在相关代码的上面使用JSDOC注释：

如：

/\*\* @type {HTMLCanvasElement} \*/

const canvas = document.querySelector('#mycanvas');

/\*\* @type {WebGLRenderingContext} \*/

const gl = someLibraryInitWebGL(canvas);

JSDOC学习网址：<https://jsdoc.app/>

JSDoc中文网址：<https://www.shouce.ren/api/view/a/13232>

**JSDoc 介绍**

**什么是JSDoc**

JSDoc是一个根据javascript文件中注释信息，生成JavaScript应用程序或库、模块的API文档 的工具。你可以使用他记录如：命名空间，类，方法，方法参数等。类似JavaDoc和PHPDoc。现在很多编辑器或IDE中还可以通过JSDoc直接或使用插件生成智能提示。从而使开发者很容易了解整个类和其中的属性和方法，并且快速知道如何使用，从而提高开发效率，降低维护成本。

**使用JSDoc**

JSDoc本质是代码注释，所以使用起来非常方便，但是他有一定的格式和规则，只要了解这些，那么后面的事情，比如生产文档，生成智能提示都可以通过工具来完成。

**JSDoc注释**

JSDoc注释一般应该放置在方法或函数声明之前，它必须以/ \*\*开始，以便由JSDoc解析器识别。其他任何以/\*，/\*\*\*或者超过3个星号的注释，都将被JSDoc解析器忽略。例如一下代码：

1. /\*\*
2. \* Book类，代表一个书本.
3. \* @constructor
4. \* @param {string} title - 书本的标题.
5. \* @param {string} author - 书本的作者.
6. \*/
7. function Book(title, author) {
8. this.title=title;
9. this.author=author;
10. }
11. Book.prototype={
12. /\*\*
13. \* 获取书本的标题
14. \* @returns {string|\*}
15. \*/
16. getTitle:function(){
17. return this.title;
18. },
19. /\*\*
20. \* 设置书本的页数
21. \* @param pageNum {number} 页数
22. \*/
23. setPageNum:function(pageNum){
24. this.pageNum=pageNum;
25. }
26. };

**JSDoc注释标签**

看了上面的代码注释是不是一目了然呢，获取你会疑惑上面以@开头的标签是什么意思。 在JSDoc 注释有一套标准的注释标签，一般以@开头。这里解释一下每个标签的意思。

**关于别名**

因为JSDoc也经历好几个版本，所以考虑向后兼容，所以存在别名，另一个方面，别名对于一部分人来说更加直观。 比如@param有两个别名：

* @arg
* @argument

**JSDoc 3中的名称路径**

如果涉及到一个JavaScript变量，这个变量在文档中的其他地方，你必须提供一个唯一标识符映射到该变量。名称路径提供了一种这样做的方法，并且消除实例成员，静态成员和内部变量之间的歧义。

**JSDoc 3 中名称路径的基本语法示例：**

1. myFunction
2. MyConstructor
3. MyConstructor#instanceMember
4. MyConstructor.staticMember
5. MyConstructor~innerMember // note that JSDoc 2 uses a dash

下面给出了例子：一个名为"say"的*实例*方法，一个名为"say"的*内部*函数,和同样名为"say"的*静态*方法。这三种不同的方法,都是彼此独立地存在的。

例如，使用一个文档化标签来描述你的代码：

1. /\*\* @constructor \*/
2. Person = function() {
3. this.say = function() {
4. return "I'm an instance.";
5. }
6. function say() {
7. return "I'm inner.";
8. }
9. }
10. Person.say = function() {
11. return "I'm static.";
12. }
13. var p = new Person();
14. p.say(); // I'm an instance. 实例
15. Person.say(); // I'm static. 静态
16. // 这里无法直接访问内部函数 there is no way to directly access the inner function from here

你可以使用三种不同的名称路径语法来表示这三种不同的方法。

例如，使用一个文档化标签来描述你的代码：

1. Person#say // 名为"say"的实例方法
2. Person.say // 名为"say"的静态方法
3. Person~say // 名为"say"的内部函数

你可能会惊讶，既然内部方法不能在它被定义的函数外部直接访问，那么为什么还有语法来引用这个内部方法，虽然这是正确的，这个“〜”语法很少使用，内部方法有可能被引用到另一种方法的容器中被返回出来，因此在你的代码其他地方的一些对象有可能借用这个内部方法。

需要注意的是，如果一个构造函数有一个实例成员，这个实例成员也是一个构造器，那么你可以简单地将名称路径连接在一起，形成一个较长名路径名：

例如，使用一个文档化标签来描述你的代码：

1. /\*\* @constructor \*/
2. Person = function() {
3. /\*\* @constructor \*/
4. this.Idea = function() {
5. this.consider = function(){
6. return "hmmm";
7. }
8. }
9. }
10. var p = new Person();
11. var i = new p.Idea();
12. i.consider();

在这种情况下，引用名为"consider"的方法，你可以使用下面的名路径名：Person#Idea#consider

链接可与连接符号（#,.,~）任意组合使用。

**特殊情况：模块，外部组件和事件。**

1. /\*\* A module. Its name is module:foo/bar.
2. \* @module foo/bar
3. \*/
4. /\*\* The built in string object. Its name is external:String.
5. \* @external String
6. \*/
7. /\*\* An event. Its name is module:foo/bar.event:MyEvent.
8. \* @event module:foo/bar.event:MyEvent
9. \*/

使用名称路径也有一些特殊的情况：@module名称由"module:"前缀，@external 名称由"external:"前缀，@event名称由"event:"前缀。

**在名称中，对象的名称路径中带有特殊字符。**

1. /\*\* @namespace \*/
2. var chat = {
3. /\*\*
4. \* Refer to this by {@link chat."#channel"}.
5. \* @namespace
6. \*/
7. "#channel": {
8. /\*\*
9. \* Refer to this by {@link chat."#channel".open}.
10. \* @type {boolean}
11. \* @defaultvalue
12. \*/
13. open: true,
14. /\*\*
15. \* Internal quotes have to be escaped by backslash. This is
16. \* {@link chat."#channel"."say-\"hello\""}.
17. \*/
18. 'say-"hello"': function (msg) {}
19. }
20. };
21. /\*\*
22. \* Now we define an event in our {@link chat."#channel"} namespace.
23. \* @event chat."#channel"."op:announce-motd"
24. \*/

上面这个例子中，一个命名空间中的成员名称有带有特殊字符（哈希字符#号，破折号，双引号）。这种情况下，你需要这样引用这些名字：chat."#channel", chat."#channel"."op:announce-motd"，等等。在名称内部的双引号应该用反斜杠转义：`chat."#channel"."say-\"hello\""。

**JSDoc中的命令行参数**

使用JSDoc最基本的，像这样使用：

1. `/path/to/jsdoc yourSourceCodeFile.js anotherSourceCodeFile.js ...`

其中...是生成文档文件的路径。

此外，可以提供一个 [Markdown file][md-file]（以“.md”结尾）或者一个名为“README”文件的路径，它将被添加到文档的头部。请参见[these instructions][including-readme]。

JSDoc支持大量的命令行选项，其中许多选项有长和短两种形式。或者，JSDoc命令行选项可以在配置文件中指定。命令行选项：

| **选项** | **描述** |
| --- | --- |
| -a <value>, --access <value> | 只显示特定access方法属性的标识符： private, protected, public, or undefined, 或者 all（表示所有的访问级别）。默认情况下， 显示除private标识符以外的所有标识符。 |
| -c <value>, --configure <value> | JSDoc配置文件的路径。默认为安装JSDoc目录下的conf.json或conf.json.EXAMPLE。 |
| -d <value>, --destination <value> | 输出生成文档的文件夹路径。JSDoc内置的Haruki模板，使用console 将数据转储到控制台。默认为./out。 |
| --debug | 打印日志信息，可以帮助调试JSDoc本身的问题。 |
| -e <value>, --encoding <value> | 当JSDoc阅读源代码时假定使用这个编码，默认为 utf8。 |
| -h, --help | 显示JSDoc的命令行选项的信息，然后退出。 |
| --match <value> | 只有运行测试，其名称中包含value。 |
| --nocolor | 当运行测试时，在控制台输出信息不要使用的颜色。在Windows中，这个选项是默认启用的。 |
| -p, --private | 将标记有[@private 标签][tags-private.md]的标识符也生成到文档中。默认情况下，不包括私有标识符。 |
| -P, --package | 包含项目名称，版本，和其他细节的package.json文件。默认为在源路径中找到的第一个package.json文件。 |
| --pedantic | 将错误视为致命错误，将警告视为错误。默认为false。 |
| -q <value>, --query <value> | 一个查询字符串用来解析和存储到全局变量env.opts.query中。示例：foo=bar&baz=true。 |
| -r, --recurse | 扫描源文件和导览时递归到子目录。 |
| -R, --readme | 用来包含到生成文档的README.md文件。默认为在源路径中找到的第一README.md 文件。 |
| -t <value>, --template <value> | 用于生成输出文档的模板的路径。默认为templates/default，JSDoc内置的默认模板。 |
| -T, --test | 运行JSDoc的测试套件，并把结果打印到控制台。 |
| -u <value>, --tutorials <value> | 导览路径，JSDoc要搜索的目录。如果省略，将不生成导览页。 |
| -v, --version | 显示JSDoc的版本号，然后退出。 |
| --verbose | 日志的详细信息到控制台JSDoc运行。默认为false。 |
| -X, --explain | 以JSON格式转储所有的doclet到控制台，然后退出。 |

