

ES6基础



成钞公司印钞管理部 李宾

2018-01-09

ECMAScript 的发展历史

总的来讲，目前浏览器运行的javascript版本是ES5，2015年6月正式发布了ES6(es2015)，2017年6月正式发布ES7(ES2017)

延伸阅读：[ECMAScript的发展](#)

浏览器的兼容性

目前几大浏览器对ES6已完美支持，但对行业来讲，有许多特性还需要用Babel转码为ES5方可使用，浏览的兼容情况可点击[这里](#)查看。

		Compilers/polyfills										Desktop browsers												
		97%	56%		71%	48%	59%	17%	5%	11%	96%	96%	96%	94%	97%	97%	97%		97%	97%	97%	99%	99%	99%
Feature name	Current browser	Traceur	Babel + core-js ^[2]	Closure	Type: Script + core-js	es6-shim	Kong 4.14 ^[3]	IE 11	Edge 15	Edge 16	Edge 17 Preview	FF 52 ESR	FF 56	FF 57	FF 58 Beta	FF 59 Nightly	CH 62, OP 49 ^[1]	CH 63, OP 50 ^[1]	CH 64, OP 51 ^[1]	CH 65, OP 52 ^[1]	SF 10.1	SF 11	SF TP	
Optimisation																								
<ul style="list-style-type: none">proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	
Syntax																								
<ul style="list-style-type: none">default function parameters	7/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	
<ul style="list-style-type: none">rest parameters	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
<ul style="list-style-type: none">spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	
<ul style="list-style-type: none">object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	
<ul style="list-style-type: none">for...of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	
<ul style="list-style-type: none">octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
<ul style="list-style-type: none">template literals	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
<ul style="list-style-type: none">RegExp "v" and "u" flags	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
<ul style="list-style-type: none">destructuring declarations	22/22	20/22	21/22	20/22	15/22	0/22	0/22	0/22	22/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	
<ul style="list-style-type: none">destructuring assignment	24/24	23/24	24/24	21/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	
<ul style="list-style-type: none">destructuring parameters	24/24	19/24	21/24	18/24	16/24	0/24	0/24	0/24	23/24	23/24	23/24	21/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	
<ul style="list-style-type: none">Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	
<ul style="list-style-type: none">new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	
Bindings																								
<ul style="list-style-type: none">const	16/16	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	
<ul style="list-style-type: none">let	12/12	10/12	10/12	10/12	10/12	0/12	0/12	10/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	
<ul style="list-style-type: none">block-level function declaration^[15]	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
Functions																								
<ul style="list-style-type: none">arrow functions	13/13	11/13	9/13	10/13	9/13	0/13	0/13	0/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	
<ul style="list-style-type: none">class	24/24	17/24	19/24	13/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	
<ul style="list-style-type: none">super	8/8	7/8	4/8	5/8	7/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	

变量定义

```
var dom = $('#container');  
let dom = $('#container');  
const PI = 3.1415;
```

js

JS 中变量可不定义也可使用，但如果不使用var关键字定义，变量的作用域会提升，污染全局变量，所以在实际使用中是 **建议变量在使用前必须定义**。

变量的作用域

! let 是块级(花括号或圆括号)作用域，let声明的变量在同一作用域内不可重新声明。

```
var a = 2;
if (true) {
  var a = 3;
}
console.log(a);
// 3
```

js

```
let a = 2;
if (true) {
  let a = 3;
}
console.log(a);
// 2
```

js

! const表示定义常量，常量不允许修改其值。但常量的内容是数组或对象时，其内容可被修改，但常量本身不能被重新定义。

```
const PI = 3.1415926;  
PI = 2; // 此时会报错
```

js

let的妙用

```
for(var i=0;i<10;i++){  
  setTimeout(function(){  
    console.log(i);  
  },0);  
}
```

js

此时将输出什么结果？

```
for(let i=0;i<10;i++){  
  setTimeout(function(){  
    console.log(i);  
  },0);  
}
```

js

在需要传入idx的异步场景里，使用let是最优解。

变量的解构赋值

```
let [a,b] = [2,3];  
a == 2;  
b == 3;
```

js

```
function add([a,b]){  
    return a+b;  
}
```

js

```
let param = {  
    name: 'zhangsan',  
    age: 17  
};  
let {name, age} = param;  
name == 'zhangsan';  
age == 17;
```

