

额外知识补充（二）

王红元 coderwhy

目录

content



- 1 CSS常见单位详解
- 2 深入理解pixel、DPR、PPI
- 3 CSS预处理器Less、Scss
- 4 CSS Pixel、DPR、PPI
- 5 浏览器视口Viewport
- 6 移动端适配rem方案
- 7 移动端适配vw方案

CSS中的单位

- 前面编写的CSS中，我们经常会使用px来表示一个长度（大小），比如font-size设置为18px，width设置为100px。
- **px是一个长度（length）单位，事实上CSS中还有非常多的长度单位。**
- **整体可以分成两类：**
 - **绝对长度单位**（Absolute length units）；
 - **相对长度单位**（Relative length units）；

CSS中的绝对单位 (Absolute length units)

■ 绝对单位：

- 它们与其他任何东西都没有关系，通常被认为总是相同的大小。
- 这些值中的大多数在用于打印时比用于屏幕输出时更有用，例如，我们通常不会在屏幕上使用cm。
- 惟一一个您经常使用的值，就是px(像素)。

绝对单位	名称	等价换算
cm	厘米	1cm = 96px/2.54
mm	毫米	1mm = 1/10th of 1cm
Q	四分之一毫米	1Q = 1/40th of 1cm
in	英寸	1in = 2.54cm = 96px
pc	十二点活字	1pc = 1/16th of 1in
pt	点	1pt = 1/72th of 1in
px	像素	1px = 1/96th of 1in

	Recommended	Occasional use	Not recommended
Screen	em, px, %	ex	pt, cm, mm, in, pc
Print	em, cm, mm, in, pt, pc, %	px, ex	

CSS中的相对单位 (Relative length units)

■ 相对长度单位

- 相对长度单位相对于其他一些东西；
- 比如父元素的字体大小，或者视图端口的大小；
- 使用相对单位的好处是，经过一些仔细的规划，您可以使文本或其他元素的大小与页面上的其他内容相对应；

■ em :

'em unit'

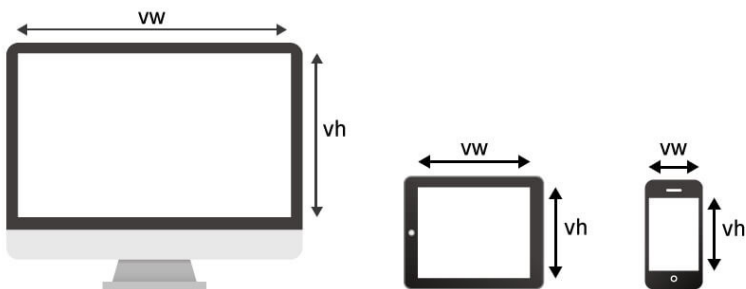
Equal to the computed value of the `'font-size'` property of the element on which it is used.

■ rem :

'rem unit'

Equal to the computed value of the `'em'` unit on the root element.

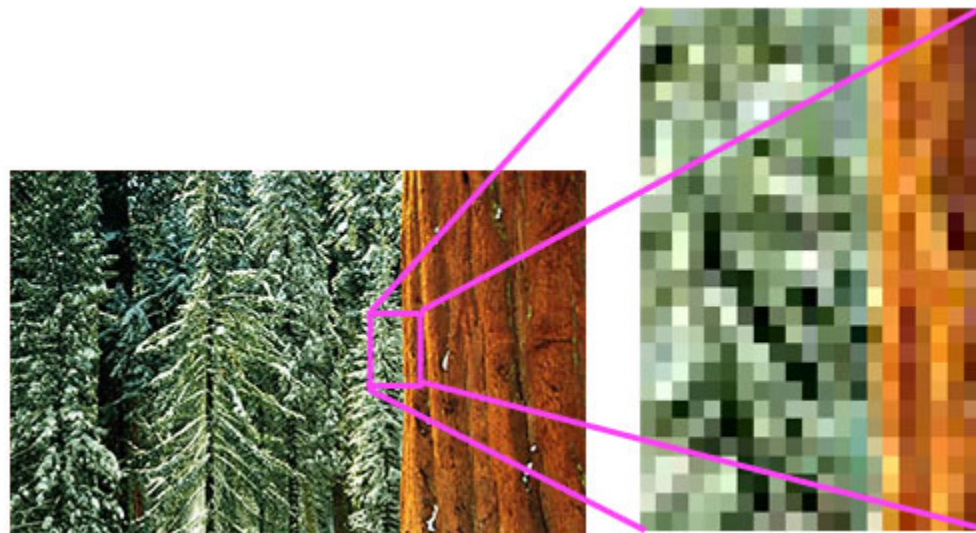
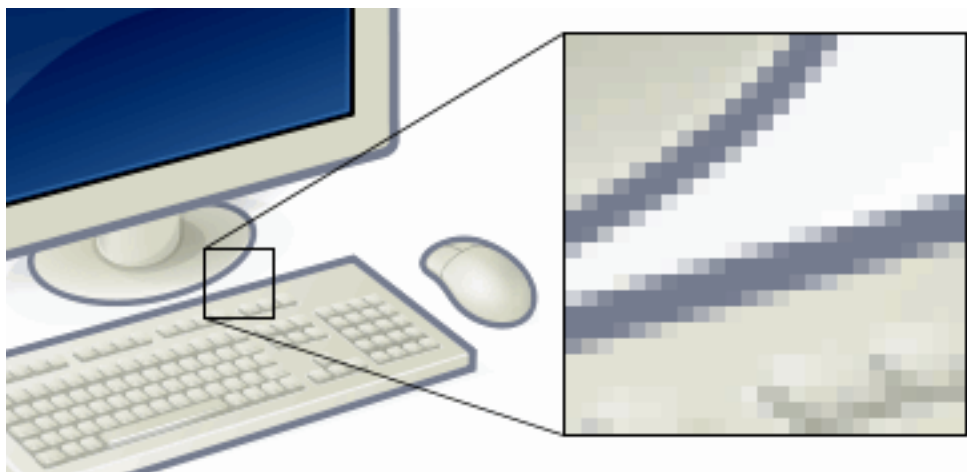
■ vw/vh



