**1 简介**

本文提供完整的Google Java 代码规范，Google风格的Java代码必须严格遵循此文档。  
像其他代码规范一样，本规范包括代码美观和一些标准约定。但是，我们更关注一些确定的规则，避免给出一些不明确的建议（无论对于人还是工具）。

**1.1 术语说明**

在本文档中，除非特殊声明：

1. 术语 *类* 包括，普通类，枚举类，接口和注解。
2. 术语 *成员* 包括内部类，字段，方法，构造方法，即除注解以外的所有顶级内容。
3. 术语 *注释* 通常是指实现的注解。我们一般不使用“文档注释”的说法而使用术语“Javadoc”  
   其他文档说明会在文中需要的地方出现。

**1.2 规范说明**

文中示例代码是非规范的。也就是说，即使示例代码符合Google规范，但不代表这是Google唯一的规范。示例中的可选规范不应被当众强制规则。

**2 源文件基础**

**2.1 文件名**

源文件名由源文件中区分大小写的顶级类名加.java后缀构成

**2.2 编码格式：UTF-8**

源文件采用UTF-8编码

**2.3 特殊字符**

**2.3.1 空格字符**

除了行终止符外，在源文件中ASCII空格字符（0x20）是唯一可以再任何地方都可能出现的空格字符  
这意味着：

1. 所有其他空格字符都是经过转义的。
2. 制表符不用于缩进。

**2.3.2 转义字符**

对于任何有特殊意义的转义序列应当用(\b,\t,\n,\f,\r,",'和\),而不是用八进制转义符(e.g.\012)或Unicode转义符(e.g.\u000a)

**2.3.3 非ASCII字符**

对于非ASCII字符，采用Unicode字符(e.g.∞)或Unicode转义符(e.g.\u221e)。这取决于那种方式更易于理解，建议不要才有Unicode转义符，即使有注释。

对于Unicode转义符，甚至是Unicode字符，注释会对其解释很有帮助。

示例

| **Example** | **Discussion** |
| --- | --- |
| String unitAbbrev = "μs"; | 最优解：清晰明了甚至不需要注释 |
| String unitAbbrev = "\u03bcs"; // "μs" | 允许：但是没必要这么做 |
| String unitAbbrev = "\u03bcs"; // 希腊字母mu，"s" | 允许：但是笨拙且易出错 |
| String unitAbbrev = "\u03bcs"; | 不建议：不易理解 |
| return '\ufeff' + content; // 字节顺序标记 | 良好：对于不会打印出的字符在需要的时候提供了注释 |

不要担心一些程序不能处理非ASCII字符而使得你的程序不可读，如果发生这种情况，程序将会被破坏，会有人修复的。

**3 源文件结构**

源文件按顺序包括：

1. 许可或版权信息（如果有的话）
2. 包声明
3. 导入声明
4. 一个顶级类

用一个空行分割每个部分。

**3.1 许可或版权信息（如果有的话）**

如果源文件中有许可或版权信息，那么它应当在这里（源文件头部）

**3.2 包声明**

包的声明是不换行的，每行的字数限制不包括包声明

**3.3 导入声明**

**3.3.1 不使用通配符**

无论是否是静态，都不适用通配符导入引用

**3.3.2 不换行**

导入声明是不换行的，每行的字数限制不包括导入声明

**3.3.3 顺序和间隔**

导入遵循以下顺序：

1. 所有静态引用在单独的一块儿
2. 所有的非静态引用在单独的一块儿

如果既有静态又有非静态，有一个空行。导入声明中只有这一个空行。  
每个导入块儿中，都是按导入名字的ASCII顺序排序的。

**3.3.4 不引入静态类**

静态导入不导入内部类，只导入普通类。

**3.4 类声明**

**3.4.1 只有一个顶级类**

每一个顶级类都写在它的源文件中

**3.4.2 类中内容的顺序**

类成员和初始化顺序会对代码的可读性造成影响.但是关于类成员排序的原则并没有一个统一的定论，不同的类肯能更适合不同的排序。  
每个类都用一些逻辑顺序在维护者给他人解释这个类时很重要。例如，新的方法不要按照习惯依照时间加到类的最后边，那不是逻辑顺序。

