2022高频前端面试题汇总（附答案）

一、CSS篇

1、什么是 BFC

`BFC（Block Formatting Context）` 格式化上下文，是 Web 页面中盒模型布局的 CSS 渲染模式，指一个独立的渲染区域或者说是一个隔离的独立容器。

形成 BFC 的条件

- 浮动元素，float 除 none 以外的值
- 定位元素，position（absolute，fixed）
- display 为以下其中之一的值 inline-block，table-cell，table-caption
- overflow 除了 visible 以外的值（hidden，auto，scroll）
- HTML 就是一个 BFC

BFC 的特性

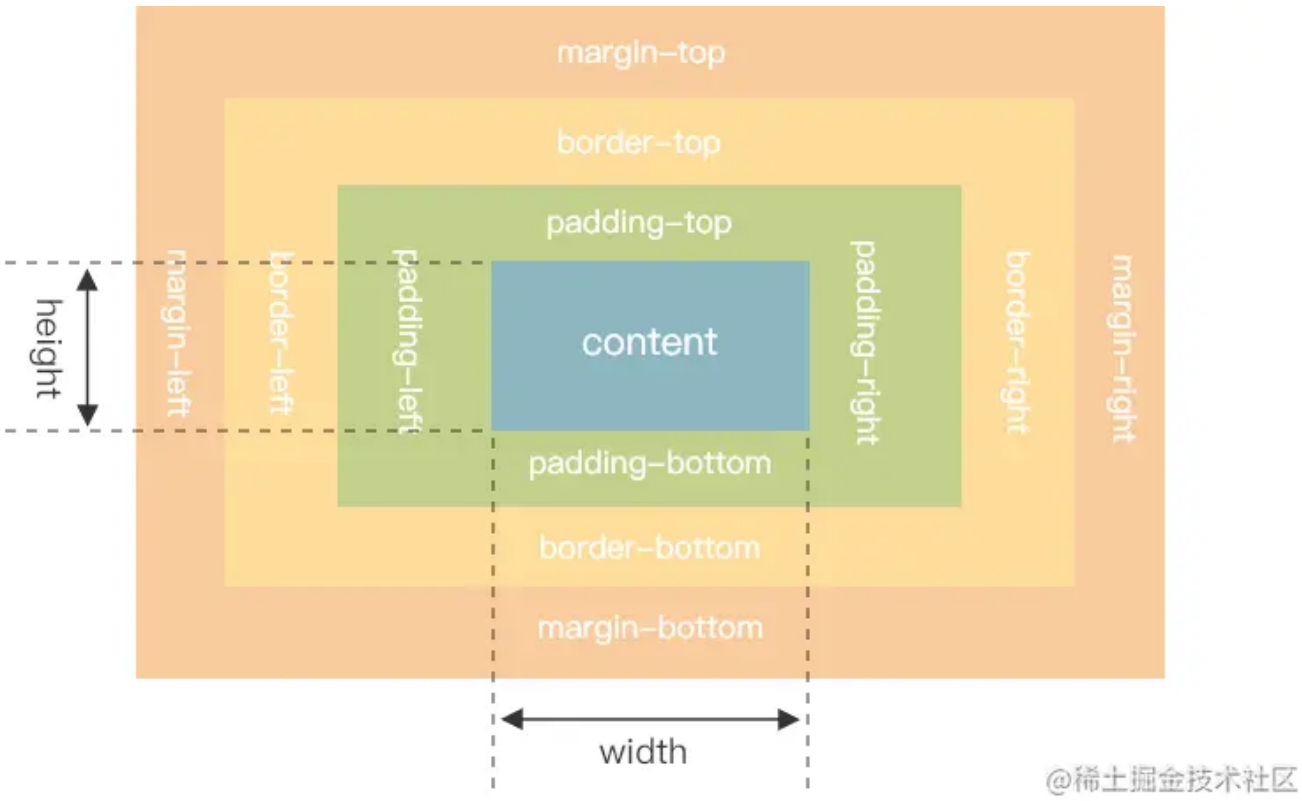
- 内部的 Box 会在垂直方向上一个接一个的放置。
- 垂直方向上的距离由 margin 决定
- bfc 的区域不会与 float 的元素区域重叠。
- 计算 bfc 的高度时，浮动元素也参与计算
- bfc 就是页面上的一个独立容器，容器里面的子元素不会影响外面元素。

2、什么是盒模型

CSS3中的盒模型有以下两种：标准盒子模型、IE盒子模型

盒模型都是由四个部分组成的，分别是margin、border、padding和content。

在标准盒模型性中



盒子在网页中实际占用:

宽 = `width + padding2 + border2 + margin2`

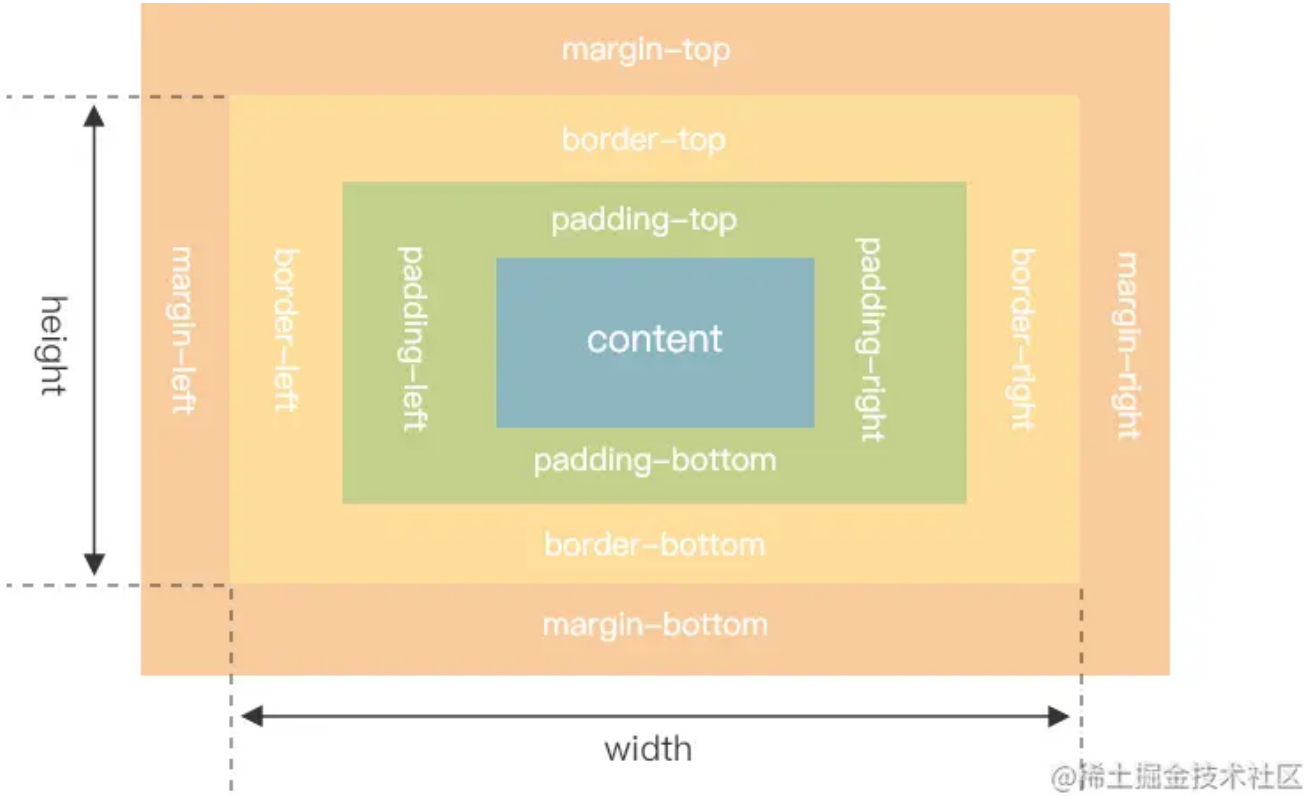
高 = `height + padding2 + border2 + margin2`

盒模型实际大小:

宽 = `width + padding2 + border2`

高 = `height + padding2 + border2`

在IE盒模型性中



盒子在网页中实际占用:

宽 = `width + margin2`

高 = `height + margin2`

盒模型实际大小:

宽 = `width`

高 = `height`

可以通过修改元素的box-sizing属性来改变元素的盒模型:

- `box-sizeing: content-box` 表示标准盒模型
- `box-sizeing: border-box` 表示IE盒模型

3、未知高度元素垂直居中的几种方案

当需要垂直居中的元素高度未知时，一般采用一下几种方案实现垂直居中:

使用绝对定位和transform

css 复制代码

```
.parent {
    position: relative;
    width: 100%;
    height: 400px;
```

```
}
.children {
    position: absolute;
    top: 50%;
    transform: translate(0, -50%);
}
```

flex实现垂直居中（最常用的）

css 复制代码

```
.parent {
    display: flex;
    align-items: center;
    justify-content: center;
    width: 100%;
    height: 400px;
    background: #fff;
}
.children {
    background: red;
}
```

通过table属性

css 复制代码

```
<div class="parent">
    <div class="children">
        <div>test</div>
    </div>
</div>

.parent {
    display: table;
    text-align:center;
}

.children {
    background: #ccc;
    display: table-cell;
    vertical-align: middle;
}

.child div {
    width: 300px;
    height: 150px;
    background: red;
    margin: 0 auto;
}
```

4、CSS3中有哪些新特性

- 新增各种CSS选择器
- 圆角 （ `border-radius:8px` ）
- 多列布局 （ `multi-column layout` ）
- 阴影和反射 （ `Shadoweflect` ）
- 文字特效 （ `text-shadow` ）
- 文字渲染 （ `Text-decoration` ）
- 线性渐变 （ `gradient` ）
- 旋转 （ `transform` ）
- 增加了旋转,缩放,定位,倾斜,动画,多背景

5、什么是CSS 预处理器 / 后处理器？大家为什么要使用他们?

