# 2022 年移动端适配方案指南 — 全网最新最全

在移动端虽然整体来说大部分浏览器内核都是 webkit，而且大部分都支持 css3 的所有语法。但是，由于手机屏幕尺寸不一样，分辨率不一样，或者你需要考虑横竖屏的问题，或者考虑到各式各样的移动端兼容性问题。这时候你也就不得不解决在不同手机上，不同情况下的展示效果，所以就需要一个开箱即用并且行之有效的移动端适配方案。

# 一、基本知识点

工欲善其事必先利其器，在具体介绍适配方案前，在本章我们会学习下适配相关的知识点，便于后续适配方案的直接上手接收。

## 1.1、响应式设计 – 像素

像素单位有设备像素、逻辑像素、CSS 像素 3 种。

### 1.1.1、设备像素、设备分辨率

设备像素（device pixels）也叫物理像素，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再改变。
设备分辨率描述的就是这个显示器的宽和高分别是多少个设备像素，例如常见的显示器的分辨率为 1920 * 1080。
设备像素和设备分辨率是由操作系统来管理的，浏览器不知道、也不必知道设备分辨率的大小，它主要根据逻辑分别率（下一小节介绍）来计算的。

### 1.1.2、设备独立像素、逻辑分辨率

设备独立像素（device independent pixels）是操作系统定义的一种像素单位，应用程序将设备独立像素告诉操作系统，操作系统再将设备独立像素转化为设备像素，从而控制屏幕上真正的物理像素点。
为什么需要在应用程序与设备像素之间定义这么一种单位呢？为什么应用程序不应该直接使用设备像素？
例如原先在 1280×720 设备分辨率的显示屏中，显示高度为 12 个设备像素的字体，现在放到设备分辨率为 2560 ×1440 的显示屏中，如果要想得到原先的大小，则需要 24 个设备像素，如果应用程序直接使用设备像素，那么编写应用程序则将变得非常困难，需要编写应用程序逻辑：字体在一些屏幕上高度为 12 个设备像素，在另一些屏幕上却需要 24 个设备像素。
因此操作系统定义了一个单位：设备独立像素。操作系统保证：用设备独立像素定义的尺寸，不管屏幕的参数如何，都能以合适的大小显示（这也是设备独立像素名字的由来）。操作系统是如何做到的呢？对于那些像素密度高的屏幕，将多个设备像素划分为一个逻辑像素。至于将多少设备像素划分为一个逻辑像素，这由操作系统决定。
对于上面的例子：“原本高度为 12 个设备像素的字体，现在高度为 24 个设备像素才能得到相同的大小”，操

作系统会将一个逻辑像素定义为 2*2 个 真实像素，从而设备独立像素尺寸不需要改变，而且不管在新、旧设备上，显示的尺寸大致相同。

> 设备独立像素与设备像素之间的比例是多少，显示器厂商和操作系统厂商会通过调查研究来得出最利于观看的比例。普遍规律是，屏幕的像素密度越高，就需要更多的设备像素来显示一个设备独立像素。

逻辑分辨率用屏幕的 宽_高 来表示（单位：设备独立像素），我们通过操作系统的分辨率设置来改变设备独立像素的大小。例如屏幕的设备分辨率是 1920_1200（单位：设备像素），我们可以在当前的分辨率下设置逻辑分辨率是 1280*800（单位：设备独立像素）。那么横、纵方向的设备像素数量恰好是设备独立像素的 1.5 倍。这也意味着，设备独立像素的边长是设备像素边长的 1.5 倍。

### 1.1.3、CSS 像素

在 CSS 中使用的 px 都是指 css 像素，比如 width: 128px。css 像素的大小是很容易变化的，当我们缩放页面的时候，元素的 css 像素数量不会改变，改变的只是每个 css 像素的大小。也就是说 width: 128px 的元素在缩放 200% 以后，宽度依然是 128 个 css 像素，只不过每个 css 像素的宽度和高度变为原来的两倍。如果原本元素宽度为 128 个设备独立像素，那么缩放 200% 以后元素宽度为 256 个设备独立像素。

（1）css 像素与设备独立像素的关系

- 缩放比例就是 css 像素边长 / 设备独立像素边长；
- 在缩放比例为 100% 的情况下，1 个 css 像素大小等于 1 个设备独立像素；
- 在缩放比例为 200% 的情况下，1 个 css 像素大小等于 (2 * 2) 个设备独立像素；

（2）css 像素与设备像素的关系

window.devicePixelRatio 设备像素比，devicePixelRatio = （在相同长度的直线上）设备像素的数量 / CSS 像素的数量。这个比例也等价于 CSS 像素边长 / 设备像素边长。如 devicePixelRatio = 2，表示在相同长度的直线上，设备像素的数量是 CSS 像素数量的 2 倍，因此 CSS 像素的边长是设备像素的 2 倍。缩放会导致 CSS 像素边长的改变，从而导致 window.devicePixelRatio 的改变！

