目录

TSG前端开发规范

TSG前端开发规范

TSG前端组Javascript开发规范

- 本规范用来指导前端开发人员用更合理的方式写 JavaScript，保证前端代码的可读写以及可维护性。
- 本规范基于ES6,用于约束ES6代码的编写

TSG前端开发规范

基本类型

本章包含如下内容

TSG前端开发规范

类型

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

```
const foo = 1;
let 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`

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

bar[0] = 9;

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

- 判断具体类型

```
const array = [];
const nodes = document.querySelectorAll("h1");

const getType = (v) => Object.prototype.toString.call(v);

console.log(getType(array)) // => "[object Array]"

console.log(getType(nodes)) // => "[object NodeList]"

console.log(typeof(nodes)) // => "object"
```

注意： 通过 `typeof` 也可以获取变量的类型，但不是太准确，酌情使用

TSG前端开发规范

- null vs undefined

  `null`是一个对象，`undefined`是`window`的一个属性。

```
typeof(null) // => "object"
typeof(undefined) // => "undefined"

const getType = (v) => Object.prototype.toString.call(v);

getType(null) // => "[object Null]"
getType(undefined) // => "[object undefined]"

console.log( null == undefined ) // => true
console.log( null === undefined ) // => true
```

TSG前端开发规范

对象

- 使用字面值创建对象。

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

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

- 不要使用保留字作为键名。

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

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

- 使用同义词替换需要使用的保留字。

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

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

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

- 创建有动态属性名的对象时，使用可被计算的属性名称。 `` javascript function getKey(k) { return a key named ${k}`; }

TSG前端开发规范

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

// good const obj = { id: 5, name: 'San Francisco',

```
[getKey('enabled')]: true
```

};

```
- 使用对象属性值的简写。

```javascript
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 在对象属性声明前把简写的属性分组。

```
    const anakinSkywalker = 'Anakin Skywalker';
    const lukeSkywalker = 'Luke Skywalker';

    // bad
    const obj = {
        episodeOne: 1,
        twoJedisWalkIntoACantina: 2,
        lukeSkywalker,
        episodeThree: 3,
        mayTheFourth: 4,
        anakinSkywalker,
    };

    // good
    const obj = {
        lukeSkywalker,
        anakinSkywalker,
        episodeOne: 1,
        twoJedisWalkIntoACantina: 2,
        episodeThree: 3,
```

TSC前端开发规范
```

```
        mayTheFourth: 4,
    };
```

TSG前端开发规范

数组

- 使用直接量创建数组。

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

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

- 向数组增加元素时使用 Array#push 来替代直接赋值。

```
const someStack = [];

// bad
someStack[someStack.length] = 'something string';

// good
someStack.push('something string');
```

- 使用拓展运算符 `...` 复制数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

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

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

- 使用 Array#slice 将类数组对象转换成数组。

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

TSG前端开发规范

TSG前端开发规范

字符串

- 使用单引号 `''` 包裹字符串。

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

// good
const name = 'Bob Parr';

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

// good
const 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。

```
// 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.';
```

- 程序化生成字符串时，使用模板字符串代替字符串连接。

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

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

TSC前端开发规范

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

TSG前端开发规范

函数

- 函数表达式：

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

// 命名函数表达式
const 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，然后把那个函数赋给一个变量。浏览器允许你这么做，但它们的解析表现不一致。

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

// good
let 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...
}
```

TSG前端开发规范

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

- 尽量少用 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('');
  }
```

- 直接给函数的参数指定默认值，不要使用一个变化的函数参数。

```
// bad
function handleThings(opts) {
  opts = opts || {};
  // ...
}

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

- 直接给函数参数赋值时需要避免副作用。

```
var b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // => 1
count();  // => 2
count(3); // => 3
count();  // => 3
```

TSG前端开发规范

TSG前端开发规范

语法相关

本章包含如下内容

TSG前端开发规范

变量

- `let` vs `var`

