- 减少使用昂贵的属性
- 不要使用@import

20.2.1. 内联首屏关键CSS

在打开一个页面，页面首要内容出现在屏幕的时间影响着用户的体验，而通过内联 `css` 关键代码能够使浏览器在下载完 `html` 后就能立刻渲染

而如果外部引用 `css` 代码，在解析 `html` 结构过程中遇到外部 `css` 文件，才会开始下载 `css` 代码，再渲染

所以， `CSS` 内联使用使渲染时间提前

注意：但是较大的 `css` 代码并不合适内联（初始拥塞窗口、没有缓存），而其余代码则采取外部引用方式

20.2.2. 异步加载CSS

在 `CSS` 文件请求、下载、解析完成之前， `CSS` 会阻塞渲染，浏览器将不会渲染任何已处理的内容

前面加载内联代码后，后面的外部引用 `css` 则没必要阻塞浏览器渲染。这时候就可以采取异步加载的方案，主要有如下

- 使用javascript将link标签插到head标签最后

JavaScript | 复制代码

```
// 创建link标签
const myCSS = document.createElement( "link" );
myCSS.rel = "stylesheet";
myCSS.href = "mystyles.css";
// 插入到header的最后位置
document.head.insertBefore( myCSS, document.head.childNodes[ document.head.childNodes.length - 1 ].nextSibling );
```

- 设置link标签media属性为noexis，浏览器会认为当前样式表不适用当前类型，会在不阻塞页面渲染的情况下再进行下载。加载完成后，将 `media` 的值设为 `screen` 或 `all` ，从而让浏览器开始解析CSS

HTML | 复制代码

```
<link rel="stylesheet" href="mystyles.css" media="noexist" onload="this.media='all'">
```

- 通过rel属性将link元素标记为alternate可选样式表，也能实现浏览器异步加载。同样别忘了加载完成之后，将rel设回stylesheet

```HTML
<link rel="alternate stylesheet" href="mystyles.css" onload="this.rel='stylesheet'">
```

20.2.3. 资源压缩

利用 `webpack` 、 `gulp/grunt` 、 `rollup` 等模块化工具，将 `css` 代码进行压缩，使文件变小，大大降低了浏览器的加载时间

20.2.4. 合理使用选择器

`css` 匹配的规则是从右往左开始匹配，例如 `#markdown .content h3` 匹配规则如下：

- 先找到h3标签元素
- 然后去除祖先不是.content的元素
- 最后去除祖先不是#markdown的元素

如果嵌套的层级更多，页面中的元素更多，那么匹配所要花费的时间代价自然更高

所以我们在编写选择器的时候，可以遵循以下规则：

- 不要嵌套使用过多复杂选择器，最好不要三层以上
- 使用id选择器就没必要再进行嵌套
- 通配符和属性选择器效率最低，避免使用

20.2.5. 减少使用昂贵的属性

在页面发生重绘的时候，昂贵属性如 `box-shadow` / `border-radius` / `filter` /透明度/ `:nth-child` 等，会降低浏览器的渲染性能

