# Code Style Reference

**目录**

**第一部分 JAVA**

**1. 前言**

**2.** **异常处理**

**3. 注释/JavaDoc**

**4. 源文件基础**

**5. 源文件结构**

**6. 格式**

**第二部分 Node.js**

**1. JavaScript编程风格**

**2. Node.js编程规范**

**第一部分 JAVA**

**前言：**

为了统一团队项目编码格式，这份文档参考了Google Java编程风格规范的定义，给出在项目中的编码统一规范。

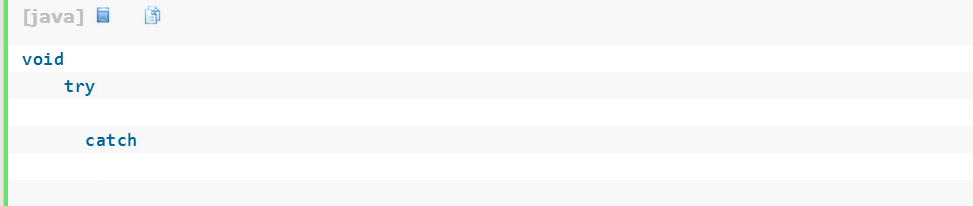
本文不仅仅是编码格式美不美观的问题，同时也讨论一些约定及编码标准。对于不是明确强制要求的，此处尽量避免提供意见。

本文档中的示例代码并不作为规范。也就是说，虽然示例代码是遵循Google编程风格，但并不意味着这是 展现这些代码的唯一方式。示例中的格式选择不应该被强制定为规则。

