JavaScript 规范（ES6）

Airbnb 的 ES5 规范写的非常好，现在添加了 ES6 的部分。

另外阮一峰老师的 ECMAScript 6 入门值得参考。

类型

- 原始类型：值传递
    - string
    - number
    - boolean
    - null
    - undefined

```
const foo = 1;
let bar = foo;

bar = 9;

console.log(foo, bar);
```

- 复杂类型：引用传递
    - object
    - array
    - function

```
const foo = [1, 2];
const bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]);
```

引用

- 为引用使用 const 关键字，而不是 var

    这样确保你不能修改引用类型，否则可能会导致一些 bug 或难以理解的代码。

```

var a = 1;
var b = 2;


const a = 1;
const b = 2;
```

- 如果你必须修改引用，使用 `let` 代替 `var`

  > 因为 `let` 是块作用域的，而 `var` 是函数作用域。

```
var count = 1;
if (true) {
  count += 1;
}


let count = 1;
if (true) {
  count += 1;
}
```

- `let` 和 `const` 都是块作用域的

对象

- 使用对象字面量创建对象

```
var item = new Object();


var item = {};
```

- 不要使用保留字（reserved words）作为键，否则在 IE8 下将出错，issue

```
// bad
var superman = {
  class: 'superhero',
  default: { clark: 'kent' },
  private: true
};

// good
var superman = {
  klass: 'superhero',
  defaults: { clark: 'kent' },
  hidden: true
};
```

- 使用易读的同义词代替保留字

```
// bad
const superman = {
  class: 'alien'
};

// bad
const superman = {
  klass: 'alien'
};

```

```
// good
const superman = {
  type: 'alien'
};
```

- 创建对象时使用计算的属性名，而不要在创建对象后使用对象的动态特性

  这样可以在同一个位置定义对象的所有属性。

```
function getKey(k) {
    return `a key named ${k}`;
  }


  const obj = {
    id: 5,
    name: 'San Francisco'
  };
  obj[getKey('enabled')] = true;


  const obj = {
    id: 5,
    name: 'San Francisco',
    [getKey('enabled')]: true
  };
```

- 使用定义对象方法的简短形式

```

const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  }
};


const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  }
};
```

- 使用定义对象属性的简短形式

  书写起来更加简单，并且可以自描述。

```
const lukeSkywalker = 'Luke Skywalker';


  const obj = {
    lukeSkywalker: lukeSkywalker
  };

```

```
8
9     const obj = {
10      lukeSkywalker
11    };
```

- 将所有简写的属性写在对象定义的最顶部

    这样可以更加方便地知道哪些属性使用了简短形式。

```
1   const anakinSkywalker = 'Anakin Skywalker';
2     const lukeSkywalker = 'Luke Skywalker';
3
4
5     const obj = {
6       episodeOne: 1,
7       twoJedisWalkIntoACantina: 2,
8       lukeSkywalker,
9       episodeThree: 3,
10      mayTheFourth: 4,
11      anakinSkywalker
12    };
13
14
15    const obj = {
16      lukeSkywalker,
17      anakinSkywalker,
18      episodeOne: 1,
19      twoJedisWalkIntoACantina: 2,
20      episodeThree: 3,
21      mayTheFourth: 4
22    };
```

数组

- 使用字面量语法创建数组

```
1
2   const items = new Array();
3
4
5   const items = [];
```

- 如果你不知道数组的长度，使用 push

```
1   const someStack = [];
2
3
4
5   someStack[someStack.length] = 'abracadabra';
6
7
8   someStack.push('abracadabra');
```

- 使用 `...` 来拷贝数组

```
1
2   const len = items.length;
```

```
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}


const itemsCopy = [...items];
```

- 使用 Array.from 将类数组对象转换为数组

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

解构 Destructuring

- 访问或使用对象的多个属性时请使用对象的解构赋值

  > 解构赋值避免了为这些属性创建临时变量或对象。

```

function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}


function getFullName(obj) {
  const { firstName, lastName } = obj;
  return `${firstName} ${lastName}`;
}


function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- 使用数组解构赋值

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


const first = arr[0];
const second = arr[1];


const [first, second] = arr;
```

