44 重点布局方案（下）

更新时间：2020-06-12 09:51:34

> 立志是事业的大门，工作是登门入室的的旅途。——巴斯德

上一节我们解决了一波关键的手写代码问题，本节我们会帮大家拆解一些具备一定理论性的 CSS 难点。大家不要背答案，记住大致的思路，面试时能够言之有物即可。

BFC 是什么？如何创造一个 BFC？

来了来了，有区分度的问题来了！BFC 相关的理解，堪称大厂 CSS 面试必问，大家一定好好把握。

理解 BFC

首先来看第一个问题：BFC 是什么？大家注意，解释 BFC 的时候，最好不要尝试从定义的角度去解释它。因为它的定义非常难以说清楚，比如 MDN 上就是这样定义的：

> 块格式化上下文（Block Formatting Context，BFC） 是Web页面的可视化CSS渲染的一部分，是块盒子的布局过程发生的区域，也是浮动元素与其他元素交互的区域。

来，告诉我，看着这个定义，你能明白 BFC 是啥吗？不仅你整不明白，面试官也多半会被这一长串的术语搞得云里雾里的。不过安全起见我这里还是给出一个相对好懂一点的说法：

BFC 就是一个**作用范围**。可以把它理解成是一个**独立**的容器。注意这个“独立”，它意味着这个作用范围和外界是毫不相关的。

在定义方面，有上面这句话就够了。不过大家要知道，我们前端是一个和工程结合得特别紧密的一个工种。解释这个问题，从定义下手不是个好思路，我们应该从**特性**下手。也就是说，**直接去解释 BFC 能解决什么问题、能够用于什么场景**。

BFC 主要被用来解决以下常见的布局问题：

- 清除浮动；
- 阻止 margin 发生重叠；
- 阻止元素被浮动的元素覆盖。

接下来，我们会结合一系列的实例来认知 BFC 的这些能力。在此之前，我们先列举几个**常见的创建 BFC 的方法**：

- float 的值不是 none；
- position 的值不是 static 或者 relative；
- display 的值是 inline-block、table-cell、flex、table-caption 或者 inline-flex；
- overflow 的值不是 visible。

注意：创建 BFC 的方法还有很多，这里是使用频率较高的几种。还是一样，大家不必追求数量哈。

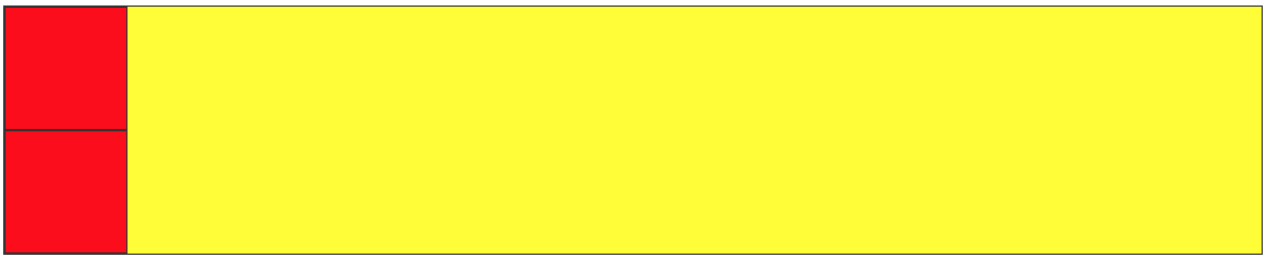
BFC 清除浮动

关于 BFC 清除浮动这一点，代码写起来没有什么难度，关键大家要理解：为什么 BFC 可以用来清除浮动。

我们先来回顾一下元素浮动之后会带来什么效果：

```
<style>
  .floatBox1 {
    width: 100px;
    height: 100px;
    float: left;
    border: 1px solid #333;
    background: red;
  }

  .floatBox2 {
    width: 100px;
    height: 100px;
    float: left;
    border: 1px solid #333;
    background: red;
  }

  .box {
    background: yellow;
    border: 1px solid #333;
  }
</style>

<body>
  <div class="box">
    <div class="floatBox1"></div>
    <div class="floatBox2"></div>
  </div>
</body>
```

