JavaScript & ECMAScript6

大象组-张巍耀

ECMAScript

我只是一个标准，具体实现
你们来做吧😏~

JavaScript

JScript(IE)

ActionScript

Chrome

Node.js

Browser API

V8引擎

System API

alert、document…

os、vm…

Safari

Safari

Safari

Safari

Safari

WebKit

Chrome

时间

Blink(V8)

Chrome

Chrome

Chrome

Opera

WeChat

从 ECMAScript 2016 开始，

ECMAScript 将进入每年发布一次新标准的阶段

配合常青浏览器每 6 周发布一次新版的快速迭代日程，
加速 JavaScript/ECMAScript 的进化

原形链

instanceof本质

```
function A () {}

var a = new A();

console.log(a instanceof A);
console.log(a.__proto__ === A.prototype);
```

检测某个构造函数的prototype是否指向要检测的对象的原形链上(__proto__)

原形链是指 a.__proto__ 或 a.__proto__.__proto__，直到找到__proto__为null为止。

看图
说话

ECMAScript6

'use strict';

未定义的变量无法使用

对象key值重复报错(新版浏览器不报错)

函数参数重名

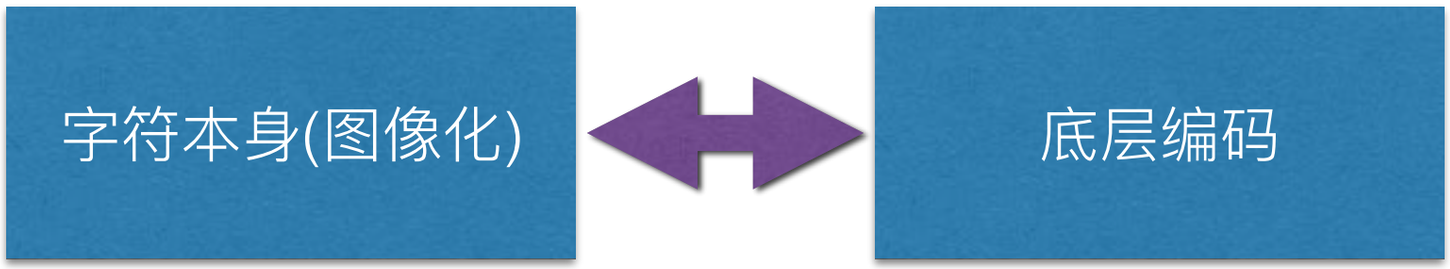
无法delete变量和只读属性

增加了一些关键字

禁用with、caller、callee、八进制表示法

建议一直开启严格模式，让错误发生的解释阶段，而不是运行阶段。
同时也能够防止他人开启严格模式后打包代码后影响自己的代码。

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict_mode/Transitioning_to_strict_mode
http://www.ruanyifeng.com/blog/2013/01/javascript_strict_mode.html
http://www.alloyteam.com/2012/06/it-is-time-to-use-the-javascript-strict-mode-strict-mode-to-enhance-the-efficiency-of-team-development/


字符本身(图像化)
底层编码

let定义变量

1. let定义的变量不会污染window中的变量。
2. 在作用域外访问不到，变量更安全。

ES5-var

```
if (true) {
  var a = 1;
}

console.log(window.a); // 1
console.log(a); // 1
```

ES6-let

```
if (true) {
  let a = 1;
}

console.log(window.a); // undefined
console.log(a); // Uncaught ReferenceError: a is not defined
```

let定义变量

```
for (var i = 0; i < 3; i++) {
  setTimeout(function () {
    console.log(i);
  }, 0)
}

// 3 3 3
```

```
for (let i = 0; i < 3; i++) {
  setTimeout(function () {
    console.log(i);
  }, 0)
}

// 0 1 2
```

let定义变量

```
for (var i = 0; i < 2; i++) {
  for (var i = 0; i < 3; i++) {
    console.log(i);
  }
}

// 0 1 2
```

```
for (let i = 0; i < 2; i++) {
  for (let i = 0; i < 3; i++) {
    console.log(i);
  }
}

// 0 1 2 0 1 2
```