**示例**

使用配置文件/path/to/my/conf.json，为./src目录的中文件生成文档，并保存输出到./docs目录中：

1. /path/to/jsdoc src -r -c /path/to/my/conf.json -d docs

运行所有JSDoc的测试，其名称包含 tag，并记录每个测试信息：

1. /path/to/jsdoc -T --match tag --verbose

**用conf.json配置JSDoc**

**Configuration File(配置文件)**

要自定义JSDoc的行为，可以使用JSON格式的配置文件格式化JSDoc，使用-c选项，例如： jsdoc -c /path/to/conf.json。

这个文件（通常命名为conf.json）提供了JSON格式化选项。看看JSDoc目录中的一个基本的例子conf.json.EXAMPLE。如果不指定配置文件，这是JSDoc将会使用什么：

1. {
2. "tags": {
3. "allowUnknownTags": true,
4. "dictionaries": ["jsdoc","closure"]
5. },
6. "source": {
7. "includePattern": ".+\\.js(doc)?$",
8. "excludePattern": "(^|\\/|\\\\)\_"
9. },
10. "plugins": [],
11. "templates": {
12. "cleverLinks": false,
13. "monospaceLinks": false
14. }
15. }

这意味着：

* JSDoc允许您使用无法识别的标签（tags.allowUnknownTags）;
* 这两个标准JSDoc标签和closure标签被启用（tags.dictionaries）;
* 只有以.js和.jsdoc结尾的文件将会被处理（source.includePattern）;
* 任何文件以下划线开始或开始下划线的目录都将被忽略（source.excludePattern）;
* 无插件加载（plugins）;
* @link标签呈现在纯文本（templates.cleverLinks，templates.monospaceLinks）。

这些选项和其他选项将在这个页面中进一步解释。

各种插件或模板等进一步设置可被添加到该文件（例如，markdown插件可以通过markdown key进行配置）。

**Specifying input files(指定输入文件)**

source选项组，结合给JSDoc命令行的路径，确定哪些文件要用JSDoc生成文档。 （在添加这个例子到你自己的.json文件之前，请先删除注释。）

1. "source": {
2. "include": [ /\* array of paths to files to generate documentation for \*/ ],
3. "exclude": [ /\* array of paths to exclude \*/ ],
4. "includePattern": ".+\\.js(doc)?$",
5. "excludePattern": "(^|\\/|\\\\)\_"
6. }

* source.include：可选的路径数组，JSDoc应该为它们生成文档。JSDoc将会结合命令行上的路径和这些文件名，以形成文件组，并且扫描。如果路径是一个目录，可以使用-r选项来递归。
* source.exclude：可选的路径数组，JSDoc应该忽略的路径。在JSDoc3.3.0或更高版本，该数组可包括source.include路径中的子目录。
* source.includePattern：一个可选的字符串，解释为一个正则表达式。如果存在，所有文件必须匹配这个正则表达式，以通过JSDoc进行扫描。默认情况下此选项设置为.+.js(doc)?$，这意味着只有以.js或者.jsdoc结尾的文件将被扫描。
* source.excludePattern：一个可选的字符串，解释为一个正则表达式。如果存在的话，任何匹配这个正则表达式的文件将被忽略。默认设置是以下划线开头的文件（或以下划线开头的目录下的所有文件）将被忽略。

这些选项中使用的顺序是：

1. 以命令行上给定的路径开始，并且在source.include中的所有文件（记得，使用-r命令行选项将在子目录中搜索）。
2. 对于在步骤1中找到的每个文件，如果正则表达式source.includePattern存在，该文件必须匹配，否则将被忽略。
3. 对于在步骤2中遗留下的每个文件，如果正则表达式source.excludePattern存在，任何匹配这个正则表达式的文件将被忽略。
4. 对于在步骤3中遗留下的每个文件，如果路径在source.exclude中，那么它将被忽略。

经过这四个步骤的所有剩余文件由JSDoc进行解析。

举个例子，假设我有以下文件结构：

1. myProject/
2. |- a.js
3. |- b.js
4. |- c.js
5. |- \_private
6. | |- a.js
7. |- lib/
8. |- a.js
9. |- ignore.js
10. |- d.txt

我把我的conf.json文件中的source部分设置成这样：

1. "source": {
2. "include": [ "myProject/a.js", "myProject/lib", "myProject/\_private" ],
3. "exclude": [ "myProject/lib/ignore.js" ],
4. "includePattern": ".+\\.js(doc)?$",
5. "excludePattern": "(^|\\/|\\\\)\_"
6. }

如果我从包含在myProject文件夹中的文件运行JSDoc，像这样：

jsdoc myProject/c.js -c /path/to/my/conf.json -r

然后JSDoc会为这些文件生产文档：

* myProject/a.js
* myProject/c.js
* myProject/lib/a.js

原因如下：

1. 根据source.include和命令行中给定的路径，我们开始扫描文件
   * myProject/c.js (来自命令行)
   * myProject/a.js (来自source.include)
   * myProject/lib/a.js, myProject/lib/ignore.js, myProject/lib/d.txt (来自source.include 并且使用 -r 选项)
   * myProject/\\_private/a.js (来自source.include)
2. 应用source.includePattern，剩下除了myProject/lib/d.txt（因为它没有以.js或.jsdoc结束）以外的所有上述文件。
3. 应用source.excludePattern，排除了myProject/\\_private/a.js。
4. 应用source.exclude，排除了myProject/lib/ignore.js。

**Incorporating command-line options into the configuration file(合并命令行选项到配置文件)**

它有可能把许多JSDoc的命令行选项放到配置文件，而不用在命令行中指定它们。要做到这一点，只要在conf.json的opts部分中使用的相关选项的longnames,值是该选项的值。

**在配置文件中设置的命令行选项**

1. "opts": {
2. "template": "templates/default", // same as -t templates/default
3. "encoding": "utf8", // same as -e utf8
4. "destination": "./out/", // same as -d ./out/
5. "recurse": true, // same as -r
6. "tutorials": "path/to/tutorials", // same as -u path/to/tutorials
7. }

这样可以通过source.include和opts，把所有的JSDoc的参数放在配置文件中，以便命令行简化为：

jsdoc -c /path/to/conf.json

在命令行中所提供的选项 优先级高于在conf.json中提供的选项。

**Plugins（插件）**

要启用插件，只要添加它们的路径（相对于JSDoc文件夹）到plugins数组中就可以了。

例如，以下将包括 markdown 插件，它转换 markdown格式的文本为HTML，和“summarize”插件，该自动生成的每个的doclet的摘要：

1. "plugins": [
2. "plugins/markdown",
3. "plugins/summarize"
4. ]

更多信息请参见插件参考，并在jsdoc/plugins 中查找插件 来构建JSDoc。

markdown插件可以通过在conf.json中设置markdown对象进行配置;请参阅配置markdown插件获得更多信息。

Output style configuration(配置输出风格)

templates选项会影响外观和生成的文档内容。自定义模板可能不会实现所有的选项。请参阅配置JSDoc的默认模板中默认模板支持的附加选项。

1. "templates": {
2. "cleverLinks": false,
3. "monospaceLinks": false
4. }

如果templates.monospaceLinks为true，从@link标签生成的所有链接文本将会以等宽字体渲染。

如果templates.cleverLinks为true，如果“asdf”是一个URL，{@link asdf} 会以正常字体呈现，否则等宽。例如，{@link http://github.com}将呈现以纯文本，但{@link MyNamespace.myFunction}将会是等宽。

如果templates.cleverLinks为true，它是引用,并且templates.monospaceLinks是被忽略的。

另外，还有一些{@linkcode…}和{@linkplain…},如果希望强制的链接被等宽或普通字体分别渲染（见@link，@linkcode和@linkplain了解更多信息）。

**Tags and tag dictionaries(标签和标签字典)**

tags选项控制哪些JSDoc标签允许被使用和解析。

1. "tags": {
2. "allowUnknownTags": true,
3. "dictionaries": ["jsdoc","closure"]
4. }

tags.allowUnknownTags属性影响JSDoc如何处理无法识别的标签。如果将此选项设置为false，JSDoc发现它不能识别（例如,@foo），JSDoc将记录一个警告。默认情况下，此选项设置为true。

该tags.dictionaries属性控制JSDoc识别哪些标签，以及JSDoc如何解析它识别标签。在JSDoc3.3.0或更高版本中，有两个内置的标签词典：

* jsdoc: 核心JSDoc标签
* closure: [Closure Compiler 标签](https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler)

