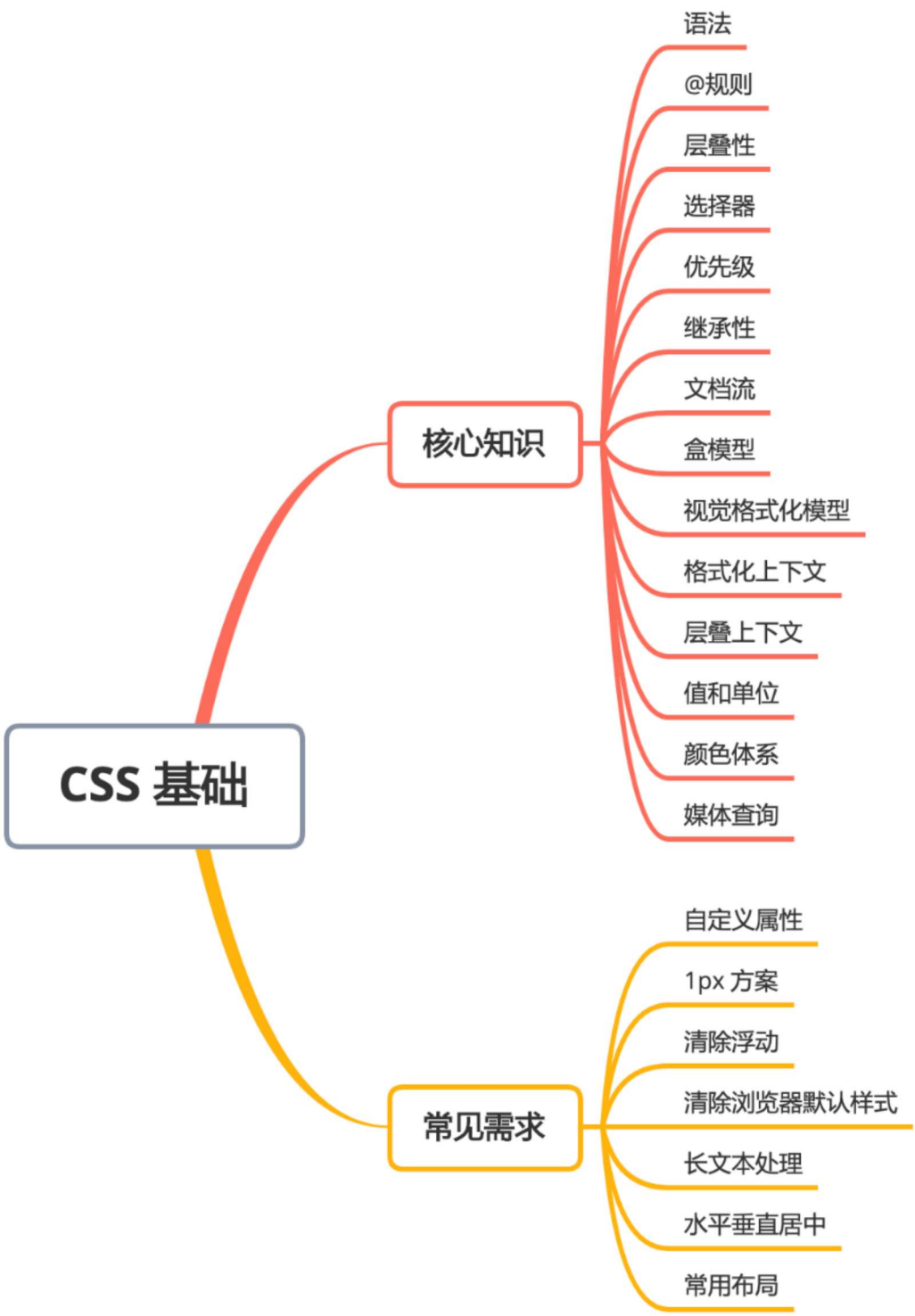
本篇文章围绕了 CSS 的核心知识点和项目中常见的需求来展开。虽然行文偏长，但较基础，适合初级中级前端阅读，阅读的时候请适当跳过已经掌握的部分。



核心概念和知识点

语法

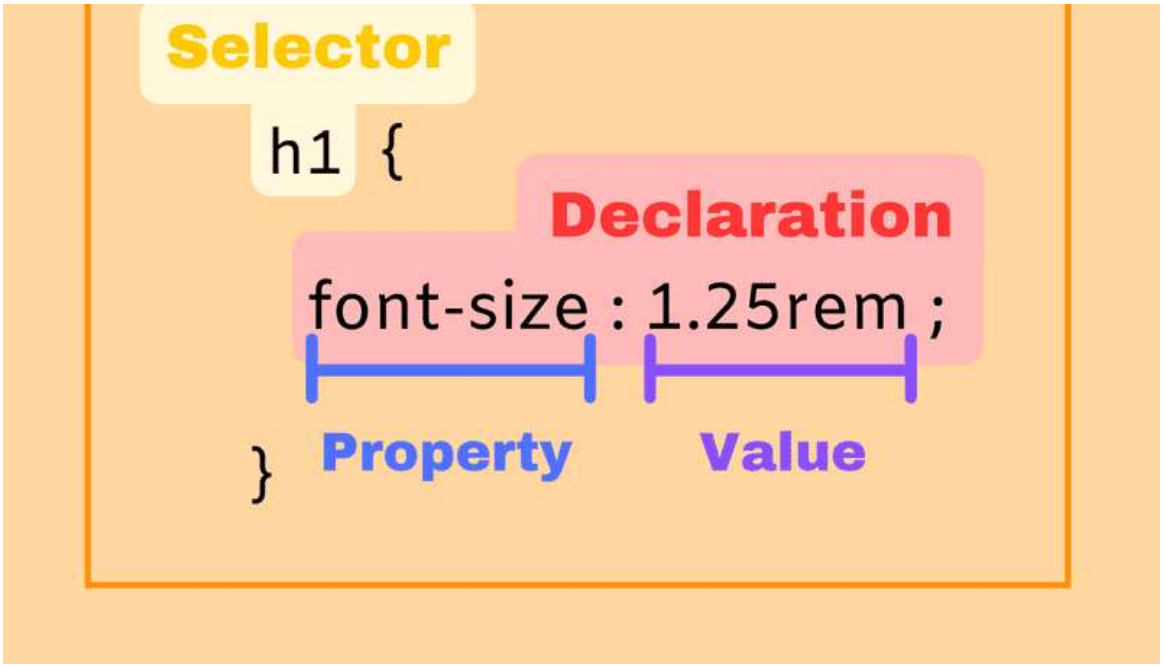
CSS 的核心功能是将 CSS 属性设定为特定的值。一个属性与值的键值对被称为**声明**（declaration）。

```
color: red;
```

而如果将一个或者多个声明用 {} 包裹起来后，那就组成了一个**声明块**（declaration block）。

```
{
    color: red;
    text-align: center;
}
```

声明块如果需要作用到对应的 HTML 元素，那还需要加上**选择器**。选择器和声明块组成了**CSS 规则集**（CSS ruleset），常简称为 CSS 规则。




```
span  {
      color: red;
      text-align: center;
}
```

> 规则集中最后一条声明可以省略分号，但是并不建议这么做，因为容易出错。

CSS 中的**注释**:

```
/* 单行注释 */

/*
```

```
        多行
        注释
*/
```

在 CSS 文件中，除了注释、CSS 规则集以及 @规则 外，定义的一些别的东西都将被浏览器忽略。

@规则

CSS 规则是样式表的主体，通常样式表会包括大量的规则列表。但有时候也需要在样式表中包括其他的一些信息，比如字符集，导入其它的外部样式表，字体等，这些需要专门的语句表示。

而 @规则 就是这样的语句。CSS 里包含了以下 @规则：

> @namespace 告诉 CSS 引擎必须考虑XML命名空间。
>
> @media 如果满足媒体查询的条件则条件规则组里的规则生效。
>
> @page描述打印文档时布局的变化.
>
> @font-face 描述将下载的外部的字体。
>
> @keyframes 描述 CSS 动画的关键帧。

@document 如果文档样式表满足给定条件则条件规则组里的规则生效。 (推延至 CSS Level 4 规范)

除了以上这几个之外，下面还将对几个比较生涩的 @规则 进行介绍。

@charset

\@charset[1] 用于定义样式表使用的字符集。它必须是样式表中的第一个元素。如果有多个 `@charset` 被声明，只有第一个会被使用，而且不能在HTML元素或HTML页面的 `<style>` 元素内使用。

注意：值必须是双引号包裹，且和

```
@charset  "UTF-8";
```

平时写样式文件都没写 @charset 规则，那这个 CSS 文件到底是用的什么字符编码的呢?

某个样式表文件到底用的是什么字符编码，浏览器有一套识别顺序（优先级由高到低）：

文件开头的 **Byte order mark**[2] 字符值，不过一般编辑器并不能看到文件头里的 BOM 值；

HTTP 响应头里的 `content-type` 字段包含的 `charset` 所指定的值，比如：

```
Content-Type: text/css; charset=utf-8
```

CSS 文件头里定义的 @charset 规则里指定的字符编码；

`<link>` 标签里的 charset 属性，该条已在 HTML5 中废除；

默认是 `UTF-8` 。

@import

\@import[3] 用于告诉 CSS 引擎引入一个外部样式表。

link 和 @import 都能导入一个样式文件，它们有什么区别嘛?

link 是 HTML 标签，除了能导入 CSS 外，还能导入别的资源，比如图片、脚本和字体等；而 @import 是 CSS 的语法，只能用来导入 CSS；

link 导入的样式会在页面加载时同时加载，@import 导入的样式需等页面加载完成后再加载；

link 没有兼容性问题，@import 不兼容 ie5 以下；

link 可以通过 JS 操作 DOM 动态引入样式表改变样式，而@import不可以。

@supports

\@supports[4] 用于查询特定的 CSS 是否生效，可以结合 not、and 和 or 操作符进行后续的操作。

```
/* 如果支持自定义属性，则把 body 颜色设置为变
@supports  (--foo:  green) {
      body  {
            color: var(--varName);
      }
}
```

层叠性

层叠样式表，这里的层叠怎么理解呢？其实它是 CSS 中的核心特性之一，用于合并来自多个源的属性值的算法。比如说针对某个 HTML 标签，有许多的 CSS 声明都能作用到的时候，那最后谁应该起作用呢？层叠性说的大概就是这个。

针对不同源的样式，将按照如下的顺序进行层叠，越往下优先级越高：

> 用户代理样式表中的声明(例如，浏览器的默认样式，在没有设置其他样式时使用)。
>
> ~~用户样式表中的常规声明(由用户设置的自定义样式。由于 Chrome 在很早的时候就放弃了用户样式表的功能，所以这里将不再考虑它的排序。)~~。
>
> 作者样式表中的常规声明(这些是我们 Web 开发人员设置的样式)。
>
> 作者样式表中的 !important 声明。
>
> ~~用户样式表中的 !important 声明S~~。

理解层叠性的时候需要结合 CSS 选择器的优先级以及继承性来理解。比如针对同一个选择器，定义在后面的声明会覆盖前面的；作者定义的样式会比默认继承的样式优先级更高。

选择器

CSS 选择器无疑是其核心之一，对于基础选择器以及一些常用伪类必须掌握。下面列出了常用的选择器。 想要获取更多选择器的用法可以看 **MDN CSS Selectors**[5]。

基础选择器

标签选择器： `h1`

类选择器： `.checked`

ID 选择器： `#picker`

通配选择器： `*`

属性选择器

`[attr]` ： 指定属性的元素；

`[attr=val]` ： 属性等于指定值的元素；

`[attr*=val]` ： 属性包含指定值的元素；

`[attr^=val]` ： 属性以指定值开头的元素；

`[attr$=val]` ： 属性以指定值结尾的元素；

`[attr~=val]` ： 属性包含指定值(完整单词)的元素(不推荐使用)；

[attr|=val]：属性以指定值(完整单词)开头的元素(不推荐使用)；

组合选择器

相邻兄弟选择器： A + B

普通兄弟选择器： A ~ B

子选择器： A > B

后代选择器： A B

伪类

条件伪类

:lang()：基于元素语言来匹配页面元素；

:dir()：匹配特定文字书写方向的元素；

:has()：匹配包含指定元素的元素；

:is()：匹配指定选择器列表里的元素；

:not()：用来匹配不符合一组选择器的元素；

行为伪类

:active：鼠标激活的元素；

:hover：鼠标悬浮的元素；

::selection：鼠标选中的元素；

状态伪类

:target ： 当前锚点的元素;

:link ： 未访问的链接元素;

:visited ： 已访问的链接元素;

:focus ： 输入聚焦的表单元素;

:required ： 输入必填的表单元素;

:valid ： 输入合法的表单元素;

:invalid ： 输入非法的表单元素;

:in-range ： 输入范围以内的表单元素;

:out-of-range ： 输入范围以外的表单元素;

:checked ： 选项选中的表单元素;

:optional ： 选项可选的表单元素;

:enabled ： 事件启用的表单元素;

:disabled ： 事件禁用的表单元素;

:read-only ： 只读的表单元素;

:read-write ： 可读可写的表单元素;

:blank ： 输入为空的表单元素;

:current() ： 浏览中的元素;

:past() ： 已浏览的元素;

:future() ： 未浏览的元素;

结构伪类

:root ： 文档的根元素;

:empty ： 无子元素的元素;

:first-letter ： 元素的首字母;

:first-line ： 元素的首行;

:nth-child(n) ： 元素中指定顺序索引的元素;

:nth-last-child(n) ： 元素中指定逆序索引的元素; ;

:first-child ： 元素中为首的元素;

:last-child ： 元素中为尾的元素;

:only-child ： 父元素仅有该元素的元素;

:nth-of-type(n) ： 标签中指定顺序索引的标签;

:nth-last-of-type(n) ： 标签中指定逆序索引的标签;

:first-of-type ： 标签中为首的标签;

:last-of-type ： 标签中为尾标签;

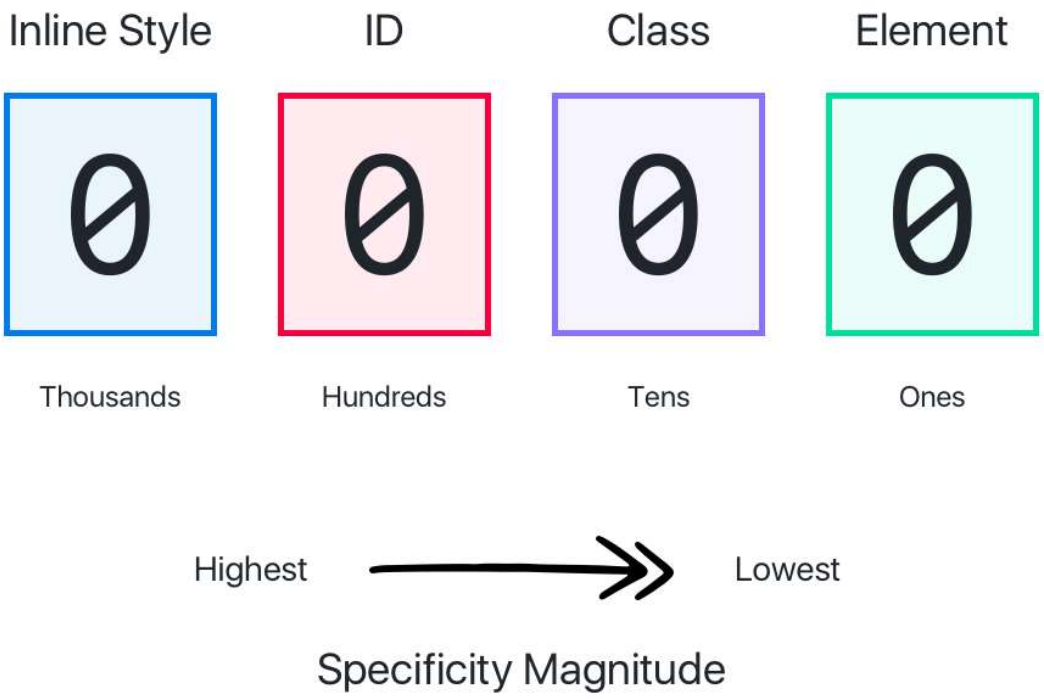
:only-of-type ： 父元素仅有该标签的标签;

伪元素

::before ： 在元素前插入内容;

::after ： 在元素后插入内容；

优先级



优先级就是分配给指定的 CSS 声明的一个权重，它由匹配的选择器中的每一种选择器类型的数值决定。为了记忆，可以把权重分成如下几个等级，数值越大的权重越高：

10000：!important；

01000：内联样式；

00100：ID 选择器；

00010：类选择器、伪类选择器、属性选择器；

00001：元素选择器、伪元素选择器；

00000：通配选择器、后代选择器、兄弟选择器；

可以看到内联样式（通过元素中 style 属性定义的样式）的优先级大于任何选择器；而给属性值加上 `!important` 又可以把优先级提至最高，就是因为它的优先级最高，所以需要谨慎使用它，以下有些使用注意事项：

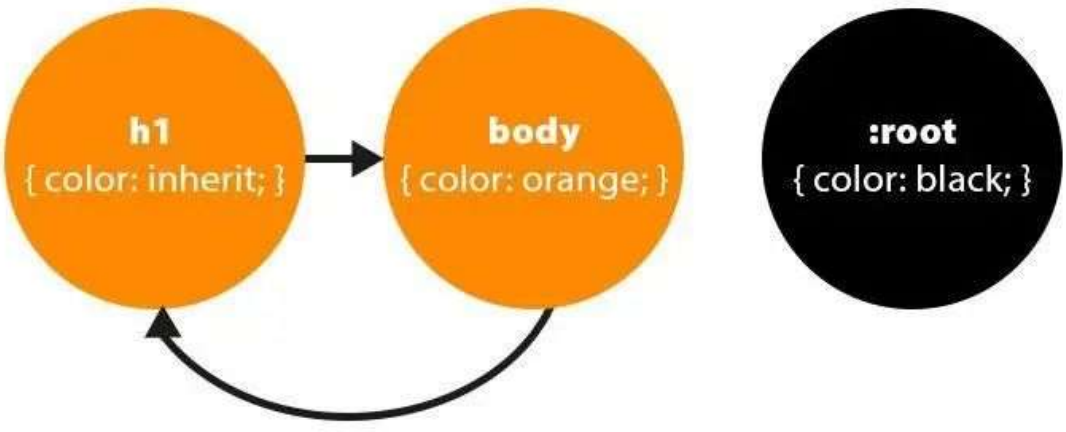
一定要优先考虑使用样式规则的优先级来解决问题而不是 !important；

只有在需要覆盖全站或外部 CSS 的特定页面中使用 !important；

永远不要在你的插件中使用 !important；

永远不要在全站范围的 CSS 代码中使用 !important；

继承性



在 CSS 中有一个很重要的特性就是子元素会继承父元素对应属性计算后的值。比如页面根元素 html 的文本颜色默认是黑色的，页面中的所有其他元素都将继承这个颜色，当申明了如下样式后，H1 文本将变成橙色。

```
body  {
      color: orange;
}
h1  {
      color: inherit;
}
```

设想一下，如果 CSS 中不存在继承性，那么我们就需要为不同文本的标签都设置一下 color，这样一来的后果就是 CSS 的文件大小就会无限增大。

CSS 属性很多，但并不是所有的属性默认都是能继承父元素对应属性的，那哪些属性存在默认继承的行为呢？一定是那些不会影响到页面布局的属性，可以分为如下几类：

- 字体相关：`font-family`、`font-style`、`font-size`、`font-weight` 等；

文本相关：text-align 、 text-indent 、 text-decoration 、 text-shadow 、 letter-spacing 、 word-spacing 、 white-space 、 line-height 、 color 等；

列表相关：list-style 、 list-style-image 、 list-style-type 、 list-style-position 等；

其他属性：visibility 、 cursor 等；

对于其他默认不继承的属性也可以通过以下几个属性值来控制继承行为：

inherit ：继承父元素对应属性的计算值；

initial ：应用该属性的默认值，比如 color 的默认值是 #000 ；

unset ：如果属性是默认可以继承的，则取 inherit 的效果，否则同 initial ；

revert ：效果等同于 unset ，兼容性差。

文档流

在 CSS 的世界中，会把内容按照从左到右、从上到下的顺序进行排列显示。正常情况下会把页面分割成一行一行的显示，而每行又可能由多列组成，所以从视觉上看起来就是从上到下从左到右，而这就是 CSS 中的流式布局，又叫文档流。文档流就像水一

