

首页 (<http://www.open-open.com/>) 代码 (<http://www.open-open.com/code/>) 文档 (<http://www.open-open.com/doc/>) 问答

全部经验分类

Android (</lib/tag/Android>) iOS (</lib/tag/iOS>) JavaScript (</lib/tag/JavaScript>)

(</lib/list/all>) 

所有分类 (</lib/list/all>) > 开发语言与工具 (</lib/list/36>) > JavaScript开发 (</lib/list/145>)

【译】寻找头绪：编写可维护的 JavaScript

JavaScript (</lib/tag/JavaScript>) 单元测试 (</lib/tag/单元测试>) 2016-05-24 17:20:46 发布

您的评价: 0.0

收藏

0收藏

几乎每个程序员都有接手维护别人遗留项目的经历。或者，有可能一个老项目某一天又被重新启动。通常情况下，接手老项目都会让人恨不得抛弃掉整个代码库从头开始。老代码凌乱、文档缺失、需要研究很多天才能完全搞明白它。然而，通过合适的规划、分解和好的工作流，项目代码可以变得干净、有组织和可扩展。



我曾经接手清理许多项目的代码，让我不得不重头开始的项目真心不多。事实上我学到了很多 JavaScript 相关的内容，学到了如何保持代码库有条理，以及最重要的是冷静，不要为你的前任抓狂。在这篇文章里，我想要让你们知道我是怎么一步步处理项目代码的，告诉你我积累的经验。

分析项目

第一步是先概览整个项目，弄明白问题所在。如果这是一个网站，通过点击测试所有功能点：打开模块、提交表单以及其他的。在做这件事的时候，打开开发者工具，看看是否有任何报错，看看控制台有没有日志。如果这是一个 Node.js 项目，打开 命令行界面 (https://en.wikipedia.org/wiki/Command-line_interface) 然后检查各个 API。最好的情况是项目通过统一的入口（例如：`main.js`，`index.js`，`app.js`，.....）来初始化所有的模块，最差的情况是整个业务逻辑散落于各处。

搞清楚项目采用了哪些工具。是 jQuery (<https://jquery.com/>)、React (<https://facebook.github.io/react/>) 或是 Express (<http://expressjs.com/>)？将所有重要的信息整理在一个清单里。假设这个项目是用 Angular 2 (<https://angular.io/>) 写的，你之前对 Angular 2 不熟悉，先去查看文档，对它有一个初步的了解，并寻找最佳实践的范例。

从更高层面上理解项目

了解技术点是一个好开端，但要进一步深入理解，我们需要看一看它的 单元测试 (https://en.wikipedia.org/wiki/Unit_testing)。单元测试通过测试代码的功能函数与方法来保证代码如预期的那样运行。不同于仅仅阅读代码，查看和运行单元测试能让你更加深入理解代码的期望运行结果。如果接手的项目没写单元测试，没关系，我们可以自己来写。

创建基线

这样做是为了确立一致性。你已经了解了项目的工具链、代码结构、模块的逻辑关系，现在该为项目创建基线了。我推荐添加一个 `.editorconfig` (<http://editorconfig.org/>) 文件让代码风格在不同的编辑器、IED 和不同的开发者之间保持一致。

一致的缩进

使用 `tab` 还是空格来缩进是一个老问题 (<http://programmers.stackexchange.com/questions/57/tabs-versus-spaces-what-is-the-proper-indentation-character-for-everything-in-e>)，常引发程序员争论不休，不过没关系，不管项目用的是空格还是 `tab`，继续使用之前的就好了。除非代码库既有空格缩进又有 `tab` 缩进的代码，那就只好在两者中做出一个取舍。每个人都可以保持自己的观点，但一个好的项目要保证所有的开发者都可以无争议地协同工作。

为什么这个很重要？因为每一个人都会有自己使用编辑器或IDE的习惯。比如，我是个代码折叠控，要是编辑器没有正确的代码折叠功能，我整个人都会迷失在文件里。如果代码的缩进不一致，就会影响到折叠功能，因此每一次我打开一个文件，不得不先修复那些缩进然后才能开始工作，这十分浪费时间。

```
// 虽然这段 JavaScript 代码是合法的，但这段代码块没办法正常折叠
// 因为这段代码的缩进不一致
function foo (data) {
  let property = String(data);

  if (property === "bar") {
    property = doSomething(property);
  }
  //... more logic.
}

// 修复缩进后，这段代码才可以被正确折叠
// 从而获得更好的编码体验和更整洁的代码库
function foo (data) {
  let property = String(data);

  if (property === "bar") {
    property = doSomething(property);
  }
  //... more logic.
}
```