相对单位	相对于
em	在 font-size 中使用是相对于父元素的字体大小，在其他属性中使用是相对于自身的字体大小，如 width
ex	字符 “x” 的高度
ch	数字 “0” 的宽度
rem	根元素的字体大小
lh	元素的line-height
vw	视窗宽度的1%
vh	视窗高度的1%

当我们聊pixel时，到底在聊些什么？

- 前面我们已经一直在使用px单位了，px是pixel单词的缩写，翻译为像素。
- 那么像素到底是什么呢？
 - 像素是影响显示的基本单位。（比如屏幕上看到的画面、一幅图片）；
 - pix是英语单词picture的常用简写，加上英语单词“元素” element，就得到pixel；
 - “像素”表示“画像元素”之意，有时亦被称为pel（picture element）；



像素的不同分类（一）

■ 但是这个100个pixel到底是多少呢？

- 我们确实可以在屏幕上看到一个大小，但是这个大小代表的真实含义是什么呢？
- 我们经常说一个电脑的分辨率、手机的分辨率，这个CSS当中的像素又是什么关系呢？

```
.box {  
  width: 100px;  
  height: 100px;  
}
```

■ 这里我们要深入到不同的像素概念中，来理解CSS中的pixel到底代表什么含义。

■ 像素单位常见的有三种像素名称：

- 设备像素（也称之为物理像素）；
- 设备独立像素（也称之为逻辑像素）；
- CSS像素；

物理像素和逻辑像素

■ 设备像素，也叫物理像素。

- 设备像素指的是显示器上的真实像素，每个像素的大小是屏幕固有的属性，屏幕出厂以后就不会改变了；
- 我们在购买显示器或者手机的时候，提到的设备分辨率就是设备像素的大小；
- 比如iPhone X的分辨率 1125x2436，指的就是设备像素；

■ 设备独立像素，也叫逻辑像素。

- 如果面向开发者我们使用设备像素显示一个100px的宽度，那么在不同屏幕上显示效果会是不同的；
- 开发者针对不同的屏幕很难进行较好的适配，编写程序必须了解用户的分辨率来进行开发；
- 所以在设备像素之上，操作系统为开发者进行抽象，提供了逻辑像素的概念；
- 比如你购买了一台显示器，在操作系统上是以1920x1080设置的显示分辨率，那么无论你购买的是2k、4k的显示器，对于开发者来说，都是1920x1080的大小。

■ CSS像素

- CSS中我们经常使用的单位也是pixel，它在默认情况下等同于设备独立像素（也就是逻辑像素）；
- 毕竟逻辑像素才是面向我们开发者的；

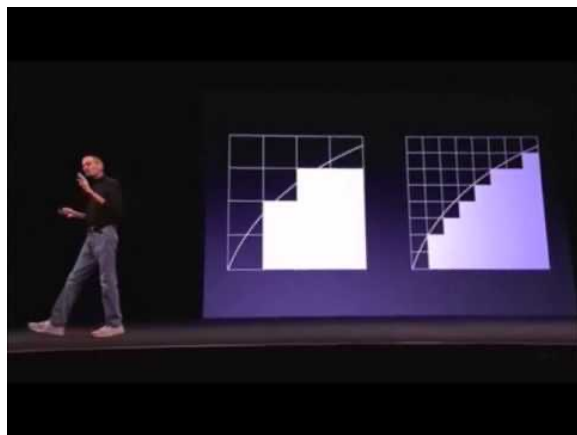
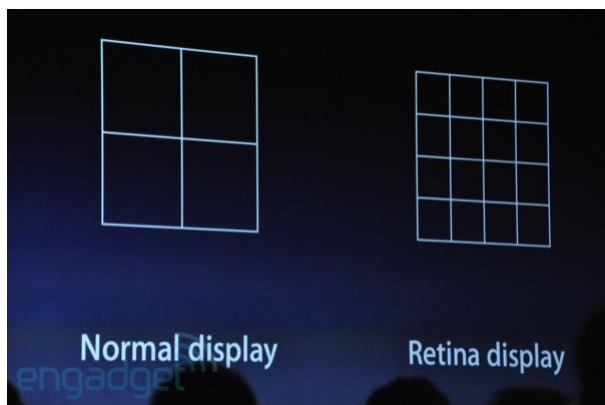
■ 我们可以通过JavaScript中的screen.width和screen.height获取到电脑的逻辑分辨率：

```
screen.height > screen.width  
1080 < 1920
```