20.2.6. 不要使用@import

css样式文件有两种引入方式，一种是 `link` 元素，另一种是 `@import`

`@import` 会影响浏览器的并行下载，使得页面在加载时增加额外的延迟，增添了额外的往返耗时

而且多个 `@import` 可能会导致下载顺序紊乱

比如一个css文件 `index.css` 包含了以下内容： `@import url("reset.css")`

那么浏览器就必须先把 `index.css` 下载、解析和执行后，才下载、解析和执行第二个文件 `reset.css`

20.2.7. 其他

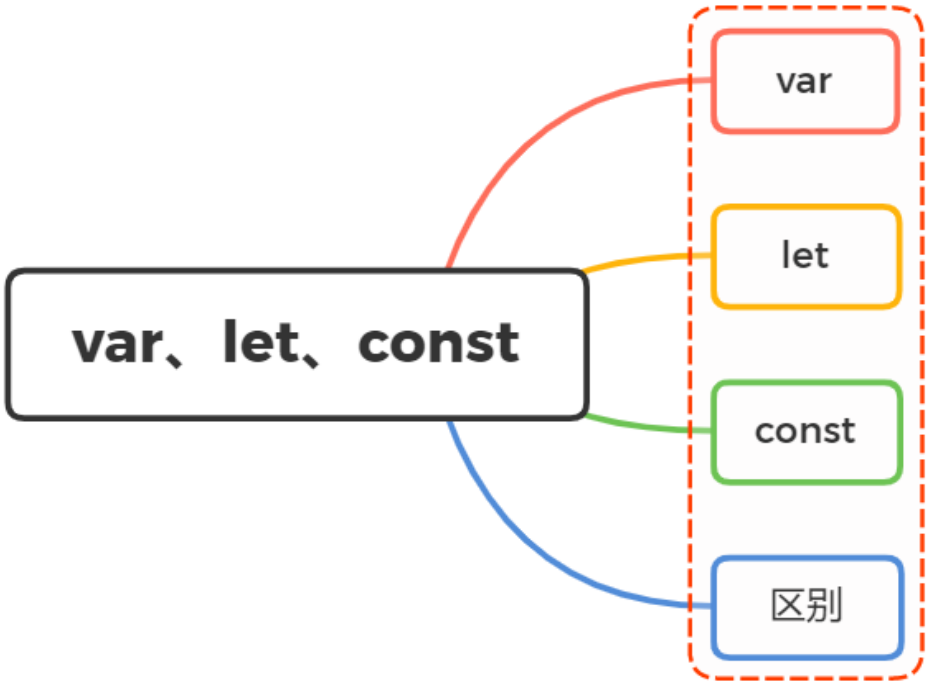
- 减少重排操作，以及减少不必要的重绘
- 了解哪些属性可以继承而来，避免对这些属性重复编写
- cssSprite，合成所有icon图片，用宽高加上backgroud-position的背景图方式显现出我们要的icon图，减少了http请求
- 把小的icon图片转成base64编码
- CSS3动画或者过渡尽量使用transform和opacity来实现动画，不要使用left和top属性

20.3. 总结

`css` 实现性能的方式可以从选择器嵌套、属性特性、减少 `http` 这三面考虑，同时还要注意 `css` 代码的加载顺序

ES6面试真题（10题）

1. 说说var、let、const之间的区别



1.1. var

在ES5中，顶层对象的属性和全局变量是等价的，用 `var` 声明的变量既是全局变量，也是顶层变量

注意：顶层对象，在浏览器环境指的是 `window` 对象，在 `Node` 指的是 `global` 对象

JavaScript 复制代码

```
var a = 10;
console.log(window.a) // 10
```

使用 `var` 声明的变量存在变量提升的情况

JavaScript 复制代码

```
console.log(a) // undefined
var a = 20
```

在编译阶段，编译器会将其变成以下执行

```JavaScript
var a
console.log(a)
a = 20
```

使用 `var` ，我们能够对一个变量进行多次声明，后面声明的变量会覆盖前面的变量声明

```JavaScript
var a = 20
var a = 30
console.log(a) // 30
```

在函数中使用使用 `var` 声明变量时候，该变量是局部的

```JavaScript
var a = 20
function change(){
    var a = 30
}
change()
console.log(a) // 20
```

而如果在函数内不使用 `var` ，该变量是全局的

```JavaScript
var a = 20
function change(){
   a = 30
}
change()
console.log(a) // 30
```

1.2. let

`let` 是 `ES6` 新增的命令，用来声明变量

用法类似于 `var` ，但是所声明的变量，只在 `let` 命令所在的代码块内有效

```JavaScript
{
    let a = 20
}
console.log(a) // ReferenceError: a is not defined.
```

