Google 开源项目风格指南

Jun 23, 2024

Contents

CHAPTER 1

Google 开源项目风格指南——中文版

- ReadTheDocs 托管地址：在线阅读最新版本
- GitHub 托管地址：zh-google-styleguide
- 离线文档下载地址：release

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 。

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”，等等诸如此类。

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。

我们已经发布了七份 **中文版**的风格指南:

1. Google C++ 风格指南
2. Google Objective-C 风格指南
3. Google Python 风格指南
4. Google JavaScript 风格指南
5. Google Shell 风格指南

6. Google JSON 风格指南
7. Google TypeScript 风格指南

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 ——一个用来帮助适应风格准则的工具，以及 google-c-style.el，Google 风格的 Emacs 配置文件。
- 另外，招募志愿者翻译 XML Document Format Style Guide ，有意者请联系 Yang.Y 。

CHAPTER 2

C++ 风格指南 - 内容目录

Contents

2.1 扉页

更新时间

2024/04/12

原作者

Benjy Weinberger

Craig Silverstein

Gregory Eitzmann

Mark Mentovai

Tashana Landray

翻译

YuleFox

Yang.Y

acgtyrant

lilinsanity

楼宇

项目主页

- Google Style Guide
- Google 开源项目风格指南 - 中文版

2.1.1 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.
  >
  > 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, 很多地方摘录的也是该版本.

以下是正文.

2.1.2 背景

C++ 是谷歌的开源项目所采用的主要编程语言之一. C++ 程序员都知道, 该语言有很多强大的特性 (feature), 但强大之处也伴随着复杂性, 让代码容易出错, 难以阅读、维护.

本指南的目标是详述 C++ 的注意事项来控制复杂性. 这些规则会在保持代码易于管理的同时, 不影响程序员高效地使用 C++ 的语言特性.

风格 (style, 亦称作可读性 (readability)) 是用于管理 C++ 代码的惯例. “风格”这一术语略有不准确, 因为这些惯例并非仅仅囊括代码格式.

谷歌主导的大部分开源项目遵守本指南的要求.

注意: 本指南并非 C++ 教程, 我们假定读者已经非常熟悉 C++.

2.1.3 本指南的目标

为什么编写这份文档?

我们认为该指南应该实现以下核心目标. 这些目标是每条规则背后的基本 **依据**. 我们希望公开这些想法, 作为讨论的基础, 让广大社区了解每条规则和特定决策背后的来由. 在理解规则所服务的目标以后, 所有人都应该清楚某条规则在哪些情况下可以忽略 (有些规则可以忽略), 以及改变规则时需要提供怎样的论据和替代品.

我们认为风格指南当前的目标如下:

风格规则应该有影响力

> 一条风格规则应该具备足够大的好处, 以至于值得所有工程师铭记. 所谓好处是相对于当前代码库的状态而言的, 所以即使某一习惯十分恶劣, 如果人们很少使用, 那么禁止这一习惯的好处依然很小. 这样可以解释为什么我们没有写下某些规则. 例如, `goto` 语句违背了许多原则, 但是现在已经很少出现, 所以风格指南不会讨论它.

为读者优化, 而非为作者优化

> 我们的代码库 (以及其中的每个组件) 应该会存在很长时间. 因此, 我们读代码的时间比写代码的时间更长. 我们明确地选择优化平均水平的软件工程师阅读、维护和调试代码的体验, 而非编写代码的舒适度. “为读者留下线索”是这一理念的一个方面. 如果代码中有特殊的情况 (例如指针所有权转移), 在此处给读者留下的文字提示很有价值 (在代码中使用 `std::unique_ptr` 就明确地表达了所有权转移).

和现有代码保持一致

> 我们让代码库的风格保持整体一致, 就能聚焦在其他 (更有价值的) 问题上. 一致性也会帮助自动化: 那些格式化代码或者调整 `#include` 顺序的工具, 只能在你的代码符合预期时才能正常工作. 很多时候, 那些用于“保持一致”的规则本质上就是“任选其一并停止内耗”; 在这些问题上, 争论的成本超过了提供自由度的价值. 不过, 一致性原则也有局限性. 在没有清晰的技术性论据和长远方向时, 这才是很好的打破平局的方式. 这一原则适合局部使用 (一个文件内, 或者一组关联性强的接口). 不应该为了一致性而采用旧风格, 忽视新风格的好处. 应该考虑到代码库会随时间推移而过渡到新风格.

恰当时与广大 C++ 社区保持一致

与其他组织保持一致性是有价值的, 这和我们保持内部一致性的原因一样. 如果 C++ 标准中的特性解决了某个问题, 或者某一范式被广泛采用, 这就是采纳它们的依据. 不过, 有时标准的特性和范式有缺陷, 或者在设计上没有考虑我们代码库的需求. 此时 (正如下文所描述的) 应该限制或者禁止这些标准特性. 有时, 相较于 C++ 标准库, 我们偏向于自研库或某些第三方库. 一般这是因为我们所选择的库具有优越性, 或者迁移到标准库的价值不值得那些工作量.

避免使用奇特或危险的语法结构

有些 C++ 的特性比表面上更加奇特或危险. 风格指南中的一些限制就是为了防止掉入这些陷阱. 你需要达到很高的标准才能豁免这些限制, 因为忽略这些规则就很可能直接引起程序错误.

避免使用那些正常水平的 C++ 程序员认为棘手或难以维护的语法结构

有些 C++ 特性会给代码带来复杂性, 因此通常不适合使用. 在用途广泛的代码中, 我们可以接受更巧妙的语法结构. 这是因为复杂的实现方式所带来的收益会被众多使用者放大, 而且在编写新代码时, 也不需要重新解读这些复杂的语法. 如有疑问, 可以请求项目主管豁免这些规则. 这对我们的代码库至关重要, 因为代码负责人和团队成员会变化: 即使所有现在修改这段代码的人都理解代码, 几年后人们就不一定还能理解了.

需要注意我们的规模

我们有上亿行代码和成千上万的工程师, 因此一位工程师的失误或者投机取巧的行为会成为很多人的负担. 举例来说, 一定要避免污染全局命名空间 (global namespace): 如果所有人都往全局命名空间里塞东西, 就很难避免上亿行代码之间的符号冲突 (name collision), 也难以修复冲突.

在必要时为优化让路

即使性能优化的手段会和此文档的其他理念冲突, 有时这些手段也是必要且恰当的.

此文档的目的是提供最大程度的指导和合理限制. 和往常一样, 你应该追随常理和正常审美. 这里我们特指整个谷歌 C++ 社区所建立的规范, 而不是你个人或者所在团队的偏好. 应该对巧妙或奇特的语法结构保持怀疑和犹豫的态度: 并不是“法无禁止即可为”. 运用你的判断力. 如有疑惑, 请不要犹豫, 随时向项目主管咨询意见.

2.1.4 C++ 版本

目前代码的目标版本是 C++20, 所以不应该使用 C++23 的特性. 本指南的 C++ 目标版本会随时间 (激进地) 升级.

禁止使用非标准扩展.

在使用 C++17 和 C++20 的特性之前, 需要权衡其他环境的可移植性.

2.2 1. 头文件

通常每个 .cc 文件应该有一个配套的 .h 文件. 常见的例外情况包括单元测试和仅有 main() 函数的 .cc 文件.

正确使用头文件会大大改善代码的可读性和执行文件的大小、性能.

下面的规则将带你规避头文件的各种误区.

2.2.1 1.1. 自给自足的头文件

Tip: 头文件应该自给自足 (self-contained, 也就是可以独立编译), 并以 .h 为扩展名. 给需要被导入 (include) 但不属于头文件的文件设置为 .inc 扩展名, 并尽量避免使用.

所有头文件应该自给自足, 也就是头文件的使用者和重构工具在导入文件时无需任何特殊的前提条件. 具体来说, 头文件要有头文件防护符 (header guards, *1.2. #define 防护符*), 并导入它所需的所有其它头文件.

若头文件声明了内联函数 (inline function) 或模版 (template), 而且头文件的使用者需要实例化 (instantiate) 这些组件时, 头文件必须直接或通过导入的文件间接提供这些组件的实现 (definition). 不要把这些实现放到另一个头文件里 (例如 -inl.h 文件) 再导入进来; 这是过去的惯例, 但现在被禁止了. 若模版的所有实例化过程都仅出现在一个 .cc 文件中, 比如采用显式 (explicit) 实例化, 或者只有这个 .cc 文件会用到模版定义 (definition), 那么可以把模版的定义放在这个文件里.

在少数情况下, 用于导入的文件不能自给自足. 它们通常是要在特殊的地方导入, 例如另一个文件中间的某处. 此类文件不需要使用头文件防护符, 也不需要导入它的依赖 (prerequisite). 此类文件应该使用 .inc 扩展名. 尽量少用这种文件, 可行时应该采用自给自足的头文件.

2.2.2 1.2. #define 防护符

Tip: 所有头文件都应该用 #define 防护符来防止重复导入. 防护符的格式是: < 项目>_< 路径>_< 文件名>_H_ .

为了保证符号的唯一性, 防护符的名称应该基于该文件在项目目录中的完整文件路径. 例如, foo 项目中的文件 foo/src/bar/baz.h 应该有如下防护:

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

2.2.3 1.3. 导入你的依赖

Tip: 若代码文件或头文件引用了其他地方定义的符号 (symbol), 该文件应该直接导入 (include) 提供该符号的声明 (declaration) 或者定义 (definition) 的头文件. 不应该为了其他原因而导入头文件.

不要依赖间接导入. 这样, 人们删除不再需要的 `#include` 语句时, 才不会影响使用者. 此规则也适用于配套的文件: 若 `foo.cc` 使用了 `bar.h` 的符号, 就需要导入 `bar.h`, 即使 `foo.h` 已经导入了 `bar.h`.

2.2.4 1.4. 前向声明

Tip: 尽量避免使用前向声明. 应该导入你所需的头文件。

定义:

前向声明 (forward declaration) 是没有对应定义 (definition) 的声明.

```
// 在 C++ 源码文件中:
class B;
void FuncInB();
extern int variable_in_b;
ABSL_DECLARE_FLAG(flag_in_b);
```

优点:

- 使用前向声明能节约编译时间, 因为 `#include` 会迫使编译器打开更多的文件并处理更多的输入.
- 使用前向声明能避免不必要的重复编译. 若使用 `#include`, 头文件中无关的改动也会触发重新编译.

缺点:

- 前向声明隐藏了依赖关系, 可能会让人忽略头文件变化后必要的重新编译过程.
- 相比 `#include`, 前向声明的存在会让自动化工具难以发现定义该符号的模块.
- 修改库 (library) 时可能破坏前向声明. 函数或模板的前向声明会阻碍头文件的负责人修改 API, 例如拓宽 (widening) 参数类型, 为模版参数添加默认值, 迁移到新的命名空间等等, 而这些操作本是无碍的.
- 为 `std::` 命名空间的符号提供前向声明会产生未定义行为 (undefined behavior).
- 很难判断什么时候该用前向声明, 什么时候该用 `#include`. 用前向声明代替 `#include` 时, 可能会悄然改变代码的含义:

```
// b.h:
struct B {};
struct D : B {};
```

(continues on next page)

(continued from previous page)

```
// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // 调用 f(B*)
```

若用 B 和 D 的前向声明替代 #include, test() 会调用 f(void*) .

- 为多个符号添加前向声明比直接 #include 更冗长.
- 为兼容前向声明而设计的代码 (比如用指针成员代替对象成员) 更慢更复杂.

结论:

尽量避免为其他项目定义的实体提供前向声明.

2.2.5 1.5. 内联函数

Tip: 只把 10 行以下的小函数定义为内联 (inline).

定义:

你可以通过声明建议编译器展开内联函数, 而不是使用正常的函数调用机制.

优点:

只要内联函数体积较小, 内联函数可以令目标代码 (object code) 更加高效. 我们鼓励对存取函数 (accessors)、变异函数 (mutators) 和其它短小且影响性能的函数使用内联展开.

缺点:

滥用内联将拖慢程序. 根据函数体积, 内联可能会增加或减少代码体积. 通常, 内联展开非常短小的存取函数会减少代码大小, 但内联一个巨大的函数将显著增加代码大小. 在现代处理器上, 通常代码越小执行越快, 因为指令缓存利用率高.

结论:

合理的经验法则是不要内联超过 10 行的函数. 谨慎对待析构函数. 析构函数往往比表面上更长, 因为会暗中调用成员和基类的析构函数!

另一个实用的经验准则: 内联那些有循环或 switch 语句的函数通常得不偿失 (除非这些循环或 switch 语句通常不执行).

应该注意, 即使函数被声明为内联函数, 也不一定真的会被内联; 比如, 虚函数和递归函数通常不会被内联. 一般不应该声明递归函数为内联. (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). 为虚函数声明内联的主要目的是在类 (class) 中定义该函数, 以便于使用该函数或注释其行为. 这常用于存取函数和变异函数.

2.2.6 1.6. #include 的路径及顺序

Tip: 推荐按照以下顺序导入头文件: 配套的头文件, C 语言系统库头文件, C++ 标准库头文件, 其他库的头文件, 本项目的头文件.

头文件的路径应相对于项目源码目录, 不能出现 UNIX 目录别名 (alias) `.` (当前目录) 或 `..` (上级目录). 例如, 应该按如下方式导入 `google-awesome-project/src/base/logging.h`:

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

在 `dir/foo.cc` 或 `dir/foo_test.cc` 这两个实现或测试 `dir2/foo2.h` 内容的文件中, 按如下顺序导入头文件:

1. `dir2/foo2.h`.
2. 空行
3. C 语言系统文件 (确切地说: 用使用方括号和 `.h` 扩展名的头文件), 例如 `<unistd.h>` 和 `<stdlib.h>`.
4. 空行
5. C++ 标准库头文件 (不含扩展名), 例如 `<algorithm>` 和 `<cstddef>`.
6. 空行
7. 其他库的 `.h` 文件.
8. 空行
9. 本项目的 `.h` 文件.

每个非空的分组之间用空行隔开.

这种顺序可以确保在 `dir2/foo2.h` 缺少必要的导入时, 构建 (build) `dir/foo.cc` 或 `dir/foo_test.cc` 会失败. 这样维护这些文件的人会首先发现构建失败, 而不是维护其他库的无辜的人.

`dir/foo.cc` 和 `dir2/foo2.h` 通常位于同一目录下 (如 `base/basictypes_unittest.cc` 和 `base/basictypes.h`), 但有时也放在不同目录下.

注意 C 语言头文件 (如 `stddef.h`) 和对应的 C++ 头文件 (`cstddef`) 是等效的. 两种风格都可以接受, 但是最好和现有代码保持一致.

每个分组内部的导入语句应该按字母序排列. 注意旧代码可能没有遵守这条规则, 应该在方便时进行修正.

