ES 新特性

📎 课程时间安排.jpeg

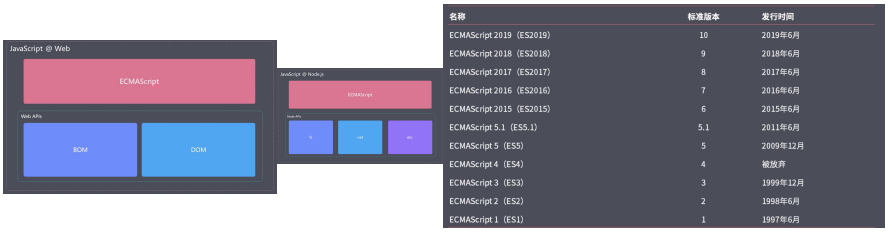
ECMAScript新特性、JavaScript异步编程

关键结果：

1. 掌握ES6以后新增的特性帮助你提高开发效率
2. 掌握解决异步回调的终极方案和原理

提前准备的问题:

1. ES6的新特性有哪些？为什么要用它们?
2. 除了ES6语法的内容之外，作者还带给了我们什么观点，进行记录！
    a. 语言和平台之间的关系
    b. 系统化的学习ECMAScript，形成一个完成的体系
    c. JavaScript是ECMAScript的扩展语言
3. 什么是异步编程？异步编程带来了什么好处，解决的问题是什么?



名称	标准版本	发行时间
ECMAScript 2019（ES2019）	10	2019年6月
ECMAScript 2018（ES2018）	9	2018年6月
ECMAScript 2017（ES2017）	8	2017年6月
ECMAScript 2016（ES2016）	7	2016年6月
ECMAScript 2015（ES2015）	6	2015年6月
ECMAScript 5.1（ES5.1）	5.1	2011年6月
ECMAScript 5（ES5）	5	2009年12月
ECMAScript 4（ES4）	4	被放弃
ECMAScript 3（ES3）	3	1999年12月
ECMAScript 2（ES2）	2	1998年6月
ECMAScript 1（ES1）	1	1997年6月

准备工作，安装nodemon

```
yarn add nodemon;
yarn nodemon xxx.js
```

ES2015 let与块级作用域

1. let声明的变量作用于块级作用域。

2. if、for、while内都是块级作用域;for的括号内声明的变量也是一个块级作用域;
3. let不会有声明提升

ES2015 const

1. const声明一个只读的常量
2. 声明时必须进行初始化定义值
3. 如果const声明了一个对象，依然可以为对象添加新属性，修改其原有属性。这是因为我们并没有改变该变量的内存地址，而是在其内存地址之内去修改它的部分值;

ES2015 数组的解构

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

const [foo, , baz, more, more2 = 'default'] = arr;

console.log(foo, baz, more, more2); // 1 3 undefined default
```

ES2015 对象的解构

```
const obj = { name: 'wubo', age: 27 };

const { name, age: myAge, sex, address = '杭州滨江' } = obj;
console.log(name, myAge, sex, address); // wubo、27、undefined、杭州滨江

const name = 1; // 报错
```

ES2015 模板字符串

```
const name = 'string';
const str = `this is es2015 ${name}`;
console.log(str); // this is es2015 string
console.log(`1 + 1 = ${1 + 1}`); // 1 + 1 = 2
```

ES2015 带标签的模板字符串

1. 对模板字符串进行加工
2. 做一个模板引擎

```
const name = 'wubo';
const gender = true;

function tagFunc (strings) {
  return strings;
}
function tagFunc2 (strings) {
  return 123;
}
function tagFunc3 (strings, name, gender) {
  const sex = gender ? 'man' : 'woman';

  return strins[0] + name + strins[1] + gender + strins[2];
}

console.log(tagFunc`hey, ${name} is a ${gender}.`); // [ 'hey, ', '
 is a ', '.' ]
console.log(tagFunc2`hey, ${name} is a ${gender}.`); // 123
console.log(tagFunc3`hey, ${name} is a ${gender}.`); // hey, wubo is
a man.
```

ES2015 字符串的扩展方法

includes()、startsWith()、endsWith()

