1. **关键技术**

1. 从使用 Zepto.js 到使用 Vue-cli 脚手架搭建单页面应用。

2. 在 Vue.js 构建的单页面应用中，使用 VueRouter 作前端路由管理器、Vuex 进行状态管理、axios http库， 并在项目中运用 ES6 的解构、箭头函数、模板字符串等新特性、使用 async 函数解决异步问题；利用 npm 命令和配置 webpack 打包静态资源；实现前后端分离。

3. 熟练使用 CSS3 和动画库实现各种动效，以及利用 Flex 实现弹性布局。

4. 熟练使用 Fiddler 抓包，调试接口数据。

5. 熟练使用 jQuery 实现多级菜单展示、订阅、树型部门结构操作等多种功能，熟练使用 AJAX 技术。

6.熟练使用 Echarts、Highcharts 进行数据可视化的图表展示

1. **AMD与CMD区别**

AMD 即Asynchronous Module Definition，中文名是异步模块定义的意思。它是一个在浏览器端模块化开发的规范

CMD 即Common Module Definition通用模块定义，CMD规范是国内发展出来的，就像AMD有个requireJS，CMD有个浏览器的实现SeaJS，SeaJS要解决的问题和requireJS一样，只不过在模块定义方式和模块加载（可以说运行、解析）时机上有所不同

关于这两个的区别网上可以搜出一堆文章，简单总结一下

最明显的区别就是在模块定义时对依赖的处理不同

1、AMD推崇依赖前置，在定义模块的时候就要声明其依赖的模块   
2、CMD推崇就近依赖，只有在用到某个模块的时候再去require   
这种区别各有优劣，只是语法上的差距，而且requireJS和SeaJS都支持对方的写法

AMD和CMD最大的区别是对依赖模块的执行时机处理不同，注意不是加载的时机或者方式不同

很多人说requireJS是异步加载模块，SeaJS是同步加载模块，这么理解实际上是不准确的，其实加载模块都是异步的，只不过AMD依赖前置，js可以方便知道依赖模块是谁，立即加载，而CMD就近依赖，需要使用把模块变为字符串解析一遍才知道依赖了那些模块，这也是很多人诟病CMD的一点，牺牲性能来带来开发的便利性，实际上解析模块用的时间短到可以忽略

为什么我们说两个的区别是依赖模块执行时机不同，为什么很多人认为ADM是异步的，CMD是同步的（除了名字的原因。。。）

同样都是异步加载模块，AMD在加载模块完成后就会执行改模块，所有模块都加载执行完后会进入require的回调函数，执行主逻辑，这样的效果就是依赖模块的执行顺序和书写顺序不一定一致，看网络速度，哪个先下载下来，哪个先执行，但是主逻辑一定在所有依赖加载完成后才执行

CMD加载完某个依赖模块后并不执行，只是下载而已，在所有依赖模块加载完成后进入主逻辑，遇到require语句的时候才执行对应的模块，这样模块的执行顺序和书写顺序是完全一致的

这也是很多人说AMD用户体验好，因为没有延迟，依赖模块提前执行了，CMD性能好，因为只有用户需要的时候才执行的原因

1. **RESTful**

一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

REST（英文：Representational State Transfer，简称REST）描述了一个架构样式的网络系统，比如 web 应用程序。

1. **BFC**

什么是BFC

在一个Web页面的CSS渲染中，块级格式化上下文 (Block Fromatting Context)是按照块级盒子布局的。W3C对BFC的定义如下：

浮动元素和绝对定位元素，非块级盒子的块级容器（例如 inline-blocks, table-cells, 和 table-captions），以及overflow值不为“visiable”的块级盒子，都会为他们的内容创建新的BFC（块级格式上下文）。

为了便于理解，我们换一种方式来重新定义BFC。一个HTML元素要创建BFC，则满足下列的任意一个或多个条件即可：

1、float的值不是none。

2、position的值不是static或者relative。

3、display的值是inline-block、table-cell、flex、table-caption或者inline-flex

4、overflow的值不是visible

BFC是一个独立的布局环境，其中的元素布局是不受外界的影响，并且在一个BFC中，块盒与行盒（行盒由一行中所有的内联元素所组成）都会垂直的沿着其父元素的边框排列。

怎么创建BFC

要显示的创建一个BFC是非常简单的，只要满足上述4个CSS条件之一就行。例如：

<div class="container">

你的内容

</div>

在类container中添加类似 overflow: scroll，overflow: hidden，display: flex，float: left，或 display: table 的规则来显示创建BFC。虽然添加上述的任意一条都能创建BFC，但会有一些副作用：

