代码风格

本代码风格包括：HTML/CSS 代码风格，Node.js 代码风格，JAVA 代码风格，Python 代码风格

HTML/CSS 的代码风格

概述

本文档定义了 HTML/CSS 的编写格式和风格规则。它旨在提高合作和代码质量,并使其支持基础架构。适用于 HTML/CSS 文件，包括 GSS 文件。 只要代码质量是可以被维护的，就能很好的被工具混淆、压缩和合并。

html

样式规则

协议

嵌入式资源书写省略协议头

省略图像、媒体文件、样式表和脚本等 URL 协议头部声明 (http: , https:)。如果不是这两个声明的 URL 则不省略。

省略协议声明，使 URL 成相对地址，防止内容混淆问题和导致小文件重复下载。

```
<!-- 不推荐 -->
<script src="http://www.google.com/js/gweb/analytics/autotrack.js"></script>
<!-- 推荐 -->
<script src="//www.google.com/js/gweb/analytics/autotrack.js"></script>
/* 不推荐 */
.example {
  background: url(http://www.google.com/images/example);
}
/* 推荐 */
.example {
  background: url(//www.google.com/images/example);
}
```

排版规则

缩进

每次缩进两个空格。

大小写

只用小写字母。

行尾空格

删除行尾白空格。

元数据规则

编码

用不带 BOM 头的 UTF-8 编码。

让你的编辑器用没有字节顺序标记的 UTF-8 编码格式进行编写。

在 HTML 模板和文件中指定编码 <meta charset="utf-8">。 不需要制定样式表的编码，它默认为 UTF-8。

（更多有关于编码的信息和怎样指定它，请查看 Character Sets & Encodings in XHTML, HTML

and CSS。）
注释
尽可能的去解释你写的代码。
用注释来解释代码：它包括什么，它的目的是什么，它能做什么，为什么使用这个解决方案，还是说只是因为偏爱如此呢？
（本规则可选，没必要每份代码都描述的很充分，它会增重 HTML 和 CSS 的代码。这取决于该项目的复杂程度。）
活动的条目
用 TODO 标记代办事项和正活动的条目
只用 TODO 来强调代办事项， 不要用其他的常见格式，例如 @@ 。
附加联系人（用户名或电子邮件列表），用括号括起来，例如 TODO(contact) 。
可在冒号之后附加活动条目说明等，例如 TODO: 活动条目说明 。

```
{# TODO(cha.jn): 重新置中 #}
<center>Test</center>
<!-- TODO: 删除可选元素 -->
<ul>
  <li>Apples</li>
  <li>Oranges</li>
</ul>
```

HTML 代码风格规则
文档类型
请使用 HTML5 标准。
HTML 代码有效性
尽量使用有效的 HTML 代码。
编写有效的 HTML 代码，否则很难达到性能上的提升。
用类似这样的工具 W3C HTML validator 来进行测试。
HTML 代码有效性是重要的质量衡量标准，并可确保 HTML 代码可以正确使用。

```
<!-- 不推荐 -->
<title>Test</title>
<article>This is only a test.
<!-- 推荐 -->
<!DOCTYPE html>
<meta charset="utf-8">
<title>Test</title>
<article>This is only a test.</article>
```

语义
根据 HTML 各个元素的用途而去使用它们。
使用元素 (有时候错称其为"标签") 要知道为什么去使用它们和是否正确。例如，用 heading 元素构造标题， p 元素构造段落, a 元素构造锚点等。
根据 HTML 各个元素的用途而去使用是很重要的，它涉及到文档的可访问性、重用和代码效率等问题。

```
<!-- 不推荐 -->
<div onclick="goToRecommendations();">All recommendations</div>
<!-- 推荐 -->
```

All recommendations
多媒体后备方案
为多媒体提供备选内容。
对于多媒体，如图像，视频，通过 canvas 读取的动画元素，确保提供备选方案。 对于图像使用有意义的备选文案（ alt ） 对于视频和音频使用有效的副本和文案说明。
提供备选内容是很重要的，原因：给盲人用户以一些提示性的文字，用 @alt 告诉他这图像是关于什么的，给可能没理解视频或音频的内容的用户以提示。
（图像的 alt 属性会产生冗余，如果使用图像只是为了不能立即用 CSS 而装饰的 ，就不需要用备选文案了，可以写 alt="" 。）
<!-- 不推荐 -->
<img src="spreadsheet.png">
<!-- 推荐 -->
<img src="spreadsheet.png" alt="电子表格截图">
关注点分离
将表现和行为分开。
严格保持结构 （标记），表现 （样式），和行为 （脚本）分离，并尽量让这三者之间的交互保持最低限度。
确保文档和模板只包含 HTML 结构， 把所有表现都放到样式表里，把所有行为都放到脚本里。
此外，尽量使脚本和样式表在文档与模板中有最小接触面积，即减少外链。
将表现和行为分开维护是很重要滴，因为更改 HTML 文档结构和模板会比更新样式表和脚本更花费成本。
<!-- 不推荐 -->
<!DOCTYPE html>
<title>HTML sucks</title>
<link rel="stylesheet" href="base.css" media="screen">
<link rel="stylesheet" href="grid.css" media="screen">
<link rel="stylesheet" href="print.css" media="print">
<h1 style="font-size: 1em;">HTML sucks</h1>
<p>I’ve read about this on a few sites but now I’m sure:
<u>HTML is stupid!!1</u>
I can’t believe there’s no way to control the styling of
my website without doing everything all over again!
<!-- 推荐 -->
<!DOCTYPE html>
<title>My first CSS-only redesign</title>
<link rel="stylesheet" href="default.css">
<h1>My first CSS-only redesign</h1>
<p>I’ve read about this on a few sites but today I’m actually
doing it: separating concerns and avoiding anything in the HTML of
my website that is presentational.
<p>It’s awesome!
实体引用
不要用实体引用。

