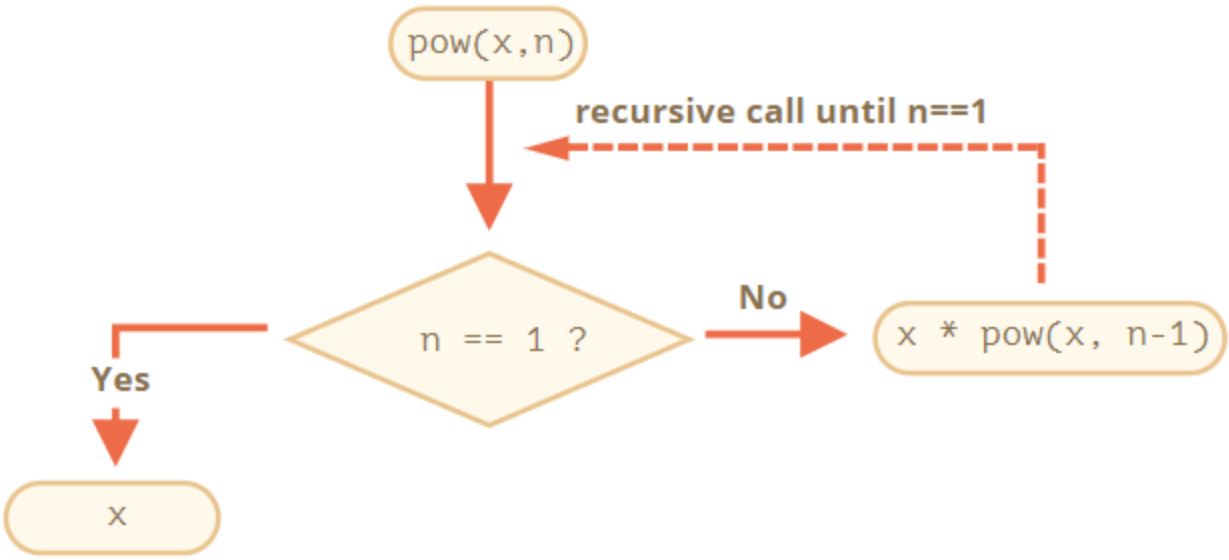
```
JavaScript | 复制代码

1                if n==1  = x
2                /
3   pow(x, n) =
4                \
5                 else     = x * pow(x, n - 1)
```

也就是说 `pow` 递归地调用自身 直到 `n == 1`



为了计算 `pow(2, 4)`，递归变体经过了下面几个步骤：

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

因此，递归将函数调用简化为一个更简单的函数调用，然后再将其简化为一个更简单的函数，以此类推，直到结果

35.2. 尾递归

尾递归，即在函数尾位置调用自身（或是一个尾调用本身的其他函数等等）。尾递归也是递归的一种特殊情形。尾递归是一种特殊的尾调用，即在尾部直接调用自身的递归函数

尾递归在普通尾调用的基础上，多出了2个特征：

- 在尾部调用的是函数自身
- 可通过优化，使得计算仅占用常量栈空间

在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储，递归次数过多容易造成栈溢出

这时候，我们就可以使用尾递归，即一个函数中所有递归形式的调用都出现在函数的末尾，对于尾递归来说，由于只存在一个调用记录，所以永远不会发生"栈溢出"错误

实现一下阶乘，如果用普通的递归，如下：

JavaScript 复制代码

```
function factorial(n) {
  if (n === 1) return 1;
  return n * factorial(n - 1);
}

factorial(5) // 120
```

如果 `n` 等于5，这个方法要执行5次，才返回最终的计算表达式，这样每次都要保存这个方法，就容易造成栈溢出，复杂度为 `O(n)`

如果我们使用尾递归，则如下：

JavaScript 复制代码

```
function factorial(n, total) {
  if (n === 1) return total;
  return factorial(n - 1, n * total);
}

factorial(5, 1) // 120
```