**异常处理：**

1. 不要忽视异常处理

如果像下面的代码这样，对catch后的异常作空处理，就会发生意想不到的后果



正确做法（1）：

在方法声明时抛出异常，由客户程序员去负责消化这个异常。



1. 不要偷懒而捕捉一般异常

下面代码一概捕捉Exception异常，大小通吃是不对的，这样会让你在错误出现时难以定位到错误原因，一般异常无法用统一方法进行异常处理。

错误做法：

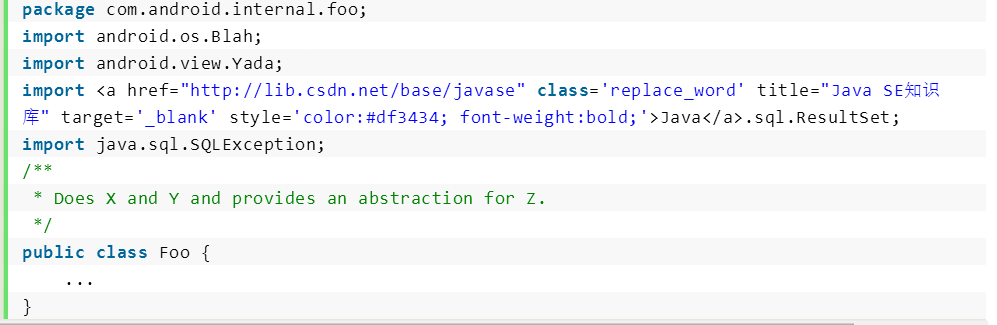


**注释/JavaDoc**

1）顶部版权声明

2）包和引入块（每一块以空白行分隔）

3）类或接口的声明。 在Javadoc注释，描述的类或接口的用途。



注：

类、接口的文档注释包含如下信息：

1.用途。 开发人员使用某个类/接口之前，需要知道采用该类/接口的用途。

2.如何使用。开发人员需要知道该类/接口应该如何使用，如果必要的话还需要注明不应该如何使用。

3.开发维护的日志。一个有关于该类/接口的维护记录：时间、作者、摘要。

方法注释的内容 ：

1.类该方法是做什么的 。

2.该方法如何工作。

3.代码修改历史纪录。

4.方法调用代码示范。

5.必须传入什么样的参数给这个方法。

6.异常处理。

7.这个方法返回什么。

**源文件基础**

1. 文件名：源文件以其最顶层的类名来命名，大小写敏感，文件扩展名为.java
2. 文件编码：源文件编码格式为UTF­8。
3. 特殊字符
4. 空白字符：

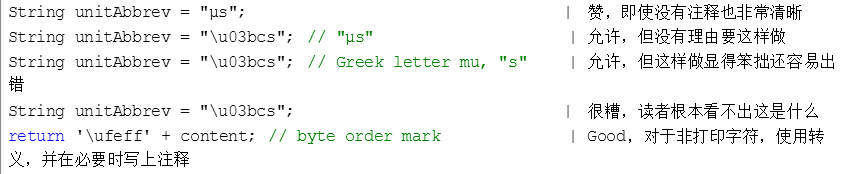
除了行结束符序列，ASCII水平空格字符(0x20，即空格)是源文件中唯一允许出现的空白字符，这意味着所有其它字符串中的空白字符都要进行转义，并且制表符不用于缩进。

1. 特殊转义序列：

对于具有特殊转义序列的任何字符(\b,\t,\n,\f,\r,\“,\‘及\)，我们使用它的转义序列，而不是相应的 八进制(比如\012)或Unicode(比如\u000a)转义。

1. 非ASCII字符：

对于剩余的非ASCII字符，是使用实际的Unicode字符(比如∞)，还是使用等价的Unicode转义符(比如 \u221e)，取决于哪个能让代码更易于阅读和理解。



**源文件结构**

一个源文件包含(按顺序地)：

1.许可证或版权信息(如有需要)

2.package语句

3.import语句

4.一个顶级类(只有一个)

以上每个部分之间用一个空行隔开。

1. 许可证或版权信息

如果一个文件包含许可证或版权信息，那么它应当被放在文件最前面。

1. package语句

package语句不换行，列限制(4.4节)并不适用于package语句。(即package语句写在一行里)

1. import语句
   1. import不要使用通配符

即，不要出现类似这样的import语句：import java.util.\*;

b）不要换行

import语句不换行，列限制(4.4节)并不适用于import语句。(每个import语句独立成行)

1. 顺序和间距

import语句可分为以下几组，按照这个顺序，每组由一个空行分隔

1.所有的静态导入独立成组

2.com.google imports(仅当这个源文件是在com.google包下)

3.第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun.

4.java imports

5.javax imports

组内不空行按字典序排列。

1. 类声明
   1. 只有一个顶级类声明：

每个顶级类都在一个与它同名的源文件中(当然，还包含.java后缀)。

* 1. 类成员顺序：

类的成员顺序对易学性有很大的影响，但这也不存在唯一的通用法则。不同的类对成员的排序可能是不同 的。 最重要的一点，每个类应该以某种逻辑去排序它的成员，维护者应该要能解释这种排序逻辑。比如， 新的方法不能总是习惯性地添加到类的结尾，因为这样就是按时间顺序而非某种逻辑来排序的。

* 1. 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间不要放进其它 函数/方法。

**格式**

术语说明：块状结构(block­like construct)指的是一个类，方法或构造函数的主体。需要注意的是，数 组初始化中的初始值可被选择性地视为块状结式

1.大括号

1）使用大括号(即使是可选的)

大括号与if, else, for, do, while语句一起使用，即使只有一条语句(或是空)，也应该把大括号写上。

1. 非空块：K&R风格

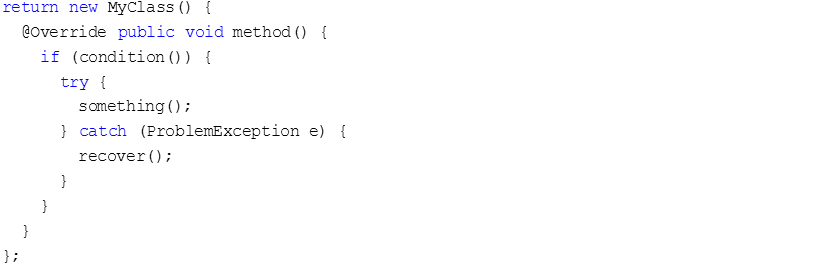
对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格 (Egyptian brackets):