样，能够自适应所在的容器，一般它有如下几个特性：

块级元素默认会占满整行，所以多个块级盒子之间是从上到下排列的；

内联元素默认会在一行里一列一列的排布，当一行放不下的时候，会自动切换到下一行继续按照列排布；

如何脱离文档流呢？

脱流文档流指节点脱流正常文档流后，在正常文档流中的其他节点将忽略该节点并填补其原先空间。文档一旦脱流，计算其父节点高度时不会将其高度纳入，脱流节点不占据空间。有两种方式可以让元素脱离文档流：浮动和定位。

使用浮动（float）会将元素脱离文档流，移动到容器左/右侧边界或者是另一个浮动元素旁边，该浮动元素之前占用的空间将被别的元素填补，另外浮动之后所占用的区域不会和别的元素之间发生重叠；

使用绝对定位（`position: absolute;`）或者固定定位（`position: fixed;`）也会使得元素脱离文档流，且空出来的位置将自动被后续节点填补。

盒模型

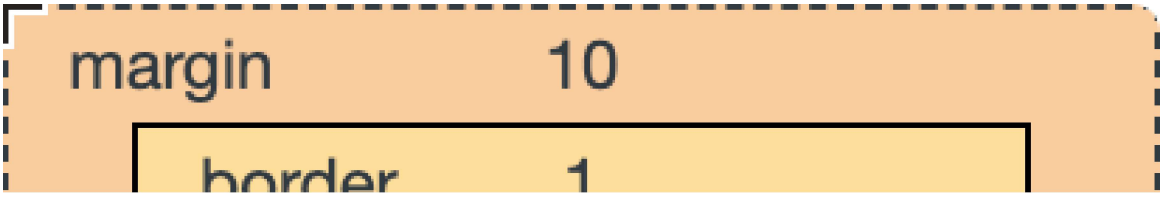
在 CSS 中任何元素都可以看成是一个盒子，而一个盒子是由 4 部分组成的：内容（content）、内边距（padding）、边框（border）和外边距（margin）。

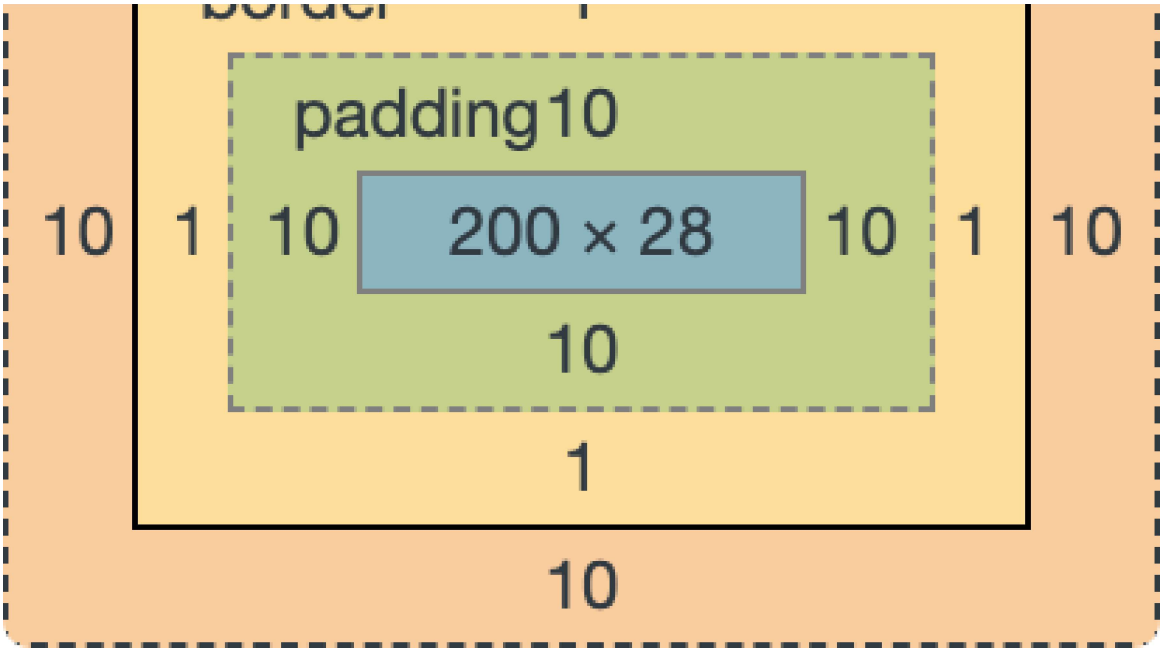
盒模型有 2 种：标准盒模型和 IE 盒模型，本别是由 W3C 和 IExplore 制定的标准。

如果给某个元素设置如下样式：

```
.box  {
      width: 200px;
      height: 200px;
      padding: 10px;
      border: 1px  solid #eee;
      margin: 10px;
}
```

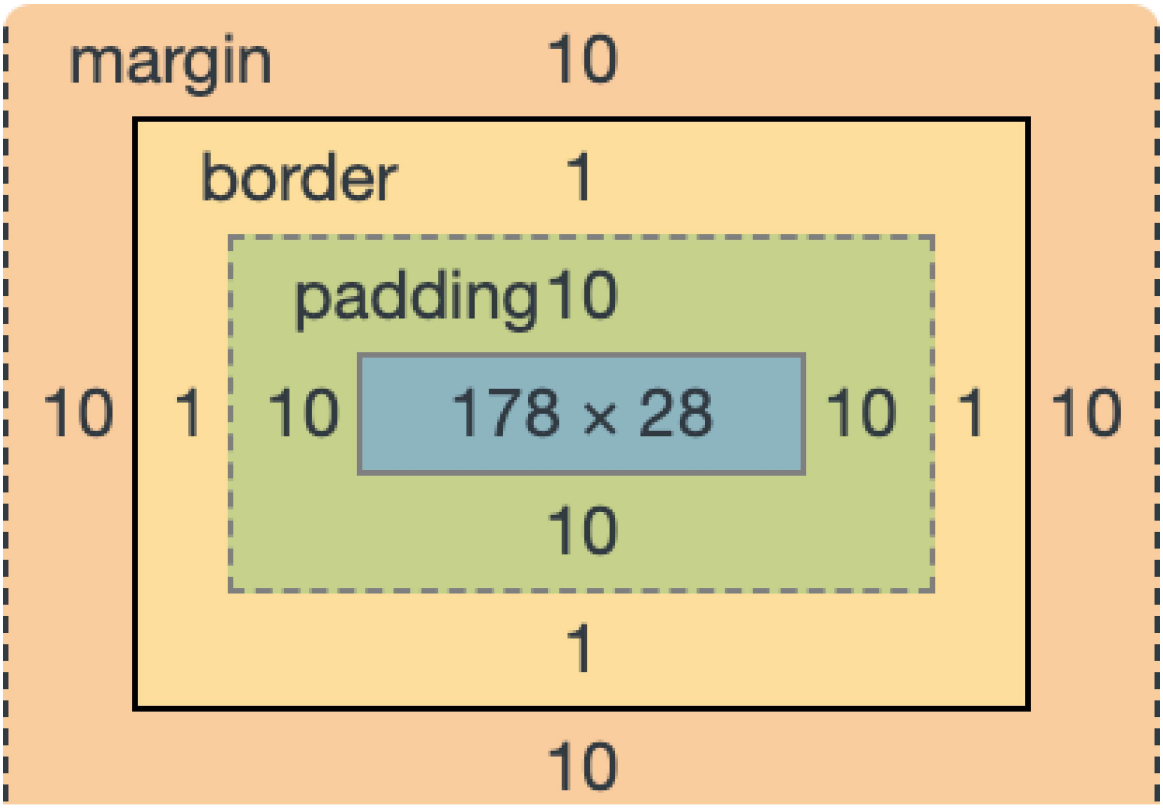
标准盒模型认为：盒子的实际尺寸 = 内容（设置的宽/高） + 内边距 + 边框





所以 `.box` 元素内容的宽度就为 `200px`，而实际的宽度则是 `width + padding-left + padding-right + border-left-width + border-right-width` = 200 + 10 + 10 + 1 + 1 = 222。

IE 盒模型认为：盒子的实际尺寸 = 设置的宽/高 = 内容 + 内边距 + 边框



`.box` 元素所占用的实际宽度为 `200px`，而内容的真实宽度则是 `width - padding-left - padding-right - border-left-width - border-right-width` = 200 - 10 - 10 - 1 - 1 = 178。

现在高版本的浏览器基本上默认都是使用标准盒模型，而像 IE6 这种老古董才是默认使用 IE 盒模型的。

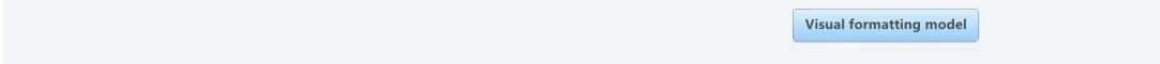
在 CSS3 中新增了一个属性 `box-sizing`，允许开发者来指定盒子使用什么标准，它有 2 个值：

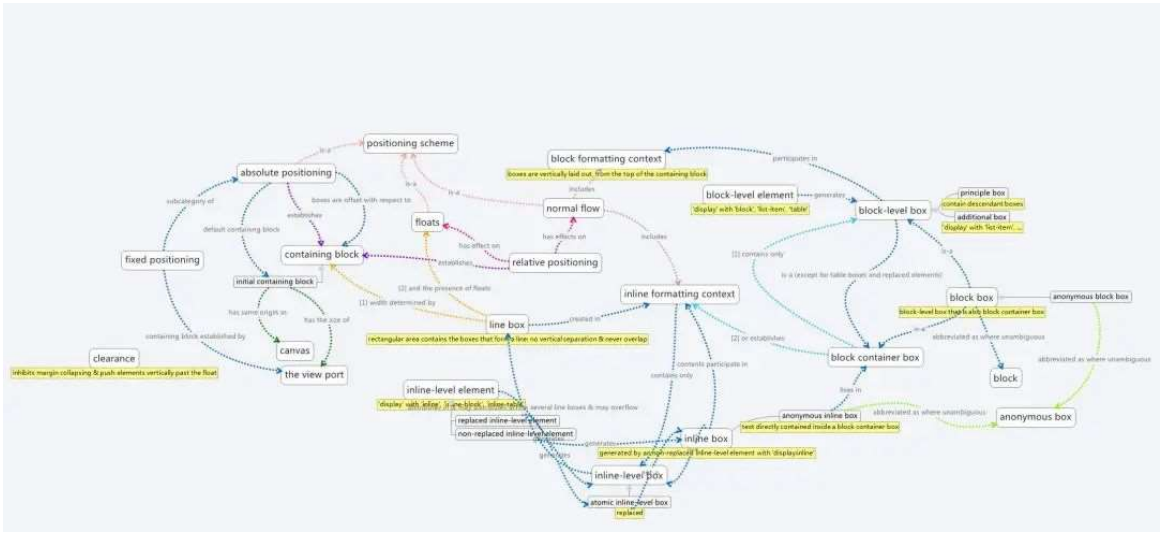
`content-box`：标准盒模型；

`border-box`：IE 盒模型；

视觉格式化模型

视觉格式化模型（Visual formatting model）是用来处理和在视觉媒体上显示文档时使用的计算规则。CSS 中一切皆盒子，而视觉格式化模型简单来理解就是规定这些盒子应该怎么样放置到页面中去，这个模型在计算的时候会依赖到很多的因素，比如：盒子尺寸、盒子类型、定位方案（是浮动还是定位）、兄弟元素或者子元素以及一些别的因素。





Visual formatting model

从上图中可以看到视觉格式化模型涉及到的内容很多，有兴趣深入研究的可以结合上图看这个 W3C 的文档 **Visual formatting model**[6]。所以这里就简单介绍下盒子类型。

Short 'display'	Full 'display'	Generated box
'none'	—	subtree omitted from box tree
'contents'	—	element replaced by contents in box tree
'block'	'block flow'	block-level block container aka block box
'flow-root'	'block flow-root'	block-level block container that establishes a new block formatting context (BFC)
'inline'	'inline flow'	inline box
'inline-block'	'inline flow-root'	inline-level block container
'run-in'	'run-in flow'	run-in box (inline box with special box-tree-munging rules)
'list-item'	'block flow list-item'	block box with additional marker box
'inline list-item'	'inline flow list-item'	inline box with additional marker box

display 屬性

block-level 元素
本身對外參與 BFC

對內建立一個塊格式化上下文(BFC)

inline-level 元素
本身對外參與 IFC

盒子类型由 display 决定，同时给一个元素设置 display 后，将会决定这个盒子的 2 个显示类型（display type）：

outer display type（对外显示）：决定了该元素本身是如何布局的，即参与何种格式化上下文；

inner display type（对内显示）：其实就相当于把该元素当成了容器，规定了其内部子元素是如何布局的，参与何种格式化上下文；

outer display type

对外显示方面，盒子类型可以分成 2 类：block-level box（块级盒子） 和 inline-level box（行内级盒子）。

依据上图可以列出都有哪些块级和行内级盒子：

块级盒子：display 为 block、list-item、table、flex、grid、flow-root 等；

行内级盒子：display 为 inline、inline-block、inline-table 等；

所有块级盒子都会参与 BFC，呈现垂直排列；而所有行内级盒子都参会 IFC，呈现水平排列。

除此之外，block、inline 和 inline-block 还有什么更具体的区别呢?

block

占满一行，默认继承父元素的宽度；多个块元素将从上到下进行排列；

设置 width/height 将会生效；

设置 padding 和 margin 将会生效；

inline

不会占满一行，宽度随着内容而变化；多个 inline 元素将按照从左到右的顺序在一行里排列显示，如果一行显示不下，则自动换行；

设置 width/height 将不会生效；

设置竖直方向上的 padding 和 margin 将不会生效；

inline-block

是行内块元素，不单独占满一行，可以看成是能够在一行里进行左右排列的块元素；

设置 width/height 将会生效；

设置 padding 和 margin 将会生效；

inner display type

对内方面，其实就是把元素当成了容器，里面包裹着文本或者其他子元素。container box 的类型依据 display 的值不同，分为 4 种：

block container：建立 BFC 或者 IFC；

flex container：建立 FFC；

grid container：建立 GFC;

ruby container：接触不多，不做介绍。

值得一提的是如果把 img 这种替换元素（replaced element）申明为 block 是不会产生 container box 的，因为替换元素比如 img 设计的初衷就仅仅是通过 src 把内容替换成图片，完全没考虑过会把它当成容器。

格式化上下文

格式化上下文（Formatting Context）是 CSS2.1 规范中的一个概念，大概说的是页面中的一块渲染区域，规定了渲染区域内部的子元素是如何排版以及相互作用的。

不同类型的盒子有不同格式化上下文，大概有这 4 类：

BFC (Block Formatting Context) 块级格式化上下文；

IFC (Inline Formatting Context) 行内格式化上下文；

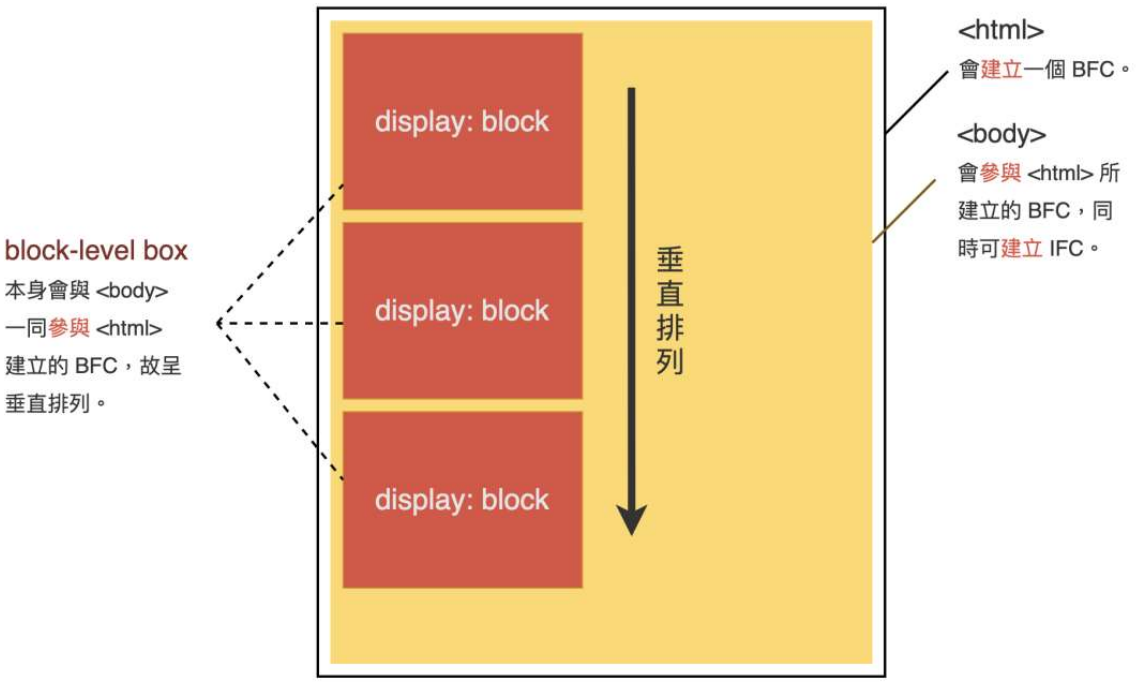
FFC (Flex Formatting Context) 弹性格式化上下文；

GFC (Grid Formatting Context) 格栅格式化上下文；

其中 BFC 和 IFC 在 CSS 中扮演着非常重要的角色，因为它们直接影响了网页布局，所以需要深入理解其原理。

BFC

块格式化上下文，它是一个独立的渲染区域，只有块级盒子参与，它规定了内部的块级盒子如何布局，并且与这个区域外部毫不相干。



图来源于 yachen168

BFC 渲染规则

内部的盒子会在垂直方向，一个接一个地放置；

盒子垂直方向的距离由 margin 决定，属于同一个 BFC 的两个相邻盒子的 margin 会发生重叠；

每个元素的 margin 的左边，与包含块 border 的左边相接触(对于从左往右的格式化，否则相反)，即使存在浮动也是如此；

BFC 的区域不会与 float 盒子重叠；

BFC 就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。

计算 BFC 的高度时，浮动元素也参与计算。

如何创建 BFC?

