**es6与模块化**

## 一、es6简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

## 二、es6 新特性

### 2.1 函数参数默认值

|  |
| --- |
| <!-- 存在0的问题？ -->  function animate(options){  this.speed = options.speed||200;  }  Es6  function animate(speed=200){  this.speed = speed;  } |

### 2.2 模板字符串

|  |
| --- |
| var name = '张三';  var age = 20;  console.log(`名哼:${name},年龄:${age}`) |

### 2.3 多行字符串

|  |
| --- |
| var str = '你好\n'+'今天吃了吗?';  console.log(str);  Es6  var str = `你好  今天吃了吗`; |

### 2.4 解构赋值

|  |
| --- |
| var user ={  name:'张三',  age:20  }  var {name,age} = user;  console.log(name);  console.log(age); |

### 2.5 对象属性简写

|  |
| --- |
| var name = '张三'  var age = 20  var user = {  name:name,  age:age  }  es6  var user = {name,age}  console.log(user) |

### 2.6 箭头函数

|  |
| --- |
| var say = function(){  return 'Hi';  }  var say =()=>'Hi';  var say = ()=>{  var word = 'Hi';  return word;  }  注意：**1.箭头函数不能改变this的指向**  function User(){  this.age = '张三'  this.say = function(){  // setTimeout(function(){  // alert('Hi '+this.age);  // },2000);  setTimeout(()=>{  alert('Hi '+this.age);  },2000)  }  }  var user = new User();  user.say();  **2.箭头函数的this在严格模式下不能指向window**  'use strict';  function User(){  this.name = '张三';  console.log(this)  }  User();  **3.箭头函数单个参数可以省略()**  var foo = a=>{  alert(a)  } |

### 2.7 Promise

|  |
| --- |
| var promise = new Promise(function(resolve,reject){  setTimeout(function(){  resolve();  }, 5000);  })  promise.then(function(){  alert('看够了没有')  }) |

### 2.8 Let与Const

### 2.9 类

|  |
| --- |
| class Animate{  constructor(type,name){  this.type = type;  this.name = name;  }  say(){  alert('Hi i am '+this.name)  }  }  class Dog extends Animate{  constructor(color,type,name){  super(type,name);  this.color = color;  }  run(){  alert('Hi i am running');  }  }  var dog = new Dog('red','Dog','欧弟');  dog.run(); |

### 2.10 模块化

* JavaScript一直没有官方的模块化解决方案，开发者在实践中主要采用CommonJS和AMD规范。而ES6制定了模块(Module)功能。
* ES6中使用export与import关键词实现模块化。

## 三、模块化

### 3.1 什么是模块化开发

* 将软件产品看作为一系列功能模块的组合
* 通过特定的方式实现软件所需模块的划分、管理、加载

### 3.2 为什么要使用模块化

* 全局变量冲突
* 函数命名冲突
* 依赖管理关系

### 3.3 解决问题

* 自执行函数

|  |
| --- |
| var modA = function(){  var say;  return{  say:function(){  }  }  }  modA.say(); |

* 命名空间

|  |
| --- |
| bufan.common.add=xx;  bufan.util.add=xxx; |

* 匿名自执行函数

|  |
| --- |
| (function(window){  window.jQuery = window.$ = jQuery;  })(window) |

## 四、模拟实现模块化

1. 原生方式
2. 面向对象

|  |
| --- |
| (function(window){  function MathExt(){  }  MathExt.prototype={  add:function(a,b){  return parseInt(a)+parseInt(b);  }  }  window.m = new MathExt();  })(window); |

1. 划分私有空间

|  |
| --- |
| var m = (function(){  function add(a,b){  return parseInt(a)+parseInt(b);  }  return{  add:add  }  })(); |

1. 模块扩展与维护

|  |
| --- |
| (function(m){  m.add = function(a,b){  return parseInt(a)+parseInt(b);  }  window.m = m;  })(window.m||{}); |
| (function(m){  m.jian = function(a,b){  return parseInt(a)-parseInt(b);  }  window.m = m;  })(window.m||{}); |

1. 第三方依赖的管理

|  |
| --- |
| (function(m,$){  m.add = function(a,b){  return parseInt(a)+parseInt(b);  }  window.m = m;  console.log($);  })(window.m||{},jQuery); |

## 五、模块化的规范

### 5.1 CommonJS规范

重要: node.js为前端工程化开发提供了支撑.

CommonJS是在浏览器环境之外构建JavaScript生态系统为目标产生的项目，比如服务器和桌面环境中。CommonJS规范是为了解决JavaScript的作用域问题而定义的模块形式，可以使每个模块在它自身的命名空间中执行。该规范的主要内容是：模块必须通过 module.exports导出对外的变量或接口，通过require()来导入其他模块的输出到当前模块。

浏览器不兼容CommonJS的根本原因，在于缺少四个Node.js环境的变量。

Module、exports、require、global

只要能够提供这四个变量，浏览器就能加载 CommonJS 模块。

举个栗子：

|  |
| --- |
| var module = {  exports: {}  };  (function(module, exports) {  exports.multiply = function (n) { return n \* 1000 };  }(module, module.exports))  var f = module.exports.multiply;  f(5) // 5000 |

