CSS单位	
相对长度单位	em、ex、ch、rem、vw、vh、vmin、vmax、%
绝对长度单位	cm、mm、in、px、pt、pc

这里我们主要讲述px、em、rem、vh、vw

16.2.1. px

px，表示像素，所谓像素就是呈现在我们显示器上的一个个小点，每个像素点都是大小等同的，所以像素为计量单位被分在了绝对长度单位中

有些人会把 `px` 认为是相对长度，原因在于在移动端中存在设备像素比， `px` 实际显示的大小是不确定的

这里之所以认为 `px` 为绝对单位，在于 `px` 的大小和元素的其他属性无关

16.2.2. em

em是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸（ `1em = 16px` ）

为了简化 `font-size` 的换算，我们需要在 `css` 中的 `body` 选择器中声明 `font-size` = `62.5%` ，这就使 em 值变为 `16px*62.5% = 10px`

这样 `12px = 1.2em` ， `10px = 1em` , 也就是说只需要将你的原来的 `px` 数值除以 10，然后换上 `em` 作为单位就行了

特点:

- em 的值并不是固定的
- em 会继承父级元素的字体大小
- em 是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸
- 任意浏览器的默认字体高都是 16px

举个例子

```HTML
<div class="big">
    我是14px=1.4rem<div class="small">我是12px=1.2rem</div>
</div>
```

样式为

```CSS
<style>
    html {font-size: 10px;  } /*  公式16px*62.5%=10px  */
    .big{font-size: 1.4rem}
    .small{font-size: 1.2rem}
</style>
```

这时候 `.big` 元素的 `font-size` 为14px，而 `.small` 元素的 `font-size` 为12px

16.2.3. rem

rem，相对单位，相对的只是HTML根元素 `font-size` 的值

同理，如果想要简化 `font-size` 的转化，我们可以在根元素 `html` 中加入 `font-size: 62.5%`

```CSS
html {font-size: 62.5%;  } /*  公式16px*62.5%=10px  */
```

这样页面中1rem=10px、1.2rem=12px、1.4rem=14px、1.6rem=16px;使得视觉、使用、书写都得到了极大的帮助

特点：

- rem单位可谓集相对大小和绝对大小的优点于一身
- 和em不同的是rem总是相对于根元素，而不像em一样使用级联的方式来计算尺寸

16.2.4. vh、vw

vw，就是根据窗口的宽度，分成100等份，100vw就表示满宽，50vw就表示一半宽。（vw 始终是针对窗口的宽），同理，`vh` 则为窗口的高度

这里的窗口分成几种情况：

- 在桌面端，指的是浏览器的可视区域

- 移动端指的就是布局视口

像 `vw` 、 `vh` ，比较容易混淆的一个单位是 `%` ，不过百分比宽泛的讲是相对于父元素：

- 对于普通定位元素就是我们理解的父元素
- 对于position: absolute;的元素是相对于已定位的父元素
- 对于position: fixed;的元素是相对于 ViewPort（可视窗口）

16.3. 总结

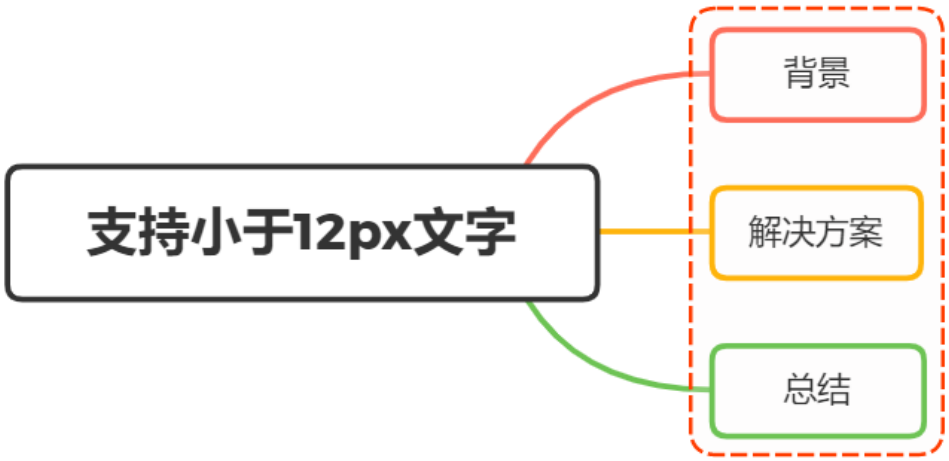
px：绝对单位，页面按精确像素展示

em：相对单位，基准点为父节点字体的大小，如果自身定义了 `font-size` 按自身来计算，整个页面内 `1em` 不是一个固定的值

rem：相对单位，可理解为 `root em` , 相对根节点 `html` 的字体大小来计算

vh、vw：主要用于页面视口大小布局，在页面布局上更加方便简单

17. 让Chrome支持小于12px 的文字方式有哪些？区别?