根元素：html

非溢出的可见元素：overflow 不为 visible

设置浮动：float 属性不为 none

设置定位：position 为 absolute 或 fixed

定义成块级的非块级元素：display: inline-block/table-cell/table-caption/flex/inline-flex/grid/inline-grid

BFC 应用场景

1、 自适应两栏布局

应用原理：BFC 的区域不会和浮动区域重叠，所以就可以把侧边栏固定宽度且左浮动，而对右侧内容触发 BFC，使得它的宽度自适应该行剩余宽度。


```
<div  class="layout">
      <div  class="aside">aside</div>
      <div  class="main">main</div>
</div>
```

```
.aside  {
      float: left;
      width: 100px;
}
.main  {
      <!-- 触发 BFC -->
      overflow: auto;
}
```

2、清除内部浮动

浮动造成的问题就是父元素高度坍塌，所以清除浮动需要解决的问题就是让父元素的高度恢复正常。而用 BFC 清除浮动的原理就是：计算 BFC 的高度时，浮动元素也参与计算。只要触发父元素的 BFC 即可。


```
.parent  {
      overflow: hidden;
}
```

3、 防止垂直 margin 合并

BFC 渲染原理之一：同一个 BFC 下的垂直 margin 会发生合并。所以如果让 2 个元素不在同一个 BFC 中即可阻止垂直 margin 合并。那如何让 2 个相邻的兄弟元素不在同一个 BFC 中呢？可以给其中一个元素外面包裹一层，然后触发其包裹层的 BFC，这样一来 2 个元素就不会在同一个 BFC 中了。

b

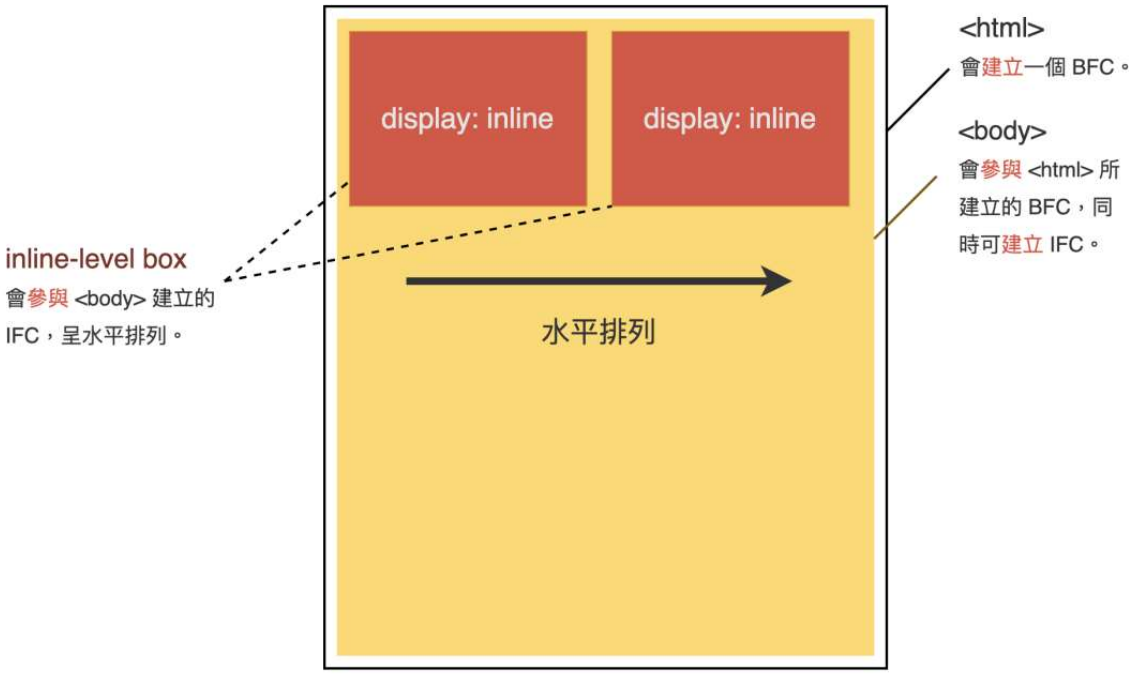
```
<div  class="layout">
      <div  class="a">a</div>
      <div  class="contain-b">
            <div  class="b">b</div>
      </div>
</div>
```

```
.demo3  .a,
.demo3  .b  {
      border: 1px  solid #999;
      margin: 10px;
}
.contain-b  {
      overflow: hidden;
}
```

针对以上 3 个 示例，可以结合这个 **BFC 应用示例** [7] 配合观看更佳。

IFC

IFC 的形成条件非常简单，块级元素中仅包含内联级别元素，需要注意的是当IFC中有块级元素插入时，会产生两个匿名块将父元素分割开来，产生两个IFC。



IFC 渲染规则

子元素在水平方向上一个接一个排列，在垂直方向上将以容器顶部开始向下排列；

节点无法声明宽高，其中 margin 和 padding 在水平方向有效在垂直方向无效；

节点在垂直方向上以不同形式对齐；

能把在一行上的框都完全包含进去的一个矩形区域，被称为该行的线盒（line box）。线盒的宽度是由包含块（containing box）和与其中的浮动来决定；

IFC 中的 line box 一般左右边贴紧其包含块，但 float 元素会优先排列。

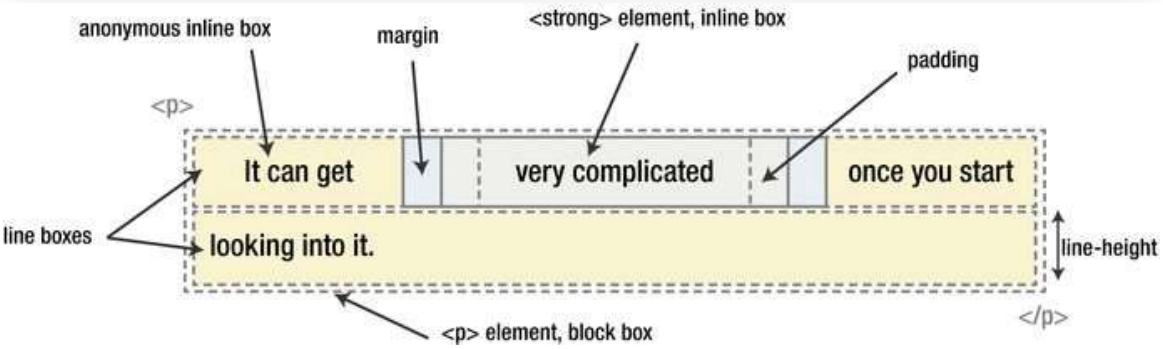
IFC 中的 line box 高度由 line-height 计算规则来确定，同个 IFC 下的多个 line box 高度可能会不同；

当内联级盒子的总宽度少于包含它们的 line box 时，其水平渲染规则由 text-align 属性值来决定；

当一个内联盒子超过父元素的宽度时，它会被分割成多盒子，这些盒子分布在多个 line box 中。如果子元素未设置强制换行的情况下，inline box 将不可被分割，将会溢出父元素。

针对如上的 IFC 渲染规则，你是不是可以分析下下面这段代码的 IFC 环境是怎么样的呢?

```
<p>It can get <strong>very complicated</stor
```



对应上面这样一串 HTML 分析如下：

p 标签是一个 block container，对内将产生一个 IFC；

由于一行没办法显示完全，所以产生了 2 个线盒（line box）；线盒的宽度就继承了 p 的宽度；高度是由里面的内联盒子的 line-height 决定；

It can get：匿名的内联盒子；

very complicated：strong 标签产生的内联盒子；

once you start：匿名的内联盒子；

looking into it.：匿名的内联盒子。

IFC 应用场景

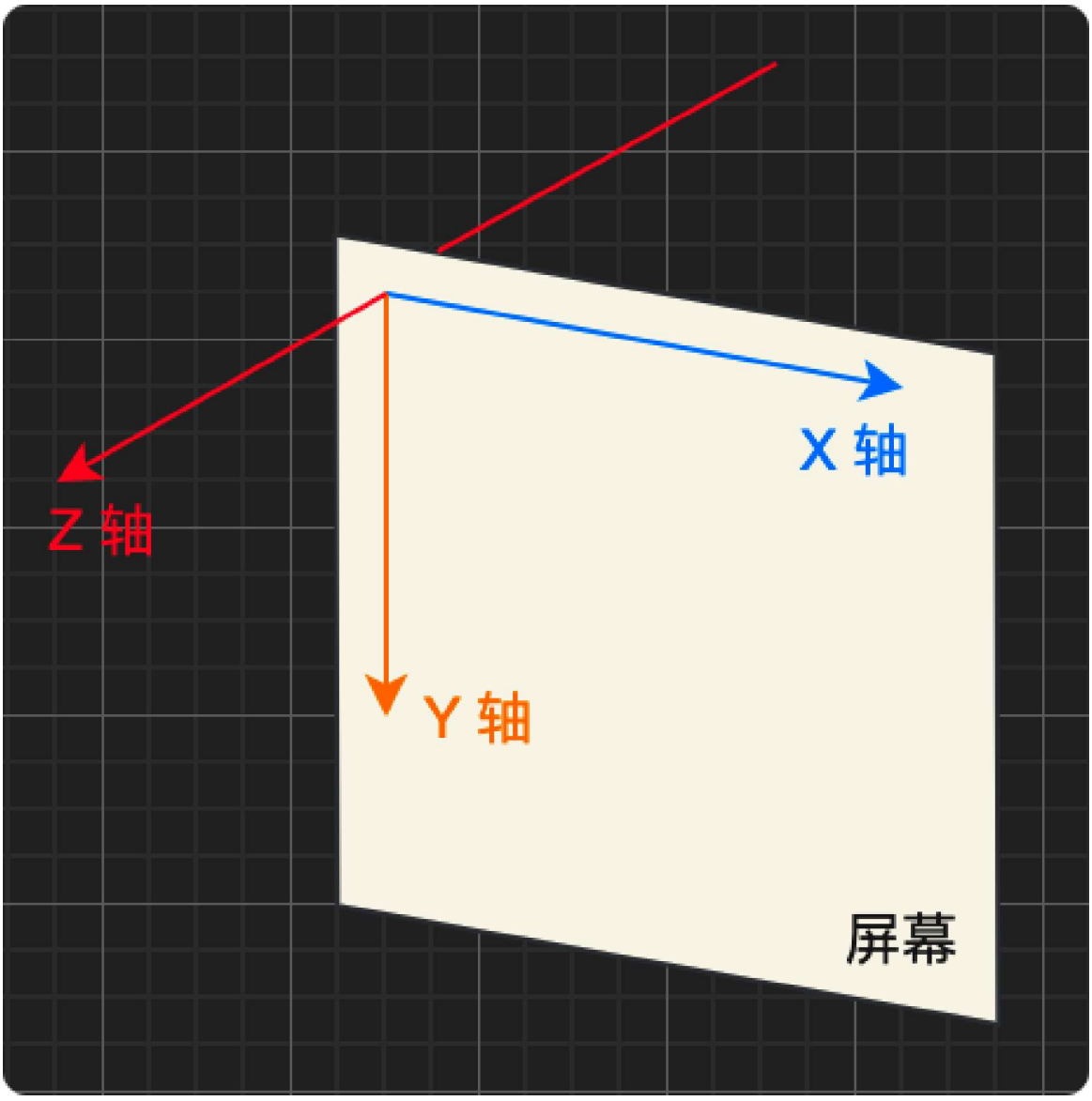
水平居中：当一个块要在环境中水平居中时，设置其为 inline-block 则会在外层产生 IFC，通过 text-align 则可以使其水平居中。

垂直居中：创建一个 IFC，用其中一个元素撑开父元素的高度，然后设置其 vertical-align: middle，其他行内元素则可以在此父元素下垂直居中。

偷个懒，demo 和图我就不做了。

层叠上下文

在电脑显示屏幕上的显示的页面其实是一个三维的空间，水平方向是 X 轴，竖直方向是 Y 轴，而屏幕到眼睛的方向可以看成是 Z 轴。众 HTML 元素依据自己定义的属性的优先级在 Z 轴上按照一定的顺序排开，而这其实就是层叠上下文所要描述的东西。



-w566

我们对层叠上下文的第一印象可能要来源于 z-index，认为它的值越大，距离屏幕观察者就越近，

那么层叠等级就越高，事实确实是这样的，但层叠上下文的内容远非仅仅如此：

- z-index 能够在层叠上下文中对元素的堆叠顺序其作用是必须配合定位才可以；
- 除了 z-index 之外，一个元素在 Z 轴上的显示顺序还受层叠等级和层叠顺序影响；

在看层叠等级和层叠顺序之前，我们先来看下如何产生一个层叠上下文，特定的 HTML 元素或者 CSS 属性产生层叠上下文，MDN 中给出了这么一个列表，符合以下任一条件的元素都会产生层叠上下文：

- html 文档根元素
- 声明 position: absolute/relative 且 z-index 值不为 auto 的元素；
- 声明 position: fixed/sticky 的元素；
- flex 容器的子元素，且 z-index 值不为 auto；
- grid 容器的子元素，且 z-index 值不为 auto；
- opacity 属性值小于 1 的元素；
- mix-blend-mode 属性值不为 normal 的元素；
- 以下任意属性值不为 none 的元素：
  - transform
  - filter

perspective

clip-path

mask / mask-image / mask-border

isolation 属性值为 isolate 的元素；

-webkit-overflow-scrolling 属性值为 touch 的元素；

will-change 值设定了任一属性而该属性在 non-initial 值时会创建层叠上下文的元素；

contain 属性值为 layout、paint 或包含它们其中之一的合成值（比如 contain: strict、contain: content）的元素。

层叠等级

层叠等级指节点在三维空间 Z 轴上的上下顺序。它分两种情况：

在同一个层叠上下文中，它描述定义的是该层叠上下文中的层叠上下文元素在 Z 轴上的上下顺序；

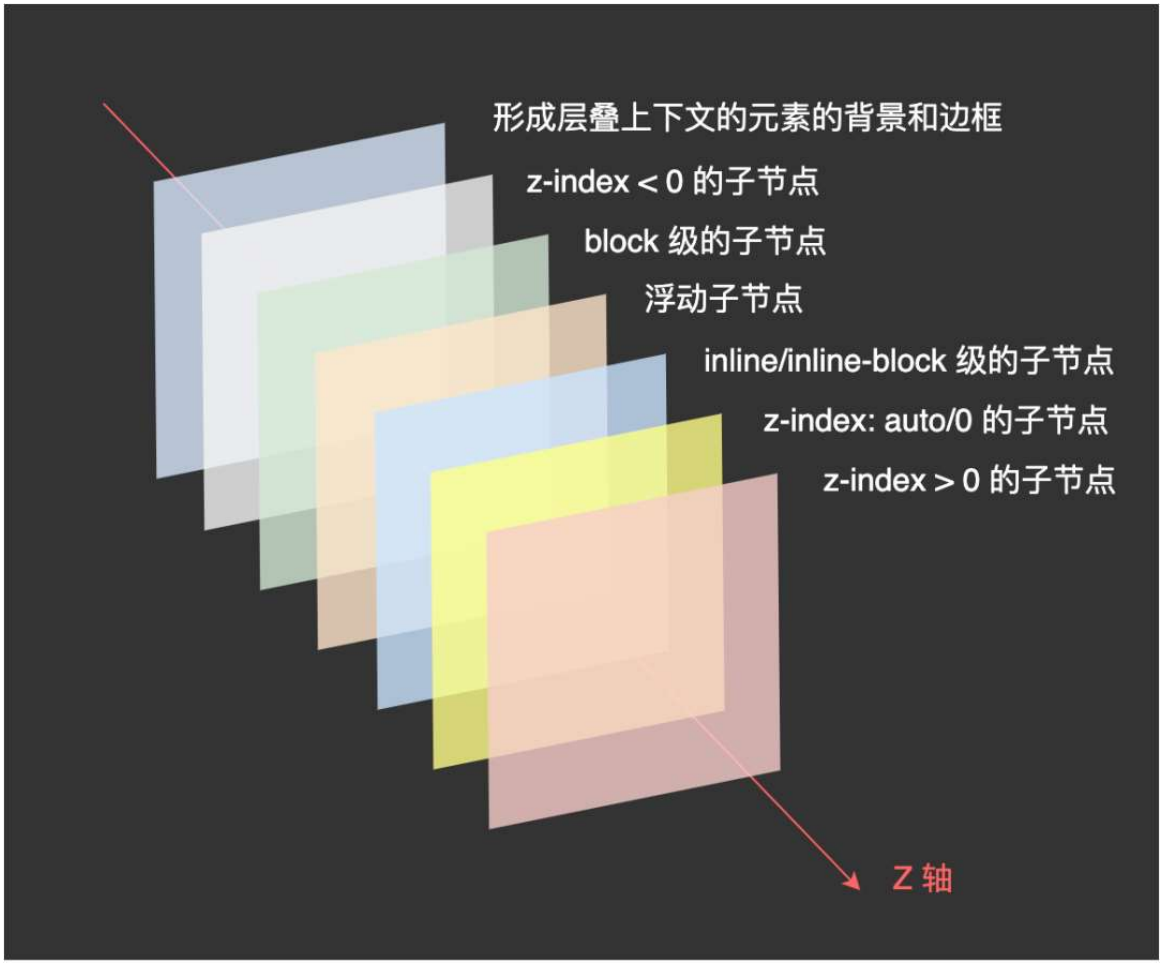
在其他普通元素中，它描述定义的是这些普通元素在 Z 轴上的上下顺序；

普通节点的层叠等级优先由其所在的层叠上下文决定，层叠等级的比较只有在当前层叠上下文中才有意

义，脱离当前层叠上下文的比较就变得无意义了。

层叠顺序

在同一个层叠上下文中如果有多个元素，那么他们之间的层叠顺序是怎么样的呢？



以下这个列表越往下层叠优先级越高，视觉上的效果就是越容易被用户看到（不会被其他元素覆盖）：

层叠上下文的 border 和 background

z-index < 0 的子节点

标准流内块级非定位的子节点

浮动非定位的子节点

标准流内行内非定位的子节点

z-index: auto/0 的子节点

z-index > 0的子节点

如何比较两个元素的层叠等级?

在同一个层叠上下文中，比较两个元素就是按照上图的介绍的层叠顺序进行比较。

如果不在同一个层叠上下文中的时候，那就需要比较两个元素分别所处的层叠上下文的等级。

如果两个元素都在同一个层叠上下文，且层叠顺序相同，则在 HTML 中定义越后面的层叠等级越高。

值和单位

CSS 的声明是由属性和值组成的，而值的类型有许多种：

数值：长度值 ，用于指定例如元素 width、border-width、font-size 等属性的值；

百分比：可以用于指定尺寸或长度，例如取决于父容器的 width、height 或默认的 font-size；

颜色：用于指定 background-color、color 等；

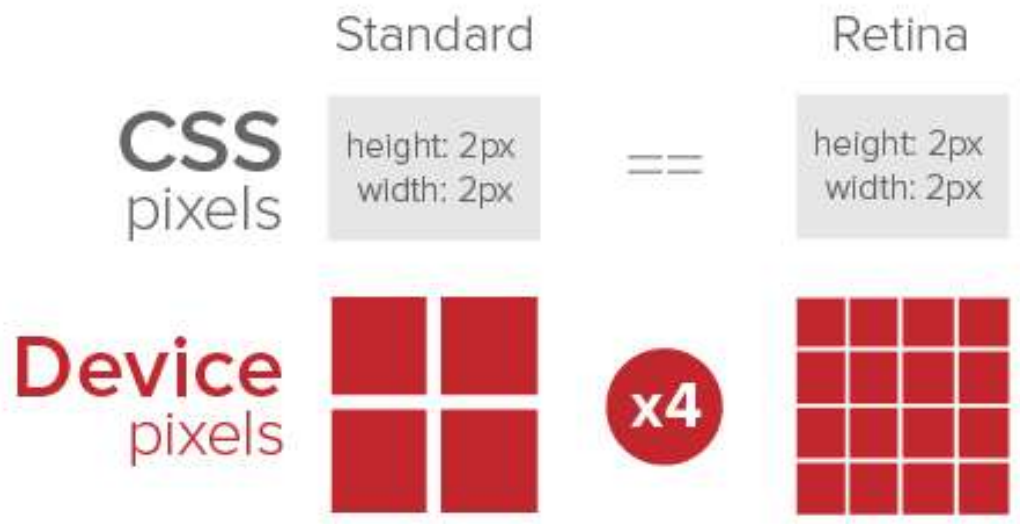
坐标位置：以屏幕的左上角为坐标原点定位元素的位置，比如常见的 background-position、top、right、bottom 和 left 等属性；

函数：用于指定资源路径或背景图片的渐变，比如 url()、linear-gradient() 等；

而还有些值是需要带单位的，比如 width: 100px，这里的 px 就是表示长度的单位，长度单位除了 px 外，比较常用的还有 em、rem、vw/vh 等。那他们有什么区别呢？又应该在什么时候使用它们呢？

px

屏幕分辨率是指在屏幕的横纵方向上的像素点数量，比如分辨率 1920×1080 意味着水平方向含有 1920 个像素数，垂直方向含有 1080 个像素数。



而 px 表示的是 CSS 中的像素，在 CSS 中它是绝对的长度单位，也是最基础的单位，其他长度单位会自动被浏览器换算成 px。但是对于设备而言，它其实又是相对的长度单位，比如宽高都为 2px，在正常的屏幕下，其实就是 4 个像素点，而在设备像素比 (devicePixelRatio) 为 2 的 Retina 屏幕下，它就有 16 个像素点。所以屏幕尺寸一致的情况下，屏幕分辨率越高，显示效果就越细腻。

讲到这里，还有一些相关的概念需要理清下：

设备像素（Device pixels）

设备屏幕的物理像素，表示的是屏幕的横纵有多少像素点；和屏幕分辨率是差不多的意思。

设备像素比（DPR）

设备像素比表示 1 个 CSS 像素等于几个物理像素。

计算公式：DPR = 物理像素数 / 逻辑像素数；

在浏览器中可以通过 window.devicePixelRatio 来获取当前屏幕的 DPR。

像素密度（DPI/PPI）

像素密度也叫显示密度或者屏幕密度，缩写为 DPI(Dots Per Inch) 或者 PPI(Pixel Per Inch)。从技术角度说，PPI 只存在于计算机显示领域，而 DPI 只出现于打印或印刷领域。