左大括号前不换行

左大括号后换行

右大括号前换行

如果右大括号是一个语句、函数体或类的终止，则右大括号后换行; 否则不换行。例如，如果右大括 号后面是else或逗号，则不换行。



3）空块：可以用简洁版本

一个空的块状结构里什么也不包含，大括号可以简洁地写成{}，不需要换行。例外：如果它是一个多块语 句的一部分(if/else 或 try/catch/finally) ，即使大括号内没内容，右大括号也要换行。



2.块缩进：2个空格

每当开始一个新的块，缩进增加2个空格，当块结束时，缩进返回先前的缩进级别。缩进级别适用于代码和注释。

3.一行一个语句

每个语句后要换行。

4.列限制：80或100

一个项目可以选择一行80个字符或100个字符的列限制，除了下述例外，任何一行如果超过这个字符数限 制，必须自动换行。

例外：

1）不可能满足列限制的行(例如，Javadoc中的一个长URL，或是一个长的JSNI方法参考)。

2）package和import语句(见3.2节和3.3节)。

3）注释中那些可能被剪切并粘贴到shell中的命令行。

5.自动换行

术语说明：一般情况下，一行长代码为了避免超出列限制(80或100个字符)而被分为多行，我们称之为自 动换行(line­wrapping)。

我们并没有全面，确定性的准则来决定在每一种情况下如何自动换行。很多时候，对于同一段代码会有好 几种有效的自动换行方式。

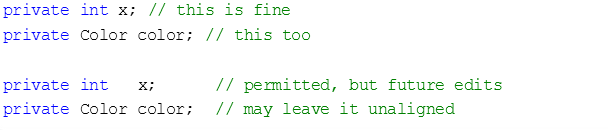
6.空白

1）垂直空白

2）水平空白

3）水平对齐

以下示例先展示未对齐的代码，然后是对齐的代码：



7. 用小括号来限定组：

除非作者和reviewer都认为去掉小括号也不会使代码被误解，或是去掉小括号能让代码更易于阅读，否则 我们不应该去掉小括号。 我们没有理由假设读者能记住整个Java运算符优先级表。

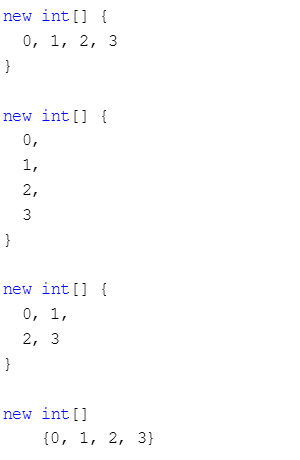
8.具体结构

1）枚举类



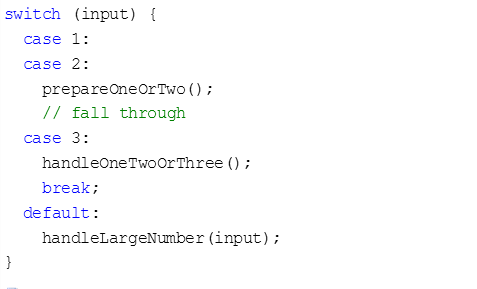
2）变量声明

3）数组

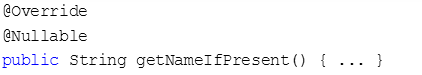


4）非C风格的数组声明

5）switch语句



6）注解



7）注释



8）Modifiers



**第二部分 Node.js**

1. JavaScript编程风格
   1. 概述

所谓"编程风格"（programming style），指的是编写代码的样式规则。不同的程序员，往往有不同的编程风格。

有人说，编译器的规范叫做"语法规则"（grammar），这是程序员必须遵守的；而编译器忽略的部分，就叫"编程风格"（programming style），这是程序员可以自由选择的。这种说法不完全正确，程序员固然可以自由选择编程风格，但是好的编程风格有助于写出质量更高、错误更少、更易于维护的程序。

所以，有一点应该明确，"编程风格"的选择不应该基于个人爱好、熟悉程度、打字工作量等因素，而要考虑如何尽量使代码清晰易读、减少出错。你选择的，不是你喜欢的风格，而是一种能够清晰表达你的意图的风格。这一点，对于Javascript这种语法自由度很高、设计不完全成熟的语言尤其重要。

1.2 大括号的位置

绝大多数的编程语言，都用大括号（{}）表示区块（block）。起首的大括号的位置，有许多不同的写法。

最流行的有两种。一种是起首的大括号另起一行：

　　block

　　{

　　　　...

