Google Java 风格指南

原文链接：https://google.github.io/styleguide/javaguide.html

目录

1 简介

本文档完整定义了谷歌 java 编程语言的代码规范。当且仅当一个 java 源文件中的代码遵循此规范才可被是 google 风格的代码。

与其他编程风格指南一样，这里的问题不仅仅覆盖编码格式的美学问题，同时也涉及一些类型的约定或者代码规范。然而，这篇文档首先重点关注的是人人都需要遵守的硬性规定，同时避免给出一些不够明显、可实施的建议（不论是人或者工具）。

1.1 术语说明

在本篇文档中，另行明确的除外：

1. 术语 class 被用作表示一个普通类，枚举类型，接口或者 annotation 类型（@interface）；
2. 术语 member （一个类中的）被用作表示一个内部类，字段，方法，构造器，即所有顶级类中除了初始化块及注释的内容；
3. 术语 comment 总是表示文档的注释，我们不使用“documentation comments”，而是使用普遍“javadoc”来称呼。

1.2 指南说明

本篇文档的示例代码不作为规范。也就是说，尽管这些示例遵循 Google 编码风格，但是并不是表示这些代码的唯一方式。这些例子的代码只是作为可选择的格式而不是强制执行的规则。

2 源文件基础

2.1 文件名

源文件名称是该文件所包含的区分大小写的顶级类的名称构成（每个源文件应该正好有一个这样的顶级类），扩展名为 .java。

2.2 文件编码：UTF-8

源文件的编码格式为 utf-8。

• UTF-8 编码是世界通用的语言编码，它可以同时对几乎所有地球上已知的文字字符进行书写和表示。

2.3 特殊字符

2.3.1 空白字符

除了换行符，ASCII 水平空格字符（0x20 即空格）是唯一允许出现在源文件任意位置的空白字符，这意味着：

1. 所有其他的空白字符或字符串都需要转义；
2. 制表符不用作缩进。

2.3.2 特殊转义序列

对具有一种特殊转义序列的字符（\b, \t, \n, \f, \r, \", \' 和 \\），我们使用它的序列而不是相应的八进制（如：\012）或 Unicode（如：\u000a）转义。

• 使用转义序列字符而不是使用相应的八进制，是因为相对于八进制转义字符而言，我们更容易记住他们转义序列的作用。

2.3.3 非 ASCII 字符

对于剩余的非 ASCII 字符，采用实际的 Unicode 字符（如：∞），还是采用等价的 Unicode 转义（如：\u221e），这取决于哪一种更容易阅读和理解，尽管强

烈不建议使用 Unicode 转义外部字符串常量和注释。

提示：在 Unicode 转义的情况下，有时甚至在实际使用 Unicode 字符时，一个说明注解将是很有帮助的。

例如：

Example	Discussion
String unitAbbrev = “μs”;	最好：完全清晰并且不需要注释
String unitAbbrev = “\u03bcs”; // “μs”	允许：但是没有理由这样做
String unitAbbrev = “\u03bcs”; // Greek letter mu, “s”	允许：看起来很尴尬并且容易犯错
String unitAbbrev = “\u03bcs”;	需要改善：读者不知道表示的是什么
return ‘\ufeff’ + content; // byte order mark	好的：使用转义字符表示空白字符，并且添加不可缺少的注解