命名

确保项目中使用命名约定是值得推崇的做法。驼峰命名 (<https://en.wikipedia.org/wiki/CamelCase>) 通常在 JavaScript 代码中使用，但是我看到过许多不一致的命名。例如：`jQuery`项目经常会有代码在命名上混淆`jQuery`对象和其他对象。

```
// 不一致的命名让代码变得难以检查和理解
// 它还会误导维护者
const $element = $(".element");

function _privateMethod () {
  const self = $(this);
  const _internalElement = $(".internal-element");
  let $data = element.data("foo");
  //... more logic.
}

// 这样改就更易于理解了
const $element = $(".element");

function _privateMethod () {
  const $this = $(this);
  const $internalElement = $(".internal-element");
  let elementData = $element.data("foo");
  //... more logic.
}
```

代码检查

之前所做的一切是在美化代码，主要是让它变得更易于检查。接下来我们介绍保证代码质量的通用最佳实践。`ESLint` (<http://eslint.org/>)，`JSLint` (<http://www.jslint.com/>)，还有 `JSHint` (<http://jshint.com/>) 是目前最受欢迎的三个 JavaScript 代码检查工具。我个人用 `JSHint` 最多，但我现在最喜欢 `ESLint`，主要是因为它可以自定义规则，也较早

地支持了 ES2015。

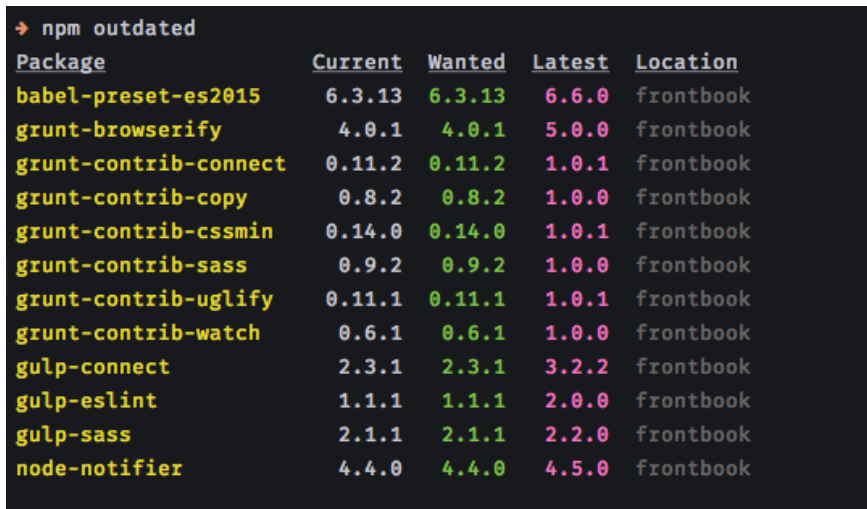
一旦你开始代码检查，如果有很多错误信息出现，立即修复它们。别跳过这些步骤，直到你的代码检查工具对你的代码彻底满意了。

升级依赖

升级依赖需要仔细些，如果你不注意依赖本身的升级带来的一些变化，就很容易导致错误。一些项目可能只能依赖某些库的固定版本（例如：`v1.12.5`），而另一些则使用版本通配符（例如：`v1.12.x`）。如果你要快速升级依赖，你需要知道依赖模块的版本号通常按如下规则建立：主版本.小版本.补丁。如果你对 `semantic versioning` (<http://semver.org/>) 的方式不熟悉，我推荐你先阅读 Tim Oxley 的这篇文章 (<https://nodesource.com/blog/semver-a-primer/>)。

升级依赖没有通用方法。每个项目是不一样的，需要区别对待。升级依赖的补丁版本通常不会出什么问题，小版本一般也还OK。但如果你要升级依赖的主版本，你就需要仔细检查版本升级带来的改变。有可能API完全改变了，那样你就得重写你项目的一大堆代码。如果非必要，我一般避免将依赖升级到下一个主版本。

