Google Java Style Guide

谷歌 Java 编程风格指南

作者：cushon & shicks

组织：Google Style Guides Program

时间：May 23, 2018

版本：5.2

译者：陶冶

工欲善其事，必先利其器。——《论语・卫灵公》

写在前面

软件构造课的前置语言是 JAVA，工欲善其事，必先利其器，故吾认为，在学习 JAVA 之前更重要的是了解 JAVA 的编程规范，在华为，谷歌，阿里，苹果等一众公司中，吾选择了谷歌的规范 (Google Java Style Guide) 作为蓝本 (因为谷歌的编程规范成体系、历史久远且在Github上开源)，目前网上的译本大多是Hawstein大佬于 January 20, 2014 译的2.2 版本, 大佬翻译的很好，不过由于版本更替，该版本有些地方已不适用。现特将最新版 (May 23, 2018) 翻译，欢迎指正[1]。

陶冶[2]

April 23, 2020

[1]1180300204@stu.hit.edu.cn

[2]哈尔滨工业大学　计算机学院

目录

第一章 引言

此文档旨在**完整**的定义 Google 对于 Java 源码的编程规范。当且仅当 Java 源码完全遵守本文档时，此源码才被称为 Google 风格 (*Google Style*)。

与其他 (Google) 的编程规范一样，本文档不仅阐述了因审美引起的格式问题，同时也阐述了其他形式的公约以及编程标准。但是，本文档仅专注于我们普遍遵守的**强制规范** (hard-and-fast rules)，并避免给出界定模糊的建议 (对人或编译工具来说)。

1.1 术语说明

本文档中，除非特意指明，否则一般遵守下列表述：

1. 类 (class)：指一个普通类，枚举类，接口或者注解 (@interface)；
2. 成员 (memeber)：指嵌套类，类变量 (field)，方法或构造函数。也即，除初始化函数 (initializers) 和注释外的所有类中的顶级内容；
3. 注释 (comment)：总是指实现用 (implementation) 注释[1]。本文档使用更常用的短语 “Javadoc’’ 代替 ‘‘文档用注释”(documentation comment) 短语。

其他文档术语将在本文档中陆续出现。

1.2 指南说明

本文档中的示例代码是**非规范化** (non-normative) 的。即，示例代码是遵循 Google 风格的，但这并不表示这是唯一的表达方式。示例代码中可选的格式优化不应该被归于强制规范 (，所以并未归入此文档中)。

[1]实现用注释是指：如/* A comment */或// Another comment。而文档用注释/Javadoc 则是指/** This is a javadoc */。具体可以参看Oracle 的文档。

第 二 章 源码文件基础守则

2.1 文件名

源码文件名由区分大小写的顶层类名 (仅有 1 个)，和.java后缀构成。

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 白空格字符 (Whitespace characters)

除换行符外，**ASCII 的横向空格字符** (0x20) 是在源码文件中唯一出现的白空格字符。这也就意味着：

1. 所有其他的位于字符或字符串中的白空格字符应当转义。
2. 制表符 (Tab) **不能**用于缩进。

2.3.2 特殊转义序列

若任意字符中包含特殊转义序列(\b, \t, \n, \f, \r, ", ’及\\)，则应直接使用该序列而非使用其八进制形式 (如\012) 或其 Unicode 转义 (如\u000a)。

2.3.3 非 ASCII 字符

对于余下的非 ASCII 字符，可使用实际 Unicode 字符 (如 ∞) 或等效 Unicode 转义 (如\u221e) 表示。尽管 (本文档) 强烈不推荐在字符串和注释外出现 Unicode 转义，(实际中) 如何取舍仅取决于哪个更容易阅读和容易理解。