计算公式：像素密度 = 屏幕对角线的像素尺寸 / 物理尺寸

比如，对于分辨率为 750 * 1334 的 iPhone 6 来说，它的像素密度为：

```
Math.sqrt(750 * 750 + 1334 * 1334) / 4.7
```

设备独立像素（DIP）

DIP 是特别针对 Android设备而衍生出来的，原因是安卓屏幕的尺寸繁多，因此为了显示能尽量和设备无关，而提出的这个概念。它是基于屏幕密度而计算的，认为当屏幕密度是 160 的时候，px = DIP。

计算公式：dip = px * 160 / dpi

em

em 是 CSS 中的相对长度单位中的一个。居然是相对的，那它到底是相对的谁呢？它有 2 层意思：

> 在 font-size 中使用是相对于**父元素**的 font-size 大小，比如父元素 font-size: 16px，当给子元素指定 font-size: 2em 的时候，经过计算后它的字体大小会是 32px；
>
> 在其他属性中使用是相对于自身的字体大小，如 width/height/padding/margin 等；

我们都知道每个浏览器都会给 HTML 根元素 html 设置一个默认的 font-size，而这个值通常是 16px。这也就是为什么 1em = 16px 的原因所在了。

em 在计算的时候是会层层计算的，比如：

```
<div>
      <p></p>
</div>
```

```
div  { font-size: 2em; }
p  { font-size: 2em; }
```

对于如上一个结构的 HTML，由于根元素 html 的字体大小是 16px，所以 p 标签最终计算出来后的字体大小会是 16 * 2 * 2 = 64px

rem

rem(root em) 和 em 一样，也是一个相对长度单位，不过 rem 相对的是 HTML 的根元素 html。

rem 由于是基于 html 的 font-size 来计算，所以通常用于自适应网站或者 H5 中。

比如在做 H5 的时候，前端通常会让 UI 给 750px 宽的设计图，而在开发的时候可以基于 iPhone X 的尺寸 375px * 812px 来写页面，这样一来的话，就可以用下面的 JS 依据当前页面的视口宽度自动计算出根元素 html 的基准 font-size 是多少。

```
(function (doc, win) {
    var docEl = doc.documentElement,
        resizeEvt = 'orientationchange'
        psdWidth = 750,    // 设计图宽度
        recalc = function () {
            var clientWidth = docE
```

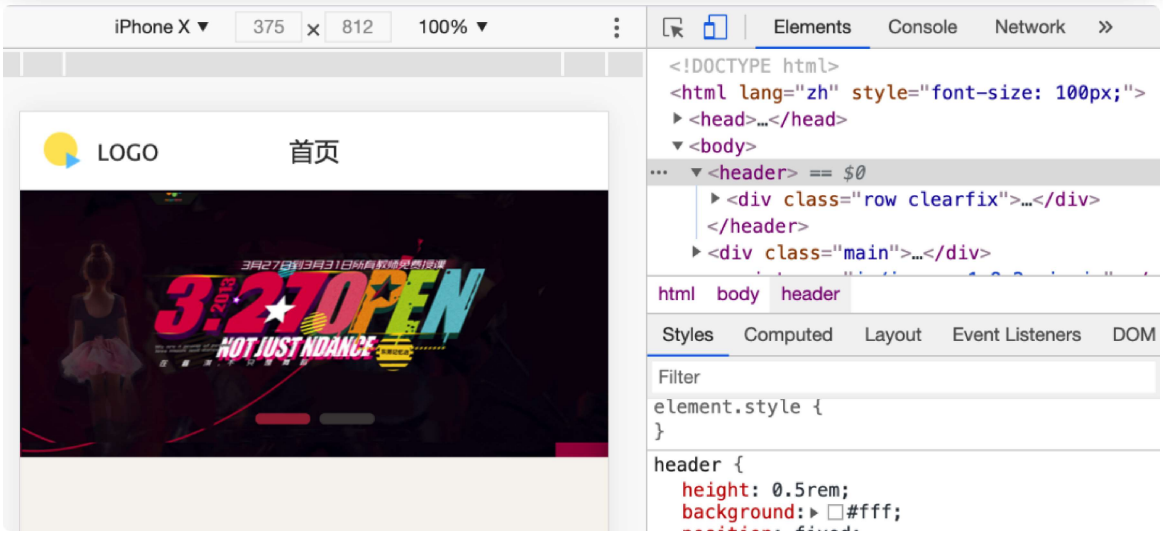
```
                if  ( !clientWidth ) re
                if  ( clientWidth >= 64
                        docEl.style.fon
                } else  {
                        docEl.style.fon
                }
        };

        if  ( !doc.addEventListener ) return;
        // 绑定事件的时候最好配合防抖函数
        win.addEventListener( resizeEvt, debou
        doc.addEventListener( 'DOMContentLoade

        function  debounce(func, wait) {
                var  timeout;
                return  function  () {
                        var  context = this;
                        var  args = arguments;
                        clearTimeout(timeout)
                        timeout = setTimeout(fu
                                func.apply(cont
                        }, wait);
                }
        }
})(document, window);
```

比如当视口是 375px 的时候，经过计算 html 的 font-size 会是 100px，这样有什么好处呢？好处就是方便写样式，比如从设计图量出来的 header 高度是 50px 的，那我们写样式的时候就可以直接写：

```
header  {
        height: 0.5rem;
}
```



每个从设计图量出来的尺寸只要除于 100 即可得到当前元素的 rem 值，都不用经过计算，非常方便。偷偷告诉你，如果你把上面那串计算 html 标签 font-size 的 JS 代码中的 200 替换成 2，那在计算 rem 的时候就不需要除于 100 了，从设计图量出多大 px，就直接写多少个 rem。

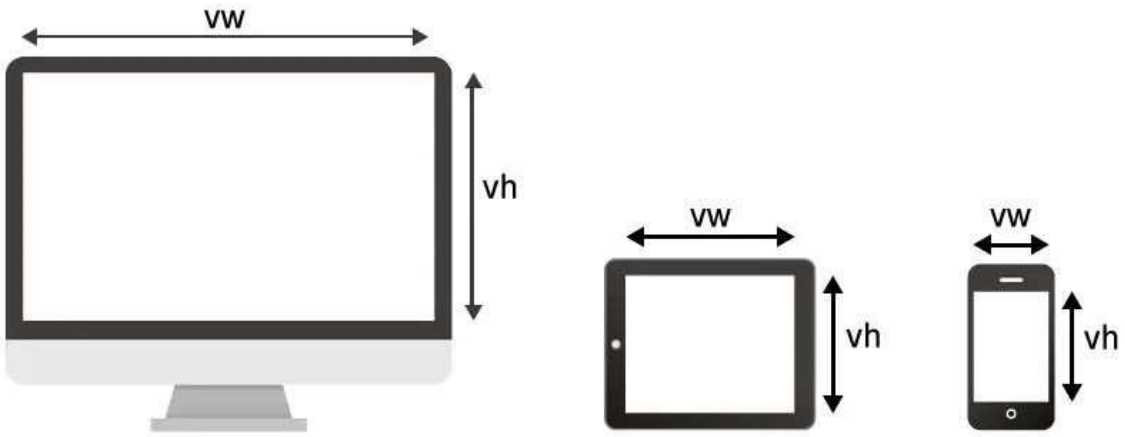
vw/vh

vw 和 vh 分别是相对于屏幕视口宽度和高度而言的长度单位：

1vw = 视口宽度均分成 100 份中 1 份的长度；

1vh = 视口高度均分成 100 份中 1 份的长度；

在 JS 中 100vw = window.innerWidth，100vh = window.innerHeight。



vw/vh 的出现使得多了一种写自适应布局的方案，开发者不再局限于 rem 了。

相对视口的单位，除了 vw/vh 外，还有 vmin 和 vmax：

vmin：取 vw 和 vh 中值较小的；

vmax：取 vw 和 vh 中值较大的；

颜色体系

CSS 中用于表示颜色的值种类繁多，足够构成一个体系，所以这里就专门拿出一个小节来讲解它。

根据 **CSS 颜色草案**[8] 中提到的颜色值类型，大概可以把它们分为这几类：

- 颜色关键字
- transparent 关键字
- currentColor 关键字
- RGB 颜色
- HSL 颜色

颜色关键字

颜色关键字（color keywords）是不区分大小写的标识符，它表示一个具体的颜色，比如 white（白），黑（black）等；

可接受的关键字列表在CSS的演变过程中发生了改变：

- CSS 标准 1 只接受 16 个基本颜色，称为 VGA 颜色，因为它们来源于 VGA 显卡所显示的颜色

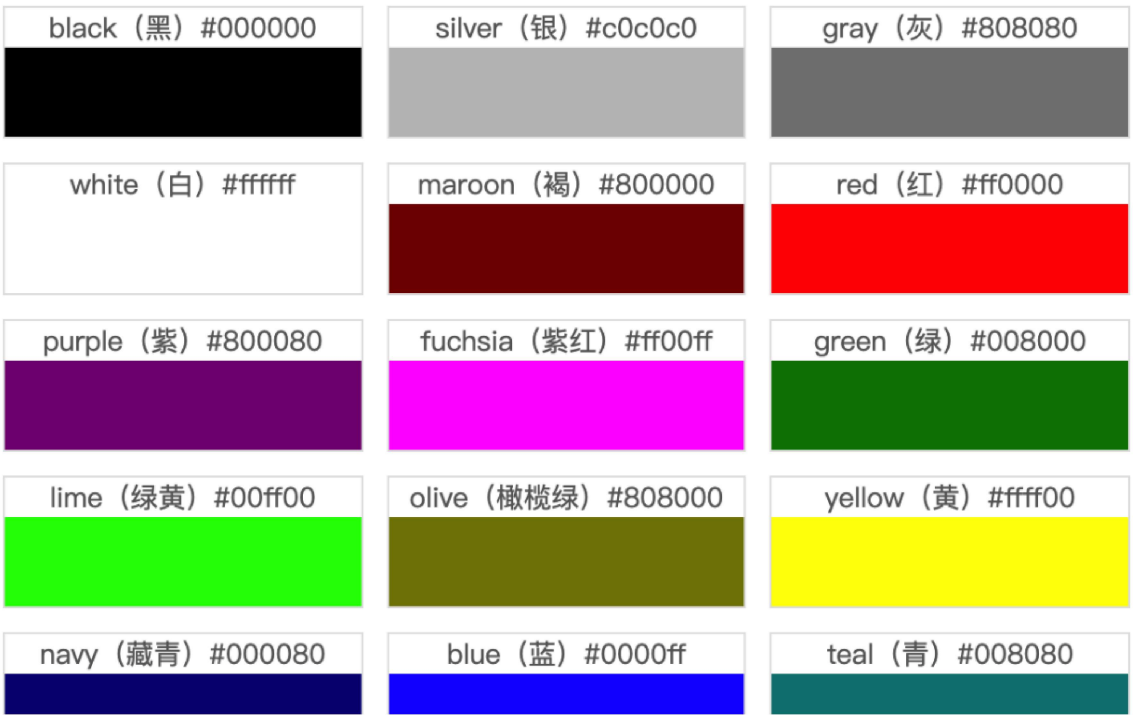
集合而被称为 VGA colors （视频图形阵列色彩）。

CSS 标准 2 增加了 orange 关键字。

从一开始，浏览器接受其它的颜色，由于一些早期浏览器是 X11 应用程序，这些颜色大多数是 X11 命名的颜色列表，虽然有一点不同。SVG 1.0 是首个正式定义这些关键字的标准；CSS 色彩标准 3 也正式定义了这些关键字。它们经常被称作扩展的颜色关键字， X11 颜色或 SVG 颜色。

CSS 颜色标准 4 添加可 rebeccapurple 关键字来纪念 web 先锋 Eric Meyer。

如下这张图是 16 个基础色，又叫 VGA 颜色。截止到目前为止 CSS 颜色关键字总共有 146 个，这里可以查看 **完整的色彩关键字列表**[9]。



VGA 颜色

需要注意的是如果声明的时候的颜色关键字是错误的，浏览器会忽略它。

transparent 关键字

transparent 关键字表示一个完全透明的颜色，即该颜色看上去将是背景色。从技术上说，它是带有 alpha 通道为最小值的黑色，是 rgba(0,0,0,0) 的简写。

透明关键字有什么应用场景呢?

实现三角形

下面这个图是用 4 条边框填充的正方形，看懂了它你大概就知道该如何用 CSS 写三角形了。

```
div  {
      border-top-color: #ffc107;
      border-right-color: #00bcd4;
      border-bottom-color: #e26b6b;
      border-left-color: #cc7cda;
      border-width: 50px;
      border-style: solid;
}
```

用 transparent 实现三角形的原理：

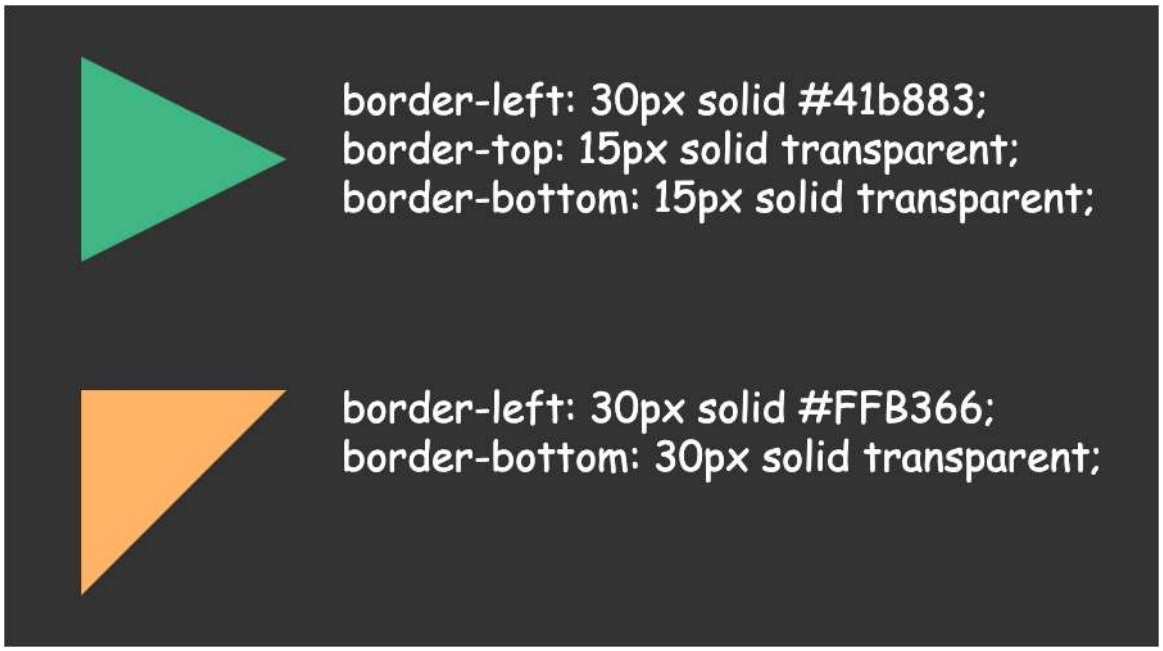
首先宽高必须是 0px，通过边框的粗细来填充内容；

那条边需要就要加上颜色，而不需要的边则用 transparent；

想要什么样姿势的三角形，完全由上下左右 4 条边的中有颜色的边和透明的边的位置决定；

等腰三角形：设置一条边有颜色，然后紧挨着的 2 边是透明，且宽度是有颜色边的一半；直角三角形：设置一条边有颜色，然后紧挨着的任何一边透明即可。

看下示例：



增大点击区域

常常在移动端的时候点击的按钮的区域特别小，但是由于现实效果又不太好把它做大，所以常用的一个手段就是通过透明的边框来增大按钮的点击区域：

```
.btn  {
      border: 5px  solid transparent;
}
```

currentColor 关键字

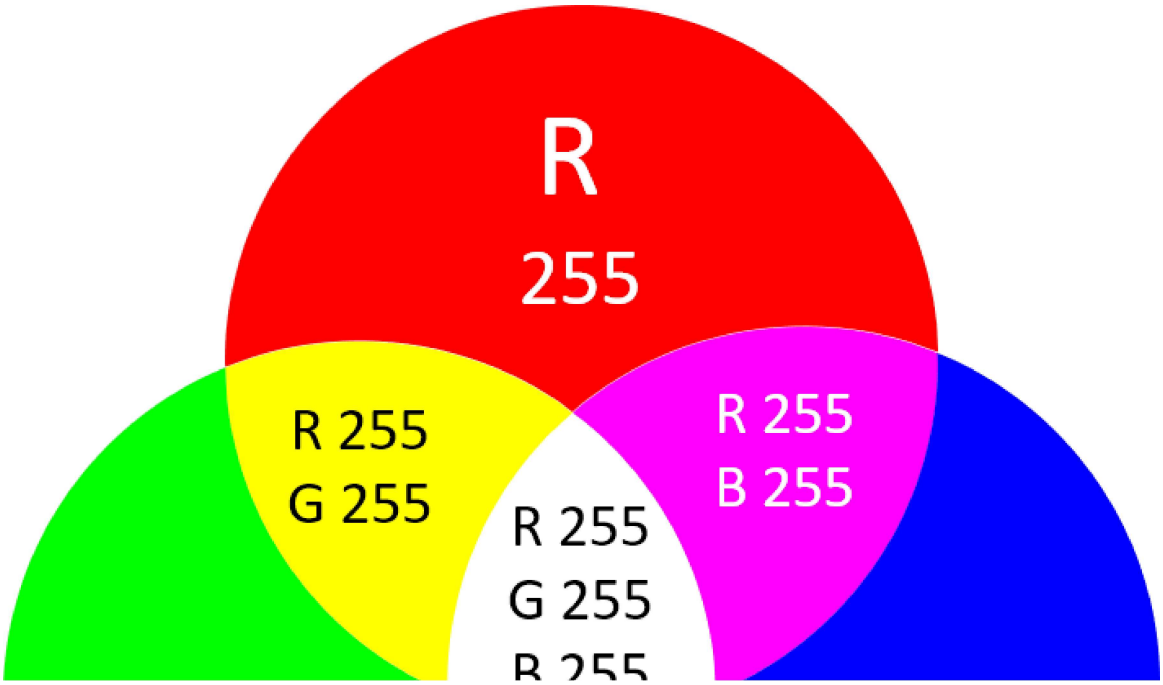
currentColor 会取当前元素继承父级元素的文本颜色值或声明的文本颜色值，即 computed 后的 color 值。

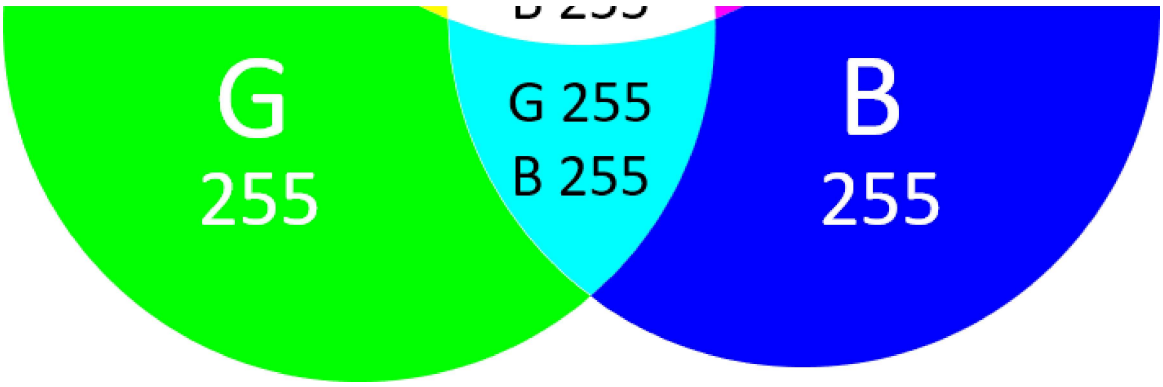
比如，对于如下 CSS，该元素的边框颜色会是 red：

```
.btn  {
      color: red;
      border: 1px  solid currentColor;
}
```