预处理器：例如LESS、Sass、Stylus，用来预编译Sass或less，增强了css代码的复用性，还有层级、mixin、变量、循环、函数等，具有很方便的UI组件模块化开发能力，极大的提高工作效率。

后处理器：例如PostCSS，通常被视为在完成的样式表中根据CSS规范处理CSS，让其更有效；目前最常做的是给CSS属性添加浏览器私有前缀，实现跨浏览器兼容性的问题。

CSS 预处理器为 CSS 增加一些编程的特性，无需考虑浏览器的兼容性问题”，例如你可以在 CSS 中使用变量、简单的逻辑程序、函数（如右侧代码编辑器中就使用了变量$color）等等在编程语言中的一些基本特性，可以让你的 CSS 更加简洁、适应性更强、可读性更佳，更易于代码的维护等诸多好处。

6、为什么会出现margin重叠的问题？怎么解决?

问题描述： 两个块级元素的上外边距和下外边距可能会合并（折叠）为一个外边距，其大小会取其中外边距值大的那个，这种行为就是外边距折叠。需要注意的是，浮动的元素和绝对定位这种脱离文档

流的元素的外边距不会折叠。重叠只会出现在垂直方向。

计算原则： 折叠合并后外边距的计算原则如下：

- 如果两者都是正数，那么就取最大者
- 如果是一正一负，就会正值减去负值的绝对值
- 两个都是负值时，用0减去两个中绝对值大的那个

解决办法： 对于折叠的情况，主要有两种： `兄弟之间重叠` 和 `父子之间重叠`

（1）兄弟之间重叠

- 底部元素变为行内盒子： `display: inline-block`
- 底部元素设置浮动： `float`
- 底部元素的position的值为 `absolute/fixed`

（2）父子之间重叠

- 父元素加入： `overflow: hidden`
- 父元素添加透明边框： `border:1px solid transparent`
- 子元素变为行内盒子： `display: inline-block`
- 子元素加入浮动属性或定位

7、flex布局

flex知识点的话，建议大家去看阮一峰老师的文章，看完应该就明白了

> Flex 布局教程：语法篇
> Flex 布局教程：实例篇

8、移动端媒体查询，rem, vw的理解

`@media媒体查询` 是针对不同的媒体类型定义不同的样式，特别是响应式页面，可以针对不同屏幕的大小，编写多套样式，从而达到自适应效果，代码如下：

```css
@media screen and (max-width: 720px) {
    body {
        background-color: #6633FF;
    }
}

@media screen and (max-width: 640px) {
    body {
        background-color: #00FF66;
    }
}
/*
   上述的代码分别对分辨率在0～640px以及640px～720px的屏幕设置了不同的背景颜色。
*/
```

rem 是一个灵活的可扩展的单位，由浏览器转化像素并显示。与em单位不同， rem 单位无论嵌套层级如何，都只相对于浏览器的根元素（HTML元素）的 font-size

由于 viewport 单位得到众多浏览器的兼容， lib-flexible 这个过渡方案已经可以放弃使用，不管是现在的版本还是以前的版本，都存有一定的问题。建议大家开始使用 viewport，代码如下：

```js
(function flexible (window, document) {
  var docEl = document.documentElement
  var dpr = window.devicePixelRatio || 1

  // adjust body font size
  function setBodyFontSize () {
    if (document.body) {
      document.body.style.fontSize = (12 * dpr) + 'px'
    } else {
      document.addEventListener('DOMContentLoaded', setBodyFontSize)
    }
  }
  setBodyFontSize();

  // set 1rem = viewWidth / 10
  function setRemUnit () {
    var rem = docEl.clientWidth / 10
    docEl.style.fontSize = rem + 'px'
  }

  setRemUnit()

  // reset rem unit on page resize
  window.addEventListener('resize', setRemUnit)
  window.addEventListener('pageshow', function (e) {
```

```
    if (e.persisted) {
      setRemUnit()
    }
  })

  // detect 0.5px supports
  if (dpr >= 2) {
    var fakeBody = document.createElement('body')
    var testElement = document.createElement('div')
    testElement.style.border = '.5px solid transparent'
    fakeBody.appendChild(testElement)
    docEl.appendChild(fakeBody)
    if (testElement.offsetHeight === 1) {
      docEl.classList.add('hairlines')
    }
    docEl.removeChild(fakeBody)
  }
}(window, document))
```

什么是vw/vh？

vw/vh是一个相对单位（类似于em,rem)

- vw：viewport width 视口宽度单位
- vh：viewport height 视口高度单位

相对视口的尺寸计算结果

- 1vw = 1/100视口宽度
- 1vh = 1/100视口高度

9、移动端1px解决方案

1px的问题经常出现在移动端边框设置上，会导致设置1px边框看起来较粗，影响用户体验，对于boder的1px问题，可以通过 伪元素 + transform 来解决, 代码如下：

css 复制代码

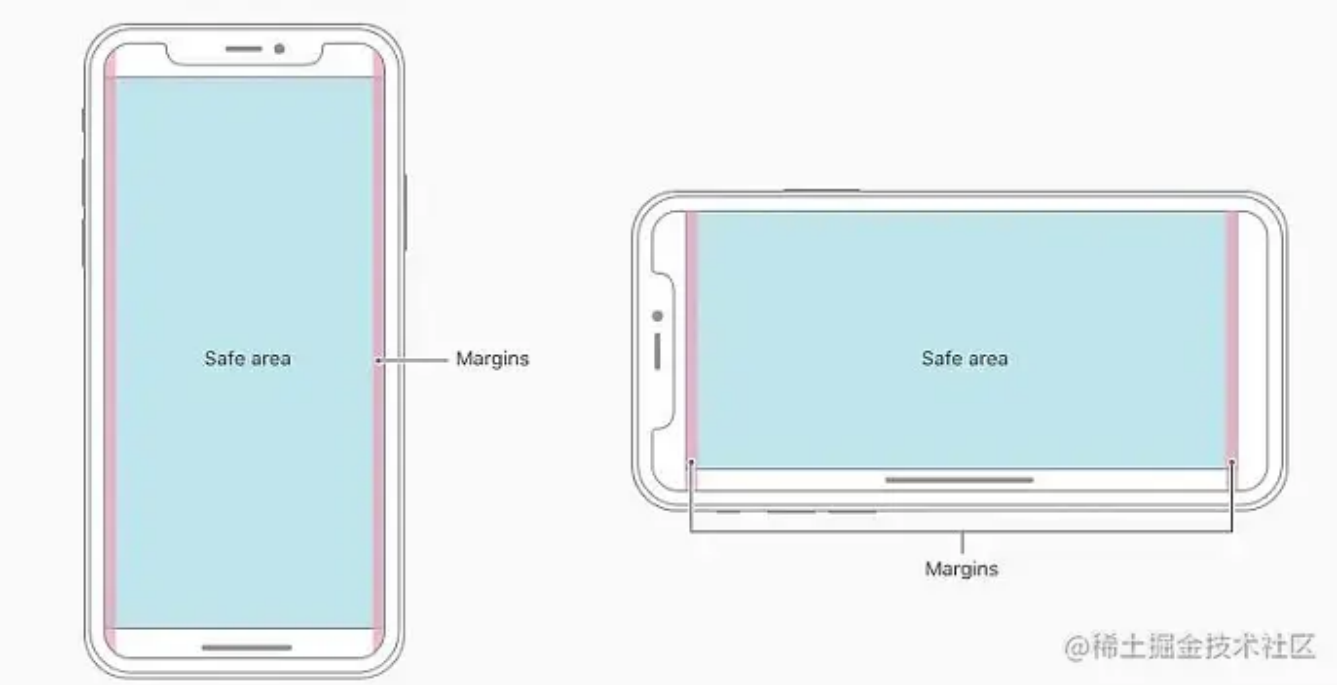
```
/* 手机端实现真正的一像素边框 */
.border-1px,
.border-bottom-1px,
.border-top-1px,
.border-left-1px,
.border-right-1px {
    position: relative;
}
```

```
/* 线条颜色 */
.border-1px::after,
.border-bottom-1px::after,
.border-top-1px::after,
.border-left-1px::after,
.border-right-1px::after {
    background-color: #000;
}

/* 底边边框一像素 */
.border-bottom-1px::after {
    content: "";
    position: absolute;
    left: 0;
    bottom: 0;
    width: 100%;
    height: 1px;
    transform-origin: 0 0;
}

/* 上边边框一像素 */
.border-top-1px::after {
    content: "";
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 1px;
    transform-origin: 0 0;
}

/* 左边边框一像素 */
.border-left-1px::after {
    content: "";
    position: absolute;
    left: 0;
    top: 0;
    width: 1px;
    height: 100%;
    transform-origin: 0 0;
}

/* 右边边框1像素 */
.border-right-1px::after {
    content: "";
    position: absolute;
    right: 0;
    top: 0;
```

```
    width: 1px;
    height: 100%;
    transform-origin: 0 0;
    box-sizing: border-box;
}

/* 边框一像素 */
.border-1px::after {
    content: "";
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    border: 1px solid gray;
    box-sizing: border-box;
}

/* 设备像素比 */
/* 显示屏最小dpr为2 */
@media (-webkit-min-device-pixel-ratio: 2) {
    .border-bottom-1px::after, .border-top-1px::after {
        transform: scaleY(0.5);
    }

    .border-left-1px::after, .border-right-1px::after {
        transform: scaleX(0.5);
    }

    .border-1px::after {
        width: 200%;
        height: 200%;
        transform: scale(0.5);
        transform-origin: 0 0;
    }
}

/* 设备像素比 */
@media (-webkit-min-device-pixel-ratio: 3)  {
    .border-bottom-1px::after, .border-top-1px::after {
        transform: scaleY(0.333);
    }

    .border-left-1px::after, .border-right-1px::after {
        transform: scaleX(0.333);
    }

    .border-1px::after {
        width: 300%;
```

```
        height: 300%;
        transform: scale(0.333);
        transform-origin: 0 0;
    }
}
```

10、IOS全面屏安全区域问题



iOS 11

css 复制代码

```
padding-top: constant(safe-area-inset-top);
padding-right: constant(safe-area-inset-right);
padding-bottom: constant(safe-area-inset-bottom);
padding-left: constant(safe-area-inset-left);
```

iOS 11.2+

css 复制代码

```
padding-top: env(safe-area-inset-top);
padding-right: env(safe-area-inset-right);
padding-bottom: env(safe-area-inset-bottom);
padding-left: env(safe-area-inset-left);
```