不需要使用类似 —、” 和 ☺ 等的实体引用，假定团队之间所用的文件和编辑器是同一编码（UTF-8）。
在 HTML 文档中具有特殊含义的字符（例如 < 和 &)为例外， 噢对了，还有 "不可见" 字符 （例如 no-break 空格）。
<!-- 不推荐 -->
欧元货币符号是 “&eur;”。
<!-- 推荐 -->
欧元货币符号是 “€”。
可选标签
省略可选标签（可选）。
出于优化文件大小和校验， 可以考虑省略可选标签，哪些是可选标签可以参考 HTML5 specification。
（这种方法可能需要更精准的规范来制定，众多的开发者对此的观点也都不同。考虑到一致性和简洁的原因，省略所有可选标记是有必要的。）
<!-- 不推荐 -->
```
<!DOCTYPE html>
<html>
  <head>
    <title>Spending money, spending bytes</title>
  </head>
  <body>
    <p>Sic.</p>
  </body>
</html>
```
<!-- 推荐 -->
```
<!DOCTYPE html>
<title>Saving money, saving bytes</title>
<p>Qed.
```
type 属性
在样式表和脚本的标签中忽略 type 属性
在样式表（除非不用 CSS）和脚本（除非不用 JavaScript）的标签中 不写 type 属性。
HTML5 默认 type 为 text/css 和 text/javascript 类型，所以没必要指定。即便是老浏览器也是支持的。
<!-- 不推荐 -->
```
<link rel="stylesheet" href="//www.google.com/css/maia.css"
  type="text/css">
```
<!-- 推荐 -->
```
<link rel="stylesheet" href="//www.google.com/css/maia.css">
```
<!-- 不推荐 -->
```
<script src="//www.google.com/js/gweb/analytics/autotrack.js"
  type="text/javascript"></script>
```
<!-- 推荐 -->
```
<script src="//www.google.com/js/gweb/analytics/autotrack.js"></script>
```
HTML 代码格式规则

每个块元素、列表元素或表格元素都独占一行，每个子元素都相对于父元素进行缩进。
独立元素的样式（as CSS allows elements to assume a different role per display property), 将块元素、列表元素或表格元素都放在新行。
另外，需要缩进块元素、列表元素或表格元素的子元素。
（如果出现了列表项左右空文本节点问题，可以试着将所有的 li 元素都放在一行。)

```
<blockquote>
  <p><em>Space</em>, the final frontier.</p>
</blockquote>
<ul>
  <li>Moe
  <li>Larry
  <li>Curly
</ul>
<table>
  <thead>
    <tr>
      <th scope="col">Income
      <th scope="col">Taxes
  <tbody>
    <tr>
      <td>$ 5.00
      <td>$ 4.50
</table>
```

NODE.JS 的代码风格

nodejs 并没有官方统一的编码风格，但是好的编码风格可以提高代码可读性。
Tabs 还是空格
引号
花括号
变量声明
变量和属性名称
类名
常量
创建对象/数组
判断相等操作
扩展原型
条件
函数代码行数
返回值
闭包命名
内嵌闭包
回调函数
Object.freeze, Object.preventExtensions, Object.seal, with, eval
Getters 和 setters 方法
没有正式的官方文档来规定 node.js 程序的代码风格.。这是一份我自己编写的规范，它可以帮助你编写出优雅的软件.

本指南假定你只针对 node.js 进行开发。如果你是编写基于浏览器或其他环境运行的 javascript 代码，请忽略.

请注意 node.js 以及它的各种包, 有其各自不同的风格. 如果你在为它们贡献代码，请遵从它们的规则.

Tabs 还是空格

让我们先开始一个信仰问题. 我们的 benevolent dictator 选择了 2 个空格缩进, 所以你最好还是照做.

引号

除非你在编写 JSON，否则使用单引号.

正确:

```
var foo = 'bar';
```

错误:

```
var foo = "bar";
```

花括号

按照下面的方式来写花括号.

正确:

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

错误:

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

另外在条件语句的前后加入一个空格。

变量声明

每行声明一个变量，每个变量后边添加分号，不要使用逗号连续声明变量。.

正确:

```
var keys = ['foo', 'bar'];
var values = [23, 42];
var object = {};
while (items.length) {
  var key = keys.pop();
  object[key] = values.pop();
}
```

错误:

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

```
        key;

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

变量和属性命名

使用小骆峰式命名法，避免使用单字符变量和缩写。

正确:

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

错误:

```
var admin_user = d.query('SELECT * FROM users ...');
```

类命名

类命名应采用大驼峰式命名法。

正确:

```
function BankAccount() {
}
```

错误:

```
function bank_Account() {
}
```

常量

全部使用大写，多个单词可以下划线连接.

正确:

```
var SECOND = 1 * 1000;

function File() {
}
File.FULL_PERMISSIONS = 0777;
```

错误:

```
const SECOND = 1 * 1000;

function File() {
```

```
}
File.fullPermissions = 0777;
```

创建对象/数组

每行声明一个短的变量，尾部使用逗号分开. 数组的键值使用引号:

正确:

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

错误:

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

判断相等操作

使用三等号===而不是双等号==，双等号会自动转换，出现意想不到的问题。

正确:

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

错误:

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

对象的扩展

不要扩展任何对象的属性，尤其是原生对象，如果不遵守这条规则的话会出现意想不到的问题。

正确:

```
var a = [];
```

```
if (!a.length) {
   console.log('winning');
}
```
错误:

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

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

任何有意义的条件都应该有一个描述的变量:

正确:

```
var isAuthorized = (user.isAdmin() || user.isModerator());
if (isAuthorized) {
   console.log('winning');
}
```
错误:

```
if (user.isAdmin() || user.isModerator()) {
   console.log('losing');
}
```
函数代码行数

让你的函数短小 。 一个良好的函数，适合在幻灯片上，在最后一排的一间大屋子的人可以舒适地阅读。最好每个函数限制在 10 行.

返回值

避免深度嵌套 if 语句，尽可能简单的返回函数值。

正确:

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

```
  if (val > 100) {
    return false;
  }

  return true;
}
```
错误:

```
function isPercentage(val) {
  if (val >= 0) {
    if (val < 100) {
      return true;
    } else {
      return false;
    }
  } else {
    return false;
  }
}
```
在这个特例中，还可以写的更短:

