text-wrap: wrap;

white-space: -moz-pre-wrap;

white-space: pre-wrap;

word-wrap: break-word;

}

pre { .wrap }

这会输出：

pre {

text-wrap: wrap;

white-space: -moz-pre-wrap;

white-space: pre-wrap;

word-wrap: break-word;

}

**Mixins With Multiple Parameters （带多个参数的mixins）**

参数可以用*分号*或者*逗号*分割。但是推荐使用*分号*分割。因为逗号符号有两个意思：它可以解释为mixins参数分隔符或者css列表分隔符。

使用逗号作为mixin的分隔符则无法用它创建逗号分割的参数列表。换句话说，如果编译器在mixin调用或者声明中看到至少一个分号，它会假设参数是由分号分割的，而所有的逗号都属于CSS列表:

* 两个参数，并且每个参数都是逗号分割的列表：.name(1,2,3;something, ele)，
* 三个参数，并且每个参数都包含一个数字：.name(1,2,3)，
* 使用伪造的分号创建mixin，调用的时候参数包含一个逗号分割的css列表：.name(1,2,3;)，
* 逗号分割默认值：.name(@param1: red, blue)。

定义多个具有相同名称和参数数量的mixins是合法的。Less会使用它可以应用的属性。如果使用mixin的时候只带一个参数，比如.mixin(green)，这个属性会导致所有的mixin都会使用强制使用这个明确的参数：

.mixin(@color) {

color-1: @color;

}

.mixin(@color; @padding: 2) {

color-2: @color;

padding-2: @padding;

}

.mixin(@color; @padding; @margin: 2) {

color-3: @color;

padding-3: @padding;

margin: @margin @margin @margin @margin;

}

.some .selector div {

.mixin(#008000);

}

会编译为:

.some .selector div {

color-1: #008000;

color-2: #008000;

padding-2: 2;

}

**Named Parameters （命名参数）**

引用mixin时可以通过参数名称而不是参数的位置来为mixin提供参数值。任何参数都已通过它的名称来引用，这样就不必按照任意特定的顺序来使用参数：

.mixin(@color: black; @margin: 10px; @padding: 20px) {

color: @color;

margin: @margin;

padding: @padding;

}

.class1 {

.mixin(@margin: 20px; @color: #33acfe);

}

.class2 {

.mixin(#efca44; @padding: 40px);

}

会编译为：

.class1 {

color: #33acfe;

margin: 20px;

padding: 20px;

}

.class2 {

color: #efca44;

margin: 10px;

padding: 40px;

}

**The @arguments variable （@arguments 变量）**

@arguments在mixins内部有特殊意义，调用mixin时，它包含所有传入的参数。如果你不想单个单个的处理参数，这一特性是很有用的：

.box-shadow(@x: 0; @y: 0; @blur: 1px; @color: #000) {

-webkit-box-shadow: @arguments;

-moz-box-shadow: @arguments;

box-shadow: @arguments;

}

.big-block {

.box-shadow(2px; 5px);

}

返回结果为：

.big-block {

-webkit-box-shadow: 2px 5px 1px #000;

-moz-box-shadow: 2px 5px 1px #000;

box-shadow: 2px 5px 1px #000;

}

**Advanced arguments and the @rest variable （高级参数和@rest变量）**

如果你希望你的mixin接受数量不定的参数，你可以使用...。在变量名后面使用它，它会将这些参数分配给变量。

.mixin(...) { // matches 0-N arguments

.mixin() { // matches exactly 0 arguments

.mixin(@a: 1) { // matches 0-1 arguments

.mixin(@a: 1; ...) { // matches 0-N arguments

.mixin(@a; ...) { // matches 1-N arguments

此外：

.mixin(@a; @rest...) {

// @rest is bound to arguments after @a

// @arguments is bound to all arguments

}

**Pattern-matching （模式匹配）**

有时候，你可能想要基于你传递给它的参数改变mixin的行为。先来看一些基础的示例：

.mixin(@s; @color) { ... }

.class {

.mixin(@switch; #888);

}

现在，比方说你想要.mixin基于@switch的值以不同的方式表现，你可这样定义这个mixin：

.mixin(dark; @color) {

color: darken(@color, 10%);

}

.mixin(light; @color) {

color: lighten(@color, 10%);

}

.mixin(@\_; @color) {

display: block;