**3.4.2.1 重载：永远放在一起**

当一个类有多个构造器或多个同名方法时，它们应该依次放在一起，中间没有其他代码，甚至是私有成员。

**4 格式化**

术语小贴士：块状构造是指类，方法或构造函数的主体。注意，根据第4.8.3.1节关于数组初始化程序的介绍，可以选择将任何数组初始化程序视为类似于块的构造。

**4.1 大括号**

**4.1.1 大括号用于可选的地方**

大括号和if,else,for,do还有while语句一起使用，即使代码块儿为空或只有一行。

**4.1.2 非空块儿：K & R风格**

在非空块儿和块状构造中，大括号准守Kernighan和Ritchie风格（“埃及风格”）：

* 左括号前不换行
* 左括号后换行
* 右括号前换行
* 仅在大括号终止语句或终止方法，构造函数或命名类的主体时，才在右括号后换行。例如，如果大括号后跟着else或逗号，则没有换行符。

例如：

return () -> {

while (condition()) {

method();

}

};

return new MyClass() {

@Override public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

} else if (otherCondition()) {

somethingElse();

} else {

lastThing();

}

}

};

枚举类有一些例外情况，在4.8.1枚举

**4.1.3 空块儿：可以简介一些**

一个空的块或构造块儿可以按K&R风格（就像4.1.2那样）也可以把左右括号紧挨着，中间没有字符或空行，除非这时多个块中的一部分（一个直观的例子就是代码块儿：if/else或try/catch/finally）。  
例如：

// 这时被允许的

void doNothing() {}

// 这也是被允许的

void doNothingElse() {

}

// 这是不允许的：在多块儿代码中不是简洁代码

try {

doSomething();

} catch (Exception e) {}

**4.2 块儿缩进：两个空格**

每个新块儿或块状构造中，都要缩进两个空格。当块结束时，缩进返回到原来的状态。缩进适用于代码和注释。（参见4.1.2，非空代码块儿：K&R风格）

**4.3 每句代码占一行**

每句代码后面跟一个换行符。

**4.4 每行字数限制：100**

Java代码限制每行最多100个字符。一个字符就是任意的一个Unicode单元，除非另有说明，任何超过限制的代码都应该换行，就像4.5换行解释的那样。

每个Unicode单位算一个字符，不管其展示如何。例如，在全角模式下，你可能比半角模式更早的换行。

例外情况：

1. 不是每一行都必须遵守字数限制（例如，Javadoc中的长URL或长JSNI方法引用）
2. 包声明和导入声明（详见，3.2包声明和3.3导入声明）
3. 换行会导致不在注释区的注释

**4.5 换行**

术语小贴士：将一行代买划分为多行的行为，叫做换行。  
没有明确的，固定的方式解决所有情况下的换行。通常一行代码有几种不同的方式。

小贴士：除了避免超过字数限制，作者也可以自己根据实际情况决定是否换行

提醒：提取方法或本地变量可以在不换行的情况下解决问题

**4.5.1 在哪里换行**

换行的基本原则是：在高级语法上换行。即：

1. 在非运算符处换行，折行发生在符号前面。（注意这条原则与其他语言的Google风格不同，例如C++和JavaScript。）

* 以下操作同样属于符号的范围：
  + 句号（.）
  + 方法引用中的两个冒号（::）
  + 类型绑定符中的&（<T扩展Foo＆Bar>）
  + catch块中的|（catch（FooException | BarException e））

1. 在赋值运算符处换行时，换行通常在符号之后，但是其他方式也可以接受

* 这同时适用于类似的赋值运算符for循环中的冒号

1. 在方法或构造器名称后的左括号后（(）
2. 逗号始终放在前面的语句后面（逗号前不换行）
3. lambda表达式中的箭头不会被折断，但如果箭头如果是单行命令的话，箭头后悔折行

MyLambda<String, Long, Object> lambda =

(String label, Long value, Object obj) -> {

...

