

内容目录

- [Google 开源项目风格指南 \(中文版\)](#)
- [C++ 风格指南 - 内容目录](#)
 - [0. 扉页](#)
 - [0.1 译者前言](#)
 - [0.2 背景](#)
 - [1. 头文件](#)
 - [1.1. Self-contained 头文件](#)
 - [1.2. #define 保护](#)
 - [1.3. 前置声明](#)
 - [1.4. 内联函数](#)
 - [1.5. #include 的路径及顺序](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
 - [2. 作用域](#)
 - [2.1. 名字空间](#)
 - [2.1.1. 匿名名字空间](#)
 - [2.1.2. 具名的名字空间](#)
 - [2.2. 嵌套类](#)
 - [2.3. 非成员函数、静态成员函数和全局函数](#)
 - [2.4. 局部变量](#)
 - [2.5. 静态和全局变量](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
 - [3. 类](#)
 - [3.1. 构造函数的职责](#)
 - [3.2. 初始化](#)
 - [3.3. 显式构造函数](#)
 - [3.4. 可拷贝类型和可移动类型](#)
 - [3.5. 委派和继承构造函数](#)
 - [3.6. 结构体 VS. 类](#)
 - [3.7. 继承](#)
 - [3.8. 多重继承](#)
 - [3.9. 接口](#)
 - [3.10. 运算符重载](#)
 - [3.11. 存取控制](#)
 - [3.11. 声明顺序](#)
 - [3.12. 编写简短函数](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [4. 来自 Google 的奇技](#)
 - [4.1. 所有权与智能指针](#)
 - [4.2. cpplint](#)
 - [译者 \(acgtyrant\) 笔记](#)

- [5. 其他 C++ 特性](#)
 - [5.1. 引用参数](#)
 - [5.2. 右值引用](#)
 - [5.3. 函数重载](#)
 - [5.4. 缺省参数](#)
 - [5.5. 变长数组和 `alloca\(\)`](#)
 - [5.6. 友元](#)
 - [5.7. 异常](#)
 - [5.8. 运行时类型识别](#)
 - [5.9. 类型转换](#)
 - [5.10. 流](#)
 - [5.11. 前置自增和自减](#)
 - [5.12. `const` 用法](#)
 - [5.13. `constexpr` 用法](#)
 - [5.14. 整型](#)
 - [5.15. 64 位下的可移植性](#)
 - [5.16. 预处理宏](#)
 - [5.17. `0`, `nullptr` 和 `NULL`](#)
 - [5.18. `sizeof`](#)
 - [5.19. `auto`](#)
 - [5.20. 列表初始化](#)
 - [5.21. Lambda 表达式](#)
 - [5.22. 模板元编程](#)
 - [5.23. Boost 库](#)
 - [5.24. C++11](#)
 - [译者\(acgtyrant\)笔记](#)
- [6. 命名约定](#)
 - [6.1. 通用命名规则](#)
 - [6.2. 文件命名](#)
 - [6.3. 类型命名](#)
 - [6.4. 变量命名](#)
 - [6.5. 常量命名](#)
 - [6.6. 函数命名](#)
 - [6.7. 名字空间命名](#)
 - [6.8. 枚举命名](#)
 - [6.9. 宏命名](#)
 - [6.10. 命名规则的特例](#)
 - [译者\(acgtyrant\)笔记](#)
- [7. 注释](#)
 - [7.1. 注释风格](#)
 - [7.2. 文件注释](#)
 - [7.3. 类注释](#)
 - [7.4. 函数注释](#)
 - [7.5. 变量注释](#)
 - [7.6. 实现注释](#)
 - [7.7. 标点, 拼写和语法](#)
 - [7.8. TODO 注释](#)

- [7.9. 弃用注释](#)
- [译者 \(YuleFox\) 笔记](#)
- [8. 格式](#)
 - [8.1. 行长度](#)
 - [8.2. 非 ASCII 字符](#)
 - [8.3. 空格还是制表位](#)
 - [8.4. 函数声明与定义](#)
 - [8.5. Lambda 表达式](#)
 - [8.6. 函数调用](#)
 - [8.7. 列表初始化格式](#)
 - [8.8. 条件语句](#)
 - [8.9. 循环和开关选择语句](#)
 - [8.10. 指针和引用表达式](#)
 - [8.11. 布尔表达式](#)
 - [8.12. 函数返回值](#)
 - [8.13. 变量及数组初始化](#)
 - [8.14. 预处理指令](#)
 - [8.15. 类格式](#)
 - [8.16. 构造函数初始值列表](#)
 - [8.17. 名字空间格式化](#)
 - [8.18. 水平留白](#)
 - [8.19. 垂直留白](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
- [9. 规则特例](#)
 - [9.1. 现有不合规范的代码](#)
 - [9.2. Windows 代码](#)
- [10. 结束语](#)
- [Objective-C 风格指南 - 内容目录](#)
 - [Google Objective-C Style Guide 中文版](#)
 - [译者的话](#)
 - [ewanke](#)
 - [Yang.Y](#)
 - [背景介绍](#)
 - [例子](#)
 - [留白和格式](#)
 - [空格 vs. 制表符](#)
 - [行宽](#)
 - [方法声明和定义](#)
 - [方法调用](#)
 - [@public 和 @private](#)
 - [异常](#)
 - [协议名](#)
 - [块\(闭包\)](#)
 - [命名](#)
 - [文件名](#)
 - [Objective-C++](#)

- [类名](#)
 - [类别名](#)
 - [Objective-C 方法名](#)
 - [变量名](#)
 - [普通变量名](#)
 - [实例变量](#)
 - [常量](#)
- [注释](#)
 - [文件注释](#)
 - [版权信息及作者](#)
 - [声明部分的注释](#)
 - [实现部分的注释](#)
 - [对象所有权](#)
- [Cocoa 和 Objective-C 特性](#)
 - [成员变量应该是 @private](#)
 - [明确指定构造函数](#)
 - [重载指定构造函数](#)
 - [重载 NSObject 的方法](#)
 - [初始化](#)
 - [避免 +new](#)
 - [保持公共 API 简单](#)
 - [#import and #include](#)
 - [使用根框架](#)
 - [构建时即设定 autorelease](#)
 - [autorelease 优先 retain 其次](#)
 - [init 和 dealloc 内避免使用访问器](#)
 - [按声明顺序销毁实例变量](#)
 - [setter 应复制 NSStrings](#)
 - [避免抛异常](#)
 - [nil 检查](#)
 - [BOOL 若干陷阱](#)
 - [属性\(Property\)](#)
 - [命名](#)
 - [位置](#)
 - [字符串应使用 copy 属性\(Attribute\)](#)
 - [原子性](#)
 - [点引用](#)
 - [没有实例变量的接口](#)
 - [自动 synthesize 实例变量](#)
- [Cocoa 模式](#)
 - [委托模式](#)
 - [模型/视图/控制器\(MVC\)](#)
- [Python 风格指南 - 内容目录](#)
 - [扉页](#)
 - [背景](#)
 - [Python语言规范](#)
 - [Lint](#)

- [导入](#)
- [包](#)
- [异常](#)
- [全局变量](#)
- [嵌套/局部/内部类或函数](#)
- [列表推导\(List Comprehensions\)](#)
- [默认迭代器和操作符](#)
- [生成器](#)
- [Lambda函数](#)
- [条件表达式](#)
- [默认参数值](#)
- [属性\(properties\)](#)
- [True/False的求值](#)
- [过时的语言特性](#)
- [词法作用域\(Lexical Scoping\)](#)
- [函数与方法装饰器](#)
- [线程](#)
- [威力过大的特性](#)
- [Python风格规范](#)
 - [分号](#)
 - [行长度](#)
 - [括号](#)
 - [缩进](#)
 - [空行](#)
 - [空格](#)
 - [Shebang](#)
 - [注释](#)
 - [类](#)
 - [字符串](#)
 - [文件和sockets](#)
 - [TODO注释](#)
 - [导入格式](#)
 - [语句](#)
 - [访问控制](#)
 - [命名](#)
 - [Main](#)
- [临别赠言](#)
- [Shell 风格指南 - 内容目录](#)
 - [扉页](#)
 - [背景](#)
 - [使用哪一种Shell](#)
 - [什么时候使用Shell](#)
 - [Shell文件和解释器调用](#)
 - [文件扩展名](#)
 - [SUID / SGID](#)
 - [环境](#)
 - [STDOUT vs STDERR](#)

- [注释](#)
 - [文件头](#)
 - [功能注释](#)
 - [实现部分的注释](#)
 - [TODO注释](#)
- [格式](#)
 - [缩进](#)
 - [行的长度和长字符串](#)
 - [管道](#)
 - [循环](#)
 - [case语句](#)
 - [变量扩展](#)
 - [引用](#)
- [特性及错误](#)
 - [命令替换](#)
 - [test,\[和\[\[](#)
 - [测试字符串](#)
 - [文件名的通配符扩展](#)
 - [Eval](#)
 - [管道导向while循环](#)
- [命名转换](#)
 - [函数名](#)
 - [变量名](#)
 - [常量和环境变量名](#)
 - [源文件名](#)
 - [只读变量](#)
 - [使用本地变量](#)
 - [函数位置](#)
 - [主函数main](#)
- [调用命令](#)
 - [检查返回值](#)
 - [内建命令和外部命令](#)
- [结论](#)

Google 开源项目风格指南 (中文版)

- 在线文档托管在 ReadTheDocs : [在线阅读最新版本](http://zh-google-styleguide.readthedocs.org/) [http://zh-google-styleguide.readthedocs.org/]
- 中文风格指南 GitHub 托管地址: [zh-google-styleguide](https://github.com/zh-google-styleguide/zh-google-styleguide) [https://github.com/zh-google-styleguide/zh-google-styleguide]

Note

声明.

本项目并非 Google 官方项目, 而是由国内程序员凭热情创建和维护.

如果你关注的是 Google 官方英文版, 请移步 [Google Style Guide](https://github.com/google/styleguide) [https://github.com/google/styleguide]

每个较大的开源项目都有自己的风格指南: 关于如何为该项目编写代码的一系列约定 (有时候会比较武断). 当所有代码均保持一致的风格, 在理解大型代码库时更为轻松.

“风格”的含义涵盖范围广, 从“变量使用驼峰格式 (camelCase)”到“决不使用全局变量”再到“决不使用异常”. 英文版项目维护的是在 Google 使用的编程风格指南. 如果你正在修改的项目源自 Google, 你可能会被引导至英文版项目页面, 以了解项目所使用的风格.

我们已经发布了五份 中文版 的风格指南:

1. [Google C++ 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-cpp-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-cpp-styleguide/]
2. [Google Objective-C 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-objc-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-objc-styleguide/]
3. [Google Python 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-python-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-python-styleguide/]
4. [Google JSON 风格指南](https://github.com/darcyliu/google-styleguide/blob/master/JSONStyleGuide.md) [https://github.com/darcyliu/google-styleguide/blob/master/JSONStyleGuide.md]
5. [Google Shell 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-shell-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-shell-styleguide/]

中文版项目采用 reStructuredText 纯文本标记语法, 并使用 Sphinx 生成 HTML / CHM / PDF 等文档格式.

- 英文版项目还包含 [cpplint](https://github.com/google/styleguide/tree/gh-pages/cpplint) [https://github.com/google/styleguide/tree/gh-pages/cpplint] - 一个用来帮助适应风格准则的工具, 以及 [google-c-style.el](https://raw.githubusercontent.com/google/styleguide/gh-pages/google-c-style.el) [https://raw.githubusercontent.com/google/styleguide/gh-pages/google-c-style.el], Google 风格的 Emacs 配置文件.
- 另外, 招募志愿者翻译 [JavaScript Style Guide](http://google.github.io/styleguide/javascriptguide.xml) [http://google.github.io/styleguide/javascriptguide.xml] 以及 [XML Document Format Style Guide](http://google.github.io/styleguide/xmlstyle.html) [http://google.github.io/styleguide/xmlstyle.html], 有意者请联系 [Yang.Y](https://github.com/yangyubo) [https://github.com/yangyubo].

C++ 风格指南 - 内容目录

Contents

- [C++ 风格指南 - 内容目录](#)

- [0. 扉页](#)
 - [0.1 译者前言](#)
 - [0.2 背景](#)
- [1. 头文件](#)
 - [1.1. Self-contained 头文件](#)
 - [1.2. #define 保护](#)
 - [1.3. 前置声明](#)
 - [1.4. 内联函数](#)
 - [1.5. #include 的路径及顺序](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
- [2. 作用域](#)
 - [2.1. 名字空间](#)
 - [2.1.1. 匿名名字空间](#)
 - [2.1.2. 具名的名字空间](#)
 - [2.2. 嵌套类](#)
 - [2.3. 非成员函数、静态成员函数和全局函数](#)
 - [2.4. 局部变量](#)
 - [2.5. 静态和全局变量](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
- [3. 类](#)
 - [3.1. 构造函数的职责](#)
 - [3.2. 初始化](#)
 - [3.3. 显式构造函数](#)
 - [3.4. 可拷贝类型和可移动类型](#)
 - [3.5. 委派和继承构造函数](#)
 - [3.6. 结构体 VS. 类](#)
 - [3.7. 继承](#)
 - [3.8. 多重继承](#)
 - [3.9. 接口](#)
 - [3.10. 运算符重载](#)
 - [3.11. 存取控制](#)
 - [3.11. 声明顺序](#)
 - [3.12. 编写简短函数](#)
 - [译者 \(YuleFox\) 笔记](#)

- [4. 来自 Google 的奇技](#)
 - [4.1. 所有权与智能指针](#)
 - [4.2. cpplint](#)
 - [译者\(acgtyrant\)笔记](#)
- [5. 其他 C++ 特性](#)
 - [5.1. 引用参数](#)
 - [5.2. 右值引用](#)
 - [5.3. 函数重载](#)
 - [5.4. 缺省参数](#)
 - [5.5. 变长数组和 alloca\(\)](#)
 - [5.6. 友元](#)
 - [5.7. 异常](#)
 - [5.8. 运行时类型识别](#)
 - [5.9. 类型转换](#)
 - [5.10. 流](#)
 - [5.11. 前置自增和自减](#)
 - [5.12. const 用法](#)
 - [5.13. constexpr 用法](#)
 - [5.14. 整型](#)
 - [5.15. 64 位下的可移植性](#)
 - [5.16. 预处理宏](#)
 - [5.17. 0, nullptr 和 NULL](#)
 - [5.18. sizeof](#)
 - [5.19. auto](#)
 - [5.20. 列表初始化](#)
 - [5.21. Lambda 表达式](#)
 - [5.22. 模板元编程](#)
 - [5.23. Boost 库](#)
 - [5.24. C++11](#)
 - [译者\(acgtyrant\)笔记](#)
- [6. 命名约定](#)
 - [6.1. 通用命名规则](#)
 - [6.2. 文件命名](#)
 - [6.3. 类型命名](#)
 - [6.4. 变量命名](#)
 - [6.5. 常量命名](#)
 - [6.6. 函数命名](#)
 - [6.7. 名字空间命名](#)
 - [6.8. 枚举命名](#)
 - [6.9. 宏命名](#)
 - [6.10. 命名规则的特例](#)
 - [译者\(acgtyrant\)笔记](#)
- [7. 注释](#)
 - [7.1. 注释风格](#)
 - [7.2. 文件注释](#)
 - [7.3. 类注释](#)
 - [7.4. 函数注释](#)

- [7.5. 变量注释](#)
- [7.6. 实现注释](#)
- [7.7. 标点, 拼写和语法](#)
- [7.8. TODO 注释](#)
- [7.9. 弃用注释](#)
- [译者 \(YuleFox\) 笔记](#)
- [8. 格式](#)
 - [8.1. 行长度](#)
 - [8.2. 非 ASCII 字符](#)
 - [8.3. 空格还是制表位](#)
 - [8.4. 函数声明与定义](#)
 - [8.5. Lambda 表达式](#)
 - [8.6. 函数调用](#)
 - [8.7. 列表初始化格式](#)
 - [8.8. 条件语句](#)
 - [8.9. 循环和开关选择语句](#)
 - [8.10. 指针和引用表达式](#)
 - [8.11. 布尔表达式](#)
 - [8.12. 函数返回值](#)
 - [8.13. 变量及数组初始化](#)
 - [8.14. 预处理指令](#)
 - [8.15. 类格式](#)
 - [8.16. 构造函数初始值列表](#)
 - [8.17. 名字空间格式化](#)
 - [8.18. 水平留白](#)
 - [8.19. 垂直留白](#)
 - [译者 \(YuleFox\) 笔记](#)
 - [译者 \(acgtyrant\) 笔记](#)
- [9. 规则特例](#)
 - [9.1. 现有不合规范的代码](#)
 - [9.2. Windows 代码](#)
- [10. 结束语](#)

0. 扉页

版本: 4.45

原作者: Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray

翻译: [YuleFox](#)
[Yang.Y](#)
[acgtyrant](#)
[lilinsanity](#)

项目主页:

- [Google Style Guide](#)
- [Google 开源项目风格指南 - 中文版](#)

0.1 译者前言

Google 经常会发布一些开源项目, 意味着会接受来自其他代码贡献者的代码. 但是如果代码贡献者的编程风格与 Google 的不一致, 会给代码阅读者和其他代码提交者造成不小的困扰. Google 因此发布了这份自己的编程风格指南, 使所有提交代码的人都能获知 Google 的编程风格.

翻译初衷:

规则的作用就是避免混乱. 但规则本身一定要权威, 有说服力, 并且是理性的. 我们所见过的大部分编程规范, 其内容或不够严谨, 或阐述过于简单, 或带有一定的武断性.

Google 保持其一贯的严谨精神, 5 万汉字的指南涉及广泛, 论证严密. 我们翻译该系列指南的主因也正是其严谨性. 严谨意味着指南的价值不仅仅局限于它罗列出的规范, 更具参考意义的是它为了列出规范而做的谨慎权衡过程.

指南不仅列出你要怎么做, 还告诉你为什么要这么做, 哪些情况下可以不这么做, 以及如何权衡其利弊. 其他团队未必要完全遵照指南亦步亦趋, 如前面所说, 这份指南是 Google 根据自身实际情况打造的, 适用于其主导的开源项目. 其他团队可以参照该指南, 或从中汲取灵感, 建立适合自身实际情况的规范.

我们在翻译的过程中, 收获颇多. 希望本系列指南中文版对你同样能有所帮助.

我们翻译时也是尽力保持严谨, 但水平所限, bug 在所难免. 有任何意见或建议, 可与我们取得联系.

中文版和英文版一样, 使用 Artistic License/GPL 开源许可.

中文版修订历史:

- 2015-08 : 热心的清华大学同学 @lilinsanity 完善了「类」章节以及其它一些小章节。至此, 对 Google CPP Style Guide 4.45 的翻译正式竣工。
- 2015-07 4.45 : acgtyrant 为了学习 C++ 的规范, 顺便重新翻译了本 C++ 风格指南, 特别是 C++11 的全新内容。排版大幅度优化, 翻译措辞更地道, 添加了新译者笔记。Google 总部 C++ 工程师 innocentim, 清华大学不愿意透露姓名的唐马儒先生, 大阪大学大学院情报科学研究科计算机科学专攻博士 farseerfc 和其它 Arch Linux 中文社区众帮了译者不少忙, 谢谢他们。因为 C++ Primer 尚未完全入门, 暂时没有翻译「类」章节和其它一些小章节。
- 2009-06 3.133 : YuleFox 的 1.0 版已经相当完善, 但原版在近一年的时间里, 其规范也发生了一些变化。

Yang.Y 与 YuleFox 一拍即合, 以项目的形式来延续中文版: [Google 开源项目风格指南 - 中文版项目](http://github.com/yangyubo/zh-google-styleguide) [http://github.com/yangyubo/zh-google-styleguide].

主要变化是同步到 3.133 最新英文版本, 做部分勘误和改善可读性方面的修改, 并改进排版效果. Yang.Y 重新翻修, YuleFox 做后续评审.

- 2008-07 1.0 : 出自 [YuleFox 的 Blog](http://www.yulefox.com/?p=207) [http://www.yulefox.com/?p=207], 很多地方摘录的也是该版本.

0.2 背景

C++ 是 Google 大部分开源项目的主要编程语言. 正如每个 C++ 程序员都知道的, C++ 有很多强大的特性, 但这种强大不可避免的导致它走向复杂, 使代码更容易产生 bug, 难以阅读和维护.

本指南的目的是通过详细阐述 C++ 注意事项来驾驭其复杂性. 这些规则在保证代码易于管理的同时, 也能高效使用 C++ 的语言特性.

风格, 亦被称作可读性, 也就是指导 C++ 编程的约定. 使用术语“风格”有些用词不当, 因为这些习惯远不止源代码文件格式化这么简单.

使代码易于管理的方法之一是加强代码一致性. 让任何程序员都可以快速读懂你的代码这点非常重要. 保持统一编程风格并遵守约定意味着可以很容易根据“模式匹配”规则来推断各种标识符的含义. 创建通用, 必需的习惯用语和模式可以使代码更容易理解. 在一些情况下可能有充分的理由改变某些编程风格, 但我们还是应该遵循一致性原则, 尽量不这么做.

本指南的另一个观点是 C++ 特性的臃肿. C++ 是一门包含大量高级特性的庞大语言. 某些情况下, 我们会限制甚至禁止使用某些特性. 这么做是为了保持代码清爽, 避免这些特性可能导致的各种问题. 指南中列举了这类特性, 并解释为什么这些特性被限制使用.

Google 主导的开源项目均符合本指南的规定.

注意: 本指南并非 C++ 教程, 我们假定读者已经对 C++ 非常熟悉.

1. 头文件

通常每一个 .cc 文件都有一个对应的 .h 文件. 也有一些常见例外, 如单元测试代码和只包含 main() 函数的 .cc 文件.

正确使用头文件可令代码在可读性、文件大小和性能上大为改观.

下面的规则将引导你规避使用头文件时的各种陷阱.

1.1. Self-contained 头文件

Tip

头文件应该能够自给自足 (self-contained, 也就是可以作为第一个头文件被引入), 以 .h 结尾. 至于用来插入文本的文件, 说到底它们并不是头文件, 所以应以 .inc 结尾. 不允许分离出 -inl.h 头文件的做法.

所有头文件要能够自给自足. 换言之, 用户和重构工具不需要为特别场合而包含额外的头文件. 详言之, 一个头文件要有 [1.2. #define 保护](#), 统统包含它所需要的其它头文件, 也不要求定义任何特别 symbols.

不过有一个例外, 即一个文件并不是 self-contained 的, 而是作为文本插入到代码某处. 或者, 文件内容实际上是其它头文件的特定平台 (platform-specific) 扩展部分. 这些文件就要用 .inc 文件扩展名.

如果 .h 文件声明了一个模板或内联函数, 同时也在该文件加以定义. 凡是有用到这些的 .cc 文件, 就得统统包含该头文件, 否则程序可能会在构建中链接失败. 不要把这些定义放到分离的 -inl.h 文件里 (译者注: 过去该规范曾提倡把定义放到 -inl.h 里过).

有个例外: 如果某函数模板为所有相关模板参数显式实例化, 或本身就是某类的一个私有成员, 那么它就只能定义在实例化该模板的 .cc 文件里.

1.2. #define 保护

Tip

所有头文件都应该使用 #define 来防止头文件被多重包含, 命名格式当是:

```
<PROJECT>_<PATH>_<FILE>_H_ .
```

为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径. 例如, 项目 foo 中的头文件 foo/src/bar/baz.h 可按如下方式保护:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

1.3. 前置声明

Tip

尽可能地避免使用前置声明。使用 `#include` 包含需要的头文件即可。

定义：

所谓「前置声明」(forward declaration)是类、函数和模板的纯粹声明,没伴随着其定义。

优点：

- 前置声明能够节省编译时间,多余的 `#include` 会迫使编译器展开更多的文件,处理更多的输入。
- 前置声明能够节省不必要的重新编译的时间。`#include` 使代码因为头文件中无关的改动而被重新编译多次。

缺点：

- 前置声明隐藏了依赖关系,头文件改动时,用户的代码会跳过必要的重新编译过程。
- 前置声明可能会被库的后续更改所破坏。前置声明函数或模板有时会妨碍头文件开发者变动其 API。例如扩大形参类型,加个自带默认参数的模板形参等等。
- 前置声明来自命名空间 `std::` 的 symbol 时,其行为未定义。
- 很难判断什么时候该用前置声明,什么时候该用 `#include`。极端情况下,用前置声明代替 `includes` 甚至都会暗暗地改变代码的含义：

如果 `#include` 被 B 和 D 的前置声明替代, `test()` 就会调用 `f(void*)`。* 前置声明了不少来自头文件的 symbol 时,就会比单单一行的 `include` 冗长。* 仅仅为了能前置声明而重构代码(比如用指针成员代替对象成员)会使代码变得更慢更复杂。

结论：

- 尽量避免前置声明那些定义在其他项目中的实体。
- 函数:总是使用 `#include`。
- 类模板:优先使用 `#include`。

至于什么时候包含头文件,参见 `name-and-order-of-includes`。

1.4. 内联函数

Tip

只有当函数只有 10 行甚至更少时才将其定义为内联函数。

定义:

当函数被声明为内联函数之后, 编译器会将其内联展开, 而不是按通常的函数调用机制进行调用。

优点:

只要内联的函数体较小, 内联该函数可以令目标代码更加高效. 对于存取函数以及其它函数体比较短, 性能关键的函数, 鼓励使用内联。

缺点:

滥用内联将导致程序变得更慢. 内联可能使目标代码量或增或减, 这取决于内联函数的大小. 内联非常短小的存取函数通常会减少代码大小, 但内联一个相当大的函数将戏剧性的增加代码大小. 现代处理器由于更好的利用了指令缓存, 小巧的代码往往执行更快。

结论:

一个较为合理的经验准则是, 不要内联超过 10 行的函数. 谨慎对待析构函数, 析构函数往往比其表面看起来要更长, 因为有隐含的成员和基类析构函数被调用!

另一个实用的经验准则: 内联那些包含循环或 switch 语句的函数常常是得不偿失 (除非在大多数情况下, 这些循环或 switch 语句从不被执行)。

有些函数即使声明为内联的也不一定会被编译器内联, 这点很重要; 比如虚函数和递归函数就不会被正常内联. 通常, 递归函数不应该声明成内联函数.(YuleFox 注: 递归调用堆栈的展开并不像循环那么简单, 比如递归层数在编译时可能是未知的, 大多数编译器都不支持内联递归函数). 虚函数内联的主要原因则是想把它函数体放在类定义内, 为了图个方便, 抑或是当作文档描述其行为, 比如精短的存取函数。

1.5. #include 的路径及顺序

Tip

使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 相关头文件, C 库, C++ 库, 其他库的 .h, 本项目内的 .h.

项目内头文件应按照项目源代码目录树结构排列, 避免使用 UNIX 特殊的快捷目录 . (当前目录) 或 .. (上级目录). 例如, google-awesome-project/src/base/logging.h 应该按如下方式包含:

```
#include "base/logging.h"
```

又如, dir/foo.cc 的主要作用是实现或测试 dir2/foo2.h 的功能, foo.cc 中包含头文件的次序如下:

1. dir2/foo2.h (优先位置, 详情如下)
2. C 系统文件
3. C++ 系统文件
4. 其他库的 .h 文件
5. 本项目内 .h 文件

这种优先的顺序排序保证当 dir2/foo2.h 遗漏某些必要的库时, dir/foo.cc 或 dir/foo_test.cc 的构建会立刻中止。因此这一条规则保证维护这些文件的人们首先看到构建中止的消息而不是维护其他包的人们。

dir/foo.cc 和 dir2/foo2.h 通常位于同一目录下 (如 base/basicctypes_unittest.cc 和 base/basicctypes.h), 但也可以放在不同目录下。

按字母顺序对头文件包含进行二次排序是不错的主意。注意较老的代码可不符合这条规则, 要在方便的时候改正它们。

您所依赖的 symbols 被哪些头文件所定义, 您就应该包含(include)哪些头文件, forward-declaration 情况除外。比如您要用到 bar.h 中的某个 symbol, 哪怕您所包含的 foo.h 已经包含了 bar.h, 也照样得包含 bar.h, 除非 foo.h 有明确说明它会自动向您提供 bar.h 中的 symbol。不过, 凡是 cc 文件所对应的「相关头文件」已经包含的, 就不用再重复包含进其 cc 文件里面了, 就像 foo.cc 只包含 foo.h 就够了, 不用再管后者所包含的其它内容。

举例来说, google-awesome-project/src/foo/internal/fooserver.cc 的包含次序如下:

```
#include "foo/public/fooserver.h" // 优先位置

#include <sys/types.h>
#include <unistd.h>

#include <hash_map>
#include <vector>

#include "base/basicctypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

例外:

有时, 平台特定(system-specific)代码需要条件编译(conditional includes), 这些代码可以放到其它 includes 之后。当然, 您的平台特定代码也要够简练且独立, 比如:

```
#include "foo/public/fooserver.h"

#include "base/port.h" // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif // LANG_CXX11
```

译者 (YuleFox) 笔记

1. 避免多重包含是学编程时最基本的要求;
2. 前置声明是为了降低编译依赖,防止修改一个头文件引发多米诺效应;
3. 内联函数的合理使用可提高代码执行效率;
4. `-in1.h` 可提高代码可读性 (一般用不到吧:D);
5. 标准化函数参数顺序可以提高可读性和易维护性 (对函数参数的堆栈空间有轻微影响, 我以前大多是相同类型放在一起);
6. 包含文件的名称使用 `.` 和 `..` 虽然方便却易混乱, 使用比较完整的项目路径看上去很清晰, 很条理, 包含文件的次序除了美观之外, 最重要的是可以减少隐藏依赖, 使每个头文件在“最需要编译” (对应源文件处 :D) 的地方编译, 有人提出库文件放在最后, 这样出错先是项目内的文件, 头文件都放在对应源文件的最前面, 这一点足以保证内部错误的及时发现了.

译者(acgtyrant)笔记

1. 原来还真有项目用 `#includes` 来插入文本, 且其文件扩展名 `.inc` 看上去也很科学。
2. Google 已经不再提倡 `-in1.h` 用法。
3. 注意, 前置声明的类是不完全类型 (incomplete type), 我们只能定义指向该类型的指针或引用, 或者声明 (但不能定义) 以不完全类型作为参数或者返回类型的函数。毕竟编译器不知道不完全类型的定义, 我们不能创建其类的任何对象, 也不能声明成类内部的数据成员。
4. 类内部的函数一般会 **自动内联**。所以某函数一旦不需要内联, 其定义就不要再放在头文件里, 而是放到对应的 `.cc` 文件里。这样可以保持头文件的类相当精炼, 也很好贯彻了声明与定义分离的原则。
5. 在 `#include` 中插入空行以分割相关头文件, C 库, C++ 库, 其他库的 `.h` 和本项目内的 `.h` 是个好习惯。

2. 作用域

2.1. 名字空间

Tip

鼓励在 .cc 文件内使用匿名名字空间. 使用具名的名字空间时, 其名称可基于项目名或相对路径. 禁止使用 using 指示(using-directive). 禁止使用内联命名空间(inline namespace)。

定义:

名字空间将全局作用域细分为独立的, 具名的作用域, 可有效防止全局作用域的命名冲突.

优点:

虽然类已经提供了(可嵌套的)命名轴线 (YuleFox 注: 将命名分割在不同类的作用域内), 名字空间在此基础上又封装了一层.

举例来说, 两个不同项目的全局作用域都有一个类 Foo, 这样在编译或运行时造成冲突. 如果每个项目将代码置于不同名字空间中, project1::Foo 和 project2::Foo 作为不同符号自然不会冲突.

内联命名空间会自动把内部的标识符放到外层作用域, 比如:

```
namespace X {  
  inline namespace Y {  
    void foo();  
  }  
}
```

X::Y::foo() 与 X::foo() 彼此可代替. 内联命名空间主要用来保持跨版本的 ABI 兼容性。

缺点:

名字空间具有迷惑性, 因为它们和类一样提供了额外的(可嵌套的)命名轴线.

命名空间很容易令人迷惑, 毕竟它们不再受其声明所在命名空间的限制. 内联命名空间只在大型版本控制里有用。

在头文件中使用匿名空间导致违背 C++ 的唯一定义原则 (One Definition Rule (ODR)).

结论:

根据下文将要提到的策略合理使用命名空间.

2.1.1. 匿名名字空间

- 在 .cc 文件中, 允许甚至鼓励使用匿名名字空间, 以避免运行时的命名冲突:

```
namespace {                                // .cc 文件中

// 名字空间的内容无需缩进
enum { kUNUSED, kEOF, kERROR };           // 经常使用的符号
bool AtEof() { return pos_ == kEOF; }     // 使用本名字空间内的符号 EOF

} // namespace
```

然而, 与特定类关联的文件作用域声明在该类中被声明为类型, 静态数据成员或静态成员函数, 而不是匿名名字空间的成员. 如上例所示, 匿名空间结束时用注释 `// namespace` 标识.

- 不要在 .h 文件中使用匿名名字空间.

2.1.2. 具名的名字空间

具名的名字空间使用方式如下:

- 用名字空间把文件包含, [gflags](http://code.google.com/p/google-gflags/) [http://code.google.com/p/google-gflags/] 的声明/定义, 以及类的前置声明以外的整个源文件封装起来, 以区别于其它名字空间:

```
// .h 文件
namespace mynamespace {

// 所有声明都置于命名空间中
// 注意不要使用缩进
class MyClass {
public:
    ...
    void Foo();
};

} // namespace mynamespace
```

```
// .cc 文件
namespace mynamespace {

// 函数定义都置于命名空间中
void MyClass::Foo() {
    ...
}

} // namespace mynamespace
```

通常的 .cc 文件包含更多, 更复杂的细节, 比如引用其他名字空间的类等.

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;                                // 全局名字空间中类 C 的前置声明
namespace a { class A; }               // a::A 的前置声明
```

```
namespace b {
...code for b...           // b 中的代码
} // namespace b
```

- 不要在名字空间 `std` 内声明任何东西, 包括**标准库**的类前置声明. 在 `std` 名字空间声明实体会导致不确定的问题, 比如不可移植. 声明**标准库**下的实体, 需要包含**对应的头文件**.
- 最好不要使用 `using` 指示, 以保证名字空间下的所有名称都可以正常使用.

```
// 禁止 — 污染名字空间
using namespace foo;
```

- 在 `.cc` 文件, `.h` 文件的函数, 方法或类中, 可以使用 `using` 声明。

```
// 允许: .cc 文件中
// .h 文件的话, 必须在函数, 方法或类的内部使用
using ::foo::bar;
```

- 在 `.cc` 文件, `.h` 文件的函数, 方法或类中, 允许使用名字空间别名.

```
// 允许: .cc 文件中
// .h 文件的话, 必须在函数, 方法或类的内部使用

namespace fbz = ::foo::bar::baz;

// 在 .h 文件里
namespace librarian {
//以下别名在所有包含了该头文件的文件中生效。
namespace pd_s = ::pipeline_diagnostics::sidetable;

inline void my_inline_function() {
    // namespace alias local to a function (or method).
    namespace fbz = ::foo::bar::baz;
    ...
}
} // namespace librarian
```

注意在 `.h` 文件的别名对包含了该头文件的所有人可见, 所以在公共头文件(在项目外可用)以及它们递归包含的其它头文件里, 不要用别名。毕竟原则上公共 API 要尽可能地精简。

- 禁止用内联命名空间

2.2. 嵌套类

Tip

当公有嵌套类作为接口的一部分时, 虽然可以直接将他们保持在全局作用域中, 但将嵌套类的声明置于 `ref: namespaces`` 内是更好的选择。

定义: 在一个类内部定义另一个类; 嵌套类也被称为 成员类 (*member class*).

```
class Foo {  
  
private:  
    // Bar是嵌套在Foo中的成员类  
    class Bar {  
        ...  
    };  
  
};
```

优点:

当嵌套 (或成员) 类只被外围类使用时非常有用; 把它作为外围类作用域内的成员, 而不是去污染外部作用域的同名类. 嵌套类可以在外围类中做前置声明, 然后在 .cc 文件中定义, 这样避免在外围类的声明中定义嵌套类, 因为嵌套类的定义通常只与实现相关.

缺点:

嵌套类只能在外围类的内部做前置声明. 因此, 任何使用了 `Foo::Bar*` 指针的头文件不得不包含类 `Foo` 的整个声明.

结论:

不要将嵌套类定义成公有, 除非它们是接口的一部分, 比如, 嵌套类含有某些方法的一组选项.

2.3. 非成员函数、静态成员函数和全局函数

Tip

使用静态成员函数或名字空间内的非成员函数, 尽量不要用裸的全局函数.

优点:

某些情况下, 非成员函数和静态成员函数是非常有用的, 将非成员函数放在名字空间内可避免污染全局作用域.

缺点:

将非成员函数和静态成员函数作为新类的成员或许更有意义, 当它们需要访问外部资源或具有重要的依赖关系时更是如此.

结论:

有时, 把函数的定义同类的实例脱钩是有益的, 甚至是必要的. 这样的函数可以被定义成静态成员, 或是非成员函数. 非成员函数不应依赖于外部变量, 应尽量置于某个名字空间内. 相比单纯为了封装若干不共享任何静态数据的静态成员函数而创建类, 不如使用 `ref.namespaces`.

定义在同一编译单元的函数, 被其他编译单元直接调用可能会引入不必要的耦合和链接时依赖; 静态成员函数对此尤其敏感. 可以考虑提取到新类中, 或者将函数置于独立库的名字空间内.

如果你必须定义非成员函数, 又只是在 .cc 文件中使用它, 可使用匿名 namespace 或 `static` 链接关键字 (如 `static int Foo() {...}`) 限定其作用域.

2.4. 局部变量

Tip

将函数变量尽可能置于最小作用域内, 并在变量声明时进行初始化.