举例来说, `google-awesome-project/src/foo/internal/fooserver.cc` 的导入语句如下:

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

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

#include <string>
```

(continues on next page)

(continued from previous page)

```
#include <vector>

#include "base/basictypes.h"
#include "foo/server/bar.h"
#include "third_party/absl/flags/flag.h"
```

例外：

有时平台相关的 (system-specific) 代码需要有条件地导入 (conditional include)，此时可以在其他导入语句后放置条件导入语句. 当然, 尽量保持平台相关的代码简洁且影响范围小. 例如：

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

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

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

2.2.7 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. `-inl.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.

2.2.8 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 `-inl.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，也很好地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。

2.3 2. 作用域

2.3.1 2.1. 命名空间

Tip: 除了少数特殊情况, 应该在命名空间 (namespace) 内放置代码. 命名空间应该有独一无二的名字, 其中包含项目名称, 也可以选择性地包含文件路径. 禁止使用 using 指令 (例如 `using namespace foo`). 禁止使用内联 (inline) 命名空间. 请参见内部链接 中关于匿名命名空间 (unamed namespace) 的内容.

定义:

命名空间可以将全局作用域 (global scope) 划分为独立的、有名字的作用域, 因此可以有效防止全局作用域中的命名冲突 (name collision).

优点:

命名空间可以避免大型程序中的命名冲突, 同时代码可以继续使用简短的名称.

举例来说, 若两个项目的全局作用域中都有一个叫 `Foo` 的类 (class), 这两个符号 (symbol) 会在编译或运行时发生冲突. 如果每个项目在不同的命名空间中放置代码, `project1::Foo` 和 `project2::Foo` 就是截然不同的符号, 不会冲突.

内联命名空间会自动把其中的标识符置入外层作用域, 比如:

```
namespace outer {
inline namespace inner {
    void foo();
}  // namespace inner
}  // namespace outer
```

此时表达式 `outer::inner::foo()` 与 `outer::foo()` 等效. 内联命名空间的主要用途是保持不同 ABI 版本之间的兼容性。

缺点:

命名空间让人难以理解, 因为难以找到一个标识符所对应的定义.

内联命名空间更难理解, 因为其中的标识符不仅仅出现在声明它的命名空间中. 因此内联命名空间只能作为版本控制策略的一部分.

部分情景下, 我们必须多次使用完全限定名称 (fully-qualified name) 来引用符号. 此时多层嵌套的命名空间会让代码冗长.

结论:

建议按如下方法使用命名空间:

- 遵守 命名空间命名 规则.

- 像前面的例子一样, 用注释给命名空间收尾. (译者注: 注明命名空间的名字.)
- 在导入语句、gflags 声明/定义以及其他命名空间的类的前向声明 (forward declaration) 之后, 用命名空间包裹整个源代码文件:

```
// .h 文件
namespace mynamespace {

// 所有声明都位于命名空间中.
// 注意没有缩进.
class MyClass {
    public:
    ...
    void Foo();
};

}  // namespace mynamespace
```

```
// .cc 文件
namespace mynamespace {

// 函数定义位于命名空间中.
void MyClass::Foo() {
    ...
}

}  // namespace mynamespace
```

更复杂的 .cc 文件有更多细节, 比如旗标 (flag) 或 using 声明.

```
#include "a.h"

DEFINE_FLAG(bool, someflag, false, "某个旗标");

namespace mynamespace {

using ::foo::Bar;

... 命名空间内的代码...  // 代码紧贴左边框.

}  // namespace mynamespace
```

- 若要将自动生成的 proto 消息代码放入命名空间, 可以在 .proto 文件中使用 package 修饰符 (specifier). 参见 Protocol Buffer 的包.
- 不要在 std 命名空间内声明任何东西. 不要前向声明 (forward declare) 标准库的类. 在 std 命名空间内声明实体是未定义行为 (undefined behavior), 也就是会损害可移植性. 若要声明标准库的实体, 应该导入对应的头文件.
- 禁止使用 *using* 指令引入命名空间的所有符号。

```
// 禁止: 这会污染命名空间.
using namespace foo;
```

- 除了在明显标注为内部使用的命名空间内, 不要让头文件引入命名空间别名 (namespace alias). 这是因为头文件的命名空间中引入的任何东西都是该文件的公开 API. 正确示例:

```
// 在 .cc 中, 用别名缩略常用的名称.
namespace baz = ::foo::bar::baz;
```

```
// 在 .h 中, 用别名缩略常用的命名空间.
namespace librarian {
namespace impl {  // 仅限内部使用, 不是 API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // 一个函数 (f 或方法) 中的局部别名.
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- 禁止内联命名空间.
- 如果命名空间的名称包含“internal”, 代表用户不应该使用这些 API.

```
// Absl 以外的代码不应该使用这一内部符号.
using ::absl::container_internal::ImplementationDetail;
```

- 我们鼓励新的代码使用单行的嵌套命名空间声明, 但不强制要求.

  译者注: 例如

```
namespace foo::bar {
...
}  // namespace foo::bar
```

2.3.2 2.2. 内部链接

Tip: 若其他文件不需要使用 `.cc` 文件中的定义, 这些定义可以放入匿名命名空间 (unnamed namespace) 或声明为 `static`, 以实现内部链接 (internal linkage). 但是不要在 `.h` 文件中使用这些手段.

定义:

所有放入匿名命名空间中的声明都会内部链接. 声明为 `static` 的函数和变量也会内部链接. 这意味着其他文件不能访问你声明的任何事物. 即使另一个文件声明了一模一样的名称, 这两个实体也都是相互独立的.

结论:

建议 .cc 文件中所有不需要外部使用的代码采用内部链接. 不要在 .h 文件中使用内部链接.

匿名命名空间的声明应与具名命名空间的格式相同. 在末尾的注释中, 不用填写命名空间名称:

```
namespace {
...
}  // namespace
```

2.3.3 2.3. 非成员函数、静态成员函数和全局函数

Tip: 建议将非成员 (nonmember) 函数放入命名空间; 尽量不要使用完全全局的函数 (completely global function). 不要仅仅为了给静态成员 (static member) 分组而使用类 (class). 类的静态方法应当和类的实例或静态数据紧密相关.

优点:

非成员函数和静态成员函数在某些情况下有用. 若将非成员函数放在命名空间内, 不会污染全局命名空间.

缺点:

有时非成员函数和静态成员函数更适合成为一个新的类的成员, 尤其是当它们需要访问外部资源或有明显的依赖关系时.

结论:

有时我们需要定义一个和类的实例无关的函数. 这样的函数可以定义为静态成员函数或非成员函数. 非成员函数不应该依赖外部变量, 且大部分情况下应该位于命名空间中. 不要仅仅为了给静态成员分组而创建一个新类; 这相当于给所有名称添加一个公共前缀, 而这样的分组通常是不必要的.

如果你定义的非成员函数仅供本 .cc 文件使用, 请用内部链接 限制其作用域.

2.3.4 2.4. 局部变量

Tip: 应该尽可能缩小函数变量的作用域 (scope), 并在声明的同时初始化.

你可以在 C++ 函数的任何位置声明变量. 我们提倡尽可能缩小变量的作用域, 且声明离第一次使用的位置越近越好. 这样读者更容易找到声明, 了解变量的类型和初始值. 特别地, 应该直接初始化变量而非先声明再赋值, 比如:

```
int i;
i = f();      // 不好: 初始化和声明分离.
```

```
int i = f(); // 良好: 声明时初始化 。
```

```
int jobs = NumJobs();
// 更多代码...
f(jobs);      // 不好: 初始化和使用位置分离.
```

```
int jobs = NumJobs();
f(jobs);      // 良好: 初始化以后立即 (或很快) 使用.
```

```
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}; // 良好: 立即初始化 v.
```

通常应该在语句内声明用于 `if`、`while` 和 `for` 语句的变量, 这样会把作用域限制在语句内. 例如:

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

需要注意的是, 如果变量是一个对象, 那么它每次进入作用域时会调用构造函数, 每次退出作用域时都会调用析构函数.

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

在循环的作用域外面声明这类变量更高效:

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

2.3.5 2.5. 静态和全局变量

Tip: 禁止使用 静态储存周期 (static storage duration) 的变量, 除非它们可以 平凡地析构 (trivially destructible). 简单来说, 就是析构函数 (destructor) 不会做任何事情, 包括成员和基类 (base) 的析构函数. 正式地说, 就是这一类型 (type) 没有用户定义的析构函数或虚析构函数 (virtual destructor), 且所有成员和基类也能平凡地析构. 函数的局部静态变量可以动态地初始化 (dynamic initialization) . 除了少数情况外, 不推荐动态初始化静态类成员变量或命名空间内的变量. 详情参见下文.

作为经验之谈: 若只看全局变量的声明, 如果该语句可以作为常量表达式 (constexpr), 则满足以上要求.

定义:

每个对象 (object) 都有与生命周期 (linetime) 相关的储存周期 (storage duration). 静态储存周期对象的存活时间是从程序初始化开始, 到程序结束为止. 这些对象可能是命名空间作用域内的变量 (全局变量)、类的静态数据成员或者用 `static` 修饰符 (specifier) 声明的函数局部变量. 对于函数局部静态变量, 初始化发生在在控制流第一次经过声明时; 所有其他对象会在程序启动时初始化. 程序退出时会销毁所有静态储存周期的对象 (这发生在未汇合 (join) 的线程终止前).

初始化过程可以是动态 (dynamic) 的, 也就是初始化过程中有不平凡 (non-trivial) 的操作. (例如, 会分配内存的构造函数, 或者用当前进程 ID 初始化的变量.) 其他初始化都是静态 (static) 初始化. 二者并非水火不容: 静态储存周期的变量 一**定**会静态初始化 (初始化为指定常量或给所有字节清零), 必要时会随后再次动态初始化.

优点:

全局或静态变量对很多场景有帮助: 具名常量 (named constants)、编译单元 (translation unit) 内部的辅助数据结构、命令行旗标 (flag)、日志、注册机制、后台基础设施等等.

缺点:

使用动态初始化或具有非平凡析构函数的全局和静态变量时, 会增加代码复杂度, 容易引发难以察觉的错误. 不同编译单元的动态初始化顺序不确定, 析构顺序也不确定 (只知道析构顺序一定是初始化顺序的逆序). 如果静态变量的初始化代码引用了另一个静态储存周期的变量, 这次访问可能发生在另一变量的生命周期开始前 (或生命周期结束后). 此外, 若有些线程没有在程序结束前汇合, 这些线程可能在静态变量析构后继续访问这些变量.

决定:

关于析构的决定

平凡的析构函数不受执行顺序影响 (他们实际上不算”执行”); 其他析构函数则有风险, 可能访问生命周期已结束的对象. 因此, 只有拥有平凡析构函数的对象才能采用静态储存周期. 基本类型 (例如指针和 `int`) 可以平凡地析构, 可平凡析构的类型所构成的数组也可以平凡地析构. 注意, 用 `constexpr` 修饰的变量可以平凡地析构.

```
const int kNum = 10;  // 允许

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};  // 允许

void foo() {
  static const char* const kMessages[] = {"hello", "world"};  // 允许
}

// 允许: constexpr 可以保证析构函数是平凡的.
constexpr std::array<int, 3> kArray = {1, 2, 3};
```

```
// 不好: 非平凡的析构.
const std::string kFoo = "foo";
```

(continues on next page)

(continued from previous page)

```
// 和上面相同的原因，即使 kBar 是引用（该规则也适用于生命周期被延长的临时对象）.
const std::string& kBar = StrCat("a", "b", "c");

void bar() {
  // 不好：非平凡的析构.
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

注意，引用不是对象，因此它们的析构函数不受限. 但是，它们仍需遵守动态初始化的限制. 特别地，我们允许形如 `static T& t = *new T;` 的函数内局部静态引用.

关于初始化的决定

初始化是更复杂的话题，因为我们不仅需要考虑构造函数的执行过程，也要考虑初始化表达式 (initializer) 的求值过程.

```
int n = 5;    // 可以
int m = f();  // ? (依赖 f)
Foo x;        // ? (依赖 Foo::Foo)
Bar y = g();  // ? (依赖 g 和 Bar::Bar)
```

除了第一行语句以外，其他语句都会受到不确定的初始化顺序影响.

我们所需的概念在 C++ 标准中的正式称谓是常量初始化 (constant initialization). 这意味着初始化表达式是常量表达式 (constant expression)，并且如果要用构造函数进行初始化，则该构造函数也必须声明为 `constexpr`:

```
struct Foo { constexpr Foo(int) {} };

int n = 5;  // 可以，5 是常量表达式.
Foo x(2);   // 可以，2 是常量表达式且被选中的构造函数也是 constexpr.
Foo a[] = { Foo(1), Foo(2), Foo(3) };  // 可以
```

可以自由使用常量初始化. 应该用 `constexpr` 或 `constinit` 标记静态变量的常量初始化过程. 应该假设任何没有这些标记的静态变量都是动态初始化的，并谨慎地检查这些代码.

作为反例，以下初始化过程有问题:

```
// 下文使用了这些声明.
time_t time(time_t*);       // 不是 constexpr!
int f();                    // 不是 constexpr!
struct Bar { Bar() {} };

// 有问题的初始化.
time_t m = time(nullptr);   // 初始化表达式不是常量表达式.
Foo y(f());                 // 同上
Bar b;                      // 被选中的构造函数 Bar::Bar() 不是 constexpr.
```

我们不建议且通常禁止动态地初始化全局变量. 不过，如果这一初始化过程不依赖于其他初始化过程的顺序，则可以允许. 若满足这一要求，则初始化的顺序变化不会产生任何区别. 例如:

```
int p = getpid();  // 若其他静态变量不会在初始化过程中使用 p, 则允许.
```

允许动态地初始化静态局部变量 (这是常见的).

常用的语法结构

- 全局字符串: 如果你需要具名的 (named) 全局或静态字符串常量, 可以采用 `constexpr` 修饰的 `string_view` 变量、字符数组或指向字符串字面量 (literal) 的字符指针. 字符串字面量具有静态储存周期, 因此通常能满足需要. 参见 第 140 号每周提示.
- 字典和集合等动态容器 (container): 若你需要用静态变量储存不会改变的数据 (例如用于搜索的集合或查找表), 不要使用标准库的动态容器, 因为这些容器拥有非平凡的析构函数. 可以考虑用平凡类型的数组替代, 例如 `int` 数组的数组 (作为把 `int` 映射到 `int` 的字典) 或者数对 (pair) 的数组 (例如一组 `int` 和 `const char*` 的数对). 对于少量数据, 线性搜索就足够了, 而且因为具有内存局部性 (memory locality) 而更加高效; 可以使用 absl/algorithm/container.h 中的工具实现常见操作. 如有需要, 可以保持数据有序并采用二分查找法 (binary search). 如果你确实需要使用标准库的动态容器, 建议使用如下文所述的函数内局部静态指针.
- 智能指针 (smart pointer, 例如 `std::unique_ptr` 和 `std::shared_ptr`) 在析构时有释放资源的操作, 因此不能作为静态变量. 请思考你的情景是否适用于本小节描述的其他模式. 简单的解决方式是, 用裸指针 (plain pointer) 指向动态分配的对象, 并且永远不删除这个对象 (参见最后一点).
- 自定义类型的静态变量: 如果静态数据或常量数据是自定义类型, 请给这一类型设置平凡的析构函数和 `constexpr` 修饰的构造函数.
- 若以上都不适用, 你可以采用函数内局部静态指针或引用, 动态分配一个对象且永不删除 (例如 `static const auto& impl = *new T(args...);`).

2.3.6 2.6. thread_local 变量

Tip: 必须使用编译期常量 (compile-time constant) 初始化在函数外定义的 `thread_local` 变量, 且必须使用 ABSL_CONST_INIT 属性来强制执行这一规则. 优先采用 `thread_local`, 而非其他定义线程内局部数据的方法.

定义:

我们可以用 `thread_local` 修饰符声明变量:

```
thread_local Foo foo = ...;
```

这样的变量本质上其实是一组不同的对象. 不同线程访问该变量时, 会访问各自的对象. `thread_local` 变量在很多方面类似于静态储存周期的变量. 例如, 可以在命名空间内、函数内或类的静态成员内声明这些变量, 但不能在类的普通成员内声明它们.

`thread_local` 实例与静态变量的初始化过程类似, 区别是 `thread_local` 实例会在每个线程启动时初始化, 而非程序启动时初始化. 这意味着函数内的 `thread_local` 变量是线程

安全的. 若要访问其他 thread_local 变量, 则有跟静态变量一样的初始化顺序问题 (而且问题更大).

thread_local 的变量也有微妙的析构顺序问题: 线程终止时, thread_local 的销毁顺序是初始化顺序的逆序 (正如 C++ 在其他部分的规则一样). 如果 thread_local 的变量在析构过程中访问了该线程中已销毁的其他 thread_local 变量, 就会出现难以调试的释放后使用 (use-after-free, 即野指针) 问题.

优点:

- 线程的局部数据可以从根本上防止竞态条件 (race) (因为通常只有一个线程访问), 因此 thread_local 能帮助并行化.
- 在创建线程局部数据的各种方法中, thread_local 是由语法标准支持的唯一方法.

缺点:

- 在线程启动或首次使用 thread_local 变量时, 可能触发很多难以预测、运行时间不可控的其他代码.
- thread_local 本质上是全局变量. 除了线程安全以外, 它具有全局变量的所有其他缺点.
- 在最坏情况下, thread_local 变量占用的内存与线程数量成正比, 占用量可能十分巨大.
- 成员数据 (data member) 必须是静态的才能声明为 thread_local.
- 若 thread_local 变量拥有复杂的析构函数, 我们可能遇到野指针. 特别地, 析构函数不能 (直接或间接地) 访问任何有可能已被销毁的其他 thread_local 变量. 我们难以检查这一规则.
- 那些用于全局/静态变量的、预防野指针的方法不适用于 thread_local. 展开来说, 我们可以跳过全局或局部变量的析构函数, 因为他们的生命周期会随着程序终止而自然结束. 因此, 操作系统很快就会回收泄露的内存和其他资源. 然而, 若跳过 thread_local 的析构函数, 那么资源泄漏量和程序运行期间创建的线程数量成正比.

决定:

位于类或命名空间中的 thread_local 变量只能用真正的编译时常量来初始化 (也就是不能动态初始化). 必须用 ABSL_CONST_INIT 修饰来保证这一点 (也可以用 constexpr 修饰, 但不常见).