默认情况下，两个词典都是启用的。此外，在默认情况下，jsdoc字典首先被解析;作为一个结果，如果jsdoc词典处理一个标签不同于closure词典，jsdoc版本的标签优先被采用(即，以jsdoc版本的标签为准)。

如果您在Closure Compiler 项目中使用JSDoc，并且你想要避免使用 Closure Compiler无法识别的标签，更改tags.dictionaries设置为[“closure”]。如果你想允许核心JSDoc标签, 但又想要确保Closure Compiler特定的标记使用Closure Compiler对其进行解释，您也可以更改此设置为[“closure”,”jsdoc”]。

**配置JSDoc的默认模板**

JSDoc的默认模板中提供了几个选项，您可以使用自定义外观和内容来生成文档。

要使用这些选项，您必须为JSDoc创建一个配置文件，并在配置文件中设置相应的选项。

**Generating pretty-printed source files（生成适合打印的文档）**

默认情况下，JSDoc的默认模板为你的源文件生成适合打印的文档。在文档中，它还链接到那些适合的打印文件。

要禁用适合打印的文件，设置选项templates.default.outputSourceFiles为false。使用该选项也将删除文档中链接到源文件的连接。此选项在JSDoc3.3.0及更高版本上是可用的。

**Copying static files to the output directory(复制静态文件到输出目录)**

JSDoc的默认模板会自动复制一些静态文件，如CSS样式表，到输出目录。在JSDoc3.3.0或更高版本，你可以告诉默认模板复制附加静态文件到输出目录。例如，您可能希望复制一个图像的目录到输出目录，所以你可以在你的文档中显示这些图像。

要将附加静态文件复制到输出目录，使用下列选项：

* templates.default.staticFiles.include：一个路径的数组，其内容应复制到输出目录。子目录也将被复制。
* templates.default.staticFiles.exclude：路径的数组，指明这些 *不* 应该被复制到输出目录。
* templates.default.staticFiles.includePattern：正则表达式，指明要复制的文件。如果这个属性没有被定义，所有的文件将被复制。
* templates.default.staticFiles.excludePattern：正则表达式，说明哪些文件跳过(不复制)。如果这个属性没有被定义，什么都不会被跳过。

**复制图片目录到输出目录**

例如，要复制./myproject/static目录中的所有静态文件到输出目录中：

1. {
2. "templates": {
3. "default": {
4. "staticFiles": {
5. "include": [
6. "./myproject/static"
7. ]
8. }
9. }
10. }
11. }

如果您的静态文件目录中包含./myproject/static/img/screen.png文件，您可以通过HTML标签 <img src="img/screen.png">在您的文档中显示该图片。

**Showing the current date in the page footer（在页脚显示当前日期）**

默认情况下，JSDoc的默认模板总是在生成文档的页脚显示当前日期。在JSDoc3.3.0或更高版本，可以通过设置选项templates.default.includeDate为false来忽略当前日期。

**Showing longnames in the navigation column（在导航栏中显示长文件名）**

默认情况下，JSDoc的默认模板在导航列中显示每个标识符缩写的名字。例如，标识符my.namespace.MyClass将简单地称为显示MyClass。相反,要显示完整的长名称，设置选项templates.default.useLongnameInNav为true。此选项在JSDoc3.4.0及更高版本中可用。

**Overriding the default template's layout file（重写默认模板的布局文件）**

默认的模板使用名为 layout.tmpl 的文件 指定每个生成文档的页面中的页眉和页脚。特别是，每个生产的文档页面会加载该文件定义了CSS和JavaScript文件。在JSDoc3.3.0或更高版本，可以指定使用自己的layout.tmpl文件，它允许你加载自己的自定义CSS和JavaScript文件，去除或替代，标准的文件。

要使用此功能，设置选项templates.default.layoutFile的路径到你的自定义布局文件。路径是相对于config.json文件，当前的工作目录，和JSDoc目录的相对路径，按照这个顺序。

**块标签和内联标签**

**概述**

JSDoc支持两种不同类型的标签：

* **块标签**, 这是在一个JSDoc注释的最高级别。
* **内联标签**, 块标签文本中的标签或说明。

块标签通常会提供有关您的代码的详细信息，如一个函数接受的参数。内联标签通常链接到文件的其他部分，类似于HTML中的锚标记（<a>）。

块标签总是以 at 符号（@）开头。除了JSDoc注释中最后一个块标记，每个块标签后面必须跟一个换行符。

内联标签也以 at 符号（@）开。然而，内联标签及其文本必须用花括号（{ and }）括起来。 { 表示行内联标签的开始，而}表示内联标签的结束。如果你的标签文本中包含右花括号（}），则必须用反斜线（ \ ）进行转义。在内联标签后,你并不需要使用一个换行符。

大多数JSDoc标签是块标签。一般来说，当这个网站上说"JSDoc 标签",我们真正的意思是"块标签"。

**示例**

在下面的例子中， @param 是一个块标签，而{@link}是一个内联标签。

例如，JSDoc注释中的块标签和内联标签：

1. /\*\*
2. \* Set the shoe's color. Use {@link Shoe#setSize} to set the shoe size.
3. \*
4. \* @param {string} color - The shoe's color.
5. \*/
6. Shoe.prototype.setColor = function(color) {
7. // ...
8. };

您可以在描述中使用内联标签，如上图所示，或在块标记中使用内联标签，如下图所示。

例如，块标签内使用内嵌标签：

1. /\*\*
2. \* Set the shoe's color.
3. \*
4. \* @param {SHOE\_COLORS} color - The shoe color. Must be an enumerated
5. \* value of {@link SHOE\_COLORS}.
6. \*/
7. Shoe.prototype.setColor = function(color) {
8. // ...
9. };

当您在JSDoc注释中使用多个块标记，它们必须使用换行符将他们分开。

例如，用换行符分隔的多个块标签：

1. /\*\*
2. \* Set the color and type of the shoelaces.
3. \*
4. \* @param {LACE\_COLORS} color - The shoelace color.
5. \* @param {LACE\_TYPES} type - The type of shoelace.
6. \*/
7. Shoe.prototype.setLaceType = function(color, type) {
8. // ...
9. };

# 关于JSDoc插件

## Creating and Enabling a Plugin（创建并启用插件）

创建并启用新JSDoc插件,需要两个步骤：

1. 创建一个包含你的插件代码的JavaScript模块.
2. 将该模块添加到JSDoc配置文件的plugins数组中。你可以指定一个绝对或相对路径。如果使用相对路径，JSDoc按照相对于配置文件所在的目录，当前的工作目录和JSDoc安装目录的顺序搜索插件。

例如，如果你的插件是在当前工作目录下，plugins/shout.js文件中定义的，你应该在JSDoc配置文件中的plugins数组中添加字符串plugins/shout。

例如，在JSDoc配置文件中添加一个插件：

1. {
2. "plugins": ["plugins/shout"]
3. }

JSDoc按配置文件plugins数组中列出的顺序执行的插件。

## Authoring JSDoc 3 Plugins（创建JSDoc3插件）

JSDoc 3的插件系统广泛的控制着解析过程。一个插件可以通过执行以下任何一项操作，影响解析结果：

* 定义事件处理程序
* 定义标签
* 定义一个抽象语法树节点的访问者

### Event Handlers（事件处理程序）

最高级别，一个插件可以注册具体命名事件的处理程序，让JSDoc触发。 JSDoc将一个事件对象传递给处理程序。你的插件模块要导出一个包含处理程序的handlers对象，像这样。

例如，事件处理程序插件 'newDoclet' 事件：

1. exports.handlers = {
2. newDoclet: function(e) {
3. // Do something when we see a new doclet
4. }
5. };

JSDoc触发事件的顺序和底层代码是一样。

事件处理程序插件可以通过设置事件对象的stopPropagation属性(e.stopPropagation = true)停止运行后面的插件。一个插件可以通过设置preventDefault属性（e.preventDefault = true）阻止触发事件。

#### 事件: parseBegin

JSDoc开始加载和解析源文件之前，parseBegin事件被触发。你的插件可以通过修改事件的内容，来控制哪些文件将被JSDoc解析。

**注意**：此事件在JSDoc3.2及更高版本有效。

该事件对象包含下列属性：

* sourcefiles：源文件的路径数组，这些源文件将被解析。

#### 事件: fileBegin

当解析器即将解析一个文件fileBegin事件触发。如果需要，你的插件可以使用此事件触发每个文件的初始化。

该事件对象包含下列属性：

* filename：文件的名称。

#### 事件: beforeParse

beforeParse事件在解析开始之前被触发。插件可以使用此方法来修改将被解析的源代码。例如，你的插件可以添加一个JSDoc注释，也可以删除不是有效JavaScript的预处理标记。

该事件对象包含下列属性：

* filename：文件的名称。
* source：文件的内容。

下面是为一个函数增加了一个虚拟的doclet到源文件的例子，这样它会被解析并添加到文档。这样做，文档的方法就可以提供给用户，但在被文档化的源代码中不可能出现，如由外部超类所提供的方法：