可以看到，每一次返回的就是一个新的函数，不带上一个函数的参数，也就不需要储存上一个函数了。尾递归只需要保存一个调用栈，复杂度 O(1)

35.3. 应用场景

数组求和

JavaScript 复制代码

```javascript
function sumArray(arr, total) {
    if(arr.length === 1) {
        return total
    }
    return sum(arr, total + arr.pop())
}
```

使用尾递归优化求斐波那契数列

JavaScript 复制代码

```javascript
function factorial2 (n, start = 1, total = 1) {
    if(n <= 2){
        return total
    }
    return factorial2 (n -1, total, total + start)
}
```

数组扁平化

JavaScript 复制代码

```javascript
let a = [1,2,3, [1,2,3, [1,2,3]]]
// 变成
let a = [1,2,3,1,2,3,1,2,3]
// 具体实现
function flat(arr = [], result = []) {
    arr.forEach(v => {
        if(Array.isArray(v)) {
            result = result.concat(flat(v, []))
        }else {
            result.push(v)
        }
    })
    return result
}
```

数组对象格式化

JavaScript | 复制代码

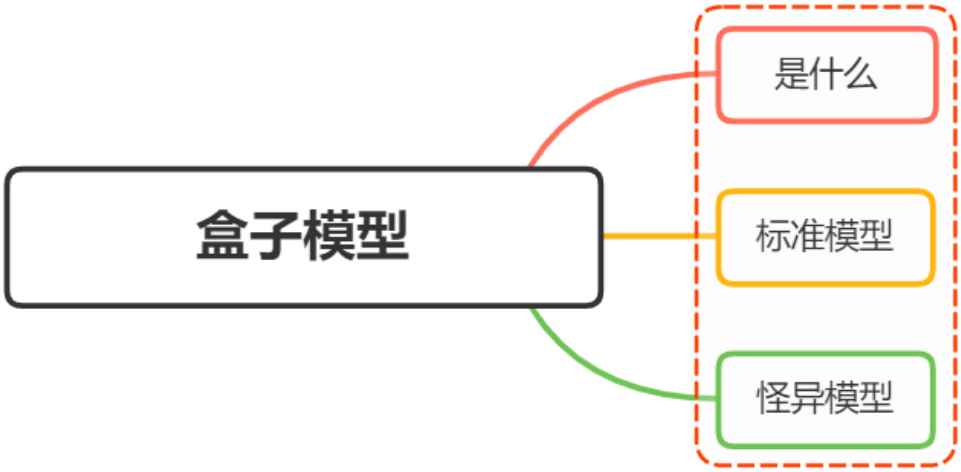
```javascript
let obj = {
    a: '1',
    b: {
        c: '2',
        D: {
            E: '3'
        }
    }
}
// 转化为如下:
let obj = {
    a: '1',
    b: {
        c: '2',
        d: {
            e: '3'
        }
    }
}

// 代码实现
function keysLower(obj) {
    let reg = new RegExp("([A-Z]+)", "g");
    for (let key in obj) {
        if (obj.hasOwnProperty(key)) {
            let temp = obj[key];
            if (reg.test(key.toString())) {
                // 将修改后的属性名重新赋值给temp，并在对象obj内添加一个转换后的属性
                temp = obj[key.replace(reg, function (result) {
                    return result.toLowerCase()
                })] = obj[key];
                // 将之前大写的键属性删除
                delete obj[key];
            }
            // 如果属性是对象或者数组，重新执行函数
            if (typeof temp === 'object' || Object.prototype.toString.call(temp) === '[object Array]') {
                keysLower(temp);
            }
        }
    }
    return obj;
};
```

35.4.

