**20180403**

* JavaScript简介

JavaScript是世界上最流行的脚本语言，电脑、手机、平板上浏览的所有网页和无数基于HTML5的手机App，交互逻辑都是由JavaScript驱动的。

JavaScript是一种运行在浏览器中的解释型的编程语言。

在Web世界里，只有JavaScript能跨平台、跨浏览器驱动网页，与用户交互。

随着HTML5在PC和移动端越来越流行，JavaScript变得更重要了。新兴的Node.js把JavaScript引入到了服务器端，JavaScript已经变成全能型选手。

-------------------------------------------------------------------------------------------------------

1995年，网景公司凭借其Navigator浏览器成为Web时代开启时最著名的第一代互联网公司。网景公司希望能在静态HTML页面添加一些动态效果，于是叫Brendan Eich这哥们用了10天设计出了JavaScript语言。

起名为JavaScript的原因是当时Java语言非常火，网景公司希望借其名气来推广；但事实上JavaScript除了语法有点像Java，基本上没啥关系。

* ECMAScript

为了让JavaScript成为全球标准，几个公司联合ECMA组织定制了JavaScript语言的标准，被称为ECMAScript标准。

ECMAScript是语言标准；JavaScript是网景公司对ECMAScript标准的实现。

ECMAScript在不断发展，2015年发布ECMAScript 6标准(简称ES6)，目前已经推出ES2017(ES8)；

而JavaScript版本实际上就是它实现了ECMAScript标准的哪个版本。

* 数据类型

① Number

不区分整数和浮点数，统一用Number表示，以下都是合法的Number类型：

|  |
| --- |
| 123; // 整数123 0.456; // 浮点数0.456 1.2345e3; // 科学计数法表示1.2345x1000，等同于1234.5 -99; // 负数 NaN; // NaN表示Not a Number, 无法计算结果时用NaN表示 Infinity; // 无穷大, 当数值超过了Number所能表示的最大值时, 就表示为Infinity  console.log(5 / 2); // 2.5,因为没有整数浮点数概念, 不取整 console.log(2 / 0); // Infinity console.log(0 / 0); // NaN |

② 字符串

1) 多行字符串

由于多行字符串用'\n'写起来费事，最新的ES6标准新增了多行字符串表示方法，用反引号` ... `表示：

|  |
| --- |
| console.log(`无可奈何花落去， 似曾相识燕归来， 小园香径独徘徊。`); |

2) 模板字符串

多个字符串连接可以用+：

|  |
| --- |
| **var** name = 'hikari'; **var** age = 25; **var** message = '你好, 我的名字是' + name + ', 今年' + age + '岁了!'; console.log(message) |

如果变量很多，用+连接麻烦。ES6新增了模板字符串，也是用反引号` ... `，但会自动替换字符串中的变量：

|  |
| --- |
| **var** name = 'hikari'; **var** age = 25; **var** message = `你好, 我的名字是${name}, 今年${age}岁了!`; console.log(message) |

结果：

-------------------------------------------------------------------------------------------------------

**注意**：字符串是不可变的

|  |
| --- |
| **var** name = 'hikari'; name[1]='a'; console.log(name); // hikari |

对字符串某个索引赋值，不会报错，字符串也不会变。

字符串索引和数组一样，越界不报错，但是返回undefined

主要方法：toUpperCase()、toLowerCase()、indexOf()、substring(start[,end])

③ 布尔类型

**注意**1：==和===的区别

==会自动转换数据类型再比较，可能会得到诡异的结果；

===如果数据类型不一致，返回false；类型一致再比较。

由于JavaScript这个设计缺陷，不要使用==比较，始终使用===比较。

-------------------------------------------------------------------------------------------------------

**注意**2：NaN与任何值都不相等，包括自己：

|  |
| --- |
| console.log(NaN === NaN); // false |

唯一能判断NaN的方法是通过isNaN()函数：

|  |
| --- |
| console.log(isNaN(NaN)); // true |

isNaN()的坑：

|  |
| --- |
| console.log(isNaN(' ')); // false, 认为是0,不是非数字,返回false console.log(isNaN('123abc')); // true, 字符串,是非数字,返回true |

更骚是是parseInt()：

|  |
| --- |
| console.log(parseInt(' ')); // NaN, 认为是非数字 console.log(parseInt('123abc')); // 123,将数字开头的字符串转为数字 |

-------------------------------------------------------------------------------------------------------

**注意**3：浮点数的相等比较

|  |
| --- |
| console.log(1 - 1 / 3 === 2 / 3); // false |

计算机无法精确表达无限循环的浮点数，一般比较两个浮点数是否相等，计算两者之差是否小于某个阈值，比如10-8：

|  |
| --- |
| console.log(Math.abs(1 - 1 / 3 - 2 / 3) < 1e-8); // true |

④ null和undefined

null表示一个空值，类似于Java的null， Python的None。

undefined和null类似，表示未定义。

JavaScript设计者希望用null表示一个空值，而undefined表示值未定义。事实证明，这并没有什么卵用，区分两者的意义不大。多数情况都应该用null。

undefined仅在判断函数参数是否传递的情况下有用。

⑤ 数组

JavaScript的数组可以包括任意数据类型。

|  |
| --- |
| **var** arr1 = [1, 2, 3.14, 'hikari', **null**, **true**]; **var** arr2 = **new** Array(1, 2, 3.14); // pycharm提示此种定义可以被简化... |

考虑到代码的可读性，强烈建议用[]直接定义数组。

访问数组不会越界，而是直接给出undefined

|  |
| --- |
| console.log(arr1[10]); // undefined |

-------------------------------------------------------------------------------------------------------

**注意**：直接给数组length赋新值会导致数组大小变化

|  |
| --- |
| **var** arr = [1, 2, 3]; console.log(arr.length); // 3 arr.length = 1; console.log(arr); // [1] |

如果设为比原来大，用undefined补充；如果越界赋值也会改变数组大小。

|  |
| --- |
| arr[10]='a'; console.log(arr.length); // 11 |

主要方法：indexOf()、slice(start[,end])：切片 、push()：尾插、pop()：尾删、 unshift()：头插、shift()：头删、sort()、reverse()、splice()、concat()、join()

-------------------------------------------------------------------------------------------------------

练习：在新生欢迎会上，根据新同学的名单，排序后显示：欢迎XXX，XXX，XXX和XXX同学！：

|  |
| --- |
| **var** arr = ['小明', '小红', '大军', '阿黄']; arr.sort(); **var** last=arr.pop(); **var** msg='欢迎'+arr.join('，')+'和'+last+'同学！'; console.log(msg); |



这顺序略奇怪…难道根据unicode排序的?

查询得：小明\u5c0f\u660e；小红\u5c0f\u7ea2；大军\u5927\u519b；阿黄\u963f\u9ec4

果然没错…

⑥ 对象

JavaScript的对象是一组由键值对组成的无序集合：

|  |
| --- |
| **var** hikari = {  name: 'hikari',  age: 25,  tags: ['js', 'web', 'python'],  city: 'Nanjing',  hasCar: **false**,  zipcode: **null** }; console.log(hikari['name']); // hikari console.log(hikari.tags); // ["js", "web", "python"] console.log(hikari.hobby); // undefined, 未定义的属性 |

键都是字符串类型，值可以是任意数据类型。获取一个对象的属性，可以使用obj.attr或obj['attr']的方式。

-------------------------------------------------------------------------------------------------------

由于JavaScript是动态语言，可以自由地给一个对象添加或删除属性：

|  |
| --- |
| console.log(hikari.hasCar); // false **delete** hikari.hasCar; console.log(hikari.hasCar); // undefined |

-------------------------------------------------------------------------------------------------------

检查对象是不是有某个属性可以用in：

|  |
| --- |
| console.log('city' **in** hikari); // true console.log('hobby' **in** hikari); // false console.log('toString' **in** hikari); // true  console.log(hikari.toString); // function toString() { [native code] } |

**注意**：in检测存在的属性不一定是对象自己的，也可能是继承来的，如toString，toString定义在object对象，所有对象最终都会在原型链上指向object，所以所有对象都拥有toString属性。

检测某个属性是不是对象自身拥有的使用hasOwnProperty()方法：

|  |
| --- |
| console.log(hikari.hasOwnProperty('zipcode')); // true console.log(hikari.hasOwnProperty('toString')); // false |

* strict模式

JavaScript在设计之初，为了方便初学者学习，并不强制要求用var申明变量。这个设计错误带来了严重的后果：如果一个变量没有通过var申明就被使用，那么该变量就自动被申明为全局变量：

|  |
| --- |
| **var** a = 1; **var** b = 2; **function** show() {  **var** a = 23;  b = 43;  c = 99;  console.log('a=' + a + ',b=' + b + ',c=' + c); // a=23,b=43,c=99 } show(); console.log('a=' + a + ',b=' + b + ',c=' + c); // a=1,b=43,c=99 |

函数show内部a是局部变量，b、c都是全局变量，修改了b的值并隐式定义全局变量c。在同一页面的不同的JavaScript文件中，如果都不用var申明，同名变量相互影响，会产生难以调试的错误结果。

为了弥补这一严重设计缺陷，ECMA在后续规范中推出了strict模式。在strict模式下未使用var申明变量就使用，将导致运行错误。

JavaScript代码第一行写上：'use strict';启用strict模式

不支持strict模式的浏览器会当做一个字符串语句执行；支持strict模式的浏览器将开启strict模式运行JavaScript。

然后console就会提示：Uncaught ReferenceError: c is not defined

* 条件判断

简单部分做下练习算了…

练习：小明身高1.75m，体重80.5kg。请根据BMI公式(体重除以身高的平方)帮小明计算他的BMI指数，并根据BMI指数：低于18.5：过轻；18.5-25：正常；25-28：过重；28-32：肥胖；高于32：严重肥胖；用if...else...判断并显示结果：

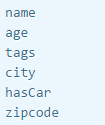
|  |
| --- |
| 'use strict'; // 弹出对话框输入 **var** height = parseFloat(prompt('请输入身高(m):')); **var** weight = parseFloat(prompt('请输入体重(kg):')); **var** bmi = weight / (height \* height); **if** (bmi < 18.5) {  console.log('过轻'); } **else if** ((bmi < 25)) {  console.log('正常'); } **else if** (bmi < 28) {  console.log('过重'); } **else if** (bmi < 32) {  console.log('肥胖'); } **else** {  console.log('严重肥胖'); } |

结果：过重 // 减肥吧少年!

* 循环

for ... in循环可以把一个对象的所有属性依次循环出来：

|  |
| --- |
| **for** (**var** i **in** hikari) {  console.log(i); } |

结果：

Array也是对象，每个元素的索引被视为对象的属性，for ... in循环可以直接循环出Array的索引：

|  |
| --- |
| **var** a = ['A', 'B', 'C']; **for** (**var** i **in** a) {  console.log(i); // '0', '1', '2', 索引是字符串类型  console.log(a[i]); // 'A', 'B', 'C' } |

**注意**：for ... in对Array的循环得到的是String而不是Number。

字符串的for…in与数组类似。

* Map和Set

JavaScript的默认对象表示方式{}相当于其他语言中的Map或Dictionary的数据结构，即一组键值对。

但是JavaScript的对象的键必须是字符串。但实际上Number或者其他数据类型作为键也非常合理。

为解决此问题，ES6引入了新的数据类型Map。

① Map：一组键值对，查找速度极快

举个栗子，根据同学名字查找对应成绩，如果用数组实现，需要两个数组：

|  |
| --- |
| **var** names = ['maki', 'rin', 'nozomi']; **var** scores = [99, 55, 78]; console.log('rin score: '+scores[names.indexOf('rin')]); |

查找给定名字对应成绩需要在names中找到对应的位置，再从scores取出对应的成绩；数组越长，耗时越长。

-------------------------------------------------------------------------------------------------------

如果用Map实现，只需要名字-成绩对照表，直接根据名字查找成绩，无论表有多大，查找速度都不会变慢。

|  |
| --- |
| **var** scores = **new** Map([['maki', 99], ['rin', 55], ['nozomi', 78]]); console.log('rin score: ' + scores.get('rin')); |

初始化Map需要一个二维数组或直接初始化为一个空Map。

Map对象方法：

① set(key,value)：设置键值对，不存在键为添加，存在键覆盖旧值

② has(key)：是否存在某个键

③ get(key)：根据键获取值，键不存在则为undefined

④ delete(key)：删除键值对，键不存在什么也不做

② Set：只存储不重复key的集合，不存储value

初始化Set需要提供数组或直接创建一个空Set：

|  |
| --- |
| **var** s1 = **new** Set(); **var** s2 = **new** Set([1, 2, '3', 3, '3', 3]); console.log(s1); // Set {} console.log(s2); // Set {1, 2, "3", 3}, 重复元素自动被过滤 |

Set对象方法主要是add(key)和delete(key)

|  |
| --- |
| s1.add(1); console.log(s1); // Set {1} s1.add(1); console.log(s1); // Set {1}, 可以重复添加但没效果 s2.delete(1); console.log(s2); // Set {2, "3", 3} |

JavaScript的Map和Set类似于Python的dict和set

* iterable

遍历Array可以采用下标循环，但遍历Map和Set无法使用下标。

为了统一集合类型，ES6标准引入了新的iterable类型，Array、Map和Set都属于iterable类型。iterable类型可以通过新的for ... of循环来遍历。

|  |
| --- |
| **var** scores = **new** Map([['maki', 99], ['rin', 55], ['nozomi', 78]]); **for** (**var** i **of** scores){ // 遍历Map  console.log(i[0]+': score = '+i[1]); } |

-------------------------------------------------------------------------------------------------------

for ... in循环由于历史遗留问题，遍历的实际上是对象的属性名称。一个Array实际也是一个对象，它的每个元素的索引被视为一个属性。

给Array对象添加额外属性：

|  |
| --- |
| **var** arr = ['A', 'B', 'C']; arr.name = 'arr'; **for** (**var** i **in** arr) {  console.log(i); // '0', '1', '2', 'name' }  console.log(arr.length); // 3 |

for ... in循环将把name包括在内，但length没变，表示不包括在内。

for ... of循环修复了这些问题，它只循环集合本身的元素：

|  |
| --- |
| **for** (**var** i **of** arr) {  console.log(i); // 'A', 'B', 'C' } |

-------------------------------------------------------------------------------------------------------

然而更好的方式是直接使用iterable内置的forEach方法 (ES5.1标准引入)，它接收一个函数，每次迭代就自动回调该函数。

遍历Array：

|  |
| --- |
| arr.forEach(**function** (element, index, array) {  // element: 指向当前元素的值; index: 指向当前索引; array: 指向Array对象本身  console.log(element + ', index = ' + index); }); |

结果：

遍历Map：

|  |
| --- |
| **var** scores = **new** Map([['maki', 99], ['rin', 55], ['nozomi', 78]]); scores.forEach(**function** (value, key) {  console.log(key + ': score = ' + value); }); |

如果对某些参数不感兴趣，可以忽略。例如只获取Array的element

|  |
| --- |
| arr.forEach(**function** (value) {  console.log(value); }) |

* 函数

JavaScript函数允许传入任意个值而不影响调用，因此传入的值比需要的多也没有问题，虽然函数并不需要这些值；

传入值个数比定义的少也没有问题，没传入值的变量收到undefined

|  |
| --- |
| **var** add = **function** add(a, b) {  **return** a + b; }; console.log(add(1, 2)); // 3 console.log(add(4, '5', 6, '7')); // 45 console.log(add(8)); // NaN console.log(add()); // NaN |

要避免收到undefined，可以对参数进行检查：

|  |
| --- |
| **var** add = **function** add(a, b) {  **if** (typeof(b) != 'number' || typeof(a) != 'number') {  **throw** 'Not a Number!';  }  **return** a + b; }; |

JavaScript还有一个免费赠送的关键字arguments，只在函数内部起作用，永远指向当前函数传入的所有值。arguments类似Array但它不是一个Array：

|  |
| --- |
| **function** foo(x) {  console.log('x = ' + x); // x = 1  **for** (**var** i **of** arguments) {  console.log('arg[' + i + '] = ' + i); // arg[1] = 1; arg[2] = 2; arg[3] = 3  } } foo(1, 2, 3); |

不能forEach但可以for-of，说明arguments不是iterable?

实际上arguments最常用于判断传入值的个数：

|  |
| --- |
| // show函数name和age都是可选参数, 没收到传值都用默认值, 传入1个值age用默认值 **function** show(name, age) {  **if** (arguments.length < 2) {  age = 25;  }  **if** (arguments.length < 1) {  name = 'hikari';  }  console.log('我的名字叫' + name + '，今年' + age + '岁了！'); } show(); // 我的名字叫hikari，今年25岁了！ show('haha'); // 我的名字叫haha，今年25岁了！ show('maki', 15); // 我的名字叫maki，今年15岁了！ |

rest参数

ES6标准引入了rest参数，只能写在参数最后，前面用...标识，类似于Python的可变参数，传入的值先给前面定义的参数，多余的值以数组形式交给rest，所以不需要遍历arguments再去除前面定义的参数就可以获取了多余参数的值。

如果传入的值连定义的参数都没填满，rest会接收一个空数组。

|  |
| --- |
| **function** foo(a, b, ...rest){  console.log('a = ' + a);  console.log('b = ' + b);  console.log(rest); } foo(1, 2, 3, 4, 5); // a = 1; b = 2; [3, 4, 5] foo(1); // a = 1; b = undefined; [] |

练习：定义一个计算圆面积的函数area\_of\_circle(r, pi)，r表示圆的半径；pi: 表示π的值，如果不传值默认3.14

|  |
| --- |
| **function** area\_of\_circle(r, pi) {  **if** (typeof(r) != 'number') { // 最好还是要进行类型检测, 但是自娱自乐又觉得没必要  **throw** 'Not a Number!';  }  **if** (arguments.length < 2) {  pi = 3.14;  }  **return** pi \* r \* r; } // 测试 **if** (area\_of\_circle(2) === 12.56 && area\_of\_circle(2, 3.1416) === 12.5664) {  console.log('测试通过'); } **else** {  console.log('测试失败'); } |

* 变量提升

JavaScript的函数定义有个特点，会先扫描整个函数体的语句，把所有申明的变量提升到函数顶部：

|  |
| --- |
| 'use strict'; **function** foo() {  **var** x = 'hello, ' + y;  console.log(x); // hello, undefined  **var** y = 'world'; } foo(); |

虽然是strict模式，但语句var x = 'hello, ' + y;并没有报错，原因是变量y在后面声明了；但控制台显示hello, undefined，说明变量y的值为undefined。

因为JavaScript引擎自动提升了变量y的声明，但不会提升变量y的赋值。

上述foo()函数，JavaScript引擎看到的代码相当于：

|  |
| --- |
| **function** foo() {  **var** y; // 提升变量y的声明, 此时y为undefined  **var** x = 'hello, ' + y;  console.log(x); // hello, undefined  y = 'world'; } |

由于JavaScript这一怪异的特性，在函数内部定义变量时要首先声明所有变量。

最常见的做法是用一个var申明函数内部用到的所有变量：

|  |
| --- |
| **function** foo() {  **var** x = 1,  y = x + 2,  i, j;  // 其它语句… } |

* 全局作用域

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象window，全局变量实际上被绑定到window的一个属性：

|  |
| --- |
| **var** name = 'hikari'; **var** show = **function** () {  console.log(name);  console.log(window.name); // 两者等价 }; show(); |

直接访问全局变量name和访问window.name是完全一样的。

由于函数定义有两种方式，以变量方式var foo = function () {}定义的函数实际上也是一个全局变量。所以上面show函数也是全局变量，绑定到window对象：

|  |
| --- |
| show(); // 直接调用 window.show(); // 通过window调用 |

其实，alert()函数也是window的一个变量：

|  |
| --- |
| window.alert('调用window.alert()'); **var** old\_alert = window.alert; // 把alert保存到另一个变量 window.alert = **function** () { }; // 给alert赋一个新函数 alert('无法用alert()显示了!'); // 此句不弹窗 window.alert = old\_alert; // 恢复alert alert('又可以用alert()了!'); |

说明JavaScript实际上只有一个全局作用域。任何变量如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则抛出ReferenceError错误。

* 名字空间 (namespace)

全局变量会绑定到window上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难发现。

减少冲突的一个方法是把自己所有变量和函数全部绑定到一个全局变量中。

例如：

|  |
| --- |
| // 唯一的全局变量MYAPP **var** MYAPP = {}; // 其他变量 MYAPP.name = 'hikari app'; MYAPP.version = 1.0; // 其他函数 MYAPP.show = **function** () {  **return** 'hikari app'; }; |

把自己的代码全部放入唯一的名字空间中，大大减少全局变量冲突的可能。

许多著名的JavaScript库如jQuery、YUI、underscore等都是如此做的。

* 局部作用域

由于JavaScript的变量作用域是函数内部，在for循环等语句块中是无法定义具有局部作用域的变量的：

|  |
| --- |
| **function** show() {  **var** arr = [1, 2, 3];  **for** (**var** i **of** arr) {  console.log(i); // 1, 2, 3  }  **var** j = i \* i;  console.log(j); // 9, 仍然可以使用变量i } show(); |

为了解决块级作用域，ES6引入了新关键字let，用let替代var可以申明一个块级作用域的变量：

|  |
| --- |
| **function** show() {  **var** arr = [1, 2, 3];  **for** (**let** i **of** arr) {  console.log(i); // 1, 2, 3  }  **var** j = i \* i; // Uncaught ReferenceError: i is not defined } |

// 也就是说var已经过时了? let碾压var?