楼上这段代码，我们浮动了两个盒子。大家可以尝试把浮动相关的代码先注释掉，先看看浮动前的样子，浮动前效果是这样的：



浮动后：

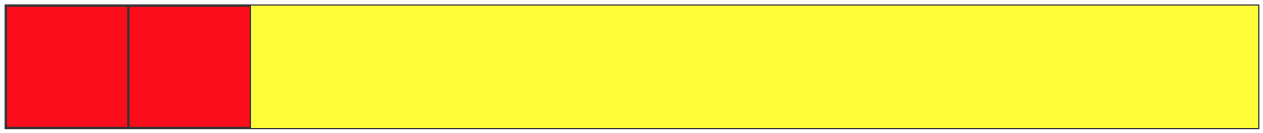
虽然现在有 Flex 大家基本都不怎么用浮动布局了，不过还是要清楚浮动布局应用的一个基本目的：它是为了改变应用了 float 的那些**目标元素的布局方式**。在这里，父元素的宽高也被间接地改变了，这是浮动布局的一个副作用。

解决这个副作用，有一个非常成熟的方案叫做“清除浮动”。创建 BFC 就是清除浮动的一种方法。

我们随机选取以上罗列的几种方法中的一种，这里我设置了 overflow 属性：

```
.box {
    background: yellow;
    border: 1px solid #333;
    overflow: hidden;
  }
```

成功创建了一个 BFC，此时效果如下：



我们看到，在 BFC 的帮助下，我们既实现了两个子元素 `float: left` 的布局效果，又避免了对父元素的影响。

那么回过头来看，为什么 BFC 可以解决浮动问题呢？答案就在 BFC 这个区域的**独立**特性上。因为独立，所以它要确保自己的作用范围不会对外界产生影响。此处我们把父元素创建为了一个 BFC，那么父元素就要保证自己的子元素不会跑出去给别人添麻烦，所以会自动把自己的宽高适应到能囊括子元素的程度。

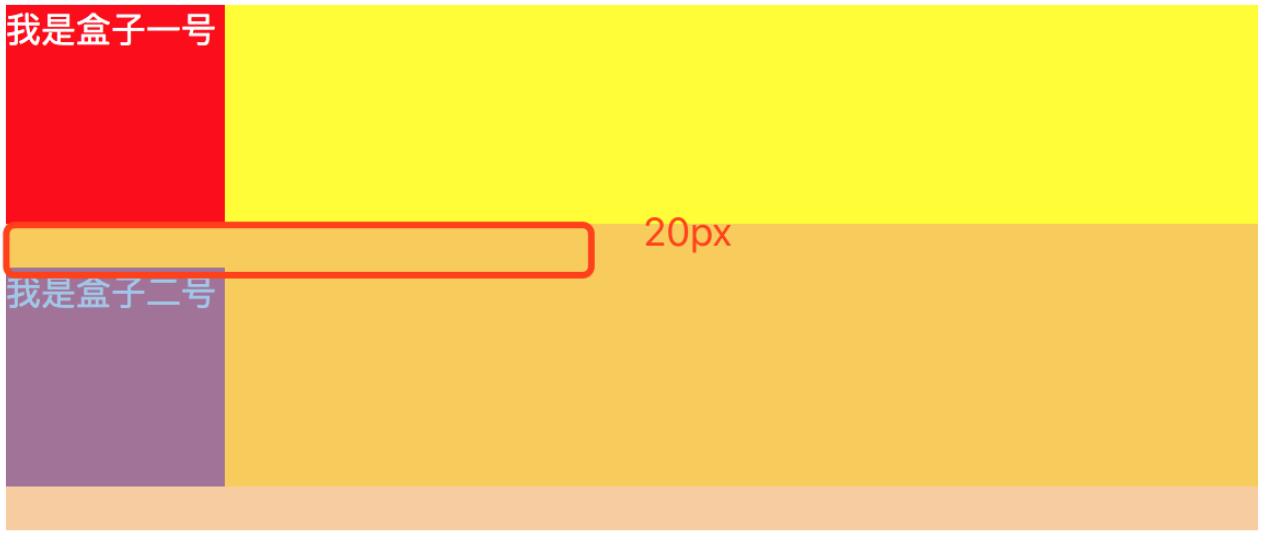
BFC 解决重叠边距问题

我们先来看看啥是外边距重叠：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>边距重叠测试</title>
  <style>
    .box {
      background-color: yellow;
    }