```
function isPercentage(val) {
  var isInRange = (val >= 0 && val <= 100);
  return isInRange;
}
```
闭包命名

给闭包一个命名，有助于代码调试:

正确:

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

```
req.on('end', function() {
  console.log('losing');
});
```
嵌套闭包

使用闭包，但是不能做嵌套.否则你的代码将是一团糟。

正确:

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

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

错误:

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

回调函数

由于 nodejs 都是非阻塞的 IO 操作，函数一般都是通过回调函数来返回他们的结果的。nodejs 采用回调函数的第一个对象作为错误对象，自己写回调函数的使用也应该遵守这条规则。

Object.freeze, Object.preventExtensions, Object.seal, with, eval

远离它们，你根本就不需要.

Getters 和 setters 方法

不要使用 setters，因为它会造成更多的问题，getters 可以任意使用.

JAVA 的代码风格

目录

前言
这份文档是 Google Java 编程风格规范的完整定义。当且仅当一个 Java 源文件符合此文档中的规则， 我们才认为它符合 Google 的 Java 编程风格。
与其它的编程风格指南一样，这里所讨论的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。然而，这份文档主要侧重于我们所普遍遵循的规则， 对于那些不是明确强制要求的，我们尽量避免提供意见。
1.1 术语说明
在本文档中，除非另有说明：
术语 class 可表示一个普通类，枚举类，接口或是 annotation 类型(@interface)
术语 comment 只用来指代实现的注释(implementation comments)，我们不使用“documentation comments”一词，而是用 Javadoc。
其他的术语说明会偶尔在后面的文档出现。
1.2 指南说明
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。
源文件基础
2.1 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。
2.2 文件编码：UTF-8
源文件编码格式为 UTF-8。
2.3 特殊字符
2.3.1 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。
制表符不用于缩进。
2.3.2 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。
2.3.3 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。
Tip: 在使用 Unicode 转义符或是一些实际的 Unicode 字符时，建议做些注释给出解释，这有助于别人阅读和理解。
例如：
String unitAbbrev = " μ s"; | 赞，即使没有注释也非常清晰
String unitAbbrev = "\u03bcs"; // " μ s" | 允许，但没有理由要这样做
String unitAbbrev = "\u03bcs"; // Greek letter mu, "s" | 允许，但这样做显得笨拙还容易出错
String unitAbbrev = "\u03bcs"; | 很糟，读者根本看不出这是什么
return '\ufeff' + content; // byte order mark | Good，对于非打印字符，使用转义，并在必要时写上注释
Tip: 永远不要由于害怕某些程序可能无法正确处理非 ASCII 字符而让你的代码可读性变差。当程序无法正确处理非 ASCII 字符时，它自然无法正确运行，你就会去 fix 这些问题的了。(言下之意就是大胆去用非 ASCII 字符，如果真的有需要的话)
源文件结构
一个源文件包含(按顺序地)：
许可证或版权信息(如有需要)
package 语句
import 语句
一个顶级类(只有一个)
以上每个部分之间用一个空行隔开。
3.1 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。
3.2 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)
3.3 import 语句
3.3.1 import 不要使用通配符
即，不要出现类似这样的 import 语句：import java.util.*;
3.3.2 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)
3.3.3 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：
所有的静态导入独立成组
com.google imports(仅当这个源文件是在 com.google 包下)
第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

java imports
javax imports
组内不空行，按字典序排列。
3.4 类声明
3.4.1 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)。
例外：package-info.java，该文件中可没有 package-info 类。
3.4.2 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。
3.4.2.1 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(block-like 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构(4.8.3.1 节)。
4.1 大括号
4.1.1 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。
4.1.2 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。
示例：

```
return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    }
  }
};
```

4.8.1 节给出了 enum 类的一些例外。
4.1.3 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。
示例：
void doNothing() {}
4.2 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。(见 4.1.2 节中的代码示例)
4.3 一行一个语句
每个语句后要换行。
4.4 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：
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)。
package 和 import 语句(见 3.2 节和 3.3 节)。
注释中那些可能被剪切并粘贴到 shell 中的命令行。
4.5 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(line-wrapping)。
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。
Tip: 提取方法或局部变量可以在不换行的情况下解决代码过长的问题(是合理缩短命名长度吧)
4.5.1 从哪里断开
自动换行的基本准则是：更倾向于在更高的语法级别处断开。
如果在非赋值运算符处断开，那么在该符号前断开(比如+，它将位于下一行)。注意：这一点与 Google 其它语言的编程风格不同(如 C++和 JavaScript)。 这条规则也适用于以下“类运算符”符号：点分隔符(.)，类型界限中的&（<T extends Foo & Bar>)，catch 块中的管道符号(catch (FooException | BarException e)
如果在赋值运算符处断开，通常的做法是在该符号后断开(比如=，它与前面的内容留在同一行)。这条规则也适用于 foreach 语句中的分号。
方法名或构造函数名与左括号留在同一行。
逗号(,)与其前面的内容留在同一行。
4.5.2 自动换行时缩进至少+4 个空格
自动换行时，第一行后的每一行至少比第一行多缩进 4 个空格(注意：制表符不用于缩进。见 2.3.1 节)。
当存在连续自动换行时，缩进可能会多缩进不只 4 个空格(语法元素存在多级时)。一般而言，两个连续行使用相同的缩进当且仅当它们开始于同级语法元素。
第 4.6.3 水平对齐一节中指出，不鼓励使用可变数目的空格来对齐前面行的符号。
4.6 空白
4.6.1 垂直空白
以下情况需要使用一个空行：
类内连续的成员之间：字段，构造函数，方法，嵌套类，静态初始化块，实例初始化块。

例外：两个连续字段之间的空行是可选的，用于字段的空行主要用来对字段进行逻辑分组。
在函数体内，语句的逻辑分组间使用空行。
类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。
要满足本文档中其他节的空行要求(比如 3.3 节：import 语句)
多个连续的空行是允许的，但没有必要这样做(我们也不鼓励这样做)。

4.6.2 水平空白

除了语言需求和其它规则，并且除了文字，注释和 Javadoc 用到单个空格，单个 ASCII 空格也出现在以下几个地方：
分隔任何保留字与紧随其后的左括号(()(如 if, for catch 等)。
分隔任何保留字与其前面的右大括号(})(如 else, catch)。
在任何左大括号前({)，两个例外：
@SomeAnnotation({a, b})(不使用空格)。
String[][] x = foo;(大括号间没有空格，见下面的 Note)。
在任何二元或三元运算符的两侧。这也适用于以下“类运算符”符号：
类型界限中的&(<T extends Foo & Bar>)。
catch 块中的管道符号(catch (FooException | BarException e)。
foreach 语句中的分号。
在, : ;及右括号())后
如果在一条语句后做注释，则双斜杠(//)两边都要空格。这里可以允许多个空格，但没有必要。
类型和变量之间：List list。
数组初始化中，大括号内的空格是可选的，即 new int[] {5, 6}和 new int[] { 5, 6 }都是可以的。
Note：这个规则并不要求或禁止一行的开关或结尾需要额外的空格，只对内部空格做要求。

4.6.3 水平对齐：不做要求

术语说明：水平对齐指的是通过增加可变数量的空格来使某一行的字符与上一行的相应字符对齐。
这是允许的(而且在不少地方可以看到这样的代码)，但 Google 编程风格对此不做要求。即使对于已经使用水平对齐的代码，我们也不需要去保持这种风格。
以下示例先展示未对齐的代码，然后是对齐的代码：