示例：

1. exports.handlers = {
2. beforeParse: function(e) {
3. var extraDoc = [
4. '/\*\*',
5. ' \* Function provided by a superclass.',
6. ' \* @name superFunc',
7. ' \* @memberof ui.mywidget',
8. ' \* @function',
9. ' \*/'
10. ];
11. e.source += extraDoc.join('\n');
12. }
13. };

#### 事件: jsdocCommentFound

每当JSDoc注释被发现,jsdocCommentFound事件就会被触发。注释可以或不与任何代码相关联。您可以在注释被处理之前使用此事件修改注释的内容。

该事件对象包含下列属性：

* filename: 该文件的名称。
* comment：JSDoc注释的文本。
* lineno: 注释被发现的行号。

#### 事件: symbolFound

当解析器在代码中遇到一个可能需要被文档化的标识符时，symbolFound 事件就会被触发。例如，解析器会为源文件中每个变量，函数和对象字面量触发一个symbolFound事件。

该事件对象包含下列属性：

* filename：该文件的名称。
* comment：与标识符相关联的任何注释文本。
* id:标识符的唯一ID。
* lineno:标识符被发现的行号。
* range:包含标识符相关联的源文件中第一个和最后一个字符的数字索引的一个数组。
* astnode：抽象语法树中标识符的节点。
* code：有关该代码的详细信息的对象。这个对象通常包含name,type, 和 node 属性。对象也可能具有value, paramnames, 或 funcscope属性，这取决于标识符。

#### 事件: newDoclet

newDoclet事件是最高级别的事件。新的doclet已被创建时，它就会被触发。这意味着一个JSDoc注释或标识符已被处理，并且实际传递给模板的doclet已被创建。

该事件对象包含下列属性：

* doclet: 已被创建的新 doclet 。

所述的doclet的属性取决于doclet表示的注释或标识符。你可能会看到一些共同的属性包括：

* comment：JSDoc注释文本，或者，如果标识符没被描述，那么该值是一个空字符串。
* meta：对象，描述doclet如何关联源文件（例如，在源文件中的位置）。
* description：被记录的标识符的说明。
* kind:被记录的标识符的种类（例如，class 或者 function）。
* name:标识符的短名称（例如，myMethod）。
* longname:全名，其中包含成员信息（例如，MyClass#myMethod）。
* memberof：该标识符所读的模块，命名空间或类（例如，MyClass），或者，如果该标识符没有父级，那么该值是一个空字符串。
* scope:标识符在其父级内的作用域范围（例如，global, static, instance,或 inner）。
* undocumented：如果标识符没有JSDoc注释，设置为true。
* defaultvalue：标识符的默认值。
* type：包含关于标识符类型详细信息的对象。
* params：包含参数列表的对象。
* tags：对象，包含JSDoc不识别的标记列表。只有当JSDoc的配置文件中allowUnknownTags设置为true时可用。

下面是一个newDoclet 处理程序的一个例子说明：

1. exports.handlers = {
2. newDoclet: function(e) {
3. // e.doclet will refer to the newly created doclet
4. // you can read and modify properties of that doclet if you wish
5. if (typeof e.doclet.description === 'string') {
6. e.doclet.description = e.doclet.description.toUpperCase();
7. }
8. }
9. };

#### 事件: fileComplete

当解析器解析完一个文件时，fileComplete 事件就会被触发。你的插件可以使用这个事件来触发每个文件的清理。

该事件对象包含下列属性：

* filename: 文件名称.
* source:该文件的内容.

#### 事件: parseComplete

JSDoc解析所有指定的源文件之后，parseComplete事件就会被触发。

**注意**：此事件在JSDoc3.2及更高版本会被触发。

该事件对象包含下列属性：

* sourcefiles:被解析的源代码文件的路径数组。
* doclets：doclet对象的数组。见[newDoclet事件](https://www.shouce.ren/api/view/a/13238#%E4%BA%8B%E4%BB%B6-newdoclet)，有关每个的doclet可以包含属性的详细信息。**注意**：这个属性在JSDoc3.2.1及更高版本中可用。

#### 事件: processingComplete

JSDoc更新反映继承和借来的标识符的解析结果后，processingComplete事件被触发。

注意：此事件在JSDoc3.2.1及更高版本中会被触发。

该事件对象包含下列属性：

* doclets：doclet对象的数组。见[newDoclet事件](https://www.shouce.ren/api/view/a/13238#%E4%BA%8B%E4%BB%B6-newdoclet)，有关每个的doclet可以包含属性的详细信息。

### Tag Definitions （标签定义）

添加标签到标签字典是影响文档生成的一个中级方式。在一个newDoclet事件被触发前，JSDoc注释块被解析以确定可能存在的说明和任何JSDoc标签。当一个标签被发现，如果它已在标签字典被定义，它就会被赋予一个修改doclet的机会。

插件可以通过导出一个defineTags函数来定义标签。该函数将传递一个可用于定义标签的字典，像这样：

示例：

1. exports.defineTags = function(dictionary) {
2. // define tags here
3. };

#### The Dictionary（字典）

字典提供了以下方法：

* defineTag(title, opts): 用于定义标签。第一个参数是标签的名称（例如，param 或 overview）。第二个参数是一个包含标签选项的对象。可以包含以下任一选项;每个选项的默认值都是false：
  + canHaveType (boolean): 如果标签文本可以包含一个类型表达式，那么设置为true（如 @param {string} name - Description 中的 {string}）。
  + canHaveName (boolean):如果标签文本可以包含一个名称，那么设置为true（如 @param {string} name - Description中的 name )。
  + isNamespace (boolean): 如果该标签是应用doclet的长名称，作为命名空间，那么设置为true。例如，@module标签应设置该选项设置为true，并在标签上使用@module myModuleName的结果为长名称 module:myModuleName。
  + mustHaveValue (boolean): 如果该标签必须有一个值，那么设置为true（如 @name TheName中的 TheName )。
  + mustNotHaveDescription (boolean): 如果该标签可能有一个值，但是不是必须有描述，那么设置为true（如 @tag {typeExpr} TheDescription中的 TheDescription )。
  + mustNotHaveValue (boolean):如果该标签必须没有值，那么设置为true。
  + onTagged (function):当该标签被发现时，执行的回调函数。该函数传递两个参数：该doclet和该标签对象。
* lookUp(tagName): 按名称检索标签对象。返回该标签对象，包括它的选项，如果标签没有定义，那么返回false。
* isNamespace(tagName):如果该标签是应用doclet的长名称，作为命名空间，那么返回true。
* normalise(tagName): 返回标签的规范名称。例如，@constant是@const标签的同义词;如果你调用normalise('const')，那么返回结果是constant字符串。
* normalize(tagName): normalise的同义词。在JSDoc3.3.0及更高版本中可用。

标签的onTagged 回调可以修改doclet或标签的内容。

定义一个 onTagged 回调

1. dictionary.defineTag('instance', {
2. onTagged: function(doclet, tag) {
3. doclet.scope = "instance";
4. }
5. });

defineTag方法返回一个Tag对象，这个对象有一个synonym方法，这个方法可用于定义该标签的一个同义词。

定义标签同义词

1. dictionary.defineTag('exception', { /\* options for exception tag \*/ })
2. .synonym('throws');

### Node Visitors（节点访问者）

在最底层，插件作者可以通过定义一个访问的每个节点的节点访问者来处理在抽象语法树（AST）中的每个节点。通过使用node-visitor插件，您可以修改注释并触发任意一段代码的解析事件。

插件可以通过导出一个包含visitNode 函数的 astNodeVisitor 对象来定义节点访问者，像这样：

示例：

1. exports.astNodeVisitor = {
2. visitNode: function(node, e, parser, currentSourceName) {
3. // do all sorts of crazy things here
4. }
5. };

该函数在每个节点上调用，具有以下参数：

* node：AST的节点。 AST节点是JavaScript对象,使用由Mozilla的解析器API定义的格式。您可以使用[Esprima的解析器演示](http://esprima.org/demo/parse.html)，查看为您的源代码创建的AST。
* e：事件。如果该节点是一个解析器处理，事件对象将已经被填充，在symbolFound事件上用相同的东西描述。否则，这将是空对象在其上设置各种属性。
* parser：本JSDoc解析器实例。
* currentSourceName：该文件的名称被解析。

#### 使事情发生

实现节点访问的首要原因是为了能够记录事情，那些不寻常的记录（创建类像函数调用），或者自动生成文档为未记录的代码。例如，一个插件可能看起来调用到\_trigger方法，因为它知道这意味着一个事件被触发，然后生成文档因为这个事件。

为了使事情发生了，visitNode函数应该修改事件参数的属性。一般来说，目标是构建一个注释，然后得到一个事件触发。在分析器可以让所有的节点的访问都来看看节点之后，它会查看是否该事件对象有一个comment释属性和event属性。如果两个都有，在事件属性命名的事件被触发。该事件通常symbolFound or jsdocCommentFound，但理论上，一个插件可以定义自己的事件和处理它们。

