

前言

曾经简单的交互都需使用JS才能完成，经历过 **jQuery时代** 的同学应该很清楚，使用原生JS写交互很艰难，但是使用 **jQuery** 封装好的交互函数那就很简单了。

如今CSS3增加了 **transform**、**transition** 和 **animation** 三大交互属性，为CSS的单调性增加了很多趣味，也为交互开发增加了新的可能。

变换

变换分为 **2D变换** 和 **3D变换**。2D变换在平面上操作，3D变换在空间上操作，2D和3D的概念相信很多同学都会了吧。变换可理解成将节点复制一份并生成新的图层，原节点隐藏，使用新节点进行变换操作。

声明 **transform-style** 可实现 **2D变换** 和 **3D变换** 间的切换，不同变换空间需使用对应的变换函数。当然 **transform-style** 需声明在父节点中，即需发生变换的节点的父节点。

- ☑ **flat**：所有变换效果在平面上呈现(**默认**)
- ☑ **preserve-3d**：所有变换效果在空间上呈现

笔者已将 **2D变换函数** 和 **3D变换函数** 整理好，在不同变换空间使用对应的变换函数即可。

- ☑ **translate()**：位移
 - ☑ **translate(x,y)**：2D位移
 - ☑ **translate3d(x,y,z)**：3D位移
 - ☑ **translateX(x)**：X轴位移，等同于 **translate(x,0)** 或 **translate3d(x,0,0)**
 - ☑ **translateY(y)**：Y轴位移，等同于 **translate(0,y)** 或 **translate3d(0,y,0)**
 - ☑ **translateZ(z)**：Z轴位移，等同于 **translate3d(0,0,z)**
 - 描述
 - 单位：**Length** 长度，可用任何长度单位，允许负值
 - 默认：XYZ轴不声明默认是 **0**
 - 正值：沿X轴向右位移/沿Y轴向上位移/沿Z轴向外位移
 - 负值：沿X轴向左位移/沿Y轴向下位移/沿Z轴向内位移
- ☑ **scale()**：缩放
 - ☑ **scale(x,y)**：2D缩放
 - ☑ **scale3d(x,y,z)**：3D缩放
 - ☑ **scaleX(x)**：X轴缩放，等同于 **scale(x,1)** 或 **scale3d(x,1,1)**
 - ☑ **scaleY(y)**：Y轴缩放，等同于 **scale(1,y)** 或 **scale3d(1,y,1)**
 - ☑ **scaleZ(z)**：Z轴缩放，等同于 **scale3d(1,1,z)**
 - 描述

- 单位: **Number** 数值或 **Percentage** 百分比, 允许负值
- 默认: XYZ轴不声明默认是 **1** 或 **100%**
- 正值: $0 < (x, y, z) < 1$ 沿X轴缩小/沿Y轴缩小/沿Z轴变厚, $(x, y, z) > 1$ 沿X轴放大/沿Y轴放大/沿Z轴变薄
- 负值: $-1 < (x, y, z) < 0$ 翻转沿X轴缩小/沿Y轴缩小/沿Z轴变厚, $(x, y, z) < -1$ 翻转沿X轴放大/沿Y轴放大/沿Z轴变薄

✓ **skew()**: 扭曲

- ✓ **skew(x,y)**: 2D扭曲
- ✓ **skewX(x)**: X轴扭曲, 等同于 **skew(x,0)**
- ✓ **skewY(y)**: Y轴扭曲, 等同于 **skew(0,y)**

◦ 描述

- 单位: **Angle** 角度或 **Turn** 周
- 默认: XY轴不声明默认是 **0**
- 正值: 沿X轴向左扭曲/沿Y轴向下扭曲
- 负值: 沿X轴向右扭曲/沿Y轴向上扭曲