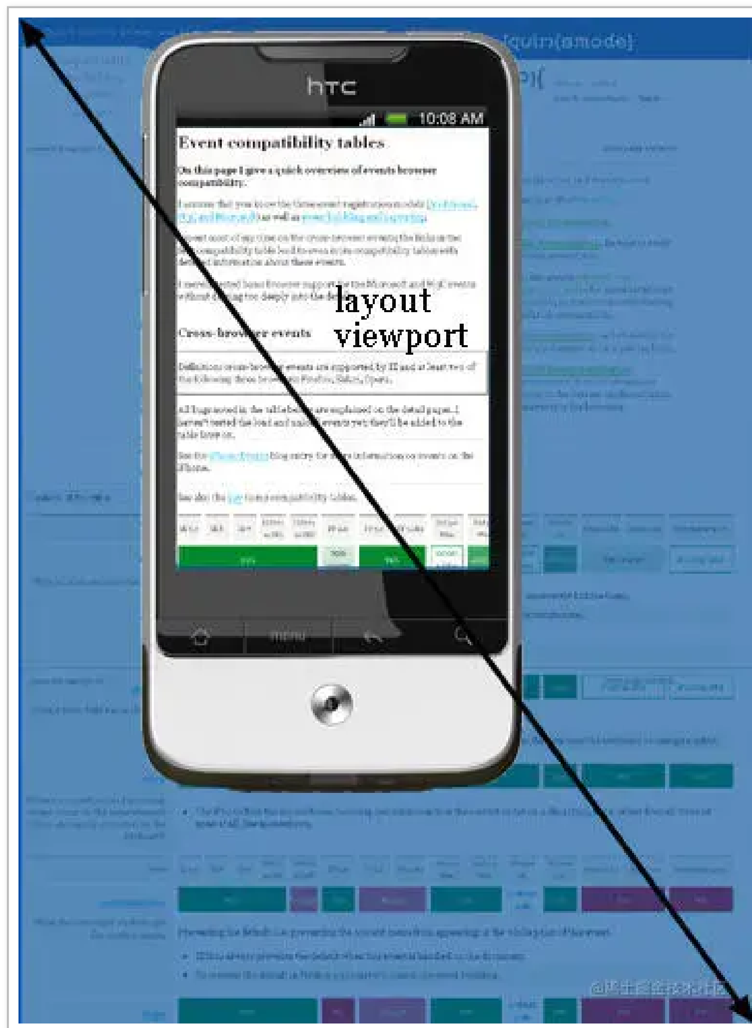
## 1.2、响应式设计 - viewport

viewport 表示浏览器的可视区域，也就是浏览器中用来显示网页的那部分区域。存在三种 viewport 分别为 layout viewport、visual viewport 以及 ideal viewport，我们接下来分别介绍三种。

### 1.2.1、layout viewport

layout viewport 为布局视口，即网页布局的区域，它是 html 元素的父容器，只要不在 css 中修改 元素的宽度， 元素的宽度就会撑满 layout viewport 的宽度。
很多时候浏览器窗口没有办法显示出 layout viewport 的全貌，但是它确实是已经被加载出来了，这个时候滚动条就出现了，你需要通过滚动条来浏览 layout viewport 其他的部分。
layout viewport 用 css 像素来衡量尺寸，在缩放、调整浏览器窗口的时候不会改变。缩放、调整浏览器窗口改变的只是 visual viewport。

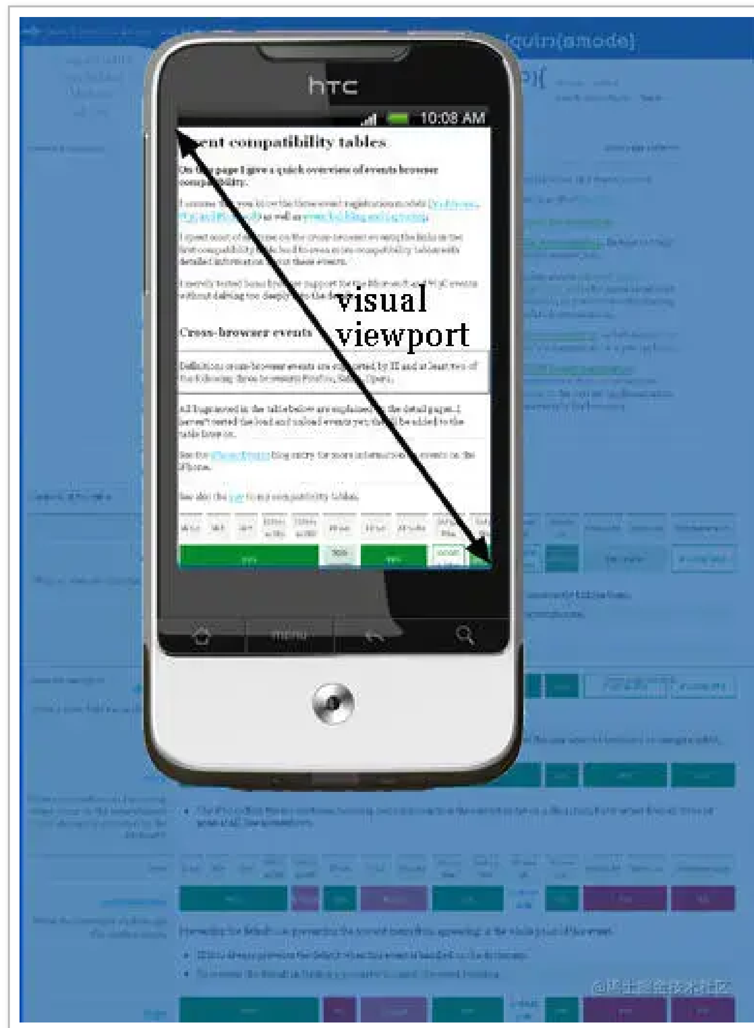
在桌面浏览器中，缩放 100% 的时候，Layout Viewport 宽度等于内容窗口的宽度。（你几乎不会在电脑上见过横向滚动条，除非你调整缩放）
但是在移动端，缩放为 100% 的时候，Layout Viewport 不一定等于内容窗口的大小。当你用手机浏览浏览宽大的网页（这些网页没有采用响应式设计）的时候，你只能一次浏览网页的一个部分，然后通过手指滑动浏览其他部分。这就说明整个网页（Layout Viewport）已经加载出来了，只不过你要一部分一部分地看。