js

```
// 交换变量值  
let a = 2, b = 1, temp = 0;;  
temp = a, a = temp, temp = b;  
[a, b] = [b, a];
```

js

字符串扩展

! String.includes()

```
let content = '烽火戏诸侯';  
if(content.indexOf('烽火')>-1){  
    // ...  
}  
content.includes('烽火');
```

js

、、

字母串拼接，以设备下拉select为例：

```
var machines = [{  
    name: '设备1',  
    value: 1  
}, {  
    name: '设备2',  
    value: 2  
}, {  
    name: '设备3',  
    value: 3  
}];  
  
var strOptions = '';  
for (var i = 0; i < machines.length; i++) {  
    var item = machines[i];  
    strOptions += '<option value="' + item.value + '>' + item.name + '</option>'  
}  
$('select').html(strOptions);
```

数组遍历

! Array.map()

接收函数 function(item,index){ }, 返回数组

```
machines.map(function(item) {  
    strOptions += `<option value="${item.value}">${item.name}</option>`;   
})  
$('select').html(strOptions);
```

js

```
<select>  
    <option value="1">设备1</option>  
</select>
```

html

数组遍历



Array.forEach()

只接收函数，不返回值

```
for (var i = 0; i < machines.length; i++) {  
  var item = machines[i];  
  console.log(item);  
}
```

js

```
machines.forEach(function(item,i)){  
  console.log(item.name);  
}
```

js

数组排序、push、pop

```
// 数组排序  
machines.sort(function(a,b){  
    return a.id-b.id  
});
```

js

```
// 向数组追加数据  
machines.push({  
    name:'设备4',  
    value:4  
})
```

js

```
// 将数组最后一个值删除  
machines.pop();
```

js

数组join



Array.join()

```
strOptions = machines.map(function(item) {  
    return `<option value="${item.value}">${item.name}</option>`;  
})  
$('select').html(strOptions.join(''));
```

js

```
let machines = [{  
    name: '设备1',  
    value: 1  
}, {  
    name: '设备2',  
    value: 2  
}, {  
    name: '设备3',  
    value: 3  
}];  
let str = machines.map(item => `<option value="${item.value}">${item.name}</option>`)  
$('select').html(str.join(''));
```

js

扩展运算符

```
let a = [2,3,4];  
let b = [3,4,5];  
let c = [...a,...b];  
c == a.concat(b);
```

js

用法示例，在数组的头部添加一个值，其余值全部后移一位。

```
let a = [2,3,4,5];  
let b = [10,...a];  
b == [10].concat(a);
```

js



Array.includes()

```
let a = [1,2,3,4];  
a.includes(3) == true;
```

js

Array.findIndex()

```
let machines = [{
  name: '设备1',
  value: 1
}, {
  name: '设备2',
  value: 2
}, {
  name: '设备3',
  value: 3
}];
let idx = machines.findIndex(item => item.name == '设备2');
// console.log(machines[idx].value);
```

js

箭头函数

```
function add(a,b){  
  return a+b;  
}  
let add = (a,b)=>a+b;
```

js

函数的默认值

```
let add = (a,b)=>{  
  if(typeof b == 'undefined'){  
    b = 0;  
  }  
  return a+b;  
}  
add(3) == 3;
```

js

```
let add = (a,b)=>{  
  b = b || 0;  
  return a+b;  
}  
add(3) == 3;
```

js

```
let add = (a,b=0)=>a+b;  
add(3) == 3;
```

js

对象扩展

Object.assign()

用途1：浅拷贝(shallow clone)

```
let classA = {  
  name: '二年级三班',  
  teacher: {  
    name: '张老师',  
    age: 25  
  }  
}  
  
let b = Object.assign({}, classA);  
b.teacher.age = 30;  
a.teacher.age == 30;
```

js

深拷贝(deep clone)

```
import _ from 'lodash';  
let classA = {  
  name: '二年级三班',  
  teacher: {  
    name: '张老师',  
    age: 25  
  }  
}  
let classB = _.cloneDeep(classA);  
// 此时A与B保持独立
```

js

与c语言中函数传值和传址的区别?