```
const message = `Error: foo is not defined.`;

const { log } = console;
log(message.startsWith('Error')); // true
log(message.endsWith('.')); // true
log(message.includes('foo')); // true
```

ES2015 参数默认值

函数的形参可以设置默认值。

问题：1. 为什么形参要在最后一个，设置默认值才能起作用？我觉得关键点不是最不最后，而是形参是否被传入赋值为非undefined了。只不过写在最后是比较好认知的。

```
function foo (flag = true) {
    console.log(flag);
}
foo(); // true
foo(2); // 2
foo(undefined); // true
```

ES2015 剩余参数

1. 以前使用arguments来访问剩余参数
2. 因为是接收剩余参数，所以...必须写在最后

```
function foo (first, ...args) {
  console.log(first, args);
}

foo(1, 2, 3, 4); // 1 [ 2, 3, 4 ]
```

ES2015 展开数组

```
const arr = ['foo', 'bar', 'baz'];

console.log(...arr); // foo bar baz
console.log.apply(console, arr); // foo bar baz
```

ES2015 箭头函数

1. 简化了代码书写，也更加易读

```
const arr = [1, 2, 3, 4, 5, 6, 7];

console.log(arr.filter(i => i % 2)); // [ 1, 3, 5, 7 ]
console.log(arr.filter(function(i) {
  return i % 2;
})); // [ 1, 3, 5, 7 ]
```

ES2015 箭头函数与this

1. 箭头函数的this指向的是，它所在的作用域的this

```
const person = {
  name: 'wubo',
  sayHi: () => {
    console.log(`hi, my name is ${this.name}`);
  },
  sayHiAsync: function () {
    setTimeout(() => {
      console.log(this.name);
    });
  }
}

person.sayHi(); // hi, my name is undefined
person.sayHiAsync(); // wubo
```

ES2015 对象字面量的增强

1. 当属性名与，值的变量名一致，这可以省略掉冒号和冒号后面的内容
2. 如果是函数，则可以省略掉function，此时函数内的this就是当前对象；当然，也可以采用app, call, bind来改变；
3. 新增了计算属性名，使用[]包括计算内容

```
const bar = 123;
const obj = {
  foo: 123,
  bar,
  method1 () {
    console.log('method1');
    console.log(this.foo);
  },
  [bar + 1]: 124,
};
console.log(obj); // { '124': 124, foo: 123, bar: 123, method1: [Fun
ction: method1] }
obj.method1.apply({ foo: 121 }) // method1 121
```

ES2015 Object.assign

1. 接口任意个对象
2. 用第一个参数对象之后的对象，来覆盖第一个对象
3. 第一个对象的值会被改变。（内存地址不会变）

```
const source1 = {
  a: 123,
  b: 123
};

const source2 = {
  b: 456,
  d: 456
};

const target = {
  a: 456,
  c: 456
}

const result = Object.assign(target, source1, source2);
console.log(result); // { a: 123, c: 456, b: 456, d: 456 }
console.log(result === target); // true
```

ES2015 Object.is

1. 判断两个值是否是一样的
2. 建议还是使用===来判断

```
console.log(Object.is(+0, -0)); // false
console.log(Object.is(NaN, NaN)); // true
```

ES2015 Proxy

1. 为对象设置访问代理器，来监听对象的读写，可以理解为门卫。