新的闭包

1. 闭包有防止变量污染的作用。
2. 闭包会生成一个独立的作用域，也能防止其他人污染。

```
;(function () {
  // ...
})();
```

```
{
  // ...
}
```

const定义变量

```
const MAX_LEN = 10;
console.log(MAX_LEN); // 10
MAX_LEN = 9; // Uncaught TypeError: Assignment to constant variable.
```

```
const MAX_LEN = 10;
console.log(MAX_LEN); // 10

if (true) {
  const MAX_LEN = 11;
  console.log(MAX_LEN); // 11
}

{
  const MAX_LEN = 12;
  console.log(MAX_LEN); // 12
}

console.log(MAX_LEN); // 10
```

1. 定以后无法修改。
2. 不同作用域可以定义同名的const变量。
3. 常亮代码书写各式约定为全大写
   用 _ 分隔单词。

数组的解构赋值

```
var arr = ['arrA', 'arrB', 'arrC', 'arrD'];

var a = arr[0];
var b = arr[1];
var c = arr[2];

console.log(a, b, c); // arrA arrB arrC
```

```
let arr = ['arrA', 'arrB', 'arrC', 'arrD'];

let [a, b, c] = arr;

console.log(a, b, c); // arrA arrB arrC
```

省略賦值

```
var arr = ['arrA', 'arrB', 'arrC', 'arrD'];

var a = arr[0];
var b = arr[1];
var d = arr[3];
var e = arr[4];

console.log(a, b, d, e); // arrA arrB arrD undefined
```

```
let arr = ['arrA', 'arrB', 'arrC', 'arrD'];

let [a, b, , d, e] = arr;

console.log(a, b, d, e); // arrA arrB arrD undefined
```

不定参数

```
let arr = ['arrA', 'arrB', 'arrC', 'arrD'];

let [a, ...other] = arr;

console.log(a, other); // arrA ["arrB", "arrC", "arrD"]
```

对象的解构赋值

```
var obj = {
  a: 'objA',
  b: 'objB'
};

var a = obj.a;
var b = obj.b;
var c = obj.c;

console.log(a, b, c); // objA objB undefined
```

```
let obj = {
  a: 'objA',
  b: 'objB'
};

let {a, b, c} = obj;

console.log(a, b, c); // objA objB undefined
```

对象的解构赋值-变量更名

```
var obj = {
  a: 'objA',
  b: 'objB'
};

var myA = obj.a;
var myB = obj.b;
var myC = obj.c;

console.log(myA, myB, myC); // objA objB undefined
```

```
let obj = {
  a: 'objA',
  b: 'objB'
};

let {a: myA, b: myB, c: myC} = obj;

console.log(myA, myB, myC); // objA objB undefined
```

嵌套解构赋值

```
let obj = {
  a: 'objA',
  b: 'objB',
  arr: ['arrA', 'arrB', {objInArr: 'objInArr'}]
};

let {a: myA, b: myB, arr: [myArrA, myArrB, {objInArr: myObj}]} = obj;

console.log(myA, myB, myArrA, myArrB, myObj); // objA objB arrA arrB objInArr
```

解构赋值默认值

```
let arr = ['arrA', 'arrB'];

let [a = 'a', b = 'b', c = 'c'] = arr;

console.log(a, b, c); // arrA arrB c
```

```
let obj = {
  a: 'objA',
  b: 'objB'
};

let {a = 'a', b = 'b', c = 'c'} = obj;

console.log(a, b, c); // objA objB c
```

嵌套+默认值

```
let obj = {
  a: 'objA',
  b: 'objB',
  arr: ['arrA', 'arrB', {objInArr: 'objInArr'}]
};

let {
  a: myA = 'myA',
  b: myB = 'myB',
  c: myC = 'myC',
  arr:
    [
      myArrA = 'myArrA',
      myArrB = 'myArrB',
      {
        objInArr: myObj = 'myObj',
        other: otherObj = 'myOtherObj'
      }
    ]
  } = obj;

console.log(myA, myB, myC, myArrA, myArrB, myObj, otherObj);
// objA objB myC arrA arrB objInArr myOtherObj
```