```
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4; // The scope is inside the if-block
  var b = 1; // The scope is inside the function

  console.log(a);  // 4
  console.log(b);  // 1
}

console.log(a); // 5
console.log(b); // 1
```

- 直使用 const 来声明变量，避免产生全局变量

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

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

- 使用 const 声明每一个变量。

  > 为什么？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
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

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

TSG前端开发规范

- 将所有的 const 和 let 分组

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

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

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

TSG前端开发规范

属性访问

- 使用 `.` 来访问对象的属性。

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

// bad
const isJedi = luke['jedi'];

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

- 当通过变量访问属性时使用中括号 `[]` 。

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

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

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

TSC前端开发规范

变量提升

- `var` 声明会被提升至该作用域的顶部，但它们赋值不会提升。`let` 和 `const` 被赋予了一种称为「暂时性死区（Temporal Dead Zones, TDZ）」的概念。这对于了解为什么 type of 不再安全相当重要。

```
// 我们知道这样运行不了
// （假设 notDefined 不是全局变量）
function example() {
    console.log(notDefined); // => throws a ReferenceError
}

// 由于变量提升的原因，
// 在引用变量后再声明变量是可以运行的。
// 注：变量的赋值 `true` 不会被提升。
function example() {
    console.log(declaredButNotAssigned); // => undefined
    var declaredButNotAssigned = true;
}

// 编译器会把函数声明提升到作用域的顶层，
// 这意味着我们的例子可以改写成这样：
function example() {
    let declaredButNotAssigned;
    console.log(declaredButNotAssigned); // => undefined
    declaredButNotAssigned = true;
}

// 使用 const 和 let
function example() {
    console.log(declaredButNotAssigned); // => throws a ReferenceError
    console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
    const declaredButNotAssigned = true;
}
```

- 匿名函数表达式的变量名会被提升，但函数内容并不会。

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

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

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

TSG前端开发规范

```
}
```

- 命名的函数表达式的变量名会被提升，但函数名和函数函数内容并不会。

```
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');
    };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
    console.log(named); // => undefined

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

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

- 函数声明的名称和函数体都会被提升。

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

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

ISC前端开发规范

比较运算符 & 等号

- 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=` 。
- 条件表达式的强制转换规则：

  1. 对象被计算为 **true**
  2. **undefined**被计算为 **false**
  3. **null**被计算为 false
  4. 布尔值被计算为布尔的值
  5. 数字如果是**+0**、**-0**、或 **NaN** 被计算为 **false**, 否则为 **true**
  6. 字符串如果是空字符串 `''` ,则被计算为**false**，否则为**true**

```
if ([0]) {
    // true
    // An array is an object, objects evaluate to true
}
```

- 使用简写。

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

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

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

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

- 想了解更多信息，参考 Angus Croll 的Truth Equality and JavaScript。

TSG前端开发规范

TSG前端开发规范

代码块

- 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();
}
```

TSG前端开发规范

TSG前端开发规范

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
const active = true;  // is current tab

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

// bad
function getType() {
    console.log('fetching type...');
    // set the default type to 'no type'
```

TSG前端开发规范

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

    return type;
}

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

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

    return type;
}
```

- 使用 `// FIXME`：标注问题。

```
class Calculator {
  constructor() {
    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 使用 `// TODO`：标注问题需要改进或者优化。

```
class Calculator {
constructor() {
 // TODO: total should be configurable by an options param
 this.total = 0;
}
}
```

TSC前端开发规范

空格

空格主要用来格式化代码，保证代码的一致性和可阅读性，一般的IDE都可以直接格式化代码。在WebStorm中，我们可以同时按住 `Ctrl + ALT + L` 来格式代码。

- 使用**4**个空格作为缩进。

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

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

- 在花括号前放一个空格。

```
// 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
```

TSG前端开发规范

```
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
const x=y+5;

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

- 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
const obj = {
    foo() {
    },
    bar() {
```

TSG前端开发规范

```
    }
};
return obj;

// good
const obj = {
    foo() {
    },

    bar() {
    }
};

return obj;
```

- 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();
```

TSG前端开发规范

逗号 & 分号

- 不要在行首出现逗号

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

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

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

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

- 在语句的介绍或者匿名函数的头部加入分号

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

// good
(() => {
    const name = 'Skywalker';
```

ITSG前端开发规范

```
    return name;
})();

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

TSG前端开发规范

ES6相关

本章包含如下内容：

TSG前端开发规范

箭头函数

- 当你必须使用函数表达式（或传递一个匿名函数）时，使用箭头函数符号。

箭头函数会创造一个新的 `this` 执行环境（参考Arrow Functions -MDN），同时会使代码更简洁。

```
// bad
[1, 2, 3].map(function (x) {
    return x * x;
});


