JSDOC

- Official website

目录

- JSDoc3 入门
  - 入门
  - 在 JSDoc 3中使用名称路径
  - JSDOC 的命令行参数
  - 使用配置文件配置 JSDOC
  - 配置 JSDoc 的默认模板
  - 块标签和内联标签
  - 关于 JSDoc 插件
  - 使用 Markdown 插件
  - Tutorials 教程
  - 包含 Package（包）文件
  - 包含 README 文件
- JSDoc 示例
  - ES 2015 Classes
  - ES 2015 Modules
  - CommonJS Modules
  - AMD Modules
- 块标签
  - @abstract
  - @access
  - @alias
  - @async
  - @augments
  - @author
  - @borrows
  - @callback
  - @class
  - @classdesc
  - @constant
  - @constructs

- @copyright
- @default
- @deprecated
- @description
- @enum
- @event
- @example
- @exports
- @external
- @file
- @fires
- @function
- @generator
- @global
- @hideconstructor
- @ignore
- @implements
- @inheritdoc
- @inner
- @instance
- @interface
- @kind
- @lends
- @license
- @listens
- @member
- @memberof
- @mixes
- @mixin
- @module
- @name
- @namespace
- @override
- @package
- @param
- @private
- @property

    - @protected
    - @public
    - @readonly
    - @requires
    - @returns
    - @see
    - @since
    - @static
    - @summary
    - @this
    - @throws
    - @todo
    - @tutorial
    - @type
    - @typedef
    - @variation
    - @version
    - @yields
- 内联标签
    - @link
    - @tutorial
- 贡献
    - JSDoc project on GitHub
    - Use JSDoc project on GitHub

JSDoc 入门

目录

- 介绍
- 向代码中添加文档注释
- 生成网站

介绍

JSDoc 3 是一个用于 JavaScript 的API文档生成器，类似于 Javadoc 或 phpDocumentor。可以将文档注释直接添加到源代码中。JSDoc 工具将扫描您的源代码并为您生成一个 HTML 文档网站。

向代码中添加文档注释

JSDoc 的目的是记录 JavaScript 应用程序或库的 API。假设您想要记录诸如模块、名称空间、类、方法、方法参数等内容。

JSDoc注释通常应该放在记录代码之前。为了被 JSDoc 解析器识别，每个注释必须以 `/**` 序列开头。以 `/*`、`/***` 开头或超过3颗星的注释将被忽略。这个特性用于控制解析注释块的功能。

最简单的文档示例就是描述：

```
/** This is a description of the foo function. */
function foo() {
}
```

添加一个描述很简单--只需在文档注释中键入所需的说明。

可以使用特殊的 `JSDoc标签` 来提供更多信息。例如，如果函数是类的构造函数，则可以通过添加 @constructor标记来表示。

```
/**
 * Represents a book.
 * @constructor
 */
function Book(title, author) {
}
```

使用标签添加更多的信息。

```
/**
 * Represents a book.
 * @constructor
 * @param {string} title - The title of the book.
 * @param {string} author - The author of the book.
 */
function Book(title, author) {
}
```

生成网站

一旦你的代码是已注释的，你可以是用 JSDoc 3的工具从源文件中生成一个 HTML 网站。

默认情况下，JSDoc 使用内置的“默认”模板将文档转换为 HTML。您可以根据自己的需要编辑此模板，或者创建一个全新的模板（如果您喜欢的话）。

在命令行上运行文档生成器：

```
jsdoc book.js
```

此命令将在当前工作目录中创建名为 `out/` 的目录。在该目录中，您将找到生成的 HTML 页面。

JSDoc 3中使用名称路径

目录

- 名称路径
- 相关链接

JSDoc3 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：

```
myFunction
MyConstructor
MyConstructor#instanceMember
MyConstructor.staticMember
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

```
/** @constructor */
Person = function() {
  this.say = function() {
    return "I'm an instance.";
  }

  function say() {
    return "I'm inner.";
  }
}
Person.say = function() {
```

```
  return "I'm static.";
}

var p = new Person();
p.say();      // I'm an instance.
Person.say(); // I'm static.
// 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：

```
Person#say  // 名为"say"的实例方法
Person.say  // 名为"say"的静态方法
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：

```
/** @constructor */
Person = function() {
  /** @constructor */
  this.Idea = function() {
    this.consider = function(){
      return "hmmm";
    }
  }
}

var p = new Person();
var i = new p.Idea();
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`

这种链接可与连接符号（ `#`，`.`，`~` ）任意组合使用。

特殊情况：模块，外部组件和事件。

```
/** A module. Its name is module:foo/bar.
 * @module foo/bar
```

```
 */

/** The built in string object. Its name is external:String.
 * @external String
 */

/** An event. Its name is module:foo/bar.event:MyEvent.
 * @event module:foo/bar.event:MyEvent
 */
```

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。

```
/** @namespace */
var chat = {
  /**
   * Refer to this by {@link chat."#channel"}.
   * @namespace
   */
  "#channel": {
    /**
     * Refer to this by {@link chat."#channel".open}.
     * @type {boolean}
     * @defaultvalue
     */
    open: true,
    /**
     * Internal quotes have to be escaped by backslash. This is
     * {@link chat."#channel"."say-\"hello\""}.
     */
    'say-"hello"': function (msg) {}
  }
};

/**
 * Now we define an event in our {@link chat."#channel"} namespace.
 * @event chat."#channel"."op:announce-motd"
 */
```

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\""`。

相关链接

- Block and inline tags
- {@link}

JSDoc中的命令行参数

目录

- 示例
- 相关链接

使用 JSDoc 最基本的，像这样使用：

```
/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（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。

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>`	只运行测试，其名称中包含 value。

选项	描述
`--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。查看导览说明，以了解更多信息。
`-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：

```
/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：

```
/path/to/jsdoc -T --match tag --verbose
```

相关链接

- 使用配置文件配置 JSDoc

使用配置文件配置 JSDoc

目录

- 配置文件格式
- 默认配置选项
- 配置插件
- 指定递归深度
- 指定输入文件
- 指定源类型
- 将命令行选项合并到配置文件中
- 配置标记和标记字典
- 配置模板
- 相关链接

配置文件格式

要自定义 JSDoc 的行为，可以使用以下格式之一向 JSDoc 提供配置文件：

- 一个JSON文件。在JSDoc 3.3.0及更高版本中，此文件可能包含注释。
- 导出单个配置对象的 CommonJS 模块。JSDoc 3.5.0及更高版本支持这种格式。

要使用配置文件运行 JSDoc，请使用 `-c` 命令行选项（例如，`jsdoc -c /path/To/conf.json` 或 `jsdoc -c /path/To/conf.js`）。

下面的例子展示了一个简单的配置文件，它启用了 JSDoc 的 Markdown 插件。JSDoc 的配置选项将在下面的章节中详细说明。

JSON 配置文件：

```
{
  "plugins": ["plugins/markdown"]
```

```
}
```

JS 配置文件：

```
'use strict';
module.exports = {
  plugins: ['plugins/markdown']
};
```

更多的信息，请查阅文件 conf.json.EXAMPLE。

默认配置选项

如果不指定配置文件，JSDoc 将使用默认如下配置：

```
{
  "plugins": [],
  "recurseDepth": 10,
  "source": {
    "includePattern": ".+\\.js(doc|x)?$",
    "excludePattern": "(^|\\/|\\\\)_"
  },
  "sourceType": "module",
  "tags": {
    "allowUnknownTags": true,
    "dictionaries": ["jsdoc", "closure"]
  },
  "templates": {
    "cleverLinks": false,
    "monospaceLinks": false
  }
}
```

这意味着：

- 无插件加载（`plugins`）；
- 如果使用 -r 命令行标志启用递归，JSDoc 将搜索 10 层深的文件（`recurseDepth`）；
- 只有以 `.js`、`.jsdoc` 和 `.jsx` 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`）；
- JSDoc 支持使用 ES2015 modules（`sourceType`）；
- JSDoc 允许您使用无法识别的标签（`tags.allowUnknownTags`）；
- 标准 JSDoc 标签和 closure 标签被启用（`tags.dictionaries`）；
- @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。

配置插件

要启用插件，将它们的路径（相对于 JSDoc 文件夹）添加到插件数组中。

例如，以下 JSON 配置文件将启用 Markdown 插件（它将 Markdown 格式的文本转换为 HTML）和“summary”插件（它自动为每个 doclet 生成摘要）。

带插件的JSON配置文件：

```
{
  "plugins": [
    "plugins/markdown",
    "plugins/summarize"
  ]
}
```

有关更多信息，请参阅 plugin 参考，并在 JSDoc 的 `plugins` 目录中查找 JSDoc 内置的插件。

通过向配置文件中添加 `markdown` 对象来配置 `Markdown` 插件。有关详细信息，请参阅配置 `Markdown` 插件。

指定递归深度

`recurseDepth` 配置控制 JSDoc 递归源文件的层级数量。该属性在 JSDoc 3.5.0以及更高版本中使用。并且仅当还指定了 `-r` 命令行标志时才使用此选项，该标志告诉 JSDoc 递归搜索输入文件。

```
{
  "recurseDepth": 10
}
```

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。

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

- `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`、`.jsdoc` 和 `.jsx` 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中找到的每个文件，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中找到的每个文件，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，假设有以下文件结构：

```
myProject/
|- a.js
|- b.js
|- c.js
|- _private
|  |- a.js
|- lib/
   |- a.js
   |- ignore.js
   |- d.txt
```

另外，假设配置文件如下：

```
{
  "source": {
    "include": ["myProject/a.js", "myProject/lib", "myProject/_private"],
    "exclude": ["myProject/lib/ignore.js"],
    "includePattern": ".+\\.js(doc|x)?$",
    "excludePattern": "(^|\\/|\\\\)_"
```

```
  }
}
```

在 `myProject` 根文件夹运行 `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：

- 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)
- 应用 `source.includePattern`，剩下除了 `myProject/lib/d.txt`，因为它没有以 `.js`，`.jsdoc` 或 `.jsx` 结束。
- 应用 `source.excludePattern`，排除了 `myProject/\_private/a.js`。
- 应用 `source.exclude`，排除了 `myProject/lib/ignore.js`。

指定源类型

`sourceType` 选项影响 JSDoc 解析 JavaScript 文件的方式。此选项在 JSDoc 3.5.0及更高版本中可用，此选项接受以下值：

- `module`（默认）：对大多数类型的JavaScript文件使用此值。
- `script`：如果 JSDoc 在解析代码时以严格模式记录错误，例如删除不合格标识符，则使用此值。

```
{
  "sourceType": "module"
}
```

将命令行选项合并到配置文件中

可以将 JSDoc 的许多命令行选项放入配置文件中，而不是在命令行中指定它们。为此，将相关选项的长名称添加到配置文件的 `opts` 部分，并将值设置为该选项的值。

在配置文件中设置的命令行选项:

```
{
  "opts": {
    "template": "templates/default",  // same as -t templates/default
    "encoding": "utf8",               // same as -e utf8
    "destination": "./out/",          // same as -d ./out/
    "recurse": true,                  // same as -r
    "tutorials": "path/to/tutorials", // same as -u path/to/tutorials
  }
}
```

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配置文件中提供的选项。

配置标记和标记字典

`tags` 选项控制哪些 JSDoc 标签允许被使用和解析。

```
{
  "tags": {
    "allowUnknownTags": true,
    "dictionaries": ["jsdoc", "closure"]
  }
}
```

`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` 。

`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

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。

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"]` 。

配置模板

`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。

```
{
  "templates": {
    "cleverLinks": false,
    "monospaceLinks": false
  }
}
```

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。

相关链接

- JSDOC 的命令行参数
- 关于 JSDoc 插件
- 使用 Markdown 插件

配置 JSDoc 的默认模板

目录

- 生成适合打印的文档
- 复制静态文件到输出目录
- 在页脚显示当前日期
- 在导航栏中显示长文件名
- 重写默认模板的布局文件
- 相关链接

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。

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。

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：

```
{
  "templates": {
    "default": {
      "staticFiles": {
        "include": [
          "./myproject/static"
        ]
      }
    }
  }
}
```

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。

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。

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。

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。

相关链接

- 使用配置文件配置 JSDoc

块标签和内联标签

目录

- 概述
- 示例

概述

JSDoc支持两种不同类型的标签：

- 块标签(Block), 这是在一个JSDoc注释的最高级别。
- 内联标签(inline), 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：

```
/**
 * Set the shoe's color. Use {@link Shoe#setSize} to set the shoe size.
 *
 * @param {string} color - The shoe's color.
 */
Shoe.prototype.setColor = function(color) {
  // ...
};
```

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：

```
/**
 * Set the shoe's color.
 *
 * @param {SHOE_COLORS} color - The shoe color. Must be an enumerated
 * value of {@link SHOE_COLORS}.
 */
Shoe.prototype.setColor = function(color) {
  // ...
};
```

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：

```
/**
 * Set the color and type of the shoelaces.
 *
 * @param {LACE_COLORS} color - The shoelace color.
 * @param {LACE_TYPES} type - The type of shoelace.
 */
Shoe.prototype.setLaceType = function(color, type) {
```

```
  // ...
};
```

关于 JSDoc 插件

目录

- 创建并启用插件
- 创建SDoc3插件
  - 事件处理程序
  - 标签定义
  - 节点访问者
- 报告错误

创建并启用插件

创建并启用新 JSDoc 插件,需要两个步骤：

- 创建一个包含你的插件代码的 JavaScript 模块.
- 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：

```
{
  "plugins": ["plugins/shout"]
}
```

JSDoc 按配置文件 `plugins` 数组中列出的顺序执行的插件。

创建SDoc3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

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，例如：

例如，事件处理程序插件 'newDoclet' 事件：

```
exports.handlers = {
  newDoclet: function(e) {
    // Do something when we see a new doclet
  }
};
```

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。

Event: 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。

Event: 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。

Event: 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：

```
exports.handlers = {
  beforeParse: function(e) {
    var extraDoc = [
      '/**',
      ' * Function provided by a superclass.',
      ' * @name superFunc',
      ' * @memberof ui.mywidget',
      ' * @function',
      ' */'
    ];
    e.source += extraDoc.join('\n');
  }
};
```

Event: 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；
- `columnno`：找到注释的列号，JSDoc 3.5.0及更高版本中提供。

Event: 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。
- `columnno`：找到注释的列号，JSDoc 3.5.0及更高版本中提供。
- `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。

Event: 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。

要查看 JSDoc 为代码生成的 doclet，请使用 -x命令行选项运行 JSDoc。

下面是一个 `newDoclet` 处理程序的一个例子说明：

```
exports.handlers = {
  newDoclet: function(e) {
    // e.doclet will refer to the newly created doclet
    // you can read and modify properties of that doclet if you wish
    if (typeof e.doclet.description === 'string') {
      e.doclet.description = e.doclet.description.toUpperCase();
    }
  }
};
```