### 5.2 AMD规范

    Asynchronous Module Definition，就是 异步模块定义，所有的模块将被异步加载，模块加载不影响后面语句运行。所有依赖某些模块的语句均放置在回调函数中。

    AMD规范定义了一个自由变量或者说是全局变量 define 的函数。

define( id?, dependencies?, factory );  

第一个参数 id 为字符串类型，表示了模块标识，为可选参数。若不存在则模块标识应该默认定义为在加载器中被请求脚本的标识。如果存在，那么模块标识必须为顶层的或者一个绝对的标识。

第二个参数，dependencies ，是一个当前模块依赖的，已被模块定义的模块标识的数组字面量。

第三个参数，factory，是一个需要进行实例化的函数或者一个对象。

**AMD的实现RequireJS**

官网 http://www.requirejs.org/

API <http://www.requirejs.org/docs/api.html>

RequireJS 是一个前端的模块化管理的工具库，遵循AMD规范，它的作者就是AMD规范的创始人 James Burke。所以说RequireJS是对AMD规范的阐述一点也不为过。

RequireJS 的基本思想为：通过一个函数来将所有所需要的或者说所依赖的模块实现装载进来，然后返回一个新的函数（模块），我们所有的关于新模块的业务代码都在这个函数内部操作，其内部也可无限制的使用已经加载进来的以来的模块。

### 5.3 CMD规范

CMD是国内玉伯大神在开发SeaJS的时候提出来的，

<https://github.com/seajs/seajs/issues/242>

#### 5.3.1 Sea.js使用步骤

1. 在页面中引入sea.js文件

2. 定义一个主模块文件，比如：main.js

3. 在主模块文件中通过define的方式定义一个模块，并导出公共成员

4. 在页面的行内脚本中通过seajs.use('path',fn)的方式使用模块

5. 回调函数的参数传过来的就是模块中导出的成员对象

|  |
| --- |
| define(function(require,exports,module){  function add(a,b){  return parseInt(a)+parseInt(b);  }  exports.add = add;  }) |
| seajs.use('./js/m.js',function(m){  ... |

#### 5.3.2 引入模块

Require

|  |
| --- |
| define(function(require,exports,module){  var convertor = require('./parse.js');  function add(a,b){  return convertor.parse(a)+convertor.parse(b);  }  exports.add = add;  }) |
| define(function(require,exports,module){  exports.parse = function(input){  return parseInt(input);  }  }) |

#### 5.3.3 导出对象

|  |
| --- |
| Person.js  define(function(require,exports,module){  function Person(name){  this.name = name;  this.say = function(){  alert('你好，'+this.name);  }  }  module.exports = Person;  }) |
| M.js  define(function(require,exports,module){  var convertor = require('./parse.js');  var Person = require('./Person.js');  function add(a,b){  return convertor.parse(a)+convertor.parse(b);  }  var zhang = new Person('张');  exports.add = add;  exports.zhang = zhang;  }) |
| Seajs.use('./js/m.js',function(m){  btn.onclick = function(){  var zhang = m.zhang;  zhang.say();  ... |

## 六、es6模块化

ES6中使用export与import关键词实现模块化。

|  |
| --- |
| common.js  export function add(x,y){  return x+y;  } |
| main.js  import add from './common.js';  console.log(add(5,3)); |

6.1 babel

Babel 是一个 JavaScript 编译器。

中文网址： <https://babeljs.cn/>

* 安装步骤：

$ npm install --global babel-cli

$ npm init

$ npm install --save-dev babel-preset-es2015

在根目录创建 .babelrc （babel的配置文件） ，内容为：

{ "presets": [ "es2015" ] }

* 执行编译：

# 转码结果输出到标准输出

$ babel example.js

# 转码结果写入一个文件# --out-file 或 -o 参数指定输出文件

$ babel example.js --out-file compiled.js

# 或者

$ babel example.js -o compiled.js

# 整个目录转码# --out-dir 或 -d 参数指定输出目录

$ babel src --out-dir lib

# 或者

$ babel src -d lib

# -s 参数生成source map文件

$ babel src -d lib -s

* 执行结果：

$ babel-node xx.js

附件:语法

|  |
| --- |
| export { name1, name2, …, nameN };  export { variable1 as name1, variable2 as name2, …, nameN };  export let name1, name2, …, nameN; // also var, const  export let name1 = …, name2 = …, …, nameN; // also var, const  export function FunctionName(){...}  export class ClassName {...}  export default expression;  export default function (…) { … } // also class, function\*  export default function name1(…) { … } // also class, function\*  export { name1 as default, … };  export \* from …;  export { name1, name2, …, nameN } from …;  export { import1 as name1, import2 as name2, …, nameN } from …;  export { default } from …; |

|  |
| --- |
| import defaultExport from "module-name";  import \* as name from "module-name";  import { export } from "module-name";  import { export as alias } from "module-name";  import { export1 , export2 } from "module-name";  import { export1 , export2 as alias2 , [...] } from "module-name";  import defaultExport, { export [ , [...] ] } from "module-name";  import defaultExport, \* as name from "module-name";  import "module-name"; |