## 1.2.2、visual viewport

visual viewport 为视觉视口，就是显示在屏幕上的网页区域，它往往只显示 layout viewport 的一部分。
visual viewport 就像一台摄像机，layout viewport 就像一张纸，摄像机对准纸的哪个部分，你就能看见哪个

部分。你可以改变摄像机的拍摄区域大小（调整浏览器窗口大小），也可以调整摄像机的距离（调整缩放比例），这些方法都可以改变 visual viewport，但是 layout viewport 始终不变。




## 1.2.3、ideal viewport

ideal viewport 为理想视口，不同的设备有自己不同的 ideal viewport，ideal viewport 的宽度等于移动设备的屏幕宽度，所以其是最适合移动设备的 viewport。只要在 css 中把某一元素的宽度设为 ideal viewport 的宽度 (单位用 px)，那么这个元素的宽度就是设备屏幕的宽度了，也就是宽度为 100% 的效果。 ideal viewport 的意义在于，无论在何种分辨率的屏幕下，那些针对 ideal viewport 而设计的网站，不需要用户手动缩放，也不需要出现横向滚动条，都可以完美的呈现给用户。

## 1.2.4、利用 meta 标签对 viewport 进行控制

移动设备默认的 viewport 是 layout viewport，也就是那个比屏幕要宽的 viewport，但在进行移动设备网站的开发时，我们需要的是 ideal viewport。那么怎么才能得到 ideal viewport 呢?
我们在开发 h5 页面时，最经常见的标签如下所示

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
```

该 meta 标签的作用是让当前 viewport 的宽度等于设备的宽度，同时不允许用户手动缩放。如果你不这样的设定的话，那就会使用那个比屏幕宽的默认 viewport（layout viewport），也就是说会出现横向滚动条。
相关的属性意义如下所示

| width | 设置 layout viewport 的宽度，为一个正整数，或字符串 "width-device" |
| --- | --- |
| height | 设置页面的初始缩放值，为一个数字，可以带小数 |
| initial-scale | 允许用户的最小缩放值，为一个数字，可以带小数 |
| minimum-scale | 允许用户的最大缩放值，为一个数字，可以带小数 |
| maximum-scale | 设置 layout viewport 的高度，这个属性对我们并不重要，很少使用 |
| user-scalable | 是否允许用户进行缩放，值为 "no" 或 "yes", no 代表不允许，yes 代表允许 |

# 二、方案选择

在前端滚滚潮流的历史发展中的不同时期分别出现了一些极具代表性的适配方案，以下分别进行简单介绍。

## 2.1、使用 css 的媒体查询 @media

基于 css 的媒体查询属性 @media 分别为不同屏幕尺寸的移动设备编写不同尺寸的 css 属性，示例如下所示。虽然此方法能在一定程度上解决移动设备适配的问题，但我们也可以看出其存在以下问题，所以其已几乎被历史潮流淘汰。

- 页面上所有的元素都得在不同的 @media 中定义一遍不同的尺寸，这个代价有点高；
- 如果再多一种屏幕尺寸，就得多写一个 @media 查询块；

```
@media only screen and (min-width: 375px) {
  .logo {
    width : 62.5px;
  }
}

@media only screen and (min-width: 360px) {
  .logo {
    width : 60px;
  }
}

@media only screen and (min-width: 320px) {
  .logo {
    width : 53.3333px;
  }
}
```

## 2.2、使用 rem 单位

rem（font size of the root element）是指相对于根元素的字体大小的单位，如果我们设置 html 的 font-size 为 16px，则如果需要设置元素字体大小为 16px，则写为 1rem。但是其还是必须得借助 @media 属性来为不同大小的设备设置不同的 font-size，相对上一种方案，可以减少重复编写相同属性的代价，简单示例如下所示。
我们也能看到该方案存在以下问题：

- 不同的尺寸需要写多个 @media；
- 所有涉及到使用 rem 的地方，全部都需要调用方法 calc()，这个也挺麻烦的；

```
@media only screen and (min-width: 375px) {
  html {
    font-size : 375px;
  }
}

@media only screen and (min-width: 360px) {
  html {
    font-size : 360px;
  }
}

@media only screen and (min-width: 320px) {
  html {
    font-size : 320px;
  }
}

//定义方法：calc
@function calc($val){
    @return $val / 1080;
}

.logo{
        width : calc(180rem);
}
```

## 2.3、flexible 适配方案

在 rem 方案上进行改进，我们可以使用 js 动态来设置根字体，这种方案的典型代表就是 flexible 适配方案 。

### 2.3.1、 使用 rem 模拟 vw 特性适配多种屏幕尺寸

它的核心代码如下所示

```
// set 1rem = viewWidth / 10
function setRemUnit () {
    var rem = docEl.clientWidth / 10
    docEl.style.fontSize = rem + 'px'
```

```
}
setRemUnit();
```

上面的代码中，将 html 节点的 font-size 设置为页面 clientWidth(布局视口) 的 1/10，即 1rem 就等于页面布局视口的 1/10，这就意味着我们后面使用的 rem 都是按照页面比例来计算的。

## 2.3.2、控制 viewport 的 width 和 scale 值适配高倍屏显示

设置 viewport 的 width 为 device-width，改变浏览器 viewport（布局视口和视觉视口）的默认宽度为理想视口宽度，从而使得用户可以在理想视口内看到完整的布局视口的内容。
等比设置 viewport 的 initial-scale、maximum-scale、minimum-scale 的值，从而实现 1 物理像素 = 1 css 像素，以适配高倍屏的显示效果（就是在这个地方规避了大家熟知的“1px 问题”）