函数的默认参数

```
function ajax ({method = 'get', url = 'default.com', data = {}, success, error}) {
  console.log(method, url, data, success, error);
}

ajax({
  data: {x: 'x'}
});

// get default.com Object {x: "x"} undefined undefined
```

函数的默认参数

```
function func1 ({x = 'defaulX', y = 'defaultY'} = {}) {
  console.log(x, y);
}

function func2 ({x, y} = {x: 'defaulX', y: 'defaultY'}) {
  console.log(x, y);
}

func1({x: 1.1, y: 1.2}); // 1.1 1.2
func2({x: 2.1, y: 2.2}); // 2.1 2.2

func1({x: 1.1}); // 1.1 "defaultY"
func2({x: 2.1}); // 2.1 undefined

func1(); // defaulX defaultY
func2(); // defaulX defaultY
```

rest参数

```
function add (one, ...values) {
  console.log(values === arguments); // false
  console.log(Array.isArray(values)); // true
  console.log(Array.isArray(arguments)); // false
  console.log(values); // [2, 3]
  console.log(one); // 1

  let sum = 0;
  values.forEach(function (value) {
    sum += value;
  });

  console.log(sum); // 5
}

add(1, 2, 3);
```

拓展运算符

```
let arr = [1, 2, 3];
console.log(arr); // [1, 2, 3]
console.log(...arr); // 1 2 3
```

```
let maxOld = Math.max.apply(null, [5, 3, 7]);
let maxNew = Math.max(...[5, 3, 7]);

console.log(maxOld); // 7
console.log(maxNew); // 7
```

```
let arr1 = [1, 3];
let arr2 = [2, 4];

console.log(arr1.concat(arr2)); // [1, 3, 2, 4]
console.log([...arr1, ...arr2]); // [1, 3, 2, 4]
```

箭头函数

```
let arrowFunc = a => a;
console.log(arrowFunc(1)); // 1

let func = function (a) {
  return a;
};
console.log(func(2)); // 2
```

```
let arrowFunc = (a) => {
  return a+1;
};
console.log(arrowFunc(1)); // 2

let func = function (a) {
  return a+1;
};
console.log(func(2)); // 3
```

```
let arr = [1, 2, 3];

let arr1 = arr.map(value => value * value);

let arr2 = arr.map(function (value) {
  return value * value;
});

console.log(arr1); // [1, 4, 9]
console.log(arr2); // [1, 4, 9]
```