17.1. 背景

Chrome 中文版浏览器会默认设定页面的最小字号是12px，英文版没有限制

原由 Chrome 团队认为汉字小于12px就会增加识别难度

- 中文版浏览器

与网页语言无关，取决于用户在Chrome的设置里（chrome://settings/languages）把哪种语言设置为默认显示语言

- 系统级最小字号

浏览器默认设定页面的最小字号，用户可以前往 chrome://settings/fonts 根据需求更改

而我们在实际项目中，不能奢求用户更改浏览器设置

对于文本需要以更小的字号来显示，就需要用到一些小技巧

17.2. 解决方案

常见的解决方案有：

- zoom
- –webkit–transform:scale()
- –webkit–text–size–adjust:none

17.2.1. Zoom

`zoom` 的字面意思是“变焦”，可以改变页面上元素的尺寸，属于真实尺寸

其支持的值类型有：

- zoom:50%，表示缩小到原来的一半
- zoom:0.5，表示缩小到原来的一半

使用 `zoom` 来”支持“ 12px 以下的字体

代码如下：

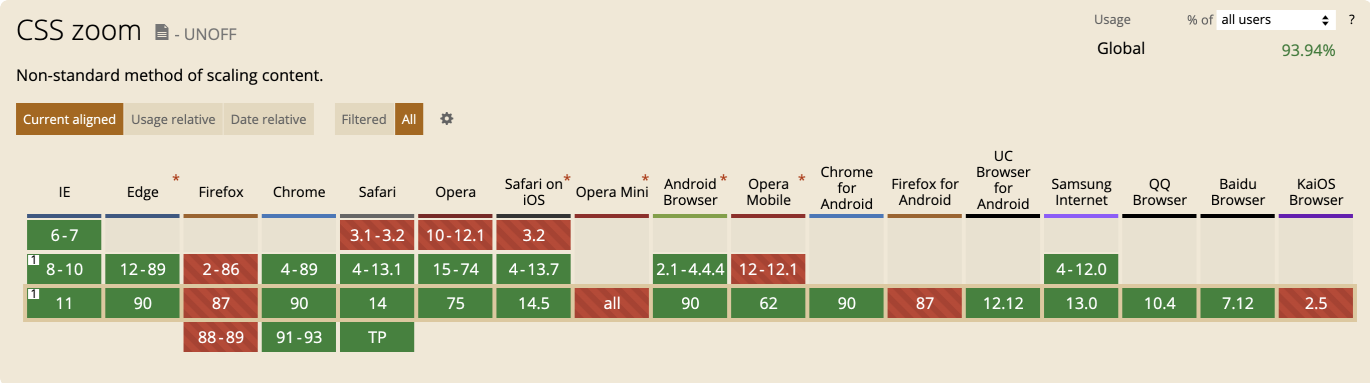
HTML | 复制代码

```html
<style type="text/css">
    .span1{
        font-size: 12px;
        display: inline-block;
        zoom: 0.8;
    }
    .span2{
        display: inline-block;
        font-size: 12px;
    }
</style>
<body>
    <span class="span1">测试10px</span>
    <span class="span2">测试12px</span>
</body>
```

效果如下：

测试10px 测试12px

> 需要注意的是，`Zoom` 并不是标准属性，需要考虑其兼容性



17.2.2. –webkit–transform:scale()

针对 `chrome` 浏览器,加 `webkit` 前缀，用 `transform:scale()` 这个属性进行放缩

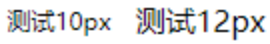
注意的是，使用 `scale` 属性只对可以定义宽高的元素生效，所以，下面代码中将 `span` 元素转为行内块元素

实现代码如下：

HTML | 复制代码

```html
<style type="text/css">
    .span1{
        font-size: 12px;
        display: inline-block;
        -webkit-transform:scale(0.8);
    }
    .span2{
        display: inline-block;
        font-size: 12px;
    }
</style>
<body>
    <span class="span1">测试10px</span>
    <span class="span2">测试12px</span>
</body>
```