如果你的项目使用 `npm` (<https://www.npmjs.com/>) 来管理依赖，你可以很方便地使用 `npm outdated` 命令来检查你的依赖是否已经过时了。我用一个项目 `FrontBook` (<https://github.com/morkro/FrontBook>) 来举例说明，在这个项目里，我经常升级所有的依赖：



→ npm outdated				
Package	Current	Wanted	Latest	Location
babel-preset-es2015	6.3.13	6.3.13	6.6.0	frontbook
grunt-browserify	4.0.1	4.0.1	5.0.0	frontbook
grunt-contrib-connect	0.11.2	0.11.2	1.0.1	frontbook
grunt-contrib-copy	0.8.2	0.8.2	1.0.0	frontbook
grunt-contrib-cssmin	0.14.0	0.14.0	1.0.1	frontbook
grunt-contrib-sass	0.9.2	0.9.2	1.0.0	frontbook
grunt-contrib-uglify	0.11.1	0.11.1	1.0.1	frontbook
grunt-contrib-watch	0.6.1	0.6.1	1.0.0	frontbook
gulp-connect	2.3.1	2.3.1	3.2.2	frontbook
gulp-eslint	1.1.1	1.1.1	2.0.0	frontbook
gulp-sass	2.1.1	2.1.1	2.2.0	frontbook
node-notifier	4.4.0	4.4.0	4.5.0	frontbook

如你所见，我这个项目里的依赖有很多主版本升级。我不会一次将他们全部升级，但是会一次升级一个。虽然这会耗费许多时间，但这是确保不会出问题的唯一办法（尤其是如果这个项目没有任何测试）。

下面该干脏活了

我必须让你知道的非常重要的一点是，清理代码并不意味着需要移除和重写大量的代码片段。当然，有时候这可能是唯一的解决办法，但是这不应该是你的首选方案。`JavaScript` 特别灵活，因此难以给出一般性的建议，通常情况下你必须对症下药。

建立单元测试

单元测试能保证你理解代码是如何工作的，这样避免一些意外导致错误。`JavaScript` 单元测试的内容足够写另一篇文章，所以我在这里不能详细介绍。目前被广泛使用的单元测试框架有 `Karma` (<http://karma-runner.github.io/0.13/index.html>)、`Jasmine` (<http://jasmine.github.io/>)、`Mocha` (<http://mochajs.org/>) 以及 `Ava` (<https://github.com/sindresorhus/ava>)。如果你还要测试你的用户界面，`Nightwatch.js` (<http://nightwatchjs.org/>) 和 `DalekJS` (<http://dalekjs.com/>) 是适合浏览器自动化测试的工具。

单元测试和浏览器自动化测试的区别是，前者测试你的 `JavaScript` 代码本身，来确保你所有的模块和主要逻辑运行无误。后者，测试用户界面，确保界面元素在正确的位置，且如预期地工作。

在你开始动手重构代码之前，认真对待单元测试，那样你的项目的稳定性将得到改善，而你甚至还没有开始考虑可扩展性。单元测试带来的另一个好处是你不再需要无时无刻担心你的改动会无意中破坏原有功能。

Rebecca Murphey 写了一篇很棒的文章关于 如何为现有代码写单元测试 (<https://rmurphey.com/blog/2014/07/13/unit-tests>)。

架构

`JavaScript` 架构是另一个大话题。重构和清理架构归结于你在这方面积累了多少经验。我们可以选择许多不同的设计模式 (<https://addyosmani.com/resources/essentialjsdesignpatterns/book/>)，但是不是所有的模式都适合于提升可扩展性。限于篇幅，我不能涵盖所有模式，但我至少可以给你一些通用的建议。

首先，你需要找出哪些设计模式在你的项目中已经使用到了。阅读有关这些模式的部分，确保它们在项目中使用上保持一致性。可扩展性的关键之一便是坚持一致的模式，避免混搭。当然，你可以针对项目中的不同目的采用不同的设计模式（例如，将单例模式

(<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#singletonpatternjavascript>) 用于数据结构和短命名空间的辅助功能函数，以及将观察者模式

(<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript>) 用于与模块），但是别在一个模块上使用一种设计模式，对另一个模块又用另一种不同的设计模式。