    .box1,
    .box2 {
      padding: 0;
      width: 100px;
      height: 100px;
      margin: 20px 0 20px 0;
      background-color: red;
      color: #fff;
    }
  </style>
</head>
<body>
  <div class="box">
    <div class="box1">我是盒子一号</div>
    <div class="box2">我是盒子二号</div>
  </div>
</body>
</html>
```

这段代码效果如下：

乍一看好像没什么问题，可我们仔细看看两个盒子之间的外边距：

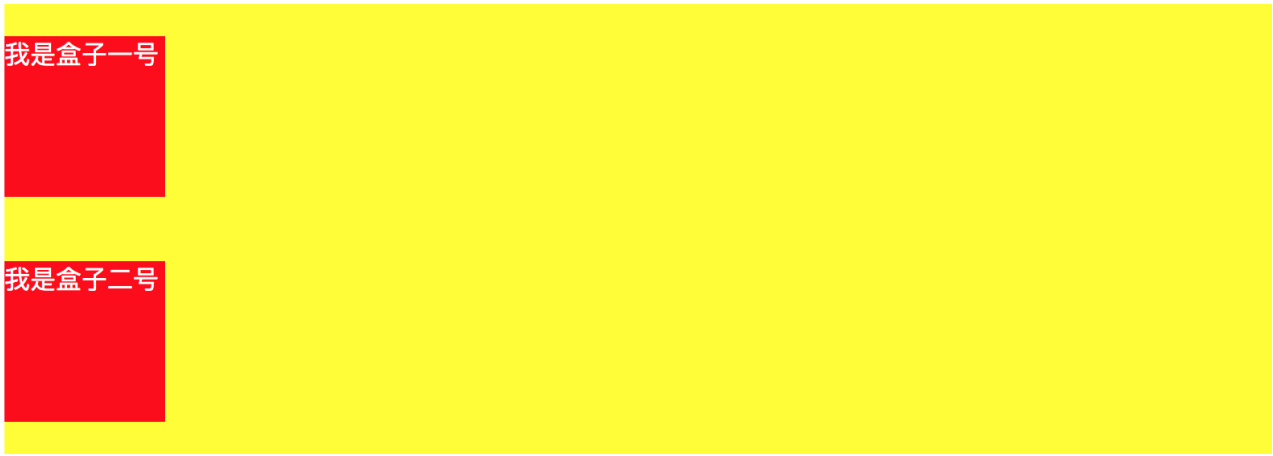


这里我选中了二号元素，变色的部分是二号元素本身和其外边距的占地面积。我们可以看出，盒子一号和盒子二号虽然都设置了 20px 的外边距，但是实际展示的只有一个 20px 的外边距。这是一个非常正常的现象：在CSS当中，相邻的两个盒子的外边距会被结合成一个单独的外边距。这种合并外边距的方式就叫**折叠**。

那如果我想要的效果不是折叠，而是每一个元素的外边距都能实际的生效呢？这时候 BFC 又能派上用场了，我给两个盒子各创建一个父元素，然后让这个父元素是一个 BFC：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>边距重叠测试</title>
  <style>
    .box {
      background-color: yellow;
    }

    .box1,
    .box2 {
      padding: 0;
      width: 100px;
      height: 100px;
      margin: 20px 0 20px 0;
      background-color: red;
      color: #fff;
    }

    .box-container {
      overflow: hidden;
    }
  </style>
</head>
<body>
  <div class="box">
    <div class="box-container">
      <div class="box1">我是盒子一号</div>
    </div>
    <div class="box-container">
      <div class="box2">我是盒子二号</div>
    </div>
  </div>
</body>
</html>
```

效果就会变成这样：



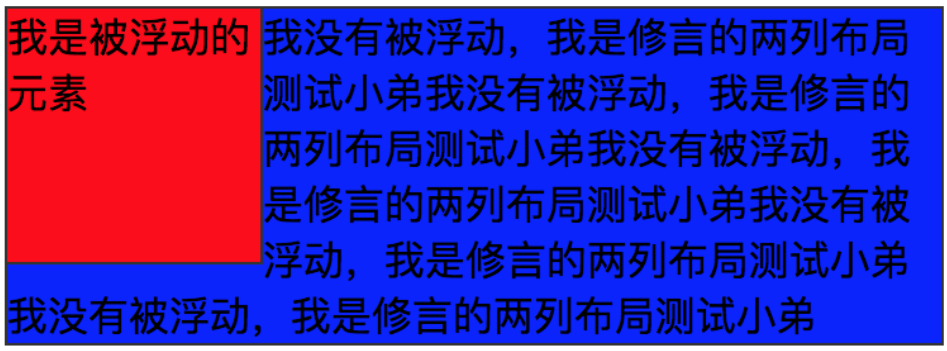
细心的同学会发现，此时不仅两个兄弟元素之间的外边距生效了，连子元素和父元素之间的外边距也生效了。这是因为早先父子元素间其实也存在边距重叠，也就是说边距重叠条件里的“相邻”不只是指兄弟元素之间的相邻关系，还包括了父子元素之间的相邻关系。通过把两个子元素放进两个不同的 BFC 里，利用 BFC “独立”的特性，可以使两个子元素的所有内容都被严严实实地包进各自的 BFC 中，避免与外界产生重叠的关系。

