CSS 笔记

Pionpill [1]

本文档为作者系统学习 CSS 时的笔记。

2022 年 12 月 21 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

本文主要参考书籍: « 深入解析 CSS»[1]。顾名思义，主要参考书籍是一本进阶书，因此本文不适合完全不了解 CSS 的入门读者。在阅读本文之前，确保有以下前置知识:

- 掌握 HTML。
- 熟悉 CSS 的部分属性以及一些选择器。

HTML 与 CSS 的诸多标签/属性不一定要全部知道，不了解的直接查 MDN 文档即可。

这本笔记完全不能代替原书，仅是对原书的一个要点记录，只有对 CSS 有经验但需要查阅资料的读者适合本书。也可以将本书当作一个概要，根据不了的内容另行查找资料。

本人的编写及开发环境如下：

- IDE: VSCode 1.72
- Chrome: 91.0
- Node.js: 18.12
- OS: Window11

2022 年 12 月 21 日

[1]«CSS IN DEPTH»: [美] Keith J. Grant 黄小璐译 2020 年 4 月第一版

目录

III 动画

I 基础

1 层叠样式表

1.1 层叠

层叠决定了如何解决冲突，当声明冲突时，层叠会依次根据三种条件解决冲突：

- 样式表的来源: 内联 > 外部 > 默认
- 选择器优先级: 例如 `id` > `class`
- 源码顺序: 样式在样式表里的声明顺序。

1.1.1 样式表的来源

开发者编写的样式表属于作者样式表，除此之外还有用户代理样式表 (浏览器默认样式)。有的浏览的允许用户定义一个用户样式表，优先级介于作者样式表的用户代理样式表之间。

样式表的优先级为: 作者样式表 > 用户样式表 > 用户代理样式表。

`!important` 声明

样式来源规则有一个例外：标记为重要（`important`）的声明:

```
color: red !important
```

标记了 `!important` 的声明会被当作更高优先级的来源。

1.1.2 理解优先级

浏览器将优先级分为两部分：HTML 的行内样式和选择器的样式。

行内样式

如果用 HTML 的 `style` 属性写样式，这个声明只会作用于当前元素。实际上行内元素属于“带作用域的”声明，它会覆盖任何来自样式表或者 `<style>` 标签的样式。行内样式没有选择器，因为它们直接作用于所在的元素。

如果要覆盖样式表里的行内声明，需要为声明添加`!important`，这样能将它提升到一个更高优先级的来源。但如果行内样式也被标记为`!important`，就无法覆盖它了。最好是只在样式表内用`!important`。

选择器优先级

一般的，选择器优先级如下: ID > Class > Tag。伪类选择器与属性选择器和类选择器优先级相同; 通用选择器和组合器对优先级没有影响。

有时候我们会混用多个选择器，一个常用的表示优先级的方式是用数值形式来标记，通常用逗号隔开每个数。如，“1,2,2”表示选择器由 1 个 ID、2 个类、2 个标签组成。优先级最高的 ID 列为第一位，紧接着是类，最后是标签。

比如 `#page-header #page-title` 有两个 Id, 表示为 “2,0,0” `ul li` 有 2 个标签，表示为 “0,0,2”，很明显 200 > 002，前者优先级更高。

如果涉及到行内样式，可以在最前面加一个数字表示是否为行内样式，此时，行内样式的优先级为 “1,0,0,0”。不过这并没有实际意义，因为行内样式总是高于外部样式，而行内样式又不会写 (一般不写) 得过于复杂。

至于 `!important`，所有使用该关键字的样式都会提升到最高的优先级来源 (升维)，因此他总是最高级的，多个 `!important` 会让一切回到起点，即撤去 `!important` 比较优先级。

1.1.3 源码顺序

如果两个声明的来源和优先级相同，其中一个声明在样式表中出现较晚，或者位于页面较晚引入的样式表中，则该声明胜出。

```
.nav a {       /* 0,1,1 */
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

a.featured {   /* 0,1,1 */
  background-color: orange;
}
```

上面两个声明块优先级都是 “0,1,1” 但下面的被选中使用。

由于这个特性多个状态的书写顺序就显得尤其重要，比如对按钮有悬停和激活两种状态，如果悬停状态在激活状态之后，激活后将会显示悬停的样式。因此状态的书写顺序应该为: `a:linked > a:visited > a:hover > a:active`。

层叠值

浏览器遵循三个步骤，即来源、优先级、源码顺序，来解析网页上每个元素的每个属性。如果一个声明在层叠中 “胜出”，它就被称作一个层叠值。元素的每个属性最多只有一个层叠值。

两条经验法则

处理层叠时有两条通用的经验法则:

- 非必要不要使用 `ID`: 如果不是要通过脚本控制节点树，应不用 ID 选择器。
- 不要使用 `!important`: 使用 `!important` 将带来一个新的维度，这让维护变得非常困难。

1.2 继承

如果一个元素的某个属性没有层叠值，则可能会继承某个祖先元素的值。比如通常会给 `<body>` 元素加上 `font-family`，里面的所有后代元素都会继承这个字体，就不必给页面的每个元素明确指定字体了。

但不是所有的属性都能被继承。默认情况下，只有特定的一些属性能被继承，通常是我们希望被继承的那些。它们主要是跟文本相关的属性：`color`、`font`、`font-family`、`font-size`、`font-weight`、`font-variant`、`font-style`、`line-height`、`letter-spacing`、`text-align`、`text-indent`、`text-transform`、`white-space`、`ord-spacing`。

此外，列表的一些属性也会被继承。

`inherit` 关键字

有时，我们想用继承代替一个层叠值。这时候可以用 `inherit` 关键字。可以用它来覆盖另一个值，这样该元素就会继承其父元素的值。

```
a {
    color: inherit;
    text-decoration: underline;
}
```

这么做的好处是如果父样式颜色改变，子样式也会随之改变。

`initial` 关键字

`initial` 关键字用于撤销作用于某个元素的样式，恢复为默认值。

```
a {
    color: initial;
    text-decoration: underline;
}
```

和 `initial` 很像的有 `auto` 值，但 `auto` 并不是所有属性的默认值。

1.3 相对单位

CSS 为网页带来了后期绑定（late-binding）的样式：直到内容和样式都完成了，二者才会结合起来。这会给设计流程增加复杂性，而这在其他类型的图形设计中是不存在的。不过这也带来了好处，即一个样式表可以作用于成百上千个网页。此外，用户还能直接改变最终的渲染效果，比如用户可以改变默认字号或者缩放浏览器窗口。

em 和 rem

em 是最常见的相对长度单位，适合基于特定的字号进行排版。在 CSS 中，1em 等于当前元素的字号，其准确值取决于作用的元素。浏览器会根据相对单位的值计算出绝对值，称作计算值（computed value）。