如果你的项目没有任何架构（可能一切都堆在一个巨大的 `app.js` 文件里），从现在开始让它有架构。不过别打算一口吃成胖子，需要一点一点来。再次强调，没有对任何项目都适用的万精油方案，每一个项目的情况都是不同的。根据规模和复杂度不同，项目文件目录结构各有不同。通常，最基本的原则是，目录结构应当将第三方库、模块、数据以及负责初始化模块与逻辑的入口文件（比如：`index.js`、`main.js`）分开来。

简而言之就是 模块化。

将一切模块化？

模块化不是解决 JavaScript 扩展性问题的唯一选择。模块化增加了一层 API，这一层对开发者来说并不陌生。这些工作虽然麻烦，但是值得去做的。模块化的基本原则是将所有功能拆分为小模块。这么做了以后，不仅让你更容易解决代码里的问题，也让项目组的其他成员更容易协同工作。每个模块只做一件事，它们不用关心外部逻辑，可以被复用在不同的地方。

如何将一大堆功能拆分成许多逻辑关联的小模块？让我们来做做看：

```
// 这个例子使用 Fetch API 来请求一个服务器的 API
// 让我们假设它返回一个 JSON 文件，包含一些基本信息
// 然后我们创建一个新的元素，统计 json 所有属性的
// content 字段中的字符数，然后将结果插入 DOM 的某个位置。
fetch("https://api.somewebsite.io/post/61454e0126ebb8a2e85d", { method: "GET" })
  .then(response => {
    if (response.status === 200) {
      return response.json();
    }
  })
  .then(json => {
    if (json) {
      Object.keys(json).forEach(key => {
        const item = json[key];
        const count = item.content.trim().replace(/\s+/gi, "").length;
        const el = `
        <div class="foo-${item.className}">
          <p>Total characters: ${count}</p>
        </div>
        `;
        const wrapper = document.querySelector(".info-element");

        wrapper.innerHTML = el;
      });
    }
  })
  .catch(error => console.error(error));
```

上面的代码不是模块化的。所有的功能都耦合在一起。想象一下，如果这是更复杂的函数，由于出了一些错误你必须调试它们，可能 API 没返回，可能某些原因 JSON 内部的值被改变或者别的什么问题。调试这一大坨代码如同噩梦般，不是吗？

让我们将代码按不同的职责拆分开来：

```
// 在前一个例子里，有一个功能是统计字符串的字符数
// 让我们将它单独抽出来成为一个模块
function countCharacters (text) {
  const removeWhitespace = /\s+/gi;
  return text.trim().replace(removeWhitespace, "").length;
}

// 这一部分，我们也独立成一个模块，使用 DOM API 来创建 HTML
// 取代之前直接插入字符串的做法
function createWrapperElement (cssClass, content) {
  const className = cssClass || "default";
  const wrapperElement = document.createElement("div");
  const textElement = document.createElement("p");
  const textNode = document.createTextNode(`Total characters: ${content}`);

  wrapperElement.classList.add(className);
  textElement.appendChild(textNode);
  wrapperElement.appendChild(textElement);

  return wrapperElement;
}

// 前一个例子 .forEach 中的匿名函数也被我们抽出来形成一个模块
function appendCharacterCount (config) {
  const wordCount = countCharacters(config.content);
  const wrapperElement = createWrapperElement(config.className, wordCount);
  const infoElement = document.querySelector(".info-element");

  infoElement.appendChild(wrapperElement);
}
```

好了，我们现在有了三个新模块，让我们看看重构之后的 `fetch` 调用：

```
fetch("https://api.somewebsite.io/post/61454e0126ebb8a2e85d", { method: "GET" })
  .then(response => {
    if (response.status === 200) {
      return response.json();
    }
  })
  .then(json => {
    if (json) {
      Object.keys(json).forEach(key => appendCharacterCount(json[key]))
    }
  })
  .catch(error => console.error(error));
```

当然我们还可以进一步将 `.then()` 中的逻辑也抽出来形成模块，不过我想我已经充分表达了模块化的含义。

如果不想模块化呢？

如我前面提到的，将你的代码拆成小模块会增加额外的一层 **API**。如果你不想这么做，但是又想要让代码易于与其他开发者一起维护，不拆函数也没问题，你依然可以将你的代码分解成更简单的部分并把重点放在可测试的代码上。