3 源文件结构

一个源文件由以下部分组成，按顺序依次为：

1. 许可证或版权信息（如果存在）
2. 包声明
3. 导入声明
4. 当且仅当一个顶级类

用一个空行分隔每一个组成部分。

3.1 许可证或版权信息（如果存在）

如果一个文件中有许可证或版权信息，即在文件开头声明。

• 将许可证或版权信息放在文件开头，一个是因为这些信息被使用时容易查看，二是因为这些信息如果放在文件中和代码的逻辑无关，造成阅读代码的不便。

3.2 包声明

包声明不能换行，单行限制（4.4 中，单行限制最多 100 字符）不适用于包声明。

3.3 导入声明

3.3.1 导入不能含有通配符

不能使用通配符导入，不论是否为静态导入。

• 使用通配符导入时，会额外导入我们不需要的类或包。

3.3.2 不换行

一条导入声明不应该换行，单行限制（4.4 中，单行限制最多 100 字符）不适用于导入声明。

• 因为我们查看导入语句时通常是针对某一个类的导入，所以放在一行更容易阅读，且不会对逻辑代码的阅读有影响。

3.3.3 顺序和空行

导入应该按以下的顺序：

1. 所有的静态导入放在一个模块。
2. 所有的非静态导入放在一个模块。

如果导入声明中既有静态导入也有非静态导入，用一个空行分隔两个模块，导入声明中不应该有其他的空行。

每个模块按导入名称的 ASCII 顺序排列（越底层的包放在越前面，因为 ASCII 中”.”在”;”之前）。

• 我认为这样做没有必要，理由和上面一样，我们阅读代码时不会从上往下按顺序看完所有的导入语句，这样将静态导入和非静态导入分开，并且排序的行为对于代码的编写、阅读和维护来说是无意义的额外操作。

3.3.4 没有静态导入类

不能静态导入静态内部类，使用普通的导入导入静态内部类。

3.4 类声明

3.4.1 有且仅有一个顶级类声明

每一个顶级类存在于他所属的一个源文件中。

3.4.2 类成员的顺序

你采用的类成员和构造器的顺序对代码的理解性有很大的影响。然而，这里没有一个简单的统一的标准，不同的类它们的类成员有不同的排序方式。

每一个类使用一些逻辑顺序是重要的，这可以使我们更好的向维护者解释这个类当他们有疑问时。例如，我们不能仅仅将一个新方法放在一个类的最后，因为这会产生一个按添加时间排序的顺序，而不是逻辑顺序。

•我认为方法的位置顺序不需要做到与逻辑上的一致，因为如果我们这样做，在开发和维护的过程中，我们需要不断的确定各个方法的位置，在方法很多时就很耗费时间，但是这样做的结果对代码的阅读和维护并没有很大的帮助，因为在阅读和维护的时候，我们可以通过方法的名称精确定位方法的位置。

3.4.2.1 重载方法：不应该分开

当一个类有复杂的构造器和相同名字的复杂方法时，让它们按顺序展示，而不要让别的代码插入其中（包括私有化的成员）。

• 这样做代码更容易阅读和维护。

4 格式

术语说明：块状结构指的是类、方法或构造器的 body({}中的部分)。注意，4.8.3.1 项数组初始化中说到，任何数组初始化器也可以被看作块状结构。

4.1 花括号

4.1.1 花括号可以在任意地方被使用

花括号被用作 if，else，for，do while 语句中，甚至即使这些语句{}中为空或者只有一条简单声明。

4.1.2 非空块状结构采用 K&R 风格

（K&R 风格在处理括花括号时，使用了一种较为紧凑的格式，将左括号留在前一行的末尾，并尽可能地压缩...）。

对于非空的块状代码，花括号遵循 K&R 风格：

* “{”前不换行
* “{”后换行
* “}”前换行
* 只有“}”是用于表示结束一个语句块，或者结束一个方法体、构造器体，一个被命名的类体，“}”后换行。例如，当“}”紧跟着一个 else 或者一个逗号，则不换行。

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();
    }
  }
};
```

对于枚举类型有一些少许的例外情况，在 4.8.1 节枚举类型中会讲到。

- 这样做代码看起来更美观和整洁。

4.1.3 空代码块：可以更加简洁

一个空的代码块或者块状构造器可以采用 K&R 风格（像 4.1.2 节描述的一样）。可以在“{”之后换行，也可以在“{”之后直接接“}”，除非它是多个块语句的一部分（一个空白块直接包含多个块：if/else 或 try/catch/finally）。

例如：

//这是可以的

void doNothing() {}