效果如下：



17.2.3. -webkit-text-size-adjust:none

该属性用来设定文字大小是否根据设备(浏览器)来自动调整显示大小

属性值：

- percentage：字体显示的大小；
- auto：默认，字体大小会根据设备/浏览器来自动调整；
- none:字体大小不会自动调整

CSS | 复制代码

```css
html { -webkit-text-size-adjust: none; }
```

这样设置之后会有一个问题，就是当你放大网页时，一般情况下字体也会随着变大，而设置了以上代码后，字体只会显示你当前设置的字体大小，不会随着网页放大而变大了

所以，我们不建议全局应用该属性，而是单独对某一属性使用

> 需要注意的是，自从 `chrome 27` 之后，就取消了对这个属性的支持。同时，该属性只对英文、数字生效，对中文不生效

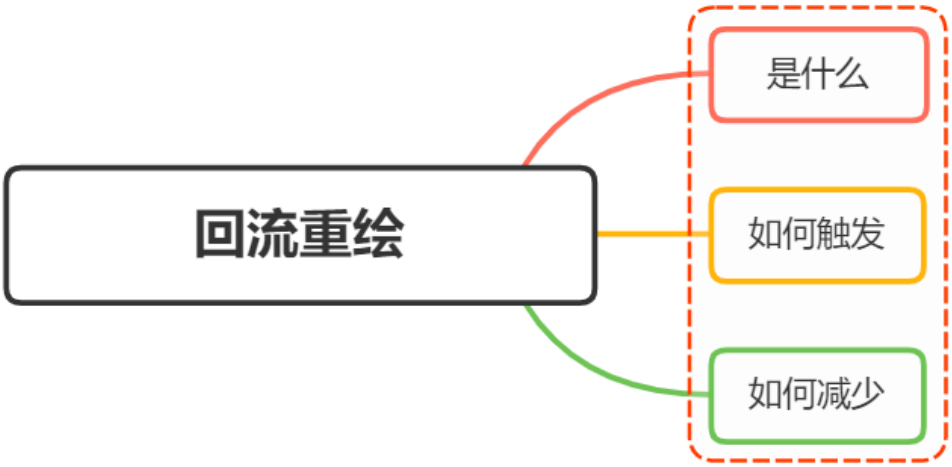
17.3. 总结

`Zoom` 非标属性，有兼容问题，缩放会改变了元素占据的空间大小，触发重排

`-webkit-transform:scale()` 大部分现代浏览器支持，并且对英文、数字、中文也能够生效，缩放不会改变了元素占据的空间大小，页面布局不会发生变化