为你的代码撰写文档

文档化是一个老生常谈的话题。程序员社区的一部分人提倡将一切文档化，而另一部分人认为好代码即是文档 (<http://stackoverflow.com/questions/209015/what-is-self-documenting-code-and-can-it-replace-well-documented-code>)。我奉行中庸之道，我觉得代码的可读和可扩展之间需要保持平衡。你可以使用 **JSDoc** (<http://usejsdoc.org/>) 来帮助你实现文档化。

JSDoc 是一个 **JavaScript** 的 **API** 文档生成器。常用的编辑器和 **IDE** 都有支持它的插件。我们看一个例子：

```
function properties (name, obj = {}) {
  if (!name) return;
  const arr = [];

  Object.keys(obj).forEach(key => {
    if (arr.indexOf(obj[key][name]) <= -1) {
      arr.push(obj[key][name]);
    }
  });

  return arr;
}
```

这个函数有两个参数，遍历一个对象，返回一个数组。这也许不是一个过于复杂的方法，但是对于没写过这段代码的人来说，搞懂它还是有点费劲。此外，这个方法具体的作用也不是很明确。让我们对它文档化：

```
/** * 遍历一个对象，将所有属性对象的 "name" push 到一个新数组中 * 如果有重复，只 push 一次 * @param {String} propertyName - 属性的名字 * @param {Object} obj - 你想要遍历的对象 * @return {Array} */
function getArrayOfProperties (propertyName, obj = {}) {
  if (!propertyName) return;
  const properties = [];
  Object.keys(obj).forEach(child => {
    if (properties.indexOf(obj[child][propertyName]) <= -1) {
      properties.push(obj[child][propertyName]);
    }
  });
  return properties;
}
```

我没有改变任何代码，只是改了一下函数名，添加了一段简短的注释，就已经让这段代码的可读性变得好些了。

有条理的代码提交 workflow

重构本身是一项艰巨的任务。为了能随时回滚你的修改（假如你破坏了一些原有功能，过了一会才意识到，你可能就需要回滚代码到之前版本），你的每一部分修改，比如重写了一个方法、重命名了一个名字空间，都应该及时提交到 git（或者 svn）。这么做也许会让你觉得麻烦，但这么做有助于让你的清理工作更有条理。

为你的重构工作开一个新的分支，千万别总是在主线 (master) 上改。因为主线版本你有可能需要临时做一些更新或者随时发布一些 bug fixes 到线上环境，而你又不能将你没有经过测试的和未完成的重构一同发布到线上，因此我建议你还是应该在不同的分支上工作。

在 GitHub 上有一份 有趣的指导 (<https://guides.github.com/introduction/flow/>)，是关于如何使用他们的版本控制流程的。

别失去理智

除了用技术解决问题之外，有一个很重要的步骤很少被人提及：别为你的前任抓狂。我无意指责任何人，但是我知道一些人经历过这种情况。我花了很多年的时间去理解和克服心理上的不爽。我曾经对前任开发者们留下的代码、解决方案感到有些抓狂，他们做的一切在我眼里看来都造成混乱。

结果，这些消极情绪没带给我任何好处。消极情绪会导致你过度重构，浪费你的时间，而且可能破坏一些原有功能，而这一切又导致你越来越恼怒。你可能会花费额外的时间去重写一个本来毫无问题的模块，没有人会因此感谢你，因为你在做无用功。先分析状况，然后做有价值的重构。在任何时候，你随时可以对一个模块做一些细节的改进。

一段代码为什么写成这样往往是有历史原因的，也许前任程序员没有足够的时间将代码写得足够好、或者不知道有更好的写法，或者别的什么原因。我们都是过来人。

整理一下

让我们从头回顾一下所有的步骤，为你的下一个项目创建一个 checklist：

1. 分析项目

- 先忘掉自己的开发者身份，以一个用户的身份来看清它的全貌。
- 浏览代码库，列出项目使用的工具。
- 阅读项目相关工具的文档和最佳实践。

- 浏览单元测试，从更高层面上了解项目。

2. 创建基线

- 引入 `.editorconfig` (<http://editorconfig.org/>) 以保证在所有的编辑器和 IDE 下保持代码风格一致。
- 使缩进风格一致，至于是用 `tab` 还是空格，无所谓。
- 执行命名约定。
- 如果代码检查工具不存在, 添加一个，可以是 ESLint (<http://eslint.org/>)、JSLint (<http://www.jshint.com/>) 或者 JSHint (<http://jshint.com/>)。
- 升级依赖，但是需要格外小心，弄清楚到底升级了什么。

