命名规范

- 当我们为变量, 函数和实例命名时, 使用 camelCase 命名法.

```
// bad
var FOOBar = {};
var foo_bar = {};
function FOOBar () {}

// good
var fooBar = {};
function fooBar () {}
```

- 当我们为类或者模块命名时, 使用 PascalCase 命名法.

```
// bad
var foobar = cc.Class({
    foo: 'foo',
    bar: 'bar',
});
var foobar = require('foo-bar');

// good
var FooBar = cc.Class({
    foo: 'foo',
    bar: 'bar',
});
var FooBar = require('foo-bar');
```

- 使用前置下划线 `_` 当我们为私有属性命名

```
// bad
this.__firstName__ = 'foobar';
this.firstName_ = 'foobar';

// good
this._firstName = 'foobar';
```

- 文件名我们采用 dash 命名法

```
// bad
fooBar.js
FooBar.js

// good
foo-bar.js
```

语法规范

- 使用 `{}` 创建一个字典

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

// bad
var map = Object.create(null);

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

- 使用 `[]` 创建一个数组

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

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

- 尽可能在 js 代码中使用单引号 `''` 来定义 string

```
// bad
var str = "Hello World";

// good
var str = 'Hello World';
```

- 多行 string 定义时, 尽可能使用 `+` 定义

```
// bad
const 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
const 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
const 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.';
```

- 使用 `===` 和 `!==` 而不是 `==` 和 `!=`

书写规范

- 根据个人习惯, 和原代码作者格式, 选择 4 个空格或者 2 个空格作为缩进

```
// bad
function() {
•var name;
}

// very bad
function() {
••<tab>••var name;
}

// good
function() {
••var name;
}

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

- 行尾不要留有空格，文件底部请留一个空行

```
// bad
function () {•
••••var name;•
}
/* EOF */

// good
function () {
••••var name;
}

/* EOF */
```

- 语句结尾请加 `;`

```
// bad
proto.foo = function () {
}

// good
proto.foo = function () {
};

// very bad
function foo () {
```

```
    return 'test'
}

// good
function foo () {
    return 'test';
}

// bad
function foo () {
};

// good, 这里不是语句结尾
function foo () {
}
```

- 尽可能将 `{` 和表达式放在同一行

```
// bad
if ( isFoobar )
{
}

// good
if ( isFoobar ) {
}

// bad
function foobar()
{
}

// good
function foobar() {
}

// bad
var obj =
{
    foo: 'foo',
    bar: 'bar',
}

// good
var obj = {
    foo: 'foo',
    bar: 'bar',
}
```

- 在 `{` 前请空一格

```
// bad
```

```
if (isJedi){
    fight();
}
else{
    escape();
}

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

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

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

- 在逻辑状态表达式（`if`，`else`，`while`，`switch`）后请空一格

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

// good
if (isJedi) {
    fight();
}
else {
    escape();
}
```

- 二元、三元运算符的左右请空一格

```
// bad
var x=y+5;
var left = rotated? y: x;

// good
```

```
var x = y + 5;
var left = rotated ? y : x;

// bad
for (let i=0; i< 10; i++) {
}

// good
for (let i = 0; i < 10; i++) {
}
```

- 一些函数的声明方式，尽量有空格分隔，保持代码清晰度：

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

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

// bad
function divisibleFunction () {
    return DEBUG ? 'foo' : 'bar';
}

// good
var divisibleFunction = DEBUG ?
    function () {
        return 'foo';
    } :
    function () {
        return 'bar';
    };

// bad
function test(){
}

// good
function test () {
}

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

// good
var obj = {
```

```
    foo () {
    }
};

// bad
array.map(x=>x + 1);
array.map(x => {
    return x + 1;
});

// good
array.map(x => x + 1);
```

- 在 代码块定义之间请空一行

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

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

return baz;

// bad
const obj = {
    x: 0,
    y: 0,
    foo() {
    },
    bar() {
    },
};
return obj;

// good
const obj = {
    x: 0,
    y: 0,

    foo() {
    },

    bar() {
    },
};

return obj;
```

- 不要使用前置逗号定义

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

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

// bad
var hero = {
      firstName: 'Ada'
    , lastName: 'Lovelace'
    , birthYear: 1815
    , superPower: 'computers'
};

// good
var hero = {
    firstName: 'Ada',
    lastName: 'Lovelace',
    birthYear: 1815,
    superPower: 'computers',
};
```