`-webkit-text-size-adjust` 对谷歌浏览器有版本要求，在27之后，就取消了该属性的支持，并且只对英文、数字生效

18. 怎么理解回流跟重绘？什么场景下会触发？

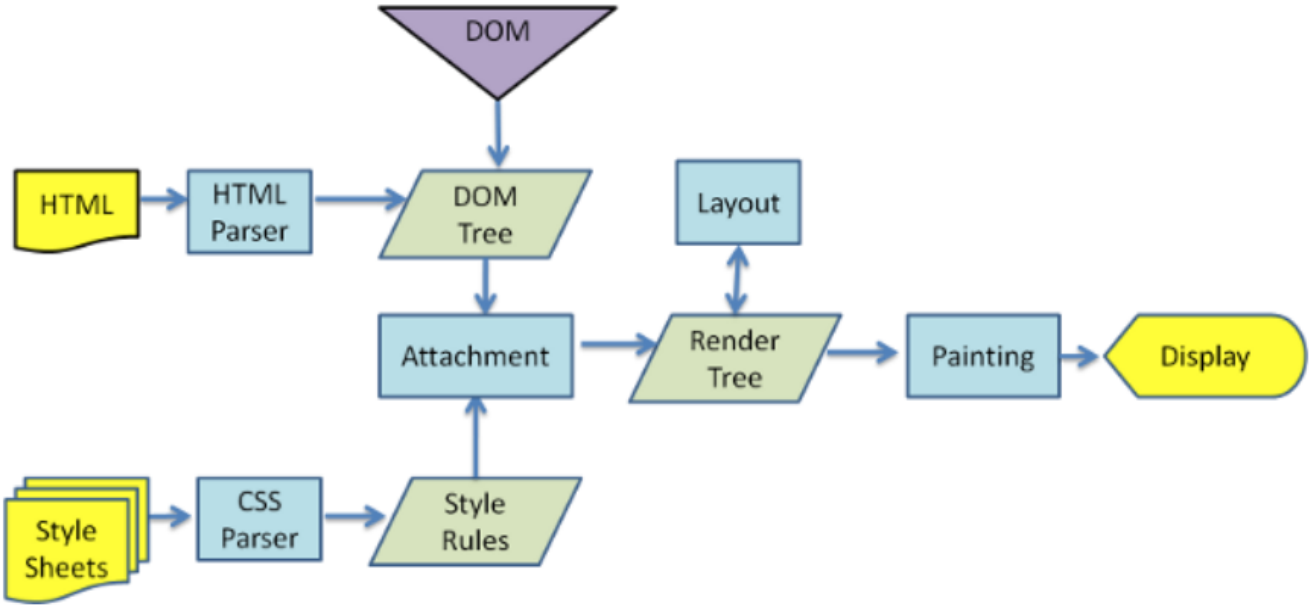


18.1. 是什么

在 `HTML` 中，每个元素都可以理解成一个盒子，在浏览器解析过程中，会涉及到回流与重绘：

- 回流：布局引擎会根据各种样式计算每个盒子在页面上的大小与位置
- 重绘：当计算好盒模型的位置、大小及其他属性后，浏览器根据每个盒子特性进行绘制

具体的浏览器解析渲染机制如下所示：



- 解析HTML，生成DOM树，解析CSS，生成CSSOM树
- 将DOM树和CSSOM树结合，生成渲染树(Render Tree)
- Layout(回流):根据生成的渲染树，进行回流(Layout)，得到节点的几何信息（位置，大小）
- Painting(重绘):根据渲染树以及回流得到的几何信息，得到节点的绝对像素
- Display:将像素发送给GPU，展示在页面上

在页面初始渲染阶段，回流不可避免的触发，可以理解成页面一开始是空白的元素，后面添加了新的元素使页面布局发生改变

当我们对 `DOM` 的修改引发了 `DOM` 几何尺寸的变化（比如修改元素的宽、高或隐藏元素等）时，浏览器需要重新计算元素的几何属性，然后再将计算的结果绘制出来

当我们对 `DOM` 的修改导致了样式的变化（ `color` 或 `background-color` ），却并未影响其几何属性时，浏览器不需重新计算元素的几何属性、直接为该元素绘制新的样式，这里就仅仅触发了重绘

18.2. 如何触发

要想减少回流和重绘的次数，首先要了解回流和重绘是如何触发的

18.2.1. 回流触发时机

回流这一阶段主要是计算节点的位置和几何信息，那么当页面布局和几何信息发生变化的时候，就需要回流，如下面情况：

- 添加或删除可见的DOM元素

- 元素的位置发生变化
- 元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）
- 内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代
- 页面一开始渲染的时候（这避免不了）
- 浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）

还有一些容易被忽略的操作：获取一些特定属性的值

> offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight

这些属性有一个共性，就是需要通过即时计算得到。因此浏览器为了获取这些值，也会进行回流

除此还包括 `getComputedStyle` 方法，原理是一样的

18.2.2. 重绘触发时机

触发回流一定会触发重绘

可以把页面理解为一个黑板，黑板上有一朵画好的小花。现在我们要把这朵从左边移到了右边，那我们要先确定好右边的具体位置，画好形状（回流），再画上它原有的颜色（重绘）

除此之外还有一些其他引起重绘行为：

- 颜色的修改
- 文本方向的修改
- 阴影的修改

18.2.3. 浏览器优化机制

由于每次重排都会造成额外的计算消耗，因此大多数浏览器都会通过队列化修改并批量执行来优化重排过程。浏览器会将修改操作放入到队列里，直到过了一段时间或者操作达到了一个阈值，才清空队列