- 函数有多个返回值时使用对象解构，而不是数组解构

  > 这样你就可以随时添加新的返回值或任意改变返回值的顺序，而不会导致调用失败。

```
function processInput(input) {

    return [left, right, top, bottom];
  }


  const [left, __, top] = processInput(input);


  function processInput(input) {

    return { left, right, top, bottom };
  }


  const { left, right } = processInput(input);
```

字符串

- 使用单引号 ''

```

var name = "Bob Parr";


var name = 'Bob Parr';


var fullName = "Bob " + this.lastName;


var fullName = 'Bob ' + this.lastName;
```

- 超过80个字符的字符串应该使用字符串连接换行
- 注：如果过度使用长字符串连接可能会对性能有影响。jsPerf & Discussion

```

var errorMessage = 'This is a super long error that was thrown because of
Batman. When you stop to think about how Batman had anything to do with
this, you would get nowhere fast.';


var errorMessage = 'This is a super long error that \
was thrown because of Batman. \
When you stop to think about \
how Batman had anything to do \
with this, you would get nowhere \
fast.';



var errorMessage = 'This is a super long error that ' +
  'was thrown because of Batman.' +
  'When you stop to think about ' +
  'how Batman had anything to do ' +
  'with this, you would get nowhere ' +
  'fast.';
```

- 编程构建字符串时，使用字符串模板而不是字符串连接

  > 模板给你一个可读的字符串，简洁的语法与适当的换行和字符串插值特性。

```
function sayHi(name) {
  return 'How are you, ' + name + '?';
}


function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}


function sayHi(name) {
  return `How are you, ${name}?`;
}
```

函数

- 使用函数声明而不是函数表达式

  > 函数声明拥有函数名，在调用栈中更加容易识别。并且，函数声明会整体提升，而函数表达式只会提升变量本身。这条规则也可以这样描述，始终使用箭头函数来代替函数表达式。

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


function foo() {
}
```

- 函数表达式

```
(() => {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- 绝对不要在一个非函数块（if，while，等等）里声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但是它们解析不同
- 注：ECMA-262 把 块 定义为一组语句，函数声明不是一个语句。阅读 ECMA-262 对这个问题的说明

```
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}
```

```
8
9   if (currentUser) {
10    var test = function test() {
11      console.log('Yup.');
12    };
13  }
```

- 绝对不要把参数命名为 arguments, 这将会覆盖函数作用域内传过来的 arguments 对象

```
1
2   function nope(name, options, arguments) {
3
4   }
5
6
7   function yup(name, options, args) {
8
9   }
```

- 永远不要使用 arguments，使用 ... 操作符来代替

> ... 操作符可以明确指定你需要哪些参数，并且得到的是一个真实的数组，而不是 arguments 这样的类数组对象。

```
1
2   function concatenateAll() {
3     const args = Array.prototype.slice.call(arguments);
4     return args.join('');
5   }
6
7
8   function concatenateAll(...args) {
9     return args.join('');
10  }
```

- 使用函数参数默认值语法，而不是修改函数的实参

箭头函数 Arrow Functions

- 当必须使用函数表达式时（例如传递一个匿名函数时），请使用箭头函数

> 箭头函数提供了更简洁的语法，并且箭头函数中 this 对象的指向是不变的，this 对象绑定定义时所在的对象，这通常是我们想要的。如果该函数的逻辑非常复杂，请将该函数提取为一个函数声明。

```
1
2   [1, 2, 3].map(function (x) {
3     return x * x;
4   });
5
6
7   [1, 2, 3].map((x) => {
8     return x * x
9   });
```

- 总是用括号包裹参数，省略括号只适用于单个参数，并且还降低了程序的可读性

```
  [1, 2, 3].map(x => x * x);


  [1, 2, 3].map((x) => x * x);
```