Event: 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。

Event: 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，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。

Event: 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，有关每个的 doclet 可以包含属性的详细信息。

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：

```
exports.defineTags = function(dictionary) {
  // define tags here
};
```

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:

```
dictionary.defineTag('instance', {
  onTagged: function(doclet, tag) {
    doclet.scope = "instance";
  }
});
```

`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:

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

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：

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

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，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`: JSDoc 解析器实例。
- `currentSourceName`: 被解析的文件名。

触发事情发生

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`）。

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：

```
var logger = require('jsdoc/util/logger');

exports.handlers = {
  newDoclet: function(e) {
    // Your code here.

    if (somethingBadHappened) {
      logger.error('Oh, no, something bad happened!');
    }
  }
};
```

使用Markdown插件

目录

- 概述
- 启用markdown插件
- 在额外的JSDoc标签中转换Markdown
- 剔除markdown默认处理的标签
- 用换行符换行文本
- 添加ID属性到标题标签

概述

JSDoc 包括 `Markdown` 插件，自动把 `Markdown` 格式文本转换成 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。

注意：当您启用 `Markdown` 插件，一定要在 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 (在 JSDoc 注释的开头包含未标记的描述)
- @param
- @property
- @returns
- @see
- @throws

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：

```
{
  "plugins": ["plugins/markdown"]
}
```

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 标签的 `Markdown` 文本。您可以通过添加一个 `markdown.tags` 属性到 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` 标签接收在一个的 doclet 的 foo 和 bar 属性中存储值，你可以通过添加下面的设置到 JSDoc 配置文件，来使用 `Markdown` 处理这些标签。

转换’ foo’ 和’ bar’ 标签中的 Markdown:

```
{
  "plugins": ["plugins/markdown"],
  "markdown": {
    "tags": ["foo", "bar"]
```

```
  }
}
```

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，添加一个 `markdown.excludeTags` 属性到 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：

```
{
  "plugins": ["plugins/markdown"],
  "markdown": {
    "excludeTags": ["author"]
  }
}
```

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。这是因为正常的 JSDoc 注释可以多行。如果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。

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

目录

- 添加教程
- 配置标题，顺序和层次结构
- 从API文档链接到教程
- @tutorial 块标签
- {@tutorial} 内联标签

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 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。

添加教程

添加教程到 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：

```
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，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。

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：

```
{
  "tutorial1": {
    "title": "Tutorial One",
    "children": {
      "childA": {
        "title": "Child A"
      },
      "childB": {
        "title": "Child B"
      }
    }
  },
  "tutorial2": {
    "title": "Tutorial Two"
  }
}
```

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：

```
{
  "tutorial1": {
    "title": "Tutorial One",
    "children": ["childA", "childB"]
  },
  "tutorial2": {
    "title": "Tutorial Two"
  },
  "childA": {
    "title": "Child A"
  },
  "childB": {
    "title": "Child B"
  }
}
```

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。

从API文档链接到教程

有多种方式从 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:

```
/**
 * Class representing a socket connection.
 *
 * @class
 * @tutorial socket-tutorial
 */
function Socket() {}
```

{@tutorial} 内联标签

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:

```
/**
 * Class representing a socket connection. See {@tutorial socket-tutorial}
 * for an overview.
 *
 * @class
 */
function Socket() {}
```

包含 Package（包）文件

目录

- 示例

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 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。

示例

在源路径中包含一个包文件:

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

使用 `-P/--package` 选项：

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

包含 README 文件

目录

- 示例

有两种方法可以将 `README` 文件中的信息合并到您的文档：

1. 在 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。

示例

在源路径中包含一个 `README` 文件:

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

使用 `-R/--readme` 选项：

```
jsdoc --readme path/to/readme/README path/to/js
```

ES 2015 Classes

目录

- 文档化一个简单的类
- 扩展类
- 相关链接

JSDoc3 描述一个遵循 ECMAScript 2015 规范的类是很简单的。你并不需要使用诸如如 `@class` 和 `@constructor` 的标签来描述 ES2015 classes，JSDoc 通过分析你的代码会自动识别类和它们的构造函数。ES2015 classes 在 JSDoc3.4.0 及更高版本支持。

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：

```
/** Class representing a point. */
class Point {
  /**
    * Create a point.
    * @param {number} x - The x value.
    * @param {number} y - The y value.
    */
  constructor(x, y) {
    // ...
  }

  /**
    * Get the x value.
    * @return {number} The x value.
    */
  getX() {
    // ...
  }

  /**
    * Get the y value.
    * @return {number} The y value.
    */
  getY() {
    // ...
  }

  /**
    * Convert a string containing two comma-separated numbers into a point.
```

```
    * @param {string} str - The string containing two comma-separated numbers.
    * @return {Point} A Point object.
    */
  static fromString(str) {
    // ...
  }
}
```

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：

```
/** Class representing a point. */
const Point = class {
  // and so on
}
```

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 ES2015 类：

```
/**
 * Class representing a dot.
 * @extends Point
 */
class Dot extends Point {
  /**
   * Create a dot.
   * @param {number} x - The x value.
   * @param {number} y - The y value.
   * @param {number} width - The width of the dot, in pixels.
   */
  constructor(x, y, width) {
    // ...
  }

  /**
   * Get the dot's width.
   * @return {number} The dot's width, in pixels.
   */
  getWidth() {
    // ...
  }
}
```

相关链接

- @augments

ES 2015 Modules

目录

- 模块标识符
- 导出值
- 相关链接

JSDoc3 能够记录遵循 ECMAScript 2015规范的模块。ES2015 模块在JSDoc3.4.0及更高版本中支持。

模块标识符

当描述一个 ES2015 module（模块）时，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：

```
/**
  * Pants module.
  * @module my/pants
  * @see module:my/shirt
  */
```

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`。

导出值

下面的示例演示如何在 ES2015 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：

```
/** @module color/mixer */

/** The name of the module. */
export const name = 'mixer';

/** The most recent blended color. */
export var lastColor = null;

/**
  * Blend two colors together.
  * @param {string} color1 - The first color, in hexadecimal format.
  * @param {string} color2 - The second color, in hexadecimal format.
  * @return {string} The blended color.
  */
export function blend(color1, color2) {}

// convert color to array of RGB values (0-255)
function rgbify(color) {}

export {
  /**
    * Get the red, green, and blue values of a color.
    * @function
    * @param {string} color - A color, in hexadecimal format.
    * @returns {Array.<number>} An array of the red, green, and blue values,
    * each ranging from 0 to 255.
    */
  rgbify as toRgb
}
```

相关链接

- 在 JSDoc 3中使用名称路径
- @module

CommonJS Modules

目录

- 概述
- 模块标识
- exports对象的属性
- 值分配给局部变量
- 值分配给module.exports
  - 对象字面量分配给module.exports

    - 函数分配给module.exports
    - 字符串，数字，或布尔值分配给module.exports
- 值分配给module.exports和局部变量
- 属性添加到this
- 相关链接

概述

为了帮助记录 CommonJS modules（模块），JSDoc 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 接受 Node.js modules（模块）的约定，它扩展了 CommonJS 的标准（例如，将值分配给 `module.exports` ）。根据所遵循编码约定，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)。

模块标识

在大多数情况下，CommonJS 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 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：

```
/**
  * Pants module.
  * @module my/pants
  * @see module:my/shirt
  */
```

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` 。

exports对象的属性

最容易记录直接指定给 `exports` 对象的属性的符号。JSDoc 将自动识别模块导出这些符号。

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：

```
/**
  * Shirt module.
  * @module my/shirt
  */

/** Button the shirt. */
exports.button = function() {
  // ...
};

/** Unbutton the shirt. */
exports.unbutton = function() {
  // ...
};
```

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。

例如，方法分配给局部变量并添加到导出对象:

```
/**
  * Shirt module.
  * @module my/shirt
  */

/** Wash the shirt. */
var wash = exports.wash = function() {
  // ...
};
```

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，长名称定义在 `@alias` 标签中：

```
/**
  * Shirt module.
```

```
  * @module my/shirt
  */

/**
  * Wash the shirt.
  * @alias module:my/shirt.wash
  */
var wash = exports.wash = function() {
  // ...
};
```

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：

```
/**
  * Shirt module.
  * @module my/shirt
  */

var wash =
/** Wash the shirt. */
exports.wash = function() {
  // ...
};
```

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。

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长名称。

例如：对象字面量分配给'module.exports'：

```
/**
 * Color mixer.
 * @module color/mixer
 */

module.exports = {
  /**
    * Blend two colors together.
    * @param {string} color1 - The first color, in hexadecimal format.
```

```
    * @param {string} color2 - The second color, in hexadecimal format.
    * @return {string} The blended color.
    */
  blend: function(color1, color2) {
    // ...
  },

  /**
    * Darken a color by the given percentage.
    * @param {string} color - The color, in hexadecimal format.
    * @param {number} percent - The percentage, ranging from 0 to 100.
    * @return {string} The darkened color.
    */
  darken: function(color, percent) {
    // ..
  }
};
```

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` ：

```
/**
 * Color mixer.
 * @module color/mixer
 */

module.exports = {
  /**
    * Blend two colors together.
    * @param {string} color1 - The first color, in hexadecimal format.
    * @param {string} color2 - The second color, in hexadecimal format.
    * @return {string} The blended color.
    */
  blend: function(color1, color2) {
    // ...
  }
};

/**
  * Darken a color by the given percentage.
  * @param {string} color - The color, in hexadecimal format.
  * @param {number} percent - The percentage, ranging from 0 to 100.
  * @return {string} The darkened color.
  */
module.exports.darken = function(color, percent) {
// ..
};
```

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长名称。

例如，函数分配给'module.exports'：

```
/**
  * Color mixer.
  * @module color/mixer
  */

/**
  * Blend two colors together.
  * @param {string} color1 - The first color, in hexadecimal format.
  * @param {string} color2 - The second color, in hexadecimal format.
  * @return {string} The blended color.
  */
module.exports = function(color1, color2) {
  // ...
};
```

同样的模式适用于构造函数。

例如，构造函数分配给 'module.exports'：

```
/**
  * Color mixer.
  * @module color/mixer
  */

/** Create a color mixer. */
module.exports = function ColorMixer() {
  // ...
};
```

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），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':

```
/**
  * Module representing the word of the day.
  * @module wotd
  * @type {string}
  */

module.exports = 'perniciousness';
```

值分配给module.exports和局部变量

如果模块导出标识符不直接分配给 `module.exports` ，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` ：

```
/**
  * Color mixer.
  * @exports color/mixer
  */
var mixer = module.exports = {
  /**
    * Blend two colors together.
    * @param {string} color1 - The first color, in hexadecimal format.
    * @param {string} color2 - The second color, in hexadecimal format.
    * @return {string} The blended color.
    */
  blend: function(color1, color2) {
    // ...
  }
};
```

属性添加到this

当一个模块添加属性到它的 `this` 对象，JSDoc 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：

```
/**
  * Module for bookshelf-related utilities.
  * @module bookshelf
  */

/**
  * Create a new Book.
  * @class
  * @param {string} title - The title of the book.
  */
this.Book = function(title) {
  /** The title of the book. */
  this.title = title;
}
```

相关链接

- 在 JSDoc 3中使用名称路径

- @exports
- @module

AMD Modules

目录

- 概述
- 模块标识符
- 函数返回一个对象字面量
- 函数返回另一个函数
- 模块声明在return语句中
- 模块对象传递给一个函数
- 多模块定义在一个文件中
- 相关链接

概述

JSDoc3 可以使用异步模块定义AMD API记录模块，这是由库实现的，如 RequireJS。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。

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 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：

```
/**
  * Pants module.
  * @module my/pants
  * @see module:my/shirt
  */
```

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`。

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：

```
define('my/shirt', function() {
 /**
  * A module representing a shirt.
  * @exports my/shirt
  */
  var shirt = {
    /** The module's `color` property. */
    color: 'black',

    /**
      * Create a new Turtleneck.
      * @class
      * @param {string} size - The size (`XS`, `S`, `M`, `L`, `XL`, or `XXL`).
      */
    Turtleneck: function(size) {
      /** The class's `size` property. */
      this.size = size;
    }
  };

  return shirt;
});
```

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 该函数使用相同的长名称作为模块。

例如，函数返回另一个函数：

```
/**
  * A module representing a jacket.
  * @module my/jacket
  */
```

```
define('my/jacket', function() {
  /**
    * Create a new jacket.
    * @class
    * @alias module:my/jacket
    */
  var Jacket = function() {
    // ...
  };

  /** Zip up the jacket. */
  Jacket.prototype.zip = function() {
    // ...
  };

  return Jacket;
});
```

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 该函数使用相同的长名称作为模块。

例如，模块声明在 `return` 语句中：

```
/**
  * Module representing a shirt.
  * @module my/shirt
  */

define('my/shirt', function() {
  // Do setup work here.

  return /** @alias module:my/shirt */ {
    /** Color. */
    color: 'black',
    /** Size. */
    size: 'unisize'
  };
});
```

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：

```
define('my/jacket', function(
  /**
    * Utility functions for jackets.
    * @exports my/jacket
    */
  module) {

  /**
    * Zip up a jacket.
    * @param {Jacket} jacket - The jacket to zip up.
    */
  module.zip = function(jacket) {
    // ...
  };
});
```

多模块定义在一个文件中

如果在一个单一的 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：

```
// one module
define('html/utils', function() {
  /**
    * Utility functions to ease working with DOM elements.
    * @exports html/utils
    */
  var utils = {
    /**
      * Get the value of a property on an element.
      * @param {HTMLElement} element      - The element.
      * @param {string}      propertyName - The name of the property.
      * @return {*}           The value of the property.
      */
    getStyleProperty: function(element, propertyName) { }
  };

  /**
    * Determine if an element is in the document head.
    * @param {HTMLElement} element - The element.
    * @return {boolean}     Set to `true` if the element is in the document head,
    *                               `false` otherwise.
    */
  utils.isInHead = function(element) { }

    return utils;
  }
);
```

```
// another module
define('tag', function() {
    /** @exports tag */
    var tag = {
    /**
      * Create a new Tag.
      * @class
      * @param {string} tagName - The name of the tag.
      */
    Tag: function(tagName) {
      // ...
    }
  };

  return tag;
});
```

相关链接

- 在 JSDoc 3中使用名称路径
- @exports
- @module

@abstract

目录

- 别名
- 概述
- 实例

别名

`@virtual`

概述

`@abstract` 标记标识必须由继承该成员的对象实现（或重写）的成员。

实例

例如，父类的抽象方法，子类实现该方法：