　　}

另一种是起首的大括号跟在关键字的后面：

　　block {

　　　　...

　　}

一般来说，这两种写法都可以接受。但是，Javascript要使用后一种，因为Javascript会自动添加句末的分号，导致一些难以察觉的错误。

　　return

　　{

　　　　key:value;

　　};

上面的代码的原意，是要返回一个对象，但实际上返回的是undefined，因为Javascript自动在return语句后面添加了分号。为了避免这一类错误，需要写成下面这样：

　　return {

　　　　key : value;

　　};

因此，

规则1：表示区块起首的大括号，不要另起一行。

1.3圆括号

圆括号（parentheses）在Javascript中有两种作用，一种表示调用函数，另一种表示不同的值的组合（grouping）。我们可以用空格，区分这两种不同的括号。

　　规则2：调用函数的时候，函数名与左括号之间没有空格。

　　规则3：函数名与参数序列之间，没有空格。

　　规则4：所有其他语法元素与左括号之间，都有一个空格。

按照上面的规则，下面的写法都是不规范的：

　　foo (bar)

　　return(a+b);

　　if(a === 0) {...}

　　function foo (b) {...}

function(x) {...}

1.4 分号

分号表示语句的结束。大多数情况下，如果你省略了句尾的分号，Javascript会自动添加。

　　var a = 1

等同于

　　var a = 1;

因此，有人提倡省略句尾的分号。但麻烦的是，如果下一行的第一个字元（token）是下面这五个字符之一，Javascript将不对上一行句尾添加分号："("、"["、"/"、"+"和"-"。

　　x = y

　　(function (){

　　　　...

　　})();

上面的代码等同于

　　x = y(function (){...})();

因此，

规则5：不要省略句末的分号。

1.5 with语句

with可以减少代码的书写，但是会造成混淆。

　　with (o) {

　　　　foo = bar;

　　}

上面的代码，可以有四种运行结果：

　　o.foo = bar;

　　o.foo = o.bar;

　　foo = bar;

　　foo = o.bar;

这四种结果都可能发生，取决于不同的变量是否有定义。因此，

规则6：不要使用with语句。

1.6 语句的合并

有些程序员追求简洁，喜欢合并不同目的的语句。比如，原来的语句是

　　a = b;

　　if (a) {...}

他喜欢写成下面这样:

　　if (a = b) {...}

虽然语句少了一行，但是可读性大打折扣，而且会造成误读，让别人误以为这行代码的意思是：

　　if （a === b）{...}

另外一种情况是，有些程序员喜欢在同一行中赋值多个变量：

　　var a = b = 0;

他以为，这行代码等同于

　　var a = 0, b = 0;

实际上不是，它的真正效果是下面这样：

　　b = 0;

　　var a = b;

因此，

规则8：不要将不同目的的语句，合并成一行。

1.7 变量声明

Javascript会自动将变量声明"提升"（hoist）到代码块（block）的头部。

　　if (!o) {

　　　　var o = {};

　　}

等同于

　　var o;

　　if (!o) {

　　　　o = {};

　　}

为了避免可能出现的问题，不如把变量声明都放在代码块的头部。

　　for (var i ...) {...}

最好写成：

　　var i;

　　for (i ...) {...,}

因此，

　　规则9：所有变量声明都放在函数的头部。

规则10：所有函数都在使用之前定义。

1.8 new命令

Javascript使用new命令，从建构函数生成一个新对象。

　　var o = new myObject();

这种做法的问题是，一旦你忘了加上new，myObject()内部的this关键字就会指向全局对象，导致所有绑定在this上面的变量，都变成全部变量。

　　规则12：不要使用new命令，改用Object.create()命令。

如果不得不使用new，为了防止出错，最好在视觉上把建构函数与其他函数区分开来。

规则13：建构函数的函数名，采用首字母大写（InitialCap）；其他函数名，一律首字母小写。

1.9 自增和自减运算符

自增（++）和自减（--）运算符，放在变量的前面或后面，返回的值不一样，很容易发生错误。

事实上，所有的++运算符都可以用"+= 1"代替。

代码变得更清晰了。有一个很可笑的例子，某个Javascript函数库的源代码中出现了下面的片段：

　　++x;