不存在变量提升

```JavaScript
console.log(a) // 报错ReferenceError
let a = 2
```

这表示在声明它之前，变量 `a` 是不存在的，这时如果用到它，就会抛出一个错误

只要块级作用域内存在 `let` 命令，这个区域就不再受外部影响

```JavaScript
var a = 123
if (true) {
    a = 'abc' // ReferenceError
    let a;
}
```

使用 `let` 声明变量前，该变量都不可用，也就是大家常说的“暂时性死区”

最后， `let` 不允许在相同作用域中重复声明

```JavaScript
let a = 20
let a = 30
// Uncaught SyntaxError: Identifier 'a' has already been declared
```

注意的是相同作用域，下面这种情况是不会报错的

```JavaScript
let a = 20
{
    let a = 30
}
```

因此，我们不能在函数内部重新声明参数

```JavaScript
function func(arg) {
  let arg;
}
func()
// Uncaught SyntaxError: Identifier 'arg' has already been declared
```

1.3. const

`const` 声明一个只读的常量，一旦声明，常量的值就不能改变

```JavaScript
const a = 1
a = 3
// TypeError: Assignment to constant variable.
```

这意味着， `const` 一旦声明变量，就必须立即初始化，不能留到以后赋值

```JavaScript
const a;
// SyntaxError: Missing initializer in const declaration
```

如果之前用 `var` 或 `let` 声明过变量，再用 `const` 声明同样会报错

```JavaScript
var a = 20
let b = 20
const a = 30
const b = 30
// 都会报错
```

`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，值就保存在变量指向的那个内存地址，因此等同于常量

对于复杂类型的数据，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，并不能确保改变量的结构不变

```javascript
const foo = {};

// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123

// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

其它情况，`const` 与 `let` 一致

1.4. 区别

`var` 、 `let` 、 `const` 三者区别可以围绕下面五点展开：

- 变量提升
- 暂时性死区
- 块级作用域
- 重复声明
- 修改声明的变量
- 使用

1.4.1. 变量提升

`var` 声明的变量存在变量提升，即变量可以在声明之前调用，值为 `undefined`

`let` 和 `const` 不存在变量提升，即它们所声明的变量一定要在声明后使用，否则报错

```javascript
// var
console.log(a)  // undefined
var a = 10

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

1.4.2. 暂时性死区

`var` 不存在暂时性死区

`let` 和 `const` 存在暂时性死区，只有等到声明变量的那一行代码出现，才可以获取和使用该变量

```javascript
// var
console.log(a)  // undefined
var a = 10

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

1.4.3. 块级作用域

`var` 不存在块级作用域

`let` 和 `const` 存在块级作用域

```javascript
// var
{
    var a = 20
}
console.log(a)  // 20

// let
{
    let b = 20
}
console.log(b)  // Uncaught ReferenceError: b is not defined

// const
{
    const c = 20
}
console.log(c)  // Uncaught ReferenceError: c is not defined
```

1.4.4. 重复声明

`var` 允许重复声明变量

`let` 和 `const` 在同一作用域不允许重复声明变量