```
/**
  * Generic dairy product.
  * @constructor
  */
function DairyProduct() {}

/**
  * Check whether the dairy product is solid at room temperature.
  * @abstract
  * @return {boolean}
  */
DairyProduct.prototype.isSolid = function() {
  throw new Error('must be implemented by subclass!');
};

/**
  * Cool, refreshing milk.
  * @constructor
  * @augments DairyProduct
  */
function Milk() {}

/**
  * Check whether milk is solid at room temperature.
  * @return {boolean} Always returns false.
  */
Milk.prototype.isSolid = function() {
  return false;
};
```

@access

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@access <package|private|protected|public>
```

概述

`@access` 指定该成员的访问级别（包 `package`，私有 `private`，公共 `public`，或保护 `protected`）。你可以使用与 `@access` 标签同义的其他标签：

- `@access package` 等价于 `@package`，属性在 JSDoc 3.5.0 以上版本可用;
- `@access private` 等价于 `@private`;
- `@access protected` 等价于 `@protected`;
- `@access public` 等价于 `@public`;

私有成员不会显示在生成的输出文档中，除非通过 `-p/--private` 命令行选项运行 JSDoc。在 JSDoc3.3.0 或更高版本，您还可以使用 `-a/--access` 命令行选项来改变这种行为。

请注意，doclet 的访问级别不用于他们的 scope (作用域)。例如，如果 Parent 有一个名为 child 的内部变量，那么这个内部变量将被记录为 `@public`，child 变量仍然是被视为一个内部变量，其 namepath 为 `Parent~child`。 换一种说法，child 变量将有一个内部作用域，即使这个变量是公开的。 要更改 doclet 的作用域，请使用 `@instance`，`@static`，和 `@global` 标签。

实例

可以使用与 `@access` 标签同义的其他标签：

```
/** @constructor */
function Thingy() {

  /** @access private */
  var foo = 0;

  /** @access protected */
  this._bar = 1;

  /** @access package */
  this.baz = 2;

  /** @access public */
  this.pez = 3;

}

// same as...

/** @constructor */
function OtherThingy() {

  /** @private */
  var foo = 0;

  /** @protected */
  this._bar = 1;

  /** @package */
```

```
  this.baz = 2;

  /** @public */
  this.pez = 3;

}
```

相关链接

- @global
- @instance
- @package
- @private
- @protected
- @public
- @static

@alias

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@alias <aliasNamepath>
```

概述

`@alias` 标签标记成员有一个别名。如果该成员有不同的名称，JSDoc 把所有引用作为这个成员。如果你在一个内部函数中定义一个类的时候，这个标签是非常有用的;在这种情况下，您可以使用 `@alias` 标签告诉 JSDoc，这个类如何在您的应用程序中暴露出来。

虽然 `@alias` 标签听起来类似于 @name 标签，但是他们的行为非常不同。`@name` 标签告诉 JSDoc 忽略与注释关联的所有代码。例如，当 JSDoc 处理下面的代码的时候，它忽略了 `bar` 的注释关联到一个 `foo` 函数：

```
/**
  * Bar function.
  * @name bar
  */
function foo() {}
```

`@alias` 标记告诉 JSDoc 这是一个伪装，成员 A 实际上叫做成员 B。例如，当 JSDoc 处理下面代码的时候，它承认 `foo` 是一个函数，然后在生产的文档中将 `foo` 改名为 `bar`：

```
/**
  * Bar function.
  * @alias bar
  */
function foo() {}
```

实例

假设你正在使用类框架，希望当你定义一个类的时候，你只要传递一个构造函数。您可以使用 `@alias` 标签告诉 JSDoc，这个类如何在您的应用程序中暴露出来。

在下面的例子中，在 `@alias` 标签告诉 JSDoc 处理匿名函数，就好像它是 `trackr.CookieManager` 类的构造函数。在这个函数中，JSDoc 将 `this` 关键字解释为 `trackr.CookieManager`，因此，“value”方法的 namepath(名称路径)为 `trackr.CookieManager#value`。

例如，匿名的构造函数使用 `@alias`：

```
Klass(
  "trackr.CookieManager",

  /**
   * @class
   * @alias trackr.CookieManager
   * @param {Object} kv
   */
  function(kv) {
    /** The value. */
    this.value = kv;
  }
);
```

您也可以在一个立即调用的函数表达式（IIFE）中创建的成员中使用 `@alias` 标签。`@alias` 标签告诉 JSDoc，这些成员都暴露在 IIFE 作用域之外的。

例如，命名空间的静态方法使用 `@alias`：

```
/** @namespace */
var Apple = {};

(function(ns) {
  /**
   * @namespace
   * @alias Apple.Core
   */
  var core = {};

  /** Documented as Apple.Core.seed */
  core.seed = function() {};

  ns.Core = core;
})(Apple);
```

对于那些对象字面量中定义的成员，可以使用 `@alias` 标签替代的 @lends标记。

```
// Documenting objectA with @alias

var objectA = (function() {
  /**
   * Documented as objectA
   * @alias objectA
   * @namespace
   */
  var x = {
    /**
     * Documented as objectA.myProperty
     * @member
     */
    myProperty: "foo"
  };

  return x;
})();

// Documenting objectB with @lends

/**
 * Documented as objectB
 * @namespace
 */
var objectB = (function() {
  /** @lends objectB */
  var x = {
    /**
     * Documented as objectB.myProperty
     * @member
     */
    myProperty: "bar"
  };
```

```
  return x;
})();
```

相关链接

- @name
- @lends

@async

目录

- 语法
- 概述
- 实例

语法

```
@async
```

概述

`@async` 标记表示函数是异步的，这意味着它是使用语法 `async function foo () {}` 声明的。不要将此标记用于其他类型的异步函数，例如提供回调的函数。JSDoc 3.5.0 及更高版本中提供了此标记。

一般来说，不需要使用此标记，因为 JSDoc 会自动检测异步函数并在生成的文档中标识它们。但是，如果您正在为代码中没有出现的异步函数编写虚拟注释，则可以使用此标记告诉JSDoc该函数是异步的。

实例

以下示例显示使用 `@async` 标记的虚拟注释：

```
/**
 * Download data from the specified URL.
 *
 * @async
 * @function downloadData
```

```
 * @param {string} url - The URL to download from.
 * @return {Promise<string>} The data from the URL.
 */
```

@augments

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@extends
```

语法

```
@augments <namepath>
```

概述

`@augments` 或 `@extends` 标签指明标识符继承自哪个父类，后面需要加父类名。你可以使用这个标签来记录基于类和并基于原型的继承。

在 JSDoc3.3.0 或更高版本中，如果一个标识符继承自多个父类，并且多个父类有同名的成员，JSDoc 使用来自列出的 JSDoc 注释中最后一个父类的文档。

实例

在下面的例子中，`Duck` 类被定义为 `Animal` 的子类。`Duck` 实例和 `Animal` 实例具有相同的属性，`speak` 方法是 `Duck` 实例所独有的。

```
/**
 * @constructor
 */
function Animal() {
```

```
  /** Is this animal alive? */
  this.alive = true;
}

/**
 * @constructor
 * @augments Animal
 */
function Duck() {}
Duck.prototype = new Animal();

/** What do ducks say? */
Duck.prototype.speak = function() {
  if (this.alive) {
    alert("Quack!");
  }
};

var d = new Duck();
d.speak(); // Quack!
d.alive = false;
d.speak(); // (nothing)
```

在下面的例子中，`Duck` 类继承自 `Flyable` 和 `Bird` 类，这两个父类都定义了一个 `takeOff` 方法。由于 `@augments Bird` 是在 `Duck` 文档列表中最后，JSDoc 自动使用 `Bird#takeOff` 注释来记录 `Duck#takeOff`。

例如，用重写方法来实现多重继承：

```
/**
 * Abstract class for things that can fly.
 * @class
 */
function Flyable() {
  this.canFly = true;
}

/** Take off. */
Flyable.prototype.takeOff = function() {
  // ...
};

/**
 * Abstract class representing a bird.
 * @class
 */
function Bird(canFly) {
  this.canFly = canFly;
}

/** Spread your wings and fly, if possible. */
Bird.prototype.takeOff = function() {
```

```
  if (this.canFly) {
    this._spreadWings()
      ._run()
      ._flapWings();
  }
};

/**
 * Class representing a duck.
 * @class
 * @augments Flyable
 * @augments Bird
 */
function Duck() {}

// Described in the docs as "Spread your wings and fly, if possible."
Duck.prototype.takeOff = function() {
  // ...
};
```

相关链接

- @borrows
- @class
- @mixes
- @mixin

@author

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@author <name> [<emailAddress>]
```

概述

`@author` 标签标识一个项目的作者。在 JSDoc3.2 和更高版本中，如果作者的名字后面跟着尖括号括起来的电子邮件地址， 默认模板将电子邮件地址转换为 `mailto:` 链接。

实例

例如，描述项目的作者:

```
/**
 * @author Jane Smith <jsmith@example.com>
 */
function MyClass() {}
```

相关链接

- @file
- @version

@borrows

目录

- 语法
- 概述
- 实例

语法

```
@borrows <that namepath> as <this namepath>
```

概述

`@borrows` 标签允许将另一个标识符的描述添加到当前描述。

如果你不止在一个地方引用同一个函数，但是你又不想重复添加同样的文档描述到多个地方，这个时候非常有用。

实例

在这个例子中， `trstr` 函数存在文档，但 `util.trim` 只是使用不同的名称引用相同的功能。

例如，复制 `trstr` 的文档描述给 `util.trim`：

```
/**
 * @namespace
 * @borrows trstr as trim
 */
var util = {
  trim: trstr
};

/**
 * Remove whitespace from around a string.
 * @param {string} str
 */
function trstr(str) {}
```

@callback

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@callback <namepath>
```

概述

`@Callback` 标签提供回调函数（可传递给其他函数）的描述，包括回调的参数和返回值。可以包涵任何一个能提供给 @method标签。

一旦你定义了一个回调，你可以像 @typedef 标签所定义的自定义类型那样使用它。尤其是，你可以使用回调的名称作为类型名称。这样您可以使你明确指明函数参数应包含那个回调。

如果你想要一个回调显示为某个特定类的类型定义，可以给回调加一个 `namepath` ，指示它是某个类的一个内部函数。还可以定义一个引用多个类引用的全局的回调类型。

实例

描述一个指定类回调:

```
/**
 * @class
 */
function Requester() {}

/**
 * Send a request.
 * @param {Requester~requestCallback} cb - The callback that handles the response.
 */
Requester.prototype.send = function(cb) {
  // code
};

/**
 * This callback is displayed as part of the Requester class.
 * @callback Requester~requestCallback
 * @param {number} responseCode
 * @param {string} responseMessage
 */
```

描述一个全局回调:

```
/**
 * @class
 */
function Requester() {}

/**
 * Send a request.
 * @param {requestCallback} cb - The callback that handles the response.
 */
Requester.prototype.send = function(cb) {
  // code
};

/**
 * This callback is displayed as a global member.
 * @callback requestCallback
 * @param {number} responseCode
 * @param {string} responseMessage
 */
```

相关链接

- @function
- @typedef

@class

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@constructor
```

语法

```
@class [<type> <name>]
```

概述

`@class` 标记将函数标记为构造函数，这意味着要使用 `new` 关键字调用它以返回实例。

实例

一个函数构建一个 `Person` 实例：

```
/**
 * Creates a new Person.
 * @class
 */
function Person() {}

var p = new Person();
```

相关链接

- @constructs

@classdesc

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@classdesc <some description>
```

概述

`@classdesc` 标签用于为类提供一个描述，这样和构造函数的描述区分开来。@classdesc标签应该与 @class (或 @constructor)标签

结合使用。

在JSDoc 3 中 @classdesc标签的功能和以前版本中的@class标签的功能是重复的。截至第3版，@class标签的语法和功能和现在的@constructor标签是完全匹配的，并且@classdesc标签更明确地传达其目的：记录一类的描述。

实例

如下所示，一个类有两个添加描述的地方，一个适用于函数本身，而另一个一般适用于类。

一个同时具有构造函数描述和类说明的 doclet：

```
/**
 * This is a description of the MyClass constructor function.
 * @class
 * @classdesc This is a description of the MyClass class.
 */
function MyClass() {}
```

相关链接

- @class

- @description

@constant

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@const
```

语法

```
@constant [<type> <name>]
```

概述

`@constant` 标记用于将文档标记为属于常量的符号。

实例

在这个实例中我们记录一个字符串常量。注意，虽然代码使用 `const` 关键字，对于 JSDoc 来说，这不是必需的。如果你的 JavaScript 宿主环境尚不支持常量声明，`@const` 描述可以很有效地用在 `var` 声明上。

一个字符串常量表示红色：

```
/** @constant
    @type {string}
    @default
*/
const RED = "FF0000";
```

```
/** @constant {number} */
var ONE = 1;
```

注意，实例中在 @type 标签中提供了一个类型是可选的。另外可选的 @default 标签用在这里也一样，这里将自动添加任何指定的值（例如， 'FF0000'）给文档。

相关链接

- @type
- @default

@constructs

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@constructs [<name>]
```

概述

当使用对象字面量形式定义类（例如使用 @lends 标签）时，可使用 `@constructs` 标签标明这个函数用来作为类的构造实例。

实例

`@constructs` 和 `@lends` 结合使用：

```
var Person = makeClass(
  /** @lends Person.prototype */
  {
    /** @constructs */
    initialize: function(name) {
      this.name = name;
    },
```

```
    /** Describe me. */
    say: function(message) {
      return this.name + " says: " + message;
    }
  }
);
```

不和 `@lends` 结合使用的时候，你必须提供一个类的名称:

```
makeClass(
  "Menu",
  /**
   * @constructs Menu
   * @param items
   */
  function(items) {},
  {
    /** @memberof Menu# */
    show: function() {}
  }
);
```

相关链接

- @lends

@copyright

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@copyright <some copyright text>
```

概述

`@copyright` 标签是用来描述一个文件的版权信息。一般和 @file 标签结合使用。

实例

```
/**
 * @file This is my cool script.
 * @copyright Michael Mathews 2011
 */
```

相关链接

- @file

@default

目录

- 别名
- 语法
- 概述
- 实例

别名

```
@defaultvalue
```

语法

```
@default [<some value>]
```

概述

`@default` 标签记录标识的赋值。可以在标签后面跟上他的值，或者当值是一个唯一被分配的简单值(可以是：一个字符串，数字，布尔值或null)的时候，可以让JSDoc从源代码中获取值，自动记录。

实例

在本实例中,一个常量被记录。该常数的值为 `0xff0000`。通过添加 `@default` 标签，这个值将自动添加到文档。