```
ABSL_CONST_INIT thread_local Foo foo = ...;
```

函数中的 thread_local 变量没有初始化的顾虑, 但是在线程退出时有释放后使用的风险. 注意, 你可以用静态方法暴露函数内的 thread_local 变量, 来模拟类或命名空间中的 thread_local 变量:

```
Foo& MyThreadLocalFoo() {
  thread_local Foo result = ComplicatedInitialization();
  return result;
}
```

注意, 线程退出时会销毁 thread_local 变量. 如果析构函数使用了任何其他 (可能已经销毁的) thead_local 变量, 我们会遇到难以调试的野指针. 建议使用平凡的类型, 或析构函数中没有自定义代码的类型, 以减少访问其他 thread_local 变量的可能性.

建议优先使用 `thread_local` 定义线程的局部数据, 而非其他机制.

2.3.7 译者 (YuleFox) 笔记

1. `cc` 中的匿名命名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.

2.3.8 译者（acgtyrant）笔记

1. 注意「using 指示（using-directive）」和「using 声明（using-declaration）」的区别。
2. 匿名命名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。

2.4 3. 类

类 (class) 是 C++ 中最基本的代码单元. 因此，类在 C++ 中被广泛使用. 本节将列出编写类时需要注意的事项.

2.4.1 3.1. 构造函数的内部操作

Tip: 构造函数 (constructor) 中不得调用虚函数 (virtual method). 不要在没有错误处理机制的情况下进行可能失败的初始化.

定义

构造函数可以执行任意初始化操作.

优点

- 无需担心类是否已经初始化.
- 由构造函数完全初始化的对象可以是 `const` 类型, 也方便在标准容器或算法中使用.

缺点

- 在构造函数内调用虚函数时, 不会分派 (dispatch) 到子类的虚函数实现 (implementation). 即使当前还没有子类, 将来也是隐患, 会引发混乱.
- 构造函数难以报告错误, 只能让程序崩溃 (有时不合适) 或者用异常 (我们禁止使用) 来表示错误.
- 如果初始化失败，对象将处于异常状态. 为了检查对象有效性，需要额外添加 `IsValid()` 等状态检查机制, 但用户容易忘记调用这些检查.
- 由于无法获取构造函数的地址, 因此难以将构造函数的工作转交给其他线程执行.

结论

构造函数不允许调用虚函数. 合适时, 终止程序也是一种处理错误的方式. 否则, 可以像 第 42 号每周提示 一样定义 `Init()` 方法或工厂函数 (factory function). 若要添加 `Init()` 方法, 应该确保可以从对象的状态中得知哪些公用方法是可用的. 若在对象未构造完成时调用其方法, 很容易导致错误.

2.4.2 3.2. 隐式类型转换

Tip: 不要定义隐式类型转换. 定义类型转换运算符和单个参数的构造函数时, 请使用 `explicit` 关键字 (keyword).

定义

可以通过隐式转换, 在需要目标类型的地方使用源类型的对象. 例如向接受 `double` 参数的函数传入 `int` 类型的参数.

除了语言内置的隐式类型转换, 用户还可以在类中定义特定的成员来添加自定义转换函数. 若要在源类型中定义隐式类型转换, 可以定义一个以目标类型命名的类型转换运算符 (例如 `operator bool()`). 若要在目标类型中定义隐式类型转换, 则可以定义一个以源类型为唯一参数 (或唯一无默认值的参数) 的构造函数.

可以在构造函数或类型转换运算符上添加 `explicit` 关键字, 确保使用者必须明确指定目标类型, 例如使用转换 (cast) 运算符. 该关键字也适用于列表初始化:

```
class Foo {
    explicit Foo(int x, double y);
    ...
};

void Func(Foo f);
```

```
Func({42, 3.14});  // 报错
```

此类代码并非真正的隐式类型转换, 但语法标准将其视为类似的转换, 因此也适用于 `explicit` 关键字.

优点

- 有时目标类型显而易见. 隐式类型转换无需明确指定类名, 使类型更易用、更简洁.

- 隐式类型转换可以简化函数重载 (overload), 例如只需定义一个接受 `string_view` 参数的函数, 即可取代 `std::string` 和 `const char*` 的重载函数.
- 初始化对象时, 列表初始化语法简洁明了.

缺点

- 隐式转换可能掩盖类型不匹配错误. 有时目标类型与用户预期不符, 甚至用户不知道会出现类型转换.
- 隐式转换降低了代码可读性. 尤其是存在函数重载的情况下, 难以判断实际调用的函数.
- 单参数构造函数可能被意外当作隐式类型转换, 这可能不是作者的本意.
- 如果单参数构造函数没有 `explicit` 标记, 读者无法判断这是隐式类型转换还是遗漏了 `explicit` 标记.
- 隐式类型转换可能导致调用歧义 (call-site ambiguity), 尤其是存在双向隐式转换的情况下. 这种情况可能是因为两种类型都定义了隐式转换, 或者一种类型同时定义了隐式构造函数和隐式转换运算符.
- 如果隐式指定列表初始化时的目标类型, 那么同样存在上述问题, 尤其是在列表只有一个元素的情况下.

结论

在类的定义中, 类型转换运算符和单参数构造函数都应该标记为 `explicit`. 拷贝和移动构造函数除外, 因为它们不执行类型转换.

对于某些可互换的类型, 隐式类型转换是必要且恰当的, 例如两种类型的对象只是同一底层值的不同表示形式. 此时请联系项目负责人申请豁免.

接受多个参数的构造函数可以省略 `explicit` 标记. 接受单个 `std::initializer_list` 参数的构造函数也应该省略 `explicit` 标记, 以支持拷贝初始化 (copy-initialization, 例如 `MyType m = {1, 2};`).

2.4.3 3.3. 可拷贝类型和可移动类型

Tip: 类的公有接口必须明确指明该类是可拷贝的、仅可移动的、还是既不可拷贝也不可移动的. 如果该类型的复制和移动操作有明确的语义并且有用，则应该支持这些操作.

定义

可移动类型 (movable type) 可以用临时变量初始化或赋值.

可拷贝类型 (copyable type) 可以用另一个相同类型的对象初始化 (因此从定义上来说也是可移动的), 同时源对象的状态保持不变. `std::unique_ptr<int>` 是可移动但不可拷贝类型的例子 (因为在赋值过程中必须修改提供初始值的 `std::unique_ptr<int>` 对象). `int` 和 `std::string` 是既可移动也可拷贝的例子. (`int` 类型的移动和拷贝操作等效. `std::string` 类型的移动操作比拷贝操作的开销更低.)

对于用户自定义的类型, 拷贝操作由拷贝构造函数 (copy constructor) 和拷贝赋值运算符 (copy-assignment operator) 定义. 移动操作由移动构造函数 (move constructor) 和移动赋值运算符

(move-assignment operator) 定义, 不存在时由拷贝构造函数和拷贝赋值运算符代替.

编译器会在某些情况下隐式调用拷贝或移动构造函数, 例如使用值传递 (pass by value) 传递参数时.

优点

可移动和可拷贝类型的对象可以通过值传递来传递参数或返回值, 使 API 更简单、安全、通用. 与指针或引用传递 (pass by reference) 不同, 值传递不会造成所有权、生命周期和可变性 (mutability) 等方面的混乱, 也无需在调用约定中进行规定, 阻止了用户和定义者之间的跨作用域交互, 使代码更可读、更好维护, 并且更容易被编译器优化. 您可以在必须使用值传递的通用 API (例如大多数容器) 中使用这些对象, 同时它们在类型组合 (type composition) 等场景下具有更好的灵活性.

拷贝、移动构造函数和赋值运算符通常比其他替代方案 (比如 `Clone()`, `CopyFrom()` 或 `Swap()`) 更容易确保正确性, 因为可以隐式或显式用 `= default` 指示编译器自动生成代码. 这种写法简洁明了, 可以保证复制所有数据成员. 拷贝和移动构造函数通常更高效, 因为不需要分配堆内存, 不需要分离初始化和赋值的过程, 同时还适用于 拷贝消除 等优化.

用户可以通过移动运算符隐式而高效地从右值对象 (rvalue) 中转移资源. 这有时可以使代码更加清晰.

缺点

某些类型不应该支持拷贝. 为这些类型提供拷贝操作可能会产生误导性的、无意义甚至错误的结果. 对于单例对象的类型 (例如注册表)、用于特定作用域的类型 (例如 `Cleanup`) 或与其他对象紧密耦合的类型 (例如互斥锁), 它们的拷贝操作没有意义. 为多态类型的基类提供拷贝运算符很危险, 因为使用此运算符时会导致 对象切割 (object slicing) . 默认或者随意编写的拷贝操作可能存在错误, 往往会引发令人困惑且难以诊断的错误.

拷贝构造函数是隐式调用的, 因此很容易忽视这些调用. 一些其他编程语言经常或者强制使用引用传递. 习惯了这些语言的程序员可能会感到困惑. 同时, 这也容易导致过度拷贝, 引发性能问题.

结论

每个类的公有接口都应该明确指明该类是否支持拷贝和移动操作. 通常应在声明的 `public` 部分显式声明或删除对应的操作.

具体来说, 可拷贝的类应该显式声明拷贝运算符, 仅能移动的类应该显式声明移动运算符, 既不能拷贝也不能移动的类应该显式删除复制运算符. 可拷贝的类也可以声明移动运算符, 以支持更高效的移动. 您可以显式声明或删除所有四个拷贝和移动运算符, 但这不是必需的. 如果您提供拷贝或移动赋值运算符, 则必须提供同类的构造函数.

```
class Copyable {
 public:
  Copyable(const Copyable& other) = default;
  Copyable& operator=(const Copyable& other) = default;
  // 上面的声明阻止了隐式移动运算符.
  // 您可以显式声明移动操作以支持更高效的移动.
};
```

(continues on next page)

(continued from previous page)

```
class MoveOnly {
 public:
  MoveOnly(MoveOnly&& other) = default;
  MoveOnly& operator=(MoveOnly&& other) = default;

  // 复制操作被隐式删除了，但您也可以显式删除：
  MoveOnly(const MoveOnly&) = delete;
  MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable {
 public:
  // 既不可复制也不可移动.
  NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&) = delete;