由于 em 指代当前元素的字号，因此 font-size 不可以指定为 em，比较字号不能是自己的多少倍，不过如果父级可以得出字号大小，这样做是可以做的。默认的字号是 16px，也即 medium 关键字的值。

em 相信绝大多数人都用过，rem 就不一定了。rem 是 root em 的缩写，表示根元素的单位。除此之外还有个 ex 表示 x 字体的大小，通常是 em 的一半。

在实际开发中，通常用用 rem 设置字号，用 px 设置边框，用 em 设置其他大部分属性。

视口相对单位

前面介绍的 em 和 rem 都是相对于 font-size 定义的，但 CSS 里不止有这一种相对单位。还有相对于浏览器视口定义长度的视口的相对单位:

- vh: 视口高度的 1/100;
- vw: 视口宽度的 1/100;
- vmin: 视口高度和宽度中较小一方的 1/100;
- vmax: 视口高度和宽度中较大一方的 1/100。

灵活使用这些相对单位可以帮助我们有效解决跨端问题。

此外，CSS 提供了一个 calc() 函数，可以对值进行基本运算，支持包括加减乘除运算，运算符两边都需要留出一个空格。比如 calc(1em + 10px)。

无单位值

当一个元素的值定义为长度（px、em、rem，等等）时，子元素会继承它的计算值。当使用 em 等单位定义行高时，它们的值是计算值。使用无单位的数值时，继承的是声明值，即在每个继承子元素上会重新算它的计算值。这样得到的结果几乎总是我们想要的。

自定义属性

自定义属性 (全称: 层叠变量的自定义属性) 给 CSS 引进了变量的概念，开启了一种全新的基于上下文的动态样式。你可以声明一个变量，为它赋一个值，然后在样式表的其他地方引用这个值。

比如我们定义一个自定义属性:

```
:root {
    --main-font: Helvetica, Arial, sans-serif;
}
```

调用函数 var() 就能使用该变量。

```
p {
    font-family: var(--main-font);
}
```

在样式表某处为自定义属性定义一个值，作为“单一数据源”，然后在其他地方复用它。这种方式特别适合反复出现的值，比如颜色值。

如果 `var()` 函数算出来的是一个非法值，对应的属性就会设置为其初始值。

2 盒模型

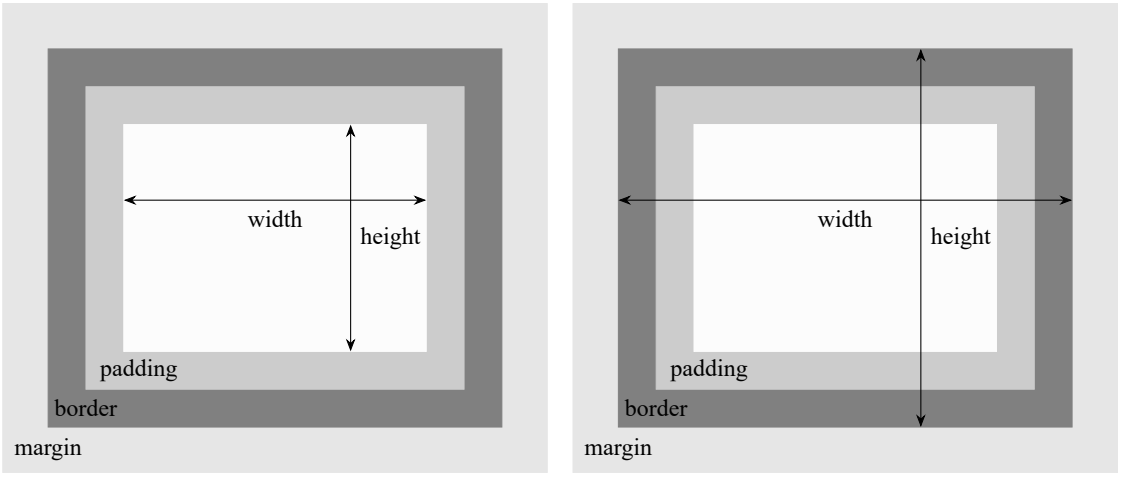
2.1 元素宽度问题

元素宽度包括基础的盒模型概念，多个盒模型占整个网页的宽度。

盒模型

盒模型的宽度有两种计算方式，使用 box-sizing 控制，有两个属性值:

- content-box: 高宽仅包含内部元素，默认值。
- border-box: 高宽包括 padding 和 border。



box-sizing: content-box　　box-sizing: border-box

图 2.1 box-sizing

在实际开发中，我们更常用 border-box 值，因为这样有助于我们对其视口。

如我们使用默认值，在指定高宽时由于 padding，border，margin 都会影响整个盒模型在布局中的占用情况，如果我们缩放界面或者使用其他平台的设备查看网页，则很难确认最终的呈现效果。

当然这个缺陷有几个解决方案，一是 padding，border，margin 都设置为 0，属于是自损 800 的方案。而是情况慢慢调整 width，height 的值，这在一个界面上也许有用，但如果在手机等其他设备上效果往往不尽人意。

box-sizing 的继承问题

前面说过，只有内容相关和表格的一些属性会被继承，很遗憾 box-sizing 是不能继承的，这会给开发带来很多繁琐的问题，下面有几个解决方案:

- **全局修改盒模型**: 这是最常用的方案，将所有盒模型均设置为 border-box，一般情况下没有问题。但如果我们使用第三方组件，如果有些组件使用的是 content-box，这会改变这些组建的样式。

```
*, ::before, ::after {
    box-sizing: border-box;
}
```

- **根节点继承**: 为了避免组件样式被修改，我们可以使用 `inherit` 强制 `box-sizing` 继承，再在需要继承的父节点 (通常是根节点，也即 `<html>`) 节点设置盒模型方案。

```
:root {
    box-sizing: border-box;
}
*, ::before, ::after {
    box-sizing: inherit;
}
```

2.2 元素高度问题

高度与宽度不同，由于网页高度是无限的，可以上下滑动，因此在高度方面我们更加关注元素的内容。为什么不用宽度来控制内容呢，因为宽度更加注重设备缩放以及分辨率问题。

控制溢出行为

当明确设置一个元素的高度时，内容可能会溢出容器。用 `overflow` 属性可以控制溢出内容的行为，该属性支持以下 4 个值:

- `visible`: 所有内容可见，即使溢出容器边缘。
- `hidden`: 溢出容器内边距边缘的内容被裁剪，无法看见。
- `scroll`: 容器出现滚动条，用户可以通过滚动查看剩余内容。在某些情况下，会出现水平和数值两种滚动条。
- `auto`: 只有内容溢出时容器才会出现滚动条，多数情况下使用 `auto` 而不是 `scroll`。