```
/**
 * @constant
 * @default
 */
const RED = 0xff0000;
```

@deprecated

目录

- 语法
- 概述
- 实例

语法

```
@deprecated [<some text>]
```

概述

`@deprecated` 标签指明一个标识在代码中已经被弃用。

实例

可以单独使用的 `@deprecated` 标签，或包括一些文本，来详细说明为什么要弃用。

```
/**
 * @deprecated since version 2.0
 */
function old() {}
```

@description

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@desc
```

语法

```
@description <some description>
```

概述

`@description` 标记允许提供正在记录一般说明。描述可以包括 HTML 标记。如果启用了 Markdown 插件，它也可包括 Markdown 格式。

实例

如果在注释开始的地方添加描述，那么可省略 `@description` 标签。

```
/**
 * Add two numbers.
 * @param {number} a
 * @param {number} b
 * @returns {number}
 */
function add(a, b) {
  return a + b;
}
```

通过使用 `@description` 标签添加的描述可放在 JSDoc 的任意地方。

```
/**
 * @param {number} a
 * @param {number} b
 * @returns {number}
 * @description Add two numbers.
 */
```

```
function add(a, b) {
  return a + b;
}
```

如果 JSDoc注 释的开头同时有一个描述和一个带有 `@description` 标记的描述，那么用 `@description` 指定的描述将覆盖注释开头的描述。

相关链接

- @classdesc
- @summary

@enum

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@enum [<type>]
```

概述

`@enum` 标记记录了一组静态属性，这些属性的值都属于同一类型。

枚举类似于属性的集合，只是枚举记录在其自己的 doc 注释中，而属性记录在其容器的 doc 注释中。此标记通常与 @readonly 一起使用，因为枚举通常表示常量的集合。

实例

这例子展示了如何记录一个表示具有三种可能状态的值的对象。请注意，如果您愿意，可以添加枚举成员的可选描述。还可以重写该类型，如“MAYBE”所示——默认情况下，枚举成员将使用与枚举本身相同的类型进行记录。

一个数字枚举，表示的3种状态：

```
/**
 * Enum for tri-state values.
 * @readonly
 * @enum {number}
 */
var triState = {
  /** The true value */
  TRUE: 1,
  FALSE: -1,
  /** @type {boolean} */
  MAYBE: true
};
```

相关链接

- @property

@event

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@event <className>#[event:]<eventName>
```

概述

`@event` 标记可以触发的事件。典型事件由一个具有一组已定义属性的对象表示。

使用 `@event` 标记定义特定类型的事件后，可以使用 `@fires` 标记指示方法可以触发该事件。还可以使用 `@listens` 标记表明用这个表示来侦听该事件。

JSDoc 会自动将名称空间 `event:` 前置到每个事件的名称。通常，当链接到另一个 doclet 中的事件时，必须包含此命名空间（ `@fires` 标签是一个特殊的例外，它可以让你忽略命名空间。 ）

注意：JSDoc 3 使用 `@event doclet` 来记录事件的内容。相反，JSDoc Toolkit 2 使用 `@event doclets` 来标识在发生同名事件时可以触发的函数。

实例

下面的示例演示如何记录一个 Hurl 类中名为 snowball 事件。该事件包含一个带有单独属性的对象。

将函数调用记录为事件：

```
/**
 * Throw a snowball.
 *
 * @fires Hurl#snowball
 */
Hurl.prototype.snowball = function() {
  /**
   * Snowball event.
   *
   * @event Hurl#snowball
   * @type {object}
   * @property {boolean} isPacked - Indicates whether the snowball is tightly packed.
   */
  this.emit("snowball", {
    isPacked: this._snowball.isPacked
  });
};
```

使用一个命名 doclet 来描述一个事件：

```
/**
 * Throw a snowball.
 *
 * @fires Hurl#snowball
 */
Hurl.prototype.snowball = function() {
  // ...
};

/**
 * Snowball event.
 *
 * @event Hurl#snowball
 * @type {object}
 * @property {boolean} isPacked - Indicates whether the snowball is tightly packed.
 */
```

相关链接

- @fires
- @listens

@example

目录

- 概述
- 实例

概述

提供如何使用文档化项的示例。此标记后面的文本将显示为突出显示的代码。

实例

注意，一个 doclet 中可以同时使用多个 `@example` 标签。

```
/**
 * Solves equations of the form a * x = b
 * @example
 * // returns 2
 * globalNS.method1(5, 10);
 * @example
 * // returns 3
 * globalNS.method(5, 15);
 * @returns {Number} Returns the value of x for the equation.
 */
globalNS.method1 = function (a, b) {
  return b / a;
};
```

`@example` 标签后面可以添加 `<caption></caption>` 标签作为示例的标题。

```
/**
 * Solves equations of the form a * x = b
 * @example <caption>Example usage of method1.</caption>
 * // returns 2
 * globalNS.method1(5, 10);
 * @returns {Number} Returns the value of x for the equation.
 */
globalNS.method1 = function (a, b) {
  return b / a;
};
```

@exports

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@exports <moduleName>
```

在 JSDoc3.3.0 或更高版本中，`<moduleName>` 可以包含 `module:` 前缀。在以前的版本中，必须忽略此前缀。

概述

使用 `@exports` 标签描述除由 JavaScript 模块的 `exports` 或 `module.exports` 属性外导出的任何内容。

实例

在模块中，当使用特定的 `exports` 模块，`@exports` 标签是不需要。JSDoc 会自动识别出该对象的导出成员。同样，JSDoc 会自动识别中的 Node.js 模块特定的 `module.exports` 属性。

CommonJS 模块:

```
/**
 * A module that says hello!
 * @module hello/world
 */

/** Say hello. */
exports.sayHello = function() {
  return "Hello world";
};
```

Node.js 模块:

```
/**
 * A module that shouts hello!
 * @module hello/world
 */

/** SAY HELLO. */
module.exports = function() {
  return "HELLO WORLD";
};
```

AMD 模块导出一个字面量对象：

```
define(function() {
  /**
   * A module that creates greeters.
   * @module greeter
   */

  /**
   * @constructor
   * @param {string} subject - The subject to greet.
   */
  var exports = function(subject) {
    this.subject = subject || "world";
  };

  /** Say hello to the subject. */
  exports.prototype.sayHello = function() {
    return "Hello " + this.subject;
  };

  return exports;
});
```

如果模块导出使用的是除 `exports` 或 `module.exports` 之外的其他方法方式，使用 `@exports` 标签来说明哪些成员用于导出。

AMD 模块导出一个对象：

```
define(function() {
  /**
   * A module that says hello!
   * @exports hello/world
   */
  var ns = {};

  /** Say hello. */
  ns.sayHello = function() {
    return "Hello world";
  };
```

```
  return ns;
});
```

相关链接

- @module
- CommonJS Modules
- AMD Modules

@external

目录

- 别名
- 语法
- 概述
- 实例

别名

```
@host
```

语法

```
@external <nameOfExternal>
```

概述

`@external` 标记标识在当前包外部定义的类、命名空间或模块。通过使用此标记，可以将包的扩展记录到外部符号，也可以向包的用户提供有关外部符号的信息。还可以在任何其他 JSDoc 标记中引用外部符号的 namepath。

外部符号的 namepath 始终使用前缀 `external:`（例如，`{@link external:Foo}` 或 `@augments external:Foo`）。但是，可以从 `@external` 标记中省略此前缀。

注意：只能在你的项目之外定义的最高级别的标识上添加 `@external` 标签，请参见“描述一个嵌套的外部标识”的例子。

实例

以下示例显示如何将内置字符串对象作为外部对象以及新的实例方法 `external:String#rot13` 进行文档记录。

记录添加到内置类的方法:

```
/**
 * The built in string object.
 * @external String
 * @see {@link https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String|String
 */

/**
 * Create a ROT13-encoded version of the string. Added by the `foo` package.
 * @function external:String#rot13
 * @example
 * var greeting = new String('hello world');
 * console.log( greeting.rot13() ); // uryyb jbeyq
 */
```

下面的例子中描述一个新 `starfairy` 功能如何添加到外部的 `jQuery.fn` 命名空间。

描述的外部的命名空间：

```
/**
 * The jQuery plugin namespace.
 * @external "jQuery.fn"
 * @see {@link http://learn.jquery.com/plugins/|jQuery Plugins}
 */

/**
 * A jQuery plugin to make stars fly around your home page.
 * @function external:"jQuery.fn".starfairy
 */
```

在下面的例子中，`EncryptedRequest` 类被描述为内置的 `XMLHttpRequest` 类的子类。

```
/**
 * The built-in class for sending HTTP requests.
 * @external XMLHttpRequest
 * @see https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
 */

/**
 * Extends the built-in `XMLHttpRequest` class to send data encoded with a secret key.
 * @class EncodedRequest
```

```
 * @extends external:XMLHttpRequest
 */
```

只能将 `@external` 标签添加到项目定义的最外最顶层。在下面的例子中，描述的是外部的 `security.TLS` 类。其结果是， `@external` 标签是用来描述外部的 `external:security` 命名空间，而不是外部类 `external:security.TLS` 。

记录一个嵌套的外部标识：

```
/**
 * External namespace for security-related classes.
 * @external security
 * @see http://example.org/docs/security
 */

/**
 * External class that provides Transport Layer Security (TLS) encryption.
 * @class TLS
 * @memberof external:security
 */
```

@file

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@fileoverview
@overview
```

语法

```
@file <descriptionOfFile>
```

概述

`@file` 标签提供文件的说明。在文件开头的 JSDoc 注释部分使用该标签。

实例

文件描述：

```
/**
 * @file Manages the configuration settings for the widget.
 * @author Rowina Sanela
 */
```

相关链接

- @author
- @version

@fires

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@emits
```

语法

```
@fires <className>#[event:]<eventName>
```

概述

`@fires` 标记表示方法在调用时可以触发指定类型的事件。使用 @event标记来记录事件的内容。

实例

方法将触发 "drain" 事件：

```
/**
 * Drink the milkshake.
 *
 * @fires Milkshake#drain
 */
Milkshake.prototype.drink = function() {
  // ...
};
```

相关链接

- @event
- @listens

@function

目录

- 别名
- 语法
- 概述
- 实例

别名

```
@func
@method
```

语法

```
@function [<functionName>]
```

概述

将一个对象标记为一个函数，即使在解析器看来它可能不是一个函数。它将 doclet 的 @kind 设置为 `'function'`。

实例

使用 `@function` 标记为一个函数：

```
/** @function */
var paginate = paginateFactory(pages);
```

如果没有 `@function` 标记，`paginate` 对象将被记录为泛型对象（一个 @member），因为从检查代码行中无法判断 `paginate` 在运行时将保存哪种类型的值。

使用带函数名的 `@function`：

```
/** @function myFunction */

// the above is the same as:
/** @function
 * @name myFunction
 */
```

相关链接

- @kind
- @member
- @name

@generator

目录

- 语法
- 概述
- 实例

语法

```
@generator
```

概述

`@generator` 标记表示函数是一个生成器函数，这意味着它是使用语法函数 `*foo () {}` 声明的。JSDoc 3.5.0 及更高版本中提供了此标记。

一般来说，不需要使用此标记，因为 JSDoc 会自动检测生成器函数并在生成的文档中标识它们。但是，如果您正在为代码中没有出现的生成器函数编写虚拟注释，则可以使用此标记告诉 JSDoc 该函数是生成器函数。

实例

以下示例显示使用 `@generator` 标记的虚拟注释。

带有 `@generator` 标记的虚拟注释:

```
/**
 * Generate numbers in the Fibonacci sequence.
 *
 * @generator
 * @function fibonacci
 * @yields {number} The next number in the Fibonacci sequence.
 */
```

@global

目录

-语法 -概述 -实例 -相关链接

语法

```
@global
```

概述

`@global` 标记指定一个符号应作为全局符号出现在文档中。JSDoc 忽略源文件中符号的实际作用域。此标记对于在本地定义然后指定给全局符号的符号特别有用。

实例

使用 `@global` 标签来指定一个标识应记录为全局。

将内部变量记录为全局变量:

```
(function() {
  /** @global */
  var foo = 'hello foo';

  this.foo = foo;
}).apply(window);
```

相关链接

- @inner
- @instance
- @memberof
- @static

@hideconstructor

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@hideconstructor
```

概述

`@hideconstructor` 标记告诉 JSDoc 生成的文档不应该显示类的构造函数。JSDoc 3.5.0 及更高版本中提供了此标记。

对于 ES2015 之前的类，请将此标记与 @class 或 @constructor标记结合使用。

对于 ES2015 类，请在构造函数的 JSDoc 注释中使用此标记。如果类没有显式构造函数，请在类的 JSDoc 注释中使用此标记。

实例

ES2015 之前的类带有 `@hideconstructor`：

```
/**
 * @classdesc Toaster singleton.
 * @class
 * @hideconstructor
 */
var Toaster = (function() {
  var instance = null;

  function Toaster() {}

  /**
   * Toast an item.
   *
   * @alias toast
   * @memberof Toaster
   * @instance
   * @param {BreadyThing} item - The item to toast.
   * @return {Toast} A toasted bready thing.
   */
  Toaster.prototype.toast = function(item) {};

  return {
    /**
     * Get the Toaster instance.
     *
     * @alias Toaster.getInstance
     * @returns {Toaster} The Toaster instance.
     */
    getInstance: function() {
      if (instance === null) {
        instance = new Toaster();
        delete instance.constructor;
      }

      return instance;
    }
  };
})();
```

ES2015 类带有 `@hideconstructor`：

```
/**
 * Waffle iron singleton.
```

```
 */
class WaffleIron {
  // Private field declarations
  #instance = null;

  /**
   * Create the waffle iron.
   *
   * @hideconstructor
   */
  constructor() {
    if (#instance) {
      return #instance;
    }

    /**
     * Cook a waffle.
     *
     * @param {Batter} batter - The waffle batter.
     * @return {Waffle} The cooked waffle.
     */
    this.cook = function(batter) {};

    this.#instance = this;
  }

  /**
   * Get the WaffleIron instance.
   *
   * @return {WaffleIron} The WaffleIron instance.
   */
  getInstance() {
    return new WaffleIron();
  }
}
```

相关链接

- @class

@ignore

目录

- 语法
- 概述
- 实例

语法

```
@ignore
```

概述

`@ignore` 标签表示在代码中的注释不应该出现在文档中，注释会被直接忽略。这个标签优先于所有其他标签。

对于大多数 JSDoc 模板来说，包括默认模板，`@ignore` 标签具有以下效果：

- 如果您和 @class 或 @module 标签结合使用 `@ignore` 标签，整个类或模块的 JSDoc 注释文档会被省略。
- 如果您和 @namespace 标签结合使用 `@ignore` 标签，必须将 `@ignore` 标签添加到任意子类和命名空间中。否则，会显示子类和命名空间的文档，但不完整。