```
const person = {
  name: 'zce',
```

```
  age: 20,
}

const personProxy = new Proxy(person, {
  get (target, property) {
    return property in target ? target[property] : 'default';
  },
  set (target, property, value) {
    if (property === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError(`${value} is not an int`);
      }
    }

    target[property] = value;
  }
});

personProxy.age = 100; // 正常
console.log(personProxy.gender); // default
personProxy.gender = 'man'; // 正常
personProxy.age = '123'; // TypeError: 123 is not an int
```

ES2015 Proxy 对比 defineProperty

1. defineProperty只能监听到对象的读写
2. Proxy可以监听到对象的读写、删除，方法的调用等等，详情见下表

handler方法	触发方式
get	读取某个属性
set	写入某个属性
has	in 操作符
deleteProperty	delete 操作符
getPropertypeOf	Object.getPropertypeOf()
setPrototypeOf	Object.setPrototypeOf()
isExtensible	Object.isExtensible()
preventExtensions	Object.preventExtensions()
getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor()
defineProperty	Object.defineProperty()
ownKeys	Object.getOwnPropertyNames(), Object.getOwnPropertySymbols()
apply	调用一个函数
construct	用 new 调用一个函数

3. Proxy还可以监听数组
4. Proxy是以非侵入的方式监管了对象的读写，就是不需要为某个值单独设置监管。

```
const person = {
  name: 'zce',
  age: 20,
```

```
}

const personProxy = new Proxy(person, {
  get (target, property) {
    return property in target ? target[property] : 'default';
  },
  set (target, property, value) {
    if (property === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError(`${value} is not an int`);
      }
    }

    target[property] = value;
  },
  deleteProperty (target, property) {
    console.log(`delete ${property}`);
    delete target[property];
  }
});

delete personProxy.age; // delete age
console.log(personProxy.gender); // default
personProxy.age = 123;
console.log(personProxy.age); // 123
personProxy.age = '123'; // TypeError: 123 is not an int
```

ES2015 Reflect

1. 统一的对象操作API（、、defineProperty、deleteProperty、get、getOwnPropertyDescriptor、getPrototypeOf、has、isExtensible、ownKeys、preventExtensions、set、setPrototypeOf），内部封装了一系列对对象的底层操作
   a. 所有API细节，请见MDN-Reflect
2. 静态类，无法使用new创建实例
3. Reflect成员方法就是Proxy处理对象的默认实现
4. 为什么要使用Reflect，其价值何在？答案：统一提供一套用于操作对象的API（以前是零散的，对新手不友好）

```
const person = {
```

```
  name: 'wubo',
  age: 27,
}
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

console.log(Reflect.has(person, 'name')); // true
console.log(Reflect.ownKeys(person)); // [ 'name', 'age' ]
console.log(Reflect.deleteProperty(person, 'name'), person.name); //
true undefined
console.log(Reflect.apply(Math.ceil, undefined, [1.75])); // 2
console.log(Reflect.apply(''.charAt, 'ponies', [3])); // i
console.log(Reflect.construct(Person, ['wubo', 27])); //
```

ES2015 Promise

1. 一种更优的异步编程解决方案
2. 解决了传统异步编程中回调函数嵌套过深的问题

ES2015 class类、静态方法、继承

```
class Person {
  constructor(name) {
    this.name = name;
  }

  sayHi () {
    console.log(`Hi, ${this.name}`);
  }

  static create (name) {
    return new Person(name);
  }
}

class Student extends Person {
```

```
  constructor(name, number) {
    super(name);
    this.number = number;
  }

  hello () {
    super.sayHi();
    console.log(`my school number is ${this.number}`);
  }
}

const person = new Person('吴博');
person.sayHi(); // Hi, 吴博
console.log(Person.create('wubo')); // { name: 'wubo' }
const s = new Student('wubo', '100');
s.hello(); // Hi, wubo、my school number is 100
```

ES2015 Set

1. 内部的值不允许重复， new Set()
2. has()、delete()、clear()
3. 数组去重