* 常量

由于var和let声明的是变量，如果要声明常量，在ES6之前是不行的。

通常用全部大写的变量来表示此变量的常量，不要修改它的值。

ES6标准引入了新关键字const定义常量，const与let都具有块级作用域：

|  |
| --- |
| **const** PI = 3.14; PI = 3; // Uncaught TypeError: Assignment to constant variable. |

* 解构赋值

从ES6开始，JavaScript引入了解构赋值，可以同时对一组变量进行赋值。

相当于Python元组和列表的拆包：a, b, c = (1, 2, 3)

① 数组解构赋值

|  |
| --- |
| **var** [a, b, c] = [1, 2, 3]; console.log('a = ' + a + ', b = ' + b + ', c = ' + c); // a = 1, b = 2, c = 3 |

如果数组本身还有嵌套，嵌套层次和位置要保持一致：

|  |
| --- |
| [a, [b, c]] = ['hehe', [22, 'c']]; console.log('a = ' + a + ', b = ' + b + ', c = ' + c); // a = 'hehe', b = 22, c = 'c' |

解构赋值还可以忽略某些元素：

|  |
| --- |
| **let** [, age,] = ['hikari', 25, 'python']; // 只对age赋值第2个元素 console.log(age); // 25 |

② 对象解构赋值

1) 解构赋值也可以从一个对象中快速获取指定属性：

|  |
| --- |
| **let** hikari = {  name: 'hikari',  age: 25,  gender: 'male',  school: '皇家幼稚园',  job: '搬砖' }; **let** {name, age, job} = hikari; // name, age, job分别被赋值为对应属性 console.log('name=' + name + ', age=' + age + ', job=' + job); {  **let** name = 'maki';  console.log('{}里面的name=' + name); // {}里面的name=maki } console.log('外面的name=' + name); // 外面的name=hikari |

pycharm将js版本改为ES6，使用var声明变量提示应该用let或const代替…

局部代码块{}中的name对外面的name没有影响。

**20180404**

2) 也可以对嵌套的对象属性进行解构赋值，要保证对应的层次一致：

|  |
| --- |
| **let** hikari = {  name: 'hikari',  age: 25,  gender: 'male',  school: '皇家幼稚园',  job: '搬砖',  address: {  province: 'Jiangsu',  city: 'Nanjing',  zipcode: '215000',  } }; **let** {name, address: {city, zipcode}} = hikari; console.log(`name: ${name}, city: ${city}, zipcode: ${zipcode}`); // name: hikari, city: Nanjing, zipcode: 215000 // 注意address不是变量, 而是为了让city和zipcode获得嵌套address对象的属性 console.log(address); // ReferenceError: address is not defined |

3) 变量名可以与属性名不一致，比如把job属性赋值给work变量：

|  |
| --- |
| **let** {job: work} = hikari; console.log('work is '+work); // work is 搬砖 console.log('job is '+job); // ReferenceError: job is not defined |

同理，job也不是变量，而是为了让变量work获得job属性

4) 解构赋值可以使用默认值，避免不存在的属性返回undefined的问题：

|  |
| --- |
| // 如果对象没有single属性默认赋值为true **let** {single = **true**} = hikari; console.log('是否单身：' + single); // 是否单身：true |

5) 先声明变量再解构赋值可能会报错：

|  |
| --- |
| **let** h,w; {h, w} = {name: 'hikari', h: 175, w: 80}; console.log('h=' + h + ',w=' + w); // SyntaxError: Unexpected token = |

因为JavaScript引擎把{}语句当作了块处理，于是=不再合法。

pycharm直接提示=错误，代码格式化后变成：

|  |
| --- |
| **let** h, w; {  h, w // 此处被当成语句块,什么都没干, } = {name: 'hikari', h: 175, w: 80}; // 此句应该是表达式, 但是=开始不合法 |

解决方法是将整个解构赋值语句用小括号括起来：

|  |
| --- |
| **let** h, w; ({h, w} = {name: 'hikari', h: 175, w: 80}); console.log('h=' + h + ',w=' + w); // h=175,w=80 |

* 解构赋值的应用

使用解构赋值可以减少代码量，但需要支持ES6解构赋值特性的浏览器。

① 交换两个变量值简化，不需要临时变量

|  |
| --- |
| **let** a = 23, b = 45; [a, b] = [b, a]; console.log(`a=${a}, b=${b}`); // a=45, b=23 |

② 获取当前页面的域名和路径：

|  |
| --- |
| **let** {hostname: domain, pathname: path} = location; console.log(`domain: ${domain}, path: ${path}`);  // domain: , path: /C:/Users/hikari星/Desktop/1.html |

③ 如果一个函数接收一个对象作为参数，那么可以使用解构直接把对象的属性绑定到变量中。

|  |
| --- |
| **function** build\_date({year, month, day, hour = 0, minute = 0, second = 0}) {  **return new** Date(year + '-' + month + '-' + day + ' ' + hour + ':' + minute + ':' + second); } console.log(build\_date({year: 2018, month: 4, day: 4})); // Wed Apr 04 2018 00:00:00 GMT+0800 (中国标准时间) console.log(build\_date({year: 2018, month: 4, day: 4, hour: 13, minute: 30, second: 45})); // Wed Apr 04 2018 13:30:45 GMT+0800 (中国标准时间) |

传入的对象只需要year、month和day三个属性；hour、minute和second属性可传可不传，不传用默认值。

* 方法

对象绑定一个函数称为对象的方法

|  |
| --- |
| **let** hikari = {  name: 'hikari',  birth: 1992,  age: **function** () {  **let** y = **new** Date().getFullYear(); // 获取当前年份  **return** y - **this**.birth;  } }; console.log(hikari.age); // function () { let y = new Date().getFullYear(); return y - this.birth; } console.log(hikari.age()); // 26 |

方法与普通函数唯一区别是有一个this关键字。

如果把方法写到对象外面：

|  |
| --- |
| **function** get\_age() {  console.log(**this**);  **let** y = **new** Date().getFullYear();  **return** y - **this**.birth; } **let** hikari = {  name: 'hikari',  birth: 1992,  age: get\_age }; console.log(hikari.age()); // [object Object]; 26 console.log(get\_age()); // [object Window]; NaN |

以对象方法形式调用this指向调用的对象；单独调用this指向全局对象window

由于这是一个巨大的设计错误。ECMA决定，在strict模式下让函数的this指向undefined。因此在strict模式下，会得到一个错误：

Uncaught TypeError: Cannot read property 'birth' of undefined

这个决定只是让错误及时暴露出来，并没有解决this应该指向的正确位置。

* call和apply

函数本身的call或apply方法可以指定函数this指向哪个对象。

两者第1个参数都是需要绑定的this变量；

call(this, var1, var2…)：函数参数按顺序传入；

apply(this, [var1, var2, …])：函数参数打包成Array再传入。

|  |
| --- |
| **function** get\_age() {  **let** y = **new** Date().getFullYear();  **return** y - **this**.birth; } **let** hikari = {  name: 'hikari',  birth: 1992,  age: get\_age }; console.log(hikari.age()); // 26 console.log(get\_age.call(hikari)); // 26, this指向hikari, 参数为空 console.log(get\_age.apply(hikari, [])); // 26, this指向hikari, 参数为空数组 |

* 装饰器

JavaScript所有对象都是动态的，即使内置的函数，也可以重新指向新的函数。

比如想统计代码一共调用了多少次parseInt()，可以在所有的调用处手动加上cnt += 1，不过这样做太傻了。最佳方案是自定义函数替换掉默认的parseInt()：

|  |
| --- |
| **let** cnt = 0; **let** oldParseInt = parseInt; // 保存原函数 window.parseInt = **function** () {  cnt+= 1;  **return** oldParseInt.apply(**null**, arguments); // 调用原函数 }; parseInt('10'); parseInt('20'); parseInt('30'); console.log('count = ' + cnt); // count = 3 |

* 高阶函数

一个函数接收另一个函数作为参数，这种函数称为高阶函数。

① map()

map()方法定义在JavaScript的Array中，传入函数f作用于Array每个元素得到的结果组成一个新的Array：

|  |
| --- |
| **function** f(x) {  **return** x \* x + 1; } **let** arr = [1, 3, 5, 7, 9]; **let** ret = arr.map(f); console.log(ret); // [2, 10, 26, 50, 82] |

② reduce()

Array的reduce()把一个函数作用Array的[x1, x2, x3...]，这个函数必须接收两个参数，reduce()把结果继续和序列的下一个元素做累积计算，其效果是：

[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)

|  |
| --- |
| **function** f(x, y) {  **return** x + '' + y; } **let** arr = [1, 3, 5, 7, 9]; **let** ret = arr.reduce(f); console.log(ret); // '13579' |

③ filter()

Array的filter()接收一个函数，作用于每个元素，返回true的留下：

|  |
| --- |
| **let** arr = [1, 2, 3, 4, 5, 6, 7]; console.log(arr.filter(**function** (x) {  **return** x % 2 === 0; })); // [2, 4, 6] |

filter()接收的回调函数可以有多个参数element, index, self。通常仅使用第一个参数，表示Array的某个元素。另外两个参数表示素的位置和数组本身。

比如用于去除Array的重复元素：

|  |
| --- |
| **let** arr = [1, 2, 4, 6, 3, 4, 5, 3, 1, 2]; **let** ret = arr.filter(**function** (val, index, self) {  **return** index === self.indexOf(val); }); console.log(ret); // [1, 2, 4, 6, 3, 5] |

④ sort()

JavaScript的Array的sort()方法的排序结果很奇葩：

|  |
| --- |
| **let** arr = [23, 56, 32, 123, 86, 9, 47, 62]; console.log(arr.sort()) // [123, 23, 32, 47, 56, 62, 86, 9] |

默认将所有元素先转换为String再排序…坑啊

不过sort()也可以接收一个函数作为排序规则：

|  |
| --- |
| **let** arr = [23, 56, 32, 123, 86, 9, 47, 62]; arr.sort(**function** (x, y) {  **if** (x < y) {  **return** -1;  }  **if** (x > y) {  **return** 1;  }  **return** 0; }); console.log(arr) // [9, 23, 32, 47, 56, 62, 86, 123] |

如果x<y return 1; x>y return -1;则为从大到小排序。

上面直接打印arr.sort()和排序后再打印arr都可以。也就是sort()方法会直接对Array进行修改，它返回的结果仍是当前Array。

* 闭包

高阶函数，还可以把函数作为结果返回。

如果返回函数在其内部引用了外部的局部变量，整体构成闭包。

返回的函数并没有立刻执行，而是直到调用f()才执行。

|  |
| --- |
| **function** count() {  **let** arr = [];  **for** (**let** i = 1; i <= 3; i++) {  arr.push(**function** () {  **return** i \* i;  });  }  **return** arr; } **let** [f1, f2, f3] = count(); console.log([f1(), f2(), f3()]); // [1, 4, 9] |

居然没问题? 说好的[16, 16, 16]呢?

好吧，如果for循环的i用var声明，结果就是[16, 16, 16]…

返回闭包时返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果要引用循环变量需要再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何改变，已绑定到函数参数的值不变。

结果let居然没错…var可以安息了…

* 闭包用途

面向对象语言里如Java和C++，要在对象内部封装一个私有变量，可以用private修饰一个成员变量。

在没有class机制，只有函数的语言里，借助闭包同样可以封装一个私有变量。

① 用JavaScript创建一个计数器：

|  |
| --- |
| **function** create\_counter(initial) {  **let** x = initial || 0; // 不传初始计数默认0  **return** {  increase: **function** () {  **return** ++x;  }  } } **let** c1 = create\_counter(); console.log(c1.increase()); // 1 console.log(c1.increase()); // 2 console.log(c1.increase()); // 3 **let** c2 = create\_counter(56); console.log(c2.increase()); // 57 console.log(c2.increase()); // 58 console.log(c2.increase()); // 59 |

在返回的对象中，实现了一个闭包，该闭包携带了局部变量x，而且从外部代码根本无法访问变量x。换句话说，闭包就是携带状态的函数，并且它的状态可以完全对外隐藏。

② 闭包可以把多参数的函数变成单参数的函数。(偏函数?)

例如：要计算xy可以用Math.pow(x, y)函数，不过考虑到经常计算x2或x3，可以利用闭包创建新的函数pow2和pow3：

|  |
| --- |
| **function** make\_pow(n) {  **return function** (x) {  **return** Math.pow(x, n);  } } // 创建两个偏函数x2和x3 **let** pow2 = make\_pow(2); **let** pow3 = make\_pow(3); console.log(pow2(25)); // 625, 25的平方 console.log(pow3(16)); // 4096, 16的立方 |

* 脑洞大开

很久很久以前，有个叫阿隆佐·邱奇的帅哥(图灵的博导)，发现只需要用函数，就可以用计算机实现运算，而不需要0123这些数字和+-\*/这些符号。

|  |
| --- |
| // 定义数字0 **let** zero = **function** (f) {  **return function** (x) {  **return** x;  } }; // 定义数字1 **let** one = **function** (f) {  **return function** (x) {  **return** f(x);  } }; // 定义加法 **function** add(n, m) {  **return function** (f) {  **return function** (x) {  **return** m(f)(n(f)(x));  }  } } // 计算数字2 = 1 + 1 **let** two = add(one, one); // 计算数字3 = 1 + 2 **let** three = add(one, two); // 计算数字5 = 2 + 3 **let** five = add(two, three); // 你说它是3就是3，你说它是5就是5，你怎么证明？ // 呵呵，看这里 // 给3传一个函数,会打印3次: (three(**function** () {  console.log('print 3 times'); }))(); // 给5传一个函数,会打印5次: (five(**function** () {  console.log('print 5 times'); }))(); // 继续接着玩一会... |

个人理解：

上面的one、two、three都是函数啦~压根不是数字

zero(f)(x)=x，没有执行函数f(x)，也就是0；

one(f)(x)=f(x)，执行函数f(x)一次，对应1；

two(f)(x)=add(one,one)(f)(x)= one(f)(one(f)(x))=one(f)(f(x))=f(f(x))，也就是嵌套执行f(x)两次，对应2；内层f(x)的输出作为外层f(x)的输入

three(f)(x)=add(one,two)(f)(x)= two(f)(one(f)(x))=two(f)(f(x))=f(f(f(x)))，执行f(x)三次，对应3；

根据add和前面的推论，后面就可以连起来了…

five(f)(x)=add(two,three)(f)(x)=three(f)(two(f)(x))=three(f)(f(f(x)))=f(f(f(f(f(x)))))，执行5次f(x)；

此处的f(x)对应于function () {console.log();}

* 箭头函数 (arrow function)

ES6标准新增的一种新函数，它的定义用一个箭头：

|  |
| --- |
| **let** f = x => x \*\* 3; // 居然和python一样可以求幂 |

等价于：

|  |
| --- |
| **let** f = **function** (x) {  **return** x \*\* 3; }; |

箭头函数相当于匿名函数，并且简化了函数定义。

简单的语句可以只包含一个表达式，连{ ... }和return都省略掉了。

如果包含多条语句，就不能省略{ ... }和return：

|  |
| --- |
| **let** f = x => {  **if** (x > 0) {  **return** x \* x;  }  **else** {  **return** -x \* x;  } }; |

简单的if用三元表达式还是可以一行：

|  |
| --- |
| **let** f = x => x > 0 ? x \* x : -x \* x; |

如果参数不是一个，就需要用括号()括起来：

|  |
| --- |
| // 两个参数 **let** f = (x, y) => x \* x + y \* y; // 无参数 **let** pi = () => 3.14; // 可变参数 **let** sum = (x, y, ...rest) => {  **let** s = x + y;  **for** (**let** i **of** rest) {  s += i;  }  **return** s; }; console.log(f(5,6)); // 61 console.log(pi()); // 3.14 console.log(sum(4,5)); // 9 console.log(sum(6,7,8,9,10)); // 40 |

如果要返回一个对象是单表达式，直接这么写会报错；因为和函数体的{ ... }有语法冲突，所以要加括号：

|  |
| --- |
| **let** f = x => ({foo: x \*\* 2}); console.log(f(5).foo); // 25 |

实际上，箭头函数和匿名函数有个明显的区别：箭头函数内部的this是词法作用域，由上下文确定。

**20180405**

* generator

生成器是ES6标准引入的新数据类型，借鉴了Python生成器的概念和语法。

generator由function\*定义，除了return语句，还可以用yield返回多次。

生成器典型例子：斐波那契数列

|  |
| --- |
| **function** fib(n) {  **let** a = 0, b = 1;  **let** arr = [];  **for** (**let** i = 0; i < n; i++) {  arr.push(b);  [a, b] = [b, a + b];  }  **return** arr; } **let** arr = fib(10); console.log(arr); // [1,1,2,3,5,8,13,21,34,55] |

函数只能返回一次，所以必须返回一个Array。

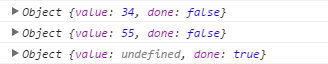
如果换成generator，就可以一次返回一个数，不断返回多次。

|  |
| --- |
| **function**\* fib(n) {  **let** a = 0, b = 1;  **for** (**let** i = 0; i < n; i++) {  **yield** b;  [a, b] = [b, a + b];  } } **let** g = fib(10); console.log(g); // fib {[[GeneratorStatus]]: "suspended", [[GeneratorReceiver]]: Window} |

fib(10)仅仅创建了一个generator对象，但没有执行。

调用生成器可以不断使用生成器的next方法：

比如调用11次console.log(g.next());



next()每次遇到yield b;就返回一个对象{value: b, done: true/false}，然后暂停。value是yield返回值，done表示生成器是否执行结束。如果done为true，表示生成器全部执行完毕，不要再继续调用next()了。如果有return，第1次done为true时，next()的结果是return的返回值。

显然这样很不友好，更好的方法是直接用for ... of循环迭代生成器，这样不需要自己判断done是否为true。

|  |
| --- |
| **for** (**let** i **of** g) {  console.log(i); // 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 } |

因为generator可以在执行过程中多次返回，就像可以记住执行状态的函数，因此写一个generator就可以实现需要用面向对象才能实现的功能。

generator的另一个好处是可以用于AJAX的异步回调。

* 标准对象

在JavaScript的世界里，一切皆对象。

但某些对象和其他对象不太一样。为了区分对象的类型，可以用typeof操作符获取对象的类型，其返回一个字符串：

|  |
| --- |
| console.log(typeof 123); // number console.log(typeof NaN); // number console.log(typeof 'abc'); // string console.log(typeof **true**); // boolean console.log(typeof undefined); // undefined console.log(typeof alert); // function console.log(typeof [1, 2, 3]); // object console.log(typeof **null**); // object console.log(typeof {name: 'hikari'}); // object console.log(typeof window); // object |

可见number、string、boolean、function和undefined有别于其他类型。

**注意**：null类型是object，Array类型也是object；用typeof无法区分null、Array和通常意义上的object{}。

* 包装对象

JavaScript还提供了包装对象，类似于Java里int和Integer的暧昧关系。

number、boolean和string都有包装对象。在JavaScript中，字符串也区分string类型和它的包装类型。包装对象用new创建：

|  |
| --- |
| **let** n = **new** Number(123); // 生成了新的包装类型 **let** b = **new** Boolean(**true**); **let** s = **new** String('hikari'); console.log(typeof n); // object console.log(typeof b); // object console.log(typeof s); // object console.log(n === 123); // false |

包装对象类型为object，所以和原始值用===比较会返回false。

所以闲的蛋疼也不要使用包装对象！尤其是针对string类型！

pycharm提醒primitive type object wrapper used，它的建议是把new去掉

此时Number()、Boolean()和String()被当做普通函数，把任何类型的数据转换为number、boolean和string类型(注意不是其包装类型)：

|  |
| --- |
| **let** n = Number('123'); // 123, 相当于parseInt()或parseFloat() console.log(typeof n); // 'number' **let** b = Boolean('true'); // true console.log(typeof b); // 'boolean' **let** b2 = Boolean('false'); // true! 'false'字符串非空, 转换结果为true **let** b3 = Boolean(''); // false, 空字符串为false console.log(`b2 ${b2}, b3 ${b3}`); // b2 true, b3 false **let** s = String(123.45); // '123.45' console.log(typeof s); // 'string' |

这就是JavaScript特有的催眠魅力！

需要遵守规则：

① 不要使用new Number()、new Boolean()、new String()创建包装对象；

② 用parseInt()或parseFloat()转换任意类型到number；

③ 用String()转换任意类型到string；或直接调用某个对象的toString()方法；

④ 可以直接写if (obj) {...}，通常不必转换为boolean再判断；

⑤ typeof可以判断出number、boolean、string、function和undefined；

⑥ 判断Array使用Array.isArray(arr)；

⑦ 判断null使用obj === null；

⑧ 判断某个全局变量是否存在用typeof window.MY\_VAR === 'undefined'；

⑨ 函数内部判断某个变量是否存在用typeof my\_var === 'undefined'。

不是任何对象都有toString()方法，null和undefined没有！虽然null还伪装成了object类型。

整数的number对象调用toString()需要特别处理：

|  |
| --- |
| console.log(123..toString()); // 多加个. console.log((123).toString()); // 加括号 console.log(1.23.toString()); // 浮点数没问题 console.log((1.23).toString()); // 加括号也没问题 |

整数只用1点或浮点数用两点都会报语法错误!

* Date

Date对象用来表示日期和时间

获取系统时间：

|  |
| --- |
| **let** now = **new** Date(); console.log(now); // Thu Apr 05 2018 22:15:37 GMT+0900 (东京标准时间) console.log(now.getFullYear()); // 2018年 console.log(now.getMonth()); // 3, 表示4月, 注意月份范围0~11 console.log(now.getDate()); // 5日 console.log(now.getDay()); // 4, 星期四 console.log(now.getHours()); // 22时 console.log(now.getMinutes()); //15分 console.log(now.getSeconds()); // 37秒 console.log(now.getMilliseconds()); // 385毫秒 console.log(now.getTime()); // 1522934137385, number类型的时间戳 |

创建指定日期和时间的Date对象：

|  |
| --- |
| // ① 指定对应参数new一个对象 **let** d1 = **new** Date(2018, 3, 5, 22, 19, 12, 123); console.log(d1); // Thu Apr 05 2018 22:19:12 GMT+0900 (东京标准时间) // ② 解析格式化的日期字符串, 返回时间戳 **let** d2 = Date.parse('2018-04-19 22:22:22.22+0900'); console.log(d2); // 1524144142220 d2 = **new** Date(d2); // 时间戳转为Date对象 console.log(d2); // Thu Apr 19 2018 22:22:22 GMT+0900 (东京标准时间)  console.log(d2.toUTCString()); // 转为UTC时间, Thu, 19 Apr 2018 13:22:22 GMT |

* 正则表达式拾遗

正则表达式具有提取子串的强大功能。用()表示要提取的分组(Group)

如果正则表达式定义了组，就可以在RegExp对象上用exec()方法提取子串。

exec()方法匹配成功返回一个Array，第一个元素是匹配到的整个字符串，后面的字符串表示匹配成功的子串；匹配失败时返回null。

|  |
| --- |
| **let** re = /^(0?\d|1\d|2[0-3]):([0-5]\d):([0-5]\d)$/; **let** ret = re.exec('19:05:30'); console.log(ret); // ["19:05:30", "19", "05", "30", index: 0, input: "19:05:30"] console.log(ret.input);// 19:05:30 console.log(Array.isArray(ret)); // true console.log(re.exec('19:59:60')); // null |

* JSON

JSON非常简单，风靡Web世界，是ECMA标准。几乎所有编程语言都有解析JSON的库；JavaScript内置了JSON的解析，可以直接使用JSON。

把任何JavaScript对象变成JSON，就是把对象序列化成一个JSON格式的字符串，这样才能够通过网络传递给其他计算机。

如果收到一个JSON格式的字符串，只需要把它反序列化成一个JavaScript对象，就可以在JavaScript中直接使用这个对象了。

① 序列化：JSON.stringify(value,replacer,space)

value：要转换的对象；

replacer：可选，用于筛选对象的键值，可以是函数(每个键值都被函数处理)或数组(指定输出键的数组)；

space：可选，表示缩进。

|  |
| --- |
| **let** hikari = {  name: 'hikari',  age: 25,  gender: **true**,  height: 1.75,  grade: **null**,  skills: ['JavaScript', 'Java', 'Python', 'PhotoShop'] }; console.log(JSON.stringify(hikari)); |

直接序列化一行显示，太难看：



指定缩进：

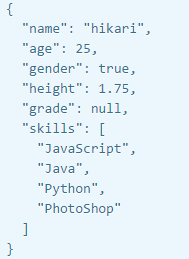
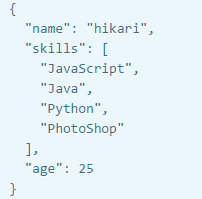
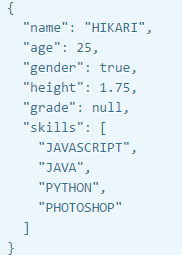
|  |
| --- |
| console.log(JSON.stringify(hikari, **null**, ' ')); |

用数组指定要输出的键及键的顺序：

|  |
| --- |
| console.log(JSON.stringify(hikari, ['name', 'skills', 'age'], ' ')); |