DPR、PPI

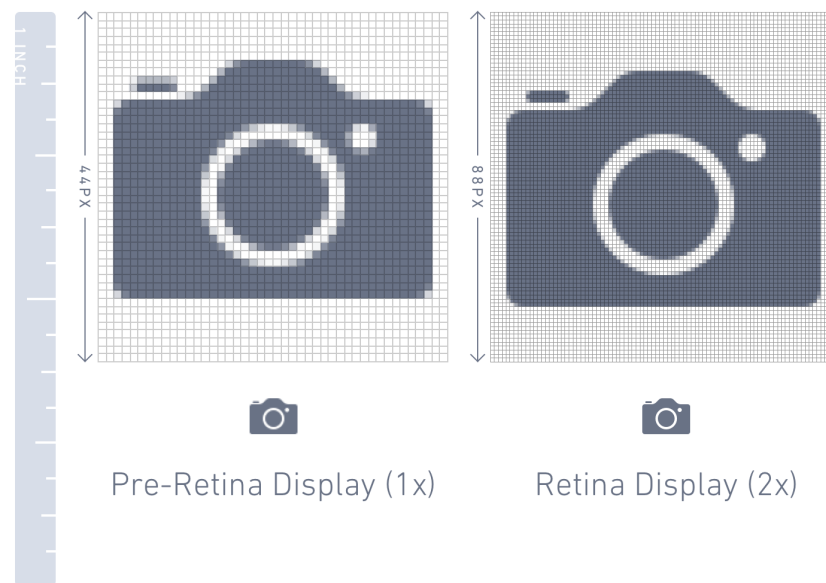
■ DPR : device pixel ratio

- 2010年，iPhone4问世，不仅仅带来了移动互联网，还带来了Retina屏幕；
- Retina屏幕翻译为视网膜显示屏，可以为用户带来更好的显示；
- 在Retina屏幕中，一个逻辑像素在长度上对应两个物理像素，这个比例称之为设备像素比（device pixel ratio）；
- 我们可以通过`window.devicePixelRatio`获取到当前屏幕上的DPR值；



■ PPI（了解）：每英寸像素（英语：Pixels Per Inch，缩写：PPI）

- 通常用来表示一个打印图像或者显示器上像素的密度；
- 前面我们提过1英寸=2.54厘米，在工业领域被广泛应用；



CSS编写的痛点

- CSS作为一种**样式语言**, 本身用来**给HTML元素添加样式**是没有问题的.
- 但是目前前端项目已经越来越复杂, 不再是简简单单的几行CSS就可以搞定的, 我们需要几千行甚至上万行的CSS来完成页面的美化工作.
- 随着代码量的增加, 必然会造成很多的编写不便 :
 - 比如大量的重复代码, 虽然可以用**类来勉强管理和抽取**, 但是**使用起来依然不方便** ;
 - 比如**无法定义变量** (当然目前已经支持), 如果一个值被修改, 那么需要**修改大量代码, 可维护性很差**; (比如主题颜色)
 - 比如**没有专门的作用域和嵌套**, 需要**定义大量的id/class来保证选择器的准确性**, 避免样式混淆;
 - 等等一系列的问题 ;
- 所以有一种对CSS称呼是 **“面向命名编程”** ;
- 社区为了解决CSS面临的大量问题, 出现了一系列的**CSS预处理器**(CSS_preprocessor)
 - CSS 预处理器是一个能让你通过**预处理器自己独有的语法**来生成CSS的程序;
 - 市面上有很多**CSS预处理器**可供选择 , 且绝大多数CSS预处理器会**增加一些原生CSS不具备的特性**;
 - 代码最终**会转化为CSS来运行**, 因为**对于浏览器来说只识别CSS**;

常见的CSS预处理器

■ 常见的预处理器有哪些呢？目前使用较多的是三种预处理器：

■ Sass/Scss：

- 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，是属于Haml（一种模板系统）的一部分；
- 目前受LESS影响，已经进化到了全面兼容CSS的SCSS；

■ Less：

- 2009年出现，受SASS的影响较大，但又使用CSS的语法，让大部分开发者更容易上手；
- 比起SASS来，可编程功能不够，不过优点是使用方式简单、便捷，兼容CSS，并且已经足够使用；
- 另外反过来也影响了SASS演变到了SCSS的时代；
- 著名的Twitter Bootstrap就是采用LESS做底层语言的，也包括React的UI框架AntDesign。

■ Stylus：

- 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；
- 语法偏向于Python，使用率相对于Sass/Less少很多

■ 什么是Less呢? 我们来看一下官方的介绍:

□ It's CSS, with just a little more.



■ Less (Leaner Style Sheets 的缩写) 是一门CSS 扩展语言, 并且兼容CSS。

□ Less增加了很多相比于CSS更好用的特性;

□ 比如定义变量、混入、嵌套、计算等等 ;

□ Less最终需要被编译成CSS运行于浏览器中 (包括部署到服务器中) ;

- 我们可以编写如下的Less代码：

```
@mainColor: #fa0112;

.box {
  color: @mainColor;

  .desc {
    font-size: 12px;
  }

  .info {
    font-size: 18px;
  }

  &:hover {
    background-color: #0f0;
  }
}
```

less代码的编译

■ 这段代码如何被编译成CSS代码运行呢？

■ 方式一：下载Node环境，通过npm包管理下载less工具，使用less工具对代码进行编译；

- 因为目前我们还没有学习过Node，更没有学习过npm工具；
- 所以先阶段不推荐大家使用less本地工具来管理；
- 后续我们学习了webpack其实可以自动完成这些操作的；

■ 方法二：通过VSCode插件来编译成CSS或者在线编译

- <https://lesscss.org/less-preview/>

■ 方式三：引入CDN的less编译代码，对less进行实时的处理；

```
<link rel="stylesheet/less" href="./less/01_测试less.less">  
<script src="https://cdn.jsdelivr.net/npm/less@4" ></script>
```

```
<script src="https://cdn.jsdelivr.net/npm/less@4" ></script>
```