11、图片变形问题如何解决

如果图片设置的是背景图

```css
background-size: cover;
```

css 复制代码

如果是img标签图片

```css
object-fit: cover;
```

css 复制代码

12、css常见布局实现

> 可参考：常见的CSS布局

二、Javascript基础

1、JavaScript数据类型

JavaScript一共有8种数据类型

七种基本数据类型：`Undefined`、`Null`、`Boolean`、`Number`、`String`、`Symbol`（es6新增）和 `BigInt`（es10新增）

一种复杂数据类型：`Object` 里面包含 Function、Array、Date等

2、JavaScript判断数据类型的几种方法

typeof

```
/*
  优点：能够快速区分基本数据类型
  缺点：不能将Object、Array和Null区分，都返回object
*/
console.log(typeof 1);              // number
console.log(typeof NaN);            // number
console.log(typeof true);           // boolean
console.log(typeof 'mc');           // string
console.log(typeof Symbol)          // function
console.log(typeof function(){});   // function
console.log(typeof console.log());  // function
console.log(typeof []);             // object
console.log(typeof {});             // object
console.log(typeof null);           // object
console.log(typeof undefined);      // undefined
```

instanceof

```
/*
  优点：能够区分Array、Object和Function，适合用于判断自定义的类实例对象
  缺点：Number，Boolean，String基本数据类型不能判断
*/
console.log(1 instanceof Number);                   // false
console.log(true instanceof Boolean);               // false
console.log('str' instanceof String);               // false
console.log([] instanceof Array);                   // true
console.log(function(){} instanceof Function);      // true
console.log(function(){} instanceof Object);        // true
console.log({} instanceof Function);                // false
console.log({} instanceof Object);                  // true
```

```
/*
  优点：精准判断数据类型
  缺点：写法繁琐不容易记，推荐进行封装后使用
*/
const toString = Object.prototype.toString;
console.log(toString.call(1));                      // [object Number]
console.log(toString.call(true));                   // [object Boolean]
console.log(toString.call('mc'));                   // [object String]
console.log(toString.call([]));                     // [object Array]
console.log(toString.call({}));                     // [object Object]
console.log(toString.call(function(){}));           // [object Function]
console.log(toString.call(undefined));              // [object Undefined]
console.log(toString.call(null));                   // [object Null]
```

3、intanceof 操作符的实现原理及实现

```js
function instanceOf(left, right) {
    let proto = left.__proto__
    let prototype = right.prototype
    while (true) {
        if (proto === null) return false
        if (proto === prototype) return true
        proto = proto.__proto__
    }
}
```

4、数组常用操作方法

- push()：在末尾添加一个或多个元素，修改原数组，返回值，数组新的长度
- pop()：删除数组最后一个元素，无参数，修改原数组，返回值，删除元素的值
- unshift()：向数组的开头添加一个或者多个元素，修改原数组，返回值，数组新的长度
- shift()：删除数组的第一个元素，数组长度减1，无参数，修改原数组，返回值，删除元素的值
- reverse()：颠倒数组中元素的顺序，无参数，修改原数组，返回值，新的数组
- sort()：对数组的元素进行排序，修改原数组，返回值，新的数组
- toString()：把数组转换成字符串，逗号分隔每一项，返回值，一个字符串
- join()：方法用于把数组中的所有元素转换成一个字符串，返回值，一个字符串
- concat()：连接两个或多个数组，不影响原数组，返回值，一个新的数组
- slice()：数组截取slice(begin,end)，返回值，返回被截取项目的新数组
- splice()：数组删除splice(第几个开始，要删除的个数)，修改原数组，返回值，返回被删除项目的新数组
- indexOf()：从前往后查找数组元素的索引号，不修改原数组，返回值，数组元素的索引号
- lastIndexOf()：从后往前查找数组元素的索引号，不修改原数组，返回值，数组元素的索引号

- `find()`：用于找出第一个符合条件的数组成员，返回值，数组元素
- `findIndex()`：用于找出第一个符合条件的数组成员的位置，返回值，索引号
- `includes()`：判断某个数组是否包含给定的值

5、数组去重

除了es6的new Set()去重外，可以通过以下几种方法实现数组去重

js 复制代码

```
// 利用filter去实现
Array.prototype.unique1 = function() {
    return this.filter((item, index, array) => {
        return this.indexOf(item) === index;
    });
}
```

js 复制代码

```
Array.prototype.unique2 = function() {
    const n = {}, r = []; // n为hash表，r为临时数组
    for (let i = 0; i < this.length; i++) {
        if (!n[this[i]]) { // 如果hash表中没有当前项
            n[this[i]] = true; //存入hash表
            r.push(this[i]); //把当前数组的当前项push到临时数组里面
        }
    }
    return r;
}
```

js 复制代码

```
// 返回后的数组顺序会乱
Array.prototype.unique3 = function() {
    this.sort();
    const r = [this[0]];
    for (let i = 1; i < this.length; i++) {
        if (this[i] !== this[i - 1]) {
            r.push(this[i]);
        }
    }
    return r;
}
```

6、执行上下文与作用域、作用域链

执行上下文（以下简称"上下文"）的概念在 JavaScript 中是颇为重要的。变量或函数的上下文决定了它们可以访问哪些数据，以及它们的行为。每个上下文都有一个关联的变量对象（variable object），而这个上下文中定义的所有变量和函数都存在于这个对象上。虽然无法通过代码访问变量对象，但后台处理数据会用到它。

全局上下文是最外层的上下文。根据 ECMAScript 实现的宿主环境，表示全局上下文的对象可能不一样。在浏览器中，全局上下文就是我们常说的 window 对象，因此所有通过 var 定义的全局变量和函数都会成为 window 对象的属性和方法。使用 let 和 const 的顶级声明不会定义在全局上下文中，但在作用域链解析上效果是一样的。上下文在其所有代码都执行完毕后会被销毁，包括定义在它上面的所有变量和函数（全局上下文在应用程序退出前才会被销毁，比如关闭网页或退出浏览器）。

每个函数调用都有自己的上下文。当代码执行流进入函数时，函数的上下文被推到一个上下文栈上。在函数执行完之后，上下文栈会弹出该函数上下文，将控制权返还给之前的执行上下文。ECMAScript 程序的执行流就是通过这个上下文栈进行控制的。

上下文中的代码在执行的时候，会创建变量对象的一个 作用域链（scope chain） 。这个作用域链决定了各级上下文中的代码在访问变量和函数时的顺序。代码正在执行的上下文的变量对象始终位于作用域链的最前端。如果上下文是函数，则其活动对象（activation object）用作变量对象。活动对象最初只有一个定义变量： arguments 。（全局上下文中没有这个变量。）作用域链中的下一个变量对象来自包含上下文，再下一个对象来自再下一个包含上下文。以此类推直至全局上下文；全局上下文的变量对象始终是作用域链的最后一个变量对象。

代码执行时的标识符解析是通过沿作用域链逐级搜索标识符名称完成的。搜索过程始终从作用域链 的最前端开始，然后逐级往后，直到找到标识符。（如果没有找到标识符，那么通常会报错。） 看一看下面这个例子：

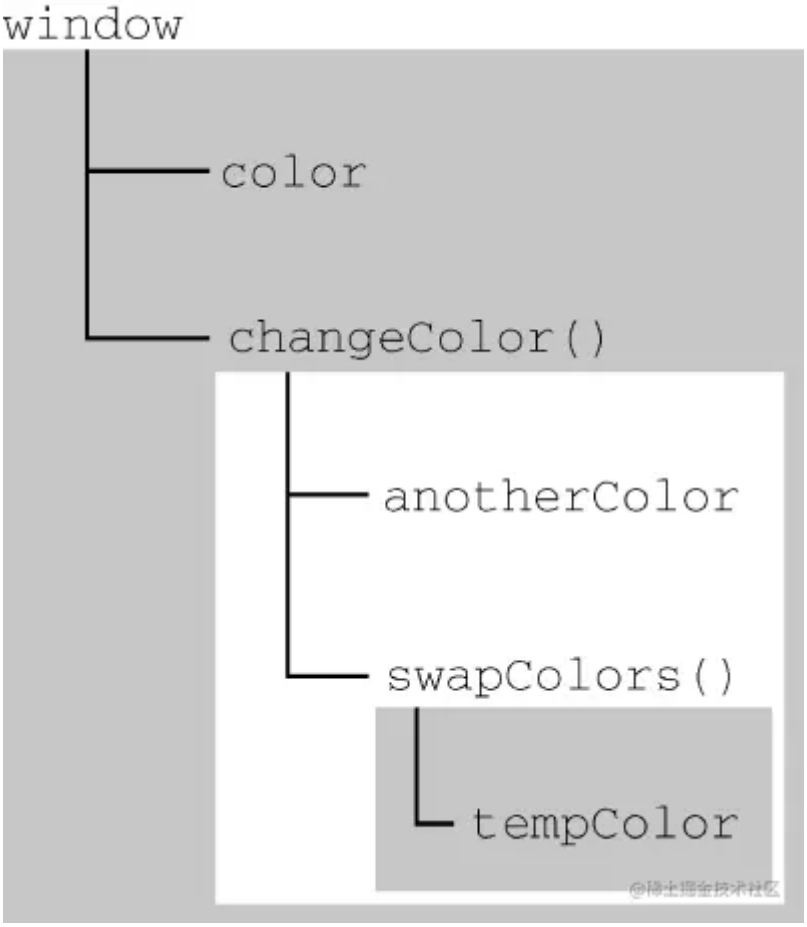

```
var color = "blue";

function changeColor() {
 if (color === "blue") {
     color = "red";
 } else {
     color = "blue";
 }
}

changeColor();
```

对这个例子而言，函数 `changeColor()` 的作用域链包含两个对象：一个是它自己的变量对象（就 是定义 arguments 对象的那个），另一个是全局上下文的变量对象。这个函数内部之所以能够访问变量 `color` ，就是因为可以在作用域链中找到它。

此外，局部作用域中定义的变量可用于在局部上下文中替换全局变量。看一看下面这个例子：

```js 复制代码
var color = "blue";

function changeColor() {
    let anotherColor = "red";

    function swapColors() {
        let tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;
        // 这里可以访问 color、anotherColor 和 tempColor
    }

    // 这里可以访问 color 和 anotherColor，但访问不到 tempColor
    swapColors();
}

// 这里只能访问 color
changeColor();
```

