Javascript格式规则

命名

通常来说，使用 `functionNamesLikeThis` ， `variableNamesLikeThis` ， `ClassNamesLikeThis` ， `EnumNamesLikeThis` ， `methodNamesLikeThis` ， `CONSTANT_VALUES_LIKE_THIS` ， `foo.namespaceNamesLikeThis.bar` 和 `filenameslikethis.js` 这种格式的命名方式。

属性和方法

- *私有* 属性和方法应该以下划线开头命名。
- *保护* 属性和方法应该以无下划线开头命名（像公共属性和方法一样）。

了解更多关于私有成员和保护成员的信息，请阅读 可见性 部分。

方法和函数参数

可选函数参数以 `opt_` 开头。

参数数目可变的函数应该具有以 `var_args` 命名的最后一个参数。你可能不会在代码里引用 `var_args` ；使用 `arguments` 对象。

可选参数和数目可变的参数也可以在注释 `@param` 中指定。尽管这两种惯例都被编译器接受，但更加推荐两者一起使用。

getter和setter

EcmaScript 5 不鼓励使用属性的getter和setter。然而，如果使用它们，那么getter就不要改变属性的可见状态。

```
/**
*错误--不要这样做.
*/
var foo = { get next() { return this.nextId++; } };
};
```

存取函数

属性的getter和setter方法不是必需的。然而，如果使用它们，那么读取方法必须以 `getFoo()` 命名，并且写入方法必须以 `setFoo(value)` 命名。（对于布尔型的读取方法，也可以使用 `isFoo()` ，并且这样往往听起来更自然。）

命名空间

JavaScript没有原生的对封装和命名空间的支持。

全局命名冲突难以调试，并且当两个项目尝试整合的时候可能引起棘手的问题。为了能共享共用的JavaScript代码，我们采用一些约定来避免冲突。

为全局代码使用命名空间

在全局范围内 *总是* 使用唯一的项目或库相关的伪命名空间进行前缀标识。如果你正在进行“Project Sloth”项目，一个合理的伪命名空间为 `sloth.*` 。

```
var sloth = {};

sloth.sleep = function() {
  ...
};
```

很多JavaScript库，包括 the Closure Library 和 Dojo toolkit 给你高级功能来声明命名空间。保持你的命名空间声明一致。

```
goog.provide('sloth');

sloth.sleep = function() {
  ...
};
```

尊重命名空间所有权

当选择一个子命名空间的时候，确保父命名空间知道你在做什么。如果你开始了一个为sloths创建hats的项目，确保Sloth这一组命名空间知道你在使用 `sloth.hats` 。

外部代码和内部代码使用不同的命名空间

“外部代码”指的是来自你的代码库外并独立编译的代码。内部名称和外部名称应该严格区分开。如果你正在使用一个能调用 `foo.hats.*` 中的东西的外部库，你的内部代码不应该定义 `foo.hats.*` 中的所有符号，因为如果其他团队定义新符号就会把它打破。

```
foo.require('foo.hats');
/**
*错误--不要这样做。
* @constructor
* @extends {foo.hats.RoundHat}
*/
foo.hats.BowlerHat = function() {
};
```

如果你在外部命名空间中需要定义新的API，那么你应该明确地导出且仅导出公共的API函数。为了一致性和编译器更好的优化你的内部代码，你的内部代码应该使用内部API的内部名称调用它们。

```
foo.provide('googleyhats.BowlerHat');

foo.require('foo.hats');
/**
* @constructor
* @extends {foo.hats.RoundHat}
*/
googleyhats.BowlerHat = function() {
  ...
};
goog.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);
```

为长类型的名称提供别名提高可读性

如果对完全合格的类型使用本地别名可以提高可读性，那么就这样做。本地别名的名称应该符合类型的最后一部分。

```
/**
* @constructor
*/
some.long.namespace.MyClass = function() {
};

/**
* @param {some.long.namespace.MyClass} a
*/
some.long.namespace.MyClass.staticHelper = function(a) {
  ...
};

myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  var staticHelper = some.long.namespace.MyClass.staticHelper;
  staticHelper(new MyClass());
};
```

不要为命名空间起本地别名。命名空间应该只能使用 goog.scope 命名别名。

```
myapp.main = function() {
  var namespace = some.long.namespace;
  namespace.MyClass.staticHelper(new namespace.MyClass());
};
```