```
private int x; // this is fine
private Color color; // this too

private int      x;           // permitted, but future edits
private Color color;    // may leave it unaligned
```

Tip：对齐可增加代码可读性，但它为日后的维护带来问题。考虑未来某个时候，我们需要修改一堆对齐的代码中的一行。 这可能导致原本很漂亮的对齐代码变得错位。很可能它会提示你调整周围代码的空白来使这一堆代码重新水平对齐(比如程序员想保持这种水平对齐的风格)， 这就会让你做许多的无用功，增加了 reviewer 的工作并且可能导致更多的合并冲突。

4.7 用小括号来限定组：推荐

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。

4.8 具体结构

4.8.1 枚举类

枚举常量间用逗号隔开，换行可选。

没有方法和文档的枚举类可写成数组初始化的格式：

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

由于枚举类也是一个类，因此所有适用于其它类的格式规则也适用于枚举类。

4.8.2 变量声明

4.8.2.1 每次只声明一个变量

不要使用组合声明，比如 int a, b;。

4.8.2.2 需要时才声明，并尽快进行初始化

不要在一个代码块的开头把局部变量一次性都声明了(这是 C 语言的做法)，而是在第一次需要使用它时才声明。 局部变量在声明时最好就进行初始化，或者声明后尽快进行初始化。

4.8.3 数组

4.8.3.1 数组初始化：可写成块状结构

数组初始化可以写成块状结构，比如，下面的写法都是 OK 的：

```
new int[] {
   0, 1, 2, 3
}

new int[] {
   0,
   1,
   2,
   3
}

new int[] {
   0, 1,
   2, 3
}

new int[]
      {0, 1, 2, 3}
```

4.8.3.2 非 C 风格的数组声明

中括号是类型的一部分：String[] args， 而非 String args[]。

4.8.4 switch 语句

术语说明：switch 块的大括号内是一个或多个语句组。每个语句组包含一个或多个 switch 标签(case FOO:或 default:)，后面跟着一条或多条语句。

4.8.4.1 缩进

与其它块状结构一致，switch 块中的内容缩进为 2 个空格。
每个 switch 标签后新起一行，再缩进 2 个空格，写下一条或多条语句。
4.8.4.2 Fall-through：注释

在一个 switch 块内，每个语句组要么通过 break, continue, return 或抛出异常来终止，要么通过一条注释来说明程序将继续执行到下一个语句组， 任何能表达这个意思的注释都是 OK 的(典型的是用// fall through)。这个特殊的注释并不需要在最后一个语句组(一般是 default)中出现。示例：

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

4.8.4.3 default 的情况要写出来

每个 switch 语句都包含一个 default 语句组，即使它什么代码也不包含。
4.8.5 注解(Annotations)
注解紧跟在文档块后面，应用于类、方法和构造函数，一个注解独占一行。这些换行不属于自动换行(第 4.5 节，自动换行)，因此缩进级别不变。例如：

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

例外：单个的注解可以和签名的第一行出现在同一行。例如：

```
@Override public int hashCode() { ... }
```

应用于字段的注解紧随文档块出现，应用于字段的多个注解允许与字段出现在同一行。例如：

```
@Partial @Mock DataLoader loader;
```

参数和局部变量注解没有特定规则。
4.8.6 注释
4.8.6.1 块注释风格

块注释与其周围的代码在同一缩进级别。它们可以是/* ... */风格，也可以是// ...风格。对于多行的/* ... */注释，后续行必须从*开始， 并且与前一行的*对齐。以下示例注释都是 OK 的。

```
/*
 * This is          // And so           /* Or you can
 * okay.            // is this.         * even do this. */
 */
```

注释不要封闭在由星号或其它字符绘制的框架里。

Tip：在写多行注释时，如果你希望在必要时能重新换行(即注释像段落风格一样)，那么使用/* ... */。

4.8.7 Modifiers

类和成员的 modifiers 如果存在，则按 Java 语言规范中推荐的顺序出现。

public protected private abstract static final transient volatile synchronized native strictfp

命名约定

5.1 对所有标识符都通用的规则

标识符只能使用 ASCII 字母和数字，因此每个有效的标识符名称都能匹配正则表达式\w+。

在 Google 其它编程语言风格中使用的特殊前缀或后缀，如 name_, mName, s_name 和 kName，在 Java 编程风格中都不再使用。

5.2 标识符类型的规则

5.2.1 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

5.2.2 类名

类名都以 UpperCamelCase 风格编写。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。现在还没有特定的规则或行之有效的约定来命名注解类型。

测试类的命名以它要测试的类的名称开始，以 Test 结束。例如，HashTest 或 HashIntegrationTest。

5.2.3 方法名

方法名都以 lowerCamelCase 风格编写。

方法名通常是动词或动词短语。

下划线可能出现在 JUnit 测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：test<MethodUnderTest>_<state>，例如 testPop_emptyStack。 并不存在唯一正确的方式来命名测试方法。

5.2.4 常量名

常量名命名模式为 CONSTANT_CASE，全部字母大写，用下划线分隔单词。那，到底什么算是一个常量？

每个常量都是一个静态 final 字段，但不是所有静态 final 字段都是常量。在决定一个字段是否是一个常量时， 考虑它是否真的感觉像是一个常量。例如，如果任何一个该实例的观测状态是可变的，则它几乎肯定不会是一个常量。 只是永远不打算改变对象一般是不够的，它要真的一直不变才能将它示为常量。

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
```

