

- 盒模型介绍
- CSS选择器和优先级
- 重排 (reflow) 和重绘 (repaint) 的理解
 - 浏览器的渲染过程
 - 生成渲染树
 - 何时发生回流重绘
 - 浏览器的优化机制
 - 减少回流和重绘
 - 最小化重绘和重排
 - 批量修改DOM
 - 避免触发同步布局事件
 - 对于复杂的动画效果, 使用绝对定位让其脱离文档流
 - css3硬件加速 (GPU加速)
- 对BFC的理解
 - 视觉格式化模型
 - 块级元素
 - 块级盒
 - 行内级元素
 - 行内盒
 - 匿名盒
 - 定位方案
 - 普通流
 - 浮动
 - 定位技术
 - BFC (块级格式上下文)
 - BFC的创建
 - BFC的范围
 - BFC的特新
 - BFC的应用
 - 自适应多栏布局
 - 防止外边距折叠
 - 清除浮动
- 参考链接

盒模型介绍

CSS3中的盒模型有以下两种:

1. 标准盒模型
2. IE (替代) 盒模型

两种盒子模型都是由`content + padding + border + margin` 构成, 其大小都是由`content + padding + border`决定的, 但是盒子的内容宽/高度 (即`width/height`) 的计算范围根据盒模型的不同会有所不同:

- 标准盒模型: 只包含`content`
- IE (替代) 盒模型: `content + padding + border`

可以通过`box-sizing`来改变元素的盒模型

- `box-sizing: content-box`: 标准盒模型 (默认值)
- `box-sizing: border-box`: IE (替代) 盒模型

CSS选择器和优先级

选择器	示例
类型选择器	<code>h1 {}</code>
通配选择器	<code>* {}</code>
类选择器	<code>.box {}</code>
ID选择器	<code>#unique {}</code>
标签属性选择器	<code>a[title] {}</code>
伪类选择器	<code>p:first-child {}</code>
伪元素选择器	<code>p::first-line {}</code>
后代选择器	<code>article p</code>
子代选择器	<code>article > p</code>
相邻兄弟选择器	<code>h1 + p</code>
通用兄弟选择器	<code>h1 ~ p</code>

通常来说, 样式的优先级一般为`!important > style (内联样式) > ID选择器 > 类 (class) 选择器 > 标签选择器`, 但是涉及多类选择器作用于一个元素的时候怎么判断优先级呢? 在改一些第三方库 (比如antd) 样式时, 理解这个会帮助很大。

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:

1. 如果存在内联样式, 那么 $A = 1$, 否则 $A = 0$
2. B的值等于ID选择器出现的次数
3. C的值等于类选择器和属性选择器和伪类出现的总次数
4. D的值等于标签选择器和伪元素出现的总次数

例如: `#nav-global > ul > li > a.nav-link`

套用上面的算法, 一次求出A, B, C, D的值:

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

练习:

```
li /* (0, 0, 0, 1) */
ul li /* (0, 0, 0, 2) */
ul ol+li /* (0, 0, 0, 3) */
ul ol+li /* (0, 0, 0, 3) */
h1 + *[REL=up] /* (0, 0, 1, 1) */
ul ol li.red /* (0, 0, 1, 3) */
li.red.level /* (0, 0, 2, 1) */
a1.a2.a3.a4.a5.a6.a7.a8.a9.a10.a11 /* (0, 0, 11, 0) */
#x34y /* (0, 1, 0, 0) */
li:first-child h2 .title /* (0, 0, 2, 2) */
#nav .selected > a:hover /* (0, 1, 2, 1) */
html body #nav .selected > a:hover /* (0, 1, 2, 3) */
```

比较连个优先级高低的规则是: 从左往右依次进行比较, 较大者胜出, 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

优先级的特殊情况

经过上面的优先级计算规则, 我们可以知道内联样式的优先级是最高的, 但是外部样式有没有什么办法覆盖内联样式呢? 有的, 那就要 `!important` 出马了。因为一般情况下, 很少会使用内联样式, 所以 `!important` 也很少会用到! 如果不是为了要覆盖内联样式, 建议尽量不要使用 `!important`。

注意: 如果在内联样式中使用了`!important`, 那么外部样式无论怎样都不能覆盖内联样式, 因此千万不要在内联样式中使用`!important`

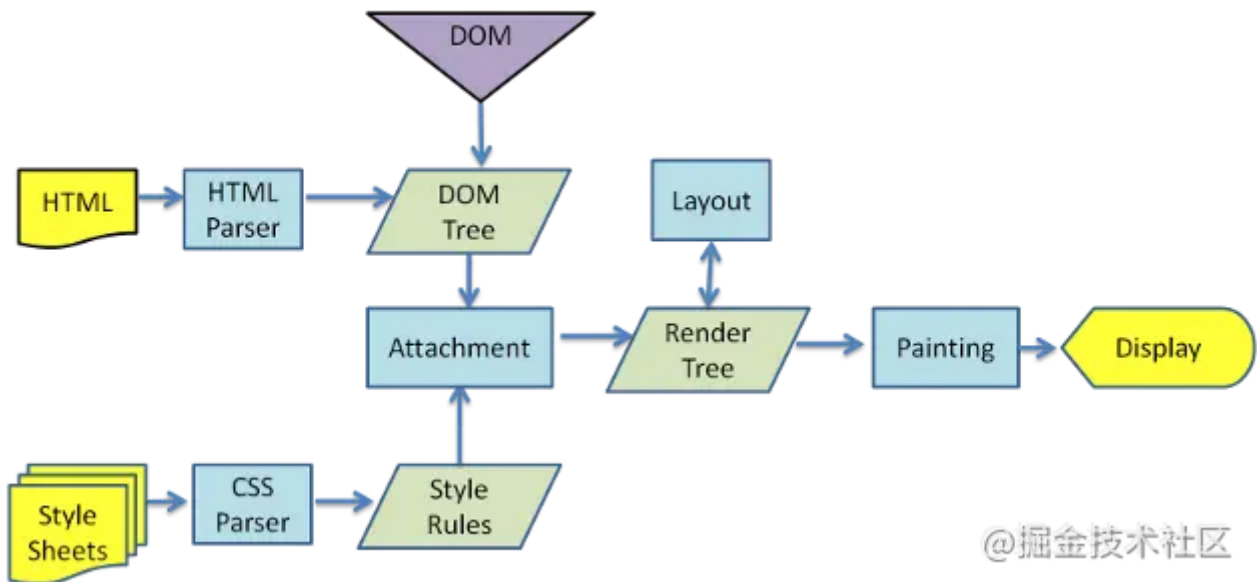
重排 (reflow) 和重绘 (repaint) 的理解

- 重排: 无论通过什么方式影响了元素的**几何信息** (元素在视口内的位置和尺寸大小), 浏览器需要**重新计算**元素在视口内的几何属性, 这个过程叫做重排。
- 重绘: 通过构造渲染树和重排 (回流) 阶段, 我们知道了哪些节点是可见的, 以及可见节点的样式和具体的几何信息 (元素在视口内的位置和尺寸大小), 接下来就可以将渲染树的每个节点都转换为屏幕上的**实际像素**, 这个阶段就叫做重绘。

如何减少重排和重绘?