```
const s = new Set();

s.add(1).add(2).add(3).add(4).add(2);
console.log(s); // Set { 1, 2, 3, 4 }

const arr = [1, 2, 3, 4, 1, 2, 3, 4];
console.log(Array.from(new Set(arr))); // [ 1, 2, 3, 4 ]
console.log([...new Set(arr)]); // [ 1, 2, 3, 4 ]
```

ES2015 Map

1. Object是键-值对的，而键只能是字符串。Map改善了这一点，Map的键可以是任意类型
2. API：set、get、clear、delete

```
const m = new Map();
const tom = { name: 'tom' };
```

```
3
4 m.set(tom, 90);
5 console.log(m); // Map { { name: 'tom' } => 100 }
6 console.log(m.get(tom)); // 90
7 console.log(m.has(tom)); // true
8
9 m.set([1, 2, 3], [1, 2, 3, 4]);
10 m.forEach((value, key) => {
11   console.log(value, key);
12 });
13 // 90 { name: 'tom' }
14 // [ 1, 2, 3, 4 ] [ 1, 2, 3 ]
```

ES2015 Symbol

1. 表示一个独一无二的值。
2. Symbol.for() 可以创建一个值，for内部建立了一个字符串与symbol的对应关系，也就意味着for接受了非字符串时也会将其转换为字符串
3. for...in 、Object.keys、JSON.stringify 都会忽视掉Symbol创建的属性；需要通过Object.getOwnPropertySymbols(obj)来获取
4. Symbol.iterator、Symbol.hasInstance、Symbol.toStringTag、
5. 可以解决的问题（ES6以前）
    a. 对象的键只能是字符串，难以避免我们为对象新增属性时，会形成覆盖的情况
    b. 创建一个私有成员

```
1 const name = Symbol();
2 const obj = {
3   [name]: 123,
4   say() {
5     console.log(this[name])
6   }
7 };
8 console.log(Symbol('foo') === Symbol('foo')); //false
9 obj.say(); // 123
```

ES2015 for...of循环

1. 作为遍历所有数据结构的统一方式
2. 因为是for循环，所以可以用break、continue;

3. 可以循环arguments、DOM节点列表、Set、Map

```
const m = new Map();
m.set('foo', '123');
m.set('bar', '345');

for (const [key, value] of m) {
  console.log(key, value); // foo 123 、 bar 345
}

const s = new Set(['foo', 'bar']);
for( const item of s) {
  console.log(item); // foo 、 bar
}

const arr = [1, 2, 3, 4];
for(const value of arr) {
  console.log(value); // 1、2、3、4
}
```

ES2015 可迭代接口

1. ES2015提供了Iterable接口、实现Iterable接口就是for...of的前提
    a. 内部挂在一个iterator方法
2. 类似于toString一样，所有类型都可以使用toString，就是因为他们都实现了统一的接口

```
const set = new Set(['foo', 'bar', 'baz']);

const iterator = set[Symbol.iterator]();
console.log(iterator.next()); // { value: 'foo', done: false }
console.log(iterator.next()); // { value: 'bar', done: false }
console.log(iterator.next()); // { value: 'baz', done: false }
console.log(iterator.next()); // { value: undefined, done: true }
console.log(iterator.next()); // { value: undefined, done: true }
```

ES2015 实现可迭代接口

1. 为对象实现可迭代接口
    a. 添加iterable接口（可迭代接口），

i. return 一个iterator接口（迭代器接口）

1. return 一个 next方法

a. return一个iterationResult（迭代结果接口）

```
const obj = {
  store: ['foo', 'bar', 'gaz'],
  [Symbol.iterator]: function () {
    let index = 0;
    const self = this;
    return {
      next: function() {
        return {
          value: self.store[index],
          done: index++ >= self.store.length
        }
      }
    }
  }
}
for (const value of obj) {
  console.log(value); // foo、bar、gaz
}
```

ES2015 迭代器模拟

1. 对外提供统一迭代接口