```
//这也是可以的
void doNothingElse() {
}

//这是不可以的，多个语句块中没有一个简明的空语句块
try{
   doSomething();
}catch(Exception e) {}
```

4.2 块缩进：+2 个空格

在一个新的代码块或块状代码开始的时候，再缩进两个空格。当块状代码结束时，缩进回到之前的缩进级别。缩进的等级在整个代码块中的代码和注释中都适用（具体可以看 4.1.2 中的例子）。

• 这样做代码更容易阅读和维护，我们在修改的时候能够更清楚自己修改的是那一层代码，尤其是在嵌套有多层 if 或 for 语句的时候，缩进可以帮助代码更少犯错。

4.3 一行一句代码

每个语句后面跟一个换行。

• 这样做代码看起来更美观和整洁。

4.4 单行限制：100

每行 java 代码限制在 100 字符之内，一个“字符”表示任何一个 Unicode 代码。除了特别说明的，任何一行超过了这个限制都需要换行，具体会在 4.5 节换行解释。

任何一个 Unicode 代码都被记作为一个字符，即使它的宽度大于或小于一个字符。例如：如果使用 fullwidth 字符，则可以字符少于 100 的时候换行。

例外情况：

1. 遵守单行限制而不能实现其功能（例如：javadoc 中超长的 URL 地址， 或

者一个超长的 JSNI 方法的引用）；

2. 包声明或导入声明（详细说明参考 3.2 和 3.3 节）；
3. 一个注释中可能会被直接复制粘贴到 shell 执行的命令行。

•这样做代码在开发，阅读，维护的时候更方便，能够一眼看到所有的代码，不用滑动滑块查看。

4.5 换行

术语说明：当将一行合法的代码分离成多行，这种行为成为换行。

没有全面的、确定的准则来描述在任何情况下如何进行换行，因此经常会出现相同的一段有多张换行的方式。

注意：换行的重要的原因是为了避免超出单行的限制，甚至代码在列限制之中的换行也取决与作者本身的判断。

提示：将很长的代码提取为一个方法或者局部变量可能就不需要换行解决问题。

4.5.1 换行的位置

换行的重要原则是：在更高一级的语法层次上换行。也就是说：

1.当换行出现在非赋值语句时，在运算符号之前换行（注意这一点与谷歌风格的其他语言的用法不同，像 C++和 JavaScript）。

*这个原则同样适用于下面一些“运算符”：

*点分隔符（.）

*方法引用的两个冒号（::）

*在类型约束中的&符号（<T extends Foo & Bar>）

*在 catch 语句块中的|（catch (FooException | BarException e)）

• 这样更容易阅读，否则多行的情况下会将操作符与后面的操作数匹配错。

2.赋值语句换行通常是在赋值符号之后，但是无论是在赋值符号之前，还是之后，都可以接受。

*这也适用于增强 for 语句（“foreach”）中的“赋值运算符”。

3.方法或构造器的名称附在（（）之前，也就是说换行在左括号之后。

• 这样使代码更容易阅读，确定调用的是对象的属性还是方法。

4. 一个逗号附在它之前的语句上，也就是说在逗号之后换行。

• 因为逗号应该是和之前的语句匹配的，所以在逗号之后换行。

5.在一个 lambda 语句中的箭头不能换行，除非 lambda 语句体是一个无支撑的表达式，这样可以在箭头符号之后换行。例如：

```
MyLambda<String, Long, Object> lambda =
    (String label, Long value, Object obj) -> {
        ...
    };

