

---

# LESS 安装

---

通过 npm 安装：



```
npm install less -g
```

不想安装到全局可以这样：



```
npm i less --save-dev
```

---

## LESS DEMO

---

### 变量 (VARIABLES)

无需多说，看代码一目了然：

```
@width: 10px;
@height: @width + 10px;

#header {
  width: @width;
  height: @height;
}
```

编译为：

```
#header {
  width: 10px;
  height: 20px;
}
```

## 混合 (MIXINS)

混合 (Mixin) 是一种将一组属性从一个规则集包含 (或混入) 到另一个规则集的方法。假设我们定义了一个类 (class) 如下：

```
.bordered {
  border-top: dotted 1px black;
  border-bottom: solid 2px black;
}
```

如果我們希望在其它规则集中使用这些属性呢？没问题，我们只需像下面这样输入所需属性的类 (class) 名称即可，如下所示：

```
#menu a {  
  color: #111;  
  .bordered();  
}  
  
.post a {  
  color: red;  
  .bordered();  
}
```

## 嵌套 (NESTING)

Less 提供了使用嵌套 (nesting) 代替层叠或与层叠结合使用的能力。假设我们有以下 CSS 代码：

```
#header {  
  color: black;  
}  
#header .navigation {  
  font-size: 12px;  
}  
#header .logo {  
  width: 300px;  
}
```

用 Less 语言我们可以这样书写代码：

```
#header {
  color: black;
  .navigation {
    font-size: 12px;
  }
  .logo {
    width: 300px;
  }
}
```

你还可以使用此方法将伪选择器（pseudo-selectors）与混合（mixins）一同使用。下面是一个经典的 clearfix 技巧，重写为一个混合（mixin）（& 表示当前选择器的父级）：

```
.clearfix {
  display: block;
  zoom: 1;

  &:after {
    content: " ";
    display: block;
    font-size: 0;
    height: 0;
    clear: both;
    visibility: hidden;
  }
}
```

## @规则嵌套和冒泡

@ 规则（例如 @media 或 @supports）可以与选择器以相同的方式进行嵌套。@ 规则会被放在前面，同一规则集中的其它元素的相对顺序保持不变。这叫做冒泡（bubbling）。



```
.component {  
  width: 300px;  
  @media (min-width: 768px) {  
    width: 600px;  
    @media (min-resolution: 192dpi) {  
      background-image: url(/img/retina2x.png);  
    }  
  }  
  @media (min-width: 1280px) {  
    width: 800px;  
  }  
}
```

编译为:



```
.component {  
  width: 300px;  
}  
@media (min-width: 768px) {  
  .component {  
    width: 600px;  
  }  
}  
@media (min-width: 768px) and (min-resolution: 192dpi) {  
  .component {  
    background-image: url(/img/retina2x.png);  
  }  
}  
@media (min-width: 1280px) {  
  .component {  
    width: 800px;  
  }  
}
```

# 运算 (OPERATIONS)

算术运算符 `+`、`-`、`*`、`/` 可以对任何数字、颜色或变量进行运算。

计算的结果以最左侧操作数的单位类型为准。如果单位换算无效或失去意义，则忽略单位。

无效的单位换算例如：px 到 cm 或 rad 到 % 的转换。

```
// 所有操作数被转换成相同的单位
@conversion-1: 5cm + 10mm; // 结果是 6cm
@conversion-2: 2 - 3cm - 5mm; // 结果是 -1.5cm

// conversion is impossible
@incompatible-units: 2 + 5px - 3cm; // 结果是 4px

// example with variables
@base: 5%;
@filler: @base * 2; // 结果是 10%
@other: @base + @filler; // 结果是 15%
```

乘法和除法不作转换。因为这两种运算在大多数情况下都没有意义，一个长度乘以一个长度就得到一个区域，而 CSS 是不支持指定区域的。Less 将按数字的原样进行操作，并将为计算结果指定明确的单位类型。

```
@base: 2cm * 3mm; // 结果是 6cm
```

你还可以对颜色进行算术运算：

```
@color: #224488 / 2; //结果是 #112244
background-color: #112244 + #111; // 结果是 #223355
```

**calc() 特例**

为了与 CSS 保持兼容，`calc()` 并不对数学表达式进行计算，但是在嵌套函数中会计算变量和数学公式的值。

```
@var: 50vh/2;  
width: calc(50% + (@var - 20px)); // 结果是 calc(50% + (25vh - 20px))
```

## 转义 (ESCAPING)

转义 (Escaping) 允许你使用任意字符串作为属性或变量值。任何 `~"anything"` 或 `~'anything'` 形式的内容都将按原样输出，除非 interpolation。

```
@min768: ~"(min-width: 768px)";  
.element {  
  @media @min768 {  
    font-size: 1.2rem;  
  }  
}
```