注 在使用 Unicode 转义，或直接使用 Unicode 字符时，添加解释性的注释会很有帮助。

例如：

示例	讨论
String unitAbbrev = "μs";	最佳：无需注释也很清晰明了。
String unitAbbrev = "\03bcs"; // "μs"	可行，但是没有理由这样做。
String unitAbbrev = "\03bcs"; // Greek letter mu, "s"	可行，但是很别扭，而且容易出错。
String unitAbbrev = "\03bcs";	较差：读代码的人不知道这是什么。
return ’\ufeff’ + content; // byte order mark	好：对不可打印的字符使用转义，并进行必要的注释。

注 永远不要因为害怕程序不能正确处理非 ASCII 字符而降低源码可读性。如果程序未能正确处理非 ASCII 字符，那么该程序是损坏的并应着手修复。

第 三 章 源文件结构

一个源文件应**按照顺序**，包含以下内容：

1. 如果包含许可或版权信息，应显示
2. 所处的包的声明 (Package statement)
3. 导包声明 (import statements)
4. 有且仅有 1 个顶层的类

仅使用一个空行去分割以上出现的内容。

3.1 如果包含许可或版权信息，应显示

若文件中包含许可或版权信息，应在此处显示。

3.2 所处的包的声明

所处的包声明**不允许自动换行**。该声明不受每行字符限制 (见 4.4，每行限制 100 个字符)。

3.3 导包声明

3.3.1 禁止通配符导入

(Google Style) **不使用通配符导入**，无论是在静态导入中，或其他 (非静态) 导入。

3.3.2 不允许自动换行

导包声明**不允许自动换行**。该声明不受每行字符限制 (见 4.4，每行限制 100 个字符)。

3.3.3 顺序与空格

导入顺序遵循以下原则：

1. 所有静态导入在同一区块
2. 所有非静态导入在同一区块

如果静态导入和非静态导入同时出现，使用一个空行分割两个区块。其他导包声明之间无空格。

每个区块的导包语句按照名字的 ASCII 顺序排序。(**注意**：这并不意味着整个导包语句都按照 ASCII 顺序排序，因为 (在 ASCII 表中) '.' 出现在 ';' 前)

3.4 禁止静态导入类

3.4.1 有且仅有一个顶层类声明

每个顶层类都处在 (以他类命名的) 源文件顶层。

3.4.2 类内容的顺序

如何排列你的类成员与初始化函数是一件很容易学习的事情。但是，排列的方法不止一种；不同的类有不同的内容排序方法。

重要的是，每个类都应使用**某种逻辑排序**，这样代码维护者在被 (审阅代码时) 问到的话就可以有充足的理由解释清楚。例如，新的方法不仅仅处于习惯性的放在类的末尾，放在末尾意味着此种顺序产生了 ‘‘按日期添加’’ 的非逻辑排序。

3.4.3 重载：不要分割开

当一个类有多个构造函数，或多个方法重名时，他们应该一个接一个的挨个卸下来，中间不能有任何其他代码 (private 方法也不行)。

第四章 格式

术语说明：块状结构 (block-like constructs) 指类的主体，方法或构造函数。注意，在 4.8.3.1 节数组初始化声明 (array initializers) 中，数组初始化声明可以被选择性的视为块状结构。

4.1 大括号

4.1.1 可用可不用的时候必用大括号

在if, else, for, do, while语句中，即便无内部语句或只有一句，也必须使用大括号。

4.1.2 非空语句块：使用 K&R 风格

对于非空语句块和块状结构，使用 Kernighan and Ritchie 风格 (埃及括号 (Egyptian brackets)) 的大括号：

- 左大括号前不换行。
- 左大括号后换行。
- 右大括号前换行。
- 仅当结束一段声明、函数、构造函数、非匿名类时，右大括号后才换行。比如，当右大括号后有else或逗号时，右大括号后就不换行。

示例：

```
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();
    }
```

```
  }
};
```