实例

在下面的例子中，`Jacket` 和 `Jacket#color` 将不会出现在文档中。

在 Class 中使用 `@ignore` 标签：

```
/**
 * @class
 * @ignore
 */
function Jacket() {
  /** The jacket's color. */
  this.color = null;
}
```

在下面的例子中，`Clothes` 命名空间包含一个 `Jacket` 类。`@ignore` 标签必须添加到 `Clothes` 和 `Clothes.Jacket` 中。`Clothes`，`Clothes.Jacket` 和 `Clothes.Jacket#color` 将不会出现在文档。

带子类的命名空间：

```
/**
 * @namespace
 * @ignore
 */
var Clothes = {
  /**
   * @class
   * @ignore
   */
```

```
  Jacket: function() {
    /** The jacket's color. */
    this.color = null;
  }
};
```

@implements

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@implements {typeExpression}
```

概述

`@implements` 标签表示一个标识实现一个接口。

添加 `@implements` 标签到实现接口（例如，一个构造函数）的顶层标识。不需要将 `@implements` 标签添加到实现接口（例如，实现的实例方法）的每个成员上。

如果没有在实现的接口中描述这个标识，JSDoc 会自动使用该接口文档的标识。

实例

在下面的例子中，`TransparentColor` 类实现 `Color` 接口，并添加了 `TransparentColor#rgba` 方法。

使用 `@implements` 标签：

```
/**
 * Interface for classes that represent a color.
 *
 * @interface
 */
function Color() {}
```

```
/**
 * Get the color as an array of red, green, and blue values, represented as
 * decimal numbers between 0 and 1.
 *
 * @returns {Array<number>} An array containing the red, green, and blue values,
 * in that order.
 */
Color.prototype.rgb = function() {
  throw new Error("not implemented");
};

/**
 * Class representing a color with transparency information.
 *
 * @class
 * @implements {Color}
 */
function TransparentColor() {}

// inherits the documentation from `Color#rgb`
TransparentColor.prototype.rgb = function() {
  // ...
};

/**
 * Get the color as an array of red, green, blue, and alpha values, represented
 * as decimal numbers between 0 and 1.
 *
 * @returns {Array<number>} An array containing the red, green, blue, and alpha
 * values, in that order.
 */
TransparentColor.prototype.rgba = function() {
  // ...
};
```

相关链接

- @interface

@inheritdoc

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@inheritdoc
```

概述

`@inheritdoc` 标记表示应该从其父类继承其文档。JSDoc 注释中包含的任何其他标记都将被忽略。

提供此标记是为了与 Closure Compiler 兼容。默认情况下，如果不向符号添加 JSDoc 注释，标识符将从其父级继承文档。

`@inheritdoc` 标记的存在意味着 @override 标记的存在。

实例

下面的例子显示了一个类的描述如何从它的父类继承文档。

一个类继承自他的父类：

```
/**
 * @classdesc Abstract class representing a network connection.
 * @class
 */
function Connection() {}

/**
 * Open the connection.
 */
Connection.prototype.open = function() {
  // ...
};

/**
 * @classdesc Class representing a socket connection.
 * @class
 * @augments Connection
 */
function Socket() {}

/** @inheritdoc */
Socket.prototype.open = function() {
  // ...
};
```

省略 `Socket#open` 的 JSDoc 注 释，可以得到同样的结果。

不带 `@inheritdoc` 标记的继承文档:

```
/**
 * @classdesc Abstract class representing a network connection.
 * @class
 */
function Connection() {}

/**
 * Open the connection.
 */
Connection.prototype.open = function() {
  // ...
};

/**
 * @classdesc Class representing a socket connection.
 * @class
 * @augments Connection
 */
function Socket() {}

Socket.prototype.open = function() {
  // ...
};
```

相关链接

- @override

{@link}

目录

- 别名
- 语法
- 概述
- 链接格式化
- 实例
- 相关链接

别名

```
{@linkcode}
{@linkplain}
```

语法

```
{@link namepathOrURL}
[link text]{@link namepathOrURL}
{@link namepathOrURL|link text}
{@link namepathOrURL link text (after the first space)}
```

概述

`{@link}` 内联标记创建指向指定的 `namepath` 或 `URL` 的链接。使用 `{@link}` 标记时，还可以使用几种不同格式之一提供链接文本。如果不提供任何链接文本，JSDoc 将使用 `namepath` 或 `URL` 作为链接文本。

如果需要链接到教程，请使用 {@tutorial} 内联标记而不是 `{@link}` 标记。

链接格式化

默认情况下，`{@link}` 生成标准的 HTML 锚点标记。但是，你可能更愿意在某些环节用等宽字体呈现，或指定单个链接的格式。您可以使用 `{@link}` 标签的同义词来控制链接的格式：

- `{@linkcode}`：强制使用等宽字体链接文本。
- `{@linkplain}`：强制显示为正常的文本，没有等宽字体链接文本。

您还可以在 JSDoc 的配置文件中设置下列选项之一;详情参见配置 JSDoc：

- `templates.cleverLinks`: 如果设置为 `true`，链接 URL 使用普通的文本，并链接到代码中使用等宽字体。
- `templates.monospaceLinks`: 当设置为 `true` 时，除了用 `{@linkplain}` 标记创建的链接外，所有链接都使用一个单空间字体。

注意：尽管默认的 JSDoc 模板正确地呈现了所有这些标记，但是其他模板可能无法识别 `{@linkcode}` 和 `{@linkplain}` 标记。此外，其他模板可能会忽略链接呈现的配置选项。

实例

下面的例子显示了提供给 `{@link}` 标签链接文本的所有方式。

提供链接文本：

```
/**
 * See {@link MyClass} and [MyClass's foo property]{@link MyClass#foo}.
 * Also, check out {@link http://www.google.com|Google} and
 * {@link https://github.com GitHub}.
 */
function myFunction() {}
```

默认情况下，上面的例子中输出类似以下内容。

`@link` 标签输出：

```
See <a href="MyClass.html">MyClass</a> and <a href="MyClass.html#foo">MyClass's foo
property</a>. Also, check out <a href="http://www.google.com">Google</a> and
<a href="https://github.com">GitHub</a>.
```

如果配置属性 `templates.cleverLinks` 设置为 `true`，上面的例子会输出：

```
See <a href="MyClass.html"><code>MyClass</code></a> and <a href="MyClass.html#foo">
<code>MyClass's foo property</code></a>. Also, check out
<a href="http://www.google.com">Google</a> and <a href="https://github.com">GitHub</a>.
```

相关链接

- 使用配置文件配置 JSDOC
- 在 JSDoc 3中使用名称路径

@tutorial

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
{@tutorial tutorialID}
[link text]{@tutorial tutorialID}
{@tutorial tutorialID|link text}
{@tutorial tutorialID link text (after the first space)}
```

概述

`{@tutorial}` 内联标记创建指向您指定的教程标识符的链接。使用 `{@tutorial}` 标记时，还可以使用几种不同格式之一提供链接文本。如果不提供任何链接文本，JSDoc 将使用教程的标题作为链接文本。

如果需要链接到 `namepath` 或 `URL`，请使用 {@link}内联标记，而不是 `{@tutorial}` 标记。

实例

以下示例显示了为 `{@tutorial}` 标记提供链接文本的所有方法。

提供链接文本:

```
/**
 * See {@tutorial gettingstarted} and [Configuring the Dashboard]{@tutorial dashboard}.
 * For more information, see {@tutorial create|Creating a Widget} and
 * {@tutorial destroy Destroying a Widget}.
 */
function myFunction() {}
```

如果定义了所有这些教程，并且 `Getting Started` 教程的标题是 `“Getting Started”`，则上面的示例将生成类似于以下内容的输出：

```
See <a href="tutorial-gettingstarted.html">Getting Started</a> and
<a href="tutorial-dashboard.html">Configuring the Dashboard</a>.
For more information, see <a href="tutorial-create.html">Creating a Widget</a> and
<a href="tutorial-destroy.html">Destroying a Widget</a>.
```

相关链接

- @tutorial

@inner

目录

- 语法
- 概述
- 实例

- 相关链接

语法

```
@inner
```

概述

使用 `@inner` 标记会将符号标记为其父符号的内部成员。这意味着它可以被 `“Parent~Child”` 引用。

使用 `@inner` 将覆盖 doclet 的默认作用域（除非它在全局作用域中，在这种情况下它将保持全局作用域）。

实例

使用 `@inner` 使一个虚拟的 doclet 作为内部成员：

```
/** @namespace MyNamespace */
/**
 * myFunction is now MyNamespace~myFunction.
 * @function myFunction
 * @memberof MyNamespace
 * @inner
 */
```

注意，在上面的代码我们也可以使用 `"@function MyNamespace~myFunction"`，代替 @memberof 和 `@inner` 标签。

使用 `@inner`：

```
/** @namespace */
var MyNamespace = {
  /**
   * foo is now MyNamespace~foo rather than MyNamespace.foo.
   * @inner
   */
  foo: 1
};
```

在上面的例子中，我们使用 `@inner` 迫使一个命名空间的成员被描述作为内部成员（默认情况下，这是一个静态成员）。这意味着，`foo` 现在有了 `MyNamespace~foo` 新名字，而不是 `MyNamespace.foo`。

相关链接

- @global
- @instance
- @static

@instance

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@instance
```

概述

使用 `@instance` 标记会将符号标记为其父符号的实例成员。这意味着它可以被称为 `Parent#Child`。

使用 `@instance` 将覆盖 doclet 的默认作用域（除非它在全局作用域中，在这种情况下，它将保持全局作用域）。

实例

下面的例子是 `@function MyNamespace#myFunction` 的一个普通写法。

使用 `@instance` 使一个虚拟的 doclet 作为实例成员:

```
/** @namespace MyNamespace */
/**
 * myFunction is now MyNamespace#myFunction.
 * @function myFunction
 * @memberof MyNamespace
 * @instance
 */
```

更有用的是，可以使用 `@instance` 标记覆盖 JSDoc 推断的范围。例如，可以指示静态成员用作实例成员。

使用 `@instance` 标识实例成员:

```
/** @namespace */
var BaseObject = {
  /**
   * foo is now BaseObject#foo rather than BaseObject.foo.
   * @instance
   */
  foo: null
};

/** Generates BaseObject instances. */
function fooFactory(fooValue) {
  var props = { foo: fooValue };
  return Object.create(BaseObject, props);
}
```

相关链接

- @global
- @inner
- @static

@interface

目录

- 语法
- 概述
- 实例
- 相关链接

语法

用作 JSDoc 标签字典 (默认开启):

```
@interface [<name>]
```

用作 Closure Compiler 标签字典:

```
@interface
```

概述

`@interface` 标签使一个标识符作为其他标识符的一个实现接口。 例如，你的代码可能定义一个父类，它的方法和属性被去掉。您可以将 `@interface` 标签添加到父类，以指明子类必须实现父类的方法和属性。

作为接口，`@interface` 标记应该添加到顶层标识符（例如，一个构造函数）。你并不需要将 `@interface` 标签添加加到实现接口（例如，实现的实例方法）的每个成员上。

如果您使用的是 JSDoc 标记字典（默认启用），你还可以定义一个接口的虚拟注释，而不是为接口编写代码。见一个例子“定义一个接口的虚拟注释”。

实例

在下面的例子中，`Color` 函数表示其它类可以实现的接口。

使用 `@interface` 标签：

```
/**
 * Interface for classes that represent a color.
 *
 * @interface
 */
function Color() {}

/**
 * Get the color as an array of red, green, and blue values, represented as
 * decimal numbers between 0 and 1.
 *
 * @returns {Array<number>} An array containing the red, green, and blue values,
 * in that order.
 */
Color.prototype.rgb = function() {
  throw new Error('not implemented');
};
```

下面的例子使用虚拟注释，而不是代码，来定义 `Color` 接口。

虚拟注释来定义一个接口：

```
/**
 * Interface for classes that represent a color.
 *
 * @interface Color
```

```
 */

/**
 * Get the color as an array of red, green, and blue values, represented as
 * decimal numbers between 0 and 1.
 *
 * @function
 * @name Color#rgb
 * @returns {Array<number>} An array containing the red, green, and blue values,
 * in that order.
 */
```

相关链接

- @implements

@kind

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@kind <kindName>
```

`<kindName>` 取值为：

- `class`
- `constant`
- `event`
- `external`
- `file`
- `function`
- `member`
- `mixin`
- `module`

- `namespace`
- `typedef`

概述

`@kind` 标签是用来指明什么样的标识符被描述（例如，一类或模块）。标识符 `kind` 不同于标识符 `type`（例如，字符串或布尔）。

通常你不需要 `@kind` 标签，因为标识符的 `kind` 是由 doclet 的其他标记来确定。例如，使用 @class 标签自动意味着 `@kind class`，使用 @namespace 标签则意味着 `@kind namespace`。

实例

使用 `@kind`：

```
// The following examples produce the same result:

/**
 * A constant.
 * @kind constant
 */
const asdf = 1;

/**
 * A constant.
 * @constant
 */
const asdf = 1;
```

`kind` 标签可能引起冲突（例如，使用 @module，表示他的 `kind` 为 `module`，同时，又使用了 `@kind constant`，表示他的 `kind` 为 `constant`），在这种情况下最后的标签决定 `kind` 的值。

冲突的 `@kind` 语句：

```
/**
 * This will show up as a constant
 * @module myModule
 * @kind constant
 */

/**
 * This will show up as a module.
 * @kind constant
 * @module myModule
 */
```

相关链接

- @type

@lends

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@lends <namepath>
```

概述

`@lends` 标签允许将一个字面量对象的所有成员标记为某个标识符的成员，就像他们是给定名称的标识符成员。如果要将对象文字传递给从其成员创建命名类的函数，则可能需要执行此操作。

实例

在本例中，我们希望使用一个 `helper` 函数来创建一个名为 `Person` 的类，以及名为 `initialize` 和 `say` 的实例方法。这与一些流行的框架处理类创建的方式类似。

示例类:

```
// We want to document this as being a class
var Person = makeClass(
  // We want to document these as being methods
  {
    initialize: function(name) {
      this.name = name;
    },
    say: function(message) {
      return this.name + " says: " + message;
    }
  }
);
```