避免访问一个别名类型的属性，除非它是一个枚举。

```
/** @enum {string} */
some.long.namespace.Fruit = {
  APPLE: 'a',
  BANANA: 'b'
};

myapp.main = function() {
  var Fruit = some.long.namespace.Fruit;
  switch (fruit) {
    case Fruit.APPLE:
      ...
    case Fruit.BANANA:
      ...
  }
};
```

```
myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  MyClass.staticHelper(null);
};
```

永远不要在全局环境中创建别名。只在函数体内使用它们。

文件名

为了避免在大小写敏感的平台上引起混淆，文件名应该小写。文件名应该以 `.js` 结尾，并且应该不包含除了 `-` 或 `_` （相比较 `_` 更推荐 `-` ）以外的其它标点。

自定义 toString() 方法

必须确保无误，并且无其他副作用。

你可以通过自定义 `toString()` 方法来控制对象如何字符串化他们自己。这没问题，但是你必须确保你的方法执行无误，并且无其他副作用。如果你的方法没有达到这个要求，就会很容易产生严重的问题。比如，如果 `toString()` 方法调用一个方法产生一个断言，断言可能要输出对象的名称，就又需要调用 `toString()` 方法。

延时初始化

可以使用。并不总在变量声明的地方就进行变量初始化，所以延时初始化是可行的。

明确作用域

时常。

经常使用明确的作用域加强可移植性和清晰度。例如，在作用域链中不要依赖 `window` 。你可能想在其他应用中使用你的函数，这时此 `window` 就非彼 `window` 了。

代码格式

我们原则上遵循 C++格式规范 ， 并且进行以下额外的说明。

大括号

由于隐含分号的插入，无论大括号括起来的是什么，总是在同一行上开始你的大括号。例如：

```
if (something) {
  // ...
} else {
  // …
}
```

数组和对象初始化表达式

当单行数组和对象初始化表达式可以在一行写开时，写成单行是允许的。

```
var arr = [1, 2, 3];  //之后无空格[或之前]
var obj = {a: 1, b: 2, c: 3};  //之后无空格[或之前]
```

多行数组和对象初始化表达式缩进两个空格，括号的处理就像块一样单独成行。

```
//对象初始化表达式
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};

//数组初始化表达式
this.rows_ = [
  '"Slartibartfast" <fjordmaster@magrathea.com>',
  '"Zaphod Beeblebrox" <theprez@universe.gov>',
  '"Ford Prefect" <ford@theguide.com>',
  '"Arthur Dent" <has.no.tea@gmail.com>',
  '"Marvin the Paranoid Android" <marv@googlemail.com>',
  'the.mice@magrathea.com'
];

//在方法调用中使用
goog.dom.createDom(goog.dom.TagName.DIV, {
  id: 'foo',
  className: 'some-css-class',
  style: 'display:none'
}, 'Hello, world!');
```

长标识符或值在对齐的初始化列表中存在问题，所以初始化值不必对齐。例如：

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
  lengthyName: 2
};
```

不要像这样：

```
WRONG_Object.prototype = {
  a          : 0,
  b          : 1,
  lengthyName: 2
};
```

函数参数

如果可能，应该在同一行上列出所有函数参数。如果这样做将超出每行80个字符的限制，参数必须以一种可读性较好的方式进行换行。为了节省空间，在每一行你可以尽可能的接近80个字符，或者把每一个参数单独放在一行以提高可读性。缩进可能是四个空格，或者和括号对齐。下面是最常见的参数换行形式：

```
// 四个空格，每行包括80个字符。适用于非常长的函数名，
// 重命名不需要重新缩进，占用空间小。
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
};

//四个空格，每行一个参数。适用于长函数名，
// 允许重命名，并且强调每一个参数。
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
    // ...
};

// 缩进和括号对齐，每行80字符。 看上去是分组的参数，
// 占用空间小。
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
            tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
}

// 和括号对齐，每行一个参数。看上去是分组的并且
// 强调每个单独的参数。
function bar(veryDescriptiveArgumentNumberOne,
            veryDescriptiveArgumentTwo,
            tableModelEventHandlerProxy,
            artichokeDescriptorAdapterIterator) {
    // ...
}
```

当函数调用本身缩进，你可以自由地开始相对于原始声明的开头或者相对于当前函数调用的开头，进行4个空格的缩进。以下都是可接受的缩进风格。