与事件处理程序的插件，一个节点访问插件可以停止后面的插件，运行通过在事件对象上设置stopPropagation属性（e.stopPropagation = true）。一个插件可以通过设置的preventDefault属性停止事件触发（e.preventDefault = true）。

## Reporting Errors(报告错误)

如果你的插件需要报告错误，使用在jsdoc/util/logger模块中的下列方法之一：

* logger.warn：发出警告给用户，可能出现的问题。
* logger.error：报告错误，从该插件可以恢复。
* logger.fatal：报告错误，应引起JSDoc停止运行。

使用这些方法创建更好的用户体验,不是简单地抛出一个错误。

**注意**：请不要使用jsdoc/util/error 模块报告错误。该模块使用，将在JSDoc的未来版本中删除。

报告一个非致命错误：

1. var logger = require('jsdoc/util/logger');
2. exports.handlers = {
3. newDoclet: function(e) {
4. // Your code here.
5. if (somethingBadHappened) {
6. logger.error('Oh, no, something bad happened!');
7. }
8. }
9. }

# 使用Markdown插件

## Overview（概述）

SDoc包括markdown插件，自动把Markdown-formatted文本转换成HTML。你可以在任何JSDoc模板中使用这个插件。在JSDoc3.2.2及以后版本中，Markdown插件使用了[marked Markdown 解析器](https://github.com/chjj/marked)。

**注意**：当您启用降价插件，一定要在您JSDoc注释的每行前面加上前导星号。如果省略前导星号，JSDoc解析器可能会删除用于markdown格式化的星号。

默认情况下，JSDoc会在以下JSDoc标签中查找markdown格式的文本：

* @author
* @classdesc
* @description (包括在JSDoc注释开始地方的未标记的)
* @param
* @property
* @returns
* @see
* @throws

## Enabling the Markdown plugin(启用markdown插件)

要启用markdown插件，只要将字符串plugins/markdown添加到JSDoc配置文件的plugins数组中即可。

示例，配置文件启用markdown插件：

1. {
2. "plugins": ["plugins/markdown"]
3. }

## Converting Markdown in additional JSDoc tags（在额外的JSDoc标签中转换Markdown）

默认情况下，markdown插件只处理[特定JSDoc标签](https://www.shouce.ren/api/view/a/13239#default-tags)的markdown文本。您可以通过添加一个 markdown.tags属性到你的JSDoc配置文件中，来处理的其他标签中的markdown文本。markdown.tags属性包含一个额外的doclet属性的数组，这个doclet属性可以包含markdown文本。 （在大多数情况下，doclet属性的名称相同的标签名。然而，一些标签存储方式不一样;例如，@param 标签存储在的doclet的params属性。如果你不知道如何标签的文本存储在一个doclet中，运行JSDoc使用-X/--explain，打印每个的doclet到控制台）

例如，如果foo和bar标签接受在一个的doclet的foo和bar属性中存储值，你可以通过添加下面的设置到您JSDoc配置文件，来使用markdown处理这些标签。

例如，转换’foo’和’bar’标签中的Markdown

1. {
2. "plugins": ["plugins/markdown"],
3. "markdown": {
4. "tags": ["foo", "bar"]
5. }
6. }

## Excluding the default tags from Markdown processing（剔除markdown默认处理的标签）

为了防止Markdown插件处理任何[默认JSDoc标签](https://www.shouce.ren/api/view/a/13239#default-tags)，添加一个markdown.excludeTags属性到您的JSDoc配置文件。该markdown.excludeTags属性包含不应该被markdown文本处理的默认标签数组。

例如，从markdown处理排除author标签：

1. {
2. "plugins": ["plugins/markdown"],
3. "markdown": {
4. "excludeTags": ["author"]
5. }
6. }

## Hard-wrapping text at line breaks （用换行符换行文本）

默认情况下，Markdown插件不处理换行符换行的文本。这是因为，这是正常的JSDoc注释可以多行。如果您更喜欢处理换行符换行的文本，设置JSDoc配置文件中的markdown.hardwrap属性为true。此属性是在JSDoc3.4.0及更高版本中可用。

## Adding ID attributes to headings（添加ID属性到标题标签）

默认情况下，Markdown插件不会给每个HTML标题标签添加id 属性。想要标题标签文本自动添加ID属性，设置JSDoc配置文件markdown.idInHeadings属性为true。此属性是在JSDoc3.4.0及更高版本中可用。

**Tutorials 教程**

JSDoc允许你的API文档的页面旁边包含教程。您可以使用此功能来为您的API提供详细的使用说明，如“入门”指南或实现一个功能的一步一步的过程。

**Adding tutorials（添加教程）**

添加教程到您的API文档，可以通过--tutorials 或 -u 选项运行JSDoc，并提供JSDoc要搜索的教程目录。例如：

1. jsdoc -u path/to/tutorials path/to/js/files

JSDoc在教程目录中搜索具有以下扩展名的文件：

* .htm
* .html
* .markdown (转换 Markdown 为 HTML)
* .md (转换 Markdown 为 HTML)
* .xhtml
* .xml (作为HTML处理)

JSDoc还搜索JSON文件，这个JSON文件包含有关标题，排序，和教程的层次结构等信息，这些将在下面的部分中讨论。

JSDoc给每个教程分配一个标识符。该标识符是不带扩展名的文件名。例如， /path/to/tutorials/overview.md分配的标识符是overview。

在教程文件中，您可以使用{@link}和{@tutorial}内联标签来链接到文档的其他部分。JSDoc将自动处理这些链接。

**Configuring titles, order, and hierarchy （配置标题，顺序和层次结构）**

默认情况下，JSDoc使用的文件名作为教程标题，并且所有的教程都在同一层次。您可以使用JSON文件为每个教程提供标题并指示文档中的教程应如何排序和分组。

JSON文件必须使用扩展.json。在JSON文件，您可以使用教程标识符为每个教程提供两个属性：

* title: 文档中显示的标题。
* children:子教程信息。

在JSDoc 3.2.0或更高版本中，你可以使用以下格式的JSON文件：

1. 对象树，子教程定义在父级教程的children属性中。例如，tutorial1有两个子教程childA 和 childB，tutorial2和tutorial1在同一层级上，并且tutorial2没有子教程：
   1. {
   2. "tutorial1": {
   3. "title": "Tutorial One",
   4. "children": {
   5. "childA": {
   6. "title": "Child A"
   7. },
   8. "childB": {
   9. "title": "Child B"
   10. }
   11. }
   12. },
   13. "tutorial2": {
   14. "title": "Tutorial Two"
   15. }
   16. }
2. 一个顶级对象，其属性都是教程对象，子教程在教程对象的children属性中列出名称。例如，tutorial1有两个子教程childA 和 childB，tutorial2和tutorial1在同一层级上，并且tutorial2没有子教程：
   1. {
   2. "tutorial1": {
   3. "title": "Tutorial One",
   4. "children": ["childA", "childB"]
   5. },
   6. "tutorial2": {
   7. "title": "Tutorial Two"
   8. },
   9. "childA": {
   10. "title": "Child A"
   11. },
   12. "childB": {
   13. "title": "Child B"
   14. }
   15. }

您也可以为每个教程提供了一个单独的 .json 文件，使用教程标识符作为文件名。此方法已过时，不应该被用于新的项目。

**Linking to tutorials from API documentation（从API文档链接到教程）**

有多种方式从您的API文档的链接到教程：

**@tutorial 块标签**

如果在JSDoc注释中包含一个@tutorial 块标签，生成的文件将包含一个链接，链接到您指定的教程。

例如，使用@tutorial 块标签：

1. /\*\*
2. \* Class representing a socket connection.
3. \*
4. \* @class
5. \* @tutorial socket-tutorial
6. \*/
7. function Socket() {}

**{@tutorial} 内联标签**

您也可以在另一个标签的文本中使用{@tutorial} 内联标签，链接到一个教程。默认情况下，JSDoc将使用教程标题作为链接文字。

例如，使用{@tutorial} 内联标签

1. /\*\*
2. \* Class representing a socket connection. See {@tutorial socket-tutorial}
3. \* for an overview.
4. \*
5. \* @class
6. \*/
7. function Socket() {}

**包含Package（包）文件**

包文件包含的信息对你的项目文档是很有用的，比如该项目的名称和版本号。当JSDoc生成的文档的时候,可以自动使用项目中package.json文件中的信息。例如，默认的模板在文档中显示项目的名称和版本号。

有两种方法可以将package.json文件中的信息合并到您的文档：

1. 在你的JavaScript文件的源路径中，包含package.json文件的路径。JSDoc将使用在源路径中发现的第一个package.json 文件。
2. 使用-P/--package 包命令行选项运行JSDoc，指定package.json文件的路径。此选项在JSDoc3.3.0及更高版本中可用。