  // 移动操作被隐式删除了，但您也可以显式声明：
  NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&&) = delete;
};
```

只有在以下显而易见的情况下, 才能省略上述声明:

- 如果一个类没有 `private` 部分 (例如结构体或纯接口基类), 则该类型的可拷贝性和可移动性可以由其公有数据成员的可拷贝性/可移动性来确定.
- 如果一个基类明显是不可移动或不可拷贝的, 那么其派生类自然也是不可移动或不可拷贝的. 如果纯接口基类只是隐式声明这些操作, 则不足以明确说明子类的可复制性或可移动性.
- 注意, 如果您显式声明或删除拷贝构造函数或拷贝赋值运算符之一, 另一个也必须显式声明或删除. 移动操作亦如此.

如果普通用户容易误解某个类的拷贝或移动操作的含义, 或者此操作会产生意想不到的开销, 那么这个类应该设计为不可拷贝或不可移动的. 可拷贝类型的移动操作只是一种性能优化, 容易增加复杂性并引发错误. 除非移动操作明显比拷贝操作更高效, 不要定义移动操作. 如果您的类可拷贝, 那么最好确保自动生成的默认 (default) 实现是正确的. 请像检查您自己的代码一样检查默认实现的正确性.

为了避免对象切割的风险, 基类最好是抽象 (abstract) 类. 若要声明抽象类, 可以将构造函数或析构函数声明为 `protected`, 或者声明纯虚 (pure virtual) 成员函数. 尽量避免继承一个具体类 (concrete class).

2.4.4 3.4. 结构体还是类

Tip: 只能用 `struct` 定义那些用于储存数据的被动对象. 其他情况应该使用 `class`.

C++ 中 `struct` 和 `class` 关键字的含义几乎一样. 我们自己为这两个关键字赋予了不同的语义, 所以您要选择合适的关键字.

应该用结构体定义用于储存数据的被动对象, 其中可能包含常量成员. 所有成员都必须是公共的. 结构体的成员之间不能存在不变式 (invariant) 关系, 因为用户直接访问这些成员时可能破坏不变式. 结构体可以有构造函数、析构函数和辅助方法, 但是这些函数不能要求或实现不变式.

如果需要实现更多功能或不变式约束, 或者该结构体用途广泛并且会在未来不断更新, 那么类更合适. 在不确定的时候, 应该选择类.

为了与 STL 保持一致, 特征 (trait)、模板元函数 *(template metafunction)* 、仿函数 (functor) 等无状态的类型可以使用结构体而不用类.

注意, 类和结构体的成员变量具有不同的命名规则.

2.4.5 3.5. 结构体、数对还是元组

Tip: 如果可以给成员起一个有意义的名字, 应该用结构体而不是数对 (pair) 或元组 (tuple).

虽然使用数对和元组可以免于定义自定义类型, 从而节省编写代码的时间, 但是有意义的成员名称通常比 `.first`, `.second` 和 `std::get<X>` 更可读. C++14 引入了 `std::get<Type>`, 只要某类型的元素唯一, 就可以根据类型而非下标来访问元组元素. 这在一定程度上缓解了问题, 但成员名称通常比类型名称更清晰、更有信息量.

数对和元组适合通用代码, 因为其中的元素没有特定含义. 与现有代码或 API 交互时也可能需要它们.

2.4.6 3.6. 继承

Tip: 通常情况下, 组合 (composition) 比继承 (inheritance) 更合适. 请使用 `public` 继承.

定义

当子类继承基类时, 子类会包含基类定义的所有数据及操作. **接口继承** (interface inheritance) 指从纯抽象基类 (pure abstract base class, 不包含状态或方法定义) 继承; 所有其他继承都是 **实现继承** (implementation inheritance).

优点

实现继承复用了基类代码, 因此可以减少代码量. 继承是在编译时声明的, 因此您和编译器都可以理解并检查错误. 接口继承可以强制一个类公开特定 API. 当类没有定义 API 中的方法时, 编译器可以检测到错误.

缺点

对于实现继承, 子类的实现代码散布在父类和子类之间, 因此更难理解. 子类不能重写 (override) 父类的非虚函数, 因此无法修改其实现.

多重继承的问题更严重, 因为这样通常会产生更大的性能开销 (事实上, 多重继承相比单继承的性能损失大于虚方法相比普通方法的性能损失). 此外, 这容易产生 **菱形继承** (diamond inheritance) 的模式, 造成歧义、混乱和严重错误.

结论

所有继承都应该使用 `public` 的访问权限. 如果要实现私有继承, 可以将基类对象作为成员变量保存. 当您不希望您的类被继承时, 可以使用 `final` 关键字.

不要过度使用实现继承. 组合 (composition) 通常更合适. 尽量只在“什么是什么”(“is-a”, YuleFox 注: 其他 “has-a” 情况下请使用组合) 关系的情况下使用继承: 如果 `Bar` 是一种 `Foo`, 那么 `Bar` 才能继承 `Foo`.

只将子类可能需要访问的成员函数设为 `protected`. 注意, 数据成员应该是私有的.

明确使用 `override` 或 `final` (较少使用) 关键字限定重写的虚函数或者虚析构函数. 重写时不要使用 `virtual` 关键字. 原因: 如果函数带有 `override` 或 `final` 关键字, 却没有正确重写基类的虚函数, 会导致编译错误, 有助于发现常见笔误. 这些限定符相当于文档. 如果不用限定符, 读者必须检查所有祖先类才能确定函数是否是虚函数.

允许多重继承, 但强烈建议避免多重实现继承.

2.4.7 3.7. 运算符重载

Tip: 谨慎使用运算符重载 (overload). 禁止自定义字面量 (user-defined literal).

定义

用户可以使用 `operator` 关键字来 重载内置运算符 , 前提是其中一个参数是用户自定义类型. 用户还可以使用 `operator""` 定义一类新的字面量, 或者定义类型转换函数 (例如 `operator bool()`).

优点

重载运算符可以让用户定义的类型拥有与内置类型相似的行为, 使得代码更简洁直观. 重载运算符相当于特定操作的惯用名称 (例如 ==, <, =, <<). 若用户定义的类型符合这些习惯, 则代码更易读, 并便于与使用这些名称的库进行互操作.

自定义字面量提供了一种创建自定义类型对象的简洁写法。

缺点

- 需要花费精力才能实现正确、一致且符合预期的一组重载运算符. 稍有不慎就会引起困惑和错误.
- 过度使用运算符会让代码难以理解, 特别是重载运算符的语义不合常理时.
- 函数重载的弊端也同样适用于运算符重载，甚至更加严重.
- 重载运算符可能会混淆视听, 让您把一些耗时的操作误以为是快速的内置运算.
- 要列出重载运算符的调用者, 需要能理解 C++ 语法的搜索工具, 无法使用 `grep` 等通用工具.
- 如果重载运算符的参数类型错误, 您可能会调用一个完全不同的重载函数, 而不是得到编译错误. 例如: `foo < bar` 和 `&foo < &bar` 会执行完全不同的代码.
- 某些运算符的重载是危险的. 例如, 重载一元运算符 `&` 时, 取决于重载声明是否在某段代码中可见, 同样的代码可能具有完全不同的含义. 重载的 `&&`, `||` 和 `,` 与内置运算符的运算顺序不一致. (译者注: 例如, 内置的 `&&` 运算符会短路求值, 左侧为假时会跳过右侧的运算, 而重载的运算符不会短路.)

- 通常我们在类的定义以外定义运算符, 所以不同的文件可能对同一个运算有不同的定义. 如果同一二进制文件中链接 (link) 了两种定义, 结果就是未定义行为 (undefined behavior), 可能出现难以发现的运行时错误.
- 自定义字面量会创造新的语法, 例如将 `std::string_view("Hello World")` 简写为 `"Hello World"sv`. 经验丰富的 C++ 程序员都对此感到陌生. 现有的语法更清晰, 尽管不够简洁.
- 自定义字面量不能限制在命名空间中, 因此使用自定义字面量时必须同时使用 using 指令 (using-directive, 我们:ref: *禁止使用 <namespaces>*) 或 using 声明 (using-declaration, 我们禁止在头文件里使用, 除非导入的名称是需要暴露的接口). 因为头文件不能使用自定义字面量, 所以源文件的字面量格式也不应该与头文件不同.

结论

重载运算符应该意义明确, 符合常理, 并且与对应的内置运算符行为一致. 例如, 应该用 `|` 表示位或/逻辑或, 而非类似 shell 的管道.

只为您自己定义的类型定义重载运算符. 具体来说, 重载运算符和对应的类型应该在同一个头文件, `.cc` 文件和命名空间中. 这样, 任何使用该类型的代码都可以使用这些运算符, 避免了多重定义的风险. 尽量避免用模板定义运算符, 否则每个可能的模板类型都必须符合以上要求. 定义一个运算符时, 请同时定义相关且有意义的运算符, 并且保证语义一致.

建议将不修改数据的二元运算符定义为非成员函数. 如果二元运算符是成员函数, 那么右侧的参数可以隐式类型转换, 左侧却不能. 此时可能 `a + b` 能编译而 `b + a` 编译失败, 令人困惑.

对可以判断相等性的类型 `T`, 请定义非成员运算符 `operator==`, 并用文档说明什么条件下认为 `T` 的两个值相等. 如果类型 `T` 的 “小于” 的概念是显而易见的, 那么您也可以定义 `operator<=>`, 并且保持与 `operator==` 的逻辑一致. 不建议重载其他的比较、排序运算符.

不要为了避免重载操作符而走极端. 比如, 应当定义 `==`, `=` 和 `<<` 而不是 `Equals()`, `CopyFrom()` 和 `PrintTo()`. 另一方面, 不要仅仅因为其他库需要运算符重载而定义运算符. 比如, 如果您的类型没有自然顺序, 但您要在 `std::set` 中存储这样的对象, 最好使用自定义比较器 (comparator) 而不是重载 `<`.

不要重载 `&&`, `||`, `,` 或一元的 (unary) `&` 运算符. 不要重载 `operator""`, 即不要引入自定义字面量. 不要使用其他人提供的任何自定义字面量 (包括标准库).

类型转换运算符的内容参见*隐式类型转换*. `=` 运算符的内容参见*拷贝构造函数*. 关于重载用于流 (stream) 操作的 `<<` 运算符, 参见*流*. 另请参阅*函数重载* 的规则, 这些规则也适用于运算符重载.

2.4.8 3.8. 访问控制

Tip: 类的 *所有*数据成员应该声明为私有 (private), 除非是常量. 这样做可以简化类的不变式 (invariant) 逻辑, 代价是需要增加一些冗余的访问器 (accessor) 代码 (通常是 const 方法).

由于技术原因, 在使用 Google Test 时, 我们允许在 `.cc` 文件中将测试夹具类 (test fixture class) 的数据成员声明为受保护的 (protected). 如果测试夹具类的声明位于使用该夹具的 `.cc` 文件之外 (例如在 `.h` 文件中), 则应该将数据成员设为私有.

2.4.9 3.9. 声明次序

Tip: 将相似的声明放在一起. 公有 (public) 部分放在最前面.

类的定义通常以 `public:` 开头, 其次是 `protected:`, 最后以 `private:` 结尾. 空的部分可以省略.

在各个部分中, 应该将相似的声明分组, 并建议使用以下顺序:

1. 类型和类型别名 (`typedef`, `using`, `enum`, 嵌套结构体和类, 友元类型)
2. (可选, 仅适用于结构体) 非静态数据成员
3. 静态常量
4. 工厂函数 (factory function)
5. 构造函数和赋值运算符
6. 析构函数
7. 所有其他函数 (包括静态与非静态成员函数, 还有友元函数)
8. 所有其他数据成员 (包括静态和非静态的)

不要在类定义中放置大段的函数定义. 通常, 只有简单、对性能至关重要且非常简短的方法可以声明为内联函数. 参见内联函数 一节.

2.4.10 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, 如果定义了其他构造函数, 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;

2.5 4. 函数

2.5.1 4.1. 输入和输出

总述

我们倾向于按值返回，否则按引用返回。避免返回指针，除非它可以为空.

说明

C++ 函数由返回值提供天然的输出，有时也通过输出参数（或输入/输出参数）提供. 我们倾向于使用返回值而不是输出参数：它们提高了可读性，并且通常提供相同或更好的性能.

C/C++ 中的函数参数或者是函数的输入, 或者是函数的输出, 或兼而有之. 非可选输入参数通常是值参或 `const` 引用, 非可选输出参数或输入/输出参数通常应该是引用（不能为空）. 对于可选的参数，通常使用 `std::optional` 来表示可选的按值输入，使用 `const` 指针来表示可选的其他输入. 使用非常量指针来表示可选输出和可选输入/输出参数.

避免定义需要 `const` 引用参数去超出生命周期的函数，因为 `const` 引用参数与临时变量绑定. 相反，要找到某种方法来消除生命周期要求（例如，通过复制参数），或者通过 `const` 指针传递它并记录生命周期和非空要求.

在排序函数参数时，将所有输入参数放在所有输出参数之前. 特别要注意, 在加入新参数时不要因为它们是新参数就置于参数列表最后, 而是仍然要按照前述的规则, 即将新的输入参数也置于输出参数之前.

这并非一个硬性规定. 输入/输出参数 (通常是类或结构体) 让这个问题变得复杂. 并且, 有时候为了其他函数保持一致, 你可能不得不有所变通.

2.5.2 4.2. 编写简短函数

总述

我们倾向于编写简短, 凝练的函数.

说明

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, 以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 / 调试起来很困难, 或者你只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.

2.5.3 4.3. 函数重载

总述

若要使用函数重载, 则必须能让读者一看调用点就胸有成竹, 而不用花心思猜测调用的重载函数到底是哪一种. 这一规则也适用于构造函数.

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*` 的函数对其进行重载:

```
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, 可以令代码更加直观. 模板化代码需要重载, 这同时也能为使用者带来便利.

缺点

如果函数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, 如果派生类只重载了某个函数的部分变体, 继承语义就容易令人困惑.

结论

如果打算重载一个函数, 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()`. 如果重载函数的目的是为了支持不同数量的同一类型参数, 则优先考虑使用 `std::vector` 以便使用者可以用列表初始化 指定参数.

2.5.4 4.4. 缺省参数

总述

只允许在非虚函数中使用缺省参数, 且必须保证缺省参数的值始终一致. 缺省参数与函数重载 遵循同样的规则. 一般情况下建议使用函数重载, 尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点的情况下.

优点

有些函数一般情况下使用默认参数, 但有时需要又使用非默认的参数. 缺省参数为这样的情形提供了便利, 使程序员不需要为了极少的例外情况编写大量的函数. 和函数重载相比, 缺省参数的语法更简洁明了, 减少了大量的样板代码, 也更好地区别了“必要参数”和“可选参数”.

缺点

缺省参数实际上是函数重载语义的另一种实现方式, 因此所有不应当使用函数重载的理由 也都适用于缺省参数.

虚函数调用的缺省参数取决于目标对象的静态类型, 此时无法保证给定函数的所有重载声明的都是同样的缺省参数.

缺省参数是在每个调用点都要进行重新求值的, 这会造成生成的代码迅速膨胀. 作为读者, 一般来说也更希望缺省的参数在声明时就已经被固定了, 而不是在每次调用时都可能会有不同的取值.

缺省参数会干扰函数指针, 导致函数签名与调用点的签名不一致. 而函数重载不会导致这样的问题.

结论

对于虚函数, 不允许使用缺省参数, 因为在虚函数中缺省参数不一定能正常工作. 如果在每个调用点缺省参数的值都有可能不同, 在这种情况下缺省函数也不允许使用. (例如, 不要写像 `void f(int n = counter++);` 这样的代码.)

在其他情况下, 如果缺省参数对可读性的提升远远超过了以上提及的缺点的话, 可以使用缺省参数. 如果仍有疑惑, 就使用函数重载.

2.5.5 4.5. 函数返回类型后置语法

总述

只有在常规写法 (返回类型前置) 不便于书写或不便于阅读时使用返回类型后置语法.

定义

C++ 现在允许两种不同的函数声明方式. 以往的写法是将返回类型置于函数名之前. 例如:

```
int foo(int x);
```

C++11 引入了这一新的形式. 现在可以在函数名前使用 `auto` 关键字, 在参数列表之后后置返回类型. 例如:

```
auto foo(int x) -> int;
```

后置返回类型为函数作用域. 对于像 `int` 这样简单的类型, 两种写法没有区别. 但对于复杂的情况, 例如类域中的类型声明或者以函数参数的形式书写的类型, 写法的不同会造成区别.

优点

后置返回类型是显式地指定*Lambda 表达式* 的返回值的唯一方式. 某些情况下, 编译器可以自动推导出 Lambda 表达式的返回类型, 但并不是在所有的情况下都能实现. 即使编译器能够自动推导, 显式地指定返回类型也能让读者更明了.

有时在已经出现了的函数参数列表之后指定返回类型, 能够让书写更简单, 也更易读, 尤其是在返回类型依赖于模板参数时. 例如:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

对比下面的例子:

```
template <class T, class U> decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

缺点

后置返回类型相对来说是非常新的语法, 而且在 C 和 Java 中都没有相似的写法, 因此可能对读者来说比较陌生.

在已有的代码中有大量的函数声明, 你不可能把它们都用新的语法重写一遍. 因此实际的做法只能是使用旧的语法或者新旧混用. 在这种情况下, 只使用一种版本是相对来说更规整的形式.

结论

在大部分情况下, 应当继续使用以往的函数声明写法, 即将返回类型置于函数名前. 只有在必需的时候 (如 Lambda 表达式) 或者使用后置语法能够简化书写并且提高易读性的时候才使用新的返回类型后置语法. 但是后一种情况一般来说是很少见的, 大部分时候都出现在相当复杂的模板代码中, 而多数情况下不鼓励写这样复杂的模板代码.

2.6 5. 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.

2.6.1 5.1. 所有权与智能指针

> 总述

动态分配出的对象最好有单一且固定的所有主, 并通过智能指针传递所有权.

> 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, 此时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.

智能指针是一个通过重载 `*` 和 `->` 运算符以表现得如指针一样的类. 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 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的所有权; 当 `std::unique_ptr` 离开作用域时, 对象就会被销毁. `std::unique_ptr` 不能被复制, 但可以把它移动（move）给新所有主. std::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, 有文档且不紊乱的话, 可读性会有很大提升.
- 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: 这时不光要操心所有权, 还要顾及别名, 生命周期, 可变性以及其它大大小小的问题.
- 其实值语义的开销经常被高估, 所以所有权传递带来的性能提升不一定能弥补可读性和复杂度的损失.
- 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.
- 如果使用智能指针, 那么资源释放发生的位置就会变得不那么明显.
- `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, 开销可能相当大.
- 某些极端情况下 (例如循环引用), 所有权被共享的对象永远不会被销毁.
- 智能指针并不能够完全代替原生指针.

> 结论

如果必须使用动态分配, 那么更倾向于将所有权保持在分配者手中. 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);
```

如果没有很好的理由, 则不要使用共享所有权. 这里的理由可以是为了避免开销昂贵的拷贝操作, 但是只有当性能提升非常明显, 并且操作的对象是不可变的 (比如说 `std::shared_ptr<const Foo>`) 时候, 才能这么做. 如果确实要使用共享所有权, 建议于使用 `std::shared_ptr` .