传入函数对键值处理，比如把字符串类型的值改为大写：

|  |
| --- |
| console.log(JSON.stringify(hikari, (key, value) => typeof value === 'string' ? value.toUpperCase() : value, ' ')); |

如果想要精确控制序列化，可以给对象定义一个toJSON()方法，直接返回JSON应该序列化的数据：

|  |
| --- |
| **let** hikari = {  name: 'hikari',  age: 25,  gender: **true**,  height: 1.75,  grade: **null**,  skills: ['JavaScript', 'Java', 'Python', 'PhotoShop'],  toJSON: **function** () { // 只输出name和age, 并且改变了key  **return** {Name: **this**.name, Age: **this**.age};  } };  console.log(JSON.stringify(hikari)); // {"Name":"hikari","Age":25} |

**20180406**

② 反序列化：JSON.parse(jsonString,reviver)

将JSON格式的字符串变成JavaScript对象

jsonString：JSON格式的字符串

reviver：可选，转换函数，对象每个键值对都会调用此函数。

|  |
| --- |
| console.log(JSON.parse('[1,2,3]')); // [1, 2, 3], 数组 **let** s = '{"name":"hikari","age":25}'; // JSON字符串要双引号 console.log(JSON.parse(s)); // Object {name: "hikari", age: 25} console.log(JSON.parse(s, (k, v) => k === 'name' ? v + '同学' : v)); // Object {name: "hikari同学", age: 25} console.log(JSON.parse('true')); // true, boolean类型 console.log(typeof JSON.parse('123.45')); // 123.45, number类型 |

* 面向对象编程

大多数编程语言都有类和对象的基本概念，而JavaScript却不太一样。

JavaScript不区分类和实例，而是通过原型(prototype)实现面向对象编程。

|  |
| --- |
| **let** Student = {  name: 'hikari',  age: 25,  study: **function** () {  console.log(**this**.name + ' is good good studying');  } }; **let** tom = {name: 'tom'}; tom.\_\_proto\_\_ = Student; tom.study(); // tom is good good studying |

JavaScript的原型链没有类的概念，所有对象都是实例，继承不过是把一个对象的原型指向另一个对象而已。

如果将tom的\_\_proto\_\_指向Bird，此时他变成了一只鸟，不能study只能fly：

|  |
| --- |
| **let** Bird = {  name: '',  fly: **function** () {  console.log(**this**.name + ' is flying');  } }; tom.\_\_proto\_\_ = Bird; tom.fly(); // tom is flying  tom.study(); // Uncaught TypeError: tom.study is not a function |

注意：不要直接用obj.\_\_proto\_\_改变一个对象的原型，上面只是演示用。

Object.create()方法可以传入一个原型对象，并创建一个基于该原型的新对象：

|  |
| --- |
| **function** create\_student(name) {  // 基于原型对象Student创建一个新对象  **let** s = Object.create(Student);  s.name = name; // 初始化新对象  **return** s; } **let** tom = create\_student('tom'); tom.study(); // tom is good good studying console.log(tom.\_\_proto\_\_ === Student); // true |

* 创建对象

JavaScript对每个创建的对象都会设置一个原型，指向它的原型对象。

当访问一个对象的属性时，JavaScript引擎先在当前对象上查找该属性；如果没有，就到其原型对象上找；如果还没有，就一直上溯到Object.prototype对象；最终如果还没有找到，返回undefined。

如一个Array对象arr = [1, 2, 3]，其原型链是：

arr --> Array.prototype --> Object.prototype --> null

Array.prototype定义了indexOf()、push()等方法，因此可以在所有的Array对象上直接调用这些方法。

函数也是一个对象，对于函数function f(){return 0;}原型链是：

f --> Function.prototype --> Object.prototype --> null

Function.prototype定义了apply()等方法，因此所有函数都可以调用这些方法。

如果原型链很长，那么访问一个对象的属性就会因为花更多的时间查找而变得更慢，因此注意原型链不要太长。

* 构造函数

除了用{...}创建一个对象，还可以用构造函数创建对象：

|  |
| --- |
| **function** Student(name, age) {  **this**.name = name;  **this**.age = age;  **this**.hello = **function** () {  console.log(`我的名字叫${**this**.name}，今年${**this**.age}岁了！`);  } } **let** hikari = **new** Student('hikari', 25); hikari.hello(); // 我的名字叫hikari，今年25岁了！ |

注意：如果不写new，Student就是一个普通函数，它返回undefined；反之写new就变成了构造函数，它绑定的this指向新创建的对象，并默认返回this，也就是说不需要在最后写return this;。

hikari的原型链：

hikari --> Student.prototype --> Object.prototype --> null

|  |
| --- |
| console.log(hikari.\_\_proto\_\_ === Student.prototype); console.log(Student.prototype.\_\_proto\_\_ === Object.prototype); console.log(Object.prototype.\_\_proto\_\_ === **null**); console.log(hikari.constructor === Student); console.log(Student.prototype.constructor === Student); console.log(Object.getPrototypeOf(hikari) === Student.prototype); console.log(hikari instanceof Student); |

上面全是true…

注意：Student.prototype指向的对象就是hikari的原型对象，这个原型对象有个属性constructor，指向Student函数本身。

Student函数恰好有个属性prototype指向hikari的原型对象，但是hikari没有prototype属性，但可以用\_\_proto\_\_这个非标准用法查看。

不过还有个问题：

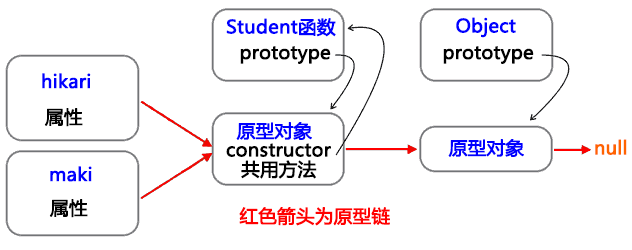
|  |
| --- |
| **let** maki = **new** Student('maki', 15); maki.hello(); // 我的名字叫maki，今年15岁了！ console.log(hikari.hello === maki.hello); // false |

通过new Student()创建多个对象，它们的hello()方法代码相同，但属于各自拥有，不是同一个方法；如果能够共享方法，可以节省很多内存空间。

根据对象的属性查找原则，只要把hello()方法移动到hikari、maki这些对象共同的原型Student.prototype上：

|  |
| --- |
| **function** Student(name, age) {  **this**.name = name;  **this**.age = age; } Student.prototype.hello = **function** () {  console.log(`我的名字叫${**this**.name}，今年${**this**.age}岁了！`); }; **let** hikari = **new** Student('hikari', 25); **let** maki = **new** Student('maki', 15); hikari.hello(); // 我的名字叫hikari，今年25岁了！ maki.hello(); // 我的名字叫maki，今年15岁了！ console.log(hikari.hello === maki.hello); // true |

原型链：



* 忘记写new怎么办?

如果一个函数被定义为用于创建对象的构造函数，但调用时忘了写new：

strict模式，this.name = name将报错，因为this绑定为undefined；非strict模式，this.name = name不报错，因为this绑定为window，于是无意间创建了全局变量name，并且返回undefined，这个结果更糟糕。

所以调用构造函数千万**不要忘记**写new。为了区分普通函数和构造函数，按照约定，构造函数首字母应当大写，而普通函数首字母应当小写。这样的话一些语法检查工具如[jslint](http://www.jslint.com/)将可以检测到漏写的new。

可以编写一个create\_student()函数，在内部封装所有的new操作：

|  |
| --- |
| **function** Student(props) {  **this**.name = props.name || '匿名'; // 默认名字为'匿名'  **this**.age = props.age || 20; // 默认年龄为20 } Student.prototype.hello = **function** () {  console.log(`我的名字叫${**this**.name}，今年${**this**.age}岁了！`); }; **function** create\_student(props) {  **return new** Student(props || {}) } **let** hikari = create\_student({name: 'hikari', age: 25}); **let** haha = create\_student({name: '哈哈'}); **let** no\_name = create\_student(); hikari.hello(); // 我的名字叫hikari，今年25岁了！ haha.hello(); // 我的名字叫哈哈，今年20岁了！ no\_name.hello(); // 我的名字叫匿名，今年20岁了！ |

这个create\_student()函数不需要用new调用，参数非常灵活。

如果创建的对象有很多属性，只需要传递需要的某些属性，剩下的属性可以用默认值。由于参数是一个Object，无需记忆参数的顺序。

**20180407**

* 原型继承

在传统的基于Class的语言如Java、C++中，继承的本质是扩展一个已有的Class，并生成新的Subclass。由于这类语言严格区分类和实例，继承实际上是类型的扩展。但是JavaScript由于采用原型继承，无法直接扩展一个Class，因为根本不存在Class这种类型。

比如要基于Student扩展出PrimaryStudent，可以先定义出PrimaryStudent：

|  |
| --- |
| **function** PrimaryStudent(props) {  Student.call(**this**, props); // 调用Student构造函数, 绑定this变量  **this**.grade = props.grade || 1; } |

此时PrimaryStudent创建的对象p的原型是：

p --> PrimaryStudent.prototype --> Object.prototype --> null

需要将原型链修改为：

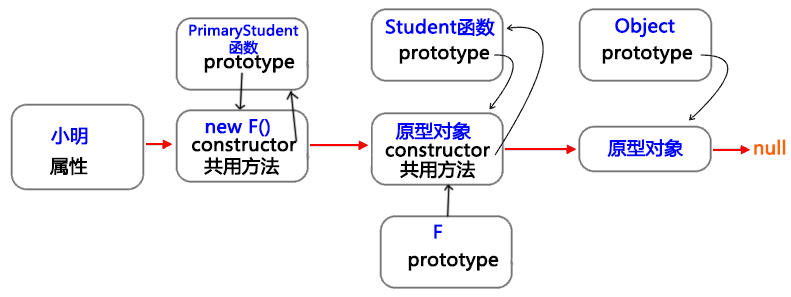
p --> PrimaryStudent.prototype --> Student.prototype --> Object.prototype --> null

原型链正确，继承关系就正确。新的基于PrimaryStudent创建的对象不但能调用PrimaryStudent.prototype定义的方法，也可以调用Student.prototype定义的方法。

必须借助一个中间对象来实现正确的原型链，这个中间对象的原型要指向Student.prototype。为了实现这一点，参考道爷 (发明JSON的道格拉斯)的代码，中间对象可以用一个空函数F来实现：

|  |
| --- |
| **function** F() { // 空函数F } // 把F的原型指向Student.prototype F.prototype = Student.prototype; // 把PrimaryStudent的原型指向一个新的F对象, F对象的原型正好指向Student.prototype PrimaryStudent.prototype = **new** F(); // 把PrimaryStudent原型的构造函数修复为PrimaryStudent PrimaryStudent.prototype.constructor = PrimaryStudent; // 在PrimaryStudent原型(就是new F()对象)上定义方法 PrimaryStudent.prototype.show\_grade = **function** () {  console.log(**this**.name + '今年' + **this**.grade + '年级。'); }; **let** p = **new** PrimaryStudent({name: '小明', age: 7}); p.hello(); // 我的名字叫小明，今年7岁了！ p.show\_grade(); // 小明今年1年级。 // 验证原型 console.log(p.\_\_proto\_\_ === PrimaryStudent.prototype); // true console.log(PrimaryStudent.prototype.\_\_proto\_\_ === Student.prototype); // true console.log(Student.prototype.\_\_proto\_\_ === Object.prototype); // true // 验证继承关系 console.log(p instanceof PrimaryStudent); // true console.log(p instanceof Student); // true |

函数F仅用于桥接，仅创建了一个new F()实例，没有改变原有的Student定义的原型链。



如果把继承用inherits()函数封装起来，还可以隐藏F的定义，并简化代码：

|  |
| --- |
| **function** inherits(Child, Parent) { // 继承函数,将继承封装和隐藏F  **let** F = **function** () {  };  F.prototype = Parent.prototype;  Child.prototype = **new** F();  Child.prototype.constructor = Child; } inherits(PrimaryStudent, Student); // 实现原型继承链 // 绑定方法到PrimaryStudent原型 PrimaryStudent.prototype.show\_grade = **function** () {  console.log(**this**.name + '今年' + **this**.grade + '年级。'); }; **let** p = **new** PrimaryStudent({name: '小明', age: 7}); p.hello(); // 我的名字叫小明，今年7岁了！ p.show\_grade(); // 小明今年1年级。 |

JavaScript的原型继承实现方式：

① 定义新的构造函数，并在内部用call()调用希望继承的构造函数，并绑定this；

② 借助中间函数F实现原型链继承，最好通过封装的inherits函数完成；

③ 在新构造函数的原型上定义新方法。

* class继承

JavaScript的原型继承理解起来比传统的类-实例模型困难，最大的缺点是继承的实现需要编写大量代码，并且需要正确实现原型链。

ES6开始被引入的新关键字class，其目的就是让定义类更简单。

|  |
| --- |
| **class** Student {  **constructor**(name, age) { // 构造函数  **this**.name = name;  **this**.age = age;  }  hello() { // 定义在原型对象上的函数, 没有function关键字  console.log(`我的名字叫${**this**.name}，今年${**this**.age}岁了！`);  } } **let** hikari = **new** Student('hikari', 25); **let** maki = **new** Student('maki', 15); console.log(hikari.hello === maki.hello); // true console.log(hikari.\_\_proto\_\_ === Student.prototype); // true |

class的定义包含了构造函数constructor和定义在原型对象上的函数(**没有**function关键字)，避免了Student.prototype.hello = function () {}这样分散的代码。

用class定义对象的另一个巨大的好处是继承更方便了。原型继承的中间对象，原型对象的构造函数等都不需要考虑了，直接通过extends实现：

|  |
| --- |
| **class** PrimaryStudent **extends** Student { // extends继承  **constructor**(name, age, grade) {  **super**(name, age); // super()调用父类的构造方法  **this**.grade = grade;  }  show\_grade() {  console.log(**this**.name + '今年' + **this**.grade + '年级。');  } } **let** p = **new** PrimaryStudent('小明', 7, 1); p.hello(); // 我的名字叫小明，今年7岁了！ p.show\_grade(); // 小明今年1年级。 |

PrimaryStudent的定义也是用class关键字，而extends则表示原型链对象来自Student。子类的构造函数参数可能与父类不一样，可以通过super()调用父类的构造函数初始化相同的参数。

ES6引入的class和原有的原型继承没有任何区别，class的作用是让JavaScript引擎实现原来需要自己编写的原型链代码。

不是所有的主流浏览器都支持ES6的class，但可以通过工具如[Babel](https://babeljs.io/)把class代码转换为传统的prototype代码。

* 浏览器

目前主流的浏览器：

① IE：国内用得最多的浏览器，历来对W3C标准支持**差**。从IE10开始支持ES6标准；

② Chrome：Google出品的基于Webkit内核浏览器，内置了非常强悍的JavaScript引擎——V8。由于Chrome一经安装就时刻保持自升级，最新版早就支持ES6；

③ Safari：Apple的Mac系统自带的基于Webkit内核的浏览器，从OS X 10.7 Lion自带的6.1版本开始支持ES6，目前最新版本的Safari早已支持ES6；

④ Firefox：Mozilla自己研制的Gecko内核和JavaScript引擎OdinMonkey。早期的Firefox按版本发布，后来终于聪明地学习Chrome的做法进行自升级，时刻保持最新；

⑤ 移动设备上目前iOS和Android两大阵营分别主要使用Apple的Safari和Google的Chrome，由于两者都是Webkit核心，结果HTML5首先在手机上全面普及(桌面绝对是Microsoft拖了后腿)，对JavaScript的标准支持也很好，最新版本均支持ES6。

⑥ 其他浏览器如Opera等由于市场份额太小就被自动忽略了。

各种国产浏览器只是做了一个壳，其核心调用的是IE，也有号称同时支持IE和Webkit的双核浏览器。

不同的浏览器对JavaScript支持的差异主要是，有些API的接口不一样，比如AJAX，File接口。对于ES6标准，不同的浏览器对各个特性支持也不一样。在编写JavaScript的时候，要充分考虑浏览器的差异，尽量让同一份代码能运行在不同的浏览器中。

* 浏览器对象

① window

window对象不但充当全局作用域，而且表示浏览器窗口。

window对象有innerWidth和innerHeight属性，可以获取浏览器窗口的内部宽度和高度。内部宽高是指除去菜单栏、工具栏、边框等，用于显示网页的净宽高。

还有outerWidth和outerHeight属性，可以获取浏览器窗口的整个宽高。

兼容性：IE<=8不支持。

|  |
| --- |
| // 按了F12后, 不按是一样的? console.log('window inner size: ' + window.innerWidth + ' x ' + window.innerHeight); // window inner size: 902 x 632 console.log('window outer size: ' + window.outerWidth + ' x ' + window.outerHeight); // window outer size: 1366 x 632 |

② navigator

navigator对象表示浏览器的信息，最常用的属性有：

1) navigator.appName：浏览器名称；

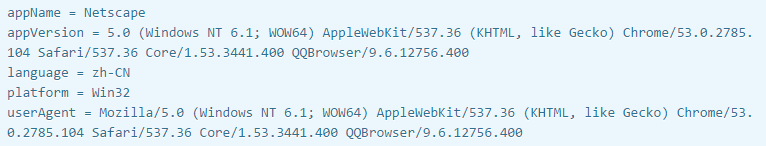
2) navigator.appVersion：浏览器版本；

3) navigator.language：浏览器设置的语言；

4) navigator.platform：操作系统类型；

5) navigator.userAgent：浏览器设定的User-Agent字符串。

|  |
| --- |
| console.log('appName = ' + navigator.appName); console.log('appVersion = ' + navigator.appVersion); console.log('language = ' + navigator.language); console.log('platform = ' + navigator.platform); console.log('userAgent = ' + navigator.userAgent); |



注意：navigator的信息可以很容易地被用户修改，所以JavaScript读取的值不一定是正确的。

③ screen

screen对象表示屏幕的信息，常用的属性有：

1) screen.width：屏幕宽度，以像素为单位；

2) screen.height：屏幕高度，以像素为单位；

3) screen.colorDepth：返回颜色位数，如8、16、24。

|  |
| --- |
| console.log(+screen.width + 'x' + screen.height + ' ' + screen.colorDepth); // 1366x768 24 |

④ location

location对象表示当前页面的URL信息。

一个完整的URL如：

http://www.example.com:8080/path/index.html?a=1&b=2#TOP

可以用location.href获取完整url。

获取URL各个部分的值：

|  |
| --- |
| console.log(location.protocol); // 'http' console.log(location.host); // 'www.example.com' console.log(location.port); // '8080' console.log(location.pathname); // '/path/index.html' console.log(location.search); // '?a=1&b=2' console.log(location.hash); // 'TOP' |

调用location.assign()加载一个新页面；调用location.reload()重新加载当前页面。

|  |
| --- |
| **if** (confirm('重新加载当前页' + location.href + '?')) { // 弹窗确认框  location.reload(); } **else** {  location.assign('http://www.baidu.com'); // 加载一个新URL地址 } |

⑤ document

document对象表示当前页面。由于HTML在浏览器中以DOM形式表示为树形结构，document对象就是整个DOM树的根结点。

document的title属性是从HTML的<title>标签读取，但是可以动态改变。

document.title = '新的title';

要查找DOM树的某个结点，需要从document对象开始查找。最常用的查找是根据ID和Tag Name。

用document对象提供的getElementById()和getElementsByTagName()可以按ID获得一个DOM结点和按Tag名称获得一组DOM结点。

document对象还有一个cookie属性，可以获取当前页面的Cookie。

Cookie是由服务器发送的key-value标示符。因为HTTP协议是无状态的，但是服务器要区分到底是哪个用户发过来的请求，就可以用Cookie来区分。当一个用户成功登录后，服务器发送一个Cookie给浏览器；此后浏览器访问该网站，会在请求头附上这个Cookie，服务器根据Cookie即可区分出用户。

JavaScript可以通过document.cookie读取到当前页面的Cookie，而用户的登录信息通常也在Cookie中，这造成了巨大的安全隐患。如果HTML引入的第三方JavaScript存在恶意代码，则黑客将直接获取到网站的用户登录信息。

为了解决这个问题，服务器在设置Cookie时可以使用httpOnly，设定了httpOnly的Cookie将不能被JavaScript读取。这个行为由浏览器实现，主流浏览器均支持httpOnly选项，IE从IE6 SP1开始支持。

为了确保安全，服务器端在设置Cookie时，应该始终坚持使用httpOnly。

⑥ history

history对象保存了浏览器的历史记录，JavaScript可以调用history对象的back()或forward()，相当于用户点击了浏览器的后退或前进按钮。

这个对象属于历史遗留产物，对于现代Web页面来说，由于大量使用AJAX和页面交互，简单粗暴地调用history.back()可能会让用户感到非常愤怒。

新手开始设计Web页面时喜欢在登录页登录成功时调用history.back()，试图回到登录前的页面。这是一种错误的方法。

现在任何情况都**不应该**使用history这个对象了。

**20180408**

* 操作DOM

由于HTML文档被浏览器解析后就是一棵DOM树，要改变HTML的结构，就需要通过JavaScript来操作DOM。

操作一个DOM结点的操作：

① 修改该DOM结点的HTML内容；

② 遍历该DOM结点下的子结点，以便进行进一步操作；

③ 在该DOM结点下添加一个子结点，相当于动态增加一个HTML结点；

④ 将该结点从HTML中删除，相当于删除该DOM结点及其所有子结点。

在操作一个DOM结点前，需要先拿到这个DOM结点。最常用的方法是document.getElementById()和document.getElementsByTagName()，以及CSS选择器document.getElementsByClassName()。

由于ID在HTML文档中是唯一的，所以getElementById()可以直接定位唯一一个DOM结点。getElementsByTagName()和getElementsByClassName()总是返回一组DOM结点。

要精确选择DOM，可以先定位父结点，再从父结点开始选择，缩小范围。

第二种方法是使用新增的选择器querySelector()和querySelectorAll()，格式类似于样式选择器，使用更加方便。

严格地讲这里的DOM结点是指Element，但是DOM结点实际上是Node。在HTML中Node有很多，大多数时候只关心Element，也就是实际控制页面结构的Node，其他忽略。根结点Document已经自动绑定为全局变量document。

练习：

HTML结构：

|  |
| --- |
| <div id="test-div">  <div class="c-red">  <p id="test-p">JavaScript</p> <p>Java</p></div>  <div class="c-red c-green">  <p>Python</p><p>Ruby</p><p>Swift</p></div>  <div class="c-green">  <p>Scheme</p><p>Haskell</p></div></div> |

选择出指定条件的结点：

|  |
| --- |
| // 选择<p>JavaScript</p>: **let** js = document.querySelector('#test-p'); // 选择<p>Python</p>,<p>Ruby</p>,<p>Swift</p>: **let** arr = document.querySelector('.c-red.c-green').children; // 选择<p>Haskell</p>: **let** haskell = document.querySelectorAll('.c-green')[1].lastElementChild; // 测试 console.log(js.innerText); // JavaScript **for** (**let** i **of** arr) {  console.log(i.innerText); // Python, Ruby, Swift } console.log(haskell.innerText); // Haskell |

js标签放在head标签里，先加载head再加载body，这样js去获取div结点还不存在，结果是null。解决方法是将script标签代码放到HTML最后；或者window.onload=function(){执行代码};