枚举类有几个例外，详情参见 4.8.1 节枚举类

4.1.3 空语句块：有可能被简化

空语句块或块状结构可以使用 K&R 风格 (在 4.1.2 节中)、除此之外，除非该空语句块是多段语句块的一部分 (如if/else或try/catch/finally)，空语句块的左右大括号也可以不加字符，不加换行的紧挨在一起 ({})。

示例：

```
// 可以接受
void doNothing() {}

// 同样可以接受
void doNothingElse() {
}
```

```
// 不可接受：多段语句块中不允许存在简化过的空语句块
try {
  doSomething();
} catch (Exception e) {}
```

4.2 区块缩进：+2 空格

每当新增一个新的区块或块状结构时，缩进 +2 空格。当该区块结束后，缩进回到之前的缩进水平 (即-2 空格)。缩进区域对于所处区块的所有代码和注释都均适用。(见 4.1.2 节非空语句块：使用 K&R 风格)

4.3 每行仅一条语句

每个语句结束后都应换行。

4.4 每行限制：100 个字符

Java 源码每行限制 100 个字符。‘‘字符’’ 是指任何一个 Unicode 码位。除以下特殊情况外，任何一行超过了 (100 个字符后) 都必须换行，详情参见 4.5 节换行。每个 Unicode 码位算作一个字符，无论该字符显示出来 (相较一般) 更宽或更窄。例如，如果使用全角字符，你可能需要在到达 100 个字符限制前提早换行。

例外：

1. 当情况不允许时 (例如，Javadoc 中出现的长 URL，或者一个很长的 JSNI(JavaScript Native Interface) 方法引用)。
2. package和声明 (见 3.2 节所处包的声明和 3.3 节导包声明)。
3. 注释中可能需要粘贴到 shell 中的命令。

4.5 换行

术语说明：当代码可以合法占据一行，但却分割成若干行时，此行为成为换行。

并没有全面的，确定性的公式定义每种情况下如何准确的换行。很多情况下，对于同一行代码，都会存在若干种合理的分割方法。

笔记 尽管一般情况下换行是为了防止超出每行的字符限制，但有时候尽管并未超出此限制，作者也会处于谨慎将代码分割成多行。

注 (将原本的代码块) 提炼成一个方法或一个局部变量可免除换行的苦恼。

4.5.1 在何处换行

换行的一个主旨是：倾向于在更高的语法层上断句。并且：

1. 当换行出现在非赋值符号处时，在该符号前换行。(注意这点与 Google 其他语言的 coding style 不太一致，比如 C++，JavaScript。)
   - 此条也适用于以下 ‘‘似操作符’’ 的符号：
     - 点分隔符 (.)
     - 双冒号代表的方法引用 (::)
     - 类型绑定中的与 (&) 符号 (<T extends Foo & Bar>)
     - catch区块中的或 (|) 符号 (catch (FooException | BarException e))
2. 当换行出现在赋值符号处时，通常在该符号后换行，但 (在前或者在后换行) 都可以接受。
   - 此条也适用于for(foreach) 语句中的 ‘‘似赋值符号’’ 冒号 (:)。
3. 一个方法或构造函数的名字总是与其左小括号 (() 连在一起。
4. 逗号 (,) 与其前面的内容留在同一行。
5. 在 Lambda 表达式中，一行从不会被断开，除了如果 Lambda 表达式的主体由单个非支持表达式组成，则可能会在箭头之后立即出现断开。

示例：

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

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

笔记 换行的主要目的是要有清晰的代码，而不必是适合最少行数的代码。

4.5.2 自动换行时缩进至少 +4 个空格

自动换行时，第一行后面的每一行 (每个连续行) 从原始行缩进至少增加 4 个空格。

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。

章节 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：

1. 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 (在它们之间没有其他代码) 是可选的。这样的空白行根据需要用于创建字段的逻辑分组。
   - **例外**：枚举常量之间的空行，章节 4.8.1 将介绍。
2. 在函数体内，语句的逻辑分组间使用空行。
3. 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。)
4. 要满足本文档中其他节的空行要求。(比如章节 3：源文件结构及章节 3.3：import 语句)