C++ 允许在函数的任何位置声明变量. 我们提倡在尽可能小的作用域中声明变量, 离第一次使用越近越好. 这使得代码浏览者更容易定位变量声明的位置, 了解变量的类型和初始值. 特别是, 应使用初始化的方式替代声明再赋值, 比如:

```
int i;
i = f(); // 坏——初始化和声明分离
int j = g(); // 好——初始化时声明

vector<int> v;
v.push_back(1); // 用花括号初始化更好
v.push_back(2);

vector<int> v = {1, 2}; // 好——v 一开始就初始化
```

注意, GCC 可正确实现了 `for (int i = 0; i < 10; ++i)` (`i` 的作用域仅限 `for` 循环内), 所以其他 `for` 循环中可以重新使用 `i`. 在 `if` 和 `while` 等语句中的作用域声明也是正确的, 如:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

Warning

如果变量是一个对象, 每次进入作用域都要调用其构造函数, 每次退出作用域都要调用其析构函数.

```
// 低效的实现
for (int i = 0; i < 1000000; ++i) {
    Foo f; // 构造函数和析构函数分别调用 1000000 次!
    f.DoSomething(i);
}
```

在循环作用域外面声明这类变量要高效的多:

```
Foo f; // 构造函数和析构函数只调用 1 次
for (int i = 0; i < 1000000; ++i) {
    f.DoSomething(i);
}
```


2.5. 静态和全局变量

Tip

禁止使用 `class` 类型的静态或全局变量:它们会导致难以发现的 bug 和不确定的构造和析构函数调用顺序。不过 `constexpr` 变量除外,毕竟它们又不涉及动态初始化或析构。

静态生存周期的对象,即包括了全局变量,静态变量,静态类成员变量和函数静态变量,都必须是原生数据类型 (POD: Plain Old Data): 即 `int`, `char` 和 `float`, 以及 POD 类型的指针、数组和结构体。

静态变量的构造函数、析构函数和初始化的顺序在 C++ 中是不确定的,甚至随着构建变化而变化,导致难以发现的 bug. 所以除了禁用类类型的全局变量,我们也不允许用函数返回值来初始化 POD 变量,除非该函数不涉及(比如 `getenv()` 或 `getpid()`)不涉及任何全局变量。(函数作用域里的静态变量除外,毕竟它的初始化顺序是有明确定义的,而且只会在指令执行到它的声明那里才会发生。)

同理,全局和静态变量在程序中断时会被析构,无论所谓中断是从 `main()` 返回还是对 `exit()` 的调用。析构顺序正好与构造函数调用的顺序相反。但既然构造顺序未定义,那么析构顺序当然也就不定了。比如,在程序结束时某静态变量已经被析构了,但代码还在跑——比如其它线程——并试图访问它且失败;再比如,一个静态 `string` 变量也许会在一个引用了前者的其它变量析构之前被析构掉。

改善以上析构问题的办法之一是用 `quick_exit()` 来代替 `exit()` 并中断程序。它们的不同之处是前者不会执行任何析构,也不会执行 `atexit()` 所绑定的任何 handlers. 如果您想在执行 `quick_exit()` 来中断时执行某 handler(比如刷新 log),您可以把它绑定到 `_at_quick_exit()`. 如果您想在 `exit()` 和 `quick_exit()` 都用上该 handler,都绑定上去。

综上所述,我们只允许 POD 类型的静态变量,即完全禁用 `vector` (使用 C 数组替代) 和 `string` (使用 `const char []`)。

如果您确实需要一个 `class` 类型的静态或全局变量,可以考虑在 `main()` 函数或 `pthread_once()` 内初始化一个指针且永不回收。注意只能用 raw 指针,别用智能指针,毕竟后者的析构函数涉及到上文指出的不定顺序问题。

Note

Yang.Y 译注:

上文提及的静态变量泛指静态生存周期的对象,包括: 全局变量, 静态变量, 静态类成员变量, 以及函数静态变量.

译者 (YuleFox) 笔记

1. `cc` 中的匿名名字空间可避免命名冲突, 限定作用域, 避免直接使用 `using` 关键字污染命名空

间;

2. 嵌套类符合局部使用原则, 只是不能在其他头文件中前置声明, 尽量不要 `public`;
3. 尽量不用全局函数和全局变量, 考虑作用域和命名空间限制, 尽量单独形成编译单元;
4. 多线程中的全局变量 (含静态成员变量) 不要使用 `class` 类型 (含 STL 容器), 避免不明确行为导致的 bug.
5. 作用域的使用, 除了考虑名称污染, 可读性之外, 主要是为降低耦合, 提高编译/执行效率.

译者(acgtyrant)笔记

1. 注意「using 指示(using-directive)」和「using 声明(using-declaration)」的区别。
2. 匿名名字空间说白了就是文件作用域, 就像 C `static` 声明的作用域一样, 后者已经被 C++ 标准提倡弃用。
3. 局部变量在声明的同时进行显式值初始化, 比起隐式初始化再赋值的两步过程要高效, 同时也贯彻了计算机体系结构重要的概念「局部性(locality)」。
4. 注意别在循环犯大量构造和析构的低级错误。

3. 类

类是 C++ 中代码的基本单元. 显然, 它们被广泛使用. 本节列举了在写一个类时的主要注意事项.

3.1. 构造函数的职责

Tip

不要在构造函数中进行复杂的初始化 (尤其是那些有可能失败或者需要调用虚函数的初始化).

定义:

在构造函数体中进行初始化操作.

优点:

排版方便, 无需担心类是否已经初始化.

缺点:

在构造函数中执行操作引起的问题有:

- 构造函数中很难上报错误, [不能使用异常](#).
- 操作失败会造成对象初始化失败, 进入不确定状态.
- 如果在构造函数内调用了自身的虚函数, 这类调用是不会重定向到子类的虚函数实现. 即使当前没有子类化实现, 将来仍是隐患.
- 如果有人创建该类型的全局变量 (虽然违背了上节提到的规则), 构造函数将先 `main()` 一步被调用, 有可能破坏构造函数中暗含的假设条件. 例如, [gflags](#) [<http://code.google.com/p/google-gflags/>] 尚未初始化.

结论:

构造函数不得调用虚函数, 或尝试报告一个非致命错误. 如果对象需要进行有意义的 (non-trivial) 初始化, 考虑使用明确的 `Init()` 方法或使用工厂模式.

3.2. 初始化

Tip

如果类中定义了成员变量, 则必须在类中为每个类提供初始化函数或定义一个构造函数. 若未声明构造函数, 则编译器会生成一个默认的构造函数, 这有可能导致某些成员未被初始化或被初始化为不恰当的值.

定义:

`new` 一个不带参数的类对象时, 会调用这个类的默认构造函数. 用 `new[]` 创建数组时, 默认构造函数则总是被调用. 在类成员里面进行初始化是指声明一个成员变量的时候使用一个结构例如 `int _count = 17` 或者 `string _name{"abc"}` 来替代 `int _count` 或者 `string _name` 这样的形式.

优点:

用户定义的默认构造函数将在没有提供初始化操作时将对象初始化. 这样就保证了对象在被构造之时就处于一个有效且可用的状态, 同时保证了对象在被创建时就处于一个显然“不可能”的状态, 以此帮助调试.

缺点:

对代码编写者来说, 这是多余的工作.

如果一个成员变量在声明时初始化又在构造函数中初始化, 有可能造成混乱, 因为构造函数中的值会覆盖掉声明中的值.

结论:

简单的初始化用类成员初始化完成, 尤其是当一个成员变量要在多个构造函数里用相同的方式初始化的时候.

如果你的类中有成员变量没有在类里面进行初始化, 而且没有提供其它构造函数, 你必须定义一个 (不带参数的) 默认构造函数. 把对象的内部状态初始化成一致 / 有效的值无疑是更合理的方式.

这么做的原因是: 如果你没有提供其它构造函数, 又没有定义默认构造函数, 编译器将为你自动生成一个. 编译器生成的构造函数并不会对对象进行合理的初始化.

如果你定义的类型继承现有类, 而你又没有增加新的成员变量, 则不需要为新类定义默认构造函数.

3.3. 显式构造函数

Tip

对单个参数的构造函数使用 C++ 关键字 `explicit`.

定义:

通常, 如果构造函数只有一个参数, 可看成是一种隐式转换. 打个比方, 如果你定义了 `Foo::Foo(string name)`, 接着把一个字符串传给一个以 `Foo` 对象为参数的函数, 构造函数

`Foo::Foo(string name)` 将被调用, 并将该字符串转换为一个 `Foo` 的临时对象传给调用函数. 看上去很方便, 但如果你并不希望如此通过转换生成一个新对象的话, 麻烦也随之而来. 为避免构造函数被调用造成隐式转换, 可以将其声明为 `explicit`.

除单参数构造函数外, 这一规则也适用于除第一个参数以外的其他参数都具有默认参数的构造函数, 例如 `Foo::Foo(string name, int id = 42)`.

优点:

避免不合时宜的变换.

缺点:

无

结论:

所有单参数构造函数都必须是显式的. 在类定义中, 将关键字 `explicit` 加到单参数构造函数前: `explicit Foo(string name);`

例外: 在极少数情况下, 拷贝构造函数可以不声明成 `explicit`. 作为其它类的透明包装器的类也是特例之一. 类似的例外情况应在注释中明确说明.

最后, 只有 `std::initializer_list` 的构造函数可以是非 `explicit`, 以允许你的类型结构可以使用列表初始化的方式进行赋值. 例如:

```
MyType m = {1, 2};
MyType MakeMyType() { return {1, 2}; }
TakeMyType({1, 2});
```

3.4. 可拷贝类型和可移动类型

Tip

如果你的类型需要, 就让它们支持拷贝 / 移动. 否则, 就把隐式产生的拷贝和移动函数禁用.

定义:

可拷贝类型允许对象在初始化时得到来自相同类型的另一对象的值, 或在赋值时被赋予相同类型的另一对象的值, 同时不改变源对象的值. 对于用户定义的类型, 拷贝操作一般通过拷贝构造函数与拷贝赋值操作符定义. `string` 类型就是一个可拷贝类型的例子.

可移动类型允许对象在初始化时得到来自相同类型的临时对象的值, 或在赋值时被赋予相同类型的临时对象的值 (因此所有可拷贝对象也是可移动的). `std::unique_ptr<int>` 就是一个可移动但不可复制的对象的例子. 对于用户定义的类型, 移动操作一般是通过移动构造函数和移动赋值操作符实现的.

拷贝 / 移动构造函数在某些情况下会被编译器隐式调用. 例如, 通过传值的方式传递对象.

优点:

可移动及可拷贝类型的对象可以通过传值的方式进行传递或者返回, 这使得 API 更简单, 更安全也更通用. 与传指针和引用不同, 这样的传递不会造成所有权, 生命周期, 可变性等方面的混乱, 也就没必要在协议中予以明确. 这同时也防止了客户端与实现在非作用域内的交互, 使得它们更容易被理解与维护. 这样的对象可以和需要传值操作的通用 API 一起使用, 例如大多数容器.

拷贝 / 移动构造函数与赋值操作一般来说要比它们的各种替代方案, 比如 Clone(), CopyFrom() or Swap(), 更容易定义, 因为它们能通过编译器产生, 无论是隐式的还是通过 = 默认. 这种方式很简洁, 也保证所有数据成员都会被复制. 拷贝与移动构造函数一般也更高效, 因为它们不需要堆的分配或者是单独的初始化和赋值步骤, 同时, 对于类似省略不必要的拷贝这样的优化它们也更加合适.

移动操作允许隐式且高效地将源数据转移出右值对象. 这有时能让代码风格更加清晰.

缺点:

许多类型都不需要拷贝, 为它们提供拷贝操作会让人迷惑, 也显得荒谬而不合理. 为基类提供拷贝 / 赋值操作是有害的, 因为在使用它们时会造成对象切割. 默认的或者随意的拷贝操作实现可能是不正确的, 这往往导致令人困惑并且难以诊断出的错误.

拷贝构造函数是隐式调用的, 也就是说, 这些调用很容易被忽略. 这会让人迷惑, 尤其是对那些所用的语言约定或强制要求传引用的程序员来说更是如此. 同时, 这从一定程度上说会鼓励过度拷贝, 从而导致性能上的问题.

结论:

如果需要就让你的类型可拷贝 / 可移动. 作为一个经验法则, 如果对于你的用户来说这个拷贝操作不是一眼就能看出来的, 那就不要把类型设置为可拷贝. 如果让类型可拷贝, 一定要同时给出拷贝构造函数和赋值操作的定义. 如果让类型可拷贝, 同时移动操作的效率高于拷贝操作, 那么就把移动的两个操作 (移动构造函数和赋值操作) 也给出定义. 如果类型不可拷贝, 但是移动操作的正确性对用户显然可见, 那么把这个类型设置为只可移动并定义移动的两个操作.

建议通过 = default 定义拷贝和移动操作. 定义非默认的移动操作目前需要异常. 时刻记得检测默认操作的正确性. 由于存在对象切割的风险, 不要为任何有可能有派生类的对象提供赋值操作或者拷贝 / 移动构造函数 (当然也不要继承有这样的成员函数的类). 如果你的基类需要可复制属性, 请提供一个 public virtual Clone() 和一个 protected 的拷贝构造函数以供派生类实现.

如果你的类不需要拷贝 / 移动操作, 请显式地通过 = delete 或其他手段禁用之.

3.5. 委派和继承构造函数

Tip

在能够减少重复代码的情况下使用委派和继承构造函数.

定义:

委派和继承构造函数是由 C++11 引进为了减少构造函数重复代码而开发的两种不同的特性. 通过特殊的初始化列表语法, 委派构造函数允许类的一个构造函数调用其他的构造函数. 例如:

```
X::X(const string& name) : name_(name) {  
    ...  
}  
  
X::X() : X("") { }
```

继承构造函数允许派生类直接调用基类的构造函数, 一如继承基类的其他成员函数, 而无需重新声明. 当基类拥有多个构造函数时这一功能尤其有用. 例如:

```
class Base {  
public:  
    Base();  
    Base(int n);  
    Base(const string& s);  
    ...  
};  
  
class Derived : public Base {  
public:  
    using Base::Base; // Base's constructors are redeclared here.  
};
```

如果派生类的构造函数只是调用基类的构造函数而没有其他行为时, 这一功能特别有用.

优点:

委派和继承构造函数可以减少冗余代码, 提高可读性. 委派构造函数对 Java 程序员来说并不陌生.

缺点:

使用辅助函数可以预估出委派构造函数的行为. 如果派生类和基类相比引入了新的成员变量, 继承构造函数就会让人迷惑, 因为基类并不知道这些新的成员变量的存在.

结论:

只在能够减少冗余代码, 提高可读性的前提下使用委派和继承构造函数. 如果派生类有新的成员变量, 那么使用继承构造函数时要小心. 如果在派生类中对成员变量使用了类内部初始化的话, 继承构造函数还是适用的.

3.6. 结构体 VS. 类

Tip

仅当只有数据时使用 struct, 其它一概使用 class.

说明:

在 C++ 中 struct 和 class 关键字几乎含义一样. 我们为这两个关键字添加我们自己的语义理解, 以便未定义的数据类型选择合适的关键字.

struct 用来定义包含数据的被动式对象, 也可以包含相关的常量, 但除了存取数据成员之外, 没有别的函数功能. 并且存取功能是通过直接访问位域, 而非函数调用. 除了构造函数, 析构函数, Initialize(), Reset(), Validate() 等类似的函数外, 不能提供其它功能的函数.

如果需要更多的函数功能, class 更适合. 如果拿不准, 就用 class.

为了和 STL 保持一致, 对于仿函数和 trait 特性可以不用 class 而是使用 struct.

注意: 类和结构体的成员变量使用不同的命名规则.

3.7. 继承

Tip

使用组合 (composition, YuleFox 注: 这一点也是 GoF 在 <<Design Patterns>> 里反复强调的) 常常比使用继承更合理. 如果使用继承的话, 定义为 public 继承.

定义:

当子类继承基类时, 子类包含了父基类所有数据及操作的定义. C++ 实践中, 继承主要用于两种场合: 实现继承 (implementation inheritance), 子类继承父类的实现代码; 接口继承 (interface inheritance), 子类仅继承父类的方法名称.

优点:

实现继承通过原封不动的复用基类代码减少了代码量. 由于继承是在编译时声明, 程序员和编译器都可以理解相应操作并发现错误. 从编程角度而言, 接口继承是用来强制类输出特定的 API. 在类没有实现 API 中某个必须的方法时, 编译器同样会发现并报告错误.

缺点:

对于实现继承, 由于子类的实现代码散布在父类和子类间之间, 要理解其实现变得更加困难. 子类不能重写父类的非虚函数, 当然也就不能修改其实现. 基类也可能定义了一些数据成员, 还要区分基类的实际布局.

结论:

所有继承必须是 public 的. 如果你想使用私有继承, 你应该替换成把基类的实例作为成员对象的方式.

不要过度使用实现继承. 组合常常更合适一些. 尽量做到只在“是一个”(“is-a”, YuleFox 注: 其他“has-a”情况下请使用组合) 的情况下使用继承: 如果 Bar 的确“是一种”Foo, Bar 才能继承 Foo.

必要的话, 析构函数声明为 virtual. 如果你的类有虚函数, 则析构函数也应该为虚函数. 注意[数据成员在任何情况下都必须是私有的](#).

当重载一个虚函数, 在衍生类中把它明确的声明为 virtual. 理论依据: 如果省略 virtual 关键字, 代码阅读者不得不检查所有父类, 以判断该函数是否是虚函数.

3.8. 多重继承

Tip

真正需要用到多重实现继承的情况少之又少. 只在以下情况我们才允许多重继承: 最多只有一个基类是非抽象类; 其它基类都是以 Interface 为后缀的[纯接口类](#).

定义:

多重继承允许子类拥有多个基类. 要将作为纯接口的基类和具有实现的基类区别开来.

优点:

相比单继承 (见[继承](#)), 多重实现继承可以复用更多的代码.

缺点:

真正需要用到多重实现继承的情况少之又少. 多重实现继承看上去是不错的解决方案, 但你通常也可以找到一个更明确, 更清晰的不同解决方案.

结论:

只有当所有父类除第一个外都是[纯接口类](#)时, 才允许使用多重继承. 为确保它们是纯接口, 这些类必须以 Interface 为后缀.

Note

关于该规则, Windows 下有个[特例](#).

3.9. 接口

Tip

接口是指满足特定条件的类, 这些类以 `Interface` 为后缀 (不强制).

定义:

当一个类满足以下要求时, 称之为纯接口:

- 只有纯虚函数 (“=0”) 和静态函数 (除了下文提到的析构造函数).
- 没有非静态数据成员.
- 没有定义任何构造函数. 如果有, 也不能带有参数, 并且必须为 `protected`.
- 如果它是一个子类, 也只能从满足上述条件并以 `Interface` 为后缀的类继承.

接口类不能被直接实例化, 因为它声明了纯虚函数. 为确保接口类的所有实现可被正确销毁, 必须为之声明虚析构造函数 (作为上述第 1 条规则的特例, 析构造函数不能是纯虚函数). 具体细节可参考 Stroustrup 的 *The C++ Programming Language, 3rd edition* 第 12.4 节.

优点:

以 `Interface` 为后缀可以提醒其他人不要为该接口类增加函数实现或非静态数据成员. 这一点对于 [多重继承](#) 尤其重要. 另外, 对于 Java 程序员来说, 接口的概念已是深入人心.

缺点:

`Interface` 后缀增加了类名长度, 为阅读和理解带来不便. 同时, 接口特性作为实现细节不应暴露给用户.

结论:

只有在满足上述需要时, 类才以 `Interface` 结尾, 但反过来, 满足上述需要的类未必一定以 `Interface` 结尾.

3.10. 运算符重载

Tip

除少数特定环境外, 不要重载运算符.

定义:

一个类可以定义诸如 `+` 和 `/` 等运算符, 使其可以像内建类型一样直接操作.

优点:

使代码看上去更加直观, 类表现的和内建类型 (如 `int`) 行为一致. 重载运算符使 `Equals()`,

Add() 等函数名黯然失色. 为了使一些模板函数正确工作, 你可能必须定义操作符.

缺点:

虽然操作符重载令代码更加直观, 但也有一些不足:

- 混淆视听, 让你误以为一些耗时的操作和操作内建类型一样轻巧.
- 更难定位重载运算符的调用点, 查找 Equals() 显然比对应的 == 调用点要容易的多.
- 有的运算符可以对指针进行操作, 容易导致 bug. Foo + 4 做的是一件事, 而 &Foo + 4 可能做的是完全不同的另一件事. 对于二者, 编译器都不会报错, 使其很难调试;

重载还有令你吃惊的副作用. 比如, 重载了 operator& 的类不能被前置声明.

结论:

一般不要重载运算符. 尤其是赋值操作 (operator=) 比较诡异, 应避免重载. 如果需要的话, 可以定义类似 Equals(), CopyFrom() 等函数.

然而, 极少数情况下可能需要重载运算符以便与模板或 “标准” C++ 类互操作 (如 operator<<(ostream&, const T&)). 只有被证明是完全合理的才能重载, 但你还是要尽可能避免这样做. 尤其是不要仅仅为了在 STL 容器中用作键值就重载 operator== 或 operator<; 相反, 你应该在声明容器的时候, 创建相等判断和大小比较的仿函数类型.

有些 STL 算法确实需要重载 operator== 时, 你可以这么做, 记得别忘了在文档中说明原因.

参考 [拷贝构造函数](#) 和 [函数重载](#).

3.11. 存取控制

Tip

将所有数据成员声明为 private, 并根据需要提供相应的存取函数. 例如, 某个名为 foo_ 的变量, 其取值函数是 foo(). 还可能需要一个赋值函数 set_foo().

特例是, 静态常量数据成员 (一般写做 kFoo) 不需要是私有成员.

一般在头文件中把存取函数定义成内联函数.

参考 [继承](#) 和 [函数命名](#)

3.11. 声明顺序

Tip

在类中使用特定的声明顺序: public: 在 private: 之前, 成员函数在数据成员 (变量) 前;

类的访问控制区段的声明顺序依次为: public:, protected:, private:. 如果某区段没内容, 可以不声明.

每个区段内的声明通常按以下顺序:

- typedefs 和枚举
- 常量
- 构造函数
- 析构函数
- 成员函数, 含静态成员函数
- 数据成员, 含静态数据成员

友元声明应该放在 private 区段. 如果用宏 DISALLOW_COPY_AND_ASSIGN 禁用拷贝和赋值, 应当将其置于 private 区段的末尾, 也即整个类声明的末尾. 参见可拷贝类型和可移动类型.

.cc 文件中函数的定义应尽可能和声明顺序一致.

不要在类定义中内联大型函数. 通常, 只有那些没有特别意义或性能要求高, 并且是比较短小的函数才能被定义为内联函数. 更多细节参考 [内联函数](#).

3.12. 编写简短函数

Tip

倾向编写简短, 凝练的函数.

我们承认长函数有时是合理的, 因此并不硬性限制函数的长度. 如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割.

即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题. 甚至导致难以发现的 bug. 使函数尽量简短, 便于他人阅读和修改代码.

在处理代码时, 你可能会发现复杂的长函数. 不要害怕修改现有代码: 如果证实这些代码使用 / 调试困难, 或者你需要使用其中的一小段代码, 考虑将其分割为更加简短并易于管理的若干函数.

译者 (YuleFox) 笔记

1. 不在构造函数中做太多逻辑相关的初始化;
2. 编译器提供的默认构造函数不会对变量进行初始化, 如果定义其他构造函数, 编译器不再提供, 需要编码者自行提供默认构造函数;
3. 为避免隐式转换, 需将单参数构造函数声明为 explicit;
4. 为避免拷贝构造函数, 赋值操作的滥用和编译器自动生成, 可将其声明为 private 且无需实现;
5. 仅在作为数据集合时使用 struct;
6. 组合 > 实现继承 > 接口继承 > 私有继承, 子类重载的虚函数也要声明 virtual 关键字, 虽然编译器允许不这样做;

7. 避免使用多重继承, 使用时, 除一个基类含有实现外, 其他基类均为纯接口;
8. 接口类名以 `Interface` 为后缀, 除提供带实现的虚析构函数, 静态成员函数外, 其他均为纯虚函数, 不定义非静态数据成员, 不提供构造函数, 提供的话, 声明为 `protected`;
9. 为降低复杂性, 尽量不重载操作符, 模板, 标准类中使用时提供文档说明;
10. 存取函数一般内联在头文件中;
11. 声明次序: `public -> protected -> private`;
12. 函数体尽量短小, 紧凑, 功能单一;

4. 来自 Google 的奇技

Google 用了很多自己实现的技巧 / 工具使 C++ 代码更加健壮, 我们使用 C++ 的方式可能和你在其它地方见到的有所不同.

4.1. 所有权与智能指针

Tip

动态分配出的对象最好有单一且固定的所有主(owner), 且通过智能指针传递所有权(ownership).

定义:

所有权是一种登记/管理动态内存和其它资源的技术。动态分配出的对象的所有主是一个对象或函数, 后者负责确保当前者无用时就自动销毁前者。所有权有时可以共享, 那么就由最后一个所有主来负责销毁它。甚至也可以不用共享, 在代码中直接把所有权传递给其它对象。

其实您可以把智能指针当成一个重载了 * 和 -> 的「对象」来看。智能指针类型被用来自动化所有权的登记工作, 来确保执行销毁义务到位。[std::unique_ptr](#)

[http://en.cppreference.com/w/cpp/memory/unique_ptr] 是 C++11 新推出的一种智能指针类型, 用来表示动态分配出的对象的「独一无二」所有权; 当 `std::unique_ptr` 离开作用域, 对象就会被销毁。不能复制 `std::unique_ptr`, 但可以把它移动(move)给新所有主。[std::shared_ptr](#)

[http://en.cppreference.com/w/cpp/memory/shared_ptr] 同样表示动态分配对象的所有权, 但可以被共享, 也可以被复制; 对象的所有权由所有复制者共同拥有, 最后一个复制者被销毁时, 对象也会随着被销毁。

优点:

- 如果没有清晰、逻辑条理的所有权安排, 不可能管理好动态分配的内存。
- 传递对象的所有权, 开销比复制来得小, 如果可以复制的话。
- 传递所有权也比「借用」指针或引用来得简单, 毕竟它大大省去了两个用户一起协调对象生命周期的工作。
- 如果所有权逻辑条理, 有文档且不乱来的话, 可读性很棒。
- 可以不用手动完成所有权的登记工作, 大大简化了代码, 也免去了一大波错误之恼。
- 对于 `const` 对象来说, 智能指针简单易用, 也比深度复制高效。

缺点:

- 不得不用指针(不管是智能的还是原生的)来表示和传递所有权。指针语义可要比值语义复杂得许多了, 特别是在 API 里: 您不光要操心所有权, 还要顾及别名, 生命周期, 可变性(mutability)以及其它大小问题。
- 其实值语义的开销经常被高估, 所以就所有权的性能来说, 可不能光只考虑可读性以及复

杂性。

- 如果 API 依赖所有权的传递,就会害得客户端不得不用单一的内存管理模型。
- 销毁资源并回收的相关代码不是很明朗。
- `std::unique_ptr` 的所有权传递原理是 C++11 的 move 语法,后者毕竟是刚刚推出的,容易迷惑程序员。
- 如果原本的所有权设计已经够完善了,那么若要引入所有权共享机制,可能不得不重构整个系统。
- 所有权共享机制的登记工作在运行时进行,开销可能相当不小。
- 某些极端情况下,所有权被共享的对象永远不会被销毁,比如引用死循环(cyclic references)。
- 智能指针并不能够完全代替原生指针。

决定:

如果必须使用动态分配,倾向于保持分配者的所有权。如果其他地方要使用这个对象,最好传递它的拷贝,或者传递一个不用改变所有权的指针或引用。倾向于使用 `std::unique_ptr` 来明确所有权传递,例如:

```
std::unique_ptr<Foo> FooFactory();  
void FooConsumer(std::unique_ptr<Foo> ptr);
```

避免使用共享所有权。如果对性能要求很高,并且操作的对象是不可变的(比如说 `std::shared_ptr<const Foo>`),这时可以用共享所有权来避免昂贵的拷贝操作。如果确实要使用共享所有权,倾向于使用 `std::shared_ptr`。

不要在新代码中使用 `scoped_ptr` `` , 除非你必须兼容老版本的C++。总是用 ```std::unique_ptr` 代替 `std::auto_ptr`。

4.2. cpplint

Tip

使用 `cpplint.py` 检查风格错误。

`cpplint.py` 是一个用来分析源文件,能检查出多种风格错误的工具. 它并不完美,甚至还会漏报和误报,但它仍然是一个非常有用的工具. 在行尾加 `// NOLINT`, 或在上一行加 `// NOLINTNEXTLINE`, 可以忽略报错。

某些项目会指导你如何使用他们的项目工具运行 `cpplint.py`. 如果你参与的项目没有提供, 你可以单独下载 [cpplint.py](http://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py) [http://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py].

译者(acgtyrant)笔记

1. 把智能指针当成对象来看待的话,就很好领会它与所指对象之间的关系了。
2. 原来 Rust 的 Ownership 思想是受到了 C++ 智能指针的很大启发啊。
3. `scoped_ptr` 和 `auto_ptr` 已过时。现在是 `shared_ptr` 和 `unique_ptr` 的天下了。

4. 按本文来说,似乎除了智能指针,还有其它所有权机制,值得留意。
5. Arch Linux 用户注意了,AUR 有对 cpplint 打包。

5. 其他 C++ 特性

5.1. 引用参数

Tip

所有按引用**传递**的参数必须加上 `const`.

定义:

在 C 语言中, 如果函数需要修改变量的**值**, 参数必须为指针, 如 `int foo(int *pval)`. 在 C++ 中, 函数还可以声明引用参数: `int foo(int &val)`.

优点:

定义引用参数防止出现 `(*pval)++` 这样丑陋的代码. 像拷贝构造函数这样的应用也是必需的. 而且更明确, 不接受 `NULL` 指针.

缺点:

容易引起误解, 因为引用在语法上是**值**变量却拥有指针的语义.

结论:

函数参数列表中, 所有引用参数都必须是 `const`:

```
void Foo(const string &in, string *out);
```

事实上这在 Google Code 是一个硬性约定: 输入参数是**值**参或 `const` 引用, 输出参数为指针. 输入参数可以是 `const` 指针, 但决不能是非 `const` 的引用参数, 除非用于交换, 比如 `swap()`.

有时候, 在输入形参中用 `const T*` 指针比 `const T&` 更明智. 比如:

- 您会传 `null` 指针。
- 函数要把指针或对地址的引用**赋值**给输入形参。

总之大多时候输入形参往往是 `const T&`. 若用 `const T*` 说明输入另有处理. 所以若您要用 `const T*`, 则应有理有据, 否则会害得读者误解。

5.2. 右值引用

Tip

只在定义移动构造函数与移动赋值操作时使用右值引用. 不要使用 `std::forward`.

定义:

右值引用是一种只能绑定到临时对象的引用的一种, 其语法与传统的引用语法相似. 例如, `void f(string&& s);` 声明了一个其参数是一个字符串的右值引用的函数.

优点:

用于定义移动构造函数 (使用类的右值引用进行构造的函数) 使得移动一个值而非拷贝之成为可能. 例如, 如果 `v1` 是一个 `vector<string>`, 则 `auto v2(std::move(v1))` 将很可能不再进行大量的数据复制而只是简单地进行指针操作, 在某些情况下这将带来大幅度的性能提升.

右值引用使得编写通用的函数封装来转发其参数到另外一个函数成为可能, 无论其参数是否是临时对象都能正常工作.

右值引用能实现可移动但不可拷贝的类型, 这一特性对那些在拷贝方面没有实际需求, 但有时又需要将它们作为函数参数传递或塞入容器的类型很有用.

要高效率地使用某些标准库类型, 例如 `std::unique_ptr`, `std::move` 是必需的.

缺点:

右值引用是一个相对比较新的特性 (由 C++11 引入), 它尚未被广泛理解. 类似引用崩溃, 移动构造函数的自动推导这样的规则都是很复杂的.

结论:

只在定义移动构造函数与移动赋值操作时使用右值引用, 不要使用 `std::forward` 功能函数. 你可能会使用 `std::move` 来表示将值从一个对象移动而不是复制到另一个对象.

5.3. 函数重载

Tip

若要用好函数重载, 最好能让读者一看调用点 (call site) 就胸有成竹, 不用花心思猜测调用的重载函数到底是哪一种. 该规则适用于构造函数.

定义:

你可以编写一个参数类型为 `const string&` 的函数, 然后用另一个参数类型为 `const char*` 的函数重载它:

```
class MyClass {
public:
    void Analyze(const string &text);
```

```
};  
void Analyze(const char *text, size_t textlen);
```

优点:

通过重载参数不同的同名函数, 令代码更加直观. 模板化代码需要重载, 同时为使用者带来便利.

缺点:

如果函数单单靠不同的参数类型而重载(acgtyrant 注:这意味着参数数量不变), 读者就得十分熟悉 C++ 五花八门的匹配规则, 以了解匹配过程具体到底如何. 另外, 当派生类只重载了某个函数的部分变体, 继承语义容易令人困惑.

结论:

如果您打算重载一个函数, 可以试试改在函数名里加上参数信息. 例如, 用 AppendString() 和 AppendInt() 等, 而不是一口气重载多个 Append().

5.4. 缺省参数

Tip

我们不允许使用缺省函数参数, 少数极端情况除外. 尽可能改用函数重载.

优点:

当您有依赖缺省参数的函数时, 您也许偶尔会修改修改这些缺省参数. 通过缺省参数, 不用再为个别情况而特意定义一大堆函数了. 与函数重载相比, 缺省参数语法更为清晰, 代码少, 也很好地区分了「必选参数」和「可选参数」.

缺点:

缺省参数会干扰函数指针, 害得后者的函数签名(function signature)往往对不上所实际要调用的函数签名. 即在一个现有函数添加缺省参数, 就会改变它的类型, 那么调用其地址的代码可能会出错, 不过函数重载就没这问题了. 此外, 缺省参数会造成臃肿的代码, 毕竟它们在每一个调用点(call site)都有重复(acgtyrant 注:我猜可能是因为调用函数的代码表面上省去了不少参数, 但编译器在编译时还是会在每一个调用代码里统统补上所有默认实参信息, 造成大量的重复). 函数重载正好相反, 毕竟它们所谓的「缺省参数」只会出现在函数定义里.

结论:

由于缺点并不是很严重, 有些人依旧偏爱缺省参数胜于函数重载. 所以除了以下情况, 我们要求必须显式提供所有参数(acgtyrant 注:即不能再通过缺省参数来省略参数了).

其一, 位于 .cc 文件里的静态函数或匿名空间函数, 毕竟都只能在局部文件里调用该函数了.

其二,可以在构造函数里用缺省参数,毕竟不可能取得它们的地址。

其三,可以用来模拟变长数组。

```
// 通过空 AlphaNum 以支持四个形参
string StrCat(const AlphaNum &a,
              const AlphaNum &b = gEmptyAlphaNum,
              const AlphaNum &c = gEmptyAlphaNum,
              const AlphaNum &d = gEmptyAlphaNum);
```

5.5. 变长数组和 `alloca()`

Tip

我们不允许使用变长数组和 `alloca()`。

优点:

变长数组具有浑然天成的语法. 变长数组和 `alloca()` 也都很高效.

缺点:

变长数组和 `alloca()` 不是标准 C++ 的组成部分. 更重要的是, 它们根据数据大小动态分配堆栈内存, 会引起难以发现的内存越界 bugs: “在我的机器上运行的好好的, 发布后却莫名其妙的挂掉了”。

结论:

改用更安全的分配器(allocator), 就像 `std::vector` 或 `std::unique_ptr<T[]>`。

5.6. 友元

Tip

我们允许合理的使用友元类及友元函数。

通常友元应该定义在同一文件内, 避免代码读者跑到其它文件查找使用该私有成员的类. 经常用到友元的一个地方是将 `FooBuilder` 声明为 `Foo` 的友元, 以便 `FooBuilder` 正确构造 `Foo` 的内部状态, 而无需将该状态暴露出来. 某些情况下, 将一个单元测试类声明成待测类的友元会很方便.

友元扩大了(但没有打破)类的封装边界. 某些情况下, 相对于将类成员声明为 `public`, 使用友元是更好的选择, 尤其是如果你只允许另一个类访问该类的私有成员时. 当然, 大多数类都只应该通过其提供的公有成员进行互操作.

5.7. 异常

Tip

我们不使用 C++ 异常。

优点:

- 异常允许应用高层决定如何处理在底层嵌套函数中「不可能发生」的失败(failures),不用管那些含糊且容易出错的错误代码(acgtyrant 注:error code, 我猜是C语言函数返回的非零 int 值)。
- 很多现代语言都用异常。引入异常使得 C++ 与 Python, Java 以及其它类 C++ 的语言更一脉相承。
- 有些第三方 C++ 库依赖异常,禁用异常就不好用了。
- 异常是处理构造函数失败的唯一途径。虽然可以用工厂函数(acgtyrant 注:factory function, 出自 C++ 的一种设计模式,即「简单工厂模式」)或 Init() 方法代替异常, but these require heap allocation or a new “invalid” state, respectively.
- 在测试框架里很好用。

缺点:

- 在现有函数中添加 throw 语句时,您必须检查所有调用点。要么让所有调用点统统具备最低限度的异常安全保证,要么眼睁睁地看异常一路欢快地往上跑,最终中断掉整个程序。举例,f() 调用 g(), g() 又调用 h(), 且 h 抛出的异常被 f 捕获。当心 g, 否则会没妥善清理好。
- 还有更常见的,异常会彻底扰乱程序的执行流程并难以判断,函数也许会在您意料不到的地方返回。您或许会加一大堆何时何处处理异常的规定来降低风险,然而开发者的记忆负担更重了。
- 异常安全需要RAII和不同的编码实践. 要轻松编写出正确的异常安全代码需要大量的支持机制. 更进一步地说, 为了避免读者理解整个调用表, 异常安全必须隔绝从持续状态写到“提交”状态的逻辑. 这一点有利有弊 (因为你也许不得不为了隔离提交而混淆代码). 如果允许使用异常, 我们就不得不时刻关注这样的弊端, 即使有时它们并不值得。
- 后用异常会增加二进制文件数据,延长编译时间(或许影响小),还可能加大地址空间的压力。
- 滥用异常会变相鼓励开发者去捕捉不合时宜,或本来就已经没法恢复的「伪异常」。比如, 用户的输入不符合格式要求时,也用不着抛异常。如此之类的伪异常列都列不完。

结论:

从表面上看来,使用异常利大于弊,尤其是在新项目中. 但是对于现有代码, 引入异常会牵连到所有相关代码. 如果新项目允许异常向外扩散, 在跟以前未使用异常的代码整合时也将是个麻烦. 因为 Google 现有的大多数 C++ 代码都没有异常处理, 引入带有异常处理的新代码相当困难.

鉴于 Google 现有代码不接受异常, 在现有代码中使用异常比在新项目中使用的代价多少要大一些. 迁移过程比较慢, 也容易出错. 我们不相信异常的使用有效替代方案, 如错误代码, 断言等会造成严重负担.

我们并不是基于哲学或道德层面反对使用异常, 而是在实践的基础上. 我们希望在 Google 使

用我们自己的开源项目, 但项目中使用异常会为此带来不便, 因此我们也建议不要在 Google 的开源项目中使用异常. 如果我们需要把这些项目推倒重来显然不太现实.

对于 Windows 代码来说, 有个 [特例](#).

(YuleFox 注: 对于异常处理, 显然不是短短几句话能够说清楚的, 以构造函数为例, 很多 C++ 书籍上都提到当构造失败时只有异常可以处理, Google 禁止使用异常这一点, 仅仅是为了自身的方便, 说大了, 无非是基于软件管理成本上, 实际使用中还是自己决定)

5.8. 运行时类型识别

TODO

Tip

我们禁止使用 RTTI.

定义:

RTTI 允许程序员在运行时识别 C++ 类对象的类型. 它通过使用 typeid 或者 dynamic_cast 完成.

优点:

RTTI 的标准替代 (下面将描述) 需要对有问题的类层级进行修改或重构. 有时这样的修改并不是我们所想要的, 甚至是不可取的, 尤其是在一个已经广泛使用的或者成熟的代码中.

RTTI 在某些单元测试中非常有用. 比如进行工厂类测试时, 用来验证一个新建对象是否为期望的动态类型. RTTI 对于管理对象和派生对象的关系也很有用.