虽然不推荐,但是少数情况下宽度也是可以控制溢出行为的,使用`overflow-y, overflow-x`可分别控制高宽的溢出行为。

和高度相关的很多问题会在后面布局章节中说明，高度往往是随着布局的改变而改变的，除非特别需要，否则不要设置元素的高度，这会带来更棘手的问题。

`min-height` 与 `max-height`

这两个属性可以指定最小或最大值，而不是明确定义高度，这样元素就可以在这些界限内自动决定高度。同样的还有 `min-width, max-width`。

2.3 外边距问题

负外边距

不同于 `border, padding`, 外边距 `margin` 是可以设置为负值得。负外边距有一些特殊用途，比如让元素重叠或者拉伸到比容器还宽。

负外边距的具体行为取决于设置在元素的哪边:

- 如果设置左边或顶部的负外边距，元素就会相应地向左或向上移动，导致元素与它前面的元素重叠。
- 如果设置右边或者底部的负外边距，并不会移动元素，而是将它后面的元素拉过来。

警告 2.1. 如果元素被别的元素遮挡，利用负外边距让元素重叠的做法可能导致元素不可点击。

外边距折叠

有一个奇怪的现象，有时候我们设置外边距为 1em，但是多个盒模型顶部/底部相邻时，理论上应该是 2em，但实际上只有 1em 距离，被吃了一半。这种现象叫做外边距折叠。

外边距折叠的主要原因与包含文字的块之间的间隔相关。段落（`<p>`）默认有 1em 的上外边距和 1em 的下外边距。这是用户代理的样式表添加的，但当前后叠放两个段落时，它们的外边距不会相加产生一个 2em 的间距，而会折叠，只产生 1em 的间隔。

折叠外边距的大小等于相邻外边距 (两个及以上) 中的最大值。

有几个方式可以防止外边距折叠:

- 对容器使用 `overflow: auto`（或者非 `visible` 的值），防止内部元素的外边距跟容器外部的外边距折叠。这种方式副作用最小。
- 在两个外边距之间加上边框或者内边距，防止它们折叠。
- 如果容器为浮动元素、内联块、绝对定位或固定定位时，外边距不会在它外面折叠。
- 当使用 `Flexbox` 布局或网络布局时，弹性布局内的元素之间不会发生外边距折叠。
- 当元素显示为 `table-cell` 时不具备外边距属性,因此它们不会折叠。此外还有 `table-row` 和大部分其他表格显示类型

II 布局

3 浮动布局

浮动布局，顾名思义，它会让元素浮动在界面上，类似于 word 文档里面的图片浮动。

浮动布局是一种很老的布局方案，现在基本已经不用了，Flexbox 正在取代浮动布局，一般只有做旧版兼容的时候会用到浮动布局。但要实现将图片移动到网页一侧，并且让文字围绕图片的效果，浮动仍然是唯一的方法。

容器折叠

浮动布局存在一些问题 (特性)，浮动元素不同于普通文档流的元素，它们的高度不会加到父元素上。类似 word 文章中图片的高度由自己确定，对其他元素没有影响。

在网页中，这个特性存在一些问题，有时候浮动元素需要被父级元素包围，最简单的方式是增加一个 `<div>` 标签，并添加 `clear: both` 属性，`clear` 属性指定一个元素是否必须移动 (清除浮动后) 到在它之前的浮动元素下面。这样直到有 `clear` 属性标签的内容都会被包括在内:

```
<div style="clear: both"></div>
```

但这样存在一个问题，我们为了添加 CSS 属性而创建了新的 HTML 标签。

进一步的解决方案是使用 `::after` 伪元素，这样就不用构建新的标签了:

```
.float::after {
    clear: both;
}
```

注意，要给包含浮动的元素清除浮动，而不是给别的元素，比如浮动元素本身，或包含浮动的元素的后面的兄弟元素。

这个清除浮动还有个一致性问题没有解决：浮动元素的外边距不会折叠到清除浮动容器的外部，非浮动元素的外边距则会正常折叠。要解决这个问题可以使用 `display: table` 使用表格布局。

BFC

BFC(block formatting context) 即块级格式化上下文, BFC 是网页的一块区域，元素基于这块区域布局。虽然 BFC 本身是环绕文档流的一部分，但它将内部的内容与外部的上下文隔离开。这种隔离为创建 BFC 的元素做出了以下 3 件事情。

- 包含了内部所有元素的上下外边距。它们不会跟 BFC 外面的元素产生外边距折叠。
- 包含了内部所有的浮动元素。
- 不会跟 BFC 外面的浮动元素重叠。

简而言之，BFC 里的内容不会跟外部的元素重叠或者相互影响。如果给元素增加 `clear` 属性，它只会清除自身所在 BFC 内的浮动。如果强制给一个元素生成一个新的 BFC，它不会跟其他 BFC 重叠。给元素添加以下的任意属性值都会创建 BFC。

- `float`: `left` 或 `right`。
- `overflow`: `hidden`、`auto`、`scroll`。
- `display`: `inline-block`、`table-cell`、`table-caption`、`flex`、`inline-flex`、`grid`、`inline-grid`。拥有这些属性的元素称为块级容器（block container）。
- `position: absolute, fixed`。

最常见的设置 BFC 的方式是: `overflow: auto`，这并不会实际上改变什么，但是设置了 BFC。

网格系统

网络系统是一种定义样式的方案，而不是具体的技术。

网格系统可以提高代码的可复用性。网格系统提供了一系列的类名，可添加到标记中，将网页的一部分构造成行和列。它应该只给容器设置宽度和定位，不给网页提供视觉样式，比如颜色和边框。需要在每个容器内部添加新的元素来实现想要的视觉样式。

要构建一个网格系统，首先要定义它的行为。通常网格系统的每行被划分为特定数量的列，一般是 12 个，但也可以是其他数。每行子元素的宽度可能等于 1 12 个列的宽度。选取 12 作为列数是因为它能够被 2、3、4、6 整除，组合起来足够灵活。

在实际操作中，我们往往给标签添加 `column-x` 这样的属性构建网络，在 CSS 中使用 `[class*="column-"]` 这类属性选择器批量控制网络。再通过 `column-x` 详细控制单个元素属性。

4 弹性布局

弹性布局是浮动布局的替代方案，Flexbox 全称弹性盒子布局 (Flexible Box Layout)，跟浮动布局相比，Flexbox 的可预测性更好，还能提供更精细的控制。

