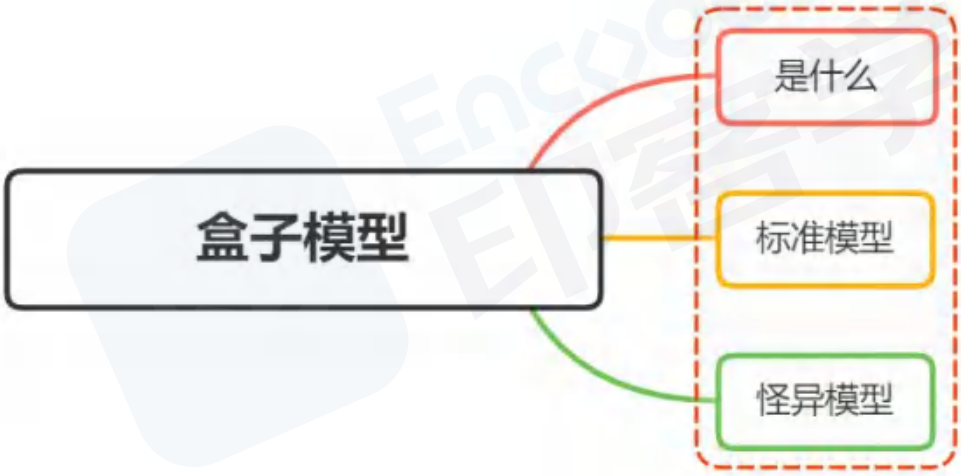
CSS面试真题（20题）

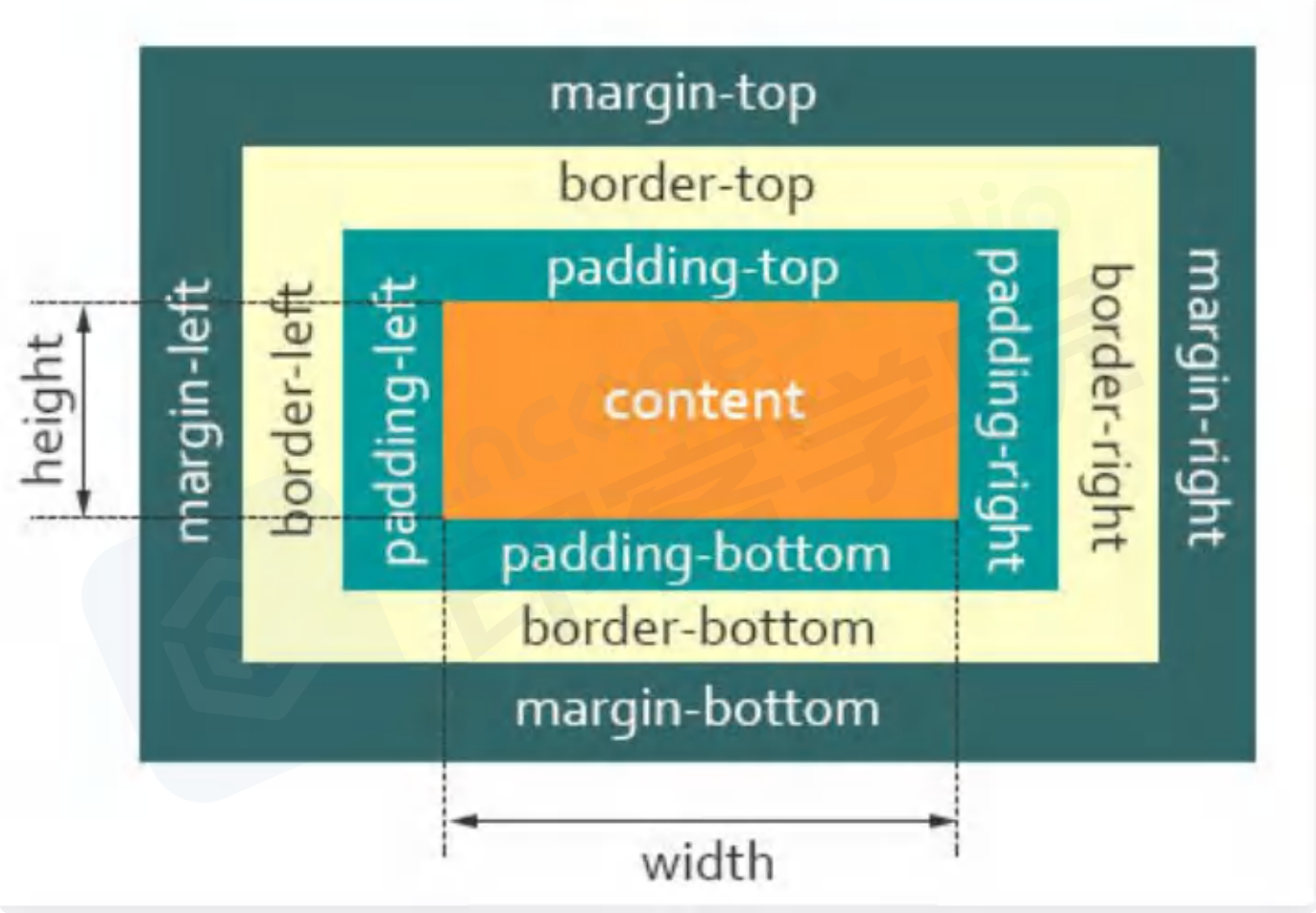
1. 说说你对盒子模型的理解?



1.1. 是什么

当对一个文档进行布局（layout）的时候，浏览器的渲染引擎会根据标准之一的 CSS 基础框盒模型（CSS basic box model），将所有元素表示为一个个矩形的盒子（box）

一个盒子由四个部分组成：`content`、`padding`、`border`、`margin`



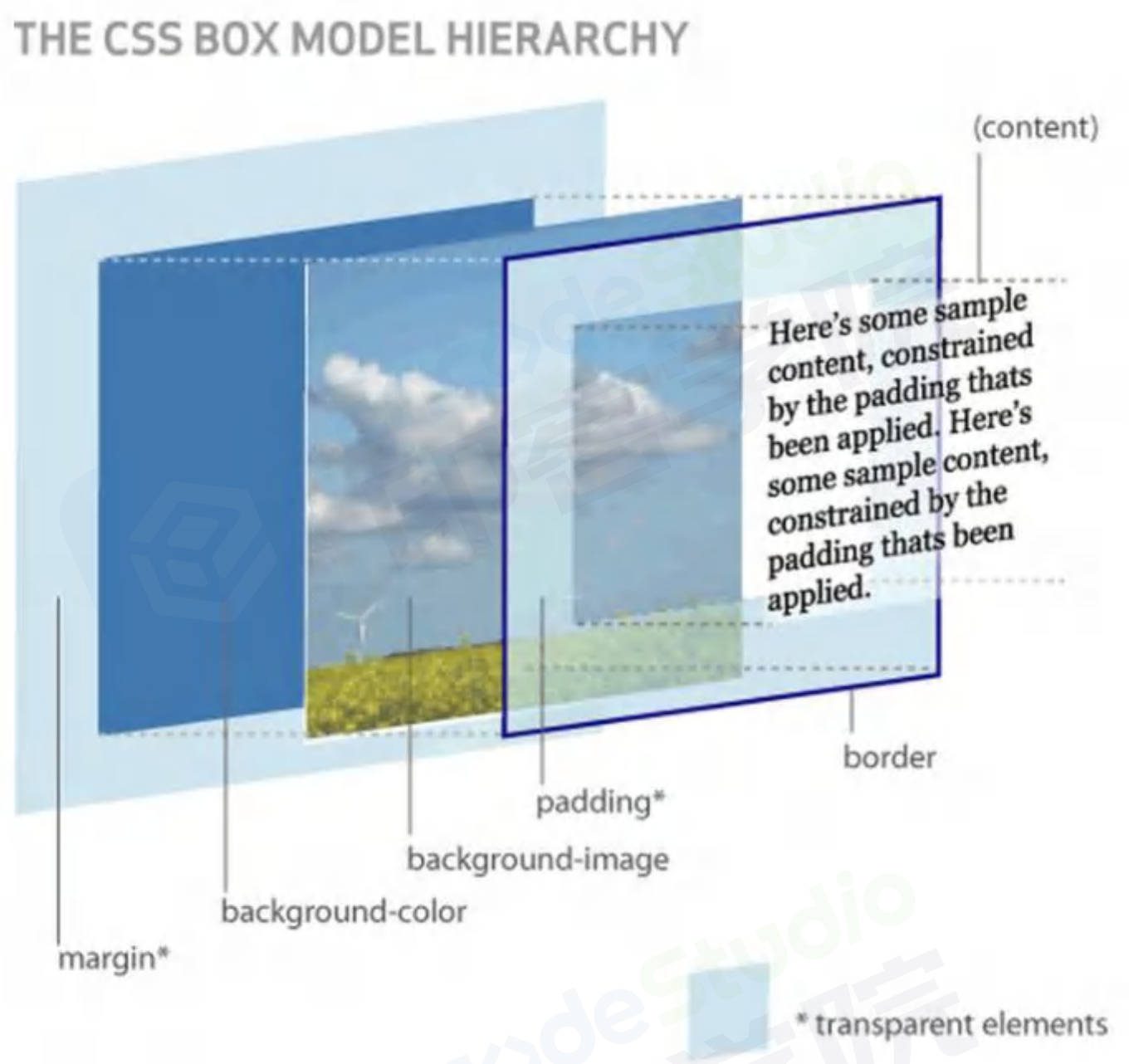
`content`，即实际内容，显示文本和图像

`boreder`，即边框，围绕元素内容的内边距的一条或多条线，由粗细、样式、颜色三部分组成

`padding`，即内边距，清除内容周围的区域，内边距是透明的，取值不能为负，受盒子的`background`属性影响

`margin`，即外边距，在元素外创建额外的空白，空白通常指不能放其他元素的区域

上述是一个从二维的角度观察盒子，下面再看看看三维图：



下面来段代码：

HTML | 复制代码

```
<style>
  .box {
    width: 200px;
    height: 100px;
    padding: 20px;
  }
</style>
<div class="box">
  盒子模型
</div>
```

当我们在浏览器查看元素时，却发现元素的大小变成了 `240px`

这是因为，在 `CSS` 中，盒子模型可以分成：

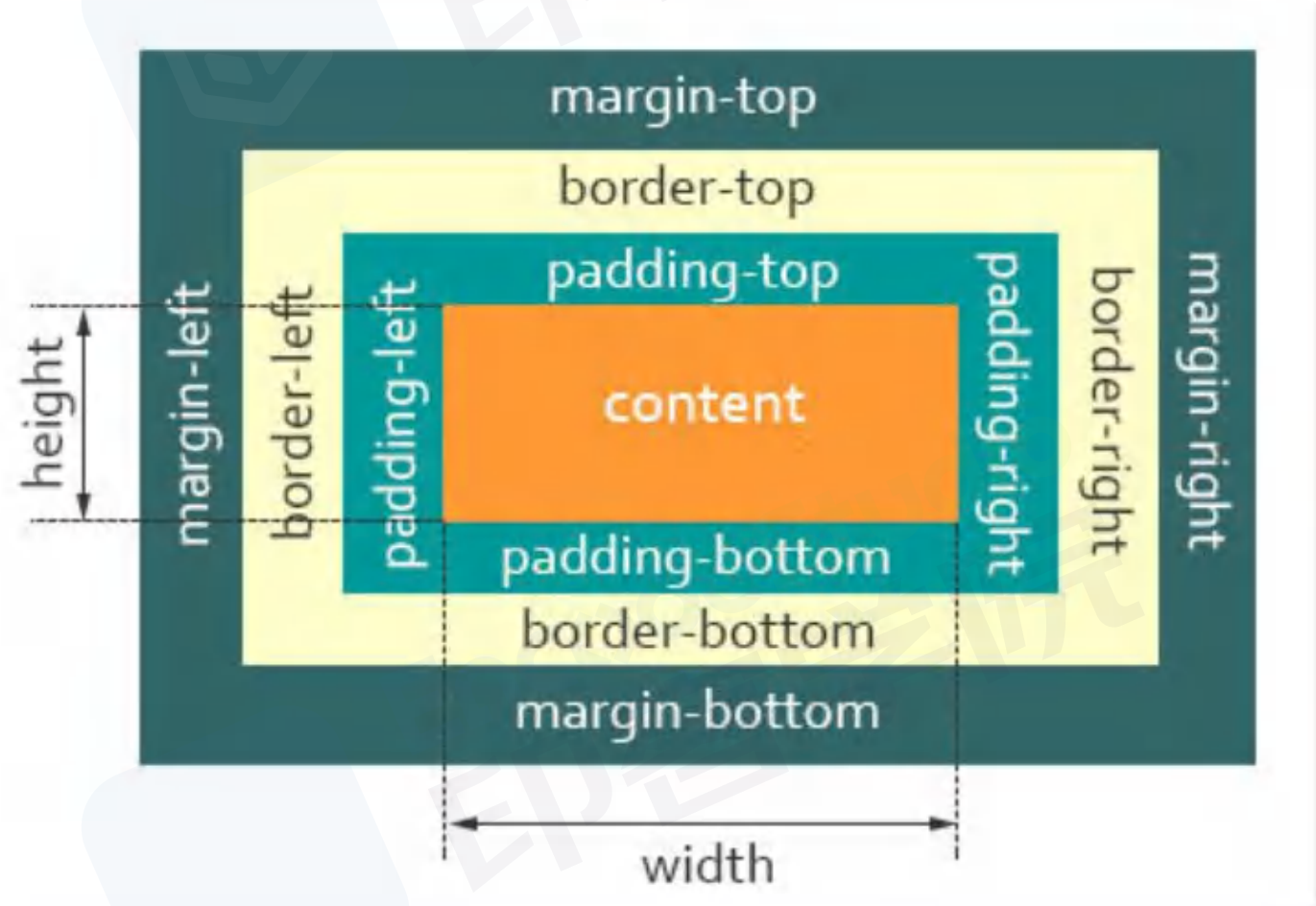
- W3C 标准盒子模型
- IE 怪异盒子模型

默认情况下，盒子模型为 `W3C` 标准盒子模型

1.2. 标准盒子模型

标准盒子模型，是浏览器默认的盒子模型

下面看看标准盒子模型的模型图：



从上图可以看到：

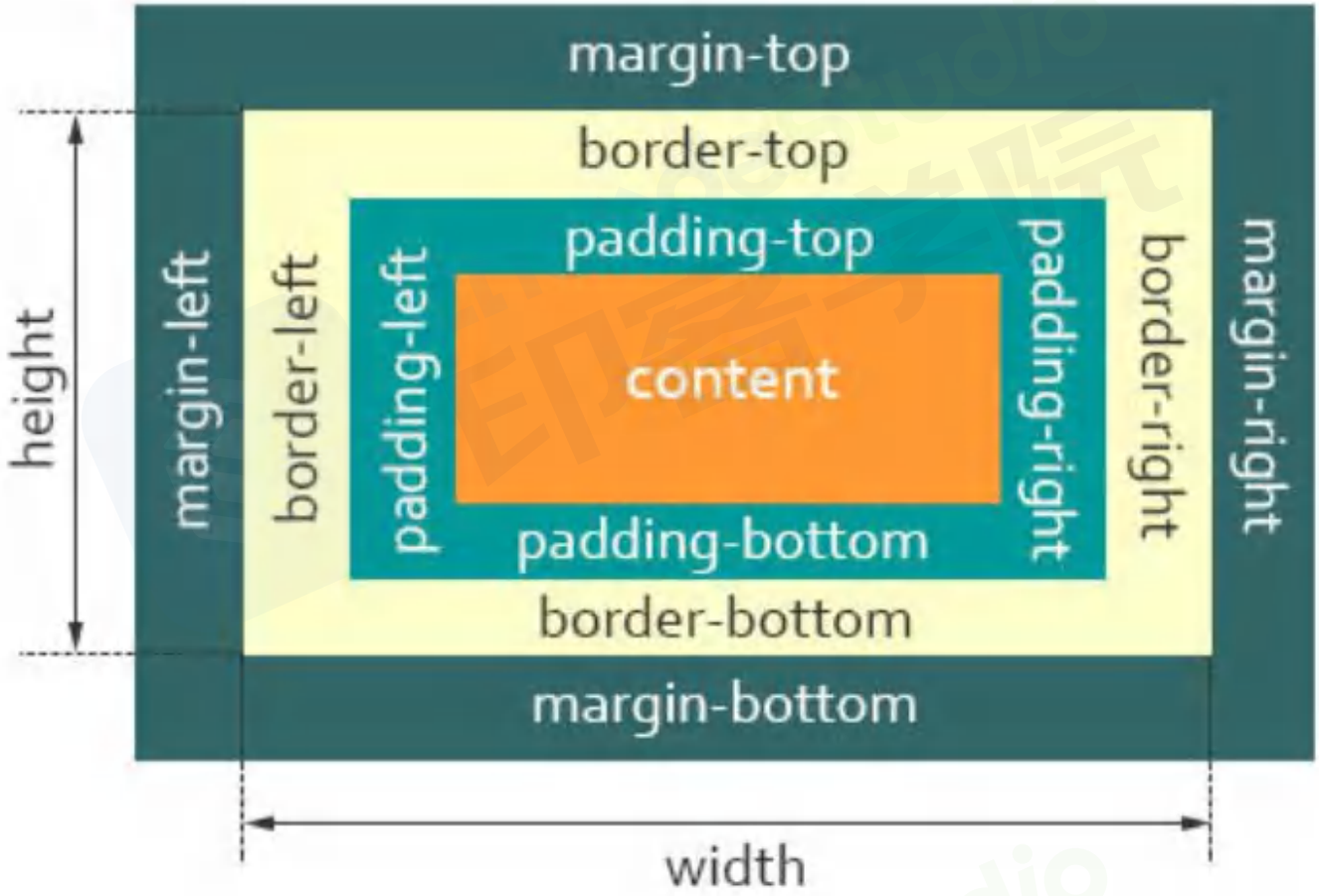
- 盒子总宽度 = width + padding + border + margin;
- 盒子总高度 = height + padding + border + margin

也就是，`width/height` 只是内容高度，不包含 `padding` 和 `border` 值

所以上面问题中，设置 `width` 为200px，但由于存在 `padding` ，但实际上盒子的宽度有240px

1.3. IE 怪异盒子模型

同样看看IE 怪异盒子模型的模型图：



从上图可以看到：

- 盒子总宽度 = width + margin;
- 盒子总高度 = height + margin;

也就是，`width/height` 包含了 `padding` 和 `border` 值

1.4. Box-sizing

CSS 中的 box-sizing 属性定义了引擎应该如何计算一个元素的总宽度和总高度

语法：

```css
box-sizing: content-box|border-box|inherit:
```

- content-box 默认值，元素的 width/height 不包含padding，border，与标准盒子模型表现一致

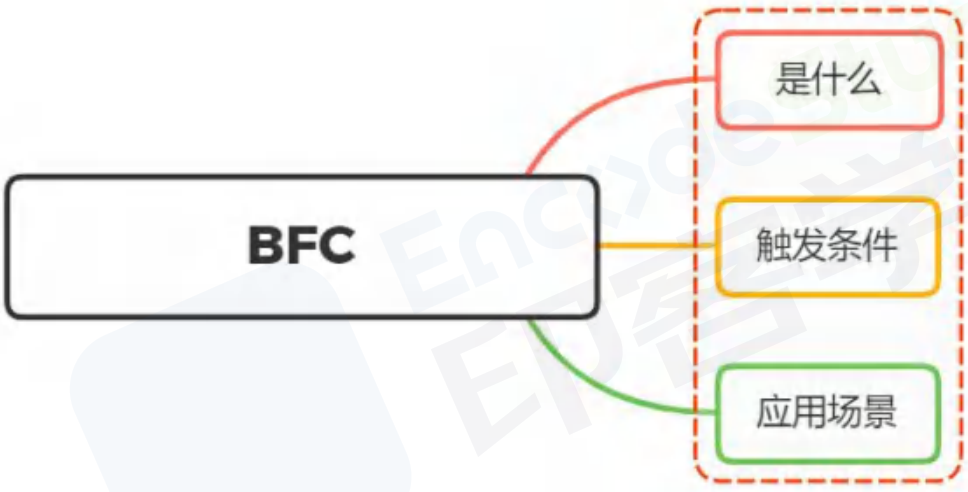
- border-box 元素的 width/height 包含 padding，border，与怪异盒子模型表现一致
- inherit 指定 box-sizing 属性的值，应该从父元素继承

回到上面的例子里，设置盒子为 border-box 模型

```HTML
<style>
  .box {
    width: 200px;
    height: 100px;
    padding: 20px;
    box-sizing: border-box;
  }
</style>
<div class="box">
  盒子模型
</div>
```

这时候，就可以发现盒子的所占据的宽度为200px

2. 谈谈你对BFC的理解?



2.1. 是什么

我们在页面布局的时候，经常出现以下情况：

- 这个元素高度怎么没了?

