js-es5代码规范

目录

类型

- **原始值**: 存取直接作用于它自身。
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`

```
var foo = 1;
var bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- **复杂类型**: 存取时作用于它自身值的引用。
  - `object`
  - `array`
  - `function`

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

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

对象

- 使用直接量创建对象。

```
// bad
var item = new Object();

// good
var item = {};
```

- 不要使用保留字作为键名，它们在 IE8 下不工作。更多信息。

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

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

- 使用同义词替换需要使用的保留字。

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

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

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

数组

- 使用直接量创建数组。

```
// bad
var items = new Array();

// good
var items = [];
```

- 向数组增加元素时使用 Array#push 来替代直接赋值。

```
var someStack = [];


// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- 当你需要拷贝数组时，使用 Array#slice。jsPerf

```
var len = items.length;
var itemsCopy = [];
var i;

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

// good
itemsCopy = items.slice();
```

- 使用 Array#slice 将类数组对象转换成数组。

```
function trigger() {
  var args = Array.prototype.slice.call(arguments);
  ...
}
```

字符串

- 使用单引号 `''` 包裹字符串。

```
// bad
var name = "Bob Parr";

// good
var name = 'Bob Parr';

// bad
var fullName = "Bob " + this.lastName;

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

- 超过 100 个字符的字符串应该使用连接符写成多行。
- 注：若过度使用，通过连接符连接的长字符串可能会影响性能。jsPerf & 讨论.

```
// bad
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.';

// bad
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.';

// good
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.';
```

- 程序化生成的字符串使用 Array#join 连接而不是使用连接符。尤其是 IE 下：jsPerf.

```
var items;
var messages;
var length;
var i;

messages = [{
  state: 'success',
  message: 'This one worked.'
}, {
  state: 'success',
  message: 'This one worked as well.'
}, {
  state: 'error',
  message: 'This one did not work.'
}];

length = messages.length;

// bad
function inbox(messages) {
  items = '<ul>';

  for (i = 0; i < length; i++) {
    items += '<li>' + messages[i].message + '</li>';
  }

  return items + '</ul>';
}

// good
```

```
function inbox(messages) {
  items = [];

  for (i = 0; i < length; i++) {
    // use direct assignment in this case because we're micro-optimizing.
    items[i] = '<li>' + messages[i].message + '</li>';
  }

  return '<ul>' + items.join('') + '</ul>';
}
```

函数

- 函数表达式：

```
// 匿名函数表达式
var anonymous = function() {
  return true;
};

// 命名函数表达式
var named = function named() {
  return true;
};

// 立即调用的函数表达式（IIFE）
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

- 永远不要在一个非函数代码块（if、while 等）中声明一个函数，浏览器允许你这么做，但它们的解析表现不一致，正确的做法是：在块外定义一个变量，然后将函数赋值给它。

- **注：** ECMA-262 把 `块` 定义为一组语句。函数声明不是语句。阅读对 ECMA-262 这个问题的说明。

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

// good
var test;
if (currentUser) {
  test = function test() {
    console.log('Yup.');
  };
}
```

- 永远不要把参数命名为 `arguments`。这将取代函数作用域内的 `arguments` 对象。

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

属性

- 使用 `.` 来访问对象的属性。

```
var luke = {
  jedi: true,
  age: 28
};

// bad
```

```
var isJedi = luke['jedi'];

// good
var isJedi = luke.jedi;
```

- 当通过变量访问属性时使用中括号 `[]`。

```
var luke = {
  jedi: true,
  age: 28
};

function getProp(prop) {
  return luke[prop];
}

var isJedi = getProp('jedi');
```

变量

- 总是使用 `var` 来声明变量。不这么做将导致产生全局变量。我们要避免污染全局命名空间。

```
// bad
superPower = new SuperPower();

// good
var superPower = new SuperPower();
```

- 使用 `var` 声明每一个变量。 这样做的好处是增加新变量将变的更加容易，而且你永远不用再担心调换错 `;` 跟 `,`。

```
// bad
var items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// （跟上面的代码比较一下，看看哪里错了）
var items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';
```

- 最后再声明未赋值的变量。当你需要引用前面的变量赋值时这将变的很有用。

```
// bad
var i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
var i;
var items = getItems();
var dragonball;
var goSportsTeam = true;
var len;

// good
var items = getItems();
var goSportsTeam = true;
var dragonball;
var length;
var i;
```

- 在作用域顶部声明变量。这将帮你避免变量声明提升相关的问题。

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

```
  //..other stuff..

  var name = getName();

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

  return name;
}

// good
function () {
  var name = getName();

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

  //..other stuff..