■ 方式四：将less编译的js代码下载到本地，执行js代码对less进行编译；

- 看上课的操作；

Less语法一：Less兼容CSS

■ Less语法一：Less是兼容CSS的

- 所以我们可以Less文件中编写所有的CSS代码；
- 只是将css的扩展名改成了.less结尾而已；

```
-Scss > less > (less) 01_less兼容css.less > ...  
.box {  
  height: 200px;  
  background-color:  orange;  
}  
  
.box .small {  
  display: inline-block;  
  width: 100px;  
  height: 100px;  
  background-color:  #f00;  
}
```

Less语法二 – 变量 (Variables)

■ 在一个大型的网页项目中，我们CSS使用到的某几种属性值往往是特定的

- 比如我们使用到的主题颜色值，那么每次编写类似于#f3c258格式的语法；
- 一方面是记忆不太方便，需要重新编写或者拷贝样式；
- 另一方面如果有一天主题颜色改变，我们需要修改大量的代码；
- 所以，我们可以将常见的颜色或者字体等定义为变量来使用；

■ 在Less中使用如下的格式来定义变量；

@变量名： 变量值；

```
@themeColor: #f3c258;
@mainFontSize: 12px;

.box p .link {
  color: @themeColor;
  font-size: @mainFontSize;
}
```


Less语法三 – 嵌套 (Nesting)

- 在之前的项目中，当我们需要找到一个内层的元素时，往往需要嵌套很多层的选择器

```
#main .section-01 .news-list-info .date {  
  font-size: 12px;  
  color: #999;  
}
```

- Less提供了选择器的嵌套

```
<div class="box">  
  <h1 class="title">我是标题</h1>  
  <p class="content">  
    我是段落内容  
    <a class="link" href="#">我是超链接</a>  
    <span class="keyword">keyword</span>  
  </p>  
</div>
```

```
.box {  
  .title {  
    color: red;  
  }  
  
  .content {  
    .link {  
      font-size: 20px;  
      color: orange;  
    }  
  
    .keyword {  
      font-size: 30px;  
      color: purple;  
    }  
  }  
}
```

- 特殊符号：**&** 表示当前选择器的父级

Less语法四 – 运算 (Operations)

- 在Less中，算术运算符 +、-、*、/ 可以对任何数字、颜色或变量进行运算。
 - 算术运算符在加、减或比较之前会进行单位换算，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；

```
.box {  
  height: 100px + 10%;  
  background-color: #ff0000 + #00ff00;  
}
```

Less语法五 – 混合 (Mixins)

- 在原来的CSS编写过程中，多个选择器中可能会有大量相同的代码
 - 我们希望能将这些代码进行抽取到一个独立的地方，任何选择器都可以进行复用；
 - 在less中提供了混入 (Mixins) 来帮助我们完成这样的操作；
- 混合 (Mixin) 是一种将一组属性从一个规则集 (或混入) 到另一个规则集的方法。

```
.bordered {  
  border-top: 2px solid ■ #f00;  
  border-bottom: 2px dotted ■ #0f0;  
}
```

```
.box {  
  height: 100px;  
  background-color: ■ orange;  
  
  .bordered()  
}  
  
.container {  
  height: 200px;  
  background-color: ■ purple;  
  
  .bordered()  
}
```

- 注意：混入在没有参数的情况下，小括号可以省略，但是不建议这样使用；

Less语法五 – 混合 (Mixins)

■ 混入也可以传入变量 (暂时了解)

```
.bordered(@borderWidth: 2px) {  
  border-top: @borderWidth solid #f00;  
  border-bottom: @borderWidth dotted #0f0;  
}
```

■ Less语法六：映射 (Maps)

```
.colors() {  
  primaryColor: #f00;  
  secondColor: #0f0;  
}  
  
.box {  
  width: 100px;  
  height: 100px;  
  color: .colors[primaryColor];  
  background-color: .colors()[secondColor];  
}
```

■ 混入和映射结合：混入也可以当做一个自定义函数来使用 (暂时了解)

```
.pxToRem(@px) {  
  result: (@px / @htmlFontSize) * 1rem;  
}  
  
.box {  
  width: .pxToRem(100)[result];  
  font-size: .pxToRem(18)[result];  
}
```

■ Less语法七：extend继承

- 和mixins作用类似，用于复用代码；
- 和mixins相比，继承代码最终会转化成并集选择器；

```
.bordered {  
  border-bottom: 10px solid #000;  
}  
  
.box {  
  &:extend(.bordered);  
}
```

```
.bordered,  
.box {  
  border-bottom: 10px solid #000;  
}
```

■ Less语法八：Less内置函数

- Less 内置了多种函数用于转换颜色、处理字符串、算术运算等。
- 内置函数手册：<https://less.bootcss.com/functions/>

```
.box {  
  color: color(■ red); // 将red转成RGB的值  
  width: convert(100px, "in"); // 单位的转换  
  font-size: ceil(18.5px); // 数学函数  
}
```

■ Less语法九：作用域（Scope）

- 在查找一个变量时，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；

```
.box {  
  .inner {  
    font-size: @fontSize; // 使用15px  
  }  
  @fontSize: 15px;  
}
```

■ Less语法十：注释（Comments）

- 在Less中，~~块注释和行注释~~都可以使用；

■ Less语法十一：导入（Importing）

- ~~导入的方式和CSS的用法是一致的~~；
- 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；

认识Sass和Scss

- 事实上，最初Sass 是Haml的一部分，Haml 是一种模板系统，由 Ruby 开发者设计和开发。
- 所以，Sass的语法使用的是类似于Ruby的语法，没有花括号，没有分号，具有严格的缩进；

```
$font-stack: Helvetica, sans-serif
$primary-color: #333

body
  font: 100% $font-stack
  color: $primary-color
```