在考虑多个抽象对象时 RTTI 也很好用. 例如:

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
    Derived* that = dynamic_cast<Derived*>(other);
    if (that == NULL)
        return false;
    ...
}
```

缺点:

在运行时判断类型通常意味着设计问题. 如果你需要在运行期间确定一个对象的类型, 这通常说明你需要考虑重新设计你的类.

随意地使用 RTTI 会使你的代码难以维护. 它使得基于类型的判断树或者 switch 语句散布在代码各处. 如果以后要进行修改, 你就必须检查它们.

结论:

RTTI 有合理的用途但是容易被滥用, 因此在使用时请务必注意. 在单元测试中可以使用 RTTI, 但是在其他代码中请尽量避免. 尤其是在新代码中, 使用 RTTI 前务必三思. 如果你的代码需要根据不同的对象类型执行不同的行为的话, 请考虑用以下的两种替代方案之一查询类型:

虚函数可以根据子类类型的不同而执行不同代码. 这是把工作交给了对象本身去处理.

如果这一工作需要在对象之外完成, 可以考虑使用双重分发的方案, 例如使用访问者设计模式. 这就能够在对象之外进行类型判断.

如果程序能够保证给定的基类实例实际上都是某个派生类的实例, 那么就可以自由使用 `dynamic_cast`. 在这种情况下, 使用 `dynamic_cast` 也是一种替代方案.

基于类型的判断树是一个很强的暗示, 它说明你的代码已经偏离正轨了. 不要像下面这样:

```
if (typeid(*data) == typeid(D1)) {  
    ...  
} else if (typeid(*data) == typeid(D2)) {  
    ...  
} else if (typeid(*data) == typeid(D3)) {  
    ...  
}
```

一旦在类层级中加入新的子类, 像这样的代码往往会崩溃. 而且, 一旦某个子类的属性改变了, 你很难找到并修改所有受影响的代码块.

不要去手工实现一个类似 RTTI 的方案. 反对 RTTI 的理由同样适用于这些方案, 比如带类型标签的类继承体系. 而且, 这些方案会掩盖你的真实意图.

5.9. 类型转换

Tip

使用 C++ 的类型转换, 如 `static_cast<>()`. 不要使用 `int y = (int)x` 或 `int y = int(x)` 等转换方式;

定义:

C++ 采用了有别于 C 的类型转换机制, 对转换操作进行归类.

优点:

C 语言的类型转换问题在于模棱两可的操作; 有时是在做强制转换 (如 `(int)3.5`), 有时是在做类型转换 (如 `(int)"hello"`). 另外, C++ 的类型转换在查找时更醒目.

缺点:

恶心的语法.

结论:

不要使用 C 风格类型转换. 而应该使用 C++ 风格.

- 用 `static_cast` 替代 C 风格的值转换, 或某个类指针需要明确的向上转换为父类指针时.
- 用 `const_cast` 去掉 `const` 限定符.
- 用 `reinterpret_cast` 指针类型和整型或其它指针之间进行不安全的相互转换. 仅在你对所作一切了然于心时使用.

至于 `dynamic_cast` 参见 [5.8. 运行时类型识别](#).

5.10. 流

Tip

只在记录日志时使用流.

定义:

流用来替代 `printf()` 和 `scanf()`.

优点:

有了流, 在打印时不需要关心对象的类型. 不用担心格式化字符串与参数列表不匹配 (虽然在 gcc 中使用 `printf` 也不存在这个问题). 流的构造和析构函数会自动打开和关闭对应的文件.

缺点:

流使得 `pread()` 等功能函数很难执行. 如果不使用 `printf` 风格的格式化字符串, 某些格式化操作 (尤其是常用的格式字符串 `%. *s`) 用流处理性能是很低的. 流不支持字符串操作符重新排序 (`%1s`), 而这一点对于软件国际化很有用.

结论:

不要使用流, 除非是日志接口需要. 使用 `printf` 之类的代替.

使用流还有很多利弊, 但代码一致性胜过一切. 不要在代码中使用流.

拓展讨论:

对这一条规则存在一些争论, 这儿给出点深层次原因. 回想一下唯一性原则 (Only One Way): 我们希望在任何时候都只使用一种确定的 I/O 类型, 使代码在所有 I/O 处都保持一致. 因此, 我们不希望用户来决定是使用流还是 `printf + read/write`. 相反, 我们应该决定到底用哪一种方式. 把日志作为特例是因为日志是一个非常独特的应用, 还有一些是历史原因.

流的支持者们主张流是不二之选, 但观点并不是那么清晰有力. 他们指出的流的每个优势也都是其劣势. 流最大的优势是在输出时不需要关心打印对象的类型. 这是一个亮点. 同时, 也是一个不足: 你很容易用错类型, 而编译器不会报警. 使用流时容易造成的这类错误:

```
cout << this;    // 输出地址
cout << *this;   // 输出值
```

由于 << 被重载, 编译器不会报错. 就因为这一点我们反对使用操作符重载.

有人说 printf 的格式化丑陋不堪, 易读性差, 但流也好不到哪儿去. 看看下面两段代码吧, 实现相同的功能, 哪个更清晰?

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
      << ":" << foo->bar()->hostname.second << ": " << strerror(errno);

fprintf(stderr, "Error connecting to '%s:%u: %s",
         foo->bar()->hostname.first, foo->bar()->hostname.second,
         strerror(errno));
```

你可能会说, “把流封装一下就会比较好了”, 这儿可以, 其他地方呢? 而且不要忘了, 我们的目标是使语言更紧凑, 而不是添加一些别人需要学习的新装备.

每一种方式都是各有利弊, “没有最好, 只有更适合”. 简单性原则告诫我们必须从中选择其一, 最后大多数决定采用 printf + read/write.

5.11. 前置自增和自减

Tip

对于迭代器和其他模板对象使用前缀形式 (++i) 的自增, 自减运算符.

定义:

对于变量在自增 (++i 或 i++) 或自减 (--i 或 i--) 后表达式的值又没有没用到的情况下, 需要确定到底是使用前置还是后置的自增 (自减).

优点:

不考虑返回值的话, 前置自增 (++i) 通常要比后置自增 (i++) 效率更高. 因为后置自增 (或自减) 需要对表达式的值 i 进行一次拷贝. 如果 i 是迭代器或其他非数值类型, 拷贝的代价是比较大的. 既然两种自增方式实现的功能一样, 为什么不总是使用前置自增呢?

缺点:

在 C 开发中, 当表达式的值未被使用时, 传统的做法是使用后置自增, 特别是在 for 循环中. 有些人觉得后置自增更加易懂, 因为这很像自然语言, 主语 (i) 在谓语动词 (++) 前.

结论:

对简单数值 (非对象), 两种都无所谓. 对迭代器和模板类型, 使用前置自增 (自减).

5.12. const 用法

Tip

我们强烈建议你在任何可能的情况下都要使用 `const`. 此外有时改用 C++11 推出的 `constexpr` 更好。

定义:

在声明的变量或参数前加上关键字 `const` 用于指明变量值不可被篡改 (如 `const int foo`). 为类中的函数加上 `const` 限定符表明该函数不会修改类成员变量的状态 (如 `class Foo { int Bar(char c) const; };`).

优点:

大家更容易理解如何使用变量. 编译器可以更好地进行类型检测, 相应地, 也能生成更好的代码. 人们对编写正确的代码更加自信, 因为他们知道所调用的函数被限定了能或不能修改变量值. 即使是在无锁的多线程编程中, 人们也知道什么样的函数是安全的.

缺点:

`const` 是入侵性的: 如果你向一个函数传入 `const` 变量, 函数原型声明中也必须对应 `const` 参数 (否则变量需要 `const_cast` 类型转换), 在调用库函数时显得尤其麻烦.

结论:

`const` 变量, 数据成员, 函数和参数为编译时类型检测增加了一层保障; 便于尽早发现错误. 因此, 我们强烈建议在任何可能的情况下使用 `const`:

- 如果函数不会修改传你入的引用或指针类型参数, 该参数应声明为 `const`.
- 尽可能将函数声明为 `const`. 访问函数应该总是 `const`. 其他不会修改任何数据成员, 未调用非 `const` 函数, 不会返回数据成员非 `const` 指针或引用的函数也应该声明成 `const`.
- 如果数据成员在对象构造之后不再发生变化, 可将其定义为 `const`.

然而, 也不要发了疯似的使用 `const`. 像 `const int * const * const x`; 就有些过了, 虽然它非常精确的描述了常量 `x`. 关注真正有帮助意义的信息: 前面的例子写成 `const int** x` 就够了.

关键字 `mutable` 可以使用, 但是在多线程中是不安全的, 使用时首先要考虑线程安全.

`const` 的位置:

有人喜欢 `int const *foo` 形式, 不喜欢 `const int* foo`, 他们认为前者更一致因此可读性也更好: 遵循了 `const` 总位于其描述的对象之后的原则. 但是一致性原则不适用于此, “不要过度使用” 的声明可以取消大部分你原本想保持一致性. 将 `const` 放在前面才更易读, 因为在自

然语言中形容词 (const) 是在名词 (int) 之前.

这是说, 我们提倡但不强制 const 在前. 但要保持代码的一致性! (Yang.Y 注: 也就是不要在一些地方把 const 写在类型前面, 在其他地方又写在后面, 确定一种写法, 然后保持一致.)

5.13. constexpr 用法

Tip

在 C++11 里, 用 constexpr 来定义真正的常量, 或实现常量初始化。

定义:

变量可以被声明成 constexpr 以表示它是真正意义上的常量, 即在编译时和运行时都不变。函数或构造函数也可以被声明成 constexpr, 以用来定义 constexpr 变量。

优点:

如今 constexpr 就可以定义浮点式的真·常量, 不用再依赖字面值了; 也可以定义用户自定义类型上的常量; 甚至也可以定义函数调用所返回的常量。

缺点:

若过早把变量优化成 constexpr 变量, 将来又要把它改为常规变量时, 挺麻烦的; Current restrictions on what is allowed in constexpr functions and constructors may invite obscure workarounds in these definitions.

结论:

靠 constexpr 特性, 方才实现了 C++ 在接口上打造真正常量机制的可能。好好用 constexpr 来定义真·常量以及支持常量的函数。Avoid complexifying function definitions to enable their use with constexpr. 千万别痴心妄想地想靠 constexpr 来强制代码「内联」。

5.14. 整型

Tip

C++ 内建整型中, 仅使用 int. 如果程序中需要不同大小的变量, 可以使用 <stdint.h> 中长度精确的整型, 如 int16_t. 如果您的变量可能不小于 2^{31} (2GiB), 就用 64 位变量比如 int64_t. 此外要留意, 哪怕您的值并不会超出 int 所能够表示的范围, 在计算过程中也可能会溢出。所以拿不准时, 干脆用更大的类型。

定义:

C++ 没有指定整型的大小. 通常人们假定 short 是 16 位, int 是 32 位, long 是 32 位, long

long 是 64 位.

优点:

保持声明统一.

缺点:

C++ 中整型大小因编译器和体系结构的不同而不同.

结论:

`<stdint.h>` 定义了 `int16_t`, `uint32_t`, `int64_t` 等整型, 在需要确保整型大小时可以使用它们代替 `short`, `unsigned long long` 等. 在 C 整型中, 只使用 `int`. 在合适的情况下, 推荐使用标准类型如 `size_t` 和 `ptrdiff_t`.

如果已知整数不会太大, 我们常常会使用 `int`, 如循环计数. 在类似的情况下使用原生类型 `int`. 你可以认为 `int` 至少为 32 位, 但不要认为它会多于 32 位. 如果需要 64 位整型, 用 `int64_t` 或 `uint64_t`.

对于大整数, 使用 `int64_t`.

不要使用 `uint32_t` 等无符号整型, 除非你是在表示一个位组而不是一个数值, 或是你需要定义二进制补码溢出. 尤其是不要为了指出数值永不会为负, 而使用无符号类型. 相反, 你应该使用断言来保护数据.

如果您的代码涉及容器返回的大小(size), 确保其类型足以应付容器各种可能的用法. 拿不准时, 类型越大越好.

小心整型类型转换和整型提升(acgtyrant 注: integer promotions, 比如 `int` 与 `unsigned int` 运算时, 前者被提升为 `unsigned int` 而有可能溢出), 总有意想不到的后果.

关于无符号整数:

有些人, 包括一些教科书作者, 推荐使用无符号类型表示非负数. 这种做法试图达到自我文档化. 但是, 在 C 语言中, 这一优点被由其导致的 bug 所淹没. 看看下面的例子:

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

上述循环永远不会退出! 有时 gcc 会发现该 bug 并报警, 但大部分情况下都不会. 类似的 bug 还会出现在比较有符合变量和无符号变量时. 主要是 C 的类型提升机制会致使无符号类型的行为出乎你的意料.

因此, 使用断言来指出变量为非负数, 而不是使用无符号型!

5.15. 64 位下的可移植性

Tip

代码应该对 64 位和 32 位系统友好. 处理打印, 比较, 结构体对齐时应切记:

- 对于某些类型, printf() 的指示符在 32 位和 64 位系统上可移植性不是很好. C99 标准定义了一些可移植的格式化指示符. 不幸的是, MSVC 7.1 并非全部支持, 而且标准中也有所遗漏, 所以有时我们不得不自己定义一个丑陋的版本 (头文件 inttypes.h 仿标准风格):

```
// printf macros for size_t, in the style of inttypes.h
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
#define __PRIS_PREFIX
#endif

// Use these macros after a % in a printf format string
// to get correct 32/64 bit behavior, like this:
// size_t size = records.size();
// printf("%"PRIuS"\n", size);
#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

| 类型 | 不要使用 | 使用 | 备注 |
|------------------|-----------------|----------------------|------------|
| void * (或其他指针类型) | %lx | %p | |
| int64_t | %qd, %lld | %"PRId64" | |
| uint64_t | %qu, %llu, %llx | %"PRIu64", %"PRIx64" | |
| size_t | %u | %"PRIuS", %"PRIxS" | C99 规定 %zu |
| ptrdiff_t | %d | %"PRIdS" | C99 规定 %zd |

注意 PRI* 宏会被编译器扩展为独立字符串. 因此如果使用非常量的格式化字符串, 需要将宏的值而不是宏名插入格式中. 使用 PRI* 宏同样可以在 % 后包含长度指示符. 例如, printf("x = %30"PRIuS"\n", x) 在 32 位 Linux 上将被展开为 printf("x = %30" "u" "\n", x), 编译器当成 printf("x = %30u\n", x) 处理 (Yang.Y 注: 这在 MSVC 6.0 上行不通, VC 6 编译器不会自动把引号间隔的多个字符串连接一个长字符串).

- 记住 sizeof(void *) != sizeof(int). 如果需要一个指针大小的整数要用 intptr_t.
- 你要非常小心的对待结构体对齐, 尤其是要持久化到磁盘上的结构体 (Yang.Y 注: 持久化 - 将数据按字节流顺序保存在磁盘文件或数据库中). 在 64 位系统中, 任何含有 int64_t/uint64_t 成员的类型/结构体, 缺省都以 8 字节在结尾对齐. 如果 32 位和 64 位代码要共用持久化的结构体, 需要确保两种体系结构下的结构体对齐一致. 大多数编译器都允许调整结构体对齐. gcc 中可使用 __attribute__((packed)). MSVC 则提供了 #pragma pack() 和 __declspec(align()) (YuleFox 注, 解决方案的项目属性里也可以直接设置).
- 创建 64 位常量时使用 LL 或 ULL 作为后缀, 如:

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

- 如果你确实需要 32 位和 64 位系统具有不同代码, 可以使用 `#ifdef _LP64` 指令来切分 32/64 位代码. (尽量不要这么做, 如果非用不可, 尽量使修改局部化)

5.16. 预处理宏

Tip

使用宏时要非常谨慎, 尽量以内联函数, 枚举和常量代替之.

宏意味着你和编译器看到的代码是不同的. 这可能会导致异常行为, 尤其因为宏具有全局作用域.

值得庆幸的是, C++ 中, 宏不像在 C 中那么必不可少. 以往用宏展开性能关键的代码, 现在可以用内联函数替代. 用宏表示常量可被 `const` 变量代替. 用宏“缩写”长变量名可被引用代替. 用宏进行条件编译... 这个, 千万别这么做, 会令测试更加痛苦 (`#define` 防止头文件重包含当然是个特例).

宏可以做一些其他技术无法实现的事情, 在一些代码库 (尤其是底层库中) 可以看到宏的某些特性 (如用 `#` 字符串化, 用 `##` 连接等等). 但在使用前, 仔细考虑一下能不能不使用宏达到同样的目的.

下面给出的用法模式可以避免使用宏带来的问题; 如果你要宏, 尽可能遵守:

- 不要在 .h 文件中定义宏.
- 在马上要使用时才进行 `#define`, 使用后要立即 `#undef`.
- 不要只是对已经存在的宏使用 `#undef`, 选择一个不会冲突的名称;
- 不要试图使用展开后会导致 C++ 构造不稳定的宏, 不然也至少要附上文档说明其行为.
- 不要用 `##` 处理函数, 类和变量的名字。

5.17. 0, nullptr 和 NULL

Tip

整数用 `0`, 实数用 `0.0`, 指针用 `nullptr` 或 `NULL`, 字符(串)用 `'\0'`.

整数用 `0`, 实数用 `0.0`, 这一点是毫无争议的.

对于指针 (地址值), 到底是用 `0`, `NULL` 还是 `nullptr`. C++11 项目用 `nullptr`; C++03 项目则用 `NULL`, 毕竟它看起来像指针. 实际上, 一些 C++ 编译器对 `NULL` 的定义比较特殊, 可以输出有用的警告, 特别是 `sizeof(NULL)` 就和 `sizeof(0)` 不一样.

字符(串)用 `'\0'`, 不仅类型正确而且可读性好.

5.18. sizeof

Tip

尽可能用 `sizeof(varname)` 代替 `sizeof(type)`。

使用 `sizeof(varname)` 是因为当代码中变量类型改变时会自动更新. 您或许会用 `sizeof(type)` 处理不涉及任何变量的代码, 比如处理来自外部或内部的数据格式, 这时用变量就不合适了。

```
Struct data;  
Struct data; memset(&data, 0, sizeof(data));
```

Warning

```
memset(&data, 0, sizeof(Struct));
```

```
if (raw_size < sizeof(int)) {  
    LOG(ERROR) << "compressed record not big enough for count: " << raw_size;  
    return false;  
}
```

5.19. auto

Tip

用 `auto` 绕过烦琐的类型名, 只要可读性好就继续用, 别用在局部变量之外的地方。

定义:

C++11 中, 若变量被声明成 `auto`, 那它的类型就会被自动匹配成初始化表达式的类型。您可以用 `auto` 来复制初始化或绑定引用。

```
vector<string> v;  
...  
auto s1 = v[0]; // 创建一份 v[0] 的拷贝。  
const auto& s2 = v[0]; // s2 是 v[0] 的一个引用。
```

优点:

C++ 类型名有时又长又臭, 特别是涉及模板或命名空间的时候。就像:

```
sparse_hash_map<string, int>::iterator iter = m.find(val);
```

返回类型好难读, 代码目的也不够一目了然。重构其:

```
auto iter = m.find(val);
```

好多了。

没有 `auto` 的话, 我们不得不在同一个表达式里写同一个类型名两次, 无谓的重复, 就像:

```
diagnostics::ErrorStatus* status = new diagnostics::ErrorStatus("xyz");
```

有了 `auto`, 可以更方便地用中间变量, 显式编写它们的类型轻松点。

缺点:

类型够明显时, 特别是初始化变量时, 代码才会够一目了然。但以下就不一样了:

```
auto i = x.Lookup(key);
```

看不出其类型是啥, `x` 的类型声明恐怕远在几百行之外了。

程序员必须会区分 `auto` 和 `const auto&` 的不同之处, 否则会复制错东西。

`auto` 和 C++11 列表初始化的合体令人摸不着头脑:

```
auto x(3); // 圆括号。
auto y{3}; // 大括号。
```

它们不是同一回事——`x` 是 `int`, `y` 则是 `std::initializer_list<int>`. 其它一般不可见的代理类型 (acgtyrant 注: normally-invisible proxy types, 它涉及到 C++ 鲜为人知的坑: [Why is vector<bool> not a STL container?](http://stackoverflow.com/a/17794965/1546088) [http://stackoverflow.com/a/17794965/1546088]) 也有大同小异的陷阱。

如果在接口里用 `auto`, 比如声明头文件里的一个常量, 那么只要仅仅因为程序员一时修改其值而导致类型变化的话——API 要翻天覆地的了。

结论:

`auto` 只能用在局部变量里用。别用在文件作用域变量, 命名空间作用域变量和类数据成员里。永远别列表初始化 `auto` 变量。

`auto` 还可以和 C++11 特性「尾置返回类型 (trailing return type)」一起用, 不过后者只能用在 `lambda` 表达式里。

5.20. 列表初始化

Tip

你可以用列表初始化。

早在 C++03 里, 聚合类型 (aggregate types) 就已经可以被列表初始化了, 比如数组和不自带构造函数的结构体:

```
struct Point { int x; int y; };
Point p = {1, 2};
```

C++11 中, 该特性得到进一步的推广, 任何对象类型都可以被列表初始化。示范如下:

```
// Vector 接收了一个初始化列表。
vector<string> v{"foo", "bar"};

// 不考虑细节上的微妙差别, 大致上相同。
```

```
// 您可以任选其一。
vector<string> v = {"foo", "bar"};

// 可以配合 new 一起用。
auto p = new vector<string>{"foo", "bar"};

// map 接收了一些 pair, 列表初始化大显神威。
map<int, string> m = {{1, "one"}, {2, "2"}};

// 初始化列表也可以用在返回类型上的隐式转换。
vector<int> test_function() { return {1, 2, 3}; }

// 初始化列表可迭代。
for (int i : {-1, -2, -3}) {}

// 在函数调用里用列表初始化。
void TestFunction2(vector<int> v) {}
TestFunction2({1, 2, 3});
```

用户自定义类型也可以定义接收 `std::initializer_list<T>` 的构造函数和赋值运算符,以自动列表初始化:

```
class MyType {
public:
    // std::initializer_list 专门接收 init 列表。
    // 得以值传递。
    MyType(std::initializer_list<int> init_list) {
        for (int i : init_list) append(i);
    }
    MyType& operator=(std::initializer_list<int> init_list) {
        clear();
        for (int i : init_list) append(i);
    }
};
MyType m{2, 3, 5, 7};
```

最后,列表初始化也适用于常规数据类型的构造,哪怕没有接收 `std::initializer_list<T>` 的构造函数。

```
double d{1.23};
// MyOtherType 没有 std::initializer_list 构造函数,
// 直接上接收常规类型的构造函数。
class MyOtherType {
public:
    explicit MyOtherType(string);
    MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// 不过如果构造函数是显式的 (explicit), 您就不能用 `= {}` 了。
MyOtherType m{"b"};
```

千万别直接列表初始化 auto 变量,看下一句,估计没人看得懂:

Warning

```
auto d = {1.23};           // d 即是 std::initializer_list<double>
```



```
auto d = double{1.23}; // 善哉 -- d 即为 double, 并非 std::initializer_list.
```

至于格式化,参见 `braced-initializer-list-format`.

5.21. Lambda 表达式

Tip

适当使用 lambda 表达式。别用默认 lambda 捕获,所有捕获都要显式写出来。

定义:

Lambda 表达式是创建匿名函数对象的一种简易途径,常用于把函数当参数传,例如:

```
std::sort(v.begin(), v.end(), [](int x, int y) {  
    return Weight(x) < Weight(y);  
});
```

C++11 首次提出 Lambdas, 还提供了一系列处理函数对象的工具,比如多态包装器 (polymorphic wrapper) `std::function`.

优点:

- 传函数对象给 STL 算法,Lambdas 最简易,可读性也好。
- Lambdas, `std::functions` 和 `std::bind` 可以搭配成通用回调机制 (general purpose callback mechanism); 写接收有界函数为参数的函数也很容易了。

缺点:

- Lambdas 的变量捕获略旁门左道,可能会造成悬空指针。
- Lambdas 可能会失控;层层嵌套的匿名函数难以阅读。

结论:

- 按 format 小用 lambda 表达式怡情。
- 禁用默认捕获,捕获都要显式写出来。打比方,比起 `[=](int x) {return x + n;}`, 您该写成 `[n](int x) {return x + n;}` 才对,这样读者也好一眼看出 `n` 是被捕获的值。
- 匿名函数始终要简短,如果函数体超过了五行,那么还不如起名 (acgtyrant 注:即把 lambda 表达式赋值给对象),或改用函数。
- 如果可读性更好,就显式写出 `lambda` 的尾置返回类型,就像 `auto`。

5.22. 模板元编程

TODO

5.23. Boost 库

Tip

只使用 Boost 中被认可的库.

定义:

[Boost 库集](http://www.boost.org/) [http://www.boost.org/] 是一个广受欢迎, 经过同行鉴定, 免费开源的 C++ 库集.

优点:

Boost 代码质量普遍较高, 可移植性好, 填补了 C++ 标准库很多空白, 如型别的特性, 更完善的绑定器, 更好的智能指针.

缺点:

某些 Boost 库提倡的编程实践可读性差, 比如元编程和其他高级模板技术, 以及过度“函数化”的编程风格.

结论:

为了向阅读和维护代码的人员提供更好的可读性, 我们只允许使用 Boost 一部分经认可的特性子集. 目前允许使用以下库:

- [Call Traits](http://www.boost.org/doc/libs/1_58_0/libs/utility/call_traits.htm) [http://www.boost.org/doc/libs/1_58_0/libs/utility/call_traits.htm] : boost/call_traits.hpp
- [Compressed Pair](http://www.boost.org/libs/utility/compressed_pair.htm) [http://www.boost.org/libs/utility/compressed_pair.htm] : boost/compressed_pair.hpp
- [<The Boost Graph Library \(BGL\)](http://www.boost.org/doc/libs/1_58_0/libs/graph/doc/index.html) [http://www.boost.org/doc/libs/1_58_0/libs/graph/doc/index.html] : boost/graph, except serialization (adj_list_serialize.hpp) and parallel/distributed algorithms and data structures(boost/graph/parallel/* and boost/graph/distributed/*)
- [Property Map](http://www.boost.org/libs/property_map/) [http://www.boost.org/libs/property_map/] : boost/property_map.hpp
- The part of [Iterator](http://www.boost.org/libs/iterator/) [http://www.boost.org/libs/iterator/] that deals with defining iterators: boost/iterator/iterator_adaptor.hpp, boost/iterator/iterator_facade.hpp, and boost/function_output_iterator.hpp
- The part of [Polygon](http://www.boost.org/libs/polygon/) [http://www.boost.org/libs/polygon/] that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: boost/polygon/voronoi_builder.hpp, boost/polygon/voronoi_diagram.hpp, and boost/polygon/voronoi_geometry_type.hpp
- [Bimap](http://www.boost.org/libs/bimap/) [http://www.boost.org/libs/bimap/] : boost/bimap
- [Statistical Distributions and Functions](http://www.boost.org/libs/math/doc/html/dist.html) [http://www.boost.org/libs/math/doc/html/dist.html] : boost/math/distributions
- [Multi-index](http://www.boost.org/libs/multi_index/) [http://www.boost.org/libs/multi_index/] : boost/multi_index
- [Heap](http://www.boost.org/libs/heap/) [http://www.boost.org/libs/heap/] : boost/heap
- The flat containers from [Container](http://www.boost.org/libs/container/) [http://www.boost.org/libs/container/]:

boost/container/flat_map, and boost/container/flat_set

我们正在积极考虑增加其它 Boost 特性, 所以列表中的规则将不断变化.

以下库可以用, 但由于如今已经被 C++ 11 标准库取代, 不再鼓励:

- [Pointer Container](http://www.boost.org/libs/ptr_container/) [http://www.boost.org/libs/ptr_container/] : boost/ptr_container, 改用 [std::unique_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr) [http://en.cppreference.com/w/cpp/memory/unique_ptr]
- [Array](http://www.boost.org/libs/array/) [http://www.boost.org/libs/array/] : boost/array.hpp, 改用 [std::array](http://en.cppreference.com/w/cpp/container/array) [http://en.cppreference.com/w/cpp/container/array]

5.24. C++11

Tip

适当用 C++11 (前身是 C++0x) 的库和语言扩展, 在贵项目用 C++11 特性前三思可移植性。

定义:

C++11 有众多语言和库上的`变革` <<https://en.wikipedia.org/wiki/C%2B%2B11>>`_`。

优点:

在二〇一四年八月之前, C++11 一度是官方标准, 被大多 C++ 编译器支持。它标准化很多我们早先就在用的 C++ 扩展, 简化了不少操作, 大大改善了性能和安全。

缺点:

C++11 相对于前身, 复杂极了: 1300 页 vs 800 页! 很多开发者也不怎么熟悉它。于是从长远来看, 前者特性对代码可读性以及维护代价难以预估。我们说不准什么时候采纳其特性, 特别是在被迫依赖老实工具的项目上。

和 [5.23. Boost 库](#) 一样, 有些 C++11 扩展提倡实则对可读性有害的编程实践——就像去除冗余检查 (比如类型名) 以帮助读者, 或是鼓励模板元编程等等。有些扩展在功能上与原有机制冲突, 容易招致困惑以及迁移代价。

缺点:

C++11 特性除了个别情况下, 可以用一用。除了本指南会有不少章节会加以讨若干 C++11 特性之外, 以下特性最好不要用:

- 尾置返回类型, 比如用 `auto foo() -> int` 代替 `int foo()`。为了兼容于现有代码的声明风格。
- 编译时合数 `<ratio>`, 因为它涉及一个重模板的接口风格。
- `<cfenv>` 和 `<fenv.h>` 头文件, 因为编译器尚不支持。
- 默认 lambda 捕获。

译者(acgtyrant)笔记

1. 实际上, [缺省参数会改变函数签名的前提是改变了它接收的参数数量](http://www.zhihu.com/question/24439516/answer/27858964)
[http://www.zhihu.com/question/24439516/answer/27858964], 比如把 `void a()` 改成 `void a(int b = 0)`, 开发者改变其代码的初衷也许是, 在不改变「代码兼容性」的同时, 又提供了可选 `int` 参数的余地, 然而这终究会破坏函数指针上的兼容性, 毕竟函数签名确实变了。
2. 此外把自带缺省参数的函数地址赋值给指针时, 会丢失缺省参数信息。
3. 我还发现 [滥用缺省参数会害得读者光只看调用代码的话, 会误以为其函数接受的参数数量比实际上还要少](http://www.zhihu.com/question/24439516/answer/27896004)。[http://www.zhihu.com/question/24439516/answer/27896004]
4. `friend` 实际上只对函数／类赋予了对其所所在类的访问权限, 并不是有效的声明语句。所以除了在头文件类内部写 `friend` 函数／类, 还要在类作用域之外正式地声明一遍, 最后在对应的 `.cc` 文件加以定义。
5. 本风格指南都强调了「友元应该定义在同一文件内, 避免代码读者跑到其它文件查找使用该私有成员的类」。那么可以把其声明放在类声明所在的头文件, 定义也放在类定义所在的文件。
6. 由于友元函数／类并不是类的一部分, 自然也不会是类可调用的公有接口, 于是我主张全集中放在类的尾部, 即 `.ref: `private` 的数据成员 <declaration-order>`` 之后。
7. [对使用 C++ 异常处理应具有怎样的态度?](http://www.zhihu.com/question/22889420) [http://www.zhihu.com/question/22889420] 非常值得一读。
8. 注意初始化 `const` 对象时, 必须在初始化的同时值初始化。
9. 用断言代替无符号整型类型, 深有启发。
10. `auto` 在涉及迭代器的循环语句里挺常用。
11. [Should the trailing return type syntax style become the default for new C++11 programs?](http://stackoverflow.com/questions/11215227/should-the-trailing-return-type-syntax-style-become-the-default-for-new-c11-programs?)
[http://stackoverflow.com/questions/11215227/should-the-trailing-return-type-syntax-style-become-the-default-for-new-c11-pr] 讨论了 `auto` 与尾置返回类型一起用的全新编码风格, 值得一看。

6. 命名约定

最重要的一致性规则是命名管理. 命名风格快速获知名字代表是什么东东: 类型? 变量? 函数? 常量? 宏 ...? 甚至不需要去查找类型声明. 我们大脑中的模式匹配引擎可以非常可靠的处理这些命名规则.

命名规则具有一定随意性, 但相比按个人喜好命名, 一致性更重, 所以不管你怎么想, 规则总归是规则.

6.1. 通用命名规则

Tip

函数命名, 变量命名, 文件命名要有描述性; 少用缩写。

尽可能给有描述性的命名, 别心疼空间, 毕竟让代码易于新读者理解很重要. 不要用只有项目开发能理解的缩写, 也不要通过砍掉几个字母来缩写单词。

```
int price_count_reader;    // 无缩写
int num_errors;           // “num” 本来就很常见
int num_dns_connections;   // 人人都知道 “DNS” 是啥
```

Warning

```
int n;                    // 莫名其妙。
int nerr;                 // 怪缩写。
int n_comp_conns;        // 怪缩写。
int wgc_connections;     // 只有贵团队知道是啥意思。
int pc_reader;            // “pc” 有太多可能的解释了。
int cstmr_id;             // 有删减若干字母。
```

6.2. 文件命名

Tip

文件名要全部小写, 可以包含下划线 (_) 或连字符 (-). 按项目约定来. 如果并没有项目约定, “ _ ”更好。

可接受的文件命名:

```
* my_useful_class.cc
* my-useful-class.cc
* myusefulclass.cc
* muusefulclass_test.cc // ``_unittest`` 和 ``_regtest`` 已弃用。
```

C++ 文件要以 `.cc` 结尾, 头文件以 `.h` 结尾. 专门插入文本的文件则以 `.inc` 结尾, 参见: [ref:self-contained headers](#)。

不要使用已经存在于 `/usr/include` 下的文件名 (Yang.Y 注: 即编译器搜索系统头文件的路径), 如 `db.h`。

通常应尽量不要让文件名更加明确. `http_server_logs.h` 就比 `logs.h` 要好. 定义类时文件名一般成对出现, 如 `foo_bar.h` 和 `foo_bar.cc`, 对应于类 `FooBar`。

内联函数必须放在 `.h` 文件中. 如果内联函数比较短, 就直接放在 `.h` 中。

6.3. 类型命名

Tip

类型名称的每个单词首字母均大写, 不包含下划线: `MyExcitingClass`, `MyExcitingEnum`。

所有类型命名 —— 类, 结构体, 类型定义 (`typedef`), 枚举 —— 均使用相同约定. 例如:

```
// classes and structs
class UriTable { ...
class UriTableTester { ...
struct UriTableProperties { ...

// typedefs
typedef hash_map<UriTableProperties *, string> PropertiesMap;

// enums
enum UriTableErrors { ...
```

6.4. 变量命名

Tip

变量名一律小写, 单词之间用下划线连接. 类的成员变量以下划线结尾, 但结构体的就不用, 如: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`。

普通变量命名:

举例:

```
string table_name; // 可 - 用下划线。
string tablename;  // 可 - 全小写。
```

Warning

```
string tableName;    // 差 - 混合大小写。
```

类数据成员：

不管是静态的还是非静态的,类数据成员都可以和普通变量一样,但要接下划线。

```
class TableInfo {  
    ...  
private:  
    string table_name_;    // 可 - 尾后加下划线。  
    string tablename_;    // 可。  
    static Pool<TableInfo>* pool_;    // 可。  
};
```

结构体变量：

不管是静态的还是非静态的,结构体数据成员都可以和普通变量一样,不用像类那样接下划线：

```
struct UrlTableProperties {  
    string name;  
    int num_entries;  
}
```

结构体与类的讨论参考 [结构体 vs. 类](#) 一节。

全局变量：

对全局变量没有特别要求,少用就好,但如果你要用,可以用 g_ 或其它标志作为前缀,以便更好的区分局部变量。

6.5. 常量命名

Tip

在全局或类里的常量名称前加 k: kDaysInAWeek. 且除去开头的 k 之外每个单词开头字母均大写。

所有编译时常量,无论是局部的,全局的还是类中的,和其他变量稍微区别一下. k 后接大写字母开头的单词:

```
const int kDaysInAWeek = 7;
```

这规则适用于编译时的局部作用域常量,不过要按变量规则来命名也可以。

6.6. 函数命名

Tip

常规函数使用大小写混合, 取值和设值函数则要求与变量名匹配: `MyExcitingFunction()`, `MyExcitingMethod()`, `my_exciting_member_variable()`, `set_my_exciting_member_variable()`.

常规函数:

函数名的每个单词首字母大写, 没有下划线。

如果您的某函数出错时就要直接 crash, 那么就在函数名加上 `OrDie`. 但这函数本身必须集成在产品代码里, 且平时也可能会出错。

```
AddTableEntry()  
DeleteUrl()  
OpenFileOrDie()
```

取值和设值函数:

取值 (Accessors) 和设值 (Mutators) 函数要与存取的变量名匹配. 这儿摘录一个类, `num_entries_` 是该类的实例变量:

```
class MyClass {  
    public:  
        ...  
        int num_entries() const { return num_entries_; }  
        void set_num_entries(int num_entries) { num_entries_ = num_entries; }  
  
    private:  
        int num_entries_;  
};
```

其它非常短小的内联函数名也可以用小写字母, 例如. 如果你在循环中调用这样的函数甚至都不用缓存其返回值, 小写命名就可以接受.

6.7. 名字空间命名

Tip

名字空间用小写字母命名, 并基于项目名称和目录结构: `google_awesome_project`.

关于名字空间的讨论和如何命名, 参考 [名字空间](#) 一节.

6.8. 枚举命名

Tip

枚举的命名应当和 [常量](#) 或 [宏](#) 一致: `kEnumName` 或是 `ENUM_NAME`.

单独的枚举值应该优先采用 [常量](#) 的命名方式. 但 [宏](#) 方式的命名也可以接受. 枚举名 `UrlTableErrors` (以及 `AlternateUrlTableErrors`) 是类型, 所以要用大小写混合的方式.

```
enum UrlTableErrors {
    kOK = 0,
    kErrorOutOfMemory,
    kErrorMalformedInput,
};
enum AlternateUrlTableErrors {
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

2009 年 1 月之前, 我们一直建议采用 [宏](#) 的方式命名枚举值. 由于枚举值和宏之间的命名冲突, 直接导致了很多问题. 由此, 这里改为优先选择常量风格的命名方式. 新代码应该尽可能优先使用常量风格. 但是老代码没必要切换到常量风格, 除非宏风格确实会产生编译期问题.

6.9. 宏命名

Tip

你并不打算:ref:使用宏 `<preprocessor-macros>`, 对吧? 如果你一定要用, 像这样命名:
`MY_MACRO_THAT_SCARES_SMALL_CHILDREN`.

参考:ref:预处理宏 `<preprocessor-macros>`; 通常 不应该 使用宏. 如果不得不用, 其命名像枚举命名一样全部大写, 使用下划线:

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

6.10. 命名规则的特例

Tip

如果你命名的实体与已有 C/C++ 实体相似, 可参考现有命名策略.

`bigopen()`:

函数名, 参照 `open()` 的形式

`uint`:

`typedef`

`bigpos`:

struct 或 class, 参照 pos 的形式

sparse_hash_map:

STL 相似实体; 参照 STL 命名约定

LONGLONG_MAX:

常量, 如同 INT_MAX

译者(acgtyrant)笔记

1. 感觉 Google 的命名约定很高明, 比如写了简单的类 QueryResult, 接着又可以直接定义一个变量 query_result, 区分度很好; 再次, 类内变量以下划线结尾, 那么就可以直接传入同名的形参, 比如 TextQuery::TextQuery(std::string word) : word_(word) {}, 其中 word_ 自然是类内私有成员。

7. 注释

注释虽然写起来很痛苦,但对保证代码可读性至关重要.下面的规则描述了如何注释以及在哪儿注释.当然也要记住:注释固然很重要,但最好的代码本身应该是自文档化.有意义的类型名和变量名,要远胜过要用注释解释的含糊不清的名字.