- 最小化重绘和重排: 比如样式集中改变, 使用添加新样式类名 `.class` 或 `cssText`
- 批量操作DOM: 比如读取某元素 `offsetWidth` 属性存到一个临时变量, 再去使用, 而不是频繁使用这个计算属性; 又比如利用 `document.createDocumentFragment()` 来添加要被添加的节点, 处理完之后再插入到实际DOM中
- 使用 `absolute` 或 `fixed` 使元素脱离文档流, 这在制作复杂的动画时对性能的影响比较明显
- 开启GPU加速: 利用css属性 `transform`、`will-change` 等, 比如改变元素的位置, 我们使用 `translate` 会比使用绝对定位改变其 `left`、`top` 等来的高效, 因为它不会触发重排或重绘, `transform` 使浏览器为元素创建一个GPU图层, 这使得动画元素在一个独立的图层进行渲染。当元素的内容没有发生改变, 就没有必要进行重绘。

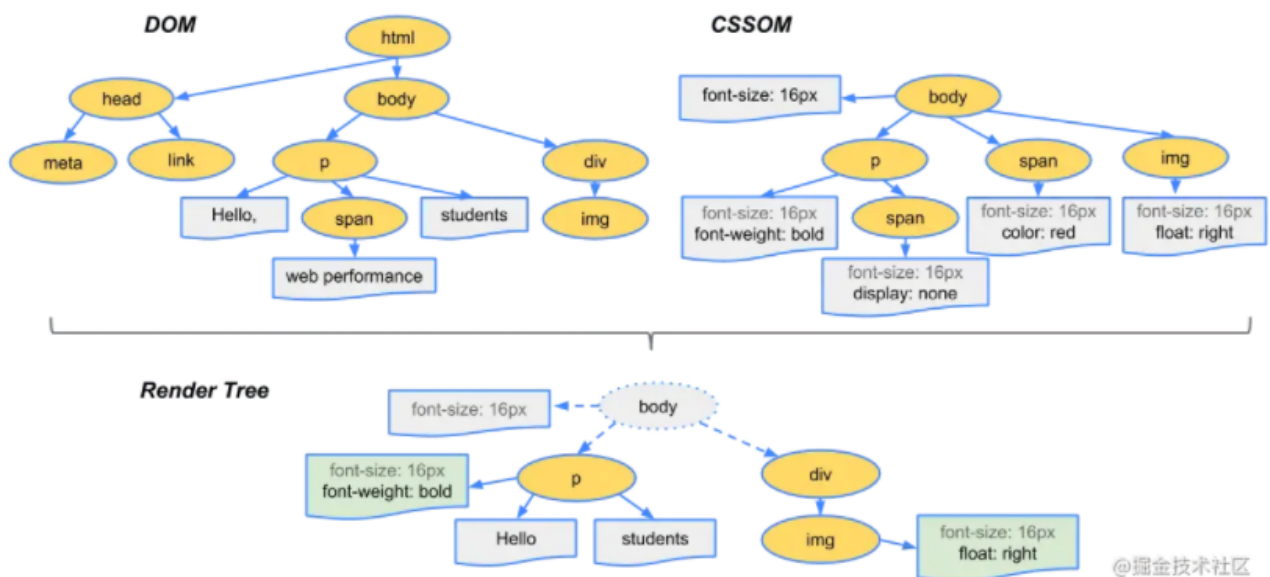
浏览器的渲染过程



从上图可以看到，浏览器的渲染过程如下：

1. 解析HTML，生成DOM树；解析CSS，生成CSSOM树
2. 将DOM树和CSSOM树，生成渲染树（Render Tree）
3. Layout/Reflow（回流）：根据生成的渲染树，进行回流，得到节点的几何信息（位置，大小）
4. 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
5. Display：将像素发送给GPU，展示在页面上。（这一步其实还有很多内容，比如会在GPU将多个层合并为同一个层，并展示在页面中。而CSS3硬件加速的原理则是新建合成层）

生成渲染树



为了构建渲染树，浏览器主要完成了以下工作：

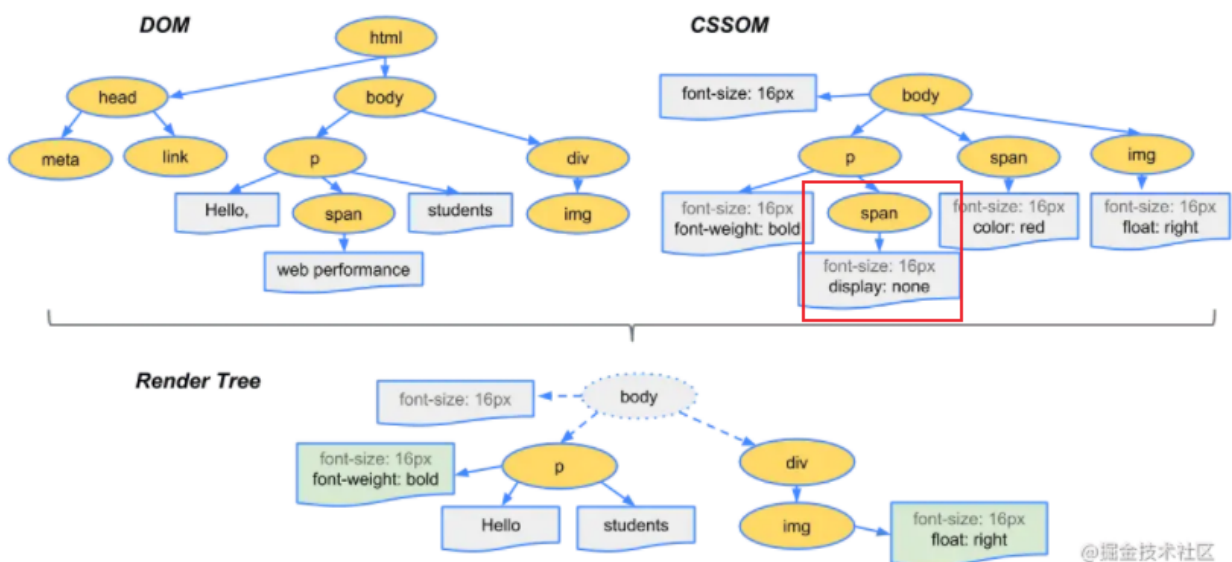
1. 从DOM树的根节点开始遍历每个可见节点
2. 对于每个可见的节点，找到CSSOM树中对应的规则，并应用它们
3. 根据每个可见节点以及其对应的样式，组合生成渲染树

第一步中，既然说了要遍历可见的节点，那么我们得先知道，什么节点是不可见的。不可见的节点包括：

- 一些不会渲染输出的节点：比如`script`, `meta`, `link`等
- 一些通过css进行隐藏的节点。比如`display: none`。注意，利用`visibility`和`opacity`隐藏的节点，还是会显示在渲染树上的。只有`display: none`的节点才不会显示在渲染树上。

从上图来说，我们可以看到`span`标签的样式有一个`display: none`，因此，它最终并没有显示在渲染树上。

注意：渲染树只包含可见的节点



@掘金技术社区

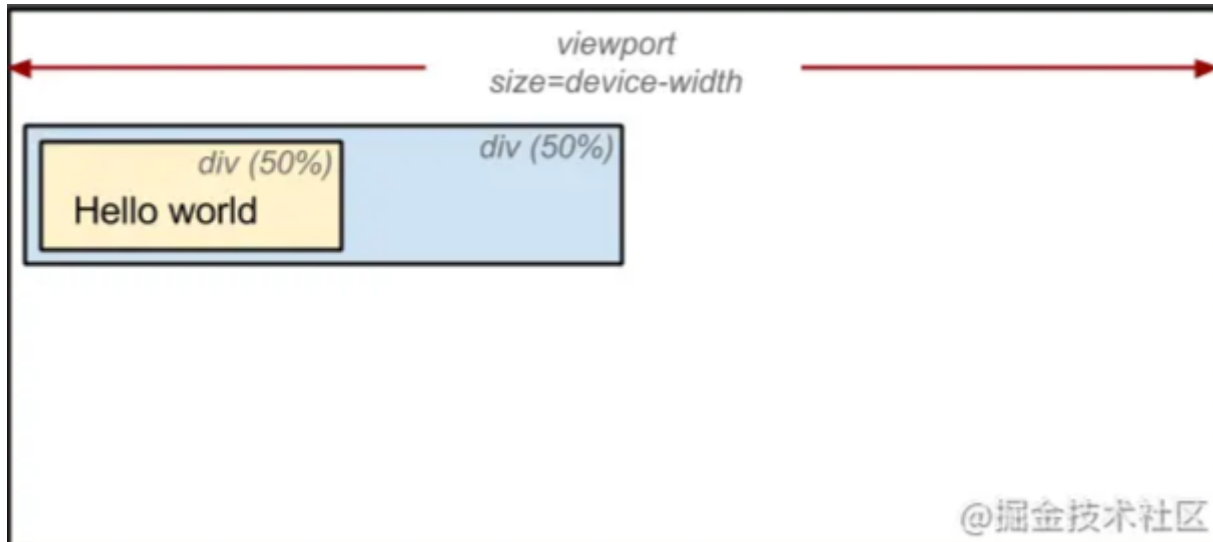
回流

前面我们通过构造渲染树，我们将可见的DOM节点以及它对应的样式结合起来，可是我们还需要计算它们在设备视口（viewport）内的确切位置和大小，这个计算阶段就是回流。