- 这两栏布局怎么没法自适应?
- 这两个元素的间距怎么有点奇怪的样子?
- ……

原因是元素之间相互的影响，导致了意料之外的情况，这里就涉及到 `BFC` 概念

`BFC` （Block Formatting Context），即块级格式化上下文，它是页面中的一块渲染区域，并且有一套属于自己的渲染规则：

- 内部的盒子会在垂直方向上一个接一个的放置
- 对于同一个BFC的俩个相邻的盒子的margin会发生重叠，与方向无关。
- 每个元素的左外边距与包含块的左边界相接触（从左到右），即使浮动元素也是如此
- BFC的区域不会与float的元素区域重叠
- 计算BFC的高度时，浮动子元素也参与计算
- 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，反之亦然

`BFC` 目的是形成一个相对于外界完全独立的空间，让内部的子元素不会影响到外部的元素

2.2. 触发条件

触发 `BFC` 的条件包含不限于：

- 根元素，即HTML元素
- 浮动元素：float值为left、right
- overflow值不为 visible，为 auto、scroll、hidden
- display的值为inline-block、inltable-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid
- position的值为absolute或fixed

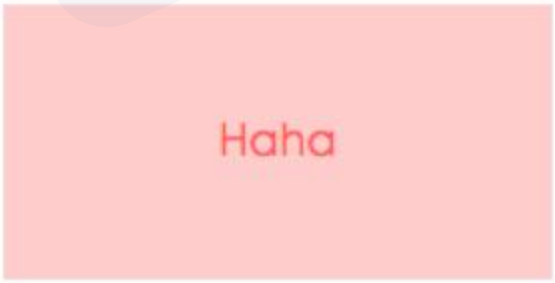
2.3. 应用场景

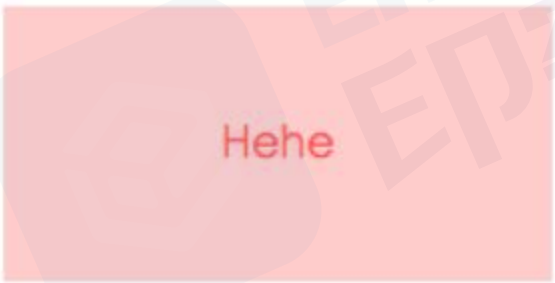
利用 `BFC` 的特性，我们将 `BFC` 应用在以下场景：

2.3.1. 防止margin重叠（塌陷）

```HTML
<style>
    p {
        color: #f55;
        background: #fcc;
        width: 200px;
        line-height: 100px;
        text-align:center;
        margin: 100px;
    }
</style>
<body>
    <p>Haha</p >
    <p>Hehe</p >
</body>
```

页面显示如下：





两个 `p` 元素之间的距离为 `100px`，发生了 `margin` 重叠（塌陷），以最大的为准，如果第一个P的 `margin` 为80的话，两个P之间的距离还是100，以最大的为准。

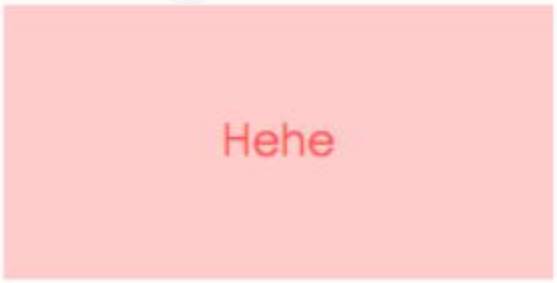
前面讲到，同一个 `BFC` 的俩个相邻的盒子的 `margin` 会发生重叠

可以在 `p` 外面包裹一层容器，并触发这个容器生成一个 `BFC` ，那么两个 `p` 就不属于同一个 `BFC` ，则不会出现 `margin` 重叠

HTML | 复制代码

```html
<style>
    .wrap {
        overflow: hidden;// 新的BFC
    }
    p {
        color: #f55;
        background: #fcc;
        width: 200px;
        line-height: 100px;
        text-align:center;
        margin: 100px;
    }
</style>
<body>
    <p>Haha</p >
    <div class="wrap">
        <p>Hehe</p >
    </div>
</body>
```

这时候，边距则不会重叠：



2.3.2. 清除内部浮动

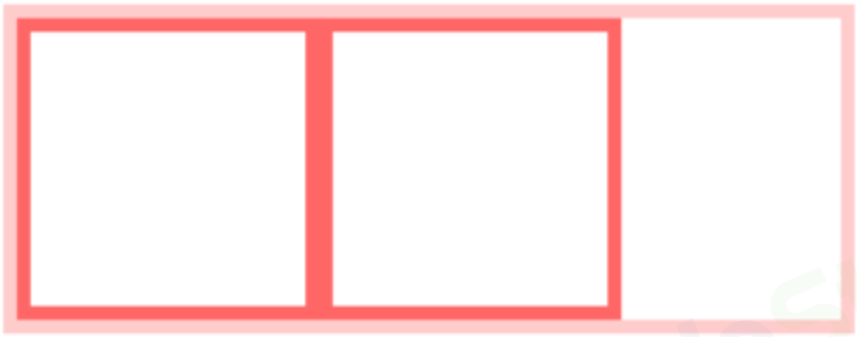
```html
<style>
    .par {
        border: 5px solid #fcc;
        width: 300px;
    }

    .child {
        border: 5px solid #f66;
        width:100px;
        height: 100px;
        float: left;
    }
</style>
<body>
    <div class="par">
        <div class="child"></div>
        <div class="child"></div>
    </div>
</body>
```

页面显示如下：

而 `BFC` 在计算高度时，浮动元素也会参与，所以我们可以触发 `.par` 元素生成 `BFC`，则内部浮动元素计算高度时候也会计算

```css
.par {
    overflow: hidden;
}
```

实现效果如下：



2.3.3. 自适应多栏布局

这里举个两栏的布局

HTML | 复制代码

```
<style>
    body {
        width: 300px;
        position: relative;
    }

    .aside {
        width: 100px;
        height: 150px;
        float: left;
        background: #f66;
    }

    .main {
        height: 200px;
        background: #fcc;
    }
</style>
<body>
    <div class="aside"></div>
    <div class="main"></div>
</body>
```

效果图如下：

前面讲到，每个元素的左外边距与包含块的左边界相接触

因此，虽然 `.aslide` 为浮动元素，但是 `main` 的左边依然会与包含块的左边相接触

而 `BFC` 的区域不会与浮动盒子重叠

所以我们可以通过触发 `main` 生成 `BFC` ，以此适应两栏布局

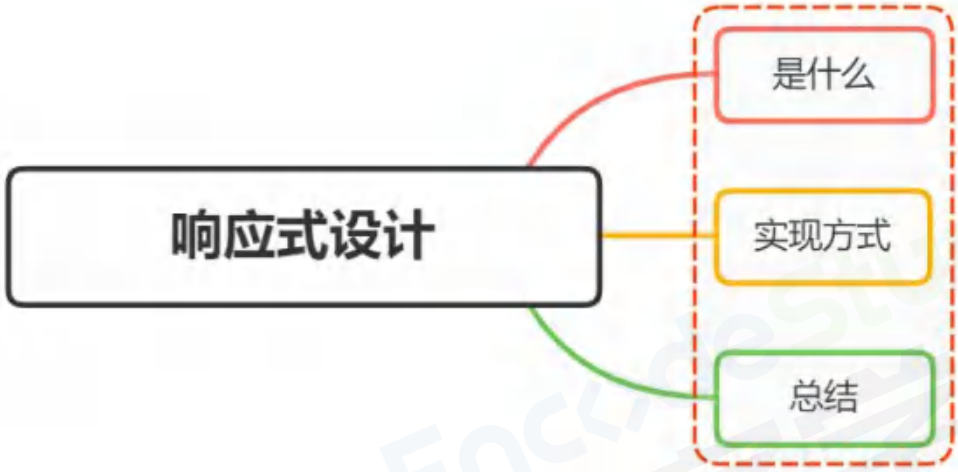
CSS | 复制代码

```css
.main {
    overflow: hidden;
}
```

这时候，新的 `BFC` 不会与浮动的 `.aside` 元素重叠。因此会根据包含块的宽度，和 `.aside` 的宽度，自动变窄

效果如下：

3. 什么是响应式设计？响应式设计的基本原理是什么？如何做？



3.1. 是什么

响应式网站设计（Responsive Web design）是一种网络页面设计布局，页面的设计与开发应当根据用户行为以及设备环境(系统平台、屏幕尺寸、屏幕定向等)进行相应的响应和调整

描述响应式界面最著名的一句话就是“Content is like water”

大白话便是“如果将屏幕看作容器，那么内容就像水一样”

响应式网站常见特点：

- 同时适配PC + 平板 + 手机等

- 标签导航在接近手持终端设备时改变为经典的抽屉式导航
- 网站的布局会根据视口来调整模块的大小和位置

3.2. 实现方式

响应式设计的基本原理是通过媒体查询检测不同的设备屏幕尺寸做处理，为了处理移动端，页面头部必须有 `meta` 声明 `viewport`

```HTML
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no”>
```

属性对应如下：

- width=device-width: 是自适应手机屏幕的尺寸宽度
- maximum-scale:是缩放比例的最大值
- inital-scale:是缩放的初始化
- user-scalable:是用户的可以缩放的操作

实现响应式布局的方式有如下：

- 媒体查询
- 百分比
- vw/vh
- rem

3.2.1. 媒体查询

`CSS3` 中的增加了更多的媒体查询，就像 `if` 条件表达式一样，我们可以设置不同类型的媒体条件，并根据对应的条件，给相应符合条件的媒体调用相对应的样式表

使用 `@Media` 查询，可以针对不同的媒体类型定义不同的样式，如：

```css
@media screen and (max-width: 1920px) { ... }
```

当视口在375px – 600px之间，设置特定字体大小18px

```css
@media screen (min-width: 375px) and (max-width: 600px) {
  body {
    font-size: 18px;
  }
}
```

通过媒体查询，可以通过给不同分辨率的设备编写不同的样式来实现响应式的布局，比如我们为不同分辨率的屏幕，设置不同的背景图片

比如给小屏幕手机设置@2x图，为大屏幕手机设置@3x图，通过媒体查询就能很方便的实现

3.2.2. 百分比

通过百分比单位 " % " 来实现响应式的效果

比如当浏览器的宽度或者高度发生变化时，通过百分比单位，可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果

`height` 、 `width` 属性的百分比依托于父标签的宽高，但是其他盒子属性则不完全依赖父元素：

- 子元素的top/left和bottom/right如果设置百分比，则相对于直接非static定位(默认定位)的父元素的高度/宽度
- 子元素的padding如果设置百分比，不论是垂直方向或者是水平方向，都相对于直接父亲元素的width，而与父元素的height无关。
- 子元素的margin如果设置成百分比，不论是垂直方向还是水平方向，都相对于直接父元素的width
- border-radius不一样，如果设置border-radius为百分比，则是相对于自身的宽度

可以看到每个属性都使用百分比，会照成布局的复杂度，所以不建议使用百分比来实现响应式

3.2.3. vw/vh

`vw` 表示相对于视图窗口的宽度， `vh` 表示相对于视图窗口高度。 任意层级元素，在使用 `vw` 单位的情况下， `1vw` 都等于视图宽度的百分之一

与百分比布局很相似，在以前文章提过与 `%` 的区别，这里就不再展开述说

3.2.4. rem

在以前也讲到， `rem` 是相对于根元素 `html` 的 `font-size` 属性，默认情况下浏览器字体大小为 `16px` ，此时 `1rem = 16px`

可以利用前面提到的媒体查询，针对不同设备分辨率改变 `font-size` 的值，如下：

CSS | 复制代码

```css
@media screen and (max-width: 414px) {
  html {
    font-size: 18px
  }
}

@media screen and (max-width: 375px) {
  html {
    font-size: 16px
  }
}

@media screen and (max-width: 320px) {
  html {
    font-size: 12px
  }
}
```

为了更准确监听设备可视窗口变化，我们可以在 `css` 之前插入 `script` 标签，内容如下：

```javascript
//动态为根元素设置字体大小
function init () {
    // 获取屏幕宽度
    var width = document.documentElement.clientWidth
    // 设置根元素字体大小。此时为宽的10等分
    document.documentElement.style.fontSize = width / 10 + 'px'
}

//首次加载应用，设置一次
init()
// 监听手机旋转的事件的时机，重新设置
window.addEventListener('orientationchange', init)
// 监听手机窗口变化，重新设置
window.addEventListener('resize', init)
```

无论设备可视窗口如何变化，始终设置 `rem` 为 `width` 的1/10，实现了百分比布局

除此之外，我们还可以利用主流 `UI` 框架，如： `element ui` 、 `antd` 提供的栅格布局实现响应式

3.2.5. 小结

响应式设计实现通常会从以下几方面思考：

- 弹性盒子（包括图片、表格、视频）和媒体查询等技术
- 使用百分比布局创建流式布局的弹性UI，同时使用媒体查询限制元素的尺寸和内容变更范围
- 使用相对单位使得内容自适应调节
- 选择断点，针对不同断点实现不同布局和内容展示

3.3. 总结

响应式布局优点可以看到：

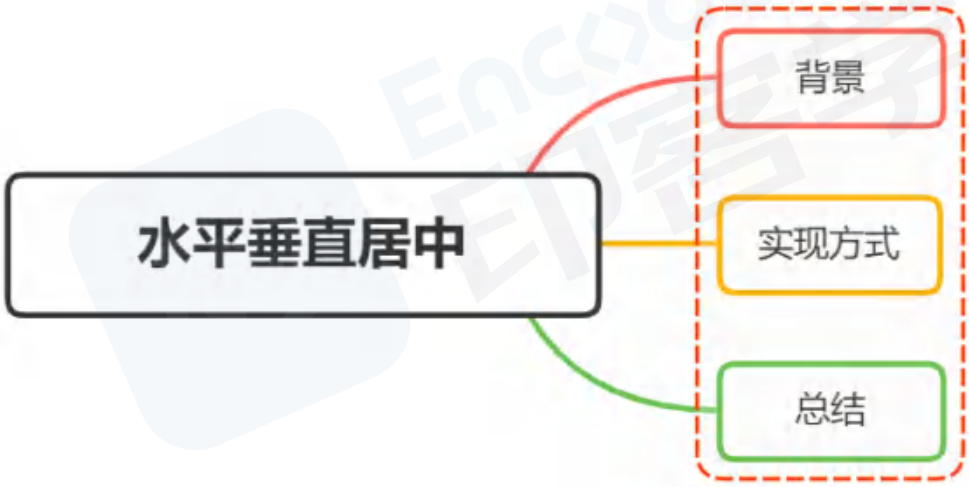
- 面对不同分辨率设备灵活性强
- 能够快捷解决多设备显示适应问题

缺点：

- 仅适用布局、信息、框架并不复杂的部门类型网站
- 兼容各种设备工作量大，效率低下
- 代码累赘，会出现隐藏无用的元素，加载时间加长

- 其实这是一种折中性质的设计解决方案，多方面因素影响而达不到最佳效果
- 一定程度上改变了网站原有的布局结构，会出现用户混淆的情况

4. 元素水平垂直居中的方法有哪些？如果元素不定宽高呢？



4.1. 背景

在开发中经常遇到这个问题，即让某个元素的内容在水平和垂直方向上都居中，内容不仅限于文字，可能是图片或其他元素

居中是一个非常基础但又是非常重要的应用场景，实现居中的方法存在很多，可以将这些方法分成两个大类：

- 居中元素（子元素）的宽高已知
- 居中元素宽高未知

4.2. 实现方式

实现元素水平垂直居中的方式：

- 利用定位+margin:auto
- 利用定位+margin:负值
- 利用定位+transform
- table布局

- flex布局
- grid布局

4.2.1. 利用定位+margin:auto

先上代码：

HTML | 复制代码

```
<style>
    .father{
        width:500px;
        height:300px;
        border:1px solid #0a3b98;
        position: relative;
    }
    .son{
        width:100px;
        height:40px;
        background: #f0a238;
        position: absolute;
        top:0;
        left:0;
        right:0;
        bottom:0;
        margin:auto;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

父级设置为相对定位，子级绝对定位 ，并且四个定位属性的值都设置了0，那么这时候如果子级没有设置宽高，则会被拉开到和父级一样宽高

这里子元素设置了宽高，所以宽高会按照我们的设置来显示，但是实际上子级的虚拟占位已经撑满了整个父级，这时候再给它一个 `margin: auto` 它就可以上下左右都居中了

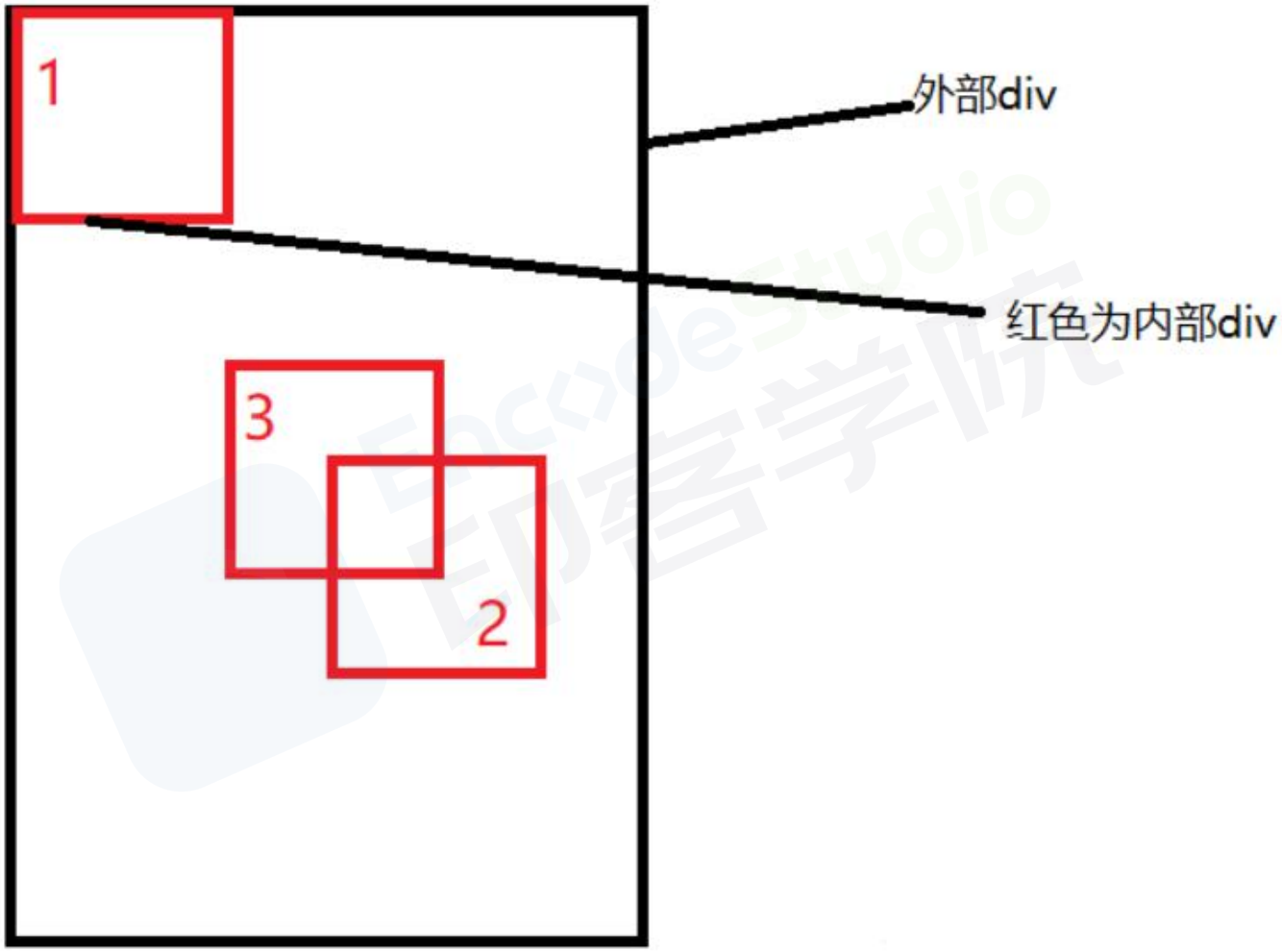
4.2.2. 利用定位+margin:负值

绝大多数情况下，设置父元素为相对定位， 子元素移动自身50%实现水平垂直居中

HTML | 复制代码

```html
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        margin-left:-50px;
        margin-top:-50px;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

整个实现思路如下图所示：



- 初始位置为方块1的位置
- 当设置left、top为50%的时候，内部子元素为方块2的位置
- 设置margin为负数时，使内部子元素到方块3的位置，即中间位置

这种方案不要求父元素的高度，也就是即使父元素的高度变化了，仍然可以保持在父元素的垂直居中位置，水平方向上是一样的操作

但是该方案需要知道子元素自身的宽高，但是我们可以通过下面 `transform` 属性进行移动

4.2.3. 利用定位+transform

实现代码如下：

```css
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%,-50%);
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

`translate(-50%, -50%)` 将会将元素位移自己宽度和高度的-50%

这种方法其实和最上面被否定掉的margin负值用法一样，可以说是 `margin` 负值的替代方案，并不需要知道自身元素的宽高

4.2.4. table布局

设置父元素为 `display:table-cell` ，子元素设置 `display: inline-block` 。利用 `vertical` 和 `text-align` 可以让所有的行内块级元素水平垂直居中

HTML | 复制代码

```html
<style>
    .father {
        display: table-cell;
        width: 200px;
        height: 200px;
        background: skyblue;
        vertical-align: middle;
        text-align: center;
    }
    .son {
        display: inline-block;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

4.2.5. flex弹性布局

还是看看实现的整体代码：

HTML | 复制代码

```html
<style>
    .father {
        display: flex;
        justify-content: center;
        align-items: center;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

`css3` 中了 `flex` 布局，可以非常简单实现垂直水平居中

这里可以简单看看 `flex` 布局的关键属性作用：

- display: flex时，表示该容器内部的元素将按照flex进行布局
- align-items: center表示这些元素将相对于本容器水平居中
- justify-content: center也是同样的道理垂直居中

4.2.6. grid网格布局

```HTML
<style>
    .father {
            display: grid;
            align-items:center;
            justify-content: center;
            width: 200px;
            height: 200px;
            background: skyblue;

        }
        .son {
            width: 10px;
            height: 10px;
            border: 1px solid red
        }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

这里看到，`gird` 网格布局和 `flex` 弹性布局都简单粗暴

4.2.7. 小结

上述方法中，不知道元素宽高大小仍能实现水平垂直居中的方法有：

- 利用定位+margin:auto
- 利用定位+transform
- flex布局
- grid布局

4.3. 总结

根据元素标签的性质，可以分为：

- 内联元素居中布局
- 块级元素居中布局

4.3.1. 内联元素居中布局

水平居中

- 行内元素可设置：text-align: center
- flex布局设置父元素：display: flex; justify-content: center

垂直居中

- 单行文本父元素确认高度：height === line-height
- 多行文本父元素确认高度：display: table-cell; vertical-align: middle

4.3.2. 块级元素居中布局

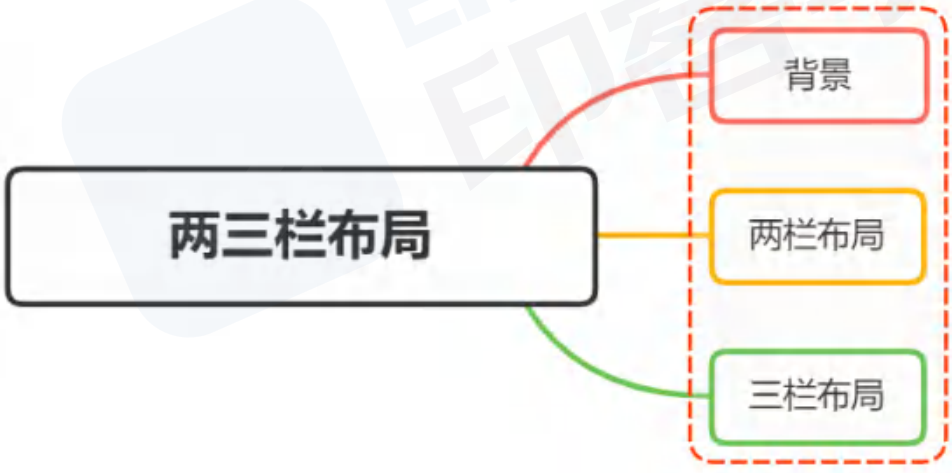
水平居中

- 定宽: margin: 0 auto
- 绝对定位+left:50%+margin:负自身一半

垂直居中

- position: absolute设置left、top、margin-left、margin-top(定高)
- display: table-cell
- transform: translate(x, y)
- flex(不定高，不定宽)
- grid(不定高，不定宽)，兼容性相对比较差