以上代码涉及 3 个上下文：全局上下文、 `changeColor()` 的局部上下文和 `swapColors()` 的局部 上下文。全局上下文中有一个变量 color 和一个函数 `changeColor()` 。 `changeColor()` 的局部上下文中有一个变量 `anotherColor` 和一个函数 `swapColors()` ，但在这里可以访问全局上下文中的变量 `color` 。 `swapColors()` 的局部上下文中有一个变量 `tempColor` ，只能在这个上下文中访问到。全局上下文和 `changeColor()` 的局部上下文都无法访问到 `tempColor` 。而在 `swapColors()` 中则可以访问另外两个 上下文中的变量，因为它们都是父上下文。下图展示了这个例子的作用域链。



上图中的矩形表示不同的上下文。内部上下文可以通过作用域链访问外部上下文中的一切，但外 部上下文无法访问内部上下文中的任何东西。上下文之间的连接是线性的、有序的。每个上下文都可以到上一级上下文中去搜索变量和函数，但任何上下文都不能到下一级上下文中去搜索。 `swapColors()` 局部上下文的作用域链中有 3 个对象： `swapColors()` 的变量对象、changeColor()的变量对象和全局变量对象。 `swapColors()` 的局部上下文首先从自己的变量对象开始搜索变量和函数，搜不到就去搜索上一级变量对象。 `changeColor()` 上下文的作用域链中只有 2 个对象：它自己的变量对象和全局变量对象。因此，它不能访问 `swapColors()` 的上下文。

7、说一下你对前端闭包的理解

闭包是指有权访问另外一个函数作用域中的变量的函数，有兴趣的可以看看几篇文章，看完会又更深刻的理解

> 破解前端面试（80% 应聘者不及格系列）：从闭包说起

8、说一下你对call/apply/bind的理解

`call/apply/bind` 都是用来修改this指向的

apply 和 call 的区别

`apply` 和 `call` 的区别是 `call` 方法接受的是若干个参数列表，而 `apply` 接收的是一个包含多个参数的数组(方便记忆：call有两个l，表示可以传递多个参数)

bind 和 apply、call 区别

call、apply都是直接调用，bind生成的this指向改变函数需要手动调用

9、说一下你对原型、原型链的理解

> JavaScript深入之从原型到原型链

10、js实现继承的几种方法

比较常见的几种：

- 原型链继承
- 借用构造函数继承
- 组合继承
- 原型式继承
- 寄生式继承
- 寄生组合式继承 一般只建议寄生组合式继承，因为其它方式的继承会在一次实例中调用两次父类的构造函数或有其它缺点，代码如下：


```
function Parent(name) {
    this.name = name;
}
Parent.prototype.sayName = function() {
    console.log('parent name:', this.name);
}
function Child(name, parentName) {
    Parent.call(this, parentName);
```

```
    this.name = name;
}
function create(proto) {
    function F(){}
    F.prototype = proto;
    return new F();
}
Child.prototype = create(Parent.prototype);
Child.prototype.sayName = function() {
    console.log('child name:', this.name);
}
Child.prototype.constructor = Child;

var parent = new Parent('father');
parent.sayName();    // parent name: father

var child = new Child('son', 'father');
```

> 其他几种继承方式，可参考：JavaScript常用八种继承方案

11、深浅拷贝

深拷贝： `JSON.parse(JSON.stringify(data))` 递归

浅拷贝： `Object.assign()` 扩展运算符（`...`） `Array.prototype.concat()` `Array.prototype.slice()`

12、事件循环机制/Event Loop

> 说说事件循环机制(满分答案来了)

13、防抖与节流

防抖： 防抖就是将一段时间内连续的多次触发转化为一次触发。一般可以使用在用户输入停止一段时间过后再去获取数据，而不是每次输入都去获取

防抖应用场景：

- 保存、跳转、登录等按钮的频繁点击
- 搜索框搜索

节流： 节流，顾名思义，控制流量。用于用户在与页面交互时控制事件发生的频率，一般场景是单位的时间或其它间隔内定时执行操作。一段时间内，事件在每次到达我们规定的间隔 n 秒时触发一次

节流应用场景：

- scroll，resize, touchmove, mousemove等极易持续性促发事件的相关动画问题，降低频率
- 浏览器播放事件

14、函数柯里化

在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数且返回结果的新函数的技术。


```
function curry(fn, args) {
    var length = fn.length;
    var args = args || [];
    return function(){
        newArgs = args.concat(Array.prototype.slice.call(arguments));
        if (newArgs.length < length) {
            return curry.call(this,fn,newArgs);
        } else {
            return fn.apply(this,newArgs);
        }
    }
}

function multiFn(a, b, c) {
    return a * b * c;
}

var multi = curry(multiFn);

multi(2)(3)(4);
multi(2,3,4);
multi(2)(3,4);
multi(2,3)(4);
```

15、手写代码

这里面包含了大部分的常见手写代码： JavaScript手写代码无敌秘籍

三、ES6

1、常用的es6语法有哪些

- let、const
- 解构赋值
- 模板字符串
- 箭头函数
- 函数默认值
- promise
- set、map结构
- class类
- symbol
- Iterator 和 for...of 循环.
- 数值的扩展方法
- 数组的扩展方法
- 正则的扩展方法
- 对象的扩展方法

2、说下var、let、const的区别

- let、const不存在变量提升，var存在变量提升
- 在严格模式下let、const不能重复声明，var可以重复声明

- let、const有块级作用域，var没有块级作用域
- const声明一个只读的常量。一旦声明，常量的值就不能改变

> 注： const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，const只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

3、说说你对promise的了解

promise是异步编程的一种解决方案。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息

> 详细的可以参考：要就来45道Promise面试题一次爽到底

4、setTimeout、Promise、Async/Await 的区别

- 其中 `setTimeout` 的回调函数放到宏任务队列里，等到执行栈清空以后执行
- `promise.then` 里的回调函数会放到相应宏任务的微任务队列里，等宏任务里面的同步代码执行完再执行
- `async` 函数表示函数里面可能会有异步方法， `await` 后面跟一个表达式， `async` 方法执行时，遇到 `await` 会立即执行表达式，然后把表达式后面的代码放到微任务队列里，让出执行栈让同步代码先执行

5、class类的理解

类定义

与函数类型相似，定义类也有两种主要方式：类声明和类表达式。这两种方式都使用 `class` 关键 字加大括号：

```js
class Person {} // 类声明
const Animal = class {}; // 类表达式
```


与函数表达式类似，类表达式在它们被求值前也不能引用。不过，与函数定义不同的是，虽然函数 声明可以提升，但类定义不能：

```js
console.log(FunctionExpression); // undefined
var FunctionExpression = function() {};
console.log(FunctionExpression); // function() {}

console.log(FunctionDeclaration); // FunctionDeclaration() {}
function FunctionDeclaration() {}
console.log(FunctionDeclaration); // FunctionDeclaration() {}

console.log(ClassExpression); // undefined
var ClassExpression = class {};
console.log(ClassExpression); // class {}

console.log(ClassDeclaration); // ReferenceError: ClassDeclaration is not defined
class ClassDeclaration {}
console.log(ClassDeclaration); // class ClassDeclaration {}
```


类构成

类可以包含构造函数方法、实例方法、获取函数、设置函数和静态类方法，但这些都不是必需的。 空的类定义照样有效。默认情况下，类定义中的代码都在严格模式下执行

构造函数 constructor

`constructor()` 方法是类的默认方法，通过 `new` 命令生成对象实例时，自动调用该方法。一个类必须有 `constructor()` 方法，如果没有显式定义，一个空的 `constructor()` 方法会被默认添加。

类的实例化

`class` 的实例化必须通过 `new` 关键字

```js
class Example {}
let exam1 = Example();
// Class constructor Example cannot be invoked without 'new'
```

使用 new 调用类的构造函数会执行如下操作。

(1) 在内存中创建一个新对象。

(2) 这个新对象内部的 `[[Prototype]]` 指针被赋值为构造函数的 `prototype` 属性。

(3) 构造函数内部的 `this` 被赋值为这个新对象（即 `this` 指向新对象）

(4) 执行构造函数内部的代码（给新对象添加属性）。

(5) 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

类的所有实例共享一个原型对象

```js
class Example {
    constructor(a, b) {
        this.a = a;
        this.b = b;
        console.log('Example');
    }
    sum() {
        return this.a + this.b;
    }
}
let exam1 = new Example(2, 1);
let exam2 = new Example(3, 1);
console.log(exam1._proto_ == exam2._proto_); // true

exam1._proto_.sub = function () {
    return this.a - this.b;
}
console.log(exam1.sub()); // 1
console.log(exam2.sub()); // 2
```

上面代码中， `exam1` 和 `exam2` 都是 `Example` 的实例，它们的原型都是 `Example.prototype` ，所以 `__proto__` 属性是相等的

setter、getter

在“类”的内部可以使用 `get` 和 `set` 关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

静态方法 static

类（ `class` ）通过 `static` 关键字定义静态方法。不能在类的实例上调用静态方法，而应该通过类本身调用。这些通常是实用程序方法，例如创建或克隆对象的功能。

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上 `static` 关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```js 复制代码
class Foo {
    static classMethod() {
        return 'hello';
    }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
```

上面代码中， `Foo` 类的 `classMethod` 方法前有 `static` 关键字，表明该方法是一个静态方法，可以直接在 `Foo` 类上调用（ `Foo.classMethod()` ），而不是在 `Foo` 类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。

关键字 super

`super` 关键字用于访问和调用一个对象的父对象上的函数。

继承 extends

`ES6` 类支持单继承。使用 extends 关键字，就可以继承任何拥有 `[[Construct]]` 和原型的对象。 很大程度上，这意味着不仅可以继承一个类，也可以继承普通的构造函数（保持向后兼容）：

```js 复制代码
class Vehicle {}

// 继承类
class Bus extends Vehicle {}

let b = new Bus();
console.log(b instanceof Bus); // true
console.log(b instanceof Vehicle); // true


function Person() {}

// 继承普通构造函数
class Engineer extends Person {}

let e = new Engineer();
console.log(e instanceof Engineer); // true
console.log(e instanceof Person); // true
```

四、浏览器、网络基础

1、如何用js去实现一个ajax

大概步骤如下，具体要针对具体业务封装：