你写的注释是给代码读者看的:下一个需要理解你的代码的人.慷慨些吧,下一个人可能就是你!

7.1. 注释风格

Tip

使用 `//` 或 `/* */`,统一就好.

`//` 或 `/* */` 都可以;但 `//` 更常用.要在如何注释及注释风格上确保统一.

7.2. 文件注释

Tip

在每一个文件开头加入版权公告,然后是文件内容描述.

法律公告和作者信息:

每个文件都应该包含以下项,依次是:

- 版权声明 (比如, Copyright 2008 Google Inc.)
- 许可证. 为项目选择合适的许可证版本 (比如, Apache 2.0, BSD, LGPL, GPL)
- 作者: 标识文件的原始作者.

如果你对原始作者的文件做了重大修改,将你的信息添加到作者信息里.这样当其他人对该文件有疑问时可以知道该联系谁.

文件内容:

紧接着版权许可和作者信息之后,每个文件都要用注释描述文件内容.

通常, `.h` 文件要对所声明的类的功能和用法作简单说明. `.cc` 文件通常包含了更多的实现细节或算法技巧讨论,如果你感觉这些实现细节或算法技巧讨论对于理解 `.h` 文件有帮助,可以将该注释挪到 `.h`,并在 `.cc` 中指出文档在 `.h`.

不要简单的在 .h 和 .cc 间复制注释. 这种偏离了注释的实际意义.

7.3. 类注释

Tip

每个类的定义都要附带一份注释, 描述类的功能和用法.

```
// Iterates over the contents of a GargantuanTable. Sample usage:
//     GargantuanTable_Iterator* iter = table->NewIterator();
//     for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//         process(iter->key(), iter->value());
//     }
//     delete iter;
class GargantuanTable_Iterator {
    ...
};
```

如果你觉得已经在文件顶部详细描述了该类, 想直接简单的来上一句“完整描述见文件顶部”也不打紧, 但务必确保有这类注释.

如果类有任何同步前提, 文档说明之. 如果该类的实例可被多线程访问, 要特别注意文档说明多线程环境下相关的规则和常量使用.

7.4. 函数注释

Tip

函数声明处注释描述函数功能; 定义处描述函数实现.

函数声明:

注释位于声明之前, 对函数功能及用法进行描述. 注释使用叙述式 (“Opens the file”) 而非指令式 (“Open the file”); 注释只是为了描述函数, 而不是命令函数做什么. 通常, 注释不会描述函数如何工作. 那是函数定义部分的事情.

函数声明处注释的内容:

- 函数的输入输出.
- 对类成员函数而言: 函数调用期间对象是否需要保持引用参数, 是否会释放这些参数.
- 如果函数分配了空间, 需要由调用者释放.
- 参数是否可以 NULL.
- 是否存在函数使用上的性能隐患.
- 如果函数是可重入的, 其同步前提是什么?

举例如下:

```
// Returns an iterator for this table. It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//     Iterator* iter = table->NewIterator();
//     iter->Seek("");
//     return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

但也要避免罗罗嗦嗦, 或做些显而易见的说明. 下面的注释就没有必要加上 “returns false otherwise”, 因为已经暗含其中了:

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

注释构造/析构函数时, 切记读代码的人知道构造/析构函数是干啥的, 所以 “destroys this object” 这样的注释是没有意义的. 注明构造函数对参数做了什么 (例如, 是否取得指针所有权) 以及析构函数清理了什么. 如果都是些无关紧要的内容, 直接省掉注释. 析构函数前没有注释是很正常的.

函数定义:

每个函数定义时要用注释说明函数功能和实现要点. 比如说说你用的编程技巧, 实现的大致步骤, 或解释如此实现的理由, 为什么前半部分要加锁而后半部分不需要.

不要从 .h 文件或其他地方的函数声明处直接复制注释. 简要重述函数功能是可以的, 但注释重点要放在如何实现上.

7.5. 变量注释

Tip

通常变量名本身足以很好说明变量用途. 某些情况下, 也需要额外的注释说明.

类数据成员:

每个类数据成员 (也叫实例变量或成员变量) 都应该用注释说明用途. 如果变量可以接受 NULL 或 -1 等警戒值, 须加以说明. 比如:

```
private:
    // Keeps track of the total number of entries in the table.
    // Used to ensure we do not go over the limit. -1 means
```

```
// that we don't yet know how many entries the table has.  
int num_total_entries_;
```

全局变量:

和数据成员一样, 所有全局变量也要注释说明含义及用途. 比如:

```
// The total number of tests cases that we run through in this regression test  
const int kNumTestCases = 6;
```

7.6. 实现注释

Tip

对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

代码前注释:

巧妙或复杂的代码段前要加注释. 比如:

```
// Divide result by two, taking into account that x  
// contains the carry from the add.  
for (int i = 0; i < result->size(); i++) {  
    x = (x << 8) + (*result)[i];  
    (*result)[i] = x >> 1;  
    x &= 1;  
}
```

行注释:

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释. 比如:

```
// If we have enough memory, mmap the data portion too.  
mmap_budget = max<int64>(0, mmap_budget - index->length());  
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))  
    return; // Error already logged.
```

注意, 这里用了两段注释分别描述这段代码的作用, 和提示函数返回时错误已经被记入日志.

如果你需要连续进行多行注释, 可以使之对齐获得更好的可读性:

```
DoSomething(); // Comment here so the comments line up.  
DoSomethingElseThatIsLonger(); // Comment here so there are two spaces between  
                                // the code and the comment.  
{ // One space before comment when opening a new scope is allowed,  
  // thus the comment lines up with the following comments and code.  
  DoSomethingElse(); // Two spaces before line comments normally.  
}
```

NULL, true/false, 1, 2, 3...:

向函数传入 `NULL`, 布尔值或整数时, 要注释说明含义, 或使用常量让代码望文知意. 例如, 对比:

Warning

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL); // What are these arguments??
```

和:

```
bool success = CalculateSomething(interesting_value,
                                  10,          // Default base value.
                                  false,       // Not the first time we're callin
                                  NULL);      // No callback.
```

或使用常量或描述性变量:

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

不允许:

注意 永远不要用自然语言翻译代码作为注释. 要假设读代码的人 C++ 水平比你高, 即便他/她可能不知道你的用意:

Warning

```
// 现在, 检查 b 数组并确保 i 是否存在,
// 下一个元素是 i+1.
...      // 天哪. 令人崩溃的注释.
```

7.7. 标点, 拼写和语法

Tip

注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

注释的通常写法是包含正确大小写和结尾句号的完整语句. 短一点的注释 (如代码行尾注释) 可以随意点, 依然要注意风格的一致性. 完整的语句可读性更好, 也可以说明该注释是完整的, 而不是一些不成熟的想法.

虽然被别人指出该用分号时却用了逗号多少有些尴尬,但清晰易读的代码还是很重要的. 正确的标点, 拼写和语法对此会有所帮助.

7.8. TODO 注释

Tip

对那些临时的, 短期的解决方案, 或已经够好但仍不完美的代码使用 TODO 注释.

TODO 注释要使用全大写的字符串 TODO, 在随后的圆括号里写上你的大名, 邮件地址, 或其它身份标识. 冒号是可选的. 主要目的是让添加注释的人 (也是可以请求提供更多细节的人) 可根据规范的 TODO 格式进行查找. 添加 TODO 注释并不意味着你要自己来修正.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.  
// TODO(Zeke) change this to use relations.
```

如果加 TODO 是为了在“将来某一天做某事”, 可以附上一个非常明确的时间“Fix by November 2005”), 或者一个明确的事项 (“Remove this code when all clients can handle XML responses.”).

7.9. 弃用注释

Tip

通过弃用注释 (DEPRECATED comments) 以标记某接口点 (interface points) 已弃用。

您可以写上包含全大写的 DEPRECATED 的注释, 以标记某接口为弃用状态. 注释可以放在接口声明前, 或者同一行。

在 DEPRECATED 一词后, 留下您的名字, 邮箱地址以及括号补充。

仅仅标记接口为 DEPRECATED 并不会让大家不约而同地弃用, 您还得亲自主动修正调用点 (callsites), 或是找个帮手。

修正好的代码应该不会再涉及弃用接口点了, 着实改用新接口点. 如果您不知从何下手, 可以找标记弃用注释的当事人一起商量。

译者 (YuleFox) 笔记

1. 关于注释风格, 很多 C++ 的 coders 更喜欢行注释, C coders 或许对块注释依然情有独钟, 或者在文件头大段大段的注释时使用块注释;
2. 文件注释可以炫耀你的成就, 也是为了捅了篓子别人可以找你;
3. 注释要言简意赅, 不要拖沓冗余, 复杂的东西简单化和简单的东西复杂化都是要被鄙视的;
4. 对于 Chinese coders 来说, 用英文注释还是用中文注释, it is a problem, 但不管怎样, 注释是为了让别人看懂, 难道是为了炫耀编程语言之外的你的母语或外语水平吗;

5. 注释不要太乱, 适当的缩进才会让人乐意看. 但也没有必要规定注释从第几列开始 (我自己写代码的时候总喜欢这样), UNIX/LINUX 下还可以约定是使用 tab 还是 space, 个人倾向于 space;
6. TODO 很不错, 有时候, 注释确实是为了标记一些未完成的或完成的不尽如人意的地方, 这样一搜索, 就知道还有哪些活要干, 日志都省了.

8. 格式

代码风格和格式确实比较随意, 但一个项目中所有人遵循同一风格是非常容易的. 个体未必同意下述每一处格式规则, 但整个项目服从统一的编程风格是很重要的, 只有这样才能让所有人能很轻松的阅读和理解代码.

另外, 我们写了一个 [emacs 配置文件](http://google-styleguide.googlecode.com/svn/trunk/google-c-style.el) [http://google-styleguide.googlecode.com/svn/trunk/google-c-style.el] 来帮助你正确的格式化代码.

8.1. 行长度

Tip

每一行代码字符数不超过 80.

我们也认识到这条规则是有争议的, 但很多已有代码都已经遵照这一规则, 我们感觉一致性更重要.

优点:

提倡该原则的人主张强迫他们调整编辑器窗口大小很野蛮. 很多人同时并排开几个代码窗口, 根本没有多余空间拉伸窗口. 大家都把窗口最大尺寸加以限定, 并且 80 列宽是传统标准. 为什么要改变呢?

缺点:

反对该原则的人则认为更宽的代码行更易阅读. 80 列的限制是上个世纪 60 年代的大型机的古板缺陷; 现代设备具有更宽的显示屏, 很轻松的可以显示更多代码.

结论:

80 个字符是最大值.

特例:

- 如果一行注释包含了超过 80 字符的命令或 URL, 出于复制粘贴的方便允许该行超过 80 字符.
- 包含长路径的 `#include` 语句可以超出 80 列. 但应该尽量避免.
- [头文件保护](#) 可以无视该原则.

8.2. 非 ASCII 字符

Tip

尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.

即使是英文, 也不应将用户界面的文本硬编码到源代码中, 因此非 ASCII 字符要少用. 特殊情况下可以适当包含此类字符. 如, 代码分析外部数据文件时, 可以适当硬编码数据文件中作为分隔符的非 ASCII 字符串; 更常见的是 (不需要本地化的) 单元测试代码可能包含非 ASCII 字符串. 此类情况下, 应使用 UTF-8 编码, 因为很多工具都可以理解和处理 UTF-8 编码.

十六进制编码也可以, 能增强可读性的情况下尤其鼓励 —— 比如 `"\xEF\xBB\xBF"` 在 Unicode 中是零宽度无间断的间隔符号, 如果不用十六进制直接放在 UTF-8 格式的源文件中, 是看不到的.

(Yang.Y 注: `"\xEF\xBB\xBF"` 通常用作 UTF-8 with BOM 编码标记)

用 `u8` 前缀以把带 `uXXXX` 转义序列的字符串字面值编码成 UTF-8. 不要用在本身就带 UTF-8 字符的字符串字面值上, 因为如果编译器不把源代码识别成 UTF-8, 输出就会出错.

别用 C++11 的 `char16_t` 和 `char32_t`, 它们和 UTF-8 文本没有关系, `wchar_t` 同理, 除非您写的代码要调用 Windows API, 后者有用到 `wchar_t` 扩展.

8.3. 空格还是制表位

Tip

只使用空格, 每次缩进 2 个空格.

我们使用空格缩进. 不要在代码中使用制符表. 你应该设置编辑器将制符表转为空格.

8.4. 函数声明与定义

Tip

返回类型和函数名在同一行, 参数也尽量放在同一行, 如果放不下就对形参分行.

函数看上去像这样:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
    DoSomething();
    ...
}
```

如果同一行文本太多, 放不下所有参数:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
```

```

Type par_name3) {
    DoSomething();
    ...
}

```

甚至连第一个参数都放不下:

```

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,    // 4 空格缩进
    Type par_name2,
    Type par_name3) {
    DoSomething();    // 2 空格缩进
    ...
}

```

注意以下几点:

- 如果返回类型和函数名在一行放不下,分行。
- 如果返回类型那个与函数声明或定义分行了,不要缩进。
- 左圆括号总是和函数名在同一行;
- 函数名和左圆括号间没有空格;
- 圆括号与参数间没有空格;
- 左大括号总在最后一个参数同一行的末尾处;
- 如果其它风格规则允许的话,右大括号总是单独位于函数最后一行,或者与左大括号同一行。
- 右大括号和左大括号间总是有一个空格;
- 函数声明和定义中的所有形参必须有命名且一致;
- 所有形参应尽可能对齐;
- 缺省缩进为 2 个空格;
- 换行后的参数保持 4 个空格的缩进;

如果有些参数没有用到, 在函数定义处将参数名注释起来:

```

// 接口中形参恒有命名。
class Shape {
public:
    virtual void Rotate(double radians) = 0;
}

// 声明中形参恒有命名。
class Circle : public Shape {
public:
    virtual void Rotate(double radians);
}

// 定义中注释掉无用变量。
void Circle::Rotate(double /*radians*/) {}

```

Warning

```

// 差 - 如果将来有人要实现, 很难猜出变量是干什么用的。
void Circle::Rotate(double) {}

```

8.5. Lambda 表达式

Tip

其它函数怎么格式化形参和函数体,Lambda 表达式就怎么格式化;捕获列表同理。

若用引用捕获,在变量名和 & 之间不留空格。

```
int x = 0;
auto add_to_x = [&x](int n) { x += n; };
```

短 lambda 就写得和内联函数一样。

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&blacklist](int i) {
    return blacklist.find(i) != blacklist.end();
}),
    digits.end());
```

8.6. 函数调用

Tip

要么一行写完函数调用,要么在圆括号里对参数分行,要么参数另起一行且缩进四格。如果没有其它顾虑的话,尽可能精简行数,比如把多个参数适当地放在同一行里。

函数调用遵循如下形式:

```
bool retval = DoSomething(argument1, argument2, argument3);
```

如果同一行放不下,可断为多行,后面每一行都和第一个实参对齐,左圆括号后和右圆括号前不要留空格:

```
bool retval = DoSomething(averyveryveryverylongargument1,
    argument2, argument3);
```

参数也可以放在次行,缩进四格:

```
if (...) {
    ...
    ...
    if (...) {
        DoSomething(
            argument1, argument2, // 4 空格缩进
            argument3, argument4);
    }
}
```

把多个参数放在同一行,是为了减少函数调用所需的行数,除非影响到可读性。有人认为把每个参数都独立成行,不仅更好读,而且方便编辑参数。不过,比起所谓的参数编辑,我们更看重可读

性,且后者比较好办:

如果一些参数本身就是略复杂的表达式,且降低了可读性。那么可以直接创建临时变量描述该表达式,并传递给函数:

```
int my_heuristic = scores[x] * y + bases[x];
bool retval = DoSomething(my_heuristic, x, y, z);
```

或者放着不管,补充上注释:

```
bool retval = DoSomething(scores[x] * y + bases[x],    // Score heuristic.
                          x, y, z);
```

如果某参数独立成行,对可读性更有帮助的话,就这么办。

此外,如果一系列参数本身就有一定的结构,可以酌情地按其结构来决定参数格式:

```
// 通过 3x3 矩阵转换 widget.
my_widget.Transform(x1, x2, x3,
                   y1, y2, y3,
                   z1, z2, z3);
```

8.7. 列表初始化格式

Tip

您平时怎么格式化函数调用,就怎么格式化:[ref:braced_initializer_list](#)。

如果列表初始化伴随着名字,比如类型或变量名,您可以当名字是函数、{} 是函数调用的括号来格式化它。反之,就当它有个长度为零的名字。

```
// 一行列表初始化示范。
return {foo, bar};
functioncall({foo, bar});
pair<int, int> p{foo, bar};

// 当不得不断行时。
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other values},
    SomeOtherType{"Slightly shorter string",
        some, other, values}};
SomeType variable{
    "This is too long to fit all in one line";
MyType m = {    // 注意了,您可以在 { 前断行。
    superlongvariablename1,
    superlongvariablename2,
```

```
{short, interior, list},
{interiorwrappinglist,
 interiorwrappinglist2}};
```

8.8. 条件语句

Tip

倾向于不在圆括号内使用空格. 关键字 `if` 和 `else` 另起一行.

对基本条件语句有两种可以接受的格式. 一种在圆括号和条件之间有空格, 另一种没有.

最常见的是没有空格的格式. 哪种都可以, 但 保持一致性. 如果你是在修改一个文件, 参考当前已有格式. 如果是写新的代码, 参考目录下或项目中其它文件. 还在徘徊的话, 就不要加空格了.

```
if (condition) {  圆括号里没空格紧邻。
    ...  // 2 空格缩进。
} else {  // else 与 if 的右括号同一行。
    ...
}
```

如果你更喜欢在圆括号内部加空格:

```
if ( condition ) {  // 圆括号与空格紧邻 - 不常见
    ...  // 2 空格缩进。
} else {  // else 与 if 的右括号同一行。
    ...
}
```

注意所有情况下 `if` 和左圆括号间都有个空格. 右圆括号和左大括号之间也要有个空格:

Warning

```
if(condition)      // 差 - IF 后面没空格。
if (condition){     // 差 - { 前面没空格。
if(condition){      // 变本加厉地差。
```

```
if (condition) {  // 可 - IF 和 { 都与空格紧邻。
```

如果能增强可读性, 简短的条件语句允许写在同一行. 只有当语句简单并且没有使用 `else` 子句时使用:

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

如果语句有 `else` 分支则不允许:

Warning


```
// 不可以这样子 - 当有 ELSE 分支时 IF 块却只有一行
if (x) DoThis();
else DoThat();
```

通常, 单行语句不需要使用大括号, 如果你喜欢用也没问题; 复杂的条件或循环语句用大括号可读性会更好. 也有一些项目要求 if 必须总是使用大括号:

```
if (condition)
    DoSomething(); // 2 空格缩进。

if (condition) {
    DoSomething(); // 2 空格缩进。
}
```

但如果语句中某个 if-else 分支使用了大括号的话, 其它分支也必须使用:

Warning

```
// 不可以这样子 - IF 有大括号 ELSE 却没有。
if (condition) {
    foo;
} else
    bar;

// 不可以这样子 - ELSE 有大括号 IF 却没有。
if (condition)
    foo;
else {
    bar;
}
```

```
// 只要其中一个分支用了大括号, 两个分支都要用上大括号。
if (condition) {
    foo;
} else {
    bar;
}
```

8.9. 循环和开关选择语句

Tip

switch 语句可以使用大括号分段, 以表明 cases 之间不是连在一起的. 在单语句循环里, 括号可用可不用. 空循环体应使用 {} 或 continue.

switch 语句中的 case 块可以使用大括号也可以不用, 取决于你的个人喜好. 如果用的话, 要按照下文所述的方法.

如果有不满足 case 条件的枚举值, switch 应该总是包含一个 default 匹配 (如果有输入值没有 case 去处理, 编译器将报警). 如果 default 应该永远执行不到, 简单的加条 assert:

```
switch (var) {
    case 0: { // 2 空格缩进
        ... // 4 空格缩进
        break;
    }
    case 1: {
        ...
        break;
    }
    default: {
        assert(false);
    }
}
```

在单语句循环里,括号可用可不用:

```
for (int i = 0; i < kSomeNumber; ++i)
    printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {
    printf("I take it back\n");
}
```

空循环体应使用 {} 或 continue, 而不是一个简单的分号.

```
while (condition) {
    // 反复循环直到条件失效。
}
for (int i = 0; i < kSomeNumber; ++i) {} // 可 - 空循环体。
while (condition) continue; // 可 - continue 表明没有逻辑。
```

Warning

```
while (condition); // 差 - 看起来仅仅只是 while/loop 的一部分。
```

8.10. 指针和引用表达式

Tip

句点或箭头前后不要有空格. 指针/地址操作符 (*, &) 之后不能有空格.

下面是指针和引用表达式的正确使用范例:

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

注意:啊

- 在访问成员时, 句点或箭头前后没有空格.
- 指针操作符 * 或 & 后没有空格.

在声明指针变量或参数时, 星号与类型或变量名紧挨都可以:

```
// 好样的, 空格前置。  
char *c;  
const string &str;  
  
// 好样的, 空格后置。  
char* c; // 但别忘了 "char* c, *d, *e, ...;"!  
const string& str;
```

Warning

```
char * c; // 差 - * 两边都有空格  
const string & str; // 差 - & 两边都有空格。
```

在单个文件内要保持风格一致, 所以, 如果是修改现有文件, 要遵照该文件的风格。

8.11. 布尔表达式

Tip

如果一个布尔表达式超过 [标准行宽](#), 断行方式要统一一下。

下例中, 逻辑与 (&&) 操作符总位于行尾:

```
if (this_one_thing > this_other_thing &&  
    a_third_thing == a_fourth_thing &&  
    yet_another & last_one) {  
    ...  
}
```

注意, 上例的逻辑与 (&&) 操作符均位于行尾. 这格式在 Google 里很常见, 您要把所有操作符放在开头也可以. 可以考虑额外插入圆括号, 合理使用的话对增强可读性是很有帮助的. 此外直接用符号形式的操作符, 比如 && 和 ~, 不要用词语形式的 and 和 compl.

8.12. 函数返回值

Tip

return 表达式里时没必要都用圆括号。

假如您写 `x = epr` 时本来就会加上括号, 那 `return expr;` 也可如法炮制。

函数返回时不要使用圆括号:

```
return result; // 返回值很简单, 没有圆括号。  
// 可以用圆括号把复杂表达式圈起来, 改善可读性。
```

```
return (some_long_condition &&
        another_condition);
```

Warning

```
return (value);           // 毕竟您从来不会写 var = (value);
return(result);           // return 可不是函数！
```

8.13. 变量及数组初始化

Tip

用 `=`, `()` 和 `{}` 均可.

您可以用 `=`, `()` 和 `{}`, 以下都对:

```
int x = 3;
int x(3);
int x{3};
string name("Some Name");
string name = "Some Name";
string name{"Some Name"};
```

请务必小心列表初始化 `{...}` 用 `std::initializer_list` 构造函数初始化出的类型。非空列表初始化就会优先调用 `std::initializer_list`, 不过空列表初始化除外, 后者原则上会调用默认构造函数。为了强制禁用 `std::initializer_list` 构造函数, 请改用括号。

```
vector<int> v(100, 1); // A vector of 100 1s.
vector<int> v{100, 1}; // A vector of 100, 1.
```

此外, 列表初始化不允许整型类型的四舍五入, 这可以用来避免一些类型上的编程失误。

```
int pi(3.14); // 可 -- pi == 3.
int pi{3.14}; // Compile error: narrowing conversion.
```

8.14. 预处理指令

Tip

预处理指令不要缩进, 从行首开始.

即使预处理指令位于缩进代码块中, 指令也应从行首开始.

```
// 可 - directives at beginning of line
if (lopsided_score) {
#ifdef DISASTER_PENDING // 正确 -- 行开头起。
    DropEverything();
#endif
```

```
BackToNormal();  
}
```

Warning

```
// 差 - indented directives  
if (lopsided_score) {  
    #if DISASTER_PENDING // 错了! "#if" 应该放在行开头  
    DropEverything();  
    #endif // 错了! "#endif" 不要缩进  
    BackToNormal();  
}
```

8.15. 类格式

Tip

访问控制块的声明依次序是 `public:`, `protected:`, `private:`, 每次缩进 1 个空格.

类声明 (对类注释不了解的话, 参考 [类注释](#)) 的基本格式如下:

```
class MyClass : public OtherClass {  
public:    // 注意有 1 空格缩进!  
    MyClass(); // 照常, 2 空格缩进.  
    explicit MyClass(int var);  
    ~MyClass() {}  
  
    void SomeFunction();  
    void SomeFunctionThatDoesNothing() {  
    }  
  
    void set_some_var(int var) { some_var_ = var; }  
    int some_var() const { return some_var_; }  
  
private:  
    bool SomeInternalFunction();  
  
    int some_var_;  
    int some_other_var_;  
    DISALLOW_COPY_AND_ASSIGN(MyClass);  
};
```

注意事项:

- 所有基类名应在 80 列限制下尽量与子类名放在同一行.
- 关键词 `public:`, `protected:`, `private:` 要缩进 1 个空格.
- 除第一个关键词 (一般是 `public`) 外, 其他关键词前要空一行. 如果类比较小的话也可以不空.
- 这些关键词后不要保留空行.
- `public` 放在最前面, 然后是 `protected`, 最后是 `private`.
- 关于声明顺序的规则请参考 [声明顺序](#) 一节.

8.16. 构造函数初始值列表

Tip

构造函数初始值列表放在同一行或按四格缩进并排几行。

下面两种初始值列表方式都可以接受：

```
// 当全放在一行合适时：  
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1) {
```

或

```
// 如果要断成多行，缩进四格，冒号放在第一行初始化句：  
MyClass::MyClass(int var)  
    : some_var_(var),           // 4 空格缩进  
    some_other_var_(var + 1) {  // 对准  
    ...  
    DoSomething();  
    ...  
}
```

8.17. 名字空间格式化

Tip

名字空间内容不缩进。

[名字空间](#) 不要增加额外的缩进层次，例如：

```
namespace {  
  
void foo() { // 正确。命名空间内没有额外的缩进。  
    ...  
}  
  
} // namespace
```

不要缩进名字空间：

Warning

```
namespace {  
  
    // 错，缩进多余了。  
    void foo() {  
        ...  
    }  
  
} // namespace
```

声明嵌套命名空间时,每命名空间都独立成行。

```
namespace foo {
namespace bar {
```

8.18. 水平留白

Tip

水平留白的使用因地制宜. 永远不要在行尾添加没意义的留白.

常规:

```
void f(bool b) { // 左大括号前恒有空格。
...
int i = 0; // 分号前不加空格。
int x[] = { 0 }; // 大括号内部可与空格紧邻也不可，不过两边都要加上。
int x[] = {0};
// 继承与初始化列表中的冒号前后恒有空格。
class Foo : public Bar {
public:
// 至于内联函数实现，在大括号内部加上空格并编写实现。
Foo(int b) : Bar(), baz_(b) {} // 大括号里面是空的话，不加空格。
void Reset() { baz_ = 0; } // 用括号把大括号与实现分开。
...
}
```

添加冗余的留白会给其他人编辑时造成额外负担. 因此, 行尾不要留空格. 如果确定一行代码已经修改完毕, 将多余的空格去掉; 或者在专门清理空格时去掉(确信没有其他人在处理). (Yang.Y 注: 现在大部分代码编辑器稍加设置后, 都支持自动删除行首/行尾空格, 如果不支持, 考虑换一款编辑器或 IDE)

循环和条件语句:

```
if (b) { // if 条件语句和循环语句关键字后均有空格。
} else { // else 前后有空格。
}
while (test) {} // 圆括号内部不紧邻空格。
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) { // 循环和条件语句的圆括号里可以与空格紧邻。
if ( test ) { // 圆括号，但这很少见。总之要一致。
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) { // 循环里内 ; 后恒有空格， ; 前可以加个空格。
switch (i) {
case 1: // switch case 的冒号前无空格。
...
case 2: break; // 如果冒号有代码，加个空格。
}
```

操作符:

```
// 赋值操作系统前后恒有空格。
```



```

x = 0;

// 其它二元操作符也前后恒有空格，不过对 factors 前后不加空格也可以。
// 圆括号内部不紧邻空格。
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// 在参数和一元操作符之间不加空格。
x = -5;
++x;
if (x && !y)
    ...

```

模板和转换:

```

// 尖叫括号(< and >) 不与空格紧邻，< 前没有空格，>( 之间也没有。
vector<string> x;
y = static_cast<char*>(x);

// 在类型与指针操作符之间留空格也可以，但要保持一致。
vector<char *> x;
set<list<string>> x;           // 在 C++11 代码里可以这样用了。
set<list<string> > x;        // C++03 中要在 > > 里留个空格。

// 您或许可以在 < < 里加上一对对称的空格。
set< list<string> > x;

```

8.19. 垂直留白

Tip

垂直留白越少越好.

这不仅仅是规则而是原则问题了: 不在万不得已, 不要使用空行. 尤其是: 两个函数定义之间的空行不要超过 2 行, 函数体首尾不要留空行, 函数体中也不要随意添加空行.

基本原则是: 同一屏可以显示的代码越多, 越容易理解程序的控制流. 当然, 过于密集的代码块和过于疏松的代码块同样难看, 取决于你的判断. 但通常是垂直留白越少越好.

空行心得如下:

- 函数体内开头或结尾的空行可读性微乎其微。
- 在多重 if-else 块里加空行或许有点可读性。

译者 (YuleFox) 笔记

1. 对于代码格式, 因人, 系统而异各有优缺点, 但同一个项目中遵循同一标准还是有必要的;
2. 行宽原则上不超过 80 列, 把 22 寸的显示屏都占完, 怎么说不过去;
3. 尽量不使用非 ASCII 字符, 如果使用的话, 参考 UTF-8 格式 (尤其是 UNIX/Linux 下, Windows

下可以考虑宽字符), 尽量不将字符串常量耦合到代码中, 比如独立出资源文件, 这不仅仅是风格问题了;

4. UNIX/Linux 下无条件使用空格, MSVC 的话使用 Tab 也无可厚非;
5. 函数参数, 逻辑条件, 初始化列表: 要么所有参数和函数名放在同一行, 要么所有参数并排分行;
6. 除函数定义的左大括号可以置于行首外, 包括函数/类/结构体/枚举声明, 各种语句的左大括号置于行尾, 所有右大括号独立成行;
7. ./-> 操作符前后不留空格, */& 不要前后都留, 一个就可, 靠左靠右依各人喜好;
8. 预处理指令/命名空间不使用额外缩进, 类/结构体/枚举/函数/语句使用缩进;
9. 初始化用 = 还是 () 依个人喜好, 统一就好;
10. return 不要加 ();
11. 水平/垂直留白不要滥用, 怎么易读怎么来.
12. 关于 UNIX/Linux 风格为什么要把左大括号置于行尾 (.cc 文件的函数实现处, 左大括号位于行首), 我的理解是代码看上去比较简约, 想想行首除了函数体被一对大括号封在一起之外, 只有右大括号的代码看上去确实也舒服; Windows 风格将左大括号置于行首的优点是匹配情况一目了然.

译者(acgtyrant)笔记

1. 80 行限制事实上有助于避免代码可读性失控, 比如超多重嵌套块, 超多重函数调用等等.
2. Linux 上设置好了 Locale 就几乎一劳永逸设置好所有开发环境的编码, 不像奇葩的 Windows.
3. Google 强调有一对 if-else 时, 不论有没有嵌套, 都要有大括号. Apple 正好`有栽过跟头`
<<http://coolshell.cn/articles/11112.html>>`_.
4. 其实我主张指针/地址操作符与变量名紧邻, `int* a, b` vs `int *a, b`, 新手会误以为前者的 `b` 是 `int *` 变量, 但后者就不一样了, 高下立判.
5. 在这风格指南里我才刚知道 C++ 原来还有所谓的 [Alternative operator representations](http://en.cppreference.com/w/cpp/language/operator_alternative) [http://en.cppreference.com/w/cpp/language/operator_alternative], 大概没人用吧.
6. 注意构造函数初始值列表(Constructor Initializer List)与列表初始化(Initializer List)是两码事, 我就差点混淆了它们的翻译.
7. 事实上, 如果您熟悉英语本身的书写规则, 就会发现该风格指南在格式上的规定与英语语法相当一脉相承. 比如普通标点符号和单词后面还有文本的话, 总会留一个空格; 特殊符号与单词之间就不用留了, 比如 `if (true)` 中的圆括号与 `true`.
8. 本风格指南没有明确规定 void 函数里要不要用 return 语句, 不过就 Google 开源项目 `leveldb` 并没有写; 此外从 [Is a blank return statement at the end of a function whos return type is void necessary?](http://stackoverflow.com/questions/9316717/is-a-blank-return-statement-at-the-end-of-a-function-whos-return-type-is-void-ne) [<http://stackoverflow.com/questions/9316717/is-a-blank-return-statement-at-the-end-of-a-function-whos-return-type-is-void-ne>] 来看, `return;` 比 `return ;` 更约定俗成(事实上 `cpplint` 会对后者报错, 指出分号前有多余的空格), 且可用来提前跳出函数栈.

9. 规则特例

前面说明的编程习惯基本都是强制性的. 但所有优秀的规则都允许例外, 这里就是探讨这些特例.

9.1. 现有不合规范的代码

Tip

对于现有不符合既定编程风格的代码可以网开一面.

当你修改使用其他风格的代码时, 为了与代码原有风格保持一致可以不使用本指南约定. 如果不放心可以与代码原作者或现在的负责人员商讨, 记住, 一致性 包括原有的一致性.

9.2. Windows 代码

Tip

Windows 程序员有自己的编程习惯, 主要源于 Windows 头文件和其它 Microsoft 代码. 我们希望大家都可以顺利读懂你的代码, 所以针对所有平台的 C++ 编程只给出一个单独的指南.

如果你习惯使用 Windows 编码风格, 这儿有必要重申一下某些你可能会忘记的指南:

- 不要使用匈牙利命名法 (比如把整型变量命名成 `iNum`). 使用 Google 命名约定, 包括对源文件使用 `.cc` 扩展名.
- Windows 定义了很多原生类型的同义词 (YuleFox 注: 这一点, 我也很反感), 如 `DWORD`, `HANDLE` 等等. 在调用 Windows API 时这是完全可以接受甚至鼓励的. 但还是尽量使用原有的 C++ 类型, 例如, 使用 `const TCHAR *` 而不是 `LPCTSTR`.
- 使用 Microsoft Visual C++ 进行编译时, 将警告级别设置为 3 或更高, 并将所有 warnings 当作 errors 处理.
- 不要使用 `#pragma once`; 而应该使用 Google 的头文件保护规则. 头文件保护的路径应该相对于项目根目录 (Yang.Y 注: 如 `#ifndef SRC_DIR_BAR_H_`, 参考 `#define` 保护一节).
- 除非万不得已, 不要使用任何非标准的扩展, 如 `#pragma` 和 `__declspec`. 允许使用 `__declspec(dllimport)` 和 `__declspec(dllexport)`; 但你必须通过宏来使用, 比如 `DLLIMPORT` 和 `DLLEXPORT`, 这样其他人在分享使用这些代码时很容易就去掉这些扩展.

在 Windows 上, 只有很少的一些情况下, 我们可以偶尔违反规则:

- 通常我们 [禁止使用多重继承](#), 但在使用 COM 和 ATL/WTL 类时可以使用多重继承. 为了实现 COM 或 ATL/WTL 类/接口, 你可能不得不使用多重实现继承.
- 虽然代码中不应该使用异常, 但是在 ATL 和部分 STL (包括 Visual C++ 的 STL) 中异常被广泛使用. 使用 ATL 时, 应定义 `_ATL_NO_EXCEPTIONS` 以禁用异常. 你要研究一下是否

能够禁用 STL 的异常, 如果无法禁用, 后用编译器异常也可以. (注意这只是为了编译 STL, 自己代码里仍然不要含异常处理.)

- 通常为了利用头文件预编译, 每个每个源文件的开头都会包含一个名为 StdAfx.h 或 precompile.h 的文件. 为了使代码方便与其他项目共享, 避免显式包含此文件 (precompile.cc), 使用 /FI 编译器选项以自动包含.
- 资源头文件通常命名为 resource.h, 且只包含宏的, 不需要遵守本风格指南.

10. 结束语

Tip

运用常识和判断力, 并保持一致.

编辑代码时, 花点时间看看项目中的其它代码, 并熟悉其风格. 如果其它代码中 `if` 语句使用空格, 那么你也要使用. 如果其中的注释用星号 (*) 围成一个盒子状, 你同样要这么做.

风格指南的重点在于提供一个通用的编程规范, 这样大家可以把精力集中在实现内容而不是表现形式上. 我们展示了全局的风格规范, 但局部风格也很重要, 如果你在一个文件中新加的代码和原有代码风格相去甚远, 这就破坏了文件本身的整体美观, 也影响阅读, 所以要尽量避免.

好了, 关于编码风格写的够多了; 代码本身才更有趣. 尽情享受吧!