5. 如何实现两栏布局，右侧自适应？三栏布局中间自适应呢？



5.1. 背景

在日常布局中，无论是两栏布局还是三栏布局，使用的频率都非常高

5.1.1. 两栏布局

两栏布局实现效果就是将页面分割成左右宽度不等的两列，宽度较小的列设置为固定宽度，剩余宽度由另一列撑满，

比如 `Ant Design` 文档，蓝色区域为主要内容布局容器，侧边栏为次要内容布局容器

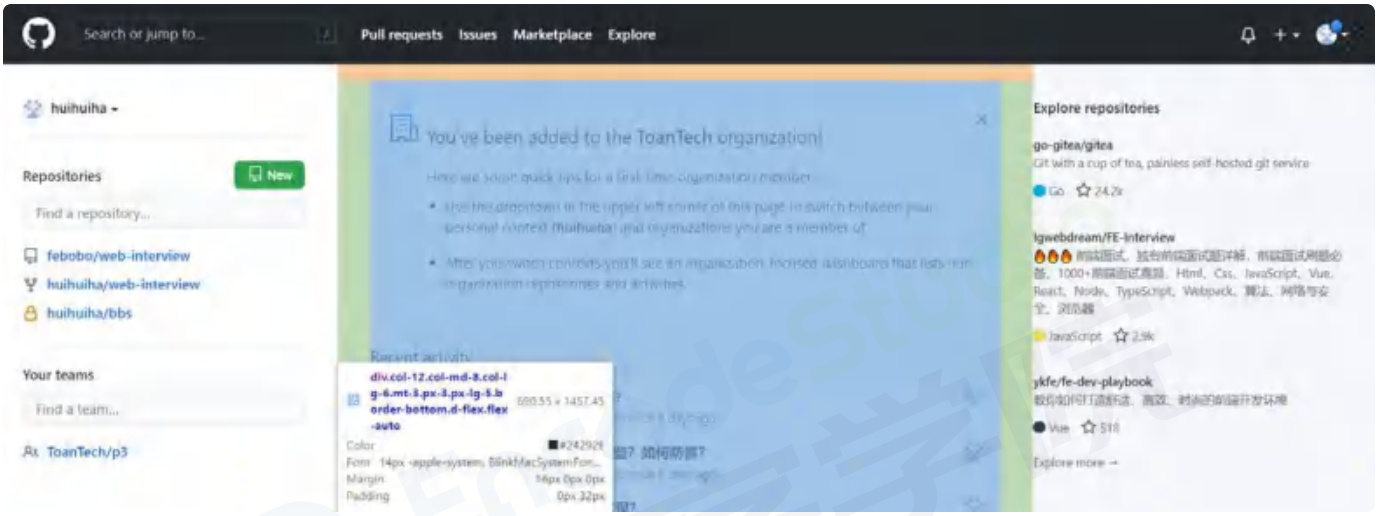
> 这里称宽度较小的列父元素为次要布局容器，宽度较大的列父元素为主要布局容器

这种布局适用于内容上具有明显主次关系的网页

5.1.2. 三栏布局

三栏布局按照左中右的顺序进行排列，通常中间列最宽，左右两列次之

大家最常见的就是 `github` ：



5.2. 两栏布局

两栏布局非常常见，往往是以一个定宽栏和一个自适应的栏并排展示存在

实现思路也非常的简单：

- 使用 float 左浮左边栏
- 右边模块使用 margin-left 撑出内容块做内容展示
- 为父级元素添加BFC，防止下方元素飞到上方内容

代码如下：

```HTML
<style>
    .box{
        overflow: hidden; 添加BFC
    }
    .left {
        float: left;
        width: 200px;
        background-color: gray;
        height: 400px;
    }
    .right {
        margin-left: 210px;
        background-color: lightgray;
        height: 200px;
    }
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>
```

还有一种更为简单的使用则是采取：flex弹性布局

5.2.1. flex弹性布局

```HTML
<style>
    .box{
        display: flex;
    }
    .left {
        width: 100px;
    }
    .right {
        flex: 1;
    }
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>
```

`flex` 可以说是最好的方案了，代码少，使用简单

注意的是， `flex` 容器的一个默认属性值: `align-items: stretch;`

这个属性导致了列等高的效果。 为了让两个盒子高度自动，需要设置: `align-items: flex-start`

5.3. 三栏布局

实现三栏布局中间自适应的布局方式有：

- 两边使用 float，中间使用 margin
- 两边使用 absolute，中间使用 margin
- 两边使用 float 和负 margin
- display: table 实现
- flex实现
- grid网格布局

5.3.1. 两边使用 float，中间使用 margin

需要将中间的内容放在 `html` 结构最后，否则右侧会臣在中间内容的下方

实现代码如下：

HTML | 复制代码

```html
<style>
    .wrap {
        background: #eee;
        overflow: hidden; <!-- 生成BFC，计算高度时考虑浮动的元素 -->
        padding: 20px;
        height: 200px;
    }
    .left {
        width: 200px;
        height: 200px;
        float: left;
        background: coral;
    }
    .right {
        width: 120px;
        height: 200px;
        float: right;
        background: lightblue;
    }
    .middle {
        margin-left: 220px;
        height: 200px;
        background: lightpink;
        margin-right: 140px;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="right">右侧</div>
    <div class="middle">中间</div>
</div>
```

原理如下：

- 两边固定宽度，中间宽度自适应。
- 利用中间元素的margin值控制两边的间距
- 宽度小于左右部分宽度之和时，右侧部分会被挤下去

这种实现方式存在缺陷：

- 主体内容是最后加载的。
- 右边在主体内容之前，如果是响应式设计，不能简单的换行展示

5.3.2. 两边使用 absolute，中间使用 margin

基于绝对定位的三栏布局：注意绝对定位的元素脱离文档流，相对于最近的已经定位的祖先元素进行定位。无需考虑HTML中结构的顺序

HTML | 复制代码

```html
<style>
  .container {
    position: relative;
  }

  .left,
  .right,
  .main {
    height: 200px;
    line-height: 200px;
    text-align: center;
  }

  .left {
    position: absolute;
    top: 0;
    left: 0;
    width: 100px;
    background: green;
  }

  .right {
    position: absolute;
    top: 0;
    right: 0;
    width: 100px;
    background: green;
  }

  .main {
    margin: 0 110px;
    background: black;
    color: white;
  }
</style>

<div class="container">
  <div class="left">左边固定宽度</div>
  <div class="right">右边固定宽度</div>
  <div class="main">中间自适应</div>
</div>
```

实现流程：

- 左右两边使用绝对定位，固定在两侧。

- 中间占满一行，但通过 margin和左右两边留出10px的间隔

5.3.3. 两边使用 float 和负 margin

HTML | 复制代码

```
<style>
  .left,
  .right,
  .main {
    height: 200px;
    line-height: 200px;
    text-align: center;
  }

  .main-wrapper {
    float: left;
    width: 100%;
  }

  .main {
    margin: 0 110px;
    background: black;
    color: white;
  }

  .left,
  .right {
    float: left;
    width: 100px;
    margin-left: -100%;
    background: green;
  }

  .right {
    margin-left: -100px; /* 同自身宽度 */
  }
</style>

<div class="main-wrapper">
  <div class="main">中间自适应</div>
</div>
<div class="left">左边固定宽度</div>
<div class="right">右边固定宽度</div>
```

实现过程：

- 中间使用了双层标签，外层是浮动的，以便左中右能在同一行展示
- 左边通过使用负 margin-left:-100%，相当于中间的宽度，所以向上偏移到左侧
- 右边通过使用负 margin-left:-100px，相当于自身宽度，所以向上偏移到最右侧

缺点：

- 增加了 .main-wrapper 一层，结构变复杂
- 使用负 margin，调试也相对麻烦

5.3.4. 使用 display: table 实现

`<table>` 标签用于展示行列数据，不适合用于布局。但是可以使用 `display: table` 来实现布局的效果

HTML 复制代码

```html
<style>
  .container {
    height: 200px;
    line-height: 200px;
    text-align: center;
    display: table;
    table-layout: fixed;
    width: 100%;
  }

  .left,
  .right,
  .main {
    display: table-cell;
  }

  .left,
  .right {
    width: 100px;
    background: green;
  }

  .main {
    background: black;
    color: white;
    width: 100%;
  }
</style>

<div class="container">
  <div class="left">左边固定宽度</div>
  <div class="main">中间自适应</div>
  <div class="right">右边固定宽度</div>
</div>
```

实现原理：

- 层通过 display: table设置为表格，设置 table-layout: fixed`表示列宽自身宽度决定，而不是自动计算。
- 内层的左中右通过 display: table-cell设置为表格单元。
- 左右设置固定宽度，中间设置 width: 100% 填充剩下的宽度

5.3.5. 使用flex实现

利用 `flex` 弹性布局，可以简单实现中间自适应

代码如下：

HTML | 复制代码

```
<style type="text/css">
    .wrap {
        display: flex;
        justify-content: space-between;
    }

    .left,
    .right,
    .middle {
        height: 100px;
    }

    .left {
        width: 200px;
        background: coral;
    }

    .right {
        width: 120px;
        background: lightblue;
    }

    .middle {
        background: #555;
        width: 100%;
        margin: 0 20px;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="middle">中间</div>
    <div class="right">右侧</div>
</div>
```

实现过程：

- 仅需将容器设置为 `display:flex;` ，
- 盒内元素两端对其，将中间元素设置为 `100%` 宽度，或者设为 `flex:1` ，即可填充空白
- 盒内元素的高度撑开容器的高度

优点：

- 结构简单直观
- 可以结合 flex的其他功能实现更多效果，例如使用 order属性调整显示顺序，让主体内容优先加载，但展示在中间

5.3.6. grid网格布局

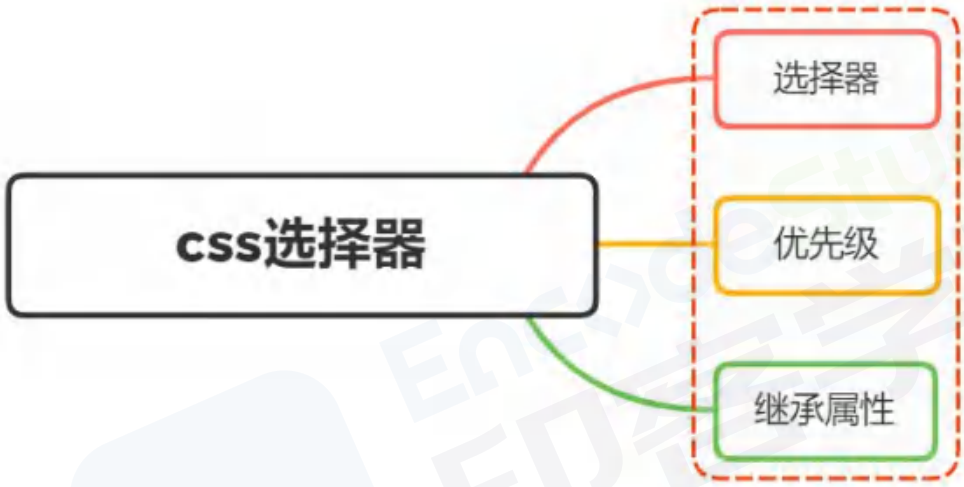
代码如下：

HTML | 复制代码

```html
<style>
    .wrap {
        display: grid;
        width: 100%;
        grid-template-columns: 300px auto 300px;
    }

    .left,
    .right,
    .middle {
        height: 100px;
    }

    .left {
        background: coral;
    }

    .right {
        background: lightblue;
    }

    .middle {
        background: #555;
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="middle">中间</div>
    <div class="right">右侧</div>
</div>
```

跟 `flex` 弹性布局一样的简单

6. css选择器有哪些？优先级？哪些属性可以继承？



6.1. 选择器

CSS选择器是CSS规则的第一部分

它是元素和其他部分组合起来告诉浏览器哪个HTML元素应当是被选为应用规则中的CSS属性值的方式

选择器所选择的元素，叫做“选择器的对象”

我们从一个 `Html` 结构开始

HTML | 复制代码

```html
<div id="box">
    <div class="one">
        <p class="one_1">
        </p >
        <p class="one_1">
        </p >
    </div>
    <div class="two"></div>
    <div class="two"></div>
    <div class="two"></div>
</div>
```

关于 `css` 属性选择器常用的有：

- id选择器（#box），选择id为box的元素
- 类选择器（.one），选择类名为one的所有元素
- 标签选择器（div），选择标签为div的所有元素

- 后代选择器（#box div），选择id为box元素内部所有的div元素
- 子选择器（.one>one_1），选择父元素为.one的所有.one_1的元素
- 相邻同胞选择器（.one+.two），选择紧接在.one之后的所有.two元素
- 群组选择器（div,p），选择div、p的所有元素

还有一些使用频率相对没那么多的选择器：

- 伪类选择器

```css
:link ： 选择未被访问的链接
:visited：选取已被访问的链接
:active：选择活动链接
:hover ： 鼠标指针浮动在上面的元素
:focus ： 选择具有焦点的
:first-child：父元素的首个子元素
```

- 伪元素选择器

```css
:first-letter ： 用于选取指定选择器的首字母
:first-line ： 选取指定选择器的首行
:before ： 选择器在被选元素的内容前面插入内容
:after ： 选择器在被选元素的内容后面插入内容
```

- 属性选择器

```css
[attribute] 选择带有attribute属性的元素
[attribute=value] 选择所有使用attribute=value的元素
[attribute~=value] 选择attribute属性包含value的元素
[attribute|=value]：选择attribute属性以value开头的元素
```

在 `CSS3` 中新增的选择器有如下：

- 层次选择器（p~ul），选择前面有p元素的每个ul元素
- 伪类选择器

```css
:first-of-type 表示一组同级元素中其类型的第一个元素
:last-of-type 表示一组同级元素中其类型的最后一个元素
:only-of-type 表示没有同类型兄弟元素的元素
:only-child 表示没有任何兄弟的元素
:nth-child(n) 根据元素在一组同级中的位置匹配元素
:nth-last-of-type(n) 匹配给定类型的元素，基于它们在一组兄弟元素中的位置，从末尾开始计数
:last-child 表示一组兄弟元素中的最后一个元素
:root 设置HTML文档
:empty 指定空的元素
:enabled 选择可用元素
:disabled 选择被禁用元素
:checked 选择选中的元素
:not(selector) 选择与 <selector> 不匹配的所有元素
```

- 属性选择器

```css
[attribute*=value]：选择attribute属性值包含value的所有元素
[attribute^=value]：选择attribute属性开头为value的所有元素
[attribute$=value]：选择attribute属性结尾为value的所有元素
```

6.2. 优先级

相信大家对 `CSS` 选择器的优先级都不陌生：

> 内联 > ID选择器 > 类选择器 > 标签选择器

到具体的计算层面，优先级是由 A 、B、C、D 的值来决定的，其中它们的值计算规则如下：

- 如果存在内联样式，那么 A = 1, 否则 A = 0
- B的值等于 ID选择器出现的次数
- C的值等于 类选择器 和 属性选择器 和 伪类 出现的总次数
- D 的值等于 标签选择器 和 伪元素 出现的总次数

这里举个例子：

```css
#nav-global > ul > li > a.nav-link
```

套用上面的算法，依次求出 `A` `B` `C` `D` 的值：

- 因为没有内联样式 ，所以 A = 0
- ID选择器总共出现了1次， B = 1
- 类选择器出现了1次， 属性选择器出现了0次，伪类选择器出现0次，所以 C = (1 + 0 + 0) = 1
- 标签选择器出现了3次， 伪元素出现了0次，所以 D = (3 + 0) = 3

上面算出的 `A` 、 `B` 、 `C` 、 `D` 可以简记作： `(0, 1, 1, 3)`

知道了优先级是如何计算之后，就来看看比较规则：

- 从左往右依次进行比较 ，较大者优先级更高
- 如果相等，则继续往右移动一位进行比较
- 如果4位全部相等，则后面的会覆盖前面的

经过上面的优先级计算规则，我们知道内联样式的优先级最高，如果外部样式需要覆盖内联样式，就需要使用 `!important`

6.3. 继承属性

在 `css` 中，继承是指的是给父元素设置一些属性，后代元素会自动拥有这些属性

关于继承属性，可以分成：

- 字体系列属性

```css
font:组合字体
font-family:规定元素的字体系列
font-weight:设置字体的粗细
font-size:设置字体的尺寸
font-style:定义字体的风格
font-variant:偏大或偏小的字体
```

- 文本系列属性

```CSS
text-indent：文本缩进
text-align：文本水平对刘
line-height：行高
word-spacing：增加或减少单词间的空白
letter-spacing：增加或减少字符间的空白
text-transform：控制文本大小写
direction：规定文本的书写方向
color：文本颜色
```

- 元素可见性

```CSS
visibility
```

- 表格布局属性

```CSS
caption-side：定位表格标题位置
border-collapse：合并表格边框
border-spacing：设置相邻单元格的边框间的距离
empty-cells：单元格的边框的出现与消失
table-layout：表格的宽度由什么决定
```

- 列表属性

```CSS
list-style-type：文字前面的小点点样式
list-style-position：小点点位置
list-style：以上的属性可通过这属性集合
```

- 引用

```CSS
quotes：设置嵌套引用的引号类型
```

- 光标属性

```css
cursor: 箭头可以变成需要的形状
```

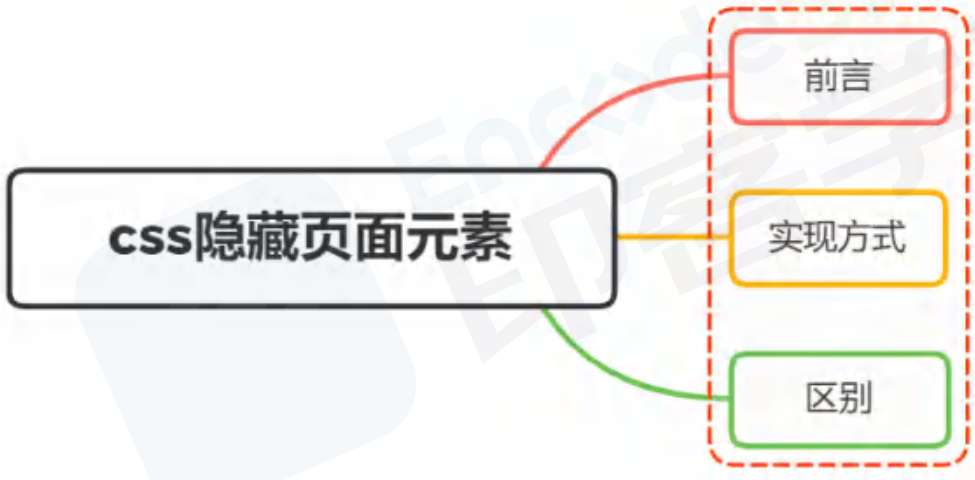
继承中比较特殊的几点：

- a 标签的字体颜色不能被继承
- h1–h6标签字体的大下也是不能被继承的

6.3.1. 无继承的属性

- display
- 文本属性：vertical–align、text–decoration
- 盒子模型的属性：宽度、高度、内外边距、边框等
- 背景属性：背景图片、颜色、位置等
- 定位属性：浮动、清除浮动、定位position等
- 生成内容属性：content、counter–reset、counter–increment
- 轮廓样式属性：outline–style、outline–width、outline–color、outline
- 页面样式属性：size、page–break–before、page–break–after

7. css中，有哪些方式可以隐藏页面元素？区别？



7.1. 前言

在平常的样式排版中，我们经常遇到将某个模块隐藏的场景

通过 `css` 隐藏元素的方法有很多种，它们看起来实现的效果是一致的

但实际上每一种方法都有一丝轻微的不同，这些不同决定了在一些特定场合下使用哪一种方法

7.2. 实现方式

通过 `css` 实现隐藏元素方法有如下：

- display:none
- visibility:hidden
- opacity:0
- 设置height、width模型属性为0
- position:absolute
- clip-path

7.2.1. display:none

设置元素的 `display` 为 `none` 是最常用的隐藏元素的方法

```css
.hide {
    display:none;
}
```

将元素设置为 `display:none` 后，元素在页面上将彻底消失

元素本身占有的空间就会被其他元素占有，也就是说它会导致浏览器的重排和重绘

消失后，自身绑定的事件不会触发，也不会有过渡效果

特点：元素不可见，不占据空间，无法响应点击事件

7.2.2. visibility:hidden

设置元素的 `visibility` 为 `hidden` 也是一种常用的隐藏元素的方法

从页面上仅仅是隐藏该元素，DOM结果均会存在，只是当时在一个不可见的状态，不会触发重排，但是会触发重绘

```css
.hidden{
    visibility:hidden
}
```

给人的效果是隐藏了，所以他自身的事件不会触发

特点：元素不可见，占据页面空间，无法响应点击事件

7.2.3. opacity:0

`opacity` 属性表示元素的透明度，将元素的透明度设置为0后，在我们用户眼中，元素也是隐藏的

不会引发重排，一般情况下也会引发重绘

> 如果利用 animation 动画，对 opacity 做变化（animation会默认触发GPU加速），则只会触发 GPU 层面的 composite，不会触发重绘

```css
.transparent {
    opacity:0;
}
```

由于其仍然是存在于页面上的，所以他自身的的事件仍然是可以触发的，但被他遮挡的元素是不能触发其事件的

需要注意的是：其子元素不能设置opacity来达到显示的效果

特点：改变元素透明度，元素不可见，占据页面空间，可以响应点击事件

7.2.4. 设置height、width属性为0

将元素的 `margin` ， `border` ， `padding` ， `height` 和 `width` 等影响元素盒模型的属性设置成0，如果元素内有子元素或内容，还应该设置其 `overflow:hidden` 来隐藏其子元素

```css
.hiddenBox {
    margin:0;
    border:0;
    padding:0;
    height:0;
    width:0;
    overflow:hidden;
}
```

特点：元素不可见，不占据页面空间，无法响应点击事件

7.2.5. position:absolute

将元素移出可视区域

```css
.hide {
   position: absolute;
   top: -9999px;
   left: -9999px;
}
```

特点：元素不可见，不影响页面布局

7.2.6. clip-path

通过裁剪的形式

```css
.hide {
  clip-path: polygon(0px 0px,0px 0px,0px 0px,0px 0px);
}
```

特点：元素不可见，占据页面空间，无法响应点击事件

7.2.7. 小结

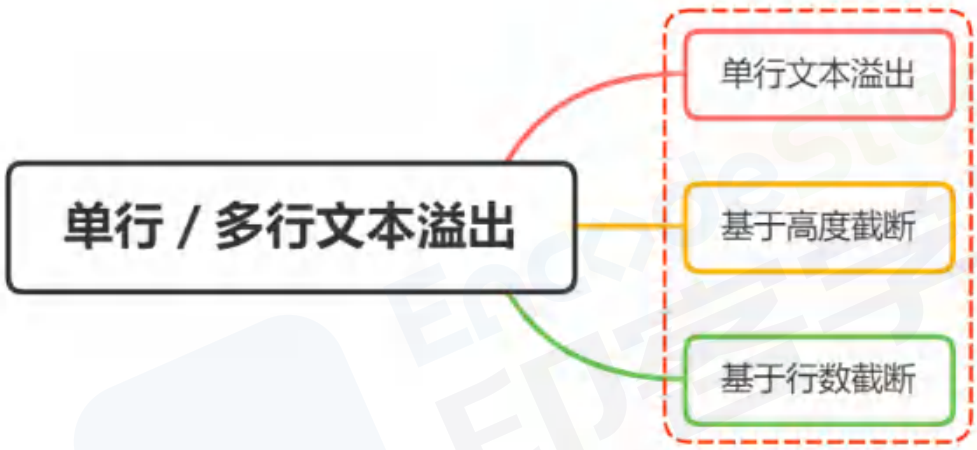
最常用的还是 `display:none` 和 `visibility:hidden` ，其他的方式只能认为是奇招，它们的真正用途并不是用于隐藏元素，所以并不推荐使用它们

7.3. 区别

关于 `display: none` 、 `visibility: hidden` 、 `opacity: 0` 的区别，如下表所示：

	display: none	visibility: hidden	opacity: 0
页面中	不存在	存在	存在
重排	会	不会	不会
重绘	会	会	不一定
自身绑定事件	不触发	不触发	可触发
transition	不支持	支持	支持
子元素可复原	不能	能	不能
被遮挡的元素可触发事件	能	能	不能

8. 如何实现单行 / 多行文本溢出的省略样式?