当你获取布局信息的操作的时候，会强制队列刷新，包括前面讲到的 `offsetTop` 等方法都会返回最新的数据

因此浏览器不得不清空队列，触发回流重绘来返回正确的值

18.3. 如何减少

我们了解了如何触发回流和重绘的场景，下面给出避免回流的经验：

- 如果想设定元素的样式，通过改变元素的 `class` 类名 (尽可能在 DOM 树的最里层)
- 避免设置多项内联样式
- 应用元素的动画，使用 `position` 属性的 `fixed` 值或 `absolute` 值(如前文示例所提)
- 避免使用 `table` 布局， `table` 中每个元素的大小以及内容的改动，都会导致整个 `table` 的重新计算
- 对于那些复杂的动画，对其设置 `position: fixed/absolute` ，尽可能地使元素脱离文档流，从而减少对其他元素的影响
- 使用css3硬件加速，可以让 `transform` 、 `opacity` 、 `filters` 这些动画不会引起回流重绘
- 避免使用 CSS 的 `JavaScript` 表达式

在使用 `JavaScript` 动态插入多个节点时, 可以使用 `DocumentFragment` . 创建后一次插入. 就能避免多次的渲染性能

但有时候，我们会无可避免地进行回流或者重绘，我们可以更好使用它们

例如，多次修改一个把元素布局的时候，我们很可能会如下操作

JavaScript | 复制代码

```
const el = document.getElementById('el')
for(let i=0;i<10;i++) {
    el.style.top  = el.offsetTop  + 10 + "px";
    el.style.left = el.offsetLeft + 10 + "px";
}
```

每次循环都需要获取多次 `offset` 属性，比较糟糕，可以使用变量的形式缓存起来，待计算完毕再提交给浏览器发出重计算请求

JavaScript | 复制代码

```
// 缓存offsetLeft与offsetTop的值
const el = document.getElementById('el')
let offLeft = el.offsetLeft, offTop = el.offsetTop

// 在JS层面进行计算
for(let i=0;i<10;i++) {
  offLeft += 10
  offTop  += 10
}

// 一次性将计算结果应用到DOM上
el.style.left = offLeft + "px"
el.style.top = offTop  + "px"
```

我们还可避免改变样式，使用类名去合并样式

```JavaScript
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

使用类名去合并样式

```HTML
<style>
    .basic_style {
        width: 100px;
        height: 200px;
        border: 10px solid red;
        color: red;
    }
</style>
<script>
    const container = document.getElementById('container')
    container.classList.add('basic_style')
</script>
```

前者每次单独操作，都去触发一次渲染树更改（新浏览器不会），

都去触发一次渲染树更改，从而导致相应的回流与重绘过程

合并之后，等于我们将所有的更改一次性发出

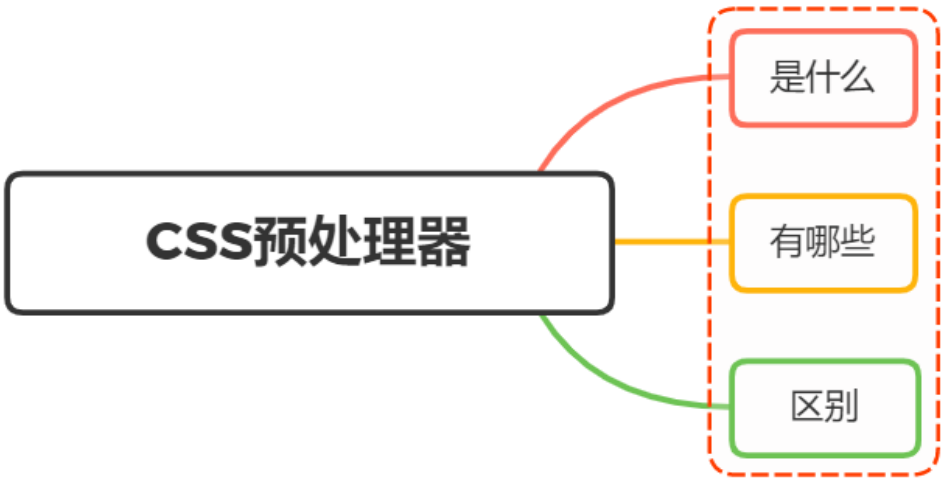
我们还可以通过通过设置元素属性 `display: none` ，将其从页面上去掉，然后再进行后续操作，这些后续操作也不会触发回流与重绘，这个过程称为离线操作

```JavaScript
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

离线操作后

```javascript
let container = document.getElementById('container')
container.style.display = 'none'
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
... (省略了许多类似的后续操作)
container.style.display = 'block'
```