　　++x;

这个程序员忘了，还有更简单、更合理的写法：

　　x += 2;

因此，

　　规则14：不要使用自增（++）和自减（--）运算符，用+=和-=代替。

2. Node.js编程规范

### 2个空格的缩进

使用2个空格进行缩进，永远也不要使用混合的tab和空格作为缩进。

### 换行

使用Unix风格的换行，每行结尾以(\n)结束，永远不要使用Windows的换行符（\r\n）。

### 无拖尾空白

永远也不要在一行后面留空格，在提交之前，你要像每顿饭刷牙一样清理你的JS文件。否则，腐烂的气味会驱走贡献者或同事。

### 使用分号

根据科学研究，分号的使用是我们社会的核心价值。考虑一下[反对派](http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding)的观点 ，但是我们需要传统，不要滥用纠错机制（省略分号）。

注\* 在JavaScript中[前置逗号代码风格](http://ourjs.com/detail/53f7ffabc1afbc6e3000000a)和省略分号一直存在争论，下同。

### 每行最多80个字符

每行最多80个字符。是的，屏幕在最近几年越来越大，但是你的脑子没怎么变，你可以使用多余的空间用来分屏。

### 使用单引号

只有在JSON文件中才使用双引号

Right:

var foo = 'bar';

Wrong:

var foo = "bar";

注\* 为什么？ JavaScript中包含双引号的字符串几乎到处都是，这样你就不需要转义了。

### 在同一行写大括号

Right:

if (true) {  
  console.log('winning');  
}

Wrong:

if (true)  
{  
  console.log('losing');  
}

同样，注意在条件前后都加个空格。

### 方法链（调用链）

如果你使用方法链，确保每行只调用一个方法。

同时你要合理使用缩进来表示他们的父对象是一致的。

Right:

User  
  .findOne({ name: 'foo' })  
  .populate('bar')  
  .exec(function(err, user) {  
    return true;  
  });

Wrong:

User  
.findOne({ name: 'foo' })  
.populate('bar')  
.exec(function(err, user) {  
  return true;  
});

User.findOne({ name: 'foo' })  
  .populate('bar')  
  .exec(function(err, user) {  
    return true;  
  });

User.findOne({ name: 'foo' }).populate('bar')  
.exec(function(err, user) {  
  return true;  
});

User.findOne({ name: 'foo' }).populate('bar')  
  .exec(function(err, user) {  
    return true;  
});

### 每行声明一个变量

每个var只声明一个变量，它可以更容易地重新排序。但是，并且变量应该在更有意义的地方声明。

Right:

var keys   = ['foo', 'bar'];  
var values = [23, 42];  
var object = {};

while (keys.length) {  
  var key = keys.pop();  
  object[key] = values.pop();  
}

Wrong:

var keys = ['foo', 'bar'],  
    values = [23, 42],  
    object = {},  
    key;

while (keys.length) {  
  key = keys.pop();  
  object[key] = values.pop();  
}

### 使用首字母小写给变量属性和函数命名

变量，属性和函数名应该使用lowerCamelCase（首字母小写）。他们也应该是描述性的。一般应避免单字符变量和不常见的缩写。

Right:

var adminUser = db.query('SELECT \* FROM users ...');

Wrong:

var admin\_user = db.query('SELECT \* FROM users ...');

### 类名首字母大写

类名的首字母应该是大写的

Right:

function BankAccount() {  
}

Wrong:

function bank\_Account() {  
}

### 常量大写

常量应该被声明为普通变量或静态类的属性，全部使用大写字母。

Node.js/V8实际上支持Mozilla的const的扩展，但遗憾的是不能用于类成员，也不是任何ECMA标准的一部分。

Right:

var SECOND = 1 \* 1000;

function File() {  
}

File.FULL\_PERMISSIONS = 0777;

Wrong:

const SECOND = 1 \* 1000;

function File() {  
}

File.fullPermissions = 0777;