CSS面试真题（20题）

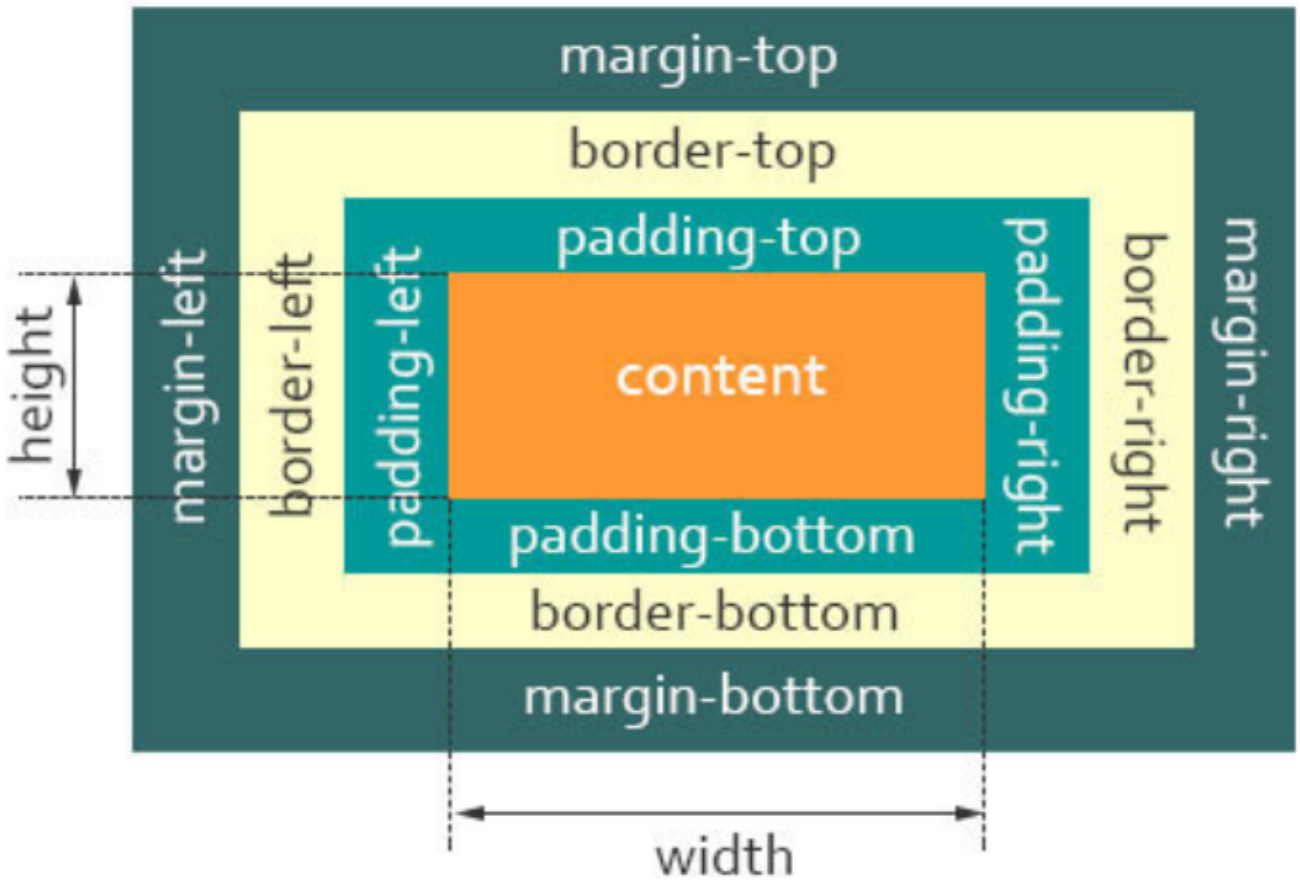
1. 说说你对盒子模型的理解?