Flexbox 唯一不算缺陷的缺陷是引入了很多新的属性: 12 种。

4.1 弹性容器基础

弹性布局原则

使用弹性布局需要给元素添加 `display: flex` 标签。该元素变成了一个弹性容器（flex container），它的直接子元素变成了弹性子元素（flex item）。

- 弹性子元素: 默认在同一行从左到右的顺序并排排列。高度相等，由内容决定。
- 弹性容器: 像块元素一样填满可用宽度。

> **注 4.1.** 可以使用 *display: inline-flex*。和 *flex* 唯一的区别是宽度不会占满视口。*display* 的其他值 *block, inline-block* 只影响对应的元素，但弹性布局会控制内部的元素。

子元素按照主轴线排列，主轴的方向为主起点（左）到主终点（右）。垂直于主轴的是副轴。方向从副起点（上）到副终点（下）。主轴与副轴的方向可以改变。

图 4.1 弹性元素

弹性子元素的大小

一般我们使用 `width, height` 属性设置元素的大小，但是 Flexbox 提供了更强大的属性: `flex`, `flex` 可以用简写，该属性包含了以下三个属性 (依次为: `flex-grow, flex-shrink, flex-grow`):

- `flex-basis`: 定义了元素大小的基准值，即一个初始的“主尺寸”。`flex-basis` 属性可以设置为任意的 `width` 值，包括 `px`、`em`、百分比。它的初始值是 `auto`，此时浏览器会检查元素是否设置了 `width` 属性值。如果有，则使用 `width` 的值作为 `flex-basis` 的值；如果没有，则用元素内容自身的大小。如果 `flex-basis` 的值不是 `auto`，`width` 属性会被忽略。

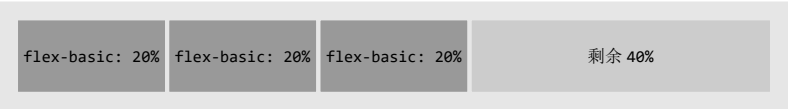


图 4.2 flex-basis

每个弹性子元素的初始主尺寸确定后，它们可能需要在主轴方向扩大或者缩小来适应（或者填充）弹性容器的大小。这时候就需要 `flex-grow` 和 `flex-shrink` 来决定缩放的规则。

• `flex-grow`: 每个弹性子元素的 `flex-basis` 值计算出来后，它们（加上子元素之间的外边距）加起来会占据一定的宽度。加起来的宽度不一定正好填满弹性容器的宽度，可能会有留白. 多出来的留白（或剩余宽度）会按照 `flex-grow`（增长因子）的值按比例分配给每个弹性子元素。

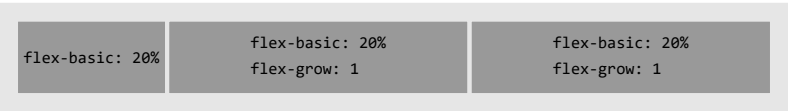


图 4.3 flex-growth

• `flex-shrink`: 前两个属性都是对于盒模型宽度来说的,如果 `margin` 设置的过大,容器内的元素完全有可能超出范围,因此需要 `flex-shrink` 来收缩,收缩的算法和 `flex-grow` 相同。

`flex` 属性最后两个值默认是 `flex-shrink: 1, flex-grow: 0%`。最好使用 `flex` 而不是精确的 `flex-xxx`，因为这样会有后两个属性的默认值。

弹性方向

在使用弹性布局时，同一容器内元素的高度是相同的，但有时会出现如下的情况:

图 4.4 弹性布局板块高度不一致

左右两列为弹性布局容器中的两个元素，虽然两个子元素是等高的，但是右边栏内部的两个板块没有扩展到填满右边栏区域。

对应的解决方案是将右侧子元素设置为弹性容器，并改变弹性方向（`flex-direction`）:

```
.column2 {
    display: flex;
    flex-direction: column;
}
```

同时将右侧的两个元素加上 flex-growth:

```
.element1 .element2 {
    flex: 1;
}
```

这样，两个子元素就会自动在竖直方向上填充。

4.2 弹性容器的属性

容器换行

如果界面缩的很小，由于弹性元素的部分内容会占固定的宽度 (即最小宽度), 这时候部分内容可能就会超出屏幕范围，如果不想这样，可以使用 flex-wrap 属性。

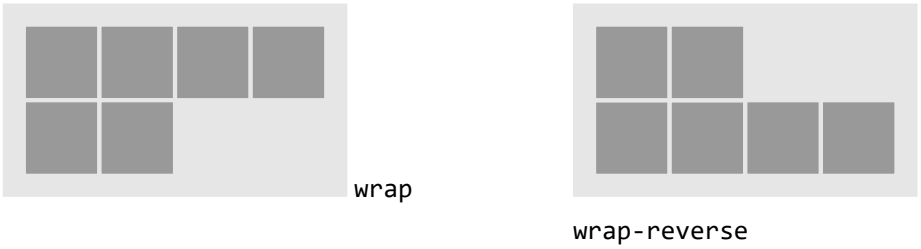


图 4.5 flex-direction

默认情况下，flex-wrap 值为 nowrap。如果修改了弹性方向，对应的换行也会改变。

此外，有一个 flex-flow 简写，包含两个值，分别为 flex-direction flex-wrap。

容器主轴对齐

justify-content 用于控制子元素主轴上的位置，主要是针对空白区域的填充方案，默认值为 center。

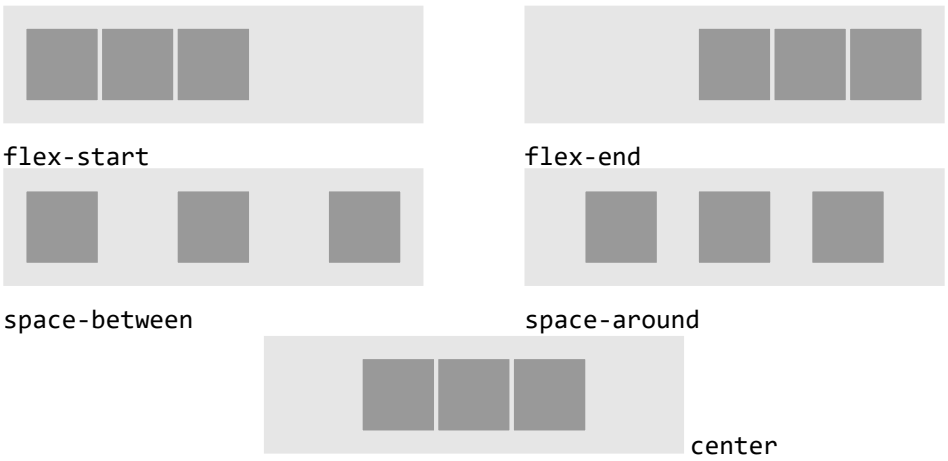


图 4.6 flex-direction

容器副轴对齐

`align-items` 用于控制副轴上的位置，默认值为 `stretch`。值和 `justify-content` 类似，`stretch` 会拉伸元素高度，`baseline` 则以字体的下划线位置对齐。