Predicate<String> predicate = str ->
    longExpressionInvolving(str);
```

注意：换行的目的是为了有一个清晰的代码，不一定是代码符合一行上的最小的数量。

4.5.2 缩进续行至少 4 个空格

当换行时，每一行比原始行至少缩进 4 个空格。

当一段连续代码需要连续多次换行，缩进可以超过 4 个空格。总的来说，当且仅当两个延续行以语法元素断行时可以使用相同级别的缩进。

•与之前的换行规则区分开，使在阅读的时候很容易就知道是换行还是续行。

4.6.3 节介绍水平对齐中，解决了使用多个空格与之前行缩进对齐的问题。

4.6 空白空间

4.6.1 垂直空白

一个空白行出现的情况：

1.在一个类的连续成员或初始化之间：构造函数、方法、字段、嵌套类、静态初始化以及实例的初始化。

*例外：两个连续的成员变量之间是否有空白行是随意的（没有其他的代码在他们之间），这些空行被用作创建成员变量逻辑上的分组。

*例外：枚举常量之间的空行在 4.8.1 节说明。

2.代码之间，根据需要组成逻辑合理的代码。

3.类中的第一个成员变量或初始化之前，或者最后一个成员变量或初始化之后，可以随意选择。

4.本文档中其他部分介绍的需要空白行的情况。（例如：3.3 节的导入声明）

多个空白行是允许出现的，但是没有必要（或者说不鼓励）。

4.6.2 水平空白

在除了文字词语分隔、注释和 javadoc 等语法和风格规则外，一个 ASCII 空格只出现在以下地方。

1. 任何保留关键字与其后面的左括号（()之间加空格，像 if，for 或 catch 与。
2. 任何保留关键字与其后面的右花括号前加空格，像 if，for 或 catch。
3. 任何左花括号之前加空格，除了两种情况：

*@SomeAnnotation({a, b})；

*String[][] x = {{"foo"}}。

4. 在所有二元或三元运算符两边都需要加空格：这也适用下面这些运算符：

*当&符号用于连接类型的范围：<T extends Foo & Bar>

*当|符号用于分隔 catch 语句中多个异常：catch (FooException | BarException e)

*当:符号在强 for 循环（foreach）语句中

*当->符号在 lambda 表达式中：(String str) -> str.length()

但是不适用与下面情况：

*当::符号作为方法的引用时，这样应该写作：Object::toString

*当.作为方法的调用，这样应该写作：object.toString()

5. 逗号，冒号，分号或右括号之后需要加空格
6. 适用//双斜线注释时，双斜线的两边都需要加空格，这里也可以适用多个空格，但是没有必要。

• 双斜线注释时，若注释在代码的上方，左边应该不加空格，使注释能够与代码对齐，更加美观，在代码之后加空格分隔代码与注释，使代码更加清晰。

7. 在类型和变量之间：List<String> list。

8. 在数组初始化的时候有两种选择：

*new int[] {5, 6}和 new int[] { 5, 6 }两种写法都是可以的

这一原则不适用与一行的开头或结束时的空格，只针对行内部字符之间的分隔。

• 能够使代码更加的整洁、美观。

4.6.3 水平对齐（不作要求）

术语说明：水平对齐是指在代码中添加多个附加的空格，使本行的某一符号与前一行的某一符号对齐。

这种做法是允许的，但是这不是谷歌风格的要求，甚至在很多习惯的地方没有保持水平对齐的要求。

以下是没有水平对齐和有水平对齐的比较：

```
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
```

注意：水平对齐能够增加代码的可读性，但是增加了未来维护代码的难度。考虑到维护时只需要改变一行代码，之前的对齐可以不需要改动。为了对齐，你更有可能改了一行代码，同时需要更改附近的好几行代码，而这几行代码的改动，可能又会引起一些为了保持对齐的代码改动。那原本这行改动，我们称之为“爆炸半径”。这种改动，在最坏的情况下可能会导致大量的无意义的工作，即使在最好的情况下，也会影响版本历史信息，减慢代码 review 的速度，引起更多 merge 代码冲突的情况。

4.7 分组括号：推荐使用

当代码的开发人员和评审人员都觉得没有分组括号不会造成对代码的误解的时，并且不会使代码更容易理解，分组括号可以加也可以不加。我们没有理由认为所有阅读代码的人都能记住所有 java 运算符的优先级。

• 能够帮助代码更容易阅读。

4.8 特殊结构

4.8.1 枚举类

每一个逗号后面都要接一个一个枚举常量，是否换行不作要求，也可以在逗号之后加一个空白行。这是一种可能情况：

```
private enum Answer {
  YES {
    @Override public String toString() {
      return "yes";
    }
  },