}

现在，如果运行它：

@switch: light;

.class {

.mixin(@switch; #888);

}

这将得到如下CSS：

.class {

color: #a2a2a2;

display: block;

}

这里传递给.mixin的贪色变淡了。如果@switch的值是dark，结果会变成暗色。

这里发生了什么：

* 第一个mixin定义并没有匹配，因为它期望第一个参数是dark。
* 第二个mixin定义匹配了，因为它接受的参数是预期的light。
* 第三个mixin定义也匹配了，因为它任何值都在其预料只用。

这里只会使用匹配的mixin。变量匹配，然后绑定给任意变量。

除了变量匹配，只有一个值与其自身相等。

你也可以基于参数数量来匹配，这里有个例子：

.mixin(@a) {

color: @a;

}

.mixin(@a; @b) {

color: fade(@a; @b);

}

现在，如果我们用一个参数来调用.mixin，这将会输出第一个定义，但是如果我们使用*两个*参数调用它，这回获取第二个定义，这就是@a淡入到@b。

**Mixins as Functions （作为函数使用的混合）**

从mixin中返回变量

所有定义在一个mixin中的变量都是可见的，还可以用于调用它的作用域中（除非调用它的作用域定义了同名变量）。

示例：

.mixin() {

@width: 100%;

@height: 200px;

}

.caller {

.mixin();

width: @width;

height: @height;

}

结果：

.caller {

width: 100%;

height: 200px;

}

因此定义在mixin中的变量还可以充当它的返回值。这样就允许我们创建一个用起来类似函数的mixin。

示例：

.average(@x, @y) {

@average: ((@x + @y) / 2);

}

div {

.average(16px, 50px); // "call" the mixin

padding: @average; // use its "return" value

}

结果：

div {

padding: 33px;

}

直接定义在调用者作用域内的变量不能被重写。然而，定义变在量调用者父级作用域内的变量是不是受保护的，将被重写：

.mixin() {

@size: in-mixin;

@definedOnlyInMixin: in-mixin;

}

.class {

margin: @size @definedOnlyInMixin;

.mixin();

}

@size: globaly-defined-value; // 调用者父级作用域 - 不受保护

结果:

.class {

margin: in-mixin in-mixin;

}

最后，定义在mixin中的mixin同样可以作为返回值：

.unlock(@value) { // 外层的 mixin

.doSomething() { // 被嵌套的 mixin

declaration: @value;

}

}

#namespace {

.unlock(5); // unlock doSomething mixin

.doSomething(); //嵌套混入被复制到这里，并可用

}

结果:

#namespace {

declaration: 5;

}

**Passing Rulesets to Mixins （传递规则集给混合）**

允许在mixin中定义包装的CSS块

Released [v1.7.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

分离（detached）规则集合是一组CSS属性，嵌套的规则集合，媒体声明或是存储在一个变量中的任何其他东西。你可以将其包含到一个规则集合或其他结构中，它的所有属性将被复制在那里。你也可以使用它作为一个mixin参数，并传递它周围的其他任何变量。

简单示例:

// 声明 detached 规则集合

@detached-ruleset: { background: red; };

// 使用 detached 规则集合

.top {

@detached-ruleset();

}

编译结果为：

.top {

background: red;

}

调用时分离（detached）规则集合后面的圆括号是必须的， @detached-ruleset; 这样调用是无效的。

当你希望定义一个mixin将一个媒体查询中的一个代码块或者一个浏览器不支持的类名抽象出来时很有用。规则集合可以传递规则集给mixin，所以该mixin会包装这些内容。比如：

.desktop-and-old-ie(@rules) {

@media screen and (min-width: 1200) { @rules(); }

html.lt-ie9 & { @rules(); }

}

header {

background-color: blue;

.desktop-and-old-ie({

background-color: red;

});

}

这里的desktop-and-old-ie mixin定义了媒体查询和祖先类，因此你可以使用mixin来包装一段代码。上面这段代码会输出：

header {

background-color: blue;

}

@media screen and (min-width: 1200) {

header {

background-color: red;

}

}

html.lt-ie9 header {

background-color: red;

}

一个规则集合现在可以分配给一个变量或传递给一个mixin，并且可以包含全部的less特性，例如

@my-ruleset: {

.my-selector {

background-color: black;

}

};