✓ **rotate()**: 旋转

- ✓ **rotate()**: 2D旋转
- ✓ **rotate3d(x,y,z,a)**: 3D旋转, **[x,y,z]** 是一个向量, 数值都是 **0~1**
- ✓ **rotateX(a)**: X轴旋转, 等同于 **rotate(1,0,0,a)**, 正值时沿X轴向上逆时针旋转, 负值时沿X轴向下顺时针旋转
- ✓ **rotateY(a)**: 3D Y轴旋转, 等同于 **rotate(0,1,0,a)**, 正值时沿Y轴向右逆时针旋转, 负值时沿Y轴向左顺时针旋转
- ✓ **rotateZ(a)**: 3D Z轴旋转, 等同于 **rotate(0,0,1,a)**, 正值时沿Z轴顺时针旋转, 负值时沿Z轴逆时针旋转

◦ 描述

- 单位: **Angle** 角度或 **Turn** 周
- 正值: 2D旋转时顺时针旋转
- 负值: 2D旋转时逆时针旋转

✓ **matrix()**: 矩阵(太过复杂, 可放弃)

- ✓ **matrix(a,b,c,d,e,f)**: 2D矩阵(位移、缩放、扭曲、旋转的综合函数)
- ✓ **matrix(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)**: 3D矩阵(位移、缩放、扭曲、旋转的综合函数)

✓ **perspective()**: 视距

- **Length**: 长度, 可用任何长度单位

transform 的使用场景很多, 不局限于某种特定场景, 若结合 **transition** 和 **animation** 使用还必须注意性能问题。

多值执行顺序

与 `background` 和 `mask` 一致可声明多重效果，使用 `逗号` 隔开。网上很多结论说 `transform` 多值执行顺序是 `从左到右` 或 `从右到左`，其实这样的结论都是比较笼统的。正确来说并无执行上的先后顺序，而是由多个变换对应的矩阵相乘，再拿该矩阵去乘以坐标，最终得出变换效果。

例如 `transform:translate(150px,0),rotate(45deg)` 和 `transform:rotate(45deg),translate(150px,0)`，最终的变换效果就有所不同。

- 第一种：先往右位移 `150px`，坐标轴不变；再顺时针旋转 `45deg`，坐标轴顺时针旋转 `45deg`
- 第二种：先顺时针旋转 `45deg`，坐标轴顺时针旋转 `45deg`；再往右位移 `150px`，坐标轴不变

```
.elem {  
  display: flex;  
  justify-content: center;  
  align-items: center;  
  position: absolute;  
  left: 50px;  
  top: 50px;  
  width: 100px;
```

```
height: 100px;
background-color: #f66;
font-size: 20px;
color: #fff;
&.transform-1 {
    transform: translate(150px, 0) rotate(45deg);
}
&.transform-2 {
    transform: rotate(45deg) translate(150px, 0);
}
}
```

正确的理解是 **从左到右**，但是还得注意坐标轴是否发生了变化，若坐标轴发生了变化，会影响到后续的变换效果。优先考虑坐标轴的变化，先分析出前后 **缩放旋转** 的变化，再分析出前后 **位移扭曲** 的变化。

缩放和旋转都能让坐标轴发生变化，这个必须谨记

视距效果

transform: perspective() 也可通过 **perspective** 声明，这个属性在开启3D变换后最好声明上，否则有些3D变换效果可能无法得到更好的展现。

- 值越小，用户与空间Z轴距离越近，视觉效果越强
- 值越大，用户与空间Z轴距离越远，视觉效果越弱

perspective/perspective()区别

perspective 和 **transform: perspective()** 都能声明视距，那为何要存在两种声明方式呢？当然是有它们的区别所以才能存在呀。

- **perspective** 与 **transform: perspective()** 的作用相同
- **perspective** 在 **舞台节点** (变换节点的父节点)上使用，**transform: perspective()** 在 **当前变换节点** 上使用，也可与其他变换函数一起使用

GPU硬件加速模式

有无发现即使很简单的动画，有时都能引起卡顿，特别是在移动端上尤其明显。在此介绍一种Hack方法，为节点声明 `transform:transition3d()` 或 `transform:translateZ()`，这两个声明都会开启**GPU硬件加速模式**，从而让浏览器在渲染动画时从CPU转向GPU，实现硬件加速。

`transform:transition3d()` 和 `transform:translateZ()` 其实是为了渲染3D样式，但声明为 `0` 后并无真正使用3D效果，但浏览器却因此开启了GPU硬件加速模式。在 **Webkit内核** 下使用 `transform:translate3d()` 加速效果会更明显。

```
.elem {  
    transform: transition3d(0, 0, 0);  
}  
/* 或 */  
.elem {  
    transform: translateZ(0);  
}
```