8.1. 前言

在日常开发展示页面，如果一段文本的数量过长，受制于元素宽度的因素，有可能不能完全显示，为了提高用户的使用体验，这个时候就需要我们把溢出的文本显示成省略号

对于文本的溢出，我们可以分成两种形式：

- 单行文本溢出

- 多行文本溢出

8.2. 实现方式

8.2.1. 单行文本溢出省略

理解也很简单，即文本在一行内显示，超出部分以省略号的形式展现

实现方式也很简单，涉及的 `css` 属性有：

- text-overflow：规定当文本溢出时，显示省略符号来代表被修剪的文本
- white-space：设置文字在一行显示，不能换行
- overflow：文字长度超出限定宽度，则隐藏超出的内容

`overflow` 设为 `hidden` ，普通情况用在块级元素的外层隐藏内部溢出元素，或者配合下面两个属性实现文本溢出省略

`white-space:nowrap` ，作用是设置文本不换行，是 `overflow:hidden` 和 `text-overflow:ellipsis` 生效的基础

`text-overflow` 属性值有如下：

- clip：当对象内文本溢出部分裁切掉
- ellipsis：当对象内文本溢出时显示省略标记（...）

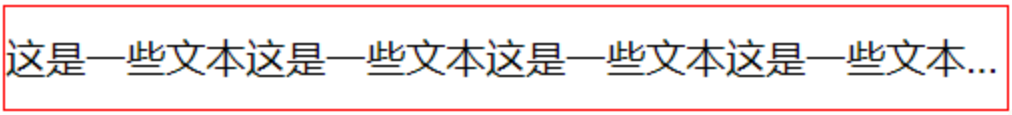
`text-overflow` 只有在设置了 `overflow:hidden` 和 `white-space:nowrap` 才能够生效的

举个例子

HTML 复制代码

```html
<style>
    p{
        overflow: hidden;
        line-height: 40px;
        width:400px;
        height:40px;
        border:1px solid red;
        text-overflow: ellipsis;
        white-space: nowrap;
    }
</style>
<p 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本</p >
```

效果如下：



可以看到，设置单行文本溢出较为简单，并且省略号显示的位置较好

8.2.2. 多行文本溢出省略

多行文本溢出的时候，我们可以分为两种情况：

- 基于高度截断
- 基于行数截断

8.2.2.1. 基于高度截断

8.2.2.2. 伪元素 + 定位

核心的 `css` 代码结构如下：

- position: relative：为伪元素绝对定位
- overflow: hidden：文本溢出限定的宽度就隐藏内容）
- position: absolute：给省略号绝对定位
- line-height: 20px：结合元素高度,高度固定的情况下,设定行高, 控制显示行数
- height: 40px：设定当前元素高度
- ::after {} ： 设置省略号样式

代码如下所示：

HTML 复制代码

```html
<style>
    .demo {
        position: relative;
        line-height: 20px;
        height: 40px;
        overflow: hidden;
    }
    .demo::after {
        content: "...";
        position: absolute;
        bottom: 0;
        right: 0;
        padding: 0 20px 0 10px;
    }
</style>

<body>
    <div class='demo'>这是一段很长的文本</div>
</body>
```

实现原理很好理解，就是通过伪元素绝对定位到行尾并遮住文字，再通过 `overflow: hidden` 隐藏多余文字

这种实现具有以下优点：

- 兼容性好，对各大主流浏览器有好的支持
- 响应式截断，根据不同宽度做出调整

一般文本存在英文的时候，可以设置 `word-break: break-all` 使一个单词能够在换行时进行拆分

8.2.2.3. 基于行数截断

纯 `css` 实现也非常简单，核心的 `css` 代码如下：

- –webkit–line–clamp: 2：用来限制在一个块元素显示的文本的行数，为了实现该效果，它需要组合其他的WebKit属性）
- display: –webkit–box：和1结合使用，将对象作为弹性伸缩盒子模型显示
- –webkit–box–orient: vertical：和1结合使用 ，设置或检索伸缩盒对象的子元素的排列方式
- overflow: hidden：文本溢出限定的宽度就隐藏内容
- text–overflow: ellipsis：多行文本的情况下，用省略号“…”隐藏溢出范围的文本

```HTML
<style>
    p {
        width: 400px;
        border-radius: 1px solid red;
        -webkit-line-clamp: 2;
        display: -webkit-box;
        -webkit-box-orient: vertical;
        overflow: hidden;
        text-overflow: ellipsis;
    }
</style>
<p>
    这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
    这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
</p >
```

可以看到，上述使用了 `webkit` 的 `CSS` 属性扩展，所以兼容浏览器范围是 `PC` 端的 `webkit` 内核的浏览器，由于移动端大多数是使用 `webkit` ，所以移动端常用该形式

需要注意的是，如果文本为一段很长的英文或者数字，则需要添加 `word-wrap: break-word` 属性

还能通过使用 `javascript` 实现配合 `css` ，实现代码如下所示：

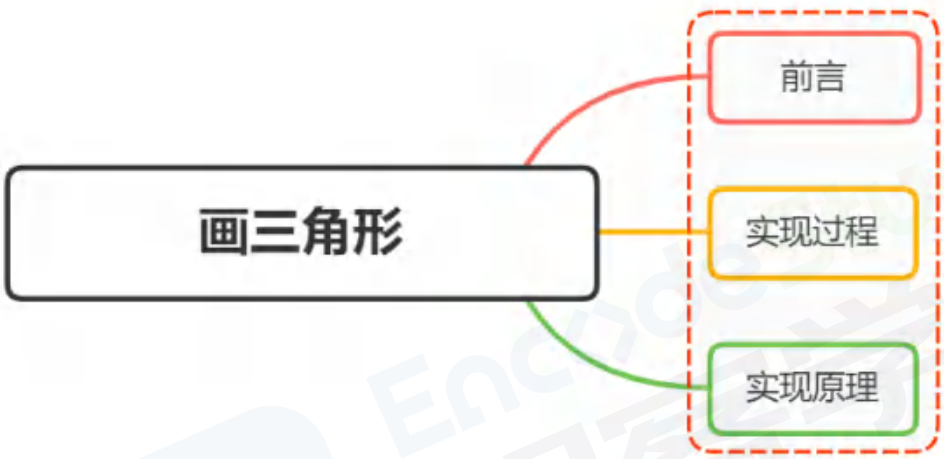
css结构如下：

```CSS
p {
    position: relative;
    width: 400px;
    line-height: 20px;
    overflow: hidden;

}
.p-after:after{
    content: "...";
    position: absolute;
    bottom: 0;
    right: 0;
    padding-left: 40px;
    background: -webkit-linear-gradient(left, transparent, #fff 55%);
    background: -moz-linear-gradient(left, transparent, #fff 55%);
    background: -o-linear-gradient(left, transparent, #fff 55%);
    background: linear-gradient(to right, transparent, #fff 55%);
}
```

javascript代码如下：

```javascript
$(function(){
 //获取文本的行高，并获取文本的高度，假设我们规定的行数是五行，那么对超过行数的部分进行限制高度，并加上省略号
   $('p').each(function(i, obj){
        var lineHeight = parseInt($(this).css("line-height"));
        var height = parseInt($(this).height());
        if((height / lineHeight) >3 ){
            $(this).addClass("p-after")
            $(this).css("height","60px");
        }else{
            $(this).removeClass("p-after");
        }
    });
})
```

9. CSS如何画一个三角形？原理是什么？



9.1. 前言

在前端开发的时候，我们有时候会需要用到一个三角形的形状，比如地址选择或者播放器里面播放按钮

通常情况下，我们会使用图片或者 `svg` 去完成三角形效果图，但如果单纯使用 `css` 如何完成一个三角形呢?

实现过程似乎也并不困难，通过边框就可完成

9.2. 实现过程

在以前也讲过盒子模型，默认情况下是一个矩形，实现也很简单

HTML | 复制代码

```html
<style>
    .border {
        width: 50px;
        height: 50px;
        border: 2px solid;
        border-color: #96ceb4 #ffeead #d9534f #ffad60;
    }
</style>
<div class="border"></div>
```

效果如下图所示：

将 `border` 设置 `50px` ，效果图如下所示：

白色区域则为 `width` 、 `height` ，这时候只需要你将白色区域部分宽高逐渐变小，最终变为0，则变成如下图所示：

这时候就已经能够看到4个不同颜色的三角形，如果需要下方三角形，只需要将上、左、右边框设置为0就可以得到下方的红色三角形

但这种方式，虽然视觉上是实现了三角形，但实际上，隐藏的部分任然占据部分高度，需要将上方的宽度去掉

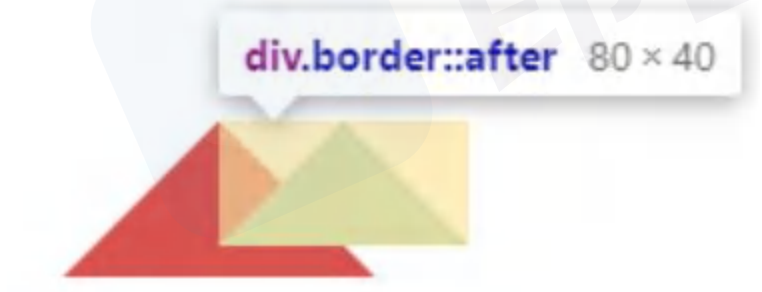
最终实现代码如下：

```css
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
}
```

如果想要实现一个只有边框是空心的三角形，由于这里不能再使用 `border` 属性，所以最直接的方法是利用伪类新建一个小一点的三角形定位上去

```css
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
    position: relative;
}
.border:after{
    content: '';
    border-style:solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 0;
    left: 0;
}
```

效果图如下所示：



伪类元素定位参照对象的内容区域宽高都为0，则内容区域即可以理解成中心一点，所以伪元素相对中心这点定位

将元素定位进行微调以及改变颜色，就能够完成下方效果图：

最终代码如下：

CSS | 复制代码

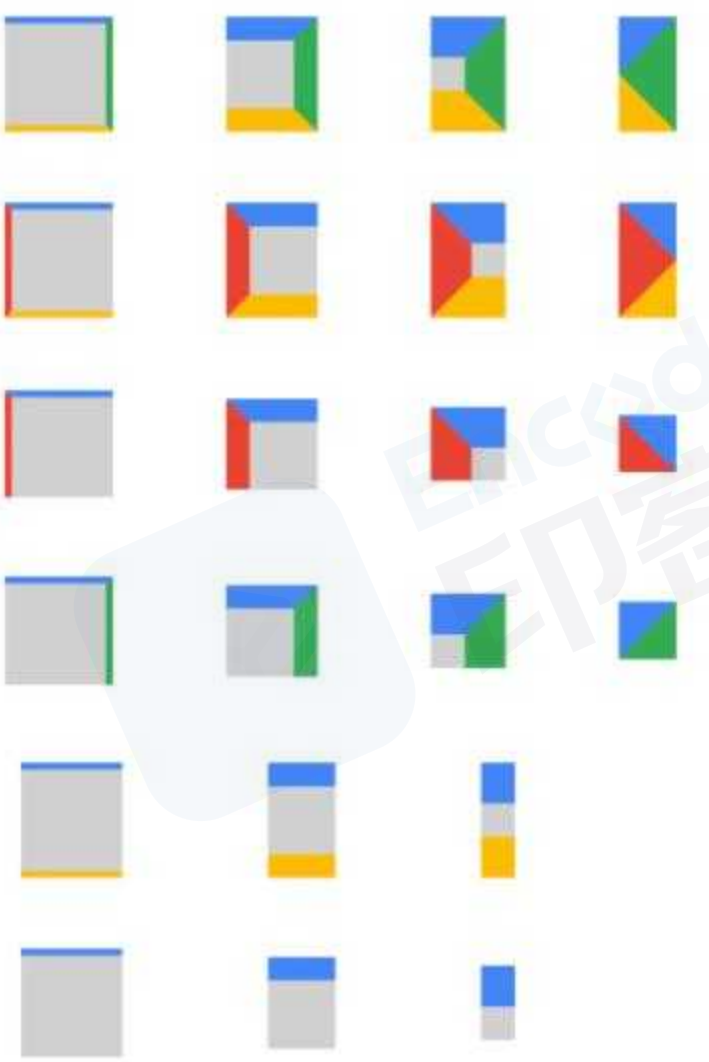
```css
.border:after {
    content: '';
    border-style: solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 6px;
    left: -40px;
}
```

9.3. 原理分析

可以看到，边框是实现三角形的部分，边框实际上并不是一个直线，如果我们将四条边设置不同的颜色，将边框逐渐放大，可以得到每条边框都是一个梯形

当分别取消边框的时候，发现下面几种情况：

- 取消一条边的时候，与这条边相邻的两条边的接触部分会变成直的
- 当仅有邻边时， 两个边会变成对分的三角
- 当保留边没有其他接触时，极限情况所有东西都会消失



通过上图的变化规则，利用旋转、隐藏，以及设置内容宽高等属性，就能够实现其他类型的三角形

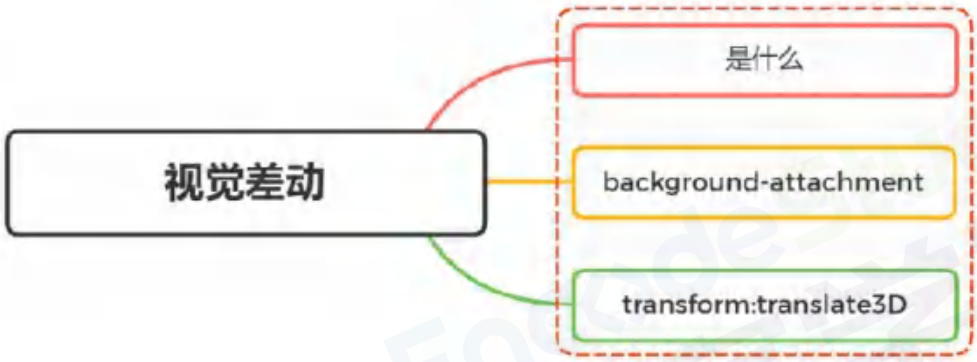
如设置直角三角形，如上图倒数第三行实现过程，我们就能知道整个实现原理

实现代码如下：

CSS | 复制代码

```css
.box {
    /* 内部大小 */
    width: 0px;
    height: 0px;
    /* 边框大小 只设置两条边*/
    border-top: #4285f4 solid;
    border-right: transparent solid;
    border-width: 85px;
    /* 其他设置 */
    margin: 50px;
}
```

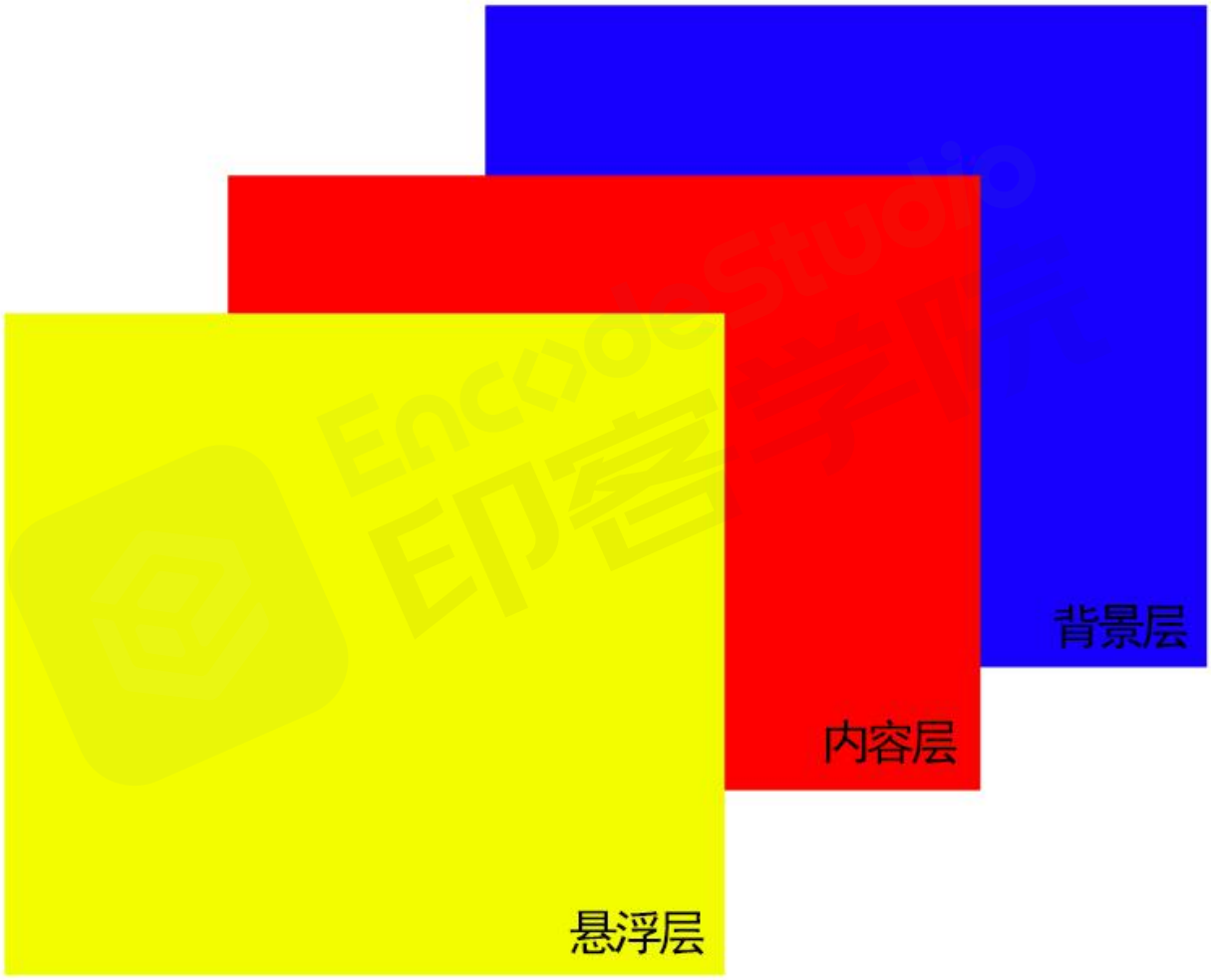
10. 如何使用css完成视差滚动效果？



10.1. 是什么

视差滚动（Parallax Scrolling）是指多层背景以不同的速度移动，形成立体的运动效果，带来非常出色的视觉体验

我们可以把网页解刨成：背景层、内容层、悬浮层



当滚动鼠标滑轮的时候，各个图层以不同的速度移动，形成视觉差的效果

10.2. 实现方式

使用 `css` 形式实现视觉差滚动效果的方式有：

- background–attachment
- transform:translate3D

10.2.1. background–attachment

作用是设置背景图像是否固定或者随着页面的其余部分滚动

值分别有如下：

- scroll：默认值，背景图像会随着页面其余部分的滚动而移动
- fixed：当页面的其余部分滚动时，背景图像不会移动
- inherit：继承父元素background–attachment属性的值

完成滚动视觉差就需要将 `background-attachment` 属性设置为 `fixed` ，让背景相对于视口固定。及时一个元素有滚动机制，背景也不会随着元素的内容而滚动

也就是说，背景一开始就已经被固定在初始的位置

核心的 `css` 代码如下：

```css
section {
    height: 100vh;
}

.g-img {
    background-image: url(...);
    background-attachment: fixed;
    background-size: cover;
    background-position: center center;
}
```

整体例子如下：

HTML | 复制代码

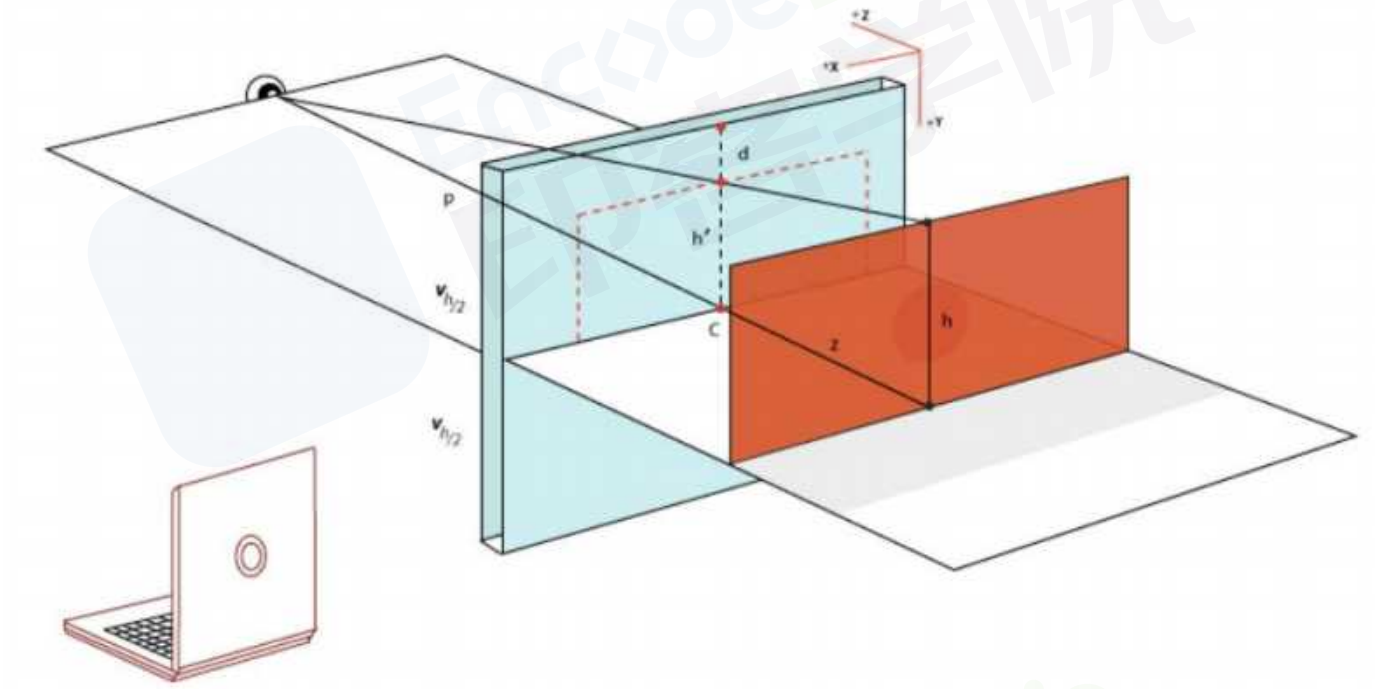
```
<style>
div {
            height: 100vh;
            background: rgba(0, 0, 0, .7);
            color: #fff;
            line-height: 100vh;
            text-align: center;
            font-size: 20vh;
        }

        .a-img1 {
            background-image: url(https://images.pexels.com/photos/1097491/pexels-photo-1097491.jpeg);
            background-attachment: fixed;
            background-size: cover;
            background-position: center center;
        }

        .a-img2 {
            background-image: url(https://images.pexels.com/photos/2437299/pexels-photo-2437299.jpeg);
            background-attachment: fixed;
            background-size: cover;
            background-position: center center;
        }

        .a-img3 {
            background-image: url(https://images.pexels.com/photos/1005417/pexels-photo-1005417.jpeg);
            background-attachment: fixed;
            background-size: cover;
            background-position: center center;
        }
</style>
 <div class="a-text">1</div>
    <div class="a-img1">2</div>
    <div class="a-text">3</div>
    <div class="a-img2">4</div>
    <div class="a-text">5</div>
    <div class="a-img3">6</div>
    <div class="a-text">7</div>
```

10.2.2. transform:translate3D

同样，让我们先来看一下两个概念 `transform` 和 `perspective`：

- transform: css3 属性，可以对元素进行变换(2d/3d)，包括平移 translate,旋转 rotate,缩放 scale,等等
- perspective: css3 属性，当元素涉及 3d 变换时，perspective 可以定义我们眼睛看到的 3d 立体效果，即空间感