如果没有任何注释，JSDoc 将无法识别这段代码使用两个方法创建一个 `Person` 类。要记录这些方法，我们必须在 doc 注释中紧靠对象文本之前使用 `@lends` 标记。 `@lends` 标记告诉 JSDoc，该对象文本的所有成员名称都被“借用”给一个名为 `Person` 的变量。我们还必须为每种方法添加注释。

下面的例子让我们更接近我们想要的：

```
/** @class */
var Person = makeClass(
  /** @lends Person */
  {
    /**
     * Create a `Person` instance.
     * @param {string} name - The person's name.
     */
    initialize: function(name) {
      this.name = name;
    },
    /**
     * Say something.
     * @param {string} message - The message to say.
     * @returns {string} The complete message.
     */
    say: function(message) {
      return this.name + " says: " + message;
    }
  }
);
```

现在名为 `initialize` 和 `say` 的函数会被文档化，但它们被标记为 `Person` 类的静态方法。这可能是你期望的，但有种情况下我们想要 `initialize` 和 `say` 属于 `Person` 类的实例。所以，我们通过少做改动,使其成为原型的方法。

标记为实例方法:

```
/** @class */
var Person = makeClass(
  /** @lends Person.prototype */
  {
    /**
     * Create a `Person` instance.
     * @param {string} name - The person's name.
     */
    initialize: function(name) {
      this.name = name;
    },
    /**
     * Say something.
     * @param {string} message - The message to say.
```

```
     * @returns {string} The complete message.
     */
    say: function(message) {
      return this.name + " says: " + message;
    }
  }
);
```

最后一步：我们的类框架使用借出的 `initialize` 函数来构造 `Person` 实例，但是 `Person` 实例没有自己的 `initialize` 方法。解决方案是将 `@constructs` 标记添加到借出的函数中。记住也要删除 `@class` 标记，否则会记录两个类。

与构造器一起记录:

```
var Person = makeClass(
  /** @lends Person.prototype */
  {
    /**
     * Create a `Person` instance.
     * @constructs
     * @param {string} name - The person's name.
     */
    initialize: function(name) {
      this.name = name;
    },
    /**
     * Say something.
     * @param {string} message - The message to say.
     * @returns {string} The complete message.
     */
    say: function(message) {
      return this.name + " says: " + message;
    }
  }
);
```

相关链接

- @borrows
- @constructs

@license

目录

- 语法

- 概述
- 实例

语法

```
@license <identifier>
```

概述

`@license` 标记标识应用于代码任何部分的软件许可证。

您可以使用任何文本来标识正在使用的许可证。如果代码使用标准的开源许可证，请考虑使用the Software Package Data Exchange (SPDX) License List中的相应标识符。

一些 JavaScript 处理工具，比如 Google 的 Closure 编译器，会自动保存任何包含 `@license` 标记的 JSDoc 注释。如果您正在使用其中一个工具，您可能希望添加一个独立的 JSDoc 注释，该注释包括 `@license` 标记以及许可证的整个文本，以便许可证文本将包含在生成的 JavaScript 文件中。

实例

在Apache 2.0 许可下分发的模块：

```
/**
 * Utility functions for the foo package.
 * @module foo/util
 * @license Apache-2.0
 */
```

一个独立的 JSDoc 注释块，包含完整的MIT许可:

```
/**
 * @license
 * Copyright (c) 2015 Example Corporation Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
```

```
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
```

@listens

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@listens <eventName>
```

概述

`@listens` 标记指示符号侦听指定的事件。使用 @event标记来记录事件的内容。

实例

下面的示例演示了如何记录名为 `module:hurler~event:snowball` 的事件，还有一个方法命名为 `module:playground/monitor.reportThrowage` 来监听事件。

描述一个事件和它的监听器:

```
define("hurler", [], function() {
  /**
   * Event reporting that a snowball has been hurled.
   *
   * @event module:hurler~snowball
   * @property {number} velocity - The snowball's velocity, in meters per second.
   */

  /**
```

```
   * Snowball-hurling module.
   *
   * @module hurler
   */
  var exports = {
    /**
     * Attack an innocent (or guilty) person with a snowball.
     *
     * @method
     * @fires module:hurler~snowball
     */
    attack: function() {
      this.emit("snowball", { velocity: 10 });
    }
  };

  return exports;
});

define("playground/monitor", [], function() {
  /**
   * Keeps an eye out for snowball-throwers.
   *
   * @module playground/monitor
   */
  var exports = {
    /**
     * Report the throwing of a snowball.
     *
     * @method
     * @param {module:hurler~event:snowball} e - A snowball event.
     * @listens module:hurler~event:snowball
     */
    reportThrowage: function(e) {
      this.log("snowball thrown: velocity " + e.velocity);
    }
  };

  return exports;
});
```

相关链接

- @event
- @fires

@member

目录

- 别名
- 语法
- 概述
- 实例

别名

```
@var
```

语法

```
@member [<type>] [<name>]
```

概述

`@member` 标记标识没有特殊类型的任意成员，例如“类”、“函数”或“常量”。成员可以选择具有类型和名称。

实例

`Data#point` 上使用 `@member` ：

```
/** @class */
function Data() {
  /** @member {Object} */
  this.point = {};
}
```

下面是使用 `@var` 的一个例子， `@member` 的别名，来描述一个（虚拟）变量'foo'。

```
/**
 * A variable in the global namespace called 'foo'.
 * @var {number} foo
 */
```

上面的例子等价于：

```
/**
 * A variable in the global namespace called 'foo'.
 * @type {number}
```

```
 */
var foo;
```

@memberof

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@memberof <parentNamepath>
@memberof! <parentNamepath>
```

概述

`@memberof` 标签标明成员隶属于哪一个父级标识符。

默认情况下，`@memberof` 标签标注的标识符是静态成员。对于内部成员和实例成员，你可以使用对应名称路径的符号，或明确标注 @inner 或 @instance 标签。

“强制的” `@memberof` 标签，`@memberof!` 强制对象被记录为属于特定的父级标识符，即使它有不同的父级标识符。

实例

在下面的示例中，`hammer` 函数通常被记录为全局函数。这是因为，事实上，它是一个全局函数，但它也是 `Tools` 命名空间的一个成员，这就是您想要记录它的方式。解决方案是添加 `@memberof` 标记。

使用 `@memberof`：

```
/** @namespace */
var Tools = {};

/** @memberof Tools */
var hammer = function() {};
```

```
Tools.hammer = hammer;
```

对于类的实例成员，可以使用语法：`"@memberof ClassName.prototype"` 或者 `"@memberof ClassName#"`。另外也可以组合使用 `"@memberof ClassName"` 和 `@instance` 达到同样的效果。

在类原型上使用 `@memberof` ：

```
/** @class Observable */
create("Observable", {
  /**
   * This will be a static member, Observable.cache.
   * @memberof Observable
   */
  cache: [],

  /**
   * This will be an instance member, Observable#publish.
   * @memberof Observable.prototype
   */
  publish: function(msg) {},

  /**
   * This will also be an instance member, Observable#save.
   * @memberof Observable#
   */
  save: function() {},

  /**
   * This will also be an instance member, Observable#end.
   * @memberof Observable
   * @instance
   */
  end: function() {}
});
```

下面的示例使用强制 `@memberof` 标签， `@memberof!` ，来描述对象(`Data#point`)的属性，它是一个类（Data）的实例成员。

当您使用 @property 标签记录一个属性的时候，则无法使用其 longname 连接到这个属性。我们可以使用 @alias 和 `@memberof!` 来强制属性为可连接，告诉 JSDoc `Data#point.y` 应记录为 `Data#` 的成员 `point.y` 而不是 `Data#` 的 `point` 中的一员 `y` 。

为对象属性使用 `@memberof!` ：

```
/** @class */
function Data() {
  /**
   * @type {object}
   * @property {number} y This will show up as a property of `Data#point`,
```

```
    * but you cannot link to the property as {@link Data#point.y}.
    */
   this.point = {
     /**
      * The @alias and @memberof! tags force JSDoc to document the
      * property as `point.x` (rather than `x`) and to be a member of
      * `Data#`. You can link to the property as {@link Data#point.x}.
      * @alias point.x
      * @memberof! Data#
      */
     x: 0,
     y: 1
   };
 }
```

相关链接

- @name

@mixes

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@mixes <OtherObjectPath>
```

概述

`@mixes` 标签指示当前对象混入了 `OtherObjectPath` 对象的所有成员,被混入的对象就是一个@mixin。

实例

用 `@mixin` 标签描述一个混入：

```
/**
 * This provides methods used for event handling. It's not meant to
 * be used directly.
 *
 * @mixin
 */
var Eventful = {
  /**
   * Register a handler function to be called whenever this event is fired.
   * @param {string} eventName - Name of the event.
   * @param {function(Object)} handler - The handler to call.
   */
  on: function(eventName, handler) {
    // code...
  },

  /**
   * Fire an event, causing all handlers for that event name to run.
   * @param {string} eventName - Name of the event.
   * @param {Object} eventData - The data provided to each handler.
   */
  fire: function(eventName, eventData) {
    // code...
  }
};
```

现在，我们添加一个 `FormButton` 类，并且调用 `mix` 函数，将 `Eventful` 的所有功能混入到 `FormButton`，这样 `FormButton` 也可以触发事件和监听了。我们使用 `@mixes` 标签，以表明 `FormButton` 混入了 `Eventful` 的功能。

使用 `@mixes` 标签：

```
/**
 * @constructor FormButton
 * @mixes Eventful
 */
var FormButton = function() {
  // code...
};
FormButton.prototype.press = function() {
  this.fire('press', {});
}
mix(Eventful).into(FormButton.prototype);
```

相关链接

- @borrows
- @class

- @mixin

@mixin

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@mixin [<MixinName>]
```

概述

`@mixin` 标签提供旨在被添加到其他对象的功能。然后，可以将 @mixes 标签添加到使用了该 mixin（混入）的对象上。

实例

使用 `@mixin`：

```
/**
 * This provides methods used for event handling. It's not meant to
 * be used directly.
 *
 * @mixin
 */
var Eventful = {
  /**
   * Register a handler function to be called whenever this event is fired.
   * @param {string} eventName - Name of the event.
   * @param {function(Object)} handler - The handler to call.
   */
  on: function(eventName, handler) {
    // code...
  },

  /**
   * Fire an event, causing all handlers for that event name to run.
   * @param {string} eventName - Name of the event.
```

```
     * @param {Object} eventData - The data provided to each handler.
     */
    fire: function(eventName, eventData) {
      // code...
    }
};
```

相关链接

- @borrows
- @class
- @mixes

@module

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@module [[{<type>}] <moduleName>]
```

在 JSDoc3.3.0 或更高版本中，`<moduleName>` 可能包括 `module:` 前缀。在以前的版本中，必须忽略此前缀。

注意：如果你提供了一个type，那 必须同时提供模块名称 `<moduleName>`。

概述

`@module` 可以将当前文件标注为一个模块，默认情况下文件内的所有标识符都隶属于此模块，除非文档另有说明。

使用 `module:moduleName` 链接到模块（例如，@link 或 @see 标记内）。例如，`@module foo/bar` 可以使用 `{@link module:foo/bar}` 链接到。

如果未提供模块名称，则该名称将从模块的路径和文件名派生。例如，假设我有一个文件 `test.js`，位于 `src` 目录中，它包含 `/** @module */`。下面是运行 JSDoc 的一些场景以及

test.js 的结果模块名：

如果没有提供导出模块的名称：

```
# from src/
jsdoc ./test.js   # module name 'test'

# from src's parent directory:
jsdoc src/test.js # module name 'src/test'
jsdoc -r src/     # module name 'test'
```

实例

下面的示例演示了在一个模块中用于标识的 `namepaths`。第一个标识符是模块私有的，或“内部”变量 - 它只能在模块内访问。第二个标识符是由模块导出一个静态函数。

使用基础的 `@module`：

```
/** @module myModule */

/** will be module:myModule~foo */
var foo = 1;

/** will be module:myModule.bar */
var bar = function() {};
```

当一个导出的标识符被定义为 `module.exports`，`exports`，或 `this` 中的成员，JSDoc 会推断该标识符是模块的静态成员。

在下面的例子中，`Book` 类被描述为一个静态成员，`module:bookshelf.Book`，带有一个实例成员，`module:bookshelf.Book#title`。

定义导出的标识符为 `this` 的成员：

```
/** @module bookshelf */
/** @class */
this.Book = function (title) {
  /** The title. */
  this.title = title;
};
```

在下面的例子中，两个函数有 `namepaths`（名称路径）`module:color/mixer.blend` 和 `module:color/mixer.darken`。

定义导出的标识符为 `module.exports` 或 `exports` 的成员:

```
/** @module color/mixer */
module.exports = {
  /** Blend two colours together. */
  blend: function (color1, color2) {}
};
/** Darkens a color. */
exports.darken = function (color, shade) {};
```

更多例子查看描述 JavaScript 模块。

相关链接

- @exports
- CommonJS Modules
- AMD Modules

@name

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@name <namePath>
```

概述

`@name` 标签强制 JSDoc 使用这个给定的名称，而忽略实际代码里的名称。这个标签最好用于"虚拟注释"，而不是在代码中随时可见的标签，如在运行时期间产生的方法。

当您使用 `@name` 标签，必须提供额外的标签，来告诉 JSDoc 什么样的标识符将被文档化;该标识符是否是另一个标识符的成员，等等。如果不提供这些信息，标识符将不会被正确文档化。

警告：通过使用 `@name` 标签告诉 JSDoc 忽略实际代码，隔离文档注释。在许多情况下，最好是使用 @alias 标签代替，这个标签只是改变了标识符的名称，但是保留了标识符的其他信息。

实例

下面的例子演示了如何使用 `@name` 标签描述一个函数，JSDoc 通常不会识别。

使用 `@name` 标签：

```
/**
 * @name highlightSearchTerm
 * @function
 * @global
 * @param {string} term - The search term to highlight.
 */
eval("window.highlightSearchTerm = function(term) {};")
```

相关链接

- 在 JSDoc 3中使用名称路径
- @alias

@namespace

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@namespace [[{<type>}] <SomeName>]
```

概述

`@namespace` 标记表示对象为其成员创建命名空间。您还可以编写一个虚拟 JSDoc 注释来定义代码使用的命名空间。

如果一个命名空间是由除对象字面量以为的标识符定义的，您可以包括一个 `type` 的表达式，跟在 `@namespace` 标签后面。如果 `@namespace` 标签包括一个 `type`，那么它也必须包含一个名称。

您可能需要描述一个命名空间，其名称中包含特殊字符，如"#" 或 "!"。在这些情况下，当你的描述或链接到这个命名空间时，你必须将命名空间中特殊符号部分使用双引号括起来。详情参见下面的例子。

实例

对象上使用 `@namespace` 标签：