不要使用 `std::auto_ptr`, 使用 `std::unique_ptr` 代替它.

2.6.2 5.2. Cpplint

> 总述

使用 `cpplint.py` 检查风格错误.

> 说明

`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. 它不并完美, 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.

2.6.3 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.

2.7 6. 其他 C++ 特性

2.7.1 6.1. 右值引用

Tip: 仅在下面列出的某些特殊情况下使用右值引用.

定义:

> 右值引用是一种能绑定到右值表达式的引用类型, 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.

当标记“`&&`”应用于函数参数中的非限定模板参数时，将应用特殊的模板参数推导规则。这种引用称为转发引用.

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.

2.7.2 6.2. 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()`.

2.7.3 6.3. 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看来

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);
```

2.7.4 6.4. 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[]>`.

2.7.5 6.5. 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.

2.7.6 6.6. 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, 但是前者要求在堆栈分配内存，后者会导致刚创建的实例处于”无效“状态。
- 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.
- 启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)

2.7.7 6.7. 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.

2.7.8 6.8. 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. 仅在你对所做一切了然于心时使用.

至于 `dynamic_cast` 参见*6.7. 运行时类型识别*.

2.7.9 6.9. 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`.

2.7.10 6.10. 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).

2.7.11 6.11. `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:

- 如果函数不会修改你传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的一致性. 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.)

2.7.12 6.12. 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；当前对 constexpr 函数和构造函数中允许的限制可能会导致这些定义中解决的方法模糊。

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。避免复杂的函数定义，以使其能够与 constexpr 一起使用。千万别痴心妄想地想靠 constexpr 来强制代码「内联」。

2.7.13 6.13. 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!

2.7.14 6.14. 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/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)

2.7.15 6.15. 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。

2.7.16 6.16. 0, nullptr 和 NULL

Tip: 指针使用 nullptr，字符使用 '\0'(而不是 0 字面值)。

对于指针 (地址值)，使用 nullptr，因为这保证了类型安全。

使用 '\0' 作为空字符。使用正确的类型使代码更具可读性。

2.7.17 6.17. 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;
}
```

2.7.18 6.18. 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?）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。

2.7.19 6.19. 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);
```

(continues on next page)

(continued from previous page)

```
  }
};
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 (explict), 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*9.7. 列表初始化格式*.

2.7.20 6.20. Lambda 表达式

Tip: 适当使用 lambda 表达式。当 lambda 将转移当前作用域时，首选显式捕获。

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);
});
```

它们还允许通过名称显式或隐式使用默认捕获从封闭范围中捕获变量。显式捕获要求将每个变量作为值或引用捕获列出：

```
int weight = 3;
int sum = 0;
// Captures `weight` by value and `sum` by reference.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

默认捕获隐式捕获 lambda 正文中引用的任何变量，包括 this 是否使用了任何成员：

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Captures `lookup_table` by reference, sorts `indices` by the value
// of the associated element in `lookup_table`.
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
});
```

变量捕获还可以具有显式初始值设定项，该初始值设定项可用于按值捕获仅移动变量，或用于普通引用或值捕获无法处理的其他情况：

```
std::unique_ptr<Foo> foo = ...;
[foo = std::move(foo)] () {
    ...
}
```

此类捕获（通常称为“初始化捕获”或“广义 lambda 捕获”）实际上不需要从封闭作用域中“捕获”任何内容，甚至不需要从封闭作用域中具有名称; 此语法是定义 Lambda 对象成员的完全通用方法：

```
[foo = std::vector<int>({1, 2, 3})] () {
  ...
}
```

优点：

- 传函数对象给 STL 算法，Lambda 最简易，可读性也好。
- 适当使用默认捕获可以消除冗余，并突出显示默认捕获中的重要异常。
- Lambda, `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：

- lambda 中的变量捕获可能是悬空指针错误的根源，尤其是当 lambda 逃逸到当前作用域时。
- 按值默认捕获可能会产生误导，因为它们不能防止悬空指针错误。按值捕获指针不会导致深度复制，因此它通常具有与按引用捕获相同的生存期问题。这在按值捕获 this 时尤其令人困惑，因为的 this 用法通常是隐式的。
- 捕获实际上声明了新变量（无论捕获是否具有初始值设定项），但它们看起来与 C++ 中的任何其他变量声明语法完全不同。特别是，变量的类型没有位置，甚至没有 auto 占位符（尽管初始化捕获可

以间接指示它，例如，使用强制转换）。这甚至可能使得很难将它们识别为声明。

- 初始化捕获本质上依赖于类型推导，并且存在许多与 auto 相同的缺点，但另一个问题是语法甚至没有提示读者正在进行推导。
- lambda 的使用可能会失控; 非常长的嵌套匿名函数会使代码更难理解。

结论：

- 在适当的情况下使用 lambda 表达式，格式如下 所述。
- 如果 lambda 可能离开当前作用域，则首选显式捕获。例如：

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); });
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `Frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

更喜欢写：

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// BETTER - The compile will fail if `Frobnicate` is a member
// function, and it's clearer that `foo` is dangerously captured by
// reference.
```

- 仅当 lambda 的生存期明显短于任何潜在捕获时，才使用默认的引用捕获（[&] ）。
- 仅使用默认的按值（[=] ）捕获作为绑定短 lambda 的几个变量的方法，其中捕获的变量集一目了然，并且不会导致隐式捕获 this 。（这意味着出现在非静态类成员函数中并在其正文中引用非静态类成员的 lambda 必须显式或 this 通过 [&] 捕获。不建议使用默认的按值捕获来编写长而复杂的 lambda。
- 仅使用捕获来实际捕获封闭范围中的变量。不要将捕获与初始值设定项一起使用来引入新名称，或实质性地更改现有名称的含义。相反，以传统方式声明一个新变量，然后捕获它，或者避免使用 lambda 简写并显式定义函数对象。
- 有关指定参数和返回类型的指导，请参阅 类型推导 部分。

2.7.21 6.21. 模板编程

Tip: 不要使用复杂的模板编程

定义:

模板编程指的是利用 c++ 模板实例化机制是图灵完备性, 可以被用来实现编译时刻的类型判断的一系列编程技巧

优点:

模板编程能够实现非常灵活的类型安全的接口和极好的性能, 一些常见的工具比如 Google Test, std::tuple, std::function 和 Boost.Spirit. 这些工具如果没有模板是实现不了的

缺点:

- 模板编程所使用的技巧对于使用 c++ 不是很熟练的人是比较晦涩, 难懂的. 在复杂的地方使用模板的代码让人更不容易读懂, 并且 debug 和维护起来都很麻烦
- 模板编程经常会导致编译出错的信息非常不友好: 在代码出错的时候, 即使这个接口非常的简单, 模板内部复杂的实现细节也会在出错信息显示. 导致这个编译出错信息看起来非常难以理解.
- 大量的使用模板编程接口会让重构工具 (Visual Assist X, Refactor for C++ 等等) 更难发挥用途. 首先模板的代码会在很多上下文里面扩展开来, 所以很难确认重构对所有的这些展开的代码有用, 其次有些重构工具只对已经做过模板类型替换的代码的 AST 有用. 因此重构工具对这些模板实现的原始代码并不有效, 很难找出哪些需要重构.

结论:

- 模板编程有时候能够实现更简洁更易用的接口, 但是更多的时候却适得其反. 因此模板编程最好只用在少量的基础组件, 基础数据结构上, 因为模板带来的额外的维护成本会被大量的使用给分担掉
- 在使用模板编程或者其他复杂的模板技巧的时候, 你一定要再三考虑一下. 考虑一下你们团队成员的平均水平是否能够读懂并且能够维护你写的模板代码. 或者一个非 c++ 程序员和一些只是在出错的时候偶尔看一下代码的人能够读懂这些错误信息或者能够跟踪函数的调用流程. 如果你使用递归的模板实例化, 或者类型列表, 或者元函数, 又或者表达式模板, 或者依赖 SFINAE, 或者 sizeof 的 trick 手段来检查函数是否重载, 那么这说明你模板用的太多了, 这些模板太复杂了, 我们不推荐使用
- 如果你使用模板编程, 你必须考虑尽可能的把复杂度最小化, 并且尽量不要让模板对外暴露. 你最好只在实现里面使用模板, 然后给用户暴露的接口里面并不使用模板, 这样能提高你的接口的可读性. 并且你应该在这些使用模板的代码上写尽可能详细的注释. 你的注释里面应该详细的包含这些代码是怎么用的, 这些模板生成出来的代码大概是什么样子的. 还需要额外注意在用户错误使用你的模板代码的时候需要输出更人性化的出错信息. 因为这些出错信息也是你的接口的一部分, 所以你的代码必须调整到这些错误信息在用户看起来应该是非常容易理解, 并且用户很容易知道如何修改这些错误

2.7.22 6.22. Boost 库

Tip: 只使用 Boost 中被认可的库.

定义:

Boost 库集 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 : `boost/call_traits.hpp`
- Compressed Pair : `boost/compressed_pair.hpp`
- <The Boost Graph Library (BGL) : `boost/graph`, except serialization (`adj_list_serialize.hpp`) and parallel/distributed algorithms and data structures(`boost/graph/parallel/*` and `boost/graph/distributed/*`)
- Property Map : `boost/property_map.hpp`
- The part of 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 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 : `boost/bimap`
- Statistical Distributions and Functions : `boost/math/distributions`
- Multi-index : `boost/multi_index`
- Heap : `boost/heap`
- The flat containers from 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：`boost/ptr_container`，改用 std::unique_ptr
- Array：`boost/array.hpp`，改用 std::array

2.7.23 6.23. 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 。

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*6.22. 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。

结论：

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。

2.7.24 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，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。

4. friend 实际上只对函数 / 类赋予了对其所在类的访问权限，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，即的数据成员之后，参考声明顺序 。
7. 对使用 C++ 异常处理应具有怎样的态度? 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? 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。

2.8 7. 命名约定

最重要的一致性规则是命名管理. 命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义: 类型, 变量, 函数, 常量, 宏, 等等, 甚至. 我们大脑中的模式匹配引擎非常依赖这些命名规则.

命名规则具有一定随意性, 但相比按个人喜好命名, 一致性更重要, 所以无论你认为它们是否重要, 规则总归是规则.

2.8.1 7.1. 通用命名规则

总述

函数命名, 变量命名, 文件命名要有描述性; 少用缩写.

说明

尽可能使用描述性的命名, 别心疼空间, 毕竟相比之下让代码易于新读者理解更重要. 不要用只有项目开发者能理解的缩写, 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" 是一个常见的写法
int num_dns_connections;   // 人人都知道 "DNS" 是什么
```

```
int n;                     // 毫无意义.
int nerr;                  // 含糊不清的缩写.
int n_comp_conns;          // 含糊不清的缩写.
int wgc_connections;       // 只有贵团队知道是什么意思.
int pc_reader;             // "pc" 有太多可能的解释了.
int cstmr_id;              // 删减了若干字母.
```

注意, 一些特定的广为人知的缩写是允许的, 例如用 `i` 表示迭代变量和用 `T` 表示模板参数.

模板参数的命名应当遵循对应的分类: 类型模板参数应当遵循类型命名 的规则, 而非类型模板应当遵循变量命名 的规则.

2.8.2 7.2. 文件命名

总述

文件名要全部小写, 可以包含下划线 (_) 或连字符 (-), 依照项目的约定. 如果没有约定, 那么 "_" 更好.

说明

可接受的文件命名示例:

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_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, 参见头文件自足.

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`.

内联函数定义必须放在 `.h` 文件中. 如果内联函数比较短, 就直接将实现也放在 `.h` 中.

2.8.3 7.3. 类型命名

总述

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

说明

所有类型命名——类, 结构体, 类型定义 (`typedef`), 枚举, 类型模板参数——均使用相同约定, 即以大写字母开始, 每个单词首字母均大写, 不包含下划线. 例如:

```
// 类和结构体
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// 类型定义
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using 别名
using PropertiesMap = hash_map<UrlTableProperties *, string>;
```

(continues on next page)

(continued from previous page)

```
// 枚举
enum UrlTableErrors { ...
```

2.8.4 7.4. 变量命名

总述

变量 (包括函数参数) 和数据成员名一律小写, 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_`.

说明

普通变量命名

举例:

```
string table_name;  // 好 - 用下划线.
string tablename;   // 好 - 全小写.

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_;  // 好 - 后加下划线.
  string tablename_;   // 好.
  static Pool<TableInfo>* pool_;  // 好.
};
```

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;
  static Pool<UrlTableProperties>* pool;
};
```

结构体与类的使用讨论, 参考*结构体 vs. 类*.

2.8.5 7.5. 常量命名

总述

声明为 constexpr 或 const 的变量, 或在程序运行期间其值始终保持不变的, 命名时以 “k” 开头, 大小写混合. 例如:

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

说明

所有具有静态存储类型的变量 (例如静态变量或全局变量, 参见 存储类型) 都应当以此方式命名. 对于其他存储类型的变量, 如自动变量等, 这条规则是可选的. 如果不采用这条规则, 就按照一般的变量命名规则.

2.8.6 7.6. 函数命名

总述

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().

说明

一般来说, 函数名的每个单词首字母大写 (即“驼峰变量名”或“帕斯卡变量名”), 没有下划线. 对于首字母缩写的单词, 更倾向于将它们视作一个单词进行首字母大写 (例如, 写作 StartRpc() 而非 StartRPC()).

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

(同样的命名规则同时适用于类作用域与命名空间作用域的常量, 因为它们是作为 API 的一部分暴露对外的, 因此应当让它们看起来像是一个函数, 因为在这时, 它们实际上是一个对象而非函数的这一事实对外不过是一个无关紧要的实现细节.)

取值和设值函数的命名与变量一致. 一般来说它们的名称与实际的成员变量对应, 但并不强制要求. 例如 int count() 与 void set_count(int count).

2.8.7 7.7. 命名空间命名

总述

命名空间以小写字母命名. 最高级命名空间的名字取决于项目名称. 要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突.

顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字. 命名空间中的代码, 应当存放于和命名空间的名字匹配的文件夹或其子文件夹中.

注意不使用缩写作为名称 的规则同样适用于命名空间. 命名空间中的代码极少需要涉及命名空间的名称, 因此没有必要在命名空间中使用缩写.

要避免嵌套的命名空间与常见的顶级命名空间发生名称冲突. 由于名称查找规则的存在, 命名空间之间的冲突完全有可能导致编译失败. 尤其是, 不要创建嵌套的 std 命名空间. 建议使用更独特的项目标

识符 (websearch::index, websearch::index_util) 而非常见的极易发生冲突的名称 (比如 websearch::util).

对于 internal 命名空间, 要当心加入到同一 internal 命名空间的代码之间发生冲突 (由于内部维护人员通常来自同一团队, 因此常有可能导致冲突). 在这种情况下, 请使用文件名以使得内部名称独一无二 (例如对于 frobber.h, 使用 websearch::index::frobber_internal).

2.8.8 7.8. 枚举命名

总述

枚举的命名应当和常量 或宏 一致: kEnumName 或是 ENUM_NAME.