构造函数

- 总是使用 class 关键字，避免直接修改 prototype

  > class 语法更简洁，也更易理解。

```
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}



class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

- 使用 extends 关键字来继承

  > 这是一个内置的继承方式，并且不会破坏 instanceof 原型检查。

```
  const inherits = require('inherits');
  function PeekableQueue(contents) {
    Queue.apply(this, contents);
  }
  inherits(PeekableQueue, Queue);
  PeekableQueue.prototype.peek = function() {
    return this._queue[0];
  }


  class PeekableQueue extends Queue {
    peek() {
      return this._queue[0];
    }
  }
```

- 在方法中返回 this 以方便链式调用

```

Jedi.prototype.jump = function() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
};

const luke = new Jedi();
luke.jump();
luke.setHeight(20);


class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}

const luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- 可以写一个自定义的toString()方法，但是确保它工作正常并且不会有副作用

```
class Jedi {
  contructor(options = {}) {
    this.name = options.name || 'no name';
  }

  getName() {
    return this.name;
  }

  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

模块

- 总是在非标准的模块系统中使用标准的 import 和 export 语法，我们总是可以将标准的模块语法转换成支持特定模块加载器的语法。

> 模块是未来的趋势，那么我们为何不现在就开始使用。

```

```

```
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;


import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;


import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 不要使用通配符 * 的 import

  这样确保了只有一个默认的 export 项

```

import * as AirbnbStyleGuide from './AirbnbStyleGuide';


import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- 不要直接从一个 import 上 export

  虽然一行代码看起来更简洁，但是有一个明确的 import 和一个明确的 export 使得代码行为更加明确。

```


export default { es6 } from './airbnbStyleGuide';



import { es6 } from './AirbnbStyleGuide';
export default es6;
```

Iterators 和 Generators

- 不要使用迭代器（Iterators）。优先使用 JavaScript 中 map 和 reduce 这类高阶函数来代替 for-of 循环

  处理纯函数的返回值更加容易并且没有副作用

```
const numbers = [1, 2, 3, 4, 5];


  let sum = 0;
  for (let num of numbers) {
    sum += num;
  }

  sum === 15;


  let sum = 0;
  numbers.forEach((num) => sum += num);
  sum === 15;
```

```
15
16
17    const sum = numbers.reduce((total, num) => total + num, 0);
18    sum === 15;
```

- 不要使用 generators

    它们不太容易转换为 ES5 的语法。

属性

- 使用点 . 操作符来访问属性

```
1   const luke = {
2     jedi: true,
3     age: 28
4   };
5
6
7   const isJedi = luke['jedi'];
8
9
10  const isJedi = luke.jedi;
```

- 当使用变量访问属性时使用中括号 [ ]

```
1   var luke = {
2     jedi: true,
3     age: 28
4   };
5
6   function getProp(prop) {
7     return luke[prop];
8   }
9
10  var isJedi = getProp('jedi');
```

变量

- 总是使用 const 来声明变量，否则将生成全局变量，我们应该避免污染全局命名空间

```
1
2  superPower = new SuperPower();
3
4
5  const superPower = new SuperPower();
```

- 为每个变量都使用 const 关键字声明

    这种方式更加容易添加新变量，并且不必担忧将 , 错误写成 ; 而导致生成全局变量。

```
1
2   const items = getItems(),
```

```
    goSportsTeam = true,
    dragonball = 'z';



const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';


const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- 将所有 const 变量放在一起，然后将所有 let 变量放在一起

```

let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;


let i;
let items = getItems();
let dragonball;
let goSportsTeam = true;
let len;


const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

- 在必要的时候声明变量，并且将其放在合适的位置

  let 和 const 是块级作用域的，而不是函数作用域。