  NO,
  MAYBE
}
```

一个没有方法没有 javadoc 的枚举类可以当作数组的初始化一样被初始化。

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

枚举类型也是一个类，其他类的规则同样适用于枚举类。

4.8.2 变量声明

4.8.2.1 一个变量一个声明

每个变量的声明只声明一个变量：不要使用像这种声明 int a, b;

除了：多个变量用一个变量声明在一个 for 循环的头部是可以接受的。

• 一个变量一个声明不仅使代码看起来更加整齐，同时也更容易阅读，方便后面维护。

4.8.2.2 当需要的时候才声明

局部变量不要习惯性的在包含它的代码块或者块状代码的一开始就声明，而是应该尽量靠近在其第一次使用的地方声明，以减小该变量的使用范围。局部变量在声明的时候就应该进行初始化或者在声名之后尽快初始化。

• 减少局部变量的作用域，避免资源的浪费；但是 for 循环中声明同一个类型的变量时可以放在循环之外，避免重复声明。

4.8.3 数组

4.8.3.1 数组初始化：可以作为块状代码来处理

任何数组的初始化，都可以当作是一个块状代码来进行处理。例如，下面这些都是有效的：

```
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 不能像 c 语言风格一样声明数组

方括号[]是类型的一部分，不是变量所以声明变量时是：String[] args， 不是 String args[]。

4.8.4 switch 语句

术语说明：switch 块是指花括号包含的一个或多个语句块，每一个语句块都是由一个或多个 switch 标签，后面跟着一条或多条语句组成。

4.8.4.1 缩进

和其他语句块一样，switch 块中的内容也需要缩进两个空格。

在 switch 标签之后,加上一个换行，并且在原来基础上再缩进两个，就像是一个块被打开，后面 switch 标签关闭的时候，返回到以前的缩进级别，仿佛这块已经关闭了。

• 缩进可以使代码看起来更加的工整有序，方便阅读。

4.8.4.2 注释：继续向下执行

在 switch 块中，每个语句组的需要通过 break, continue, return 或者抛出一个异常结束，否则需要通过一个注释来声明，代码可能继续需要执行到下一个语句组。任何能够说明继续往下执行的说明注释都可以（通常都是使用：// fall through），这个特别的注释不需要添加在最后一个 switch 语句组中。例如：

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

•注释需要向下执行的代码是很有必要的，因为 switch 语句块中的代码有很多，而且格式相似。如果在需要向下执行的地方不使用注释，会使阅读起来产生误解，给维护造成不便。

4.8.4.3 default 标签需要显示的声明

每一个 switch 语句都包括一个 default 语句组，即使这个语句组里没有包含代码。

除了下面情况之外：一个枚举类型的 switch 语句可以省略 default 语句组，如果它所有的标签覆盖了所有枚举类型的值。如果没有覆盖所有的值，IDEs 或其他静态分析工具都会发出警告。

4.8.5 Annotations

Annotations 应用于一个类、方法、构造器时，这个类、方法、构造器应紧接在这个代码块之后。每个 annotation 都单独放在一行上（即一行只有一个 annotation），这里的换行不属于续行，所以不需要增加缩进的级别。例如：

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

例外情况：一个无参的 annotation 可以和类或方法名一起出现在第一行。像这样：

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

Annotations 应用于一个字段时，同样字段紧跟在这个代码块之后，但是在这种情况，多个无参的 annotation 也同样可以和字段出现在同一行；例如：

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

这里没有特别的规则针对于 Annotations 应用在参数，局部变量或类型的情况。

• 我认为使用 Annotations 时，无论 annotation 是否有参数都应该放在单独的一行，这样做一个是使代码更加的整齐美观，同时也就方便阅读代码。而且我们只要知道但凡 annotation 就要单独占一行，不用区分有参无参，也不用区分是类名、方法名还是字段。更利于代码风格的统一。