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`



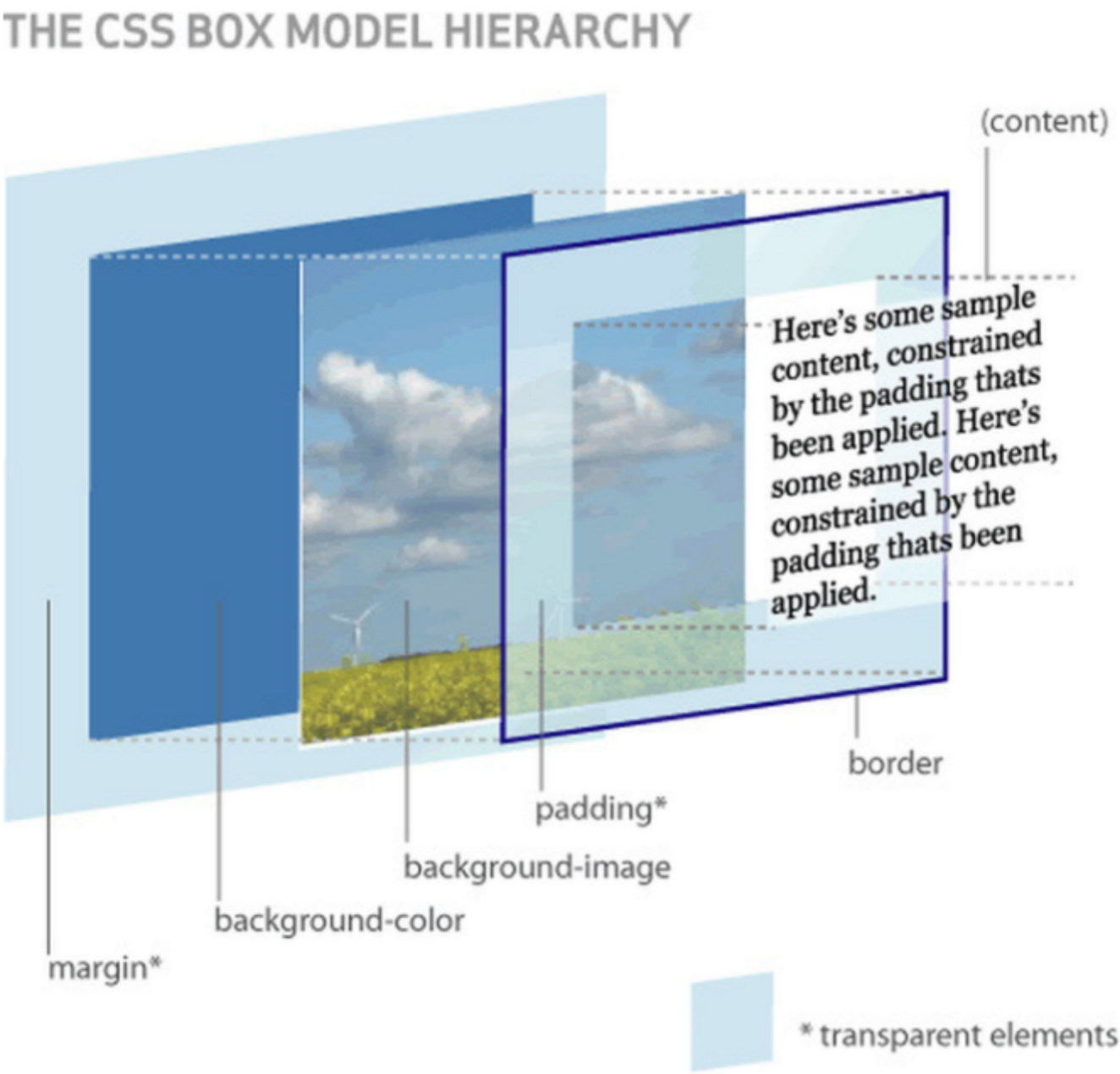
`content`，即实际内容，显示文本和图像