* 修改DOM

① 修改innerHTML属性，不但可以修改一个DOM结点的文本内容，还可以直接通过HTML片段修改DOM结点内部的子树：

|  |
| --- |
| <script>'use strict';  **let** p = document.querySelector('#para');  p.innerHTML = 'hello, <span class="name">hikari<span>'; </script> <style> .name{ color: red; font: 16px/16px "Microsoft YaHei";}</style>  <p id="para"></p> |



用innerHTML时要**注意**：如果字符串是通过网络拿到的，要对字符编码来避免XSS攻击(跨站脚本攻击，Cross Site Scripting)。

② 修改innerText或textContent属性，自动对字符串进行HTML编码，保证无法设置任何HTML标签：

p.innerText = 'hello, <span class="name">hikari<span>';

显示：

因为HTML被自动编码：hello, &lt;span class="name"&gt;hikari&lt;span&gt;

浏览器显示：hello, <span class="name">hikari<span>没有当成HTML标签

两者区别是：innerText不返回隐藏元素的文本；textContent返回所有文本。

DOM结点的style属性对应CSS，可以直接获取或设置。CSS允许font-size这样的名称，但它并非JavaScript有效的属性名，需要改写为驼峰式命名fontSize。

|  |
| --- |
| **let** p = document.querySelector('#para'); p.style.color = '#ff00ff'; p.style.fontSize = '14px'; p.style.paddingLeft = '2em'; p.innerHTML = 'hello, <span class="name">hikari<span>'; |



* 插入DOM

如果这个DOM结点是空的，如<div></div>，可以直接使用innerHTML = '<span>child</span>'就可以修改DOM结点的内容，相当于插入了新的结点。

如果这个DOM结点不是空的，innerHTML会直接替换原来的所有子结点。

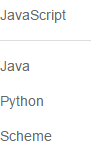
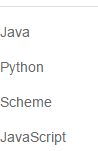
① appendChild

parent.appendChild(new\_node)：添加子结点作为父结点最后一个子结点。

|  |
| --- |
| <p id="js">JavaScript</p><hr> <div id="list">  <p id="java">Java</p>  <p id="python">Python</p>  <p id="scheme">Scheme</p></div> |

把"#js"添加到"#list"的最后一项：

|  |
| --- |
| **let** js = document.querySelector('#js'); **let** list = document.querySelector('#list'); list.appendChild(js); |

因为插入的js结点已经存在于当前文档树，所以该结点首先会从原先的位置删除，再插入到新的位置。

更多的时候会从零创建一个新结点，然后插入到指定位置：

|  |
| --- |
| **let** haskell = document.createElement('p'); haskell.id = 'haskell'; haskell.innerText = 'Haskell'; list.appendChild(haskell); |

动态创建一个结点添加到DOM树中，可以实现很多功能。

比如动态创建一个<style>结点，然后添加到<head>结点末尾，可以动态地给文档添加新的CSS：

|  |
| --- |
| **let** s = document.createElement('style'); // 创建style结点 s.setAttribute('type', 'text/css'); //设置属性 s.innerHTML = 'p { color: red }'; document.getElementsByTagName('head')[0].appendChild(s); // 动态添加样式 |

② insertBefore

parent.insertBefore(new\_node, node)：子结点插入到node之前

|  |
| --- |
| **let** cpp = document.createElement('p'); cpp.id = 'cpp'; cpp.innerText = 'C++'; **let** python = document.querySelector('#python'); list.insertBefore(cpp, python); // 将C++插到Python之前 |

练习：HTML结构：

|  |
| --- |
| <ol id="test-list">  <li class="lang">Scheme</li>  <li class="lang">Ruby</li>  <li class="lang" style="color: red">JavaScript</li>  <li class="lang">Haskell</li>  <li class="lang" style="color: blue">Python</li></ol> |

按字符串顺序重新排序DOM结点。

|  |
| --- |
| **let** list = document.getElementById('test-list'); **let** arr = []; // children得到的不是数组, 但可以迭代 **for** (**let** i **of** list.children) {  arr.push(i); } // 按照文本内容字符串顺序将结点排序 arr.sort((x, y) => x.innerText > y.innerText); list.innerHTML = ''; **for** (**let** i **of** arr) { // 清空list, 再按顺序添加结点  list.appendChild(i); } |

另一方法：

|  |
| --- |
| **let** list = document.getElementById('test-list'); **let** arr = list.children; // 冒泡排序 **for** (**let** i = arr.length - 1; i > 0; i--) {  **for** (**let** j = 0; j < i; j++) {  // 如果后面的字符串小, 将其往前面插  **if** (arr[j].innerText > arr[j + 1].innerText) {  list.insertBefore(arr[j + 1], arr[j]);  }  } } |

* 删除DOM

要删除一个结点，首先要获得该结点本身以及它的父结点，然后调用父结点的removeChild把自己删除：

|  |
| --- |
| // 获取待删除结点 **let** node = document.getElementById('to\_remove'); // 获取其父结点 **let** parent = node.parentElement; // 调用父结点的removeChild方法删除子结点 **let** removed = parent.removeChild(node); console.log(node === removed); // true |

注意：删除后的结点虽然不在文档树中，但还在内存中，可以随时再次被添加到别的位置。

当遍历一个父结点的子结点并进行删除操作时，要注意children属性是一个只读属性，并且它在子结点变化时会实时更新。

比如上面test-list的例子，要删除所有的子结点：

|  |
| --- |
| **let** parent = document.getElementById('test-list'); **let** arr = parent.children; **for** (**let** i = 0; i < arr.length; i++) {  parent.removeChild(arr[i]); } |



结果0,2,4被删除了，1,3没有。原因是当第0个被删除后，后面结点的下标都减1，接着删除下标1的结点其实是原来的下标2。

将i++删除即可，每次都删除第0个元素，删除length次，刚好删完。

* 操作表单

表单的输入框、下拉框等可以接收用户输入，用JavaScript操作表单，可以获得用户输入内容，或者对一个输入框设置新的内容。

HTML表单的输入控件主要有：

① 文本框<input type="text">，用于输入文本；

② 密码框<input type="password">，用于输入密码；

③ 单选框<input type="radio">，用于选择一项；

④ 复选框<input type="checkbox">，用于选择多项；

⑤ 下拉框<select>，用于选择一项；

⑥ 隐藏文本<input type="hidden">，用户不可见，但表单提交时会把隐藏文本发送到服务器。

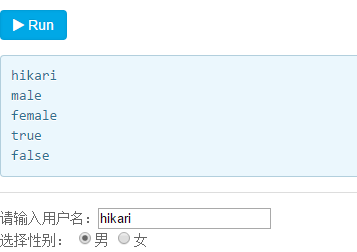
1. 获取值

<input>结点调用value获得对应的用户输入值，适用于text、password、hidden和select。但对于单选框和复选框，value属性返回的永远是HTML预设的值，而需要获得的实际是用户是否勾上了选项，应该用checked判断。

|  |
| --- |
| <label for="user">请输入用户名：</label><input type="text" id="user"><br> <label>选择性别：</label> <input type="radio" name="gender" value="male" checked="checked" id="male"><label for="male">男</label> <input type="radio" name="gender" value="female" id="female"><label for="female">女</label> |

|  |
| --- |
| **let** user = document.getElementById('user'); console.log(user.value); // 用户输入的值 **let** male = document.getElementById('male'); **let** female = document.getElementById('female'); console.log(male.value); // 单选框和复选框的value是预设的值 console.log(female.value); console.log(male.checked); // checked判断选中还是没选中 console.log(female.checked); |

输入后点击Run：



2. 设置值

对于text、password、hidden和select，直接设置value；

对于单选框和复选框，设置checked为true或false即可。

|  |
| --- |
| **let** user = document.getElementById('user'); user.value = 'maki'; // 用户输入的值 **let** female = document.getElementById('female'); female.checked = **true**; |



3. HTML5控件

HTML5新增了大量标准控件，常用的包括date、datetime、datetime-local、color等，它们都使用<input>标签：

|  |
| --- |
| <input type="date" value="2018-04-08" title=""> <input type="datetime-local" value="2018-04-08T16:00:00" title=""> <input type="color" value="#ff0000" title=""> |



点击可以选择日期、日期时间、颜色。

不支持HTML5的浏览器无法识别新的控件，会当做type="text"来显示。支持HTML5的浏览器将获得格式化的字符串。例如，type="date"类型的input的value将保证是一个有效的YYYY-MM-DD格式的日期或空字符串。

4. 提交表单

① 通过<form>的submit()方法提交一个表单，例如绑定一个<button>的click事件，在JavaScript代码中提交表单：

|  |
| --- |
| <form id="info">  请输入用户名：<input type="text" name="user"><br>  请输入密码：<input type="password" name="pwd"><br>  <button type="button" onclick="doSubmitForm()">提交</button></form> |

|  |
| --- |
| **function** doSubmitForm() {  **let** form = document.getElementById('info');  // do sth.  form.submit(); } |

这种方式的缺点是扰乱了浏览器对form的正常提交。浏览器默认点击<button type="submit">时提交表单，或者用户在最后一个输入框按回车键。

② 响应<form>本身的onsubmit事件，在提交form时作修改：

|  |
| --- |
| <form id="info" method="post" onsubmit="**return** checkForm()">  <label for="user">请输入用户名：</label><input type="text" name="user" id="user"><br>  <label for="pwd">请输入密码：</label><input type="password" name="pwd" id="pwd"><br>  <button type="submit">提交</button></form> |

|  |
| --- |
| **function** checkForm() {  **let** form = document.getElementById('info');  // do sth.  // return true告诉浏览器继续提交, 如果return false浏览器将不会继续提交form  // 此时通常用户输入有误, 提示用户错误信息后终止提交form  **return true**; } |

③ <input type="hidden">

在检查和修改<input>时，要充分利用<input type="hidden">来传递数据。

例如很多登录表单希望用户输入用户名和密码，但出于安全考虑，提交表单时不传输明文密码，而是密码的MD5。普通JavaScript开发人员会直接修改<input>：

|  |
| --- |
| <script src="http://cdn.bootcss.com/blueimp-md5/1.1.0/js/md5.min.js"></script> <script>  'use strict';  **function** checkForm() {  **let** pwd = document.getElementById('pwd');  // 把用户输入的密码转为MD5  pwd.value = md5(pwd.value);  alert(pwd.value);  // do sth.  **return true**;  } </script> |

这个做法看上去没啥问题，但用户输入了密码提交时，密码框的显示会突然从几个\*变成32个\* (一瞬间…)

要想不改变用户的输入，可以利用<input type="hidden">实现：

|  |
| --- |
| <form id="info" method="post" onsubmit="**return** checkForm()">  <label for="user">请输入用户名：</label><input type="text" name="user" id="user"><br>  <label for="input-pwd">请输入密码：</label><input type="password" id="input-pwd">  <input type="hidden" id="md5-pwd" name="pwd"><br>  <button type="submit">提交</button></form> |

|  |
| --- |
| **function** checkForm() {  **let** input\_pwd = document.getElementById('input-pwd');  **let** md5\_pwd = document.getElementById('md5-pwd');  // 把用户输入的密码变为MD5存到hidden  md5\_pwd.value = md5(input\_pwd.value);  alert(md5\_pwd.value);  // 继续下一步:  **return true**; } |

hidden控件标记了name="pwd"，且md5赋值给了其value，数据会被提交；而用户输入的密码框的<input>没有name属性，数据不会被提交。

POST提交的表单数据：



**20180409**

* 操作文件

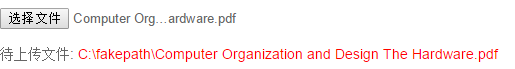
在HTML表单中，可以上传文件的唯一控件就是<input type="file">。

**注意**：当一个表单包含<input type="file">时，表单的enctype必须指定为multipart/form-data，method必须指定为post，浏览器才能正确编码并以multipart/form-data格式发送表单的数据。

出于安全考虑，浏览器只允许用户点击<input type="file">选择本地文件，用JavaScript对<input type="file">的value赋值是没有任何效果的。当用户选择了上传某个文件后，JavaScript也无法获得该文件的真实路径：

|  |
| --- |
| $(**function** () { // 原生JS也可以$(function)?  **let** fileInput = document.getElementById('test-file-upload'),  filePath = document.getElementById('test-get-filename');  fileInput.addEventListener('change', **function** () { // 绑定change事件  filePath.innerText = fileInput.value;  }); }); |

|  |
| --- |
| <form action="" method="post" enctype="multipart/form-data">  <p><input type="file" id="test-file-upload" name="test"></p>  <p>待上传文件: <span id="test-get-filename" style="color:red"></span></p></form> |

选择D盘的文件，结果显示C:\fakepath\文件名

通常上传的文件都由后台服务器处理，JavaScript可以在提交表单时对文件扩展名做检查，防止用户上传无效格式的文件。

* File API

由于JavaScript对用户上传的文件操作非常有限，尤其是无法读取文件内容，使得很多需要操作文件的网页不得不用Flash这样的第三方插件实现。

随着HTML5的普及，新增的File API允许JavaScript读取文件内容，提供了File和FileReader两个主要对象，可以获得文件信息并读取文件。

示例：读取用户选取图片文件，并在一个<div>中预览图像：

|  |
| --- |
| $(**function** () {  **let** fileInput = document.getElementById('test-image-file'),  info = document.getElementById('test-file-info'),  preview = document.getElementById('test-image-preview');  fileInput.addEventListener('change', **function** () { // 绑定change事件  preview.style.backgroundImage = ''; // 清除背景图片  **if** (!fileInput.value) { // 检查文件是否选择  info.innerHTML = '没有选择文件';  **return**;  }  **let** file = fileInput.files[0]; // 获取File引用  // 获取File信息  info.innerHTML = `文件：${file.name}<br>大小：${file.size}<br>修改：${file.lastModifiedDate}`;  **if** (file.type !== 'image/jpeg' && file.type !== 'image/png' && file.type !== 'image/gif') {  alert('不是有效的图片文件!');  **return**;  }  // 读取文件  **let** reader = **new** FileReader();  reader.onload = **function** (e) {  **let** data = e.target.result;  console.log(data); // data:image/png;base64,iVBORw0K...(base64编码)  preview.style.backgroundImage = `url(${data})`;  };  // 以DataURL的形式读取文件  reader.readAsDataURL(file);  }); }); |

|  |
| --- |
| #test-image-preview {  border: 1px solid rgb(204, 204, 204);  width: 500px;  height: 200px;  background-size: contain;  background-repeat: no-repeat;  background-position: center center; } |

|  |
| --- |
| <form action="" method="post" enctype="multipart/form-data">  <p>图片预览：</p>  <div id="test-image-preview"></div><br>  <p><input type="file" id="test-image-file" name="test"></p>  <p id="test-file-info">没有选择文件</p></form> |



以DataURL的形式读取到的文件是一个base64编码的字符串 (data:image/png;

base64,iVBORw0K...)，常用于设置图像。把字符串'base64,'后面的字符发给服务器并用Base64解码就可以得到原始文件的二进制内容。

* 回调

JavaScript的一个重要特性就是单线程执行模式。浏览器的JavaScript执行引擎在执行JavaScript代码时，总是以单线程模式执行。任何时候JavaScript代码都不可能同时有多于1个线程在执行。

JavaScript执行多任务实际上都是异步调用，比如上面的代码：

reader.readAsDataURL(file);

就会发起一个异步操作来读取文件内容。因为是异步操作，所以在JavaScript代码中就不知道什么时候操作结束，因此需要先设置一个回调函数：

|  |
| --- |
| reader.onload = **function** (e) {  // 当文件读取完成后, 自动调用此函数 }; |

当文件读取完成后，JavaScript引擎将自动调用回调函数。执行回调函数时，文件已经读取完毕，所以可以在回调函数内部安全地获得文件内容。

**20180411**

* AJAX

在Form表单提交，一般提交后在新页面里显示操作成功与否。如果由于网络太慢或者其他原因，就会得到一个404页面。

这就是Web的运作原理：一次HTTP请求对应一个页面。

如果要让用户留在当前页面，同时发出新的HTTP请求，就必须用JavaScript发送这个新请求，接收到数据后，再用JavaScript更新页面。这样，用户就感觉仍然停留在当前页面，但是数据却可以不断地更新。

AJAX (Asynchronous JavaScript and XML)，用JavaScript执行异步网络请求。

最早大规模使用AJAX的是Gmail，Gmail的页面在首次加载后，剩下的所有数据都依赖于AJAX更新。

用JavaScript写一个完整的AJAX代码需要注意：AJAX请求是异步执行的，要通过回调函数获得响应。

在现代浏览器上写AJAX主要依靠XMLHttpRequest对象：

|  |
| --- |
| **function** success(text) { // 请求成功做什么  **let** textarea = document.getElementById('test-response-text');  textarea.value = text; } **function** fail(code) { // 请求失败做什么  **let** textarea = document.getElementById('test-response-text');  textarea.value = 'Error code: ' + code; } $(**function** () {  **let** request = **new** XMLHttpRequest(); // 新建XMLHttpRequest对象  request.onreadystatechange = **function** () { // 状态发生变化时, 函数被回调  **if** (request.readyState === 4) { // 成功完成  // 判断响应结果  **if** (request.status === 200) {  // 成功, 通过responseText拿到响应的文本  **return** success(request.responseText);  } **else** {  // 失败, 根据响应码判断失败原因  **return** fail(request.status);  }  } **else** {  // HTTP请求还在继续...  }  };  // ajax不能跨域请求, 自己写个服务器呗…  request.open('GET', '/api/test');  request.send(); // 发送请求  alert('请求已发送，请等待响应...'); }); |

由于ajax不能跨域请求，在之前blog的Day14基础上添加视图函数：

|  |
| --- |
| @get('/api/test') def api\_get():  return {'name': 'hikari', 'age': 25, 'school': '皇家幼稚园', 'job': '搬砖'} @get('/test') def test():  return {'\_\_template\_\_': 'test.html'} |



对于低版本的IE，需要换一个ActiveXObject对象：

|  |
| --- |
| **function** success(text) {  **let** textarea = document.getElementById('test-ie-response-text');  textarea.value = text; } **function** fail(code) {  **let** textarea = document.getElementById('test-ie-response-text');  textarea.value = 'Error code: ' + code; } $(**function** () {  **let** request = **new** ActiveXObject('Microsoft.XMLHTTP'); // 新建Microsoft.XMLHTTP对象  // 与之前一样… }); |

通过检测window对象是否有XMLHttpRequest属性确定浏览器是否支持标准的XMLHttpRequest。**注意**：不要根据浏览器的navigator.userAgent检测浏览器是否支持某个JavaScript特性，因为这个字符串可以伪造，而且通过IE版本判断JavaScript特性将非常复杂。

-------------------------------------------------------------------------------------------------------

当创建XMLHttpRequest对象后，先设置onreadystatechange回调函数。在回调函数中，通常只需通过readyState === 4判断请求是否完成；如果已完成，再根据status === 200判断是否是一个成功的响应。

XMLHttpRequest对象的open(method, url, async?=true)方法，method指定GET或POST，url指定URL地址，async?指定是否使用异步，默认true。

**注意**：不要把async?指定为false，否则浏览器将停止响应，直到AJAX请求完成。如果这个请求耗时10秒，那么10秒内浏览器会处于假死状态。

最后调用的send()方法才是真正发送请求。GET请求不需要参数，POST请求需要把body部分以字符串或者FormData对象传进去。

* 安全限制

上面ajax请求URL可以是相对路径'/api/test'或绝对路径'127.0.0.1:8000/api/test'。但是不能跨域请求，比如改为'http://www.sina.com.cn'，肯定报错。

这是因为浏览器的同源策略导致的。默认情况下，JavaScript在发送AJAX请求时，URL的域名必须和当前页面完全一致：

① 域名要相同，www.example.com和example.com不同；

② 协议要相同，http和https不同；

③ 端口号要相同， 80端口和8000不同。

有的浏览器松一点，允许端口不同；大多数浏览器都会严格遵守这个限制。

-------------------------------------------------------------------------------------------------------

JavaScript请求外域的URL方法：

① 通过Flash插件发送HTTP请求；可以绕过浏览器的安全限制，但必须安装Flash，并且跟Flash交互。不过Flash用起来麻烦，现在用得也越来越少。

② 通过在同源域名下架设一个代理服务器转发，JavaScript负责把请求发送到代理服务器：'/proxy?url=http://www.sina.com.cn'；代理服务器把结果返回，这样就遵守了浏览器的同源策略，但需要服务器端额外做开发。

③ JSONP，限制是只能用GET请求，并且要求返回JavaScript。实际上是利用了浏览器允许跨域引用JavaScript资源：

<script src="http://example.com/abc.js"></script>

JSONP通常以函数调用的形式返回，例如返回JavaScript内容：foo(data);

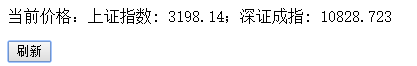
如果在页面中先准备好foo()函数，然后给页面动态加一个<script>结点，相当于动态读取外域的JavaScript资源，最后就等着接收回调。

示例：163股票查询URL：http://api.money.126.net/data/feed/0000001,1399001?

callback=refreshPrice

|  |
| --- |
| **function** refreshPrice(data) { // 回调函数  // data是getPrice()返回的JSON  **let** p = document.getElementById('test-jsonp');  // 从得到的data中获取数据显示  p.innerHTML = `当前价格：${data['0000001'].name}: ${data['0000001'].price}；${data['1399001'].name}: ${data['1399001'].price}`; } **function** getPrice() {  // 点击按钮, head标签插入一个script标签, 跨域引用资源  **let** js = document.createElement('script'),  head = document.getElementsByTagName('head')[0];  // callback=refreshPrice指定回调函数是refreshPrice(), 可以起别的函数名  js.src = 'http://api.money.126.net/data/feed/0000001,1399001?callback=refreshPrice';  head.appendChild(js); } |

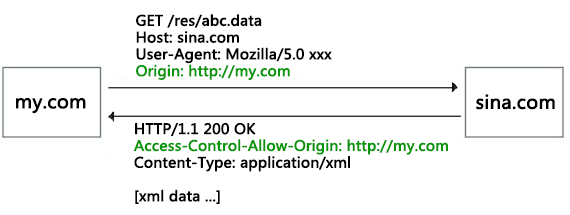
|  |
| --- |
| <div>  <p id="test-jsonp"></p>  <p><!-- 绑定按钮点击事件 -->  <button type="button" onclick="getPrice()">刷新</button></p></div> |



* CORS

CORS (Cross-Origin Resource Sharing)是HTML5规范定义的如何跨域访问资源。

Origin表示本域，浏览器当前页面的域。当JavaScript向外域(如sina.com)发起请求，浏览器收到响应后，首先检查Access-Control-Allow-Origin是否包含本域，如果是则跨域请求成功；反之请求失败，将无法获取到响应的任何数据。



假设本域是my.com，外域是sina.com，只要响应头Access-Control-Allow-Origin为http://my.com，或者是\*，本次请求就可以成功。

可见跨域能否成功取决于对方服务器是否愿意给设置一个正确的Access-Control-Allow-Origin，决定权始终在对方手中。

上面这种跨域请求，称之为简单请求。简单请求包括GET、HEAD和POST(POST的Content-Type类型仅限application/x-www-form-urlencoded、multipart/form-data和text/plain)，并且不能出现任何自定义头(如X-Custom: 1234)，通常能满足90%的需求。