4.8.6 注释

本节主要讨论注释的使用。Javadoc 注释在第 7 节 Javadoc 独立介绍.
任何一次换行后先加上空白字符再跟上注释的内容。就像是个注释使这一行没有成为空白行。

• 注释使用这样统一的格式可以使代码整洁美观。

4.8.6.1 注释块的风格

注释块的缩进级别和它所注释的代码的缩进级别相同。可以采用/*...*/或//...格式注释。使用多行注释/*...*/时，每一行都应该以*开始，并且*后面的内容应该前后对齐。

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

注释不会被*或其他的字符封闭起来。

提示：当在写多行注释的时候，如果你觉得注释内容自动对齐是必要的，那么使用/*...*/格式注释。//...注释一般不会自动对齐。

• 虽然上述的多行注释有三种写法，但是统一使用第一种注释可以使代码看起来更加的整齐。

4.8.7 修饰符

使用类和成言修饰符时，推荐出现的顺序按照 java 语言规范中定义的顺序：public protected private abstract default static final transient volatile synchronized native strictfp。

4.8.8 数值常量

长整数常量值使用大写字母 L 后缀，没有小写（避免与数字 1 混淆）。例如，3000000000l 而不是 3000000000l。

5 命名

5.1 所有标识符的一般规则

标识符只使用 ASCII 的字母和数字，少数情况下会有下划线。因此每一个合法的标识符都可以使用正则表达式匹配 \w+ 。在谷歌风格中，特殊的前缀或者后缀，如这些例子中的 name_，mName，s_name 和 kname 不能被用作标识符。

• 这样可以是代码看起来更加整洁，并且使用特殊名字容易造成很多误解。

5.2 不同类型的标识符规范

5.2.1 包命名

包名称全部使用小写字母，若有多个词语组合则简单的连接在一起（不加下划线），例如：com.example.deepspace,而不是 com.example.deepSpace 或 com.example.deep_space.。

5.2.2 类名

类名应该写成 UpperCamelCase（采用大写字母开头，大小写字母间隔的方式命名）。

类名采用具有代表性的名词或名词短语。例如，字符 Character 或不可变列表 ImmutableList。接口命名同样也采用具有代表性的名词或名词短语（如列表 List），但是一些情况下可以用形容词或形容词短语代替（如：Readable）。
测试类命名由要测试的类名开始，以 Test 结尾。例如，HashTest 或 HashIntegrationTest。

5.2.3 方法名

方法名应该写成 lowerCamelCase（采用小写字母开头，大小写字母间隔的方式命名）。

方法名采用具有代表性的动词或动词短语。例如：sendMessage 或 stop。
单元测试的方法名用下划线区分名字中的不同组成成分，每种组成成分都采用 lowerCamelCase 的格式，一个典型的模式时是：<methodUnderTest>_<state>, 例如 pop_emptyStack。这里没有一个公认的准则对于如何命名单元测试的名称。

5.2.4 常量名

常量名被写作 CONSTANT_CASE 的形式，常量名中所有的字母大写，并且用下划线分隔不同的单词。那什么是常量？

常量是指那些静态最终字段，这些字段的内容不可被修改，并且他们的方法只有一种结果。这里包括原始类型，字符串，不可变类型和具有多个类型的不可变集合。如果任何实例应该遵守的状态能够改变，那么他就不是一个常量。仅仅是一个不可变对象是不够的。例如：

```
// Constants
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 }

// 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 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"};
```

这些名称为具有代表性的名词或名词短语。

• 以上 5.2.1、5.2.2、5.2.3、5.2.4 的命名方式，既能够使代码具有相对统一的格式，同时，不同种类的命名方式又有区别，这能够给阅读提供便利。同时，上面的命名方式可以让我们通过命名就知道类，方法，常量的大致作用，更加方便阅读，同时也方便开发和维护。