在使用该方案时可能会出现诡异的缺陷。当有多个绝对定位的节点声明 `transform:transition3d()` 开启GPU硬件加速模式后会有几个节点凭空消失，是不是很诡异。这种现象不能完全解决，只能尽量避免。

- 尽量不要对节点及其父节点声明 `position:absolute/fixed`，当然这个很难避免不使用
- 减少声明 `transform:transition3d()` 的节点数量，减少至6个以下即可
- 声明 `will-change` 代替 `transform:transition3d()`，详情请戳[这里](#)

笔者比较推荐第二种方法，节点的数量可通过JS动态控制，保持在6个以下。而 `will-change` 会存在另一些问题，大量使用还是会引发更严重的性能问题，笔者后续会在本章更新详细的分析。

动感心形

`transform` 有一个很实用的场景，就是通过 `transform:translate()` 补位。补位指实现效果的最终位置还差一点距离就能完成，通过 `margin` 或 `transform:translate()` 将该距离补充完整，将节点调整到最终位置。

还记得第6章布局方式的居中布局吗？有一种方式就是通过 `transform:translate(-50%,-50%)` 将节点拉回最中央，节点无需声明位移的距离是宽高的二分之一，使用 `50%` 自动计算其距离为宽高的二分之一即可。

描绘一个心形虽然不是一个很常用的场景，作为一名雄性程序猿，`214` 和 `520` 等具有示爱性质的节日，当然少不了用纯CSS描绘一个动感心形啦。

使用单个 `<div>` 结合两个伪元素 `::before` 和 `::after` 通过错位叠加的方式合并成一个心形。



- 声明 `<div>` 的尺寸为一个 正方形 并以中心顺时针旋转 `45deg`
- 声明两个伪元素继承 `<div>` 的尺寸并实行绝对定位
- 声明两个伪元素的圆角率为 `100%` 并平移到相应位置

巧妙利用了 `transform` 将两个伪元素平移到相应位置产生叠加错觉。



```
<div class="heart-shape"></div>
```

```
.heart-shape {  
  position: relative;  
  width: 200px;  
  height: 200px;  
  background-color: #f66;  
  transform: rotate(45deg);  
  &::before,  
  &::after {  
    position: absolute;  
    left: 0;  
    top: 0;  
    border-radius: 100%;  
    width: 100%;  
    height: 100%;  
    background-color: #f66;  
    content: "";  
  }  
  &::before {  
    transform: translateX(-50%);  
  }  
  &::after {  
    transform: translateY(-50%);  
  }  
}
```

☑ 在线演示: [Here](#)

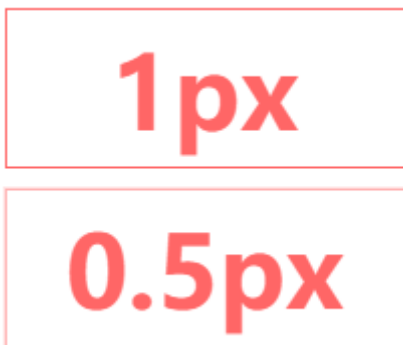
☑ 在线源码: [Here](#)

像素边框

1px边框 在桌面端网站看上去没什么大问题，但在移动端网站看上去却觉得很粗。由于大部分移动端都具有细腻的屏幕，像iPhone的 **Retina屏幕**，一个像素可由4个点或9个点组成，在接近视网膜极限的情况下，**1px边框** 看起来确实会有点粗。