-P/--package 命令行选项优先于你的源路径。如果使用-P/--package 命令行选项，JSDoc会忽略源路径中任何的package.json文件。

package.json 文件必须使用[npm 的包格式](https://docs.npmjs.com/files/package.json)。

**例子**

在源路径中包含一个包文件:

1. jsdoc path/to/js path/to/package/package.json

使用 -P/--package 选项：

1. jsdoc --package path/to/package/package-docs.json path/to/js

**包含 README 文件**

有两种方法可以将 README 文件中的信息合并到您的文档：

1. 在你的JavaScript文件的源路径中，包含一个名为README.md的Markdown文件的路径。JSDoc将使用在源路径中发现的第一个 README.md 文件。
2. 使用-R/--readme 包命令行选项运行JSDoc，指定 README 文件的路径。此选项在JSDoc3.3.0及更高版本中可用。README文件可以使用任何名称和扩展名，但它必须是Markdown格式。

-R/--readme 命令行选项优先于你的源路径。如果使用-R/--readme命令行选项，JSDoc会忽略源路径中任何的README.md文件。

如果您正在使用JSDoc的默认模板，README文件的内容将渲染成HTML，生成在文档 index.html 文件中。

**例子**

在源路径中包含一个 README 文件:

1. jsdoc path/to/js path/to/readme/README.md

使用 -R/--readme 选项：

1. jsdoc --readme path/to/readme/README path/to/js

**ES 2015 Classes（类）**

JSDoc3 描述一个遵循 [ECMAScript 2015规范](http://www.ecma-international.org/ecma-262/6.0/#sec-class-definitions)的类是很简单的。你并不需要使用诸如如@class 和 @constructor的标签来描述 ES 2015 classes，JSDoc通过分析你的代码会自动识别类和它们的构造函数。ES 2015 classes在JSDoc3.4.0及更高版本支持。

**Documenting a simple class（文档化一个简单的类）**

下面的例子演示了如何通过一个构造函数，两个实例方法和一个静态方法文档化一个简单的类：

例如，简单的 ES 2015 类 ：

1. /\*\* Class representing a point. \*/
2. class Point {
3. /\*\*
4. \* Create a point.
5. \* @param {number} x - The x value.
6. \* @param {number} y - The y value.
7. \*/
8. constructor(x, y) {
9. // ...
10. }
11. /\*\*
12. \* Get the x value.
13. \* @return {number} The x value.
14. \*/
15. getX() {
16. // ...
17. }
18. /\*\*
19. \* Get the y value.
20. \* @return {number} The y value.
21. \*/
22. getY() {
23. // ...
24. }
25. /\*\*
26. \* Convert a string containing two comma-separated numbers into a point.
27. \* @param {string} str - The string containing two comma-separated numbers.
28. \* @return {Point} A Point object.
29. \*/
30. static fromString(str) {
31. // ...
32. }
33. }

您还可以记录类表达式中定义的类，将其分配给一个变量或常量：

例如，ES 2015 类表达式：

1. /\*\* Class representing a point. \*/
2. const Point = class {
3. // and so on
4. }

**Extending classes（扩展类）**

当您使用 extends关键字来扩展一个现有的类的时候，你还需要告诉JSDoc哪个类是你要扩展的。 为此，您可以使用 @augments (或 @extends) 标签。

例如，扩展如上所示Point 类，扩展一个 ES 2015 类：

1. /\*\*
2. \* Class representing a dot.
3. \* @extends Point
4. \*/
5. class Dot extends Point {
6. /\*\*
7. \* Create a dot.
8. \* @param {number} x - The x value.
9. \* @param {number} y - The y value.
10. \* @param {number} width - The width of the dot, in pixels.
11. \*/
12. constructor(x, y, width) {
13. // ...
14. }
15. /\*\*
16. \* Get the dot's width.
17. \* @return {number} The dot's width, in pixels.
18. \*/
19. getWidth() {
20. // ...
21. }
22. }

**ES 2015 Modules（模块）**

JSDoc3 能够记录遵循[ECMAScript 2015规范](http://www.ecma-international.org/ecma-262/6.0/#sec-modules)的模块。ES 2015 模块在JSDoc3.4.0及更高版本中支持。

**Module identifiers（模块标识符）**

当你描述一个 ES 2015 module（模块）时，您将使用@module 标签来描述模块的标识符。例如，如果用户通过调用import \* as myShirt from 'my/shirt' 加载模块，你会写一个包含@module my/shirt标签的JSDoc注释。

如果使用@module标签不带值，JSDoc会基于文件路径尝试猜测正确的模块标识符。

当您使用一个 JSDoc namepath（名称路径）从另一个JSDoc注释中引用一个模块，您必须添加前缀module:。例如，如果你想模块my/pants的文档 连接到模块my/shirt，您可以使用@see 标签来描述my/pants，如下：

1. /\*\*
2. \* Pants module.
3. \* @module my/pants
4. \* @see module:my/shirt
5. \*/

同样，模块中每个成员的namepath （名称路径）将以module: 开始：，后面跟模块名字。例如，如果你的my/pants模块输出一个Jeans类，并且Jeans 有一个名为hem的实例方法，那么这个实例方法longname（长名称）是module:my/pants.Jeans#hem。

**Exported values （导出值）**

下面的示例演示如何在ES 2015 模块中描述不同种类的导出值。在多数情况下，你可以简单地在export语句上添加一个JSDoc注释来定义导出值。如果要以其他名称导出一个值，您可以在其export块中描述导出值。

例如，文档化一个模块的导出值：

1. /\*\* @module color/mixer \*/
2. /\*\* The name of the module. \*/
3. export const name = 'mixer';
4. /\*\* The most recent blended color. \*/
5. export var lastColor = null;
6. /\*\*
7. \* Blend two colors together.
8. \* @param {string} color1 - The first color, in hexidecimal format.
9. \* @param {string} color2 - The second color, in hexidecimal format.
10. \* @return {string} The blended color.
11. \*/
12. export function blend(color1, color2) {}
13. // convert color to array of RGB values (0-255)
14. function rgbify(color) {}
15. export {
16. /\*\*
17. \* Get the red, green, and blue values of a color.
18. \* @function
19. \* @param {string} color - A color, in hexidecimal format.
20. \* @returns {Array.<number>} An array of the red, green, and blue values,
21. \* each ranging from 0 to 255.
22. \*/
23. rgbify as toRgb
24. }

**CommonJS Modules（模块）**

**Overview（概述）**

为了帮助您记录[CommonJS modules（模块）](http://wiki.commonjs.org/wiki/Modules/1.1)，JSDoc3能理解很多在CommonJS的规范中使用的约定（例如，添加属性到exports对象）。此外，JSDoc承认的[Node.js modules（模块）](http://nodejs.org/api/modules.html)的约定，它扩展了CommonJS的标准（例如，将值分配给module.exports）。根据您所遵循编码约定，您可能需要提供一些额外的标签，以帮助JSDoc理解你的代码。

本页面说明如何记录使用几种不同编码约定的CommonJS和Node.js的模块。如果你要记录异步模块定义（AMD）模块（比如大家熟知的"RequireJS模块"），请查看AMD Modules(模块)。

**Module identifiers（模块标识符）**

在大多数情况下，CommonJS的或Node.js的模块应该包含一个独立的JSDoc注释,这个JSDoc注释应该包含@module 标签。@module 标签的值应该是传递给require() 函数的模块标识符。例如，如果用户通过调用require('my/shirt')来加载模块，你的JSDoc注释将包含@module my/shirt标签。

如果你使用不带值的@module 标签，JSDoc会基于文件路径尝试猜测正确的模块标识符。

当您使用 JSDoc namepath(名称路径)从另一个JSDoc注释中引用一个模块，您必须添加前缀module:。例如，如果你想模块my/pants的文档 连接到模块my/shirt，您可以使用@see 标签来描述my/pants，如下：

1. /\*\*
2. \* Pants module.
3. \* @module my/pants
4. \* @see module:my/shirt
5. \*/

同样，模块中每个成员的namepath （名称路径）将以module: 开始：，后面跟模块名字。例如，如果你的my/pants模块输出一个Jeans构造函数，并且Jeans 有一个名为hem的实例方法，那么这个实例方法longname（长名称）是module:my/pants.Jeans#hem。

**Properties of the 'exports' object（'exports'对象的属性）**

这是最简单的记录标识符，直接分配'exports'对象的属性。JSDoc会自动识别模块导出的这些标识符。

在下面的例子中，my/shirt 模块导出button 和 unbutton 方法。JSDoc会自动检测模块导出的这些方法。

例如，方法添加到导出对象：

1. /\*\*
2. \* Shirt module.
3. \* @module my/shirt
4. \*/
5. /\*\* Button the shirt. \*/
6. exports.button = function() {
7. // ...
8. };
9. /\*\* Unbutton the shirt. \*/
10. exports.unbutton = function() {
11. // ...
12. };