```js
// 创建 XMLHttpRequest 对象
var ajax = new XMLHttpRequest();
// 规定请求的类型、URL 以及是否异步处理请求。
ajax.open('GET', url);
// 发送信息至服务器时内容编码类型
ajax.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
// 发送请求
ajax.send(null);
// 接受服务器响应数据
ajax.onreadystatechange = function () {
    if (obj.readyState == 4 && (obj.status == 200 || obj.status == 304)) {
    }
};
```

2、常见的http状态码

状态码	含义
200	表示从客户端发来的请求在服务器端被正常处理了
204	无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档
301	永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替
302	临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI
304	未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。 客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源

状态码	含义
400	表示请求报文中存在语法错误。当错误发生时，需修改请求的内容后再次发送请求
401	表示未授权（Unauthorized），当前请求需要用户验证
403	表示对请求资源的访问被服务器拒绝了
404	表示服务器上无法找到请求的资源。除此之外，也可以在服务器端拒绝请求且不想说明理由时使用
500	表示服务器端在执行请求时发生了错误。也有可能是Web应用存在的bug或某些临时的故障
502	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
503	表示服务器暂时处于超负载或正在进行停机维护，现在无法处理请求
504	网关超时，服务器作为网关或代理，但是没有及时从上游服务器收到请求

3、ajax的请求头和响应头包含哪些东西

Request header	解释
Accept	指定客户端能够接收的内容类型，如：application/json, text/plain
Accept-Encoding	指定浏览器可以支持的web服务器返回内容压缩编码类型，如：gzip, deflate, br
Accept-Language	浏览器所希望的语言种类
Cache-Control	缓存机制，默认no-cache
Connec-Length	请求头的长度
Content-Type	发送的数据类型，如：application/x-www-form-urlencoded，application/json，multipart/form-data（可用来做文件上传），
Connection	表示是否需要持久连接。（HTTP 1.1默认进行持久连接）
Cookie	HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器
Host	指定请求的服务器的域名和端口号

Request header	解释
If-Modified-Since	只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回304“Not Modified”应答
Referer	包含一个URL，用户从该URL代表的页面出发访问当前请求的页面
User-Agent	浏览器信息，如果Servlet返回的内容与浏览器类型有关则该值非常有用
Cookie	这是最重要的请求头信息之一

Response header	解释
Allow	服务器支持哪些请求方法（如GET、POST等）
Content-Encoding	文档的编码（Encode）方法。 只有在解码之后才可以得到Content-Type头指定的内容类型。 利用gzip压缩文档能够显著地减少HTML文档的下载时间。 Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE 4、IE 5才支持它。 因此，Servlet应该通过查看Accept-Encoding头（即request.getHeader("Accept-Encoding")）检查浏览器是否支持gzip， 为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面
Content-Length	表示内容长度。 只有当浏览器使用持久HTTP连接时才需要这个数据。 如果你想要利用持久连接的优势，可以把输出文档写入ByteArrayOutputStream， 完成后查看其大小，然后把该值放入Content-Length头， 最后通过byteArrayStream.writeTo发送内容
Content-Type	表示后面的文档属于什么MIME类型。 Servlet默认为text/plain，但通常需要显式地指定为text/html。 由于经常要设置Content-Type，因此HttpServletResponse提供了一个专用的方法setContentType
Date	当前的GMT时间。你可以用setDateHeader来设置这个头以避免转换时间格式的麻烦
Expires	应该在什么时候认为文档已经过期，从而不再缓存它?
Last-Modified	文档的最后改动时间。客户可以通过If-Modified-Since请求头提供一个日期， 该请求将被视为一个条件GET，只有改动时间迟于指定时间的文档才会返回， 否则返回一个304（Not Modified）状态。Last-Modified也可用setDateHeader方法来设置。

Response header	解释
`Location`	表示客户应当到哪里去提取文档。Location通常不是直接设置的，而是通过HttpServletResponse的sendRedirect方法，该方法同时设置状态代码为302
`Refresh`	表示浏览器应该在多少时间之后刷新文档，以秒计
`Server`	服务器名字。Servlet一般不设置这个值，而是由Web服务器自己设置
`Set-Cookie`	设置和页面关联的Cookie。Servlet不应使用response.setHeader("Set-Cookie", ...)，而是应使用HttpServletResponse提供的专用方法addCookie
`WWW-Authenticate`	客户应该在Authorization头中提供什么类型的授权信息? 在包含401（Unauthorized）状态行的应答中这个头是必需的

4、为什么会产生跨域？怎么去解决跨域问题？

`跨域` 是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对JavaScript实施的安全限制。浏览器从一个域名的网页去请求另一个域名的资源时，出现 `域名、端口、协议` 任一不同，都属于跨域。

跨域解决方案：

- 通过jsonp跨域
- 跨域资源共享（CORS）
- nginx代理跨域
- nodejs中间件代理跨域
- WebSocket协议跨域
- postMessage跨域
- document.domain + iframe跨域
- location.hash + iframe
- window.name + iframe跨域

> 详情见：九种跨域方式实现原理（完整版）

5、post的预请求了解过吗?

预请求就是复杂请求（可能对服务器数据产生副作用的HTTP请求方法，如 `put，delete` 都会对服务器数据进行更修改，所以要先询问服务器）。
跨域请求中，浏览器自发的发起的预请求,浏览器会查询到两次请求，第一次的请求参数是 `options` ，以检测试实际请求是否可以被浏览器接受

什么情况下发生

- 请求方法不是get head post
- post 的content-type不是application/x-www-form-urlencode,multipart/form-data,text/plain ([也就是把content-type设置成" `application/json` "])
- 请求设置了自定义的header字段: 比如业务需求，传一个字段，方便后端获取，不需要每个接口都传

> 例如设置了post请求的content-type：application/json,就会发生预请求

6、从输入URL到页面展示，这中间发生了什么?

- DNS 解析:将域名解析成 IP 地址
- TCP 连接：TCP 三次握手
- 发送 HTTP 请求
- 服务器处理请求并返回 HTTP 报文
- 浏览器解析渲染页面
- 断开连接：TCP 四次挥手

> 详细可以参考： 从输入URL开始建立前端知识体系

7、说一下tcp三次握手和四次挥手

三次握手

第一次握手： 客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN(c)。此时客户端处于 SYN_SEND 状态。

第二次握手： 服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)。同时会把客户端的 ISN + 1 作为ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 SYN_RCVD 的状态。

第三次握手： 客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接

四次挥手

第一次挥手： 客户端先发送FIN报文（第24帧），用来关闭主动方到被动关闭方的数据传送，也就是客户端告诉服务器：我已经不会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，客户端依然会重发这些数据)，但此时客户端还可以接受数据。

第二次挥手： Server端接到FIN报文后，如果还有数据没有发送完成，则不必急着关闭Socket，可以继续发送数据。所以服务器端先发送ACK（第25帧），告诉Client端：请求已经收到了，但是我还没准备好，请继续等待停止的消息。这个时候Client端就进入FIN_WAIT状态，继续等待Server端的FIN报文。

第三次挥手： 当Server端确定数据已发送完成，则向Client端发送FIN报文（第26帧），告诉Client端：服务器这边数据发完了，准备好关闭连接了。

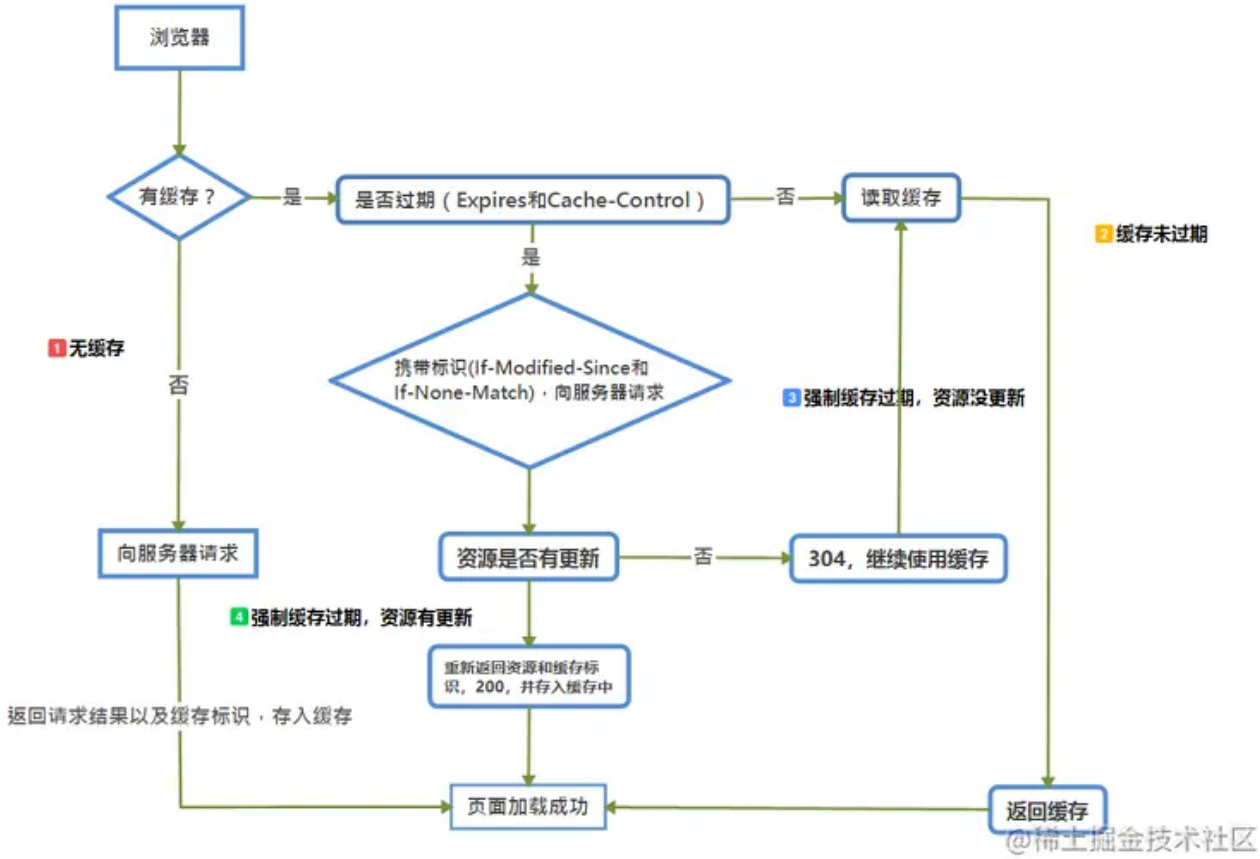
第四次挥手： Client端收到FIN报文后，就知道可以关闭连接了，但是他还是不相信网络，所以发送ACK后进入TIME_WAIT状态（第27帧）， Server端收到ACK后，就知道可以断开连接了。Client端等待了2MSL后依然没有收到回复，则证明Server端已正常关闭，最后，Client端也可以关闭连接了至此，TCP连接就已经完全关闭了！

为了方便记忆，整理个白话文版本，以找工作和离职举例（客户端=小明，服务端=人事）

> 第一次握手：由小明发起，小明向招聘人事发送了一份简历，里面包含了小明的个人基本信息
>
> 第二次握手：由人事发起，人事收到小明的简历后，安排面试，面试通过后，发送了一份offer给小明
>
> 第三次握手：由小明发起，小明收到offer后，会发送一个确认offer的回复，表示已经收到offer

> 第一次挥手： 由小明发起，告诉服务器我要离职跑路了
>
> 第二次挥手： 由人事发起，告诉小明我知道你要离职了，但是你先把工作交接好
>
> 第三次挥手： 由人事发起，我看你的交接流程都走完了，你可以走人了
>
> 第四次挥手： 由小明发起，收到人事通知离职流程已经走完了，告诉人事我明白就不来了，人事收到消息把此人从公司注销

8、说一下浏览器缓存策略



大概流程如下：

1. 第一次请求，无缓存，直接向服务器发请求，并将请求结果存入缓存中
2. 第二次请求，在强制缓存时间内，缓存未过期，浏览器接使用缓存作为结果返回 200
3. 第三次请求，强制缓存时间已过期，进入协商缓存，向服务器请求，通过在Header中携带 `If-Modified-Since` (对应浏览器返回的 `last-Modify`) 或 `If-None-Match` (对应浏览器返回的Etag) 校验缓存内容是否有更新， `Etag` 优先级更高

4. 缓存资源没有更新，返回 304，浏览器继续使用缓存，更新强制缓存时间

   缓存资源有更新，缓存失效，返回 200，重新返回资源和缓存标识，再存入浏览器缓存中

9、能不能说一说浏览器的本地存储?

浏览器的本地存储主要分为 `Cookie` 、 `WebStorage` 和 `IndexedDB` ，其中 `WebStorage` 又可以分为 `localStorage` 和 `sessionStorage` 。接下来我们就来一一分析这些本地存储方案。

Cookie

HTTP Cookie，通常叫做Cookie，一开始是在客户端用于存储会话信息的。

`Cookie主要构成`

- name：名称，一个唯一确定的cookie的名称，cookie的名称必须经过URL编码。
- value：值，存储在cookie中的字符串值。值必须被URL编码。
- Domain：域，指明cookie对哪个域有效，所有向该域发送的请求都会包含这个信息。
- path：路径，对于指定域中的那个路径，应该向服务器发送cookie。
- Expires/Max-Age：有效期，表示cookie的有效期。
- HttpOnly：如果这个这个值设置为true，就不能通过JS脚本获取cookie的值。通过这个值可以有效防止XSS攻击。
- Secure：安全标志，指定后，cookie只有在使用SSL连接的时候才能发送到服务器。

`Cookie的原理`

第一次访问网站时，浏览器发出请求，服务器响应请求后，会在响应头中添加一个Set-Cookie，将cookie放入响应请求中。

在第二次发起请求时，浏览器通过Cookie请求头部将cookie信息送给服务器，服务端根据cookie信息辨别用户身份。

Cookie的过期时间、域、路径、有效期、适用站点都可以根据需要来指定。

`Cookie的生成`

Cookie的生成方式主要有两种：

- 服务端设置Cookie
- 客户端设置Cookie

服务端设置方式参考上面Cookie的原理，具体的实现方式自行查阅相关资料。客户端设置Cookie方法如下：