};

Predicate<String> predicate = str ->

longExpressionInvolving(str);

小贴士：换行的主要目的是为了有一个更整洁的代码，不一定要追求更少的代码行数

**4.5.2 缩进连续至少4个空格**

当发生换行的时候，下一行必须必原来的一行缩进至少4个空格。  
当有多行换行时，缩进可以调整超过4个空格，通常，两条连续在且仅当它们在句法上平行行才使用相同的缩进级别。

**4.6 空格**

**4.6.1 空行**

空行总是出现在以下地方：

1. 在类中连续的成员或初始化器之间：字段，构造器，方法，内部类，静态初始化器，和实例初始化器。

* 例外情况：两个连续字段之间肯能会有空行。空行用来给逻辑分组。
* 例外情况：枚举之间的空行参见4.8.1

1. 本文描述的其他情况（像第3节源文件结构和第3.3节导入声明）

只要对可读性有帮助，都可以使用一个空行。比如一个代码块中的逻辑子块儿。不鼓励在类的第一个成员或初始化器之前或最后一个成员或构造器之后添加空行。  
运行有连续多个空行，但不鼓励这么做。

**4.6.2 横向空格**

除了语言或其他样式规则的要求之外，除了文字，注释和Javadoc外，单个ASCII空间也仅出现在以下位置：

1. 将保留字例如if,for或catch与后面的左括号（(）分开。
2. 将保留字例如else或catch与前面的右括号（}）分开
3. 在所有左括号前（{），有两个例外：

* SomeAnnotation({a, b}) (不需要空格)
* String[][] x = {{"foo"}};(早{{之间不需要空格，参见第8条)

1. 在所有的二元或三元操作符两边，这条规则同样适用于以下情况：

* 类型绑定的连字符中：<T extend Foo & Bar>
* catch块中的用于区分不同异常的竖线：catch (FooException | BarException e)
* for循环中的冒号（:）
* lambda表达式中的箭头：(String str) -> str.length()

例外情况：

* 表示方法引用的两个冒号之间::，例如：Object::toString
* 分割点（.）,例如：object.toString()

1. 在,:;或者表示强制转型的（)）之后
2. 在句尾注释//两边，这里允许有多个空格，但不强制
3. 在声明类型和变量之间，例如：List<String> list
4. 在数组初始化的时候两个括号的里边允许添加括号

* new int[] {5, 6} 和 new int[] { 5, 6 }都是合法的。

1. 在类型声明和[]或...。

此规则并不要求或静止在一行的开头或结尾添加空格；此规则仅处理一行之间的空格

**4.6.3 水平对齐：不需要**

属于小贴士：水平对齐是通过添加若干空格，使上下几行的代码对齐。  
在Google风格中这种做法是允许的，但是不是必要的。甚至不需要在已经使用过的地方保持水平对齐。  
下面有一个没对齐的例子和一个对齐的例子：

private int x; // 这时规范的

private Color color; // 这也是规范的

private int x; // 允许，但以后可以被改写

private Color color; // 可能会使其不对齐

提示：对齐的目标是可读性，但会给未来的维护带来问题。考虑到未来的改动将会涉及到这一行

**4.7 括号分组：推荐**

只有在代码作者和review人确认代码不会被误解，切不会对可读性造成影响的情况下允许省略。假设每个阅读者都可以正确理解代码是不合理的。

**4.8 特殊结构体**

**4.8.1 枚举类**

每个枚举常量后面跟一个逗号，后面再换行。下面可以再加一个空行。例如：

private enum Answer {

YES {

@Override public String toString() {

return "yes";