那么可声明 **0.5px边框** 吗。答案是可行的，即使声明成功，但有些浏览器还是按照 **1px** 的值去渲染，这样就导致不同设备的边框参差不齐了。

换个思路，使用一个伪元素的边框去当作节点边框，声明 **border** 为 **1px** 并将其宽高声明成 **200%**，最终效果是该节点的2倍大小，再通过声明 **transform:scale(.5)** 将该伪元素缩小到原来的 **0.5倍**，现在和节点尺寸一致了，而 **border** 也通过浏览器自动计算成 **0.5px** 了，最终实现 **0.5px边框**。其实现原理就是将边框宽度计算交由浏览器处理。



```
<div class="onepx-border normal">1px</div>
<div class="onepx-border thin">0.5px</div>
```

```
.onepx-border {
  width: 200px;
  height: 80px;
  cursor: pointer;
  line-height: 80px;
  text-align: center;
  font-weight: bold;
  font-size: 50px;
  color: #f66;
  & + .onepx-border {
    margin-top: 10px;
  }
  &.normal {
    border: 1px solid #f66;
```



```
}  
&.thin {  
  position: relative;  
  &::after {  
    position: absolute;  
    left: 0;  
    top: 0;  
    border: 1px solid #f66;  
    width: 200%;  
    height: 200%;  
    content: "";  
    transform: scale(.5);  
    transform-origin: left top;  
  }  
}
```

☒ 在线演示: [Here](#)

☒ 在线源码: [Here](#)

内容翻转

遇到一些内容翻转的场景, 有些同学可能会声明 `transform: rotate3d()` 将内容沿着Y轴旋转 `180deg` 水平翻转。

其实可声明 `transform: scale()` 为负值将内容直接翻转, 细心的同学应该注意到上述有谈到。

- 水平翻转: `transform: scale(1,-1)`
- 垂直翻转: `transform: scale(-1,1)`
- 倒序翻转: `transform: scale(-1,-1)`

正常文本

水平翻转

垂直翻转

倒序翻转

```
<ul class="flip-content">
  <li>正常文本</li>
  <li class="x-axis">水平翻转</li>
  <li class="y-axis">垂直翻转</li>
  <li class="reverse">倒序翻转</li>
</ul>
```

```
.flip-content {
  li {
    position: relative;
    width: 121px;
    height: 51px;
    line-height: 51px;
    text-align: center;
    font-weight: bold;
    font-size: 30px;
    color: #f66;
    &::before,
    &::after {
      position: absolute;
      background-color: #66f;
      content: "";
    }
    & + li {
      margin-top: 10px;
    }
  }
}
```

```
&.x-axis {
  transform: scale(1, -1);
  &::after {
    left: 0;
    top: 25px;
    width: 100%;
    height: 1px;
  }
}

&.y-axis {
  transform: scale(-1, 1);
  &::after {
    left: 60px;
    top: 0;
    width: 1px;
    height: 100%;
  }
}

&.reverse {
  transform: scale(-1, -1);
  &::before {
    left: 0;
    top: 25px;
    width: 100%;
    height: 1px;
  }
  &::after {
    left: 60px;
    top: 0;
    width: 1px;
    height: 100%;
  }
}
}
```

☑ 在线源码: [Here](#)

过渡

有时在不同状态间切换属性可能会显得很生硬, 此时 `transition` 就派上用场了, 它能让状态间的切换变得更丝滑。

☑ **transition-property**: 属性

- `all`: 全部属性过渡(默认)
- `none`: 无属性过渡
- `String`: 某个属性过渡

☑ **transition-duration**: 时间

- `Time`: 秒或毫秒(默认 0)

☑ **transition-timing-function**: 缓动函数

- `ease`: 逐渐变慢, 等同于 `cubic-bezier(.25,.1,.25,1)` (默认)
- `linear`: 匀速, 等同于 `cubic-bezier(0,0,1,1)`
- `ease-in`: 加速, 等同于 `cubic-bezier(.42,0,1,1)`
- `ease-out`: 减速, 等同于 `cubic-bezier(0,0,.58,1)`
- `ease-in-out`: 先加速后减速, 等同于 `cubic-bezier(.42,0,.58,1)`
- `cubic-bezier`: 贝塞尔曲线, `(x1,y1,x2,y2)` 四个值特定于曲线上的点 P1 和 P2, 所有值需在 `[0,1]` 区域内

☑ **transition-delay**: 时延

- `Time`: 秒或毫秒(默认 0)