开启换行后多行元素对齐

`align-content` 需要开启 `flex-wrap` 能使用，将每一行多个元素当作单个元素看，效果就是副轴上的 `justify-content` 和 `text-align` 结合。

弹性元素的属性

除了前面提到的 `flex`，弹性元素属性只有两个:

- `align-self`: 用于控制单个元素的 `align-item` 对齐方案，值和 `align-item` 一致。
- `Order`: 用于改变 HTML 中元素的顺序，不是很推荐使用。

5 网格布局

CSS 网格可以定义由行和列组成的二维布局，然后将元素放置到网格中。有些元素可能只占据网格的一个单元，另一些元素则可能占据多行或多列。网格的大小既可以精确定义，也可以根据自身内容自动计算。你既可以将元素精确地放置到网格某个位置，也可以让其在网格内自动定位，填充划分好的区域。

5.1 网格布局基础

网格布局和弹性布局类似，是两级 DOM 结构。设置为 `display: grid` 的元素成为一个网格容器。子元素成为网格元素。网格布局有如下概念:

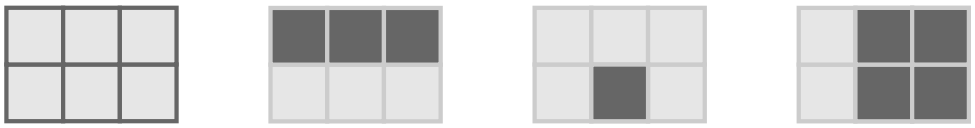


图 5.1 网络的组成部分

网格容器有以下三个主要属性:

```
.grid {
    display: grid;
    grid-template-column: 1fr 1fr 1fr; /* 三列 */
    grid-template-rows: 1fr 1fr;     /* 两行 */
    grid-gap: 0.5em;                 /* 间隔 */
}
```

其中 `grid-template-column` 和 `grid-template-rows` 的数值和 `flex-grow` 类似，是用于计算权重的因子。

如果需要声明很多行/列，可以使用 `repeat()` 函数，接收两个参数，第一个为重复次数，第二个为重复的内容，例如 `repeat(3, 2fr 1fr)` 等价于 `2fr 1fr 2fr 1fr 2fr 1fr`。

网格元素有两个相关的属性:

```
.element {
    grid-column: 1 / 3; /* 1号竖网格线到3号竖网格线 */
    grid-row: span 1;  /* 占据第一条网格轨道 */
}
```

这两个属性有以上两种写法 `num/num` 表示占据哪两条网格线之间的网格，`span num` 则表示占据第几网格轨道。

具体的网格编号如下所示:

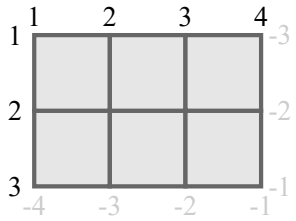


图 5.2 网格编号

与 Flexbox 配合

网格布局与弹性布局并不冲突,这两种布局方案几乎是一起开发出来的,不存在替代关系。Flexbox 本质上是一维布局，而 Grid 是二维布局。往往在网格中的部分网络区域使用 Flexbox 进行微操。

虽然 Flexbox 可以通过 `flex-wrap` 让元素换行进而达到伪二维的效果，但是这存在诸多弊端。一，`flex-wrap` 并不是为了二位布局而被设计出来的，而是为了兼容不同的设备与屏幕。二，`flex-wrap` 中换行的元素并不能达到很好的对齐效果，换行的元素往往是凌乱的。

网格布局时两级结构，每个网格元素都会扩展并填满整个网格区域，但是子元素不会，我们控制网格子元素时需要另行想办法控制高宽等属性，常饮用的方法就是使用 `Flexbox`。

替代语法

布局网格元素还有另外两个替代语法：命名的网格线和命名的网格区域。

有时候记录所有网格线的编号实在太麻烦了，尤其是在处理很多网格轨道时。为了能简单点，可以给网格线命名，并在布局时使用网格线的名称而不是编号:

```
grid-template-columns: [start] 2fr [center] 1fr [end];
```

这条声明定义了两列的网格，三条垂直的网格线分别叫作 start、center 和 end。之后定义网格元素在网格中的位置时，可以不用编号而是用这些名称来声明:

```
grid-column: start / center;
```

同时还可以给网格线提供多个名称:

```
    grid-template-columns: [left-start] 2fr [left-end right-start] 1fr [right-end];
```

在这条声明里，2 号网格线既叫作 `left-end` 也叫作 `right-start`，之后可以任选一个名称使用。这里还有一个彩蛋：将网格线命名为 `left-start` 和 `left-end`，就定义了一个叫作 `left` 的区域，这个区域覆盖两个网格线之间的区域。`-start` 和`-end` 后缀作为关键字，定义了两者之间的区域。如果给元素设置 `grid-column: left`，它就会跨越从 `left-start` 到 `left-end` 的区域。

此外还可以这样声明: `grid-template-columns: repeat(3, [col] 1fr 1fr)` 这样会出现三个 `col` 区域，使用 `grid-column: col 2 / span 2` (从第二个 `col` 开始跨越两个区

域) 可以定位到第二组网格列上。

命名网格区域不用计算或者命名网格线，直接用命名的网格区域将元素定位到网格中。实现这一方法需要借助网格容器的 `grid-template-areas` 属性和网格元素的 `grid-area` 属性。

```
.container {
    display: grid;
    grid-template-areas: "title title"
                         "nav  nav"
                         "main  aside1"
                         "main  aside2";
    grid-template-columns: 2fr 1fr;
    grid-template-rows: repeat(4, auto);
    grid-gap: 1.5em;
    max-width: 1080px;
    margin: 0 auto;
}
header {
    grid-area: title;
}
```

每个被命名的网格区域必须组成一个矩形。不能创造更复杂的形状。

还可以用句点（.）作为名称，这样便能空出一个网格单元。

```
grid-template-areas: "top top right"
                     "left . right"
                     "left bottom bottom";
```

网格布局共设计了三种语法：编号的网格线、命名的网格线、命名的网格区域。最后一个可能更受广大开发人员喜爱，尤其是明确知道每个网格元素的位置时，这种方式用起来更舒服。