编译为：

```
@media (min-width: 768px) {  
  .element {  
    font-size: 1.2rem;  
  }  
}
```

注意，从 Less 3.5 开始，可以简写为：

```
@min768: (min-width: 768px);
.element {
  @media @min768 {
    font-size: 1.2rem;
  }
}
```

## 函数 (FUNCTIONS)

Less 内置了多种函数用于转换颜色、处理字符串、算术运算等。这些函数在 [Less 函数手册](#) 中有详细介绍。

函数的用法非常简单。下面这个例子将介绍如何利用 percentage 函数将 0.5 转换为 50%，将颜色饱和度增加 5%，以及颜色亮度降低 25% 并且色相值增加 8 等用法：

```
@base: #f04615;
@width: 0.5;

.class {
  width: percentage(@width); // returns `50%`
  color: saturate(@base, 5%);
  background-color: spin(lighten(@base, 25%), 8);
}
```

## 命名空间和访问符

(不要和 [CSS @namespace](#) 或 [namespace selectors](#) 混淆了)。

有时，出于组织结构或仅仅是为了提供一些封装的目的，你希望对混合 (mixins) 进行分组。你可以用 Less 更直观地实现这一需求。假设你希望将一些混合 (mixins) 和变量置于 `#bundle` 之下，为了以后方便重用或分发：



```
#bundle() {
  .button {
    display: block;
    border: 1px solid black;
    background-color: grey;
    &:hover {
      background-color: white;
    }
  }
  .tab { ... }
  .citation { ... }
}
```

现在，如果我们希望把 `.button` 类混合到 `#header a` 中，我们可以这样做：

```
#header a {
  color: orange;
  #bundle.button(); // 还可以书写为 #bundle > .button 形式
}
```

注意：如果不希望它们出现在输出的 CSS 中，例如 `#bundle .tab`，请将 `()` 附加到命名空间（例如 `#bundle()`）后面。

## 映射（MAPS）

从 Less 3.5 版本开始，你还可以将混合（mixins）和规则集（rulesets）作为一组值的映射（map）使用。

```
#colors() {
  primary: blue;
  secondary: green;
}

.button {
  color: #colors[primary];
  border: 1px solid #colors[secondary];
}
```

输出符合预期：

```
.button {
  color: blue;
  border: 1px solid green;
}
```

## 作用域（SCOPE）

Less 中的作用域与 CSS 中的作用域非常类似。首先在本地查找变量和混合（mixins），如果找不到，则从“父”级作用域继承。

```
@var: red;

#page {
  @var: white;
  #header {
    color: @var; // white
  }
}
```

与 CSS 自定义属性一样，混合（mixin）和变量的定义不必在引用之前事先定义。因此，下面的 Less 代码示例和上面的代码示例是相同的：

```
@var: red;

#page {
  #header {
    color: @var; // white
  }
  @var: white;
}
```

## 导入 (IMPORTING)

“导入”的工作方式和你的预期的一样。你可以导入一个 `.less` 文件，此文件中的所有变量就可以全部使用了。如果导入的文件是 `.less` 扩展名，则可以将扩展名省略掉：

```
@import "library"; // library.less
@import "typo.css";
```

---

## LESS 内置函数

---

# 逻辑函数(LOGICAL FUNCTIONS)

## Usage:



```
if((逻辑表达式), 真时的值, 假时的值)
```

## Examples:



```
@some:foo;  
div{  
    //判断如果2>1, 就是3px, 反之为0px  
    margin:if((2>1),3px,0px);  
    //判断@some是不是一种颜色, 是的话就是some,反之为black  
    color:if((iscolor(@some)),@some,black);  
}
```

## Result:



```
div{  
    margin:0;  
    color:black;  
}
```

同时也支持这样:

```

@bg:black;
@bg-light:boolean(luma(@bg) > 50%);
div{
  background:@bg;
  color: if(@bg-light,black,white)
}

```

Result:

```

div{
  background:black;
  color:white;
}

```

# 字符串函数(String Functions)

## 01. escape():

将输入字符串的url特殊字符进行编码处理。

未编码: , , , , , , , and . , ' / ' ? ' ' @ ' ' & ' ' + ' ' ' ' ~ ' ' ! ' ' \$ ;

转 义 的 编 码 : , , , , , , , , , , , , , , , and . \ <space> \> ' ' # ' ' ^ ' ' ( ' ' ) ' ' { ' ' } ' ' | ' ' : ' ' > ' ' < ' ' ; ' ' ] ' ' [ ' ' =

Example:

```

escape('a = 1')

```

Output:



```
a%3D1
```


## 02. e()

字符串转义，用~"值"符号代替。

参数：- 要转义的字符串。 `string`

返回：- 不带引号的转义字符串。 `string`

Example:



```
@mscode: "ms:alwaysHasItsOwnSyntax.For.Stuff()"
filter: e(@mscode);
```