总体来说, `transition` 可用到所有可能发生属性变更的节点上, 但有一些情况是绝对不能使用的。





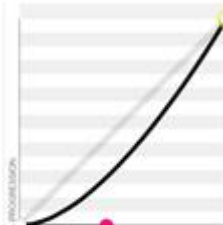
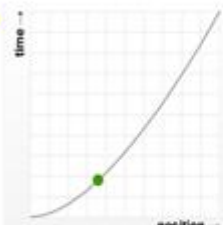
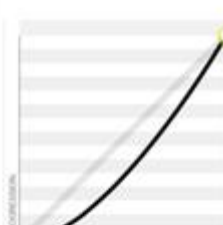
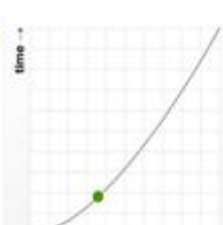
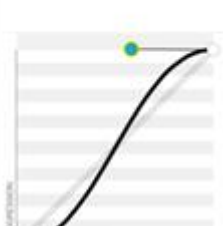

`transition` 延缓某些属性的变更过程, 若通过 `鼠标事件` 给某个节点属性赋值, 会导致属性在变更过程中发生卡顿。

例如通过鼠标的 `mousemove` 事件将 `top` 从 `10px` 变更到 `20px`。由于声明了 `transition:300ms`, 那么从 `10px` 变更到 `11px` 时会将该过程延缓 `300ms`, 导致了该过程的执行时长是 `300ms`, 而此刻想要的效果是瞬间从 `10px` 变更到 `11px`, 再依次变更到 `20px`。整个过程是鼠标移动就立刻赋值, 这样才能实时显示 `top` 的变化, 而声明了 `transition` 反而起到副作用导致看上去很卡顿。

由于 `duration` 和 `delay` 的取值都是时间, 所以可能会发生混淆。

- `duration` 和 `delay` 作用于所有节点, 包括自身的 `::before` 和 `::after`
- `transition` 中出现两个时间值时, 第一个解析为 `duration`, 第二个解析为 `delay`
- `transition` 中出现一个时间值时, 解析为 `duration`

缓动函数其实就是贝塞尔曲线，相关原理可自行百度。推荐一个设置缓动函数形状的网站[CubicBezier](https://cubic-bezier.com/)，可根据需求设置想要的缓动函数。