延伸阅读：[lodash](#)

C 语言

```
void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
void main()
{
    int a = 1, b = 2;
    swap(a, b);
    // 此时a,b不会交换
}
```

C 语言

```
void swap(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
void main()
{
    int a = 1, b = 2;
    swap(&a, &b);
    // 传入a,b的地址, 此时a,b交换
}
```

用途2：对象合并

```
let config = {  
  url: './dist/images/pictureA.jpg',  
  size: '150kb'  
}  
let imageSize = {  
  width: 300,  
  height: 400  
}  
let pictureAttr = Object.assign(imageSize, config);
```

js

用途3：函数传参的默认值

```
let getEChartOption(option){  
  let param = Object.assign({  
    type: 'line',  
    title: '纸张定量变化曲线图',  
    dataSource: 'xxx数据系统',  
  }, option);  
  // ... 一些其它的逻辑  
}
```

js



Object.keys(); Object.values()

```
{  
  type: 'line',  
  title: '纸张定量变化曲线图',  
  dataSource: 'xxx数据系统',  
}.keys() == ['type', 'title', 'dataSource'];
```

js

简便写法

```
let [name,age] = ['张三',23];  
let user = {  
  name,  
  age,  
  insert2DB(name,age){  
    // ...  
  }  
}
```

null 传导运算符

目前仅是一个提案，但在开发中经常遇到；

```
let userDept = article.authors.text.dept;  
if(userDept == '某部门'){  
    // ...  
}
```

js

以上代码在对象的后续任意结点不存在时将报错

```
let userDept = article && article.authors && article.authors.text && article.authors.  
typeof article != 'undefined' && typeof article.authors != 'undefined' // ...  
  
// null传导  
let userDept = article?.authors?.text?.dept || '';
```

js

Reflect 对象

```
if(typeof user.name!='undefined' && user.name == '张三'){  
    // ...  
}  
  
if('name' in user && user.name == '张三'){  
  
}  
  
if(Reflect.has(user,'name') && user.name == '张三'){  
    // ...  
}  
  
delete user.name;  
Reflect.deleteProperty(user,'name');
```

js

所有操作都变成了函数行为

Promise 对象

```
const promise = new Promise(function(resolve, reject) {  
  // ... some code  
  
  if (/* 异步操作成功 */) {  
    resolve(value);  
  } else {  
    reject(error);  
  }  
});  
  
promise.then(function(value) {  
  // success  
}, function(error) {  
  // failure  
});
```

js

在es7 的async出现之后，对promise的使用变得更少，但需要了解这种对象的一些特性，建议仔细阅读以下章节。

延伸阅读：[Promise](#)

JS中的异步与同步

```
let curPeople = 30;
let lastTime = '2018-01-11 03:23:33';

// 获取在当前时间之后增加的人数;
$.ajax({
  url: './yourdata.json',
  data: {
    lastTime
  },
  success: function(res) {
    curPeople += res.addedPeople;
  }
});

$('info').text(`当前活动总参与人数为${curPeople}人`);
```

js

文件I/O，数据库操作，setTimeout定时器这三类操作。

ajax可以同步获取数据

// 定义一个同步的ajax读取的函数

```
let getData = (url, data = {}) => {  
  let response = '';  
  $.ajax({  
    url,  
    data,  
    async: false,  
    success: function(res) {  
      response = res;  
    }  
  });  
  return response;  
};
```

```
let curPeople = 30;  
let lastTime = '2018-01-11 03:23:33';  
let curPeople = getData('mydata.json',{lastTime});  
$('info').text(`当前活动总参与人数为${curPeople}人`);
```