```
/**
 * My namespace.
 * @namespace
 */
var MyNamespace = {
  /** documented as MyNamespace.foo */
  foo: function() {},
  /** documented as MyNamespace.bar */
  bar: 1
};
```

为虚拟注释加上 `@namespace` 标签：

```
/**
 * A namespace.
 * @namespace MyNamespace
 */

/**
 * A function in MyNamespace (MyNamespace.myFunction).
 * @function myFunction
 * @memberof MyNamespace
 */
```

如果 `@namespace` 的名称包括特殊字符，你必须将命名空间中特殊符号部分使用双引号括起来。如果名称已经包含一个或多个双引号，那么使用反斜线（ `\` ）转义双引号。

在特殊的成员名称上使用 `@namespace` 标签：

```
/** @namespace window */

/**
 * Shorthand for the alert function.
 * Refer to it as {@link window."!"} (note the double quotes).
 */
window["!"] = function(msg) { alert(msg); };
```

相关链接

- @module

@override

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@override
```

概述

`@override` 标签指明一个标识符覆盖其父类同名的标识符。

这个标签为 Closure Compiler提供了兼容性。默认情况下，JSDoc 自动识别，覆盖其父类同名的标识符。

如果您的 JSDoc 注释块包含 @inheritdoc 标签，就不需要在包含 `@Override` 标签了。`@inheritdoc` 标签的存在就意味着 `@override` 的存在。

实例

下面的例子说明一个方法如何重写父类的方法。

重写父类方法：

```
/**
 * @classdesc Abstract class representing a network connection.
 * @class
 */
function Connection() {}

/**
 * Open the connection.
 */
Connection.prototype.open = function() {
  // ...
```

```
};

/**
 * @classdesc Class representing a socket connection.
 * @class
 * @augments Connection
 */
function Socket() {}

/**
 * Open the socket.
 * @override
 */
Socket.prototype.open = function() {
  // ...
};
```

相关链接

- @inheritdoc

@package

目录

- 语法
- 概述
- 实例
- 相关链接

语法

使用 JSDoc 标记字典（默认情况下启用）：

```
@package
```

使用 Closure Compiler 编译器标记字典:

```
@package [{typeExpression}]
```

概述

`@package` 标记将符号内容标记为 package private。通常，此标记指示符号仅可用于与此符号的源文件位于同一目录中的代码。JSDoc 3.5.0 及更高版本中提供了此标记。

默认情况下，文档中将显示用 `@package` 标记标记的符号。在 JSDoc 3.3.0 和更高版本中，可以使用 `-a/--access` 命令行选项来更改此行为。

`@package` 标记等同于 `@access package`。

实例

在下面的示例中，实例成员 `Thingy#_bar` 显示在生成的文档中，但带有一个注释，指示它是包专用的

使用 `@package` 标记:

```
/** @constructor */
function Thingy() {
  /** @package */
  this._bar = 1;
}
```

相关链接

- @access
- @global
- @instance
- @private
- @protected
- @public
- @static

@param

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@arg
@argument
```

语法

```
@param [<type>] <name> [<description>]
```

概述

`@param` 标记提供函数参数的名称、类型和描述。

`@param` 标记要求您指定要记录的参数的名称。您还可以包括括在大括号中的参数类型和参数说明。

参数类型可以是内置的 JavaScript 类型，例如字符串或对象，也可以是代码中另一个符号的 JSDoc namepath。如果您已经在 namepath 中为该符号编写了文档，JSDoc 将自动链接到该符号的文档。您还可以使用类型表达式来指示参数不可为空或可以接受任何类型；有关详细信息，请参阅 @type 标记文档。

如果您提供了一个描述，那么可以通过在描述之前插入一个连字符，使 JSDoc 注释更具可读性。务必在连字符前后加一个空格。

实例

名称, 类型, 和说明

下面的示例演示如何在 `@param` 标签中包含名称，类型，和说明。

只注释变量名称:

```
/**
 * @param somebody
 */
function sayHello(somebody) {
  alert('Hello ' + somebody);
}
```

注释变量名和变量类型:

```
/**
 * @param {string} somebody
 */
function sayHello(somebody) {
  alert('Hello ' + somebody);
}
```

注释变量名、变量类型和变量说明:

```
/**
 * @param {string} somebody Somebody's name.
 */
function sayHello(somebody) {
  alert('Hello ' + somebody);
}
```

可以在变量说明前加个连字符，使之更加容易阅读:

```
/**
 * @param {string} somebody - Somebody's name.
 */
function sayHello(somebody) {
  alert('Hello ' + somebody);
}
```

具有属性的参数

如果参数需要具有特定属性，则可以通过提供额外的 `@param` 标记来记录该属性。例如，如果希望 `employee` 参数具有 `name` 和 `department` 属性，则可以按如下方式记录该参数.

记录参数的属性:

```
/**
 * Assign the project to an employee.
 * @param {Object} employee - The employee who is responsible for the project.
 * @param {string} employee.name - The name of the employee.
 * @param {string} employee.department - The employee's department.
 */
Project.prototype.assign = function(employee) {
  // ...
};
```

如果一个参数在没有显式名称的情况下被解构，您可以给这个对象一个合适的名称并记录它的属性。

记录解构参数:

```
/**
 * Assign the project to an employee.
 * @param {Object} employee - The employee who is responsible for the project.
 * @param {string} employee.name - The name of the employee.
 * @param {string} employee.department - The employee's department.
 */
Project.prototype.assign = function({ name, department }) {
  // ...
};
```

还可以将此语法与 JSDoc 的数组参数语法相结合。例如，如果可以将多个 `employees` 分配给一个 `project`。

记录数组中值的属性：

```
/**
 * Assign the project to a list of employees.
 * @param {Object[]} employees - The employees who are responsible for the project.
 * @param {string} employees[].name - The name of an employee.
 * @param {string} employees[].department - The employee's department.
 */
Project.prototype.assign = function(employees) {
  // ...
};
```

可选参数和默认值

下面的例子说明如何描述一个参数是可选的，并且具有默认值。

一个可选参数（使用 JSDoc 语法）：

```
/**
 * @param {string} [somebody] - Somebody's name.
 */
function sayHello(somebody) {
  if (!somebody) {
    somebody = "John Doe";
  }
  alert("Hello " + somebody);
}
```

一个可选参数（ 使用Google Closure Compiler 语法）：

```
/**
 * @param {string=} somebody - Somebody's name.
 */
function sayHello(somebody) {
  if (!somebody) {
```

```
    somebody = 'John Doe';
  }
  alert('Hello ' + somebody);
}
```

一个可选参数和默认值：

```
/**
 * @param {string} [somebody=John Doe] - Somebody's name.
 */
function sayHello(somebody) {
  if (!somebody) {
    somebody = 'John Doe';
  }
  alert('Hello ' + somebody);
}
```

多种类型和可重复参数

下面的例子演示了如何使用类型的表达式来表示一个参数可以接受多种类型（或任何类型），还有一个参数可以被多次使用。有关 JSDoc 支持的类型表达式细节请参阅 @type 标签文档。

允许一个类型或另一个类型：

```
/**
 * @param {(string|string[])} [somebody=John Doe] - Somebody's name, or an array of names.
 */
function sayHello(somebody) {
  if (!somebody) {
    somebody = 'John Doe';
  } else if (Array.isArray(somebody)) {
    somebody = somebody.join(', ');
  }
  alert('Hello ' + somebody);
}
```

允许任何类型：

```
/**
 * @param {*} somebody - Whatever you want.
 */
function sayHello(somebody) {
  console.log('Hello ' + JSON.stringify(somebody));
}
```

可重复使用的参数：

```
/**
 * Returns the sum of all numbers passed to the function.
 * @param {...number} num - A positive or negative number.
 */
function sum(num) {
  var i = 0, n = arguments.length, t = 0;
  for (; i < n; i++) {
    t += arguments[i];
  }
  return t;
}
```

回调函数

如果参数接受一个回调函数，您可以使用 @callback 标签来定义一个回调类型，然后回调类型包含到 `@param` 标签中。

```
/**
 * This callback type is called `requestCallback` and is displayed as a global symbol.
 *
 * @callback requestCallback
 * @param {number} responseCode
 * @param {string} responseMessage
 */

/**
 * Does something asynchronously and executes the callback on completion.
 * @param {requestCallback} cb - The callback that handles the response.
 */
function doSomethingAsynchronously(cb) {
  // code
};
```

相关链接

- @callback
- @returns
- @type
- @typedef

@private

目录

- 语法
- 概述
- 实例
- 相关链接

语法

使用 JSDoc 标记字典（默认情况下启用）：

```
@private
```

使用 Closure Compiler 编译器标记字典:

```
@private [{typeExpression}]
```

概述

`@private` 标记将符号标记为 `private`，或不用于一般用途。除非使用 `-p/--private` 命令行选项运行 JSDoc，否则生成的输出中不会显示私有成员。在 JSDoc 3.3.0 和更高版本中，还可以使用 `-a/--access` 命令行选项来更改此行为。

`@private` 标记不被子成员继承。例如，如果 `@private` 标记添加到命名空间，命名空间的成员仍然会输出到生成的文档中;因为命名空间是私有的，成员的 `namepath` 不包含在命名空间中。

`@private` 标记等同于 `@access private`。

实例

在下面的例子中，`Documents` 和 `Documents.Newspaper` 会被输出到生成的文档中，但是 `Documents.Diary` 不会。

```
/** @namespace */
var Documents = {
  /**
   * An ordinary newspaper.
   */
  Newspaper: 1,
  /**
   * My diary.
   * @private
   */
  Diary: 2
};
```

相关链接

- @access
- @global
- @instance
- @package
- @protected
- @public
- @static

@property

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@prop
```

语法

```
@property <type> <name> [<description>]
```

概述

`@property` 标记是一种方便地记录类、命名空间或其他对象的静态属性列表的方法。

通常，JSDoc 模板将创建一个完整的新页面，以显示有关嵌套命名空间层次结构的每个级别的信息。有时，您真正想要的是在同一页上列出所有属性，包括嵌套属性。

请注意，属性标记必须在文档注释中使用，例如它们是命名空间或类的属性。此标记用于静态属性的简单集合，不允许为每个属性提供 `@examples` 或类似的复杂信息，仅提供类型、名称和

描述。

实例

在这个例子中，我们有一个名为 `config` 的命名空间。我们想要所有有关默认属性及嵌套值的信息，输出到与 `config` 同一个页面上。

描述命名空间的默认属性及嵌套属性：

```
/**
 * @namespace
 * @property {object}  defaults               - The default values for parties.
 * @property {number}  defaults.players       - The default number of players.
 * @property {string}  defaults.level         - The default level for the party.
 * @property {object}  defaults.treasure      - The default treasure.
 * @property {number}  defaults.treasure.gold - How much gold the party starts with.
 */
var config = {
  defaults: {
    players: 1,
    level:   'beginner',
    treasure: {
      gold: 0
    }
  }
};
```

下面的示例演示如何指示属性是可选的。

```
/**
 * User type definition
 * @typedef {Object} User
 * @property {string} email
 * @property {string} [nickName]
 */
```

相关链接

- @enum
- @member

@protected

目录

- 语法
- 概述
- 实例
- 相关链接

语法

使用 JSDoc 标记字典（默认情况下启用）：

```
@protected
```

使用 Closure Compiler 编译器标记字典:

```
@protected [{typeExpression}]
```

概述

`@protected` 标记将符号标记为受保护。通常，此标记表示符号仅在当前模块中可用或应仅使用。

默认情况下，文档中将显示用 `@protected` 标记标记的符号。在 JSDoc 3.3.0 和更高版本中，可以使用 `-a/--access` 命令行选项来更改此行为。

`@protected` 标记等同于 `@access protected`。

实例

在下面的例子中，该实例成员 `Thingy#_bar` 会被导出到生成的文档中，但使用注释说明它是被保护的。

使用 `@protected` 标签：

```
/** @constructor */
function Thingy() {
  /** @protected */
  this._bar = 1;
}
```

相关链接

- @access

- @global
- @instance
- @package
- @private
- @public
- @static

@public

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@public
```

概述

`@public` 标签标记标识符为公开的。

默认情况下，JSDoc 把所有标识符当做公开的，因此使用这个标记一般不会影响生成的文档。然而，你可能更愿意明确地使用 `@public` 标签，这样可以更加清晰的标明你要公开的标识符。

在 JSDoc3 中，`@public` 标签不影响标识符的作用域。使用 @instance, @static, 和 @global 标签会改变标识符的作用域。

实例

使用 `@public` 标签：

```
/**
 * The Thingy class is available to all.
 * @public
 * @class
 */
function Thingy() {
```

```
  /**
   * The Thingy~foo member. Note that 'foo' is still an inner member
   * of 'Thingy', in spite of the @public tag.
   * @public
   */
  var foo = 0;
}
```

相关链接

- @access
- @global
- @instance
- @package
- @private
- @protected
- @static

@readonly

目录

- 语法
- 概述
- 实例

语法

```
@readonly
```

概述

标记一个标识符为只读。JSDoc 不会检查某个代码是否真是只读的，只要标上 `@readonly` ，在文档中就体现为只读的。

实例

使用 `@readonly` 标签：

```
/**
 * A constant.
 * @readonly
 * @const {number}
 */
const FOO = 1;
```

带有 `@readonly` 标签的 `getter`：

```
/**
 * Options for ordering a delicious slice of pie.
 * @namespace
 */
var pieOptions = {
  /**
   * Plain.
   */
  plain: "pie",
  /**
   * A la mode.
   * @readonly
   */
  get aLaMode() {
    return this.plain + " with ice cream";
  }
};
```

@requires

目录

- 语法
- 概述
- 实例

语法

```
@requires <someModuleName>
```

概述

`@requires` 标签可以记录一个模块需要的依赖项。一个 JSDoc 注释块可以有多个 `@require` 标签。模块名可以被指定为 "moduleName" 或者 "module:moduleName";这两种形式将被解

析为模块。

JSDoc 不会尝试处理被包含的模块。如果您希望该模块包含到文档中，您必须将模块包含到 JavaScript 文件列表进行处理。

实例

使用 `@requires` 标签:

```
/**
 * This class requires the modules {@link module:xyzcorp/helper} and
 * {@link module:xyzcorp/helper.ShinyWidget#polish}.
 * @class
 * @requires module:xyzcorp/helper
 * @requires xyzcorp/helper.ShinyWidget#polish
 */
function Widgetizer() {}
```

@returns

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@return
```