为了弄清每个对象在网站上的确切大小和位置，浏览器从渲染树的根节点开始遍历，我们可以以下面这个实例来表示：

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Reflow</title>
  </head>
  <body>
    <div style="width: 50%">
      <div style="width: 50%">Hello World!</div>
    </div>
  </body>
</html>
```

我们可以看到，第一个div节点将节点的显示尺寸设置为视口宽度的50%，第二个div将其尺寸设置为父节点的50%。而在回流这个阶段，我们就需要根据视口具体的宽度，将其转为实际的像素值。



重绘

最终，我们通过构造渲染树和回流阶段，我们知道了哪些节点是可见的，以及可见节点的样式和具体的几何信息（位置、大小），那么我们就可以将渲染树的每个节点都转换为屏幕上的实际像素，这个阶段就叫做重绘节点。

何时发生回流重绘

我们前面知道了，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。比如以下情况：

- 添加或删除可见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（这个无法避免）
- 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）

注意：回流一定会触发重绘，而重绘不一定会回流

根据改变的范围和程度，渲染树中或大或小的部分需要重新计算，有些改变会触发整个页面的重排，比如，滚动条出现的时候或者修改了根节点

浏览器的优化机制

由于每次重排都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操作达到了一个阈值，才清空队列。但是，**当你获取布局信息的操作的时候，会强制队列刷新**，比如当你访问以下属性或使用以下方法：

- offsetTop、offsetLeft、offsetWidth、offsetHeight
- scrollTop、scrollLeft、scrollWidth、scrollHeight
- clientTop、clientLeft、clientWidth、clientHeight
- getComputedStyle()

- `getBoundingClientRect`
- 具体可以访问这个网站: <https://gist.github.com/paulirish/5d52fb081b3570c81e3a>

以上属性和方法都需要返回最新的布局信息, 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。因此, 我们在修改样式的时候, 最好避免使用上面列出的属性, 它们都会刷新渲染队列。如果要使用它们, 最好将值缓存起来。

减少回流和重绘

最小化重绘和重排

由于重绘和重排可能代价比较昂贵, 因此最好就是可以减少它的发生次数。为了减少发生次数, 我们可以合并多次对DOM和样式的修改, 然后一次处理掉。考虑这个例子

```
const el = document.getElementById("div");
el.style.padding = "5px";
el.style.borderLeft = "1px";
el.style.borderRight = "2px";
```

例子中, 有三个样式属性被修改了, 每一个都会影响元素的几何结构, 引起回流。当然, 大部分现代浏览器都对其做了优化, 因此, 只会触发一次重排。但是如果在旧版的浏览器或者在上面代码执行的时候, 有其他代码访问了布局信息(上文中的会触发回流的布局信息), 那么就会导致三次重排。

因此, 我们可以合并所有的改变然后依次处理, 比如我们可以采取一下的方式:

- 使用`cssText`

```
const el = document.getElementById("div");
el.style.cssText += "border-left: 1px; border-right: 2px; padding: 5px;";
```

- 修改CSS的class

```
const el = document.getElementById("div");
el.className += " active";
```

批量修改DOM

当我们需要对DOM对一系列修改的时候, 可以通过以下步骤减少回流重绘次数:

1. 使元素脱离文档流
2. 对其进行多次修改
3. 将元素带回到文档中

该过程的第一步和第三部可能引起回流, 但是经过第一步之后, 对DOM的所有修改都不会引起回流重绘, 因为它已经不在渲染树了。

有三种方式可以让DOM脱离文档流：

- 隐藏元素，应用修改，重新显示
- 使用文档片段（document fragment）在当前DOM之外构建一个子树，再把它拷贝回文档
- 将原始元素拷贝到一个脱离文档的节点中，修改节点后，再替换原始的元素

考虑我们要执行一段批量插入节点的代码：

```
function appendDataToElement(appendToElement, data) {
  let li;
  for (let i = 0; i < data.length; i++) {
    li = document.createElement("li");
    li.textContent = "text";
    appendToElement.appendChild(li);
  }
}

const ul = document.getElementById("list");
appendDataToElement(ul, data);
```

如果我们直接这样执行的话，由于每次循环都会插入一个新的节点，会导致浏览器回流一次。

我们可以使用这三种方式进行优化：

1. 隐藏元素，应用修改，重新显示

这个会在展示和隐藏节点的时候，产生两次回流

```
function appendDataToElement(appendToElement, data) {
  let li;
  for (let i = 0; i < data.length; i++) {
    li = document.createElement("li");
    li.textContent = "text";
    appendToElement.appendChild(li);
  }
}

const ul = document.getElementById("list");
ul.style.display = "none";
appendDataToElement(ul, data);
ul.style.display = "block";
```

2. 使用文档片段（document fragment）在当前DOM之外构建一个子树，再把它拷贝回文档

```
const ul = document.getElementById("list");
const fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
ul.appendChild(fragment);
```


3. 将原始元素拷贝到一个脱离文档流的节点中，修改节点后，再替换原始的元素

```
const ul = document.getElementById('list');
const clone = ul.cloneNode(true);
appendDataToElement(clone, data);
ul.parentNode.replaceChild(clone, ul);
```

避免触发同步布局事件

上文我们说过，当我们访问元素的一些属性的时候，会导致浏览器强制清空队列，进行强制同步布局。比如我们将一个p标签数组的宽度赋值为一个元素的宽度，我们可能写出这样的代码：

```
function initP() {
  for (let i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = box.offsetWidth + "px";
  }
}
```

这段代码看上去没有什么问题，可是其实会造成很大的性能问题。每次在循环的时候，都读取了box的一个offsetWidth属性值，然后利用它来更新p标签的width属性。这就导致了每一次循环的时候，浏览器都必须先使上一次循环中的样式更新操作生效，才能响应本次循环的样式读取操作。**每一次循环都会强制浏览器刷新队列。**我们可以优化为：

```
const width = box.offsetWidth;
function initP () {
  for (let i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = width + 'px';
  }
}
```

对于复杂的动画效果，使用绝对定位让其脱离文档流

对于复杂动画效果，由于会经常的引起回流重绘，因此，我们使用绝对定位，让它脱离文档流。否则会引起父元素以及后续元素频繁的回流。

css3硬件加速（GPU加速）

比起考虑如何减少回流重绘，我们更期望的是，根本不要回流重绘。这个时候，就需要css3硬件加速了

1. 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
2. 对于动画的其它属性，比如background-color这些，还是会引起回流重绘的，不过它还是可以提升这些动画的性能。

如何使用

常见的触发硬件加速的css属性：

- transform
- opacity
- filters
- will-change

css3硬件加速的坑

当然，任何美好的东西都是会有对应的代价的，过犹不及。css3硬件加速还是有坑的：

- 如果你为太多元素使用css3硬件加速，会导致内存占用较大，会有性能问题
- 在GPU渲染字体会导致抗锯齿无效。这是因为GPU和CPU的算法不同。因此如果你不在动画结束的时候关闭硬件加速，会产生字体模糊

对BFC的理解

BFC（Block Formatting Contexts）即块级格式上下文，根据盒模型可知，每个元素都被定义为一个矩形盒子，然而盒子的布局会受到**尺寸**，**定位**，**盒子的子元素或兄弟元素**，**视口的尺寸**等因素决定，所以这里有一个浏览器计算的过程，计算的规则就是由一个叫做**视觉格式化模型**的东西所定义的，BFC就是来自这个概念，它是CSS视觉渲染的一部分，**用于决定块级盒的布局及浮动相互影响范围的一个区域**。

BFC具有一些特性：

1. 块级元素会在垂直方向一个接一个的排列，和文档流的排列方式一致
2. 在BFC中上下相邻的两个容器的margin会重叠，创建新的BFC可以避免外边距重叠
3. 计算BFC的高度时，需要计算浮动元素的高度
4. BFC区域不会与浮动的容器发生重叠
5. BFC是独立的容器，容器内部元素不会影响外部元素
6. 每个元素的左margin值和容器的左border相接触

利用这些特性，我们可以解决以下问题：

- 利用4和6，我们可以实现三栏（或两栏）自适应布局
- 利用2，我们可以避免margin重叠问题
- 利用3，我们可以避免高度塌陷

创建BFC的方式：

- 绝对定位元素（position为absolute或fixed）
- 行内块元素，即display为inline-block
- overflow的值不为visible

视觉格式化模型

CSS视觉格式化模型描述了盒子是怎样生成的，简单来说，它定义了盒子生成的计算规则，通过规则将文档元素转换为一个个盒子。

每一个盒子的布局由**尺寸**、**类型**、**定位**、**盒子的子元素或兄弟元素**、**视口的尺寸和位置**等因素决定

视觉格式化模型的计算，取决于一个矩形的边界，这个矩形边界，就是**包含块**（containing block），比如：