```
var metaEL= doc.querySelector('meta[name="viewport"]');
var dpr = window.devicePixelRatio;
var scale = 1 / dpr
metaEl.setAttribute('content', 'width=device-width, initial-scale=' + scale + ', maximum-scale=' + scale + '
```

## 2.3.3、flexible 的缺陷

不可否认 flexible 在兼容性不友好的某个时期还是极大帮助来成千上万的开发者，但是该方案自身是存在一些问题的。

- 由于其缩放的缘故，video 标签的视频频播放器的样式在不同 dpr 的设备上展示差异很大；
- 如果你去研究过 lib-flexible 的源码，那你一定知道 lib-flexible 对安卓手机的特殊处理，即：一律按 dpr = 1 处理；

```
if (isIPhone) {
  // iOS下，对于2和3的屏，用2倍的方案，其余的用1倍方案
  if (devicePixelRatio >= 3 && (!dpr || dpr >= 3)) {
    dpr = 3;
  } else if (devicePixelRatio >= 2 && (!dpr || dpr >= 2)){
    dpr = 2;
  } else {
    dpr = 1;
  }
} else {
  // 其他设备下，仍旧使用1倍的方案
  dpr = 1;
}
```

- 不再兼容 @media 的响应式布局，因为 @media 语法中涉及到的尺寸查询语句，查询的尺寸依据是当前设备的物理像素，和 flexible 的布局理论（即针对不同 dpr 设备等比缩放视口的 scale 值，从而同

时改变布局视口和视觉视口大小）相悖，因此响应式布局在“等比缩放视口大小”的情境下是无法正常工作的；

其实 flexible 方案是在 模拟 viewport 功能，只是随着浏览器的发展及兼容性增强，viewport 已经能兼容绝大部分主流浏览器，并且 flexible 方案自身存在的问题，所有其也已几乎退出历史潮流。引用 lib-flexible 的 github 主页的原话：

> 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 来替代此方案。

## 2.4、viewport 适配方案

由于 viewport 单位得到众多浏览器的兼容，所以目前基于 viewport 的移动端适配方案被各大厂团队所采用。

vw 作为布局单位，从底层根本上解决了不同尺寸屏幕的适配问题，因为每个屏幕的百分比是固定的、可预测、可控制的。 viewport 相关概念如下：

- vw：是 viewport's width 的简写，1vw 等于 window.innerWidth 的 1%；
- vh：和 vw 类似，是 viewport's height 的简写，1vh 等于 window.innerHeihgt 的 1%；
- vmin：vmin 的值是当前 vw 和 vh 中较小的值；
- vmax：vmax 的值是当前 vw 和 vh 中较大的值；

假设我们拿到的视觉稿宽度为 750px，视觉稿中某个字体大小为 75px，则我们的 css 属性只要如下这么写，不需要额外的去用 js 进行设置，也不需要去缩放屏幕等；

```
.logo {
  font-size: 10vw; // 1vw = 750px * 1% = 7.5px
}
```

### 2.4.1、设置 meta 标签

在 html 头部设置 mata 标签如下所示，让当前 viewport 的宽度等于设备的宽度，同时不允许用户手动缩放。

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
```

### 2.4.2、px 自动转换为 vw

设计师一般给宽度大小为 375px 或 750px 的视觉稿，我们采用 vw 方案的话，需要将对应的元素大小单位 px 转换为 vw 单位，这是一项影响开发效率（需要手动计算将 px 转换为 vw）且不利于后续代码维护（css 代码中一堆 vw 单位，不如 px 看的直观）的事情；好在社区提供了 postcss-px-to-viewport 插件，来将 px 自动转换为 vw，相关配置步骤如下：

（1） 安装插件

```
npm install postcss-px-to-viewport --save-dev
```

（2）webpack 配置

官网是使用 glup 进行配置，但是我们项目模版中是使用 webpack 进行 postcss 插件以及相关样式插件的配置，所以我们就使用 webpack 进行配置使用，不需要额外引入 gulp 编译；webpack 相关配置如下，且每个属性表示的意义进行了备注：

```
module.exports = {
  plugins: {
    // ...
    'postcss-px-to-viewport': {
      // options
      unitToConvert: 'px',    // 需要转换的单位，默认为"px"
      viewportWidth: 750,     // 设计稿的视窗宽度
      unitPrecision: 5,       // 单位转换后保留的精度
      propList: ['*', '!font-size'],        // 能转化为 vw 的属性列表
      viewportUnit: 'vw',     // 希望使用的视窗单位
      fontViewportUnit: 'vw', // 字体使用的视窗单位
      selectorBlackList: [],  // 需要忽略的 CSS 选择器，不会转为视窗单位，使用原有的 px 等单位
      minPixelValue: 1,       // 设置最小的转换数值，如果为 1 的话，只有大于 1 的值会被转换
      mediaQuery: false,      // 媒体查询里的单位是否需要转换单位
      replace: true,          // 是否直接更换属性值，而不添加备用属性
      exclude: undefined,     // 忽略某些文件夹下的文件或特定文件，例如 'node_modules' 下的文件
      include: /\/src\//,     // 如果设置了include，那将只有匹配到的文件才会被转换
      landscape: false,       // 是否添加根据 landscapeWidth 生成的媒体查询条件
      landscapeUnit: 'vw',    // 横屏时使用的单位
      landscapeWidth: 1125,   // 横屏时使用的视窗宽度
    },
  },
};
```