```js
document.cookie = "name=zhangsan; age=20"
```


Cookie的缺点

- 每个特定域名下的cookie数量有限，不同浏览器数量限制不同。如果超过数量限制后再设置Cookie，浏览器就会清除以前设置的Cookie。
- 大小只有4kb。
- 每次HTTP请求都会默认带上Cookie，影响获取资源的效率。
- Cookie的获取、设置、删除方法需要我们自己去封装。

Web Storage

Web Storage分为localStorage和sessionStorage

localStorage

localStorage有以下几个特点：

- 保持的数据永久有效，除非手动删除；
- 大小为5M
- 仅在客户端使用，不和服务端进行通信
- 接口封装较好

使用方法：

```js
// 设置
localStorage.setItem('name', '张三')
localStorage.age = '25'
// 取值
localStorage.getItem('name')
let age = localStorage.age
// 移除
localStorage.removeItem('name')
// 移除所有
localStorage.clear()
```


sessionStorage

sessionStorage对象存储特定于某个会话的数据，当这个会话的页签或浏览器关闭，sessionStorage也就消失了。

页面刷新之后，存储在sessionStorage中的数据仍然存在可用。

sessionStorage的特点：

- 会话级别的浏览器存储
- 大小为5M
- 仅在客户端使用，不和服务端通信
- 接口封装较好

使用方法：

```
js 复制代码
// 设置
sessionStorage.setItem('name', '张三')
sessionStorage.age = '25'
// 取值
sessionStorage.getItem('name')
let age = sessionStorage.age
// 移除
sessionStorage.removeItem('name')
// 移除所有
sessionStorage.clear()
```

sessionStorage和localStorage的区别：localStorage的数据可以长期保留，sessionStorage的数据在关闭页面后即被清空

IndexedDB

IndexedDB，全称Indexed Database API，是浏览器中保持结构化数据的一种数据库。

IndexedDB的思想是创建一套API，方便保存和读取JavaScript对象，同时支持查询和搜索。

IndexedDB特点

- 键值对存储：IndexedDB采用对象仓库存储数据，可以存储所有类型的数据。仓库中数据以键值对的形式保持。
- 异步：IndexedDB操作时不会锁死浏览器，用户依然可以进行其他操作。
- 支持事务：有学过数据库的对事务肯定不陌生。事务意味着在一系列操作中，只要有一步失败，整个事务就都取消，数据库回滚到事务执行之前，不存在只改写一部分数据的情况。
- 同源限制：IndexedDB受到同源限制，每一个数据库对应创建它的域名。网页只能访问自身域名下的数据库，而不能访问跨域的数据库。
- 储存空间大: IndexedDB 的储存空间比 localStorage大得多，一般来说不少于 250MB，甚至没有上限。

- 支持二进制储存: IndexedDB不仅可以储存字符串，还可以储存二进制数据（ArrayBuffer 对象和 Blob 对象）。

IndexedDB的入门教程，可以查看阮一峰老师的文章：浏览器数据库 IndexedDB 入门教程

总结

- Cookie主要用于“维持状态”，而非本地存储数据
- Web Storage是专门为浏览器提供的数据存储机制，不与服务端发生通信
- IndexedDB 用于客户端存储大量结构化数据

10、浏览器垃圾回收机制

介绍

浏览器的 Javascript 具有自动垃圾回收机制(GC:Garbage Collecation)，也就是说，执行环境会负责管理代码执行过程中使用的内存。其原理是：垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存。但是这个过程不是实时的，因为其开销比较大并且GC时停止响应其他操作，所以垃圾回收器会按照固定的时间间隔周期性的执行。

不再使用的变量也就是生命周期结束的变量，当然只可能是局部变量，全局变量的生命周期直至浏览器卸载页面才会结束。局部变量只在函数的执行过程中存在，而在这个过程中会为局部变量在栈或堆上分配相应的空间，以存储它们的值，然后在函数中使用这些变量，直至函数结束，而闭包中由于内部函数的原因，外部函数并不能算是结束。

还是上代码说明吧：


```
function fn1() {
    var obj = { name: 'hanzichi', age: 10 };
}
function fn2() {
    var obj = { name:'hanzichi', age: 10 };
    return obj;
}

var a = fn1();
var b = fn2();
```

我们来看代码是如何执行的。首先定义了两个function，分别叫做fn1和fn2，当fn1被调用时，进入fn1的环境，会开辟一块内存存放对象{name: 'hanzichi', age: 10}，而当调用结束后，出了fn1的环

境，那么该块内存会被js引擎中的垃圾回收器自动释放；在fn2被调用的过程中，返回的对象被全局变量b所指向，所以该块内存并不会被释放。

这里问题就出现了：到底哪个变量是没有用的？所以垃圾收集器必须跟踪到底哪个变量没用，对于不再有用的变量打上标记，以备将来收回其内存。用于标记的无用变量的策略可能因实现而有所区别，通常情况下有两种实现方式： 标记清除 和 引用计数 。引用计数不太常用，标记清除较为常用。

标记清除

js中最常用的垃圾回收方式就是标记清除。当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。

垃圾回收器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记（闭包）。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾回收器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。 到目前为止，IE9+、Firefox、Opera、Chrome、Safari的js实现使用的都是标记清除的垃圾回收策略或类似的策略，只不过垃圾收集的时间间隔互不相同。

引用计数

引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，则这个值的引用次数就是1。如果同一个值又被赋给另一个变量，则该值的引用次数加1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减1。当这个值的引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾回收器下次再运行时，它就会释放那些引用次数为0的值所占用的内存。


```
function test() {
    var a = {}; // a的引用次数为0
    var b = a; // a的引用次数加1，为1
    var c = a; // a的引用次数再加1，为2
    var b = {}; // a的引用次数减1，为1
}
```

Netscape Navigator3是最早使用引用计数策略的浏览器，但很快它就遇到一个严重的问题：循环引用。循环引用指的是对象A中包含一个指向对象B的指针，而对象B中也包含一个指向对象A的引用。


```
function fn() {
    var a = {};
    var b = {};
```

```
    a.pro = b;
    b.pro = a;
}
fn();
```

以上代码a和b的引用次数都是2，fn()执行完毕后，两个对象都已经离开环境，在标记清除方式下是没有问题的，但是在引用计数策略下，因为a和b的引用次数不为0，所以不会被垃圾回收器回收内存，如果fn函数被大量调用，就会造成内存泄露。在IE7与IE8上，内存直线上升。

我们知道，IE中有一部分对象并不是原生js对象。例如，其内存泄露DOM和BOM中的对象就是使用C++以COM对象的形式实现的，而COM对象的垃圾回收机制采用的就是引用计数策略。因此，即使IE的js引擎采用标记清除策略来实现，但js访问的 COM对象依然是基于引用计数策略 的。换句话说，只要在IE中涉及COM对象，就会存在循环引用的问题。

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.e = element;
element.o = myObject;
```