```

  function() {
    test();
    console.log('doing stuff..');



    const name = getName();

    if (name === 'test') {
      return false;
    }

    return name;
  }


  function() {
    const name = getName();

```

```
    if (!arguments.length) {
      return false;
    }

    return true;
  }


  function() {
    if (!arguments.length) {
      return false;
    }

    const name = getName();

    return true;
  }
```

变量提升 Hoisting

- 通过 `var` 声明的变量将被提升到作用域的顶部，但他们的赋值不会被提升。通过 `const` 和 `let` 声明的变量不存在变量提升，这里有一个新概念，称为“暂时性死区（Temporal Dead Zones (TDZ)）”。有必要理解 `typeof` 不再是一个百分之百安全的操作。

- 匿名函数表达式提升了对应的变量名，但赋值过程没有被提升

```
function example() {
  console.log(anonymous);

  anonymous();

  let anonymous = function() {
    console.log('anonymous function expression');
  };
}
```

- 命名的函数表达式提升了对应的变量名，函数名和函数体没有被提升

```
function example() {
  console.log(named);

  named();

  superPower();

  var named = function superPower() {
    console.log('Flying');
  };
}



function example() {
  console.log(named);

  named();

  var named = function named() {
```

```
    console.log('named');
  }
}
```

- 函数声明将被提升

```
function example() {
  superPower();

  function superPower() {
    console.log('Flying');
  }
}
```

- 更多细节可以参考 Ben Cherry 的 JavaScript Scoping & Hoisting

比较运算符和等号

- 使用 `===` 和 `!==` 而不是 `==` 和 `!=`
- 比较运算通过 `ToBoolean` 强制转换并遵循一下规则：
  - `Object` - `true`
  - `Undefined` - `false`
  - `Null` - `false`
  - `Booleans` - 被转换为对应的值
  - `Number` - 值为 `+0`，`-0`，`NaN` 时为 `false`，否则为 `true`
  - `String` - 空字符串 `''` 为 `false`，否则为 `true`
- 使用快捷方式

```

if (name !== '') {

}


if (name) {

}


if (collection.length > 0) {

}


if (collection.length) {

}
```

- 更多细节请阅读 Truth Equality and JavaScript

块

- 给所有多行的块使用大括号

```

if (test)
  return false;


if (test) return false;


if (test) {
  return false;
}


function() { return false; }


function() {
  return false;
}
```

- 使用 `if...else` 这样的多行块时，请将 `else` 和 `if` 的结束括号放在同一行

```

if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}


if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

注释

- 使用 `/** ... */` 进行多行注释，包括描述，指定类型以及参数值和返回值

```
```

```
function make(tag) {


  return element;
```

```
}


 * make() returns a new element
 * based on the passed in tag name
 *
 * @param <String> tag
 * @return <Element> element
 */
function make(tag) {



  return element;
}
```

- 使用 // 进行单行注释，将注释放在被注释对象的上面，并在注释之前保留一个空行

```

const active = true;



const active = true;


function getType() {
  console.log('fetching type...');

  const type = this._type || 'no type';

  return type;
}


function getType() {
  console.log('fetching type...');


  const type = this._type || 'no type';

  return type;
}
```

- 使用 // FIXME: 来注释一个问题

```
function Calculator() {


  total = 0;

  return this;
}
```

- 使用 // TODO: 来注释一个问题的解决方案

```
function Calculator() {

```

```
3
4     this.total = 0;
5
6     return this;
7   }
```

空白

- 将 tab 设置为 2 个空格缩进

```
1
2   function() {
3   ····const name;
4   }
5
6
7   function() {
8   ·const name;
9   }
10
11
12  function() {
13  ··const name;
14  }
```

- 前大括号前放置一个空格

```
1
2   function test(){
3     console.log('test');
4   }
5
6
7   function test() {
8     console.log('test');
9   }
10
11
12  dog.set('attr',{
13    age: '1 year',
14    breed: 'Bernese Mountain Dog'
15  });
16
17
18  dog.set('attr', {
19    age: '1 year',
20    breed: 'Bernese Mountain Dog'
21  });
```