`3D` 视角示意图如下所示：



举个例子：

HTML | 复制代码

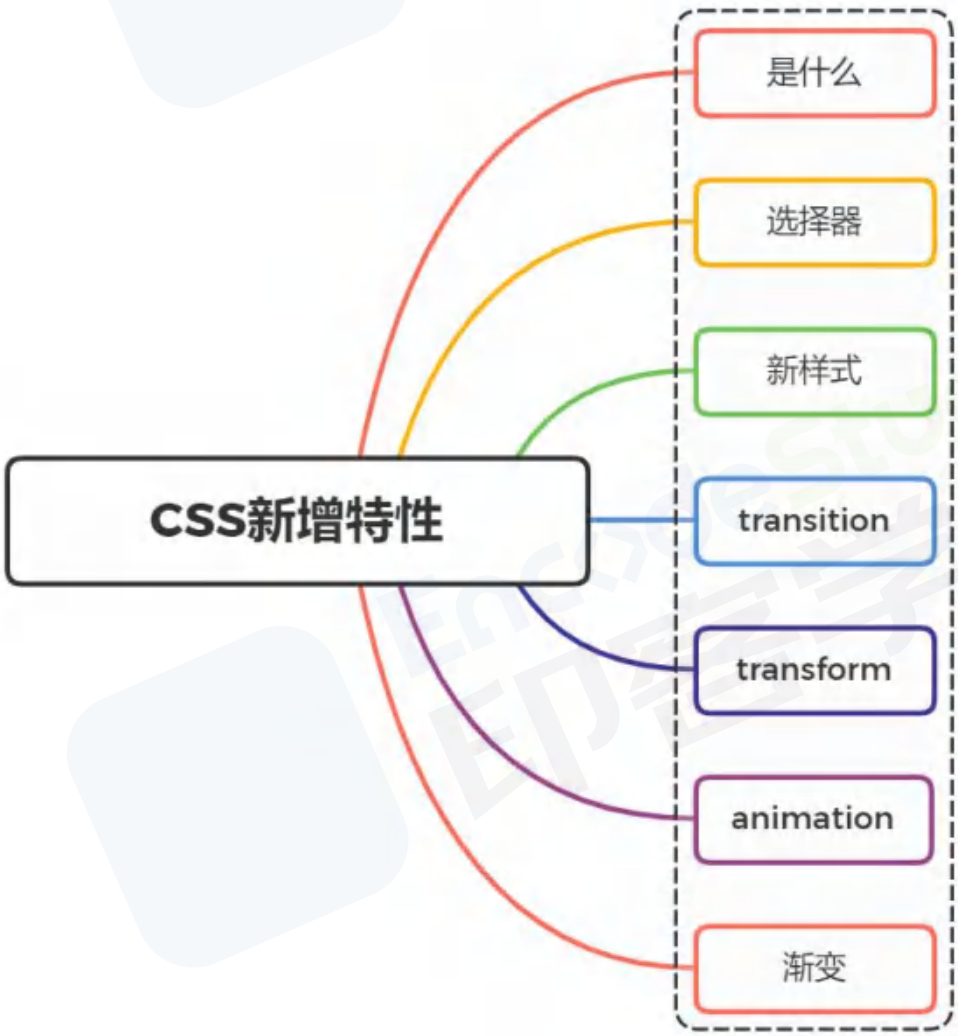
```html
<style>
    html {
        overflow: hidden;
        height: 100%
    }

    body {
        /* 视差元素的父级需要3D视角 */
        perspective: 1px;
        transform-style: preserve-3d;
        height: 100%;
        overflow-y: scroll;
        overflow-x: hidden;
    }
    #app{
        width: 100vw;
        height:200vh;
        background:skyblue;
        padding-top:100px;
    }
    .one{
        width:500px;
        height:200px;
        background:#409eff;
        transform: translateZ(0px);
        margin-bottom: 50px;
    }
    .two{
        width:500px;
        height:200px;
        background:#67c23a;
        transform: translateZ(-1px);
        margin-bottom: 150px;
    }
    .three{
        width:500px;
        height:200px;
        background:#e6a23c;
        transform: translateZ(-2px);
        margin-bottom: 150px;
    }
</style>
<div id="app">
    <div class="one">one</div>
    <div class="two">two</div>
```

```
        <div class="three">three</div>
    </div>
```

而这种方式实现视觉差动的原理如下：

- 容器设置上 transform-style: preserve-3d 和 perspective: xpx，那么处于这个容器的子元素就将位于3D空间中，
- 子元素设置不同的 transform: translateZ()，这个时候，不同元素在 3D Z轴方向距离屏幕（我们的眼睛）的距离也就不一样
- 滚动滚动条，由于子元素设置了不同的 transform: translateZ()，那么他们滚动的上下距离 translateY 相对屏幕（我们的眼睛），也是不一样的，这就达到了滚动视差的效果

11. CSS3新增了哪些新特性?



11.1. 是什么

`css`，即层叠样式表（Cascading Style Sheets）的简称，是一种标记语言，由浏览器解释执行用来使页面变得更美观

`css3` 是 `css` 的最新标准，是向后兼容的，`CSS1/2` 的特性在 `CSS3` 里都是可以使用的

而 `CSS3` 也增加了很多新特性，为开发带来了更佳的开发体验

11.2. 选择器

`css3` 中新增了一些选择器，主要为如下图所示：

选择器	例子	例子描述
element1~element2	p~ul	选择前面有 <p> 元素的每个 元素。
[attribute^=value]	a[src^="https"]	选择其 src 属性值以 "https" 开头的每个 <a> 元素。
[attribute$=value]	a[src$=".pdf"]	选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。
[attribute*=value]	a[src*="abc"]	选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。
:first-of-type	p:first-of-type	选择属于其父元素的首个 <p> 元素的每个 <p> 元素。
:last-of-type	p:last-of-type	选择属于其父元素的最后 <p> 元素的每个 <p> 元素。
:only-of-type	p:only-of-type	选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。
:only-child	p:only-child	选择属于其父元素的唯一子元素的每个 <p> 元素。
:nth-child(n)	p:nth-child(2)	选择属于其父元素的第二个子元素的每个 <p> 元素。
:nth-last-child(n)	p:nth-last-child(2)	同上，从最后一个子元素开始计数。
:nth-of-type(n)	p:nth-of-type(2)	选择属于其父元素第二个 <p> 元素的每个 <p> 元素。
:nth-last-of-type(n)	p:nth-last-of-type(2)	同上，但是从最后一个子元素开始计数。
:last-child	p:last-child	选择属于其父元素最后一个子元素每个 <p> 元素。

11.3. 新样式

11.3.1. 边框

`css3` 新增了三个边框属性，分别是：

- border-radius：创建圆角边框
- box-shadow：为元素添加阴影

- border-image：使用图片来绘制边框

11.3.1.1. box-shadow

设置元素阴影，设置属性如下：

- 水平阴影
- 垂直阴影
- 模糊距离(虚实)
- 阴影尺寸(影子大小)
- 阴影颜色
- 内/外阴影

其中水平阴影和垂直阴影是必须设置的

11.3.2. 背景

新增了几个关于背景的属性，分别是 `background-clip` 、 `background-origin` 、 `background-size` 和 `background-break`

11.3.2.1. background-clip

用于确定背景画区，有以下几种可能的属性：

- background-clip: border-box; 背景从border开始显示
- background-clip: padding-box; 背景从padding开始显示
- background-clip: content-box; 背景显content区域开始显示
- background-clip: no-clip; 默认属性，等同于border-box

通常情况，背景都是覆盖整个元素的，利用这个属性可以设定背景颜色或图片的覆盖范围

11.3.2.2. background-origin

当我们设置背景图片时，图片是会以左上角对齐，但是是以 `border` 的左上角对齐还是以 `padding` 的左上角或者 `content` 的左上角对齐？ `border-origin` 正是用来设置这个的

- background-origin: border-box; 从border开始计算background-position
- background-origin: padding-box; 从padding开始计算background-position
- background-origin: content-box; 从content开始计算background-position

默认情况是 `padding-box` ，即以 `padding` 的左上角为原点

11.3.2.3. background-size

background-size属性常用来调整背景图片的大小，主要用于设定图片本身。有以下可能的属性：

- background-size: contain; 缩小图片以适合元素（维持像素长宽比）
- background-size: cover; 扩展元素以填补元素（维持像素长宽比）
- background-size: 100px 100px; 缩小图片至指定的大小
- background-size: 50% 100%; 缩小图片至指定的大小，百分比是相对包 含元素的尺寸

11.3.3. background-break

元素可以被分成几个独立的盒子（如使内联元素span跨越多行），`background-break` 属性用来控制背景怎样在这些不同的盒子中显示

- background-break: continuous; 默认值。忽略盒之间的距离（也就是像元素没有分成多个盒子，依然是一个整体一样）
- background-break: bounding-box; 把盒之间的距离计算在内；
- background-break: each-box; 为每个盒子单独重绘背景

11.3.4. 文字

11.3.5. word-wrap

语法：`word-wrap: normal|break-word`

- normal：使用浏览器默认的换行
- break-all：允许在单词内换行

11.3.6. text-overflow

`text-overflow` 设置或检索当当前行超过指定容器的边界时如何显示，属性有两个值选择：

- clip：修剪文本
- ellipsis：显示省略符号来代表被修剪的文本

11.3.7. text-shadow

`text-shadow` 可向文本应用阴影。能够规定水平阴影、垂直阴影、模糊距离，以及阴影的颜色

11.3.8. text-decoration

CSS3里面开始支持对文字的更深层次的渲染，具体有三个属性可供设置：

- text-fill-color: 设置文字内部填充颜色
- text-stroke-color: 设置文字边界填充颜色
- text-stroke-width: 设置文字边界宽度

11.3.9. 颜色

`css3` 新增了新的颜色表示方式 `rgba` 与 `hsla`

- rgba分为两部分，rgb为颜色值，a为透明度
- hala分为四部分，h为色相，s为饱和度，l为亮度，a为透明度

11.4. transition 过渡

`transition` 属性可以被指定为一个或多个 `CSS` 属性的过渡效果，多个属性之间用逗号进行分隔，必须规定两项内容：

- 过度效果
- 持续时间

语法如下：

CSS | 复制代码

```
transition:  CSS属性，花费时间，效果曲线(默认ease)，延迟时间(默认0)
```

上面为简写模式，也可以分开写各个属性

CSS | 复制代码

```
transition-property: width;
transition-duration: 1s;
transition-timing-function: linear;
transition-delay: 2s;
```

11.4.1. transform 转换

`transform` 属性允许你旋转，缩放，倾斜或平移给定元素

`transform-origin`：转换元素的位置（围绕那个点进行转换），默认值为`(x,y,z):(50%,50%,0)`

使用方式：

- transform: translate(120px, 50%)：位移
- transform: scale(2, 0.5)：缩放
- transform: rotate(0.5turn)：旋转
- transform: skew(30deg, 20deg)：倾斜

11.4.2. animation 动画

动画这个平常用的也很多，主要是做一个预设的动画。和一些页面交互的动画效果，结果和过渡应该一样，让页面不会那么生硬

animation也有很多的属性

- animation-name：动画名称
- animation-duration：动画持续时间
- animation-timing-function：动画时间函数
- animation-delay：动画延迟时间
- animation-iteration-count：动画执行次数，可以设置为一个整数，也可以设置为infinite，意思是无限循环
- animation-direction：动画执行方向
- animation-paly-state：动画播放状态
- animation-fill-mode：动画填充模式

11.5. 渐变

颜色渐变是指在两个颜色之间平稳的过渡，`css3`渐变包括

- linear-gradient：线性渐变

```
background-image: linear-gradient(direction, color-stop1, color-stop2, ...);
```

- radial-gradient：径向渐变

```
linear-gradient(0deg, red, green);
```

11.6. 其他

关于 `css3` 其他的新特性还包括 `flex` 弹性布局、 `Grid` 栅格布局，这两个布局在以前就已经讲过，这里就不再展示

除此之外，还包括多列布局、媒体查询、混合模式等等.....

12. css3动画有哪些?

12.1. 是什么

CSS动画（CSS Animations）是为层叠样式表建议的允许可扩展标记语言（XML）元素使用CSS的动画的模块

即指元素从一种样式逐渐过渡为另一种样式的过程

常见的动画效果有很多，如平移、旋转、缩放等等，复杂动画则是多个简单动画的组合

`css` 实现动画的方式，有如下几种：

- transition 实现渐变动画
- transform 转变动画
- animation 实现自定义动画

12.2. 实现方式

12.2.1. transition 实现渐变动画

`transition` 的属性如下：

- property:填写需要变化的css属性
- duration:完成过渡效果需要的时间单位(s或者ms)
- timing-function:完成效果的速度曲线
- delay: 动画效果的延迟触发时间

其中 `timing-function` 的值有如下：

值	描述
linear	匀速（等于 cubic-bezier(0,0,1,1)）
ease	从慢到快再到慢（cubic-bezier(0.25,0.1,0.25,1)）
ease-in	慢慢变快（等于 cubic-bezier(0.42,0,1,1)）
ease-out	慢慢变慢（等于 cubic-bezier(0,0,0.58,1)）
ease-in-out	先变快再到慢（等于 cubic-bezier(0.42,0,0.58,1)），渐显渐隐效果
cubic-bezier(*n*,*n*,*n*,*n*)	在 cubic-bezier 函数中定义自己的值。可能的值是 0 至 1 之间的数值

注意：并不是所有的属性都能使用过渡的，如 `display:none<->display:block`

举个例子，实现鼠标移动上去发生变化动画效果

```html
<style>
        .base {
            width: 100px;
            height: 100px;
            display: inline-block;
            background-color: #0EA9FF;
            border-width: 5px;
            border-style: solid;
            border-color: #5daf34;
            transition-property: width, height, background-color, border-width;
            transition-duration: 2s;
            transition-timing-function: ease-in;
            transition-delay: 500ms;
        }

        /*简写*/
        /*transition: all 2s ease-in 500ms;*/
        .base:hover {
            width: 200px;
            height: 200px;
            background-color: #5daf34;
            border-width: 10px;
            border-color: #3a8ee6;
        }
</style>
<div class="base"></div>
```

12.2.2. transform 转变动画

包含四个常用的功能：

- translate：位移
- scale：缩放
- rotate：旋转
- skew：倾斜

一般配合 `transition` 过度使用

注意的是， `transform` 不支持 `inline` 元素，使用前把它变成 `block`

举个例子

HTML 复制代码

```html
<style>
    .base {
        width: 100px;
        height: 100px;
        display: inline-block;
        background-color: #0EA9FF;
        border-width: 5px;
        border-style: solid;
        border-color: #5daf34;
        transition-property: width, height, background-color, border-width;
        transition-duration: 2s;
        transition-timing-function: ease-in;
        transition-delay: 500ms;
    }
    .base2 {
        transform: none;
        transition-property: transform;
        transition-delay: 5ms;
    }

    .base2:hover {
        transform: scale(0.8, 1.5) rotate(35deg) skew(5deg) translate(15px, 25px);
    }
</style>
 <div class="base base2"></div>
```

可以看到盒子发生了旋转，倾斜，平移，放大

12.2.3. animation 实现自定义动画

`animation` 是由 8 个属性的简写，分别如下：

属性	描述	属性值
animation-duration	指定动画完成一个周期所需要时间，单位秒（s）或毫秒（ms），默认是 0	
animation-timing-function	指定动画计时函数，即动画的速度曲线，默认是 "ease"	linear、ease、ease-in、ease-out、ease-in-out

animation–delay	指定动画延迟时间，即动画何时开始，默认是 0	
animation–iteration–count	指定动画播放的次数，默认是 1	
animation–direction 指定动画播放的方向	默认是 normal	normal、reverse、alternate、alternate–reverse
animation–fill–mode	指定动画填充模式。默认是 none	forwards、backwards、both
animation–play–state	指定动画播放状态，正在运行或暂停。默认是 running	running、pauser
animation–name	指定 @keyframes 动画的名称	

`CSS` 动画只需要定义一些关键的帧，而其余的帧，浏览器会根据计时函数插值计算出来，

通过 `@keyframes` 来定义关键帧

因此，如果我们想要让元素旋转一圈，只需要定义开始和结束两帧即可：

CSS 复制代码

```css
@keyframes rotate{
    from{
        transform: rotate(0deg);
    }
    to{
        transform: rotate(360deg);
    }
}
```

`from` 表示最开始的那一帧， `to` 表示结束时的那一帧

也可以使用百分比刻画生命周期

```css
@keyframes rotate{
    0%{
        transform: rotate(0deg);
    }
    50%{
        transform: rotate(180deg);
    }
    100%{
        transform: rotate(360deg);
    }
}
```

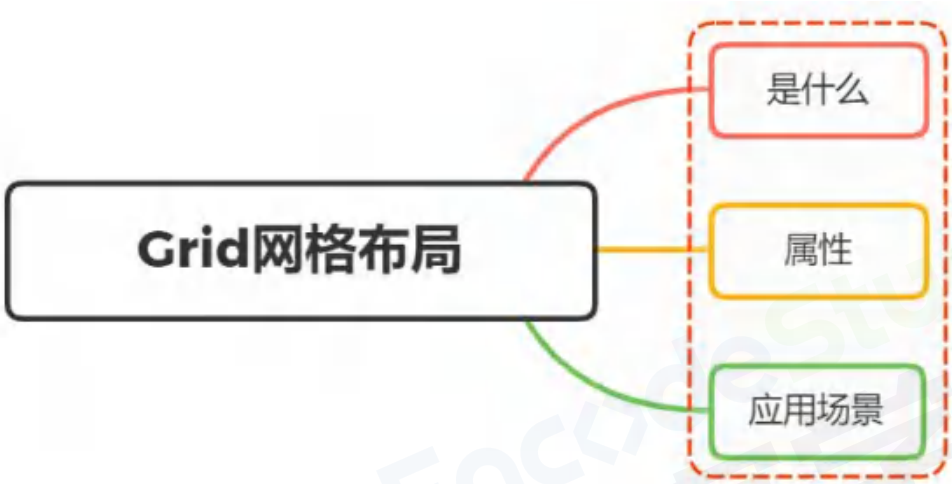
定义好了关键帧后，下来就可以直接用它了：

```css
animation: rotate 2s;
```

12.3. 总结

属性	含义
transition（过度）	用于设置元素的样式过度，和animation有着类似的效果，但细节上有很大的不同
transform（变形）	用于元素进行旋转、缩放、移动或倾斜，和设置样式的动画并没有什么关系，就相当于color一样用来设置元素的“外表”
translate（移动）	只是transform的一个属性值，即移动
animation（动画）	用于设置动画属性，他是一个简写的属性，包含6个属性

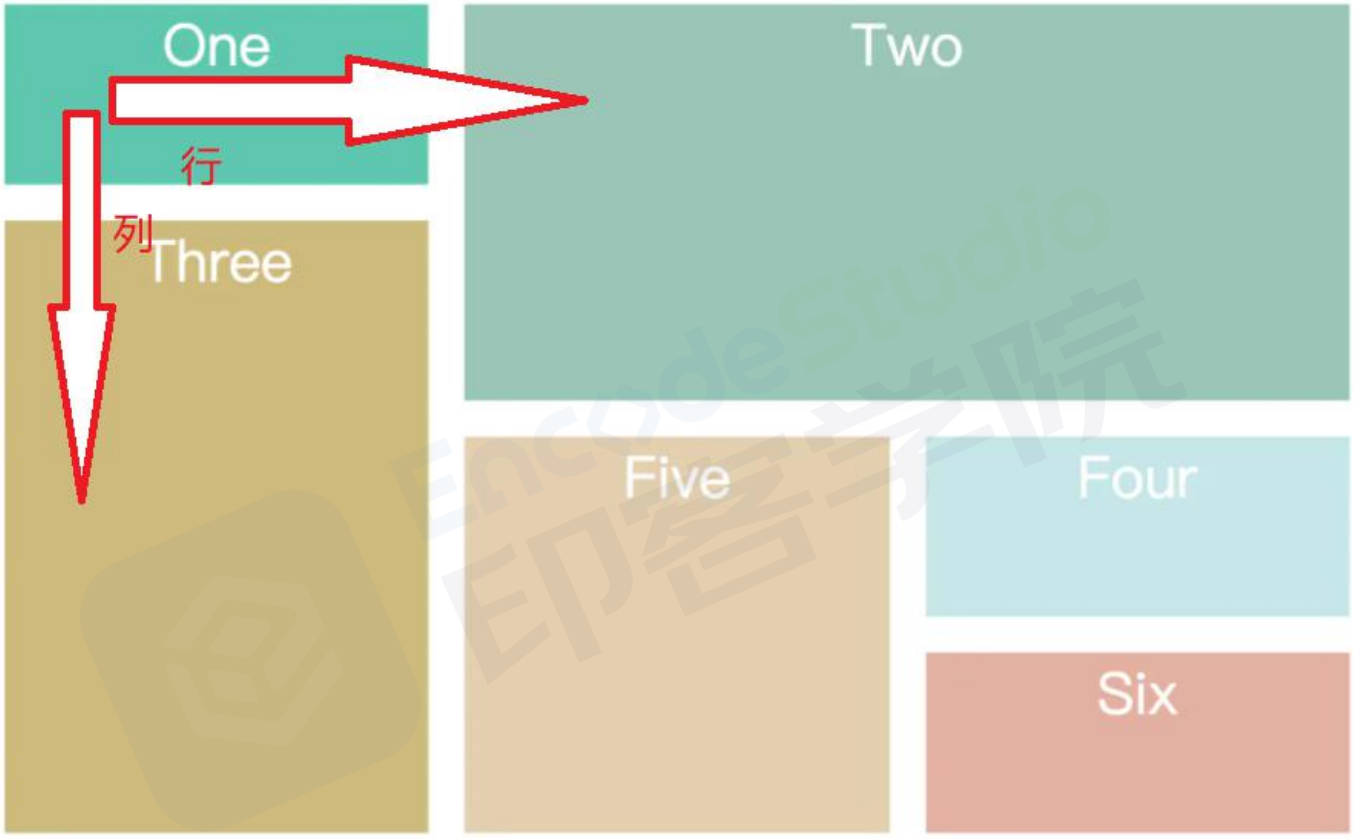
13. 介绍一下grid网格布局



13.1. 是什么

`Grid` 布局即网格布局，是一个二维的布局方式，由纵横相交的两组网格线形成的框架性布局结构，能够同时处理行与列

擅长将一个页面划分为几个主要区域，以及定义这些区域的大小、位置、层次等关系



这与之前讲到的 `flex` 一维布局不相同

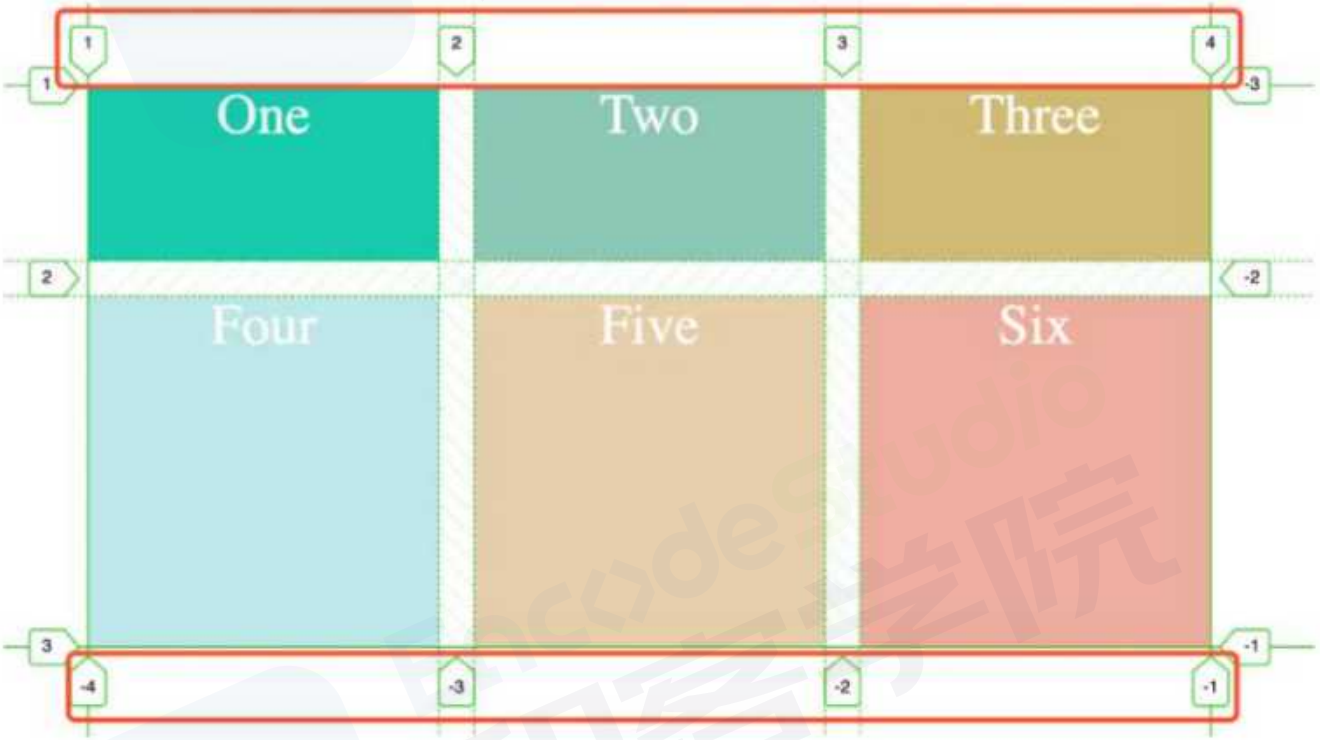
设置 `display:grid/inline-grid` 的元素就是网格布局容器，这样就能出发浏览器渲染引擎的网格布局算法

```JavaScript
<div class="container">
    <div class="item item-1">
        <p class="sub-item"></p >
 </div>
    <div class="item item-2"></div>
    <div class="item item-3"></div>
</div>
```