static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
这些名字通常是名词或名词短语。

5.2.5 非常量字段名

非常量字段名以 lowerCamelCase 风格编写。
这些名字通常是名词或名词短语。

5.2.6 参数名

参数名以 lowerCamelCase 风格编写。
参数应该避免用单个字符命名。

5.2.7 局部变量名

局部变量名以 lowerCamelCase 风格编写，比起其它类型的名称，局部变量名可以有更为宽松的缩写。
虽然缩写更宽松，但还是要避免用单字符进行命名，除了临时变量和循环变量。
即使局部变量是 final 和不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

5.2.8 类型变量名

类型变量可用以下两种风格之一进行命名：
单个的大写字母，后面可以跟一个数字(如：E, T, X, T2)。
以类命名方式(5.2.2 节)，后面加个大写的 T(如：RequestT, FooBarT)。

5.3 驼峰式命名法(CamelCase)

驼峰式命名法分大驼峰式命名法(UpperCamelCase)和小驼峰式命名法(lowerCamelCase)。 有时，我们有不只一种合理的方式将一个英语词组转换成驼峰形式，如缩略语或不寻常的结构(例如"IPv6"或"iOS")。Google 指定了以下的转换方案。
名字从散文形式(prose form)开始:
把短语转换为纯 ASCII 码，并且移除任何单引号。例如："Müller’s algorithm"将变成"Muellers algorithm"。
把这个结果切分成单词，在空格或其它标点符号(通常是连字符)处分割开。
推荐：如果某个单词已经有了常用的驼峰表示形式，按它的组成将它分割开(如"AdWords"将分割成"ad words")。 需要注意的是"iOS"并不是一个真正的驼峰表示形式，因此该推荐对它并不适用。
现在将所有字母都小写(包括缩写)，然后将单词的第一个字母大写：
每个单词的第一个字母都大写，来得到大驼峰式命名。
除了第一个单词，每个单词的第一个字母都大写，来得到小驼峰式命名。
最后将所有的单词连接起来得到一个标识符。
示例：

Prose form	Correct	Incorrect
"XML HTTP request"	XmlHttpRequest	XMLHTTPRequest
"new customer ID"	newCustomerId	newCustomerID
"inner stopwatch"	innerStopwatch	innerStopWatch
"supports IPv6 on iOS?"	supportsIpv6OnIos	supportsIPv6OnIOS
"YouTube importer"	YouTubeImporter YoutubeImporter*	

加星号处表示可以，但不推荐。

Note：在英语中，某些带有连字符的单词形式不唯一。例如："nonempty"和"non-empty"都是正确的，因此方法名 checkNonempty 和 checkNonEmpty 也都是正确的。

编程实践

6.1 @Override：能用则用

只要是合法的，就把@Override 注解给用上。

6.2 捕获的异常：不能忽视

除了下面的例子，对捕获的异常不做响应是极少正确的。(典型的响应方式是打印日志，或者如果它被认为是不可能的，则把它当作一个 AssertionError 重新抛出。)

如果它确实是不需要在 catch 块中做任何响应，需要做注释加以说明(如下面的例子)。

```
try {
   int i = Integer.parseInt(response);
   return handleNumericResponse(i);
} catch (NumberFormatException ok) {
   // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

例外：在测试中，如果一个捕获的异常被命名为 expected，则它可以被不加注释地忽略。下面是一种非常常见的情形，用以确保所测试的方法会抛出一个期望中的异常， 因此在这里就没有必要加注释。