```
const todos = {
  life: ['吃饭', '睡觉', '打豆豆'],
  learn: ['语文', '数学', '外语'],
  work: ['喝茶'],
  each: function(callback) {
    const all = [...this.life, ...this.learn, ...this.work];

    for (const item of all) {
      callback(item);
    }
  },
  [Symbol.iterator]: function() {
```

```
    const all = [...this.life, ...this.learn, ...this.work];
    let index = 0;

    return {
      next: function () {
        return {
          value: all[index],
          done: index++ >= all.length
        }
      }
    }
  }
}

todos.each(function (item) {
  console.log(item); // 吃饭、睡觉、打豆豆、语文、数学、外语、喝茶
});

for (const item of todos) {
  console.log(item); // 吃饭、睡觉、打豆豆、语文、数学、外语、喝茶
}
```

ES2015 生成器(Generator)

1. 避免异步编程中回调嵌套过深、提供更好的异步编程解决方案

```
function * foo() {
  console.log('1111');
  yield 100;
  console.log('2222');
  yield 200;
  console.log('3333');
  yield 300;
}

const generator = foo();
console.log(generator.next()); // 1111、{ value: 100, done: false }
console.log(generator.next()); // 2222、{ value: 200, done: false }
console.log(generator.next()); // 3333、{ value: 300, done: false }
```

```
console.log(generator.next()); // { value: undefined, done: true }
```

ES2015 生成器应用

```
// 案例1：发号器

function * createIdMaker() {
  let id = 1;
  while (true) {
    yield id++;
  }
}

const idMaker = createIdMaker();

console.log(idMaker.next().value); // 1
console.log(idMaker.next().value); // 2
console.log(idMaker.next().value); // 3
console.log(idMaker.next().value); // 4

// 案例2：使用Generator 函数实现 iterator 方法

const todos = {
  life: ['吃饭', '睡觉', '打豆豆'],
  learn: ['语文', '数学', '外语'],
  work: ['喝茶'],
  [Symbol.iterator]: function * () {
    const all = [...this.life, ...this.learn, ...this.work];
    for (const item of all) {
      yield item;
    }
  }
}

for (const item of todos) {
  console.log(item); // 吃饭、睡觉、打豆豆、语文、数学、外语、喝茶
}
```

ES2015 ES Modules

1. 语言层面的模块化标准
2. 后面会介绍到它与commonjs，以及其他标准做一个对比

ES2016 概述

1. Array.prototype.includes：返回布尔值；indexOf不能查找到NaN，includes可以；
2. 指数运算符：Math.pow(2, 10) 相当于 2 ** 10

ES2017 概述

1. Object.values
2. Object.entries
3. Object.getOwnPropertyDescriptors
4. String.prototype.padStart、String.prototype.padEnd
5. 函数参数中添加尾逗号
6. async、await

```
// ECMAScript 2017

const obj = {
  foo: 'value1',
  bar: 'value2',
}

// Object.values()
// 以数组的形式返回对象内所有的value
console.log(Object.values(obj)); // [ 'value1', 'value2' ]

// Object.entries(obj)
// 以数组的形式返回对象内所有的键值对
console.log(Object.entries(obj)); // [ [ 'foo', 'value1' ], [ 'bar',
'value2' ] ]
for (const [key, value] of Object.entries(obj)) {
  console.log(key, value); // foo value1、bar value2
}
console.log(new Map(Object.entries(obj))); // Map { 'foo' => 'value
1', 'bar' => 'value2' }

// Object.getOwnPropertyDescriptors
// 用来帮助我们获取对象中属性完整的描述信息
```

```
// ES2015中出现的对象内的get、set是不能通过Object.assign复制过去的, 因为Object.assign会把它们当做普通的属性去复制
const p1 = {
  firstName: 'Lei',
  LastName: 'Wang',
  get fullName() {
    return this.firstName + ' ' + this.LastName;
  }
};
console.log(p1.fullName); // Lei Wang

const p2 = Object.assign({}, p1);
p2.firstName = 'zce';
// 可能有人会以为此处会是zce Wang, 造成这种现象的原因是Object.assign会把fullName当做普通的属性去复制
console.log(p2.fullName); // Lei Wang;

const descriptors = Object.getOwnPropertyDescriptors(p1);
const p3 = Object.defineProperties({}, descriptors);
p3.firstName = 'zce';
console.log(p3.fullName); // zce Wang