在引用外域资源时，除了JavaScript和CSS外都要验证CORS。

如引用某个第三方CDN上的字体文件：

|  |
| --- |
| @font-face {  font-family: 'FontAwesome';  src: url('http://cdn.com/fonts/fontawesome.ttf') format('truetype'); } |

如果该CDN服务商未正确设置Access-Control-Allow-Origin，那么浏览器将无法加载字体资源。

对于PUT、DELETE以及其他类型如application/json的POST请求，在发送AJAX请求之前，浏览器会先发送一个OPTIONS请求(称为preflighted请求)到这个URL，询问目标服务器是否接受；

服务器必须响应并明确指出允许的Method：Access-Control-Allow-Methods字段

浏览器确认服务器响应的Access-Control-Allow-Methods头包含将要发送的AJAX请求的Method，才会继续发送AJAX，否则抛出错误。

由于以POST、PUT方式传送JSON格式的数据在REST中很常见，所以要跨域正确处理POST和PUT请求，服务器端必须正确响应OPTIONS请求。

// CORS怎么看不懂…

* Promise

JavaScript中，所有代码都是单线程执行的。

由于这个缺陷，导致JavaScript的所有网络操作、浏览器事件，都必须异步执行。异步执行可以用回调函数实现：

|  |
| --- |
| **function** callback() {  console.log('Done'); } console.log('before setTimeout()'); setTimeout(callback, 1000); // 暂停1秒钟后, 调用callback函数 console.log('after setTimeout()'); |

先显示before和after，等1秒后显示Done

异步操作会在将来某个时间点触发一个函数调用。AJAX就是典型的异步操作。

之前把回调函数success(request.responseText)和fail(request.status)写到一个AJAX操作里，正常但不好看，而且不利于代码复用。

|  |
| --- |
| **let** ajax = ajaxGet('/api/test'); ajax.ifSuccess(success).ifFail(fail); |

链式写法的好处：先统一执行AJAX逻辑，不关心如何处理结果；然后根据成功还是失败，在将来的某个时候调用success函数或fail函数。

这种承诺将来会执行的对象在JavaScript中称为Promise对象。

Promise有各种开源实现，在ES6中被统一规范，由浏览器直接支持。

示例：生成0~2随机数，小于1则等待一段时间后返回成功；否则返回失败：

|  |
| --- |
| window.onload = **function** () {  // 清除log  **let** logging = document.getElementById('test-promise-log');  **while** (logging.children.length > 1) {  logging.removeChild(logging.children[logging.children.length - 1]);  }  **function** log(s) { // 输出log到页面  **let** p = document.createElement('p');  p.innerHTML = s;  logging.appendChild(p);  }  **function** test(resolve, reject) {  **let** timeout = (Math.random() \* 2).toFixed(2); // 0~2随机数, 保留2位小数  log(`set timeout to: ${timeout} seconds.`);  setTimeout(**function** () {  **if** (timeout < 1) { // 暂停<1s, OK  log('call resolve()...');  resolve('200 OK');  }  **else** { // 暂停>=1s，失败  log('call reject()...');  reject(`timeout in ${timeout} seconds.`);  }  }, timeout \* 1000);  }  **let** p1 = **new** Promise(test); // Promise对象执行test()函数  **let** p2 = p1.then(**function** (result) { // 成功则调用resolve函数  log('成功：' + result);  });  **let** p3 = p2.catch(**function** (reason) { // 失败则调用reject函数  log('失败：' + reason);  }); }; |

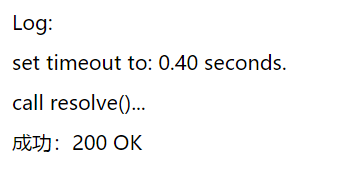
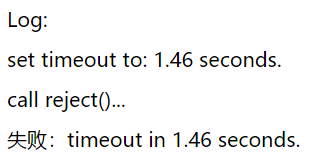
// 之前$(function(){})能用貌似是引入了all.js的原因…

|  |
| --- |
| .box { width: 500px; border: solid 1px #ccc; padding: 1em; margin: 20px; font: 20px/20px "Microsoft YaHei"; }  <div id="test-promise-log" class="box"><p>Log:</p></div> |

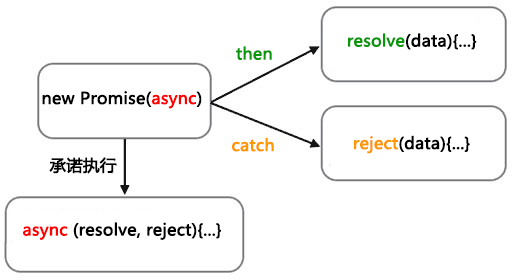
test()函数两个参数都是函数，如果执行成功，调用resolve()；否则调用reject()。test()函数只关心自身逻辑，并不关心具体的resolve和reject如何处理结果。

Promise对象可以链式调用：

|  |
| --- |
| **new** Promise(test).then(**function** (result) {  log(`成功：${result}`); }).catch(**function** (reason) {  log(`失败：${reason}`); }) |

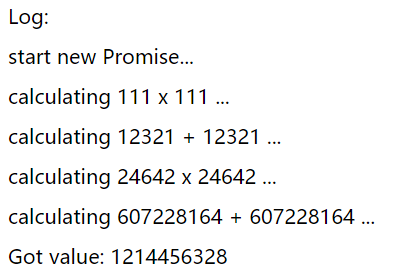
 

Promise最大的好处是在异步执行的流程中，把执行的代码和处理结果的代码清晰地分离了：



示例：串行执行一系列需要异步计算获得结果的任务

|  |
| --- |
| // 0.5秒后返回input\*input的计算结果 **function** multiply(input) {  **return new** Promise(**function** (resolve, reject) {  log(`calculating ${input} x ${input} ...`);  setTimeout(resolve, 500, input \* input);  }); } // 0.5秒后返回input+input的计算结果 **function** add(input) {  **return new** Promise(**function** (resolve, reject) {  log(`calculating ${input} + ${input} ...`);  setTimeout(resolve, 500, input + input);  }); } **let** p = **new** Promise(**function** (resolve, reject) {  log('start new Promise...');  resolve(111); }); p.then(multiply)  .then(add)  .then(multiply)  .then(add)  .then(**function** (result) {  log(`Got value: ${result}`);  }); |

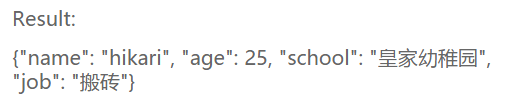


setTimeout可以看成一个模拟网络等异步执行的函数。

示例：将之前AJAX异步执行函数转换为Promise对象：

|  |
| --- |
| <link rel="stylesheet" href="/static/css/all.css"> <script src="/static/js/all.js"></script>  **function** ajax(method, url, data) {  **let** request = **new** XMLHttpRequest(); // 新建XMLHttpRequest对象  **return new** Promise(**function** (resolve, reject) { // 返回Promise对象  request.onreadystatechange = **function** () { // 状态发生变化时, 函数被回调  **if** (request.readyState === 4) {  **if** (request.status === 200) {  // 成功, 通过responseText拿到响应的文本  resolve(request.responseText);  } **else** {  // 失败, 根据响应码判断失败原因  reject(request.status);  }  }  };  request.open(method, url);  request.send(); // 发送请求  alert('请求已发送，请等待响应...');  }); } $(**function** () {  **let** ret = document.getElementById('test-promise-ajax-result');  **let** p = ajax('GET', '/api/test');  p.then(**function** (text) { // 如果AJAX成功, 获得响应内容  ret.innerText = text;  }).catch(**function** (status) { // 如果AJAX失败, 获得响应代码  ret.innerText = `ERROR: ${status}`;  }); }); |

|  |
| --- |
| <div class="box">  <p>Result:</p>  <p id="test-promise-ajax-result"></p></div>  .box { width: 500px; border: solid 1px #ccc; padding: 1em; margin: 20px; font: 20px/24px "Microsoft YaHei"; } |



// 为什么字是灰色的?过长的匿名函数嵌套看得很蛋疼…

除了串行执行若干异步任务外，Promise还可以并行执行异步任务。

如一个页面聊天系统，需要从两个不同的URL分别获得用户的个人信息和好友列表，这两个任务是可以并行执行的，用Promise.all()实现：

|  |
| --- |
| **let** p1 = **new** Promise(**function** (resolve, reject) {  setTimeout(resolve, 500, 'P1'); // 毫秒后面是传入resolve函数的参数, 可选 }); **let** p2 = **new** Promise(**function** (resolve, reject) {  setTimeout(resolve, 600, 'P2'); }); // 同时执行p1和p2, 并在它们都完成后执行then Promise.all([p1, p2]).then(**function** (results) {  console.log(results); // 获得一个Array: ['P1', 'P2'] }); |

有时候多个异步任务是为了容错。比如同时向两个URL读取用户的个人信息，只需要获得先返回的结果即可，用Promise.race()实现：

|  |
| --- |
| **let** p1 = **new** Promise(**function** (resolve, reject) {  setTimeout(resolve, 500, 'P1'); }); **let** p2 = **new** Promise(**function** (resolve, reject) {  setTimeout(resolve, 600, 'P2'); }); Promise.race([p1, p2]).then(**function** (result) {  console.log(result); // 'P1', 相当于赛跑, 获得先返回的值 }); |

race()就是赛跑，p1执行较快，Promise的then()将获得结果'P1'；p2仍在继续执行，但执行结果将被丢弃。

组合使用Promise，可以把很多异步任务以并行和串行的方式组合起来执行。

**20180413**

* Canvas

Canvas是HTML5新增组件，就像一块幕布，可以用JavaScript在上面绘制各种图表、动画等。

没有Canvas的年代，绘图只能借助Flash插件实现，页面需要JavaScript和Flash交互。有了Canvas再也不需要Flash了，直接使用JavaScript完成绘制。

一个Canvas定义了一个指定尺寸的矩形框，在这个范围内可以随意绘制。

由于浏览器对HTML5标准支持不一致，通常在<canvas>内部添加一些说明性HTML代码，如果浏览器支持Canvas，它将忽略<canvas>内部的HTML，如果浏览器不支持Canvas，它将显示<canvas>内部的HTML：

|  |
| --- |
| #test-canvas { width: 200px; height: 100px; border: 1px solid #cccccc;} <canvas id="test-canvas"><p>浏览器不支持Canvas...</p></canvas>  **let** canvas = document.getElementById('test-canvas'); **if** (canvas.getContext) {  console.log('浏览器支持Canvas!'); } **else** {  console.log('浏览器不支持Canvas!'); } |

矩形框什么都没有，console显示：浏览器支持Canvas!

getContext('2d')方法得到CanvasRenderingContext2D对象，所有绘图操作都需要通过这个对象完成。

**let** context = canvas.getContext('2d');

HTML5还有一个WebGL规范，允许在Canvas中绘制3D图形：

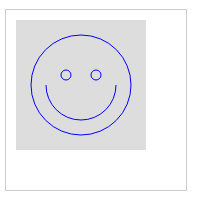
gl = canvas.getContext("webgl");

* 绘制形状

Canvas的坐标以左上角为原点，水平向右为X轴，垂直向下为Y轴，以像素为单位，所以每个点都是非负整数。

CanvasRenderingContext2D对象有若干方法绘制图形：

|  |
| --- |
| #test-shape-canvas { border: 1px solid #cccccc; margin-top: 15px;}  <canvas id="test-shape-canvas" width="180" height="180"><!--为什么宽高写在style标签图变形了...--></canvas>  **let** canvas = document.getElementById('test-shape-canvas'),  context = canvas.getContext('2d'); context.clearRect(0, 0, 180, 180); // 擦除(0,0)位置大小为200x200的矩形, 把该区域变为透明 context.fillStyle = '#dddddd'; // 设置填充颜色 context.fillRect(10, 10, 130, 130); // 把(10,10)位置大小为130x130的矩形涂色 // 利用Path绘制复杂路径 **let** path = **new** Path2D(); // arc(x, y, radius, startAngle, endAngle, anticlockwise?) path.arc(75, 75, 50, 0, Math.PI \* 2, **true**); // 大圆 path.moveTo(110, 75); path.arc(75, 75, 35, 0, Math.PI, **false**); // 半圆 path.moveTo(65, 65); path.arc(60, 65, 5, 0, Math.PI \* 2, **true**); // 小圆 path.moveTo(95, 65); path.arc(90, 65, 5, 0, Math.PI \* 2, **true**); // 小圆 context.strokeStyle = '#0000ff'; // 绘制颜色 context.stroke(path); // 绘制路径 |



* 绘制文本

绘制文本是在指定位置输出文本，可以设置文本的字体、样式、阴影等，与CSS完全一致：

|  |
| --- |
| #test-text-canvas { border: 1px solid #cccccc; margin-top: 15px;}  <canvas id="test-text-canvas" width="180" height="80"></canvas>  **let** canvas = document.getElementById('test-text-canvas'),  context = canvas.getContext('2d'); context.clearRect(0, 0, canvas.width, canvas.height); context.shadowOffsetX = 2; context.shadowOffsetY = 2; context.shadowBlur = 2; context.shadowColor = '#ff6666'; context.font = '24px "Meiryo'; context.fillStyle = '#ff0000'; context.fillText('西木野真姫', 20, 40); |



* jQuery

江湖传言，全世界大约有80~90%的网站直接或间接使用了jQuery。

jQuery为什么这么火：

① 消除浏览器差异：不需要写冗长代码来针对不同的浏览器绑定事件，编写AJAX等代码；

② 简洁的操作DOM的方法：写$('#test')肯定比document.getElementById('test')简洁；

③ 轻松实现动画、修改CSS等各种操作。

jQuery的理念'Write Less, Do More'，写更少的代码，完成更多的工作！

目前jQuery有1.x、2.x、3.x版本，区别是1.x兼容低版本浏览器，2.x和3.x不支持。选择哪个版本主要取决于是否想支持IE 6~8。

* $符号

$是著名的jQuery符号。jQuery把所有功能全部封装在一个全局变量jQuery中，而$也是一个合法的变量名，它是变量jQuery的别名：

|  |
| --- |
| console.log(window.jQuery); // function (e,t){return new Q.fn.init(e,t)} console.log(window.$); // function (e,t){return new Q.fn.init(e,t)} console.log($ === jQuery); // true console.log(**typeof** $); // 'function' |

$本质是一个函数，但函数也是对象，$除了可以直接调用，也可以有很多属性。

绝大多数时候都直接用$，但如果$这个变量不幸被占用了，而且还不能改，那只能让jQuery把$变量交出来，然后使用jQuery这个变量：

|  |
| --- |
| console.log($); // function (e,t){return new Q.fn.init(e,t)} jQuery.noConflict(); console.log($); // undefined console.log(jQuery); // function (e,t){return new Q.fn.init(e,t)} |

这种黑魔法的原理是jQuery在占用$之前，先在内部保存了原来的$，调用jQuery.noConflict()时会把原来保存的变量还原。

* 选择器

选择器是jQuery的核心。

用document.getElementById()、document.getElementsByTagName()等获取DOM结点太繁琐了。

在层级关系中，例如查找<table class="test">里面的所有<tr>，一层循环实际上是错的，因为<table>的标准写法是：<table><tbody><tr>...</tr></tbody></table>

jQuery的选择器可以快速定位到一个或多个DOM结点。

① 按ID查找

|  |
| --- |
| $(**function** () {  **let** $test = $('#test');// 查找<div id="test">, 返回jQuery对象, 类似数组  **let** $p=$('<p>hehe</p>'); // 创建jQuery对象  $p.css({color:'red',font:'24px/24px "Microsoft Yahei"'}); // 设置p标签的css  $test.append($p); // p标签作为#test的子结点  console.log($test); }); |

jQuery对象类似数组，每个元素都是一个引用了DOM结点的对象。

如果id为test的<div>存在，返回的jQuery对象：[div#test]

如果不存在，返回的jQuery对象：w.fn.init {}

jQuery的选择器不会返回undefined或者null，好处是不必进行判断：

if (div === undefined){}

jQuery对象和DOM对象之间可以互相转化：

|  |
| --- |
| **let** div\_dom = $test.get(0); // 假设存在div, 获取第1个DOM元素 console.log(div\_dom); // <div id="test">...</div> **let** $test1 = $(div\_dom); // 重新把DOM包装为jQuery对象 |

通常不需要获取DOM对象，直接使用jQuery对象更方便。如果拿到一个DOM对象，可以简单调用$()变成jQuery对象，以方便使用jQuery的API。

② 按tag查找

|  |
| --- |
| **let** $div = $('div'); // 返回所有div结点 console.log($div.length); // 页面div结点个数, 1 |

③ 按class查找

|  |
| --- |
| $(**function** () {  **let** $red = $('.red'); // 返回class包含red的所有结点  **let** $rg = $('.red.big'); // 返回class包含red和big的所有结点  $red.text('red');  $rg.text('red big'); });  .red {color: red;} .big {font: 24px/30px "Microsoft Yahei";}  <div class="red"></div> <div class="red big"></div> <div class="red">haha</div> |

④ 按属性查找

一个DOM结点除了id和class还可以有很多属性，很多时候按属性查找会非常方便，比如在一个表单中按属性来查找：

|  |
| --- |
| **let** $email = $('[name=email]'); // 找出<??? name="email"> **let** $pwd = $('[type=password]'); // 找出<??? type="password"> **let** $a = $('[items="A B"]'); // 找出<??? items="A B"> |

当属性的值包含空格等特殊字符时，需要用双引号括起来。

按属性查找还可以使用前缀查找或者后缀查找：

|  |
| --- |
| // 例如：name="icon-1", name="icon-2" **let** $icons = $('[name^=icon]'); // 找出所有name属性值以icon开头的DOM // 例如：name="startswith", name="endswith" **let** $names = $('[name$=with]'); // 找出所有name属性值以with结尾的DOM |

这个方法尤其适合通过class属性查找，且不受class包含多个名称的影响：

|  |
| --- |
| // 例如：class="icon-clock", class="abc icon-home" **let** icons = $('[class^="icon-"]'); // 找出所有class包含至少一个以icon-开头的DOM |

⑤ 组合查找

把上述简单选择器组合使用。如果查找$('[name=email]')，很可能把表单外的<div name="email">也找出来，但只希望查找<input>：

**let** $email\_input = $('input[name=email]'); // 只会找出<input name="email">

同样，根据tag和class来组合查找也很常见：

**let** $box = $('div.box'); // 找出<div class="box">...</div>

⑥ 多项选择器

把多个选择器用,组合起来使用：

|  |
| --- |
| **let** $a = $('p,div'); // 把<p>和<div>都选出来 **let** $b = $('p.red,p.green'); // 把<p class="red">和<p class="green">都选出来 |

注意：选出来的元素按照在HTML中出现顺序排列，而且不会有重复元素。例如：<p class="red green">不会被$('p.red,p.green')选择两次。

* 层级选择器 (Descendant Selector)

因为DOM结构就是层级结构，所以经常要根据层级关系进行选择。

如果两个DOM元素具有层级关系，就可以用$('ancestor descendant')选择，层级之间用空格隔开。

|  |
| --- |
| **let** $js1 = $('ul.lang li.js'); **let** $js2 = $('div.test li.js'); console.log($js1); // [li.js, prevObject: w.fn.init[1]] console.log($js2); // [li.js, prevObject: w.fn.init[1]]  <div class="test">  <ul class="lang">  <li class="js">JavaScript</li>  <li class="python">Python</li>  <li class="cpp">C++</li></ul></div> |

<div>和<ul>都是<li>的祖先节点，所以上面两种方式都可以选出<li>结点

层级选择器相比单个选择器好处在于缩小了选择范围，因为要先定位父结点，才能选择相应的子结点，避免了其他不相关的元素。

**let** $input = $('form[name=upload] input');把选择范围限定在name属性为upload的表单里。如果页面有很多表单，其他表单的<input>不会被选择。

* 子选择器 (Child Selector)

子选择器$('parent>child')类似层级选择器，但限定了层级关系必须是父子关系，就是child必须是parent的直属子结点。

|  |
| --- |
| **let** $js1 = $('ul.lang>li.js'); **let** $js2 = $('div.test>li.js'); console.log($js1); // [li.js, prevObject: w.fn.init[1]], length:1 console.log($js2); // [prevObject: w.fn.init[1]], length:0就当是[]吧... |

li的直接父结点的ul，所以上面第2种不能选择出结果。

* 过滤器 (Filter)

过滤器一般不单独使用，通常附加在选择器上，用于更精确地定位元素。

|  |
| --- |
| **let** $lis = $('ul.lang li'); // 选出ul.lang全部li结点 **let** $li1 = $('ul.lang li:first-child'); // 仅选出JavaScript **let** $li\_last = $('ul.lang li:last-child'); // 仅选出C++ **let** $li2 = $('ul.lang li:nth-child(2)'); // 选出第n个元素, n从1开始 **let** $even = $('ul.lang li:nth-child(even)'); // 选出序号为偶数的元素 **let** $odd = $('ul.lang li:nth-child(odd)'); // 选出序号为奇数的元素 |

* 表单相关选择器

① :input：可选择<input>，<textarea>，<select>和<button>；

② :file：可以选择<input type="file">，和input[type=file]一样；

③ :checkbox：可以选择复选框，和input[type=checkbox]一样；

④ :radio：可以选择单选框，和input[type=radio]一样；

⑤ :focus：可以选择当前输入焦点的元素，如把光标放到一个<input>上，用$('input:focus')就可以选出；

⑥ :checked：选择当前勾上的单选框和复选框，用这个选择器可以获得用户选择的项目，如$('input[type=radio]:checked')；

⑦ :enabled：可以选择正常输入的<input>、<select>等，也就是没有灰掉的输入；

⑧ :disabled：和:enabled正好相反，选择那些不能输入的。

此外，jQuery还有很多有用的选择器。例如选出可见的或隐藏的元素：

|  |
| --- |
| **let** $visible = $('div:visible'); // 所有可见的div **let** $hidden = $('div:hidden'); // 所有隐藏的div |

* 查找和过滤

通常情况下选择器可以直接定位到元素，但是当拿到一个jQuery对象后，还能以此对象为基准，进行查找和过滤。

① 查找

最常见的查找是在某个结点的所有子结点中查找，使用find()方法，它本身又接收一个任意的选择器。

|  |
| --- |
| **let** $ul = $('ul.lang'); // 获得<ul> **let** $dynamic = $ul.find('.dynamic'); // 获得动态语言JavaScript, Python, Scheme  **let** $swift = $ul.find('#swift'); // 获得Swift **let** $haskell = $ul.find('[name=haskell]'); // 获得Haskell  <ul class="lang">  <li class="js dynamic">JavaScript</li>  <li class="dynamic">Python</li>  <li id="swift">Swift</li>  <li class="dynamic">Scheme</li>  <li name="haskell">Haskell</li></ul> |

获取当前结点的父结点用parent()方法：

|  |
| --- |
| **let** $parent = $swift.parent(); // 获得Swift的上层节点<ul> **let** $a = $swift.parent('.red'); // 获得Swift的上层节点<ul>, 同时传入过滤条件; 如果ul不符合条件, 返回空jQuery对象 console.log($parent.length); // 1 console.log($a.length); // 0 |

对于位于同一层级的结点，可以通过prev()和next()方法获取前后结点

|  |
| --- |
| **let** $next = $swift.next(); // swift后一个li结点 console.log($next.text()); // Scheme **let** $next2 = $swift.next('[name=cpp]'); // swift后一个li结点且name属性为cpp console.log($next2.text()); // '', 什么也没有 **let** $prev = $swift.prev('.dynamic'); // swift前一个li结点且class为dynamic console.log($prev.text()); // Python |

② 过滤

和函数式编程的map、filter类似，jQuery对象也有类似的方法。

1) filter()方法可以过滤掉不符合选择器条件的结点：

|  |
| --- |
| **let** $langs = $ul.children('li'); // ul的所有li结点 console.log($langs.length); // 5 $dynamic = $langs.filter('.dynamic'); // 得到class为dynamic的li结点 $dynamic.addClass('red'); // 添加red类 |

filter()也可以传入一个函数，注意函数内部的this被绑定为DOM对象而不是jQuery对象：

|  |
| --- |
| **let** $a = $langs.filter(**function** () {  // this是DOM对象  **return this**.innerHTML.startsWith('S'); }); $a.each(**function** () {  console.log(**this**); // <li id="swift">Swift</li>, <li class="dynamic red">Scheme</li> }); |

说明上面给dynamic类添加red类，添加成功!