5.2 隐式网格

当处理大量的网格元素时，挨个指定元素的位置未免太不方便。当元素是从数据库获取时，元素的个数可能是未知的。在这些情况下，以一种宽松的方式定义网格可能更合理，剩下的交给布局算法来放置网格元素。

这时需要用到隐式网格（implicit grid）。使用 `grid-template-*` 属性定义网格轨道时，创建的是显式网格（explicit grid），但是有些网格元素仍然可以放在显式轨道外面，此时会自动创建隐式轨道以扩展网格，从而包含这些元素。

如果网格元素放在声明的网格轨道之外，就会创建隐式轨道，直到包含该元素。

隐式网格轨道默认大小为 `auto`，也就是它们会扩展到能容纳网格元素内容。可以给网格容器设置 `grid-auto-columns` 和 `grid-auto-rows`，为隐式网格轨道指定一个大小（比如，`grid-auto-columns: 1fr`）。

使用隐式网络时，由于不确定元素数量，可以给 grid-template-columns 使用 auto-fill 值，代表会根据屏幕宽度自动设置列数量，配合 minmax() 函数可以有效地适配各种屏幕分辨率。如果网格元素不够填满所有网格轨道，auto-fill 就会导致一些空的网格轨道。如果不希望出现空的网格轨道，可以使用 auto-fit 关键字代替 auto-fill。它会让非空的网格轨道扩展，填满可用空间。

添加变化

熟悉 Win10 磁铁的读者肯定知道磁铁的大小可以改变，网格布局也可以改变为 2x1, 2x2 大小。网格布局提供了一个属性 grid-auto-flow，它可以控制布局算法的行为。它的初始值是 row，如果值为 column，它就会将元素优先放在网格列中，只有当一列填满了，才会移动到下一行。

但是这样存在一定的问题，无论按行还是按列，有时候大网格区域后紧跟的小网格区域会造成很多空网格单元，只需要为 grid-auto-flow 添加 dense 值即可，小元素就会“回填”大元素造成的空白区域。代价是会改变元素的排列顺序。

对齐

最后讲一下对齐，针对主轴和副轴有两种对齐属性，分别以 justify 和 align 开头，对象有所不同，如下表所示:

表 2.1 网格对其属性

属性	作用于	对齐
justify-items, align-items	网络容器	网格区域内的所有元素
justify-self, align-self	网络元素	网格区域内的单个元素
justify-content, align-content	网络容器	网格区域内的网络轨道

它们属性值和 Flexbox 类似。

6 定位和层叠上下文

6.1 定位

前面几章介绍了主流的三种文档流布局，定位不同于文档流，position 的默认值是 static 表示违背定位，如果设置了其他值，元素会被定位，也将彻底从文档流中移走。

固定定位

固定定位最好理解，给一个元素设置 position: fixed 就能将元素放在屏幕视口的任意位置。这需要搭配四种属性一起使用：top、right、bottom 和 left。这些属性的值决定了固定定位的元素与浏览器视口边缘的距离。

固定定位的一个作用是使用脚本控制要填写的表单，但点击某个按钮时将 display 设置为非 none 值以显示表单。

绝对定位

绝对定位 display: absolute 和固定定位的唯一区别是绝对定位是相对已定位的父元素块的，如果父元素未定位，则向上冒泡查找定位的元素。

相对定位

相对定位 display: relative 不同于前两个定位值，它并没有让元素离开文本流，而是让元素相对于它本来的位置进行偏移。top、right、bottom 和 left 也不再指定元素与视口边距的位置，而是元素相对位移的位置。

粘性定位

粘性定位，position: sticky 是相对定位和固定定位的结合: 正常情况下，元素会随着页面滚动，当到达屏幕的特定位置时，如果用户继续滚动，它就会“锁定”在这个位置。最常见的用例是侧边栏导航。

6.2 层叠上下文

浏览器将 HTML 解析为 DOM 的同时还创建了另一个树形结构，叫作渲染树（render tree）。它代表了每个元素的视觉样式和位置。同时还决定浏览器绘制元素的顺序。顺序很重要，因为如果元素刚好重叠，后绘制的元素就会出现在先绘制的元素前面。

浏览器会先绘制所有非定位的元素，然后绘制定位元素。默认情况下，所有的定位元素会出现在非定位元素前面。

z-index 属性的值可以是任意整数（正负都行）。拥有较高 z-index 的元素出现在拥有较低 z-index 的元素前面。拥有负数 z-index 的元素出现在静态元素后面。

给一个定位元素加上 z-index 可以创建层叠上下文。默认情况下，z-index 的值是 auto，不创建层叠上下文。一个层叠上下文包含一个元素或者由浏览器一起绘制的一组元素。其中一个元素会作为层叠上下文的根，比如给一个定位元素加上 z-index 的时候，它就变成了一个新的层叠上下文的根。所有后代元素就是这个层叠上下文的一部分。

使用层叠上下文有一个注意点，如果有两个父元素 one 和 two，他们的 z-index 值为 1，后定义的元素 (two) 会出现在先定义的元素上方 (顺序覆盖)。如果 one 中有一个元素 element 的 z-index 值为 100，他也可能显示在 two 的下方，子元素的 z-index 值只再父元素内产生层叠优先级影响，对父元素本身没有影响。

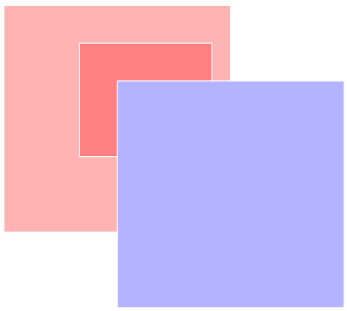


图 6.1 z-index

所有层叠上下文内的元素按照如下顺序叠放:

- 层叠上下文的根
- z-index 为负的定位元素 (及子元素)
- 非定位元素
- z-index 为 auto 的定位元素 (及子元素)
- z-index 为正的定位元素 (及子元素)

在实际开发中，一般不会直接给 z-index 设置一个具体的数值，如果 z-index 被使用多次，这会让代码很难维护，根本不知道元素，根元素的层叠顺序。更推荐统一管理 z-index，比如这种做法:

```
:root {
    --z-loading-indicator: 100;
    --z-nav -menu:        200;
    --z-dropdown-menu:    300;
}

nav {
    z-index: var(--z-nav);
}
```

7 响应式设计

这节讲的都是抽象概念，粒子请查看原书。如果只做桌面端设计也可以跳过这章。

响应式设计主要用于解决多设备屏幕分辨率不同，响应式设计的主要目的是只需要创建一个网站，就可以在智能手机、平板，或者其他任何设备上运行，它有三大原则。

- **移动优先**: 实现桌面布局之前先构建移动版的布局。
- **@media 规则**: 使用这个样式规则，可以为不同大小的视口定制样式。用这一语法，通常叫作媒体查询（media queries），写的样式只在特定条件下才会生效。
- **流式布局**: 这种方式允许容器根据视口宽度缩放尺寸。