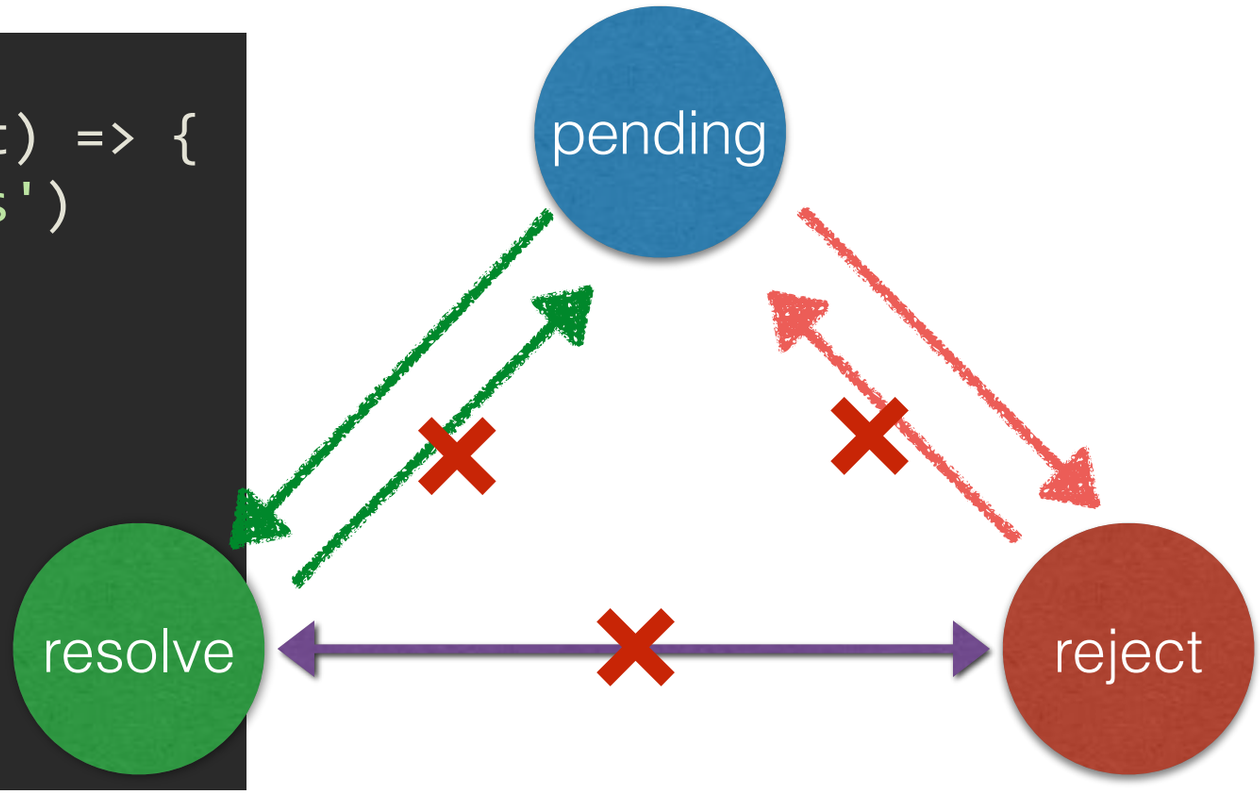
箭头函数this

```
let obj = {
  id: 123,
  func1: function () {
    setTimeout(function () {
      console.log(this.id);
    })
  },
  func2: function () {
    setTimeout(() => {
      console.log(this.id);
    })
  },
  func3: () => {
    console.log(this.id, );
    setTimeout(() => {
      console.log(this.id);
    });
  }
};

obj.func1(); // undefined
obj.func2(); // 123
obj.func3(); // undefined
```

Promise

1. Promise实例是一个状态机，pending、resolve、reject三个状态。
2. 状态转移只能从pending到resolve，或者pending到reject。
3. 实例默认为pending，一段时后会变成resolve或reject状态。
4. resolve触发then回调，reject触发catch回调。
5. 可以链式写多个then，但是最后一定是catch，catch还可以捕获then中的异常。

```
let p = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'success')
});

p.then((value) => {
  console.log(value); // success
}).catch((error) => {
  console.error(error);
});
```



类

```
class Parent {
  constructor (p = 'p') {
    this.p = p;
  }

  sayP () {
    console.log(this.p);
  }
}

class Child extends Parent{
  constructor (c = 'c') {
    super('ppp');
    this.c = c;
  }

  sayC () {
    console.log(this.c);
  }
}

var c = new Child();
c.sayC(); // c
c.sayP(); // ppp
```

```
function Parent(p) {
  this.p = p || 'p';
}
Parent.prototype.sayP = function () {
  console.log(this.p);
};

function Child (c) {
  this.c = c || 'c';
}
Child.prototype = new Parent('ppp');
Child.prototype.constructor = Child;
Child.prototype.sayC = function () {
  console.log(this.c);
};

var c = new Child();
c.sayC(); // c
c.sayP(); // ppp
```

1. 增加了class、extends、super关键字。
2. constructor就是构造函数
3. sayP、sayC就是prototype中的方法。
4. 子类的constructor方法中必须运行super，要先实例父类。

谢谢收听QwQ