相关配置属性，通过注释一目了然其作用，其中需要强调的点为 propList 属性，我们配置了 font-size 不进行转换 vw，也就是说在不同手机屏幕尺寸下的字体大小是一样的。 其中 font-size 是否需要根据屏幕大小做适配，或者怎么做，一直是个争论不休的话题；考虑到我们移动端没有平板的需求，且咨询过团队业务设计师的意见，所以对模版进行以上默认配置；当然如果你的视觉要求你的项目要做字体大小适配，修改 propList 属性的配置即可。

（3）效果展示 我们在项目代码中，进行如下 css 编码：

```
.hello {
  color: #333;
  font-size: 28px;
}
```

启动项目，我们可以看到浏览器渲染的页面中，postcss-px-to-viewport 已经帮我们做进行了 px -> vw 的转换；如下所示：

```
.src-pages-Home-index__hello {
  color: #333;
  font-size: 8.75vw;
}
```

## 2.4.3、标注不需要转换的属性

在项目中，如果设计师要求某一场景不做自适配，需为固定的宽高或大小，这时我们就需要利用 postcss-px-to-viewport 插件的 Ignoring 特性，对不需要转换的 css 属性进行标注，示例如下所示：

- /* px-to-viewport-ignore-next */ —> 下一行不进行转换.
- /* px-to-viewport-ignore */ —> 当前行不进行转换

```
/* example input: */
.class {
  /* px-to-viewport-ignore-next */
  width: 10px;
  padding: 10px;
  height: 10px; /* px-to-viewport-ignore */
}

/* example output: */
.class {
  width: 10px;
  padding: 3.125vw;
  height: 10px;
}
```

## 2.4.4、Retina 屏预留坑位

考虑 Retina 屏场景，可能对图片的高清程度、1px 等场景有需求，所以我们预留判断 Retina 屏坑位。 相关方案如下：在入口的 html 页面进行 dpr 判断，以及 data-dpr 的设置；然后在项目的 css 文件中就可以根据 data-dpr 的值根据不同的 dpr 写不同的样式类；

（1）index.html 文件

```
// index.html 文件
const dpr = devicePixelRatio >= 3? 3: devicePixelRatio >= 2? 2: 1;
document.documentElement.setAttribute('data-dpr', dpr);
```

（2）样式文件

```
[data-dpr="1"] .hello {
  background-image: url(image@1x.jpg);

[data-dpr="2"] .hello {
  background-image: url(image@2x.jpg);
}

[data-dpr="3"] .hello {
  background-image: url(image@3x.jpg);
}
```

# 三、若干特定场景最佳实践

## 3.1、行内样式的场景

**场景**：当你需要写行内样式的代码（style）时， postcss-px-to-viewport 插件 无法进行 px 单位无法转换，需要自己手动计算好 vw；

**最佳实践**：通过添加、修改、删除 className 的方式进行处理此类场景，不直接操作行内样式，这更符合将 js 和 css 隔离开的更佳实践。

## 3.2、1px 的问题

retina 屏下 1px 问题是个常谈的问题，相比较普通屏，retina 屏的 1px 线会显得比较粗，设计美感欠缺；在视觉设计师眼里的 1px 是指设备像素 1px，而如果我们直接写 css 的大小 1px，那在 dpr = 2 时，则等于 2px 设备像素，dpr = 3 时，等于 3px 设备像素。所以对于要求处理 1px 的场景，我们要进行特殊处理。
以下介绍常用的几种方法

### 3.2.1、transform: scale(0.5)

可以使用 transform: scale(0.5) 进行 X、Y 轴的缩放，如下示例所示

```
.class1 {
  height: 1px;
  transform: scaleY(0.5);
}
```

优点是编写简单，但是如果实现上下左右四条边框会比较难搞，并且如果有嵌套存在的话，会对包含的元素产生影响，所以结合 :before 和 :after 来使用。

### 3.2.2、transform: scale(0.5) + :before / :after （推荐）

此种方式能解决例如 标签上下左右边框 1px 的场景，以及有嵌套元素存在的场景，比较通用，示例如下所示

```
.calss1 {
  position: relative;
  &::after {
    content:"";
    position: absolute;
    bottom:0px;
    left:0px;
    right:0px;
    border-top:1px solid #666;
    transform: scaleY(0.5);
  }
}
```

### 3.2.3、box-shadow

利用 css 对阴影处理来模拟边框，示例如下所示，底部一条线，缺点是存在阴影不好看。

```
  .class1 {
    box-shadow: 0 1px 1px -1px rgba(0, 0, 0, 0.5);
  }
```

### 3.2.4、其它

还有如下等方式处理 1px 问题，但不推荐，了解即可

- viewport: 将页面进行缩小处理；
- border-image：切个 1px 图片来模拟；
- background-image：切个 1px 图片来模拟；
- linear-gradient：通过线性渐变，来实现移动端 1px 的线；
- svg：基于矢量图形 (svg) 在不同设备屏幕特性下具有伸缩性。

## 3.3、图片高清的问题

图片高清的问题：

- 适用普通屏的图片在 retina 屏中，图片展示就会显得模糊；
- 适用 retina 屏的图片在普通屏中，图片展示就会缺少色差、没有锐利度，并且浪费带宽； 所以如果对性能、美观要求很高的场景，需要根据 dpr 区分使用对应的图片，我们在文章 viewport 适配方案中针