2) map()方法把一个jQuery对象包含的若干DOM节点转化为其他对象：

|  |
| --- |
| **let** $arr = $langs.map(**function** () {  **return this**.innerHTML; }).get(); // get()方法获取数组, 每个成员是字符串 console.log($arr); // ["JavaScript", "Python", "Swift", "Scheme", "Haskell"] |

此外，一个jQuery对象如果包含了不止一个DOM结点，first()、last()和slice()方法可以返回一个新的jQuery对象，把不需要的DOM结点去掉：

|  |
| --- |
| **let** $first = $langs.first(); // JavaScript，相当于$('ul.lang li:first-child') **let** $last = $langs.last(); // Haskell, 相当于$('ul.lang li:last-child') **let** $sub = $langs.slice(1, 3); // 获取[1,3)的结点, 参数和数组的slice()方法一致 $sub.each(**function** () {  console.log(**this**.innerHTML); // Python, Swift }); |

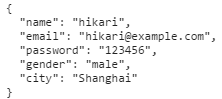
练习：

表单如下：

|  |
| --- |
| <form id="test-form" action="#0" onsubmit="**return false**;">  <p><label>Name: <input name="name"></label></p>  <p><label>Email: <input name="email"></label></p>  <p><label>Password: <input name="password" type="password"></label></p>  <p>Gender: <label><input name="gender" type="radio" value="male" checked> Male</label>  <label><input name="gender" type="radio" value="female"> Female</label></p>  <p><label>City: <select name="city">  <option value="Beijing" selected>Beijing</option>  <option value="Shanghai">Shanghai</option>  <option value="Chengdu">Chengdu</option>  <option value="Xiamen">Xiamen</option></select></label></p>  <p><button type="button" onclick="show()">Submit</button></p></form> |

输入值后，用jQuery获取表单的JSON字符串，key和value分别对应每个输入的name和相应的value，例如：{"name":"hikari","email":...}

|  |
| --- |
| **function** show() {  **let** dct = {};  $('#test-form :text, :password, input:checked, select').each(**function** () {  dct[**this**.name] = **this**.value;  });  **let** json = JSON.stringify(dct, **null**, ' ');  console.log(**typeof**(json) === 'string' ? json : 'json变量不是string!'); } |



**20180416**

* jQuery操作DOM

之前修改DOM的CSS、文本、设置HTML很麻烦，而且有的浏览器只有innerHTML，有的浏览器支持innerText，有了jQuery对象，不需要考虑浏览器差异了，全部统一操作。

① 修改Text和HTML

jQuery对象的text()和html()方法分别获取结点的文本和原始HTML文本；

如何设置Text或HTML? jQuery的API设计非常巧妙：如text()，无参数调用是获取文本，传入参数就变成设置文本，其他函数也类似。

|  |
| --- |
| **function** print(s) {  **let** $show = $('#show');  **let** $p = $('<p></p>');  $p.text(s);  $show.append($p); }  $(**function** () {  $('#btn').click(**function** () {  **let** $ul = $('#test');  **let** $book = $ul.find('li[name=book]');  print($book.text()); // Java & JavaScript  print($book.html()); // Java &amp; JavaScript  $book.text('JavaScript从入门到弃坑'); // 修改text  // jQuery对象可以包含0个或任意个DOM结点, 调用方法会作用每个DOM结点  $ul.find('li').addClass('red');  // 如果不存在id为not-exist的结点, 代码不会报错, 什么也没有发生  $('#not-exist').text('Hello');  }); });  ul { list-style: none; font: 20px/24px "Microsoft YaHei"} #show { color: #666666; font: 16px/20px "Microsoft YaHei"} .blue { color: blue;} .red { color: red;}  <ul id="test">  <li class="js">JavaScript</li>  <li name="book">Java &amp; JavaScript</li></ul><hr> <button id="btn">快按</button><hr> <div id="show"></div> |

jQuery对象的另一个好处是可以执行一个操作，作用在对应的一组DOM结点上。即使选择器没有返回任何DOM结点，调用jQuery对象的方法仍然不会报错，意味着免去了许多if语句。

② 修改CSS

css('name', 'value')或css({k1:v1,k2:v2,…})

css()方法作用于DOM结点的style属性，具有最高优先级。

|  |
| --- |
| $book.css('color', 'red').css('background-color', 'gold'); // 链式调用 print($book.css('color')); // rgb(255, 0, 0) $book.css('color', '') // 清除color属性 |

jQuery对象的所有方法都返回一个jQuery对象(新的对象或自身)，这样可以进行链式调用，非常方便。

如果要修改class属性，可以使用：hasClass()、addClass()、removeClass()等

③ 显示和隐藏DOM

要隐藏一个DOM，可以设置CSS的display属性为none。要显示就需要恢复原有的display属性，需要先记录原有的display属性(block、inline等)。

jQuery直接提供show()和hide()方法，不用关心它是如何实现的。

**注意**：隐藏DOM结点并未改变DOM树的结构，只影响DOM结点的显示，与删除结点不同。

④ 获取DOM信息

jQuery可以获取DOM的高宽等信息，无需针对不同浏览器编写特定代码：

|  |
| --- |
| // 浏览器可视窗口大小 print(`window: ${$(window).width()} x ${$(window).height()}`); // window: 1366 x 632 // HTML文档大小 print(`document: ${$(document).width()} x ${$(document).height()}`); // document: 1366 x 632 **let** $div = $('#test'); // 某个div的大小 print(`div: ${$div.width()} x ${$div.height()}`); // div: 200 x 200 $div.width(400); // 设置CSS属性 width: 400px, 是否生效要看CSS是否有效 $div.height('400px'); // 设置CSS属性 height: 400px, 是否生效要看CSS是否有效  #test { width: 200px; height: 200px; background-color: gold;}  <div id="test"></div> |

attr()和removeAttr()方法用于获取设置、移除DOM结点的属性

prop()和attr()类似，但是HTML5规定有一种属性在DOM结点中可以没有值，只有出现与不出现两种，例如：

<input id="gender" type="radio" name="gender" checked value="1">

等价于：

<input id="gender" type="radio" name="gender" checked="checked" value="1">

attr()和prop()对于属性checked处理有所不同，attr()返回'checked'，prop()返回true，

prop()返回值更合理一些。不过用is()方法判断更好：

|  |
| --- |
| **let** $gender = $('#gender'); print($gender.attr('checked')); // checked print($gender.prop('checked')); // true  print($gender.is(':checked')) // true |

类似的属性还有selected，处理时最好用is(':selected')。

⑤ 操作表单

对于表单元素，jQuery对象统一提供val()方法获取和设置对应的value属性。

* 修改DOM结构

直接使用浏览器提供的API对DOM结构进行修改，不但代码复杂，而且要针对浏览器写不同的代码。

jQuery专注于操作jQuery对象本身，底层的DOM操作由jQuery完成，修改DOM大大地简化。

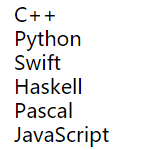
① 添加DOM

要添加新的DOM结点，除了通过jQuery的html()这种暴力方法外，还可以用append()和prepend()把DOM添加到尾部或头部，例如：

|  |
| --- |
| <div id="test">  <ul>  <li><span>JavaScript</span></li>  <li><span>Python</span></li>  <li><span>Swift</span></li></ul></div> |

向列表新增语言：

|  |
| --- |
| **let** $ul = $('#test').find('ul').eq(0); // 字符串形式 $ul.append('<li><span>Haskell</span></li>'); // 添加创建的DOM对象 **let** $ps = document.createElement('li'); $ps.innerHTML = '<span>Pascal</span>'; $ul.append($ps); // 添加jQuery对象, js从原来位置删除添加至末尾 **let** $js = $ul.find('li').first(); $ul.append($js); // 添加函数对象 $ul.prepend(**function** (index, html) {  **return** '<li><span>C++</span></li>'; }); |



要把新结点插入到指定位置，可以先定位到其他结点，然后用before()或after()方法插入到该结点前面或后面：

$js.before('<li><span>Java</span></li>'); 将Java结点插入到js结点前面。

② 删除结点

拿到jQuery对象后直接调用remove()方法就能删除DOM结点。如果jQuery对象包含若干DOM节点，实际上可以一次删除多个DOM结点：

|  |
| --- |
| **let** $li = $ul.find('li'); $li.remove(); // $ul的li结点全部删除 |

练习：如上面#test的HTML结构，除了列出的3种语言，再添加Pascal、Lua和Ruby，按字母顺序排序结点：

|  |
| --- |
| $(**function** () {  $('#test').find('>ul>li').css('color', 'red'); // 当前的li字体颜色设为红色  $('#btn').click(**function** () { // 按钮点击事件  **let** arr = ['Pascal', 'Lua', 'Ruby'];  **let** $ul = $('#test').find('ul').eq(0);  //添加li结点  **for** (**let** i **of** arr) {  $ul.append(`<li><span>${i}</span></li>`)  }  **let** $li = $ul.find('li');  // sort里面匿名函数x,y是DOM对象, 需要$(x)转为jQuery对象才能使用jQuery方法  $li.sort((x, y) => $(x).text() > $(y).text());  // 将原来所有li从$ul删除后添加排序后的  $ul.append($li);  test();  });  **function** test() { // 测试  **let** s = $('#test').find('>ul>li').map(**function** () {  **return** $(**this**).text();  }).get().join(','); // 将结点映射为字符串数组拼接成字符串  **if** (s === 'JavaScript,Lua,Pascal,Python,Ruby,Swift') {  print('测试通过!');  } **else** {  print('测试失败: ' + s);  }  } }); |



* jQuery事件

JavaScript在浏览器中以单线程模式运行，一旦页面上所有的JavaScript代码被执行完后，就只能依赖触发事件来执行JavaScript代码。

浏览器在接收到用户的鼠标或键盘输入后，会自动在对应的DOM结点上触发相应的事件。如果该结点绑定了对应的处理函数，该函数就会自动调用。

由于不同浏览器绑定事件的代码不太一样，用jQuery写代码，就屏蔽了不同浏览器的差异。

jQuery事件主要包括：

① 鼠标事件

1) click：鼠标单击时触发；

2) dblclick：鼠标双击时触发；

3) mouseenter：鼠标进入时触发；

4) mouseleave：鼠标移出时触发；

5) mousemove：鼠标在DOM内部移动时触发；

6) hover：鼠标进入和退出时触发两个函数，相当于mouseenter和mouseleave

② 键盘事件

键盘事件仅作用在当前焦点的DOM上，通常是<input>和<textarea>。

1) keydown：键盘按下时触发；

2) keyup：键盘松开时触发；

3) keypress：按一次键后触发。

③ 其他事件

1) focus：当DOM获得焦点时触发；

2) blur：当DOM失去焦点时触发；

3) change：当<input>、<select>或<textarea>的内容改变时触发；

4) submit：当<form>提交时触发；

5) ready：当页面被载入并且DOM树完成初始化后触发。

参考201712.pdf~29

如click事件：

|  |
| --- |
| **let** $btn = $('#btn'); $btn.on('click', **function** () {  alert('hello'); }); |

on()方法绑定事件需要传入事件名字和处理函数。

更简化的写法是直接调用click()方法：

|  |
| --- |
| $btn.click(**function** () {  alert('hello'); }); |

两者完全等价，通常用后面的写法。

* 事件参数

有些事件如mousemove和keypress，需要获取鼠标位置和按键值，否则监听这些事件就没意义了。所有事件都会传入Event对象作为参数，可以从Event对象上获取到更多的信息：

|  |
| --- |
| $('#testMouseMoveDiv').mousemove(**function** (e) {  $('#testMouseMoveSpan').text(`pageX = ${e.pageX}, pageY = ${e.pageY}`); });  #testMouseMoveDiv { display: block; width: 300px; height: 120px; border: 1px solid #ccc;}  <p>mousemove: <span id="testMouseMoveSpan"></span></p> <div id="testMouseMoveDiv">在此区域移动鼠标试试</div> |

* 取消绑定

一个已被绑定的事件可以解除绑定，通过off('click', function)实现：

|  |
| --- |
| **function** hello() {  alert('hello'); } **let** $btn = $('#btn'); $btn.click(hello); // 绑定点击事件 setTimeout(**function** () { // 3秒后取消绑定点击事件  $btn.off('click', hello) }, 3000); |

需要特别注意，off('click',匿名函数)是无效的：

|  |
| --- |
| $btn.click(**function** () {  alert('hello'); }); // off匿名函数无法取消绑定 setTimeout(**function** () { //  $btn.off('click', **function** () {  alert('hello');  }) }, 3000); |

因为两个匿名函数不是同一个函数。

为了实现移除效果，可以使用off('click')移除绑定的click事件的所有处理函数。

无参数调用off()移除已绑定的所有类型的事件处理函数。

* 事件触发条件

事件的触发总是由用户操作引发的。例如监控文本框的内容改动：

|  |
| --- |
| **let** $user = $('#user'); $user.change(**function** () {  print(`change : ${$user.val()}`) });  <label>用户名：<input type="text" id="user"></label> |

当用户在文本框中输入后失去焦点时，就会触发change事件。

但如果用JavaScript代码去改动文本框的值，将不会触发change事件：

|  |
| --- |
| setTimeout(**function** () {  $user.val('haha'); }, 1000); |

文本框的值1秒后变为haha，但是并没有触发change事件。

有时候希望用代码触发change事件，可以调用无参数的change()方法触发事件：

|  |
| --- |
| setTimeout(**function** () {  $user.val('haha');  $user.change(); // 触发change事件 }, 1000); |



$user.change()相当于$user.trigger('change')，是trigger()方法的简写。

手动触发一个事件是为了避免写两份一模一样的代码。

**20180417**

* 曜さん、お誕生日おめでとうございます！



ヨーソロー！

* 练习

Form表单：

|  |
| --- |
| <form id="test-form" action="">  <legend>请选择想要学习的编程语言：<!--居然有legend标签--></legend>  <fieldset><p><label class="selectAll"><input type="checkbox">  <span class="selectAll">全选</span>  <span class="deselectAll">全不选</span></label>  <a href="javascript:;" class="invertSelect">反选</a></p>  <p><label><input type="checkbox" name="lang" value="javascript"> JavaScript</label></p>  <p><label><input type="checkbox" name="lang" value="python"> Python</label></p>  <p><label><input type="checkbox" name="lang" value="ruby"> Ruby</label></p>  <p><label><input type="checkbox" name="lang" value="haskell"> Haskell</label></p>  <p><label><input type="checkbox" name="lang" value="scheme"> Scheme</label></p>  <p><button type="submit">Submit</button></p></fieldset></form>  \* { font: 16px/20px "Microsoft YaHei"} a { text-decoration: none;} |

绑定合适的事件处理函数，实现以下逻辑：

① 勾上全选时自动选中所有语言，并把全选变成全不选；

② 点击全不选时，自动不选中所有语言；

③ 点击反选时，自动把所有语言状态反转(选中变为未选，未选变为选中)；

④ 所有语言都手动勾上时，全选被自动勾上，并变为全不选；

⑤ 当手动去掉选中至少一种语言时，全不选自动被去掉选中，并变为全选。

|  |
| --- |
| $(**function** () {  **let** form = $('#test-form'),  langs = form.find('[name=lang]'), // 所有语言  selectAll = form.find('label.selectAll :checkbox'), //选择框  selectAllLabel = form.find('label.selectAll span.selectAll'), // 全选span文字部分  deselectAllLabel = form.find('label.selectAll span.deselectAll'), // 全不选span文字部分  invertSelect = form.find('a.invertSelect'); // 反选超链接  // 重置初始化状态  form.find('\*').show().off();  form.find(':checkbox').prop('checked', **false**).off();  deselectAllLabel.hide();  // 拦截form提交事件  form.off().submit(**function** (e) {  e.preventDefault();  alert(form.serialize());  });  // write your code below  // 全选选择框change事件  selectAll.change(**function** () {  // 将所有语言选择框与全选状态一致  langs.prop('checked', selectAll.is(':checked'));  langs.change(); // 全选和全不选会触发语言选择框状态变化  });  // 反选超链接的点击事件  invertSelect.click(**function** () {  langs.each(**function** () {  // 将每个语言选择状态取反  $(**this**).prop('checked', !$(**this**).is(':checked'));  });  langs.change();  });  // 任意一个语言选择框状态变化触发  langs.change(**function** () {  **let** isAllChecked = **true**;  langs.each(**function** () { // 与运算求所有语言是不是全选  isAllChecked &= $(**this**).is(':checked');  });  **if** (isAllChecked) { // 是全选, 则把全选隐藏、全不选显示、全选框选中  selectAllLabel.hide();  deselectAllLabel.show();  selectAll.prop('checked', **true**);  } **else** { // 反之相反  selectAllLabel.show();  deselectAllLabel.hide();  selectAll.prop('checked', **false**);  }  }); }); |

* jQuery动画

JavaScript实现动画原理非常简单：只需要以固定的时间间隔(如0.1秒)，每次把DOM元素CSS样式修改一点(如宽高各增加10%)，看起来就像动画了。

但是用JavaScript手动实现动画效果，需要编写非常复杂的代码。

使用jQuery实现动画，只需要一行代码！

* jQuery内置的动画样式：

① show()、hide()和toggle()

以无参数形式调用show()和hide()，会显示和隐藏DOM元素。传入一个时间参数，就变成了动画。时间以毫秒为单位，也可以是'slow'、'fast'等字符串

toggle()方法根据当前状态决定show()还是hide()

|  |
| --- |
| $(**function** () {  **let** $img = $('#img');  $('#btn1').click(**function** () {  $img.hide('slow');  });  $('#btn2').click(**function** () {  $img.show(3000);  });  $('#btn3').click(**function** () {  $img.toggle('ease');  }); });  .box { width: 500px; border: 1px solid #ccc; padding: 10px; margin: 10px 0;} .img\_con { height: 128px;} #img { width: 128px; height: 128px; background: #cccccc url('404.png') no-repeat center center;}  <div class="box">  <div>  <button id="btn1">hide</button>  <button id="btn2">show</button>  <button id="btn3">toggle</button></div><br>  <div class="img\_con"><div id="img"></div></div></div> |

// 这些东西不用过一段时间就忘了…

② slideUp()、slideDown()和slideToggle()

在垂直方向逐渐展开或收缩。slideUp()把可见的DOM元素收起来，效果跟拉上窗帘似的；slideDown()相反；slideToggle()则根据当前状态决定展开或收缩。

③ fadeIn()、fadeOut()和fadeToggle()

动画效果是淡入淡出，通过设置DOM元素的opacity属性实现，fadeToggle()则根据元素当前状态决定淡入或淡出。

* 自定义动画

如果内置动画效果不能满足要求，那就祭出最后大招：animate()，可以实现任意动画效果，需要传入的参数是DOM元素最终的CSS状态和时间，jQuery在时间段内不断调整CSS直到达到设定值：

|  |
| --- |
| $('#animate').click(**function** () {  $img.animate({  opacity: 0.25,  width: '256px',  height: '256px',  }, 3000); // 在3秒内CSS过渡到设定值 });  <button id="animate">animate</button> |

animate()还可以传入一个回调函数，当动画结束时该函数将被调用：

|  |
| --- |
| $('#animate').click(**function** () {  $img.animate({  opacity: 0.25,  width: '256px',  height: '256px',  }, 3000, **function** () {  console.log('动画已结束');  // 恢复至初始状态  $(**this**).css({opacity: '1', width: '128px', height: '128px'});  }); }); |

* 串行动画

jQuery的动画效果可以串行执行，通过delay()方法可以实现暂停，这样可以实现更复杂的动画效果，而代码却相当简单：

|  |
| --- |
| $('#animate').click(**function** () {  // 动画效果：先隐藏 -> slideDown -> 暂停 -> 放大 -> 暂停 -> 缩小  $img.hide()  .slideDown(3000) // 为什么slide的颜色不一样?  .delay(2000)  .animate({  width: '256px',  height: '256px',  }, 3000)  .delay(2000)  .animate({  width: '128px',  height: '128px',  }, 3000); }); |

因为动画需要执行一段时间，所以jQuery必须不断返回新的Promise对象才能后续执行操作。简单地把动画封装在函数中是不够的。

-------------------------------------------------------------------------------------------------------

jQuery动画的原理是改变CSS的值，如slideUp()将height从某个值逐渐变为0。但是很多不是block的DOM元素设置height不起作用，动画也就没有效果。

此外，jQuery也没有实现对background-color的动画效果，用animate()设置background-color没有效果。此时可以使用CSS3的transition实现动画效果。

**20180418**

* AJAX

JavaScript写AJAX，不同浏览器要写不同代码，且状态和错误处理很麻烦。

jQuery处理AJAX，不需要考虑浏览器问题，代码大大简化。

① $.ajax()

jQuery的全局对象jQuery(也就是$)绑定了ajax(url, settings)函数，可以处理AJAX请求。ajax()需要接收一个URL和一个可选的settings对象：

1) async：是否异步执行AJAX请求，默认为true，千万**不要**指定为false；

2) method：发送的Method，默认为'GET'，可指定为'POST'、'PUT'等；

3) contentType：发送POST请求的格式，默认值为'application/x-www-form-urlencoded; charset=UTF-8'，也可以指定为text/plain、application/json；

4) data：发送的数据，可以是字符串、数组或object。如果是GET请求，data将被转换成query附加到URL上；如果是POST请求，根据contentType把data序列化成合适的格式；

5) headers：发送的额外的HTTP头，必须是一个object；

6) dataType：接收的数据格式，可以指定为'html'、'xml'、'json'、'text'等，默认情况下根据响应的Content-Type猜测。

由于ajax不能跨域请求，又要在hikari\_web\_app里演示：

www/templates/test.html：

|  |
| --- |
| <script src="https://cdn.bootcss.com/jquery/3.3.1/jquery.min.js"></script> <script src="/static/js/hikari.js"></script>  $(**function** () {  // jQuery的jqXHR对象类似于Promise对象, 可以用链式写法处理各种回调  **let** jqxhr = $.ajax('/api/test', {  dataType: 'json'  }).done(**function** (data) {  print(`成功收到数据：${JSON.stringify(data)}`);  }).fail(**function** (xhr, status) {  print(`失败：${xhr.status}', 原因：${status}`);  }).always(**function** () {  print('请求完成: 无论成功或失败都会调用');  }); });  \* { font: 16px/20px "Microsoft Yahei";} |

test视图和test\_api视图见P45

hikari.js瞎写的print()函数：

|  |
| --- |
| **function** print(s) {  **let** $show = $('#show');  **if** ($show.length === 0) {  $show = $('<div id="show"></div>');  $('body').append($show);  }  **let** $p = $('<p></p>');  $p.text(s);  $show.append($p); } |