`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：

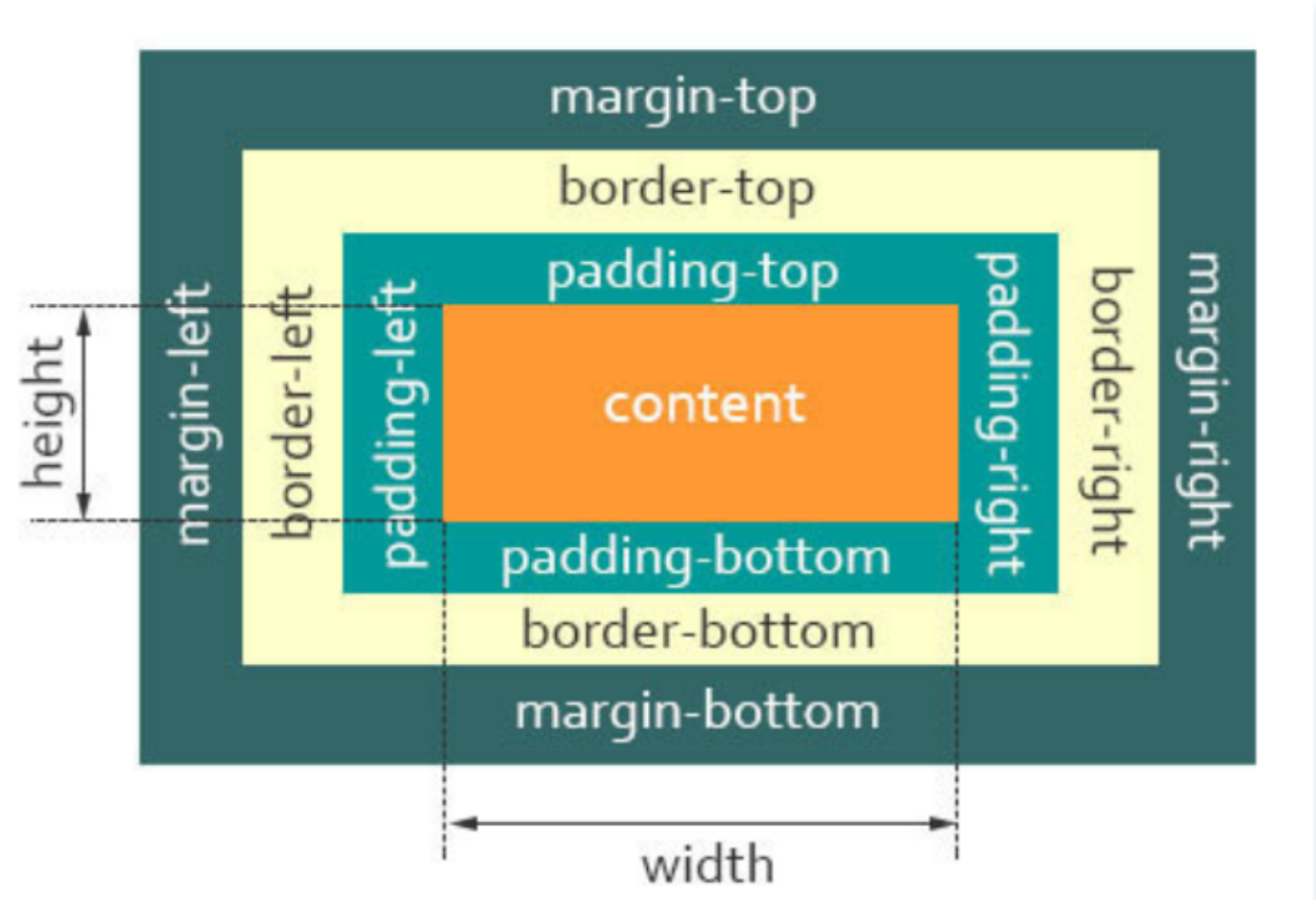
- W3C 标准盒子模型
- IE 怪异盒子模型

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：