上述代码实例中，`.container` 元素就是网格布局容器，`.item` 元素就是网格的项目，由于网格元素只能是容器的顶层子元素，所以 `p` 元素并不是网格元素

这里提一下，网格线概念，有助于下面对 `grid-column` 系列属性的理解

网格线，即划分网格的线，如下图所示：



上图是一个 2 x 3 的网格，共有3根水平网格线和4根垂直网格线

13.2. 属性

同样，`Grid` 布局属性可以分为两大类：

- 容器属性，
- 项目属性

关于容器属性有如下：

13.2.1. display 属性

文章开头讲到，在元素上设置 `display: grid` 或 `display: inline-grid` 来创建一个网格容器

- display：grid 则该容器是一个块级元素
- display: inline-grid 则容器元素为行内元素

13.2.2. grid-template-columns 属性，grid-template-rows 属性

`grid-template-columns` 属性设置列宽，`grid-template-rows` 属性设置行高

CSS | 复制代码

```css
.wrapper {
  display: grid;
  /*  声明了三列，宽度分别为 200px 200px 200px */
  grid-template-columns: 200px 200px 200px;
  grid-gap: 5px;
  /*  声明了两行，行高分别为 50px 50px  */
  grid-template-rows: 50px 50px;
}
```

以上表示固定列宽为 200px 200px 200px，行高为 50px 50px

上述代码可以看到重复写单元格宽高，通过使用 `repeat()` 函数，可以简写重复的值

- 第一个参数是重复的次数
- 第二个参数是重复的值

所以上述代码可以简写成

CSS | 复制代码

```css
.wrapper {
  display: grid;
  grid-template-columns: repeat(3,200px);
  grid-gap: 5px;
  grid-template-rows:repeat(2,50px);
}
```

除了上述的 `repeact` 关键字，还有：

- auto-fill：示自动填充，让一行（或者一列）中尽可能的容纳更多的单元格

> `grid-template-columns: repeat(auto-fill, 200px)` 表示列宽是 200 px，但列的数量是不固定的，只要浏览器能够容纳得下，就可以放置元素

- fr：片段，为了方便表示比例关系

> `grid-template-columns: 200px 1fr 2fr` 表示第一个列宽设置为 200px，后面剩余的宽度分为两部分，宽度分别为剩余宽度的 1/3 和 2/3

- minmax：产生一个长度范围，表示长度就在这个范围之中都可以应用到网格项目中。第一个参数就是最小值，第二个参数就是最大值

> `minmax(100px, 1fr)` 表示列宽不小于 `100px`，不大于 `1fr`

- auto：由浏览器自己决定长度

> `grid-template-columns: 100px auto 100px` 表示第一第三列为 100px，中间由浏览器决定长度

13.2.3. grid-row-gap 属性， grid-column-gap 属性， grid-gap 属性

`grid-row-gap` 属性、`grid-column-gap` 属性分别设置行间距和列间距。`grid-gap` 属性是两者的简写形式

`grid-row-gap: 10px` 表示行间距是 10px

`grid-column-gap: 20px` 表示列间距是 20px

`grid-gap: 10px 20px` 等同上述两个属性

13.2.4. grid-template-areas 属性

用于定义区域，一个区域由一个或者多个单元格组成

CSS | 复制代码

```css
.container {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 100px 100px 100px;
  grid-template-areas: 'a b c'
                       'd e f'
                       'g h i';
}
```

上面代码先划分出9个单元格，然后将其定名为 `a` 到 `i` 的九个区域，分别对应这九个单元格。

多个单元格合并成一个区域的写法如下

```css
grid-template-areas: 'a a a'
                     'b b b'
                     'c c c';
```

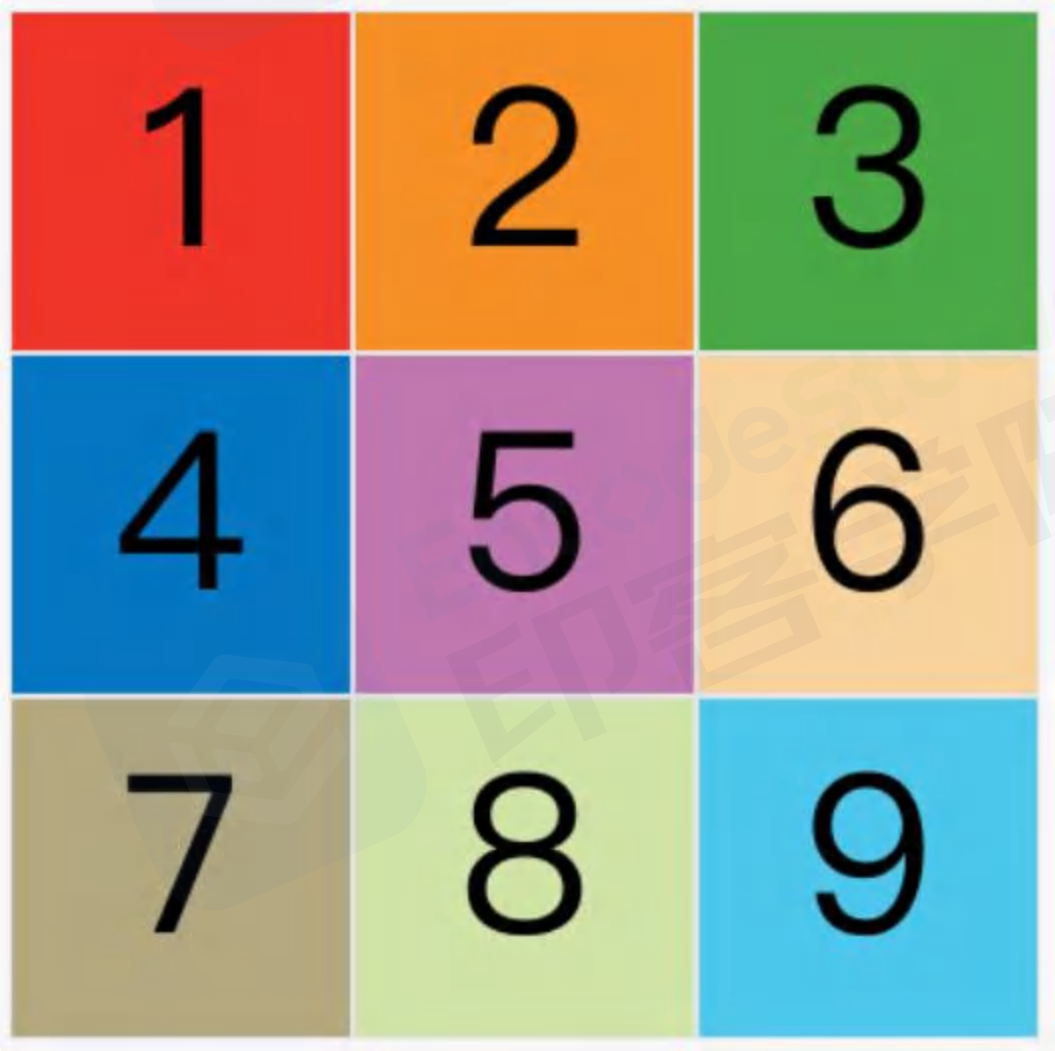
上面代码将9个单元格分成 `a` 、 `b` 、 `c` 三个区域

如果某些区域不需要利用，则使用"点"（ `.` ）表示

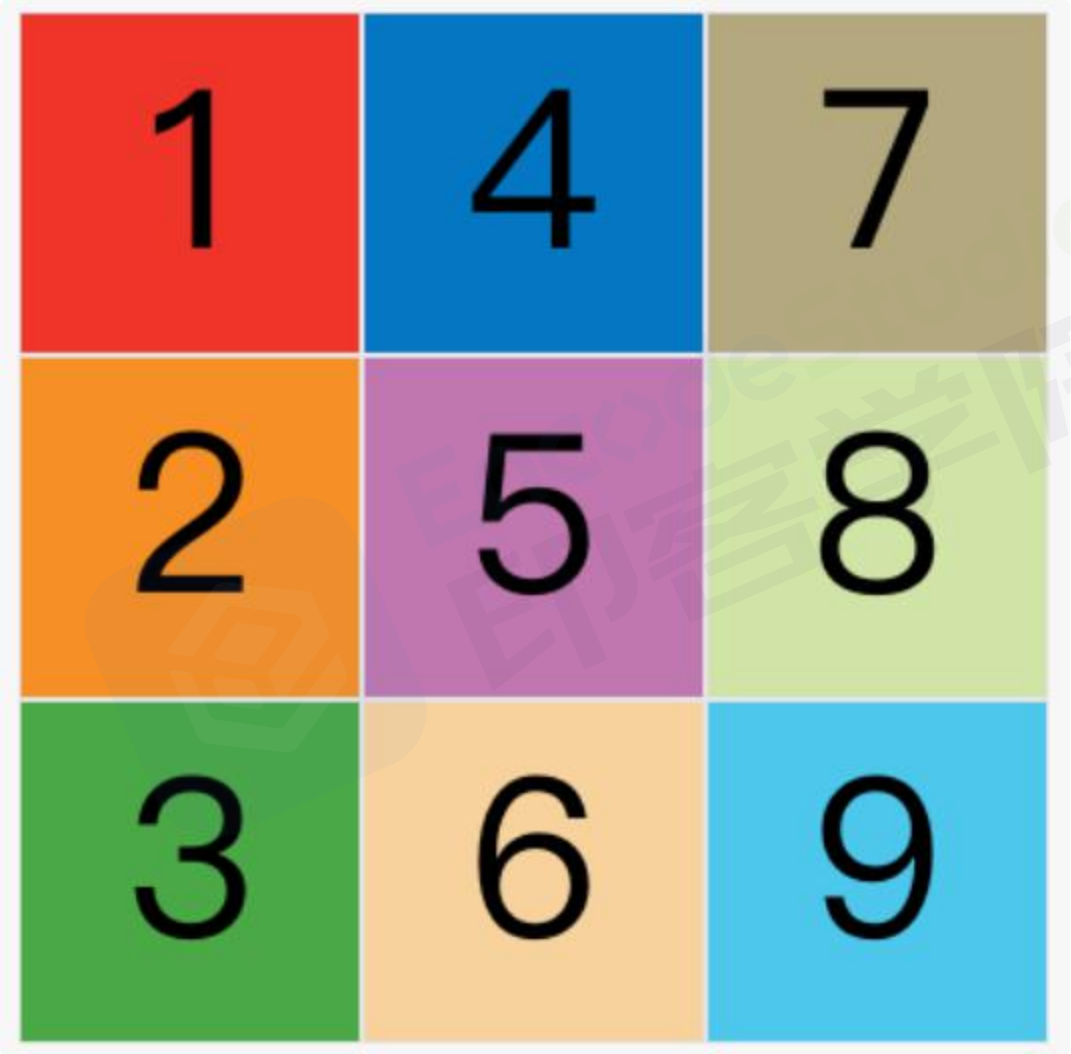
13.2.5. grid-auto-flow 属性

划分网格以后，容器的子元素会按照顺序，自动放置在每一个网格。

顺序就是由 `grid-auto-flow` 决定，默认为行，代表"先行后列"，即先填满第一行，再开始放入第二行



当修改成 `column` 后，放置变为如下：



13.2.6. justify-items 属性， align-items 属性， place-items 属性

`justify-items` 属性设置单元格内容的水平位置（左中右），`align-items` 属性设置单元格的垂直位置（上中下）

两者属性的值完成相同

CSS | 复制代码

```css
.container {
  justify-items: start | end | center | stretch;
  align-items: start | end | center | stretch;
}
```

属性对应如下：

- start：对齐单元格的起始边缘
- end：对齐单元格的结束边缘

- center：单元格内部居中
- stretch：拉伸，占满单元格的整个宽度（默认值）

`place-items` 属性是 `align-items` 属性和 `justify-items` 属性的合并简写形式

13.2.7. justify-content 属性， align-content 属性， place-content 属性

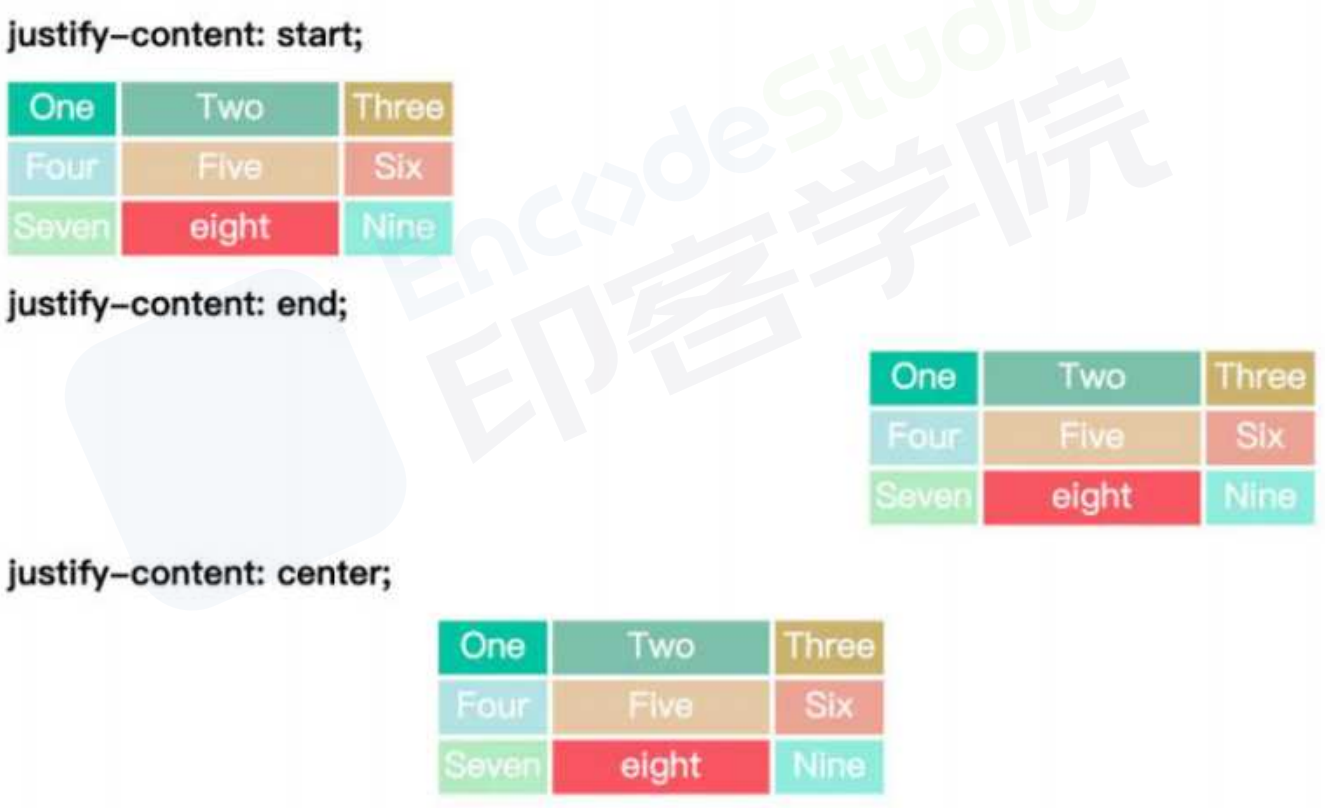
`justify-content` 属性是整个内容区域在容器里面的水平位置（左中右），`align-content` 属性是整个内容区域的垂直位置（上中下）

CSS | 复制代码

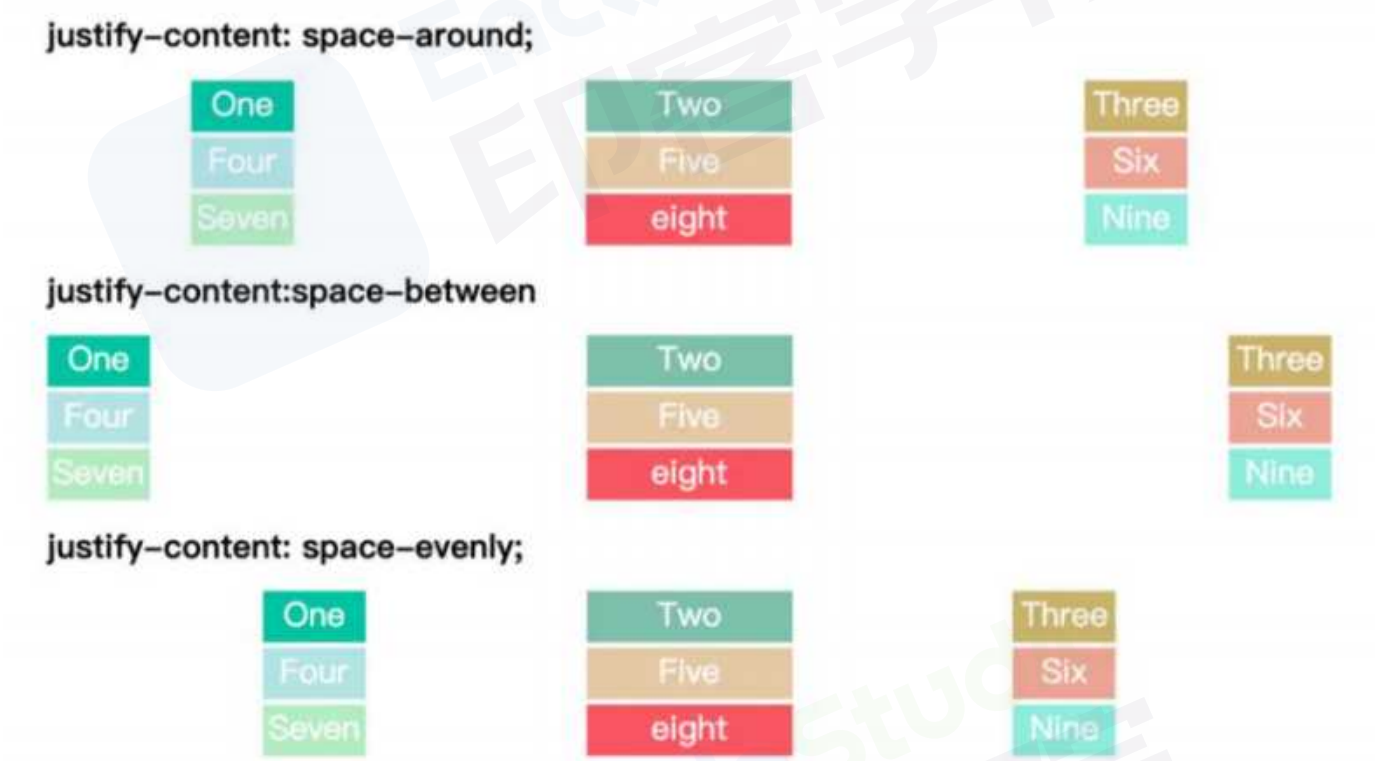
```css
.container {
  justify-content: start | end | center | stretch | space-around | space-between | space-evenly;
  align-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```

两个属性的写法完全相同，都可以取下面这些值：

- start – 对齐容器的起始边框
- end – 对齐容器的结束边框
- center – 容器内部居中



- space-around – 每个项目两侧的间隔相等。所以，项目之间的间隔比项目与容器边框的间隔大一倍
- space-between – 项目与项目的间隔相等，项目与容器边框之间没有间隔
- space-evenly – 项目与项目的间隔相等，项目与容器边框之间也是同样长度的间隔
- stretch – 项目大小没有指定时，拉伸占据整个网格容器



13.2.8. grid-auto-columns 属性和 grid-auto-rows 属性

有时候，一些项目的指定位置，在现有网格的外部，就会产生显示网格和隐式网格

比如网格只有3列，但是某一个项目指定在第5行。这时，浏览器会自动生成多余的网格，以便放置项目。超出的部分就是隐式网格

而 `grid-auto-rows` 与 `grid-auto-columns` 就是专门用于指定隐式网格的宽高

关于项目属性，有如下：

13.2.9. grid-column-start 属性、grid-column-end 属性、grid-row-start 属性以及grid-row-end 属性

指定网格项目所在的四个边框，分别定位在哪根网格线，从而指定项目的位置

- grid-column-start 属性：左边框所在的垂直网格线
- grid-column-end 属性：右边框所在的垂直网格线
- grid-row-start 属性：上边框所在的水平网格线
- grid-row-end 属性：下边框所在的水平网格线

举个例子：

HTML 复制代码

```html
<style>
    #container{
        display: grid;
        grid-template-columns: 100px 100px 100px;
        grid-template-rows: 100px 100px 100px;
    }
    .item-1 {
        grid-column-start: 2;
        grid-column-end: 4;
    }
</style>

<div id="container">
    <div class="item item-1">1</div>
    <div class="item item-2">2</div>
    <div class="item item-3">3</div>
</div>
```

通过设置 `grid-column` 属性，指定1号项目的左边框是第二根垂直网格线，右边框是第四根垂直网格线

13.2.10. grid-area 属性

`grid-area` 属性指定项目放在哪一个区域

```css
.item-1 {
  grid-area: e;
}
```

意思为将1号项目位于 `e` 区域

与上述讲到的 `grid-template-areas` 搭配使用

13.2.11. justify-self 属性、align-self 属性以及 place-self 属性

`justify-self` 属性设置单元格内容的水平位置（左中右），跟 `justify-items` 属性的用法完全一致，但只作用于单个项目。

`align-self` 属性设置单元格内容的垂直位置（上中下），跟 `align-items` 属性的用法完全一致，也是只作用于单个项目