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

  return name;
}

// bad - 不必要的函数调用
function () {
  var name = getName();

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

  this.setFirstName(name);

  return true;
}

// good
function () {
  var name;

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

  name = getName();
  this.setFirstName(name);

  return true;
}
```

提升

- 变量声明会提升至作用域顶部，但赋值不会。

```
// 我们知道这样不能正常工作（假设这里没有名为 notDefined 的全局变量）
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// 但由于变量声明提升的原因，在一个变量引用后再创建它的变量声明将可以正常工作。
// 注：变量赋值为 `true` 不会提升。
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// 解释器会把变量声明提升到作用域顶部，意味着我们的例子将被重写成：
function example() {
  var declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
```

```
  declaredButNotAssigned = true;
}
```

- 匿名函数表达式会提升它们的变量名，但不会提升函数的赋值。

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

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

- 命名函数表达式会提升变量名，但不会提升函数名或函数体。

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

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

// 当函数名跟变量名一样时，表现也是如此。
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

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

- 函数声明提升它们的名字和函数体。

```
function example() {
  superPower(); // => Flying

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

- 了解更多信息在 JavaScript Scoping & Hoisting by Ben Cherry.

比较运算符 & 等号

- 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`.
- 条件表达式例如 `if` 语句通过抽象方法 `ToBoolean` 强制计算它们的表达式并且总是遵守下面的规则：
  - **对象** 被计算为 **true**
  - **Undefined** 被计算为 **false**
  - **Null** 被计算为 **false**
  - **布尔值** 被计算为 **布尔的值**
  - **数字** 如果是 **+0、-0 或 NaN** 被计算为 **false**，否则为 **true**
  - **字符串** 如果是空字符串 `''` 被计算为 **false**，否则为 **true**

```
if ([0]) {
  // true
  // 一个数组就是一个对象，对象被计算为 true
}
```

- 使用快捷方式。

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

- 了解更多信息在 Truth Equality and JavaScript by Angus Croll.

块

- 使用大括号包裹所有的多行代码块。

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function () { return false; }

// good
function () {
  return false;
}
```

- 如果通过 `if` 和 `else` 使用多行代码块，把 `else` 放在 `if` 代码块关闭括号的同一行。

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

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

注释

- 使用 `/** ... */` 作为多行注释。包含描述、指定所有参数和返回值的类型和值。

```
// bad
// make() returns a new element
// based on the passed in tag name
//
```

```
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...stuff...

  return element;
}

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

  // ...stuff...

  return element;
}
```

- 使用 `//` 作为单行注释。在评论对象上面另起一行使用单行注释。在注释前插入空行。

```
// bad
var active = true;  // is current tab

// good
// is current tab
var active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  var type = this.type || 'no type';

  return type;
}

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

  // set the default type to 'no type'
  var type = this.type || 'no type';

  return type;
}
```

- 给注释增加 `FIXME` 或 `TODO` 的前缀可以帮助其他开发者快速了解这是一个需要复查的问题，或是给需要实现的功能提供一个解决方式。这将有别于常见的注释，因为它们是可操作的。使用 `FIXME -- need to figure this out` 或者 `TODO -- need to implement`。
- 使用 `// FIXME:` 标注问题。

```
function Calculator() {

  // FIXME: shouldn't use a global here
  total = 0;

  return this;
}
```

- 使用 `// TODO:` 标注问题的解决方式。

```
function Calculator() {

  // TODO: total should be configurable by an options param
  this.total = 0;

  return this;
}
```

空白

- 使用 2 个空格作为缩进。

```
// bad
function () {
····var name;
}

// bad
function () {
·var name;
}

// good
function () {
··var name;
}
```

- 在大括号前放一个空格。

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});
```