js

当有多个数据同时需要读取时？

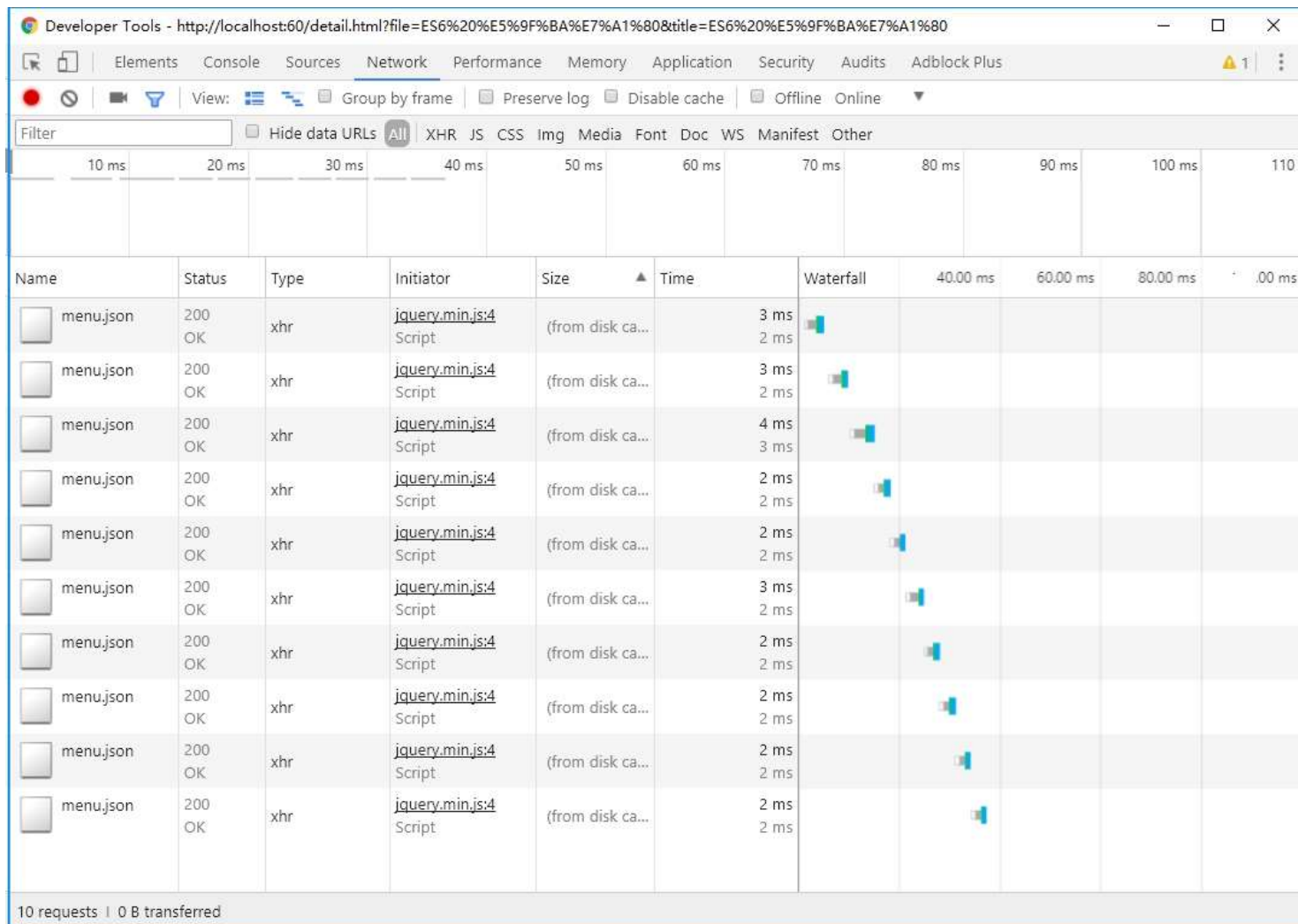
```
let getData = (url, data = {}) => {  
  let response = '';  
  $.ajax({  
    url,  
    data,  
    async: false,  
    success: function(res) {  
      response = res;  
    }  
  });  
  return response;  
};  
let arrs = [];  
for(var i=0;i<10;i++){  
  arrs.push(getData('./menu.json'));  
}  
// render table data
```

js

! 问题似乎解决了，但是看一下网络瀑布图？

网络瀑布图

在需要高并发的场景下，这样也丧失了js异步的优势



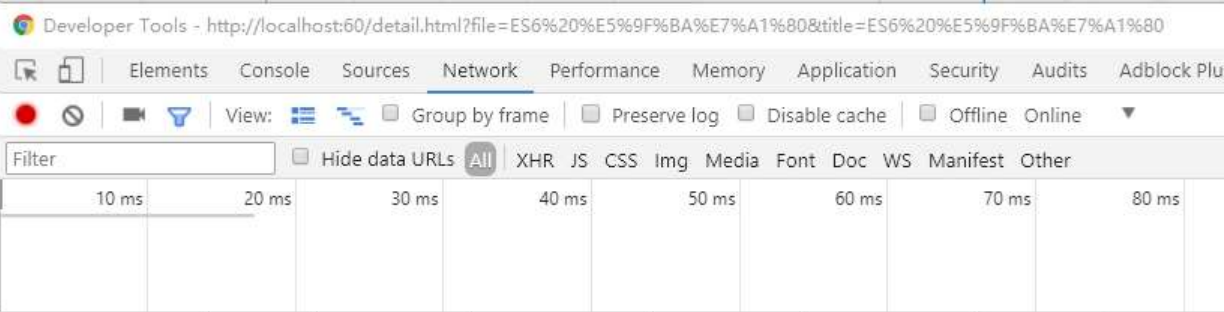
回到前面的例子

```
let arrs= [];  
for(let i=0;i<10;i++){  
  $.ajax({  
    url: './menu.json',  
    async: true,  
    success: function(data) {  
      arrs.push(data);  
      // do something else;  
    }  
  });  
}
```