- 我们会发现它的语法和css区别很大，后来官方推出了全新的语法SCSS，意思是Sassy CSS，他是完全兼容CSS的。
- 目前在前端学习SCSS直接学习SCSS即可：
 - SCSS的语法也包括变量、嵌套、混入、函数、操作符、作用域等；
 - 通常也包括更为强大的控制语句、更灵活的函数、插值语法等；
 - 大家可以根据之前学习的less语法来学习一些SCSS语法；
 - <https://sass-lang.com/guide>
- 目前大家掌握Less的使用即可；

什么是移动端适配？

■ 移动互联网的快速发展，让人们已经越来越习惯于使用手机来完成大部分日常的事务。

- 前端我们已经学习了大量HTML、CSS的前端开发知识，并且也进行了项目实战；
- 这些知识也同样适用于移动端开发，但是如果想让一个页面真正适配于移动端，我们最好多了解一些移动端的知识；

■ 移动端开发目前主要包括三类：

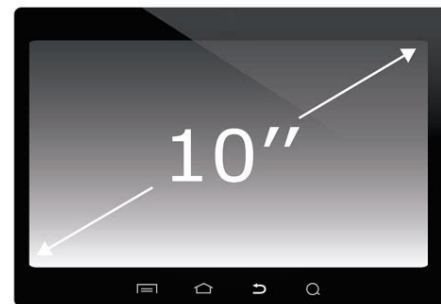
- 原生App开发（iOS、Android、RN、uniapp、Flutter等）
- 小程序开发（原生小程序、uniapp、Taro等）
- Web页面（移动端的Web页面，可以使用浏览器或者webview浏览）

■ 因为目前移动端设备较多，所以我们需要对其进行一些适配。

■ 这里有两个概念：

- 自适应：根据不同的设备屏幕大小来自动调整尺寸、大小；
- 响应式：会随着屏幕的实时变动而自动调整，是一种自适应；

Mobile devices vector set with sizes



认识视口viewport

■ 在前面我们已经简单了解过视口的概念了：

- 在一个浏览器中，我们可以看到的区域就是视口（viewport）；
- 我们说过fixed就是相对于视口来进行定位的；
- 在PC端的页面中，我们是不需要对视口进行区分，因为我们的布局视口和视觉视口是同一个；

■ 但是在移动端，不太一样，你布局的视口和你可见的视口是不太一样的。

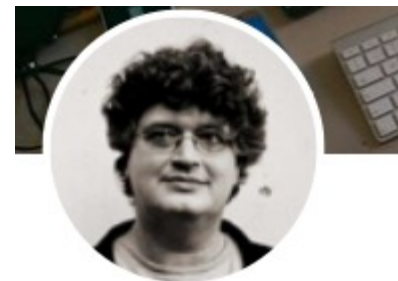
- 这是因为移动端的网页窗口往往比较小，我们可能会希望一个大的网页在移动端可以完整的显示；
- 所以在默认情况下，移动端的布局视口是大于视觉视口的；

■ 所以在移动端，我们可以将视口划分为三种情况：

- 布局视口（layout viewport）
- 视觉视口（visual layout）
- 理想视口（ideal layout）

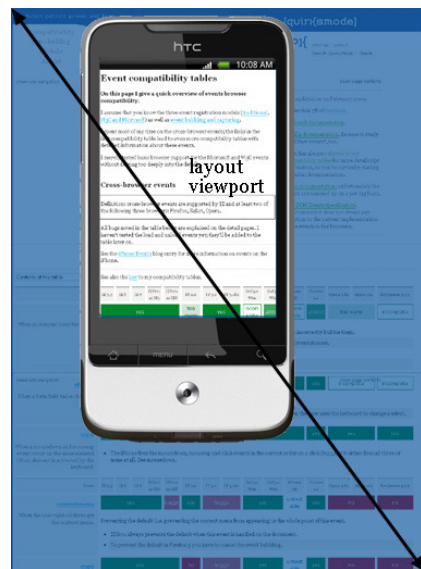
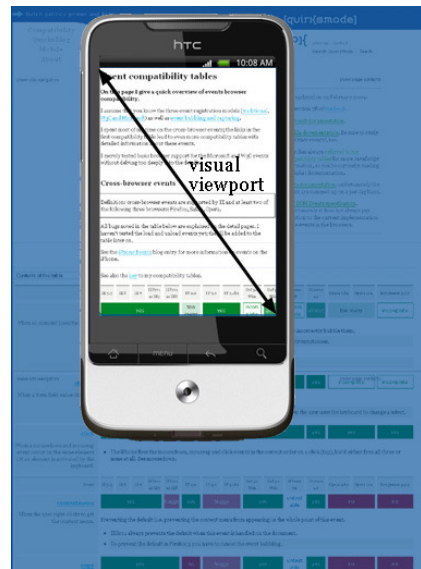
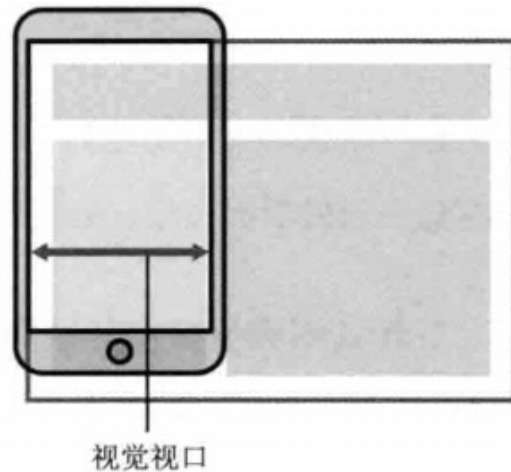
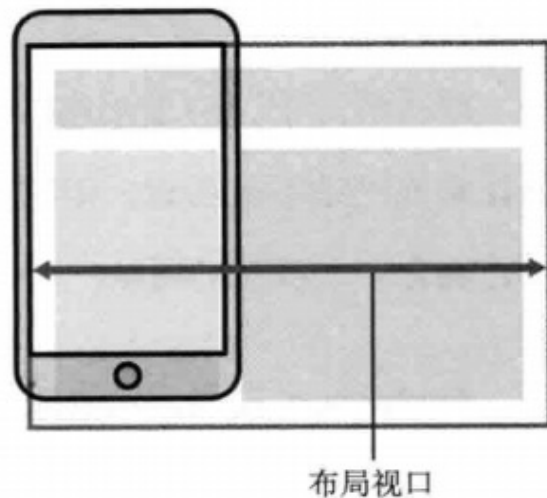
■ 这些概念的区分，事实上来自ppk，他也是对前端贡献比较大的一个人（特别是在移动端浏览器）