```
try {
   emptyStack.pop();
   fail();
} catch (NoSuchElementException expected) {
}
```

6.3 静态成员：使用类进行调用

使用类名调用静态的类成员，而不是具体某个对象或表达式。

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

6.4 Finalizers：禁用

极少会去重载 Object.finalize。

Tip：不要使用 finalize。如果你非要使用它，请先仔细阅读和理解 Effective Java 第 7 条款：“Avoid Finalizers”，然后不要使用它。

Javadoc

7.1 格式

7.1.1 一般形式

Javadoc 块的基本格式如下所示：

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

或者是以下单行形式：

/** An especially short bit of Javadoc. */
基本格式总是 OK 的。当整个 Javadoc 块能容纳于一行时(且没有 Javadoc 标记@XXX)，可以使用单行形式。

7.1.2 段落

空行(即，只包含最左侧星号的行)会出现在段落之间和 Javadoc 标记(@XXX)之前(如果有的话)。 除了第一个段落，每个段落第一个单词前都有标签<p>，并且它和第一个单词间没有空格。

7.1.3 Javadoc 标记

标准的 Javadoc 标记按以下顺序出现：@param, @return, @throws, @deprecated, 前面这 4 种标记如果出现，描述都不能为空。 当描述无法在一行中容纳，连续行需要至少再缩进 4 个空格。

7.2 摘要片段

每个类或成员的 Javadoc 以一个简短的摘要片段开始。这个片段是非常重要的，在某些情况下，它是唯一出现的文本，比如在类和方法索引中。

这只是一个小片段，可以是一个名词短语或动词短语，但不是一个完整的句子。它不会以 A {@code Foo} is a...或 This method returns...开头，它也不会是一个完整的祈使句，如 Save the record...。然而，由于开头大写及被加了标点，它看起来就像是个完整的句子。

Tip：一个常见的错误是把简单的 Javadoc 写成/** @return the customer ID */，这是不正确的。它应该写成/** Returns the customer ID. */。

7.3 哪里需要使用 Javadoc

至少在每个 public 类及它的每个 public 和 protected 成员处使用 Javadoc，以下是一些例外：

7.3.1 例外：不言自明的方法

对于简单明显的方法如 getFoo，Javadoc 是可选的(即，是可以不写的)。这种情况下除了写“Returns the foo”，确实也没有什么值得写了。

单元测试类中的测试方法可能是不言自明的最常见例子了，我们通常可以从这些方法的描述性命名中知道它是干什么的，因此不需要额外的文档说明。

Tip：如果有一些相关信息是需要读者了解的，那么以上的例外不应作为忽视这些信息的理由。例如，对于方法名 getCanonicalName， 就不应该忽视文档说明，因为读者很可能不知道词语 canonical name 指的是什么。

7.3.2 例外：重载

如果一个方法重载了超类中的方法，那么 Javadoc 并非必需的。

7.3.3 可选的 Javadoc

对于包外不可见的类和方法，如有需要，也是要使用 Javadoc 的。如果一个注释是用来定义一个类，方法，字段的整体目的或行为， 那么这个注释应该写成 Javadoc，这样更统一更友好。

Python 的代码风格

分号

Tip

不要在行尾加分号, 也不要用分号将两条命令放在同一行.
行长度

Tip

每行不超过 80 个字符
例外:

长的导入模块语句
注释里的 URL
不要使用反斜杠连接行.

Python 会将 圆括号, 中括号和花括号中的行隐式的连接起来 , 你可以利用这个特点. 如果需要, 你可以在表达式外围增加一对额外的圆括号.

```
Yes: foo_bar(self, width, height, color='black', design=None, x='foo',
                     emphasis=None, highlight=0)

        if (width == 0 and height == 0 and
              color == 'red' and emphasis == 'strong'):
```

如果一个文本字符串在一行放不下, 可以使用圆括号来实现隐式行连接:

```
x = ('This will build a very long long '
        'long long long long long long string')
```

在注释中，如果必要，将长的 URL 放在一行上。

```
Yes:    # See details at
          #
http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extens
ion_full_specification.html
No:    # See details at
        # http://www.example.com/us/developer/documentation/api/content/\
        # v2.0/csv_file_name_extension_full_specification.html
```

注意上面例子中的元素缩进; 你可以在本文的 缩进 部分找到解释.

括号

Tip

宁缺毋滥的使用括号
除非是用于实现行连接，否则不要在返回语句或条件语句中使用括号．不过在元组两边使用括号是可以的．

```
Yes: if foo:
        bar()
     while x:
        x = bar()
     if x and y:
        bar()
     if not x:
        bar()
     return foo
     for (x, y) in dict.items(): ...
No:  if (x):
        bar()
     if not(x):
        bar()
     return (foo)
```

缩进

Tip

用 4 个空格来缩进代码
绝对不要用 tab，也不要 tab 和空格混用．对于行连接的情况，你应该要么垂直对齐换行的元素(见 行长度 部分的示例)，或者使用 4 空格的悬挂式缩进(这时第一行不应该有参数)：

```
Yes:   # Aligned with opening delimiter
       foo = long_function_name(var_one, var_two,
                                var_three, var_four)

       # Aligned with opening delimiter in a dictionary
       foo = {
           long_dictionary_key: value1 +
                                value2,
           ...
       }

       # 4-space hanging indent; nothing on first line
       foo = long_function_name(
```

```
        var_one, var_two, var_three,
        var_four)

    # 4-space hanging indent in a dictionary
    foo = {
        long_dictionary_key:
            long_dictionary_value,
        ...
    }
No:  # Stuff on first line forbidden
    foo = long_function_name(var_one, var_two,
        var_three, var_four)

    # 2-space hanging indent forbidden
    foo = long_function_name(
      var_one, var_two, var_three,
      var_four)

    # No hanging indent in a dictionary
    foo = {
        long_dictionary_key:
            long_dictionary_value,
            ...
    }
```

空行

Tip

顶级定义之间空两行，方法定义之间空一行
顶级定义之间空两行，比如函数或者类定义．方法定义，类定义与第一个方法之间，都应该空一行．函数或方法中，某些地方要是你觉得合适，就空一行．

空格

Tip

按照标准的排版规范来使用标点两边的空格
括号内不要有空格．

```
Yes: spam(ham[1], {eggs: 2}, [])
No:   spam( ham[ 1 ], { eggs: 2 }, [ ] )
```

不要在逗号，分号，冒号前面加空格，但应该在它们后面加(除了在行尾).

```
Yes: if x == 4:
```

```
        print x, y
    x, y = y, x
No: if x == 4 :
        print x , y
    x , y = y , x
```

参数列表，索引或切片的左括号前不应加空格.

```
Yes: spam(1)
no: spam (1)
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

在二元操作符两边都加上一个空格，比如赋值(=)，比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not)，布尔(and, or, not). 至于算术操作符两边的空格该如何使用，需要你自己好好判断. 不过两侧务必要保持一致.

```
Yes: x == 1
No:  x<1
```

当’=’用于指示关键字参数或默认参数值时，不要在其两侧使用空格.

```
Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)
No:  def complex(real, imag = 0.0): return magic(r = real, i = imag)
```

不要用空格来垂直对齐多行间的标记，因为这会成为维护的负担(适用于:, #, =等):

```
Yes:
    foo = 1000  # comment
    long_name = 2  # comment that should not be aligned

    dictionary = {
        "foo": 1,
        "long_name": 2,
        }
No:
    foo       = 1000  # comment
    long_name = 2     # comment that should not be aligned

    dictionary = {
        "foo"     : 1,
        "long_name": 2,
        }
```

Shebang

Tip

大部分.py 文件不必以#!作为文件的开始. 根据 PEP-394，程序的 main 文件应该以

#!/usr/bin/python2 或者 #!/usr/bin/python3 开始.
(译者注: 在计算机科学中, Shebang (也称为 Hashbang)是一个由井号和叹号构成的字符串行(#!), 其出现在文本文件的第一行的前两个字符. 在文件中存在 Shebang 的情况下, 类 Unix 操作系统的程序载入器会分析 Shebang 后的内容, 将这些内容作为解释器指令, 并调用该指令, 并将载有 Shebang 的文件路径作为该解释器的参数. 例如, 以指令#!/bin/sh 开头的文件在执行时会实际调用/bin/sh 程序.)