阻止元素被浮动的元素覆盖

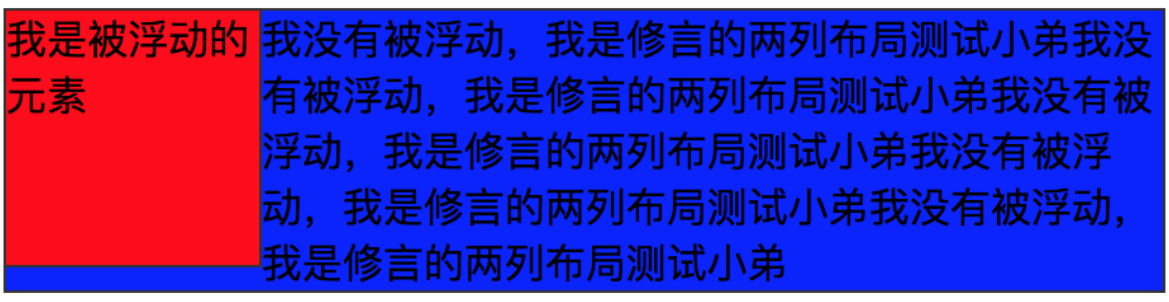
CSS 中有一种常见的布局效果：两列布局，一列宽度固定，另一列自适应。实现这种效果，有一条路就是基于浮动来做。这里我给大家做个示范：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=`, initial-scale=1.0">
  <title>两列布局自适应</title>
  <style>
    .floatBox1 {
      width: 100px;
      height: 100px;
      float: left;
      border: 1px solid #333;
      background: red;
    }

    .box2 {
      height:100%;
      border: 1px solid #333;
      background: blue;
    }
  </style>
  <body>
      <div class="floatBox1">我是被浮动的元素</div>
      <div class="box2">
        我没有被浮动，我是修言的两列布局测试小弟我没有被浮动，我是修言的两列布局测试小弟我没有被浮动，我是修言的两列布局测试小弟我没有被浮动，我是修言的两列布局测试小弟我没有被浮动，我是修言的两列布局测试小弟我没有被浮动，我是修言的两列布局测试小弟</div>
  </body>
</head>
<body>

</body>
</html>
```

这个例子对应的效果是这样的：



我们看到，右边这列自适应确实是实现了，比如我如果拉伸浏览器窗口，它就会跟着变宽：

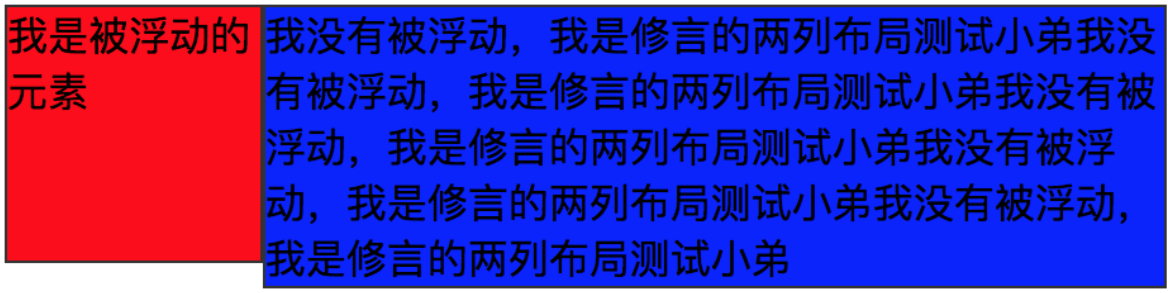


但是“两列”却是个问题，左侧的盒子因为使用了浮动布局，它会脱离原有的文档流，“浮”在普通文档流的上方。这样一来，就遮盖了没有浮动的元素。

B 元素被 A 元素覆盖，这意味着什么？这意味着 B 元素被外界影响了！一个 BFC 的基本修养是什么？是高度的独立性，不被外界的布局干扰。因此，要想解决覆盖问题，我们为 B 元素（这里就是蓝色盒子）创建一个 BFC：