Objective-C 风格指南 - 内容目录

- [Google Objective-C Style Guide 中文版](#)

- [译者的话](#)
 - [ewanke](#)
 - [Yang.Y](#)

- [背景介绍](#)
- [例子](#)

- [留白和格式](#)

- [空格 vs. 制表符](#)
- [行宽](#)
- [方法声明和定义](#)
- [方法调用](#)
- [@public 和 @private](#)
- [异常](#)
- [协议名](#)
- [块\(闭包\)](#)

- [命名](#)

- [文件名](#)
- [Objective-C++](#)
- [类名](#)
- [类别名](#)
- [Objective-C 方法名](#)
- [变量名](#)
 - [普通变量名](#)
 - [实例变量](#)
 - [常量](#)

- [注释](#)

- [文件注释](#)
 - [版权信息及作者](#)
- [声明部分的注释](#)
- [实现部分的注释](#)
- [对象所有权](#)

- [Cocoa 和 Objective-C 特性](#)

- [成员变量应该是 @private](#)
- [明确指定构造函数](#)
- [重载指定构造函数](#)
- [重载 NSObject 的方法](#)
- [初始化](#)
- [避免 +new](#)
- [保持公共 API 简单](#)
- [#import and #include](#)
- [使用根框架](#)
- [构建时即设定 autorelease](#)

- [autorelease](#) 优先 [retain](#) 其次
- [init](#) 和 [dealloc](#) 内避免使用访问器
- [按声明顺序销毁实例变量](#)
- [setter](#) 应复制 [NSStrings](#)
- [避免抛异常](#)
- [nil 检查](#)
- [BOOL](#) 若干陷阱
- [属性\(Property\)](#)
 - [命名](#)
 - [位置](#)
 - [字符串应使用 copy 属性\(Attribute\)](#)
 - [原子性](#)
 - [点引用](#)
- [没有实例变量的接口](#)
- [自动 synthesize 实例变量](#)
- [Cocoa 模式](#)
 - [委托模式](#)
 - [模型/视图/控制器\(MVC\)](#)

Google Objective-C Style Guide 中文版

版本: 2.36

原作者: Mike Pinkerton
Greg Miller
Dave MacLachlan

翻译: [ewangke](#)
[Yang.Y](#)

项目主页:

- [Google Style Guide](#)
- [Google 开源项目风格指南 - 中文版](#)

译者的话

ewanke

一直想翻译这个 [style guide](#) [<http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml>] ,终于在周末花了7个小时的时间用vim敲出了HTML。很多术语的翻译很难,平时看的中文技术类书籍有限,对很多术语的中文译法不是很清楚,难免有不恰当之处,请读者指出并帮我改进:王轲 "ewangke at gmail.com" 2011.03.27

Yang.Y

对 Objective-C 的了解有限,凭着感觉和 C/C++ 方面的理解:

- 把指南更新到 2.36 版本
- 调整了一些术语和句子

背景介绍

Objective-C 是 C 语言的扩展,增加了动态类型和面对对象的特性。它被设计成具有易读易用的,支持复杂的面向对象设计的编程语言。它是 Mac OS X 以及 iPhone 的主要开发语言。

Cocoa 是 Mac OS X 上主要的应用程序框架之一。它由一组 Objective-C 类组成,为快速开发出功能齐全的 Mac OS X 应用程序提供支持。

苹果公司已经有一份非常全面的 Objective-C 编码指南。Google 为 C++ 也写了一份类似的编码指南。而这份 Objective-C 指南则是苹果和 Google 常规建议的最佳结合。因此,在阅读本指南之前,请确定你已经阅读过:

- [Apple's Cocoa Coding Guidelines](http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/index.html) [http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/index.html]
- [Google's Open Source C++ Style Guide](http://codinn.com/projects/google-cpp-styleguide/) [http://codinn.com/projects/google-cpp-styleguide/]

Note

所有在 Google 的 C++ 风格指南中所禁止的事情,如未明确说明,也同样不能在Objective-C++ 中使用。

本文档的目的在于为所有的 Mac OS X 的代码提供编码指南及实践。许多准则是在实际的项目和小组中经过长期的演化、验证的。Google 开发的开源项目遵从本指南的要求。

Google 已经发布了遵守本指南开源代码,它们属于 [Google Toolbox for Mac project](http://code.google.com/p/google-toolbox-for-mac/) [http://code.google.com/p/google-toolbox-for-mac/] 项目(本文以缩写 GTM 指代)。GTM 代码库中的代码通常为了可以在不同项目中复用。

注意,本指南不是 Objective-C 教程。我们假定读者对 Objective-C 非常熟悉。如果你刚刚接触 Objective-C 或者需要温习,请阅读 [The Objective-C Programming Language](http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html) [http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html] 。

例子

都说一个例子顶上一千句话,我们就从一个例子开始,来感受一下编码的风格、留白以及命名等等。

一个头文件的例子,展示了在 @interface 声明中如何进行正确的注释以及留白。

```
// Foo.h
// AwesomeProject
//
// Created by Greg Miller on 6/13/08.
// Copyright 2008 Google, Inc. All rights reserved.
//

#import <Foundation/Foundation.h>

// A sample class demonstrating good Objective-C style. All interfaces,
// categories, and protocols (read: all top-level declarations in a header)
// MUST be commented. Comments must also be adjacent to the object they're
// documenting.
//
// (no blank line between this comment and the interface)
@interface Foo : NSObject {
    @private
    NSString *bar_;
    NSString *bam_;
}

// Returns an autoreleased instance of Foo. See -initWithBar: for details
// about |bar|.
+ (id)fooWithBar:(NSString *)bar;
```

```

// Designated initializer. |bar| is a thing that represents a thing that
// does a thing.
- (id)initWithBar:(NSString *)bar;

// Gets and sets |bar_|.
- (NSString *)bar;
- (void)setBar:(NSString *)bar;

// Does some work with |blah| and returns YES if the work was completed
// successfully, and NO otherwise.
- (BOOL)doWorkWithBlah:(NSString *)blah;

@end

```

一个源文件的例子,展示了 @implementation 部分如何进行正确的注释、留白。同时也包括了基于引用实现的一些重要方法,如 getters、setters、init 以及 dealloc。

```

//
//  Foo.m
//  AwesomeProject
//
//  Created by Greg Miller on 6/13/08.
//  Copyright 2008 Google, Inc. All rights reserved.
//

#import "Foo.h"

@implementation Foo

+ (id)fooWithBar:(NSString *)bar {
    return [[[self alloc] initWithBar:bar] autorelease];
}

// Must always override super's designated initializer.
- (id)init {
    return [self initWithBar:nil];
}

- (id)initWithBar:(NSString *)bar {
    if ((self = [super init])) {
        bar_ = [bar copy];
        bam_ = [[NSString alloc] initWithFormat:@"hi %d", 3];
    }
    return self;
}

- (void)dealloc {
    [bar_ release];
    [bam_ release];
    [super dealloc];
}

- (NSString *)bar {
    return bar_;
}

- (void)setBar:(NSString *)bar {
    [bar_ autorelease];
}

```

```
    bar_ = [bar copy];  
}  
  
- (BOOL)doWorkWithBlah:(NSString *)blah {  
    // ...  
    return NO;  
}  
  
@end
```

不要求在 @interface、@implementation 和 @end 前后空行。如果你在 @interface 声明了**实例变量**，则须在关括号 } 之后空一行。

除非接口和**实现**非常短，比如少量的私有方法或**桥接类**，空行方有助于可读性。

留白和格式

空格 vs. 制表符

Tip

只使用空格,且一次**缩进**两个空格。

我们使用空格**缩进**。不要在代码中使用制表符。你应该将编辑器设置成自动将制表符替换成空格。

行宽

尽量让你的代码保持在 80 列之内。

我们深知 Objective-C 是一门繁冗的语言,在某些情况下略超 80 列可能有助于提高可读性,但这也只能是特例而已,不能成为开脱。

如果阅读代码的人认为把某行行宽保持在 80 列仍然有不失可读性,你应该按他们说的去做。

我们意识到这条规则是有争议的,但很多已经存在的代码坚持了本规则,我们觉得保证一致性更重要。

通过设置 *Xcode > Preferences > Text Editing > Show page guide*,来使越界更容易被发现。

方法声明和定义

Tip

- / + 和返回类型之间须使用一个空格,参数列表中只有参数之间可以有空格。

方法应该像这样:

```
- (void)doSomethingWithString:(NSString *)theString {  
    ...  
}
```

星号前的空格是可选的。当写新的代码时,要与先前代码保持一致。

如果一行有非常多的参数,更好的方式是将每个参数单独拆成一行。如果使用多行,将每个参数前的冒号对齐。

```
- (void)doSomethingWith:(GTMFoo *)theFoo
```

```

        rect:(CGRect)theRect
        interval:(float)theInterval {
    ...
}

```

当第一个关键字比其它的短时,保证下一行至少有 4 个空格的缩进。这样可以使关键字垂直对齐,而不是使用冒号对齐:

```

- (void)short:(GTMFoo *)theFoo
  longKeyword:(CGRect)theRect
  evenLongerKeyword:(float)theInterval {
    ...
}

```

方法调用

Tip

方法调用应尽量保持与方法声明的格式一致。当格式的风格有多种选择时,新的代码要与已有代码保持一致。

调用时所有参数应该在同一行:

```

[myObject doFooWith:arg1 name:arg2 error:arg3];

```

或者每行一个参数,以冒号对齐:

```

[myObject doFooWith:arg1
           name:arg2
           error:arg3];

```

不要使用下面的缩进风格:

```

[myObject doFooWith:arg1 name:arg2 // some lines with >1 arg
           error:arg3];

[myObject doFooWith:arg1
           name:arg2 error:arg3];

[myObject doFooWith:arg1
           name:arg2 // aligning keywords instead of colons
           error:arg3];

```

方法定义与方法声明一样,当关键字的长度不足以以冒号对齐时,下一行都要以四个空格进行缩进。

```

[myObj short:arg1
  longKeyword:arg2
  evenLongerKeyword:arg3];

```

@public 和 **@private**

Tip

@public 和 @private 访问修饰符应该以一个空格缩进。

与 C++ 中的 public, private 以及 protected 非常相似。

```
@interface MyClass : NSObject {
    @public
    ...
    @private
    ...
}
@end
```

异常

Tip

每个 @ 标签应该有独立的一行,在 @ 与 {} 之间需要有一个空格, @catch 与被捕捉到的异常对象的声明之间也要有一个空格。

如果你决定使用 Objective-C 的异常,那么就按下面的格式。不过你最好先看看 [避免抛出异常](#) 了解下为什么不要使用异常。

```
@try {
    foo();
}
@catch (NSException *ex) {
    bar(ex);
}
@finally {
    baz();
}
```

协议名

Tip

类型标识符和尖括号内的协议名之间,不能有任何空格。

这条规则适用于类声明、实例变量以及方法声明。例如：

```
@interface MyProtocolClass : NSObject<NSWindowDelegate> {
    @private
    id<MyFancyDelegate> delegate_;
}
- (void)setDelegate:(id<MyFancyDelegate>)aDelegate;
```


块(闭包)

Tip

块(block)适合用在 target/selector 模式下创建回调方法时,因为它使代码更易读。块中的代码应该缩进 4 个空格。

取决于块的长度,下列都是合理的风格准则:

- 如果一行可以写完块,则没必要换行。
- 如果不得不换行,关括号应与块声明的第一个字符对齐。
- 块内的代码须按 4 空格缩进。
- 如果块太长,比如超过 20 行,建议把它定义成一个局部变量,然后再使用该变量。
- 如果块不带参数,^{ 之间无须空格。如果带有参数,^(之间无须空格,但) { 之间须有一个空格。
- 块内允许按两个空格缩进,但前提是和项目的其它代码保持一致的缩进风格。

```
// The entire block fits on one line.
[operation setCompletionBlock:^( [self onOperationDone]; ]];

// The block can be put on a new line, indented four spaces, with the
// closing brace aligned with the first character of the line on which
// block was declared.
[operation setCompletionBlock:^(
    [self.delegate newDataAvailable];
)];

// Using a block with a C API follows the same alignment and spacing
// rules as with Objective-C.
dispatch_async(fileIOQueue_, ^{
    NSString* path = [self sessionFilePath];
    if (path) {
        // ...
    }
});

// An example where the parameter wraps and the block declaration fits
// on the same line. Note the spacing of |(SessionWindow *window) {|
// compared to |^{| above.
[[SessionService sharedService]
    loadWindowWithCompletionBlock:^(SessionWindow *window) {
        if (window) {
            [self windowDidLoad:window];
        } else {
            [self errorLoadingWindow];
        }
    }];

// An example where the parameter wraps and the block declaration does
// not fit on the same line as the name.
[[SessionService sharedService]
```

```
loadWindowWithCompletionBlock:
    ^(SessionWindow *window) {
        if (window) {
            [self windowDidLoad:window];
        } else {
            [self errorLoadingWindow];
        }
    }];

// Large blocks can be declared out-of-line.
void (^largeBlock)(void) = ^{
    // ...
};

[operationQueue_ addOperationWithBlock:largeBlock];
```

命名

对于易维护的代码而言,命名规则非常重要。Objective-C 的方法名往往十分长,但代码块读起来就像散文一样,不需要太多的代码注释。

当编写纯粹的 Objective-C 代码时,我们基本遵守标准的 [Objective-C naming rules](#) [<http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html>],这些命名规则可能与 C++ 风格指南中的大相径庭。例如,Google 的 C++ 风格指南中推荐使用下划线分隔的单词作为变量名,而(苹果的)风格指南则使用驼峰命名法,这在 Objective-C 社区中非常普遍。

任何的类、类别、方法以及变量的名字中都使用全大写的 [首字母缩写](#) [<http://en.wikipedia.org/wiki/Initialism>]。这遵守了苹果的标准命名方式,如 URL、TIFF 以及 EXIF。

当编写 Objective-C++ 代码时,事情就不这么简单了。许多项目需要实现跨平台的 C++ API,并混合一些 Objective-C、Cocoa 代码,或者直接以 C++ 为后端,前端用本地 Cocoa 代码。这就导致了两种命名方式直接不统一。

我们的解决方案是:编码风格取决于方法/函数以哪种语言实现。如果在一个 @implementation 语句中,就使用 Objective-C 的风格。如果实现一个 C++ 的类,就使用 C++ 的风格。这样避免了一个函数里面实例变量和局部变量命名规则混乱,严重影响可读性。

文件名

Tip

文件名须反映出其实现了什么类 – 包括大小写。遵循你所参与项目的约定。

文件的扩展名应该如下:

| | |
|-----|------------------------|
| .h | C/C++/Objective-C 的头文件 |
| .m | Objective-C 实现文件 |
| .mm | Objective-C++ 的实现文件 |
| .cc | 纯 C++ 的实现文件 |
| .c | 纯 C 的实现文件 |

类别的文件名应该包含被扩展的类名,如:GTMNSString+Utils.h 或 ``GTMNSTextView+Autocomplete.h``。

Objective-C++

Tip

源代码文件内,Objective-C++ 代码遵循你正在实现的函数/方法的风格。

为了最小化 Cocoa/Objective-C 与 C++ 之间命名风格的冲突,根据待实现的函数/方法选择编码风格。实现 @implementation 语句块时,使用 Objective-C 的命名规则;如果实现一个 C++ 的类,就使用 C++ 命名规则。

```
// file: cross_platform_header.h

class CrossPlatformAPI {
public:
    ...
    int DoSomethingPlatformSpecific(); // impl on each platform
private:
    int an_instance_var_;
};

// file: mac_implementation.mm
#include "cross_platform_header.h"

// A typical Objective-C class, using Objective-C naming.
@interface MyDelegate : NSObject {
    @private
    int instanceVar_;
    CrossPlatformAPI* backEndObject_;
}
- (void)respondToSomething:(id)something;
@end

@implementation MyDelegate
- (void)respondToSomething:(id)something {
    // bridge from Cocoa through our C++ backend
    instanceVar_ = backEndObject->DoSomethingPlatformSpecific();
    NSString* tempString = [NSString stringWithInt:instanceVar_];
    NSLog(@"%@", tempString);
}
@end

// The platform-specific implementation of the C++ class, using
// C++ naming.
int CrossPlatformAPI::DoSomethingPlatformSpecific() {
    NSString* temp_string = [NSString stringWithInt:an_instance_var_];
    NSLog(@"%@", temp_string);
    return [temp_string intValue];
}
```

类名

Tip

类名(以及类别、协议名)应首字母大写,并以驼峰格式分割单词。

应用层 的代码,应该尽量避免不必要的前缀。为每个类都添加相同的前缀无助于可读性。当编写的代码期望在不同应用程序间复用时,应使用前缀(如:GTMSendMessage)。

类别名

Tip

类别名应该有两三个字母的前缀以表示类别是项目的一部分或者该类别是通用的。类别名应该包含它所扩展的类的名字。

比如我们要基于 NSString 创建一个用于解析的类别,我们将把类别放在一个名为 GTMNSString+Parsing.h 的文件中。类别本身命名为 GTMStringParsingAdditions (是的,我们知道类别名和文件名不一样,但是这个文件中可能存在多个不同的与解析有关类别)。类别中的方法应该以 gtm_myCategoryMethodOnAString: 为前缀以避免命名冲突,因为 Objective-C 只有一个名字空间。如果代码不会分享出去,也不会运行在不同的地址空间中,方法名字就不那么重要了。

类名与包含类别名的括号之间,应该以一个空格分隔。

Objective-C 方法名

Tip

方法名应该以小写字母开头,并混合驼峰格式。每个具名参数也应该以小写字母开头。

方法名应尽量读起来就像句子,这表示你应该选择与方法名连在一起读起来通顺的参数名。(例如,convertPoint:fromRect: 或 replaceCharactersInRange:withString:)。详情参见 [Apple's Guide to Naming Methods](http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingMethods.html)

[<http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingMethods.html>]。

访问器方法应该与他们要获取的成员变量的名字一样,但不应该以get作为前缀。例如:

```
- (id)getDelegate;    // AVOID  
- (id)delegate;      // GOOD
```

这仅限于 Objective-C 的方法名。C++ 的方法与函数的命名规则应该遵从 C++ 风格指南中的规则。

变量名

Tip

变量名应该以小写字母开头,并使用驼峰格式。类的成员变量应该以下划线作为后缀。例如:myLocalVariable、myInstanceVariable_。如果不能使用 Objective-C 2.0 的 @property,使用 KVO/KVC 绑定的成员变量可以以一个下划线作为前缀。

普通变量名

对于静态的属性(int 或指针),不要使用匈牙利命名法。尽量为变量起一个描述性的名字。不要担心浪费列宽,因为让新的代码阅读者立即理解你的代码更重要。例如:

- 错误的命名:

```
int w;  
int nerr;  
int nCompConns;  
tix = [[NSMutableArray alloc] init];  
obj = [someObject object];  
p = [network port];
```

- 正确的命名:

```
int numErrors;  
int numCompletedConnections;  
tickets = [[NSMutableArray alloc] init];  
userInfo = [someObject object];  
port = [network port];
```

实例变量

实例变量应该混合大小写,并以下划线作为后缀,如 usernameTextField_。然而,如果不能使用 Objective-C 2.0(操作系统版本的限制),并且使用了 KVO/KVC 绑定成员变量时,我们允许例外(译者注:KVO=Key Value Observing, KVC=Key Value Coding)。这种情况下,可以以一个下划线作为成员变量名字的前缀,这是苹果所接受的键/值命名惯例。如果可以使用 Objective-C 2.0,@property 以及 @synthesize 提供了遵从这一命名规则的解决方案。

常量

常量名(如宏定义、枚举、静态局部变量等)应该以小写字母 k 开头,使用驼峰格式分隔单词,如: kInvalidHandle, kWritePerm。

注释

虽然写起来很痛苦,但注释是保证代码可读性的关键。下面的规则给出了你应该什么时候、在哪进行注释。记住:尽管注释很重要,但最好的代码应该自成文档。与其给类型及变量起一个晦涩难懂的名字,再为它写注释,不如直接起一个有意义的名字。

当你写注释的时候,记得你是在给你的听众写,即下一个需要阅读你所写代码的贡献者。大方一点,下一个读代码的人可能就是你!

记住所有 C++ 风格指南里的规则在这里也同样适用,不同的之处后续会逐步指出。

文件注释

Tip

每个文件的开头以文件内容的简要描述起始,紧接着是作者,最后是版权声明和/或许可证样板。

版权信息及作者

每个文件应该按顺序包括如下项:

- 文件内容的简要描述
- 代码作者
- 版权信息声明(如:Copyright 2008 Google Inc.)
- 必要的话,加上许可证样板。为项目选择一个合适的授权样板(例如,Apache 2.0, BSD, LGPL, GPL)。

如果你对其他人的原始代码作出重大的修改,请把你自己的名字添加到作者里面。当另外一个代码贡献者对文件有问题时,他需要知道怎么联系你,这十分有用。

声明部分的注释

Tip

每个接口、类别以及协议应辅以注释,以描述它的目的及与整个项目的关系。

```
// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
    ...
}
@end
```


如果你已经在文件头部详细描述了接口,可以直接说明“完整的描述请参见文件头部”,但是一定要有这部分注释。

另外,公共接口的每个方法,都应该有注释来解释它的作用、参数、返回值以及其它影响。

为类的线程安全性作注释,如果有的话。如果类的实例可以被多个线程访问,记得注释多线程条件下的使用规则。

实现部分的注释

Tip

使用 `|` 来引用注释中的变量名及符号名而不是使用引号。

这会避免二义性,尤其是当符号是一个常用词汇,这使用语句读起来很糟糕。例如,对于符号 `count` :

```
// Sometimes we need |count| to be less than zero.
```

或者当引用已经包含引号的符号:

```
// Remember to call |StringWithoutSpaces("foo bar baz")|
```

对象所有权

Tip

当与 Objective-C 最常规的作法不同时,尽量使指针的所有权模型尽量明确。

继承自 `NSObject` 的对象的实例变量指针,通常被假定是强引用关系(`retained`),某些情况下也可以注释为弱引用(`weak`)或使用 `__weak` 生命周期限定符。同样,声明的属性如果没有被类 `retained`,必须指定是弱引用或赋予 `@property` 属性。然而,Mac 软件中标记上 `IBOutlet`s 的实例变量,被认为是不会被类 `retained` 的。

当实例变量指向 `CoreFoundation`、C++ 或者其它非 Objective-C 对象时,不论指针是否会被 `retained`,都需要使用 `__strong` 和 `__weak` 类型修饰符明确指明。`CoreFoundation` 和其它非 Objective-C 对象指针需要显式的内存管理,即便使用了自动引用计数或垃圾回收机制。当不允许使用 `__weak` 类型修饰符(比如,使用 clang 编译时的 C++ 成员变量),应使用注释替代说明。

注意:Objective-C 对象中的 C++ 对象的自动封装,缺省是不允许的,参见 [这里](http://chanson.livejournal.com/154253.html) [http://chanson.livejournal.com/154253.html] 的说明。

强引用及弱引用声明的例子:

```
@interface MyDelegate : NSObject {
```

```

@private
IBOutlet NSButton *okButton_; // normal NSControl; implicitly weak on Mac only

AnObjcObject* doohickey_; // my doohickey
__weak MyObjcParent *parent_; // so we can send msgs back (owns me)

// non-NSObject pointers...
__strong CWackyCPPClass *wacky_; // some cross-platform object
__strong CFDictionaryRef *dict_;
}
@property(strong, nonatomic) NSString *doohickey;
@property(weak, nonatomic) NSString *parent;
@end

```

(译注:强引用 - 对象被类 retained。弱引用 - 对象没有被类 retained,如委托)

Cocoa 和 Objective-C 特性

成员变量应该是 @private

Tip

成员变量应该声明为 @private

```
@interface MyClass : NSObject {  
    @private  
    id myInstanceVariable_;  
}  
  
// public accessors, setter takes ownership  
- (id)myInstanceVariable;  
- (void)setMyInstanceVariable:(id)theVar;  
@end
```

明确指定构造函数

Tip

注释并且明确指定你的类的构造函数。

对于需要继承你的类的人来说,明确指定构造函数十分重要。这样他们就可以只重写一个构造函数(可能是几个)来保证他们的子类的构造函数会被调用。这也有助于将来别人调试你的类时,理解初始化代码的工作流程。

重载指定构造函数

Tip

当你写子类的时候,如果需要 init... 方法,记得重载父类的指定构造函数。

如果你没有重载父类的指定构造函数,你的构造函数有时可能不会被调用,这会导致非常隐秘而且难以解决的 bug。

重载 NSObject 的方法

Tip

如果重载了 `NSObject` 类的方法,强烈建议把它们放在 `@implementation` 内的起始处,这也是常见的操作方法。

通常适用(但不局限)于 `init...`,`copyWithZone:`,以及 `dealloc` 方法。所有 `init...` 方法应该放在一起,`copyWithZone:` 紧随其后,最后才是 `dealloc` 方法。

初始化

Tip

不要在 `init` 方法中,将成员变量初始化为 `0` 或者 `nil`;毫无必要。

刚分配的对象,默认值都是 `0`,除了 `isa` 指针(译者注:`NSObject` 的 `isa` 指针,用于标识对象的类型)。所以不要在初始化器里面写一堆将成员初始化为 `0` 或者 `nil` 的代码。

避免 `+new`

Tip

不要调用 `NSObject` 类方法 `new`,也不要子类中重载它。使用 `alloc` 和 `init` 方法创建并初始化对象。

现代的 Objective-C 代码通过调用 `alloc` 和 `init` 方法来创建并 `retain` 一个对象。由于类方法 `new` 很少使用,这使得有关内存分配的代码审查更困难。

保持公共 API 简单

Tip

保持类简单;避免“厨房水槽(kitchen-sink)”式的 API。如果一个函数压根没必要公开,就不要这么做。用私有类别保证公共头文件整洁。

与 C++ 不同,Objective-C 没有方法来区分公共的方法和私有的方法 – 所有的方法都是公共的(译者注:这取决于 Objective-C 运行时的方法调用的消息机制)。因此,除非客户端的代码期望使用某个方法,不要把这个方法放进公共 API 中。尽可能的避免了你你不希望被调用的方法却被调用到。这包括重载父类的方法。对于内部实现所需要的方法,在实现的文件中定义一个类别,而不是把它们放进公有的头文件中。

```
// GTMFoo.m
#import "GTMFoo.h"

@interface GTMFoo (PrivateDelegateHandling)
```

```

- (NSString *)doSomethingWithDelegate; // Declare private method
@end

@implementation GTMFoo(PrivateDelegateHandling)
...
- (NSString *)doSomethingWithDelegate {
    // Implement this method
}
...
@end

```

Objective-C 2.0 以前,如果你在私有的 @interface 中声明了某个方法,但在 @implementation 中忘记定义这个方法,编译器不会抱怨(这是因为你没有在其它的类别中实现这个私有的方法)。解决方案是将方法放进指定类别的 @implementation 中。

如果你在使用 Objective-C 2.0,相反你应该使用 [类扩展](#)

[http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_4_section_5.html] 来声明你的私有类别,例如:

```

@interface GMFoo () { ... }

```

这么做确保如果声明的方法没有在 @implementation 中实现,会触发一个编译器告警。

再次说明,“私有的”方法其实不是私有的。你有时可能不小心重载了父类的私有方法,因而制造出很难查找的 Bug。通常,私有的方法应该有一个相当特殊的名字以防止子类无意地重载它们。

Objective-C 的类别可以用来将一个大的 @implementation 拆分成更容易理解的小块,同时,类别可以为最适合的类添加新的、特定应用程序的功能。例如,当添加一个“middle truncation”方法时,创建一个 NSString 的新类别并把方法放在里面,要比创建任意的一个新类把方法放进里面好得多。

#import and #include

Tip

#import Objective-C/Objective-C++ 头文件,#include C/C++ 头文件。

基于你所包括的头文件的编程语言,选择使用 #import 或是 #include:

- 当包含一个使用 Objective-C、Objective-C++ 的头文件时,使用 #import。
- 当包含一个使用标准 C、C++ 头文件时,使用 #include。头文件应该使用 [#define 保护](#)
[http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml?showone=The__define_Guard#The__define_Guard].

一些 Objective-C 的头文件缺少 #define 保护,需要使用 #import 的方式包含。由于 Objective-C 的头文件只会被 Objective-C 的源文件及头文件包含,广泛地使用 #import 是可以的。

文件中没有 Objective-C 代码的标准 C、C++ 头文件,很可能会被普通的 C、C++ 包含。由于标准 C、C++ 里面没有 #import 的用法,这些文件将被 #include。在 Objective-C 源文件中使用 #include 包含这些头文件,意味着这些头文件永远会在相同的语义下包含。

这条规则帮助跨平台的项目避免低级错误。某个 Mac 开发者写了一个新的 C 或 C++ 头文件,如果忘记使用 `#define` 保护,在 Mac 下使用 `#import` 这个头文件不回引起问题,但是在其它平台下使用 `#include` 将可能编译失败。在所有的平台上统一使用 `#include`,意味着构造更可能全都成功或者失败,防止这些文件只能在某些平台下能够工作。

```
#import <Cocoa/Cocoa.h>
#include <CoreFoundation/CoreFoundation.h>
#import "GTMFoo.h"
#include "base/basictypes.h"
```

使用根框架

Tip

`#import` 根框架而不是单独的零散文件

当你试图从框架(如 Cocoa 或者 Foundation)中包含若干零散的系统头文件时,实际上包含顶层根框架的话,编译器要做的工作更少。根框架通常已经经过预编译,加载更快。另外记得使用 `#import` 而不是 `#include` 来包含 Objective-C 的框架。

```
#import <Foundation/Foundation.h>           // good

#import <Foundation/NSArray.h>               // avoid
#import <Foundation/NSString.h>
...
```

构建时即设定 **autorelease**

Tip

当创建临时对象时,在同一行使用 `autorelease`,而不是在同一个方法的后面语句中使用一个单独的 `release`。

尽管运行效率会差一点,但避免了意外删除 `release` 或者插入 `return` 语句而导致内存泄露的可能。例如:

```
// AVOID (unless you have a compelling performance reason)
MyController* controller = [[MyController alloc] init];
// ... code here that might return ...
[controller release];

// BETTER
MyController* controller = [[[MyController alloc] init] autorelease];
```

autorelease 优先 **retain** 其次

Tip

给对象赋值时遵守 autorelease``之后 ``retain 的模式。

当给一个变量赋值新的对象时,必须先释放掉旧的对象以避免内存泄露。有很多“正确的”方法可以处理这种情况。我们则选择“autorelease 之后 retain”的方法,因为事实证明它不容易出错。注意大的循环会填满 autorelease 池,并且可能效率上会差一点,但权衡之下我们认为是可以接受的。

```
- (void)setFoo:(GMFoo *)aFoo {
    [foo_ autorelease]; // Won't dealloc if |foo_| == |aFoo|
    foo_ = [aFoo retain];
}
```

init 和 dealloc 内避免使用访问器

Tip

在 init 和 dealloc 方法执行的过程中,子类可能会处在一个不一致的状态,所以这些方法中的代码应避免调用访问器。

子类尚未初始化,或在 init 和 dealloc 方法执行时已经被销毁,会使访问器方法很可能不可靠。实际上,应在这些方法中直接对 ivals 进行赋值或释放操作。

正确:

```
- (id)init {
    self = [super init];
    if (self) {
        bar_ = [[NSMutableString alloc] init]; // good
    }
    return self;
}

- (void)dealloc {
    [bar_ release]; // good
    [super dealloc];
}
```

错误:

```
- (id)init {
    self = [super init];
    if (self) {
        self.bar = [NSMutableString string]; // avoid
    }
    return self;
}

- (void)dealloc {
```



```
self.bar = nil; // avoid  
[super dealloc];  
}
```

按声明顺序销毁实例变量

Tip

dealloc 中实例变量被释放的顺序应该与它们在 @interface 中声明的顺序一致,这有助于代码审查。

代码审查者在评审新的或者修改过的 dealloc 实现时,需要保证每个 retained 的实例变量都得到了释放。

为了简化 dealloc 的审查,retained 实例变量被释放的顺序应该与他们在 @interface 中声明的顺序一致。如果 dealloc 调用了其它方法释放成员变量,添加注释解释这些方法释放了哪些实例变量。

setter 应复制 NSStrings

Tip

接受 NSString 作为参数的 setter,应该总是 copy 传入的字符串。

永远不要仅仅 retain 一个字符串。因为调用者很可能在你不知情的情况下修改了字符串。不要假定别人不会修改,你接受的对象是一个 NSString 对象而不是 NSMutableString 对象。

```
- (void)setFoo:(NSString *)aFoo {  
    [foo_ autorelease];  
    foo_ = [aFoo copy];  
}
```

避免抛异常

Tip

不要 @throw Objective-C 异常,同时也要时刻准备捕获从第三方或 OS 代码中抛出的异常。

我们的确允许 -fobjc-exceptions 编译开关(主要因为我们要用到 @synchronized),但我们不使用 @throw。为了合理使用第三方的代码,@try、@catch 和 @finally 是允许的。如果你确实使用了异常,请明确注释你期望什么方法抛出异常。

不要使用 NS_DURING、NS_HANDLER、NS_ENDHANDLER、NS_VALUEReturn 和 NS_VOIDRETURN 宏,除非你写的代码需要在 Mac OS X 10.2 或之前的操作系统中运行。

注意:如果抛出 Objective-C 异常,Objective-C++ 代码中基于栈的对象不会被销毁。比如:

```
class exceptiontest {
public:
    exceptiontest() { NSLog(@"Created"); }
    ~exceptiontest() { NSLog(@"Destroyed"); }
};

void foo() {
    exceptiontest a;
    NSException *exception = [NSException exceptionWithName:@"foo"
                                                             reason:@"bar"
                                                             userInfo:nil];

    @throw exception;
}

int main(int argc, char *argv[]) {
    GMAutoreleasePool pool;
    @try {
        foo();
    }
    @catch(NSException *ex) {
        NSLog(@"exception raised");
    }
    return 0;
}
```

会输出:

注意:这里析构函数从未被调用。这主要会影响基于栈的 smartptr,比如 shared_ptr、linked_ptr,以及所有你可能用到的 STL 对象。因此我们不得不痛苦的说,如果必须在 Objective-C++ 中使用异常,就只用 C++ 的异常机制。永远不应该重新抛出 Objective-C 异常,也不应该在 @try、@catch 或 @finally 语句块中使用基于栈的 C++ 对象。

nil 检查

Tip

nil 检查只用在逻辑流程中。

使用 nil 的检查来检查应用程序的逻辑流程,而不是避免崩溃。Objective-C 运行时会处理向 nil 对象发送消息的情况。如果方法没有返回值,就没关系。如果有返回值,可能由于运行时架构、返回值类型以及 OS X 版本的不同而不同,参见 [Apple's documentation](http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_2_section_3.html)

[http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_2_section_3.html]。

注意,这和 C/C++ 中检查指针是否为 ``NULL`` 很不一样,C/C++ 运行时不做任何检查,从而导致应用程序崩溃。因此你仍然需要保证你不会对一个 C/C++ 的空指针解引用。

BOOL 若干陷阱

Tip

将普通整形转换成 BOOL 时要小心。不要直接将 BOOL 值与 YES 进行比较。

Objective-C 中把 BOOL 定义成无符号字符型,这意味着 BOOL 类型的值远不止 YES (1)或 NO (0)。不要直接把整形转换成 BOOL。常见的错误包括将数组的大小、指针值及位运算的结果直接转换成 BOOL,取决于整型结果的最后一个字节,很可能会产生一个 NO 值。当转换整形至 BOOL 时,使用三目操作符来返回 YES 或者 NO。(译者注:读者可以试一下任意的 256 的整数的转换结果,如 256, 512 ...)

你可以安全在 BOOL、_Bool 以及 bool 之间转换(参见 C++ Std 4.7.4, 4.12 以及 C99 Std 6.3.1.2)。你不能安全在 BOOL 以及 Boolean 之间转换,因此请把 Boolean 当作一个普通整形,就像之前讨论的那样。但 Objective-C 的方法标识符中,只使用 BOOL。

对 BOOL 使用逻辑运算符(&&, || 和 !)是合法的,返回值也可以安全地转换成 BOOL,不需要使用三目操作符。

错误的用法:

```
- (BOOL)isBold {
    return [self fontTraits] & NSFontBoldTrait;
}
- (BOOL)isValid {
    return [self stringValue];
}
```

正确的用法:

```
• (BOOL)isBold { return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
} - (BOOL)isValid {
    return [self stringValue] != nil;
} - (BOOL)isEnabled {
    return [self isValid] && [self isBold];
}
```

同样,不要直接比较 YES/NO 和 BOOL 变量。不仅仅因为影响可读性,更重要的是结果可能与你想的不同。

错误的用法:

```
BOOL great = [foo isGreat];
if (great == YES)
    // ...be great!
```

正确的用法：

```
BOOL great = [foo isGreat];
if (great)
    // ...be great!
```

属性(Property)

Tip

属性(Property)通常允许使用,但需要清楚的了解:属性(Property)是 Objective-C 2.0 的特性,会限制你的代码只能跑在 iPhone 和 Mac OS X 10.5 (Leopard) 及更高版本上。点引用只允许访问声明过的 @property。

命名

属性所关联的实例变量的命名必须遵守以下划线作为后缀的规则。属性的名字应该与成员变量去掉下划线后缀的名字一模一样。

使用 @synthesize 指示符来正确地重命名属性。

```
@interface MyClass : NSObject {
    @private
    NSString *name_;
}
@property(copy, nonatomic) NSString *name;
@end

@implementation MyClass
@synthesize name = name_;
@end
```

位置

属性的声明必须紧靠着类接口中的实例变量语句块。属性的定义必须在 @implementation 的类定义的最上方。他们的缩进与包含他们的 @interface 以及 @implementation 语句一样。

```
@interface MyClass : NSObject {
    @private
    NSString *name_;
}
@property(copy, nonatomic) NSString *name;
@end

@implementation MyClass
@synthesize name = name_;
- (id)init {
    ...
}
@end
```

字符串应使用 **copy** 属性(**Attribute**)

应总是用 copy 属性(attribute)声明 NSString 属性(property)。

从逻辑上,确保遵守 NSString 的 setter 必须使用 copy 而不是 retain 的原则。

原子性

一定要注意属性(property)的开销。缺省情况下,所有 synthesize 的 setter 和 getter 都是原子的。这会给每个 get 或者 set 带来一定的同步开销。将属性(property)声明为 nonatomic,除非你需要原子性。

点引用

点引用是地道的 Objective-C 2.0 风格。它被使用于简单的属性 set、get 操作,但不应该用它来调用对象的其它操作。

正确的做法:

```
NSString *oldName = myObject.name;  
myObject.name = @"Alice";
```

错误的做法:

```
NSArray *array = [[NSArray arrayWithObject:@"hello"] retain];  
  
NSUInteger numberOfItems = array.count; // not a property  
array.release; // not a property
```

没有实例变量的接口

Tip

没有声明任何实例变量的接口,应省略空花括号。

正确的做法:

```
@interface MyClass : NSObject // Does a lot of stuff - (void)fooBarBam; @end
```

错误的做法:

```
@interface MyClass : NSObject { } // Does a lot of stuff - (void)fooBarBam; @end
```

自动 **synthesize** 实例变量

Tip

只运行在 iOS 下的代码,优先考虑使用自动 synthesize 实例变量。

synthesize 实例变量时,使用 @synthesize var = var_; 防止原本想调用 self.var = blah; 却不慎写成了 var = blah;。

不要synthesize CType的属性 CType应该永远使用@dynamic实现指示符。尽管CType不能使用retain属性特性,开发者必须自己处理retain和release。很少有情况你需要仅仅对它进行赋值,因此最好显示地实现getter和setter,并作出注释说明。列出所有的实现指示符 尽管@dynamic是默认的,显示列出它以及其它的实现指示符会提高可读性,代码阅读者可以一眼就知道类的每个属性是如何实现的。

```
// Header file
@interface Foo : NSObject
// A guy walks into a bar.
@property(nonatomic, copy) NSString *bar;
@end

// Implementation file
@interface Foo ()
@property(nonatomic, retain) NSArray *baz;
@end

@implementation Foo
@synthesize bar = bar_;
@synthesize baz = baz_;
@end
```

Cocoa 模式

委托模式

Tip

委托对象不应该被 retain

实现委托模式的类应：

1. 拥有一个名为 `delegate_` 的实例变量来引用委托。
2. 因此,访问器方法应该命名为 `delegate` 和 `setDelegate:`。
3. `delegate_` 对象不应该被 retain。

模型/视图/控制器(MVC)

Tip

分离模型与视图。分离控制器与视图、模型。回调 API 使用 `@protocol`。

- 分离模型与视图: 不要假设模型或者数据源的表示方法。保持数据源与表示层之间的接口抽象。视图不需要了解模型的逻辑(主要的规则是问问你自己,对于数据源的一个实例,有没有可能有多种不同状态的表示方法)。
- 分离控制器与模型、视图: 不要把所有的“业务逻辑”放进跟视图有关的类中。这使代码非常难以复用。使用控制器类来处理这些代码,但保证控制器不需要了解太多表示层的逻辑。
- 使用 `@protocol` 来定义回调 API,如果不是所有的方法都必须实现,使用 `@optional` `` (特例: 使用 Objective-C 1.0 时, ``@optional 不可用,可使用类别来定义一个“非正规的协议”)。

Python 风格指南 - 内容目录

- [扉页](#)
- [背景](#)
- [Python语言规范](#)
 - [Lint](#)
 - [导入](#)
 - [包](#)
 - [异常](#)
 - [全局变量](#)
 - [嵌套/局部/内部类或函数](#)
 - [列表推导\(List Comprehensions\)](#)
 - [默认迭代器和操作符](#)
 - [生成器](#)
 - [Lambda函数](#)
 - [条件表达式](#)
 - [默认参数值](#)
 - [属性\(properties\)](#)
 - [True/False的求值](#)
 - [过时的语言特性](#)
 - [词法作用域\(Lexical Scoping\)](#)
 - [函数与方法装饰器](#)
 - [线程](#)
 - [威力过大的特性](#)
- [Python风格规范](#)
 - [分号](#)
 - [行长度](#)
 - [括号](#)
 - [缩进](#)
 - [空行](#)
 - [空格](#)
 - [Shebang](#)
 - [注释](#)
 - [类](#)
 - [字符串](#)
 - [文件和sockets](#)
 - [TODO注释](#)
 - [导入格式](#)
 - [语句](#)
 - [访问控制](#)
 - [命名](#)
 - [Main](#)
- [临别赠言](#)

扉页

版本: 2.59

原作者: Amit Patel
Antoine Picard
Eugene Jhong
Jeremy Hylton
Matt Smart
Mike Shields

翻译: [guoqiao](#) v2.19
[xuxinkun](#) v2.59

项目主页:

- [Google Style Guide](#)
- [Google 开源项目风格指南 - 中文版](#)

背景

Python 是 Google 主要的脚本语言。这本风格指南主要包含的是针对python的编程准则。

为帮助读者能够将代码准确格式化,我们提供了针对 [Vim的配置文件](#)
[https://github.com/google/styleguide/blob/gh-pages/google_python_style.vim] 。对于Emacs用户,保持默认设置即可。

Python语言规范

Lint

Tip

对你的代码运行pylint

定义:

pylint是一个在Python源代码中查找bug的工具. 对于C和C++这样的不那么动态的(译者注: 原文是less dynamic)语言, 这些bug通常由编译器来捕获. 由于Python的动态特性, 有些警告可能不对. 不过伪告警应该很少.