- 在控制语句（`if`、`while` 等）的小括号前放一个空格。在函数调用及声明中，不在函数的参数列表前加空格。

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

- 使用空格把运算符隔开。

```
// bad
var x=y+5;

// good
var x = y + 5;
```

- 在文件末尾插入一个空行。

```
// bad
(function (global) {
  // ...stuff...
})(this);
```

```
// bad
(function (global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function (global) {
  // ...stuff...
})(this);↵
```

- 在使用长方法链时进行缩进。使用前面的点 `.` 强调这是方法调用而不是新语句。

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

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

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

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

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

- 在块末和新语句前插入空行。

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
var obj = {
  foo: function () {
  },
```

```
  bar: function () {
  }
};
return obj;

// good
var obj = {
  foo: function () {
  },

  bar: function () {
  }
};

return obj;
```

逗号

- 行首逗号: **不需要**。

```
// bad
var story = [
    once
  , upon
  , aTime
];

// good
var story = [
  once,
  upon,
  aTime
];

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

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

- 额外的行末逗号：**不需要**。这样做会在 IE6/7 和 IE9 怪异模式下引起问题。同样，多余的逗号在某些 ES3 的实现里会增加数组的长度。在 ES5 中已经澄清了 (source)：

> Edition 5 clarifies the fact that a trailing comma at the end of an ArrayInitialiser does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

```
```javascript
// bad
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn',
};

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

// good
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn'
```
```

```
};

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

分号

- **使用分号。**

```
// bad
(function () {
  var name = 'Skywalker'
  return name
})()

// good
(function () {
  var name = 'Skywalker';
  return name;
})();

// good (防止函数在两个 IIFE 合并时被当成一个参数
;(function () {
  var name = 'Skywalker';
  return name;
})();
```

了解更多.

类型转换

- 在语句开始时执行类型转换。
- 字符串：

```
//  => this.reviewScore = 9;

// bad
var totalScore = this.reviewScore + '';

// good
var totalScore = '' + this.reviewScore;

// bad
var totalScore = '' + this.reviewScore + ' total score';

// good
var totalScore = this.reviewScore + ' total score';
```

- 使用 `parseInt` 转换数字时总是带上类型转换的基数。

```
var inputValue = '4';

// bad
var val = new Number(inputValue);

// bad
var val = +inputValue;

// bad
var val = inputValue >> 0;

// bad
var val = parseInt(inputValue);

// good
var val = Number(inputValue);

// good
```

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

- 如果因为某些原因 `parseInt` 成为你所做的事的瓶颈而需要使用位操作解决性能问题时，留个注释说清楚原因和你的目的。

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

- **注：** 小心使用位操作运算符。数字会被当成 64 位值，但是位操作运算符总是返回 32 位的整数（source）。位操作处理大于 32 位的整数值时还会导致意料之外的行为。讨论。最大的 32 位整数是 2,147,483,647：

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- 布尔:

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// good
var hasAge = Boolean(age);

// good
var hasAge = !!age;
```

命名规则

- 避免单字母命名。命名应具备描述性。

```
// bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- 使用驼峰式命名对象、函数和实例。

```
// bad
var OBJEcttssss = {};
var this_is_my_object = {};
var o = {};
function c() {}

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 使用帕斯卡式命名构造函数或类。

```
// bad
function user(options) {
  this.name = options.name;
}

var bad = new user({
  name: 'nope'
});

// good
function User(options) {
  this.name = options.name;
```

```
}

var good = new User({
  name: 'yup'
});
```

- 不要使用下划线前/后缀。

> 为什么？JavaScript 并没有私有属性或私有方法的概念。虽然使用下划线是表示「私有」的一种共识，但实际上这些属性是完全公开的，它本身就是你公共接口的一部分。这种习惯或许会导致开发者错误的认为改动它不会造成破坏或者不需要去测试。长话短说：如果你想要某处为「私有」，它必须不能是显式提出的。

```
```javascript
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';
this._firstName = 'Panda';

// good
this.firstName = 'Panda';
```
```

- 不要保存 `this` 的引用。使用 Function#bind。

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

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

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

// good
function () {
  return function () {
    console.log(this);
  }.bind(this);
}
```

- 给函数命名。这在做堆栈轨迹时很有帮助。