3. 清理代码

- 建立单元测试与浏览器自动化测试，可以使用一些工具，例如 Karma (<http://karma-runner.github.io/0.13/index.html>)、Jasmine (<http://jasmine.github.io/>)、或者 Nightwatch.js (<http://nightwatchjs.org/>)。
- 确保架构和设计模式保持一致。
- 不要混用设计模式 (<https://addyosmani.com/resources/essentialjsdesignpatterns/book/>)，坚持使用已经存在的设计模式。
- 决定你是否需要将代码库拆分成模块。每一个模块应当具有单一的目的，模块不用关心自身之外的其他逻辑。
- 如果你不想拆分模块，把重点放在可测试的代码上，把它们分解成更简单的代码块。
- 恰当地命名你的函数，为代码适当撰写文档，保持可读和可扩展的平衡。
- 使用 JSDoc (<http://usejsdoc.org/>) 来生成文档。
- 定期提交代码，特别在有重要改变时。这样如果有什么改错了，可以方便回滚。

4. 别失去理智

- 别为你的前任开发者抓狂。负面情绪只会导致过度重构而浪费时间。
- 一段代码为什么写成这样总是有原因的。要牢记我们都是过来人。

我非常希望这篇文章能帮到你。如果你正在为代码重构做这些努力，或者你有一些我没有提到过的好建议，我希望你可以告诉我。

英文原文：[https://www.sitepoint.com/write-maintainable-javascript/?](https://www.sitepoint.com/write-maintainable-javascript/?utm_source=javascriptweekly&utm_medium=email)

[utm_source=javascriptweekly&utm_medium=email](https://www.sitepoint.com/write-maintainable-javascript/?utm_source=javascriptweekly&utm_medium=email) ([https://www.sitepoint.com/write-maintainable-javascript/?](https://www.sitepoint.com/write-maintainable-javascript/?utm_source=javascriptweekly&utm_medium=email)

[utm_source=javascriptweekly&utm_medium=email](https://www.sitepoint.com/write-maintainable-javascript/?utm_source=javascriptweekly&utm_medium=email))

来自：<https://www.h5jun.com/post/untangling-spaghetti-code-writing-maintainable-javascript.html>

(<https://www.h5jun.com/post/untangling-spaghetti-code-writing-maintainable-javascript.html>)

同类热门经验

1. Node.js 初体验 (/lib/view/open1326870121968.html)
2. JavaScript开发规范要求 (/lib/view/open1352263831610.html)
3. 使用拖拉操作来自定义网页界面布局并保存结果 (/lib/view/open1325064347889.html)
4. Nodejs入门学习, nodejs web开发入门, npm、express、socket配置安装、nodejs聊天室开发 (/lib/view/open1329050007640.html)
5. 利用HTML5同时上传多个文件 - resumable.js (/lib/view/open1327591300671.html)
6. nide: 一个不错的Node.js开发工具IDE (/lib/view/open1325834128750.html)

相关文档 — 更多 (<http://www.open-open.com/doc>)

相关经验 — 更多

相关讨论 — 更多 (<http://www.open-open.com/solution>)

- 编写可维护的JavaScript.pdf (<http://www.open-open.com/doc/view/35daed013d5b415e90f358ed2ab83609>)
- jQuery的TDD框架 - QUnit.doc (<http://www.open-open.com/doc/view/0ed9f73ae46e4d9bb4bd3524531d60fa>)
- Test-Driven JavaScript Development.pdf (<http://www.open-open.com/doc/view/daeb9f76b4ad4a7ab29c023d0a9b5c55>)
- 编写可维护的JavaScript
- 浅谈JavaScript编程语言的编码规范 (<http://www.open-open.com/solution/view/1318472833218>)
- 那些年，追过的开源软件和技术

2016/6/3

【译】寻找头绪：编写可维护的 JavaScript - OPEN 开发经验库

• JavaScript 基础2.ppt (<http://www.open-open.com/doc/view/2b30576dea2c44e4a9ee649513d8224b>)