语法

```
@return [{type}] [description]
```

概述

`@returns` 标记记录函数返回的值。

如果要记录生成器函数，请使用 @yiels 标记而不是此标记。

实例

返回值的类型:

```
/**
 * Returns the sum of a and b
 * @param {number} a
 * @param {number} b
 * @returns {number}
 */
function sum(a, b) {
  return a + b;
}
```

返回值的类型和描述:

```
/**
 * Returns the sum of a and b
 * @param {number} a
 * @param {number} b
 * @returns {number} Sum of a and b
 */
function sum(a, b) {
  return a + b;
}
```

返回多类型的值:

```
/**
 * Returns the sum of a and b
 * @param {number} a
 * @param {number} b
 * @param {boolean} retArr If set to true, the function will return an array
 * @returns {(number|Array)} Sum of a and b or an array that contains a, b and the sum of a and b.
 */
function sum(a, b, retArr) {
  if (retArr) {
    return [a, b, a + b];
  }
  return a + b;
}
```

返回 `Promise`:

```
/**
 * Returns the sum of a and b
 * @param {number} a
 * @param {number} b
 * @returns {Promise} Promise object represents the sum of a and b
 */
function sumAsync(a, b) {
  return new Promise(function(resolve, reject) {
    resolve(a + b);
  });
}
```

相关链接

- @param
- @yields

@see

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@see <namepath>
@see <text>
```

概述

`@see` 标签表示可以参考另一个标识符的说明文档，或者一个外部资源。您可以提供一个标识符的 `namepath` 或自由格式的文本。如果你提供了一个 `namepath` ，JSDoc 的默认模板会自动将 `namepath` 转换成链接。

实例

使用 `@see` 标签：

```
/**
 * Both of these will link to the bar function.
 * @see {@link bar}
 * @see bar
 */
function foo() {}

// Use the inline {@link} tag to include a link within a free-form description.
/**
 * @see {@link foo} for further information.
 * @see {@link http://github.com|GitHub}
 */
function bar() {}
```

相关链接

- {@link}

@since

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@since <versionDescription>
```

概述

`@since` 标记表示在特定版本中添加了类、方法或其他符号。

实例

使用 `@since`：

```
/**
 * Provides access to user information.
```

```
 * @since 1.0.1
 */
function UserRecord() {}
```

相关链接

- @version

@static

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@static
```

概述

`@static` 标记表示符号包含在父项中，可以在不实例化父项的情况下访问。

使用 `@static` 标记将覆盖符号的默认作用域，但有一个例外：全局作用域中的符号将保持全局。

实例

下面的例子可以写成 `"@function MyNamespace.myFunction"` 并省略 `@memberof` 和 `@static` 标签，他们的效果是一样的.

```
/** @namespace MyNamespace */

/**
 * @function myFunction
 * @memberof MyNamespace
 * @static
 */
```

下面的示例强制模块的内部成员被描述为静态成员。

使用 `@static` 来覆盖默认作用域：

```
/** @module Rollerskate */

/**
 * The 'wheel' variable is documented as Rollerskate.wheel
 * rather than Rollerskate~wheel.
 * @static
 */
var wheel = 1;
```

相关链接

- @global
- @inner
- @instance

@summary

目录

- 语法
- 概述
- 实例

语法

```
@summary Summary goes here.
```

概述

`@summary` 标签是完整描述的一个简写版本。它可以被添加到任何的 doclet。

实例

```
/**
 * A very long, verbose, wordy, long-winded, tedious, verbacious, tautological,
 * profuse, expansive, enthusiastic, redundant, flowery, eloquent, articulate,
```

```
 * loquacious, garrulous, chatty, extended, babbling description.
 * @summary A concise summary.
 */
function bloviate() {}
```

@this

目录

- 语法
- 概述
- 实例

语法

```
@this <namePath>
```

概述

`@this` 标签指明 `this` 关键字的指向。

实例

在下面的例子中，`@this` 标签迫使 `"this.name"` 被描述为 `"Greeter#name"`，而不是全局变量 `"name"`。

```
/** @constructor */
function Greeter(name) {
  setName.apply(this, name);
}

/** @this Greeter */
function setName(name) {
  /** document me */
  this.name = name;
}
```

@throws

目录

- 别名
- 语法
- 概述
- 实例

别名

```
@exception
```

语法

```
@throws free-form description
@throws {<type>}
@throws {<type>} free-form description
```

概述

`@throws` 标记记录函数可能引发的错误。可以在一个 JSDoc 注释中多次包含 `@throws` 标记。

实例

使用带有类型的 `@throws` 标记:

```
/**
 * @throws {InvalidArgumentException}
 */
function foo(x) {}
```

使用带有描述的 `@throws` 标记:

```
/**
 * @throws Will throw an error if the argument is null.
 */
function bar(x) {}
```

使用带有类型和描述的 `@throws` 标记:

```
/**
 * @throws {DivideByZero} Argument x must be non-zero.
 */
function baz(x) {}
```

@todo

目录

- 语法
- 概述
- 实例

语法

```
@todo text describing thing to do.
```

概述

`@todo` 标签记录要完成的任务。在一个 JSDoc 注释块中您可以包含多个 `@todo` 标签。

实例

使用 `@todo` 标签：

```
/**
 * @todo Write the documentation.
 * @todo Implement this function.
 */
function foo() {
  // write me
}
```

@tutorial

目录

- 语法

- 概述
- 实例
- 相关链接

语法

```
@tutorial <tutorialID>
```

概述

`@tutorial` 标签插入一个指向向导教程的链接，作为文档的一部分。有关创建教程指导请参阅 tutorials overview。

可以在一个 JSDoc 注释中多次使用 `@tutorial` 标记。

实例

在下面的示例中，`MyClass` 的文档将链接到具有标识符 `tutorial-1` 和 `tutorial-2` 的教程。

```
/**
 * Description
 * @class
 * @tutorial tutorial-1
 * @tutorial tutorial-2
 */
function MyClass() {}
```

相关链接

- Tutorials
- {@tutorial}
- @see

@type

目录

- 语法
- 概述

- 实例
- 相关链接

语法

```
@type {typeName}
```

概述

`@type` 标记允许您提供一个类型表达式，用于标识符号可能包含的值类型或函数返回的值类型。您还可以将类型表达式包含在许多其他 JSDoc 标记中，比如 @param 标记。

类型表达式可以包括符号的 JSDoc namepath（例如，`myNamespace.MyClass`）、内置的 JavaScript 类型（例如，string）或它们的组合。您可以使用任何 Google Closure Compiler 编译器类型表达式，以及特定于JSDoc的其他几种格式。

如果 JSDoc 确定类型表达式无效，它将显示错误并停止运行。通过使用 `--lenient` 选项运行 JSDoc，可以将此错误转换为警告。

注意：JSDoc 3.2 和更高版本中完全支持 Google Closure Compiler 编译器样式的类型表达式。JSDoc 的早期版本包括对闭包编译器类型表达式的部分支持。

通过使用下面描述的格式之一提供类型表达式来指定每个类型。在适当的情况下，JSDoc 将自动为其他符号创建指向文档的链接。例如，如果该符号已被文档化，`@type {MyClass}`将链接到 `MyClass` 文档。

Symbol name (name expression)

```
{boolean}
{myNamespace.MyClass}
```

指定符号的名称。如果标识符已经被文档化，JSDoc 将创建一个链接到该标识符的文档。

Multiple types (type union)

```
{(number|boolean)}
```

这意味着值可能是几种类型中的一种，用括号括起来，并用"|"分隔类型的完整列表。

Arrays and objects (type applications and record types)

MyClass的实例的数组:

```
{Array.<MyClass>}
// or:
{MyClass[]}
```

具有字符串键和数值的对象：

```
{Object.<string, number>}
```

名为 `myObj` 的对象，属性为“a”（数字）、“b”（字符串）和“c”（任何类型）：

```
{{a: number, b: string, c}} myObj
// or:
{Object} myObj
{number} myObj.a
{string} myObj.b
{*} myObj.c
```

JSDoc 支持 Closure Compiler 语法定义的数组和对象类型。

还可以通过数组后面附加 `[]` 指示包含在数组中的类型。例如，表达式 `string[]` 表示字符串数组。

对于具有一组已知的属性的对象，你可以使用 Closure Compiler 语法文档化标注的类型。也可以分别描述每个属性，这使您能够提供有关每个属性的更多详细信息。

Nullable type

一个数字或空值：

```
{?number}
```

指明类型为指定的类型，或者为 `null`。

Non-nullable type

一个数字，但是绝对不会是 `null`：

```
{!number}
```

指明类型为指定的类型，但是绝对不会是 `null`。

Variable number of that type

此函数接受可变数量的数值参数:

```
@param {...number} num
```

指示函数接受可变数目的参数，并指定参数的类型。例如：

```
/**
 * Returns the sum of all numbers passed to the function.
 * @param {...number} num A positive or negative number
 */
function sum(num) {
  var i=0, n=arguments.length, t=0;
  for (; i<n; i++) {
    t += arguments[i];
  }
  return t;
}
```

Optional parameter

一个可选参数 `foo`：

```
@param {number} [foo]
// or:
@param {number=} foo
```

一个可选参数 `foo`，默认值为 `1`：

```
@param {number} [foo=1]
```

表示该参数是可选的。当对可选参数使用 JSDoc 的语法时，还可以显示在省略参数时将使用的值。

Callbacks

```
/**
 * @callback myCallback
 * @param {number} x - ...
 */

/** @type {myCallback} */
var cb;
```

使用 `@callback` 标签指明一个回调。和 `@typedef` 标签是相同的，不同之处在于回调的类型始终是"function"。

Type definitions

记录 `id`，`name`，`age` 属性的类型：

```
/**
 * @typedef PropertiesHash
 * @type {object}
 * @property {string} id - an ID.
 * @property {string} name - your name.
 * @property {number} age - your age.
 */

/** @type {PropertiesHash} */
var props;
```

您可以使用 `@typedef` 标签记录复杂类型，然后参考类型定义在你文档的其他地方。

实例

```
/** @type {(string|Array.)} */
var foo;
/** @type {number} */
var bar = 1;
```

在许多情况下，您可以包含一个类型表达式作为另一个标签的一部分，而不是在 JSDoc 注释块中包含独立 `@type` 标签。

类型表达式可以有多个标签：

```
/**
 * @type {number}
 * @const
 */
var FOO = 1;

// same as:

/** @const {number} */
var FOO = 1;
```

相关链接

- @callback

- @typedef
- @param
- @property

@typedef

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@typedef [<type>] <namepath>
```

概述

`@typedef` 标签在描述自定义类型时是很有用的，特别是如果你要反复引用它们的时候。这些类型可以在其它标签内使用，如 `@type` 和 `@param`。

使用 `@callback` 标签表明回调函数的类型。

实例

这个例子定义了一个联合类型的参数，表示可以包含数字或字符串。

使用 `@typedef` 标签：

```
/**
 * A number, or a string containing a number.
 * @typedef {(number|string)} NumberLike
 */

/**
 * Set the magic number.
 * @param {NumberLike} x - The magic number.
 */
function setMagicNumber(x) {
}
```

这个例子定义了一个更复杂的类型，一个具有多个属性的对象，并设置了它的 `namepath`，这样它将与使用该类型的类一起显示。因为类型定义实际上不是由类公开的，所以通常将类型定义记录为内部成员。

使用 `@typedef` 记录类的复杂类型:

```
/**
 * The complete Triforce, or one or more components of the Triforce.
 * @typedef {Object} WishGranter~Triforce
 * @property {boolean} hasCourage - Indicates whether the Courage component is present.
 * @property {boolean} hasPower - Indicates whether the Power component is present.
 * @property {boolean} hasWisdom - Indicates whether the Wisdom component is present.
 */

/**
 * A class for granting wishes, powered by the Triforce.
 * @class
 * @param {...WishGranter~Triforce} triforce - One to three {@link WishGranter~Triforce} objects
 * containing all three components of the Triforce.
 */
function WishGranter(triforce) {}
```

相关链接

- @callback
- @param
- @type

@variation

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@variation <variationNumber>
```

概述

有时代码可能包含多个具有相同长名称的符号。例如，您可能同时拥有一个全局类和一个名为 `Widget` 的顶级命名空间。在这种情况下，`{@link Widget}` 或 `@memberof Widget` 是什么意思？全局命名空间，还是全局类？

变体有助于 JSDoc 区分具有相同长名称的不同符号。例如，如果将 `“@variation 2”` 添加到小部件类的JSDoc注释中，`“{@link Widget(2)}”` 将引用该类，`“{@link Widget}”` 将引用命名空间。或者，当您使用诸如 `@alias` 或 `@name` 之类的标记（例如，`“@alias Widget(2)”`）指定符号时，也可以包含变体。

您可以使用 `@variation` 标记提供任何值，只要该值和 longname 的组合产生了 longname 的全局唯一版本。作为最佳实践，使用可预测的模式来选择值，这将使您更容易编写代码文档。

实例

下面的示例使用 `@variation` 标签来区分 `Widget` 类和 `Widget` 命名空间。

```
/**
 * The Widget namespace.
 * @namespace Widget
 */

// you can also use '@class Widget(2)' and omit the @variation tag
/**
 * The Widget class. Defaults to the properties in {@link Widget.properties}.
 * @class
 * @variation 2
 * @param {Object} props - Name-value pairs to add to the widget.
 */
function Widget(props) {}

/**
 * Properties added by default to a new {@link Widget(2)} instance.
 */
Widget.properties = {
  /**
   * Indicates whether the widget is shiny.
   */
  shiny: true,
  /**
   * Indicates whether the widget is metallic.
   */
  metallic: true
};
```

相关链接

- @alias
- @name

@version

目录

- 语法
- 概述
- 实例
- 相关链接

语法

```
@version <version>
```

概述

`@version` 标签后面的文本将被用于表示该项的版本。

实例

使用 `@version` 标签：

```
/**
 * Solves equations of the form a * x = b. Returns the value
 * of x.
 * @version 1.2.3
 * @tutorial solver
 */
function solver(a, b) {
  return b / a;
}
```

相关链接

- @since

@yields

目录

- 别名
- 语法
- 概述
- 实例
- 相关链接

别名

```
@yield
```

语法

```
@yields [{type}] [description]
```

概述

`@yields` 标记记录生成器函数生成的值。JSDoc 3.5.0 及更高版本中提供了此标记。

如果要记录常规函数，请使用 @returns 标记而不是此标记。

实例

带有类型的 `@yields`：

```
/**
 * Generate the Fibonacci sequence of numbers.
 *
 * @yields {number}
 */
function* fibonacci() {}
```

带有类型和描述的 `@yields`：

```
/**
 * Generate the Fibonacci sequence of numbers.
 *
 * @yields {number} The next number in the Fibonacci sequence.
 */
function* fibonacci() {}
```

相关链接

- @returns