**Values assigned to local variables （值分配给局部变量）**

在一些情况下，导出的标识符在它被加入到exports对象之前，可以被分配给一个局部变量。例如，如果你的模块导出一个wash方法，而模块本身往往就叫做wash方法，您可以编写模块，如下所示：

例如，方法分配给局部变量并添加到导出对象:

1. /\*\*
2. \* Shirt module.
3. \* @module my/shirt
4. \*/
5. /\*\* Wash the shirt. \*/
6. var wash = exports.wash = function() {
7. // ...
8. };

在这种情况下，JSDoc *不会*自动记录wash作为导出的方法，因为JSDoc注释呈现在局部变量wash之前，而不是呈现在exports.wash之前。一个解决办法是增加一个 @alias 标签，用来正确定义方法的longname（长名称）。在这种情况下，该方法是模块 my/shirt 的静态成员，所以正确的长名字是module:my/shirt.wash：

例如，longname（长名称）定义在 @alias 标签中：

1. /\*\*
2. \* Shirt module.
3. \* @module my/shirt
4. \*/
5. /\*\*
6. \* Wash the shirt.
7. \* @alias module:my/shirt.wash
8. \*/
9. var wash = exports.wash = function() {
10. // ...
11. };

另一种解决方案是将方法的JSDoc注释移动到exports.wash的上面。这种变化使得JSDoc检测到wash是由模块my/shirt导出的：

例如，JSDoc注释放在exports.wash之前：

1. /\*\*
2. \* Shirt module.
3. \* @module my/shirt
4. \*/
5. var wash =
6. /\*\* Wash the shirt. \*/
7. exports.wash = function() {
8. // ...
9. };

**Values assigned to 'module.exports' （值分配给'module.exports'）**

在Node.js的模块中，您可以直接给module.exports赋值。本节介绍如何记录分配给module.exports的不同类型的值。

**Object literal assigned to 'module.exports'（对象字面量分配给'module.exports'）**

如果一个模块将一个对象字面量分配给module.exports。JSDoc自动识别这个模块的exports只有这个值。此外，JSDoc自动为每个属性设置正确的longname（长名称）：

例如：对象字面量分配给'module.exports'：

1. /\*\*
2. \* Color mixer.
3. \* @module color/mixer
4. \*/
5. module.exports = {
6. /\*\* Blend two colors together. \*/
7. blend: function(color1, color2) {
8. // ...
9. },
10. /\*\* Darken a color by the given percentage. \*/
11. darken: function(color, percent) {
12. // ..
13. }
14. };

您也可以使用另一种模式，在module.exports对象字面量以外添加属性：

例如，通过属性定义，分配给module.exports：

1. /\*\*
2. \* Color mixer.
3. \* @module color/mixer
4. \*/
5. module.exports = {
6. /\*\* Blend two colors together. \*/
7. blend: function(color1, color2) {
8. // ...
9. }
10. };
11. /\*\* Darken a color by the given percentage. \*/
12. module.exports.darken = function(color, percent) {
13. // ..
14. };

**Function assigned to 'module.exports'（函数分配给'module.exports'）**

如果你分配一个函数给module.exports，JSDoc将自动这个函数设置正确的longname（长名称）。

例如，函数分配给'module.exports'：

1. /\*\*
2. \* Color mixer.
3. \* @module color/mixer
4. \*/
5. /\*\* Blend two colors together. \*/
6. module.exports = function(color1, color2) {
7. // ...
8. };

同样的模式适用于构造函数：

例如，构造函数分配给'module.exports'：

1. /\*\*
2. \* Color mixer.
3. \* @module color/mixer
4. \*/
5. /\*\* Create a color mixer. \*/
6. module.exports = function ColorMixer() {
7. // ...
8. };

**String, number, or boolean assigned to 'module.exports'（字符串，数字，或布尔值分配给'module.exports'）**

对于分配给module.exports值类型（字符串，数字和布尔值），你必须在和@module标签同一JSDoc注释块中通过使用 @type 标签记录导出的值类型。

例如，字符串分配给'module.exports'

1. /\*\*
2. \* Module representing the word of the day.
3. \* @module wotd
4. \* @type {string}
5. \*/
6. module.exports = 'perniciousness';

**Values assigned to 'module.exports' and local variables （值分配给'module.exports'和局部变量）**

如果你的模块导出标识符不直接分配给module.exports，您可以使用@exports 标签代替@module标签。@exports 标签告诉JSDoc，这个标识符表示是模块导出的值。

例如，对象字面量分配给一个局部变量和module.exports：

1. /\*\*
2. \* Color mixer.
3. \* @exports color/mixer
4. \*/
5. var mixer = module.exports = {
6. /\*\* Blend two colors together. \*/
7. blend: function(color1, color2) {
8. // ...
9. }
10. };

**Properties added to 'this'（属性添加到'this'）**

当一个模块添加属性到它的this对象，JSDoc3自动识别新的属性会被该模块导出。

例如，属性添加到一个模块的'this'对象：

1. /\*\* @module bookshelf \*/
2. /\*\* @class \*/
3. this.Book = function(title) {
4. /\*\* The title of the book. \*/
5. this.title = title;
6. }

**AMD Modules（模块）**

**Overview（概述）**

JSDoc3 可以使用[异步模块定义（AMD）API](https://github.com/amdjs/amdjs-api/blob/master/AMD.md)记录模块，这是由库实现的，如[RequireJS](http://requirejs.org/)。本页面说明根据您的模块使用的编码约定，如何使用JSDoc记录AMD模块。

如果你记录CommonJS或 Node.js的模块，见CommonJS 模块的说明。

**Module identifiers（模块标识符）**

当您记录一个AMD模块时，你要使用 @exports 标签或@module 标签 来记录真实传递给require()函数的标识符。例如，如果用户通过调用require('my/shirt', /\* callback \*/)来加载该模块，你会写一个包含@exports my/shirt 或 @module my/shirt标签的JSDoc注释。下面的例子可以帮助你决定使用哪些标签。

如果你使用不带值的@exports 或 @module 标签，JSDoc会基于文件路径尝试猜测正确的模块标识符。

当您使用 JSDoc namepath(名称路径)从另一个JSDoc注释中引用一个模块，您必须添加前缀module:。例如，如果你想模块my/pants的文档 连接到模块my/shirt，您可以使用@see 标签来描述my/pants，如下：

1. /\*\*
2. \* Pants module.
3. \* @module my/pants
4. \* @see module:my/shirt
5. \*/

同样，模块中每个成员的namepath （名称路径）将以module: 开始：，后面跟模块名字。例如，如果你的my/pants模块输出一个Jeans构造函数，并且Jeans 有一个名为hem的实例方法，那么这个实例方法longname（长名称）是module:my/pants.Jeans#hem。

**Function that returns an object literal（函数返回一个对象字面量）**

如果你定义AMD模块作为一个函数，该函数返回一个对象字面量， 使用@exports 标签来记录模块的名称。JSDoc会自动检测该对象的属性是模块的成员。

例如，函数返回一个对象字面量：

1. define('my/shirt', function() {
2. /\*\*
3. \* A module representing a shirt.
4. \* @exports my/shirt
5. \*/
6. var shirt = {
7. /\*\* The module's `color` property. \*/
8. color: 'black',
9. /\*\* @constructor \*/
10. Turtleneck: function(size) {
11. /\*\* The class' `size` property. \*/
12. this.size = size;
13. }
14. };
15. return shirt;
16. });

**Function that returns another function（函数返回另一个函数）**

如果你定义模块作为一个函数，导出的另一个函数，比如构造函数，你可以使用一个独立的带有@module 标签的注释来记录模块。然后，您可以使用一个@alias 标签告诉JSDoc该函数使用相同的longname（长名称）作为模块。

例如，函数返回另一个函数：

1. /\*\*
2. \* A module representing a jacket.
3. \* @module my/jacket
4. \*/
5. define('my/jacket', function() {
6. /\*\*
7. \* @constructor
8. \* @alias module:my/jacket
9. \*/
10. var Jacket = function() {
11. // ...
12. };
13. /\*\* Zip up the jacket. \*/
14. Jacket.prototype.zip = function() {
15. // ...
16. };
17. return Jacket;
18. });

**Module declared in a return statement （模块声明在return语句中）**

如果你在一个函数的return语句中声明你的模块对象，你可以使用一个独立的带有@module 标签的注释来记录模块。然后，您可以使用一个@alias 标签告诉JSDoc该函数使用相同的longname（长名称）作为模块。

例如，模块声明在return语句中：

1. /\*\*
2. \* Module representing a shirt.
3. \* @module my/shirt
4. \*/
5. define('my/shirt', function() {
6. // Do setup work here.
7. return /\*\* @alias module:my/shirt \*/ {
8. /\*\* Color. \*/
9. color: 'black',
10. /\*\* Size. \*/
11. size: 'unisize'
12. };
13. });