这个例子在一个DOM元素（element)与一个原生js对象（myObject)之间创建了循环引用。其中，变量myObject有一个属性e指向element对象；而变量element也有一个属性o回指myObject。由于存在这个循环引用，即使例子中的DOM从页面中移除，它也永远不会被回收。

举个栗子：

```
myObject.element = null;
element.o = null;

window.onload=function outerFunction() {
    var obj = document.getElementById("element");
    obj.onclick=function innerFunction(){};
    obj=null;
};
```

这段代码看起来没什么问题，但是obj引用了document.getElementById('element')，而document.getElementById('element')的onclick方法会引用外部环境中的变量，自然也包括obj，是不是很隐蔽啊。(在比较新的浏览器中在移除Node的时候已经会移除其上的event了，但是在老的浏览器，特别是ie上会有这个bug)

解决办法：

最简单的方式就是自己手工解除循环引用，比如刚才的函数可以这样

```js
myObject.element = null;
element.o = null;

window.onload=function outerFunction(){
    var obj = document.getElementById("element");
    obj.onclick=function innerFunction(){};
    obj=null;
};
```

将变量设置为null意味着切断变量与它此前引用的值之间的连接。当垃圾回收器下次运行时，就会删除这些值并回收它们占用的内存。

要注意的是，IE9+并不存在循环引用导致Dom内存泄露问题，可能是微软做了优化，或者Dom的回收方式已经改变

11、谈谈你对重绘和回流的理解

在讨论回流与重绘之前，我们要知道：

1. 浏览器使用流式布局模型 (Flow Based Layout)。
2. 浏览器会把 HTML 解析成 DOM ，把 CSS 解析成 CSSOM ， DOM 和 CSSOM 合并就产生了 Render Tree 。
3. 有了 RenderTree ，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上。
4. 由于浏览器使用流式布局，对 Render Tree 的计算通常只需要遍历一次就可以完成，但 table 及其内部元素除外，他们可能需要多次计算，通常要花3倍于同等元素的时间，这也是为什么要避免使用 table 布局的原因之一。

一句话：回流必将引起重绘，重绘不一定会引起回流

回流 (Reflow)

当 Render Tree 中部分或全部元素的尺寸、结构、或某些属性发生改变时，浏览器重新渲染部分或全部文档的过程称为回流。

会导致回流的操作：

- 页面首次渲染

- 浏览器窗口大小发生改变
- 元素尺寸或位置发生改变
- 元素内容变化（文字数量或图片大小等等）
- 元素字体大小变化
- 添加或者删除可见的 DOM 元素
- 激活 CSS 伪类（例如： :hover ）
- 查询某些属性或调用某些方法

一些常用且会导致回流的属性和方法：

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

重绘 (Repaint)

当页面中元素样式的改变并不影响它在文档流中的位置时（例如： color 、 background-color 、 visibility 等），浏览器会将新样式赋予给元素并重新绘制它，这个过程称为重绘。

性能影响

回流比重绘的代价要更高。

有时即使仅仅回流一个单一的元素，它的父元素以及任何跟随它的元素也会产生回流。

现代浏览器会对频繁的回流或重绘操作进行优化：

浏览器会维护一个队列，把所有引起回流和重绘的操作放入队列中，如果队列中的任务数量或者时间间隔达到一个阈值的，浏览器就会将队列清空，进行一次批处理，这样可以把多次回流和重绘变成一次。

当你访问以下属性或方法时，浏览器会立刻清空队列：

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

- `width` 、 `height`
- `getComputedStyle()`
- `getBoundingClientRect()`

因为队列中可能会有影响到这些属性或方法返回值的操作，即使你希望获取的信息与队列中操作引发的改变无关，浏览器也会强行清空队列，确保你拿到的值是最精确的。

如何避免

`CSS`

- 避免使用 `table` 布局。
- 尽可能在 `DOM` 树的最末端改变 `class` 。
- 避免设置多层内联样式。
- 将动画效果应用到 `position` 属性为 `absolute` 或 `fixed` 的元素上。
- 避免使用 `CSS` 表达式（例如： `calc()` ）。

`JavaScript`

- 避免频繁操作样式，最好一次性重写 `style` 属性，或者将样式列表定义为 `class` 并一次性更改 `class` 属性。
- 避免频繁操作 `DOM` ，创建一个 `documentFragment` ，在它上面应用所有 `DOM操作` ，最后再把它添加到文档中。
- 也可以先为元素设置 `display: none` ，操作结束后再把它显示出来。因为在 `display` 属性为 `none` 的元素上进行的 `DOM` 操作不会引发回流和重绘。
- 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。
- 对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

12、http和https的区别

- HTTP是 `明文传输` ，不安全的，HTTPS是 `加密传输` ，安全的多
- HTTP标准端口是 `80` ，HTTPS标准端口是 `443`
- HTTP不用认证证书 `免费` ，HTTPS需要认证证书 `要钱`
- `连接方式不同` ，HTTP三次握手，HTTPS中TLS1.2版本7次，TLS1.3版本6次
- HTTP在OSI网络模型中是在 `应用层` ，而HTTPS的TLS是在 `传输层`
- HTTP是无状态的，HTTPS是 `有状态` 的

13、HTTP2.0和HTTP1.X相比的新特性

- 新的二进制格式 （Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
- 多路复用 （MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。
- header压缩 ，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
- 服务端推送 （server push），同SPDY一样，HTTP2.0也具有server push功能

14、DNS

> 通俗易懂，了解什么是DNS及查询过程

15、CDN

> CDN是什么？使用CDN有什么优势？

五、Vue

最全的 Vue 面试题+详解答案

Vue.js 技术揭秘

六、React

高频前端面试题汇总之React篇（上）

高频前端面试题汇总之React篇（下）

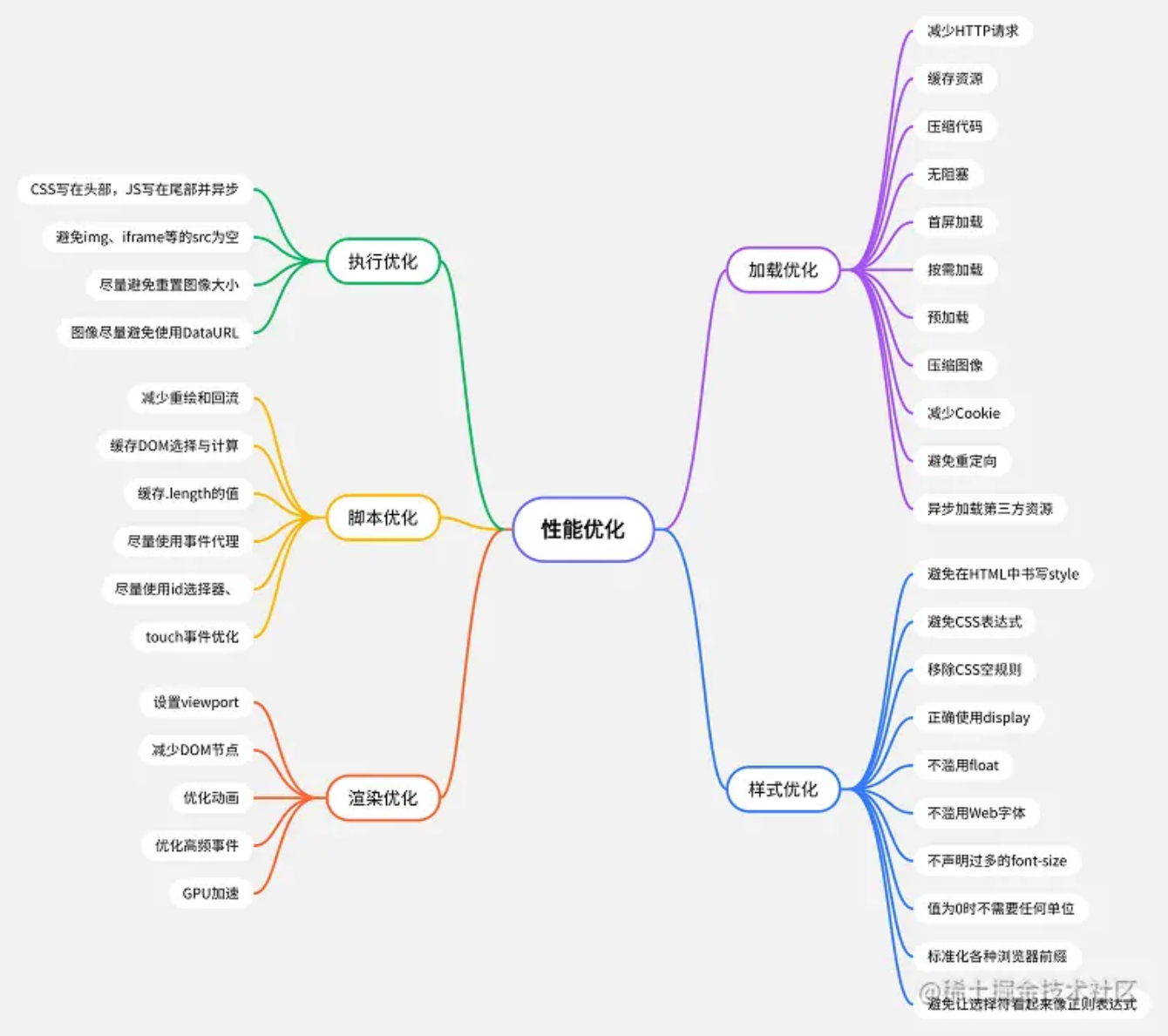
搞懂这12个Hooks，保证让你玩转React

React 源码剖析系列 - 生命周期的管理艺术

React Hooks 原理

深入理解 React 高阶组件

七、性能优化



前端性能优化手段从以下几个方面入手： 加载优化 、 执行优化 、 渲染优化 、 样式优化 、 脚本优化

加载优化 ： 减少HTTP请求、缓存资源、压缩代码、无阻塞、首屏加载、按需加载、预加载、压缩图像、减少Cookie、避免重定向、异步加载第三方资源

执行优化 ： CSS写在头部，JS写在尾部并异步、避免img、iframe等的src为空、尽量避免重置图像大小、图像尽量避免使用DataURL

渲染优化 ： 设置viewport、减少DOM节点、优化动画、优化高频事件、GPU加速

样式优化 ： 避免在HTML中书写style、避免CSS表达式、移除CSS空规则、正确使用display：display、不滥用float等

脚本优化 ： 减少重绘和回流、缓存DOM选择与计算、缓存.length的值、尽量使用事件代理、尽量使用id选择器、touch事件优化

加载优化

1. 减少HTTP请求 ： 尽量减少页面的请求数(首次加载同时请求数不能超过4个)，移动设备浏览器同时响应请求为4个请求(Android支持4个，iOS5+支持6个)