}

},

NO,

MAYBE

}

对于没有方法和注释的枚举可以格式化成像数组初始化那样（参见数组初始化 4.8.3.1）

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

由于枚举也是类，类的规则同样适用于枚举

**4.8.2 变量声明**

**4.8.2.1 每次声明一个变量**

每个声明（字段或局部变量）只声明一个变量：例如int a, b;是不被允许的。  
例外：多变量声明在for语句中是可以接受的

**4.8.2.2 在需要的地方声明变量**

变量声明不应该习惯性的在代码块的头部声明，而是应该在靠近它第一次被使用的地方。变量声明要么附带赋值，要么马上赋值。

**4.8.3 数组**

**4.8.3.1 数组初始化：块儿状**

任何数组都可以初始化为块状的，例如下面的都是合法的：

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 不要采用C语言的风格声明数组**

方括号是类型的一部分，不是参数的一部分：String[] args而不是String args[]。

**4.8.4 Switch语句**

术语小贴士：以一个switch块的大括号中，有一个或多个状态组。每个组包含一个多个判断标签（像case FOO:或default:）,后面跟一个或多个语句（对于最后一个标签，则为0个或多个语句）。

**4.8.4.1 缩进**

和其他块一样，switch块缩进+2。

每个标签后面一个换行，并且像一个新的代码块儿一样缩进+2，下一个标签像块儿结束一样回到当前位置。

**4.8.4.2 穿透条件：添加注释**

在switch块内，，每个语句组要么终止判断（通过break,continue,return或抛出异常），要么添加注释，说明会执行到下一个条件中。任何能说明要执行到下一个语句的注释都是有效的（通常使用// fall through）。最后一个语句组不需要注释。例如：

switch (input) {

case 1:

case 2:

prepareOneOrTwo();

// fall through

case 3:

handleOneTwoOrThree();

break;

default:

handleLargeNumber(input);

}

提示case 1：后面不需要注释，只有语句组后面需要注释。

**4.8.4.3 提供default条件**

每个switch语句要包含一个default语句。即使后面没有代码。  
例外：如果包含了所有的枚举情况，枚举的switch语句可以省略default。IDE或者静态检查工具可以分析出是否漏掉了某些情况。

**4.8.5 注解**

注解应用于类，方法，构造器。在注释之后，每个注解单独占一行。这些换行不被算作折行，所以不需要缩进。例如：

@Override

@Nullable

public String getNameIfPresent() { ... }

例外：单行无参注解可能会和签名一同出现，例如：

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

注解也可以应用于字段，同样是在注释后面，但是多个注释可以放在同一行;例如：

@Partial @Mock DataLoader loader;

没有格式化参数，局部变量，或类型的特殊规定。

**4.8.6 注释**

本节主要讲普通注释。Javadoc在第7节单独讲。  
任何换行符前都可以加任意的空格然后添加普通注释，这中注释使这行成为非空白行。

**4.8.6.1 块儿注释风格**

块儿注释和它所注释的代码保持相同的缩进。他们可能是/\* ... \*/或// ...。对于多行/\* ... \*/注释后续行必须以\*开头并且与上一行的\*对齐。

/\*

\* This is

\* okay.

\*/

// And so

// is this.

/\* Or you can

\* even do this. \*/

注释不包含在有型号或其他字符的框中。

贴士：当写多行注释时，如果你想在注释自动换行时。使用/\* ... \*/。大多数格式化工具对// ...没有处理。

**4.8.7 修饰符**

如果有类和成员的修饰符，按Java语言的推荐顺序来写：

public protected private abstract default static final transient volatile synchronized native strictfp

**4.8.8 数字**

long类型的值用大写字母L，不要用小写l（避免和数字1混淆）,例如，使用3000000000L而不是3000000000l

**5 命名**

**5.1 所有标识的通用规则**

标识使用ASCII字母或数字并且仅在少量的情况下使用下划线。因此每个有效的表示都可以用正则表达式\w+匹配。  
在Google风格中，特殊的前缀和后缀是不允许的。例如，这些名字在Google风格中是非法的name\_,mName，s\_name和kName。