优点:

可以捕获容易忽视的错误, 例如输入错误, 使用未赋值的变量等.

缺点:

pylint不完美. 要利用其优势, 我们有时候需要: a) 围绕着它来写代码 b) 抑制其告警 c) 改进它, 或者d) 忽略它.

结论:

确保对你的代码运行pylint.抑制不准确的警告,以便能够将其他警告暴露出来。

你可以通过设置一个行注释来抑制告警. 例如:

```
dict = 'something awful' # Bad Idea... pylint: disable=redefined-builtin
```

pylint警告是以一个数字编号(如 C0112)和一个符号名(如 empty-docstring)来标识的. 在编写新代码或更新已有代码时对告警进行医治, 推荐使用符号名来标识.

如果警告的符号名不够见名知意,那么请对其增加一个详细解释。

采用这种抑制方式的好处是我们可以轻松查找抑制并回顾它们.

你可以使用命令 `pylint --list-msgs` 来获取pylint告警列表. 你可以使用命令 `pylint --help-msg=C6409` , 以获取关于特定消息的更多信息.

相比较于之前使用的 `pylint: disable-msg` , 本文推荐使用 `pylint: disable` .

要抑制”参数未使用”告警, 你可以用”_”作为参数标识符, 或者在参数名前加”unused_”. 遇到不能改变参数名的情况, 你可以通过在函数开头”提到”它们来消除告警. 例如:

```
def foo(a, unused_b, unused_c, d=None, e=None):
    _ = d, e
    return a
```

导入

Tip

仅对包和模块使用导入

定义:

模块间共享代码的重用机制.

优点:

命名空间管理约定十分简单. 每个标识符的源都用一种一致的方式指示. `x.Obj`表示Obj对象定义在模块x中.

缺点:

模块名仍可能冲突. 有些模块名太长, 不太方便.

结论:

使用 `import x` 来导入包和模块.

使用 `from x import y`, 其中x是包前缀, y是不带前缀的模块名.

使用 `from x import y as z`, 如果两个要导入的模块都叫做y或者y太长了.

例如, 模块 `sound.effects.echo` 可以用如下方式导入:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

导入时不要使用相对名称. 即使模块在同一个包中, 也要使用完整包名. 这能帮助你避免无意间导入一个包两次.

包

Tip

使用模块的全路径名来导入每个模块

优点:

避免模块名冲突. 查找包更容易.

缺点:

部署代码变难, 因为你必须复制包层次.

结论:

所有的新代码都应该用完整包名来导入每个模块.

应该像下面这样导入:

```
# Reference in code with complete name.  
import sound.effects.echo  
  
# Reference in code with just module name (preferred).  
from sound.effects import echo
```

异常

Tip

允许使用异常, 但必须小心

定义:

异常是一种跳出代码块的正常控制流来处理错误或者其它异常条件的方式.

优点:

正常操作代码的控制流不会和错误处理代码混在一起. 当某种条件发生时, 它也允许控制流跳过多个框架. 例如, 一步跳出N个嵌套的函数, 而不必继续执行错误的代码.

缺点:

可能会导致让人困惑的控制流. 调用库时容易错过错误情况.

结论:

异常必须遵守特定条件:

1. 像这样触发异常: `raise MyException("Error message")` 或者 `raise MyException`. 不要使用两个参数的形式(`raise MyException, "Error message"`)或者过时的字符串异常(`raise "Error message"`).
2. 模块或包应该定义自己的特定域的异常基类, 这个基类应该从内建的Exception类继承. 模块的异常基类应该叫做"Error".

```
class Error(Exception):  
    pass
```

3. 永远不要使用 `except:` 语句来捕获所有异常, 也不要捕获 `Exception` 或者 `StandardError`, 除非你打算重新触发该异常, 或者你已经在当前线程的最外层(记得还是要打印一条错误消息). 在异常这方面, Python非常宽容, `except:` 真的会捕获包括Python语法错误在内的任何错误. 使用 `except:` 很容易隐藏真正的bug.
4. 尽量减少try/except块中的代码量. try块的体积越大, 期望之外的异常就越容易被触发. 这种情况下, try/except块将隐藏真正的错误.
5. 使用finally子句来执行那些无论try块中有没有异常都应该被执行的代码. 这对于清理资源常常很有用, 例如关闭文件.
6. 当捕获异常时, 使用 `as` 而不要用逗号. 例如

```
try:
    raise Error
except Error as error:
    pass
```

全局变量

Tip

避免全局变量

定义:

定义在模块级的变量.

优点:

偶尔有用.

缺点:

导入时可能改变模块行为, 因为导入模块时会对模块级变量赋值.

结论:

避免使用全局变量, 用类变量来代替. 但也有一些例外:

1. 脚本的默认选项.
2. 模块级常量. 例如: `PI = 3.14159`. 常量应该全大写, 用下划线连接.
3. 有时候用全局变量来缓存值或者作为函数返回值很有用.
4. 如果需要, 全局变量应该仅在模块内部可用, 并通过模块级的公共函数来访问.

嵌套/局部/内部类或函数

Tip

鼓励使用嵌套/本地/内部类或函数

定义:

类可以定义在方法, 函数或者类中. 函数可以定义在方法或函数中. 封闭区间中定义的变量对嵌套函数是只读的.

优点:

允许定义仅用于有效范围的工具类和函数.

缺点:

嵌套类或局部类的实例不能序列化(pickled).

结论:

推荐使用.

列表推导(List Comprehensions)

Tip

可以在简单情况下使用

定义:

列表推导(list comprehensions)与生成器表达式(generator expression)提供了一种简洁高效的方式来创建列表和迭代器, 而不必借助map(), filter(), 或者lambda.

优点:

简单的列表推导可以比其它的列表创建方法更加清晰简单. 生成器表达式可以十分高效, 因为它们避免了创建整个列表.

缺点:

复杂的列表推导或者生成器表达式可能难以阅读.

结论:

适用于简单情况. 每个部分应该单独置于一行: 映射表达式, for语句, 过滤器表达式. 禁止多重for语句或过滤器表达式. 复杂情况下还是使用循环.

Yes:

```
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))

for x in xrange(5):
    for y in xrange(5):
        if x != y:
            for z in xrange(5):
                if y != z:
                    yield (x, y, z)

return ((x, complicated_transform(x))
        for x in long_generator_function(parameter)
        if x is not None)

squares = [x * x for x in range(10)]

eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')
```

No:

```
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]

return ((x, y, z)
        for x in xrange(5)
        for y in xrange(5)
        if x != y
        for z in xrange(5)
        if y != z)
```

默认迭代器和操作符

Tip

如果**类型**支持, 就使用默认迭代器和操作符. 比如列表, 字典及文件等.

定义:

容器**类型**, 像字典和列表, 定义了默认的迭代器和关系**测试**操作符(in和not in)

优点:

默认操作符和迭代器**简单**高效, 它们直接表达了操作, 没有**额外**的方法调用. 使用默认操作符的函数是通用的. 它可以用于支持**该**操作的任何**类型**.

缺点:

你没法通过**阅读**方法名来区分对象的**类型**(例如, has_key()意味着字典). 不过这也是优点.

结论:

如果**类型**支持, 就使用默认迭代器和操作符, 例如列表, 字典和文件. 内建**类型**也定义了迭代器方法. **优先考虑**这些方法, 而不是那些返回列表的方法. 当然, **这样遍历容器时**, 你将不能修改容器.

```
Yes:  for key in adict: ...
      if key not in adict: ...
      if obj in alist: ...
      for line in afile: ...
      for k, v in dict.iteritems(): ...
```

```
No:   for key in adict.keys(): ...
      if not adict.has_key(key): ...
      for line in afile.readlines(): ...
```

生成器

Tip

按需使用生成器.

定义:

所谓生成器函数, 就是每当它**执行**一次生成(yield)语句, 它就返回一个迭代器, **这个**迭代器生成一个**值**. 生成**值**后, 生成器函数的运行状态将被挂起, 直到下一次生成.

优点:

简化代码, 因为每次调用时, 局部**变量**和控制流的状态都会被保存. 比起一次**创建**一系列**值**的函数, 生成器使用的内存更少.

缺点:

没有.

结论:

鼓励使用. 注意在生成器函数的文档字符串中使用”Yields:”而不是”Returns:”.

(译者注: 参看 [注释](#))

Lambda函数

Tip

适用于**单行函数**

定义:

与**语句**相反, lambda在一个表达式中定义匿名函数. 常用于为 `map()` 和 `filter()` 之类的高阶函数定义回调函数或者操作符.

优点:

方便.

缺点:

比本地函数更**难阅读和调试**. 没有函数名意味着堆栈跟踪更**难理解**. 由于lambda函数通常只包含一个表达式, 因此其表达能力有限.

结论:

适用于**单行函数**. 如果代码超过60-80个字符, 最好还是定义成**常规(嵌套)函数**.

对于常见的操作符, 例如乘法操作符, 使用 `operator` 模块中的函数以代替lambda函数. 例如, 推荐使用 `operator.mul`, 而不是 `lambda x, y: x * y`.

条件表达式

Tip

适用于**单行函数**

定义:

条件表达式是对于**if语句**的一种更为简短的句法规则. 例如: `x = 1 if cond else 2`.

优点:

比**if语句**更加简短和方便.

缺点:

比**if语句****难于阅读**. 如果表达式很长, **难于定位条件**.

结论:

适用于**单行函数**. 在其他情况下, 推荐使用完整的**if语句**.

默认参数值

Tip

适用于大部分情况.

定义:

你可以在函数参数列表的最后指定变量的值, 例如, `def foo(a, b = 0):`. 如果调用`foo`时只带一个参数, 则`b`被设为0. 如果带两个参数, 则`b`的值等于第二个参数.

优点:

你经常会碰到一些使用大量默认值的函数, 但偶尔(比较少见)你想要覆盖这些默认值. 默认参数值提供了一种简单的方法来完成这件事, 你不需要为这些罕见的例外定义大量函数. 同时, Python也不支持重载方法和函数, 默认参数是一种”仿造”重载行为的简单方式.

缺点:

默认参数只在模块加载时求值一次. 如果参数是列表或字典之类的可变类型, 这可能会导致问题. 如果函数修改了对象(例如向列表追加项), 默认值就被修改了.

结论:

鼓励使用, 不过有如下注意事项:

不要在函数或方法定义中使用可变对象作为默认值.

```
Yes: def foo(a, b=None):  
    if b is None:  
        b = []
```

```
No: def foo(a, b=[]):
```

```
    ...  
No: def foo(a, b=time.time()): # The time the module was loaded???
```

```
    ...  
No: def foo(a, b=FLAGS.my_thing): # sys.argv has not yet been parsed...  
    ...
```

属性(properties)

Tip

访问和设置数据成员时, 你通常会使用简单, 轻量级的访问和设置函数. 建议用属性(properties)来代替它们.

定义:

一种用于包装方法调用的方式. 当运算量不大, 它是获取和设置属性(attribute)的标准方式.

优点:

通过消除简单的属性(attribute)访问时显式的`get`和`set`方法调用, 可读性提高了. 允许懒惰的计

算. 用Pythonic的方式来**维护**类的接口. 就性能而言, 当直接访问变量是合理的, 添加**访问方法**就显得**琐碎而无意义**. 使用属性(properties)可以**绕过这个问题**. 将来也可以在不破坏接口的情况下将**访问方法**加上.

缺点:

属性(properties)是在get和set方法声明后指定, **这需要使用者在接下来的代码中注意**: set和get是用于属性(properties)的(除了用 @property 装饰器创建的只读属性). 必须继承自object类. 可能**隐藏**比如操作符重载之类的副作用. **继承时**可能会让人困惑.

结论:

你通常习惯于使用**访问或设置方法**来访问或设置数据, 它们简单而轻量. 不过我们建议你在新的代码中使用属性. 只读属性应该用 @property **装饰器** [\[http://google-styleguide.googlecode.com/svn/trunk/pyguide.html#Function_and_Method_Decorators\]](http://google-styleguide.googlecode.com/svn/trunk/pyguide.html#Function_and_Method_Decorators) 来创建.

如果子类没有覆盖属性, 那么属性的**继承**可能看上去不明显. 因此使用者必须确保访问方法间接被调用, 以保证子类中的**重载**方法被属性调用(使用模板方法设计模式).

```
Yes: import math

class Square(object):
    """A square with two properties: a writable area and a read-only perimeter

    To use:
    >>> sq = Square(3)
    >>> sq.area
    9
    >>> sq.perimeter
    12
    >>> sq.area = 16
    >>> sq.side
    4
    >>> sq.perimeter
    16
    """

    def __init__(self, side):
        self.side = side

    def __get_area(self):
        """Calculates the 'area' property."""
        return self.side ** 2

    def __get_area(self):
        """Indirect accessor for 'area' property."""
        return self.__get_area()

    def __set_area(self, area):
        """Sets the 'area' property."""
        self.side = math.sqrt(area)

    def __set_area(self, area):
        """Indirect setter for 'area' property."""
        self._SetArea(area)

    area = property(__get_area, __set_area,
                    doc="""Gets or sets the area of the square.""")
```

```
@property
def perimeter(self):
    return self.side * 4
```

(译者注: 老实说, 我觉得这段示例代码很不恰当, 有必要这么蛋疼吗?)

True/False的求值

Tip

尽可能使用隐式false

定义:

Python在布尔上下文中会将某些值求值为false. 按简单的直觉来讲, 就是所有的”空”值都被认为是false. 因此0, None, [], {}, “” 都被认为是false.

优点:

使用Python布尔值的条件语句更易读也更不易犯错. 大部分情况下, 也更快.

缺点:

对C/C++开发人员来说, 可能看起来有点怪.

结论:

尽可能使用隐式的false, 例如: 使用 `if foo:` 而不是 `if foo != []:`. 不过还是有一些注意事项需要你铭记在心:

1. 永远不要用`==`或者`!=`来比较单件, 比如None. 使用`is`或者`is not`.
2. 注意: 当你写下 `if x:` 时, 你其实表示的是 `if x is not None`. 例如: 当你要测试一个默认值是None的变量或参数是否被设为其它值. 这个值在布尔语义下可能是false!
3. 永远不要用`==`将一个布尔量与false相比较. 使用 `if not x:` 代替. 如果你需要区分false和None, 你应该用像 `if not x and x is not None:` 这样的语句.
4. 对于序列(字符串, 列表, 元组), 要注意空序列是false. 因此 `if not seq:` 或者 `if seq:` 比 `if len(seq):` 或 `if not len(seq):` 要更好.
5. 处理整数时, 使用隐式false可能会得不偿失(即不小心将None当做0来处理). 你可以将一个已知是整型(且不是len()的返回结果)的值与0比较.

```
Yes: if not users:
    print 'no users'

    if foo == 0:
        self.handle_zero()

    if i % 10 == 0:
        self.handle_multiple_of_ten()
```

```
No: if len(users) == 0:
```

```
print 'no users'

if foo is not None and not foo:
    self.handle_zero()

if not i % 10:
    self.handle_multiple_of_ten()
```

6. 注意‘0’(字符串)会被当做true.

过时的语言特性

Tip

尽可能使用字符串方法取代字符串模块. 使用函数调用语法取代apply(). 使用列表推导, for循环取代filter(), map()以及reduce().

定义:

当前版本的Python提供了大家通常更喜欢的替代品.

结论:

我们不使用不支持这些特性的Python版本, 所以没理由不用新的方式.

Yes: `words = foo.split(':')`

```
[x[1] for x in my_list if x[2] == 5]
```

```
map(math.sqrt, data)    # Ok. No inlined lambda expression.
```

```
fn(*args, **kwargs)
```

No: `words = string.split(foo, ':')`

```
map(lambda x: x[1], filter(lambda x: x[2] == 5, my_list))
```

```
apply(fn, args, kwargs)
```

词法作用域(Lexical Scoping)

Tip

推荐使用

定义:

嵌套的Python函数可以引用外层函数中定义的变量, 但是不能够对它们赋值. 变量绑定的解析是使用词法作用域, 也就是基于静态的程序文本. 对一个块中的某个名称的任何赋值都会导致Python将对该名称的全部引用当做局部变量, 甚至是赋值前的处理. 如果碰到global声明, 该名称就会被视作全局变量.

一个使用这个特性的例子:

```
def get_adder(summand1):  
    """Returns a function that adds numbers to a given number."""  
    def adder(summand2):  
        return summand1 + summand2  
  
    return adder
```

(译者注: 这个例子有点诡异, 你应该这样使用这个函数: `sum = get_adder(summand1)(summand2)`)

优点:

通常可以带来更加清晰, 优雅的代码. 尤其会让有经验的Lisp和Scheme(还有Haskell, ML等)程序员感到欣慰.

缺点:

可能导致让人迷惑的bug. 例如下面这个依据 [PEP-0227](http://www.python.org/dev/peps/pep-0227/) [http://www.python.org/dev/peps/pep-0227/] 的例子:

```
i = 4  
def foo(x):  
    def bar():  
        print i,  
        # ...  
        # A bunch of code here  
        # ...  
    for i in x: # Ah, i is local to Foo, so this is what Bar sees  
        print i,  
    bar()
```

因此 `foo([1, 2, 3])` 会打印 1 2 3 3, 不是 1 2 3 4.

(译者注: x是一个列表, for循环其实是将x中的值依次赋给i.这样对i的赋值就隐式的发生了, 整个foo函数体中的i都会被当做局部变量, 包括bar()中的那个. 这一点与C++之类的静态语言还是有很大差别的.)

结论:

鼓励使用.

函数与方法装饰器

Tip

如果好处很显然, 就明智而谨慎的使用装饰器

定义:

[用于函数及方法的装饰器](https://docs.python.org/release/2.4.3/whatsnew/node6.html) [https://docs.python.org/release/2.4.3/whatsnew/node6.html] (也就是@标记). 最常见的装饰器是@classmethod 和@staticmethod, 用于将常规函数转换成类方法或静态方法. 不过, 装饰器语法也允许用户自定义装饰器. 特别地, 对于某个函数 `my_decorator`, 下面的两段代码

是等效的:

```
class C(object):
    @my_decorator
    def method(self):
        # method body ...
```

```
class C(object):
    def method(self):
        # method body ...
    method = my_decorator(method)
```

优点:

优雅的在函数上指定一些**转换**. 该转换可能减少一些重复代码, 保持已有函数不变(enforce invariants), 等.

缺点:

装饰器可以在函数的参数或返回值上**执行**任何操作, 这可能导致让人惊异的**隐藏行为**. 而且, 装饰器在**导入时执行**. 从装饰器代码的失败中恢复更加不可能.

结论:

如果好处很显然, 就明智而谨慎的使用装饰器. 装饰器应该遵守和函数一样的导入和命名规则. 装饰器的python文档应该清晰的说明该函数是一个装饰器. 请为装饰器编写单元测试.

避免装饰器自身对外界的依赖(即不要依赖于文件, socket, 数据库连接等), 因为装饰器运行时这些资源可能不可用(由 pydoc 或其它工具导入). 应该保证一个用有效参数调用的装饰器在所有情况下都是成功的.

装饰器是一种特殊形式的”顶级代码”. 参考后面关于 [Main](#) 的话题.

线程

Tip

不要依赖内建类型的原子性.

虽然Python的内建类型例如字典看上去拥有原子操作, 但是在某些情形下它们仍然不是原子的(即: 如果__hash__或__eq__被实现为Python方法)且它们的原子性是靠不住的. 你也不能指望原子变量赋值(因为这个反过来依赖字典).

优先使用Queue模块的 queue 数据类型作为线程间的数据通信方式. 另外, 使用threading模块及其锁原语(locking primitives). 了解条件变量的合适使用方式, 这样你就可以使用 threading.Condition 来取代低级别的锁了.

威力过大的特性

Tip

避免使用这些特性

定义:

Python是一种异常灵活的语言, 它为你提供了很多花哨的特性, 诸如元类(metaclasses), 字节码访问, 任意编译(on-the-fly compilation), 动态继承, 对象父类重定义(object reparenting), 导入黑客(import hacks), 反射, 系统内修改(modification of system internals), 等等.

优点:

强大的语言特性, 能让你的代码更紧凑.

缺点:

使用这些很”酷”的特性十分诱人, 但不是绝对必要. 使用奇技淫巧的代码将更加难以阅读和调试. 开始可能还好(对原作者而言), 但当你回顾代码, 它们可能会比那些稍长一点但是很直接的代码更加难以理解.

结论:

在你的代码中避免这些特性.

Python风格规范

分号

Tip

不要在行尾加分号, 也不要加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:

1. 长的导入模块语句
2. 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](#)

[http://docs.python.org/2/reference/lexical_analysis.html#implicit-line-joining] , 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_
```

```
No: # See details at
     # http://www.example.com/us/developer/documentation/api/content/\
```

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](http://www.python.org/dev/peps/pep-0394/) [http://www.python.org/dev/peps/pep-0394/], 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](http://en.wikipedia.org/wiki/Shebang_(Unix)) [http://en.wikipedia.org/wiki/Shebang_(Unix)] (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](http://www.python.org/dev/peps/pep-0257/) [http://www.python.org/dev/peps/pep-0257/]). 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:

1. 外部不可见
2. 非常短小
3. 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](#) [http://en.wikipedia.org/wiki/Code_review] 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](http://www.python.org/dev/peps/pep-3000/) [http://www.python.org/dev/peps/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__`, `__getattr__`, `__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.

(译者注: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:

1. 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.
2. 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.
3. 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:

1. 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.

2. 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](http://docs.python.org/reference/compound_stmts.html#the-with-statement) [http://docs.python.org/reference/compound_stmts.html#the-with-statement] 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:

1. 标准库导入
2. 第三方库导入
3. 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, 那么就不要再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

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

命名约定

- 1. 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.
- 2. 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).
- 3. 用双下划线(__)开头的实例变量或方法表示类内私有.
- 4. 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.
- 5. 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 | <code>lower_with_under</code> | <code>_lower_with_under</code> |
| Packages | <code>lower_with_under</code> | |
| Classes | <code>CapWords</code> | <code>_CapWords</code> |
| Exceptions | <code>CapWords</code> | |
| Functions | <code>lower_with_under()</code> | <code>_lower_with_under()</code> |
| Global/Class Constants | <code>CAPS_WITH_UNDER</code> | <code>_CAPS_WITH_UNDER</code> |
| Global/Class Variables | <code>lower_with_under</code> | <code>_lower_with_under</code> |
| Instance Variables | <code>lower_with_under</code> | <code>_lower_with_under</code> (protected) or <code>__lower_with_under</code> (private) <code>_lower_with_under()</code> (protected) or |

| | | |
|----------------------------|--------------------|--------------------------------|
| Method Names | lower_with_under() | __lower_with_under() (private) |
| Function/Method Parameters | lower_with_under | |
| Local Variables | lower_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**.

临别赠言

请务必保持代码的一致性

如果你正在编辑代码, 花几分钟看一下周边代码, 然后决定风格. 如果它们在所有的算术操作符两边都使用空格, 那么你也应该这样做. 如果它们的注释都用标记包围起来, 那么你的注释也要这样.

制定风格指南的目的在于让代码有规可循, 这样人们就可以专注于”你在说什么”, 而不是”你在怎么说”. 我们在这里给出的是全局的规范, 但是本地的规范同样重要. 如果你加到一个文件里的代码和原有代码大相径庭, 它会让读者不知所措. 避免这种情况.

Revision 2.59

Amit Patel
Antoine Picard
Eugene Jhong
Gregory P. Smith
Jeremy Hylton
Matt Smart
Mike Shields
Shane Liebling

Shell 风格指南 - 内容目录

Contents

- [Shell 风格指南 - 内容目录](#)

- [扉页](#)
- [背景](#)
 - [使用哪一种Shell](#)
 - [什么时候使用Shell](#)
- [Shell文件和解释器调用](#)
 - [文件扩展名](#)
 - [SUID / SGID](#)
- [环境](#)
 - [STDOUT vs STDERR](#)
- [注释](#)
 - [文件头](#)
 - [功能注释](#)
 - [实现部分的注释](#)
 - [TODO注释](#)
- [格式](#)
 - [缩进](#)
 - [行的长度和长字符串](#)
 - [管道](#)
 - [循环](#)
 - [case语句](#)
 - [变量扩展](#)
 - [引用](#)
- [特性及错误](#)
 - [命令替换](#)
 - [test,\[和\[\[](#)
 - [测试字符串](#)
 - [文件名的通配符扩展](#)
 - [Eval](#)
 - [管道导向while循环](#)
- [命名转换](#)
 - [函数名](#)
 - [变量名](#)
 - [常量和环境变量名](#)
 - [源文件名](#)
 - [只读变量](#)
 - [使用本地变量](#)

- [函数位置](#)
 - [主函数main](#)
- [调用命令](#)
 - [检查返回值](#)
 - [内建命令和外部命令](#)
- [结论](#)

扉页

版本: 1.26

原作者: Paul Armstrong
等等

翻译: [Bean Zhang](#) v1.26

项目主页:

- [Google Style Guide](#)
- [Google 开源项目风格指南 - 中文版](#)

背景

使用哪一种Shell

Tip

Bash是唯一被允许执行的shell脚本语言。

可执行文件必须以 `#!/bin/bash` 和最小数量的标志开始。请使用 `set` 来设置shell的选项,使得用 `bash <script_name>` 调用你的脚本时不会破坏其功能。

限制所有的可执行shell脚本为bash使得我们安装在所有计算机中的shell语言保持一致性。

无论你是为什么而编码,对此唯一例外的是当你被迫时可以不这么做的。其中一个例子是Solaris SVR4包,编写任何脚本都需要用纯Bourne shell。

什么时候使用Shell

Tip

Shell应该仅仅被用于小功能或者简单的包装脚本。

尽管Shell脚本不是一种开发语言,但在整个谷歌它被用于编写多种实用工具的脚本。这个风格指南更多的是认同它的使用,而不是一个建议,即它可被用于广泛部署。

以下是一些准则:

- 如果你主要是在调用其他的工具并且做一些相对很小数据量的操作,那么使用shell来完成任**务**是一种可接受的选择。
- 如果你在在乎性能,那么**请选择**其他工具,而不是使用shell。
- 如果你**发现**你需要使用数据而不是**变量赋值**(如 `${PHPESTATUS}`),那么你应该使用Python脚本。
- 如果你将要编写的脚本会超过100行,那么你可能**应该**使用Python来编写,而不是Shell。请记住,当脚本行数增加,尽早使用另外一种语言重写你的脚本,以避免之后花更多的时间来重写。

Shell文件和解释器调用

文件扩展名

Tip

可执行文件应该没有扩展名(强烈建议)或者使用.sh扩展名。库文件必须使用.sh作为扩展名,而且应该是不可执行的。

当执行一个程序时,并不需要知道它是用什么语言编写的。而且shell脚本也不要求有扩展名。所以我们更喜欢可执行文件没有扩展名。

然而,对于库文件,知道其用什么语言编写的是很重要的,有时候会需要使用不同语言编写的相似的库文件。使用.sh这样特定语言后缀作为扩展名,就使得用不同语言编写的具有相同功能的库文件可以采用一样的名称。

SUID / SGID

Tip

SUID(Set User ID)和SGID(Set Group ID)在shell脚本中是被禁止的。

shell存在太多的安全问题,以致于如果允许SUID/SGID会使得shell几乎不可能足够安全。虽然bash使得运行SUID非常困难,但在某些平台上仍然有可能运行,这就是为什么我们明确提出要禁止它。

如果你需要较高权限的访问请使用 `sudo`。

环境

STDOUT vs STDERR

Tip

所有的错误信息都应该被导向STDERR。

这使得从实际问题中分离出正常状态变得更容易。

推荐使用类似如下函数,将错误信息和其他状态信息一起打印出来。

```
err() {  
    echo "[$(date +%Y-%m-%dT%H:%M:%S%z)]: $@" >&2  
}  
  
if ! do_something; then  
    err "Unable to do_something"  
    exit "${E_DID_NOTHING}"  
fi
```

注释

文件头

Tip

每个文件的开头是其文件内容的描述。

每个文件必须包含一个**顶层注释**,对其内容进行**简要概述**。**版权声明**和**作者信息**是**可选的**。

例如:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.
```

功能注释

Tip

任何不是**既明显又短**的函数都必须被**注释**。任何**库函数**无论其**长短**和**复杂性**都必须被**注释**。

其他人通过**阅读注释**(和**帮助信息**,如果有的话)就能**够**学会如何使用你的程序或**库函数**,而不需要**阅读代码**。

所有的函数**注释**应该包含:

- 函数的描述
- 全局**变量**的使用和修改
- 使用的**参数说明**
- 返回**值**,而不是上一条命令运行后默认的退出状态

例如:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.

export PATH='/usr/xpg4/bin:/usr/bin:/opt/csw/bin:/opt/goog/bin'

#####
# Cleanup files from the backup dir
# Globals:
#   BACKUP_DIR
```

```
# ORACLE_SID
# Arguments:
# None
# Returns:
# None
#####
cleanup() {
    ...
}
```

实现部分的注释

Tip

注释你代码中含有技巧、不明显、有趣的或者重要的部分。

这部分遵循谷歌代码注释的通用做法。不要注释所有代码。如果有一个复杂的算法或者你正在做一些与众不同的,放一个简单的注释。

TODO注释

Tip

使用TODO注释临时的、短期解决方案的、或者足够好但不够完美的代码。

这与C++指南中的约定相一致。

TODOs应该包含全部大写的字符串TODO,接着是括号中你的用户名。冒号是可选的。最好在TODO条目之后加上 bug 或者ticket 的序号。

例如:

```
# TODO(mrmonkey): Handle the unlikely edge cases (bug ####)
```

格式

缩进

Tip

缩进两个空格,没有制表符。

在代码块之间请使用空行以提升可读性。**缩进**为两个空格。无论你做什么,请不要使用制表符。对于已有文件,保持已有的**缩进**格式。

行的长度和长字符串

Tip

行的最大长度为80个字符。

如果你必须写长度超过80个字符的字符串,如果可能的话,尽量使用here document或者嵌入的换行符。长度超过80个字符的文字串且不能被合理地分割,这是正常的。但强烈建议找到一个方法使其变短。

```
# DO use 'here document's
cat <<END;
I am an exceptionally long
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
long string."
```

管道

Tip

如果一行容不下整个管道操作,那么**请**将整个管道操作分割成每行一个管段。

如果一行容得下整个管道操作,那么**请**将整个管道操作写在同一行。

否则,应该将整个管道操作分割成每行一个管段,管道操作的下一部分**应该**将管道符放在新行并且**缩进**2个空格。这适用于使用管道符'|'的合并命令链以及使用'|'和'&&'的逻辑运算链。

```
# All fits on one line
```

```
command1 | command2
```

```
# Long commands
```

```
command1 \  
| command2 \  
| command3 \  
| command4
```

循环

Tip

请将 ; do , ; then 和 while , for , if 放在同一行。

shell中的循环略有不同,但是我们遵循跟声明函数时的大括号相同的原则。也就是说, ; do , ; then 应该和 if/for/while 放在同一行。else 应该单独一行,结束语句应该单独一行并且跟开始语句垂直对齐。

例如:

```
for dir in ${dirs_to_cleanup}; do  
  if [[ -d "${dir}/${ORACLE_SID}" ]]; then  
    log_date "Cleaning up old files in ${dir}/${ORACLE_SID}"  
    rm "${dir}/${ORACLE_SID}/*"  
    if [[ "$?" -ne 0 ]]; then  
      error_message  
    fi  
  else  
    mkdir -p "${dir}/${ORACLE_SID}"  
    if [[ "$?" -ne 0 ]]; then  
      error_message  
    fi  
  fi  
done
```

case语句

Tip

- 通过2个空格缩进可选项。
- 在同一行可选项的模式右圆括号之后和结束符 ;; 之前各需要一个空格。
- 长可选项或者多命令可选项应该被拆分成多行,模式、操作和结束符 ;; 在不同的行。

匹配表达式比 case 和 esac 缩进一级。多行操作要再缩进一级。一般情况下,不需要引用匹配表达式。模式表达式前面不应该出现左括号。避免使用 ;& 和 ;;& 符号。

```
case "${expression}" in  
  a)
```



```

variable="..."
some_command "${variable}" "${other_expr}" ...
;;
absolute)
actions="relative"
another_command "${actions}" "${other_expr}" ...
;;
*)
error "Unexpected expression '${expression}'"
;;
esac

```

只要整个表达式可读,简单的命令可以跟模式和 `;;` 写在同一行。这通常适用于单字母选项的处理。当单行容不下操作时,请将模式单独放一行,然后是操作,最后结束符 `;;` 也单独一行。当操作在同一行时,模式的右括号之后和结束符 `;;` 之前请使用一个空格分隔。

```

verbose='false'
aflag=''
bflag=''
files=''
while getopts 'abf:v' flag; do
  case "${flag}" in
    a) aflag='true' ;;
    b) bflag='true' ;;
    f) files="${OPTARG}" ;;
    v) verbose='true' ;;
    *) error "Unexpected option ${flag}" ;;
  esac
done

```

变量扩展

Tip

按优先级顺序:保持跟你所发现的一致;引用你的变量;推荐用 `${var}` 而不是 `$var`,详细解释如下。

这些仅仅是指南,因为作为强制规定似乎饱受争议。

以下按照优先顺序列出。

1. 与现存代码中你所发现的保持一致。
2. 引用变量参阅下面一节,引用。
3. 除非绝对必要或者为了避免深深的困惑,否则不要用大括号将单个字符的shell特殊变量或定位变量括起来。推荐将其他所有变量用大括号括起来。

```

# Section of recommended cases.

# Preferred style for 'special' variables:
echo "Positional: $1" "$5" "$3"
echo "Specials: !=$, -=$-, _=$_. ?=$?, #=$# *=$* @=$@ \=$$ ..."

```

```

# Braces necessary:
echo "many parameters: ${10}"

# Braces avoiding confusion:
# Output is "a0b0c0"
set -- a b c
echo "${1}0${2}0${3}0"

# Preferred style for other variables:
echo "PATH=${PATH}, PWD=${PWD}, mine=${some_var}"
while read f; do
    echo "file=${f}"
done < <(ls -l /tmp)

# Section of discouraged cases

# Unquoted vars, unbraced vars, brace-quoted single letter
# shell specials.
echo a=$avar "b=$bvar" "PID=${$}" "${1}"

# Confusing use: this is expanded as "${1}0${2}0${3}0",
# not "${10}${20}${30}"
set -- a b c
echo "$10$20$30"

```

引用

Tip

- 除非需要小心不带引用的扩展,否则总是引用包含变量、命令替换符、空格或shell元字符的字符串。
- 推荐引用是单词的字符串(而不是命令选项或者路径名)。
- 千万不要引用整数。
- 注意 [[中模式匹配的引用规则。
- 请使用 \$@ 除非你有特殊原因需要使用 \$* 。

```

# 'Single' quotes indicate that no substitution is desired.
# "Double" quotes indicate that substitution is required/tolerated.

# Simple examples
# "quote command substitutions"
flag="$(some_command and its args "$@" 'quoted separately')"

# "quote variables"
echo "${flag}"

# "never quote literal integers"
value=32
# "quote command substitutions", even when you expect integers
number="$(generate_number)"

# "prefer quoting words", not compulsory
readonly USE_INTEGER='true'

# "quote shell meta characters"
echo 'Hello stranger, and well met. Earn lots of $$$'

```

```

echo "Process $$: Done making \$\$\$."