• JavaScript设计模式与开发实践.pdf (<http://www.open-open.com/doc/view/91e78191c05b4966bd5ce9b19a2dab62>)

• 编写javascript独有风格的高质量代码.pdf (<http://www.open-open.com/doc/view/7c96da548d554dc7a8229f2683c9671b>)

• JavaScript 语言精粹.pdf (<http://www.open-open.com/doc/view/bdf08c335cf74ad98ace3a2753628449>)

• 编写高质量代码：web前端开发修炼之道.pdf (<http://www.open-open.com/doc/view/f2f479cf1fb349609faea583ac439bef>)

• JavaScript语言精粹_完整版.pdf (<http://www.open-open.com/doc/view/b358c242ac1b438ca01f6d2ec87580c3>)

• 编写高质量代码-源代码.pdf (<http://www.open-open.com/doc/view/03497cb3e1a24c92ac165b07fb236c0a>)

• 编写高质量代码：Web前端开发修炼之道-源代码.doc (<http://www.open-open.com/doc/view/6bac4cf3fa014e55887604062004674c>)

• 编写高质量代码--web前端开发修炼之道.pdf (<http://www.open-open.com/doc/view/ec443ce436624ffbaac1139f4bf0844a>)

• 精通AngularJS优雅的单页面Web用开发.pdf (<http://www.open-open.com/doc/view/4c3ccb375d9c43ae8e51751a3d90240b>)

• JavaScript 语言精粹（修订版）.pdf (<http://www.open-open.com/doc/view/f78e2634b8ce46e09479d7373936122e>)

• 高效Javascript英文原版.pdf (<http://www.open-open.com/doc/view/386281c6828842a68ca8620ba9716804>)

• Secrets of the JavaScript Ninja.pdf (<http://www.open-open.com/doc/view/158adb44a3624a22afc4686a6182ca22>)

• JavaScript语言精粹(JavaScript.The.Good.Parts) .pdf (<http://www.open-open.com/doc/view/f4f756a28b654da8a97371c3f5519829>)

• JavaScript最佳入门教程(中文版).pdf (<http://www.open-open.com/doc/view/b06f11da0ad844ba9ca6feba07f4f1cc>)

• JavaScript 最佳入门教程(中文版).pdf (<http://www.open-open.com/doc/view/9cd8084a977b4fdaa4967c556be22227>)

• 《JavaScript语言精粹》(JavaScript The Good Parts).pdf (<http://www.open-open.com/doc/view/133f0ccafc12418c99fc78ad0842127c>)

• 解开面条代码：怎样书写可维护JavaScript ([open.com/solution/view/1425959150201](http://www.open-open.com/solution/view/1425959150201))

• 以优美方式编写JavaScript代码 ([open.com/solution/view/1348305460369](http://www.open-open.com/solution/view/1348305460369))

• Dojo 敏捷开发：集成 DOH 单元测试到 Ant build ([open.com/solution/view/1427071752309](http://www.open-open.com/solution/view/1427071752309))

• 码农周刊分类整理 ([open.com/solution/view/1427071752309](http://www.open-open.com/solution/view/1427071752309))

• Web开发人员最喜爱的10款流行JavaScript库 ([open.com/solution/view/1379903308476](http://www.open-open.com/solution/view/1379903308476))

• JavaScript 单元测试工具：Buster.JS 76个JavaScript教程资源免费下载 ([open.com/solution/view/1372818526987](http://www.open-open.com/solution/view/1372818526987))

• JavaScript 单元测试：Jest 再谈JavaScript的数据类型问题 ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• JavaScript 单元测试工具：Venus ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• JavaScript 单元测试框架 DOH ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• ASP.NET 开发人员不必担心 Node 的五大理由 ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• JsMockito - JavaScript 单元测试工具 ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• JavaScript 单元测试框架：Jasmine 初探 ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

• 开发维护大型项目的Java的建议 ([open.com/solution/view/1318472797249](http://www.open-open.com/solution/view/1318472797249))

©2006-2016 深度开源

 (<http://www.open-open.com/>)

浙ICP备09019653号-31

(<http://www.miibeian.gov.cn/>) 站长统计

([http://www.cnzz.com/stat/website.php?](http://www.cnzz.com/stat/website.php?web_id=1257892335)

web_id=1257892335)

<http://www.open-open.com/lib/view/open1464081646045.html>

8/8