RGB[A] 颜色

RGB[A] 颜色是由 R(red)-G(green)-B(blue)-A(alpha) 组成的色彩空间。





在 CSS 中，它有两种表示形式：

十六进制符号；

函数符；

十六进制符号

RGB 中的每种颜色的值范围是 00~ff，值越大表示颜色越深。所以一个颜色正常是 6 个十六进制字符加上 # 组成，比如红色就是 #ff0000。

如果 RGB 颜色需要加上不透明度，那就需要加上 alpha 通道的值，它的范围也是 00~ff，比如一个带不透明度为 67% 的红色可以这样写 #ff0000aa。

使用十六进制符号表示颜色的时候，都是用 2 个十六进制表示一个颜色，如果这 2 个字符相同，还可以缩减成只写 1 个，比如，红色 #f00；带 67% 不透明度的红色 #f00a。

函数符

当 RGB 用函数表示的时候，每个值的范围是 0~255 或者 0%~100%，所以红色是 rgb(255, 0, 0)， 或者 rgb(100%, 0, 0)。

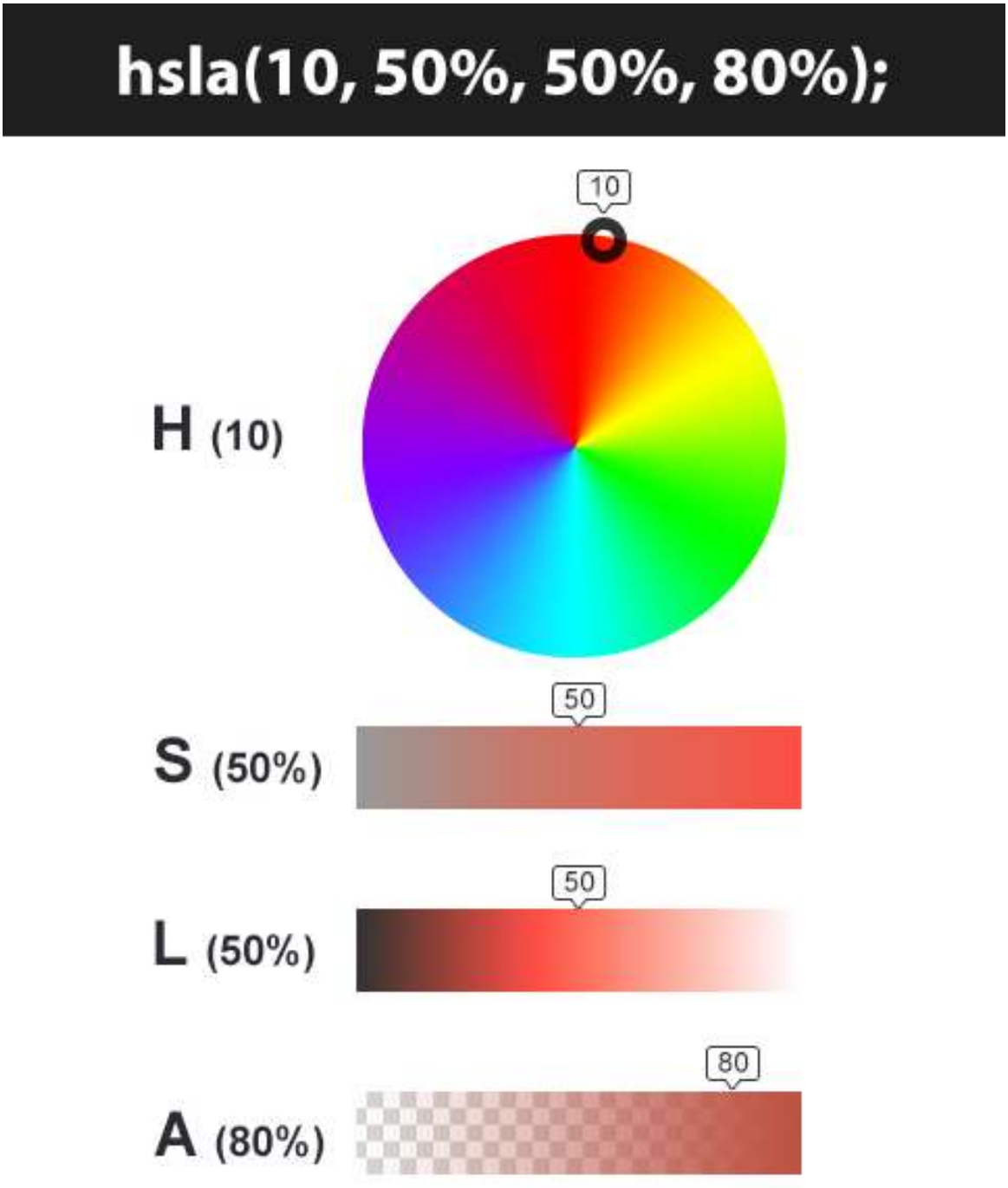
如果需要使用函数来表示带不透明度的颜色值，值的范围是 0~1 及其之间的小数或者 0%~100%，比如带 67% 不透明度的红色是 rgba(255, 0, 0, 0.67) 或者 rgba(100%, 0%, 0%, 67%)

> 需要注意的是 RGB 这 3 个颜色值需要保持一致的写法，要嘛用数字要嘛用百分比，而不透明度的值的可以不用和 RGB 保持一致写法。比如 rgb(100%, 0, 0) 这个写法是无效的；而 rgb(100%, 0%, 0%, 0.67) 是有效的。

在第 4 代 CSS 颜色标准中，新增了一种新的函数写法，即可以把 RGB 中值的分隔逗号改成空格，而把 RGB 和 alpha 中的逗号改成 /，比如带 67% 不透明度的红色可以这样写 rgba(255 0 0 / 0.67)。另外还把 rgba 的写法合并到 rgb 函数中了，即 rgb 可以直接写带不透明度的颜色。

HSL[A] 颜色

HSL[A] 颜色是由色相(hue)-饱和度(saturation)-亮度(lightness)-不透明度组成的颜色体系。



色相（H）是色彩的基本属性，值范围是 0~~360 或者 0deg~~360deg， 0 (或 360) 为红色, 120 为绿色, 240 为蓝色；

饱和度（S）是指色彩的纯度，越高色彩越纯，低则逐渐变灰，取 0~100% 的数值；0% 为灰色， 100% 全色；

亮度（L），取 0~100%，0% 为暗，100% 为白；

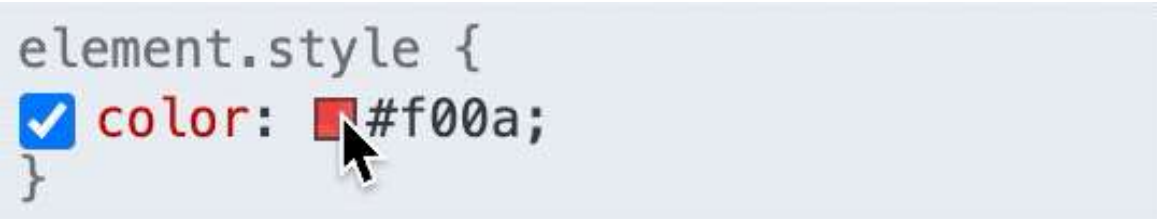
不透明度（A），取 0~~100%，或者~~1及之间的小数；

写法上可以参考 RGB 的写法，只是参数的值不一样。

给一个按钮设置不透明度为 67% 的红色的 color 的写法，以下全部写法效果一致：

```
button  {
      color: #ff0000aa;
      color: #f00a;
      color: rgba(255, 0, 0, 0.67);
      color: rgb(100%  0%  0%  / 67%);
      color: hsla(0, 100%, 50%, 67%);
      color: hsl(0deg  100%  50%  / 67%);
}
```

小提示：在 Chrome DevTools 中可以按住 shift + 鼠标左键可以切换颜色的表示方式。



媒体查询

媒体查询是指针对不同的设备、特定的设备特征或者参数进行定制化的修改网站的样式。

你可以通过给 `<link>` 加上 media 属性来指定该样式文件只能对什么设备生效，不指定的话默认是 all，即对所有设备都生效：

```
<link  rel="stylesheet"  src="styles.css"  m
<link  rel="stylesheet"  src="styles.css"  m
```

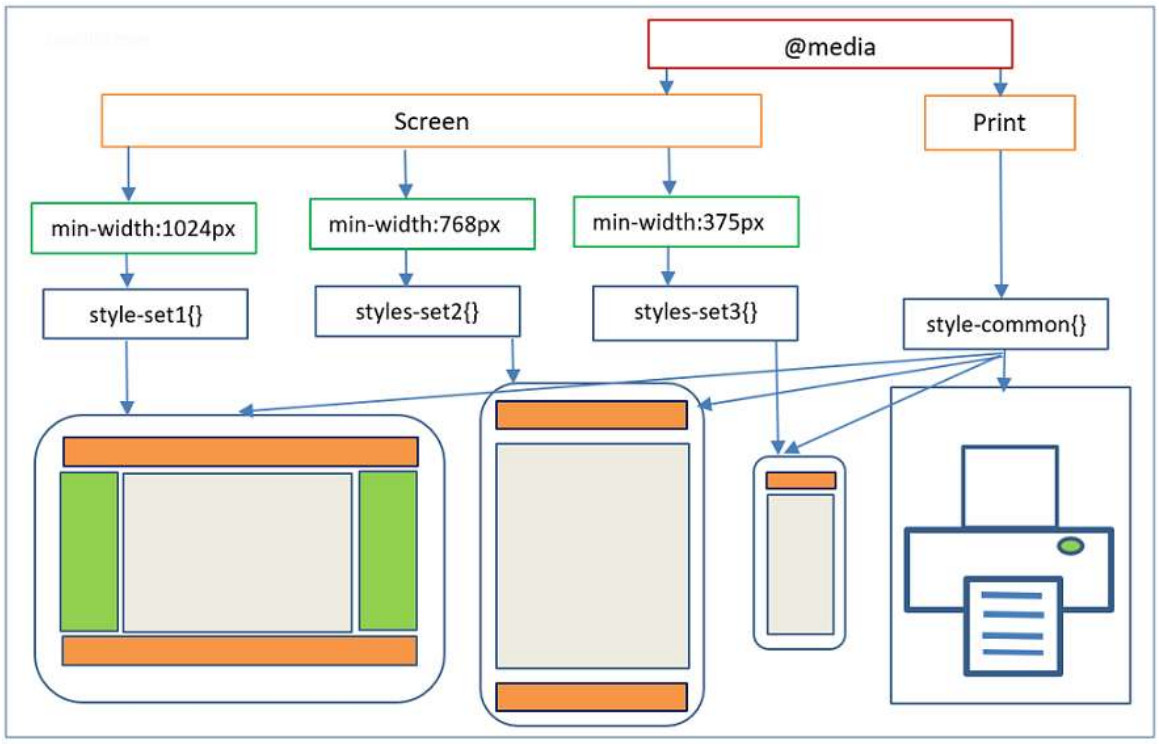
都支持哪些设备类型？

- all：适用于所有设备；
- print：适用于在打印预览模式下在屏幕上查看的分页材料和文档；

screen：主要用于屏幕；

speech：主要用于语音合成器。

> 需要注意的是：通过 media 指定的 资源尽管不匹配它的设备类型，但是浏览器依然会加载它。



除了通过 `<link>` 让指定设备生效外，还可以通过 `@media` 让 CSS 规则在特定的条件下才能生效。响应式页面就是使用了 @media 才让一个页面能够同时适配 PC、Pad 和手机端。

```
@media (min-width: 1000px) {}
```

媒体查询支持逻辑操作符：

and：查询条件都满足的时候才生效；

not：查询条件取反；

only：整个查询匹配的时候才生效，常用语兼容旧浏览器，使用时候必须指定媒体类型；

逗号或者 or：查询条件满足一项即可匹配；

媒体查询还支持**众多的媒体特性**[10]，使得它可以写出很复杂的查询条件：

```
/* 用户设备的最小高度为680px或为纵向模式的屏
@media (min-height: 680px), screen and (o
```

常见需求

自定义属性

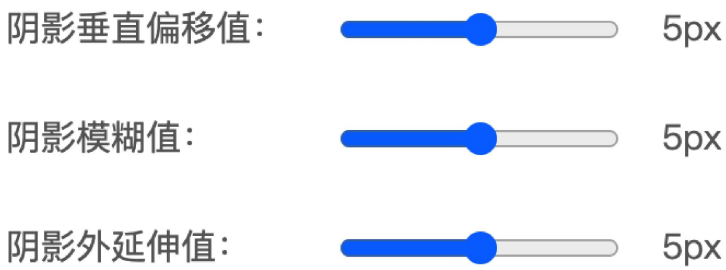
之前我们通常是在预处理器里才可以使用变量，而现在 CSS 里也支持了变量的用法。通过自定义属性就可以在想要使用的地方引用它。

自定义属性也和普通属性一样具有级联性，申明在 :root 下的时候，在全文档范围内可用，而如果是在某个元素下申明自定义属性，则只能在它及它的子元素下才可以使用。

自定义属性必须通过 `--x` 的格式申明，比如：--theme-color: red; 使用自定义属性的时候，需要用 var 函数。比如：

```
<!--  定义自定义属性 -->
:root  {
      --theme-color: red;
}

<!--  使用变量 -->
h1  {
      color: var(--theme-color);
}
```



上图这个是使用 CSS 自定义属性配合 JS 实现的动态调整元素的 box-shadow，具体可以看这个 **codepen demo**[11]。

1px 边框解决方案

Retina 显示屏比普通的屏幕有着更高的分辨率，所以在移动端的 1px 边框就会看起来比较粗，为了美观通常需要把这个线条细化处理。

而这里附上最后一种通过伪类和 transform 实现的相对完美的解决方案：

只设置单条底部边框：

```
.scale-1px-bottom  {
      position: relative;
      border:none;
}
.scale-1px-bottom::after  {
      content: '';
```

```
        content: '';
        position: absolute;
        left: 0;
        bottom: 0;
        background: #000;
        width: 100%;
        height: 1px;
        -webkit-transform: scaleY(0.5);
        transform: scaleY(0.5);
        -webkit-transform-origin: 0  0;
        transform-origin: 0  0;
}
```

同时设置 4 条边框：

```
.scale-1px  {
        position: relative;
        margin-bottom: 20px;
        border:none;
}
.scale-1px::after  {
        content: '';
        position: absolute;
        top: 0;
        left: 0;
```

```
        border: 1px  solid #000;
        -webkit-box-sizing: border-box;
        box-sizing: border-box;
        width: 200%;
        height: 200%;
        -webkit-transform: scale(0.5);
        transform: scale(0.5);
        -webkit-transform-origin: left top;
        transform-origin: left top;
}
```

清除浮动

什么是浮动：浮动元素会脱离文档流并向左/向右浮动，直到碰到父元素或者另一个浮动元素。

为什么要清楚浮动，它造成了什么问题？

因为浮动元素会脱离正常的文档流，并不会占据文档流的位置，所以如果一个父元素下面都是浮动元素，那么这个父元素就无法被浮动元素所撑开，这样一来父元素就丢失了高度，这就是所谓的浮动造成的父元素高度坍塌问题。

父元素高度一旦坍塌将对后面的元素布局造成影响，为了解决这个问题，所以需要清除浮动，让父元素恢复高度，那该如何做呢?

这里介绍两种方法：通过 BFC 来清除、通过 clear 来清除。

BFC 清除浮动

前面介绍 BFC 的时候提到过，计算 BFC 高度的时候浮动子元素的高度也将计算在内，利用这条规则就可以清楚浮动。

假设一个父元素 parent 内部只有 2 个子元素 child，且它们都是左浮动的，这个时候 parent 如果没有设置高度的话，因为浮动造成了高度坍塌，所以 parent 的高度会是 0，此时只要给 parent 创造一个 BFC，那它的高度就能恢复了。

而产生 BFC 的方式很多，我们可以给父元素设置 overflow: auto 来简单的实现 BFC 清除浮动，但是为了兼容 IE 最好用 overflow: hidden。

```
.parent  {
      overflow: hidden;
```

```
}
```

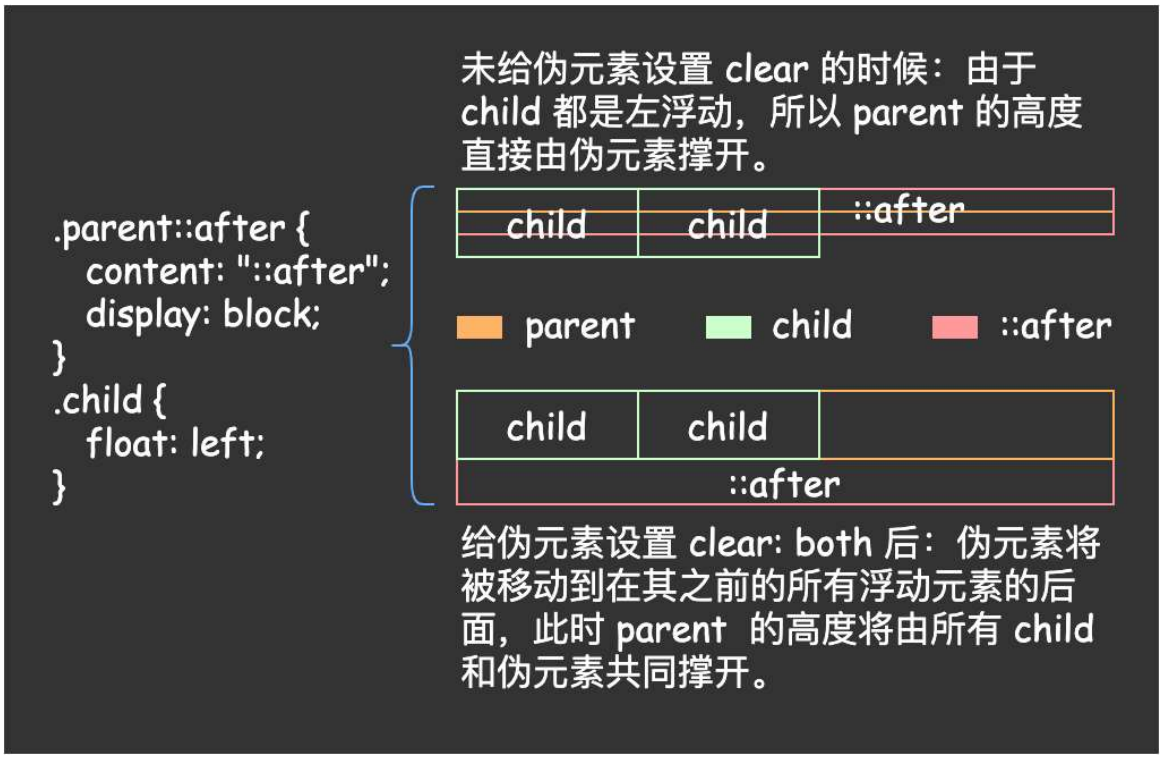
通过 overflow: hidden 来清除浮动并不完美，当元素有阴影或存在下拉菜单的时候会被截断，所以该方法使用比较局限。

通过 clear 清除浮动

我先把结论贴出来：

```
.clearfix  {
      zoom: 1;
}
.clearfix::after  {
      content: "";
      display: block;
      clear: both;
}
```

这种写法的核心原理就是通过 ::after 伪元素为在父元素的最后一个子元素后面生成一个内容为空的块级元素，然后通过 clear 将这个伪元素移动到所有它之前的浮动元素的后面，画个图来理解一下。



可以结合这个 **codepen demo**[12] 一起理解上图的 clear 清楚浮动原理。

上面这个 demo 或者图里为了展示需要所以给伪元素的内容设置为了 ::after，实际使用的时候需要设置为空字符串，让它的高度为 0，从而父元素的高度都是由实际的子元素撑开。

该方式基本上是现在人人都在用的清除浮动的方案，非常通用。

消除浏览器默认样式

针对同一个类型的 HTML 标签，不同的浏览器往往有不同的表现，所以在网站制作的时候，开发者通常

都是需要将这些浏览器的默认样式清除，让网页在不同的浏览器上能够保持一致。

针对清除浏览器默认样式这件事，在很早之前 CSS 大师 Eric A. Meyer 就干过。它就是写一堆通用的样式用来重置浏览器默认样式，这些样式通常会放到一个命名为 reset.css 文件中。比如大师的 **reset.css**[13] 是这么写的：

```
html, body, div, span, applet, object, iframe
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video  {
      margin: 0;
      padding: 0;
      border: 0;
```

```
        font-size: 100%;
        font: inherit;
        vertical-align: baseline;
}
/* HTML5 display-role reset for older browse
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section
        display: block;
}
body  {
        line-height: 1;
}
ol, ul  {
        list-style: none;
}
blockquote, q  {
        quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after  {
        content: '';
        content: none;
}
table  {
        border-collapse: collapse;
```

```
    border-spacing: 0;
}
```