你甚至可以利用 [媒体查询 media query bubbling](http://www.css88.com/doc/less/features/#features-overview-feature-media-query-bubbling-and-nested-media-queries), for instance

@my-ruleset: {

.my-selector {

@media tv {

background-color: black;

}

}

};

@media (orientation:portrait) {

@my-ruleset();

}

它将输出：

@media (orientation: portrait) and tv {

.my-selector {

background-color: black;

}

}

分离规则集合可以解锁（返回）所有它的mixins给调用者，可以以同样的方式调用mixin。但是，它不会返回变量。

返回 mixin:

// 带有mixin的分离规则集合

@detached-ruleset: {

.mixin() {

color:blue;

}

};

// 调用分离规则集合

.caller {

@detached-ruleset();

.mixin();

}

结果:

.caller {

color: blue;

}

私有变量:

detached-ruleset: {

@color:blue; // 私有变量

};

.caller {

color: @color; // 语法错误

}

**Scoping （作用域）**

分离规则集合可以在它被 *定义* 和被 *调用* 的地方使用所有变量和混入。换句话说， 定义和调用的作用域对它都是有效的。如果这两个作用域包含相同的变量或混入，声明的作用域中的值优先。

定义分离规则集合体的 *声明的作用域* 是独立的。从一个变量复制分离规则集合到另一个不能修改其作用域。规则集合不会获得新的作用域，只是在那里被引用。

最后，分离规则集合可以通过被解锁（导入）获得到它作用域。

**Definition and Caller Scope Visibility （定义和调用者作用域的可见性）**

独立的混合可以调用变量和混合：

@detached-ruleset: {

caller-variable: @callerVariable; // 这里变量是 undefined

.callerMixin(); // 这里混合是 undefined

};

selector {

// 使用分离规则集合

@detached-ruleset();

// 需要在分离规则集合内定义变量和混合

@callerVariable: value;

.callerMixin() {

variable: declaration;

}

}

编译后的结果:

selector {

caller-variable: value;

variable: declaration;

}

访问形式定义的变量和混合（mixins）优先于调用者内部提供的值（愚人码头注：考虑其作用域，更好懂）：

@variable: global;

@detached-ruleset: {

// 将使用全局变量，因为他是访问形式定义的分离规则集合

variable: @variable;

};

selector {

@detached-ruleset();

@variable: value; // 在调用者内部定义的变量 - 将被忽略

}

编译为:

selector {

variable: global;

}

**Referencing *Won't* Modify Detached Ruleset Scope （引用 *不会* 修改分离规则集合的作用域）**

规则集合不会获得新的作用域，只是被引用：

@detached-1: { scope-detached: @one @two; };

.one {

@one: visible;

.two {

@detached-2: @detached-1; // 拷贝/重命名 规则集合

@two: visible; // 规则集合不能使用visible

}

}

.usePlace {

.one > .two();

@detached-2();

}

跑出一个错误:

ERROR 1:32 The variable "@one" was not declared.

**Unlocking *Will* Modify Detached Ruleset Scope （解锁 *会* 修改分离规则集合的作用域）**

规则集合在解锁（导入）会获得新的作用域：

#space {

.importer1() {

@detached: { scope-detached: @variable; }; // 定义分离规则集合

}

}

.importer2() {

@variable: value; // 解锁分离规则集合能使用这个变量

#space > .importer1(); // 解锁/导入分离规则集合

}

.usePlace {

.importer2(); // 第二次解锁/导入分离规则集合

@detached();

}

编译为:

.usePlace {

scope-detached: value;

}

**Import Directives （导入准则）**

从其他样式表中导入样式。

在标准的CSS中，@import必须在所有其他类型的规则之前。但是Less.js不在乎你把@import语句放在什么位置。

示例：

.foo {

background: #900;

}

@import "this-is-valid.less";

**File extensions （文件扩展名）**

@import语句会通过Less依赖文件扩展名的方式区别对待不同的文件：

* 如果文件有一个.css扩展名，则将它作为CSS对象，同时@import语句保持不变（查看下面的inline选项）
* 如果有*其他扩展名*，则作为Less对象，然后导入它。
* 如果没有扩展名，则插入.less，然后将它作为Less文件导入包含进来。