```javascript
// var
var a = 10
var a = 20 // 20

// let
let b = 10
let b = 20 // Identifier 'b' has already been declared

// const
const c = 10
const c = 20 // Identifier 'c' has already been declared
```

1.4.5. 修改声明的变量

`var` 和 `let` 可以

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变

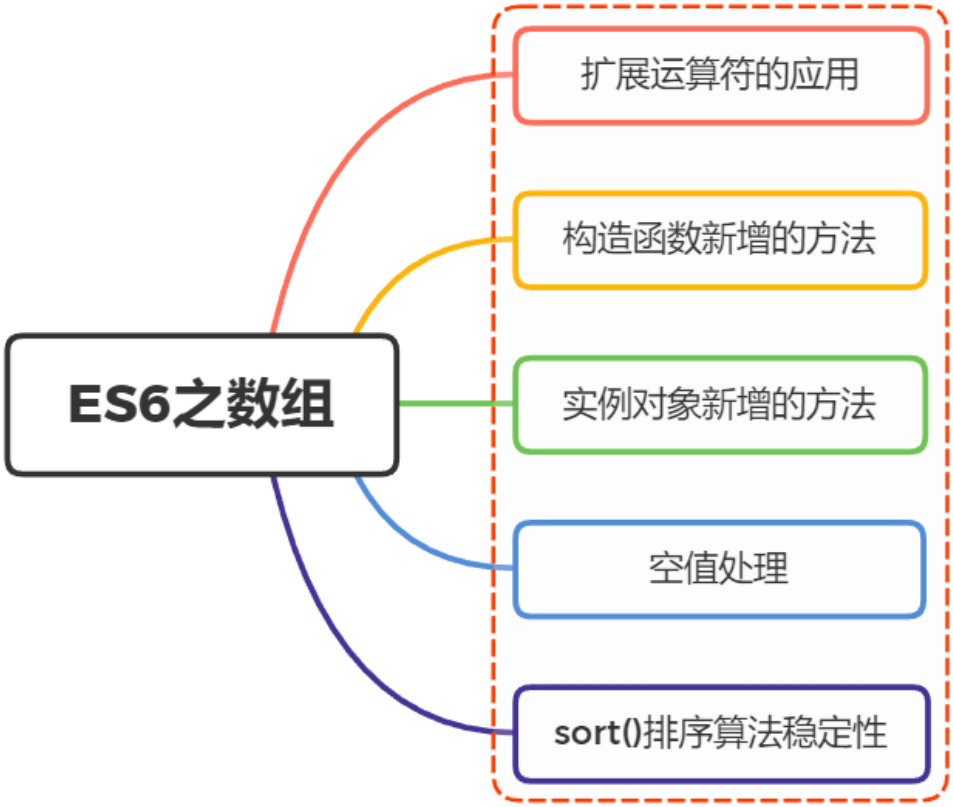
JavaScript | 复制代码

```
// var
var a = 10
a = 20
console.log(a)  // 20

//let
let b = 10
b = 20
console.log(b)  // 20

// const
const c = 10
c = 20
console.log(c) // Uncaught TypeError: Assignment to constant variable
```

1.4.6. 使用

能用 `const` 的情况尽量使用 `const` ，其他情况下大多数使用 `let` ，避免使用 `var`

2. ES6中数组新增了哪些扩展?



2.1. 扩展运算符的应用

ES6通过扩展元素符 `...` ，好比 `rest` 参数的逆运算，将一个数组转为用逗号分隔的参数序列

JavaScript | 复制代码

```
console.log(...[1, 2, 3])
// 1 2 3

console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5

[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

主要用于函数调用的时候，将一个数组变为参数序列

```JavaScript
function push(array, ...items) {
  array.push(...items);
}

function add(x, y) {
  return x + y;
}

const numbers = [4, 38];
add(...numbers) // 42
```

可以将某些数据结构转为数组

```JavaScript
[...document.querySelectorAll('div')]
```

能够更简单实现数组复制

```JavaScript
const a1 = [1, 2];
const [...a2] = a1;
// [1,2]
```

数组的合并也更为简洁了

```JavaScript
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

注意：通过扩展运算符实现的是浅拷贝，修改了引用指向的值，会同步反映到新数组

下面看个例子就清楚多了

```javascript
const arr1 = ['a', 'b',[1,2]];
const arr2 = ['c'];
const arr3  = [...arr1,...arr2]
arr[1][0] = 9999 // 修改arr1里面数组成员值
console.log(arr[3]) // 影响到arr3,['a','b',[9999,2],'c']
```

扩展运算符可以与解构赋值结合起来，用于生成数组

```javascript
 const [first, ...rest] = [1, 2, 3, 4, 5];
 first // 1
 rest  // [2, 3, 4, 5]

 const [first, ...rest] = [];
 first // undefined
 rest  // []

 const [first, ...rest] = ["foo"];
 first  // "foo"
 rest   // []
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错

```javascript
const [...butLast, last] = [1, 2, 3, 4, 5];
// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];
// 报错
```