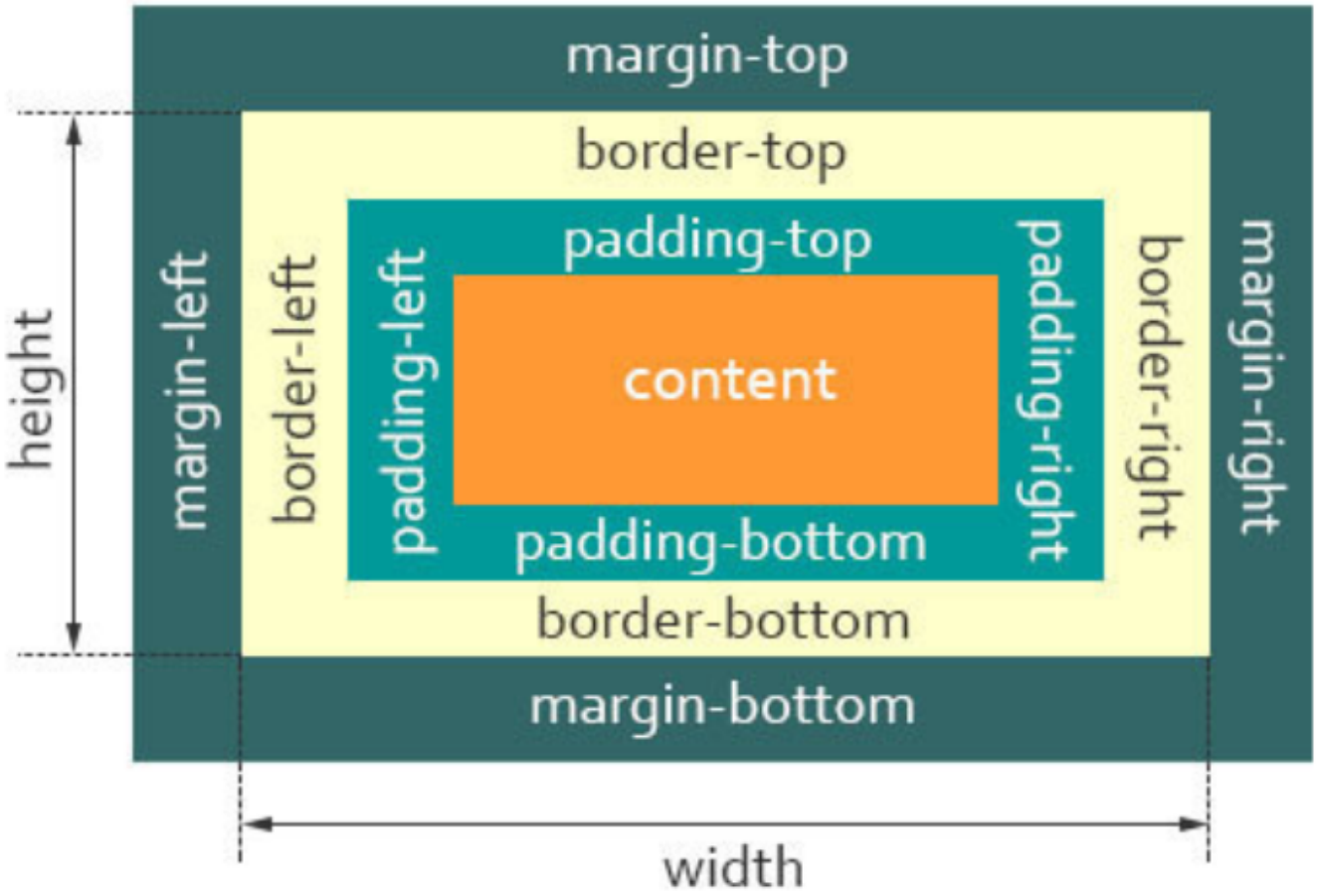
- 盒子总宽度 = width + padding + border + margin;
- 盒子总高度 = height + padding + border + margin