```
if (veryLongFunctionNameA(
        veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
  veryLongFunctionNameC(veryLongFunctionNameD(
      veryLongFunctioNameE(
          veryLongFunctionNameF)));
}
```

匿名函数传递

当在一个函数的参数列表中声明一个匿名函数时，函数体应该与声明的左边缘缩进两个空格，或者与function关键字的左边缘缩进两个空格。这是为了匿名函数体更加可读（即不被挤在屏幕的右侧）。

```
prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
```

```
  }
});

var names = prefix.something.myExcellentMapFunction(
    verboselyNamedCollectionOfItems,
    function(item) {
      return item.name;
    });
```

使用goog.scope命名别名

goog.scope 可用于在使用 the Closure Library 的工程中缩短命名空间的符号引用。

每个文件只能添加一个 `goog.scope` 调用。始终将它放在全局范围内。

开放的 `goog.scope(function() {` 调用必须在之前有一个空行，并且紧跟 `goog.provide` 声明、 `goog.require` 声明或者顶层的注释。调用必须在文件的最后一行闭合。在scope声明闭合处追加 `// goog.scope` 。注释与分号间隔两个空格。

和C++命名空间相似，不要在 `goog.scope` 声明下面缩进。相反，从第0列开始。

只取不会重新分配给另一个对象（例如大多数的构造函数、枚举和命名空间）的别名。不要这样做：

```
goog.scope(function() {
var Button = goog.ui.Button;

Button = function() { ... };
...
```

别名必须和全局中的命名的最后一个属性相同。

```
goog.provide('my.module');

goog.require('goog.dom');
goog.require('goog.ui.Button');

goog.scope(function() {
var Button = goog.ui.Button;
var dom = goog.dom;

// Alias new types after the constructor declaration.
my.module.SomeType = function() { ... };
var SomeType = my.module.SomeType;

// Declare methods on the prototype as usual:
SomeType.prototype.findButton = function() {
  // Button as aliased above.
  this.button = new Button(dom.getElement('my-button'));
};
...
});  // goog.scope
```

更多的缩进

事实上，除了 初始化数组和对象 和传递匿名函数外，所有被拆开的多行文本应与之前的表达式左对齐，或者以4个（而不是2个）空格作为一缩进层次。

```
someWonderfulHtml = '' +
                    getEvenMoreHtml(someReallyInterestingValues, moreValues,
                                    evenMoreParams, 'a duck', true, 72,
                                    slightlyMoreMonkeys(0xfff)) +
                    '';

thisIsAVeryLongVariableName =
```

```
    hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();

thisIsAVeryLongVariableName = 'expressionPartOne' + someMethodThatIsLong() +
    thisIsAnEvenLongerOtherFunctionNameThatCannotBeIndentedMore();

someValue = this.foo(
    shortArg,
    'Some really long string arg - this is a pretty common case, actually.',
    shorty2,
    this.bar());

if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&
    !ambientNotification.isActive() && (client.isAmbientSupported() ||
                                        client.alwaysTryAmbientAnyways())) {
  ambientNotification.activate();
}
```

空行

使用新的空行来划分一组逻辑上相关联的代码片段。例如：

```
doSomethingTo(x);
doSomethingElseTo(x);
andThen(x);

nowDoSomethingWith(y);

andNowWith(z);
```

二元和三元操作符

操作符始终跟随着前行, 这样你就不用顾虑分号的隐式插入问题。否则换行符和缩进还是遵循其他谷歌规范指南。

```
var x = a ? b : c;  // All on one line if it will fit.