```
// bad
var log = function (msg) {
  console.log(msg);
};

// good
var log = function log(msg) {
  console.log(msg);
};
```

- **注：** IE8 及以下版本对命名函数表达式的处理有些怪异。了解更多信息到 http://kangax.github.io/nfe/。
- 如果你的文件导出一个类，你的文件名应该与类名完全相同。

```
// file contents
class CheckBox {
  // ...
}
module.exports = CheckBox;

// in some other file
// bad
var CheckBox = require('./checkBox');

// bad
var CheckBox = require('./check_box');

// good
var CheckBox = require('./CheckBox');
```

存取器

- 属性的存取函数不是必须的。
- 如果你需要存取函数时使用 `getVal()` 和 `setVal('hello')`。

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- 如果属性是布尔值，使用 `isVal()` 或 `hasVal()`。

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

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

- 创建 get() 和 set() 函数是可以的，但要保持一致。

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

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

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

构造函数

- 给对象原型分配方法，而不是使用一个新对象覆盖原型。覆盖原型将导致继承出现问题：重设原型将覆盖原有原型！

```
function Jedi() {
  console.log('new jedi');
}

// bad
Jedi.prototype = {
```

```
  fight: function fight() {
    console.log('fighting');
  },

  block: function block() {
    console.log('blocking');
  }
};

// good
Jedi.prototype.fight = function fight() {
  console.log('fighting');
};

Jedi.prototype.block = function block() {
  console.log('blocking');
};
```

- 方法可以返回 `this` 来实现方法链式使用。

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

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

var luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

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

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

var luke = new Jedi();

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

- 写一个自定义的 `toString()` 方法是可以的，但是确保它可以正常工作且不会产生副作用。

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

Jedi.prototype.getName = function getName() {
  return this.name;
};

Jedi.prototype.toString = function toString() {
  return 'Jedi - ' + this.getName();
};
```

事件

- 当给事件附加数据时（无论是 DOM 事件还是私有事件），传入一个哈希而不是原始值。这样可以让后面的贡献者增加更多数据到事件数据而无需找出并更新事件的每一个处理器。例如，不好的写法：

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

...
```

```
$(this).on('listingUpdated', function (e, listingId) {
  // do something with listingId
});
```

更好的写法：

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

...

$(this).on('listingUpdated', function (e, data) {
  // do something with data.listingId
});
```

模块

- 模块应该以 `!` 开始。这样确保了当一个不好的模块忘记包含最后的分号时，在合并代码到生产环境后不会产生错误。详细说明
- 文件应该以驼峰式命名，并放在同名的文件夹里，且与导出的名字一致。
- 增加一个名为 `noConflict()` 的方法来设置导出的模块为前一个版本并返回它。
- 永远在模块顶部声明 `'use strict';`。

```
// fancyInput/fancyInput.js

!function (global) {
  'use strict';

  var previousFancyInput = global.FancyInput;

  function FancyInput(options) {
    this.options = options || {};
  }

  FancyInput.noConflict = function noConflict() {
    global.FancyInput = previousFancyInput;
    return FancyInput;
  };

  global.FancyInput = FancyInput;
}(this);
```

jQuery

- 使用 `$` 作为存储 jQuery 对象的变量名前缀。

```
// bad
var sidebar = $('.sidebar');

// good
var $sidebar = $('.sidebar');
```

- 缓存 jQuery 查询。

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

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

// good
function setSidebar() {
  var $sidebar = $('.sidebar');
  $sidebar.hide();
```

```
  // ...stuff...

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

- 对 DOM 查询使用层叠 `$('.sidebar ul')` 或 父元素 > 子元素 `$('.sidebar > ul')`。 jsPerf
- 对有作用域的 jQuery 对象查询使用 `find`。

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

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

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

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

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

测试

- **Yup.**

```
function () {
  return true;
}
```

资源

推荐阅读

- Annotated ECMAScript 5.1

工具

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

其它风格指南

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

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

进一步阅读

- 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
- Eloquent JavaScript - Marijn Haverbeke
- You Don't Know JS - Kyle Simpson

博客

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

播客

- JavaScript Jabber