Result:



```
filter: ms:alwaysHasItsOwnSyntax.For.Stuff();
```

## 03. %()

函数%(string,arguments...) 格式化一个字符串。

参数：

`string`:占位符的格式字符串

`anything`:用于替占位符的值

该函数设置字符串的格式。 `%(string, arguments ...)`

第一个参数是带占位符的字符串。所有占位符都以百分比符号开头，后跟字母 `,` `,` `,` `,` 或 `.`。其余参数包含用于替换占位符的表达式。如果需要打印百分比符号，请将其转义为 另一个百分比。 `%`s`S`d`D`a`A`%%`

小写：占位符

大写：使用 `utf-8` 编码转义。

Example:

```
format-a-d: %("repetitions: %a file: %d", 1 + 2,
"directory/file.less");
format-a-d-upper: %('repetitions: %A file: %D', 1 + 2,
"directory/file.less");
format-s: %("repetitions: %s file: %s", 1 + 2,
"directory/file.less");
format-s-upper: %('repetitions: %S file: %S', 1 + 2,
"directory/file.less");
```

Result:

```
format-a-d: "repetitions: 3 file: "directory/file.less"";
format-a-d-upper: "repetitions: 3 file:
%22directory%2Ffile.less%22";
format-s: "repetitions: 3 file: directory/file.less";
format-s-upper: "repetitions: 3 file: directory%2Ffile.less";
```

#### 04. replace()

```
replace(string, pattern, replacement, flags(可选))
```

参数:

**string**: 要搜索的字符串。

**pattern**: 要搜索的字符串或者正则表达式。

**replacement**: 搜索字符串模式'g'。

**flags**: (可选) 正则表达式标志。

返回:

包含替换值的字符串。

Example:



```
replace("Hello, Mars?", "Mars\\?", "Earth!");  
replace("One + one = 4", "one", "2", "gi");  
replace('This is a string.', "(string)\\.\"", "new $1.");  
replace(~"bar-1", '1', '2');
```

Result:



```
"Hello, Earth!";  
"2 + 2 = 4";  
'This is a new string.';  
bar-2;
```

# 列表函数(LIST FUNCTIONS)

## 01. length()

返回集合中值的个数。

Example:



```
length(1px solid #0080ff)
```

Output:



```
3
```

Example:





```
@list: "banana", "tomato", "potato", "peach";  
n: length(@list);
```

Output:



```
n: 4;
```

## 02. `extract()`

`extract(list, index)`

返回数组中指定索引的值;

参数:

`list`: 数组

`index`: 索引

Example:



```
@list: apple, pear, coconut, orange;  
value: extract(@list, 3);
```

Output:



```
value: coconut;
```

## 03. `range()`

生成一个跨越一系列值得列表。

`range(start, end, step)`

参数:

`start` : 起始值。

`end` : 结束值。

`step` : 每次递增的数字。

Example:



```
value: range(4);
```

Output:



```
value: 1, 2, 3, 4
```

Example:



```
value: range(10px, 30px, 10);
```

Output:



```
value: 10px, 20px, 30px;
```

#### 04. `each()`

将规则集的计算绑定到列表的每个成员。

```
each(list, rules)
```

参数:

`list` : 集合

`rules` : 匿名规则集

Example:

```
@selectors: blue, green, red;
each(@selectors, {
    .sel-@{value} {
        color: b;
    }
})
```

Output:

```
.sel-blue {
    color: b;
}
.sel-green {
    color: b;
}
.sel-red {
    color: b;
}
```

缺省情况下，每个规则集都按列表成员绑定到 `、` 和变量。对于大多数列表，将分配相同的值（数字位置，从 1 开始）。但是，您也可以将规则集 本身 用作结构化列表。

如： `@value``@key``@index``@key``@index`

```
@set: {
    one: blue;
    two: green;
    three: red;
}
.set {
    each(@set, {
        @{key}-@{index}: @value;
    });
}
```

这将输出：

```
.set {  
  one-1: blue;  
  two-2: green;  
  three-3: red;  
}
```

当然，由于您可以为每个规则集调用使用保护来调用 mixin，因此这具有非常强大的功能。 `each()`

匿名 mixin 使用的形式或以常规 mixin 开头或就像常规 mixin 一样。在 中，您可以按如下方式使用它： `#()``.()``.``#``each()`

```
.set-2() {  
  one: blue;  
  two: green;  
  three: red;  
}  
.set-2 {  
  // Call mixin and iterate each rule  
  each(.set-2(), .(@v, @k, @i) {  
    @{k}-@{i}: @v;  
  });  
}
```