- 单行注释请在斜杠后面加一个空格

```
//bad
// good
```

- 多行注释写法

```
/*
 * good
 */
```

- 需要导出到 API 文档的多行注释写法

```
/**
 * good
 */
```

其他杂项：

- var通常声明全局变量，尽可能多地使用let。
- const为常量，常量命名方式全大写分隔形式：

```
const A_CONST_VAL = 100;
```

- 对于bool型函数或变量，使用is,has,can等等表示判断的单词为前缀：

```
let isDead = false;

function isGameOver() {return false;}

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

- 用对象展开操作符浅复制对象，优先于 `Object.assign` 。使用对象剩余操作符来获得一个省略某些属性的新对象。

```
// very bad
const original = { a: 1, b: 2 };
const copy = Object.assign(original, { c: 3 }); //  `original` 是可变的 ʕ·ᴥ·ʔ
delete copy.a; // so does this
// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }
// good
const original = { a: 1, b: 2 };
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }
const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

- 使用数组展开操作符 `...` 复制数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;
for (i = 0; i < len; i += 1) {
itemsCopy[i] = items[i];
}
// good
const itemsCopy = [...items];
```

- 使用展开操作符 `...` 代替 Array.from，来将一个类数组(array-like) 对象转换成数组。

```
const foo = document.querySelectorAll('.foo');
// good
const nodes = Array.from(foo);
// best
const nodes = [...foo];

// bad
const baz = [...foo].map(bar);
// good
const baz = Array.from(foo, bar);
```

- 当访问和使用对象的多个属性时，请使用对象解构.解构可以在你建这些属性的临时引用时，为你节省时间

```
// bad
function getFullName(user) {
const firstName = user.firstName;
const lastName = user.lastName;
return `${firstName} ${lastName}`;
}
// good
function getFullName(user) {
const { firstName, lastName } = user;
return `${firstName} ${lastName}`;
}
// best
function getFullName({ firstName, lastName }) {
return `${firstName} ${lastName}`;
}


const arr = [1, 2, 3, 4];
// bad
const first = arr[0];
const second = arr[1];
// good
const [first, second] = arr;
```

- 使用对象解构来实现多个返回值，而不是数组解构

```
// bad
function processInput(input) {
// 那么奇迹发生了
return [left, right, top, bottom];
}
// 调用者需要考虑返回数据的顺序
const [left, __, top] = processInput(input);
// good
function processInput(input) {
// 那么奇迹发生了
return { left, right, top, bottom };
```

```
}
// 调用者只选择他们需要的数据
const { left, top } = processInput(input);
```

- 以编程方式构建字符串时，请使用模板字符串而不是字符串连接。 模板字符串为你提供了更好的可读性，简洁的语法，正确的换行符和字符串插值功能

```
// bad
function sayHi(name) {
    return 'How are you, ' + name + '?';
}
// bad
function sayHi(name) {
    return ['How are you, ', name, '?'].join();
}
// bad
function sayHi(name) {
    return `How are you, ${ name }?`;
}
// good
function sayHi(name) {
    return `How are you, ${name}?`;
}
```

- 函数中，不要使用 `arguments`。可以选择 rest 语法 `...` 替代。

```
// bad
function concatenateAll() {
    const args = Array.prototype.slice.call(arguments);
    return args.join('');
}
// good
function concatenateAll(...args) {
    return args.join('');
}
```

- 使用默认参数语法，而不要使用一个变化的函数参数。

```
// really bad
function handleThings(opts) {
    // 不! 我们不应该改变函数参数。
    // 更加糟糕: 如果参数 opts 是 falsy(假值) 的话，它将被设置为一个对象，
    // 这可能是你想要的，但它可以引起一些小的错误。
    opts = opts || {};
    // ...
}
```

```
// still bad
function handleThings(opts) {
    if (opts === void 0) {
        opts = {};
    }
    // ...
}
// good
function handleThings(opts = {}) {
    // ...
}
```

- 始终将默认参数放在最后。

```
// bad
function handleThings(opts = {}, name) {
// ...
}
// good
function handleThings(name, opts = {}) {
// ...
}
```

- 优先使用展开运算符 `...` 来调用可变参数函数。它更清洁，你不需要提供一个上下文，而且你不能轻易地实用 `apply` 和 `new`。

```
// bad
const x = [1, 2, 3, 4, 5];
console.log.apply(console, x);
// good
const x = [1, 2, 3, 4, 5];
console.log(...x);
// bad
new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));
// good
new Date(...[2016, 8, 5]);
```