- <https://www.quirksmode.org/mobile/viewports2.html>



布局视口和视觉视口

- 布局视口 (layout viewport)
- 默认情况下，一个在PC端的网页在移动端会如何显示呢？
 - 第一，它会按照宽度为980px来布局一个页面的盒子和内容；
 - 第二，为了显示可以完整的显示在页面中，对整个页面进行缩小；
- 我们相对于980px布局的这个视口，称之为**布局视口 (layout viewport)**；
 - 布局视口的默认宽度是980px；
- 视觉视口 (visual viewport)
 - 如果默认情况下，我们按照980px显示内容，那么右侧有一部分区域就会无法显示，所以手机端浏览器会默认对页面进行缩放以显示到用户的可见区域中；
 - 那么显示在可见区域的这个视口，就是视觉视口 (visual viewport)
- 在Chrome上按shift+鼠标左键可以进行缩放。



理想视口 (ideal viewport)

■ 如果所有的网页都按照980px在移动端布局，那么最终页面都会被缩放显示。

- 事实上这种方式是不利于我们进行移动的开发的，我们希望的是设置100px，那么显示的就是100px；
- 如何做到这一点呢？通过设置理想视口 (ideal viewport)；

■ 理想视口 (ideal viewport)：

- 默认情况下的layout viewport并不适合我们进行布局；
- 我们可以对layout viewport进行宽度和缩放的设置，以满足正常在一个移动端窗口的布局；
- 这个时候可以设置meta中的viewport；



值	可能的附加值	描述
width	一个正整数，或者字符串 device-width	定义 viewport 的宽度。
height	一个正整数，或者字符串 device-height	定义 viewport 的高度。未被任何浏览器使用。
initial-scale	一个 0.0 和 10.0 之间的正数	定义设备宽度与 viewport 大小之间的缩放比例。
maximum-scale	一个 0.0 和 10.0 之间的正数	定义缩放的最大值，必须大于等于 minimum-scale，否则表现将不可预测。
minimum-scale	一个 0.0 和 10.0 之间的正数	定义缩放的最小值，必须小于等于 maximum-scale，否则表现将不可预测。
user-scalable	yes 或者 no	默认为 yes，如果设置为 no，将无法缩放当前页面。浏览器可以忽略此规则；

移动端适配方案

■ 移动端的屏幕尺寸通常是非常繁多的，很多时候我们希望在不同的屏幕尺寸上显示不同的大小；

□ 比如我们设置一个100x100的盒子

- ✓ 在375px的屏幕上显示是100x100;
- ✓ 在320px的屏幕上显示是90+x90+;
- ✓ 在414px的屏幕上显示是100+x100+;

□ 其他尺寸也是类似，比如padding、margin、border、left，甚至是font-size等等；

■ 这个时候，我们可能可以想到一些方案来处理尺寸：

□ 方案一：百分比设置；

- ✓ 因为不同属性的百分比值，相对的可能是不同参照物，所以百分比往往很难统一；
- ✓ 所以百分比在移动端适配中使用是非常少的；

□ 方案二：rem单位+动态html的font-size；

□ 方案三：vw单位；

□ 方案四：flex的弹性布局；

适配方案 – rem+动态html的font-size

■ rem单位是相对于html元素的font-size来设置的，那么如果我们需要在不同的屏幕下有不同的尺寸，可以动态的修改html的font-size尺寸。

■ 比如如下案例：

- 1.设置一个盒子的宽度是2rem；
- 2.设置不同的屏幕上html的font-size不同；

屏幕尺寸	html的font-size	盒子的设置宽度	盒子的最终宽度
375px	37.5px	1rem	37.5px
320px	32px	1rem	32px
414px	41.4px	1rem	41.4px

■ 这样在开发中，我们只需要考虑两个问题：

- 问题一：针对不同的屏幕，设置html不同的font-size；
- 问题二：将原来要设置的尺寸，转化成rem单位；

rem的font-size尺寸

■ 方案一：媒体查询

- 可以通过媒体查询来设置不同尺寸范围内的屏幕html的font-size尺寸；
- 缺点：
 - ✓ 1.我们需要针对不同的屏幕编写大量的媒体查询；
 - ✓ 2.如果动态改变尺寸，不会实时的进行更新；