- 合并CSS和JS
- 使用CSS精灵图

2. 缓存资源 ： 使用缓存可减少向服务器的请求数，节省加载时间，所有静态资源都要在服务器端设置缓存，并且尽量使用长缓存(使用时间戳更新缓存)

- 缓存一切可缓存的资源
- 使用长缓存
- 使用外联的样式和脚本

3. 压缩代码 ： 减少资源大小可加快网页显示速度，对代码进行压缩，并在服务器端设置GZip

- 压缩代码(多余的缩进、空格和换行符)
- 启用Gzip

4. 无阻塞 ： 头部内联的样式和脚本会阻塞页面的渲染，样式放在头部并使用link方式引入，脚本放在尾部并使用异步方式加载

5. 首屏加载 ： 首屏快速显示可大大提升用户对页面速度的感知，应尽量针对首屏的快速显示做优化

6. 按需加载 ： 将不影响首屏的资源和当前屏幕不用的资源放到用户需要时才加载，可大大提升显示速度和降低总体流量(按需加载会导致大量重绘，影响渲染性能)

- 懒加载
- 滚屏加载
- Media Query加载

7. 预加载 ： 大型资源页面可使用Loading，资源加载完成后再显示页面，但加载时间过长，会造成用户流失

- 可感知Loading：进入页面时Loading
- 不可感知Loading：提前加载下一页

8. 压缩图像 ： 使用图像时选择最合适的格式和大小，然后使用工具压缩，同时在代码中用srcset来按需显示(过度压缩图像大小影响图像显示效果)

- 使用TinyJpg和TinyPng压缩图像
- 使用CSS3、SVG、IconFont代替图像
- 使用img的srcset按需加载图像
- 选择合适的图像：webp优于jpg，png8优于gif
- 选择合适的大小：首次加载不大于1014kb、不宽于640px
- PS切图时D端图像保存质量为80，M端图像保存质量为60

9. 减少Cookie：Cookie会影响加载速度，静态资源域名不使用Cookie
10. 避免重定向：重定向会影响加载速度，在服务器正确设置避免重定向
11. 异步加载第三方资源：第三方资源不可控会影响页面的加载和显示，要异步加载第三方资源

执行优化

1. CSS写在头部，JS写在尾部并异步
2. 避免img、iframe等的src为空：空src会重新加载当前页面，影响速度和效率
3. 尽量避免重置图像大小：多次重置图像大小会引发图像的多次重绘，影响性能
4. 图像尽量避免使用DataURL：DataURL图像没有使用图像的压缩算法，文件会变大，并且要解码后再渲染，加载慢耗时长

渲染优化

1. 设置viewport：HTML的viewport可加速页面的渲染

```
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1, minimum-
```

2. 减少DOM节点：DOM节点太多影响页面的渲染，尽量减少DOM节点
3. 优化动画

- 尽量使用CSS3动画
- 合理使用requestAnimationFrame动画代替setTimeout
- 适当使用Canvas动画：5个元素以内使用CSS动画，5个元素以上使用Canvas动画，iOS8+可使用WebGL动画

4. 优化高频事件：scroll、touchmove等事件可导致多次渲染

- 函数节流
- 函数防抖

- 使用requestAnimationFrame监听帧变化：使得在正确的时间进行渲染
- 增加响应变化的时间间隔：减少重绘次数

5. GPU加速 ：使用某些HTML5标签和CSS3属性会触发GPU渲染，请合理使用(过渡使用会引发手机耗电量增加)
    - HTML标签：video、canvas、webgl
    - CSS属性：opacity、transform、transition

样式优化

1. 避免在HTML中书写style
2. 避免CSS表达式 ：CSS表达式的执行需跳出CSS树的渲染
3. 移除CSS空规则 ：CSS空规则增加了css文件的大小，影响CSS树的执行
4. 正确使用display ：display会影响页面的渲染

- display:inline后不应该再使用float、margin、padding、width和height
- display:inline-block后不应该再使用float
- display:block后不应该再使用vertical-align
- display:table-*后不应该再使用float和margin

5. 不滥用float ：float在渲染时计算量比较大，尽量减少使用
6. 不滥用Web字体 ：Web字体需要下载、解析、重绘当前页面，尽量减少使用
7. 不声明过多的font-size ：过多的font-size影响CSS树的效率
8. 值为0时不需要任何单位 ：为了浏览器的兼容性和性能，值为0时不要带单位
9. 标准化各种浏览器前缀
    - 无前缀属性应放在最后
    - CSS动画属性只用-webkit-、无前缀两种
    - 其它前缀为-webkit-、-moz-、-ms-、无前缀四种：Opera改用blink内核，-o-已淘汰
10. 避免让选择符看起来像正则表达式 ：高级选择符执行耗时长且不易读懂，避免使用

脚本优化

1. 减少重绘和回流

- 避免不必要的DOM操作
- 避免使用document.write
- 减少drawImage
- 尽量改变class而不是style，使用classList代替className

2. 缓存DOM选择与计算 ： 每次DOM选择都要计算和缓存
3. 缓存.length的值 ： 每次.length计算用一个变量保存值
4. 尽量使用事件代理 ： 避免批量绑定事件
5. 尽量使用id选择器 ： id选择器选择元素是最快的
6. touch事件优化 ： 使用tap(touchstart和touchend)代替click(注意touch响应过快，易引发误操作)

八、webpack

> Webpack面试题
>
> 当面试官问Webpack的时候他想知道什么

九、TypeScript

> TypeScript免费视频图文教程（2W字）

十、数据结构与算法

算法这块呢，应该是很多人头疼的地方，没有其他方法，只能去LeetCode老老实实刷题

> JavaScript 数据结构与算法之美

十一、前端安全

1、说一说XSS攻击

就是攻击者想尽一切办法将可以执行的代码注入到网页中, 主要分为以下几种

存储型（server端）

场景： 见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等

攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中
2. 用户打开目标网站时，服务端将恶意代码从数据库中取出来，拼接在HTML中返回给浏览器
3. 用户浏览器在收到响应后解析执行，混在其中的恶意代码也同时被执行
4. 恶意代码窃取用户数据，并发送到指定攻击者的网站，或者冒充用户行为，调用目标网站的接口，执行恶意操作

反射型（Server端）

与存储型的区别在于，存储型的恶意代码存储在数据库中，反射型的恶意代码在URL上

场景： 通过 URL 传递参数的功能，如网站搜索、跳转等

攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

Dom 型(浏览器端)

DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞

场景： 通过 URL 传递参数的功能，如网站搜索、跳转等

攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

预防方案：

防止攻击者提交恶意代码，防止浏览器执行恶意代码

1. 对数据进行严格的输出编码 ：如HTML元素的编码，JS编码，CSS编码，URL编码等等； 避免拼接 HTML；Vue/React 技术栈，避免使用 v-html / dangerouslySetInnerHTML

2. CSP HTTP Header ，即 Content-Security-Policy、X-XSS-Protection

- 增加攻击难度，配置CSP(本质是建立白名单，由浏览器进行拦截)
- Content-Security-Policy: default-src 'self'-所有内容均来自站点的同一个源（不包括其子域名）
- Content-Security-Policy: default-src 'self' *.trusted.com-允许内容来自信任的域名及其子域名(域名不必须与CSP设置所在的域名相同)
- Content-Security-Policy: default-src yideng.com-该服务器仅允许通过HTTPS方式并仅从yideng.com域名来访问文档

3. 输入验证 ：比如一些常见的数字、URL、电话号码、邮箱地址等等做校验判断

4. 开启浏览器XSS防御 ： Http Only cookie，禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie

5. 验证码

2、说一说CSRF攻击

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的

攻击流程举例

1. 受害者登录 a.com，并保留了登录凭证（Cookie）
2. 攻击者引诱受害者访问了b.com
3. b.com 向 a.com 发送了一个请求：a.com/act=xx浏览器会默认携带a.com的Cookie
4. a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
5. a.com以受害者的名义执行了act=xx
6. 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

攻击类型

1. GET型 ： 如在页面的某个 img 中发起一个 get 请求
2. POST型 ： 通过自动提交表单到恶意网站
3. 链接型 ： 需要诱导用户点击链接

预防方案：

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。）

1. 同源检测 ： 通过Header中的Origin Header 、Referer Header 确定，但不同浏览器可能会有不一样的实现，不能完全保证

2. CSRF Token 校验 ： 将CSRF Token输出到页面中（通常保存在Session中），页面提交的请求携带这个Token，服务器验证Token是否

正确

3. 双重cookie验证 ：

   流程：

   - 步骤1：在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串（例如csrfcookie=v8g9e4ksfhw）
   - 步骤2：在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例POST www.a.com/comment?csr...
   - 步骤3：后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝

   优点：

   - 无需使用Session，适用面更广，易于实施
   - Token储存于客户端中，不会给服务器带来压力
   - 相对于Token，实施成本更低，可以在前后端统一拦截校验，而不需要一个个接口和页面添加

   缺点：

- Cookie中增加了额外的字段
- 如果有其他漏洞（例如XSS），攻击者可以注入Cookie，那么该防御方式失效
- 难以做到子域名的隔离
- 为了确保Cookie传输安全，采用这种防御方式的最好确保用整站HTTPS的方式，如果还没切HTTPS的使用这种方式也会有风险

4. Samesite Cookie属性 ：Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个 Cookie是个“同站 Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，Strict 为任何情况下都不可以作为第三方 Cookie ， Lax 为可以作为第三方 Cookie , 但必须是Get请求