对 retina 屏预留了 dpr 方案，相关 css 写法如下：

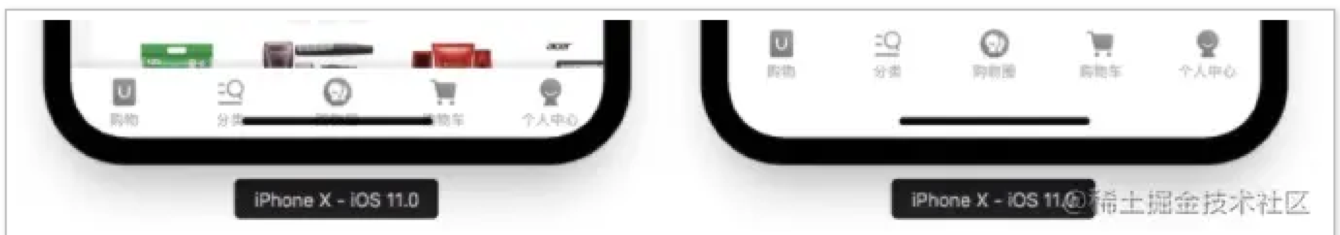
```
[data-dpr="1"] .hello {
  background-image: url(image@1x.jpg);

[data-dpr="2"] .hello {
  background-image: url(image@2x.jpg);
}

[data-dpr="3"] .hello {
  background-image: url(image@3x.jpg);
}
```

# 四、iPhoneX 适配方案

iPhoneX 取消了物理按键，改成底部小黑条，这一改动导致网页出现了比较尴尬的屏幕适配问题。对于网页而言，顶部（刘海部位）的适配问题浏览器已经做了处理，所以我们只需要关注底部与小黑条的适配问题即可（即常见的吸底导航、返回顶部等各种相对底部 fixed 定位的元素）。 比如一些需要贴在底部的按钮，和呼起的 tabBar 和底部弹出框，在 iphoneX 上就会出现被小黑条遮挡内容，或者页面上出现白色空隙的问题。处理前后截图如下所示

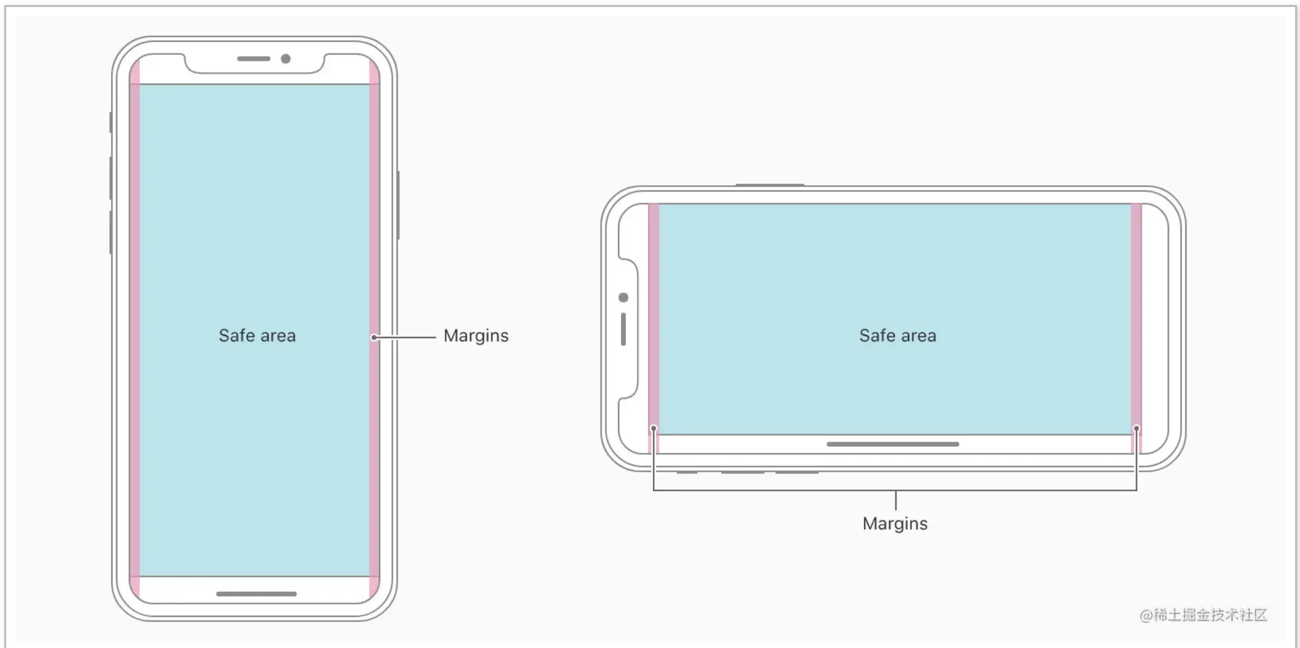


@稀土掘金技术社区

## 4.1、适配之前需要了解的几个新知识

### 4.1.1、安全区域

安全区域指的是一个可视窗口范围，处于安全区域的内容不受圆角（corners）、齐刘海（sensor housing）、小黑条（Home Indicator）影响，如下图蓝色区域：