说明

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.

2.8.9 7.9. 宏命名

总述

你并不打算使用宏, 对吧? 如果你一定要用, 像这样命名: MY_MACRO_THAT_SCARES_SMALL_CHILDREN.

说明

参考预处理宏; 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
```

2.8.10 7.10. 命名规则的特例

总述

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 型实体; 参照 STL 命名约定

`LONGLONG_MAX`: 常量, 如同 `INT_MAX`

2.8.11 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.

2.9 8. 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, 但最好的代码应当本身就是文档. 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, 也就是下一个需要理解你的代码的人. 所以慷慨些吧, 下一个读者可能就是你!

2.9.1 8.1. 注释风格

总述

使用 `//` 或 `/* */`, 统一就好.

说明

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

2.9.2 8.2. 文件注释

总述

在每一个文件开头加入版权公告.

文件注释描述了该文件的内容. 如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

说明

法律公告和作者信息

每个文件都应该包含许可证引用. 为项目选择合适的许可证版本.(比如, Apache 2.0, BSD, LGPL, GPL)

如果你对原始作者的文件做了重大修改, 请考虑删除原作者信息.

文件内容

如果一个 `.h` 文件声明了多个概念, 则文件注释应当对文件的内容做一个大致的说明, 同时说明各概念之间的联系. 一个一到两行的文件注释就足够了, 对于每个概念的详细文档应当放在各个概念中, 而不是文件注释中.

不要在 `.h` 和 `.cc` 之间复制注释, 这样的注释偏离了注释的实际意义.

2.9.3 8.3. 类注释

总述

每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.

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

说明

类注释应当为读者理解如何使用与何时使用类提供足够的信息, 同时应当提醒读者在正确使用此类时应当考虑的因素. 如果类有任何同步前提, 请用文档说明. 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.

如果你想用一小段代码演示这个类的基本用法或通常用法, 放在类注释里也非常合适.

如果类的声明和定义分开了 (例如分别放在了 .h 和 .cc 文件中), 此时, 描述类用法的注释应当和接口定义放在一起, 描述类的操作和实现的注释应当和实现放在一起.

2.9.4 8.4. 函数注释

总述

函数声明处的注释描述函数功能; 定义处的注释描述函数实现.

说明

函数声明

基本上每个函数声明处前都应当加上注释, 描述函数的功能和用途. 只有在函数的功能简单而明显时才能省略这些注释 (例如, 简单的取值和设值函数). 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.
- 函数是否分配了必须由调用者释放的空间.
- 参数是否可以为空指针.
- 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, 或者对显而易见的内容进行说明. 下面的注释就没有必要加上 “否则返回 false” , 因为已经暗含其中了:

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

注释函数重载时, 注释的重点应该是函数中被重载的部分, 而不是简单的重复被重载的函数的注释. 多数情况下, 函数重载不需要额外的文档, 因此也没有必要加上注释.

注释构造/析构函数时, 切记读代码的人知道构造/析构函数的功能, 所以 “销毁这一对象” 这样的注释是没有意义的. 你应当注明的是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

如果函数的实现过程中用到了很巧妙的方式, 那么在函数定义处应当加上解释性的注释. 例如, 你所使用的编程技巧, 实现的大致步骤, 或解释如此实现的理由. 举个例子, 你可以说明为什么函数的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.

2.9.5 8.5. 变量注释

总述

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.

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. 如果有非变量的参数 (例如特殊值, 数据成员之间的关系, 生命周期等) 不能够用类型与变量名明确表达, 则应当加上注释. 然而, 如果变量类型与变量名已经足以描述一个变量, 那么就不再需要加上注释.

特别地, 如果变量可以接受 `NULL` 或 `-1` 等警戒值, 须加以说明. 比如:

```
private:
 // Used to bounds-check table accesses. -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, 以及作为全局变量的原因. 比如:

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

2.9.6 8.6. 实现注释

总述

对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

说明

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();  // 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.
}
std::vector<string> list{
```

(continues on next page)

(continued from previous page)

```
                    // Comments in braced lists describe the next element...
                    "First item",
                    // .. and should be aligned appropriately.
"Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

函数参数注释

如果函数参数的意义不明显, 考虑用下面的方式进行弥补:

- 如果参数是一个字面常量, 并且这一常量在多处函数调用中被使用, 用以推断它们一致, 你应当用一个常量名让这一约定变得更明显, 并且保证这一约定不会被打破.
- 考虑更改函数的签名, 让某个 `bool` 类型的参数变为 `enum` 类型, 这样可以让这个参数的值表达其意义.
- 如果某个函数有多个配置选项, 你可以考虑定义一个类或结构体以保存所有的选项, 并传入类或结构体的实例. 这样的方法有许多优点, 例如这样的选项可以在调用处用变量名引用, 这样就能清晰地表明其意义. 同时也减少了函数参数的数量, 使得函数调用更易读也易写. 除此之外, 以这样的方式, 如果你使用其他的选项, 就无需对调用点进行更改.
- 用具名变量代替大段而复杂的嵌套表达式.
- 万不得已时, 才考虑在调用点用注释阐明参数的意义.

比如下面的示例的对比:

```
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

和

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

哪个更清晰一目了然.

不允许的行为

不要描述显而易见的现象, 永远不要用自然语言翻译代码作为注释, 除非即使对深入理解 C++ 的读者来说代码的行为都是不明显的. 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:

你所提供的注释应当解释代码 *为什么* 要这么做和代码的目的, 或者最好是让代码自文档化.

比较这样的注释:

```
// Find the element in the vector.  <-- 差: 这太明显了!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

和这样的注释:

```
// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

自文档化的代码根本就不需要注释. 上面例子中的注释对下面的代码来说就是毫无必要的:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

2.9.7 8.7. 标点, 拼写和语法

总述

注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

说明

注释的通常写法是包含正确大小写和结尾句号的完整叙述性语句. 大多数情况下, 完整的句子比句子片段可读性更高. 短一点的注释, 比如代码行尾注释, 可以随意点, 但依然要注意风格的一致性.

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很大帮助.

2.9.8 8.8. TODO 注释

总述

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名字, 邮件地址, bug ID, 或其它身份标识和与这一 TODO 相关的 issue. 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, 因此当你加上带有姓名的 TODO 时, 一般都是写上自己的名字.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

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.”).

2.9.9 8.9. 弃用注释

总述

通过弃用注释（DEPRECATED comments）以标记某接口点已弃用.

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, 在括号中留下您的名字, 邮箱地址以及其他身份标识.

弃用注释应当包涵简短而清晰的指引, 以帮助其他人修复其调用点. 在 C++ 中, 你可以将一个弃用函数改造成一个内联函数, 这一函数将调用新的接口.

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.

2.9.10 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.

2.10 9. 格式

每个人都可能有自己的代码风格和格式, 但如果一个项目中的所有人都遵循同一风格的话, 这个项目就能更顺利地进行. 每个人未必能同意下述的每一处格式规则, 而且其中的不少规则需要一定时间的适应, 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轻松地阅读和理解代码.

为了帮助你正确的格式化代码, 我们写了一个 emacs 配置文件.

2.10.1 9.1. 行长度

总述

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, 但很多已有代码都遵照这一规则, 因此我们感觉一致性更重要.

优点

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

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, 可以很轻松地显示更多代码.

结论

80 个字符是最大值.

如果无法在不伤害易读性的条件下进行断行, 那么注释行可以超过 80 个字符, 这样可以方便复制粘贴. 例如, 带有命令示例或 URL 的行可以超过 80 个字符.

包含长路径的 `#include` 语句可以超出 80 列.

头文件保护 可以无视该原则.

2.10.2 9.2. 非 ASCII 字符

总述

尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.

说明

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 字符应当很少被用到. 特殊情况下可以适当包含此类字符. 例如, 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"`, 或者更简洁地写作 `u8"\uFEFF"`, 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)

使用 `u8` 前缀把带 `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, 除非你写的代码要调用 Windows API, 后者广泛使用了 `wchar_t`.

2.10.3 9.3. 空格还是制表位

总述

只使用空格, 每次缩进 2 个空格.

说明

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

2.10.4 9.4. 函数声明与定义

总述

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, 分行方式与函数调用 一致.

说明

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 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

注意以下几点:

- 使用好的参数名.
- 只有在参数未被使用或者其用途非常明显时, 才能省略参数名.
- 如果返回类型和函数名在一行放不下, 分行.
- 如果返回类型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永远没有空格.
- 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, 不另起新行.
- 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.
- 右圆括号和左大括号间总是有一个空格.
- 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.

未被使用的参数, 或者根据上下文很容易看出其用途的参数, 可以省略参数名:

```
class Foo {
 public:
  Foo(Foo&&);
  Foo(const Foo&);
  Foo& operator=(Foo&&);
  Foo& operator=(const Foo&);
};
```

未被使用的参数如果其用途不明显的话, 在函数定义处将参数名注释起来:

```
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}
```

```
// 差 - 如果将来有人要实现, 很难猜出变量的作用.
void Circle::Rotate(double) {}
```

属性, 和展开为属性的宏, 写在函数声明或定义的最前面, 即返回类型之前:

```
MUST_USE_RESULT bool IsOK();
```

2.10.5 9.5. Lambda 表达式

总述

Lambda 表达式对形参和函数体的格式化和其他函数一致; 捕获列表同理, 表项用逗号隔开.

说明

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());
```

2.10.6 9.6. 函数调用

总述

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.

说明

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以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, 那也可以如此做. 参数的格式处理应当以可读性而非其他作为最重要的原则.

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);
```

2.10.7 9.7. 列表初始化格式

总述

您平时怎么格式化函数调用, 就怎么格式化列表初始化.

说明

如果列表初始化伴随着名字, 比如类型或变量名, 格式化时将将名字视作函数调用名, {} 视作函数调用的括号. 如果没有名字, 就视作名字长度为零.

```
// 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 {"},  // 假设在 { 前有长度为零的名字.
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},  // 假设在 { 前有长度为零的名字.
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",  // 非常长的字符串, 前后
都需要断行.
```

(continues on next page)

(continued from previous page)

```
        some, other values},
    SomeOtherType{"Slightly shorter string",  // 稍短的字符串.
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};  // 字符串过长, 因此无法放在同一行.
MyType m = {  // 注意了, 您可以在 { 前断行.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

2.10.8 9.8. 条件语句

总述

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

说明

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. 哪一种都可以, 最重要的是 *保持一致*. 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. 还在犹豫的话, 就不要加空格了.

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

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:

```
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) {  // 好 - 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:

```
// 不允许 - 当有 ELSE 分支时 IF 块却写在同一行
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:

```
// 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;
}
```

2.10.9 9.9. 循环和开关选择语句

总述

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.

说明

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, 编译器将给出 warning). 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 - contunue 表明没有逻辑.
```

```
while (condition);  // 差 - 看起来仅仅只是 while/loop 的部分之一.
```

2.10.10 9.10. 指针和引用表达式

总述

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.

说明

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:

- 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, 空格前置.
char *c;
const string &str;

// 好, 空格后置.
char* c;
const string& str;
```

```
int x, *y;   // 不允许 - 在多重声明中不能使用 & 或 *
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.

2.10.11 9.11. 布尔表达式

总述

如果一个布尔表达式超过标准行宽, 断行方式要统一一下.

说明

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. 这个格式在 Google 里很常见, 虽然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.

2.10.12 9.12. 函数返回值

总述

不要在 return 表达式里加上非必须的圆括号.

说明

只有在写 x = expr 要加上括号的时候才在 return expr; 里使用括号.

```
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);
```

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

2.10.13 9.13. 变量及数组初始化

总述

用 =, () 和 {} 均可.

说明

您可以用 =, () 和 {}, 以下的例子都是正确的:

```
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);  // 内容为 100 个 1 的向量.
vector<int> v{100, 1};  // 内容为 100 和 1 的向量.
```

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);  // 好 - pi == 3.
int pi{3.14};  // 编译错误: 缩窄转换.
```

2.10.14 9.14. 预处理指令

总述

预处理指令不要缩进, 从行首开始.

说明

即使预处理指令位于缩进代码块中, 指令也应从行首开始.

```
// 好 - 指令从行首开始
  if (lopsided_score) {
#if DISASTER_PENDING      // 正确 - 从行首开始
    DropEverything();
# if NOTIFY               // 非必要 - # 后跟空格
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

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

2.10.15 9.15. 类格式

总述

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

说明

类声明 (下面的代码中缺少注释, 参考类注释) 的基本格式如下:

```
class MyClass : public OtherClass {
 public:      // 注意有一个空格的缩进
  MyClass();  // 标准的两空格缩进
  explicit MyClass(int var);
  ~MyClass() {}
```

(continues on next page)

(continued from previous page)

```
  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_;
};
```

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.

2.10.16 9.16. 构造函数初始值列表

总述

构造函数初始化列表放在同一行或按四格缩进并排多行.

说明

下面两种初始值列表方式都可以接受:

```
// 如果所有变量能放在同一行:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}

// 如果不能放在同一行,
// 必须置于冒号后, 并缩进 4 个空格
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// 如果初始化列表需要置于多行, 将每一个成员放在单独的一行
```

(continues on next page)

(continued from previous page)

```
// 并逐行对齐
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}

// 右大括号 } 可以和左大括号 { 放在同一行
// 如果这样做合适的话
MyClass::MyClass(int var)
    : some_var_(var) {}
```

2.10.17 9.17. 命名空间格式化

总述

命名空间内容不缩进.

说明

命名空间 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
```

不要在命名空间内缩进:

```
namespace {

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

}  // namespace
```

声明嵌套命名空间时, 每个命名空间都独立成行.

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

2.10.18 9.18. 水平留白

总述

水平留白的使用根据在代码中的位置决定. 永远不要在行尾添加没意义的留白.