- 运算符之间用空格分隔

```
1
2   const x=y+5;
3
4
5   const x = y + 5;
```

- 文件末尾使用单个换行符

```

(function(global) {

})(this);
```

```

(function(global) {

})(this);↵
↵
```

```

(function(global) {

})(this);↵
```

- 方法链式调用时保持适当的缩进，并且使用前置的 . 来表示该行是一个方法调用，而不是一个新语句

```
$('#items').find('.selected').highlight().end().find('.open').updateCount();


$('#items').
  find('selected').
    highlight().
    end().
  find('.open').
    updateCount();


$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();


const leds =
stage.selectAll('.led').data(data).enter().append('svg:svg').class('led',
true)
    .attr('width',  (radius + margin) * 2).append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius +
margin) + ')')
    .call(tron.led);


const leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .class('led', true)
    .attr('width',  (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius +
margin) + ')')
    .call(tron.led);
```

- 在语句块之后和下一语句之前都保持一个空行

```

if (foo) {
  return bar;
}
return baz;


if (foo) {
  return bar;
}

return baz;


const obj = {
  foo: function() {
  },
  bar: function() {
  }
};
return obj;


const obj = {
  foo: function() {
  },

  bar: function() {
  }
};

return obj;
```

逗号

- 不要将逗号放前面

```

const story = [
    once
  , upon
  , aTime
];


const story = [
  once,
  upon,
  aTime
];


const hero = {
    firstName: 'Bob'
  , lastName: 'Parr'
  , heroName: 'Mr. Incredible'
  , superPower: 'strength'
};


```

```
const hero = {
  firstName: 'Bob',
  lastName: 'Parr',
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};
```

- 不要添加多余的逗号，否则将在 IE6/7 和 IE9 的怪异模式下导致错误。同时，某些 ES3 的实现会计算多数组的长度，这在 ES5 中有澄清

```

  const hero = {
    firstName: 'Kevin',
    lastName: 'Flynn',
  };

  const heroes = [
    'Batman',
    'Superman',
  ];


  const hero = {
    firstName: 'Kevin',
    lastName: 'Flynn'
  };

  const heroes = [
    'Batman',
    'Superman'
  ];
```

分号

- 句末一定要添加分号

```

(function() {
  const name = 'Skywalker'
  return name
})()


(() => {
  const name = 'Skywalker';
  return name;
})();


;(() => {
  const name = 'Skywalker';
  return name;
})();
```

类型转换

- 在语句的开始执行类型转换

- 字符串：

```


const totalScore = this.reviewScore + '';

const totalScore = String(this.reviewScore);
```

- 对数字使用 parseInt 并且总是带上类型转换的基数

```
const inputValue = '4';

const val = new Number(inputValue);

const val = +inputValue;

const val = inputValue >> 0;

const val = parseInt(inputValue);

const val = Number(inputValue);

const val = parseInt(inputValue, 10);
```

- 不管是出于一些奇特的原因，还是
  parseInt 是一个瓶颈而需要位运算来解决某些性能问题，请为你的代码注释为什么要这样做

```


 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
const val = inputValue >> 0;
```

- 注意：使用位移运算时要特别小心。Number 在 JavaScript 中表示为 64 位的值，但位移运算总是返回一个 32 位的整数（source），对大于 32 位的整数进行位移运算会导致意外的结果（讨论）。32 位最大整数为 2,147,483,647：

- 布尔值

```
var age = 0;

var hasAge = new Boolean(age);

```

```
6
7  var hasAge = Boolean(age);
8
9
10 var hasAge = !!age;
```

命名约定

- 避免单个字符名，让你的变量名有描述意义

```
1
2  function q() {
3
4  }
5
6
7  function query() {
8
9  }
```

- 命名对象、函数和实例时使用小驼峰命名规则

```
1
2  var OBJEcttsssss = {};
3  var this_is_my_object = {};
4  var this-is-my-object = {};
5  function c() {};
6  var u = new user({
7    name: 'Bob Parr'
8  });
9
10
11 var thisIsMyObject = {};
12 function thisIsMyFunction() {};
13 var user = new User({
14   name: 'Bob Parr'
15 });
```