他的这份 reset.css 据说是被使用最广泛的重设样式的方案了。

除了 reset.css 外，后来又出现了 **Normalize.css**[14] 。关于 Normalize.css, 其作者 necolas 专门写了一篇文章介绍了它，并谈到了它和 reset.css 的区别。这个是他写那篇文章的翻译版：**让我们谈一谈 Normalize.css**[15]。

文章介绍到：Normalize.css 只是一个很小的CSS文件，但它在默认的 HTML 元素样式上提供了跨浏览器的高度一致性。相比于传统的 CSS reset，Normalize.css 是一种现代的、为 HTML5 准备的优质替代方案，现在已经有很多知名的框架和网站在使用它了。

Normalize.css 的具体样式可以看这里 **Normalize.css**[16]

区别于 reset.css，Normalize.css 有如下特点：

> reset.css 几乎为所有标签都设置了默认样式，而 Normalize.css 则是有选择性的保护了部分有价值

的默认值；

修复了很多浏览器的 bug，而这是 reset.css 没做到的；

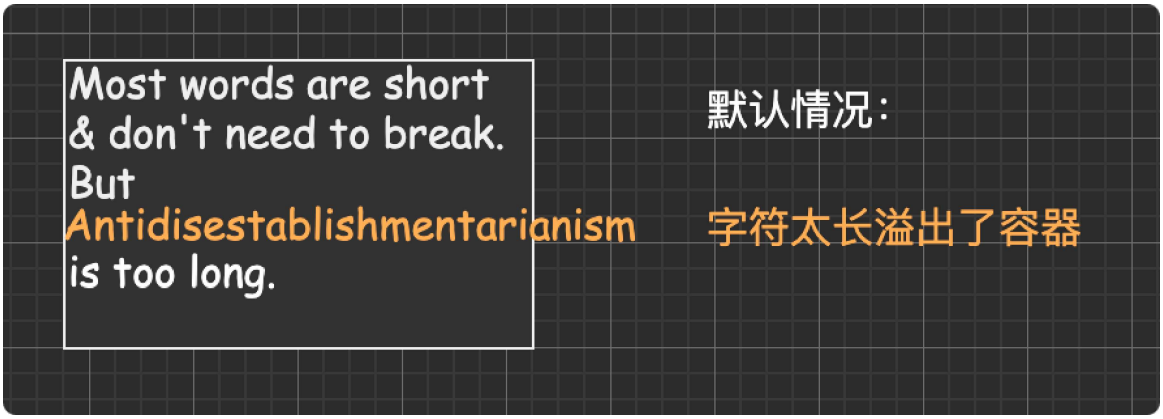
不会让你的调试工具变的杂乱，相反 reset.css 由于设置了很多默认值，所以在浏览器调试工具中往往会看到一大堆的继承样式，显得很杂乱；

Normalize.css 是模块化的，所以可以选择性的去掉永远不会用到的部分，比如表单的一般化；

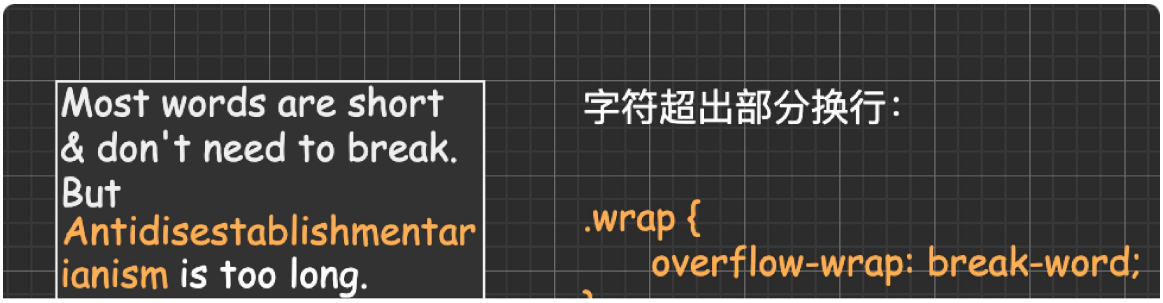
Normalize.css 有详细的说明文档；

长文本处理

默认：字符太长溢出了容器

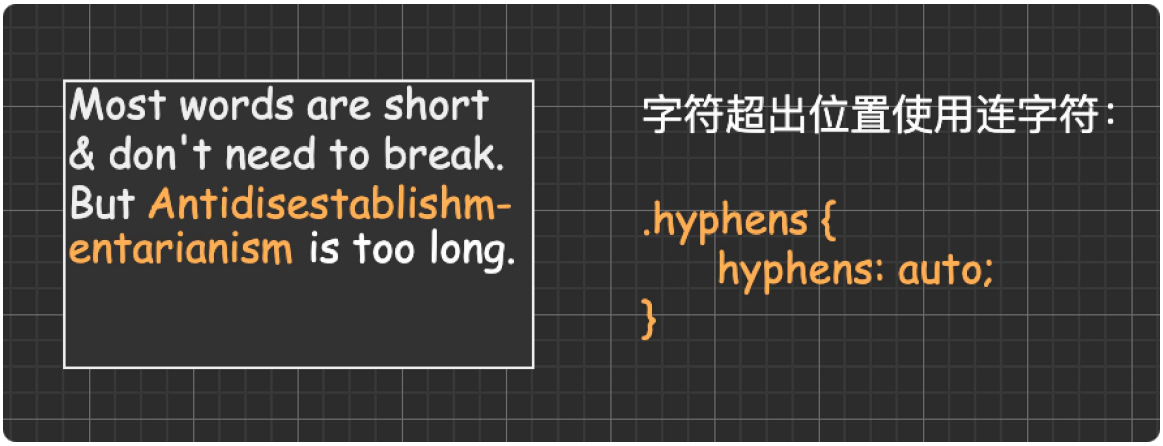


字符超出部分换行

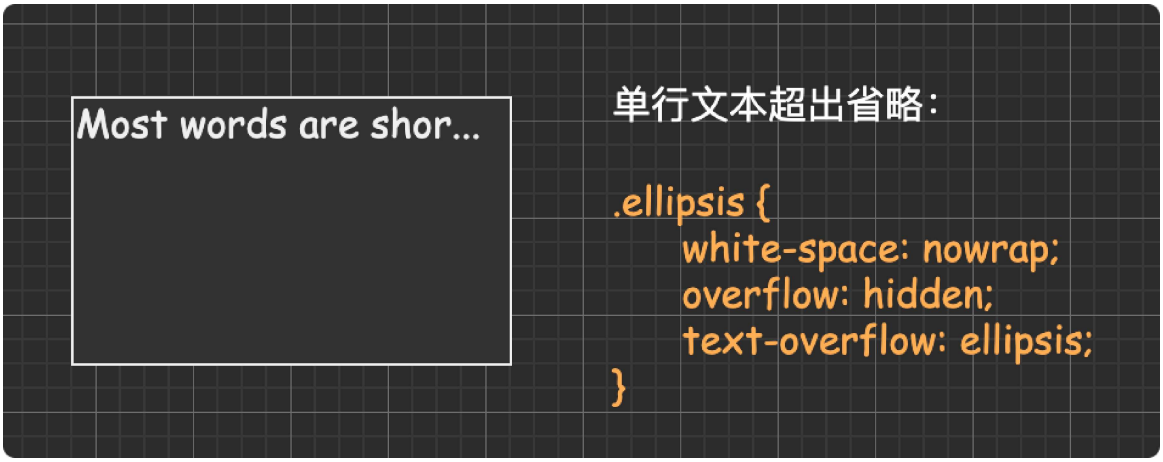


```
}
```

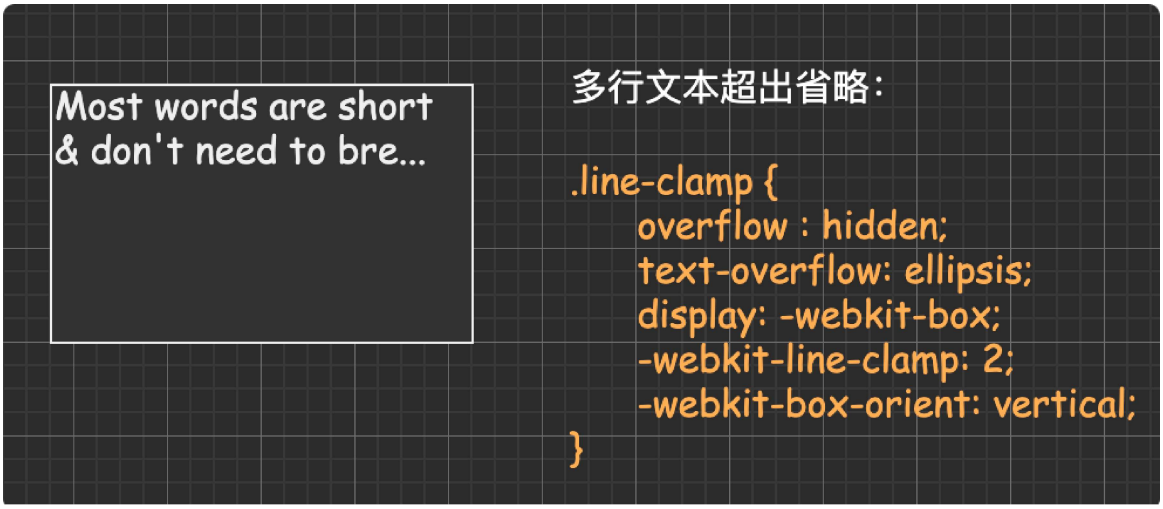
字符超出位置使用连字符



单行文本超出省略

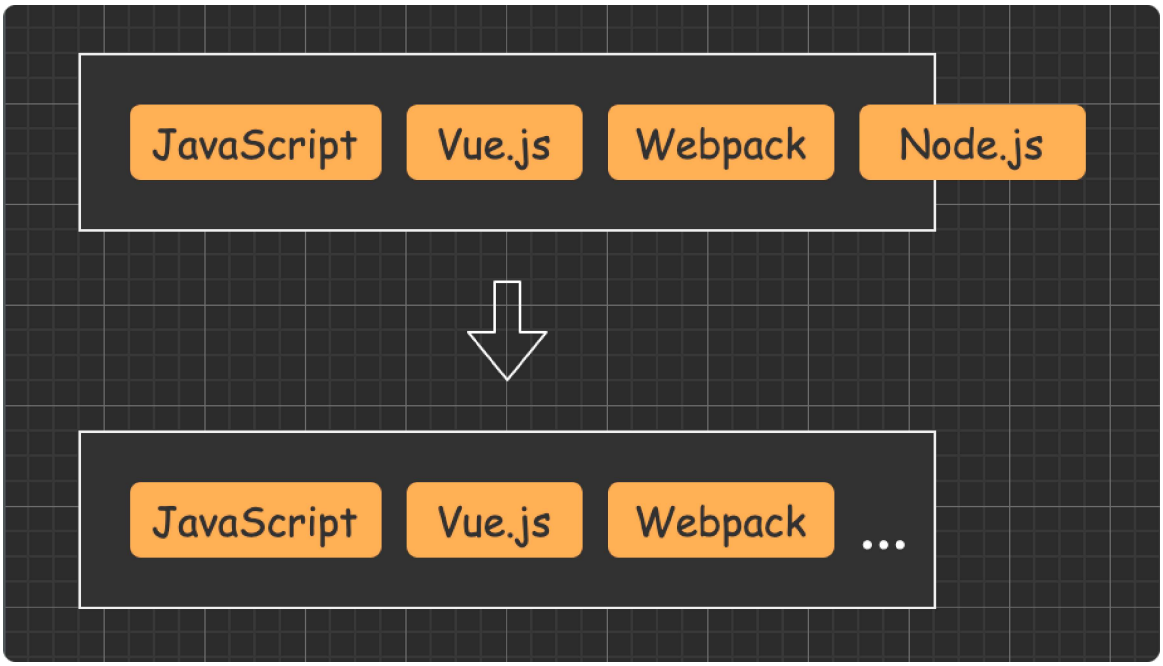


多行文本超出省略



查看以上这些方案的示例： **codepen demo**[17]

有意思的是刚好前两天看到 chokcoco 针对文本溢出也写了一篇文章，主要突出的是对整块的文本溢出处理。啥叫整块文本？比如，下面这种技术标签就是属于整块文本：



另外他还对 iOS/Safari 做了兼容处理，感兴趣的可以去阅读下：**CSS 整块文本溢出省略特性探究**[18]。

水平垂直居中

让元素在父元素中呈现出水平垂直居中的形态，无非就 2 种情况：

单行的文本、inline 或者 inline-block 元素；

固定宽高的块级盒子；

不固定宽高的块级盒子；

以下列到的所有水平垂直居中方案这里写了个 **codepen demo**[19]，配合示例阅读效果更佳。

单行的文本、inline 或 inline-block 元素

水平居中

此类元素需要水平居中，则父级元素必须是块级元素（ block level ），且父级元素上需要这样设置样式：

```
.parent  {
      text-align: center;
}
```

垂直居中

方法一：通过设置上下内间距一致达到垂直居中的效果：

```
.single-line  {
      padding-top: 10px;
```

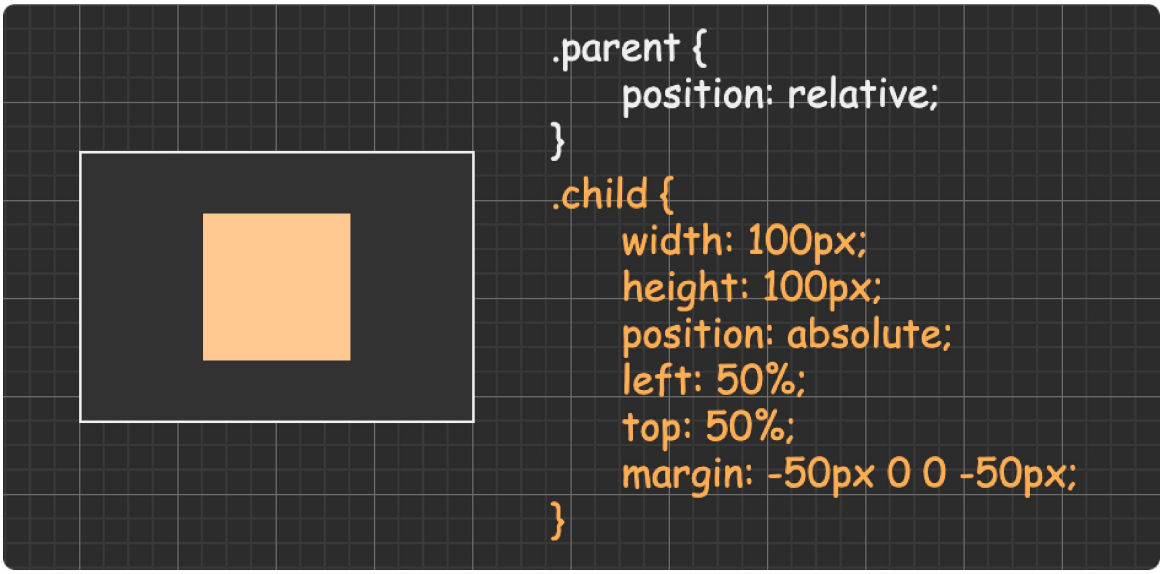
```
        padding-bottom: 10px;
}
```

方法二：通过设置 height 和 line-height 一致达到垂直居中：

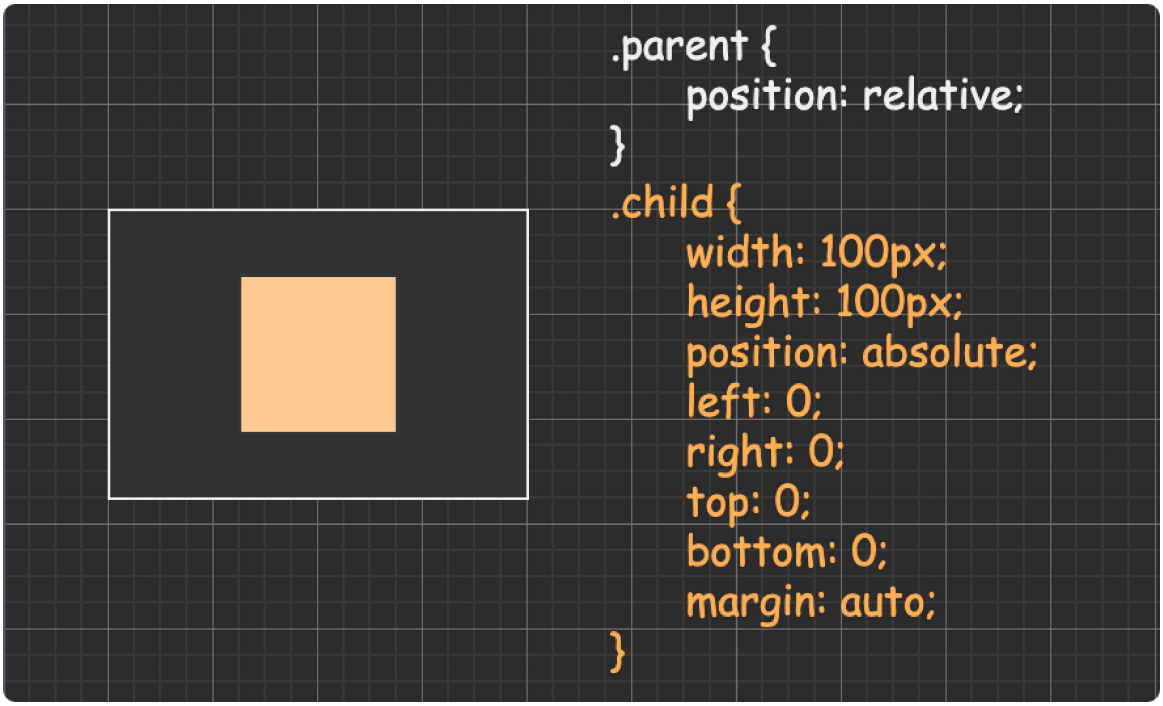
```
.single-line  {
        height: 100px;
        line-height: 100px;
}
```

固定宽高的块级盒子

方法一：absolute + 负 margin

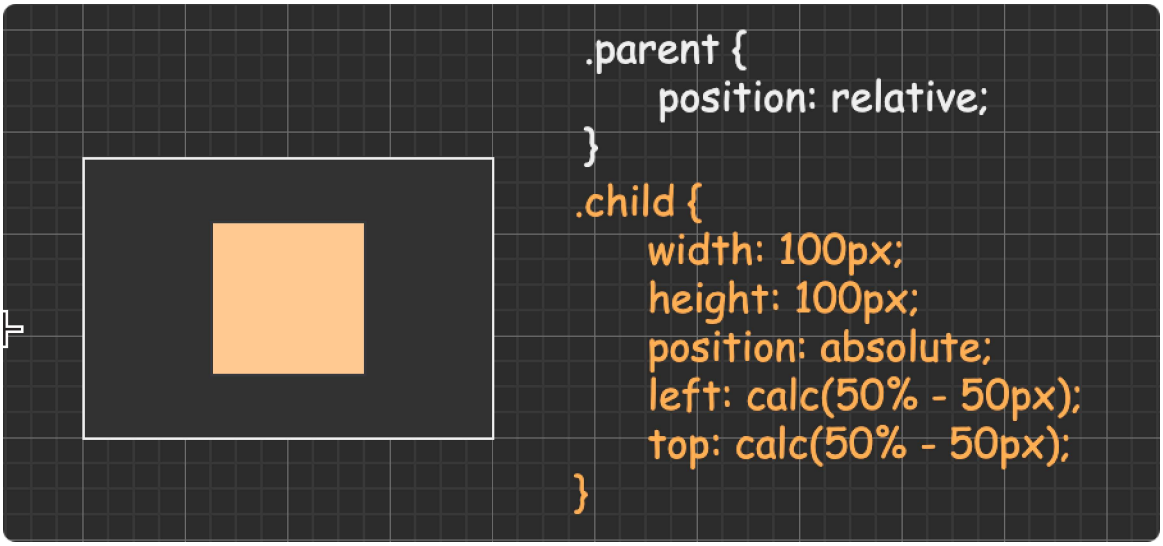


方法二：absolute + margin auto



```
.parent {
    position: relative;
}
.child {
    width: 100px;
    height: 100px;
    position: absolute;
    left: 0;
    right: 0;
    top: 0;
    bottom: 0;
    margin: auto;
}
```