// Indentation +4 is OK.
var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

// Indenting to the line position of the first operand is also OK.
var z = a ?
        moreComplicatedB :
        moreComplicatedC;
```

点号也应如此处理。

```
var x = foo.bar().
    doSomething().
    doSomethingElse();
```

括号

只用在有需要的地方。

通常只在语法或者语义需要的地方有节制地使用。

绝对不要对一元运算符如 `delete` 、 `typeof` 和 `void` 使用括号或者在关键词如 `return` 、 `throw` 和其他的（ `case` 、 `in` 或者 `new` ）之后使用括号。

字符串

使用 ' 代替 " 。使用单引号（ ' ）代替双引号（ " ）来保证一致性。当我们创建包含有HTML的字符串时这样做很有帮助。

```
var msg = 'This is some HTML';
```

可见性（私有和保护类型字段）

鼓励使用 @private 和 @protected JSDoc注释。

我们建议使用JSDoc注释 @private 和 @protected 来标识出类、函数和属性的可见程度。

设置 --jscomp_warning=visibility 可令编译器对可见性的违规进行编译器警告。可见 封闭的编译器警告 。

加了 @private 标记的全局变量和函数只能被同一文件中的代码所访问。

被标记为 @private 的构造函数只能被同一文件中的代码或者它们的静态和实例成员实例化。 @private 标记的构造函数可以被相同文件内它们的公共静态属性和 instanceof 运算符访问。

全局变量、函数和构造函数不能注释 @protected 。

```
// 文件1
// AA_PrivateClass_ 和 AA_init_ 是全局的并且在同一个文件中所以能被访问

/**
* @private
* @constructor
*/
AA_PrivateClass_ = function() {
};

/** @private */
function AA_init_() {
  return new AA_PrivateClass_();
}

AA_init_();
```

标记 @private 的属性可以被同一文件中的所有的代码访问，如果属性属于一个类，那么所有自身含有属性的类的静态方法和实例方法也可访问。它们不能被不同文件下的子类访问或者重写。

标记 @protected 的属性可以被同一文件中的所有的代码访问，任何含有属性的子类的静态方法和实例方法也可访问。

注意这些语义和C++、JAVA中private 和 protected的不同，其许可同一文件中的所有代码访问的权限，而不是仅仅局限于同一类或者同一类层次。此外，不像C++中，子类不可重写私有属性。

```
// File 1.

/** @constructor */
AA_PublicClass = function() {
  /** @private */
  this.privateProp_ = 2;

  /** @protected */
  this.protectedProp = 4;
};

/** @private */
AA_PublicClass.staticPrivateProp_ = 1;

/** @protected */
AA_PublicClass.staticProtectedProp = 31;

/** @private */
AA_PublicClass.prototype.privateMethod_ = function() {};
```

```
/** @protected */
AA_PublicClass.prototype.protectedMethod = function() {};

// File 2.

/**
* @return {number} The number of ducks we've arranged in a row.
*/
AA_PublicClass.prototype.method = function() {
  // Legal accesses of these two properties.
  return this.privateProp_ + AA_PublicClass.staticPrivateProp_;
};

// File 3.

/**
* @constructor
* @extends {AA_PublicClass}
*/
AA_SubClass = function() {
  // Legal access of a protected static property.
  AA_PublicClass.staticProtectedProp = this.method();
};
goog.inherits(AA_SubClass, AA_PublicClass);