这将按预期输出：

```
.set-2 {  
  one-1: blue;  
  two-2: green;  
  three-3: red;  
}
```

## 05. 使用、创建循环 `for range each`

Example:

```
each(range(4), {  
  .col-@{value} {  
    height: (@value * 50px);  
  }  
});
```

Output:

```
.col-1 {  
  height: 50px;  
}  
.col-2 {  
  height: 100px;  
}  
.col-3 {  
  height: 150px;  
}  
.col-4 {  
  height: 200px;  
}
```

# 数学函数(MATH FUNCTIONS)

## 01. `ceil()`

向上取整。

Example:

```
width: ceil(2.4px);
```

Output:



```
width: 3px;
```

## 02. `floor()`

向下取整。

Example:



```
height: floor(2.99999px);
```

Output:




```
height: 2px;
```

## 03. `percentage()`

将浮点数转换为百分比。

Example:



```
width: percentage(0.5);
```

Output:



```
width: 50%;
```

## 04. `round()`

四舍五入和保留小数点。

参数：

`number`：浮点数；

`decimalPlaces`：可选参数，要保留的小数点位数。

Example:



```
round(1.67);  
round(1.67, 1)
```

Output:



```
2;  
1.7;
```

## 05. `sqrt()`

计算数字的平方根，保持单位不变。

Example:



```
sqrt(25cm);  
sqrt(18.6%)
```

Output:



```
5cm;  
4.312771730569565%;
```

## 06. `abs()`

计算数字的绝对值，保持单位不变。

Example:

```
abs(25cm);  
abs(-18.6%);
```

Output:

```
25cm;  
18.6%;
```

## 07. `sin()`

计算正弦函数。

Example:

```
width: sin(1);  
// 1弧度角的正弦值  
width: sin(1deg);  
// 1角度角的正弦值  
width: sin(1grad);  
// 1百分度角的正弦值
```

Output:

```
0.8414709848078965; // sine of 1 radian  
0.01745240643728351; // sine of 1 degree  
0.015707317311820675; // sine of 1 gradian
```

## 08. `asin()`

反正弦函数。



Example:

```
width: asin(-0.84147098);  
width: asin(0);  
width: asin(2);
```

Output:

```
-1rad  
0rad  
NaNrad
```

## 09. `cos()`

余弦函数。

Example:

```
cos(1) // cosine of 1 radian  
cos(1deg) // cosine of 1 degree  
cos(1grad) // cosine of 1 gradian
```

Output:

```
0.5403023058681398 // cosine of 1 radian  
0.9998476951563913 // cosine of 1 degree  
0.9998766324816606 // cosine of 1 gradian
```

## 10. `acos()`

反余弦函数。

Example:



```
acos(0.5403023058681398)
acos(1)
acos(2)
```

Output:



```
1rad
0rad
NaNrad
```

## 11. `tan()`

正切函数。

Example:



```
tan(1) // tangent of 1 radian
tan(1deg) // tangent of 1 degree
tan(1grad) // tangent of 1 gradian
```

Output:



```
1.5574077246549023 // tangent of 1 radian
0.017455064928217585 // tangent of 1 degree
0.015709255323664916 // tangent of 1 gradian
```

## 12. `atan()`

Example:



```
atan(-1.5574077246549023)
atan(0)
round(atan(22), 6) // arctangent of 22 rounded to 6 decimal places
```

Output:



```
-1rad
0rad
1.525373rad;
```

### 13. `pi()`

返回 $\pi$ (pi)。

Example:



```
pi()
```

Output:



```
3.141592653589793
```

### 14. `pow()`

乘方运算。

参数:

`number`: 要乘方的数字;

`number`: 乘方的倍数。

Example:



```
pow(0cm, 0px)
pow(25, -2)
pow(25, 0.5)
pow(-25, 0.5)
pow(-25%, -0.5)
```

Output:



```
1cm
0.0016
5
NaN
NaN%
```

#### 15. `mod()`

取余操作。

Example:



```
width: mod(3px, 2);
```

Output:



```
1px;
```

#### 16. `min()`

最小值计算。

Example:



```
width: min(3px, 2px, 6px);
```

Output:



```
width: 2px;
```

## 17. `max()`

最大值运算。

Example:



```
width: max(3px, 2px, 6px);
```

Output:




```
width: 6px;
```

# 类型函数(TYPE FUNCTIONS)

## 01. `isnumber()`

判断参数是否是数字，是返回true，反之false。

Example:




```
isnumber(#ff0);    // false
isnumber(blue);    // false
isnumber("string"); // false
isnumber(1234);    // true
isnumber(56px);    // true
isnumber(7.8%);    // true
isnumber(keyword); // false
isnumber(url( ... )); // false
```