示例：

@import "foo"; // foo.less is imported

@import "foo.less"; // foo.less is imported

@import "foo.php"; // foo.php imported as a less file

@import "foo.css"; // statement left in place, as-is

下面的选项可以用来重写这一行为。

**Import Options （导入选项）**

Less提供了一系列的CSS扩展来让你使用@import更灵活的导入第三方CSS文件。

语法：@import (keyword) "filename";

下面导入指令已经被实现了：

* reference：使用Less文件但不输出
* inline：在输出中包含源文件但不加工它
* less：将文件作为Less文件对象，无论是什么文件扩展名
* css：将文件作为CSS文件对象，无论是什么文件扩展名
* once：只包含文件一次（默认行为）
* multiple：包含文件多次

**reference （引用）**

使用@import (reference)导入外部文件，但是不添加导入的样式到编译输出中，只引用。

发布于 [v1.5.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例: @import (reference) "foo.less";

reference 是Less语言中最强大的特性之一。想象以下，reference会使用一个*引用标记*在导入的文件中标记每个指令和选择器，正常导入它，但是生成CSS的时候，"引用的"选择器不会输出（和media query一样只包含选择器引用）。reference样式不会显示在生成的CSS中，除了应用作为[mixins](http://www.css88.com/doc/less/features/#mixins-feature) 或者 [extended](http://www.css88.com/doc/less/features/#extend-feature)的样式。

此外，**reference**还依赖于使用的方法（mixin或者extend）生成不同的结果：

* [**extend**](http://www.css88.com/doc/less/features/#extend-feature)：当一个选择是extended时，之后新的选择器会标记为*非引用*，然后将它插入引用@import语句的位置。
* [**mixins**](http://www.css88.com/doc/less/features/#mixins-feature)：当reference样式用作 [隐性的 mixin](http://www.css88.com/doc/less/features/#mixins-feature) 时，混合它的规则，标记为*非引用*，然后正常出现在引用它的地方。

**reference example （引用示例）**

这允许你通过做一些像下面这样的工作从诸如[Bootstrap](https://github.com/twbs/bootstrap)的库中拉入特定的，目标样式：

.navbar:extend(.navbar all) {}

这样就只会从Bootstrap中拉入.navbar相关的样式。

**inline**

使用@import (inline)引入外部文件，但不加工他们。

发布于 [v1.5.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例: @import (inline) "not-less-compatible.css";

当一个CSS文件可能不兼容Less的时候可以使用这一技术，这是因为尽管Less支持大多数熟知的标准的CSS，但是在有些地方它还是不支持注释，在不修改CSS的情况下它也不支持所有已知的CSS hacks。

因此你可以使用它来在输出中引入文件，最终CSS文件都会在一个地方。

**less**

使用@import (less)会将导入的文件作为Less文件对象，不管文件扩展名是什么。

发布于 [v1.4.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例：

@import (less) "foo.css";

**css**

使用@import (css)会将带入的文件作为普通的CSS文件对象，也不会管扩展名是什么。这意味着import语句把持不变。

发布于 [v1.4.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例：

@import (css) "foo.less";

输出：

@import "foo.less";

**once**

@import语句的默认行为。这意味着文件只会被导入一次，而随后的导入文件的语句都会被忽略。

发布于 [v1.4.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

这个@import语句的默认行为。

示例：

@import (once) "foo.less";

@import (once) "foo.less"; // this statement will be ignored

**multiple**

使用@import (multiple)允许导入多个同名文件。这与只能导入一次的行为是对立的。

发布于 [v1.4.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例：

// file: foo.less

.a {

color: green;

}

// file: main.less

@import (multiple) "foo.less";

@import (multiple) "foo.less";

输出：

.a {

color: green;

}

.a {

color: green;

}

**Mixin Guards**

带条件的mixins。

当你想要匹配*表达式*，而不是简单的值或者参数数量时，guard是很有用的。如果你熟悉函数式编程，那么你肯定遇到过这类问题。

为了尽可能的保持CSS声明的本质，Less选择实现了**guarded mixins**，而不是if/else语句，也就是说并不是一脉相承的实现@media查询的规范。

让我们从一个例子开始：

.mixin (@a) when (lightness(@a) >= 50%) {

background-color: black;

}

.mixin (@a) when (lightness(@a) < 50%) {

background-color: white;

}

.mixin (@a) {

color: @a;

}

这里有一个when关键字，它引进了一个guard序列（在这里只有一个guard）。现在，假设我们运行以下代码：

.class1 { .mixin(#ddd) }

.class2 { .mixin(#555) }

会得到：

.class1 {

background-color: black;

color: #ddd;