19. 说说对Css预编语言的理解？有哪些区别？



19.1. 是什么

`Css` 作为一门标记性语言，语法相对简单，对使用者的要求较低，但同时也带来一些问题

需要书写大量看似没有逻辑的代码，不方便维护及扩展，不利于复用，尤其对于非前端开发工程师来讲，往往会因为缺少 `Css` 编写经验而很难写出组织良好且易于维护的 `Css` 代码

`Css` 预处理器便是针对上述问题的解决方

19.1.1. 预处理语言

扩充了 `Css` 语言，增加了诸如变量、混合（mixin）、函数等功能，让 `Css` 更易维护、方便

本质上，预处理是 `Css` 的超集

包含一套自定义的语法及一个解析器，根据这些语法定义自己的样式规则，这些规则最终会通过解析器，编译生成对应的 `Css` 文件

19.2. 有哪些

`Css` 预编译语言在前端里面有三大优秀的预编处理器，分别是：

- sass
- less
- stylus

19.2.1. sass

2007 年诞生，最早也是最成熟的 `Css` 预处理器，拥有 Ruby 社区的支持和 `Compass` 这一最强大的 `Css` 框架，目前受 `LESS` 影响，已经进化到了全面兼容 `Css` 的 `Scss`

文件后缀名为 `.sass` 与 `scss`，可以严格按照 sass 的缩进方式省去大括号和分号

19.2.2. less

2009年出现，受 `SASS` 的影响较大，但又使用 `Css` 的语法，让大部分开发者和设计师更容易上手，在 `Ruby` 社区之外支持者远超过 `SASS`

其缺点是比起 `SASS` 来，可编程功能不够，不过优点是简单和兼容 `Css`，反过来也影响了 `SASS` 演变到了 `Scss` 的时代

19.2.3. stylus

`Stylus` 是一个 `Css` 的预处理框架，2010 年产生，来自 `Node.js` 社区，主要用来给 `Node` 项目进行 `Css` 预处理支持

所以 `Stylus` 是一种新型语言，可以创建健壮的、动态的、富有表现力的 `Css`。比较年轻，其本质上做的事情与 `SASS/LESS` 等类似

19.3. 区别

虽然各种预处理器功能强大，但使用最多的，还是以下特性：

- 变量（variables）

- 作用域（scope）
- 代码混合（ mixins）
- 嵌套（nested rules）
- 代码模块化（Modules）

因此，下面就展开这些方面的区别

19.3.1. 基本使用

less和scss

```CSS
.box {
  display: block;
}
```

sass

```CSS
.box
  display: block
```

stylus

```CSS
.box
  display: block
```

19.3.2. 嵌套

三者的嵌套语法都是一致的，甚至连引用父级选择器的标记 & 也相同

区别只是 Sass 和 Stylus 可以用没有大括号的方式书写

less

```css
.a {
  &.b {
    color: red;
  }
}
```

19.3.3. 变量

变量无疑为 Css 增加了一种有效的复用方式，减少了原来在 Css 中无法避免的重复「硬编码」

`less` 声明的变量必须以 `@` 开头，后面紧跟变量名和变量值，而且变量名和变量值需要使用冒号 `:` 分隔开

```css
@red: #c00;

strong {
  color: @red;
}
```

`sass` 声明的变量跟 `less` 十分的相似，只是变量名前面使用 `@` 开头

```css
$red: #c00;

strong {
  color: $red;
}
```

`stylus` 声明的变量没有任何的限定，可以使用 `$` 开头，结尾的分号 `;` 可有可无，但变量与变量值之间需要使用 `=`

在 `stylus` 中我们不建议使用 `@` 符号开头声明变量

```css
red = #c00

strong
  color: red
```

19.3.4. 作用域

`Css` 预编译器把变量赋予作用域，也就是存在生命周期。就像 `js` 一样，它会先从局部作用域查找变量，依次向上级作用域查找

`sass` 中不存在全局变量

```css
$color: black;
.scoped {
  $bg: blue;
  $color: white;
  color: $color;
  background-color:$bg;
}
.unscoped {
  color:$color;
}
```

编译后

```css
.scoped {
  color:white;/*是白色*/
  background-color:blue;
}
.unscoped {
  color:white;/*白色（无全局变量概念）*/
}
```

所以，在 `sass` 中最好不要定义相同的变量名

`less` 与 `stylus` 的作用域跟 `javascript` 十分的相似，首先会查找局部定义的变量，如果没有找到，会像冒泡一样，一级一级往下查找，直到根为止

```css
@color: black;
.scoped {
  @bg: blue;
  @color: white;
  color: @color;
  background-color:@bg;
}
.unscoped {
  color:@color;
}
```

