

系统分析与设计课程项目

团队代码风格 (coding style)

目录

一，Java 部分	1
二，JavaScript 部分	24
三，SQL 部分	34

一，Java 部分

1.1 术语说明

在本文档中，除非另有说明：

1. 术语 class 可表示一个普通类，枚举类，接口或是 annotation 类型
(@interface)
2. 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：

1. 所有其它字符串中的空白字符都要进行转义。
2. 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。

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。

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：

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。

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：

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。

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。

1. 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)`)

2. 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。
3. 方法名或构造函数名与左括号留在同一行。
4. 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：

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是可选的，用于字段的空行主要用来对字段进行逻辑分组。
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 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:

1. 分隔任何保留字与紧随其后的左括号()`(`(如 `if`, `for` `catch` 等)。
2. 分隔任何保留字与其前面的右大括号()`)`(如 `else`, `catch`)。
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, 见下面的 Note)。
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`&`(`<T extends Foo & Bar>`)。
 - `catch` 块中的管道符号(`catch (IOException | BarException e)`)。
 - `foreach` 语句中的分号。
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 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。

Note：这个规则并不要求或禁止一行的开头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 getNamelfPresent() { ... }
```

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：

1. 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"。
2. 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。
3. 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。
4. 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	
	YoutubelImporter*	

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。

二，JavaScript 部分

一、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。

二、 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) {...}
```

三、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。

四、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。

五、相等和严格相等

Javascript 有两个表示"相等"的运算符："相等" (==) 和"严格相等" (===)。

因为"相等"运算符会自动转换变量类型，造成很多意想不到的情况：

```
0 == "" // true
```

```
1 == true // true
```

```
2 == true // false
```

```
0 == '0' // true
```

```
false == 'false' // false
```

```
false == '0' // true
```

```
"\t\r\n" == 0 // true
```

因此,

规则 7：不要使用"相等" (==) 运算符，只使用"严格相等" (===) 运算符。

六、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。

七、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。

八、全局变量

Javascript 最大的语法缺点，可能就是全局变量对于任何一个代码块，都是可读可写。这对代码的模块化和重复使用，非常不利。

规则 11：避免使用全局变量；如果不得不使用，用大写字母表示变量名，比如 UPPER_CASE。

九、new 命令

Javascript 使用 new 命令，从构造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，都变成全局变量。

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，最好在视觉上把构造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。

十、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。

```
++x
```

等同于

```
x += 1;
```

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。

十一、区块

如果循环和判断的代码体只有一行，Javascript 允许该区块（block）省略大括号。

下面的代码

```
if (a) b(); c();
```

原意可能是

```
if (a) { b(); c(); }
```

但是，实际效果是

```
if (a) { b(); } c();
```

因此，

规则 15：总是使用大括号表示区块。

三，SQL 部分

格式良好的SQL并不会比乱七八糟的SQL运行效果更好。数据库其实不怎么关心SQL语句中你把逗号放到了字段名的前面还是后面。为了你自己思路清楚，应该做一个有效率的SQL编写者，我建议你遵守以下这些格式规则。在本文中我将分享如何通过格式良好的SQL语句提升生产率。我定义的效率指的是能从SQL输出准确的结果，并且代码清晰易于理解、修改和调试。我只列出了

“SELECT” 语句，因为我写的SQL语句99%都是查询语句。格式化SQL代码是非常个性化的事，我也很清楚因人而异，开发者都认为自己的格式化规则是最合理的。

样例问题

下面是一个典型的SQL应用场景，业务报表的数据来自三张表，客户表、销售表和地域表。基于2015年一月份的数据，该报表需要展示在每个行政区内的客户总数和销量总数。该需求通过一个简单的SQL语句就可以实现，需要关联查询三张表。

数据可能出现的问题

虽然SQL很简单，但保证你的结果正确仍然是真正的关键，因为下面一些原因可能导致错误：

数据可能来自不同的数据源。也就是说你不能保证这几个表之间的完整性。具体举例来说，你不能假定客户表中所有的邮政编码都是有效的邮政编码，并且一定在地域表中存在。

录入客户表数据的应用可能捕获到未经验证的地点数据，可能会包括错误的邮政编码。

邮政编码表可能不是完整的。新发布的邮政编码可能没有在发布后及时导入到表中。

cust_id	cust_name	↓ cust_post_code
2	Bradley Keller	60000
1	May McGee	8000
9	Danielle Miles	8000
8	Constance Moss	5040
3	Bethany Bradley	2000
4	Lucille Scott	2000
5	Abraham Graham	1803
6	Mildred Fields	1803
10	Shannon Harmon	800
7	Kent Ortiz	180

post_code	location
8000	Cape Town
6000	Port Elizabeth
2000	Johannesburg
700	Polokwane
1803	Soweto

第一原则

对我来说，相比于编写清晰易读的SQL，从SQL得到正确的结果才是第一要务。

我要做的第一件事就是编写下面的SQL语句来获取客户总数。在我写完整个语句之后我会再调整它。

我写的第一个语句是这样的：

```
SELECT COUNT(DISTINCT cust_id) as count_customers FROM customers
```