说明

通用

```
void f(bool b) {  // 左大括号前总是有空格.
  ...
int i = 0;  // 分号前不加空格.
// 列表初始化中大括号内的空格是可选的.
// 如果加了空格, 那么两边都要加上.
int x[] = { 0 };
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:
  // 对于单行函数的实现, 在大括号内加上空格
  // 然后是函数实现
  Foo(int b) : Bar(), baz_(b) {}  // 大括号里面是空的话, 不加空格.
  void Reset() { baz_ = 0; }  // 用空格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（尤其是在没有其他人在处理这件事的时候). (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.
```

(continues on next page)

(continued from previous page)

```
    ...
  case 2: break;  // 如果冒号有代码, 加个空格.
```

操作符

```
// 赋值运算符前后总是有空格.
x = 0;

// 其它二元操作符也前后恒有空格, 不过对于表达式的子式可以不加空格.
// 圆括号内部没有紧邻空格.
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

```
// 尖括号 (< and >) 不与空格紧邻, < 前没有空格, > 和 ( 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;
```

2.10.19 9.19. 垂直留白

总述

垂直留白越少越好.

说明

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, 这取决于你的判断. 但通常是垂直留白越少越好.

下面的规则可以让加入的空行更有效:

- 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.

2.10.20 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, 怎么也说不过去;
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.

2.10.21 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 .
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, 大概没人用吧.
6. 注意构造函数初始值列表（Constructer 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? 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.

2.11 10. 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.

2.11.1 10.1. 现有不合规范的代码

总述

对于现有不符合既定编程风格的代码可以网开一面.

说明

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, 一致性也包括原有的一致性.

2.11.2 10.2. Windows 代码

总述

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. 我们希望任何人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.

说明

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. 即使如此, 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, 并将所有警告 (warnings) 当作错误 (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. 使用 __declspec(dllimport) 和 __declspec(dllexport) 是允许的, 但必须通过宏来使用, 比如 DLLIMPORT 和 DLLEXPORT, 这样其他人在分享使用这些代码时可以很容易地禁用这些扩展.

然而, 在 Windows 上仍然有一些我们偶尔需要违反的规则:

- 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. 你需要研究一下是否能够禁用 STL 的异常, 如果无法禁用, 可以启用编译器异常. (注意这只是为了编译 STL, 自己的代码里仍然不应当包含异常处理).
- 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, 请避免显式包含此文件 (除了在 `precompile.cc` 中), 使用 `/FI` 编译器选项以自动包含该文件.
- 资源头文件通常命名为 `resource.h` 且只包含宏, 这一文件不需要遵守本风格指南.

2.12 11. 结束语

运用常识和判断力, 并且 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, 那么你同样要这么做.

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. 我们展示的是一个总体的的风格规范, 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, 也让打乱读者在阅读代码时的节奏, 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!

CHAPTER 3

Objective-C 风格指南 - 内容目录

3.1 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

3.1.1 译者的话

ewanke

一直想翻译这个 style guide ，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

3.1.2 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
- Google’s Open Source C++ Style Guide

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 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 。

3.1.3 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;
```

(continues on next page)

(continued from previous page)

```
}

@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。

3.2 留白和格式

3.2.1 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。

3.2.2 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。

3.2.3 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:(NSRect)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:(NSRect)theRect
    evenLongerKeyword:(float)theInterval {
  ...
}
```

3.2.4 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];
```

3.2.5 @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
```

3.2.6 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();
}
```

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

3.2.8 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) {
```

(continues on next page)

(continued from previous page)

```
      // ...
    }
});

// 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];
```

3.3 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，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。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。

3.3.1 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``。

3.3.2 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"
```

(continues on next page)

(continued from previous page)

```
// 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];
}
```

3.3.3 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`）。

3.3.4 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。

3.3.5 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。

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。

3.3.6 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`。

3.4 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。

3.4.1 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。

3.4.2 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。

3.4.3 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")|
```

3.4.4 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 `IBOutlets` 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 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）

3.5 Cocoa 和 Objective-C 特性

3.5.1 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_;
```

(continues on next page)

(continued from previous page)

```
}
// public accessors, setter takes ownership
- (id)myInstanceVariable;
- (void)setMyInstanceVariable:(id)theVar;
@end
```

3.5.2 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。

3.5.3 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。

3.5.4 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。

3.5.5 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。

3.5.6 避免 +new

Tip: 不要调用 NSObject 类方法 new，也不要在子类中重载它。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。

3.5.7 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）。解决文案是将方法放进指定类别的 @implemenation 中。

如果你在使用 Objective-C 2.0，相反你应该使用 类扩展 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。

3.5.8 #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。

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"
```

3.5.9 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>
...
```

3.5.10 构建时即设定 autorelease

Tip: 当创建临时对象时，在同一行使用 autolease，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];
```

3.5.11 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];
}
```

3.5.12 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];
```

(continues on next page)

(continued from previous page)

```
  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];
}
```

3.5.13 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。

3.5.14 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];
}
```

3.5.15 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();
  }
```

(continues on next page)

(continued from previous page)

```
  @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。

3.5.16 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 。

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。

3.5.17 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!
```

3.5.18 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
```

3.5.19 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

3.5.20 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 CFType 的属性 CFType 应该永远使用 @dynamic 实现指示符。尽管 CFType 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
```

3.6 Cocoa 模式

3.6.1 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`。

3.6.2 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”）。

CHAPTER 4

Python 风格指南 - 内容目录

4.1 扉页

原作者

Amit Patel

Antoine Picard

Eugene Jhong

Jeremy Hylton

Matt Smart

Mike Shields

详细作者列表请参见英文原版的 Git 仓库.

翻译

guoqiao v2.19

xuxinkun v2.59

captainfffsama v2.6

楼宇 2023 年 4 月 16 日更新

项目主页

- Google Style Guide (英文原版)
- Google 开源项目风格指南 - 中文版

协议

4.2 背景

Python 是谷歌使用的重要动态语言。这本风格指南列举了 Python 程序应采纳和避免的风格。

为帮助你正确地格式化代码，我们创建了 Vim 的配置文件 。对于 Emacs 用户，保持默认设置即可。

许多团队采用 Black 和 Pyink 作为自动格式化工具, 以避免格式上的争论。

4.3 Python 语言规范

4.3.1 Lint

Tip: 用 pylintrc 运行 pylint, 以检查你的代码.

定义:

pylint 是在 Python 代码中寻找 bug 和格式问题的工具. 它寻找的问题就像 C 和 C++ 这些更静态的 (译者注: 原文是 less dynamic) 语言中编译器捕捉的问题. 出于 Python 的动态特性, 部分警告可能有误. 不过, 误报应该不常见.

优点:

可以发现疏忽, 例如拼写错误, 使用未赋值的变量等.

缺点:

pylint 不完美. 要利用其优势, 我们有时侯需要: a) 绕过它 b) 抑制它的警告或者 c) 改进它.

结论:

一定要用 pylint 检查你的代码.

抑制不恰当的警告, 以免其他问题被警告淹没。你可以用行注释来抑制警告. 例如:

```
def do_PUT(self):  # WSGI 接口名, 所以 pylint: disable=invalid-name
    ...
```

pylint 的警告均以符号名 (如 `empty-docstring`) 来区分. 谷歌特有的警告以 `g-` 为前缀.

如果警告的符号名不够见名知意，那么请添加注释。

这种抑制方式的好处是, 我们可以轻易搜索并重新评判这些注释.

你可以用命令 `pylint --list-msgs` 来列出 pylint 的所有警告. 你可以用命令 `pylint --help-msg=invalid-name` 来查询某个警告的详情.

相较于旧的格式 `pylint: disable-msg` , 本文推荐使用 `pylint: disable` .

如果有“参数未使用”的警告，你可以在函数体开头删除无用的变量，以消除警告. 一定要用注释说明你为什么删除这些变量. 注明”未使用.”即可. 例如:

```
def viking_cafe_order(spam: str, beans: str, eggs: str | None = None) -> str:
    del beans, eggs  # 未被维京人使用.
    return spam + spam + spam
```

(译者注：Viking 意为维京人.)

其他避免这种警告的常用方法还有: 用`_`作为未使用参数的名称; 给这些参数名加上前缀 `unused_`; 或者把它们赋值给变量 _. 我们允许但是不再推荐这些方法. 这会导致调用者无法通过参数名来传参，也不能保证变量确实没被引用。

4.3.2 导入

Tip: 使用 `import` 语句时, 只导入包和模块, 而不单独导入函数或者类。

定义:

用于方便模块间共享代码的重用机制.

优点:

命名空间的管理规范十分简单. 每个标识符的来源都用一致的方式来表示. `x.Obj` 表示 `Obj` 对象定义在模块 `x` 中.

缺点:

模块名可能有命名冲突. 有些模块名的长度过长以至于不方便.

结论:

1. 用 `import x` 来导入包和模块.
2. 用 `from x import y`, 其中 x 是包前缀, y 是不带前缀的模块名.
3. 在以下情况使用 `from x import y as z`: 如果有两个模块都叫 `y`; 如果 `y` 和当前模块的某个全局名称冲突; 如果 `y` 是长度过长的名称.
4. 仅当缩写 `z` 是标准缩写时才能使用 `import y as z`.(比如 `np` 代表 `numpy`.)

例如, 可以用如下方式导入模块 `sound.effects.echo`:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

导入时禁止使用相对包名. 即使模块在同一个包中, 也要使用完整包名. 这能避免无意间重复导入同一个包.

例外:

这一规定的例外是:

1. 以下用于静态分析和类型检查的模块:
    1. `typing` 模块

2. `collections.abc` 模块
3. `typing_extensions` 模块

2. six.moves 模块中的重定向.

4.3.3 包

Tip: 使用每个模块的完整路径名来导入模块.

优点:

避免模块名冲突, 或是因模块搜索路径与作者的想法不符而导入错误的包. 也更容易找到模块.

缺点:

部署代码更难, 因为你必须完整复刻包的层次. 在现代的部署模式下不再是问题.

结论:

所有新的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:

正确:

```
# 在代码中引用完整名称 absl.flags (详细版).
import absl.flags
from doctor.who import jodie

_FOO = absl.flags.DEFINE_string(...)
```

```
# 在代码中仅引用模块名 flags (常见情况).
from absl import flags
from doctor.who import jodie

_FOO = flags.DEFINE_string(...)
```

错误: (假设当前文件和 *jodie.py* 都在目录 *doctor/who/* 下)

```
# 没有清晰地表达作者想要导入的模块和最终导入的模块.
# 实际导入的模块取决于由外部环境控制的 sys.path.
# 那些名为 jodie 的模块中, 哪个才是作者想导入的?
import jodie
```

不能臆测 *sys.path* 包含主程序所在的目录, 即使这种环境的确存在. 因此, 代码必须认定 *import jodie* 表示的是名为 *jodie* 的第三方库或者顶层的包，而非当前目录的 *jodie.py*.

4.3.4 异常

Tip: 允许使用异常, 但必须谨慎使用.

定义:

异常是一种跳出正常的控制流, 以处理错误或其它异常情况的方法.

优点:

处理正常情况的控制流不会和错误处理代码混在一起. 在特定情况下, 它也能让控制流跳出多层调用帧. 例如, 一步跳出 N 多层嵌套的函数, 而不必逐层传递错误代码.

缺点:

可能导致控制流晦涩难懂. 调用库函数时容易忘记处理异常.

结论:

使用异常时必须遵守特定要求:

1. 优先使用合适的内置异常类. 比如, 用 `ValueError` 表示前置条件错误 (例如给必须为正数的参数传入了负值). 不要使用 `assert` 语句来验证公开 API 的参数值. 应该用 `assert` 来保证内部正确性, 不应该用 `assert` 来纠正参数或表示意外情况. 若要用异常来表示意外情况, 应该用 `raise`. 例如:

   正确:

```
def connect_to_next_port(self, minimum: int) -> int:
    """ 连接到下一个可用的端口.

    参数:
        minimum: 一个大于等于 1024 的端口号.

    返回:
        新的最小端口.

    抛出:
        ConnectionError: 没有可用的端口.
    """
    if minimum < 1024:
        # 注意这里抛出 ValueError 的情况没有在文档里说明, 因为 API 的
        # 错误用法应该是未定义行为.
        raise ValueError(f'最小端口号至少为 1024, 不能是 {minimum}.')
    port = self._find_next_open_port(minimum)
    if port is None:
        raise ConnectionError(
            f'未能通过 {minimum} 或更高的端口号连接到服务.')
    assert port >= minimum, (
        f'意外的端口号 {port}, 端口号不应小于 {minimum}.')
    return port
```

   错误:

```
def connect_to_next_port(self, minimum: int) -> int:
    """ 连接到下一个可用的端口.

    参数:
        minimum: 一个大于等于 1024 的端口号.

    返回:
        新的最小端口.
    """
    assert minimum >= 1024, '最小端口号至少为 1024.'
    port = self._find_next_open_port(minimum)
    assert port is not None
    return port
```

2. 模块或包可以定义自己的异常类型, 这些类必须继承已有的异常类. 异常类型名应该以 `Error` 为后缀, 并且不应该有重复 (例如 `foo.FooError`).

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` , 除非你想:

    1. 重新抛出异常.
    2. 在程序中创造一个隔离点, 记录并抑制异常, 让异常不再继续传播. 这种写法可以用在线程的最外层, 以避免程序崩溃.

    如果你使用这种写法, Python 将非常宽容. `except:` 真的会捕获任何错误, 包括拼写错误的符号名、`sys.exit()` 调用、`Ctrl+C` 中断、单元测试错误和各种你不想捕获的错误.

4. 最小化 `try/except` 代码块中的代码量. `try` 的范围越大, 就越容易把你没想到的那些能抛出异常的代码囊括在内. 这样的话, `try/except` 代码块就掩盖了真正的错误.

5. 用 `finally` 表示无论异常与否都应执行的代码. 这种写法常用于清理资源, 例如关闭文件.

4.3.5 全局变量

Tip: 避免全局变量.

定义:

在程序运行时可以发生变化的模块级变量和类属性 (class attribute).

优点:

偶尔有用.

缺点:

1. 破坏封装: 这种设计会阻碍一些有用的目标. 例如, 如果用全局变量来管理数据库连接, 那就难以同时连接两个数据库 (比如为了在数据迁移时比较差异). 全局注册表也有类似的问题.

2. 导入模块时可能改变模块的行为, 因为首次导入模块时会对全局变量赋值.

结论:

避免使用全局变量.

在特殊情况下需要用到全局变量时, 应将全局变量声明为模块级变量或者类属性, 并在名称前加 _ 以示为内部状态. 如需从外部访问全局变量, 必须通过公有函数或类方法实现. 详见 命名规则 章节. 请用注释或文档链接解释这些全局变量的设计思想.

我们允许并鼓励使用模块级常量, 例如 `_MAX_HOLY_HANDGRENADE_COUNT = 3` 表示内部常量, `SIR_LANCELOTS_FAVORITE_COLOR = "blue"` 表示公开 API 的常量. 注意常量名必须全部大写, 用下划线分隔单词. 详见 命名规则 章节.

4.3.6 嵌套/局部/内部类和函数

Tip: 可以用局部类和局部函数来捕获局部变量. 可以用内部类.

定义:

可以在方法、函数和类中定义内部类. 可以在方法和函数中定义嵌套函数. 嵌套函数可以只读访问外层作用域中的变量. (译者注: 即内嵌函数可以读外部函数中定义的变量, 但是无法改写, 除非使用 *nonlocal*)

优点:

方便定义作用域有限的工具类和函数. 便于实现 抽象数据类型. 常用于实现装饰器.

缺点:

无法直接测试嵌套的函数和类. 嵌套函数和嵌套类会让外层函数的代码膨胀, 可读性变差.

结论:

可以谨慎使用. 尽量避免使用嵌套函数和嵌套类, 除非需要捕获 `self` 和 `cls` 以外的局部变量. 不要仅仅为了隐藏一个函数而使用嵌套函数. 应将需要隐藏的函数定义在模块级别, 并给名称加上 _ 前缀, 以便在测试代码中调用此函数.

4.3.7 推导式 (comprehension expression) 和生成式 (generator expression)

Tip: 适用于简单情况.

定义:

列表、字典和集合的推导式和生成式可以用于简洁高效地创建容器和迭代器, 而无需借助循环、`map()`、`filter()`, 或者 `lambda`. (译者注: 元组是没有推导式的, `()` 内加类似推导式的句式返回的是个生成器)

优点:

相较于其它创建字典、列表和集合的方法, 简单的列表推导式更加清晰和简洁. 生成器表达式十分高效, 因为无需创建整个列表.

缺点:

复杂的列表推导式和生成式难以理解.

结论:

可以用于简单情况. 以下每个部分不应超过一行: 映射表达式、for 语句和过滤表达式. 禁止多重 for 语句和多层过滤. 情况复杂时, 应该用循环.