方法三：absolute + calc

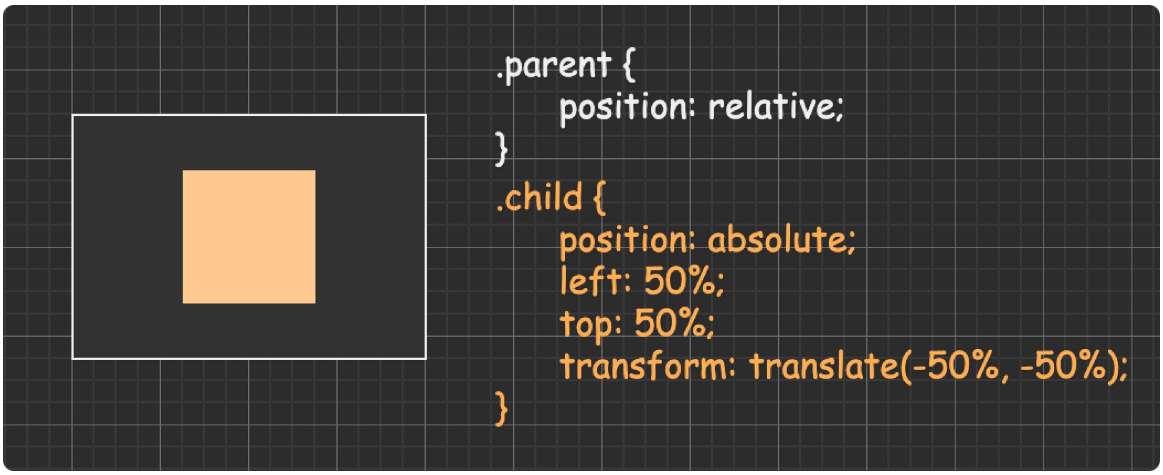


```
.parent {
    position: relative;
}
.child {
    width: 100px;
    height: 100px;
    position: absolute;
    left: calc(50% - 50px);
    top: calc(50% - 50px);
}
```

不固定宽高的块级盒子

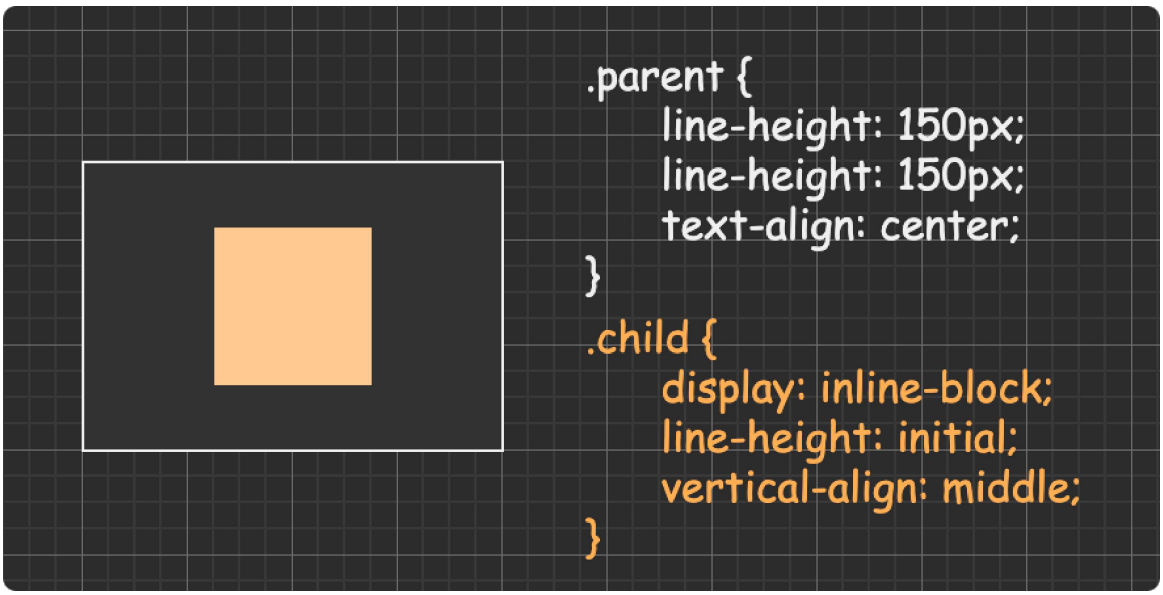
这里列了 6 种方法，其中的两种 line-height 和 writing-mode 方案看后让我惊呼：还有这种操作？学到了学到了。

方法一：absolute + transform



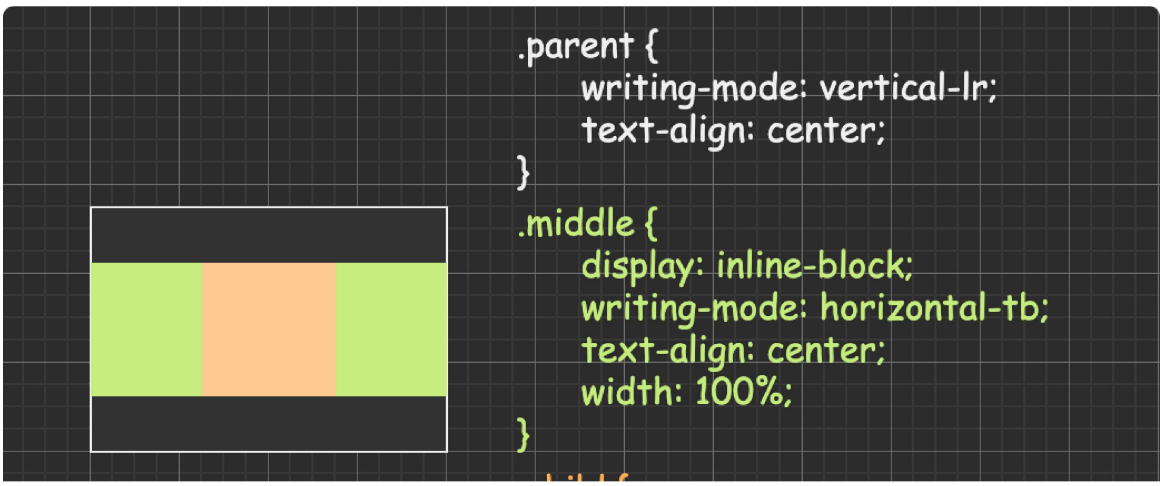
```
.parent {
    position: relative;
}
.child {
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
}
```

方法二：line-height + vertical-align



```
.parent {
    line-height: 150px;
    line-height: 150px;
    text-align: center;
}
.child {
    display: inline-block;
    line-height: initial;
    vertical-align: middle;
}
```

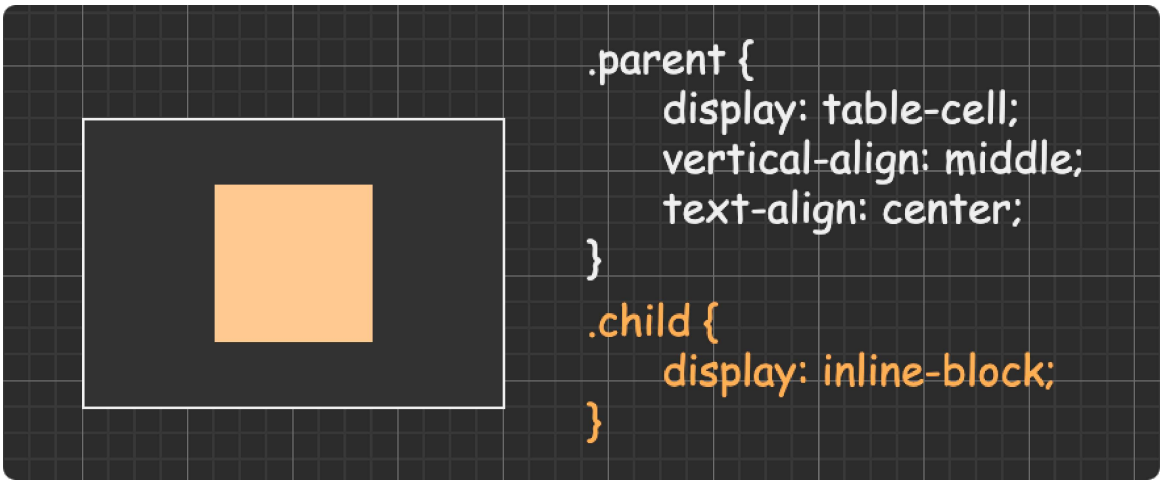
方法三：writing-mode



```
.parent {
    writing-mode: vertical-lr;
    text-align: center;
}
.middle {
    display: inline-block;
    writing-mode: horizontal-tb;
    text-align: center;
    width: 100%;
}
```

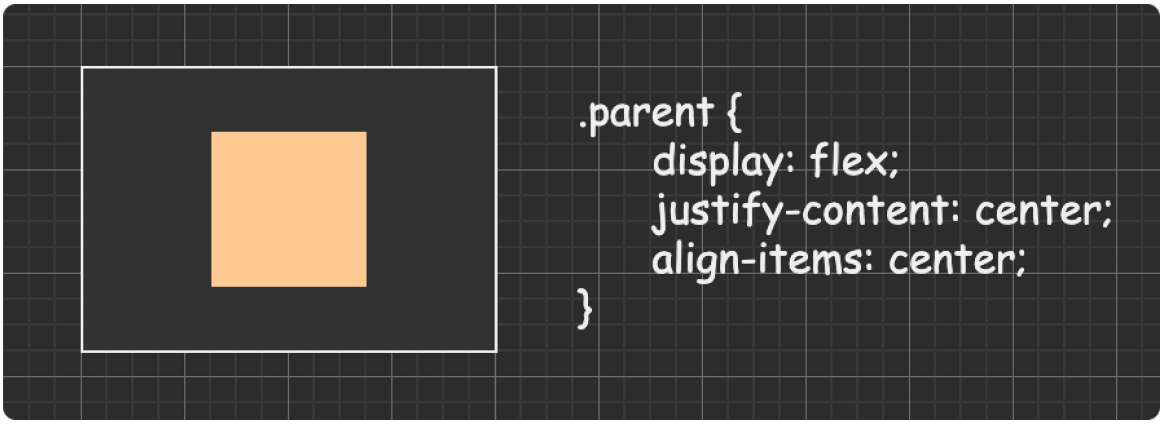
```
.child {
    display: inline-block;
}
```

方法四：table-cell



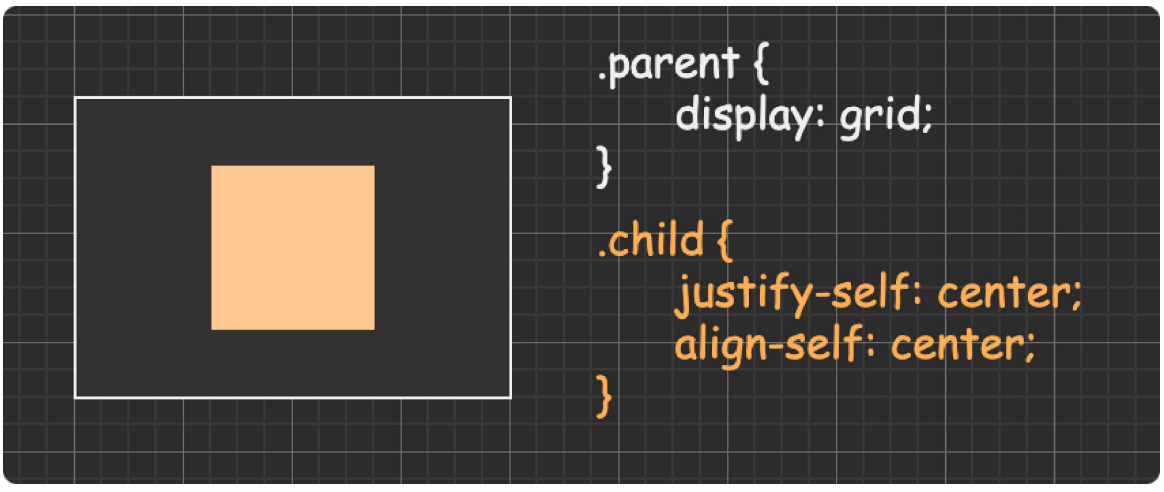
```
.parent {
    display: table-cell;
    vertical-align: middle;
    text-align: center;
}
.child {
    display: inline-block;
}
```