# "command options or path names"
# ($1 is assumed to contain a value here)
grep -li Hugo /dev/null "$1"

# Less simple examples
# "quote variables, unless proven false": ccs might be empty
git send-email --to "${reviewers}" ${ccs:+"--cc" "${ccs}"}

# Positional parameter precautions: $1 might be unset
# Single quotes leave regex as-is.
grep -cP '([Ss]pecial|\\|?characters*)$' ${1:+"$1"}

# For passing on arguments,
# "$@" is right almost everytime, and
# "$*" is wrong almost everytime:
#
# * "$*" and "$@" will split on spaces, clobbering up arguments
#   that contain spaces and dropping empty strings;
# * "$@" will retain arguments as-is, so no args
#   provided will result in no args being passed on;
#   This is in most cases what you want to use for passing
#   on arguments.
# * "$*" expands to one argument, with all args joined
#   by (usually) spaces,
#   so no args provided will result in one empty string
#   being passed on.
# (Consult 'man bash' for the nit-grits ;-))

set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$*"; echo "$#, $@")
set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$@"; echo "$#, $@")

```

特性及错误

命令替换

Tip

使用 `$(command)` 而不是反引号。

嵌套的反引号要求用反斜杠转义内部的反引号。而 `$(command)` 形式嵌套时不需要改变,而且更易于阅读。

例如:

```
# This is preferred:
var="$(command "$(command1)")"

# This is not:
var="`command \`command1\``"
```

test,[和[[

Tip

推荐使用 `[[...]]`,而不是 `[, test ,` 和 `/usr/bin/ [` 。

因为在 `[[` 和 `]]` 之间不会有路径名称扩展或单词分割发生,所以使用 `[[...]]` 能够减少错误。而且 `[[...]]` 允许正则表达式匹配,而 `[...]` 不允许。

```
# This ensures the string on the left is made up of characters in the
# alnum character class followed by the string name.
# Note that the RHS should not be quoted here.
# For the gory details, see
# E14 at http://tiswww.case.edu/php/chet/bash/FAQ
if [[ "filename" =~ ^[:alnum:]+name ]]; then
    echo "Match"
fi

# This matches the exact pattern "f*" (Does not match in this case)
if [[ "filename" == "f*" ]]; then
    echo "Match"
fi

# This gives a "too many arguments" error as f* is expanded to the
# contents of the current directory
if [ "filename" == f* ]; then
    echo "Match"
```

```
fi
```

测试字符串

Tip

尽可能使用引用,而不是过滤字符串。

Bash足以在测试中处理空字符串。所以,请使用空(非空)字符串测试,而不是过滤字符,使得代码更易于阅读。

```
# Do this:
if [[ "${my_var}" = "some_string" ]]; then
    do_something
fi

# -z (string length is zero) and -n (string length is not zero) are
# preferred over testing for an empty string
if [[ -z "${my_var}" ]]; then
    do_something
fi

# This is OK (ensure quotes on the empty side), but not preferred:
if [[ "${my_var}" = "" ]]; then
    do_something
fi

# Not this:
if [[ "${my_var}X" = "some_stringX" ]]; then
    do_something
fi
```

为了避免对你测试的目的产生困惑,请明确使用`-z`或者`-n`

```
# Use this
if [[ -n "${my_var}" ]]; then
    do_something
fi

# Instead of this as errors can occur if ${my_var} expands to a test
# flag
if [[ "${my_var}" ]]; then
    do_something
fi
```

文件名的通配符扩展

Tip

当进行文件名的通配符扩展时,请使用明确的路径。

因为文件名可能以 - 开头,所以使用扩展通配符 ./ * 比 * 来得安全得多。

```
# Here's the contents of the directory:
# -f -r somedir somefile

# This deletes almost everything in the directory by force
psa@bilby$ rm -v *
removed directory: `somedir'
removed `somefile'

# As opposed to:
psa@bilby$ rm -v ./ *
removed `./-f'
removed `./-r'
rm: cannot remove `./somedir': Is a directory
removed `./somefile'
```

Eval

Tip

应该避免使用eval。

当用于给变量赋值时,Eval解析输入,并且能够设置变量,但无法检查这些变量是什么。

```
# What does this set?
# Did it succeed? In part or whole?
eval $(set_my_variables)

# What happens if one of the returned values has a space in it?
variable="$(eval some_function)"
```

管道导向while循环

Tip

请使用过程替换或者for循环,而不是管道导向while循环。在while循环中被修改的变量是不能传递给父shell的,因为循环命令是在一个子shell中运行的。

管道导向while循环中的隐式子shell使得追踪bug变得很困难。

```
last_line='NULL'
your_command | while read line; do
    last_line="${line}"
done

# This will output 'NULL'
echo "${last_line}"
```

如果你确定输入中不包含空格或者特殊符号(通常意味着不是用户输入的),那么可以使用一个for

循环。

```
total=0
# Only do this if there are no spaces in return values.
for value in $(command); do
    total+="${value}"
done
```

使用过程替换允许重定向输出,但是请将命令放入一个显式的子shell中,而不是bash为while循环创建的隐式子shell。

```
total=0
last_file=
while read count filename; do
    total+="${count}"
    last_file="${filename}"
done < <(your_command | uniq -c)

# This will output the second field of the last line of output from
# the command.
echo "Total = ${total}"
echo "Last one = ${last_file}"
```

当不需要传递复杂的结果给父shell时可以使用while循环。这通常需要一些更复杂的“解析”。请注意简单的例子使用如awk这类工具可能更容易完成。当你特别不希望改变父shell的范围变量时这可能也是有用的。

```
# Trivial implementation of awk expression:
# awk '$3 == "nfs" { print $2 " maps to " $1 }' /proc/mounts
cat /proc/mounts | while read src dest type opts rest; do
    if [[ ${type} == "nfs" ]]; then
        echo "NFS ${dest} maps to ${src}"
    fi
done
```

命名转换

函数名

Tip

使用小写字母,并用下划线分隔单词。使用双冒号 :: 分隔库。函数名之后必须有圆括号。关键词 `function` 是可选的,但必须在一个项目中保持一致。

如果你正在写单个函数,请用小写字母来命名,并用下划线分隔单词。如果你正在写一个包,使用双冒号 :: 来分隔包名。大括号必须和函数名位于同一行(就像在Google的其他语言一样),并且函数名和圆括号之间没有空格。

```
# Single function
my_func() {
    ...
}

# Part of a package
mypackage::my_func() {
    ...
}
```

当函数名后存在 `()` 时,关键词 `function` 是多余的。但是其促进了函数的快速辨识。

变量名

Tip

如函数名。

循环的变量名应该和循环的任何变量同样命名。

```
for zone in ${zones}; do
    something_with "${zone}"
done
```

常量和环境变量名

Tip

全部大写,用下划线分隔,声明在文件的顶部。

常量和任何导出到环境中的都应该大写。

```
# Constant
readonly PATH_TO_FILES='/some/path'

# Both constant and environment
declare -xr ORACLE_SID='PROD'
```

第一次设置时有一些就变成了常量(例如,通过getopts)。因此,可以在getopts中或基于条件来设定常量,但之后应该立即设置其为只读。值得注意的是,在函数中 declare 不会对全局变量进行操作。所以推荐使用 readonly 和 export 来代替。

```
VERBOSE='false'
while getopts 'v' flag; do
  case "${flag}" in
    v) VERBOSE='true' ;;
  esac
done
readonly VERBOSE
```

源文件名

Tip

小写,如果需要的话使用下划线分隔单词。

这是为了和在Google中的其他代码风格保持一致: maketemplate 或者 make_template ,而不是 make-template 。

只读变量

Tip

使用 readonly 或者 declare -r 来确保变量只读。

因为全局变量在shell中广泛使用,所以在它们的过程中捕获错误是很重要的。当你声明了一个变量,希望其只读,那么请明确指出。

```
zip_version="$(dpkg --status zip | grep Version: | cut -d ' ' -f 2)"
if [[ -z "${zip_version}" ]]; then
  error_message
else
  readonly zip_version
fi
```

使用本地变量

Tip

使用 `local` 声明特定功能的变量。声明和赋值应该在不同行。

使用 `local` 来声明局部变量以确保其只在函数内部和子函数中可见。这避免了污染全局命名空间和不经意间设置可能具有函数之外重要性的变量。

当赋值的值由命令替换提供时,声明和赋值必须分开。因为内建的 `local` 不会从命令替换中传递退出码。

```
my_func2() {
    local name="$1"

    # Separate lines for declaration and assignment:
    local my_var
    my_var="$(my_func)" || return

    # DO NOT do this: $? contains the exit code of 'local', not my_func
    local my_var="$(my_func)"
    [[ $? -eq 0 ]] || return

    ...
}
```

函数位置

Tip

将文件中所有的函数一起放在常量下面。不要在函数之间隐藏可执行代码。

如果你有函数,请将他们一起放在文件头部。只有`includes`, `set` 声明和常量设置可能在函数声明之前完成。不要在函数之间隐藏可执行代码。如果那样做,会使得代码在调试时难以跟踪并出现意想不到的讨厌结果。

主函数main

Tip

对于包含至少一个其他函数的足够长的脚本,需要称为 `main` 的函数。

为了方便查找程序的开始,将主程序放入一个称为 `main` 的函数,作为最下面的函数。这使其和代码库的其余部分保持一致性,同时允许你定义更多变量为局部变量(如果主代码不是一个函数就不能这么做)。文件中最后的非注释行应该是对 `main` 函数的调用。

```
main "$@"
```

显然,对于仅仅是线性流的短脚本, `main` 是矫枉过正,因此是不需要的。

调用命令

检查返回值

Tip

总是检查返回值,并给出信息返回值。

对于非管道命令,使用 `$?` 或直接通过一个 `if` 语句来检查以保持其简洁。

例如:

```
if ! mv "${file_list}" "${dest_dir}/" ; then
    echo "Unable to move ${file_list} to ${dest_dir}" >&2
    exit "${E_BAD_MOVE}"
fi

# Or
mv "${file_list}" "${dest_dir}/"
if [[ "$?" -ne 0 ]]; then
    echo "Unable to move ${file_list} to ${dest_dir}" >&2
    exit "${E_BAD_MOVE}"
fi
```

Bash也有 `PIPESTATUS` 变量,允许检查从管道所有部分返回的代码。如果仅仅需要检查整个管道是成功还是失败,以下的方法是可以接受的:

```
tar -cf - ./* | ( cd "${dir}" && tar -xf - )
if [[ "${PIPESTATUS[0]}" -ne 0 || "${PIPESTATUS[1]}" -ne 0 ]]; then
    echo "Unable to tar files to ${dir}" >&2
fi
```

可是,只要你运行任何其他命令, `PIPESTATUS` 将会被覆盖。如果你需要基于管道中发生的错误执行不同的操作,那么你需要在运行命令后立即将 `PIPESTATUS` 赋值给另一个变量(别忘了 `[` 是一个会将 `PIPESTATUS` 擦除的命令)。

```
tar -cf - ./* | ( cd "${DIR}" && tar -xf - )
return_codes=("${PIPESTATUS[*]}")
if [[ "${return_codes[0]}" -ne 0 ]]; then
    do_something
fi
if [[ "${return_codes[1]}" -ne 0 ]]; then
    do_something_else
fi
```

内建命令和外部命令

Tip

可以在调用shell内建命令和调用另外的程序之间选择,请选择内建命令。

我们更喜欢使用内建命令,如在 `bash(1)` 中参数扩展函数。因为它更强健和便携(尤其是跟像 `sed` 这样的命令比较)

例如:

```
# Prefer this:
addition=$(( ${X} + ${Y} ))
substitution="${string/#foo/bar}"

# Instead of this:
addition="$(expr ${X} + ${Y})"
substitution="$(echo "${string}" | sed -e 's/^foo/bar/')
```

结论

使用常识并保持一致。

请花几分钟阅读在C++风格指南底部的赠别部分。

Index

Google 开源项目风格指南 (中文版)

- 在线文档托管在 ReadTheDocs : [在线阅读最新版本](http://zh-google-styleguide.readthedocs.org/) [http://zh-google-styleguide.readthedocs.org/]
- 中文风格指南 GitHub 托管地址: [zh-google-styleguide](https://github.com/zh-google-styleguide/zh-google-styleguide) [https://github.com/zh-google-styleguide/zh-google-styleguide]

Note

声明.

本项目并非 Google 官方项目, 而是由国内程序员凭热情创建和维护.

如果你关注的是 Google 官方英文版, 请移步 [Google Style Guide](https://github.com/google/styleguide) [https://github.com/google/styleguide]

每个较大的开源项目都有自己的风格指南: 关于如何为该项目编写代码的一系列约定 (有时候会比较武断). 当所有代码均保持一致的风格, 在理解大型代码库时更为轻松.

“风格”的含义涵盖范围广, 从“变量使用驼峰格式 (camelCase)”到“决不使用全局变量”再到“决不使用异常”. 英文版项目维护的是在 Google 使用的编程风格指南. 如果你正在修改的项目源自 Google, 你可能会被引导至英文版项目页面, 以了解项目所使用的风格.

我们已经发布了五份 中文版 的风格指南:

1. [Google C++ 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-cpp-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-cpp-styleguide/]
2. [Google Objective-C 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-objc-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-objc-styleguide/]
3. [Google Python 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-python-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-python-styleguide/]
4. [Google JSON 风格指南](https://github.com/darcyliu/google-styleguide/blob/master/JSONStyleGuide.md) [https://github.com/darcyliu/google-styleguide/blob/master/JSONStyleGuide.md]
5. [Google Shell 风格指南](http://zh-google-styleguide.readthedocs.org/en/latest/google-shell-styleguide/) [http://zh-google-styleguide.readthedocs.org/en/latest/google-shell-styleguide/]

中文版项目采用 reStructuredText 纯文本标记语法, 并使用 Sphinx 生成 HTML / CHM / PDF 等文档格式.

- 英文版项目还包含 [cpplint](https://github.com/google/styleguide/tree/gh-pages/cpplint) [https://github.com/google/styleguide/tree/gh-pages/cpplint] - 一个用来帮助适应风格准则的工具, 以及 [google-c-style.el](https://raw.githubusercontent.com/google/styleguide/gh-pages/google-c-style.el) [https://raw.githubusercontent.com/google/styleguide/gh-pages/google-c-style.el], Google 风格的 Emacs 配置文件.
- 另外, 招募志愿者翻译 [JavaScript Style Guide](http://google.github.io/styleguide/javascriptguide.xml) [http://google.github.io/styleguide/javascriptguide.xml] 以及 [XML Document Format Style Guide](http://google.github.io/styleguide/xmlstyle.html) [http://google.github.io/styleguide/xmlstyle.html], 有意者请联系 [Yang.Y](https://github.com/yangyubo) [https://github.com/yangyubo].

背景

本文档定义了HTML/CSS的排版以及风格的规则。旨在促进合作编程、提高代码质量并且支持基本的架构。它适用于原生的HTML和CSS文件,包括GSS文件。只要保证代码的整体质量,就可以很容易地使用工具进行混淆、压缩和合并。

CSS格式化规则

声明顺序

按字母顺序排列声明。

css文件书写按字母顺序排列的方式,容易记忆和维护,以达到一致的代码。

在排序时忽略浏览器特定的前缀。但是,特定CSS属性的多个浏览器前缀应按字母顺序排列(如-moz书写在-webkit前面)。

```
background: fuchsia;
border: 1px solid;
-moz-border-radius: 4px;
-webkit-border-radius: 4px;
border-radius: 4px;
color: black;
text-align: center;
text-indent: 2em;
```

块内容的缩进

缩进块内容。

将包括嵌套及声明的 [块内容](http://www.w3.org/TR/CSS21/syndata.html#block) 进行缩进,以体现层次并提高可读性。

```
@media screen, projection {

    html {
        background: #fff;
        color: #444;
    }

}
```

声明结束

每个属性后使用分号结束。

以分号结束每个属性,提高一致性和可扩展性。

```
/* 不推荐 */
.test {
    display: block;
    height: 100px
}
```

```
/* 推荐 */
.test {
  display: block;
  height: 100px;
}
```

CSS属性名结束

属性名称的冒号后有一个空格。

为保证一致性,在属性名与属性值之间添加一个空格(但是属性名和冒号间没有空格)。

```
/* 不推荐 */
h3 {
  font-weight:bold;
}

/* 推荐 */
h3 {
  font-weight: bold;
}
```

声明块间隔

在选择器和后面的声明块之间使用一个空格。

最后一个选择器与表示 [声名称](http://www.w3.org/TR/CSS21/syndata.html#rule-sets) 开始的左大花括号在同行,中间有一个字符空格。

表示开始的左大花括号和选择器在同行。

```
/* 不推荐：缺少空间 */
#video{
  margin-top: 1em;
}

/* 不推荐：不必要的换行符 */
#video
{
  margin-top: 1em;
}

/* 推荐 */
#video {
  margin-top: 1em;
}
```

选择器及声明分离

每个选择器和声明独立成行。

总是让每个选择器和声明单独成行。

```

/* 不推荐 */
a:focus, a:active {
    position: relative; top: 1px;
}

/* 推荐 */
h1,
h2,
h3 {
    font-weight: normal;
    line-height: 1.2;
}

```

CSS代码块分离

使用新空行分离规则。

始终把一个空行(两个换行符)放在代码块规则之间。

```

html {
    background: #fff;
}

body {
    margin: auto;
    width: 50%;
}

```

CSS引号

属性选择器和属性值中使用单引号。

在属性选择器及属性值中使用单引号(‘’)而不是双引号(“”).在 url () 中不要使用引号。

特例:如果你确实需要定义 @charset ,由于 [不允许使用单引号](http://www.w3.org/TR/CSS21/syndata.html#charset)
[http://www.w3.org/TR/CSS21/syndata.html#charset] ,故请使用双引号。

```

/* 不推荐 */
@import url("//www.google.com/css/maia.css");

html {
    font-family: "open sans", arial, sans-serif;
}

/* 推荐 */
@import url(//www.google.com/css/maia.css);

html {
    font-family: 'open sans', arial, sans-serif;
}

```

CSS元规则

分段规则

组的分段由一段注释完成(可选)。

尽可能地用注释来将css分段,段与段之间采用新行。

```
/* Header */  
  
#adw-header {}  
  
/* Footer */  
  
#adw-footer {}  
  
/* Gallery */  
  
.adw-gallery {}
```

css样式规则

CSS有效性

尽可能使用有效的CSS。

使用有效的CSS代码,除非在处理css验证器bug或者是专有的语法时。

使用诸如 [W3C CSS validator](http://jigsaw.w3.org/css-validator/) [http://jigsaw.w3.org/css-validator/] 等工具验证测试。

使用有效的CSS代码是一个可衡量CSS代码质量的指标,可帮你找出不起作用可被删除的CSS代码,从而确保CSS的合理使用。

id与class的命名

使用有意义的或者通用的id和class名称

用能反映出元素目的或者通用的id、class名称,代替那些很表象的、难懂的名称。

如果名称需要是易懂的,或不容易被修改,应该首选特定的或者能反映出元素目的的名称。

通用的名称适用于非特殊元素或与兄弟元素无区别的元素。他们常被称为“辅助元素”。

使用功能性或者通用的名称,可减少不必要的文档或者模板变化。

```
/* 不推荐：无意义 */
#yee-1901 {}
```

```
/* 不推荐：表象 */
.button-green {}
.clear {}
```

```
/* 推荐：具体的 */
#gallery {}
#login {}
.video {}
```

```
/* 推荐：通用的 */
.aux {}
.alt {}
```

id与class的命名规范

ID和class命名要尽可能简短,但必要的话就别怕长。

尽可能简洁地传达id或者class名称的含义。

使用简洁的id或者class名称有助于提高可读性和代码效率。

```
/* 不推荐 */
#navigation {}
.atr {}

/* 推荐 */
#nav {}
.author {}
```

选择器的类型

应当避免在id和class前添加类型选择器。

除了必要情况下(例如辅助的类),不要将元素与id或class名称结合做为选择器。

避免不必要的祖先选择器也是出于 [性能原因](http://www.stevesouders.com/blog/2009/06/18/simplifying-css-selectors/) [http://www.stevesouders.com/blog/2009/06/18/simplifying-css-selectors/] 的考虑。

```
/* 不推荐 */
ul#example {}
div.error {}

/* 推荐 */
#example {}
.error {}
```

简写属性

尽可能使用简写的属性书写方式。

CSS提供了多种属性 [简写](http://www.w3.org/TR/CSS21/about.html#shorthand) [http://www.w3.org/TR/CSS21/about.html#shorthand] 的方式(如 font),即使只显式设置一个值,也应该尽可能地使用。

使用简写属性有助于提高代码效率及可读性。

```
/* 不推荐 */
border-top-style: none;
font-family: palatino, georgia, serif;
font-size: 100%;
line-height: 1.6;
padding-bottom: 2em;
padding-left: 1em;
padding-right: 1em;
padding-top: 0;

/* 推荐 */
border-top: 0;
font: 100%/1.6 palatino, georgia, serif;
padding: 0 1em 2em;
```

0与单位

省略“0”后的单位。

除非必需,否则0后不要加单位。

```
margin: 0;
padding: 0;
```

前导0

省略前导“0”值。

在-1至1之间的值无需保留整数位的0。

```
font-size: .8em;
```

十六进制表示法

在可能的情况下使用3个字符的十六进制表示法。

对于可用3字符十六进制表示的颜色值,按此规则书写更短、更简洁。

```
/* 不推荐 */
color: #eebbcc;

/* 推荐 */
color: #ebc;
```

前缀选择器

加特定应用前缀(可选)

大型项目中以及嵌入在其它项目或外部网站上的代码需要给id和class添加前缀(命名空间)。使用短的、独特的标识符,并在其后跟一个破折号。使用命名空间有助于防止命名冲突,可以让维护变得简单,例如在搜索和替换操作时。

```
.adw-help {} /* AdWords */
#maia-note {} /* Maia */
```

id与class名称分隔符

用连字符分隔ID和类名中的单词。

选择器中的词语和缩写中不要使用除了连字符以外的任何字符(包括空字符),以提高可理解性和可读性。

```
/* 不推荐: 单词未分开 */
.demoimage {}

/* 不推荐: 使用下划线而不是连字符 */
```



```
.error_status {}
```

```
/* 推荐 */  
#video-id {}  
.ads-sample {}
```

Hacks

请先尝试其他的方法,避免用户代理检测以及CSS的“hacks”。

进行用户代理检测或使用特殊的CSS选择器及hacks看起来是处理样式差异的捷径。但为了实现和保持高效性以及代码的可维护性,这两种方案应该放到最后考虑。换句话说,用户代理检测和使用hacks会增大项目推进的阻力,所以从项目的长远利益考虑应尽力避免。一旦允许并无顾忌地使用用户代理检测和hacks便很容易滥用,最终一发而不可收。

总体排版规则

缩进

每次**缩进**使用两个空格。

不使用**TAB键**或者混合使用**TAB键**和**空格**进行**缩进**。

```
<ul>
  <li>Fantastic
  <li>Great
</ul>
```

```
.example {
  color: blue;
}
```

大小写

只使用小写字母。

所有的代**码**都使用小写字母: 适用于HTML元素、属性、属性**值**(除了text/CDATA)、CSS**选择器**、属性名以及属性**值**(字符串除外)。

```
<!-- 不推荐 -->
<A HREF="/">Home</A>
```

```
<!-- 推荐 -->

```

```
/* 不推荐 */
color: #E5E5E5;
```

```
/* 推荐 */
color: #e5e5e5;
```

尾部的空格

删除尾部的空格。

尾部的空格是多余的, 不**删除**则形成无意义的文件差异。

```
<!-- 不推荐 -->
<p>What?_
```

```
<!-- 推荐 -->
<p>Yes please.
```

整体的元数据规则

编码

使用UTF-8无BOM编码。

让你的编辑器使用无字节顺序标记的UTF-8编码。

在HTML模板和文档中使用 `<meta charset="utf-8">` 指定编码。不需要为样式表指定编码,它默认是UTF-8。

(想了解更多关于应该何时并如何指定编码,请查看 [Handling character encodings in HTML and CSS](http://www.w3.org/International/tutorials/tutorial-char-enc/) [http://www.w3.org/International/tutorials/tutorial-char-enc/])

注释

在需要时尽可能去解释你的代码。

用注释去解释你的代码,包括它的应用范围、用途、此方案的选择理由等。

(这一条是可选的,没必要为每个文件写上详细的注释,会增重HTML/CSS的代码,主要取决于项目的复杂度。)

处理内容

用TODO标记待办事宜和处理内容。

只用TODO来标记待办事宜,不要使用其他格式,例如@@。

在括号里添加联系方式(姓名或邮箱),格式为TODO(联系方式)。

在冒号后面添加处理内容,格式为TODO:处理内容。

```
{# TODO( john.doe): 重新处理水平居中 #}  
<center>Test</center>  
  
<!-- TODO: 移除可选的标签 -->  
<ul>  
  <li>Apples</li>  
  <li>Oranges</li>  
</ul>
```

整体样式规则

协议

嵌入式资源省略协议头。

省略图片、媒体文件、样式表以及脚本的URL协议头部分(http:、https:),不使用这两个协议的文件则不省略。省略协议头,即让URL变成相对地址,可以避免协议混合及小文件重复下载。

```
<!-- 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);  
}
```

HTML 格式规则

常规格式化

对每个块、列表、表格元素都另起一行,每个子元素都缩进。

每个块元素、列表元素或表格元素另起一行,而不必考虑元素的样式(因CSS可以改变元素的 display 属性)。

同样的,如果他们是块、列表或者表格元素的子元素,则将之缩进。

(如果你遇到列表项之间有空白的问题,可以把所有 li 元素放到一行。Linter鼓励抛出警告而不是错误。)

```
<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
    </tr>
  <tbody>
    <tr>
      <td>$ 5.00
      <td>$ 4.50
    </tr>
  </tbody>
</table>
```

HTML 引号

当引用属性值时,使用双引号。

使用双引号而不是单引号来包裹属性值。

```
<!-- 不推荐 -->
<a class='maia-button maia-button-secondary'>Sign in</a>
```

```
<!-- 推荐 -->
<a class="maia-button maia-button-secondary">Sign in</a>
```

HTML样式规则

文档类型

使用HTML5。

HTML5(HTML语法)是所有HTML文档的首选：`<!DOCTYPE html>`。

(推荐使用HTML,即text/html。不要使用XHTML。XHTML,即 [application/xhtml+xml](http://hixie.ch/advocacy/xhtml) [http://hixie.ch/advocacy/xhtml],缺乏浏览器和基础结构的支持,并且优化的空间比HTML小。)虽然HTML闭合标签没有问题,但是不要自闭合空标签。即写 `
` 而不是 `
`。

HTML合法性

在可能的情况下使用合法的HTML。

使用合法的HTML代码,除非由于文件大小导致的不可达到的性能目标而不能使用。

利用已用工具对合法性进行测试,例如 [W3C HTML validator](http://validator.w3.org/nu/) [http://validator.w3.org/nu/]。

使用合法的HTML是一个可度量的基准质量属性,该属性有助于了解技术需求和约束,从而确保合理的HTML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>
```

语义化

根据HTML的目的使用它。

根据元素(有时被错误的叫做“标签”)被创造的用途使用他们。比如,对标题使用标题元素,对段落使用 p 元素,对锚点使用 a 元素等。

语义化的使用HTML对于可访问性、复用性和代码的高效性等因素非常重要。

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

<!-- 推荐 -->
```

```
<a href="recommendations/">All recommendations</a>
```

多媒体降级

为多媒体提供替代内容。

对于图片、视频、通过 canvas 实现的动画等多媒体来说,确保提供可访问的替代内容。对于图片,可提供有意义的替代文本(alt);对于视频和音频,如有条件可提供对白和字幕。

提供替代内容对辅助功能很重要:没有 alt ,一位盲人用户很难知道一张图片的内容,其他用户可能不能了解视频和音频的内容。(对于 alt 属性会引起冗余的图片和你不打算添加CSS的纯粹装饰性的图片,不用添加替代文本,写成 alt="" 即可。)

```
<!-- 不推荐 -->


<!-- 推荐 -->

```

关注点分离

将结构、表现、行为分离。

严格保持结构(标识),表现(样式),行为(脚本)分离,尽量使三者之间的相互影响最小。

就是说,确保文档和模板只包含HTML,并且HTML只用来表现结构。把任何表现性的东西都移到样式表,任何行为性的东西都移到脚本中。

此外,尽可能少的从文档和模板中引用样式表和脚本来减少三者的相互影响。

结构、表现、行为分离对维护非常重要。更改HTML文档和模板总是比更新样式表和脚本成本更高。

```
<!-- 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>
<center>I can't believe there's no way to control the styling of
  my website without doing everything all over again!</center>

<!-- 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

不要使用**实体引用**。

假设文件、编辑器和团队之间使用相同的**编码**(UTF-8),则没有必要使用例如 `—`、`”`或 `☺` 这样的**实体引用**。

唯一的例外适用于HTML中具有特殊意义的字符(比如`<`和`&`),和控制或者**隐藏**的字符(比如不**换行**空格)。

```
<!-- 不推荐 -->
```

```
The currency symbol for the Euro is &ldquo;&eur;&rdquo; .
```

```
<!-- 推荐 -->
```

```
The currency symbol for the Euro is "€".
```

可选的标签

省略**可选的标签**(可选)。

为了优化文件大小和可扫描,考虑省略**可选标签**。[HTML5规范](http://www.whatwg.org/specs/web-apps/current-work/multipage/syntax.html#syntax-tag-omission) [http://www.whatwg.org/specs/web-apps/current-work/multipage/syntax.html#syntax-tag-omission] 定义了哪些**标签**可以被省略。

(这种方法可能要求一段**宽限期**去建立一个更加广泛的**准则**,因为它和Web开发人员通常所了解的有着**显著**不同。考虑到一致性和**简单性**,最好省略所有**可选标签**。)

```
<!-- 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

为**样式表**和**脚本**省略 type 属性。

引用**样式表**(除非不是使用CSS)和**脚本**(除非不是使用JavaScript)不要使用type属性。

HTML5将 [text/css](http://www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#attr-style-type) [http://www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#attr-style-type] 和 [text/javascript](http://www.whatwg.org/specs/web-apps/current-work/multipage/scripting-1.html#attr-script-type) [http://www.whatwg.org/specs/web-apps/current-work/multipage/scripting-1.html#attr-script-type] 设置为**默认值**,在这种情况下指定type属性并不必要。甚至同样兼容老版本的**浏览器**。

赠言

请与周围保持一致。

如果你正在**编辑代码**,花几分钟时间看看上下文**代码**的格式,确定他们的**编码风格**。如果在上下文**代码**中,**算术运算符**前后有空格,或**注释**前后添加了“**#**”,你也**应该这样做**。

编写这个风格指导的目标是让人们可以**专注于**“我们在讨论什么”而不是“我们该怎么描述”。我们提供了一些通用的**编码规范**,大家就可以基于**这些规范而继续**,但特定情况下的**规范也同样重要**。如果你在一个文件中添加的**代码**看上去跟其他**代码明显**不同,你就把**阅读**此文件的人的节奏打乱了。避免这种情况出现。

背景

在Google的开源项目中,JavaScript是最主要的客户端脚本语言。本指南是使用JavaScript时建议和不建议做法的清单。

Javascript语言规则

var关键字

总是用 var 关键字定义变量。

描述

如果不显式使用 var 关键字定义变量,变量会进入到全局上下文中,可能会和已有的变量发生冲突。另外,如果不使用var声明,很难说变量存在的作用域是哪个(可能在局部作用域里,也可能在document或者window上)。所以,要一直使用 var 关键字定义变量。

常量

- 使用字母全部大写(如 NAMES_LIKE_THIS)的方式命名
- 可以使用 @const 来标记一个常量 指针(指向变量或属性,自身不可变)
- 由于IE的兼容问题,不要使用 [const关键字](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript%2FReference%2FStatements%2Fconst) [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript%2FReference%2FStatements%2Fconst]

描述

常量值

如果一个值是恒定的,它命名中的字母要全部大写(如 CONSTANT_VALUE_CASE)。字母全部大写意味着这个值不可以被改写。

原始类型(number 、 string 、 boolean)是常量值。

对象的表现会更主观一些,当它们没有暴露出变化的时候,应该认为它们是常量。但是这个不是由编译器决定的。

常量指针(变量和属性)

用 @const 注释的变量和属性意味着它是不能更改的。使用const关键字可以保证在编译的时候保持一致。使用 [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript%2FReference%2FStatements%2Fconst) [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const?redirectlocale=en-US&redirectslug=JavaScript%2FReference%2FStatements%2Fconst] 效果相同,但是由于IE的兼容问题,我们不使用const关键字。

另外,不应该修改用 @const 注释的方法。

例子

注意, `@const` 不一定是常量值,但命名类似 `CONSTANT_VALUE_CASE` 的一定是常量指针。

```
/**
 * 以毫秒为单位的超时时长
 * @type {number}
 */
goog.example.TIMEOUT_IN_MILLISECONDS = 60;
```

1分钟60秒永远也不会改变,这是个常量。全部大写的命名意味其为常量值,所以它不能被重写。开源的编译器允许这个符号被重写,这是因为没有 `@const` 标记。

```
/**
 * Map of URL to response string.
 * @const
 */
MyClass.fetchedUrlCache_ = new goog.structs.Map();
```

在这个例子中,指针没有变过,但是值却是可以变化的,所以这里用了驼峰式的命名,而不是全部大写的命名。

分号

一定要使用分号。

依靠语句间隐式的分割,可能会造成细微的调试的问题,千万不要这样做。

很多时候不写分号是很危险的:

```
// 1.
MyClass.prototype.myMethod = function() {
    return 42;
} // 这里没有分号.

(function() {
    // 一些局部作用域中的初始化代码
})();

var x = {
    'i': 1,
    'j': 2
} //没有分号.

// 2. 试着在IE和firefox下做一样的事情.
//没人会这样写代码, 别管他.
[normalVersion, ffVersion][isIE]();

var THINGS_TO_EAT = [apples, oysters, sprayOnCheese] //这里没有分号

// 3. 条件语句
-1 == resultOfOperation() || die();
```

发生了什么?

1. **js错误**。返回42的函数运行了,因为后面有一对括号,而且传入的参数是一个方法,然后返回的

42被调用,导致出错了。

2. 你可能会得到一个“no sush property in undefined”的错误,因为在执行的时候,解释器将会尝试执行 `x[normalVersion, ffVersion][isIE]()` 这个方法。
3. `die` 这个方法只有在 `resultOfOperation()` 是 `NaN` 的时候执行,并且 `THINGS_TO_EAT` 将会被赋值为 `die()` 的结果。

为什么?

js语句要求以分号结尾,除非能够正确地推断分号的位置。在这个例子当中,函数声明、对象和数组字面量被写在了一个语句当中。右括号(`)`、`}`、`]`)不足以证明这条语句已经结束了,如果下一个字符是运算符或者`(`、`{`、`[`,js将不会结束语句。

这个错误让人震惊,所以一定要确保用分号结束语句。

澄清:分号和函数

函数表达式后面要分号结束,但是函数声明就不需要。例如:

```
var foo = function() {  
    return true;  
}; // 这里要分号  
  
function foo() {  
    return true;  
} // 这里不用分号
```

嵌套函数

可以使用。

嵌套函数非常有用,比如在创建持续任务或者隐藏工具方法的时候。可以放心的使用。

块内函数声明

不要使用块内函数声明。

不要这样做:

```
if (x) {  
    function foo() {}  
}
```

虽然大多数脚本引擎支持功能区块内声明,但ECMAScript并未认可(见 [ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm) [http://www.ecma-international.org/publications/standards/Ecma-262.htm],第13条和第14)。若与他人的及EcmaScript所建议的不一致,即可视为不好的实现方式。ECMAScript只允许函数声明语句列表,在根语句列表脚本或者函数。相反,使用一个变量初始化函数表达式在块内定义一个函数块:

```
if (x) {  
    var foo = function() {}  
}
```

```
}
```

异常

可以抛出异常。

如果你做一些比较复杂的项目你基本上无法避免异常,比如使用一个应用程序开发框架。可以大胆试一试。

自定义异常

可以自定义异常。

如果没有自定义异常,返回的错误信息来自一个有返回值的函数是难处理的,是不雅的。坏解决方案包括传递引用的类型来保存错误信息或总是返回有一个潜在的错误成员的对象。这些基本上为原始的异常处理hack。在适当的时候使用自定义的异常。

标准功能

总是优先于非标准功能。

为了最大的可移植性和兼容性,总是使用标准功能而不是非标准功能(例如,采用 `string.charAt(3)` 而非 `string[3]` ,用DOM的功能访问元素而不是使用特定于一个具体应用的简写)。

原始类型的包装对象

没有理由使用原始类型的包装对象,更何况他们是危险的:

```
var x = new Boolean(false);
if (x) {
    alert('hi'); //显示“hi”。
}
```

不要这样做!

然而类型转换是可以的。

```
var x = Boolean(0);
if (x) {
    alert('hi'); //永远都不显示。
}
typeof Boolean(0) == 'boolean';
typeof new Boolean(0) == 'object';
```

这是非常有用的进行数字、字符串和布尔值转换的方式。

多重的原型继承

不可取。

多重原型继承是Javascript实现继承的方式。如果你有一个以用户定义的class B作为原型的用户自定义class D,则得到多重原型继承。这样的继承出现容易但难以正确创造！

出于这个原因,最好是使用 [Closure库](https://developers.google.com/closure/library/?csw=1) [https://developers.google.com/closure/library/?csw=1] 中的 goog.inherits() 或类似的东西。

```
function D() {
    goog.base(this)
}
goog.inherits( D, B );

D.prototype.method =function() {
    ...
};
```

方法和属性定义

```
/**构造函数*/ function SomeConstructor() { this.someProperty = 1; }
Foo.prototype.someMethod = function() { ... };
```

虽然有多种使用“new”关键词来创建对象方法和属性的途径,首选的创建方法的途径是:

```
Foo.prototype.bar = function() {
    /* ... */
};
```

其他特性的首选创建方式是在构造函数中初始化字段:

```
/** @constructor */
function Foo() {
    this.bar = value;
}
```

为什么？

当前的JavaScript引擎优化基于一个对象的“形状”，[给对象添加一个属性\(包括覆盖原型设置的值\)改变了形式,会降低性能](https://developers.google.com/v8/design#prop_access) [https://developers.google.com/v8/design#prop_access] 。

删除

请使用 this.foo = null 。

```
o.prototype.dispose = function() {
    this.property_ = null;
};
```

而不是:

```
Foo.prototype.dispose = function() {
    delete this.property_;
```



```
};
```

在现代的JavaScript引擎中,改变一个对象属性的数量比重新分配值慢得多。应该避免删除关键字,除非有必要从一个对象的迭代的關鍵字列表删除一个属性,或改变 `if (key in obj)` 结果。

闭包

可以使用,但是要小心。

创建闭包可能是JS最有用的和经常被忽视的功能。在 [这里](http://jibbering.com/faq/notes/closures/) [http://jibbering.com/faq/notes/closures/] 很好地描述说明了闭包的工作。

要记住的一件事情,一个闭包的指针指向包含它的范围。因此,附加一个闭包的DOM元素,可以创建一个循环引用,所以,内存会泄漏。例如,下面的代码:

```
function foo(element, a, b) {
    element.onclick = function() { /* 使用 a 和 b */ };
}
```

闭包能保持元素a和b的引用即使它从未使用。因为元素还保持对闭包的一个引用,我们有一个循环引用,不会被垃圾收集清理。在这些情况下,代码的结构可以如下:

```
function foo(element, a, b) {
    element.onclick = bar(a, b);
}

function bar(a, b) {
    return function() { /* 使用 a 和 b */ }
}
```

eval()函数

只用于反序列化(如评估RPC响应)。

若用于 `eval()` 的字符串含有用户输入,则 `eval()` 会造成混乱的语义,使用它有风险。通常有一个更好 更清晰、更安全的方式来编写你的代码,所以一般是不会允许其使用的。然而,eval相对比非eval使反序列化更容易,因此它的使用是可以接受的(例如评估RPC响应)。

反序列化是将一系列字节存到内存中的数据结构转化过程。例如,你可能会写的对象是:

```
users = [
  {
    name: 'Eric',
    id: 37824,
    email: 'jellyvore@myway.com'
  },
  {
    name: 'xtof',
    id: 31337,
    email: 'b4d455h4x0r@google.com'
  },
  ...
]
```

```
];
```

将这些数据读入内存跟得出文件的字符串表示形式一样容易。

同样, `eval()` 函数可以简化解码RPC的返回值。例如,您可以使用 `XMLHttpRequest` 生成RPC,在响应时服务器返回JavaScript:

```
var userOnline = false;
var user = 'nusrat';
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://chat.google.com/isUserOnline?user=' + user, false);
xmlhttp.send('');
// 服务器返回:
// userOnline = true;
if (xmlhttp.status == 200) {
    eval(xmlhttp.responseText);
}
// userOnline 现在为 true
```

with() {}

不建议使用。

使用 `with` 会影响程序的语义。因为 `with` 的目标对象可能会含有和局部变量冲突的属性,使你程序的语义发生很大的变化。例如,这是做什么用?