```
<table>
  <tr>
    <td></td>
  </tr>
</table>
```

上述代码片段中，**table**和**tr**都是包含块，**table**是**tr**的包含块，同时**tr**又是**td**的包含块

需要注意的是，**盒子不受包含块的限制，当盒子的布局跑到包含块的外面时，就是我们说的溢出（overflow）**

视觉格式化模型定义了盒（box）的生成，其中的盒主要包括了**块级盒**，**行内盒**和**匿名盒**

块级元素

CSS属性值**display**为**block**，**list-item**，**table**的元素

块级盒

块级盒具有以下特性：

- CSS属性值**display**为**block**，**list-item**，**table**时，它就是块级元素
- 视觉上，块级盒呈现垂直排列的块
- 每个块级盒都会参与BFC的创建
- 每个块级元素都会至少生成一个块级盒，称为主块级盒；一些元素可能会生成额外的块级盒，比如****，用来存放项目符号

行内级元素

CSS属性值**display**为**inline**，**inline-block**，**inline-table**的元素

行内盒

行内盒具有以下特性：

- CSS属性值**display**为**inline**，**inline-block**，**inline-table**时，它就是行内级元素
- 视觉上，行内盒与其它行内级元素排列为多行
- 所有的可替换元素（**display**值为**inline**，如****，**<iframe>**，**<video>**，**<embed>**等）生成的盒都是行内盒，它们会参与**IFC**（行内格式化上下文）的创建
- 所有的非可替换行内元素（**display**值为**inline-block**或**inline-table**）生成盒称为**原子行内级盒**，不参与**IFC**创建

匿名盒

匿名盒指不能被CSS选择器选中的盒子，比如：

```
<div>
  匿名盒1
```

```
<p>块盒</p>  
匿名盒2  
</div>
```

上述代码片段中，`div`元素和`p`元素都会生成一个块级盒，`p`元素的前后会生成两个匿名盒

匿名盒所有可继承的CSS属性值都为`inherit`，所有不可继承的CSS属性值都为`initial`

匿名盒

有时为了需要会添加补充性盒，这些盒称为匿名盒（anonymous boxes），它们没有名字，不能被 CSS 选择器选中。所以匿名盒不能为其设置样式，所有样式均来自继承（inherit）或初始值（initial）。

匿名块盒（Anonymous Block Boxes）

如下，我们有个 `div` 元素，里面有两个直接文本及一个 `p` 元素

```
<div class="block">  
  我是直接文本  
  <p class="block-p">我是块级元素，但是我的前后都不是</p>  
  我也是直接文本  
</div>
```

这个时候就会为 `p` 元素上下的直接文本创建匿名块盒，大概如下（红色圈起来的我们可以理解为匿名块盒，它是不能被 CSS 选择器选中的）：

我是直接文本

我是块级元素，但是我的前后都不是

我也是直接文本

匿名行内盒（Anonymous Inline Boxes）

同样也有匿名行内盒，HTML 代码如下：

```
<p><span>课程名称：</span>前端工程师 NEXT 学位</p>
```

因为“前端工程师 NEXT 学位”没有元素包裹，属于直接文本，所以 CSS 引擎将会为其创建匿名行内盒。

定位方案

CSS页面布局技术允许我们拾取网页中的元素，并且控制它们相对正常布局流（普通流）、周边元素、父容器或者主视口/窗口的位置。技术布局从宏观上来说受定位方案影响，定位方案包括**普通流**（Normal Flow，也叫常规流，正常布局流），**浮动**（Float），**定位技术**（Position）

普通流

浏览器默认的**HTML**布局方式，此时浏览器不对页面做任何布局控制

当**position**为**static**或**relative**，并且**float**为**none**时会触发普通流，普通流有以下特性：

- 普通流中，所有的盒一个接一个排列
- **BFC**中，盒子会竖着排列
- **IFC**中，盒子会横着排列
- 静态定位中（**position**为**static**），盒的位置就是普通流里布局的位置
- 相对定位中（**position**为**relative**），盒的偏移位置由**top**, **right**, **bottom**, **left**定义，**即使有偏移，仍然保留原有的位置，其它普通流不能占用这个位置）

浮动

- 浮动定位中，盒称为浮动盒（Floating Box）
- 盒位于当前行的开头或结尾
- 普通流会环绕在浮动盒周围，除非设置**clear**属性

定位技术

1. 静态定位

position: static：该关键字指定元素使用正常的布局行为，即元素在文档常规流中当前的布局位置，此时**top**, **right**, **bottom**, **left** 和 **z-index**属性无效

2. 相对定位

position: relative：该关键字下，元素先放置在未添加定位时的位置，再在不改变页面布局的前提下调整元素位置（因此会在此元素未添加定位时所在位置留下空白）。

3. 绝对定位

position: absolute：元素会被移出正常的文档流，并不为元素预留空间，通过指定元素相对于最近的非**static**定位祖先元素（因为默认所有元素都是**static**定位）的偏移，来确定元素的位置。绝对定位的元素可以设置外边距（**margins**），且不会与其它边距合并。

4. 固定定位

position: fixed：元素会被移出正常文档流，并不为元素预留空间，而是通过指定元素相对于屏幕视口（**viewport**）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变。打印时，元素会出现在每页的固定位置。**fixed**属性会创建新的层叠上下文。当元素祖先的**transform**, **perspective**或**filter**属性非**none**时，容器由视口改为该祖先。

5. 粘性定位

position: sticky：元素根据正常文档流进行定位，然后相对它最近*滚动祖先（nearest scrolling ancestor）*和**containing block**（最近块级祖先nearest block-level ancestor），包括**table-related**元素，基于**top**, **right**,

`bottom`, `left` 的值进行偏移。偏移值不会影响任何其它元素的位置。

该值总是创建一个新的层叠上下文 (*stacking context*)，注意，一个 `sticky` 元素会“固定”在离它最近的一个拥有“滚动机制”的祖先上（当该祖先的 `overflow` 是 `hidden`, `scroll`, `auto` 或 `overlay` 时），即便这个祖先不是最近的真实可滚动祖先。这有效地抑制了任何“sticky”行为

BFC（块级格式上下文）

通过对CSS盒模型，定位，布局等信息的了解，我们知道BFC这个概念其实来自于**视觉格式化模型**

它是页面CSS视觉渲染的一部分，用于决定块级盒的布局及浮动相互影响范围的一个区域

BFC的创建

一下元素会创建BFC:

- 根元素 (`<html>`)
- 浮动元素 (`float`不为`none`)
- 绝对定位元素 (`position`为`absolute`或`fixed`)
- 表格的标题和单元格 (`display`为`table-caption`, `table-cell`)
- `overflow`的值不为`visible`的元素
- 弹性元素 (`display`为`flex`或`inline-flex`的元素的直接子元素)
- 网格元素 (`display`为`grid`或`inlien-grid`的元素的直接子元素)

块格式上下文对浮动定位与清除浮动都很重要。浮动定位和清除浮动时只会应用于同一个BFC内的元素。浮动不会影响其它BFC中元素的布局，而清除浮动只能清除同一BFC中在它前面的元素的浮动。外边距折叠 (Margin collapsing) 也只会发生在属于同一BFC的块级元素之间。

以上是 **CSS2.1** 规范定义的 **BFC** 触发方式，在最新的 **CSS3** 规范中，弹性元素和网格元素会创建 **F(Flex)FC** 和 **G(Grid)FC**。

BFC的范围

BFC包含创建它的元素的所有子元素但是不包括创建了新的BFC的子元素的内部元素

简单来说，子元素如果又创建了一个新的 BFC，那么它里面的内容就不属于上一个 BFC 了，这体现了 BFC 隔离的思想，我们还是以 table 为例：