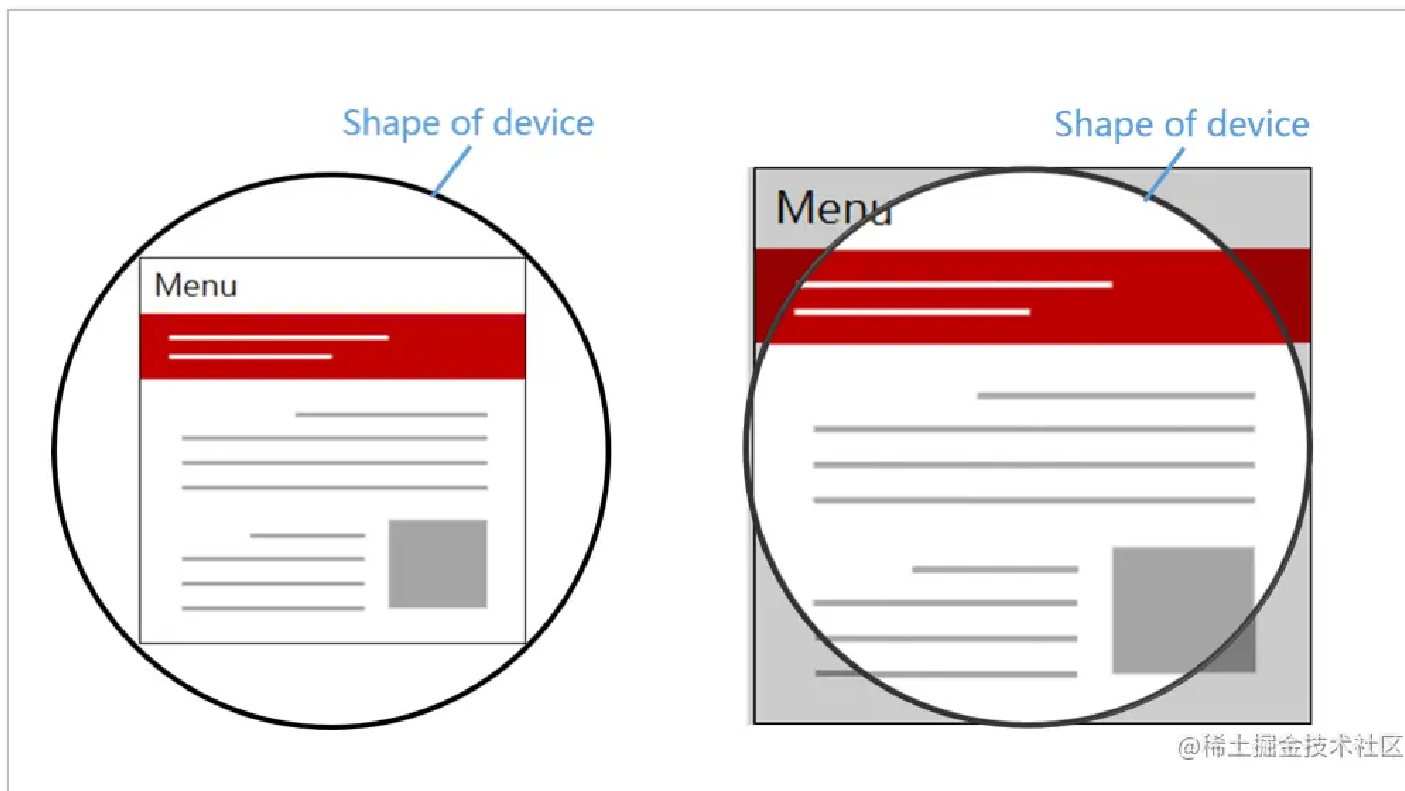

也就是说，我们要做好适配，必须保证页面可视、可操作区域是在安全区域内。 更详细说明，参考文档：
Human Interface Guidelines – iPhoneX

## 4.1.2、viewport–fit

iOS11 新增特性，苹果公司为了适配 iPhoneX 对现有 viewport meta 标签的一个扩展，用于设置网页在可视窗口的布局方式，可设置三个值。

- contain: 可视窗口完全包含网页内容（左图）
- cover：网页内容完全覆盖可视窗口（右图）
- auto：默认值，跟 contain 表现一致

需要注意：网页默认不添加扩展的表现是 viewport–fit=contain，需要适配 iPhoneX 必须设置 viewport–fit=cover，这是适配的关键步骤。更详细说明，参考文档： viewport–fit–descriptor




### 4.1.3、env() 和 constant()

iOS11 新增特性，Webkit 的一个 CSS 函数，用于设定安全区域与边界的距离，有四个预定义的变量：

- safe-area-inset-left：安全区域距离左边边界距离
- safe-area-inset-right：安全区域距离右边边界距离
- safe-area-inset-top：安全区域距离顶部边界距离
- safe-area-inset-bottom：安全区域距离底部边界距离

这里我们只需要关注 safe-area-inset-bottom 这个变量，因为它对应的就是小黑条的高度（横竖屏时值不一样）。

注意：当 viewport-fit=contain 时 env() 是不起作用的，必须要配合 viewport-fit=cover 使用。对于不支持 env() 的浏览器，浏览器将会忽略它。

需要注意的是之前使用的 constant() 在 iOS11.2 之后就不能使用的，但我们还是需要做向后兼容，像这样：

```
padding-bottom: constant(safe-area-inset-bottom); /* 兼容 iOS < 11.2 */
padding-bottom: env(safe-area-inset-bottom); /* 兼容 iOS >= 11.2 */
```

注意：env() 跟 constant() 需要同时存在，而且顺序不能换。 更详细说明，参考文档： Designing Websites for iPhone X

## 4.2、适配步骤

### 4.2.1、设置网页在可视窗口的布局方式

新增 viweport-fit 属性，使得页面内容完全覆盖整个窗口，前面也有提到过，只有设置了 viewport-fit=cover，才能使用 env()

```
<meta name="viewport" content="width=device-width, viewport-fit=cover">
```