**5.2 标识类型的规则**

**5.2.1 包名**

包名采用全小写，通过连续的简单单词组合在一起（没有下划线），例如采用com.example.deepspace，而不是com.example.deepSpace或者com.exaple.deep\_space。

**5.2.2 类名**

类名采用大写驼峰UpperCamelCase 。  
类名通常是名称或名词短语。例如，Charater或ImmutabelList。接口名字  
也可能是名词或名词短语（例如，List），但有时也可能是形容词或形容词短语。  
注解的命名没有特殊的或者已经完备的规则。  
测试类命名以被测试类名开头，以Test结尾。例如,HashTest或者HashIntegrationTast。

**5.2.3 方法名**

方法名采用小写小写驼峰lowerCamelCase 。  
方法名通常是动词或动词短语。例如，sendMessage或stop 。  
下划线在JUnit测试方法中划分不同的逻辑部分，每个部分用小写的驼峰来写。通常采用<methodUnderTest>\_<state>，例如pop\_emptyStack。对于测试方法没唯一的正确名命方法。

**5.2.4 常量名**

常量名使用CONSTANT\_CASE：全大写字母，单词之间使用下划线分割。什么是常量呢？  
常量是静态的final字段，其值确定并且方法不能改变。包括，基本类型，字符串，不可变类型，以及不可变类型的集合。如果有一个实例可以改变，那这个实例不是常量。一般的变量即使不打算改变其值也不是常量。例如：

// 常量

static final int NUMBER = 5;

static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);

static final Joiner COMMA\_JOINER = Joiner.on(','); // because Joiner is immutable

static final SomeMutableType[] EMPTY\_ARRAY = {};

enum SomeEnum { ENUM\_CONSTANT }

// 非常量

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 ImmutableMap<String, SomeMutableType> mutableValues =

ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);

static final Logger logger = Logger.getLogger(MyClass.getName());

static final String[] nonEmptyArray = {"these", "can", "change"};

他们的名字通常都是名词或名词短语

**5.2.5 非常量名**

变量命名（静态或其他）采用小驼峰。  
这些名字通常是名词或名词短语，例如computeValues或index。

**5.2.6 参数名**

参数名采用小驼峰。  
在公有方法中应该避免一个字母的命名。

**5.2.7 局部变量名**

局部变量采用小驼峰  
即使是final并且不变的字段也不是变量，不应该使用变量的规则。

**5.2.8 泛型名**

泛型遵守以下规则之一：

* 单个大写字母，后面可以跟单个数字（例如E, T, X, T2）
* 以类的形式命名，后面跟上大写字母T(例如：RequestT,FooBarT)。

**5.3 驼峰：定义**

有时候，有多重将英语转换为驼峰写法的合理方法，例如类似于IPv6或iOS这种单词。为了确保唯一，Google风格做了一下统一方案：  
按一下顺序开始：

1. 将单词转化为ASCII码并且移除撇号。例如， "Müller's algorithm"转化为"Muellers algorithm".
2. 划分结果，将结果按空格和其他标点符号划分（通常是连字号）。

* 推荐：如果一个单词已经是具有驼峰结构，见其拆开（例如“AdWords”拆成“ad words”）。提示像“iOS”这种单词并不是驼峰写法，这时一种习惯写法，所以此推荐对其不适用

1. 将所有字母改为小写，然后将以下首字母改为大写：

* ...每个单词，生成大驼峰，或
* ...除第一个外的每个单词，生成小驼峰。

1. 最后将所有单词应用在定义中。

提示这种写法几乎忽略了原单词的大小写。例如：

| **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\* |  |

* 可以接受但不推荐

有些单词在英文中有无连字符都可以：例如“nonempty”和“not-empty”都是正确的，所以方法名“checkNonEmpty”和“checkNonempty”都是正确的。