■ 方案二：编写js代码

- 如果希望实时改变屏幕尺寸时，font-size也可以实时更改，可以通过js代码；
- 方法：
 - ✓ 1.根据html的宽度计算出font-size的大小，并且设置到html上；
 - ✓ 2.监听页面的实时改变，并且重新设置font-size的大小到html上；

■ 方案三：lib-flexible库

- 事实上，lib-flexible库做的事情是相同的，你也可以直接引入它；

```
@media screen and (min-width: 320px) {  
  html { font-size: 32px; }  
}  
  
@media screen and (min-width: 375px) {  
  html { font-size: 37.5px; }  
}  
  
@media screen and (min-width: 414) {  
  html { font-size: 41.1px; }  
}
```

```
const htmlEl = document.documentElement  
function setRemUnit() {  
  const unit = htmlEl.clientWidth / 10  
  htmlEl.style.fontSize = unit + "px"  
}  
  
setRemUnit()  
window.addEventListener("resize", function() {  
  setRemUnit()  
})  
window.addEventListener("pageshow", function(e) {  
  if (e.persisted) {  
    setRemUnit()  
  }  
})
```

rem的单位换算

■ 方案一：手动换算

- 比如有一个在375px屏幕上，100px宽度和高度的盒子；
- 我们需要将100px转成对应的rem值；
- $100/37.5=2.6667$ ，其他也是相同的方法计算即可；

■ 方案二：less/scss函数

```
.pxToRem(@px) {  
  result: (@px / @htmlFontSize) * 1rem  
}  
  
.box {  
  width: .pxToRem(100)[result];  
  font-size: .pxToRem(18)[result];  
}
```

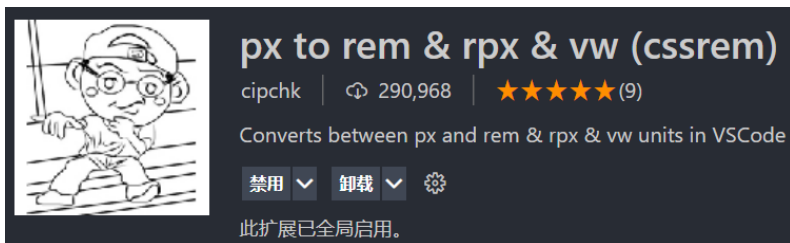
rem的单位换算

■ 方案三：postcss-pxtorem（后续学习）

- 目前在前端的工程化开发中，我们可以借助于webpack的工具来完成自动的转化；

■ 方案四：VSCode插件

- px to rem 的插件，在编写时自动转化；



全部

未分组

考拉海购

考拉海购 · 版本1



Web 750 x 3642 px

样式信息

图层 考拉海购

位置 0px 0px

大小 750px 3642px

颜色

#F5F5F5 100%

HEX

代码

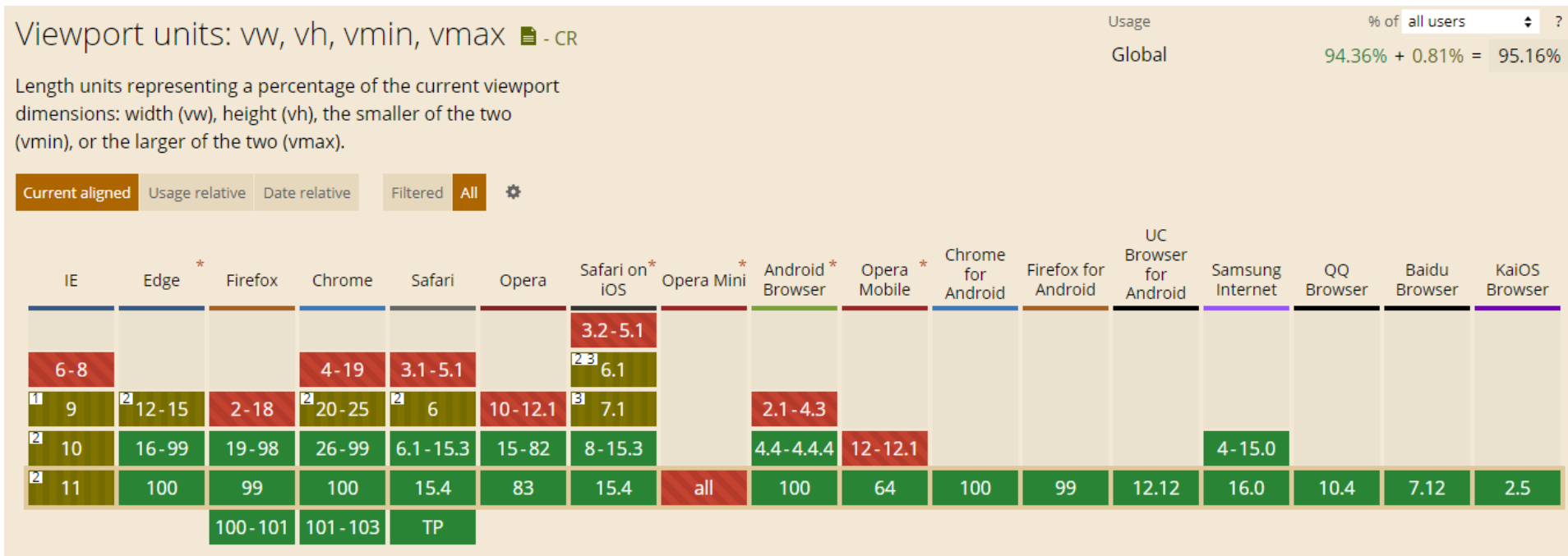
复制代码

```
width: 750px;  
height: 3642px;  
background: #F5F5F5;
```