## 02. `isstring()`

判断是否是字符串。

Example:




```
isstring(#ff0);    // false
isstring(blue);    // false
isstring("string"); // true
isstring(1234);    // false
isstring(56px);    // false
isstring(7.8%);    // false
isstring(keyword); // false
isstring(url( ... )); // false
```

## 03. `iscolor()`

判断是否是颜色。

Example:



```
iscolor(#ff0);    // true
iscolor(blue);    // true
iscolor("string"); // false
iscolor(1234);    // false
iscolor(56px);    // false
iscolor(7.8%);    // false
iscolor(keyword); // false
iscolor(url( ... )); // false
```

#### 04. `iskeyword()`

判断是否是关键字。

Example:

```
iskeyword(#ff0);    // false
iskeyword(blue);    // false
iskeyword("string"); // false
iskeyword(1234);    // false
iskeyword(56px);    // false
iskeyword(7.8%);    // false
iskeyword(keyword); // true
iskeyword(url( ... )); // false
```

```
//如果一个值是一个关键字, 返回'真(true)', 否则返回'假(false)'.
.m(@x) when (iskeyword(@x)) {
    x:@x
}
div{
    .m(123);
    .m("ABC");
    .m(red);
    .m(ABC); //x:ABC
}
```

#### 05. `isurl()`

判断是否是url地址。

Example:

```

isurl(#ff0);      // false
isurl(blue);      // false
isurl("string");  // false
isurl(1234);      // false
isurl(56px);      // false
isurl(7.8%);      // false
isurl(keyword);   // false
isurl(url( ... )); // true

```

```

//如果一个值是一个url地址, 返回'真(true)', 否则返回'假(false)'.
.m(@x) when (isurl(@x)) {
    x:@x
}
div{
    .m(ABC);
    .m(url(arr.jpg)); //x:url(arr.jpg)
}

```

## 06. ispixel()

如果值是以 px 为单位的数字, 返回true, 反之为false。

Example:

```

ispixel(#ff0);    // false
ispixel(blue);    // false
ispixel("string"); // false
ispixel(1234);    // false
ispixel(56px);    // true
ispixel(7.8%);    // false
ispixel(keyword); // false
ispixel(url( ... )); // false

```

## 07. isem()

如果值是以 em 为单位的数字, 返回true, 反之为false。

Example:



```
isem(#ff0);      // false
isem(blue);      // false
isem("string");  // false
isem(1234);      // false
isem(56px);      // false
isem(7.8em);     // true
isem(keyword);   // false
isem(url( ... )); // false
```

## 08. ispercentage()

如果值是以 % 为单位的数字，返回true，反之为false。

Example:

```
ispercentage(#ff0);      // false
ispercentage(blue);      // false
ispercentage("string");  // false
ispercentage(1234);      // false
ispercentage(56px);      // false
ispercentage(7.8%);      // true
ispercentage(keyword);   // false
ispercentage(url( ... )); // false
```

## 09. isunit()

如果值是带指定单位的数字，返回true，否则返回false。

如果第一个参数的单位是第二个参数就返回true。

参数:

**value**: 要判断的数字;

**unit**: 要查找的单位。

Example:

```
isunit(11px, px); // true
isunit(2.2%, px); // false
isunit(33px, rem); // false
isunit(4rem, rem); // true
isunit(56px, "%"); // false
isunit(7.8%, '%'); // true
isunit(1234, em); // false
isunit(#ff0, pt); // false
isunit("mm", mm); // false
```

## 10. isruleset()

如果参数是规则集，就是true，反之为false

Example:

```
@rules: {
  color: red;
}

isruleset(@rules); // true
isruleset(#ff0); // false
isruleset(blue); // false
isruleset("string"); // false
isruleset(1234); // false
isruleset(56px); // false
isruleset(7.8%); // false
isruleset(keyword); // false
isruleset(url(...)); // false
```

# 颜色值定义函数

## 01. rgb()

通过十进制红、绿、蓝（RGB）创建不透明的颜色对象。

参数：

`red` : 0-255 的整数或 0-100% 百分数;

`green` : 0-255 的整数或 0-100% 百分数;

`blue` : 0-255 的整数或 0-100% 百分数;

Example: `rgb(90, 129, 32)`

Output: `#5a8120`

## 02. `rgba()`

通过十进制红、绿、蓝 (RGB) , 以及alpha四种值 (RGBA) 创建带alpha透明的颜色对象

参数:

`red` : 0-255 的整数或 0-100% 百分数;

`green` : 0-255 的整数或 0-100% 百分数;

`blue` : 0-255 的整数或 0-100% 百分数;

`alpha` : 0-1的分数或 0-100% 百分数;

Example: `rgba(90, 129, 32, 0.5)`

Output: `rgba(90, 129, 32, 0.5)`

## 03. `argb()`

创建格式为#AARRGGBB的十六进制颜色 , 用于IE滤镜, .net和安卓开发

Example: `argb(rgba(90, 23, 148, 0.5));`

Output: `#805a1794`

## 04. `hls()`

通过色相, 饱和度, 亮度 (HLS) 三种值创建不透明的颜色对象。

Example: `hsl(90, 100%, 50%)`

Output: `#80ff00`

## 05. `hsla()`

通过色相, 饱和度, 亮度, 以及alpha四种值 (HLSA) 创建带alpha透明的颜色对象.