函数	功能描述	图例																								
ease	默认值，元素样式从初始状态过渡到终止状态时速度由快到慢，逐渐变慢	  <table><thead><tr><th>Time (s)</th><th>Position</th></tr></thead><tbody><tr><td>0.000</td><td>0%</td></tr><tr><td>0.500</td><td>9%</td></tr><tr><td>1.000</td><td>29%</td></tr><tr><td>1.500</td><td>51%</td></tr><tr><td>2.000</td><td>69%</td></tr><tr><td>2.500</td><td>81%</td></tr><tr><td>3.000</td><td>89%</td></tr><tr><td>3.500</td><td>94%</td></tr><tr><td>4.000</td><td>98%</td></tr><tr><td>4.500</td><td>99%</td></tr><tr><td>5.000</td><td>100%</td></tr></tbody></table>	Time (s)	Position	0.000	0%	0.500	9%	1.000	29%	1.500	51%	2.000	69%	2.500	81%	3.000	89%	3.500	94%	4.000	98%	4.500	99%	5.000	100%
Time (s)	Position																									
0.000	0%																									
0.500	9%																									
1.000	29%																									
1.500	51%																									
2.000	69%																									
2.500	81%																									
3.000	89%																									
3.500	94%																									
4.000	98%																									
4.500	99%																									
5.000	100%																									
linear	元素样式从初始状态过渡到终止状态速度是恒速	  <table><thead><tr><th>Time (s)</th><th>Position</th></tr></thead><tbody><tr><td>0.000</td><td>0%</td></tr><tr><td>0.500</td><td>10%</td></tr><tr><td>1.000</td><td>20%</td></tr><tr><td>1.500</td><td>30%</td></tr><tr><td>2.000</td><td>40%</td></tr><tr><td>2.500</td><td>50%</td></tr><tr><td>3.000</td><td>60%</td></tr><tr><td>3.500</td><td>70%</td></tr><tr><td>4.000</td><td>80%</td></tr><tr><td>4.500</td><td>90%</td></tr><tr><td>5.000</td><td>100%</td></tr></tbody></table>	Time (s)	Position	0.000	0%	0.500	10%	1.000	20%	1.500	30%	2.000	40%	2.500	50%	3.000	60%	3.500	70%	4.000	80%	4.500	90%	5.000	100%
Time (s)	Position																									
0.000	0%																									
0.500	10%																									
1.000	20%																									
1.500	30%																									
2.000	40%																									
2.500	50%																									
3.000	60%																									
3.500	70%																									
4.000	80%																									
4.500	90%																									
5.000	100%																									
ease-in	元素样式从初始状态过渡到终止状态时，速度越来越快，呈一种加速状态。常称这种效果为渐显效果	  <table><thead><tr><th>Time (s)</th><th>Position</th></tr></thead><tbody><tr><td>0.000</td><td>0%</td></tr><tr><td>0.500</td><td>2%</td></tr><tr><td>1.000</td><td>6%</td></tr><tr><td>1.500</td><td>13%</td></tr><tr><td>2.000</td><td>21%</td></tr><tr><td>2.500</td><td>32%</td></tr><tr><td>3.000</td><td>43%</td></tr><tr><td>3.500</td><td>55%</td></tr><tr><td>4.000</td><td>69%</td></tr><tr><td>4.500</td><td>84%</td></tr><tr><td>5.000</td><td>100%</td></tr></tbody></table>	Time (s)	Position	0.000	0%	0.500	2%	1.000	6%	1.500	13%	2.000	21%	2.500	32%	3.000	43%	3.500	55%	4.000	69%	4.500	84%	5.000	100%
Time (s)	Position																									
0.000	0%																									
0.500	2%																									
1.000	6%																									
1.500	13%																									
2.000	21%																									
2.500	32%																									
3.000	43%																									
3.500	55%																									
4.000	69%																									
4.500	84%																									
5.000	100%																									
ease-out	元素样式从初始状态过渡到终止状态时，速度越来越慢，呈一种减速状态。常称这种效果为渐隐效果	  <table><thead><tr><th>Time (s)</th><th>Position</th></tr></thead><tbody><tr><td>0.000</td><td>0%</td></tr><tr><td>0.500</td><td>2%</td></tr><tr><td>1.000</td><td>6%</td></tr><tr><td>1.500</td><td>13%</td></tr><tr><td>2.000</td><td>21%</td></tr><tr><td>2.500</td><td>32%</td></tr><tr><td>3.000</td><td>43%</td></tr><tr><td>3.500</td><td>55%</td></tr><tr><td>4.000</td><td>69%</td></tr><tr><td>4.500</td><td>84%</td></tr><tr><td>5.000</td><td>100%</td></tr></tbody></table>	Time (s)	Position	0.000	0%	0.500	2%	1.000	6%	1.500	13%	2.000	21%	2.500	32%	3.000	43%	3.500	55%	4.000	69%	4.500	84%	5.000	100%
Time (s)	Position																									
0.000	0%																									
0.500	2%																									
1.000	6%																									
1.500	13%																									
2.000	21%																									
2.500	32%																									
3.000	43%																									
3.500	55%																									
4.000	69%																									
4.500	84%																									
5.000	100%																									
ease-in-out	元素样式从初始状态到终止状态时，先加速再减速。常称这种效果为渐显渐隐效果	  <table><thead><tr><th>Time (s)</th><th>Position</th></tr></thead><tbody><tr><td>0.000</td><td>0%</td></tr><tr><td>0.500</td><td>2%</td></tr><tr><td>1.000</td><td>8%</td></tr><tr><td>1.500</td><td>19%</td></tr><tr><td>2.000</td><td>33%</td></tr><tr><td>2.500</td><td>50%</td></tr><tr><td>3.000</td><td>67%</td></tr><tr><td>3.500</td><td>81%</td></tr><tr><td>4.000</td><td>92%</td></tr><tr><td>4.500</td><td>98%</td></tr><tr><td>5.000</td><td>100%</td></tr></tbody></table>	Time (s)	Position	0.000	0%	0.500	2%	1.000	8%	1.500	19%	2.000	33%	2.500	50%	3.000	67%	3.500	81%	4.000	92%	4.500	98%	5.000	100%
Time (s)	Position																									
0.000	0%																									
0.500	2%																									
1.000	8%																									
1.500	19%																									
2.000	33%																									
2.500	50%																									
3.000	67%																									
3.500	81%																									
4.000	92%																									
4.500	98%																									
5.000	100%																									