// 华为畅玩卡死了，美团点外卖用支付宝支付卡了10min…

② $.get()

对常用的AJAX操作，jQuery提供了一些辅助方法。由于GET请求最常见，所以jQuery提供了get()方法：

|  |
| --- |
| **let** jqxhr = $.get('/path/to/resource', {  name: 'hikari',  age: 25 }); |

第二个参数如果是object，jQuery自动转为query string加到URL后面，

实际URL是：/path/to/resource?name=hikari&age=25

可以不用关心URL编码，构造一个query string。

③ $.post()

与get()类似，但第二个参数默认被序列化为application/x-www-form-urlencoded：

|  |
| --- |
| **let** jqxhr = $.post('/path/to/resource', {  name: 'hikari',  age: 25 }); |

实际构造的数据name=hikari&age=25作为POST的body(请求体)被发送。

④ $.getJSON()

由于JSON越来越普遍，jQuery也提供了getJSON()方法通过GET快速获取一个JSON对象：

|  |
| --- |
| **let** jqxhr = $.getJSON('/path/to/resource', {  name: 'hikari',  age: 25 }).done(**function** (data) {  // 请求完成执行的回调函数? data是解析后的JSON对象 }); |

* 安全限制

jQuery的AJAX完全封装的是JavaScript的AJAX操作，所以它的安全限制和用JavaScript写AJAX完全一样。

如果需要使用JSONP，可以在ajax()中设置jsonp: 'callback'，让jQuery实现JSONP跨域加载数据。

* jQuery扩展

jQuery内置方法不可能满足所有的需求。比如想要高亮某些DOM元素：

|  |
| --- |
| // 为什么不直接用$('.hilite').css({...})? $('span.hilite').css({backgroundColor: '#fffceb', color: '#d85030'}); $('p a.hilite').css({backgroundColor: '#fffceb', color: '#d85030'}); |

复制代码是不好的习惯，修改时就麻烦了。

可以扩展jQuery实现自定义方法，也称为编写jQuery插件：

|  |
| --- |
| $('span.hilite').highlight(); $('p a.hilite').highlight(); |

将来如果要修改高亮的逻辑，只需修改一处扩展代码。

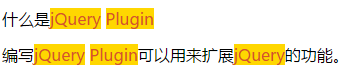
给jQuery对象绑定一个新方法是通过扩展$.fn对象实现的。

|  |
| --- |
| $(**function** () {  $.fn.highlight1 = **function** () {  // this已绑定为当前jQuery对象  **this**.css({backgroundColor: '#fffceb', color: '#d85030'});  **return this**; // 为了支持链式调用  };  $('#btn').click(**function** () {  $('#test').find('span').highlight1().hide().fadeIn('fast');  }); });  \* { font: 16px/20px "Microsoft YaHei";}  <div id="test">  <p>什么是<span>jQuery</span></p>  <p><span>jQuery</span>是目前最流行的<span>JavaScript</span>库。</p></div><hr> <button id="btn">快按</button> |



如果希望高亮的颜色能自己指定，可以给此方法加个参数：

|  |
| --- |
| $.fn.highlight2 = **function** (options) {  // 巧妙运用短路逻辑运算设定默认值  **let** bgcolor = options && options.backgroundColor || '#fffceb';  **let** color = options && options.color || '#d85030';  **this**.css({backgroundColor: bgcolor, color: color});  **return this**; };  $('#test2').find('span').highlight2({backgroundColor: 'gold'});  <div id="test2">  <p>什么是<span>jQuery</span> <span>Plugin</span></p>  <p>编写<span>jQuery</span> <span>Plugin</span>可以用来扩展<span>jQuery</span>的功能。</p></div> |



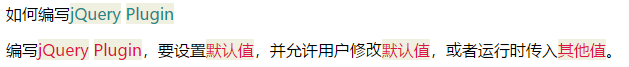
另一种方法是使用jQuery提供的辅助方法$.extend(target, obj1, obj2, ...)，它把多个object对象的属性合并到第一个target对象中，遇到同名属性，总是使用靠后对象的值，也就是越往后优先级越高：

|  |
| --- |
| // 把默认值和用户传入的options合并到对象{}中 **let** opts = $.extend({}, {  backgroundColor: '##fffceb',  color: 'd85030' }, options); |

但是这样每次都要传入参数也很麻烦，需要实现能够修改默认值，以后每次都可以调用无参数的函数highlight()。

默认值放全局变量不合适，最好是$.fn.highlight这个函数对象本身上。

|  |
| --- |
| $.fn.highlight = **function** (options) {  // 合并默认值和用户设定值  **let** opts = $.extend({}, $.fn.highlight.defaults, options);  **this**.css({backgroundColor: opts.backgroundColor, color: opts.color});  **return this**; }; // 设定默认值 $.fn.highlight.defaults = {backgroundColor: '#fff8de', color: '#d85030'}; // 修改默认值 $.fn.highlight.defaults.color = '#228888'; $.fn.highlight.defaults.backgroundColor = '#f0f0e0';  **let** $test = $('#test-highlight'); $test.find('p:first-child span').highlight(); $test.find('p:last-child span').highlight({color: '#dd1144'});  <div id="test-highlight">  <p>如何编写<span>jQuery</span> <span>Plugin</span></p>  <p>编写<span>jQuery</span> <span>Plugin</span>，要设置<span>默认值</span>，并允许用户修改<span>默认值</span>，或者运行时传入<span>其他值</span>。</p></div> |



* 编写jQuery插件的原则

① 给$.fn对象绑定函数，实现插件的代码逻辑；

② 插件函数最后要return this;以支持链式调用；

③ 插件函数如果有默认值，绑定在$.fn.<pluginName>.defaults对象上；

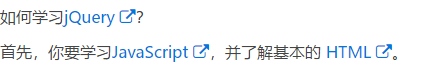
④ 用户在调用时可以传值以便自定义默认值。

* 针对特定元素的扩展

jQuery对象的有些方法只能作用在特定DOM元素上，比如submit()方法只能针对form。用jQuery选择器的filter()方法可以实现只针对某些类型DOM元素的扩展。

示例：编写一个external扩展，给所有指向外链的超链接加上跳转提示

|  |
| --- |
| <link rel="stylesheet" href="uikit-2.25.0/css/uikit.min.css">  $.fn.external = **function** () {  // return返回的each()返回结果, 支持链式调用  **return this**.filter('a').each(**function** () {  // each()内部的回调函数的this绑定为DOM本身  **let** url = $(**this**).attr('href');   // 如果是http或https开头说明是外域, 弹出确认框  **if** (url && (url.startsWith('http://') || url.startsWith('https://'))) {  $(**this**).attr('href', '#0')  .removeAttr('target') // 移去target属性  .append('<i class="uk-icon-external-link"></i>') // uikit外链图标  .click(**function** () {  **if** (confirm(`你确定要前往 ${url} ？`)) {  window.open(url);  }  });  }  }); };  $('#test-external').find('a').external(); // 把此句放在一个button的click事件里  a { text-decoration: none;}  <div id="test-external">  <p>如何学习<a href="http://jquery.com" target="\_blank">jQuery </a>？</p>  <p>首先，你要学习<a href="http://www.w3school.com.cn/js/">JavaScript </a>，并了解基本的<a href="https://developer.mozilla.org/en-US/docs/Web/HTML" target="\_blank">HTML </a>。</p></div> |



**20180419**

* 真姫ちゃん、お誕生日おめでとう！



なにそれ？意味わかんない！

* 错误处理

在执行JavaScript代码时，有些情况下会发生错误。

① 程序逻辑错误，导致代码执行异常。例如：

|  |
| --- |
| **let** arr = **null**; console.log(arr.length); // Uncaught TypeError: Cannot read property 'length' of null |

对于这种错误，要修复程序。

② 执行过程中，程序可能遇到无法预测的异常情况而报错。如网络连接中断、读取不存在的文件、没有操作权限等。

对于这种错误，需要处理它，并可能要给用户反馈。

-------------------------------------------------------------------------------------------------------

错误处理是程序设计必须要考虑的问题。

对于贴近系统底层的C语言，错误是通过错误码返回的：

|  |
| --- |
| int fd = open("/path/to/file", O\_RDONLY);  **if** (fd == -1) {  printf("Error when open file!");  } **else** {  *// TODO*  } |

通过错误码返回错误，需要约定正确的返回值和错误的返回值是什么。上面的open()函数约定返回-1表示错误。

显然，用错误码表示错误在编写程序时十分不便。

高级语言通常都提供了更抽象的错误处理逻辑try ... catch ... finally …

JavaScript也是类似，// pass…

注意：catch和finally可以不必都出现，但至少有一个出现。

* 错误类型

JavaScript有一个标准的Error对象表示错误，还有从Error派生的TypeError、ReferenceError等错误对象。

|  |
| --- |
| **try** {  console.log(**null**.length); } **catch** (e) {  **if** (e **instanceof** TypeError) {  alert('Type error!');  } **else if** (e **instanceof** Error) {  alert(e.message);  } **else** {  alert('Error: ' + e);  } } |

* 抛出错误

使用throw语句抛出错误。

|  |
| --- |
| **let** s = prompt('请输入一个数字'); **let** n = parseInt(s); **try** {  **if** (isNaN(n) || isNaN(s)) {  // pycharm不建议在try里面抛出异常  **throw new** Error('输入不是数字！');  }  **let** ret = n \* n;  print(`${n} \* ${n} = ${ret}`) } **catch** (e) {  print('出错了：' + e); } |

实际上，JavaScript允许抛出任意对象，包括数字、字符串，但最好抛出一个Error对象。

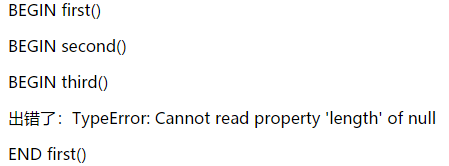
当用catch捕获错误时，一定要编写错误处理语句，哪怕仅仅把错误打印出来，也不要什么也不干，否则就不知道程序执行过程中有没有发生错误。

* 错误传递

如果在一个函数内部发生错误，而自身没有捕获，错误就会抛到外层调用函数；如果外层函数也没有捕获，错误会一直沿着函数调用链向上抛出，直到被JavaScript引擎捕获，代码终止执行。

所以不必在每个函数内部捕获错误，只需在合适的地方统一捕获：

|  |
| --- |
| **function** first(s) {  print('BEGIN first()');  **try** {  second(s);  } **catch** (e) {  print('出错了：' + e);  }  print('END first()'); } **function** second(s) {  print('BEGIN second()');  third(s);  print('END second()'); } **function** third(s) {  print('BEGIN third()');  print('length = ' + s.length);  print('END third()'); } first(**null**); |



当third()传入参数null时，代码会报错，错误向上抛给调用者second()，它没有try语句捕获错误，错误继续向上抛给调用者first()，first()函数有try语句，所以错误在first()函数被处理了。

* 异步错误处理

牢记JavaScript引擎是一个事件驱动的执行引擎，代码总以单线程执行，而回调函数的执行需要等到下一个满足条件的事件出现后，才会被执行。

如setTimeout()函数可以传入回调函数，并在指定若干毫秒后执行：

|  |
| --- |
| **function** show() {  **let** now = **new** Date();  print(now); } setTimeout(show, 1000); print('over!'); |

先打印over，过1s再打印时间

如果回调函数内部发生错误，用try包裹setTimeout()是无效的：

|  |
| --- |
| **function** show() {  **throw new** Error(); // Uncaught Error } **try** {  setTimeout(show, 1000);  print('over!'); } **catch** (e) {  print('error!') } |

打印over，然后console显示Uncaught Error

原因是调用setTimeout()函数时，传入的回调函数并未立刻执行。先打印over，此时并没有错误发生；1秒后，执行show()时才发生错误，但此时除了在show()函数内部捕获错误外，外层代码无法捕获错误。

所以，异步无法在调用时捕获错误，原因是在捕获的当时，回调函数并未执行。

类似在绑定事件代码处，无法捕获事件处理函数的错误，需要把try语句写在处理函数里面。

**20180420**

* underscore

JavaScript是函数式编程语言，支持高阶函数和闭包。函数式编程非常强大，可以写出非常简洁的代码。例如Array的map()和filter()方法：

|  |
| --- |
| **let** arr = [2, 3, 4, 5, 6]; **let** arr2 = arr.map(x => x \*\* 3); **let** arr3 = arr.filter(x => x % 3 === 0); print(arr2); // [8, 27, 64, 125, 216] print(arr3); // [3, 6] |

然而Object和低版本的浏览器例如IE6～8没有这些方法。怎么使用这些方法？

① 自己将这些方法添加到Array.prototype中，然后给Object.prototype也加上mapObject()等类似的方法。

② 直接用成熟可靠的第三方开源库，使用统一的函数实现map()、filter()操作。

此处选择第三方库underscore。

jQuery统一了不同浏览器之间的DOM操作差异，可以简单地对DOM进行操作；underscore则提供了一套完善的函数式编程的接口，更方便实现函数式编程。

jQuery在加载时，会把自身绑定到唯一的全局变量$上；而underscore将自身绑定到唯一的全局变量\_上，这也是其名叫underscore的原因。

用underscore实现map()操作：

|  |
| --- |
| <script src="https://cdn.bootcss.com/underscore.js/1.8.3/underscore-min.js"></script>  **let** arr = \_.map([11, 22, 33], x => x \*\* 2 + x + 1); print(arr); // [133, 507, 1123] |

看上去比直接用Array.map()麻烦，但是underscore的map()还可以作用于Object：

|  |
| --- |
| **let** arr = \_.map({a: 1, b: 2, c: 3}, (v, k) => `${k}=${v}`); print(arr); // ["a=1", "b=2", "c=3"] |

* underscore的Collections

underscore为集合类对象提供了一致的接口。集合类是指Array和Object，暂不支持Map和Set。

① \_.map()和\_.filter()

\_.map()和\_.filter()用法与Array的类似，但可以作用于Object。当作用于Object时，传入函数为function (value, key) // key和value倒过来?

|  |
| --- |
| **let** hikari = {name: 'hikari', age: 25, job: '搬砖'}; // map()返回结果是Array **let** arr = \_.map(hikari, (v, k) => [k, v]); print(arr); // [["name", "hikari"], ["age", 25], ["job", "搬砖"]] // mapObject返回Object **let** obj = \_.mapObject(hikari, (v, k) => v); print(obj); // {"name":"hikari","age":25,"job":"搬砖"} |

map()返回结果是Array? 不合理啊! mapObject()貌似就是返回Object。

② \_.every()和\_.some()

\_.every()：集合所有元素都满足条件时返回true；

\_.some()：集合至少一个元素满足条件时返回true。

|  |
| --- |
| **let** arr = [2, -4, 6, -8, 10]; // arr全部元素>0? print(\_.every(arr, x => x > 0)); // false // arr至少一个元素>0? print(\_.some(arr, x => x > 0)); // true |

当集合是Object时，可以同时获得value和key：

|  |
| --- |
| **let** p = /^[a-z]+$/; // 判断是否key和value全部小写, 或部分key和value小写 **let** r1 = \_.every(hikari, (v, k) => p.test(v) && p.test(k)); // false **let** r2 = \_.some(hikari, (v, k) => p.test(v) && p.test(k)); // true print(`every key-value are lowercase: ${r1}\nsome key-value are lowercase: ${r2}`); |

③ \_.max()和\_.min()

返回集合中最大和最小的元素。

|  |
| --- |
| **let** arr = [3, -5, 7, -9]; **let** a = \_.max(arr); **let** b = \_.min(arr); print(`max=${a}, min=${b}`); // max=7, min=-9 // 空集合会返回-Infinity和Infinity, 所以要先判断集合不为空 print(\_.max([])); // -Infinity print(\_.min([])); // Infinity |

注意：如果集合是Object，\_.max()和\_.min()只作用于value而忽略key

|  |
| --- |
| **let** obj = {a: 3, b: -5, c: 7, d: -9}; print(\_.max(obj)); // 7 |

④ \_.groupBy()

\_.groupBy()把集合的元素按照key归类，由每个元素传入函数的返回值确定：

|  |
| --- |
| **let** arr = [53, 66, 81, 61, 28, 43, 13, 20, 87, 74]; // 按照元素除4余数将元素分组 **let** group = \_.groupBy(arr, x => x % 4); print(group); // {"0":[28,20],"1":[53,81,61,13],"2":[66,74],"3":[43,87]} |

⑤ \_.shuffle()和\_.sample()

\_.shuffle()用洗牌算法随机打乱一个集合；\_.sample()随机选择一个或多个元素。

|  |
| --- |
| **let** arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]; // 将集合打乱 print(\_.shuffle(arr)); // [1, 7, 5, 8, 2, 9, 6, 4, 3] // 从集合随机选取一个元素 print(\_.sample(arr)); // 7 // 从集合随机选取3个元素 print(\_.sample(arr, 3)); // [8, 6, 4] |

Collections更多完整的函数参考[underscore文档](http://underscorejs.org/#collections)

* underscore的Arrays

underscore为Array提供了许多工具类方法，可以更方便操作Array。

① \_.first()和\_.last()

分别取Array第一个和最后一个元素。

|  |
| --- |
| **let** arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]; // 空数组first和last都是undefined print(\_.first(arr)); // 1 print(\_.last(arr)); // 9 |

② \_.flatten()

\_.flatten()将嵌套Array变成一维Array：

|  |
| --- |
| arr = [1, [2, [3, [4], [5, [6]]]]]; print(\_.flatten(arr)); // [1, 2, 3, 4, 5, 6] |

③ \_.zip()和\_.unzip()

\_.zip()把两个或多个数组所有元素按索引对齐并按索引合并成新数组。\_.unzip()则是相反。

|  |
| --- |
| **let** names = ['maki', 'rin', 'nozomi']; **let** scores = [99, 56, 78]; **let** sheet = \_.zip(names, scores); print(sheet); // [["maki", 99], ["rin", 56], ["nozomi", 78]] print(\_.unzip(sheet)); // [["maki", "rin", "nozomi"], [99, 56, 78]] |

④ \_.object()

与其用zip()变成二维数组，不如直接变为Object，用\_.object()函数：

|  |
| --- |
| **let** names = ['maki', 'rin', 'nozomi']; **let** scores = [99, 56, 78]; // 注意：\_.object()是一个函数, 不是JavaScript的Object对象 **let** sheet = \_.object(names, scores); print(sheet); // {"maki":99,"rin":56,"nozomi":78} |

⑤ \_.range()

快速生成一个序列，从此不再需要用for循环实现了：

|  |
| --- |
| // 用法与Python的range类似 **let** arr = \_.range(2, 20, 3); print(arr); // [2, 5, 8, 11, 14, 17] |

Arrays完整的函数参考[underscore文档](http://underscorejs.org/#arrays)

练习：根据underscore官方文档，用\_.uniq()对数组不区分大小写去重：

|  |
| --- |
| **let** arr = ['Apple', 'orange', 'banana', 'ORANGE', 'apple', 'PEAR']; /\* \_.uniq(array, [isSorted], [iteratee]) isSorted如果事先知道数组已经排序可以设为true, 默认false iteratee迭代函数, 每个元素传入函数, 根据返回值判断是不是相同 \*/ **let** result = \_.uniq(arr, x => x.toLowerCase()); print(result); // ["Apple", "orange", "banana", "PEAR"] |

* underscore的Functions

① \_.bind()

|  |
| --- |
| **let** s = ' nishikino maki '; print(s.trim()); // nishikino maki **let** f = s.trim; // f(); // Uncaught TypeError: String.prototype.trim called on null or undefined // 直接调用f()传入的this指针是undefined, 调用会报错 // 调用call()并传入s对象作为this print(f.call(s)); // nishikino maki |

这样太麻烦，不如直接用s.trim()。\_.bind()可以将s对象直接绑定在f()的this指针上，以后调用f()就可以正常调用了：

|  |
| --- |
| **let** s = ' nishikino maki '; **let** f = \_.bind(s.trim, s); print(f()); // nishikino maki |

// 这样做的意义何在?