5.2.5 非常量字段命名

非常量字段(静态字段或其他)被写作 lowerCamelCase(采用小写字母开头，大小写字母间隔的方式命名)。

这些名称为具有代表性的名词或名词短语。例如，computedValue 或 index。

5.2.6 参数名

参数名被写作 lowerCamelCase（采用小写字母开头，大小写字母间隔的方式命名）。

在公共方法中应该避免出现一个字符的参数名。

5.2.7 局部变量名

局部变量名被写作 lowerCamelCase（采用小写字母开头，大小写字母间隔的方式命名）。

即使局部变量是不可变并且是静态的，它也不能被当作是一个常量，也不能采用常量的方式去命名。

5.2.8 类型变量名

每一个类型变量名采用下面两个方式之一命名：

*一个大写字母，后面可以跟着一个数字（像 E，T，X，T2）

*一个类名（见 5.2.2 节 类名），后面跟一个大写字母 T（例如：RequestT，FooBarT）。

5.3 Camel case 的定义（驼峰式命名）

一些时候有超过一种合理的方式将英语短语转换为驼峰的格式，像一些缩写或者组合词语，如“IPv6”或“iOS”。为了提高统一性，谷歌风格指定了以下转换方式。

从名字的散文形式开始：

1. 将短语转换为普通的 ASCII 字符，并且删去'等符号。例如：“Müller's algorithm”转换为“Muellers algorithm”。
2. 将上面的结果拆分为单词，通过空格和任何其他标点划分（通常是连字符）。

*建议：如果任何单词已经作为一个常用的驼峰样式出现，分隔其组

成部分（例如："AdWords" 转化为 "ad words"）。注意一个像“iOS”这样的词本质上并不是真正的符合驼峰规则，而是习惯是用的一个词，所以不适合拆分。

3. 将所有词语转换为小写格式（包括缩写词），然后将每个单词的首字母大写：

   *每个单词都符合大写驼峰规则，或者

   *除了第一个单词，符合小写驼峰规则

4. 最后，聚合所有单词成为一个的标识符。

注意这种方式几乎完全忽视原词的格式，例如：

散文格式	正确命名	错误命名
"XML HTTP request"	XmlHttpRequest	XMLHTTPRequest
"new customer ID"	newCustomerId	newCustomerID
"inner stopwatch"	innerStopwatch	innerStopWatch
"supports IPv6 on iOS?"	supportsIpv6OnIos	supportsIPv6OnIOS
"YouTube importer"	YouTubeImporter YoutubeImporter *	

*表示可以接受但不推荐的命名

注意：在英语单词中有一些容易引起歧义的连字符号：例如“nonempty”和“non-empty”都是正确的，所以方法名 checkNonempty 和 checkNonEmpty 也是正确的。

- 命名规则的统一能够使代码整体看起来更加整洁美观，并且使用上述的命名规则，可以通过名称就大致了解代码的功能，方便阅读代码。

6 编程实践

6.1 @Override：总是使用

当一个方法使用@Override 是合法的，那么就使用@Override 注解。这包括了一个类方法重写了一个超类的方法，一个类方法实现了一个接口的方法，一个接口方法重新指定一个超接口方法。

除下面情况外：@Override 可以省略当他的父级方法为@Deprecated 注解时。

- 这样使用@Override 注解能够帮助我们阅读时候区分是重写超类的方法，

还是实现方法的重载。

6.2 异常捕获，不能被忽略

除下面情况之外，对一个被捕获的异常不作出反应是不正确的（代表性的反应是日志，如果他被认为是“不可能的”，抛出一个断言错误）。
如果在一个异常的捕获块中不作任何操作是恰当的，应该在注释中说明这个不作操作的理由。

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

除了下面情况外：在测试中，如果捕获的异常是预期的异常，则有可能被忽略，并且没有注释说明。下面是一个很常见的用法用于确保被测试的代码不会抛出预期中的异常，所以这里的注释并不重要。