// good
[1, 2, 3].map((x) => {
return x * x;
});
```

- 如果一个函数适合用一行写出并且只有一个参数，那就把花括号、圆括号和 `return` 都省略掉。如果不是，那就不要省略。

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

// good
[1, 2, 3].reduce((total, n) => {
    return total + n;
}, 0);
```

TSG前端开发规范

类与构造函数

- 总是使用 `class` 。避免直接操作 `prototype` 。

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


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

- 使用 `extends` 继承。

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

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

TSG前端开发规范

- 方法可以返回 `this` 来帮助链式调用。

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

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

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

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

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

const luke = new Jedi();

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

TSG前端开发规范

模块

- 总是使用 `import/export` 进行模块化导入或者导出，而不是其他非标准模块系统。

```
 // bad
const MigStyleGuide = require('./MigbnbStyleGuide');
module.exports = MigStyleGuide.es6;

// ok
import MigStyleGuide from './MigStyleGuide';
export default MigStyleGuide.es6;

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

- 不要从 `import` 中直接 `export`，让两者各司其职。

```
// bad
// filename es6.js
export { es6 as default } from './MigStyleGuide';

// good
// filename es6.js
import { es6 } from './MigStyleGuide';
export default es6;
```

TSG前端开发规范

迭代器

- 使用高阶函数例如 `map()` 和 `reduce()` 替代 `for-of`

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

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

sum === 15;

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

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;
```

TSG前端开发规范

解构

- 使用解构存取多属性对象。

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

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

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

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

- 对数组使用解构赋值。

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

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

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

- 需要回传多个值时，使用对象解构，而不是数组解构。

> 为什么？这样可以保证增加属性或者改变排序时，不会改变方法调用时的位置。

```
// bad
function processInput(input) {
    // then a miracle occurs
    return [left, right, top, bottom];
}

// 调用时需要考虑回调数据的顺序。
```

FSG前端开发规范

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

// good
function processInput(input) {
    // then a miracle occurs
    return { left, right, top, bottom };
}

// 调用时只选择需要的数据
const { left, right } = processInput(input);
```

TSG前端开发规范

基本类型

本章包含如下内容

TSG前端开发规范

类型转换

- 字符串类型转换：

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

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

// good
const totalScore = '' + this.reviewScore

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

- 对数字使用 parseInt 转换，并带上类型转换的基数。

```
const inputValue = '4';

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

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

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

- 布尔值类型转换

```
const age = 0;

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

// good
```

TSC前端开发规范

```
const hasAge = Boolean(age);

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

TSG前端开发规范

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
const OBJEcttssss = {};
const this_is_my_object = {};
function c() {}

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

- 使用Pascal式命名构造函数或类。

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

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

// good
class User {
    constructor(options) {
        this.name = options.name;
    }
}

const good = new User({
    name: 'yup',
```

FSG前端开发规范

```
});
```

- 使用下划线 `_` 开头命名私有属性。

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

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

- 别保存 `this` 的引用。使用箭头函数或 `Function#bind` 。

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

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

// good
function foo() {
    return () => {
        console.log(this);
    };
}
```

- 如果文件只输出一个类，文件名必须和类名完全保持一致。

```
// file contents
class CheckBox {
    // ...
}

export default CheckBox;

// in some other file
```

TSG前端开发规范

```
// bad
import CheckBox from './checkBox';

// bad
import CheckBox from './check_box';

// good
import CheckBox from './CheckBox';
```

- 导出默认的函数时使用驼峰式命名

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 导出单例、函数库、空对象时使用帕斯卡式命名。

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

export default MigStyleGuide;
```

TSG前端开发规范

属性访问和修改

- 属性读取函数和设置函数使用 `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。

```
class Jedi {
  constructor(options = {}) {
    const lightsaber = options.lightsaber || 'blue';
    this.set('lightsaber', lightsaber);
  }

  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

TSG前端开发规范

TSG前端开发规范

事件处理

- 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,这样会使程序具有更好的扩展性。

```
// 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
});
```

TSG前端开发规范

jQuery最佳实践

- 使用 `$` 作为 `jQuery` 对象变量名的前缀。

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

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

- 缓存 jQuery 查询。

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

    // ...stuff...

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

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

    // ...stuff...

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

- 尽量使用 `jQuery` 的 `find` 方法来获取 `DOM`

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


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

// good
```

TSG前端开发规范

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

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

TSG前端开发规范