可以将字符串转为真正的数组

```javascript
[...'hello']
// [ "h", "e", "l", "l", "o" ]
```

定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组

```JavaScript
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];

let map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three'],
]);

let arr = [...map.keys()]; // [1, 2, 3]
```

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错

```JavaScript
const obj = {a: 1, b: 2};
let arr = [...obj]; // TypeError: Cannot spread non-iterable object
```

2.2. 构造函数新增的方法

关于构造函数，数组新增的方法有如下：

- Array.from()
- Array.of()

2.2.1. Array.from()

将两类对象转为真正的数组：类似数组的对象和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）

```JavaScript
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

还可以接受第二个参数，用来对每个元素进行处理，将处理后的值放入返回的数组

```JavaScript
Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

2.2.2. Array.of()

用于将一组值，转换为数组

```JavaScript
Array.of(3, 11, 8) // [3,11,8]
```

没有参数的时候，返回一个空数组

当参数只有一个的时候，实际上是指定数组的长度

参数个数不少于 2 个时，`Array()` 才会返回由参数组成的新数组

```JavaScript
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

2.2.3. 实例对象新增的方法

关于数组实例对象新增的方法有如下：

- copyWithin()
- find()、findIndex()
- fill()
- entries()，keys()，values()
- includes()
- flat()，flatMap()

2.2.4. copyWithin()

将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组

参数如下：

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
- start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
- end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

```JavaScript
[1, 2, 3, 4, 5].copyWithin(0, 3) // 将从 3 号位直到数组结束的成员（4 和 5），复制到从 0 号位开始的位置，结果覆盖了原来的 1 和 2
// [4, 5, 3, 4, 5]
```

2.2.5. find()、findIndex()

`find()` 用于找出第一个符合条件的数组成员

参数是一个回调函数，接受三个参数依次为当前的值、当前的位置和原数组

```JavaScript
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

`findIndex` 返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回 `-1`

```JavaScript
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

这两个方法都可以接受第二个参数，用来绑定回调函数的 `this` 对象。

```JavaScript
function f(v){
  return v > this.age;
}
let person = {name: 'John', age: 20};
[10, 12, 26, 15].find(f, person);    // 26
```

2.2.6. fill()

使用给定值，填充一个数组

```javascript
['a', 'b', 'c'].fill(7)
// [7, 7, 7]

new Array(3).fill(7)
// [7, 7, 7]
```

还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置

```javascript
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

注意，如果填充的类型为对象，则是浅拷贝

2.2.7. entries()，keys()，values()

`keys()` 是对键名的遍历、 `values()` 是对键值的遍历， `entries()` 是对键值对的遍历

```javascript
or (let index of ['a', 'b'].keys()) {
  console.log(index);
}
// 0
// 1

for (let elem of ['a', 'b'].values()) {
  console.log(elem);
}
// 'a'
// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {
  console.log(index, elem);
}
// 0 "a"
```

2.2.8. includes()

用于判断数组是否包含给定的值

```JavaScript
[1, 2, 3].includes(2)     // true
[1, 2, 3].includes(4)     // false
[1, 2, NaN].includes(NaN) // true
```

方法的第二个参数表示搜索的起始位置，默认为 `0`

参数为负数则表示倒数的位置

```JavaScript
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
```

2.2.9. flat()，flatMap()

将数组扁平化处理，返回一个新数组，对原数据没有影响

```JavaScript
[1, 2, [3, 4]].flat()
// [1, 2, 3, 4]
```

`flat()` 默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将 `flat()` 方法的参数写成一个整数，表示想要拉平的层数，默认为1

```JavaScript
[1, 2, [3, [4, 5]]].flat()
// [1, 2, 3, [4, 5]]

[1, 2, [3, [4, 5]]].flat(2)
// [1, 2, 3, 4, 5]
```