```
<table>
  <tr>
    <td></td>
  </tr>
</table>
```

假设 table 元素创建的 BFC 我们记为 BFC_table，tr 元素创建的 BFC 记为 BFC_tr，根据规则，两个 BFC 的范围分别为：

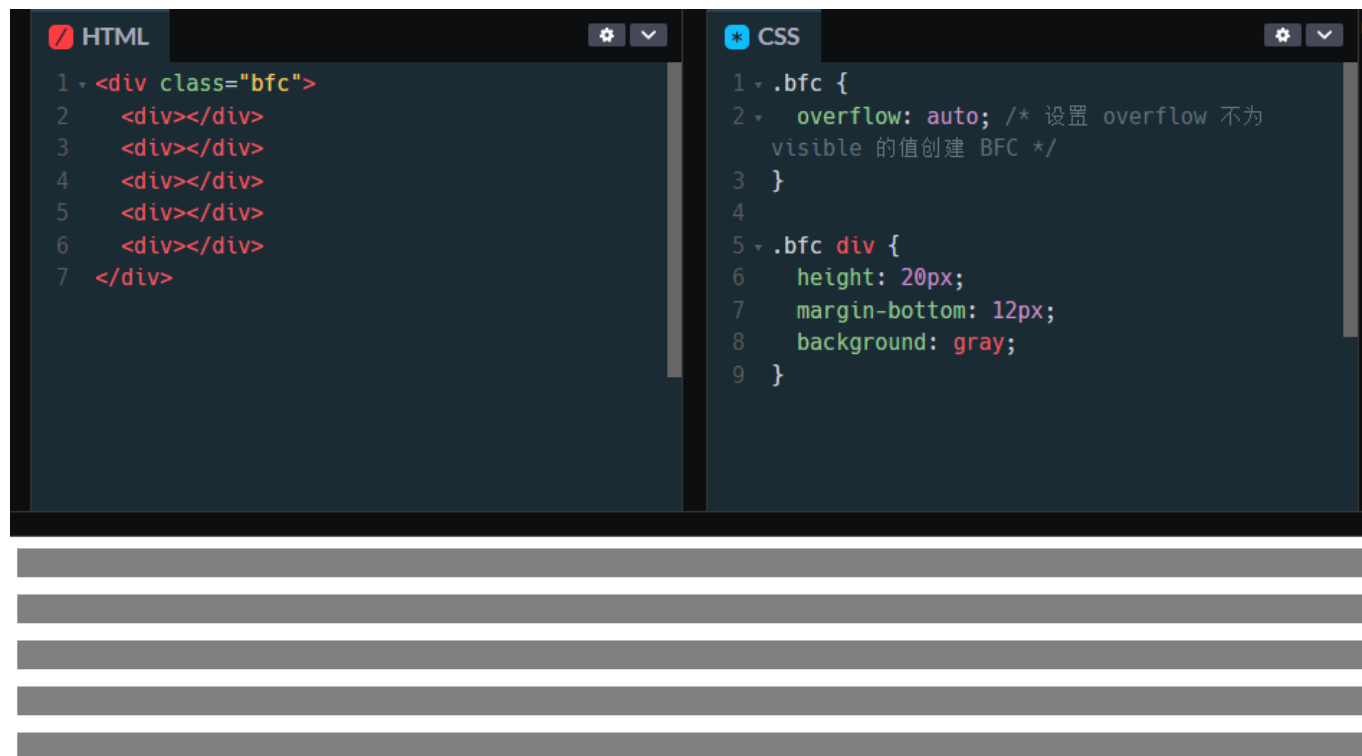
- BFC_tr: td 元素
- BFC_table: 只有 tr 元素，不包括 tr 里的 td 元素

也就是说，**一个元素不能同时存在于两个 BFC 中。**

BFC的特新

BFC 除了会创建一个隔离的空间外，还具有以下特性：

1. BFC 内部的块级盒会在垂直方向上一个接一个排列



2. 同一个 BFC 下的相邻块级元素可能发生外边距折叠，创建新的 BFC 可以避免的外边距折叠

HTML

```
1 <div class="top">TOP</div>
2 <div class="outer">
3   <div class="inner">INNER</div>
4 </div>
5
```

CSS

```
1 .top {
2   height: 50px;
3   margin: 10px 0;
4   background: yellow;
5 }
6
7 .outer {
8   background: gray;
9   height: 50px;
10  /* 创建新的 BFC 就可以避免这两个相邻块级盒
    间的外边距折叠，尝试去掉注释后看效果 */
11  /* overflow: auto; */
12 }
13
14 .inner {
15   height: 20px;
16   margin: 10px 0; /* 这里设置了inner box的
    margin，很明显没有生效 */
17   background: green;
18 }
```

TOP

INNER

HTML

```
1 <div class="top">TOP</div>
2 <div class="outer">
3   <div class="inner">INNER</div>
4 </div>
5
```

CSS