方法五：flex



```
.parent {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

方法六：grid



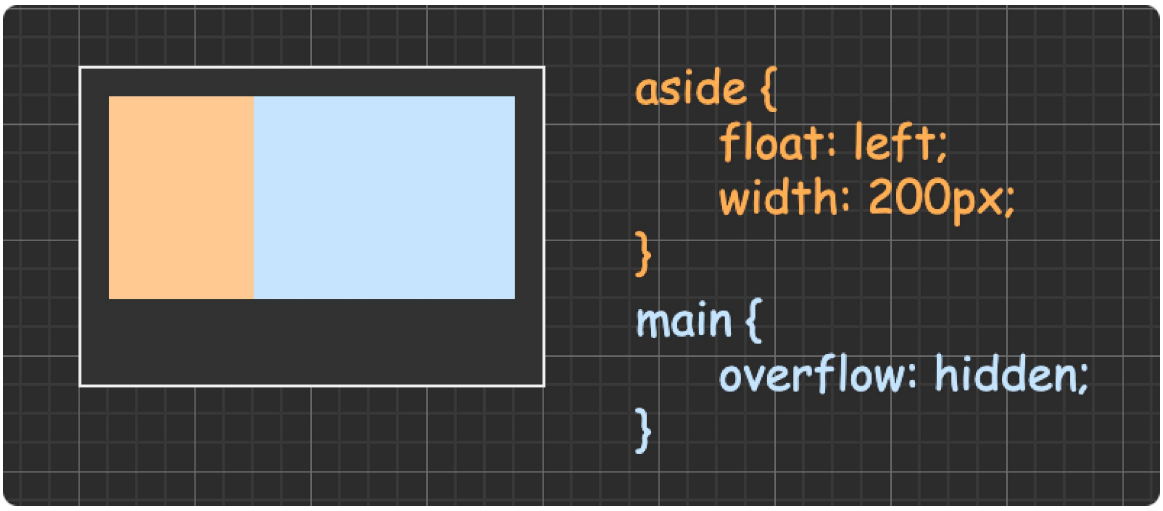
```
.parent {
    display: grid;
}
.child {
    justify-self: center;
    align-self: center;
}
```

常用布局

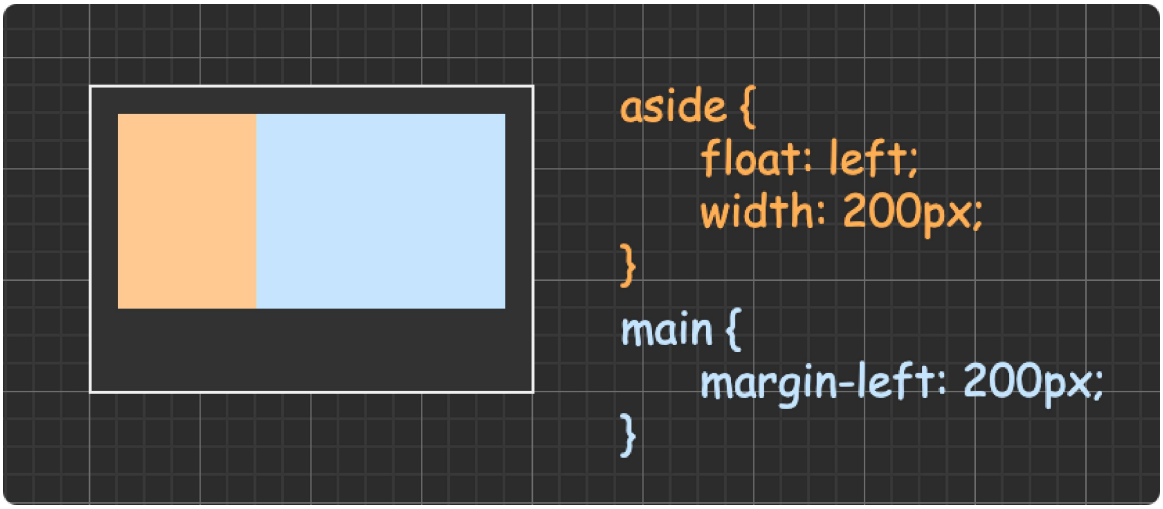
两栏布局（边栏定宽主栏自适应）

针对以下这些方案写了几个示例： codepen demo[20]

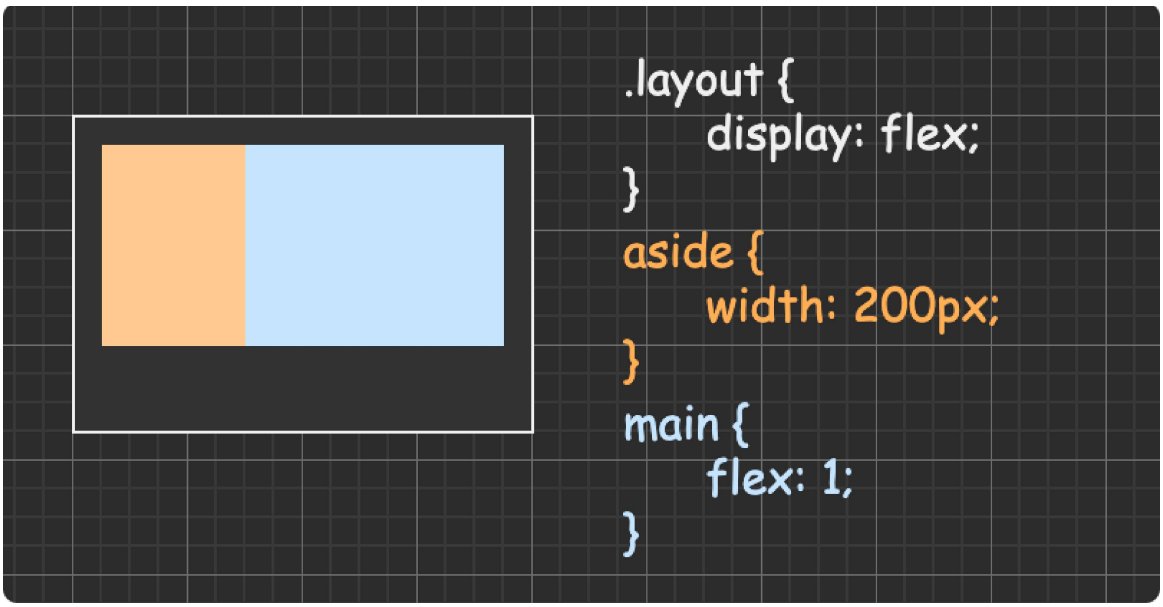
方法一：float + overflow（BFC 原理）



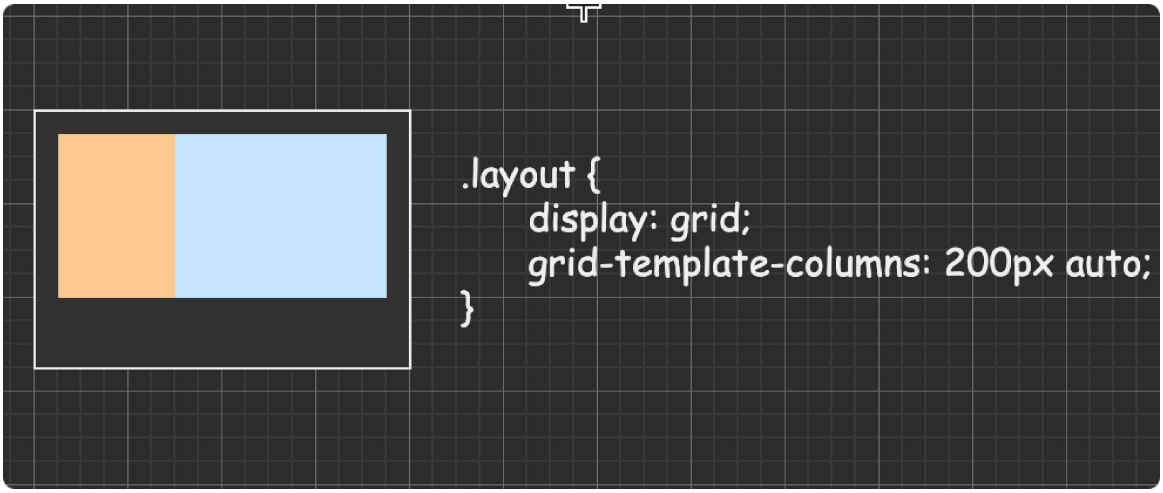
方法二：float + margin



方法三：flex



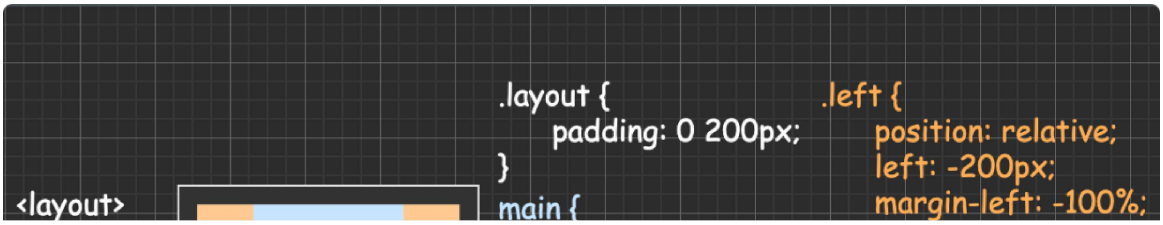
方法四：grid

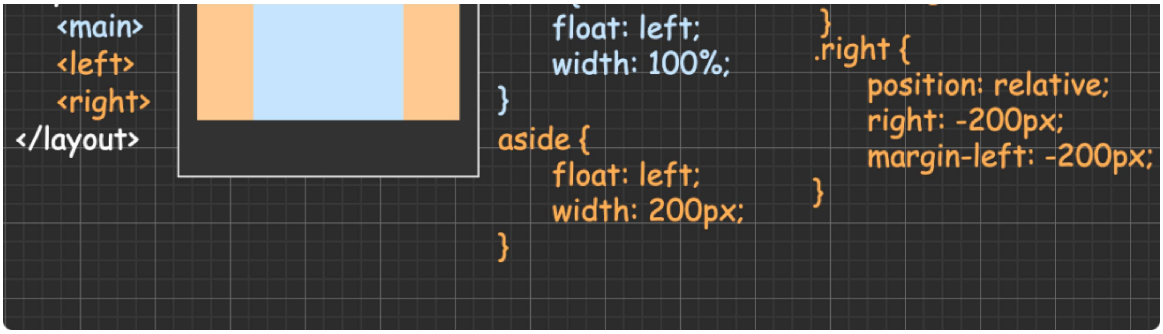


三栏布局（两侧栏定宽主栏自适应）

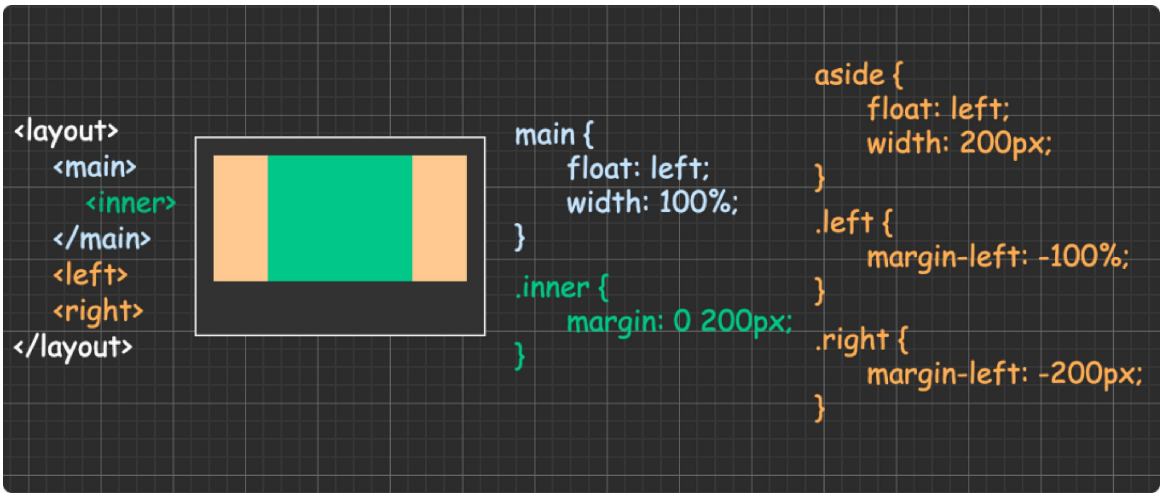
针对以下这些方案写了几个示例： codepen demo[21]

方法一：圣杯布局

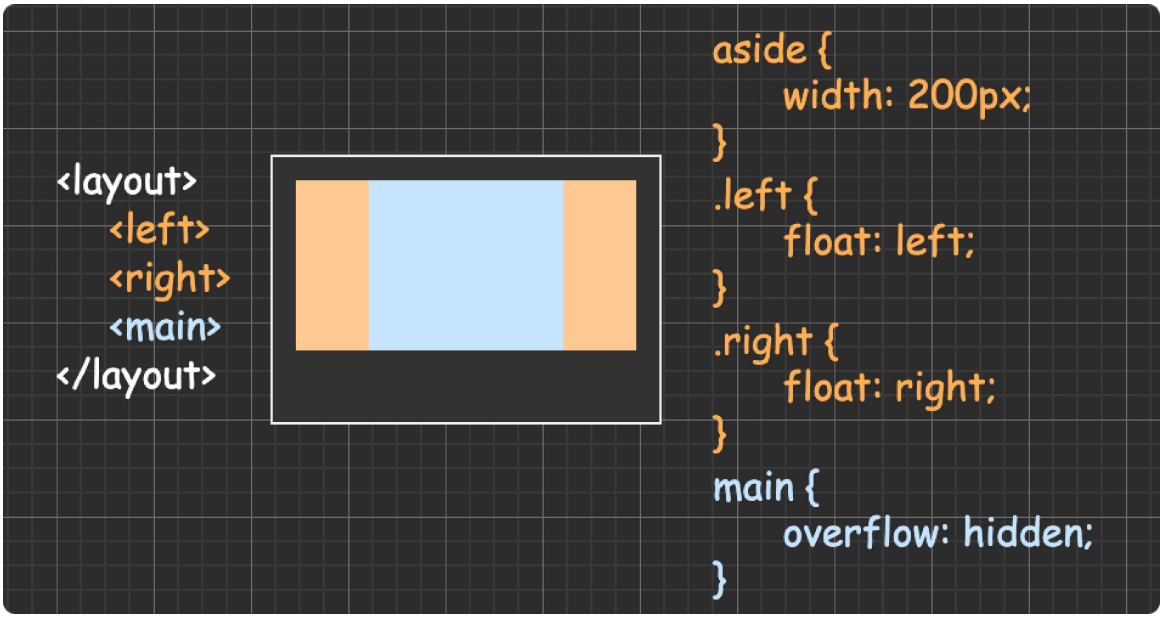




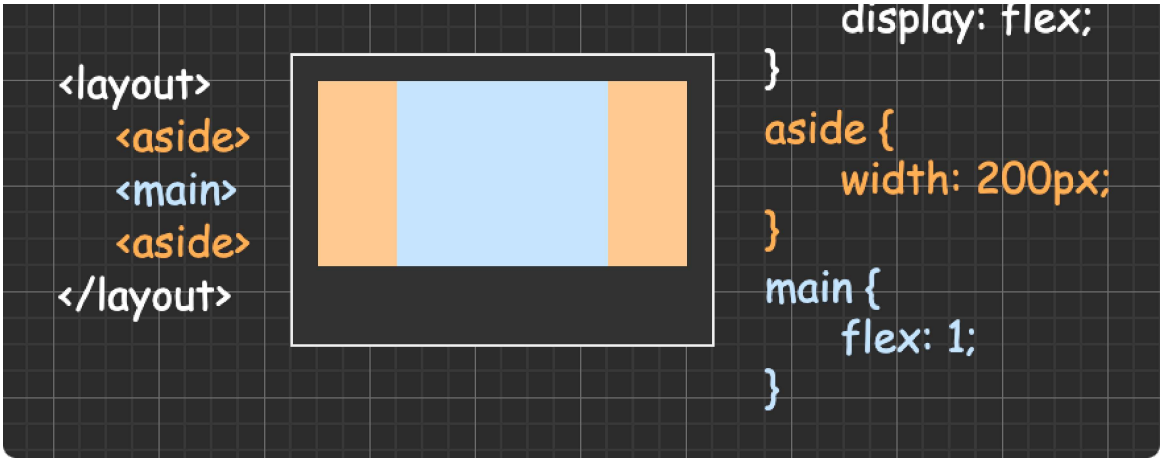
方法二：双飞翼布局



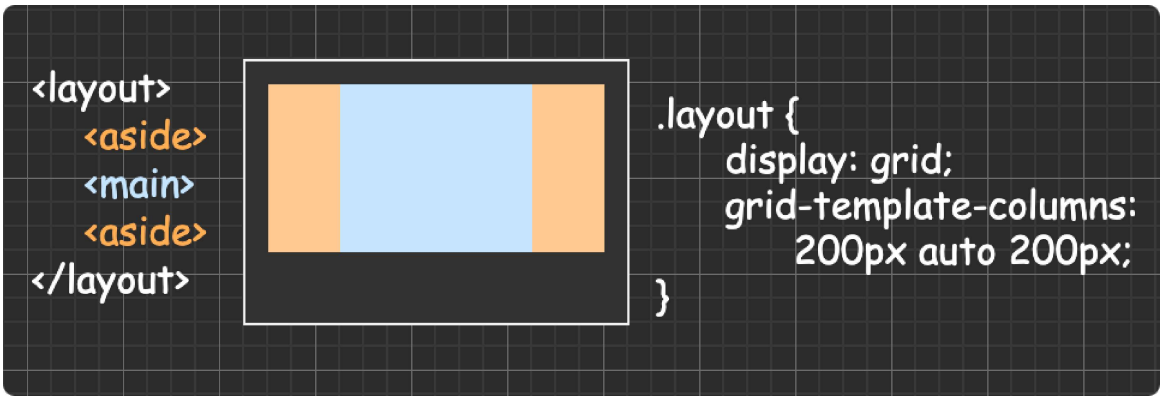
方法三：float + overflow（BFC 原理）



方法四：flex



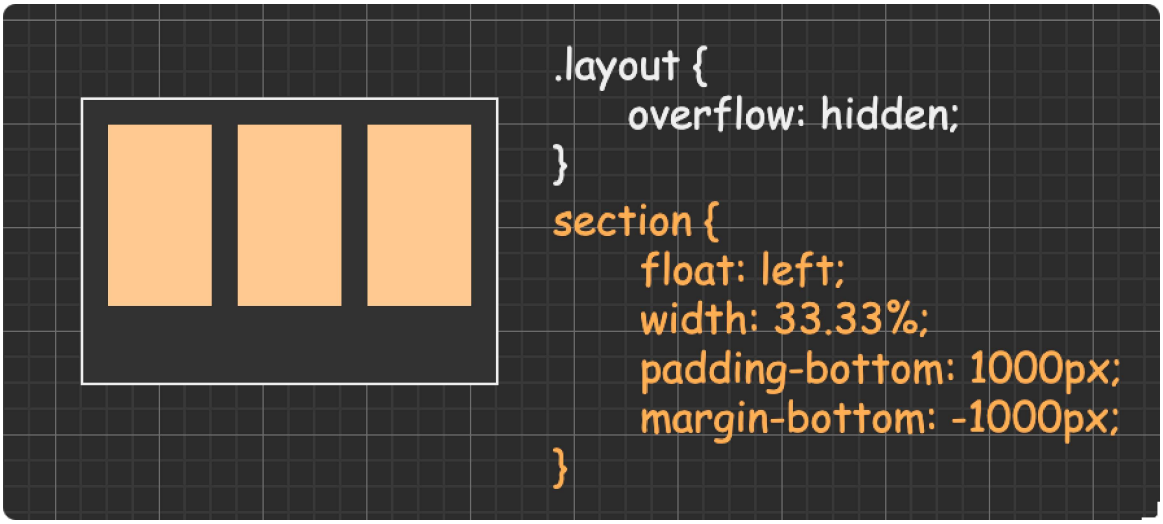
方法五：grid



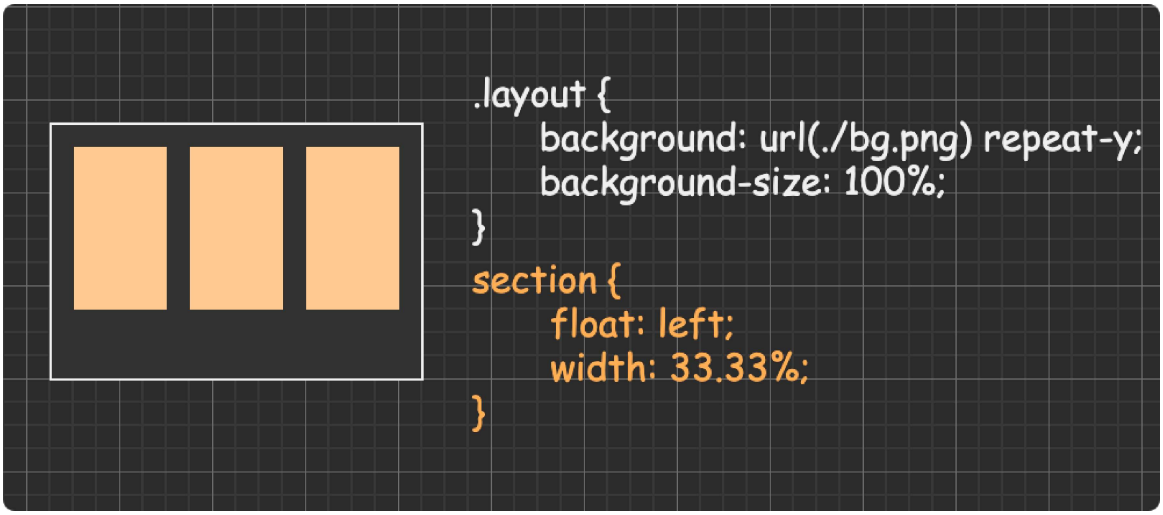
多列等高布局

结合示例阅读更佳：**codepen demo**[22]

方法一：padding + 负margin



方法二：设置父级背景图片



三行布局（头尾定高主栏自适应）

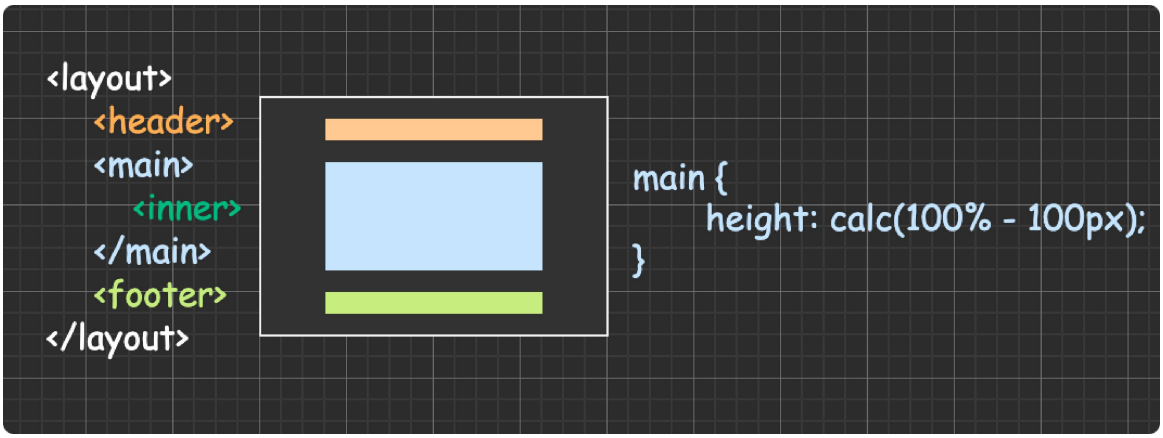
列了 4 种方法，都是基于如下的 HTML 和 CSS 的，结合示例阅读效果更佳：**codepen demo**[23]

```
<div  class="layout">
      <header></header>
      <main>
            <div  class="inner"></div>
      </main>
      <footer></footer>
</div>
```

```
html,
```

```
body,
.layout  {
        height: 100%;
}
body  {
        margin: 0;
}
header,
footer  {
        height: 50px;
}
main  {
        overflow-y: auto;
}
```

方法一：calc



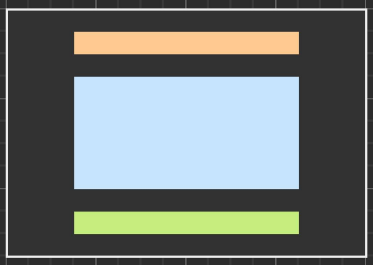
方法二：absolute

```
<layout>
    <header>
    <main>
        <inner>
    </main>
    <footer>
</layout>
```

```
        position: relative;
}
header {
        position: absolute;
        width: 100%;
}
main {
        height: 100%;
        padding: 50px 0;
        box-sizing: border-box;
}
footer {
        position: absolute;
        bottom: 0;
        width: 100%;
}
```

方法三：flex

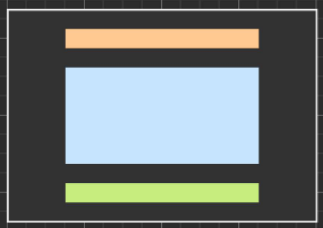
```
<layout>
    <header>
    <main>
        <inner>
    </main>
    <footer>
</layout>
```



```
.layout {
        display: flex;
        flex-direction: column;
}
main {
        flex: 1;
}
```

方法四：grid

```
<layout>
    <header>
    <main>
        <inner>
    </main>
    <footer>
</layout>
```



```
.layout {
        display: grid;
        grid-template-rows: 50px 1fr 50px;
}
```

参考资料

[1] @charset: *https://developer.mozilla.org/zh-CN/docs/Web/CSS/@charset*

[2] Byte order mark: *https://en.wikipedia.org/wiki/Byte_order_mark*

[3] @import: *https://developer.mozilla.org/zh-CN/docs/Web/CSS/@import*

[4] @supports: *https://developer.mozilla.org/zh-CN/docs/Web/CSS/@supports*

[5] MDN CSS Selectors: *https://developer.mozilla.org/zh-CN/docs/Web/CSS/CSS_Selectors*

[6] Visual formatting model: *https://www.w3.org/TR/CSS2/visuren.html*

[7] BFC 应用示例: *https://codepen.io/bulandent/pen/eYBVpEm*

[8] CSS 颜色草案: *https://drafts.csswg.org/css-color-3/*

[9] 完整的色彩关键字列表: *https://codepen.io/bulandent/pen/gOLovwL*

[10] 众多的媒体特性: *https://developer.mozilla.org/zh-CN/docs/Web/Guide/CSS/Media_queries#%E5%AA%92%E4%BD%93%E7%89%B9%E6%80%A7*

[11] codepen demo: *https://codepen.io/bulandent/pen/GVjxLJ*

[12] codepen demo: *https://codepen.io/bulandent/pen/LYb*

OvOa

[13] reset.css: *https://meyerweb.com/eric/tools/css/reset/*

[14] Normalize.css: *https://github.com/necolas/normalize.css*

[15] 让我们谈一谈 Normalize.css: *https://jerryzou.com/posts/aboutNormalizeCss/*

[16] Normalize.css: *https://necolas.github.io/normalize.css/latest/normalize.css*

[17] codepen demo: *https://codepen.io/bulandent/pen/abBrNby*

[18] CSS 整块文本溢出省略特性探究: *https://juejin.cn/post/6938583040469762055*

[19] codepen demo: *https://codepen.io/bulandent/pen/ymaKoM*

[20] codepen demo: *https://codepen.io/bulandent/pen/JjbqxbM*

[21] codepen demo: *https://codepen.io/bulandent/pen/abBrXrj*

[22] codepen demo: *https://codepen.io/bulandent/pen/jOVogdj*

[23] codepen demo: *https://codepen.io/bulandent/pen/yLVdpvr*