`flatMap()` 方法对原数组的每个成员执行一个函数相当于执行 `Array.prototype.map()`，然后对返回值组成的数组执行 `flat()` 方法。该方法返回一个新数组，不改变原数组

```JavaScript
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()
[2, 3, 4].flatMap((x) => [x, x * 2])
// [2, 4, 3, 6, 4, 8]
```

`flatMap()` 方法还可以有第二个参数，用来绑定遍历函数里面的 `this`

2.2.10. 数组的空位

数组的空位指，数组的某一个位置没有任何值

ES6 则是明确将空位转为 `undefined` ，包括 `Array.from` 、扩展运算符、 `copyWithin()` 、 `fill()` 、 `entries()` 、 `keys()` 、 `values()` 、 `find()` 和 `findIndex()`

建议大家在日常书写中，避免出现空位

2.2.11. 排序稳定性

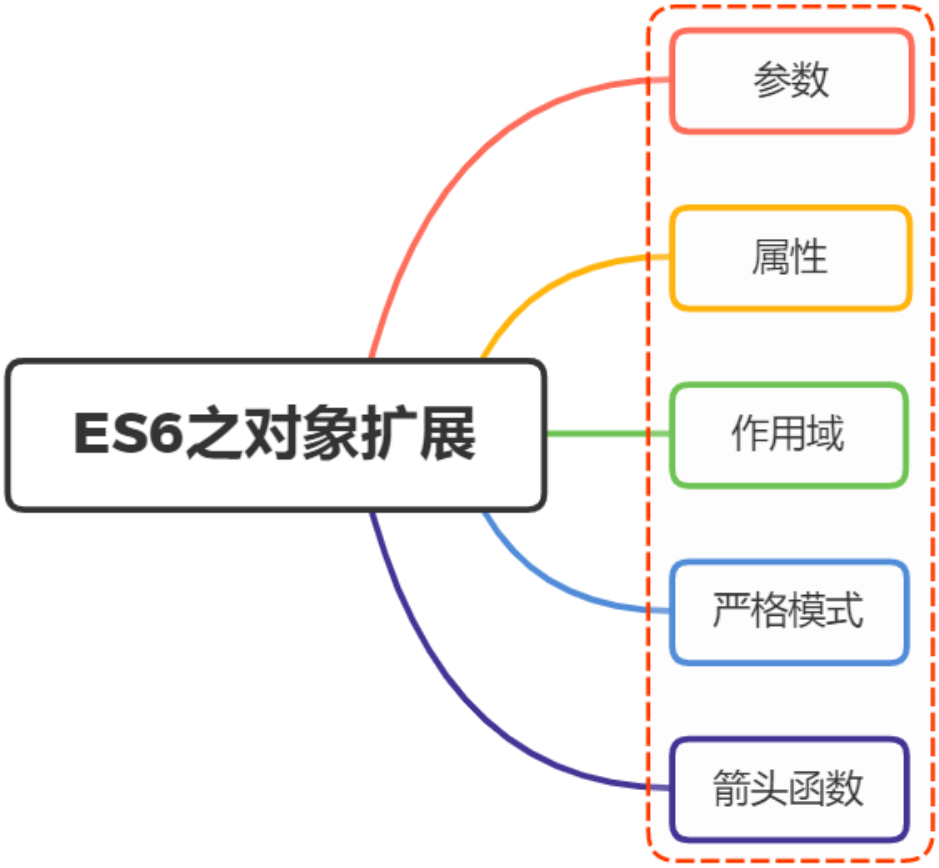
将 `sort()` 默认设置为稳定的排序算法

JavaScript | 复制代码

```
const arr = [
  'peach',
  'straw',
  'apple',
  'spork'
];

const stableSorting = (s1, s2) => {
  if (s1[0] < s2[0]) return -1;
  return 1;
};

arr.sort(stableSorting)
// ["apple", "peach", "straw", "spork"]
```

排序结果中， `straw` 在 `spork` 的前面，跟原始顺序一致

3. 函数新增了哪些扩展?