```
1 .top {
2   height: 50px;
3   margin: 10px 0;
4   background: yellow;
5 }
6
7 .outer {
8   background: gray;
9   height: 50px;
10  /* 创建新的 BFC 就可以避免这两个相邻块级盒
    间的外边距折叠，尝试去掉注释后看效果 */
11  overflow: auto;
12 }
13
14 .inner {
15   height: 20px;
16   margin: 10px 0; /* 这里设置了inner box的
    margin，很明显没有生效 */
17   background: green;
18 }
```

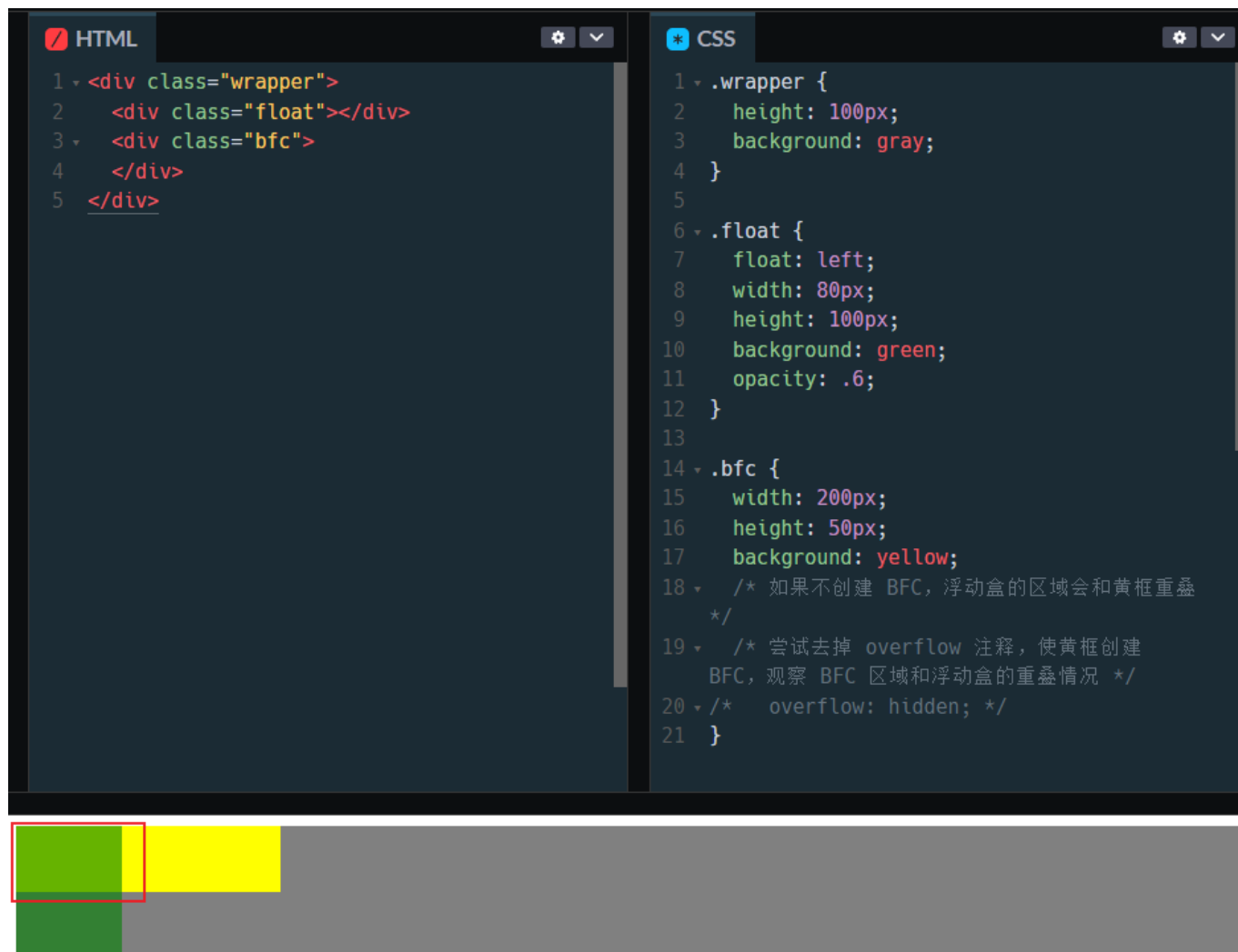
TOP

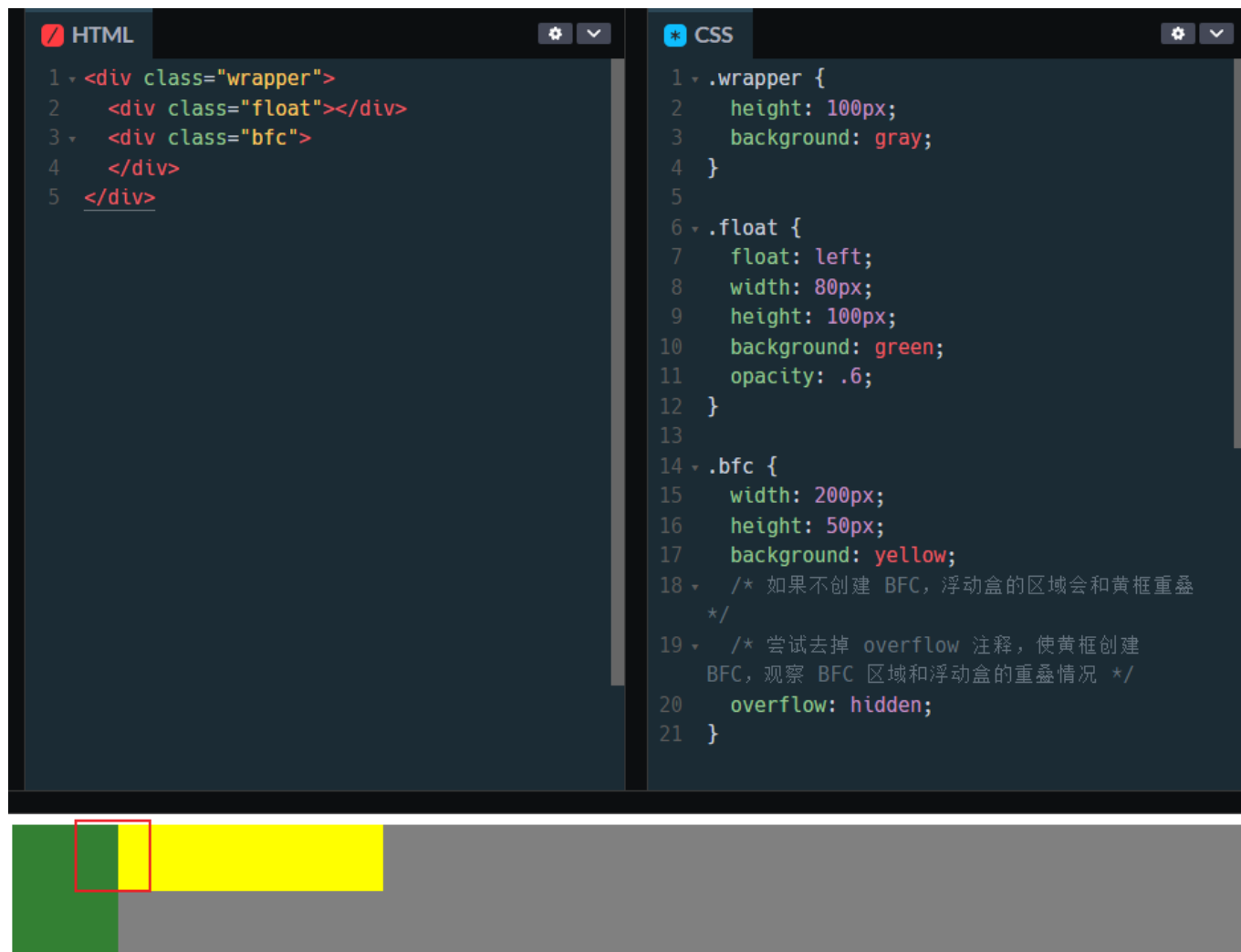
INNER

3. 每个元素的外边距盒（margin box）的左边与包含块边框盒（border box）的左边相接触（从右向左的格式化，则相反），即使存在浮动也是如此

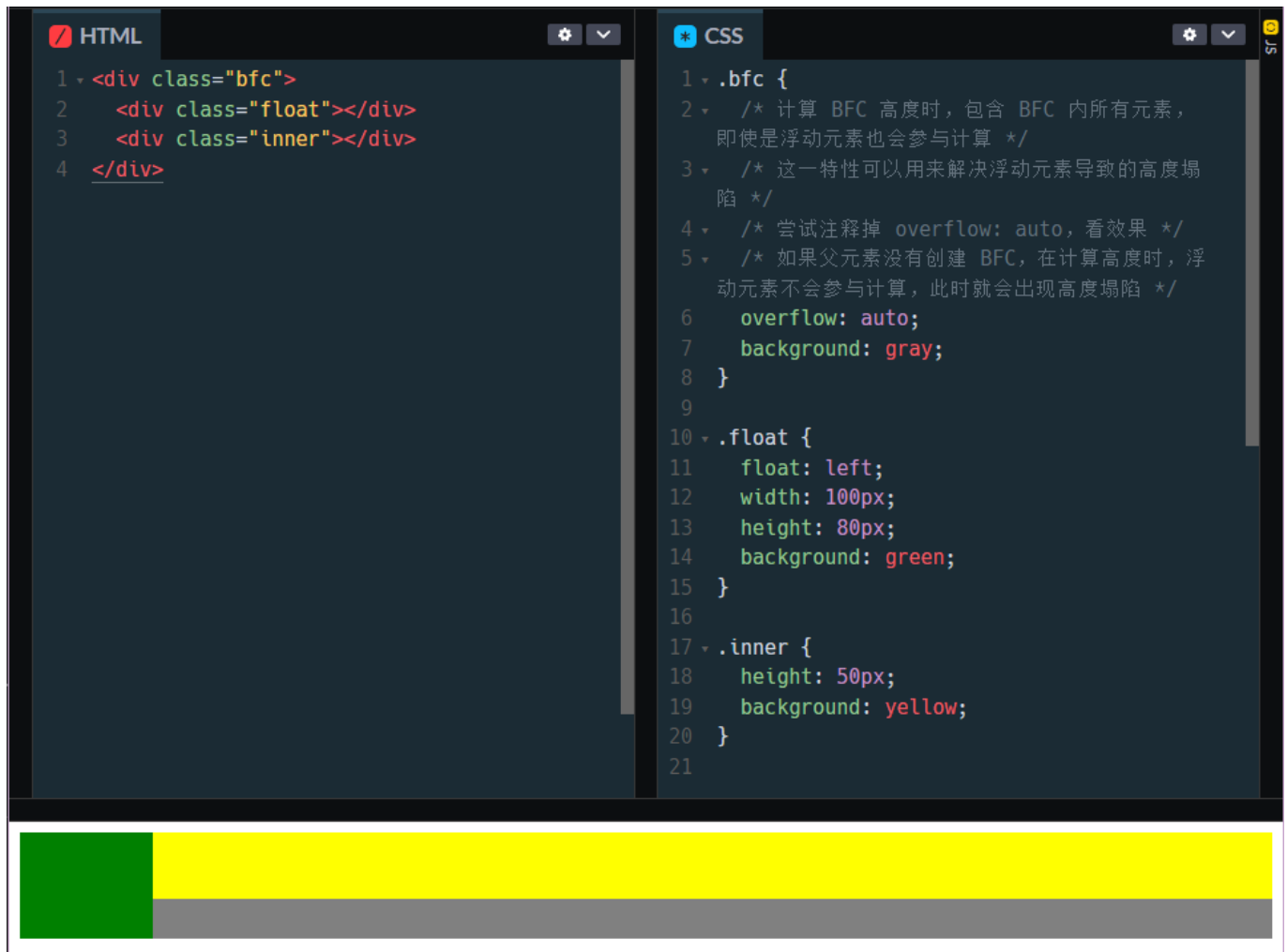


4. 浮动盒的区域不会和 BFC 重叠





5. 计算 BFC 的高度时，浮动元素也会参与计算



The image shows a code editor with two panels: HTML and CSS. The HTML panel contains a simple structure with a `div` of class `bfc` containing two `div` elements of classes `float` and `inner`. The CSS panel defines styles for these classes. The `.bfc` class sets `overflow: auto` and a gray background. The `.float` class sets `float: left`, `width: 100px`, `height: 80px`, and a green background. The `.inner` class sets `height: 50px` and a yellow background. Below the code editor, a visual representation of the layout is shown: a green rectangle (representing the `.float` element) is positioned to the left of a yellow rectangle (representing the `.inner` element), which is contained within a larger gray rectangle (representing the `.bfc` container).