#!先用于帮助内核找到 Python 解释器, 但是在导入模块时, 将会被忽略. 因此只有被直接执行的文件中才有必要加入#!.

注释

Tip

确保对模块, 函数, 方法和行内注释使用正确的风格
文档字符串

Python 有一种独一无二的的注释方式: 使用文档字符串. 文档字符串是包, 模块, 类或函数里的第一个语句. 这些字符串可以通过对象的__doc__成员被自动提取, 并且被 pydoc 所用. (你可以在你的模块上运行 pydoc 试一把, 看看它长什么样). 我们对文档字符串的惯例是使用三重双引号”“”(PEP-257). 一个文档字符串应该这样组织: 首先是一行以句号, 问号或惊叹号结尾的概述(或者该文档字符串单纯只有一行). 接着是一个空行. 接着是文档字符串剩下的部分, 它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.
模块

每个文件应该包含一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板.
函数和方法

下文所指的函数,包括函数, 方法, 以及生成器.

一个函数必须要有文档字符串, 除非它满足以下条件:

外部不可见
非常短小
简单明了
文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述”怎么做”, 除非是一些复杂的算法. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 对于复杂的代码, 在代码旁边加注释会比使用文档字符串更有意义.

关于函数的几个方面应该在特定的小节中进行描述记录， 这几个方面如下文所述. 每节应该以一个标题行开始. 标题行以冒号结尾. 除标题行外, 节的其他内容应被缩进 2 个空格.

Args:
列出每个参数的名字，并在名字后使用一个冒号和一个空格，分隔对该参数的描述.如果描述太长超过了单行 80 字符,使用 2 或者 4 个空格的悬挂缩进(与文件其他部分保持一致). 描述应该包括所需的类型和含义. 如果一个函数接受*foo(可变长度参数列表)或者**bar (任意关键字参数)，应该详细列出*foo 和**bar.
Returns: (或者 Yields: 用于生成器)
描述返回值的类型和语义. 如果函数返回 None，这一部分可以省略.
Raises:
列出与接口有关的所有异常.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

类

类应该在其定义下有一个用于描述该类的文档字符串. 如果你的类有公共属性(Attributes)，那么文档中应该有一个属性(Attributes)段. 并且应该遵守和函数参数相同的格式.

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

块注释和行注释

最需要写注释的是代码中那些技巧性的部分. 如果你在下次 代码审查 的时候必须解释一下, 那么你应该现在就给它写注释. 对于复杂的操作, 应该在其操作开始前写上若干行注释. 对于不是一目了然的代码, 应在其行尾添加注释.

```
# We use a weighted dictionary search to find out where i is in
# the array.    We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:              # true iff i is a power of 2
```

为了提高可读性, 注释应该至少离开代码 2 个空格.

另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂 Python, 他只是不知道你的代码要做什么.

```
# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

类

Tip

如果一个类不继承自其它类, 就显式的从 object 继承. 嵌套类也一样.

Yes: class SampleClass(object):

```
        pass


    class OuterClass(object):

        class InnerClass(object):
            pass


    class ChildClass(ParentClass):
        """Explicitly inherits from another class already."""
```
No:
```
class SampleClass:
        pass


class OuterClass:

        class InnerClass:
            pass
```
继承自 object 是为了使属性(properties)正常工作，并且这样可以保护你的代码，使其不受 PEP-3000 的一个特殊的潜在不兼容性影响. 这样做也定义了一些特殊的方法，这些方法实现了对象的默认语义，包括 __new__, __init__, __delattr__, __getattribute__, __setattr__, __hash__, __repr__, and __str__ .

字符串

Tip

即使参数都是字符串，使用%操作符或者格式化方法格式化字符串. 不过也不能一概而论，你需要在+和%之间好好判定.

Yes:
```
x = a + b
    x = '%s, %s!' % (imperative, expletive)
    x = '{}, {}!'.format(imperative, expletive)
    x = 'name: %s; score: %d' % (name, n)
    x = 'name: {}; score: {}'.format(name, n)
```
No:
```
x = '%s%s' % (a, b)    # use + in this case
x = '{}{}'.format(a, b)    # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```
避免在循环中用+和+=操作符来累加字符串. 由于字符串是不可变的，这样做会创建不必要的临时对象，并且导致二次方而不是线性的运行时间. 作为替代方案，你可以将每个子串加入列表，然后在循环结束后用 .join 连接列表. (也可以将每个子串写入一个 cStringIO.StringIO 缓存中.)

```
Yes: items = ['<table>']
        for last_name, first_name in employee_list:
            items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
        items.append('</table>')
        employee_table = ''.join(items)
No: employee_table = '<table>'
      for last_name, first_name in employee_list:
          employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
      employee_table += '</table>'
```

在同一个文件中，保持使用字符串引号的一致性. 使用单引号’或者双引号”之一用以引用字符串，并在同一文件中沿用. 在字符串内可以使用另外一种引号，以避免在字符串中使用. GPyLint 已经加入了这一检查.