js

再看一下网络瀑布图？

在多张报表查询的场景里，这种处理方式尤为必要。而服务端查询由于是同步进行，如果一张页面需要查询多条数据，系统需要等待所有数据查询完毕再一并返回。通过前后端分离+前端渲染可极大地提升用户体验。



Developer Tools - http://localhost:60/detail.html?file=ES6%20%E5%9F%BA%E7%A1%80&title=ES6%20%E5%9F%BA%E7%A1%80

Elements Console Sources Network Performance Memory Application Security Audits Adblock Plus

View: [Icons] Group by frame [] Preserve log [] Disable cache [] Offline Online ▼

Filter [] Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Name	Status	Type	Initiator	Size	Time	Waterfall	40.00 ms
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	7 ms 4 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	7 ms 5 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	8 ms 8 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	8 ms 7 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	8 ms 7 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	9 ms 7 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	9 ms 8 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	10 ms 8 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	10 ms 8 ms		
menu.json	200 OK	xhr	jquery.min.js:4 Script	(from disk ca...	11 ms 9 ms		

回调地狱(callback hell)

有这样的场景：数据C的查询参数依赖于数据B的查询结果，数据B的查询参数依赖于数据A的查询结果。(A-->B-->C).

```
$.ajax({  
  url: './menu.json',  
  success: function(dataA) {  
    $.ajax({  
      url: './menu.json',  
      data: dataA,  
      success: function(dataB) {  
        $.ajax({  
          url: './menu.json',  
          data: dataB,  
          success: function(dataC) {  
            setTimeout(function(){  
              // do something else;  
            }, 1000);  
          }  
        });  
      }  
    });  
  }  
});
```

js

ES7 async 函数

```
function foo(){};  
async function foo(){};  
const foo = async function(){};  
const foo = async ()=>{};
```

js

```
const foo =async param=>{  
  return param + 'response data';  
}  
foo('this is').then(data=>{  
  console.log(data);  
  // this is response data;  
})
```

js

延伸阅读：[async函数的含义和用法](#)、[axios](#)

链式调用

```
const foo = async param=>{  
  return 'foo1';  
}  
const bar = async (param)=>{  
  console.log(param);  
  return 'bar1';  
}  
foo('this is')  
  .then(data=>bar(data))  
  .then(param=>{  
    console.log(param);  
  })  
  // foo1  
  // bar1
```

js

此时只是当作promise在使用，解决了回调地狱的问题，但代码中有大量的then.

async/await

```
const foo = async param => {  
  return 'foo1';  
}  
  
const bar = async (param) => {  
  console.log(param);  
  return 'bar1';  
}  
  
const init = async () => {  
  let dataFoo = await foo();  
  let dataBar = await bar(dataFoo);  
  console.log(dataBar);  
}
```

js

async的实战运用

```
let read = require('../shop/jd');  
// 查询数据库店铺列表  
async function getShopList() {  
    return await query(sql.query.jd_shopList);  
}  
// 保存信息至数据库  
async function setShopDetail(shopDetail) {  
    let url = sqlParser.handleJDShops(shopDetail);  
    if (url.includes('undefined')) {  
        console.log('数据提取失败,id:' + shopDetail.shopId);  
        return;  
    }  
    await query(url);  
    if (shopDetail.shopCategories.length == 0) return;  
    await query(sqlParser.handleJDCategory(shopDetail.shopCategories));  
}  
  
// 获取京东店铺列表信息  
async function initJDShopInfo() {  
    let shopInfo;  
    let localShopList = await getShopList();  
    for (let i = 0; i < shopList.length; i++) {  
        let shopId = shopList[i].id;  
        let needSave = localShopList.filter(item => item.id == shopId);  
        if (needSave.length) continue;  
        shopInfo = await read.getShopTemplate(shopList[i]);  
        await setShopDetail(shopInfo);  
    }  
}
```

js

Q & A