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 | 复制代码

```
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

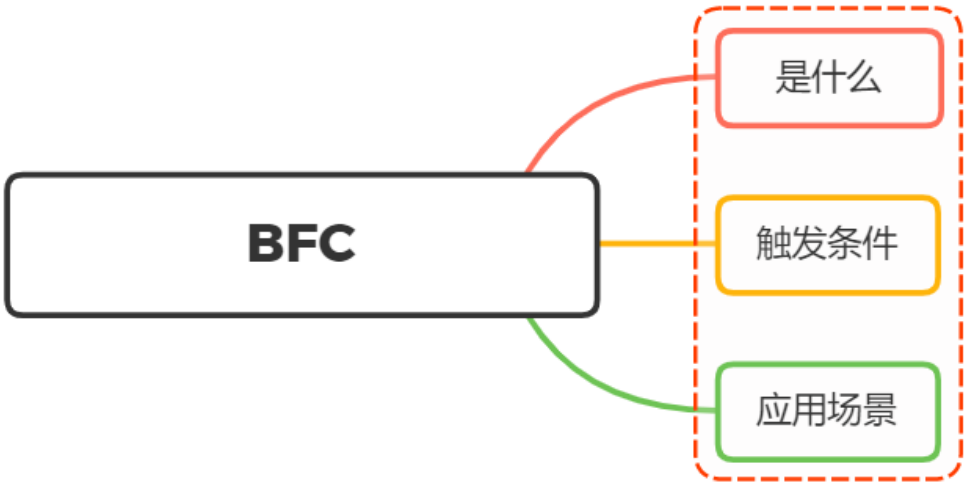
- border-box 元素的 width/height 包含 padding，border，与怪异盒子模型表现一致
- inherit 指定 box-sizing 属性的值，应该从父元素继承

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

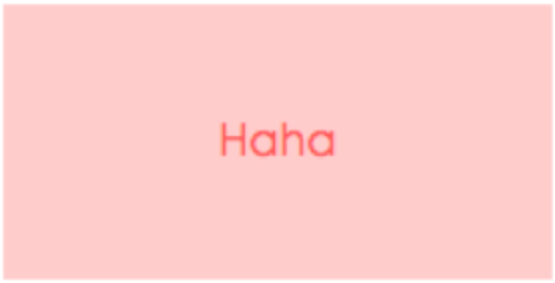
利用 `BFC` 的特性，我们将 `BFC` 应用在以下场景：

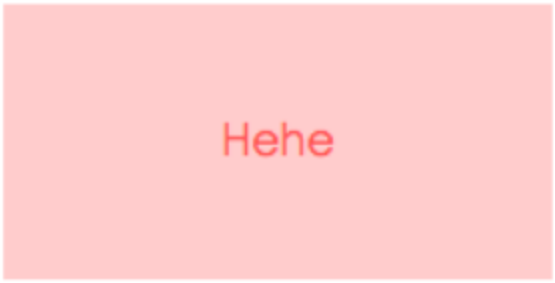
2.3.1. 防止margin重叠（塌陷）

HTML | 复制代码

```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。

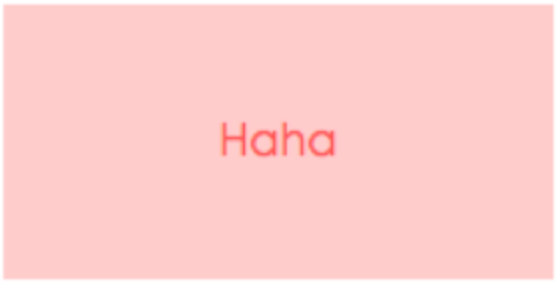
前面讲到，同一个 `BFC` 的俩个相邻的盒子的 `margin` 会发生重叠

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 | 复制代码

```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>
```

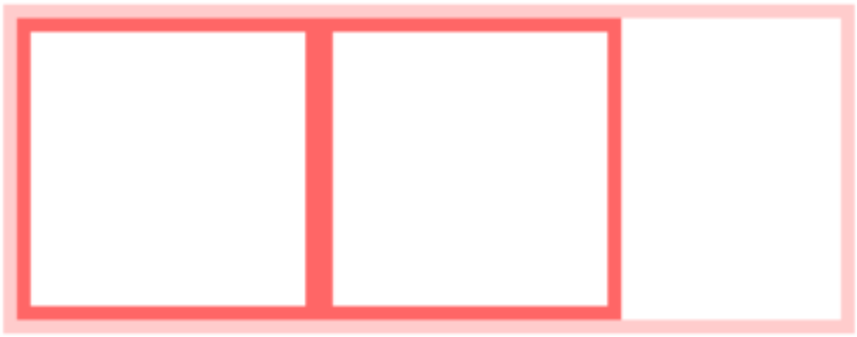
页面显示如下：

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 | 复制代码

```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