Example: `hsla(90, 100%, 50%, 0.5)`

Output: `rgba(128, 255, 0, 0.5)`

#### 06. `hsv()`

通过色相，饱和度，色调（HSV）创建不透明的颜色对象。

Example: `hsv(90, 100%, 50%)`

Output: `#408000`

#### 07. `hsva()`

通过色相，饱和度，亮度，以及alpha四种值（HSVA）创建带alpha透明的颜色对象

Example: `hsva(90, 100%, 50%, 0.5)`

Output: `rgba(64, 128, 0, 0.5)`

## 颜色值通道提取函数

#### 01. `hue()`

从HSL色彩空间中提取色相值。

Example: `hue(hsl(90, 100%, 50%))`

Output: `90`

#### 02. `saturation()`

从HSL色彩空间中提取饱和度。

Example: `saturation(hsl(90, 100%, 50%))`

Output: `100%`

#### 03. `lightness()`

从HSL色彩空间中提取亮度值。

Example: `lightness(hsl(90, 100%, 50%))`

Output: `50%`

#### 04. `hsvhue()`

从HSV色彩空间中提取色相值。

Example: `hsvhue(hsv(90, 100%, 50%))`

Output: `90`

#### 05. `hsvsaturation()`

从HSV色彩空间中提取饱和度值。

Example: `hsvsaturation(hsv(90, 100%, 50%))`

Output: `100%`

#### 06. `hsvvalue()`

从HSV色彩空间中提取色调值。

Example: `hsvvalue(hsv(90, 100%, 50%))`

Output: `50%`

#### 07. `red()`

提取颜色对象的红色值。

Example: `red(rgb(10, 20, 30))`

Output: `10`

#### 08. `green()`

提取颜色对象的绿色值。

Example: `green(rgb(10, 20, 30))`

Output: `20`

#### 09. `blue()`

提取颜色对象的蓝色值。

Example: `blue(rgb(10, 20, 30))`

Output: `30`

#### 10. `alpha()`

提取颜色对象的透明度。

Example: `alpha(rgba(10, 20, 30, 0.5))`

Output: 0.5

#### 11. luma()

计算颜色对象luma的值（亮度的百分比表示法）

Example: `luma(rgb(100, 200, 30))`

Output: 44%

#### 12. luminance()

计算没有伽玛校正的亮度值。

Example: `luminance(rgb(100, 200, 30))`

Output: 65%

## 颜色操作函数

#### 01. saturate()

增加一定数值的颜色饱和度。

Example: `saturate(hsl(90, 80%, 50%), 20%)`

Output: `#80ff00 // hsl(90, 100%, 50%)`

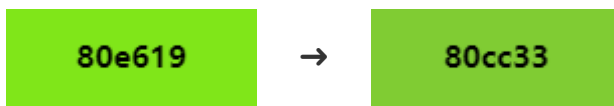


#### 02. desaturate()

降低一定数值的颜色饱和度。

Example: `desaturate(hsl(90, 80%, 50%), 20%)`

Output: `#80cc33 // hsl(90, 60%, 50%)`



### 03. `lighten()`

增加一定数值的颜色亮度。

Example: `lighten(hsl(90, 80%, 50%), 20%)`

Output: `#b3f075 // hsl(90, 80%, 70%)`



### 04. `darken()`

降低一定数值的颜色亮度。

Example: `darken(hsl(90, 80%, 50%), 20%)`

Output: `#4d8a0f // hsl(90, 80%, 30%)`



### 05. `fadein()`

降低颜色的透明度（或增加不透明度），令其更不透明。

Example: `fadein(hsla(90, 90%, 50%, 0.5), 10%)`

Output: `rgba(128, 242, 13, 0.6) // hsla(90, 90%, 50%, 0.6)`

### 06. `fadeout()`

增加颜色的透明度（或降低不透明度），令其更透明。

Example: `fadeout(hsla(90, 90%, 50%, 0.5), 10%)`

Output: `rgba(128, 242, 13, 0.4) // hsla(90, 90%, 50%, 0.4)`

## 07. `fade()`

给颜色（包括不透明的颜色）设定一定数值的透明度。

Example: `fade(hsl(90, 90%, 50%), 10%)`

Output: `rgba(128, 242, 13, 0.1) //hsla(90, 90%, 50%, 0.1)`

## 08. `spin()`

任意方向旋转颜色的色相角度。

Example:



```
spin(hsl(10, 90%, 50%), 30)
spin(hsl(10, 90%, 50%), -30)
```

Output:



```
#f2a60d // hsl(40, 90%, 50%)
#f20d59 // hsl(340, 90%, 50%)
```



## 09. `mix()`

根据比例混合两种颜色，包括计算不透明度。

Example:



```
mix(#ff0000, #0000ff, 50%)
mix(rgba(100,0,0,1.0), rgba(0,100,0,0.5), 50%)
```