}

.class2 {

background-color: white;

color: #555;

}

**Guard comparison operators （Guard中的比较运算符）**

guards中可用的比较运算符的完整列表为： >, >=, =, =<, <。此外，关键字true是让两个mixins等价的唯一真值：

.truth (@a) when (@a) { ... }

.truth (@a) when (@a = true) { ... }

除了关键字true，其他任何值都是假值：

.class {

.truth(40); // 将不符合任何上述定义。

}

注意，你也可以比较其他每个参数或者不使用参数：

@media: mobile;

.mixin (@a) when (@media = mobile) { ... }

.mixin (@a) when (@media = desktop) { ... }

.max (@a; @b) when (@a > @b) { width: @a }

.max (@a; @b) when (@a < @b) { width: @b }

**Guard logical operators （Guard逻辑运算符）**

您可以在guards之间使用逻辑运算符。语法是基于CSS媒体查询。

使用and关键字来组合guards：

.mixin (@a) when (isnumber(@a)) and (@a > 0) { ... }

你可以通过用逗号 , 分隔guards来模仿 *or* 运算符。如果任何 guards 为 true，那么它认为是匹配的：

.mixin (@a) when (@a > 10), (@a < -10) { ... }

使用 not 关键字来否定条件：

.mixin (@b) when not (@b > 0) { ... }

**Type checking functions （类型检查函数）**

最后，如果你想基于值类型匹配mixins，那么你可以使用is函数：

.mixin (@a; @b: 0) when (isnumber(@b)) { ... }

.mixin (@a; @b: black) when (iscolor(@b)) { ... }

下面是一些基本的类型检查函数：

* iscolor
* isnumber
* isstring
* iskeyword
* isurl

如果你想检查一个值除了数字是否是一个特定的单位，你可以使用下列方法之一：

* ispixel
* ispercentage
* isem
* isunit

**Conditional mixins （带条件的mixins）**

*(****FIXME****)* 此外，default函数可以用于让一个mixin匹配依赖于其他mixin匹配，然后你可以使用它来创建类似于else或者default语句（分别属于if和case结构）的“条件式mixins”：

.mixin (@a) when (@a > 0) { ... }

.mixin (@a) when (default()) { ... } // matches only if first mixin does not, i.e. when @a <= 0

**CSS Guards**

类似"if"形式的选择器

发布于 [v1.5.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

约束也适用于CSS选择器，这是一个声明mixin的语法糖，会立即调用它。

例如，在1.5.0之前你不得不这样做。

.my-optional-style() when (@my-option = true) {

button {

color: white;

}

}

.my-optional-style();

现在你可以直接在样式上编写约束。

button when (@my-option = true) {

color: white;

}

你还可以通过与&特性结合实现'if'类型的语句，从而允许组合多个约束。

& when (@my-option = true) {

button {

color: white;

}

a {

color: blue;

}

}

**Loops （循环）**

编写循环

在Less中，混合可以调用它自身。这样，当一个混合递归调用自己，再结合[Guard表达式](http://www.css88.com/doc/less/features/#mixin-guards-feature)和[模式匹配](http://www.css88.com/doc/less/features/#mixins-parametric-feature-pattern-matching)这两个特性，就可以写出循环结构。

示例：

.loop(@counter) when (@counter > 0) {

.loop((@counter - 1)); // 递归调用自身

width: (10px \* @counter); // 每次调用时产生的样式代码

}

div {

.loop(5); // 调用循环

}

输出：

div {

width: 10px;

width: 20px;

width: 30px;

width: 40px;

width: 50px;

}

使用递归循环最常见的情况就是生成栅格系统的CSS：

.generate-columns(4);

.generate-columns(@n, @i: 1) when (@i =< @n) {

.column-@{i} {

width: (@i \* 100% / @n);

}

.generate-columns(@n, (@i + 1));

}

输出：

.column-1 {

width: 25%;

}

.column-2 {

width: 50%;

}

.column-3 {

width: 75%;

}

.column-4 {

width: 100%;

}

**Merge （合并）**

合并属性

merge特性可以从多个属性中将值集合集合到一个单一属性之下的逗号或空格分割属性列表中。对于诸如background和transform之类的属性来说，merge非常有用。