3.1. 参数

`ES6` 允许为函数的参数设置默认值

JavaScript | 复制代码

```
function log(x, y = 'World') {
  console.log(x, y);
}

console.log('Hello') // Hello World
console.log('Hello', 'China') // Hello China
console.log('Hello', '') // Hello
```

函数的形参是默认声明的，不能使用 `let` 或 `const` 再次声明

JavaScript 复制代码

```
function foo(x = 5) {
    let x = 1; // error
    const x = 2; // error
}
```

参数默认值可以与解构赋值的默认值结合起来使用

JavaScript 复制代码

```
function foo({x, y = 5}) {
  console.log(x, y);
}

foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面的 `foo` 函数，当参数为对象的时候才能进行解构，如果没有提供参数的时候，变量 `x` 和 `y` 就不会生成，从而报错，这里设置默认值避免

JavaScript 复制代码

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}

foo() // undefined 5
```

参数默认值应该是函数的尾参数，如果不是非尾部的参数设置默认值，实际上这个参数是没发省略的

JavaScript 复制代码

```
function f(x = 1, y) {
  return [x, y];
}

f() // [1, undefined]
f(2) // [2, undefined]
f(, 1) // 报错
f(undefined, 1) // [1, 1]
```

3.2. 属性

3.2.1. 函数的length属性

`length` 将返回没有指定默认值的参数个数

```JavaScript
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
```

`rest` 参数也不会计入 `length` 属性

```JavaScript
(function(...args) {}).length // 0
```

如果设置了默认值的参数不是尾参数，那么 `length` 属性也不再计入后面的参数了

```JavaScript
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
```

3.2.2. name属性

返回该函数的函数名

```JavaScript
var f = function () {};

// ES5
f.name // ""

// ES6
f.name // "f"
```

如果将一个具名函数赋值给一个变量，则 `name` 属性都返回这个具名函数原本的名字

```JavaScript
const bar = function baz() {};
bar.name // "baz"
```

`Function` 构造函数返回的函数实例， `name` 属性的值为 `anonymous`

```JavaScript
(new Function).name // "anonymous"
```

`bind` 返回的函数， `name` 属性值会加上 `bound` 前缀

```JavaScript
function foo() {};
foo.bind({}).name // "bound foo"

(function(){}).bind({}).name // "bound "
```

3.3. 作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域

等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的

下面例子中， `y=x` 会形成一个单独作用域， `x` 没有被定义，所以指向全局变量 `x`

```JavaScript
let x = 1;

function f(y = x) {
  // 等同于 let y = x
  let x = 2;
  console.log(y);
}

f() // 1
```

3.4. 严格模式

只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错

JavaScript | 复制代码

```
// 报错
function doSomething(a, b = a) {
  'use strict';
  // code
}

// 报错
const doSomething = function ({a, b}) {
  'use strict';
  // code
};

// 报错
const doSomething = (...a) => {
  'use strict';
  // code
};

const obj = {
  // 报错
  doSomething({a, b}) {
    'use strict';
    // code
  }
};
```

3.5. 箭头函数

使用“箭头”（`=>`）定义函数

JavaScript | 复制代码

```
var f = v => v;

// 等同于
var f = function (v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分

```JavaScript
var f = () => 5;
// 等同于
var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用 `return` 语句返回

```JavaScript
var sum = (num1, num2) => { return num1 + num2; }
```

如果返回对象，需要加括号将对象包裹

```JavaScript
let getTempItem = id => ({ id: id, name: "Temp" });
```

注意点：

- 函数体内的 `this` 对象，就是定义时所在的对象，而不是使用时所在的对象
- 不可以当作构造函数，也就是说，不可以使用 `new` 命令，否则会抛出一个错误
- 不可以使用 `arguments` 对象，该对象在函数体内不存在。如果要用，可以用 `rest` 参数代替
- 不可以使用 `yield` 命令，因此箭头函数不能用作 Generator 函数

4. 对象新增了哪些扩展?