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

•捕获异常并进行处理，能够提高功能的稳定性。通过打印日志及其他操作，可以快速定位功能存在的问题。

6.3 静态成员：合理使用类

当需要引用一个静态类成员时是有条件的，这个条件就是引用时是通过类名，而不是这个类的引用或表达式去引用这个静态：

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

• 使用类名调用静态成员是因为静态成员在类在加载的时候就已经被初始化，他不必通过实例化类去初始化，使用类实例化的对象调用静态成员显得画蛇添足。

6.4 不要使用 Finalizers 方法

重写 Object.finalize 方法是特别罕见的。

提示：不要重写 Object.finalize 方法，如果你绝对必须这样做，你首先需要仔细阅读和理解 Effective Java 书中的第七条，“避免使用终结器”，然后不使用它。

7 Javadoc

7.1 格式

7.1.1 一般形式

Javadoc 的基本格式就像下面展示的例子一样：

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

或者像这样的单行注释：

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

无论什么情况，使用基本的格式都是可以的，而单行格式是当整个 javadoc 注释块可以放在一行时可以被使用。请注意单行注释仅仅适用于没有像@return 这样的块标签。

7.1.2 段落

空白行指的是上下两个段落之间，只包含一个用于对齐的星号（*），并且出现在下一个块标签之前。每一个段落在第一个单词之前都有一个<p>标签，之后紧跟注释内容，没有空格。

• 这样做能够更加方便阅读代码，使注释看起来更加整洁明了。

7.1.3 块标签

任何标准的“块标签”使用时依次出现的顺序应该是：@param,，@return，@throws，@deprecated，并且这四种类型出现时，它的说明都不会为空。当一个块标签的说明超过一行时，延续行以@符号开始，需要缩进四个或更多的空格。

• 一般来说@param,，@return，@throws，@deprecated 后的注释内容不会超过一行，当超过一行时，缩进四个空格用以区别其他的换行。

7.2 摘要片段

每一个 javadoc 都是以一个简明的摘要片段开头。这个片段特别重要：它是唯一出现在文本中的类和方法的说明。

这是一个片段——是由一个动词短语或名词短语构成，而不是一个完整的句子。它不是以 A {@code Foo} is a...，或 This method returns...开头，也不是一个像 Save the record.这样的完整祈使句。但是，这个片段也会像完整的句子一样使用大写字母和标点符号。

提示：一个普遍的错误是以/** @return the customer ID */这样的形式写一个简单的 javadoc。这是不对的，应该被改为/** Returns the customer ID. */。

• 因为 javadoc 会被提取为说明文档，因此对于一个说明文档来说，大小写和标点符号是重要的，不过我们主要是用中文注释，所以标点符号尤为重要。

7.3 什么地方使用 Javadoc

至少，Javadoc 出现在每一个公共类，或者一个类的 public 或 protected 成员，除了少数情况下。

一些额外的 Javadoc 内容也可能出现，在 7.3.4 节说明，非必需 Javadoc.

• 说明文档能够帮助用户更好的理解和使用接口，了解类，接口，方法的作用，所以对于公共类和方法，说明文档是必要的。

7.3.1 例外：方法名本身能够说明方法的信息

Javadoc 对于像 getFoo 这样的“简单明显”方法的说明是可以选择的，如果真的没有其他可以说明的地方，可以直接说明“返回 foo”。

重要的：为了省略一些读者可能需要知道的相关信息而引用这个例外情况是不恰当的。例如，一个方法名为 getCanonicalName 的方法，不要省略它的文档（理论上仅仅说明/** Returns the canonical name. */），一个典型的问题是读者不知道“canonical name”是指什么。

7.3.2 例外：重载方法

当一个方法重写它超类的方法时，可以不用 Javadoc 说明。

- 需要在它的超类中进行说明

7.3.4 非必需的 Javadoc

其他类和成员具有期望或需要的 Javadoc.

当使用一个注释可以定义一个类或成员的总体目标和行为，这里的注释就可以写成 Javadoc 代替使用多行注释。

非必需 Javadoc 不必严格的遵循 7.1.2、7.1.3、7.1.4 中的规则格式，尽管这是推荐的格式。

- 当一个注释可以定义一个类或成员的总体目标和行为，那么用户就可以通过这个注释来了解到类和成员的信息，放在说明文档更加便于用户的阅读和使用。