```
with (foo) {
    var x = 3;
    return x;
}
```

答案:什么都有可能。局部变量 `x` 可能会被 `foo` 的一个属性覆盖,它甚至可能有setter方法,在此情况下将其赋值为3可能会执行很多其他代码。不要使用 `with`。

this

只在构造函数对象、方法,和创建闭包的时候使用。

`this` 的语义可能会非常诡异。有时它指向全局对象(很多时候)、调用者的作用域链(在 `eval` 里)、DOM树的一个节点(当使用HTML属性来做为事件句柄时)、新创建的对象(在一个构造函数中)、或者其他对象(如果函数被 `call()` 或 `apply()` 方式调用)。

正因为 `this` 很容易被弄错,故将其使用限制在以下必须的地方:

- 在构造函数中
- 在对象的方法中(包括闭包的创建)

for-in 循环

只使用在对象、映射、哈希的键值迭代中。

`for-in` 循环经常被不正确的用在元素数组的循环中。由于并不是从 0 到 `length-1` 进行循环,而是遍历对象中和它原型链上的所有的键,所以很容易出错。这里有一些失败的例子:

```
function printArray(arr) {
    for (var key in arr) {
        print(arr[key]);
    }
}

printArray([0,1,2,3]); //这样可以

var a = new Array(10);
printArray(a); //这样不行

a = document.getElementsByTagName('*');
printArray(a); //这样不行

a = [0,1,2,3];
a.buhu = 'wine';
printArray(a); //这样不行

a = new Array;
a[3] = 3;
printArray(a); //这样不行
```

在数组循环时常用的一般方式:

```
function printArray(arr) {
    var l = arr.length;
    for (var i = 0; i < l; i++) {
        print(arr[i]);
    }
}
```

关联数组

不要将映射,哈希,关联数组当作一般数组来使用。

不允许使用关联数组.....确切的说在数组,你不可以使用非数字的索引。如果你需要一个映射或者哈希,在这种情况下你应该使用对象来代替数组,因为在功能上你真正需要的是对象的特性而不是数组的。

数组仅仅是用来拓展对象的(像在JS中你曾经使用过的 `Date`、`RegExp` 和 `String` 对象一样的)。

多行的字符串字面量

不要使用。

不要这样做:

```
var myString = 'A rather long string of English text, an error message \
    actually that just keeps going and going -- an error \
    message to make the Energizer bunny blush (right through \
```

```
those Schwarzenegger shades)! Where was I? Oh yes, \
you've got an error and all the extraneous whitespace is \
just gravy. Have a nice day.';
```

在编译时每一行头部的空白符不会被安全地去除掉;斜线后的空格也会导致棘手的问题;虽然大部分脚本引擎都会支持,但是它不是ECMAScript规范的一部分。

使用字符串连接来代替:

```
var myString = 'A rather long string of English text, an error message ' +
  'actually that just keeps going and going -- an error ' +
  'message to make the Energizer bunny blush (right through ' +
  'those Schwarzenegger shades)! Where was I? Oh yes, ' +
  'you've got an error and all the extraneous whitespace is ' +
  'just gravy. Have a nice day.';
```

数组和对象字面量

建议使用。

使用数组和对象字面量来代替数组和对象构造函数。

数组构造函数容易在参数上出错。

```
// 长度为3
var a1 = new Array(x1, x2, x3);

// 长度为 2
var a2 = new Array(x1, x2);

// If x1 is a number and it is a natural number the length will be x1.
// If x1 is a number but not a natural number this will throw an exception.
// Otherwise the array will have one element with x1 as its value.
var a3 = new Array(x1);

// 长度为0
var a4 = new Array();
```

由此,如果有人将代码从2个参数变成了一个参数,那么这个数组就会有一个错误的长度。

为了避免这种怪异的情况,永远使用可读性更好的数组字面量。

```
var a = [x1, x2, x3];
var a2 = [x1, x2];
var a3 = [x1];
var a4 = [];
```

对象构造函数虽然没有相同的问题,但是对于可读性和一致性,还是应该使用对象字面量。

```
var o = new Object();

var o2 = new Object();
o2.a = 0;
o2.b = 1;
o2.c = 2;
```

```
o2['strange key'] = 3;
```

应该写成：

```
var o = {};  
var o2 = {  
  a: 0,  
  b: 1,  
  c: 2,  
  'strange key': 3  
};
```

修改内置对象原型

不建议。

强烈禁止修改如 `Object.prototype` 和 `Array.prototype` 等对象的原型。修改其他内置原型如 `Function.prototype` 危险性较小,但在生产环境中还是会引发一些难以调试的问题,也应当避免。

Internet Explorer 中的条件注释

不要使用。

不要这样做：

```
var f = function () {  
  /*@cc_on if (@_jscript) { return 2* @*/ 3; /*@ } @*/  
};
```

条件注释会在运行时改变JavaScript语法树,阻碍自动化工具。

Javascript格式规则

命名

通常来说,使用 `functionNamesLikeThis` , `variableNamesLikeThis` , `ClassNamesLikeThis` , `EnumNamesLikeThis` , `methodNamesLikeThis` , `CONSTANT_VALUES_LIKE_THIS` , `foo.namespaceNamesLikeThis.bar` 和 `filenameslikethis.js` 这种格式的命名方式。

属性和方法

- 私有 属性和方法应该以下划线开头命名。
- 保护 属性和方法应该以无下划线开头命名(像公共属性和方法一样)。

了解更多关于私有成员和保护成员的信息,请阅读 [可见性](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#Visibility__private_and_protected_fields_) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#Visibility__private_and_protected_fields_] 部分。

方法和函数参数

可选函数参数以 `opt_` 开头。

参数数目可变的函数应该具有以 `var_args` 命名的最后一个参数。你可能不会在代码里引用 `var_args` ;使用 `arguments` 对象。

可选参数和数目可变的参数也可以在注释 `@param` 中指定。尽管这两种惯例都被编译器接受,但更加推荐两者一起使用。

getter和setter

EcmaScript 5 不鼓励使用属性的getter和setter。然而,如果使用它们,那么getter就不要改变属性的可见状态。

```
/**
 * 错误 - 不要这样做。
 */
var foo = { get next() { return this.nextId++; } };
};
```

存取函数

属性的getter和setter方法不是必需的。然而,如果使用它们,那么读取方法必须以 `getFoo()` 命名,并且写入方法必须以 `setFoo(value)` 命名。(对于布尔型的读取方法,也可以使用 `isFoo()` ,并且这样往往听起来更自然。)

命名空间

JavaScript没有原生的对封装和命名空间的支持。

全局命名冲突难以调试,并且当两个项目尝试整合的时候可能引起棘手的问题。为了能共享共用的JavaScript代码,我们采用一些约定来避免冲突。

为全局代码使用命名空间

在全局范围内总是使用唯一的项目或库相关的伪命名空间进行前缀标识。如果你正在进行“Project Sloth”项目,一个合理的伪命名空间为 `sloth.*`。

```
var sloth = {};  
  
sloth.sleep = function() {  
    ...  
};
```

很多JavaScript库,包括 [the Closure Library](https://developers.google.com/closure/library/?csw=1) [https://developers.google.com/closure/library/?csw=1] 和 [Dojo toolkit](http://dojotoolkit.org/) [http://dojotoolkit.org/] 给你高级功能来声明命名空间。保持你的命名空间声明一致。

```
goog.provide('sloth');  
  
sloth.sleep = function() {  
    ...  
};
```

尊重命名空间所有权

当选择一个子命名空间的时候,确保父命名空间知道你在做什么。如果你开始了一个为sloths创建hats的项目,确保Sloth这一组命名空间知道你在使用 `sloth.hats`。

外部代码和内部代码使用不同的命名空间

“外部代码”指的是来自你的代码库外并独立编译的代码。内部名称和外部名称应该严格区分开。如果你正在使用一个能调用 `foo.hats.*` 中的东西的外部库,你的内部代码不应该定义 `foo.hats.*` 中的所有符号,因为如果其他团队定义新符号就会把它打破。

```
foo.require('foo.hats');  
/**  
 * 错误 - 不要这样做。  
 * @constructor  
 * @extends {foo.hats.RoundHat}  
 */  
foo.hats.BowlerHat = function() {  
};
```

如果你在外部的命名空间中需要定义新的API,那么你应该明确地导出且仅导出公共的API函数。为了一致性和编译器更好的优化你的内部代码,你的内部代码应该使用内部API的内部名称调用它们。

```
foo.provide('googleyhats.BowlerHat');
```

```

foo.require('foo.hats');
/**
 * @constructor
 * @extends {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
    ...
};
goog.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);

```

为长类型的名称提供别名提高可读性

如果对完全合格的类型使用本地别名可以提高可读性,那么就这样做。本地别名的名称应该符合类型的最后一部分。

```

/**
 * @constructor
 */
some.long.namespace.MyClass = function() {
};

/**
 * @param {some.long.namespace.MyClass} a
 */
some.long.namespace.MyClass.staticHelper = function(a) {
    ...
};

myapp.main = function() {
    var MyClass = some.long.namespace.MyClass;
    var staticHelper = some.long.namespace.MyClass.staticHelper;
    staticHelper(new MyClass());
};

```

不要为命名空间起本地别名。命名空间应该只能使用 [goog.scope](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#goog-scope) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#goog-scope] 命名别名。

```

myapp.main = function() {
    var namespace = some.long.namespace;
    namespace.MyClass.staticHelper(new namespace.MyClass());
};

```

避免访问一个别名类型的属性,除非它是一个枚举。

```

/** @enum {string} */
some.long.namespace.Fruit = {
    APPLE: 'a',
    BANANA: 'b'
};

myapp.main = function() {
    var Fruit = some.long.namespace.Fruit;
    switch (fruit) {
        case Fruit.APPLE:
            ...
        case Fruit.BANANA:
            ...
    }
};

```



```
}  
};  
  
myapp.main = function() {  
    var MyClass = some.long.namespace.MyClass;  
    MyClass.staticHelper(null);  
};
```

永远不要在全局环境中创建别名。只在函数体内使用它们。

文件名

为了避免在大小写敏感的平台引起混淆,文件名**应该**小写。文件名**应该**以 `.js` 结尾,并且**应该**不包含除了 `-` 或 `_` (相比较 `_` 更推荐 `-`) 以外的其它标点。

自定义 `toString()` 方法

必须确保**无误**,并且无其他副作用。

你可以通过自定义 `toString()` 方法来控制对象如何字符串化他们自己。**这没问题**,但是你必须确保你的方法**执行无误**,并且无其他副作用。如果你的方法没有达到**这个**要求,就会很容易产生**严重的问题**。比如,如果 `toString()` 方法调用一个方法产生一个断言,断言可能要输出对象的名称,就又需要调用 `toString()` 方法。

延时初始化

可以使用。

并不总在变量声明的地方就**进行**变量初始化,所以延时初始化是可行的。

明确作用域

时常。

经常使用明确的作用域加强可移植性和清晰度。例如,在作用域链中不要**依赖** `window`。你可能想在其他应用中使用你的函数,这时此 `window` 就非彼 `window` 了。

代码格式

我们原则上遵循 [C++格式规范](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Formatting) [http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Formatting], 并且**进行**以下**额外**的说明。

大括号

由于**隐含**分号的插入,无论大括号括起来的是**什么**,总是在**同一行**上开始你的大括号。例如:

```
if (something) {
```

```
// ...
} else {
  // ...
}
```

数组和对象初始化表达式

当单行数组和对象初始化表达式可以在一行写开时,写成单行是允许的。

```
var arr = [1, 2, 3]; //之后无空格[或之前]
var obj = {a: 1, b: 2, c: 3}; //之后无空格[或之前]
```

多行数组和对象初始化表达式缩进两个空格,括号的处理就像块一样单独成行。

```
//对象初始化表达式
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};

//数组初始化表达式
this.rows_ = [
  "Slartibartfast" <fjordmaster@magrathea.com>',
  "Zaphod Beeblebrox" <theprez@universe.gov>',
  "Ford Prefect" <ford@theguide.com>',
  "Arthur Dent" <has.no.tea@gmail.com>',
  "Marvin the Paranoid Android" <marv@googlemail.com>',
  'the.mice@magrathea.com'
];

//在方法调用中使用
goog.dom.createDom(goog.dom.TagName.DIV, {
  id: 'foo',
  className: 'some-css-class',
  style: 'display:none'
}, 'Hello, world!');
```

长标识符或值在对齐的初始化列表中存在**问题**,所以初始化**值**不必**对齐**。例如:

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
  lengthyName: 2
};
```

不要像这样:

```
WRONG_Object.prototype = {
  a      : 0,
  b      : 1,
  lengthyName: 2
};
```

函数参数

如果可能,应该在同一行上列出所有函数参数。如果这样做将超出每行80个字符的限制,参数必须以一种可读性较好的方式进行换行。为了节省空间,在每一行你可以尽可能的接近80个字符,或者把每一个参数单独放在一行以提高可读性。缩进可能是四个空格,或者和括号对齐。下面是最常见的参数换行形式:

```
// 四个空格, 每行包括80个字符。适用于非常长的函数名,
// 重命名不需要重新缩进, 占用空间小。
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
};

//四个空格, 每行一个参数。适用于长函数名,
// 允许重命名, 并且强调每一个参数。
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
    // ...
};

// 缩进和括号对齐, 每行80字符。 看上去是分组的参数,
// 占用空间小。
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
}

// 和括号对齐, 每行一个参数。看上去是分组的并且
// 强调每个单独的参数。
function bar(veryDescriptiveArgumentNumberOne,
             veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy,
             artichokeDescriptorAdapterIterator) {
    // ...
}
```

当函数调用本身缩进,你可以自由地开始相对于原始声明的开头或者相对于当前函数调用的开头,进行4个空格的缩进。以下都是可接受的缩进风格。

```
if (veryLongFunctionNameA(
    veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
    veryLongFunctionNameC(veryLongFunctionNameD(
        veryLongFunctionNameE(
            veryLongFunctionNameF)));
}
```

匿名函数传递

当在一个函数的参数列表中声明一个匿名函数时,函数体应该与声明的左边缘缩进两个空格,或者与function关键字的左边缘缩进两个空格。这是为了匿名函数体更加可读(即不被挤在屏幕的右侧)。

```

prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

var names = prefix.something.myExcellentMapFunction(
  verboselyNamedCollectionOfItems,
  function(item) {
    return item.name;
  });

```

使用goog.scope命名别名

[goog.scope](https://docs.google.com/document/d/1ETFAuh2kaXMVL-vafUYhaWlhl6b5D9TOvboVg7Zl68Y/pub) [https://docs.google.com/document/d/1ETFAuh2kaXMVL-vafUYhaWlhl6b5D9TOvboVg7Zl68Y/pub] 可用于在使用 [the Closure Library](https://developers.google.com/closure/library/?csw=1) [https://developers.google.com/closure/library/?csw=1] 的工程中缩短命名空间的符号引用。

每个文件只能添加一个 goog.scope 调用。始终将它放在全局范围内。

开放的 goog.scope(function() { 调用必须在之前有一个空行,并且紧跟 goog.provide 声明、goog.require 声明或者顶层的注释。调用必须在文件的最后一行闭合。在scope声明闭合处追加 // goog.scope。注释与分号间隔两个空格。

和C++命名空间相似,不要在 goog.scope 声明下面缩进。相反,从第0列开始。

只取不会重新分配给另一个对象(例如大多数的构造函数、枚举和命名空间)的别名。不要这样做:

```

goog.scope(function() {
var Button = goog.ui.Button;

Button = function() { ... };
...

```

别名必须和全局中的命名的最后一个属性相同。

```

goog.provide('my.module');

goog.require('goog.dom');
goog.require('goog.ui.Button');

goog.scope(function() {
var Button = goog.ui.Button;
var dom = goog.dom;

// Alias new types after the constructor declaration.
my.module.SomeType = function() { ... };
var SomeType = my.module.SomeType;

// Declare methods on the prototype as usual:
SomeType.prototype.findButton = function() {
  // Button as aliased above.
  this.button = new Button(dom.getElement('my-button'));
};

```

```
};  
...  
}); // goog.scope
```

更多的缩进

事实上,除了 [初始化数组和对象](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#Array_and_Object_literals) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#Array_and_Object_literals] 和传递匿名函数外,所有被拆开的多行文本应与之前的表达式左对齐,或者以4个(而不是2个)空格作为一缩进层次。

```
someWonderfulHtml = '' +  
    getEvenMoreHtml(someReallyInterestingValues, moreValues,  
        evenMoreParams, 'a duck', true, 72,  
        slightlyMoreMonkeys(0xffff)) +  
    '';  
  
thisIsAVeryLongVariableName =  
    hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();  
  
thisIsAVeryLongVariableName = 'expressionPartOne' + someMethodThatIsLong() +  
    thisIsAnEvenLongerOtherFunctionNameThatCannotBeIndentedMore();  
  
someValue = this.foo(  
    shortArg,  
    'Some really long string arg - this is a pretty common case, actually.',  
    shorty2,  
    this.bar());  
  
if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&  
    !ambientNotification.isActive() && (client.isAmbientSupported() ||  
        client.alwaysTryAmbientAnyways())) {  
    ambientNotification.activate();  
}
```

空行

使用新的空行来划分一组逻辑上相关联的代码片段。例如：

```
doSomethingTo(x);  
doSomethingElseTo(x);  
andThen(x);  
  
nowDoSomethingWith(y);  
  
andNowWith(z);
```

二元和三元操作符

操作符始终跟随着前行, 这样你就不用顾虑分号的隐式插入问题。否则换行符和缩进还是遵循其他谷歌规范指南。

```
var x = a ? b : c; // All on one line if it will fit.  
  
// Indentation +4 is OK.  
var y = a ?
```

```
longButSimpleOperandB : longButSimpleOperandC;
```

```
// Indenting to the line position of the first operand is also OK.  
var z = a ?  
    moreComplicatedB :  
    moreComplicatedC;
```

点号也应如此处理。

```
var x = foo.bar().  
    doSomething().  
    doSomethingElse();
```

括号

只用在有需要的地方。

通常只在语法或者语义需要的地方有节制地使用。

绝对不要对一元运算符如 `delete`、`typeof` 和 `void` 使用括号或者在关键词如 `return`、`throw` 和其他的(`case`、`in` 或者 `new`)之后使用括号。

字符串

使用 `'` 代替 `"`。

使用单引号(`'`)代替双引号(`"`)来保证一致性。当我们创建包含有HTML的字符串时这样做很有帮助。

```
var msg = 'This is some HTML';
```

可见性(私有和保护类型字段)

鼓励使用 `@private` 和 `@protected` JSDoc注释。

我们建议使用JSDoc注释 `@private` 和 `@protected` 来标识出类、函数和属性的可见程度。

设置 `--jscomp_warning=visibility` 可令编译器对可见性的违规进行编译器警告。可见 [封闭的编译器警告](https://code.google.com/p/closure-compiler/wiki/Warnings) [https://code.google.com/p/closure-compiler/wiki/Warnings] 。

加了 `@private` 标记的全局变量和函数只能被同一文件中的代码所访问。

被标记为 `@private` 的构造函数只能被同一文件中的代码或者它们的静态和实例成员实例化。`@private` 标记的构造函数可以被相同文件内它们的公共静态属性和 `instanceof` 运算符访问。

全局变量、函数和构造函数不能注释 `@protected` 。

```
// 文件1  
// AA_PrivateClass_ 和 AA_init_ 是全局的并且在同一个文件中所以能被访问
```

```

/**
 * @private
 * @constructor
 */
AA_PrivateClass_ = function() {
};

/** @private */
function AA_init_() {
    return new AA_PrivateClass_();
}

AA_init_();

```

标记 `@private` 的属性可以被同一文件中的所有的代码访问,如果属性属于一个类,那么所有自身含有属性的类的静态方法和实例方法也可访问。它们不能被不同文件下的子类访问或者重写。

标记 `@protected` 的属性可以被同一文件中的所有的代码访问,任何含有属性的子类的静态方法和实例方法也可访问。

注意这些语义和C++、JAVA中`private` 和 `protected`的不同,其许可同一文件中的所有代码访问的权限,而不是仅仅局限于同一类或者同一类层次。此外,不像C++中,子类不可重写私有属性。

```

// File 1.

/** @constructor */
AA_PublicClass = function() {
    /** @private */
    this.privateProp_ = 2;

    /** @protected */
    this.protectedProp = 4;
};

/** @private */
AA_PublicClass.staticPrivateProp_ = 1;

/** @protected */
AA_PublicClass.staticProtectedProp = 31;

/** @private */
AA_PublicClass.prototype.privateMethod_ = function() {};

/** @protected */
AA_PublicClass.prototype.protectedMethod = function() {};

// File 2.

/**
 * @return {number} The number of ducks we've arranged in a row.
 */
AA_PublicClass.prototype.method = function() {
    // Legal accesses of these two properties.
    return this.privateProp_ + AA_PublicClass.staticPrivateProp_;
};

// File 3.

```



```
/**
 * @constructor
 * @extends {AA_PublicClass}
 */
AA_SubClass = function() {
  // Legal access of a protected static property.
  AA_PublicClass.staticProtectedProp = this.method();
};
goog.inherits(AA_SubClass, AA_PublicClass);

/**
 * @return {number} The number of ducks we've arranged in a row.
 */
AA_SubClass.prototype.method = function() {
  // Legal access of a protected instance property.
  return this.protectedProp;
};
```

注意在Javascript中,一个类(如 `AA_PrivateClass_`)和其构造函数类型是没有区别的。没办法确定一种类型是public而它的构造函数是private。(因为构造函数很容易重命名从而躲避隐私检查)。

JavaScript类型

鼓励和强制执行的编译器。

JSDoc记录类型时,要尽可能具体和准确。我们支持的类型是基于 [EcmaScript 4规范](http://wiki.ecmascript.org/doku.php?id=spec:spec) [http://wiki.ecmascript.org/doku.php?id=spec:spec] 。

JavaScript类型语言

ES4提案包含指定JavaScript类型的语言。我们使用JsDoc这种语言表达函数参数和返回值的类型。

随着ES4提议的发展,这种语言已经改变了。编译器仍然支持旧的语法类型,但这些语法已经被弃用了。

| 语法名称 | 语法 | 描述 | 弃用语法 |
|------|---|---|------|
| 原始类型 | 在JavaScript中有5种原始类型： {null} , {undefined} , {boolean} , {number} ,和 {string} | 类型的名称。 | |
| | {Object} 实例对象或空。 | 一个实例构造函数或接口函数。 构造函数是 @constructor JSDoc 标记定义的函数。接口函数是 @interface JSDoc标记定义的函 数。 | |
| 实例类型 | {Function} 一个实例函数或 空。 | | |
| | {EventTarget} 构造函数实现的 EventTarget接口,或者为null 的一个实例。 | 默认情况下,实例类型将接受空。 这是唯一的类型语法,使得类型 为空。此表中的其他类型的语法 | |

不会接受空。

枚举类型

`{goog.events.EventType}` 字面量初始化对象的属性之一 `goog.events.EventType`。

一个枚举必须被初始化为一个字面量对象,或作为另一个枚举的别名,加注 `@enum JSDoc` 标记。这个属性是枚举实例。[下面](#) 是枚举语法的定义。

请注意,这是我们的类型系统中为数不多的ES4规范以外的事情之一。

应用类型

`{Array.<string>}` 字符串数组。

`{Object.<string, number>}` 一个对象,其中键是字符串,值是数字。

参数化类型,该类型应用一组参数类型。这个想法是类似于Java泛型。

联合类型

`{(number|boolean)}` 一个数字或布尔值。

表明一个值可能有A型或B型。

括号在顶层表达式可以省略,但在子表达式不能省略,以避免歧义。

`{number|boolean}`

`{function(): (number|boolean)}`

`{(number, boolean)}`
,
`{(number||boolean)}`

可为空的类型

`{?number}`
一个数字或空。

空类型与任意其他类型组合的简称。这仅仅是语法糖(syntactic sugar)。

`{number?}`

非空类型

`{!Object}`
一个对象,值非空。

从非空类型中过滤掉null。最常用于实例类型,默认可为空。

`{Object!}`

记录类型

`{{myNum: number, myObject}}`
给定成员类型的匿名类型。

表示该值有指定的类型的成员。在这种情况下, `myNum` 是 `number` 类型而 `myObject` 可为任何类型。

注意花括号是语法类型的一部分。例如,表示一个数组对象有一个 `length` 属性,你可以写 `Array.<{length}>`。

`{function(string, boolean)}`

| | | |
|--|---|---------------------|
| 函数类型 | 一个函数接受两个参数(一个字符串和一个布尔值),并拥有一个未知的返回值。 | 指定一个函数。 |
| 函数 | {function(): number} | |
| 返回类型 | 一个函数没有参数并返回一个数字。 | 指定函数的返回类型。 |
| 函数this类型 | {function(this:goog.ui.Menu, string)} | |
| | 一个需要一个参数(字符串)的函数,执行上下文是 goog.ui.Menu | 指定函数类型的上下文类型。 |
| 函数new 类型 | {function(new:goog.ui.Menu, string)} | |
| | 一个构造函数接受一个参数(一个字符串),并在使用“new”关键字时创建一个 goog.ui.Menu 新实例。 | 指定构造函数所构造的类型。 |
| 可变参数 | {function(string, ... [number]): number} | |
| | 一个函数,它接受一个参数(一个字符串),然后一个可变数目的参数,必须是数字。 | 指定函数的变量参数。 |
| 可变参数(@param 注释) | @param {...number} var_args 带注释函数的可变数目参数。 | 指定带注释函数接受一个可变数目的参数。 |
| 函数 可选参数 | {function(?string=, number=)} | |
| | 一个函数,它接受一个可选的、可以为空的字符串和一个可选的数字作为参数。“=”只用于函数类型声明。 | 指定函数的可选参数。 |
| 函数 可选参数 (@param 注释) | @param {number=} opt_argument number 类型的可选参数。 | 指定带注释函数接受一个可选的参数。 |
| 所有类型 | {*} | 表明该变量可以接受任何类型。 |

未知类型

{?}

表明该变量可以接受任何类型，编译器不应该检查其类型。

JavaScript中的类型

| 类型举例 | 取值举例 | 描述 |
|----------------|---|---------------------------|
| number | <pre>1 1.0 -5 1e5 Math.PI</pre> | |
| Number | <pre>new Number(true)</pre> | Number对象 |
| string | <pre>'Hello' "World" String(42)</pre> | 字符串 |
| String | <pre>new String('Hello') new String(42)</pre> | String对象 |
| boolean | <pre>true false Boolean(0)</pre> | Boolean值 |
| Boolean | <pre>new Boolean(true)</pre> | Boolean对象 |
| RegExp | <pre>new RegExp('hello') /world/g</pre> | |
| Date | <pre>new Date new Date()</pre> | |
| null | <pre>null</pre> | |
| undefined | <pre>undefined</pre> | |
| void | <pre>function f() { return; }</pre> | 没有返回值 |
| Array | <pre>['foo', 0.3, null] []</pre> | 无类型数组 |
| Array.<number> | <pre>[11, 22, 33]</pre> | 数字数组 |

| | | |
|----------------------------------|---|--|
| Array.<Array.<string>> | <pre>[['one', 'two', 'three'], ['foo', 'bar']]</pre> | 以字符串为元素的数组,作为另一个数组的元素 |
| Object | <pre>{ foo: 'abc', bar: 123, baz: null}</pre> | |
| Object.<string> | <pre>{ 'foo': 'bar' }</pre> | 值为字符串的对象 |
| Object.<number, string> | <pre>var obj = {}; obj[1] = 'bar';</pre> | 键为整数,值为字符串的对象。注意,js当中键总是会隐式转换为字符串。所以 obj['1'] == obj[1]。键在for...in...循环中,总是字符串类型。但在对象中索引时编译器会验证键的类型。 |
| Function | <pre>function(x, y) { return x * y; }</pre> | Function对象 |
| function(number, number): number | <pre>function(x, y) { return x * y; }</pre> | 函数值 |
| 类 | <pre>/** @constructor */ function SomeClass() {} new SomeClass();</pre> | |
| 接口 | <pre>/** @interface */ function SomeInterface() {} SomeInterface.prototype.draw = function() {};</pre> | |
| project.MyClass | <pre>/** @constructor */ project.MyClass = function () {} new project.MyClass()</pre> | |
| project.MyEnum | <pre>/** @enum {string} */ project.MyEnum = { /** The color blue. */ BLUE: '#0000dd', /** The color red. */ RED: '#dd0000' };</pre> | 枚举 JSDoc中枚举的值都是可选的. |

| | | |
|------------------|--|------------|
| Element | <code>document.createElement('div')</code> | DOM元素 |
| Node | <code>document.body.firstChild</code> | DOM节点 |
| HTMLInputElement | <code>htmlDocument.getElementsByTagName('input')[0]</code> | 指明类型的DOM元素 |

类型转换

在类型检测不准确的情况下,有可能需要添加类型的注释,并且把类型转换的表达式写在括号里,括号是必须的。如:

```
/** @type {number} */ (x)
```

可为空与可选的参数和属性

因为Javascript是一个弱类型的语言,明白函数参数、类属性的可选、可为空和未定义之间的细微差别是非常重要的。

对象类型和引用类型默认可为空。如以下表达式:

```
/**
 * 传入值初始化的类
 * @param {Object} value某个值
 * @constructor
 */
function MyClass(value) {
  /**
   * Some value.
   * @type {Object}
   * @private
   */
  this.myValue_ = value;
}
```

告诉编译器 myValue_ 属性为一对象或null。如果 myValue_ 永远都不会为null,就应该如下声明:

```
/**
 * 传入非null值初始化的类
 * @param {!Object} value某个值
 * @constructor
 */
function MyClass(value) {
  /**
   * Some value.
   * @type {!Object}
   * @private
   */
  this.myValue_ = value;
}
```

这样,如果编译器可以识别出 MyClass 初始化传入值为null,就会发出一个警告。

函数的可选参数在运行时可能会是undefined,所以如果他们是类的属性,那么必须声明:

```
/**
 * 传入可选值初始化的类
 * @param {Object=} opt_value某个值 (可选)
 * @constructor
 */
function MyClass(opt_value) {
  /**
   * Some value.
   * @type {Object|undefined}
   * @private
   */
  this.myValue_ = opt_value;
}
```

这告诉编译器 myValue_ 可能是一个对象,或 null ,或 undefined 。

注意: 可选参数 opt_value 被声明成 {Object=} ,而不是 {Object|undefined} 。这是因为可选参数可能是undefined。虽然直接写undefined也并无害处,但鉴于可阅读性还是写成上述的样子。

最后,属性的可为空和可选并不矛盾,下面的四种声明各不相同:

```
/**
 * 接受四个参数, 两个可为空, 两个可选
 * @param {!Object} nonNull 必不为null
 * @param {Object} maybeNull 可为null
 * @param {!Object=} opt_nonNull 可选但必不为null
 * @param {Object=} opt_maybeNull 可选可为null
 */
function strangeButTrue(nonNull, maybeNull, opt_nonNull, opt_maybeNull) {
  // ...
};
```

类型定义

有时类型可以变得复杂。一个函数,它接受一个元素的内容可能看起来像:

```
/**
 * @param {string} tagName
 * @param {(string|Element|Text|Array.<Element>|Array.<Text>)} contents
 * @return {!Element}
 */
goog.createElement = function(tagName, contents) {
  ...
};
```

你可以定义带 @typedef 标记的常用类型表达式。例如:

```
/** @typedef {(string|Element|Text|Array.<Element>|Array.<Text>)} */
goog.ElementContent;

/**
 * @param {string} tagName
 * @param {goog.ElementContent} contents
 * @return {!Element}
```

```
*/
goog.createElement = function(tagName, contents) {
  ...
};
```

模板类型

编译器对模板类型提供有限支持。它只能从字面上通过 `this` 参数的类型和 `this` 参数是否丢失推断匿名函数的 `this` 类型。

```
/**
 * @param {function(this:T, ...)} fn
 * @param {T} thisObj
 * @param {...*} var_args
 * @template T
 */
goog.bind = function(fn, thisObj, var_args) {
  ...
};
//可能出现属性丢失警告
goog.bind(function() { this.someProperty; }, new SomeClass());
//出现this未定义警告
goog.bind(function() { this.someProperty; });
```

注释

使用JSDoc。

我们使用 [c++的注释风格](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Comments) [http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Comments]。所有的文件、类、方法和属性都应该用合适的 [JSDoc](https://code.google.com/p/jsdoc-toolkit/) [https://code.google.com/p/jsdoc-toolkit/] 的 [标签](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#JSDoc_Tag_Reference) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#JSDoc_Tag_Reference] 和 [类型](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#JsTypes) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#JsTypes] 注释。除了直观的方法名称和参数名称外,方法的描述、方法的参数以及方法的返回值也要包含进去。

行内注释应该使用 `//` 的形式。

为了避免出现语句片段,要使用正确的大写单词开头,并使用正确的标点符号作为结束。

注释语法

JSDoc的语法基于 [JavaDoc](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html) [http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html],许多编译工具从JSDoc注释中获取信息从而进行代码验证和优化,所以这些注释必须符合语法规则。

```
/**
 * A JSDoc comment should begin with a slash and 2 asterisks.
 * Inline tags should be enclosed in braces like {@code this}.
 * @desc Block tags should always start on their own line.
 */
```

JSDoc 缩进

如果你不得不进行换行,那么你应该像在代码里那样,使用四个空格进行缩进。

```
/**
 * Illustrates line wrapping for long param/return descriptions.
 * @param {string} foo This is a param with a description too long to fit in
 *   one line.
 * @return {number} This returns something that has a description too long to
 *   fit in one line.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

不必在 @fileoverview 标记中使用缩进。

虽然不建议,但依然可以对描述文字进行排版。

```
/**
 * This is NOT the preferred indentation method.
 * @param {string} foo This is a param with a description too long to fit in
 *   one line.
 * @return {number} This returns something that has a description too long to
 *   fit in one line.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

JSDoc中的HTML

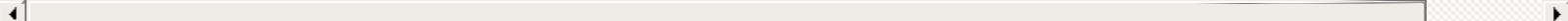
像JavaDoc一样, JSDoc 支持很多的HTML标签,像 <code> , <pre> , <tt> , , , , , <a> 等。

这就意味着不建议采用纯文本的格式。所以,不要在JSDoc里使用空白符进行格式化。

```
/**
 * Computes weight based on three factors:
 *   items sent
 *   items received
 *   last timestamp
 */
```

上面的注释会变成这样:

```
Computes weight based on three factors: items sent items received items received last tim
```



所以,用下面的方式代替:

```
/**
 * Computes weight based on three factors:
 * <ul>
 * <li>items sent
 * <li>items received
 * <li>last timestamp
 * </ul>
```


*/

[JavaDoc](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html) [http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html] 风格指南对于如何编写良好的doc注释是非常有帮助的。

顶层/文件层注释

[版权声明](http://google-styleguide.googlecode.com/svn/trunk/copyright.html) [http://google-styleguide.googlecode.com/svn/trunk/copyright.html] 和作者信息是可选的。顶层注释的目的是为了让不熟悉代码的读者了解文件中有什么。它需要描述文件内容,依赖关系以及兼容性的信息。例如:

```
/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 */
```

Class评论

类必须记录说明与描述和 [一个类型的标签](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#constructor-tag) [http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#constructor-tag] ,标识的构造函数。类必须加以描述,若是构造函数则需标注出。

```
/**
 * Class making something fun and easy.
 * @param {string} arg1 An argument that makes this more interesting.
 * @param {Array.<number>} arg2 List of numbers to be processed.
 * @constructor
 * @extends {goog.Disposable}
 */
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

方法和功能注释

参数和返回类型应该被记录下来。如果方法描述从参数或返回类型的描述中明确可知则可以省略。方法描述应该由一个第三人称表达的句子开始。

```
/**
 * Operates on an instance of MyClass and returns something.
 * @param {project.MyClass} obj Instance of MyClass which leads to a long
 *   comment that needs to be wrapped to two lines.
 * @return {boolean} Whether something occurred.
 */
function PR_someMethod(obj) {
  // ...
}
```

属性评论

```
/** @constructor */
project.MyClass = function() {
```

```

/**
 * Maximum number of things per pane.
 * @type {number}
 */
this.someProperty = 4;
}

```

JSDoc标签参考

你也许在第三方代码中看到其他类型JSDoc注释,这些注释出现在 [JSDoc Toolkit标签的参考](https://code.google.com/p/jsdoc-toolkit/wiki/TagReference) [https://code.google.com/p/jsdoc-toolkit/wiki/TagReference],但目前在谷歌的代码中不鼓励使用。你应该将他们当作“保留”字,他们包括:

- @augments
- @argument
- @borrows
- @class
- @constant
- @constructs
- @default
- @event
- @example
- @field
- @function
- @ignore
- @inner
- @link
- @memberOf
- @name
- @namespace
- @property
- @public
- @requires
- @returns
- @since
- @static
- @version

为goog.provide提供依赖

只提供顶级符号。

一个类上定义的所有成员应该放在一个文件中。所以,在一个在相同类中定义的包含多个成员的文件中只应该提供顶级的类(例如枚举、内部类等)。

要这样写:

```
goog.provide('namespace.MyClass');
```

不要这样写：

```
goog.provide('namespace.MyClass');
goog.provide('namespace.MyClass.Enum');
goog.provide('namespace.MyClass.InnerClass');
goog.provide('namespace.MyClass.TypeDef');
goog.provide('namespace.MyClass.CONSTANT');
goog.provide('namespace.MyClass.staticMethod');
```

命名空间的成员也应该提供：

```
goog.provide('foo.bar');
goog.provide('foo.bar.method');
goog.provide('foo.bar.CONSTANT');
```

编译

必需。

对于所有面向客户的代码来说,使用JS编辑器是必需的,如使用 [Closure Compiler](https://developers.google.com/closure/compiler/?csw=1)

[<https://developers.google.com/closure/compiler/?csw=1>] 。

技巧和诀窍

JavaScript帮助信息

True和False布尔表达式

下边的布尔表达式都返回false：

- null
- undefined
- “”空字符串
- 数字0

但是要小心,因为以下这些返回true：

- 字符串”0”
- []空数组
- {}空对象

下面这样写不好：

```
while (x != null) {
```

你可以写成这种更短的代码(只要你不期望x为0、空字符串或者false)：

```
while (x) {
```

如果你想检查字符串是否为null或空,你可以这样写：

```
if (y != null && y != '') {
```

但是以下这样会更简练更好：

```
if (y) {
```

注意：还有很多不直观的关于布尔表达式的例子，这里是一些：

- Boolean('0') == true '0' != true
- 0 != null 0 == [] 0 == false
- Boolean(null) == false null != true null != false
- Boolean(undefined) == false undefined != true undefined != false
- Boolean([]) == true [] != true [] == false
- Boolean({}) == true {} != true {} != false

条件(三元)操作符(?:)

以下这种写法可以三元操作符替换：

```
if (val != 0) {  
  return foo();  
} else {  
  return bar();  
}
```

你可以这样写来代替：

```
return val ? foo() : bar();
```

三元操作符在生成HTML代码时也是很有用的：

```
var html = '<input type="checkbox"' +  
  (isChecked ? ' checked' : '') +  
  (isEnabled ? '' : ' disabled') +  
  ' name="foo">';
```

&& 和 ||

二元布尔操作符是可短路的，只有在必要时才会计算到最后一项。

“||”被称作为‘default’操作符，因为可以这样：

```
/** @param {*=} opt_win */  
function foo(opt_win) {  
  var win;  
  if (opt_win) {  
    win = opt_win;  
  } else {  
    win = window;  
  }  
  // ...  
}
```

你可以这样写：

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

“&&”也可以用来缩减代码。例如，以下这种写法可以被缩减：

```
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```

你可以这样写：

```
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

或者这样写：

```
var kid = node && node.kids && node.kids[index];
if (kid) {
  foo(kid);
}
```

然而以下这样写就有点过头了：

```
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

遍历节点列表

节点列表是通过给节点迭代器加一个过滤器来实现的。这表示获取他的属性,如length的时间复杂度为O(n),通过length来遍历整个列表需要O(n^2)。

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

这样写更好：

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

这种方法对所有的集合和数组(只要数组不包含被认为是false值的元素) 都适用。

在上面的例子中,你也可以通过firstChild和nextSibling属性来遍历子节点。

```
var parentNode = document.getElementById('foo');  
for (var child = parentNode.firstChild; child; child = child.nextSibling) {  
    doSomething(child);  
}
```