**Module object passed to a function（模块对象传递给一个函数）**

如果模块对象传递到定义你的模块的函数，你可以通过给函数参数添加@exports 标签来记录模块。这种模式在JSDoc3.3.0及更高版本中支持。

例如，模块对象传递给一个函数：

1. define('my/jacket', function(
2. /\*\*
3. \* Utility functions for jackets.
4. \* @exports my/jacket
5. \*/
6. module) {
7. /\*\*
8. \* Zip up a jacket.
9. \* @param {Jacket} jacket - The jacket to zip up.
10. \*/
11. module.zip = function(jacket) {
12. // ...
13. };
14. });

**Multiple modules defined in one file（多模块定义在一个文件中）**

如果你在一个单一的JavaScript文件中定义多个AMD模块，你应该使用@exports 标签来记录每个模块对象。

例如，多模块定义在一个文件中：

1. // one module
2. define('html/utils', function() {
3. /\*\*
4. \* Utility functions to ease working with DOM elements.
5. \* @exports html/utils
6. \*/
7. var utils = {
8. /\*\* Get the value of a property on an element. \*/
9. getStyleProperty: function(element, propertyName) { }
10. };
11. /\*\* Determine if an element is in the document head. \*/
12. utils.isInHead = function(element) { }
13. return utils;
14. }
15. );
16. // another module
17. define('tag', function() {
18. /\*\* @exports tag \*/
19. var tag = {
20. /\*\* @class \*/
21. Tag: function(tagName) {
22. // ...
23. }
24. };
25. return tag;
26. });

# 块标签（Block Tags）

| 标签名 | **别名** | **描述** |
| --- | --- | --- |
| [@abstract](https://www.shouce.ren/api/view/a/13247) | @virtual | 这个成员必须在继承类中实现（或重写）。 |
| [@access](https://www.shouce.ren/api/view/a/13248) |  | 指定该成员的访问级别（私有 private，公共 public，或保护 protected） |
| [@alias](https://www.shouce.ren/api/view/a/13249) |  | 标记成员有一个别名 |
| [@augments](https://www.shouce.ren/api/view/a/13250) | @extends | 指名这个子类继承至哪个父类，后面需要加父类名 |
| [@author](https://www.shouce.ren/api/view/a/13251) |  | 指定项目的作者 |
| [@borrows](https://www.shouce.ren/api/view/a/13252) |  | 这个对象使用另一个对象的某些东西 |
| [@callback](https://www.shouce.ren/api/view/a/13253) |  | 描述一个回调函数。 |
| [@class](https://www.shouce.ren/api/view/a/13254) | @constructor | 此函数旨在需要使用”new”关键字调用，即构造函数。 |
| [@classdesc](https://www.shouce.ren/api/view/a/13255) |  | 使用后面的文字来描述整个类 |
| [@constant](https://www.shouce.ren/api/view/a/13256) | @const | 记录一个对象作为一个常量 |
| [@constructs](https://www.shouce.ren/api/view/a/13257) |  | 这个函数成员将成为类的构造函数 |
| [@copyright](https://www.shouce.ren/api/view/a/13258) |  | 描述一些版权信息 |
| [@default](https://www.shouce.ren/api/view/a/13259) | @defaultvalue | 记录默认值 |
| [@deprecated](https://www.shouce.ren/api/view/a/13260) |  | 说明这已不再是首选方法,即已经弃用 |
| [@description](https://www.shouce.ren/api/view/a/13261) | @desc | 描述一个标识 |
| [@enum](https://www.shouce.ren/api/view/a/13262) |  | 描述一个相关属性的集合。 |
| [@event](https://www.shouce.ren/api/view/a/13263) |  | 描述一个事件。 |
| [@example](https://www.shouce.ren/api/view/a/13264) |  | 提供一个如何使用描述项的例子。 |
| [@exports](https://www.shouce.ren/api/view/a/13265) |  | 标识一个由JavaScript模块导出的成员。 |
| [@external](https://www.shouce.ren/api/view/a/13266) |  | 标识一个外部的类，命名空间，或模块。 |
| [@file](https://www.shouce.ren/api/view/a/13267) | @fileoverview 或 @overview | 描述一个文件 |
| [@fires](https://www.shouce.ren/api/view/a/13268) | @emits | 描述事件这个方法可能会触发。 |
| [@function](https://www.shouce.ren/api/view/a/13269) | @func 或 @method | 描述一个函数或方法。 |
| [@global](https://www.shouce.ren/api/view/a/13270) |  | 记录一个全局对象。 |
| [@ignore](https://www.shouce.ren/api/view/a/13271) |  | 忽略文档中的一个标识。 |
| [@implements](https://www.shouce.ren/api/view/a/13272) |  | 这个标识实现一个接口。 |
| [@inheritdoc](https://www.shouce.ren/api/view/a/13273) |  | 指明这个标识应继承其父类的文档 |
| [@inner](https://www.shouce.ren/api/view/a/13274) |  | 描述一个内部对象。 |
| [@instance](https://www.shouce.ren/api/view/a/13275) |  | 记录一个实例成员。 |
| [@interface](https://www.shouce.ren/api/view/a/13276) |  | 这是别人可以实现的一个接口。 |
| [@kind](https://www.shouce.ren/api/view/a/13277) |  | 标识的类型。 |
| [@lends](https://www.shouce.ren/api/view/a/13278) |  | 将一个字面量对象的所有属性标记为某个标识符（类或模块）的成员。 |
| [@license](https://www.shouce.ren/api/view/a/13279) |  | 标识你的代码采用何种软件许可协议。 |
| [@listens](https://www.shouce.ren/api/view/a/13280) |  | 列出一个标识的监听事件。 |
| [@member](https://www.shouce.ren/api/view/a/13281) | @var | 记录一个成员。 |
| [@memberof](https://www.shouce.ren/api/view/a/13282) |  | 标明这个标识属于哪个父级标识。 |
| [@mixes](https://www.shouce.ren/api/view/a/13283) |  | 此对象混入了另一个对象中的所有成员。 |
| [@mixin](https://www.shouce.ren/api/view/a/13284) |  | 记录一个mixin（混入）对象。 |
| [@module](https://www.shouce.ren/api/view/a/13285) |  | 记录一个 JavaScript 模块。 |
| [@name](https://www.shouce.ren/api/view/a/13286) |  | 记录一个对象的名称。 |
| [@namespace](https://www.shouce.ren/api/view/a/13287) |  | 记录一个命名空间对象。 |
| [@override](https://www.shouce.ren/api/view/a/13288) |  | 指明一个标识符覆盖其父类同名的标识符。 |
| [@param](https://www.shouce.ren/api/view/a/13289) | @arg 或 @argument | 记录传递给一个函数的参数 |
| [@private](https://www.shouce.ren/api/view/a/13290) |  | 标记为私有的。 |
| [@property](https://www.shouce.ren/api/view/a/13291) | @prop | 记录一个对象的属性。 |
| [@protected](https://www.shouce.ren/api/view/a/13292) |  | 标记为受保护的。 |
| [@public](https://www.shouce.ren/api/view/a/13293) |  | 标记为公开的。 |
| [@readonly](https://www.shouce.ren/api/view/a/13294) |  | 标记为只读的。 |
| [@requires](https://www.shouce.ren/api/view/a/13295) |  | 这个文件需要一个 JavaScript 模块。 |
| [@returns](https://www.shouce.ren/api/view/a/13296) | @return | 记录一个函数的返回值。 |
| [@see](https://www.shouce.ren/api/view/a/13297) |  | 更多详细信息请参阅其他一些文档。 |
| [@since](https://www.shouce.ren/api/view/a/13298) |  | 此功能何时被添加进来的? |
| [@static](https://www.shouce.ren/api/view/a/13299) |  | 记录一个静态成员。 |
| [@summary](https://www.shouce.ren/api/view/a/13300) |  | 完整描述的一个简写版本。 |
| [@this](https://www.shouce.ren/api/view/a/13301) |  | this关键字的指向 |
| [@throws](https://www.shouce.ren/api/view/a/13302) |  | 说明可能会被抛出什么样的错误。 |
| [@todo](https://www.shouce.ren/api/view/a/13303) |  | 记录一个将要完成的任务。 |
| [@tutorial](https://www.shouce.ren/api/view/a/13304) |  | 插入一个链接到包含教程文件。 |
| [@type](https://www.shouce.ren/api/view/a/13305) |  | 记录一个对象的类型。 |
| [@typedef](https://www.shouce.ren/api/view/a/13306) |  | 记录一个自定义的类型。 |
| [@variation](https://www.shouce.ren/api/view/a/13307) |  | 区分具有相同名称的不同的对象。 |
| [@version](https://www.shouce.ren/api/view/a/13308) |  | 记录版本信息 |

**内联标签（Inline Tags）**

| 标签名 | **别名** | **描述** |
| --- | --- | --- |
| [@link](https://www.shouce.ren/api/view/a/13309) | @linkcode 或 @linkplain | 链接到文档中的另一个项目。 |
| [@tutorial](https://www.shouce.ren/api/view/a/13310) |  | 链接到一个教程 |