(GPyLint 疑为笔误，应为 PyLint.)

```
Yes:
        Python('Why are you hiding your eyes?')
        Gollum("I'm scared of lint errors.")
        Narrator('"Good!" thought a happy Python reviewer.')
No:
        Python("Why are you hiding your eyes?")
        Gollum('The lint. It burns. It burns us.')
        Gollum("Always the great lint. Watching. Watching.")
```

为多行字符串使用三重双引号”“”而非三重单引号’‘’. 当且仅当项目中使用单引号’来引用字符串时，才可能会使用三重’‘’为非文档字符串的多行字符串来标识引用. 文档字符串必须使用三重双引号”“”. 不过要注意，通常用隐式行连接更清晰，因为多行字符串与程序其他部分的缩进方式不一致.

```
Yes:
      print ("This is much nicer.\n"
                "Do it this way.\n")
No:
        print """This is pretty ugly.
   Don't do this.
   """
```

文件和 sockets

Tip

在文件和 sockets 结束时，显式的关闭它.
除文件外, sockets 或其他类似文件的对象在没有必要的情况下打开，会有许多副作用，例如:

它们可能会消耗有限的系统资源，如文件描述符. 如果这些资源在使用后没有及时归还系统，那么用于处理这些对象的代码会将资源消耗殆尽.

持有文件将会阻止对于文件的其他诸如移动、删除之类的操作.
仅仅是从逻辑上关闭文件和 sockets，那么它们仍然可能会被其共享的程序在无意中进行读或者写操作. 只有当它们真正被关闭后，对于它们尝试进行读或者写操作将会跑出异常，并使得问题快速显现出来.
而且，幻想当文件对象析构时，文件和 sockets 会自动关闭，试图将文件对象的生命周期和文件的状态绑定在一起的想法，都是不现实的. 因为有如下原因:

没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的 Python 实现采用不同的内存管理技术，比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长.
对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪，包含有全局变量等).
推荐使用 “with”语句 以管理文件:

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print line
```

对于不支持使用”with”语句的类似文件的对象,使用 contextlib.closing():

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front_page:
    for line in front_page:
        print line
```

Legacy AppEngine 中 Python 2.5 的代码如使用”with”语句，需要添加 “from __future__ import with_statement”.

TODO 注释

Tip

为临时代码使用 TODO 注释，它是一种短期解决方案. 不算完美，但够好了.
TODO 注释应该在所有开头处包含”TODO”字符串，紧跟着是用括号括起来的你的名字, email 地址或其它标识符. 然后是一个可选的冒号. 接着必须有一行注释，解释要做什么. 主要目的是为了有一个统一的 TODO 格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节). 写了 TODO 注释并不保证写的人会亲自解决问题. 当你写了一个 TODO，请注上你的名字.

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

如果你的 TODO 是”将来做某事”的形式，那么请确保你包含了一个指定的日期(“2009 年 11 月解决”)或者一个特定的事件(“等到所有的客户都可以处理 XML 请求就移除这些代码”).

导入格式

Tip

每个导入应该独占一行
Yes: import os
     import sys
No: import os, sys
导入总应该放在文件顶部，位于模块注释和文档字符串之后，模块全局变量和常量之前．导入应该按照从最通用到最不通用的顺序分组:

标准库导入
第三方库导入
应用程序指定导入
每种分组中，应该根据每个模块的完整包路径按字典序排序，忽略大小写.

import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
语句

Tip

通常每个语句应该独占一行
不过，如果测试结果与测试语句在一行放得下，你也可以将它们放在同一行．如果是 if 语句，只有在没有 else 时才能这样做．特别地，绝不要对 try/except 这样做，因为 try 和 except 不能放在同一行.

Yes:

```
if foo: bar(foo)
```
No:

```
if foo: bar(foo)
else:    baz(foo)

try:                    bar(foo)
except ValueError: baz(foo)

try:
    bar(foo)
except ValueError: baz(foo)
```
访问控制

Tip

在 Python 中，对于琐碎又不太重要的访问函数，你应该直接使用公有变量来取代它们，这样可以避免额外的函数调用开销．当添加更多功能时，你可以用属性(property)来保持语法的一致性．

(译者注：重视封装的面向对象程序员看到这个可能会很反感，因为他们一直被教育：所有成员变量都必须是私有的！其实，那真的是有点麻烦啊．试着去接受 Pythonic 哲学吧)
另一方面，如果访问更复杂，或者变量的访问开销很显著，那么你应该使用像 get_foo() 和 set_foo() 这样的函数调用．如果之前的代码行为允许通过属性(property)访问 ，那么就不要将新的访问函数与属性绑定．这样，任何试图通过老方法访问变量的代码就没法运行，使用者也就会意识到复杂性发生了变化．

命名

Tip

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_VAR_NAME, instance_var_name, function_parameter_name, local_var_name.
应该避免的名称

单字符名称，除了计数器和迭代器．
包/模块名中的连字符(-)
双下划线开头并结尾的名称(Python 保留，例如__init__)
命名约定

所谓”内部(Internal)”表示仅模块内可用，或者，在类内是保护或私有的．
用单下划线(_)开头表示模块变量或函数是 protected 的(使用 import * from 时不会包含)．
用双下划线(__)开头的实例变量或方法表示类内私有．
将相关的类和顶级函数放在同一个模块里．不像 Java，没必要限制一个类一个模块．
对类名使用大写字母开头的单词(如 CapWords，即 Pascal 风格)，但是模块名应该用小写加下划线的方式(如 lower_with_under.py)．尽管已经有很多现存的模块使用类似于 CapWords.py 这样的命名，但现在已经不鼓励这样做，因为如果模块名碰巧和类名一致，这会让人困扰．
Python 之父 Guido 推荐的规范

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under

Instance Variables lower_with_under _lower_with_under (protected) or __lower_with_under (private)
Method Names lower_with_under() _lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters lower_with_under
Local Variableslower_with_under
Main

Tip

即使是一个打算被用作脚本的文件，也应该是可导入的．并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行，这是一种副作用．主功能应该放在一个 main()函数中．在 Python 中, pydoc 以及单元测试要求模块必须是可导入的．你的代码应该在执行主程序前总是检查 if __name__ == '__main__' ，这样当模块被导入时主程序就不会被执行．

```
def main():
        ...

if __name__ == '__main__':
    main()
```

所有的顶级代码在模块导入时都会被执行．要小心不要去调用函数，创建对象，或者执行那些不应该在使用 pydoc 时执行的操作．

Next Previous