```html
<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

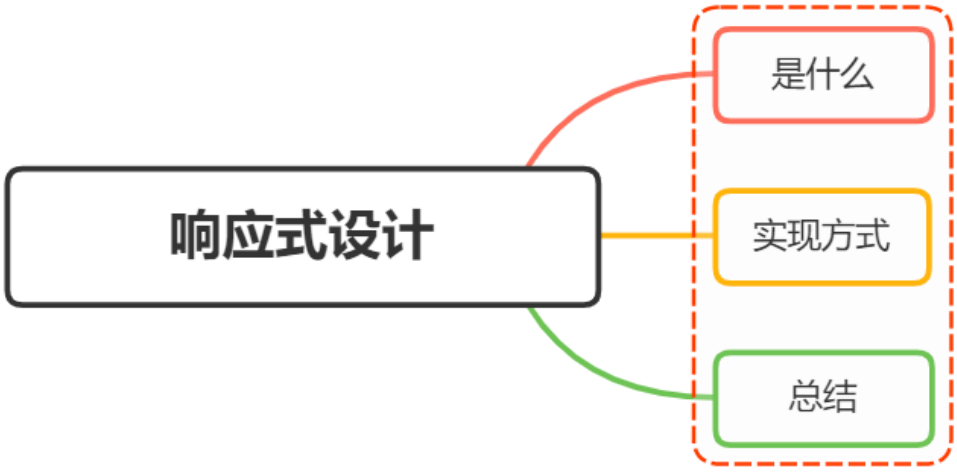
CSS | 复制代码

```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

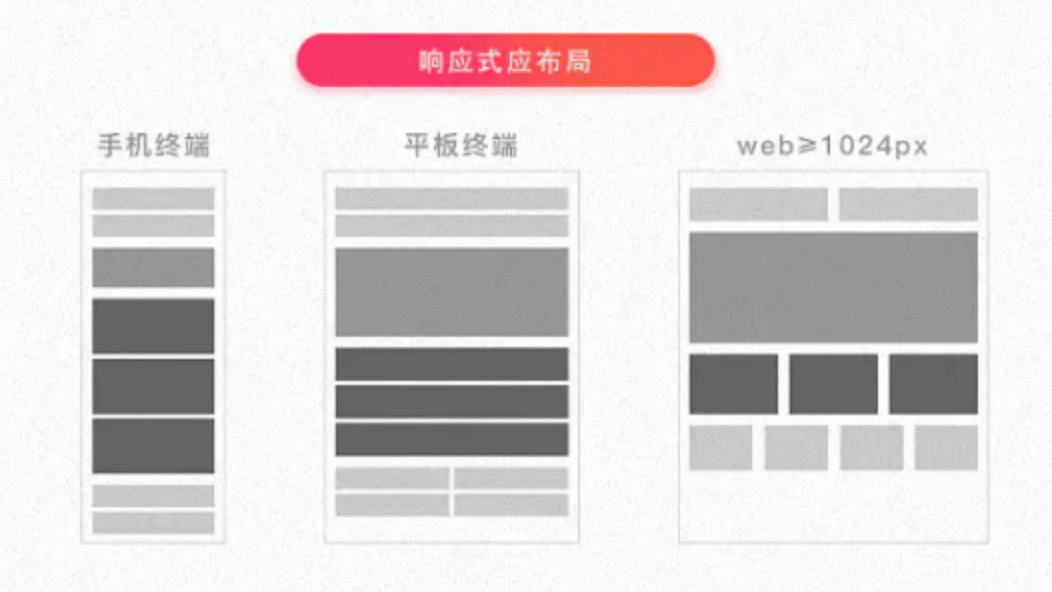
描述响应式界面最著名的一句话就是“Content is like water”

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 复制代码

```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 复制代码

```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 | 复制代码

```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