### Object / Array 声明

使用尾随逗号，把短的声明在一行：

Right:

var a = ['hello', 'world'];  
var b = {  
  good: 'code',  
  'is generally': 'pretty'  
};

Wrong:

var a = [  
  'hello', 'world'  
];  
var b = {"good": 'code'  
        , "is generally": 'pretty'  
        };

### 使用 === 操作符

编写不应该只记事规则，还要学会使用。

Right:

var a = 0;  
if (a !== '') {  
  console.log('winning');  
}

Wrong:

var a = 0;  
if (a == '') {  
  console.log('losing');  
}

注\* === 即会判断类型，又会判断结果。

### 使用多行三元运算符

三元运算符不应该用在一行。分割成多行来代替。

Right:

var foo = (a === b)  
  ? 1  
  : 2;

Wrong:

var foo = (a === b) ? 1 : 2;

### 不要扩展内置对象

不要扩展原生JavaScript对象的原型。以后会后悔的。

Right:

var a = [];  
if (!a.length) {  
  console.log('winning');  
}

Wrong:

Array.prototype.empty = function() {  
  return !this.length;  
}

var a = [];  
if (a.empty()) {  
  console.log('losing');  
}

注\* 扩展String.prototype是比较常见的，如format, trim。

### 使用描述性的条件

任何判断条件应该分配给一个描述性命名的变量或函数：

Right:

var isValidPassword = password.length >= 4 && /^(?=.\*\d).{4,}$/.test(password);

if (isValidPassword) {  
  console.log('winning');  
}

Wrong:

if (password.length >= 4 && /^(?=.\*\d).{4,}$/.test(password)) {  
  console.log('losing');  
}

### 写小而短的函数

保持你的函数短一点。对一个大房间的最后一排的人民群众而言可以轻松读取幻灯片上的代码是比较合适的。不要指望他们有的视力保持每个函数〜15行代码。

### 早点从函数返回

为了避免if语句的深层嵌套，总是尽可能早地在函数返回值。

Right:

function isPercentage(val) {  
  if (val < 0) {  
    return false;  
  }

  if (val > 100) {  
    return false;  
  }

  return true;  
}

Wrong:

function isPercentage(val) {  
  if (val >= 0) {  
    if (val < 100) {  
      return true;  
    } else {  
      return false;  
    }  
  } else {  
    return false;  
  }  
}

### 命名您的闭包

随时给你闭包名称。这表明你关心他们，并能产生更好的堆栈跟踪，heap和cpu profiles。

Right:

req.on('end', function onEnd() {  
  console.log('winning');  
});

Wrong:

req.on('end', function() {  
  console.log('losing');  
});

### 不要嵌套闭包

使用闭包，但不能嵌套他们。否则，你的代码会变得一团糟。

Right:

setTimeout(function() {  
  client.connect(afterConnect);  
}, 1000);

function afterConnect() {  
  console.log('winning');  
}

Wrong:

setTimeout(function() {  
  client.connect(function() {  
    console.log('losing');  
  });  
}, 1000);

### 使用斜线注释

使用斜线为单行和多行注释。尝试从更高层次说明你代码的意图。不要重申琐碎的事情。

Right:

// 'ID\_SOMETHING=VALUE' -> ['ID\_SOMETHING=VALUE', 'SOMETHING', 'VALUE']  
var matches = item.match(/ID\_([^\n]+)=([^\n]+)/));

// This function has a nasty side effect where a failure to increment a  
// redis counter used for statistics will cause an exception. This needs  
// to be fixed in a later iteration.  
function loadUser(id, cb) {  
  // ...  
}

var isSessionValid = (session.expires < Date.now());  
if (isSessionValid) {  
  // ...  
}

Wrong:

// Execute a regex  
var matches = item.match(/ID\_([^\n]+)=([^\n]+)/));

// Usage: loadUser(5, function() { ... })  
function loadUser(id, cb) {  
  // ...  
}

// Check if the session is valid  
var isSessionValid = (session.expires < Date.now());  
// If the session is valid  
if (isSessionValid) {  
  // ...  
}

### 禁止使用Object.freeze, Object.preventExtensions, Object.seal, with, eval

这些功能很可笑，远离他们。