查看代码

由于 `viewport` 单位得到众多浏览器的兼容，`lib-flexible` 这个过渡方案已经可以放弃使用，不管是现在的版本还是以前的版本，都存有一定的问题。建议大家开始使用 `viewport` 来替代此方。

■ vw的兼容性如何呢？



vw和rem的对比

■ rem事实上是作为一种过渡的方案，它利用的也是vw的思想。

- 前面不管是我们自己编写的js，还是flexible的源码；
- 都是将1rem等同于设计稿的1/10，在利用1rem计算相对于整个屏幕的尺寸大小；
- 那么我们来思考，1vw不是刚好等于屏幕的1/100吗？
- 而且相对于rem还更加有优势；

■ vw相比于rem的优势：

- 优势一：不需要去计算html的font-size大小，也不需要给html设置这样一个font-size；
- 优势二：不会因为设置html的font-size大小，而必须给body再设置一个font-size，防止继承；
- 优势三：因为不依赖font-size的尺寸，所以不用担心某些原因html的font-size尺寸被篡改，页面尺寸混乱；
- 优势四：vw相比于rem更加语义化，1vw刚才是1/100的viewport的大小；
- 优势五：可以具备rem之前所有的优点；

■ vw我们只面临一个问题，将尺寸换算成vw的单位即可；

■ 所以，目前相比于rem，更加推荐大家使用vw（但是理解rem依然很重要）

■ 方案一：手动换算

- 比如有一个在375px屏幕上，100px宽度和高度的盒子；
- 我们需要将100px转成对应的vw值；
- $100/3.75=26.667$ ，其他也是相同的方法计算即可；

■ 方案二：less/scss函数

```
@vwUnit: 3.75;

.pxToVw(@px) {
  result: (@px / @vwUnit) * 1vw
}

.box {
  width: .pxToVw(100)[result];
  height: .pxToVw(100)[result];
}
```

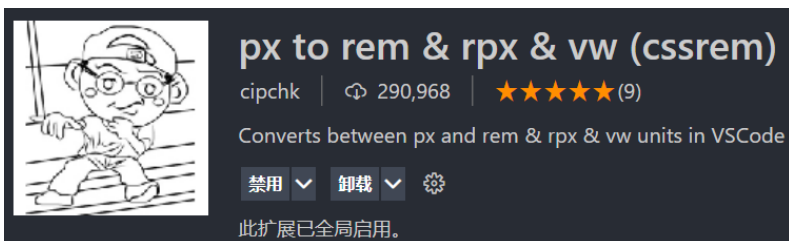
vw的单位换算

■ 方案三：postcss-px-to-viewport-8-plugin（后续学习）

□ 和rem一样，在前端的工程化开发中，我们可以借助于webpack的工具来完成自动的转化；

■ 方案四：VSCode插件

□ px to vw 的插件，在编写时自动转化；



- [illegible]

grid布局重要的概念

■ Grid Container

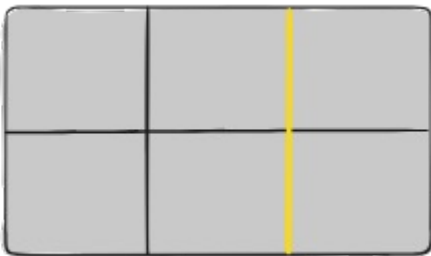
- 元素设置display为grid的盒子。

■ Grid Item , 单元格称之为grid cell

- grid container的直接子项（必须是直接子代）。

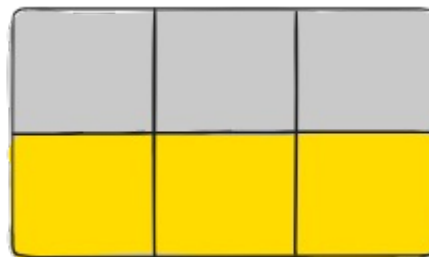
■ Grid Line

- 构成网格结构的分割线。
- 它们可以是垂直的（“列网格线”）或水平的（“行网格线”）。



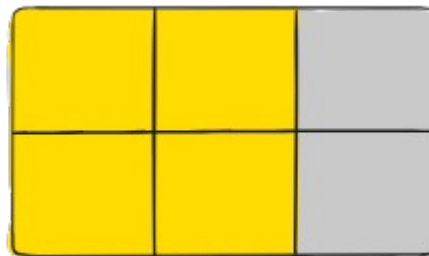
■ Grid Track

- 两条相邻网格线之间的空间；
- 你可以看成是网格的行或者列；



■ Grid Area

- 由四条网格线包围的总空间。
- 一个网格区域可以由任意数量的网格单元组成。



grid布局常见属性

■ grid container常见属性：

- ❑ display
- ❑ grid-template-columns
- ❑ grid-template-rows
- ❑ grid-template-areas
 - ✓ grid-template
- ❑ grid-column-gap
- ❑ grid-row-gap
 - ✓ grid-gap
- ❑ justify-items
- ❑ align-items
 - ✓ place-items
- ❑ justify-content
- ❑ align-content
 - ✓ place-content
- ❑ grid-auto-columns
- ❑ grid-auto-rows
 - ✓ grid-auto-flow
- ❑ grid

■ grid item常见属性：

- ❑ grid-column-start
- ❑ grid-column-end
- ❑ grid-row-start
- ❑ grid-row-end
- ❑ grid-column
- ❑ grid-row
- ❑ grid-area
- ❑ justify-self
- ❑ align-self
- ❑ place-self

- <https://css-tricks.com/snippets/css/complete-guide-grid/>