移动优先

考虑用户使用移动端的场景，一般通过手机/平板打开网站都处于快速浏览的状态，而桌面端多处于工作/学习等状态。因此移动端需要的提供更直观简要的信息。

移动端的需求和网页设计的逻辑是相符的，网页设计也应该先完成主要的核心功能，再考虑添加其他边缘功能。

如果你失陪了小屏设备，需要在 `meta` 加上这么一句，不然浏览器会假定你的界面不是响应式的:

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

`meta` 标签的 `content` 属性里包含两个选项。首先，它告诉浏览器当解析 CSS 时将设备的宽度作为假定宽度，而不是一个全屏的桌面浏览器的宽度。其次当页面加载时，它使用 `initial-scale` 将缩放比设置为 100%。

此外 `content` 属性还有第三个选项 `user-scalable=no`，阻止用户在移动设备上用两个手指缩放。通常这个设置在实践中并不友好，不推荐使用。

媒体查询

响应式设计的第二个原则是使用媒体查询。媒体查询允许某些样式只在页面满足特定条件时才生效。这样就可以根据屏幕大小定制样式。可以针对小屏设备定义一套样式，针对中等屏幕设备定义另一套样式，针对大屏设备再定义一套样式，这样就可以让页面的内容拥有多种布局。

媒体查询使用 `@media` 规则选择满足特定条件的设备。简单的媒体查询如下代码所示:

```
@media (min-width: 560px) {
    .title > h1 {
        font-size: 2.25rem;
    }
}
```

`@media` 规则会进行条件检查，只有满足所有的条件时，才会将这些样式应用到页面上。

本例中浏览器会检查 `min-width: 560px`。只有当设备的视口宽度大于等于 `560px` 的时候，才会给标题设置 `2.25rem` 的字号。如果视口宽度小于 `560px`，那么里面的所有规则都会被忽略。

在媒体查询断点中推荐使用 em 单位。在各大主流浏览器中，当用户缩放页面或者改变默认的字号时，只有 em 单位表现一致。

可以整复杂一点，设置多个条件:

```
@media (max-width: 20em), (min-width: 35em) { ... }
```

媒体查询的所有条件如下:

- `min-height`: 高度大于等于。
- `max-height`: 高度小于等于。
- `max-width`: 宽度小于等于。
- `max-width`: 宽度小于等于。
- `orientation: landscape`: 宽度大于高度。
- `orientation: portrait`: 高度大于宽度。
- `min-resolution: 2dppx`: 匹配屏幕分辨率大于等于 2dppx（dppx 指每个 CSS 像素里包含的物理像素点数）的设备。
- `max-resolution: 2dppx`: 匹配屏幕分辨率小等于 2dppx 的设备。

比较少见的，还可以使用 `@media screen` 控制打印时的网页布局。

流式布局

流式布局，有时被称作液体布局（liquid layout），指的是使用的容器随视口宽度而变化。它跟固定布局相反，固定布局的列都是用 px 或者 em 单位定义。固定容器（比如，设定了 `width: 800px` 的元素）在小屏上会超出视口范围，导致需要水平滚动条，而流式容器会自动缩小以适应视口。

在流式布局中，主页面容器通常不会有明确宽度，也不会给百分比宽度，但可能会设置左右内边距，或者设置左右外边距为 auto，让其与视口边缘之间产生留白。也就是说容器可能比视口略窄，但永远不会比视口宽。

响应式图片

图片传输往往要消耗大量流量，在不同大小的设备下可以放置不同分辨率的图片，比如:

```
@media (min-width: 35em) {
    .hero {
        padding        : 5em 3em;
        font-size      : 1.2rem;
        background-image: url(coffee-beans-medium.jpg);
    }
}
```

```
8
9  @media (min-width: 50em) {
10     .hero {
11         padding        : 7em 6em;
12         background-image: url(coffee-beans.jpg);
13     }
14 }
```

在不同屏幕的浏览器上加载这样的网页，根本看不出有什么区别。

HTML 里的 `<img>` 标签有一个 `srcset` 属性可以根据条件指定不同的图片 URL:

```
1 <img alt="A white coffee mug on a bed of coffee beans"
2     src="coffee-beans-small.jpg"
3     srcset="coffee-beans-small.jpg 560w,
4         coffee-beans-medium.jpg 800w,
5         coffee-beans.jpg 1280w"
6 />
```

这种方式允许针对不同的屏幕尺寸优化图片。更棒的是，浏览器会针对高分辨率的屏幕做出调整。如果设备的屏幕像素密度是 2 倍，浏览器就会相应地加载更高分辨率的图片。

III 动画

8 样式

原书这几节有很多设计相关理论的讲解，如有兴趣，请查看原文。本文只记录技术相关的内容。

8.1 背景，阴影，颜色

背景

与背景相关的属性一共有八个，background 属性是他们的缩写:

- background-image: 指定一个文件或生成的颜色渐变作为背景图片。
- background-position: 设置背景图片的初始位置
- background-size: 指定元素内背景图片的渲染尺寸
- background-repeat: 决定在需要填充整个元素时，是否平铺图片。
- background-origin: 决定背景相对于元素的边框盒，内边距框盒或内容盒子来定位。
- background-clip: 指定背景是否应该填充。
- background-attachment: 指定背景元素是跟着元素上下滚动还是固定。
- background-color: 指定纯色背景，渲染到背景图片下方。

。background-image 属性可以接受一个图片 URL 路径 (background-image: url(coffee-beans.jpg)), 也可以接受一个渐变函数。

最基础的渐变函数是线性渐变: background-image: linear-gradient(to right, white, blue);，第二三个参数是颜色，可以增加颜色参数。第一个确定渐变角度，单位可以是以下:

- deg: 角度
- rad: 弧度
- turn: 代表环绕圆周的圈数，0.25turn 相当于 90deg。
- grad: 百分度。一个完整的圆是 400 百分度（400grad），100grad 相当于 90deg。

在颜色参数后可以加上位置，比如:

```
background-image: linear-gradient(90deg,red 40\%, white 40\%,white 60\%, blue 60\%);
```

表示在 40%, 60% 处开始渐变，上面代码会生成法国国旗。

还可以生成重复的渐变:

```
background-image: repeating-linear-gradient(-45deg,#57b, #57b 10px, #148 10px, #148 20px);
```

上述渐变会填充满整个容器。

类似的，还有镜像渐变: `radial-gradient()`，根据参数个数不同产生不同的效果，请自行查阅具体功能。

阴影

CSS 提供了两种阴影:

- `text-shadow`: 文字阴影。
- `box-shadow`: 盒子阴影。

最完整的语法包含六个参数: `box-shadow: insert, offset-x, offset-y, radius, spread-radius, color`。`insert` 表示内阴影, `radius` 表示扩展半径，`spread-radius` 表示对扩展半径的缩放，这两个组合对阴影进行扩展。

现在主流的阴影设计方案是扁平化，low-poly 风格。

混合模式

CSS 支持 15 种混合模式，每一种都使用不同的计算原理来控制生成最终的混合结果。这个用的比较少，如果由 PS 等图像处理技术基础，很好理解。

颜色

颜色有多种表示方法，对应 CSS 有多个处理颜色的方法:

- 名称: CSS 为我们定义了常用的颜色，可以直接通过名称获取，如 `white, black`。
- 十六进制: #000000 代表纯黑，#000 效果相同，后者能现实的颜色精度略低，三个十六进制数依次拆分成三组，对应 rgb 的颜色。六个则是 32 位颜色。
- `rgba()`: 最常见的 `rgb(0,0,0)` 代表黑色 (等效于: #000)，`rgba()` 多出了一个 `alpha` 通道，表示不透明度。
- `hsl()`: 同样接受三个值，分别代表色相，饱和度，明度；设计师用的比较多，熟悉颜色关系推荐使用这个。

在实际应用中，更好的方式是全局定义颜色，然后通过变量名获取颜色，这样能保证颜色统一，也利于维护。

8.2 字体与段落

字体相关的常用属性有以下几个:

- `font-family`: 字体体系，一般是一套字体。
- `font-style`: 字体类型，斜体，黑体，正文等。
- `font-weight`: 字体粗细。

- `font-variant`: 字体异体。
- `font-size`: 字体大小。
- `line-height`: 字体行高，默认为 1.2em。
- `letter-spacing`: 字符间距，一般以 1/100em 调整。

9 过渡与变换

9.1 过渡

过渡是通过一系列 `transition-*` 属性来实现的。如果某个元素设置了过渡，那么当它的属性值发生变化时，并不是直接变成新值，而是使用过渡效果。`transition` 是该系列属性的简写，包括如下主要属性:

- `transition-property`: 过度对象，值为 CSS 属性。
- `transition-duration`: 过度经历的时间。
- `transition-timing-function`: 过度速度曲线。
- `transition-delay`: 过渡动画延迟时间。

使用 `transition` 比较简单，也是常用做法，比如在 `:hover` 伪类选择器上写入:

```
color: hsl(180,0.5,0.5);
transition: color .5s;
```

就会在 0.5s 内改变 `color` 属性的值。也可以单独设置每个属性，`transition-property` 可以设置为 `all`，代表改变所有属性。

`transition-timing-function` 有以下几个常见值:

- `liner`: 线性变化。
- `ease`: 慢-快-满
- `ease-in`: 慢-快
- `ease-out`: 快-满
- `ease-in-out`: 前两者组合

这个值本质上是调用贝塞尔函数: `cubic-bezier(n1,n2,n3,n4)`，这四个值依次分为两队，代表了两个坐标 `(n1,n2)`, `(n3,n4)`。如果有类似 Illustrator 矢量作图软件经验，就知道这两个坐标决定了曲线控制柄的位置，贝塞尔曲线随之变换。

还有一种离散过度函数: `steps()`，他会让元素突然过度，没有渐变效果，比如 `width` 属性从 100 到 200，会在相同时间间隔内经历这几个值变化 100,125,150,175,200。`step()` 有两个参数，阶跃次数和阶跃位置，次数即变化几次，位置只有两个值 `start,end`，即什么时候发生节约。这个比较抽象，建议实战看看就知道了。

并非所有的属性都可以过渡，比如 `display`；也并不是所有属性的任意值都可以过渡，比如只有 `background-color` 仅有一个值时可以过度。一般值为数值，颜色，可以使用 `calc()` 计算的的属性值可以过度，而非连续性属性不可以过度。

有两个属性可以替代 `display: none` 的效果；一个是 `opacity`，代表不透明度；另一个是 `visibility` 有两个值 `visible, hidden`。不过它们最多做到看不到元素，但其实还存在于 DOM 树中。

9.2 变换

变换通过 transform 属性实现。它的值为变换函数，可以有一个或多个值，常用的如下:

- rotate(deg): 顺时针旋转。
- translate(x,y): 平面位移。
- scale(scale): 缩放。
- skew(deg): 倾斜。

使用 transform-origin 属性可以改变基点，默认自然是盒模型中心。

CSS 还支持以上四类函数的变种: 三维版。三维的三个轴与屏幕的对应关系为:

- x 轴: 屏幕宽度方向。
- y 轴: 屏幕高度方向。
- z 轴: 屏幕内方向，即人眼的方向。

三维属性命名也十分简单，例如 rotate 的三维版: rotateX，rotateY，rotateZ。此外还有一个 perspective() 函数，表示透视距离，越小效果越明显。视距也可以设置基点，对应属性为 perspective-origin。

此外还有几个常用属性:

- backface-visibility: 当旋转到一定角度后，我们可能看不到元素正面，如果想穿透显示背面元素，使用该属性。
- transform-style（preserve-3d）: 看 3D？没用过。

过渡与变换的区别在于发生在时间段还是时间刻: 过渡是一个过程，需要一段时间去完成, 而变换只是一个状态，它和常规样式属性没有本质区别。

10 动画

CSS 中的动画包括两部分：用来定义动画的 @keyframes 规则和为元素添加动画的 animation 属性。

关键帧定义了某个时间段内，不同时间刻的状态，具体的状态使用变换以及常规属性指定；至于具体的时间以及时间变换函数由调用关键帧的属性给出。关键帧语法如下:

```
@keyframes over-and-back {
    0% {
      background-color: hsl(0, 50%, 50%);
      transform: translate(0);
    }

    50% {
      transform: translate(50px);
    }

    100% {
      background-color: hsl(270, 50%, 90%);
      transform: translate(0);
    }
}
```

调用关键帧需要 `animation` 属性，这个属性的具体用法和 `transition` 类似 (毕竟都是动态的)。

```
animation: over-and -back 1.5s linear
```

`animation` 也是一个简写，它的值依次对应如下属性:

- `animation-name`: 动画名称，对应关键帧名。
- `animation-duration`: 动画持续的时间。
- `animation-timing-function`: 定时函数，前面讲过。
- `animation-`: 动画重复次数。

主要的动画技术就这些，动画需要实战经验，比较主流的动画框架是 `tree.js`。