**6 编码实践**

**6.1 @Override:必须添加**

方法在可以添加@Override的时候添加上他。这包括重写父类方法，实现接口方法。一个接口方法重新指定父接口方法。  
例外：当父方法被标记为@Deprecated时@Override可以被省略。

**6.2 捕获异常：不可忽略**

除以下情况外，不处理异常是不正确的。（典型的是打印日志，如果认为这种情况不可能出现，重新以AssertionError的形式抛出。）  
当在捕获块中确实应当什么都不做的话，原因应该写在注释里面。

try {

int i = Integer.parseInt(response);

return handleNumericResponse(i);

} catch (NumberFormatException ok) {

// 这不是数字，正常，继续执行

}

return handleTextResponse(response);

例外：在测试中，如果捕获的异常为预期异常或者以预期开通可以在没有注释的情况下省略。下面是确保测试代码抛出预期异常，所以注释在这里不是必要的。

try {

emptyStack.pop();

fail();

} catch (NoSuchElementException expected) {

}

**6.3 静态成员：通过类引用**

当使用一个静态的类成员，应当使用该类的名词而不是该类对象的引用。

Foo aFoo = ...;

Foo.aStaticMethod(); // 好

aFoo.aStaticMethod(); // 坏

somethingThatYieldsAFoo().aStaticMethod(); // 非常坏

**6.4 回收器：不要用**

极少重写Object.finalize。

提示：不要这么做，如果你必须这么做请先仔细阅读《Effective Java Item 7》,"Avoid Finalizers"。然后放弃这么做。

**7 Javadoc**

**7.1 格式化**

**7.1.1 一般格式**

在此示例中可以看到一般的JavaDoc块的格式：

/\*\*

\* Multiple lines of Javadoc text are written here,

\* wrapped normally...

\*/

public int method(String p1) { ... }

或者单行的例子

/\*\* An especially short bit of Javadoc. \*/

基本格式总是可以接受的。当JavaDoc可以在一行内放下时可以采用单行格式代替。注意，这仅适用于没有@return之类的标签的时候。

**7.1.2 段落**

段落之间以一个空行--仅在这行开头有一个星号\*，隔开。除第一个单词外，每个段落的第一个单词前面加上<p>后面没有空格。

**7.1.3 块标签**

所有的“块标签”按顺序出现@param,@return,@throws,@desprecate。这四种标签都不会出现空描述，当一行放不下的时候，下一行缩进到@后面四个或者更多空格的地方。

**7.2 摘要碎片**

每个JavaDoc都会以一个简短的摘要碎片开始。这个碎片非常重要：它是上下文的一部分文本，例如类和方法的索引。  
这是一个碎片，是一个名词或动词，而不是完整的句子，它不以A {@code Foo} is a...或者This method returns...开头，也不会形成完整的命令式句子Save the record.，但是该碎片被大写并且被标点，就像他是完整句子一样。

提示：一个常见的错误是简单JavaDoc被写成了这种格式/\*\* @return the customer ID \*/。这是不正确的，应当写成/\*\* Returns the customer ID. \*/。

**7.3 哪里需要Javadoc**

至少，在public的类中，每个public或protect成员需要JavaDoc，下面有少数特殊情况。  
如第7.3.4节“不需要的Javadoc”中所述，还可能存在其他Javadoc内容。

**7.3.1 例外：不言自明的方法**

Javadoc在一个简单明显的方法像getFoo。如果真的没有其他话，除了“返回foo”之外，没有什么好说的。

**7.3.2 例外：重写**

对于重写的方法，不必总是写Javadoc

**7.3.3 不需要的Javadoc**

当其他类或成员有需要的Javadoc  
当注释被用来描述类或方法的目的或行为时，注释应当写成Javadoc的形式（使用/\*\*）  
7.1.2和7.1.3中的注释中的非必要Javadoc可以省略，既是是建议的。