/**
* @return {number} The number of ducks we've arranged in a row.
*/
AA_SubClass.prototype.method = function() {
  // Legal access of a protected instance property.
  return this.protectedProp;
};
```

注意在Javascript中，一个类（如 `AA_PrivateClass_` ）和其构造函数类型是没有区别的。没办法确定一种类型是public而它的构造函数是private。（因为构造函数很容易重命名从而躲避隐私检查）。

JavaScript类型

鼓励和强制执行的编译器。

JSDoc记录类型时，要尽可能具体和准确。我们支持的类型是基于 EcmaScript 4规范 。

JavaScript类型语言

ES4提案包含指定JavaScript类型的语言。我们使用JsDoc这种语言表达函数参数和返回值的类型。

随着ES4提议的发展，这种语言已经改变了。编译器仍然支持旧的语法类型，但这些语法已经被弃用了。

语法名称	语法	描述	弃用语法
原始类型	在JavaScript中有5种原始类型：`{null}` ， `{undefined}` ， `{boolean}` ， `{number}` ，和 `{string}`	类型的名称。	
实例类型	`{Object}` 实例对象或空。 `{Function}` 一个实例函数或空。 `{EventTarget}` 构造函数实现的EventTarget接口，或者为null的一个实例。	一个实例构造函数或接口函数。构造函数是 `@constructor` JSDoc标记定义的函数 。接口函数是 `@interface` JSDoc标记定义的函数。 默认情况下，实例类型将接受空。这是唯一的类型语法，使得类型为空。此表中的其他类型的语法不会接受空。	

语法名称	语法	描述	弃用语法
枚举类型	`{goog.events.EventType}` 字面量初始化对象的属性之一 `goog.events.EventType` 。	一个枚举必须被初始化为一个字面量对象，或作为另一个枚举的别名,加注 `@enum` JSDoc标记。这个属性是枚举实例。 下面 是枚举语法的定义。 请注意，这是我们的类型系统中为数不多的ES4规范以外的事情之一。	
应用类型	`{Array.<string>}` 字符串数组。 `{Object.<string, number>}` 一个对象，其中键是字符串，值是数字。	参数化类型，该类型应用一组参数类型。这个想法是类似于Java泛型。	

联合类型	`{(number|boolean)}` 一个数字或布尔值。	表明一个值可能有A型或B型。
		
		括号在顶层表达式可以省略，但在子表达式不能省略，以避免歧义。
		
		`{number|boolean}`
		
		`{function(): (number|boolean)}`
			`{(number,boolean)}` ，`{(number||boolean)}`

可为空的类型	`{?number}` 一个数字或空。	空类型与任意其他类型组合的简称。这仅仅是语法糖（syntactic sugar）。	`{number?}`
非空类型	`{!Object}` 一个对象，值非空。	从非空类型中过滤掉null。最常用于实例类型，默认可为空。	`{Object!}`
记录类型	`{{myNum: number, myObject}}` 给定成员类型的匿名类型。	表示该值有指定的类型的成员。在这种情况下，`myNum` 是 `number` 类型而 `myObject` 可为任何类型。 注意花括号是语法类型的一部分。例如，表示一个数组对象有一个 `length` 属性，你可以写 `Array.<{length}>` 。	
函数类型	`{function(string, boolean)}` 一个函数接受两个参数（一个字符串和一个布尔值），并拥有一个未知的返回值。	指定一个函数。	
函数返回类型	`{function(): number}` 一个函数没有参数并返回一个数字。	指定函数的返回类型。	
函数 `this` 类型	`{function(this:goog.ui.Menu, string)}` 一个需要一个参数（字符串）的函数，执行上下文是 `goog.ui.Menu`	指定函数类型的上下文类型。	

语法名称	语法	描述	弃用语法
函数 new 类型	`{function(new:goog.ui.Menu, string)}` 一个构造函数接受一个参数（一个字符串），并在使用“new”关键字时创建一个 goog.ui.Menu 新实例。	指定构造函数所构造的类型。	
可变参数	`{function(string, ...[number]): number}` 一个函数，它接受一个参数（一个字符串），然后一个可变数目的参数，必须是数字。	指定函数的变量参数。	
可变参数（ @param 注释）	`@param {...number} var_args` 带注释函数的可变数目参数。	指定带注释函数接受一个可变数目的参数。	
函数 可选参数	`{function(?string=, number=)}` 一个函数，它接受一个可选的、可以为空的字符串和一个可选的数字作为参数。“=”只用于函数类型声明。	指定函数的可选参数。	
函数 可选参数 （ @param 注释）	`@param {number=} opt_argument` number 类型的可选参数。	指定带注释函数接受一个可选的参数。	
所有类型	`{*}`	表明该变量可以接受任何类型。	
未知类型	`{?}`	表明该变量可以接受任何类型，编译器不应该检查其类型。	

JavaScript中的类型

类型举例	取值举例	描述

number	`1`
	`1.0`
	`-5`
	`1e5`
	`Math.PI`	

Number	`new Number(true)`	Number对象

string	`'Hello'`
	`"World"`
	`String(42)`
		字符串

类型举例	取值举例	描述
String	`new String('Hello')` `new String(42)`	String对象

boolean	`true`
	`false`
	`Boolean(0)`
		Boolean值

Boolean	`new Boolean(true)`	Boolean对象
RegExp	`new RegExp('hello')` `/world/g`	
Date	`new Date` `new Date()`	
null	`null`	
undefined	`undefined`	

void	`function f() {`
	`  return;`
	`}`
		没有返回值

Array	`['foo', 0.3, null]` `[]`	无类型数组
Array.<number>	`[11, 22, 33]`	数字数组
Array.<Array.<string>>	`[['one', 'two', 'three'], ['foo', 'bar']]`	以字符串为元素的数组，作为另一个数组的元素
Object	`{}` `{foo: 'abc', bar: 123, baz: null}`	
Object.<string>	`{'foo': 'bar'}`	值为字符串的对象

类型举例	取值举例	描述
Object.<number, string>	`var obj = {};` `obj[1] = 'bar';`	键为整数，值为字符串的对象。 注意，js当中键总是会隐式转换为字符串。所以 `obj['1'] == obj[1]` 。键在for...in...循环中，总是字符串类型。但在对象中索引时编译器会验证键的类型。

Function	`function(x, y) {`
	`  return x * y;`
	`}`
		Function对象

function(number, number): number	`function(x, y) {`
	`  return x * y;`
	`}`
		函数值

类	`/** @constructor */`
	`function SomeClass() {}`
	
	`new SomeClass();`	

接口	`/** @interface */`
	`function SomeInterface() {}`
	
	`SomeInterface.prototype.draw = function() {};`	

project.MyClass	`/** @constructor */`
	`project.MyClass = function () {}`
	
	`new project.MyClass()`	

project.MyEnum	`/** @enum {string} */`
	`project.MyEnum = {`
	`  /** The color blue. */`
	`  BLUE: '#0000dd',`
	`  /** The color red. */`
	`  RED: '#dd0000'`
	`};`
		枚举
		
		JSDoc中枚举的值都是可选的.

Element	`document.createElement('div')`	DOM元素
Node	`document.body.firstChild`	DOM节点
HTMLInputElement	`htmlDocument.getElementsByTagName('input')[0]`	指明类型的DOM元素

类型转换

在类型检测不准确的情况下，有可能需要添加类型的注释，并且把类型转换的表达式写在括号里，括号是必须的。如：