### 4.2.2、fixed 完全吸底元素场景的适配

可以通过加内边距 padding 扩展高度：

```
{
  padding-bottom: constant(safe-area-inset-bottom);
  padding-bottom: env(safe-area-inset-bottom);
}
```

或者通过计算函数 calc 覆盖原来高度：

```
{
  height: calc(60px(假设值) + constant(safe-area-inset-bottom));
  height: calc(60px(假设值) + env(safe-area-inset-bottom));
}
```

注意，这个方案需要吸底条必须是有背景色的，因为扩展的部分背景是跟随外容器的，否则出现镂空情况。

还有一种方案就是，可以通过新增一个新的元素（空的颜色块，主要用于小黑条高度的占位），然后吸底元素可以不改变高度只需要调整位置，像这样：

```
{
  margin-bottom: constant(safe-area-inset-bottom);
  margin-bottom: env(safe-area-inset-bottom);
}
```

空的颜色块：

```
{
  position: fixed;
  bottom: 0;
  width: 100%;
  height: constant(safe-area-inset-bottom);
  height: env(safe-area-inset-bottom);
  background-color: #fff;
```

```
}
```

4.2.3、fixed 非完全吸底元素场景的适配

像这种只是位置需要对应向上调整，可以仅通过下外边距 margin-bottom 来处理

```
{
  margin-bottom: constant(safe-area-inset-bottom);
  margin-bottom: env(safe-area-inset-bottom);
}
```

或者，你也可以通过计算函数 calc 覆盖原来 bottom 值：

```
{
  bottom: calc(50px(假设值) + constant(safe-area-inset-bottom));
  bottom: calc(50px(假设值) + env(safe-area-inset-bottom));
}
```

# 五、VW 兼容方案

Android 4.4 之下和 iOS 8 以下的版本有一定的兼容性问题（ps: 几乎绝迹，大家可以统计下你们的用户使用的系统版本占比），但是社区提供了兼容性解决方案，其为 viewport 的 buggyfill： Viewport Units Buggyfill ，可以访问其 github 官网查看。
我们也做了对应的实践，但是考虑到性能，我们项目模版中不会进行引入，有兴趣的同学可以查看以下实践总结；

## 5.1、Viewport Units Buggyfill 引入

viewport-units-buggyfill 主要有两个 JavaScript 文件：viewport-units-buggyfill.js 和 viewport-units-buggyfill.hacks.js。你只需要在你的 HTML 文件中引入这两个文件，比如在 react 项目中的 index.html 引入它们；

```
<script src="//g.alicdn.com/fdilab/lib3rd/viewport-units-buggyfill/0.6.2/??viewport-units-buggyfill.hacks.mi
```

第二步，在 HTML 文件中调用 viewport-units-buggyfill，比如：

```
<script>
    window.onload = function () {
        window.viewportUnitsBuggyfill.init({
            hacks: window.viewportUnitsBuggyfillHacks
```

```
        });
    }
</script>
```

但是为保证 Viewport Units Buggyfill 起作用，我们必须在我们样式文件中用到了 viewport 的单位（vw、vh、vmin 或 vmax ）地方添加 content，如下所示：

```
.my-viewport-units-using-thingie {
  width: 50vmin;
  height: 50vmax;
  top: calc(50vh - 100px);
  left: calc(50vw - 100px);

  /* hack to engage viewport-units-buggyfill */
  content: 'viewport-units-buggyfill; width: 50vmin; height: 50vmax; top: calc(50vh - 100px); left: calc(50v
}
```

## 5.2、postcss-viewport-units 引入

在 1 步骤中，我们人肉引入 content 属性，效率是非常低下的，好在社区提供了 postcss-viewport-units 插件，帮我们自动处理 content：

### 5.2.1、postcss-viewport-units 安装配置

我们执行以下命令，进行 postcss-viewport-units 插件的安装：

```
tnpm i postcss-viewport-units --save-dev
```

在我们的项目配置文件 webpack.config.js 中进行对应的插件引入配置：

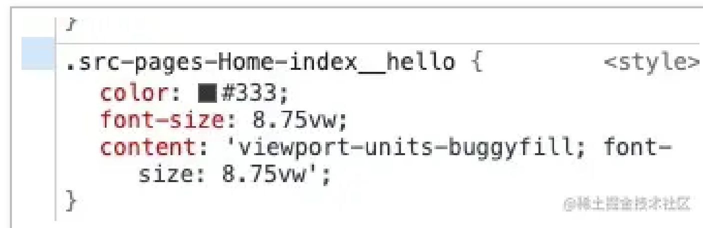
```
{
  loader: 'postcss-loader',
  options: {
    ident: 'postcss',
    plugins: () => [
      // 我们加的配置
      require('postcss-viewport-units'),
    ],
    sourceMap: isProductionEnv,
  },
},
```

### 5.2.2、效果展示

我们在项目代码中，进行如下编码：

```
.hello {
  color: #333;
  font-size: 28px;
}
```

展示的页面中，postcss-viewport-units 已经帮我们添加了 content 属性；如下所示：




博客 github 地址为： github.com/fengshi123/… ，汇总了作者的所有博客，欢迎关注及 star ~

# 六、参考文献：


1. 响应式设计 - 理解设备像素、设备独立像素和 css 像素
2. 移动前端开发之 viewport 的深入理解
3. 使用 Flexible 实现手淘 H5 页面的终端适配
4. VW: 是时候放弃 REM 布局了
5. lib-flexible
6. postcss-px-to-viewport
7. 网页适配 iPhoneX


全文完

本文由 简悦 SimpRead 转码，用以提升阅读体验，原文地址