还记得第9章选择器的切换按钮的刹车动画吗？点击按钮后，圆点从左到右有一个细微的刹车动画，这个不是通过JS捣鼓出来的，而是笔者细心地调制了一个缓动函数`cubic-bezier(.4,.4,.25,1.35)`实现的。具体实现可通过笔者推荐的网站自行调制喔。



动画

上述 `transform` 能让节点拥有更多形态，而 `animation` 能让节点拥有更多状态。正是有了 `animation`，所以才让交互效果更精彩。

CSS动画可通过设置多个点精确控制一个或一组动画，用来实现复杂的动画效果。

动画由多个点组成，每个点拥有独立的状态，这些状态通过浏览器处理成过渡效果，点与点间的过渡效果串联起来就是一个完整的动画。

`animation` 可声明的两种动画，每种动画各有自身特点。

- ☑ **关键帧动画**：在时间轴的关键帧上绘制关键状态并使之有效过渡组成动画
- ☑ **逐帧动画**：在时间轴的每一帧上绘制不同内容并使之连续播放组成动画

关键帧动画 可看作是一个连续的动画片段，**逐帧动画** 可看作是一个断续的动画片段，两种动画都是通过时间流逝将多个动画片段串联在一起。浏览器可将关键帧动画的关键帧自动过渡成片段，而将逐帧动画的每一帧按顺序播放成片段，可认为逐帧动画是一个 **GIF**。

- ☑ **animation-name**：名称
 - `none`：无动画(默认)
 - `String`：动画名称
- ☑ **animation-duration**：时间
 - `Time`：秒或毫秒(默认 0)
- ☑ **animation-timing-function**：缓动函数
 - `ease`：逐渐变慢，等同于 `cubic-bezier(.25,.1,.25,1)` (默认)
 - `linear`：匀速，等同于 `cubic-bezier(0,0,1,1)`
 - `ease-in`：加速，等同于 `cubic-bezier(.42,0,1,1)`
 - `ease-out`：减速，等同于 `cubic-bezier(0,0,.58,1)`
 - `ease-in-out`：先加速后减速，等同于 `cubic-bezier(.42,0,.58,1)`
 - `cubic-bezier`：贝塞尔曲线，`(x1,y1,x2,y2)` 四个值特定于曲线上的点 **P1** 和 **P2**，所有值需在 `[0,1]` 区域内
 - `steps([, [start|end]]?)`：把动画平均划分成 **n等分**，直到平均走完该动画
 - `step-start`：等同于 `steps(1,start)`，把动画分成一步，动画执行时以左侧端点 **0%** 为开始

- `step-end` : 等同于 `steps(1,end)` , 把动画分成一步, 动画执行时以右侧端点 `100%` 为开始
- ☑ `animation-delay`: 时延
 - `Time` : 秒或毫秒(默认 `0`)
- ☑ `animation-iteration-count`: 播放次数
 - `Number` : 数值(默认 `1`)
 - `infinite` : 无限次
- ☑ `animation-direction`: 轮流反向播放(播放次数为一次则该属性无效果)
 - `normal` : 正常播放(默认)
 - `alternate` : 轮流反向播放, 奇数次数正常播放, 偶数次数反向播放
- ☑ `animation-play-state`: 播放状态
 - `running` : 正在播放(默认)
 - `paused` : 暂停播放
- ☑ `animation-fill-mode`: 播放前后其效果是否可见
 - `none` : 不改变默认行为(默认)
 - `backwards` : 在时延所指定时间内或在动画开始前应用开始属性(在第一个关键帧中定义)
 - `forwards` : 在动画结束后保持最后一个属性(在最后一个关键帧中定义)
 - `both` : 向前和向后填充模式都被应用