// String.prototype.padStart、String.prototype.padEnd
//
const books = {
  html: 5,
  css: 16,
  javascript: 128,
};
for (const [name, count] of Object.entries(books)) {
  console.log(name, count);
}
/*
html 5
css 16
javascript 128
*/
for (const [name, count] of Object.entries(books)) {
  console.log(`${name.padEnd(16, '-')}|${count.toString().padStart(3
```

```
  , 0)}`)
}
/*
html------------|005
css-------------|016
javascript------|128
*/

// 在函数参数中添加尾逗号
// 方便后续参数的添加，方便添加了新参数后merge request中减少行改变
function foo (
  bar,
  baz,
) {
  console.log(bar, baz);
}
foo('wubo', 'wubo2', ) // wubo、wubo2
```

随堂测验

您的手机号：*

您的回答：17671686923

1.下列说法中正确的是 () [分值：5]

您的回答：B.JavaScript是ECMAScript的扩展语言 ✗

正确答案为：B.JavaScript是ECMAScript的扩展语言|D.JavaScript语言本身指的就是ECMAScript

2.以选项中下不正确的是 () [分值：5]

您的回答：C.ECMAScript2016就是ES6 ✗

正确答案为：B.2016年开始ES保持每年一个版本的迭代|C.ECMAScript2016就是ES6

3.ES6在原有基础上进行了哪些改动() [分值：5]

您的回答：A.解决原有语法上的一些不足和问题 ¦ B.对原有语法进行了加强 ¦ D.全新的数据类型和数据结构 ✓ （得分：5）

4. `let`关键字声明的变量可以在声明前使用（） [分值：5]

您的回答：B.错 ✓ （得分：5）

5. 下列有关函数参数说法正确的是 （） [分值：5]

您的回答：A.函数中的默认参数最好写在其他参数的后面 ¦ C.`rest` 参数之后不能再有其他参数 ¦ D.`rest`参数可以替换`arguments`类数组的使用 ✓ （得分：5）

6. 下列关于Promise的说法正确的是（） [分值：5]

您的回答：A.`promise`在任务队列中属于微任务 ¦ D.`promise`可以实现链式写法 ✗

正确答案为：A.`promise`在任务队列中属于微任务|B.`promise`中不管有没有异步函数，它的执行机制都是异步的|D.`promise`可以实现链式写法

7. 下列列出的形式可以对对象的数据实施拦截的是（） [分值：5]

您的回答：A.`Object.defineProperty()` ¦ B.`new Proxy()` ¦ C.`new Reflect()` ✗

正确答案为：A.`Object.defineProperty()`|B.`new Proxy()`

8.关于浅复制和深复制的说法，下列说法正确的是 [分值：5]

您的回答：A.浅层复制：只复制指向对象的指针，而不复制引用对象本身。 ¦ C.如果是浅复制，修改一个对象可能会影响另外一个对象 ✗

正确答案为：A.浅层复制：只复制指向对象的指针，而不复制引用对象本身。|B.深层复制：复制引用对象本身。|C.如果是浅复制，修改一个对象可能会影响另外一个对象

9. 下列有关class的说法正确的是（） [分值：5]

您的回答：B.子类的构造器中`super`关键字的前面不能出现`this`关键字 ¦ C.`new`实例对象的时候，其实就是调用类的构造器方法，且返回这个实例对象 ¦ D.静态方法的实现需要在方法名的前面加上`static`关键字 ✓ （得分：5）