Output:

```
#800080
rgba(75, 25, 0, 0.75)
```



#### 10. `greyscale()`

完全移除颜色的饱和度，与 `desaturate (@color, 100%)` 函数效果相同。

Example: `greyscale(hsl(90, 90%, 50%))`

Output: `#808080 // hsl(90, 0%, 50%)`



#### 11. `contrast()`

旋转两种颜色相比较，得出哪种颜色的对比度更大就倾向于对比度最大的颜色。

Example:

```
p {
  a: contrast(#bbbbbb);
  b: contrast(#222222, #101010);
  c: contrast(#222222, #101010, #dddddd);
  d: contrast(hsl(90, 100%, 50%), #000000, #ffffff, 30%);
  e: contrast(hsl(90, 100%, 50%), #000000, #ffffff, 80%);
}
```

Output:

```
p {
  a: #000000 // black
  b: #ffffff // white
  c: #dddddd
  d: #000000 // black
  e: #ffffff // white
}
```

## 颜色混合函数

### 01. multiply()

分别将两种颜色的红绿蓝三数值做乘法运算，然后再除以255，输出结果更深的颜色（对应ps中的“变暗/正片叠底”）。

**Examples :**

```
multiply(#ff6600, #000000);
```

ff6600

000000

000000



```
multiply(#ff6600, #333333);
```

ff6600

333333

331400



```
multiply(#ff6600, #666666);
```

ff6600

666666

662900



```
multiply(#ff6600, #999999);
```

ff6600

999999

993d00



```
multiply(#ff6600, #cccccc);
```

ff6600

cccccc

cc5200



```
multiply(#ff6600, #ffffff);
```

ff6600

ffffff

ff6600



```
multiply(#ff6600, #ff0000);
```

ff6600

ff0000

ff0000



```
multiply(#ff6600, #00ff00);
```

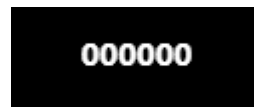
ff6600

00ff00

006600



```
multiply(#ff6600, #0000ff);
```



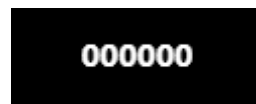
## 02. `screen()`

与multiply函数效果相反，输出结果更亮的颜色。（对应ps中“变亮/滤色”）

Example:



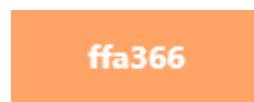
```
screen(#ff6600, #000000);
```



```
screen(#ff6600, #333333);
```



```
screen(#ff6600, #666666);
```





```
screen(#ff6600, #999999);
```

**ff6600**

**999999**

**ffc299**



```
screen(#ff6600, #cccccc);
```

**ff6600**

**cccccc**

**ffe0cc**



```
screen(#ff6600, #ffffff);
```

**ff6600**

**ffffff**

**ffffff**



```
screen(#ff6600, #ff0000);
```

**ff6600**

**ff0000**

**ff6600**



```
screen(#ff6600, #00ff00);
```



```
screen(#ff6600, #0000ff);
```



### 03. overlay()

结合multiply与screen两个函数的效果，令浅的颜色更浅，深的颜色更深（对应ps中的叠加），输出结果由第一个颜色参数决定。

Example:



```
overlay(#ff6600, #000000);
```



```
overlay(#ff6600, #333333);
```

ff6600

333333

ff2900



```
overlay(#ff6600, #666666);
```

ff6600

666666

ff5200



```
overlay(#ff6600, #999999);
```

ff6600

999999

ff7a00



```
overlay(#ff6600, #cccccc);
```

ff6600

cccccc

ffa300



```
overlay(#ff6600, #ffffff);
```





```
overlay(#ff6600, #ff0000);
```



```
overlay(#ff6600, #00ff00);
```



```
overlay(#ff6600, #0000ff);
```