```css
.item {
  justify-self: start | end | center | stretch;
  align-self: start | end | center | stretch;
}
```

这两个属性都可以取下面四个值。

- start：对齐单元格的起始边缘。
- end：对齐单元格的结束边缘。
- center：单元格内部居中。
- stretch：拉伸，占满单元格的整个宽度（默认值）

13.3. 应用场景

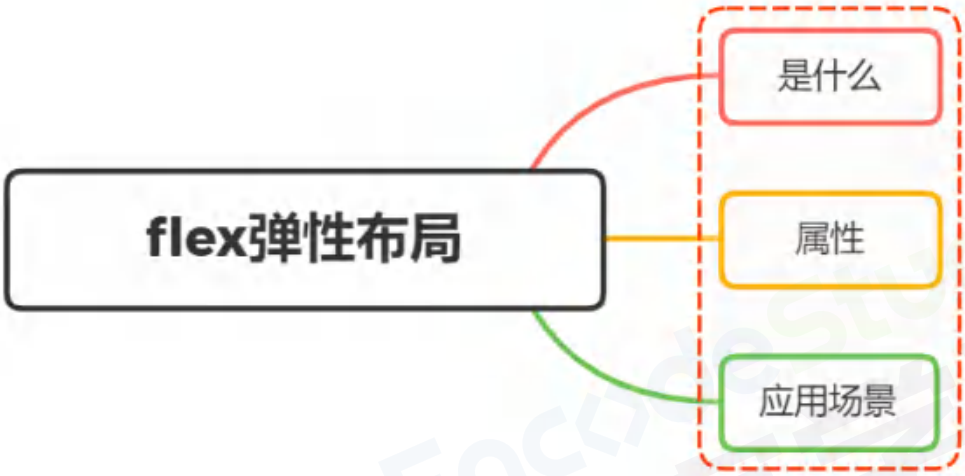
文章开头就讲到，`Grid` 是一个强大的布局，如一些常见的 CSS 布局，如居中，两列布局，三列布局等等是很容易实现的，在以前的文章中，也有使用 `Grid` 布局完成对应的功能

关于兼容性问题，结果如下：

总体兼容性还不错，但在 IE 10 以下不支持

目前，`Grid` 布局在手机端支持还不算太友好

14. 说说flexbox（弹性盒布局模型）,以及适用场景?

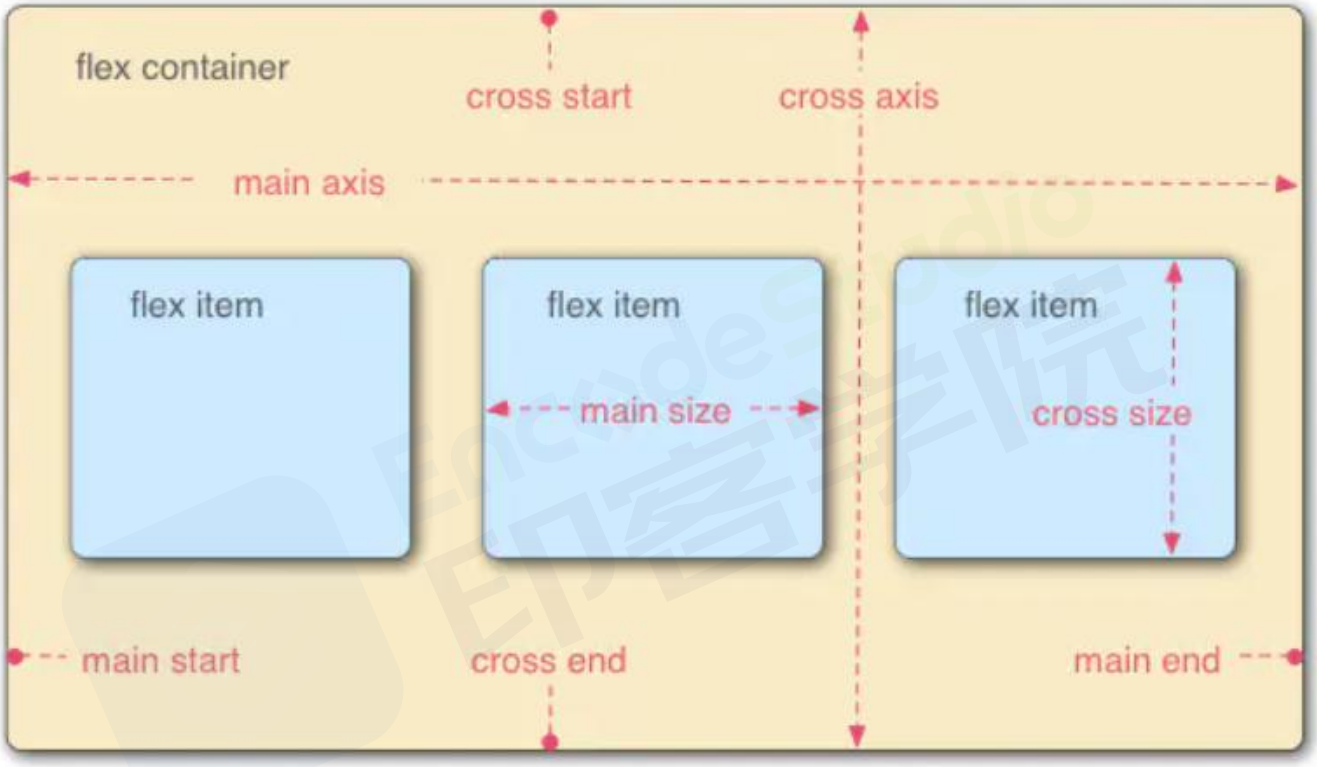


14.1. 是什么

`Flexible Box` 简称 `flex`，意为”弹性布局”，可以简便、完整、响应式地实现各种页面布局

采用Flex布局的元素，称为 `flex` 容器 `container`

它的所有子元素自动成为容器成员，称为 `flex` 项目 `item`



容器中默认存在两条轴，主轴和交叉轴，呈90度关系。项目默认沿主轴排列，通过 `flex-direction` 来决定主轴的方向

每根轴都有起点和终点，这对于元素的对齐非常重要

14.2. 属性

关于 `flex` 常用的属性，我们可以划分为容器属性和容器成员属性

容器属性有：

- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

14.2.1. flex-direction

决定主轴的方向(即项目的排列方向)

```css
.container {
    flex-direction: row | row-reverse | column | column-reverse;
}
```

属性对应如下：

- row（默认值）：主轴为水平方向，起点在左端
- row-reverse：主轴为水平方向，起点在右端
- column：主轴为垂直方向，起点在上沿。
- column-reverse：主轴为垂直方向，起点在下沿

如下图所示：

14.2.2. flex-wrap

弹性元素永远沿主轴排列，那么如果主轴排不下，通过 `flex-wrap` 决定容器内项目是否可换行

```css
.container {
    flex-wrap: nowrap | wrap | wrap-reverse;
}
```

属性对应如下：

- nowrap（默认值）：不换行
- wrap：换行，第一行在下方
- wrap-reverse：换行，第一行在上方

默认情况是不换行，但这里也不会任由元素直接溢出容器，会涉及到元素的弹性伸缩

14.2.3. flex-flow

是 `flex-direction` 属性和 `flex-wrap` 属性的简写形式，默认值为 `row nowrap`

```css
.box {
  flex-flow: <flex-direction> || <flex-wrap>;
}
```

14.2.4. justify-content

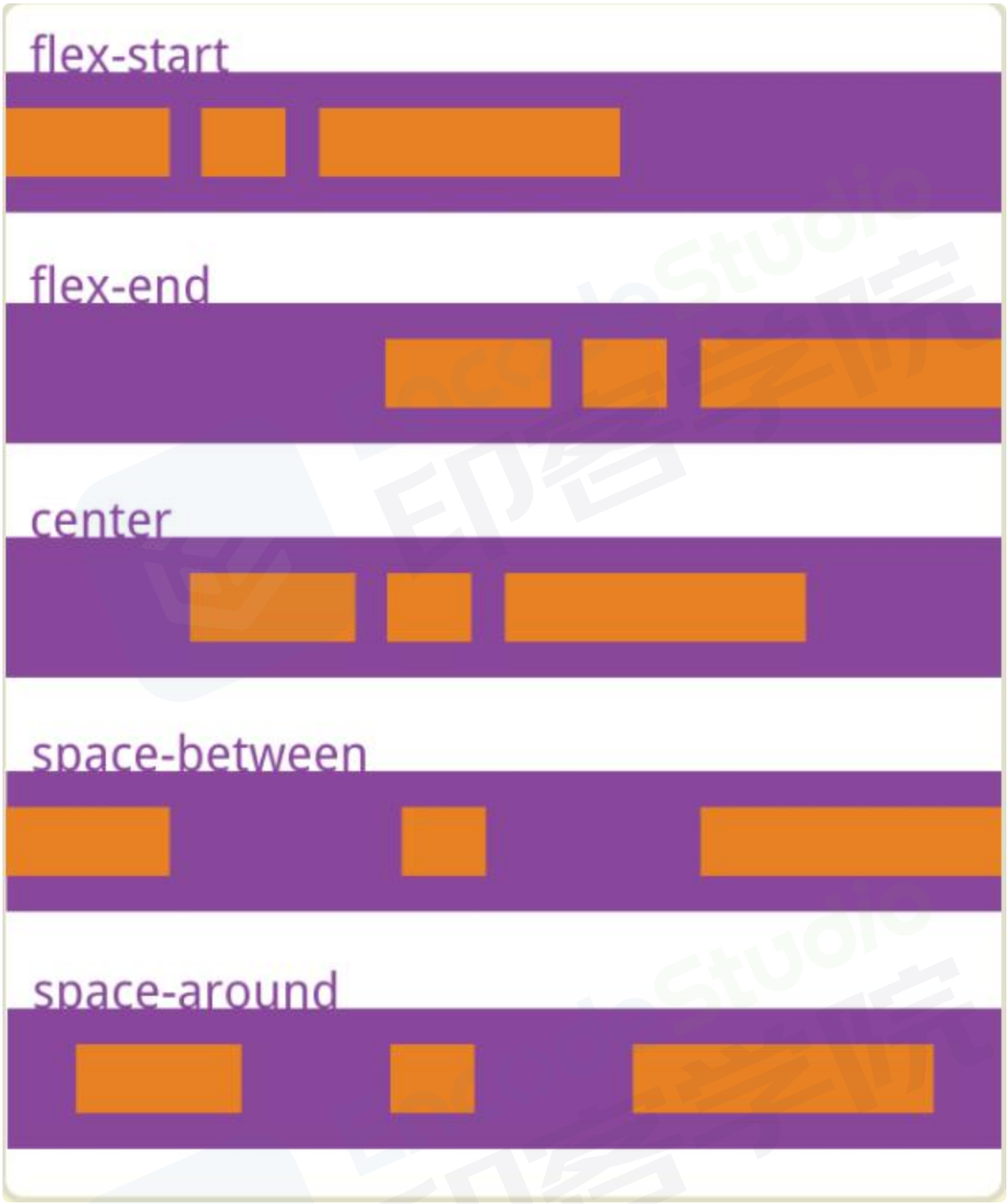
定义了项目在主轴上的对齐方式

```css
.box {
    justify-content: flex-start | flex-end | center | space-between | space-around;
}
```

属性对应如下：

- flex-start（默认值）：左对齐
- flex-end：右对齐
- center：居中
- space-between：两端对齐，项目之间的间隔都相等
- space-around：两个项目两侧间隔相等

效果图如下：



14.2.5. align-items

定义项目在交叉轴上如何对齐

CSS | 复制代码

```
.box {
  align-items: flex-start | flex-end | center | baseline | stretch;
}
```

属性对应如下：

- flex-start：交叉轴的起点对齐
- flex-end：交叉轴的终点对齐
- center：交叉轴的中点对齐
- baseline: 项目的第一行文字的基线对齐
- stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度

14.2.6. align-content

定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用

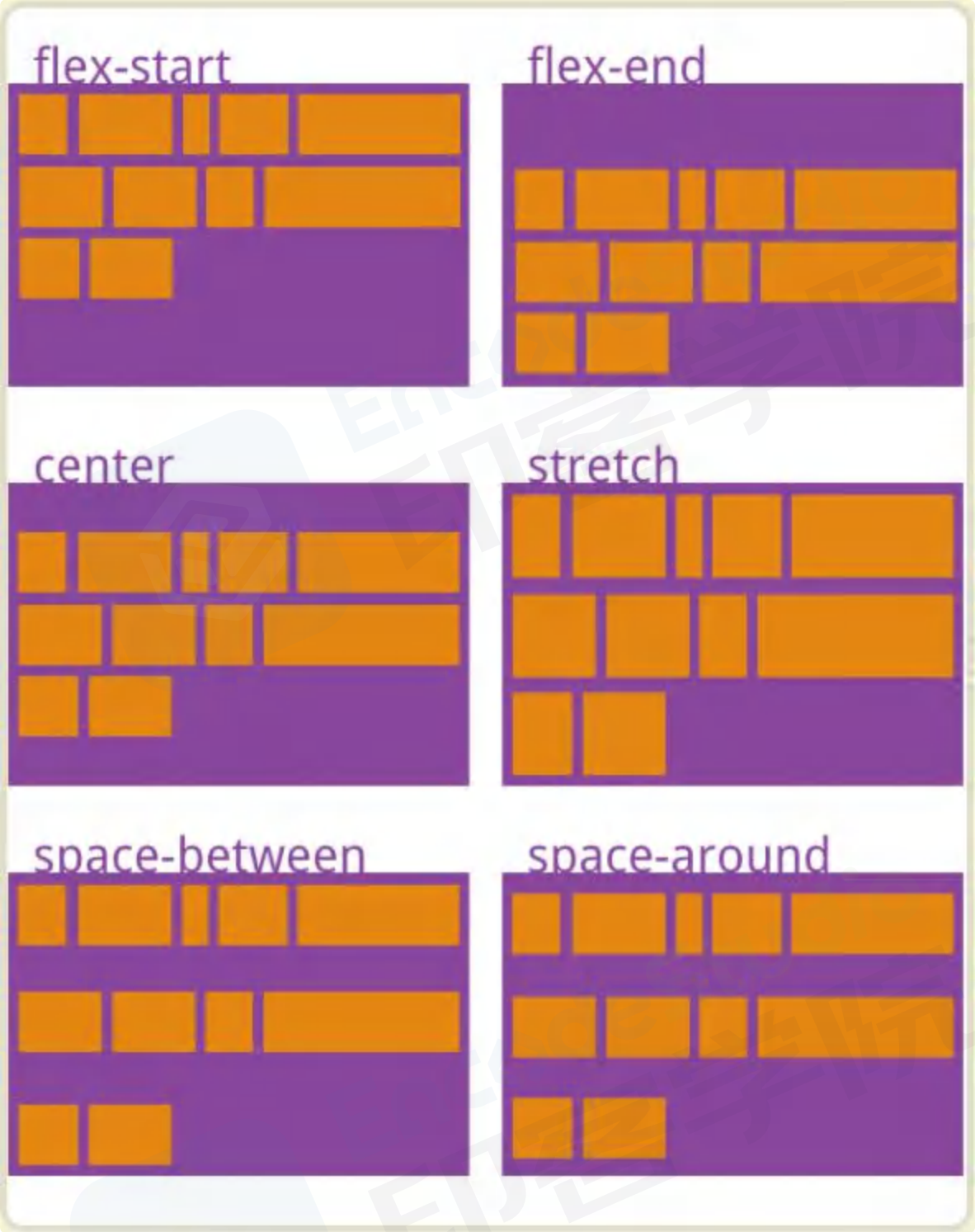
CSS 复制代码

```css
.box {
    align-content: flex-start | flex-end | center | space-between | space-around | stretch;
}
```

属性对应如下：

- flex-start：与交叉轴的起点对齐
- flex-end：与交叉轴的终点对齐
- center：与交叉轴的中点对齐
- space-between：与交叉轴两端对齐，轴线之间的间隔平均分布
- space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍
- stretch（默认值）：轴线占满整个交叉轴

效果图如下：



容器成员属性如下：

- `order`
- `flex-grow`
- `flex-shrink`
- `flex-basis`
- `flex`

- `align-self`

14.2.7. order

定义项目的排列顺序。数值越小，排列越靠前，默认为0

```CSS
.item {
    order: <integer>;
}
```

14.2.8. flex-grow

上面讲到当容器设为 `flex-wrap: nowrap;` 不换行的时候，容器宽度有不够分的情况，弹性元素会根据 `flex-grow` 来决定

定义项目的放大比例（容器宽度>元素总宽度时如何伸展）

默认为 `0`，即如果存在剩余空间，也不放大

```CSS
.item {
    flex-grow: <number>;
}
```

如果所有项目的 `flex-grow` 属性都为1，则它们将等分剩余空间（如果有的话）

如果一个项目的 `flex-grow` 属性为2，其他项目都为1，则前者占据的剩余空间将比其他项多一倍

弹性容器的宽度正好等于元素宽度总和，无多余宽度，此时无论 `flex-grow` 是什么值都不会生效

14.2.9. flex-shrink

定义了项目的缩小比例（容器宽度<元素总宽度时如何收缩），默认为1，即如果空间不足，该项目将缩小

```CSS
.item {
    flex-shrink: <number>; /* default 1 */
}
```

如果所有项目的 `flex-shrink` 属性都为1，当空间不足时，都将等比例缩小

如果一个项目的 `flex-shrink` 属性为0，其他项目都为1，则空间不足时，前者不缩小

在容器宽度有剩余时， `flex-shrink` 也是不会生效的

14.2.10. flex-basis

设置的是元素在主轴上的初始尺寸，所谓的初始尺寸就是元素在 `flex-grow` 和 `flex-shrink` 生效前的尺寸

浏览器根据这个属性，计算主轴是否有多余空间，默认值为 `auto` ，即项目的本来大小，如设置了 `width` 则元素尺寸由 `width/height` 决定（主轴方向），没有设置则由内容决定

```css
.item {
    flex-basis: <length> | auto; /* default auto */
}
```

当设置为0的是，会根据内容撑开

它可以设为跟 `width` 或 `height` 属性一样的值（比如350px），则项目将占据固定空间

14.2.11. flex

`flex` 属性是 `flex-grow` ，`flex-shrink` 和 `flex-basis` 的简写，默认值为 `0 1 auto` ，也是比较难懂的一个复合属性

```css
.item {
  flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]
}
```

一些属性有：

- flex: 1 = flex: 1 1 0%
- flex: 2 = flex: 2 1 0%
- flex: auto = flex: 1 1 auto
- flex: none = flex: 0 0 auto，常用于固定尺寸不伸缩

`flex:1` 和 `flex:auto` 的区别，可以归结于 `flex-basis:0` 和 `flex-basis:auto` 的区别

当设置为0时（绝对弹性元素），此时相当于告诉 `flex-grow` 和 `flex-shrink` 在伸缩的时候不需要考虑我的尺寸

当设置为 `auto` 时（相对弹性元素），此时则需要在伸缩时将元素尺寸纳入考虑

注意：建议优先使用这个属性，而不是单独写三个分离的属性，因为浏览器会推算相关值

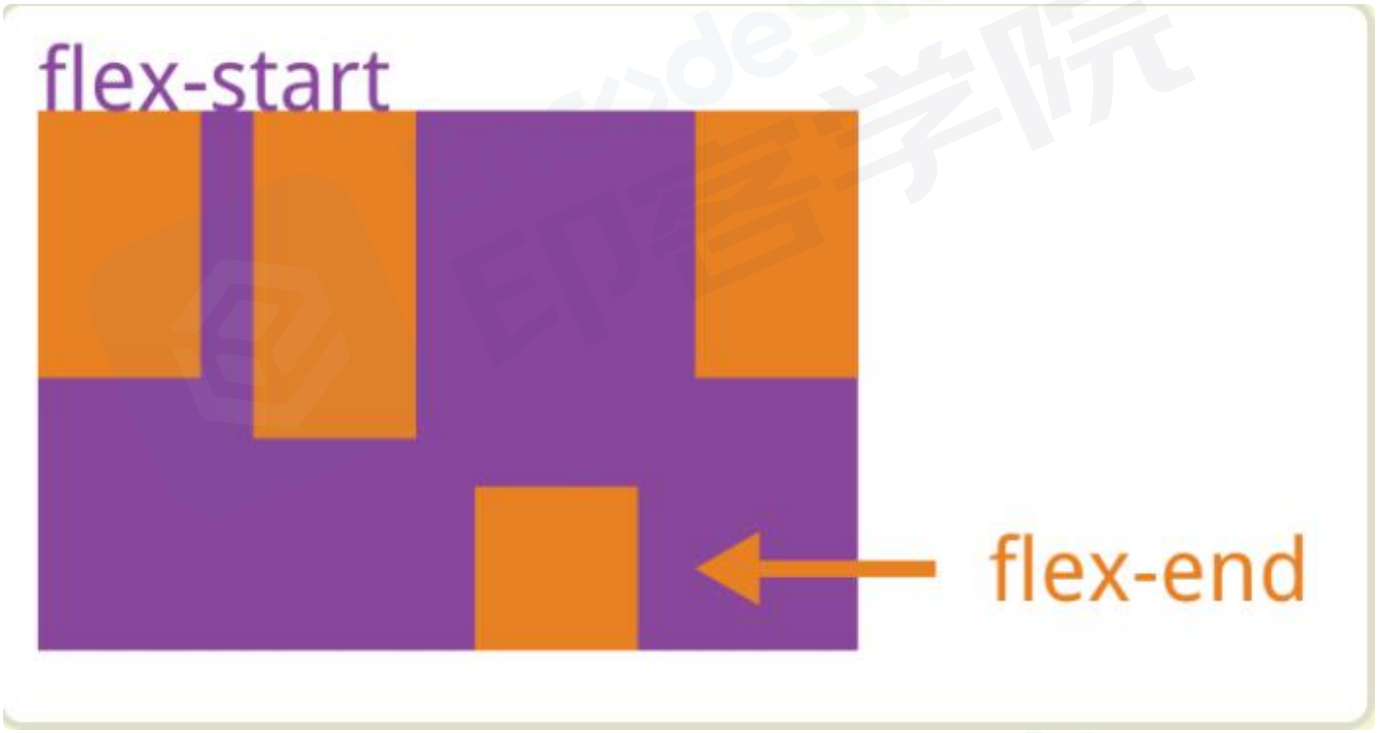
14.2.12. align-self

允许单个项目有与其他项目不一样的对齐方式，可覆盖 `align-items` 属性

默认值为 `auto` ，表示继承父元素的 `align-items` 属性，如果没有父元素，则等同于 `stretch`