多个连续的空行是允许的，但没有必要这样做 (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 空格仅出现在以下几个地方：

1. 分隔任何保留字 (如：if, for catch) 与紧随其后的左括号 (()。
2. 分隔任何保留字 (如else, catch) 与其前面的右大括号 (})。
3. 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)。
4. 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 & 符 (<T extends Foo & Bar>)。
   - catch 块中的管道符号 (catch (FooException | BarException e)。
   - foreach 语句中的冒号 (:)。
5. 在, : ;及右括号 ()) 后

6. 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。
7. 类型和变量之间：List<String> list。
8. 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。

此规则不应理解为在行的开始或结束处要求或禁止额外的空格; 它只涉及行的内部空格。

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，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
```

注 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。还允许附加空行 (通常只有一个)。这是一个样例：

```
private enum Answer {
```

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

  NO,
  MAYBE
}
```

没有方法和文档的枚举类可写成数组初始化的格式 (详见章节 4.8.3.1)。

```
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可行的 (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中出现。示例：

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

注意，在case 1后面不需要注释，只有在语句组的末尾需要。

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。

例外：enum类型的 switch 语句可以省略default语句组，如果它包括覆盖该类型的所有可能值的显式情况。这使 IDE 或其他静态分析工具能够在错过任何情况时发出警告。

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

本节讨论实现注释。Javadoc 在章节 7 Javadoc 中单独讲解。

任何换行符之前可以有任意空格，然后是实现注释。这样的注释使该行非空白。

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。

```
/*
 * This is
 * okay.
 */

// And so
// is this.

/* Or you can
 * even do this. */
```

注释不要封闭在由星号或其它字符绘制的框架里。

注 当编写多行注释时，如果您希望自动代码格式化程序在必要时重新换行 (段落样式)，请使用/* ... */样式。大多数格式化程序不会在// ... 样式注释块中重新换行。

4.8.7 修饰符

类和成员的修饰符如果存在，则按 Java 语言规范中推荐的顺序出现。

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

4.8.8 数字字面量

long数值使用大写L后缀，而非小写 (以避免与数字1混淆)。例如，3000000000L而不是30000000000l。

第 五 章 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 Style 中，特殊的前缀或后缀不使用在示例中看到的如name_, mName, s_name和kName。

5.2 标识符类型的规则

5.2.1 包名

包名称都是小写，连续的单词连接在一起 (无下划线)。如com.example.deepspace而非com.example.deepSpace或com.example.deep_space。

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。

类名通常是名词或名词短语 (如：Character或ImmutableList)，接口名称通常也是名词或名词短语 (如List)，但有时可能是形容词或形容词短语 (如Readable)。

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。

测试类从它们正在测试的类的名称开始命名，并以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。例如，sendMessage或stop。

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。一个典型的模式是：test<MethodUnderTest>_<state>，例如testPop_emptyStack。尚未出现给测试方法命名的标准命名准则。

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？

常数是静态 final 字段，其内容是不可变的，并且其方法没有可检测的函数副作用。这包括基元，字符串，不可变类型和不可变类型的不可变集合。如果任何实例的观测状态可以改变，它就不是一个常量。静态 final 字段不一定都是常量。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 (静态或其他) 以lowerCamelCase风格编写。

这些名字通常是名词或名词短语。例如，computedValues或index。

5.2.6 参数名

参数名以lowerCamelCase风格编写。

应该避免公共方法中的单字符参数名称。

5.2.7 局部变量名

局部变量名以lowerCamelCase风格编写。

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 CamelCase：定义

有时不止一种合理的方式可以将短语转换为 CamelCase 模式，例如首字母缩略词或不寻常的结构，如 “IPv6’’ 或 “iOS’’。为了提高可预测性，Google Style 指定 (尽量) 使用以下确定性方案。

从名称的文本形式开始：

1. 将短语转换为纯 ASCII 并删除任何省略号。例如：“Müller’s algorithm’’ 可改为 “Muellers algorithm’’。
2. 以空格和任何剩余的标点符号 (通常为连字符) 将结果划分为单词。
   - 推荐：如果有任何词在常用的情况下已经具有常规的 CamelCase 外观，将其拆分为其组成部分 (例如，“AdWords’’ 成为 “ad words”)。注意，诸如 “iOS’’ 这样的词本身不是真正的骆驼情况；它违反任何惯例，因此本建议不适用。
3. 将单词 (包括首字母缩略词) 第一个字符大写其他字符全小写：
   - 将所有字符全大写
   - 除第一个字符外，将所有字符全小写
4. 最后，将所有单词连接成一个词。

注意，原始单词样式几乎完全被忽略。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*	

* 可接受，但不推荐。

笔记 一些单词在英语中有不明确的连字符：例如"nonempty" 和"non-empty" 都是正确的，所以方法名称 checkNonempty 和 checkNonEmpty 也都是正确的。

第六章 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。这包括重写超类方法的类方法，实现接口方法的类方法，以及重定义超接口方法的接口方法。

例外：当父方法为@Deprecated时，可以省略@Override。

6.2 捕获的异常：不能忽视

除下面的例子，对捕获的异常不做响应极少是正确的。(典型的响应方式是打印日志，如果不行，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。

注 不要这么做，不得已非要这么做的话，请先仔细阅读并理解Effective Java Item 7: "Avoid Finalizers"，非常小心，最后还是不要这么做。

第七章 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. */
```