```
/** @type {number} */ (x)
```

可为空与可选的参数和属性

因为Javascript是一个弱类型的语言，明白函数参数、类属性的可选、可为空和未定义之间的细微差别是非常重要的。

对象类型和引用类型默认可为空。如以下表达式：

```
/**
* 传入值初始化的类
* @param {Object} value某个值
* @constructor
*/
function MyClass(value) {
  /**
   * Some value.
   * @type {Object}
   * @private
   */
  this.myValue_ = value;
}
```

告诉编译器 `myValue_` 属性为一对象或null。如果 `myValue_` 永远都不会为null, 就应该如下声明:

```
/**
* 传入非null值初始化的类
* @param {!Object} value某个值
* @constructor
*/
function MyClass(value) {
  /**
   * Some value.
   * @type {!Object}
   * @private
   */
  this.myValue_ = value;
}
```

这样，如果编译器可以识别出 `MyClass` 初始化传入值为null，就会发出一个警告。

函数的可选参数在运行时可能会是undefined，所以如果他们是类的属性，那么必须声明：

```
/**
* 传入可选值初始化的类
* @param {Object=} opt_value某个值（可选）
* @constructor
*/
function MyClass(opt_value) {
  /**
   * Some value.
   * @type {Object|undefined}
   * @private
   */
  this.myValue_ = opt_value;
}
```

这告诉编译器 `myValue_` 可能是一个对象，或 `null` ，或 `undefined` 。

注意: 可选参数 `opt_value` 被声明成 `{Object=}` ，而不是 `{Object|undefined}` 。这是因为可选参数可能是undefined。虽然直接写undefined也并无害处，但鉴于可阅读性还是写成上述的样子。

最后，属性的可为空和可选并不矛盾，下面的四种声明各不相同：

```
/**
* 接受四个参数，两个可为空，两个可选
* @param {!Object} nonNull 必不为null
* @param {Object} mayBeNull 可为null
* @param {!Object=} opt_nonNull 可选但必不为null
* @param {Object=} opt_mayBeNull 可选可为null
*/
function strangeButTrue(nonNull, mayBeNull, opt_nonNull, opt_mayBeNull) {
  // ...
};
```

类型定义

有时类型可以变得复杂。一个函数，它接受一个元素的内容可能看起来像：

```
/**
* @param {string} tagName
* @param {(string|Element|Text|Array.<Element>|Array.<Text>)} contents
* @return {!Element}
*/
goog.createElement = function(tagName, contents) {
  ...
};
```

你可以定义带 `@typedef` 标记的常用类型表达式。例如：