**Comma （逗号）**

通过逗号添加属性的值

发布于 [v1.5.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例：

.mixin() {

box-shadow+: inset 0 0 10px #555;

}

.myclass {

.mixin();

box-shadow+: 0 0 20px black;

}

输出：

.myclass {

box-shadow: inset 0 0 10px #555, 0 0 20px black;

}

**Space （作用域）**

作用域内附加属性

发布于 [v1.7.0](https://github.com/less/less.js/blob/master/CHANGELOG.md)

示例:

.mixin() {

transform+\_: scale(2);

}

.myclass {

.mixin();

transform+\_: rotate(15deg);

}

输出：

.myclass {

transform: scale(2) rotate(15deg);

}

为避免任何非有意的添加，merge需要在每个待加入的声明中显示的设置一个+或者+\_标记。

*注意：尽管transform规范中属性使用空格分割，但它仍然支持使用逗号分割；这也是为什么这个特性中没有选项来配置用空格还是逗号分割的原因。*

**Parent Selectors （父级选择器）**

使用&引用父选择器

& 运算符表示一个 [嵌套规则](http://www.css88.com/doc/less/features/#features-overview-feature-nested-rules) 的父选择器，它在应用修改类或者应用伪类给现有选择器时最常用：

a {

color: blue;

&:hover {

color: green;

}

}

结果为：

a {

color: blue;

}

a:hover {

color: green;

}

注意，如果上面的示例没有使用&，那么它的结果就是a :hover（一个匹配<a>标签内的hovered元素的后代选择器），这通常并不是我么想要的嵌套的:hover的结果。

“父选择器”有各种各样的用法。基本上，任何时候你都需要以不同的方式来组合选择器嵌套的规则，而不是默认规则。比如，一个使用&的典型的场景就是生成重复的类名：

.button {

&-ok {

background-image: url("ok.png");

}

&-cancel {

background-image: url("cancel.png");

}

&-custom {

background-image: url("custom.png");

}

}

输出：

.button-ok {

background-image: url("ok.png");

}

.button-cancel {

background-image: url("cancel.png");

}

.button-custom {

background-image: url("custom.png");

}

**Multiple & （多个 &）**

&可以在一个选择器中出现不止一次。这就使得它可以反复引用父选择器，而不是重复父选择器的类名。

.link {

& + & {

color: red;

}

& & {

color: green;

}

&& {

color: blue;

}

&, &ish {

color: cyan;

}

}

将输出：

.link + .link {

color: red;

}

.link .link {

color: green;

}

.link.link {

color: blue;

}

.link, .linkish {

color: cyan;

}

注意，&代表所有的父选择器（而不只是最近的长辈），因此下面的例子：

.grand {

.parent {

& > & {

color: red;

}

& & {

color: green;

}

&& {

color: blue;

}

&, &ish {

color: cyan;

}

}

}

结果为：

.grand .parent > .grand .parent {

color: red;

}

.grand .parent .grand .parent {

color: green;

}

.grand .parent.grand .parent {

color: blue;

}

.grand .parent,

.grand .parentish {

color: cyan;

}

**Changing selector order （改变选择器顺序）**

要前置插入一个选择器给继承的(父)选择器时它是很有用的。用过将&放到当前选择器之后就可以做到这一点。

比如，使用Modernizr时，你可能希望基于要支持的特性来指定不同的规则：

.header {

.menu {

border-radius: 5px;

.no-borderradius & {

background-image: url('images/button-background.png');

}

}

}

选择器.no-borderradius &会前置插入.no-borderradius给它的父选择器.header .menu，最后变成.no-borderradius .header .menu形式输出：

.header .menu {

border-radius: 5px;

}

.no-borderradius .header .menu {

background-image: url('images/button-background.png');

}

**Combinatorial explosion**

&还可以用于生成一个逗号分割列表的所有可能的选择器排列：

p, a, ul, li {

border-top: 2px dotted #366;

& + & {

border-top: 0;

}

}

这个组合可以扩展出指定元素的所有（16种）可能的组合：

p,

a,

ul,

li {

border-top: 2px dotted #366;

}

p + p,

p + a,

p + ul,

p + li,

a + p,

a + a,

a + ul,

a + li,

ul + p,

ul + a,

ul + ul,

ul + li,

li + p,

li + a,

li + ul,

li + li {

border-top: 0;

}