```
HTML
1 <div class="bfc">
2   <div class="float"></div>
3   <div class="inner"></div>
4 </div>

CSS
1 .bfc {
2   /* 计算 BFC 高度时，包含 BFC 内所有元素，
3    即使是浮动元素也会参与计算 */
4   /* 这一特性可以用来解决浮动元素导致的高度塌陷 */
5   /* 尝试注释掉 overflow: auto，看效果 */
6   /* 如果父元素没有创建 BFC，在计算高度时，浮
7    动元素不会参与计算，此时就会出现高度塌陷 */
8   overflow: auto;
9   background: gray;
10 }
11
12 .float {
13   float: left;
14   width: 100px;
15   height: 80px;
16   background: green;
17 }
18
19 .inner {
20   height: 50px;
21   background: yellow;
22 }
```

```
HTML
1 <div class="bfc">
2   <div class="float"></div>
3   <div class="inner"></div>
4 </div>

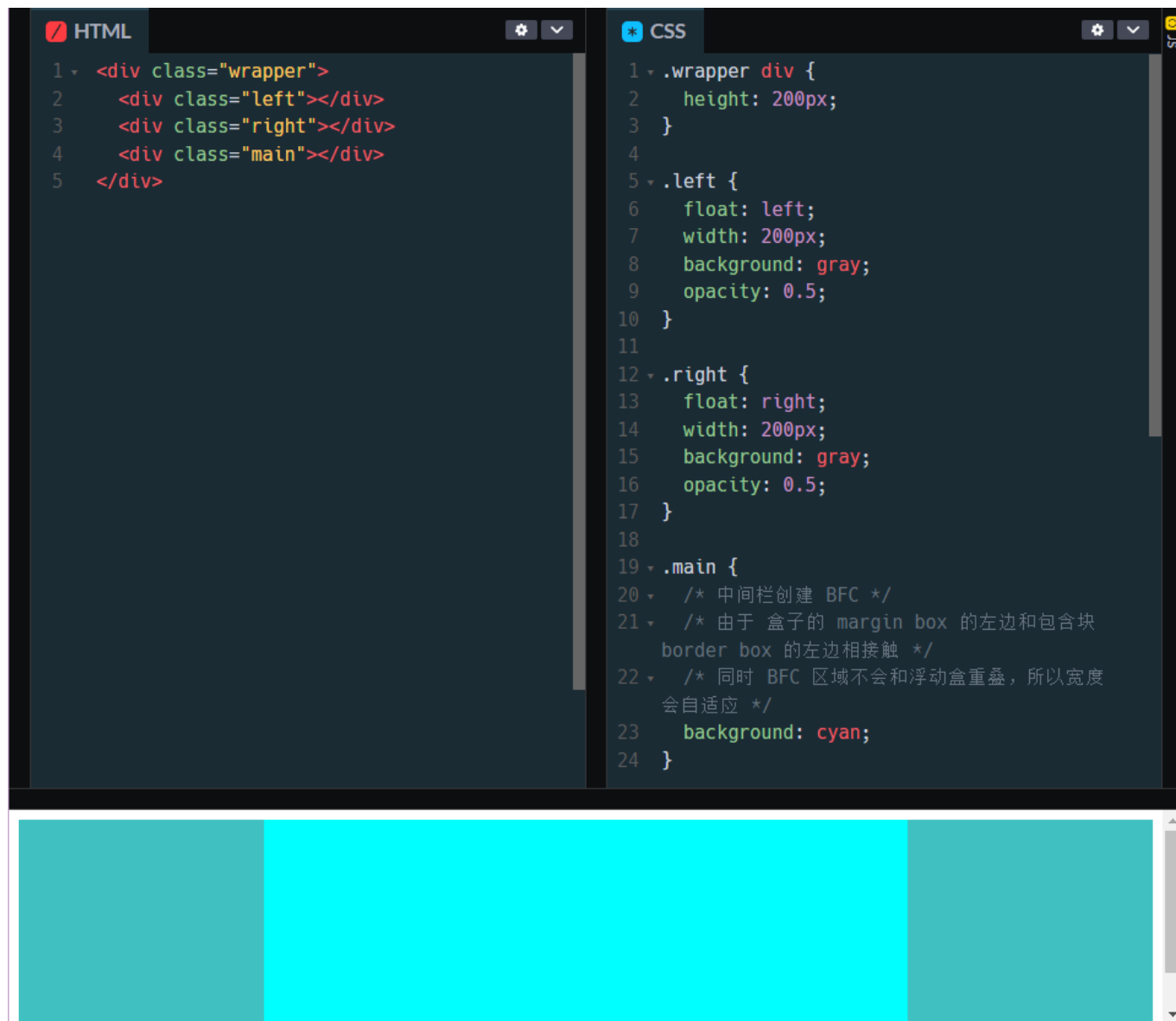
CSS
1 .bfc {
2   /* 计算 BFC 高度时，包含 BFC 内所有元素，
3    即使是浮动元素也会参与计算 */
4   /* 这一特性可以用来解决浮动元素导致的高度塌陷 */
5   /* 尝试注释掉 overflow: auto，看效果 */
6   /* 如果父元素没有创建 BFC，在计算高度时，浮
7    动元素不会参与计算，此时就会出现高度塌陷 */
8   overflow: auto;
9   background: gray;
10 }
11
12 .float {
13   float: left;
14   width: 100px;
15   height: 80px;
16   background: green;
17 }
18
19 .inner {
20   height: 50px;
21   background: yellow;
22 }
```



BFC的应用

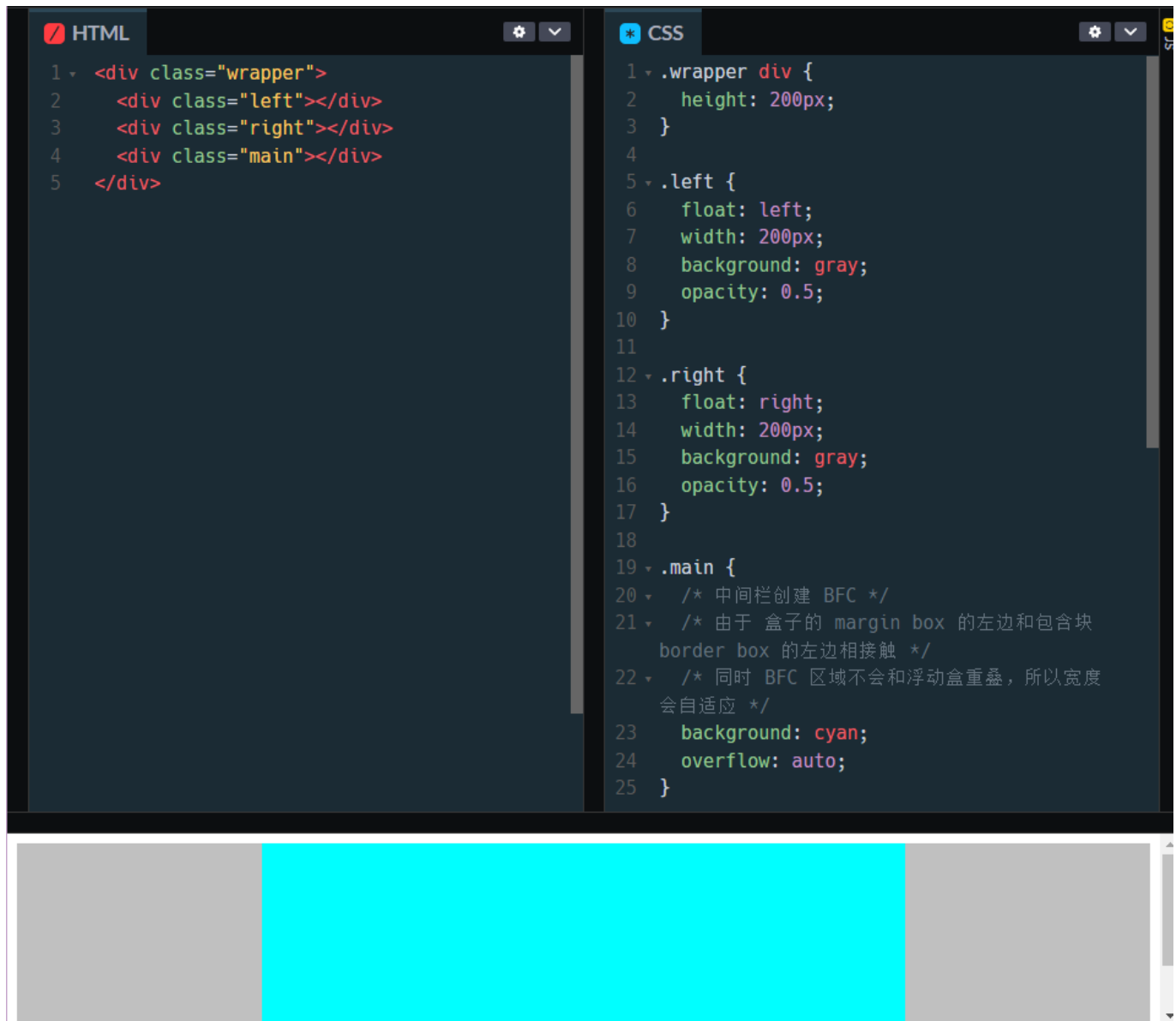
自适应多栏布局

利用特性3和4，中间栏创建BFC，左右栏宽度固定后浮动。由于盒子的margin box的左边和包含块border box的左边相接触，同时浮动盒的区域不会和BFC重叠，所以中间栏的宽度会自适应。