1、display: table 可能引发响应性问题

2、overflow: scroll 可能产生多余的滚动条

3、float: left 将把元素移至左侧，并被其他元素环绕

4、overflow: hidden 将裁切溢出元素

因而无论什么时候需要创建BFC，都要基于自身的需要来考虑。对于本文，将采用 overflow: hidden 方式：

.container {

overflow: hidden;

}

再说两点

BFC中盒子怎么对齐

如前文所说，在一个BFC中，块盒与行盒（行盒由一行中所有的内联元素所组成）都会垂直的沿着其父元素的边框排列。W3C给出得规范是：

在BFC中，每一个盒子的左外边缘（margin-left）会触碰到容器的左边缘(border-left)（对于从右到左的格式来说，则触碰到右边缘）。浮动也是如此（尽管盒子里的行盒子 Line Box 可能由于浮动而变窄），除非盒子创建了一个新的BFC（在这种情况下盒子本身可能由于浮动而变窄）。

img

外边距折叠

常规流布局时，盒子都是垂直排列，两者之间的间距由各自的外边距所决定，但不是二者外边距之和。

<div class="container">

<p>Sibling 1</p>

<p>Sibling 2</p>

</div>

对应的CSS：

.container {

background-color: red;

overflow: hidden; /\* creates a block formatting context \*/

}

p {

background-color: lightgreen;

margin: 10px 0;

}

渲染结果如图：

img2

在上图中，一个红盒子（div）包含着两个兄弟元素（p），一个BFC已经创建了出来。

理论上，两个p元素之间的外边距应当是二者外边距之和（20px）但实际上却是10px，这是外边距折叠(Collapsing Margins)的结果。

在CSS当中，相邻的两个盒子（可能是兄弟关系也可能是祖先关系）的外边距可以结合成一个单独的外边距。这种合并外边距的方式被称为折叠，并且因而所结合成的外边距称为折叠外边距。折叠的结果按照如下规则计算：

1、两个相邻的外边距都是正数时，折叠结果是它们两者之间较大的值。

2、两个相邻的外边距都是负数时，折叠结果是两者绝对值的较大值。

3、两个外边距一正一负时，折叠结果是两者的相加的和。

产生折叠的必备条件：margin必须是邻接的! (对于不产生折叠的情况，见参考文章的链接)

BFC可以做什么呢？

利用BFC避免外边距折叠

BFC可能造成外边距折叠，也可以利用它来避免这种情况。BFC产生外边距折叠要满足一个条件：两个相邻元素要处于同一个BFC中。所以，若两个相邻元素在不同的BFC中，就能避免外边距折叠。

改进前面的例子：

<div class="container">

<p>Sibling 1</p>

<p>Sibling 2</p>

<p>Sibling 3</p>

</div>

对应的CSS：

.container {

background-color: red;

overflow: hidden; /\* creates a block formatting context \*/

}

p {

background-color: lightgreen;

margin: 10px 0;

}

结果和上面一样，由于外边距折叠，三个相邻P元素之间的垂直距离是10px，这是因为三个 p 标签都从属于同一个BFC。

修改第三个P元素，使之创建一个新的BFC：

<div class="container">

<p>Sibling 1</p>

<p>Sibling 2</p>

<div class="newBFC">

<p>Sibling 3</p>

</div>

</div>

对应的CSS：

.container {

background-color: red;

overflow: hidden; /\* creates a block formatting context \*/

}

p {

margin: 10px 0;

background-color: lightgreen;

}

.newBFC {

overflow: hidden; /\* creates new block formatting context \*/

}

现在的结果如图：

img

因为第二个和第三个P元素现在分属于不同的BFC，它们之间就不会发生外边距折叠了。

BFC包含浮动

浮动元素是会脱离文档流的(绝对定位元素会脱离文档流)。如果一个没有高度或者height是auto的容器的子元素是浮动元素，则该容器的高度是不会被撑开的。我们通常会利用伪元素(:after或者:before)来解决这个问题。BFC能包含浮动，也能解决容器高度不会被撑开的问题。

img

看个例子：

<div class="container">

<div>Sibling</div>

<div>Sibling</div>

</div>

CSS：

.container {

background-color: green;

}

.container div {

float: left;

background-color: lightgreen;

margin: 10px;

}

在上面这个例子中，容器没有任何高度，并且它包不住浮动子元素，容器的高度并不会被撑开。为解决这个问题，可以在容器中创建一个BFC：

.container {

overflow: hidden; /\* creates block formatting context \*/

background-color: green;

}

.container div {

float: left;

background-color: lightgreen;

margin: 10px;

}

现在容器可以包住浮动子元素，并且其高度会扩展至包住其子元素，在这个新的BFC中浮动元素又回归到页面的常规流之中了。

使用BFC避免文字环绕

img

如上图所示，对于浮动元素，可能会造成文字环绕的情况(Figure1)，但这并不是我们想要的布局(Figure2才是想要的)。要解决这个问题，我们可以用外边距，但也可以用BFC。

First let us understand why the text wraps. For this we have to understand how the box model works when an element is floated. This is the part I left earlier while discussing the alignment in a block formatting context. Let us understand what is happening in Figure 1 in the diagram below:

img

假设HTML是：

<div class="container">

<div class="floated">

Floated div

</div>

<p>

Quae hic ut ab perferendis sit quod architecto,

dolor debitis quam rem provident aspernatur tempora

expedita.