基本形式总是可以接受的。当整个 Javadoc 块 (包括注释标记) 可以放在单个行上时，可以将其替换为单行形式。注意，这只适用于没有块标签的情形，如@return。

7.1.2 段落

空行 (即仅包含对齐的前导星号 (*) 的行) 出现在段落之间，并在 Javadoc 标记 (如果存在) 之前。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。

7.2 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。

注 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。

也可能存在其他 Javadoc 内容，如章节 7.3.3 所述。

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。这种情况下除了写"Returns the foo"，确实也没有什么值得写了。

笔记 如果有一些相关信息是需要读者了解的，那么以上的示例不应作为忽视这些信息的理由。例如，对于名为getCanonicalName，的方法，不要省略其文档 (文档可以仅仅是/** Returns the canonical name. */)，可能读者并不明白"canonical name" 具体指什么。

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 非必需 Javadoc[1]

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(使用/**)。

不一定要求非必需的 Javadoc 遵循章节 7.1.2、章节 7.1.3 和章节 7.2 的格式化规则，当然也是建议遵循的。

[1]译者著：英文版此为 7.3.4(缺失 7.3.3)，但是在 Mar 22, 2014 更新的 2.2 版中是有 7.3.3 的。应该是作者失误

第 八 章 版本更新历史

将更新历史记录于此，查看更新时间节点。

2020/04/23 更新：V1.4 正式发布

① 修正了一些翻译失误，录入错误。

② LaTeX 版正式于 GitHub 发布。

③ 修改了CSDN上的版本

2020/04/22 更新：V1.3 正式发布

① 正式于CSDN发布。

② 启动 LaTeX 编译计划 (采用 TeXLive2019+TeXstudio 配合ElegantBook 模板)。

2020/04/17 更新：V1.2

① 与 Hawstein 大佬的版本进行对比，修改了若干翻译不当。

② 美化 Markdown 版本。

2020/04/15 更新：V1.1

① 修改了若干翻译错误。

② Markdown 编排完毕。

2020/04/13 更新：全部翻译完毕，记事本 V1.0 出世

2020/04/02 更新：至第四章格式翻译完毕毕

2020/03/17 更新：至第三章源文件结构翻译完

2020/03/06 更新：至第二章源码文件基础守则翻译完毕