```
/** @typedef {(string|Element|Text|Array.<Element>|Array.<Text>)} */
goog.ElementContent;

/**
* @param {string} tagName
* @param {goog.ElementContent} contents
* @return {!Element}
*/
goog.createElement = function(tagName, contents) {
...
};
```

模板类型

编译器对模板类型提供有限支持。它只能从字面上通过 `this` 参数的类型和 `this` 参数是否丢失推断匿名函数的 `this` 类型。

```
/**
* @param {function(this:T, ...)} fn
* @param {T} thisObj
* @param {...*} var_args
* @template T
*/
goog.bind = function(fn, thisObj, var_args) {
...
};
//可能出现属性丢失警告
goog.bind(function() { this.someProperty; }, new SomeClass());
//出现this未定义警告
goog.bind(function() { this.someProperty; });
```

注释

使用JSDoc。

我们使用 c++的注释风格 。 所有的文件、类、方法和属性都应该用合适的 JSDoc 的 标签 和 类型 注释。除了直观的方法名称和参数名称外，方法的描述、方法的参数以及方法的返回值也要包含进去。

行内注释应该使用 `//` 的形式。

为了避免出现语句片段，要使用正确的大写单词开头，并使用正确的标点符号作为结束。

注释语法

JSDoc的语法基于 JavaDoc ， 许多编译工具从JSDoc注释中获取信息从而进行代码验证和优化，所以这些注释必须符合语法规则。

```
/**
* A JSDoc comment should begin with a slash and 2 asterisks.
* Inline tags should be enclosed in braces like {@code this}.
* @desc Block tags should always start on their own line.
*/
```

JSDoc 缩进

如果你不得不进行换行，那么你应该像在代码里那样，使用四个空格进行缩进。

```
/**
* Illustrates line wrapping for long param/return descriptions.
* @param {string} foo This is a param with a description too long to fit in
*     one line.
* @return {number} This returns something that has a description too long to
*     fit in one line.
*/
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

不必在 `@fileoverview` 标记中使用缩进。

虽然不建议，但依然可以对描述文字进行排版。

```
/**
* This is NOT the preferred indentation method.
* @param {string} foo This is a param with a description too long to fit in
*                     one line.
* @return {number} This returns something that has a description too long to
*                  fit in one line.
*/
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

JSDoc中的HTML

像JavaDoc一样, JSDoc 支持很多的HTML标签，像 `<code>` ， `<pre>` ， `<tt>` ， `<strong>` ， `<ul>` ， `<ol>` ， `<li>` ， `<a>` 等。

这就意味着不建议采用纯文本的格式。所以，不要在JSDoc里使用空白符进行格式化。

```
/**
* Computes weight based on three factors:
*  items sent
*  items received
*  last timestamp
*/
```

上面的注释会变成这样：

```
Computes weight based on three factors: items sent items received items received last timestamp
```

所以，用下面的方式代替：

```
/**
* Computes weight based on three factors:
* <ul>
* <li>items sent
* <li>items received
* <li>last timestamp
* </ul>
*/
```

JavaDoc 风格指南对于如何编写良好的doc注释是非常有帮助的。

顶层/文件层注释

版权声明 和作者信息是可选的。顶层注释的目的是为了让不熟悉代码的读者了解文件中有什么。它需要描述文件内容，依赖关系以及兼容性的信息。例如：

```
/**
* @fileoverview Description of file, its uses and information
* about its dependencies.
*/
```

Class评论

类必须记录说明与描述和 一个类型的标签 ， 标识的构造函数。类必须加以描述，若是构造函数则需标注出。

```
/**
* Class making something fun and easy.
* @param {string} arg1 An argument that makes this more interesting.
* @param {Array.<number>} arg2 List of numbers to be processed.
* @constructor
* @extends {goog.Disposable}
*/
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

方法和功能注释

参数和返回类型应该被记录下来。如果方法描述从参数或返回类型的描述中明确可知则可以省略。方法描述应该由一个第三人称表达的句子开始。

```
/**
* Operates on an instance of MyClass and returns something.
* @param {project.MyClass} obj Instance of MyClass which leads to a long
*    comment that needs to be wrapped to two lines.
* @return {boolean} Whether something occured.
*/
function PR_someMethod(obj) {
  // ...
}
```