Result:

count_customers

“10”

这个查询很重要，因为它紧紧围绕第一原则。因为没有SQL管理查询，也就没有依赖，我知道这就是客户数量的正确结果。我把这个结果记下来，因为我总需要拿这个数字来衡量后面的SQL(是否正确)，在本文后面也会多次提到。

下一步要做的事就是添加必要的字段和表完成查询。我特意把“添加”这个词高亮标注出来，因为根据我的规则，我会在应用第一原则时把能获取相同结果的查询注释掉。下面就是我最终格式化的查询语句。

格式化SQL

下面就是根据我的格式化思路推荐的格式化SQL。

```
SELECT
```

```
0
```

```
,c.cust_post_code
```

```
,p.location
```

```
,COUNT(DISTINCT c.cust_id) number_customers
```

```
,SUM(s.total_amount) as total_sales
```

```
FROM
```

customers c

JOIN post_codes p ON c.cust_post_code = p.post_code

JOIN sales s ON c.cust_id = s.cust_id

WHERE

1=1

AND s.sales_date BETWEEN '2015-01-01' AND '2015-01-31'

—AND s.order_id = 5

GROUP BY

c.cust_post_code

,p.location

总是使用表别名

时间会证明这么做是有必要的。如果你没有对SQL语句中用到的每个字段使用别名，在将来某个时候可能会给这个查询语句添加进来别的同名字段。到那时候你的查询乃至报表就会产生错误(出现了重名字段名)。

逗号放到字段之前

在调试或者测试我的查询语句时，这么做可以方便地注释掉某个字段，而不需要修改其它行，所有的逗号都没有缺少或多余。不这么做的话你可能总要调整逗号才能保证语句正确。如果你经常要调试语句，这么做会带来极大方便，效率会更高。这个做法对“SELECT”部分和“GROUP BY”子句部分同样适用。

在开发时我使用“SELECT 0”作为语句的开始，迁移到正式环境时它很容易删除掉。这样我们就可以在后面所有字段前面都写都好了。没有这个“0”的话，如果我想注释掉第一个字段(本例中是“c.cust_post_code”)，我就必须处理后面的逗号问题。我必须临时注释掉它，将来还要加回来。在“GROUP BY”语句中也是一样的。这个“0”是额外加的。

把“JOIN”放到独立行

把“JOIN”语句放到独立行有以下**好处**：

这么做很容易看到本查询语句涉及的所有表，只需要看滚动“JOIN”语句就可以了。

使用“JOIN”相比于在“WHERE”子句中列出所有表和表达式关系，可以把所有逻辑关系都放到一个地方。我们不可能总是把“JOIN”语句放到一行中，但是至少应该放到一起。

这么做的话要注释掉“JOIN”语句也是相对容易的。这在调试时非常有用，你可能需要知道是否是“JOIN”引起了数据问题。

列模式编辑

在处理大量字段的情况时，列模式编辑非常方便。下面是我曾经做过的第一个动态GIF展示，你可以注释掉所有非聚集字段。我使用了列模式编辑，而不仅仅是注释掉字段：

创建全部索引

在使用字段较多的UNION语句时：

注释掉“GROUP BY”子句的字段清单

```
1  SELECT
2      0
3      --,c.cust_post_code
4      --,p.post_code
5      --,p.location
6      ,COUNT(DISTINCT c.cust_id) number_customers
7      ,SUM(s.total_amount) AS total_sales
8  FROM
9      customers c
10     JOIN post_codes p ON c.cust_post_code = p.post_code
11     JOIN sales s ON c.cust_id = s.cust_id
12  WHERE
13      1=1
14      AND s.sales_date BETWEEN '2015-01-01' AND '2015-01-31'
15      --AND s.order_id = 5 GROUP BY c.cust_post_code ,p.location
16  GROUP BY
17      c.cust_post_code
18      ,p.location
19      ,p.post_code
```

测试查询结果

我必须使用外连接“OUTER”列出所有客户，因为不是所有客户的邮政编码都在地域表里有相应的邮政编码。我可以通过包含和排除不同字段和表反复操作来确保我查询的结果与最开始那个查询(单独查询客户的那个语句)结果相同，这其实是对第一原则的遵守。

```
SELECT 0,c.cust_post_code—,p.location,COUNT(DISTINCT c.cust_id)
number_customers,SUM(s.total_amount) as total_salesFROM customers c—LEFT
OUTER JOIN post_codes p ON c.cust_post_code = p.post_codeJOIN sales s ON
```

```
c.cust_id = s.cust_id WHERE 1=1 AND s.sales_date BETWEEN '2015-01-01' AND  
    '2015-01-31'—AND c.cust_post_code = 2000—AND p.post_code =  
200 GROUP BY c.cust_post_code—,p.location
```

像这样的 SQL 对我来说意味着我必须写独立的测试来检查数据。通过注释掉的那几行语句我可以使用第一原则验证我查询数据的准确性。这么做提高了我的效率和报表。