关键帧动画必须通过 `animation` 和 `@keyframes` 声明, 逐帧动画只能通过 `animation-timing-function:steps()` 声明。总体来说, 逐帧动画的声明比较简单, 可用一张 `逐帧长图` 完成整个动画效果, 而关键帧动画需结合 `@keyframes` 为每个关键帧声明当前对应的状态, 若涉及的点较多, 可能比较繁琐。

关键帧动画声明步骤

- 在 `@keyframes` 里声明动画名称和动画每个关键帧的状态
- 动画名称不能重复否则会被覆盖, 关键帧通过百分比分割出每个关键帧并声明对应的状态
- 在指定节点中声明 `animation` 调用动画

逐帧动画声明步骤

- 准备一张 `逐帧长图` , 该图像包含动画效果的每一帧且每帧宽高必须一致
- 在 `steps()` 里声明逐帧长图及其展示方式
- 在指定节点中声明 `animation` 调用动画

@keyframes注意事项

关键帧动画的声明通过 `@keyframes` 完成, 编写形式如下。

```
@keyframes animation-name {  
  from {}  
  to {}  
}  
/* 或 */  
@keyframes animation-name {  
  p1 {}  
  p2 {}  
  p3 {}  
}
```

关键帧的取值必须是 `from`、`to` 或 `Percentage`。`from` 可用 `0%` 代替，`to` 可用 `100%` 代替，若开始或结束的关键帧无对应的状态，可不用声明 `from` 或 `to`。`0%` 的 `%` 不能省略，否则关键帧解析会失败。

后面声明的关键帧状态会覆盖前面声明的关键帧状态，动画结束后会回到 `animation-fill-mode` 声明的状态。

自动打字器

很多在线编辑器网站都有一些自动打字的效果，例如[CodePen](#)。很多同学都以为是JS实现的效果，其实查看 [Chrome Devtools](#) 发现是纯CSS实现的。观察多几次自动打字器，可发现其存在以下特点。

- 字体都是等宽字体，等宽字体可保证每次打字时光标的移动距离都是一致的
- 打字器的宽度由最初的 `0px` 逐渐增加内容后变成最终固定字数的宽度，宽度以等宽字体的个数为准
- 光标随着每打一个字就闪烁一次，打字速度均匀，打字完成后再次重复打字
- 整个打字过程存在两个动画，一个是打字器自增宽度，一个是光标闪烁
- 整个打字过程一闪一闪地完成，根据其断断续续的特点可判断该动画为逐帧动画

还记得第5章**样式计算**的长度单位吗？有一个叫做 `ch` 的长度单位，它是一个等宽字体的特有长度单位，准确宽度为 `0` 的宽度。因此一个等宽字体就是 `1ch`，两个等宽字体就是 `2ch`。通过等宽字体个数定制打字器长度最合适不过了，而常用的设备自带等宽字体有 `Consolas`、`Monaco` 和 `Monospace` 三种。

打字器自增宽度 可用 `0px` 到等宽字体指定个数的宽度 `nch` 为一个自增周期，使用动画完成其自增过程即可。

光标闪烁 可用 `border-right` 模拟，具体形象现在可脑补一下，有无想出什么效果？节点里包含文本，在最右边声明 `border-right`，那不就是一个具有静态光标的输入状态吗？文本右边就是光标，很符合常理，为 `border-right` 声明一个闪烁动画即可。

```
<div class="auto-typing">Do You Want To Know More About CSS Development
```

```
@mixin typing($count: 0, $duration: 0, $delay: 0) {
  overflow: hidden;
  border-right: 1px solid transparent;
  width: #{ $count + 1 }ch;
  font-family: Consolas, Monaco, monospace;
  white-space: nowrap;
  animation: typing #{ $duration }s steps($count + 1) #{ $delay }s infinite;
  caret 500ms steps(1) #{ $delay }s infinite forwards;
}

.auto-typing {
  font-weight: bold;
  font-size: 30px;
  color: #09f;
  @include typing(52, 5);
}

@keyframes caret {
  50% {
    border-right-color: currentColor;
  }
}

@keyframes typing {
  from {
    width: 0;
  }
}
```

☑ 在线演示: [Here](#)

☑ 在线源码: [Here](#)