```
.box2 {
  height:100%;
  border: 1px solid #333;
  background: blue;
  overflow: hidden;
}
```

如此一来，浮动的红色盒子就再也不能拿我们具备 BFC 特性的蓝色盒子怎么样了：



谈谈你对 CSS 工程化的理解

来了来了，有区分度的第二个问题来了。前端基本每个人都会写 CSS，可并不是每个人都了解 CSS 工程化。工程化相关问题，往往出现在二面/三面/各种项目面试中，考察候选人 CSS 相关实践的深度。

这个问题的答案是灵活的，我这里给出普适性较强的几个方向。如果你对 CSS 工程化确实有过深入的实践，那么更建议你在这些方向的基础上，结合自己的项目经历进行扩充，切勿强记答案。

首先，大家要清楚的是，CSS 工程化是为了解决哪些方面的问题：

1. 宏观设计：CSS 代码如何组织、如何拆分、模块结构怎样设计？
2. 编码优化：怎样写出更好的 CSS？
3. 构建：如何处理我的 CSS，才能让它的打包结果最优？
4. 可维护性：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

大家首先可以就这四个方面，回溯自己的 CSS 开发经历，看有没有可以匹配上其中某个方面的点。很多时候，你不是没有做过工程化，而是你不清楚自己做的事情就是工程化。如果实在没有，也没关系，我们至少有三个点是可以和面试官聊下去的：

- 预处理器：Less、 Sass 等；
- 重要的工程化插件： PostCss；
- Webpack loader 等 。

这三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践。基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。大家如果是初识预处理器，你可以类比 JS 世界的 Vue、React来理解它。 预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码：

为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这个道理就和我们本来用 JS 也可以一把梭实现所有功能，但最后却一窝蜂跑去写 React 的 jsx 或者 Vue 的模板语法是一样的——为了爽！

要想知道有了预处理器有多爽，我们首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高，前端工程中对 CSS 提出了以下的诉求：

1. 宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；
2. 编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；
3. 可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器五花八门，它们普遍会具备这样的特性：

- 嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系 ；
- 支持定义 css 变量；
- 提供计算函数；
- 允许对代码片段进行 extend 和 mixin；
- 支持循环语句的使用；
- 支持将 CSS 文件模块化，实现复用。

以上这些，大概率就是你在面试环节需要口头表述给面试官的全部。我这里基本是在给大家写答案了，安全起见，各位还是需要对市面上最常见的两种预处理器：Sass、Less 有所了解。不用你写多么复杂的项目，只需要你花上几个小时的时间，跟着 less 文档 和 sass 文档 把关键的特性手动敲一遍，然后你再回来看我上面写的这些字，一定会有一种恍然大悟的感觉。

PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss？

PostCss 仍然是一个对 CSS 进行解析和处理的工具，它会对 CSS 做这样的事情：

它和预处理器的不同就在于，预处理器处理的是 类CSS，而 PostCss 处理的就是 CSS 本身。

理解 PostCss，我们可以类比 JS 世界里的 Babel。大家知道，Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多，这里我列举几个最高频的：

- 提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；
- 当我们的 CSS 代码需要适配低版本浏览器时，PostCss 的 Autoprefixer 插件可以帮助我们自动增加浏览器前缀；
- 允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

和预处理器一样，这里也建议大家在掌握面试所需的这些知识点之余，亲自去试用一下 PostCss，相信你的记忆和理解都会更深一层。

Webpack 能处理 CSS 吗？如何实现？

这里有两个问题，我们一个一个来看：

Webpack 能处理 CSS 吗？这个问题回答时要尽量严谨，我建议大家这样来答：

1. **Webpack 在裸奔的状态下，是不能处理 CSS 的**，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；
2. Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理？

大家不要想太多，这个问题唯一的目的就是考察你到底有没有真的在综合性项目中用过　Webpack。这道题我们需要答出两个点：

> Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader
>
> 注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：

-

  - css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；
  - style-loader：创建style标签，把 CSS 内容写入标签。

在实际使用中，**大家要切记 css-loader 的执行顺序一定要安排在 style-loader 的前面**。因为只有完成了编译过程，才可以对 css 代码进行插入；若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的，它会无情报错。

}

43 重点布局方案（上）

45 响应式布局原理与实践（上）