属性评论

```
/** @constructor */
project.MyClass = function() {
/**
  * Maximum number of things per pane.
  * @type {number}
  */
  this.someProperty = 4;
}
```

JSDoc标签参考

你也许在第三方代码中看到其他类型JSDoc注释，这些注释出现在 JSDoc Toolkit标签的参考 ，但目前在谷歌的代码中不鼓励使用。你应该将他们当作"保留"字，他们包括：

- @augments
- @argument
- @borrows
- @class
- @constant
- @constructs
- @default
- @event
- @example
- @field
- @function
- @ignore
- @inner
- @link
- @memberOf
- @name
- @namespace
- @property
- @public
- @requires
- @returns
- @since
- @static
- @version

为goog.provide提供依赖

只提供顶级符号。

一个类上定义的所有成员应该放在一个文件中。所以，在一个在相同类中定义的包含多个成员的文件中只应该提供顶级的类（例如枚举、内部类等）。

要这样写：

```
goog.provide('namespace.MyClass');
```

不要这样写：

```
goog.provide('namespace.MyClass');
goog.provide('namespace.MyClass.Enum');
goog.provide('namespace.MyClass.InnerClass');
goog.provide('namespace.MyClass.TypeDef');
goog.provide('namespace.MyClass.CONSTANT');
goog.provide('namespace.MyClass.staticMethod');
```

命名空间的成员也应该提供：

```
goog.provide('foo.bar');
goog.provide('foo.bar.method');
goog.provide('foo.bar.CONSTANT');
```

编译

必需。

对于所有面向客户的代码来说，使用JS编辑器是必需的，如使用 Closure Compiler 。

技巧和诀窍

JavaScript帮助信息

True和False布尔表达式

下边的布尔表达式都返回false：

- null
- undefined
- ''空字符串
- 数字0

但是要小心，因为以下这些返回true：

- 字符串"0"
- []空数组
- {}空对象

下面这样写不好：

```
while (x != null) {
```

你可以写成这种更短的代码（只要你不期望x为0、空字符串或者false）：

```
while (x) {
```

如果你想检查字符串是否为null或空，你可以这样写：

```
if (y != null && y != '') {
```

但是以下这样会更简练更好：

```
if (y) {
```

注意：还有很多不直观的关于布尔表达式的例子，这里是一些：

- Boolean('0') == true '0' != true
- 0 != null 0 == [] 0 == false
- Boolean(null) == false null != true null != false
- Boolean(undefined) == false undefined != true undefined != false
- Boolean([]) == true [] != true [] == false
- Boolean({}) == true {} != true {} != false

条件（三元）操作符（? : ）

以下这种写法可以三元操作符替换：

```
if (val != 0) {
  return foo();
} else {
  return bar();
}
```

你可以这样写来代替：

```
return val ? foo() : bar();
```

三元操作符在生成HTML代码时也是很有用的：

```
var html = '<input type="checkbox"' +
    (isChecked ? ' checked' : '') +
    (isEnabled ? '' : ' disabled') +
    ' name="foo">';
```

&& 和 ||

二元布尔操作符是可短路的,，只有在必要时才会计算到最后一项。

"||" 被称作为 'default' 操作符，因为可以这样：

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win;
  if (opt_win) {
    win = opt_win;
  } else {
    win = window;
  }
  // ...
}
```

你可以这样写：

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

"&&" 也可以用来缩减代码。例如，以下这种写法可以被缩减：

```
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```

你可以这样写：

```
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

或者这样写：

```
var kid = node && node.kids && node.kids[index];
  if (kid) {
    foo(kid);
}
```

然而以下这样写就有点过头了：

```
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

遍历节点列表

节点列表是通过给节点迭代器加一个过滤器来实现的。这表示获取他的属性，如length的时间复杂度为O(n)，通过length来遍历整个列表需要O(n^2)。

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

这样写更好：

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

这种方法对所有的集合和数组(只要数组不包含被认为是false值的元素) 都适用。

在上面的例子中，你也可以通过firstChild和nextSibling属性来遍历子节点。

```
var parentNode = document.getElementById('foo');
for (var child = parentNode.firstChild; child; child = child.nextSibling) {
  doSomething(child);
}
```