</p>

</div>

上图整个黑色区域表示 p 元素。p 元素没有移位但它叠在了浮动元素之下，而p元素的文本(行盒子)却移位了，行盒子水平变窄来给浮动元素腾出了空间。随着文本的增加，最后文本将环绕在浮动元素之下，因为那时候行盒子不再需要移位，也就成了图Figure1的样子。

再回顾一下W3C的描述：

在BFC上下文中，每个盒子的左外侧紧贴包含块的左侧（从右到左的格式里，则为盒子右外侧紧贴包含块右侧），甚至有浮动也是如此（尽管盒子里的行盒子 Line Box 可能由于浮动而变窄），除非盒子创建了一个新的BFC（在这种情况下盒子本身可能由于浮动而变窄）。

因而，如果p元素创建一个新的BFC那它就不会再紧贴包含块的左侧了。

在多列布局中使用BFC

如果我们创建一个占满整个容器宽度的多列布局，在某些浏览器中最后一列有时候会掉到下一行。这可能是因为浏览器四舍五入了列宽从而所有列的总宽度会超出容器。但如果我们在多列布局中的最后一列里创建一个新的BFC，它将总是占据其他列先占位完毕后剩下的空间。

例如：

<div class="container">

<div class="column">column 1</div>

<div class="column">column 2</div>

<div class="column">column 3</div>

</div>

对应的CSS：

.column {

width: 31.33%;

background-color: green;

float: left;

margin: 0 1%;

}

/\* Establishing a new block formatting

context in the last column \*/

.column:last-child {

float: none;

overflow: hidden;

}

现在尽管盒子的宽度稍有改变，但布局不会打破。当然，对多列布局来说这不一定是个好办法，但能避免最后一列下掉。这个问题上弹性盒或许是个更好的解决方案，但这个办法可以用来说明元素在这些环境下的行为。

1. **用CSS实现一个简单的幻灯片效果页面**

.myDiv{

width:500px;

height:500px;

overflow: hidden;

background-position: center;

box-shadow:0 0 5px rgba(0,0,0,1);

-webkit-animation-name:'loop';

-webkit-animation-duration:10s;

-webkit-animation-iteration-count: infinite;

}

@-webkit-keyframes 'loop'{

0%{background:url(../slider/images/1.jpg) no-repeat;}

15%{background:url(../slider/images/2.jpg) no-repeat;}

30%{background:url(../slider/images/3.jpg) no-repeat;}

45%{background:url(../slider/images/4.jpg) no-repeat;}

60%{background:url(../slider/images/5.jpg) no-repeat;}

75%{background:url(../slider/images/6.jpg) no-repeat;}

100%{background:url(../slider/images/1.jpg) no-repeat;}

}

1. **call、apply、bind的区别**

// 有只猫叫小黑，小黑会吃鱼

const cat = {

name: '小黑',

eatFish(...args) {

console.log('this指向=>', this);

console.log('...args', args);

console.log(this.name + '吃鱼');

},

}

// 有只狗叫大毛，大毛会吃骨头

const dog = {

name: '大毛',

eatBone(...args) {

console.log('this指向=>', this);

console.log('...args', args);

console.log(this.name + '吃骨头');

},

}

console.log('=================== call =========================');

// 有一天大毛想吃鱼了，可是它不知道怎么吃。怎么办？小黑说我吃的时候喂你吃

cat.eatFish.call(dog, '汪汪汪', 'call')

// 大毛为了表示感谢，决定下次吃骨头的时候也喂小黑吃

dog.eatBone.call(cat, '喵喵喵', 'call')

console.log('=================== apply =========================');

cat.eatFish.apply(dog, ['汪汪汪', 'apply'])

dog.eatBone.apply(cat, ['喵喵喵', 'apply'])

console.log('=================== bind =========================');

// 有一天他们觉得每次吃的时候再喂太麻烦了。干脆直接教对方怎么吃

const test1 = cat.eatFish.bind(dog, '汪汪汪', 'bind')

const test2 = dog.eatBone.bind(cat, '喵喵喵', 'bind')

test1()

test2()