② \_.partial()

为一个函数创建偏函数。

|  |
| --- |
| // 2的n次方, 固定第1个参数为2 **let** pow2n = \_.partial(Math.pow, 2); // x的立方, 固定第2个参数为3, \_是占位符 **let** cube = \_.partial(Math.pow, \_, 3); print(`2的16次方是${pow2n(16)}`); // 2的16次方是65536 print(`23的立方是${cube(23)}`); // 23的立方是12167 |

③ \_.memoize()

如果一个函数调用开销很大，希望能把结果缓存下来以便后续调用时直接获得结果。如计算阶乘就比较耗时：

|  |
| --- |
| **function** factorial(n) {  print(`start calculate ${n}! ...`);  **let** s = 1;  **for** (**let** i = n; i > 1; i--) {  s \*= i;  }  print(`${n}! = ${s}`);  **return** s; } // memoize()可以自动缓存函数计算结果 **let** f = \_.memoize(factorial); print(f(10)); // 3628800, 函数执行时会打印两句废话 print(f(10)); // 3628800, 有结果但是函数没有执行 |

对于相同的调用，如连续两次调用f(10)，第二次调用并没有计算，而是直接返回上次计算后缓存的结果。不过计算f(8)时仍然会重新计算。

可以将factorial()写成递归形式，调用f(10)时，f(1)~f(10)全部缓存；下一次调用f(8)就不会计算直接返回结果。

|  |
| --- |
| **function** factorial(n) {  print(`start calculate ${n}! ...`);  **if** (n < 2) {  **return** 1;  }  // 递归调用缓存函数  **return** n \* f(n - 1); } **let** f = \_.memoize(factorial); print(f(10)); // 3628800, 打印了10句start... print(f(8)); // 40320, 直接得到缓存结果, 没有计算 |

④ \_.once()

保证函数执行且仅执行一次。如果用户点击页面上两个按钮的任何一个都可以执行register()，可以用once()保证无论用户点击多少次，函数仅调用一次：

|  |
| --- |
| **let** $btn = $('#btn1,#btn2'); **let** register = \_.once(**function** () {  alert('注册成功！'); // 此匿名函数只执行1次 }); $btn.click(**function** () {  register(); });  <button id="btn1">按钮1</button> <button id="btn2">按钮2</button> |

⑤ \_.delay()

让一个函数延迟执行，效果和setTimeout()一样，但是代码明显简单了：

|  |
| --- |
| // 2秒后调用alert() \_.delay(alert, 2000); **let** log = \_.bind(console.log, console); // 回调函数有参数, 加在后面 \_.delay(log, 2000, 'Hello,', 'world!'); |

为什么要bind? 不用bind也可以?

|  |
| --- |
| **let** log = console.log; \_.delay(log, 2000, 'Hello,', 'world!'); |

更多完整的函数参考[underscore文档](http://underscorejs.org/#functions)

**20180423**

* underscore的Objects

① \_.keys()和\_.allKeys()

\_.keys()返回一个object自身所有的key，但不包含从原型链继承的；

\_.allKeys()除了object自身的key，还包含从原型链继承的。

|  |
| --- |
| **function** Student(name, age) {  **this**.name = name;  **this**.age = age; } Student.prototype.school = '皇家幼稚园'; **let** hikari = **new** Student('hikari', 25); // 自身所有key, 不包含从原型链继承的key print(\_.keys(hikari)); // ["name", "age"] // 自身所有key+从原型链继承的key print(\_.allKeys(hikari)); // ["name", "age", "school"] |

② \_.values()：返回object自身但不包含原型链继承的所有值：

|  |
| --- |
| print(\_.values(hikari)); // ["hikari", 25] |

注意：没有\_.allValues()，原因不明。

③ \_.mapObject()：针对object的map版本：

|  |
| --- |
| **let** obj = {a: 1, b: 2, c: 3}; // 匿名函数参数还是很奇葩的value在前, key在后 print(\_.mapObject(obj, (v, k) => v \* v \* (v + 1))); // {"a":2,"b":12,"c":36} |

④ \_.invert()：把object的每个key变成value，value变成key：

|  |
| --- |
| **let** obj = {rin: 55, maki: 99, nozomi: 78, umi: 78}; // 如果多个key对应一个value, 翻转后一个value对应最后一个key? print(\_.invert(obj)); // {"55":"rin","78":"umi","99":"maki"} |

⑤ \_.extend()和\_.extendOwn()

\_.extend()把多个object的key-value合并到第一个object并返回；如果有相同的key，后面object的value将覆盖前面object的value。

\_.extendOwn()获取属性时忽略从原型链继承下来的属性。

|  |
| --- |
| \_.extend(hikari,  {age: 15, city: 'Nanjing'},  {name: 'hikari', skill: ['JavaScript', 'Python']} ); print(hikari); // {"name":"hikari","age":15,"city":"Nanjing","skill":["JavaScript","Python"]} |

⑥ \_.clone()：把原有对象所有属性都复制到新对象中：

\_.clone()是浅拷贝，即两个对象相同key所引用的value是同一对象：

|  |
| --- |
| **let** copy = \_.clone(hikari); // 浅拷贝 print(copy); // {"name":"hikari","age":15,"city":"Nanjing","skill":["JavaScript","Python"],"school":  "皇家幼稚园"}, 原型链继承属性也复制了? print(copy.skill===hikari.skill); // true copy.skill.push('Java'); print(hikari.skill); // ["JavaScript", "Python", "Java"] |

⑦ \_.isEqual()：对两个object进行深度比较，如果内容完全相同返回true：

|  |
| --- |
| **let** obj1 = {name: 'hikari', skills: {Java: 70, JavaScript: 80}}; **let** obj2 = {name: 'hikari', skills: {JavaScript: 80, Java: 70}}; print(obj1 === obj2); // false print(\_.isEqual(obj1, obj2)); // true  **let** arr1 = [1, 2, 3]; **let** arr2 = [1, 2, 3]; print(arr1 === arr2); // false print(\_.isEqual(arr1, arr2)); // true |

更多完整的函数参考[underscore文档](http://underscorejs.org/#objects)。

* Chaining

jQuery支持链式调用，underscore也支持链式调用。

underscore提供了chain()函数，把对象包装成能链式调用。

|  |
| --- |
| // 普通写法, 每个元素开方, 保留奇数 **let** arr = \_.filter(\_.map([1, 4, 9, 16, 25], Math.sqrt), x => x % 2 === 1); print(arr); // [1, 3, 5] // 链式调用 arr = \_.chain([1, 4, 9, 16, 25])  .map(Math.sqrt)  .filter(x => x % 2 === 1)  .value(); // 每一步返回的都是包装对象, 最后结果需要调用value()获得最终结果 print(arr); // [1, 3, 5] |

* Node.js

① 浏览器大战

众所周知，在Netscape设计出JavaScript后短短几个月，JavaScript事实上已经是前端开发的唯一标准。后来，微软通过IE击败了Netscape后一统桌面；结果几年时间，浏览器毫无进步(2001年推出的古老IE 6至今仍然有人在使用!)。微软认为IE6已经非常完善而解散了IE6开发团队! 而Google却认为支持现代Web应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行JavaScript的引擎性能还可提升10倍。

先是Mozilla借助已壮烈牺牲的Netscape遗产在2002年推出了Firefox浏览器；接着Apple于2003年在开源的KHTML浏览器基础上推出了WebKit内核的Safari浏览器，不过仅限于Mac平台。随后Google也看中了WebKit内核，基于WebKit内核推出了Chrome浏览器。

Chrome浏览器是跨Windows和Mac平台的，并且Google认为要运行现代Web应用，浏览器必须有一个性能非常强劲的JavaScript引擎，于是Google开发了一个名叫V8的高性能JavaScript引擎，以BSD许可证开源。

现代浏览器大战让微软的IE浏览器远远落后了，因为解散了最有经验、战斗力最强的浏览器团队，回过头再追赶却发现支持HTML5的WebKit已经成为手机端的标准了，IE浏览器从此与主流移动端设备绝缘。

② Node.js

有个叫Ryan Dahl的歪果仁，工作是用C/C++写高性能Web服务。对于高性能，异步IO、事件驱动是基本原则，但是用C/C++写太痛苦了。于是这位仁兄开始设想用高级语言开发Web服务。他发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，就再也懒得写异步IO了。最终，Ryan选择了JavaScript。因为JavaScript是单线程执行，根本不能进行同步IO操作。所以JavaScript的这一缺陷导致了只能使用异步IO。

这位仁兄曾考虑自己写一个引擎，不过明智地放弃了，因为V8就是开源的JavaScript引擎。让Google去优化V8，自己改造一下拿来用，还不用付钱，这买卖很划算。

2009年Ryan正式推出了基于JavaScript语言和V8引擎的开源Web服务器项目，命名为Node.js。Node第一次把JavaScript带入到后端服务器开发，加上世界上已经有无数的JavaScript开发人员，所以Node一下子就火了。

在Node上运行的JavaScript相比其他后端开发语言，最大的优势是借助JavaScript天生的事件驱动机制和V8高性能引擎，容易编写高性能Web服务。其次，在Node环境下，通过模块化的JavaScript代码，加上函数式编程，无需考虑浏览器兼容性问题，直接使用最新的ECMAScript标准，可以完全满足工程上的需求。

③ io.js

Node.js是开源项目，虽然由社区推动，但幕后一直由Joyent公司资助。由于一些开发者对Joyent公司策略不满，于2014年从Node.js项目fork出了io.js项目，决定单独发展，但两者实际上是兼容的。然而分家后没多久，Joyent公司表示要和解，io.js项目又回归Node.js。

结果是要添加新特性先加到io.js，如果大家都很满意，就把新特性加入Node.js；于是io.js变成尝鲜版，而Node.js相当于线上稳定版。

* 安装Node.js

Node.js是在后端运行JavaScript代码，必须先安装Node环境。

目前Node.js的最新版本是9.11.1，推荐安装[8.11.1 LTS](https://nodejs.org/dist/v8.11.1/node-v8.11.1-x64.msi)。

在Windows安装必须选择全部组件，包括勾选Add to Path。

安装完成后，命令行输入node -v，如果安装正常，可以看到版本号。

命令行输入node，将进入Node.js交互环境。在交互环境可以输入任意JavaScript语句，回车后将得到输出结果。

连按两次Ctrl+C退出Node.js环境。

* npm

npm是Node.js的包管理工具(package manager)。

相当于Python的pip? 直接下载其他人开源的模块。

npm还可以根据依赖关系，把所有依赖的包都下载并管理。

npm在Node.js安装时顺便安装了；命令行输入npm -v，输出npm的版本号，说明npm已经正确安装了。

* 第一个Node程序

hello.js：

|  |
| --- |
| 'use strict'; console.log('hello world !'); |

第一行总是写'use strict';严格模式运行JavaScript代码，避免各种潜在陷阱。

命令行切到hello.js目录：

PS C:\Users\hikari\Desktop> node hello.js

hello world !

* Node交互模式和命令行模式

Node交互式环境会把每一行JavaScript代码的结果自动打印；但是命令行运行JavaScript文件却不会。

例如：在Node交互式环境下：

> 1+2+3;

6

但是写一个calc.js的文件，命令行模式下执行：

PS C:\Users\hikari\Desktop> node calc.js

什么也没有输出，必须用console.log()才能输出结果。

输入node进入交互模式，相当于启动了Node解释器，每输入一行就执行一行；

运行node calc.js相当于启动了Node解释器，一次性把calc.js的代码执行了。

js文件就是用于写大段代码；交互模式一般用于验证部分代码。

* 使用严格模式

如果在JavaScript文件开头写上'use strict';，那么Node在执行该JavaScript时将使用严格模式。但是在服务器环境下，如果有很多JavaScript文件，每个文件都写上'use strict';很麻烦。可以给Nodejs传递--use\_strict参数开启严格模式：

node --use\_strict calc.js

* Node集成开发环境

使用文本编辑器开发Node程序，最大的缺点是效率太低，运行Node程序还需在命令行单独敲命令。如果还需要调试程序，就更加麻烦了。

所以需要一个IDE集成开发环境，能在一个环境里编码、运行、调试，这样就可以大大提升开发效率。

Java集成开发环境有Eclipse，Intellij idea等；C#集成开发环境有Visual Studio，那么问题来了：Node.js的集成开发环境到底哪家强？

此处推荐[Visual Studio Code](http://code.visualstudio.com/)，微软出品，是Visual Studio发精简版，并且Visual Studio Code可以跨平台。

安装时推荐勾上：桌面快捷方式、资源管理器目录上下文菜单、添加到PATH

为VS Code创建工作目录，新建文件hello.js

|  |
| --- |
| let name = 'world';  let s = `hello, ${name} !`;  console.log(s); |

点击调试，发现没有配置，点击齿轮自动生成配置文件launch.json：

{

"version": "0.2.0",

"configurations": [

{

"type": "node",

"request": "launch",

"name": "启动程序",

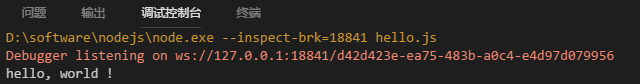
"program": "${workspaceFolder}/hello.js" // 运行的js文件

}

]

}

点击开始调试。调试控制台输出结果：



**20180424**

* 模块

随着程序代码越写越多，在一个文件里代码越来越长，越来越不容易维护。

为了编写可维护的代码，把函数分组放到不同的js文件，在Node环境一个js文件就是一个模块(module)，很多语言都采用这种代码组织方式。

除了提高代码可维护性，模块还可以避免重复造轮子，直接引用Node内置模块或第三方模块就可以使用；可以避免函数名和变量名冲突，相同名字的函数和变量可以存在不同模块中，因此在编写模块时，不必考虑名字与其他模块冲突。

示例：

hello.js：

function greet(name) {

console.log(`hello, ${name} !`);

}

// 把greet()函数作为模块输出暴露出去, 其他模块就可以使用greet()

module.exports = greet;

main.js：

let greet = require('./hello'); // 相对路径

let s = 'hikari';

greet(s); // hello, hikari !

引入的模块作为变量保存在greet变量中，就是在hello.js输出的greet()函数。所以main.js成功地引用了hello.js模块的greet()函数，然后可以直接使用了。

如果写成require('hello')，Node会依次在内置模块、全局模块和当前模块下查找hello.js，很可能会得到一个错误：Error: Cannot find module 'hello'

VSCode推荐使用ES6标准：

export default greet;

import greet from "./hello";

然后报错了…也就是不支持export和import?

* CommonJS规范

这种模块加载机制称为CommonJS规范。此规范下，每个js文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突。

要在模块中对外输出变量用：module.exports = variable;

输出的变量可以是任意对象、函数、数组等等。

要引入其他模块输出的对象用：let foo = require('other\_module');

引入的对象具体是什么，取决于引入模块输出的对象。

* 深入了解模块原理

在浏览器中大量使用全局变量不好。如果在a.js中使用了全局变量s，那么在b.js中也使用全局变量s，将造成冲突，b.js中对s赋值会改变a.js的运行逻辑。

JavaScript本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

但由于JavaScript支持闭包，如果把一段JavaScript代码用函数包装起来，其所有全局变量就变成了函数内部的局部变量，从而支持不同模块同名变量。

Node.js加载了js后，把代码外面套一层匿名函数，包装一下变成：

(function () {

// js代码

})();

原来的全局变量变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的同名变量也互不干扰。

模块的输出module.exports实现：

// node在加载js文件前准备的module对象

let module = {

id: 'hello',

exports: {}

};

let load = function (module) {

// 读取的hello.js代码

function greet(name) {

console.log(`hello, ${name} !`);

}

module.exports = greet;

return module.exports;

};

let exported = load(module);

// 保存module

save(module, exported);

变量module是Node在加载js文件前准备的一个变量，并将其传入加载函数，在hello.js中可以直接使用变量module原因在于它实际上是函数的一个参数。

通过把参数module传递给load()函数，hello.js就顺利地把一个变量传递给了Node执行环境，Node会把module变量保存到某个地方。

由于Node保存了所有导入的module，当用require()获取module时，Node找到对应的module，把这个module的exports变量返回，这样另一个模块就顺利拿到了模块的输出。

练习：编写hello.js，输出多个函数；编写main.js，引入hello模块，调用其函数。

hello.js：

let s = 'hello';

function greet(name) {

console.log(`${s}, ${name} !`);

}

function hi(name) {

console.log(`hi, ${name} !`);

}

function goodbye(name) {

console.log(`goodbye, ${name} !`);

}

module.exports = {

greet: greet,

hi: hi,

goodbye: goodbye

};

main.js：

const hello = require('./hello');

var s = 'hikari';

hello.greet(s); // hello, hikari !

hello.hi(s); // hi, hikari !

hello.goodbye(s); // goodbye, hikari !

**20180425**

* 基本模块

Node.js是运行在服务区端的JavaScript环境。服务器程序和浏览器程序相比最大的特点是没有浏览器的安全限制。服务器程序必须能接收网络请求、读写文件、处理二进制内容，所以Node.js内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中无法执行，因为它们的底层代码是用C/C++在Node.js运行环境中实现的。

* Node.js常用对象

① global

JavaScript有且仅有一个全局对象，浏览器中是window对象。而在Node.js环境中也有唯一的全局对象global，其属性和方法也和浏览器的window不同。

② process

process也是Node.js提供的一个对象，代表当前Node.js进程。

// 获取js文件名和目录

console.log('current js file: ' + \_\_filename); // d:\hello\01 global.js

console.log('current js dir: ' + \_\_dirname); // d:\hello

// ver:v8.11.1; platform: win32; arch: x64

console.log(`ver:${process.version}; platform: ${process.platform}; arch: ${process.arch}`)

// process.argv 存储了命令行参数

console.log('arguments: ' + JSON.stringify(process.argv)); // arguments: ["D:\\software\\nodejs\\node.exe","d:\\hello\\01 global.js"]

// process.cwd() 返回当前工作目录

console.log('cwd: ' + process.cwd()); // cwd: d:\hello

// 切换当前工作目录

let d = '/private/tmp';

if (process.platform === 'win32') {

// 如果是Windows，切换到 C:\Windows\System32

d = 'C:\\Windows\\System32';

}

process.chdir(d);

console.log('cwd: ' + process.cwd()); // cwd: C:\Windows\System32

JavaScript是由事件驱动执行的单线程模型，Node.js也不例外。Node.js不断执行响应事件函数，直到没有任何响应事件函数可以执行时，Node.js退出。

如果想在下一次事件响应中执行代码，可以调用process.nextTick()：

// process.nextTick()将在下一轮事件循环中调用

process.nextTick(function () {

console.log('nextTick callback!');

});

console.log('nextTick was set!');

// nextTick was set! nextTick callback!

说明传入process.nextTick()的函数不是立刻执行，而要等到下一次事件循环。

Node.js进程本身的事件就由process对象来处理。如果响应exit事件，就可以在程序即将退出时执行某个回调函数：

// 程序即将退出时的回调函数

process.on('exit', function (code) {

console.log('about to exit with code: ' + code);

}); // about to exit with code: 0

* 判断JavaScript执行环境

很多JavaScript代码既能在浏览器中执行，也能在Node环境执行。

有时程序本身需要判断自己到底在什么环境下执行，常用方式是根据浏览器和Node环境提供的全局变量名称来判断：

let env = typeof window === 'undefined' ? 'node.js' : 'browser';

console.log(`environment: ${env}`); // environment: node.js

**20180427**

* fs模块

Node.js内置的fs模块是文件系统模块，负责读写文件。

和所有其它JavaScript模块不同的是fs模块同时提供了异步和同步的方法。

① 异步读文件

const fs = require('fs');

console.log('>>> BEGIN >>>')

fs.readFile('sample.in', 'utf-8', function (err, data) {

if (err) {

console.log(err);

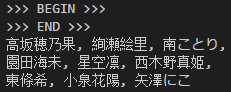
} else {

console.log(data);

}

});

console.log('>>> END >>>');



异步读取时，回调函数接收err和data两个参数。正常读取时，err为null，data为读取到的字符串；读取发生错误时，err代表一个错误对象，data为undefined。这也是Node.js标准的回调函数：第一个参数为错误信息，第二个参数为结果。

当读取二进制文件时，不传入文件编码：

fs.readFile('sample.png', function (err, data) {

if (err) {

console.log(err);

} else {

console.log(Array.isArray(data)); // false

console.log(`${data.length} bytes`); // 73770 bytes

}

});

回调函数的data是返回一个Buffer对象。在Node.js中，Buffer对象是一个包含任意个字节的数组(注意和Array不同)，length是字节数。

Buffer对象可以和String转换：

// Buffer --> String

let text = data.toString('utf-8');

console.log(text); // 乱码

// String --> Buffer

let buf = Buffer.from(text, 'utf-8');

console.log(`buf.length=${buf.length}; data.length=${data.length}`);

// buf.length=133830; data.length=73770, 两者大小都不一样?

② 同步读文件

readFileSync()函数为同步读取文件函数，不接收回调函数，函数直接返回结果。

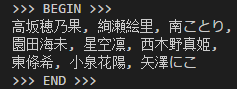
const fs = require('fs');

console.log('>>> BEGIN >>>');

let data = fs.readFileSync('sample.in', 'utf-8');

console.log(data);

console.log('>>> END >>>');

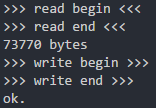


如果同步读取文件发生错误，可以使用try...catch...捕获

③ 写入文件

写入同样有异步的writeFile()和同步的writeFileSync()

|  |
| --- |
| const fs = require('fs');  *//异步写入文件*  function write(f, data) {  console.log('>>> write begin >>>');  fs.writeFile(f, data, function (err) {  if (err) {  console.log(err);  } else {  console.log('ok.');  }  });  console.log('>>> write end >>>');  }  *// 异步读取文件, 异步将data写入另一文件, 相当于复制*  function read\_write(f) {  console.log('>>> read begin <<<')  fs.readFile(f, function (err, data) {  if (err) {  console.log(err);  } else {  console.log(`${data.length} bytes`); *// 73770 bytes*  write('output.png', data);  }  });  console.log('>>> read end <<<');  }  read\_write('sample.png'); |



writeFile()的参数依次为文件名、数据和回调函数。如果传入的data是String，默认按UTF-8编码写入文本文件；如果data是Buffer，写入的是二进制文件。

回调函数只关心成功与否，只需要一个err参数。

同步写入，不需要回调函数，直接写入。

|  |
| --- |
| *// 同步写入文件*  let s = 'hello hikari\nhello world';  fs.writeFileSync('hello.out', s); |

④ stat

fs.stat()返回一个Stat对象，能获取文件或目录的详细信息：

|  |
| --- |
| const fs = require('fs');  fs.stat('sample.in', function (err, stat) {  if (err) {  console.log(err);  } else {  *// 是不是文件, 是不是目录?*  console.log('isFile: ' + stat.isFile());  console.log('isDirectory: ' + stat.isDirectory());  if (stat.isFile()) { *// 是文件, 打印文件大小, 创建日期, 修改日期*  console.log('size: ' + stat.size);  console.log('birth time: ' + stat.birthtime); *// date对象*  console.log('modified time: ' + stat.mtime); *// date对象*  }  }  }); |

结果：

isFile: true

isDirectory: false

size: 128

birth time: Fri Apr 27 2018 11:11:19 GMT+0800 (中国标准时间)

modified time: Fri Apr 27 2018 11:20:55 GMT+0800 (中国标准时间)

stat()对应同步函数statSync()：

|  |
| --- |
| try {  let info = fs.statSync('sample.in');  console.log('birth time: ' + info.birthtime);  } catch (err) {  console.log(err);  } |

* 异步还是同步

在fs模块中，提供同步方法是为了方便使用。

绝大部分在服务器端反复执行业务逻辑的JavaScript代码，必须使用异步；否则同步代码在执行时，服务器将停止响应，因为JavaScript是单线程。

服务器启动时读取配置文件或结束时写入状态文件时，可以使用同步代码。因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

* stream模块

stream是仅在服务区端可用的模块，目的是支持流这种数据结构。

流是一种抽象的数据结构，可以把数据看成是数据流。比如敲键盘时每个字符连起来看成字符流，此流是从键盘输入到应用程序，称为标准输入流(stdin)。

反之应用程序把字符一个一个输出到显示器上，称为标准输出流(stdout)。

流的特点是数据有序，必须依次读取或依次写入。

Node.js中流也是一个对象，只要响应流的事件就可以：data事件表示流的数据可以读取；end事件表示流结束，没有数据可读取了；error事件表示出错。

① 文件流读取文本

|  |
| --- |
| const fs = require('fs');  *// 打开一个输入流*  let rs = fs.createReadStream('sample.in', 'utf-8');  *// data事件可能有多次, 每次传递的chunk是流的一部分数据*  rs.on('data', function (chunk) {  console.log('data: ');  console.log(chunk);  });  rs.on('end', function () {  console.log('read stream end...');  });  rs.on('error', function (err) {  console.log(err);  }); |

② 以流的形式写入文件，只要不断调用write()，最后以end()结束：

|  |
| --- |
| let ws1 = fs.createWriteStream('output1.txt', 'utf-8');  ws1.write('使用Stream写入文本数据...\n');  ws1.write('END.');  ws1.end();  let ws2 = fs.createWriteStream('output2.txt');  ws2.write(Buffer.from('使用Stream写入二进制数据...\r\n', 'utf-8'));  ws2.write(Buffer.from('END.', 'utf-8'));  ws2.end(); |

用Windows自带的记事本'\n'没有换行；而'\r\n'换行。然而sublime、vscode都是有换行的...怪不得记事本总是被黑...

stream.Readable和stream.Writable是读取流和写入流的父类。

③ pipe

一个Readable流和一个Writable流串起来后，所有数据自动从Readable流进入Writable流，这种操作叫pipe。

Node.js中Readable流的pipe()方法，可以将一个文件流和另一个文件流串起来，源文件的数据自动写入到目标文件中，所以实际上是一个复制文件的过程：

|  |
| --- |
| function copy\_file(f1, f2) {  let rs = fs.createReadStream(f1);  let ws = fs.createWriteStream(f2);  rs.pipe(ws);  }  copy\_file('sample.png', 'copy.png'); |

默认当Readable流的数据读取完毕，end事件触发将自动关闭Writable流。

如果不希望自动关闭Writable流，需要传入参数：

readable.pipe(writable, { end: false });