#### 04. softlight()

与overlay函数效果相似，只是当纯黑色或纯白色作为参数时输出结果不会是纯黑色或纯白色（对应ps中的“柔光”）

Example:



```
softlight(#ff6600, #000000);
```

**ff6600**

**000000**

**ff2900**



```
softlight(#ff6600, #333333);
```

**ff6600**

**333333**

**ff4100**



```
softlight(#ff6600, #666666);
```

**ff6600**

**666666**

**ff5a00**



```
softlight(#ff6600, #999999);
```

ff6600

999999

ff7200



```
softlight(#ff6600, #cccccc);
```

ff6600

cccccc

ff8a00



```
softlight(#ff6600, #ffffff);
```

ff6600

ffffff

ffa100



```
softlight(#ff6600, #ff0000);
```

ff6600

ff0000

ff2900



```
softlight(#ff6600, #00ff00);
```



**ff6600**



**00ff00**



**ffa100**



```
softlight(#ff6600, #0000ff);
```



**ff6600**



**0000ff**



**ff2900**

## 05. `hardlight()`

与`overlay`函数效果相似，不过由第二个颜色参数决定输出颜色的亮度或黑度，而不是第一个颜色参数决定（对应ps中“强光/亮光/线性光/点光”）

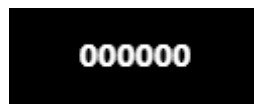
Example:



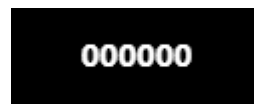
```
hardlight(#ff6600, #000000);
```



**ff6600**



**000000**



**000000**



```
hardlight(#ff6600, #333333);
```

ff6600

333333

662900



```
hardlight(#ff6600, #666666);
```

ff6600

666666

cc5200



```
hardlight(#ff6600, #999999);
```

ff6600

999999

ff8533



```
hardlight(#ff6600, #cccccc);
```

ff6600

cccccc

ffc299



```
hardlight(#ff6600, #ffffff);
```



ffffff

ffffff



```
hardlight(#ff6600, #ff0000);
```



```
hardlight(#ff6600, #00ff00);
```



```
hardlight(#ff6600, #0000ff);
```



## 06. exclusion()

效果与difference函数效果相似，只是输出结果差别更小（对应ps中“差值/排除”）

Example:



```
exclusion(#ff6600, #000000);
```

**ff6600**

**000000**

**ff6600**



```
exclusion(#ff6600, #333333);
```

**ff6600**

**333333**

**cc7033**



```
exclusion(#ff6600, #666666);
```

**ff6600**

**666666**

**997a66**



```
exclusion(#ff6600, #999999);
```

**ff6600**

**999999**

**668599**



```
exclusion(#ff6600, #cccccc);
```

ff6600

cccccc

338fcc



```
exclusion(#ff6600, #ffffff);
```

ff6600

ffffff

0099ff



```
exclusion(#ff6600, #ff0000);
```

ff6600

ff0000

006600



```
exclusion(#ff6600, #00ff00);
```

ff6600

00ff00

ff9900





```
exclusion(#ff6600, #0000ff);
```



**ff6600**



**0000ff**



**ff66ff**

## 07. average()

分别对RGB三个颜色值取平均值，然后输出结果。

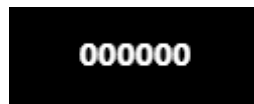
Example:



```
average(#ff6600, #000000);
```



**ff6600**



**000000**



**803300**



```
average(#ff6600, #333333);
```



**ff6600**



**333333**



**994d1a**



```
average(#ff6600, #666666);
```

**ff6600**

**666666**

**b36633**



```
average(#ff6600, #999999);
```

**ff6600**

**999999**

**cc804d**



```
average(#ff6600, #cccccc);
```

**ff6600**

**cccccc**

**e69966**



```
average(#ff6600, #ffffff);
```

**ff6600**

**ffffff**

**ffb380**



```
average(#ff6600, #ff0000);
```



```
average(#ff6600, #00ff00);
```



```
average(#ff6600, #0000ff);
```



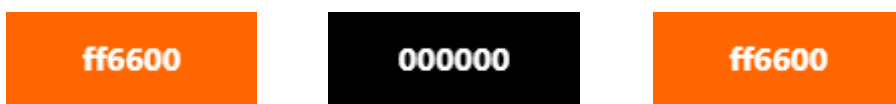
## 08. `negation()`

与difference函数效果相反，输出结果是更亮的颜色。（效果相反不代表做加法运算）

Example:



```
negation(#ff6600, #000000);
```





```
negation(#ff6600, #333333);
```

ff6600

333333

cc9933



```
negation(#ff6600, #666666);
```

ff6600

666666

99cc66



```
negation(#ff6600, #999999);
```

ff6600

999999

66ff99



```
negation(#ff6600, #cccccc);
```

ff6600

cccccc

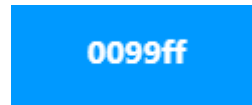
33cccc



```
negation(#ff6600, #ffffff);
```



ffffff



```
negation(#ff6600, #ff0000);
```



```
negation(#ff6600, #00ff00);
```



```
negation(#ff6600, #0000ff);
```