编译后：

```css
.scoped {
  color:white;/*白色（调用了局部变量）*/
  background-color:blue;
}
.unscoped {
  color:black;/*黑色（调用了全局变量）*/
}
```

19.3.5. 混入

混入（mixin）应该说是预处理器最精髓的功能之一了，简单点来说，`Mixins` 可以将一部分样式抽出，作为单独定义的模块，被很多选择器重复使用

可以在 `Mixins` 中定义变量或者默认参数

在 `less` 中，混合的用法是指将定义好的 `ClassA` 中引入另一个已经定义的 `Class`，也能使用够传递参数，参数变量为 `@` 声明

```css
.alert {
  font-weight: 700;
}

.highlight(@color: red) {
  font-size: 1.2em;
  color: @color;
}

.heads-up {
  .alert;
  .highlight(red);
}
```

编译后

```css
.alert {
  font-weight: 700;
}
.heads-up {
  font-weight: 700;
  font-size: 1.2em;
  color: red;
}
```

`Sass` 声明 `mixins` 时需要使用 `@mixinn` ，后面紧跟 `mixin` 的名，也可以设置参数，参数名为变量 `$` 声明的形式

CSS | 复制代码

```css
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}

.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

`stylus` 中的混合和前两款 `Css` 预处理器语言的混合略有不同，他可以不使用任何符号，就是直接声明 `Mixins` 名，然后在定义参数和默认值之间用等号（=）来连接

CSS | 复制代码

```css
error(borderWidth= 2px) {
  border: borderWidth solid #F00;
  color: #F00;
}
.generic-error {
  padding: 20px;
  margin: 4px;
  error(); /* 调用error mixins */
}
.login-error {
  left: 12px;
  position: absolute;
  top: 20px;
  error(5px); /* 调用error mixins，并将参数$borderWidth的值指定为5px */
}
```

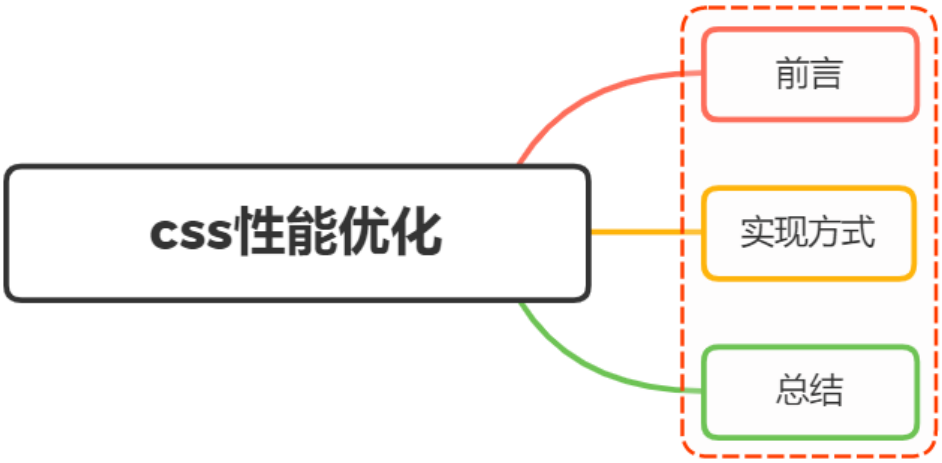
19.3.6. 代码模块化

模块化就是将 `Css` 代码分成一个个模块

`scss` 、 `less` 、 `stylus` 三者的使用方法都如下所示

```css
@import './common';
@import './github-markdown';
@import './mixin';
@import './variables';
```

20. 如果要做优化，CSS提高性能的方法有哪些?



20.1. 前言

每一个网页都离不开 `css` ，但是很多人又认为， `css` 主要是用来完成页面布局的，像一些细节或者优化，就不需要怎么考虑，实际上这种想法是不正确的

作为页面渲染和内容展现的重要环节， `css` 影响着用户对整个网站的第一体验

因此，在整个产品研发过程中， `css` 性能优化同样需要贯穿全程

20.2. 实现方式

实现方式有很多种，主要有如下：

- 内联首屏关键CSS
- 异步加载CSS
- 资源压缩
- 合理使用选择器