```
HTML
1 <div class="wrapper">
2   <div class="left"></div>
3   <div class="right"></div>
4   <div class="main"></div>
5 </div>

CSS
1 .wrapper div {
2   height: 200px;
3 }
4
5 .left {
6   float: left;
7   width: 200px;
8   background: gray;
9   opacity: 0.5;
10 }
11
12 .right {
13   float: right;
14   width: 200px;
15   background: gray;
16   opacity: 0.5;
17 }
18
19 .main {
20   /* 中间栏创建 BFC */
21   /* 由于 盒子的 margin box 的左边和包含块
22      border box 的左边相接触 */
23   /* 同时 BFC 区域不会和浮动盒重叠，所以宽度
24      会自适应 */
25   background: cyan;
26 }
```

防止外边距折叠

利用特性²，创建新的BFC,让相邻的块级盒位于不同BFC下可以防止外边距折叠。

```
HTML
1 <div class="box"></div>
2 <div class="bfc">
3   <div class="box"></div>
4 </div>
5

CSS
1 .box {
2   width: 100px;
3   height: 100px;
4   background: cyan;
5   /* 这里设置两个 box 的 margin, 但是
   margin 在垂直方向上发生了折叠, 因为他们位于同
   一个 BFC 下 */
6   margin: 50px 0;
7 }
8
9 .bfc {
10  /* 为了避免外边距折叠, 可以让包裹层创建新的
   BFC, 使两个 box 不在同一个 BFC 下 */
11  /* 尝试去掉 overflow 注释, 看效果 */
12  /* overflow: hidden; */
13 }
```



```
HTML
1 <div class="box"></div>
2 <div class="bfc">
3   <div class="box"></div>
4 </div>
5

CSS
1 .box {
2   width: 100px;
3   height: 100px;
4   background: cyan;
5   /* 这里设置两个 box 的 margin，但是
   margin 在垂直方向上发生了折叠，因为他们位于同
   一个 BFC 下 */
6   margin: 50px 0;
7 }
8
9 .bfc {
10  /* 为了避免外边距折叠，可以让包裹层创建新的
   BFC，使两个 box 不在同一个 BFC 下 */
11  /* 尝试去掉 overflow 注释，看效果 */
12  overflow: hidden;
13 }
```

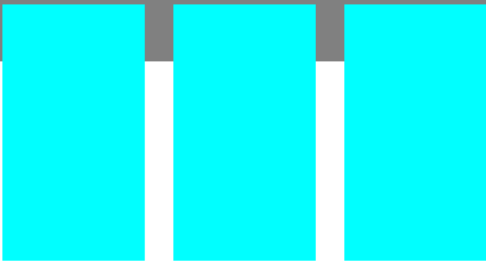


清除浮动

利用特性5，BFC内部的浮动元素也会参与高度计算，可以清除BFC内部的浮动

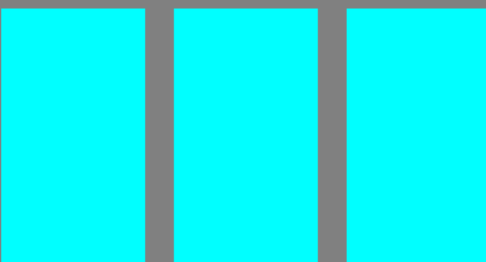
```
HTML
1 <div class="bfc">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>

CSS
1 .box {
2   float: left;
3   width: 100px;
4   height: 180px;
5   margin: 10px;
6   background: cyan;
7 }
8
9 .bfc {
10  /* 计算 BFC 高度时，浮动元素也会参与计算，
    可以利用这一点清除 BFC 内部的浮动 */
11  /* 尝试去掉 overflow 的注释，看效果 */
12  /* overflow: auto; */
13  min-height: 50px;
14  background: gray;
15 }
```



```
HTML
1 <div class="bfc">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>

CSS
1 .box {
2   float: left;
3   width: 100px;
4   height: 180px;
5   margin: 10px;
6   background: cyan;
7 }
8
9 .bfc {
10  /* 计算 BFC 高度时，浮动元素也会参与计算，
    可以利用这一点清除 BFC 内部的浮动 */
11  /* 尝试去掉 overflow 的注释，看效果 */
12  overflow: auto;
13  min-height: 50px;
14  background: gray;
15 }
```



参考链接

[深入理解CSS选择器优先级](#)

[做了一份前端面试复习计划，保熟~](#)

[你真的了解回流和重绘吗](#)

[可能是最好的BFC解析了...](#)