- 命名构造函数或类时使用大驼峰命名规则

```
1
2  function user(options) {
3    this.name = options.name;
4  }
5
6  const bad = new user({
7    name: 'nope'
8  });
9
10
11 class User {
12   constructor(options) {
13     this.name = options.name;
14   }
15 }
16
17 const good = new User({
18   name: 'yup'
```

```
});
```

- 命名私有属性时前面加个下划线 _

```

this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';


this._firstName = 'Panda';
```

- 保存对 this 的引用时使用 _this

```

function() {
  var self = this;
  return function() {
    console.log(self);
  };
}


function() {
  var that = this;
  return function() {
    console.log(that);
  };
}


function() {
  var _this = this;
  return function() {
    console.log(_this);
  };
}
```

- 导出单一一个类时，确保你的文件名就是你的类名

```

class CheckBox {

}
module.exports = CheckBox;



const CheckBox = require('./checkBox');


const CheckBox = require('./check_box');


const CheckBox = require('./CheckBox');
```

- 导出一个默认小驼峰命名的函数时，文件名应该就是导出的方法名

```
function makeStyleGuide() {
```

```
}

export default makeStyleGuide;
```

- 导出单例、函数库或裸对象时，使用大驼峰命名规则

```
const AirbnbStyleGuide = {
  es6: {
  }
};

export default AirbnbStyleGuide;
```

访问器

- 属性的访问器函数不是必须的
- 如果你确实有存取器函数的话使用 `getVal()` 和 `setVal('hello')`

```


dragon.age();


dragon.getAge();


dragon.age(25);


dragon.setAge(25);
```

- 如果属性是布尔值，使用 `isVal()` 或 `hasVal()`

```

if (!dragon.age()) {
  return false;
}


if (!dragon.hasAge()) {
  return false;
}
```

- 可以创建get()和set()函数，但是要保持一致性

```
function Jedi(options) {
  options || (options = {});
  var lightsaber = options.lightsaber || 'blue';
  this.set('lightsaber', lightsaber);
}

Jedi.prototype.set = function(key, val) {
  this[key] = val;
};

Jedi.prototype.get = function(key) {
  return this[key];
};
```

事件

- 当给事件附加数据时，传入一个哈希而不是原始值，这可以让后面的贡献者加入更多数据到事件数据里而不用找出并更新那个事件的事件处理器

```

$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function(e, listingId) {

});
```

```

$(this).trigger('listingUpdated', { listingId : listing.id });

...

$(this).on('listingUpdated', function(e, data) {

});
```

jQuery

- 为 jQuery 对象命名时添加 $ 前缀

```

const sidebar = $('.sidebar');


const $sidebar = $('.sidebar');
```

- 缓存 jQuery 的查询结果

```

function setSidebar() {
  $('.sidebar').hide();



  $('.sidebar').css({
    'background-color': 'pink'
  });
}


function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();



  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- 对DOM查询使用级联的 `$('.sidebar ul')` 或 `$('.sidebar ul')`，jsPerf
- 在指定作用域进行查询时使用 `find`

```
$('ul', '.sidebar').hide();


$('.sidebar').find('ul').hide();


$('.sidebar ul').hide();


$('.sidebar > ul').hide();


$sidebar.find('ul').hide();
```

ECMAScript 5 兼容性

- 参考 Kangax 的 ES5 compatibility table

ECMAScript 6 新特性

下面是本文涉及到的 ES6 新特性：

性能

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

资源

Read This

- Annotated ECMAScript 5.1

工具

- Code Style Linters
  - JSHint - Airbnb Style .jshintrc
  - JSCS - Airbnb Style Preset

其它规范

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

其它风格

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

更多文章

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban
- Frontend Guidelines - Benjamin De Cock

书籍

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg
- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript - David Herman

播客

- DailyJS

- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts

Podcasts

- JavaScript Jabber