```css
.item {
    align-self: auto | flex-start | flex-end | center | baseline | stretch;
}
```

效果图如下：



14.3. 应用场景

在以前的文章中，我们能够通过 `flex` 简单粗暴的实现元素水平垂直方向的居中，以及在两栏三栏自适应布局中通过 `flex` 完成，这里就不再展开代码的演示

包括现在在移动端、小程序这边的开发，都建议使用 `flex` 进行布局

15. 说说设备像素、css像素、设备独立像素、dpr、ppi 之间的区别?

15.1. 背景

在 `css` 中我们通常使用px作为单位，在PC浏览器中 `css` 的1个像素都是对应着电脑屏幕的1个物理像素

这会造成一种错觉，我们会认为 `css` 中的像素就是设备的物理像素

但实际情况却并非如此， `css` 中的像素只是一个抽象的单位，在不同的设备或不同的环境中， `css` 中的1px所代表的设备物理像素是不同的

当我们做移动端开发时，同为1px的设置，在不同分辨率的移动设备上显示效果却有很大差异

这背后就涉及了css像素、设备像素、设备独立像素、dpr、ppi的概念

15.2. 介绍

15.2.1. CSS像素

CSS像素（css pixel, px）：适用于web编程，在 CSS 中以 px 为后缀，是一个长度单位

在 CSS 规范中，长度单位可以分为两类，绝对单位以及相对单位

px是一个相对单位，相对的是设备像素（device pixel）

一般情况，页面缩放比为1，1个CSS像素等于1个设备独立像素

`CSS` 像素又具有两个方面的相对性：

- 在同一个设备上，每1个 CSS 像素所代表的设备像素是可以变化的（比如调整屏幕的分辨率）
- 在不同的设备之间，每1个 CSS 像素所代表的设备像素是可以变化的（比如两个不同型号的手机）

在页面进行缩放操作也会 引起 `css` 中 `px` 的变化，假设页面放大一倍，原来的 1px 的东西变成 2px，在实际宽度不变的情况下1px 变得跟原来的 2px 的长度（长宽）一样了（元素会占据更多的设备像素）

假设原来需要 320px 才能填满的宽度现在只需要 160px

px会受到下面的因素的影响而变化：

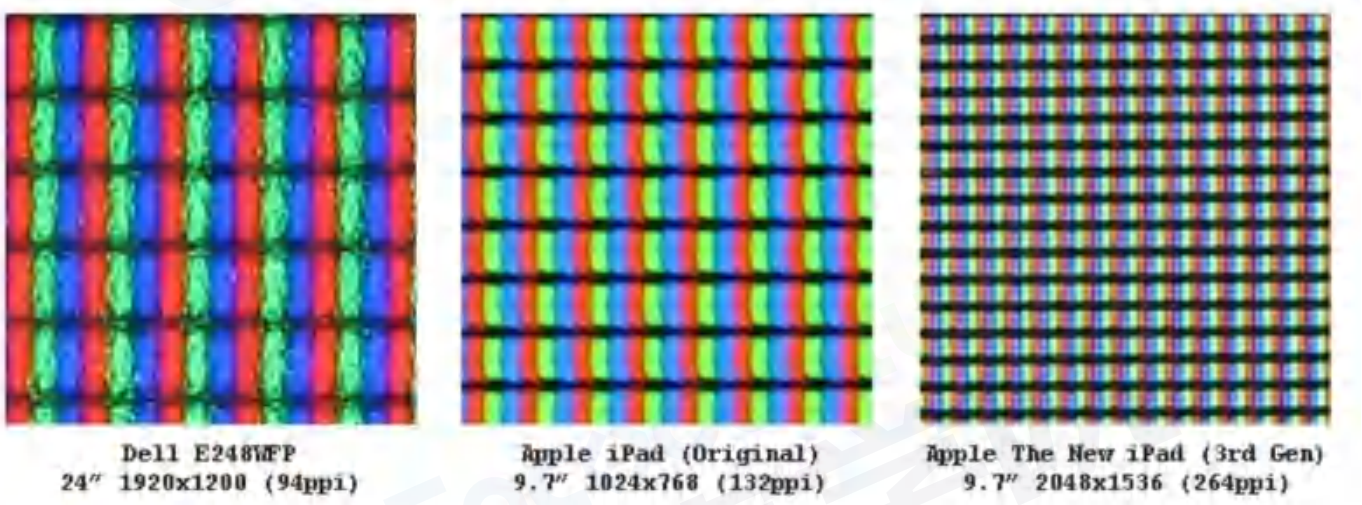
- 每英寸像素（PPI）
- 设备像素比（DPR）

15.2.2. 设备像素

设备像素（device pixels），又称为物理像素

指设备能控制显示的最小物理单位，不一定是一个小正方形区块，也没有标准的宽高，只是用于显示丰富色彩的一个“点”而已

可以参考公园里的景观变色彩灯，一个彩灯(物理像素)由红、蓝、绿小灯组成，三盏小灯不同的亮度混合出各种色彩



从屏幕在工厂生产出的那天起，它上面设备像素点就固定不变了，单位为 `pt`

15.2.3. 设备独立像素

设备独立像素（Device Independent Pixel）：与设备无关的逻辑像素，代表可以通过程序控制使用的虚拟像素，是一个总体概念，包括了CSS像素

在 `javaScript` 中可以通过 `window.screen.width/ window.screen.height` 查看

比如我们会说“电脑屏幕在 2560x1600分辨率下不适合玩游戏，我们把它调为 1440x900”，这里的“分辨率”（非严谨说法）指的就是设备独立像素

一个设备独立像素里可能包含1个或者多个物理像素点，包含的越多则屏幕看起来越清晰

至于为什么出现设备独立像素这种虚拟像素单位概念，下面举个例子：

iPhone 3GS 和 iPhone 4/4s 的尺寸都是 3.5 寸，但 iPhone 3GS 的分辨率是 320x480，iPhone 4/4s 的分辨率是 640x960

这意味着，iPhone 3GS 有 320 个物理像素，iPhone 4/4s 有 640 个物理像素

如果我们按照真实的物理像素进行布局，比如说我们按照 320 物理像素进行布局，到了 640 物理像素的手机上就会有一半的空白，为了避免这种问题，就产生了虚拟像素单位

我们统一 iPhone 3GS 和 iPhone 4/4s 都是 320 个虚拟像素，只是在 iPhone 3GS 上，最终 1 个虚拟像素换算成 1 个物理像素，在 iphone 4s 中，1 个虚拟像素最终换算成 2 个物理像素

至于 1 个虚拟像素被换算成几个物理像素，这个数值我们称之为设备像素比，也就是下面介绍的 `dpr`

15.2.4. dpr

dpr（device pixel ratio），设备像素比，代表设备独立像素到设备像素的转换关系，
在 `JavaScript` 中可以通过 `window.devicePixelRatio` 获取

计算公式如下：

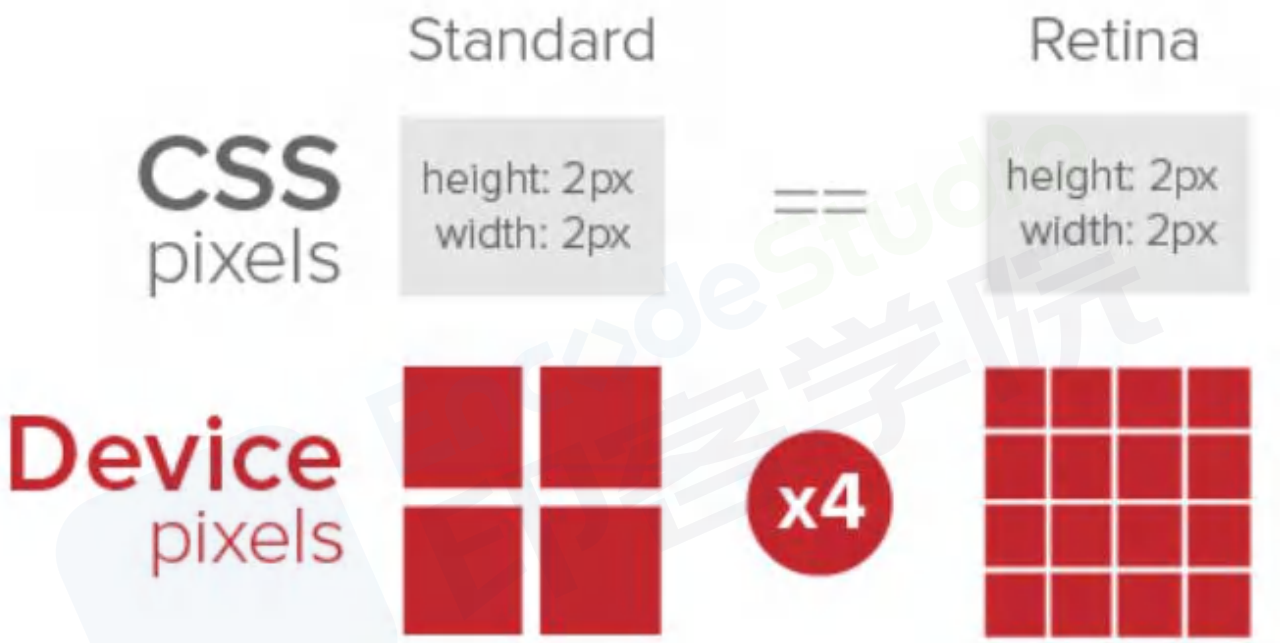
DPR = 设备像素 / 设备独立像素

当设备像素比为1:1时，使用1（1×1）个设备像素显示1个CSS像素

当设备像素比为2:1时，使用4（2×2）个设备像素显示1个CSS像素

当设备像素比为3:1时，使用9（3×3）个设备像素显示1个CSS像素

如下图所示：

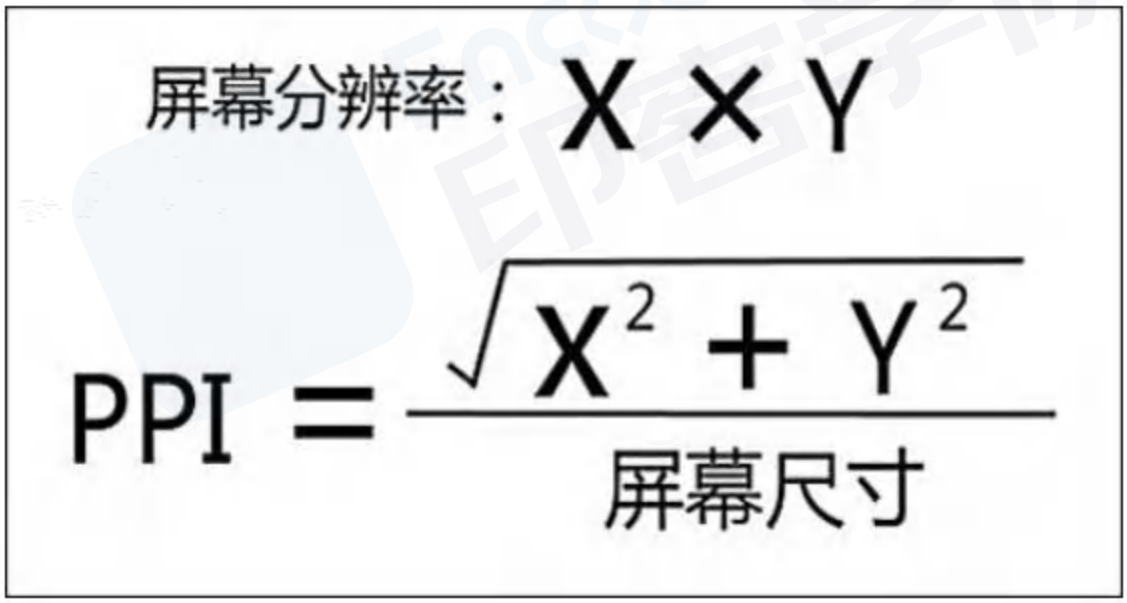


当 `dpr` 为3，那么 `1px` 的 `CSS` 像素宽度对应 `3px` 的物理像素的宽度，1px的 `CSS` 像素高度对应 `3px` 的物理像素高度

15.2.5. ppi

ppi （pixel per inch），每英寸像素，表示每英寸所包含的像素点数目，更确切的说法应该是像素密度。数值越高，说明屏幕能以更高密度显示图像

计算公式如下：



15.3. 总结

无缩放情况下，1个CSS像素等于1个设备独立像素

设备像素由屏幕生产之后就不发生改变，而设备独立像素是一个虚拟单位会发生改变

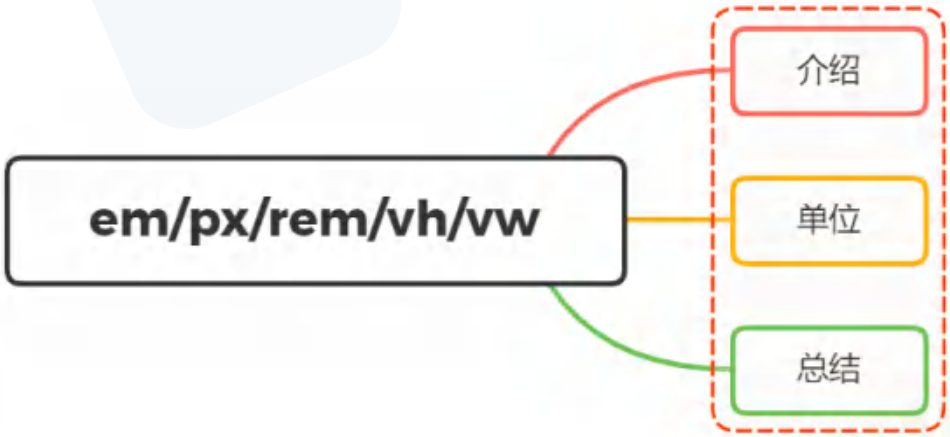
PC端中，1个设备独立像素 = 1个设备像素 （在100%，未缩放的情况下）

在移动端中，标准屏幕（160ppi）下 1个设备独立像素 = 1个设备像素

设备像素比（dpr） = 设备像素 / 设备独立像素

每英寸像素（ppi），值越大，图像越清晰

16. 说说em/px/rem/vh/vw区别?



16.1. 介绍

传统的项目开发中，我们只会用到 `px` 、 `%` 、 `em` 这几个单位，它可以适用于大部分的项目开发，且拥有比较良好的兼容性

从 `CSS3` 开始，浏览器对计量单位的支持又提升到了另外一个境界，新增了 `rem` 、 `vh` 、 `vw` 、 `vm` 等一些新的计量单位

利用这些新的单位开发出比较良好的响应式页面，适应多种不同分辨率的终端，包括移动设备等

16.2. 单位

在 `css` 单位中，可以分为长度单位、绝对单位，如下表所指示

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

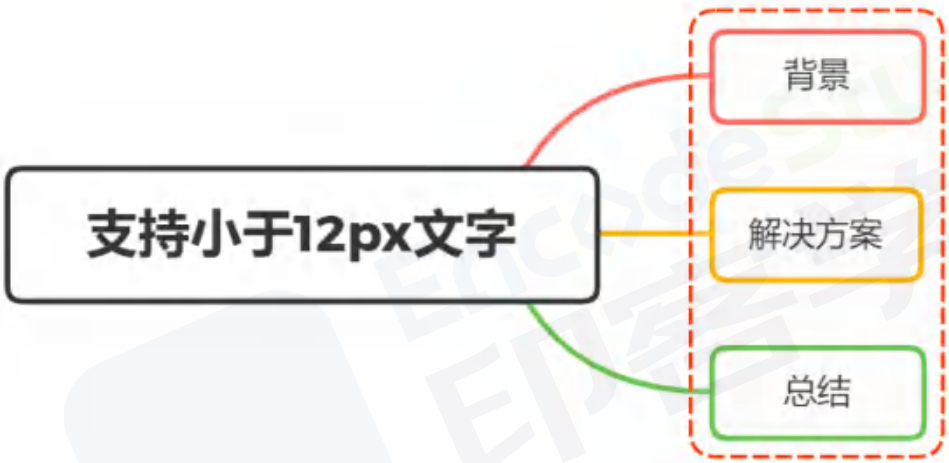
px：绝对单位，页面按精确像素展示

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

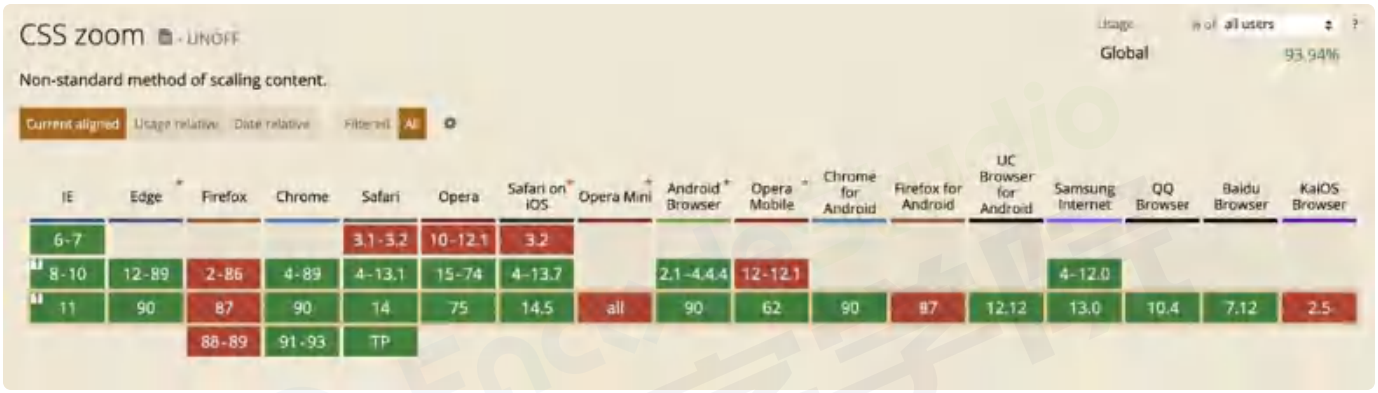
使用 `zoom` 来”支持“ 12px 以下的字体

代码如下：

```html
<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：

> 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
<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
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

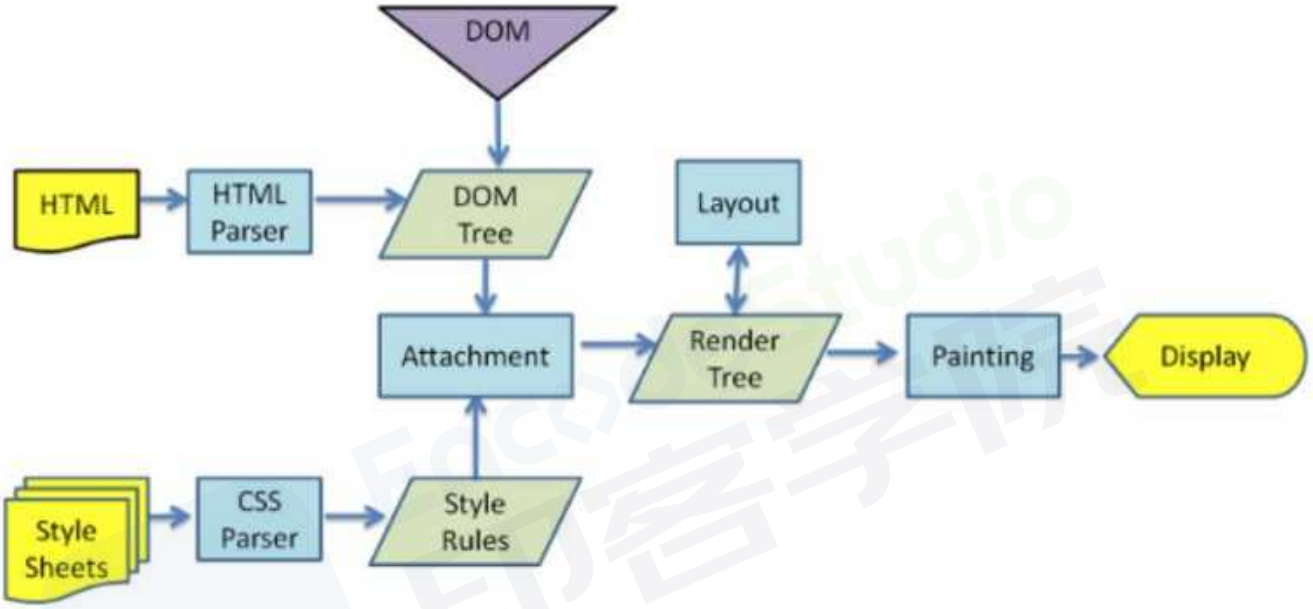
18. 怎么理解回流跟重绘？什么场景下会触发？

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
@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
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

- 减少使用昂贵的属性
- 不要使用@import

20.2.1. 内联首屏关键CSS

在打开一个页面，页面首要内容出现在屏幕的时间影响着用户的体验，而通过内联 `css` 关键代码能够使浏览器在下载完 `html` 后就能立刻渲染

而如果外部引用 `css` 代码，在解析 `html` 结构过程中遇到外部 `css` 文件，才会开始下载 `css` 代码，再渲染

所以， `CSS` 内联使用使渲染时间提前

注意：但是较大的 `css` 代码并不合适内联（初始拥塞窗口、没有缓存），而其余代码则采取外部引用方式

20.2.2. 异步加载CSS

在 `CSS` 文件请求、下载、解析完成之前， `CSS` 会阻塞渲染，浏览器将不会渲染任何已处理的内容

前面加载内联代码后，后面的外部引用 `css` 则没必要阻塞浏览器渲染。这时候就可以采取异步加载的方案，主要有如下

- 使用javascript将link标签插到head标签最后

JavaScript | 复制代码

```
// 创建link标签
const myCSS = document.createElement( "link" );
myCSS.rel = "stylesheet";
myCSS.href = "mystyles.css";
// 插入到header的最后位置
document.head.insertBefore( myCSS, document.head.childNodes[ document.head.childNodes.length - 1 ].nextSibling );
```

- 设置link标签media属性为noexis，浏览器会认为当前样式表不适用当前类型，会在不阻塞页面渲染的情况下再进行下载。加载完成后，将 `media` 的值设为 `screen` 或 `all` ，从而让浏览器开始解析CSS

HTML | 复制代码

```
<link rel="stylesheet" href="mystyles.css" media="noexist" onload="this.media='all'">
```

- 通过rel属性将link元素标记为alternate可选样式表，也能实现浏览器异步加载。同样别忘了加载完成之后，将rel设回stylesheet

```HTML
<link rel="alternate stylesheet" href="mystyles.css" onload="this.rel='stylesheet'">
```

20.2.3. 资源压缩

利用 `webpack` 、 `gulp/grunt` 、 `rollup` 等模块化工具，将 `css` 代码进行压缩，使文件变小，大大降低了浏览器的加载时间

20.2.4. 合理使用选择器

`css` 匹配的规则是从右往左开始匹配，例如 `#markdown .content h3` 匹配规则如下：

- 先找到h3标签元素
- 然后去除祖先不是.content的元素
- 最后去除祖先不是#markdown的元素

如果嵌套的层级更多，页面中的元素更多，那么匹配所要花费的时间代价自然更高

所以我们在编写选择器的时候，可以遵循以下规则：

- 不要嵌套使用过多复杂选择器，最好不要三层以上
- 使用id选择器就没必要再进行嵌套
- 通配符和属性选择器效率最低，避免使用

20.2.5. 减少使用昂贵的属性

在页面发生重绘的时候，昂贵属性如 `box-shadow` / `border-radius` / `filter` /透明度/ `:nth-child` 等，会降低浏览器的渲染性能

20.2.6. 不要使用@import

css样式文件有两种引入方式，一种是 `link` 元素，另一种是 `@import`

`@import` 会影响浏览器的并行下载，使得页面在加载时增加额外的延迟，增添了额外的往返耗时

而且多个 `@import` 可能会导致下载顺序紊乱

比如一个css文件 `index.css` 包含了以下内容： `@import url("reset.css")`

那么浏览器就必须先把 `index.css` 下载、解析和执行后，才下载、解析和执行第二个文件 `reset.css`

20.2.7. 其他

- 减少重排操作，以及减少不必要的重绘
- 了解哪些属性可以继承而来，避免对这些属性重复编写
- cssSprite，合成所有icon图片，用宽高加上backgroud-position的背景图方式显现出我们要的icon图，减少了http请求
- 把小的icon图片转成base64编码
- CSS3动画或者过渡尽量使用transform和opacity来实现动画，不要使用left和top属性

20.3. 总结

`css` 实现性能的方式可以从选择器嵌套、属性特性、减少 `http` 这三面考虑，同时还要注意 `css` 代码的加载顺序。