正确:

```
result = [mapping_expr for value in iterable if filter_expr]

result = [{'key': value} for value in iterable
          if a_long_filter_expression(value)]

result = [complicated_transform(x)
          for x in iterable if predicate(x)]

descriptive_name = [
    transform({'key': key, 'value': value}, color='black')
    for key, value in generate_iterable(some_input)
    if complicated_condition_is_met(key, value)
]

result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))

return {x: complicated_transform(x)
        for x in long_generator_function(parameter)
        if x is not None}

squares_generator = (x**2 for x in range(10))

unique_names = {user.name for user in users if user is not None}

eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')
```

错误:

```
result = [complicated_transform(
              x, some_argument=x+1)
          for x in iterable if predicate(x)]

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
```

(continues on next page)

(continued from previous page)

```
        for z in xrange(5)
        if y != z)
```

4.3.8 默认迭代器和操作符

Tip: 只要可行, 就用列表、字典和文件等类型的默认迭代器和操作符.

定义:

字典和列表等容器类型具有默认的迭代器和关系运算符 (`in` 和 `not in`).

优点:

默认迭代器和操作符简单高效. 这种写法可以直白地表达运算, 无需调用额外的函数. 使用默认操作符的函数是泛型函数, 可以用于任何支持该操作符的类型.

缺点:

你不能通过方法名来辨别对象的类型 (除非变量有类型注解). 不过这也是优点.

结论:

只要是支持的类型 (例如列表、字典和文件), 就使用默认迭代器和操作符. 内置类型也定义了一些返回迭代器的方法. 优先使用返回迭代器的方法, 而非返回列表的方法, 不过注意使用迭代器时不能修改容器.

正确:

```
for key in adict: ...
if obj in alist: ...
for line in afile: ...
for k, v in adict.items(): ...
```

错误:

```
for key in adict.keys(): ...
for line in afile.readlines(): ...
```

4.3.9 生成器

Tip: 按需使用生成器.

定义:

生成器函数会返回一个迭代器. 每当函数执行 `yield` 语句时, 迭代器就生成一个值. 随后, 生成器的运行状态将暂停, 直到需要下一个值的时候.

优点:

代码简单, 因为生成器可以保存局部变量和控制流. 相较于直接创建整个列表的函数, 生成器使用的内存更少.

缺点:

必须等到生成结束或者生成器本身被内存回收的时候, 生成器的局部变量才能被内存回收.

结论:

可以使用. 生成器的文档字符串中应使用”Yields:”而不是”Returns:”.

(译者注: 参看注释)

如果生成器占用了大量资源, 一定要强制清理资源.

一种清理资源的好方法是用上下文管理器包裹生成器 PEP-0533.

4.3.10 Lambda 函数

Tip: 适用于单行函数. 建议用生成式替代 `map()/filter()` 与 `lambda` 的组合.

定义:

lambda 定义匿名函数, 不像语句那样定义具名函数.

优点:

方便.

缺点:

比局部函数更难理解和调试. 缺失函数名会导致调用栈晦涩难懂. 由于 lambda 函数只能包含一个表达式, 因此其表达能力有限.

结论:

适用于单行函数. 如果函数体超过 60-80 个字符, 最好还是定义为常规的嵌套函数.

对于乘法等常见操作, 应该用 `operator` 模块中的函数代替 lambda 函数. 例如, 推荐用 `operator.mul` 代替 `lambda x, y: x * y`.

4.3.11 条件表达式

Tip: 适用于简单情况.

定义:

条件表达式 (又名三元运算符) 是 if 语句的缩略版. 例如: `x = 1 if cond else 2`.

优点:

比 if 语句更简短, 更方便.

缺点:

有时比 if 语句更难理解. 如果表达式很长，就难以一眼望到条件.

结论:

适用于简单情况. 以下每部分均不得长于一行: 真值分支, if 部分和 else 部分. 情况复杂时应使用完整的 if 语句.

正确:

```
one_line = 'yes' if predicate(value) else 'no'
slightly_split = ('yes' if predicate(value)
                  else 'no, nein, nyet')
the_longest_ternary_style_that_can_be_done = (
    'yes, true, affirmative, confirmed, correct'
    if predicate(value)
    else 'no, false, negative, nay')
```

错误:

```
bad_line_breaking = ('yes' if predicate(value) else
                     'no')  # 换行位置错误
portion_too_long = ('yes'
                    if some_long_module.some_long_predicate_function(
                        really_long_variable_name)
                    else 'no, false, negative, nay')    # 过长
```

4.3.12 默认参数值

Tip: 大部分情况下允许.

定义:

你可以为参数列表的最后几个参数赋予默认值, 例如, `def foo(a, b = 0):` . 如果调用 foo 时只带一个参数, 则 b 为 0. 如果调用时带两个参数, 则 b 的值等于第二个参数.

优点:

很多时候, 你需要一个拥有大量默认值的函数, 并且偶尔需要覆盖这些默认值. 通过默认参数值可以轻松实现这种功能, 不需要为了覆盖默认值而编写大量额外的函数. 同时, Python 不支持重载方法和函数, 而默认参数的写法可以轻松”仿造”重载行为.

缺点:

默认参数在模块被导入时求值且只计算一次. 如果值是列表和字典等可变类型, 就可能引发问题. 如果函数修改了这个值 (例如往列表内添加元素), 默认值就变化了.

结论:

可以使用, 不过有如下注意事项:

函数和方法的默认值不能是可变 (mutable) 对象.

正确:

```
def foo(a, b=None):
    if b is None:
        b = []
def foo(a, b: Optional[Sequence] = None):
    if b is None:
        b = []
def foo(a, b: Sequence = ()):  # 允许空元组, 因为元组是不可变的
```

错误:

```
from absl import flags
_FOO = flags.DEFINE_string(...)

def foo(a, b=[]):
    ...
def foo(a, b=time.time()):  # 确定要用模块的导入时间吗???
    ...
def foo(a, b=_FOO.value):  # 此时还没有解析 sys.argv...
    ...
def foo(a, b: Mapping = {}):  # 可能会赋值给未经过静态检查 (unchecked) 的代码
    ...
```

4.3.13 特性 (properties)

(译者注: 参照 fluent python. 这里将“property”译为”特性”, 而“attribute”译为属性. python 中数据的属性和处理数据的方法统称属性”(arrtibute)”, 而在不改变类接口的前提下用来修改数据属性的存取方法我们称为”特性 (property)”.)

Tip: 可以用特性来读取或设置涉及简单计算、逻辑的属性. 特性的实现必须和属性 (attribute) 一样满足这些通用要求: 轻量、直白、明确.

定义:

把读取、设置属性的函数包装为常规属性操作的写法.

优点:

1. 可以直接实现属性的访问、赋值接口, 而不必添加获取器 (getter) 和设置器 (setter).
2. 可以让属性变为只读.
3. 可以实现惰性求值.
4. 类的内部实现发生变化时, 可以用这种方法让用户看到的公开接口保持不变.

缺点:

1. 可能掩盖副作用, 类似运算符重载 (operator overload).
2. 子类继承时可能产生困惑.

结论:

允许使用特性. 但是, 和运算符重载一样, 只能在必要时使用, 并且要模仿常规属性的存取特点. 若无法满足要求, 请参考获取器和写入器 的规则.

举个例子, 一个特性不能仅仅用于获取和设置一个内部属性: 因为不涉及计算, 没有必要用特性 (应该把该属性设为公有). 而用特性来限制属性的访问或者计算 **简单**的衍生值则是正确的: 这种逻辑简单明了.

应该用 `@property` 装饰器 (decorator) 来创建特性. 自行实现的特性装饰器属于威力过大的功能.

特性的继承机制难以理解. 不要用特性实现子类能覆写 (override) 或扩展的计算功能.

4.3.14 True/False 的求值

Tip: 尽可能使用”隐式”假值.

定义:

Python 在计算布尔值时会把一些值视为 `False`. 简单来说, 所有的”空”值都是假值. 因此, `0, None, [], {}, ""` 作为布尔值使用时相当于 `False`.

优点:

Python 布尔值可以让条件语句更易懂, 减少失误. 多数时候运行速度也更快.

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”假值, 例如: 使用 `if foo:` 而非 `if foo != []:`. 不过还是有一些注意事项需要你铭记在心:

1. 一定要用 `if foo is None:` (或者 `is not None`) 来检测 `None` 值. 例如, 如果你要检查某个默认值为 `None` 的参数有没有被调用者覆盖, 覆盖的值在布尔语义下可能也是假值!
2. 永远不要用 == 比较一个布尔值是否等于 `False`. 应该用 `if not x:` 代替. 如果你需要区分 `False` 和 `None`, 你应该用复合表达式, 例如 `if not x and x is not None:`.
3. 多利用空序列 (字符串, 列表, 元组) 是假值的特点. 因此 `if not seq:` 比 `if len(seq):` 更好, `if not seq:` 比 `if not len(seq):` 更好.
4. 处理整数时, 使用隐式 False 可能会得不偿失 (例如不小心将 `None` 当做 0 来处理). 你可以显式比较整型值与 0 的关系 (`len()` 的返回值例外).

   正确:

```
if not users:
    print('无用户')

if i % 10 == 0:
    self.handle_multiple_of_ten()
```

(continues on next page)

(continued from previous page)

```
def f(x=None):
    if x is None:
        x = []
```

错误:

```
if len(users) == 0:
    print '无用户'

if not i % 10:
    self.handle_multiple_of_ten()

def f(x=None):
    x = x or []
```

5. 注意, '0' (字符串, 不是整数) 作为布尔值时等于 `True`.
6. 注意, 把 Numpy 数组转换为布尔值时可能抛出异常. 因此建议用 *.size* 属性检查 `np.array` 是否为空 (例如 `if not users.size`).

4.3.15 词法作用域 (Lexical Scoping, 又名静态作用域)

Tip: 可以使用.

定义:

嵌套的 Python 函数可以引用外层函数中定义的变量, 但是不能对这些变量赋值. 变量的绑定分析基于词法作用域, 也就是基于静态的程序文本. 任何在代码块内给标识符赋值的操作, 都会让 Python 将该标识符的所有引用变成局部变量, 即使读取语句写在赋值语句之前. 如果有全局声明, 该标识符会被视为全局变量.

一个使用这个特性的例子:

```
def get_adder(summand1: float) -> Callable[[float], float]:
    """ 返回一个函数, 该函数会给一个数字加上指定的值."""
    def adder(summand2: float) -> float:
        return summand1 + summand2

    return adder
```

(译者注: 这个函数的用法大概是: `fn = get_adder(1.2); sum = fn(3.4)`, 结果是 `sum == 4.6`.)

优点:

通常会产生更清晰、更优雅的代码. 尤其是让熟练使用 Lisp 和 Scheme(还有 Haskell, ML 等) 的程序员感到舒适.

缺点:

可能引发让人困惑的 bug, 例如下面这个依据 PEP-0227 改编的例子:

```
i = 4
def foo(x: Iterable[int]):
    def bar():
        print(i, end='')
    # ...
    # 很多其他代码
    # ...
    for i in x:  # 啊哈, i 是 Foo 的局部变量, 所以 bar 得到的是这个变量
        print(i, end='')
    bar()
```

因此 `foo([1, 2, 3])` 会输出 `1 2 3 3`, 而非 `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++ 之类的语言还是有很大差别的.)

结论:

可以使用.

4.3.16 函数与方法装饰器

Tip: 仅在有显著优势时, 审慎地使用装饰器. 避免使用 `staticmethod`. 减少使用 `classmethod`.

定义:

装饰器 (也就是 @ 标记) 作用在函数和方法上. 常见的装饰器是 `@property`, 用于把方法转化为动态求值的属性. 不过, 也可以用装饰器语法自行定义装饰器. 具体地说, 若有一个函数 `my_decorator`, 下面两段代码是等效的:

```
class C(object):
   @my_decorator
   def method(self):
       # 函数体 ...
```

```
class C(object):
    def method(self):
        # 函数体 ...
    method = my_decorator(method)
```

优点:

优雅地实现函数的变换; 这种变换可用于减少重复的代码, 或帮助检查不变式 (invariant).

缺点:

装饰器可以在函数的参数和返回值上执行任何操作, 这可能产生意外且隐蔽的效果. 而且, 装饰是在

定义对象时执行. 模块级对象 (类、模块级函数) 的装饰器在导入模块时执行. 当装饰器代码出错时, 很难恢复正常控制流.

结论:

仅在有显著优势时, 审慎地使用装饰器. 装饰器的导入和命名规则与函数相同. 装饰器的 pydoc 注释应清楚地说明该函数是装饰器. 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, 套接字, 数据库连接等), 因为执行装饰器时 (即导入模块时. `pydoc` 和其他工具也会导入你的模块) 可能无法连接到这些环境. 只要装饰器的调用参数正确, 装饰器应该 (尽最大努力) 保证运行成功.

装饰器是一种特殊形式的”顶级代码”. 参见关于《Python 风格规范》中“主程序”的章节.

不得使用 `staticmethod`, 除非为了兼容老代码库的 API 不得已而为之. 应该把静态方法改写为模块级函数.

仅在以下情况可以使用 `classmethod`: 实现具名构造函数 (named constructor); 在类方法中修改必要的全局状态 (例如进程内共享的缓存等)。

4.3.17 线程

Tip: 不要依赖内置类型的原子性.

虽然 Python 的内置类型表面上有原子性, 但是在特定情形下可能打破原子性 (例如用 Python 实现 `__hash__` 或 `__eq__` 的情况下). 因此它们的原子性不可靠. 你也不能臆测赋值是原子性的 (因为赋值的原子性依赖于字典的原子性).

选择线程间的数据传递方式时, 应优先考虑 `queue` 模块的 `Queue` 数据类型. 如果不适用, 则使用 `threading` 模块及其提供的锁原语 (locking primitives). 如果可行, 应该用条件变量和 `threading.Condition` 替代低级的锁.

4.3.18 威力过大的功能

Tip: 避开这些功能.

定义:

Python 是一种异常灵活的语言, 有大量花哨的功能, 诸如自定义元类 (metaclasses), 读取字节码 (bytecode), 及时编译 (on-the-fly compilation), 动态继承, 对象基类重设 (object reparenting), 导入 (import) 技巧, 反射 (例如 `getattr()`), 系统内部状态的修改, `__del__` 实现的自定义清理等等.

优点:

强大的语言功能让代码紧凑.

缺点:

这些很”酷”的功能十分诱人, 但多数情况下没必要使用. 包含奇技淫巧的代码难以阅读、理解和调

试. 一开始可能还好 (对原作者而言), 但以后回顾代码时, 这种代码通常比那些长而直白的代码更加深奥.

结论:

避开这些功能.

可以使用那些在内部利用了这些功能的标准模块和类, 比如 `abc.ABCMeta`, `dataclasses` 和 `enum`.

4.3.19 现代 python: from __future__ imports

Tip: 可以通过导入 `__future__` 包, 在较老的运行时上启用新语法, 并且只在特定文件上生效.

定义:

通过使用 `from __future__ import` 并启用现代的语法, 可以提前使用未来的 Python 特性.

优点:

实践表明, 该功能可以让版本升级过程更稳定, 因为可以逐步修改各个文件, 并用这样的兼容性声明来防止退化 (regression). 现代的代码便于维护, 因为不容易积累那些阻碍运行时升级的技术债.

缺点:

此类代码无法在过老的运行时上运行, 过老的版本可能没有实现所需的 `future` 功能. 这个问题在那些需要支持大量不同环境的项目中尤为明显.

结论:

from __future__ imports

鼓励使用 `from __future__ import` 语句. 这样, 你的源代码从今天起就能使用更现代的 Python 语法. 当你不再需要支持老版本时, 请自行删除这些导入语句.

如果你的代码要支持 3.5 版本, 而不是常规的 `>=3.7`, 请导入:

```
from __future__ import generator_stop
```

详情参见 Python future 语句 的文档.

除非你确定代码的运行环境已经足够现代, 否则不要删除 future 语句. 即使你用不到 future 语句, 也要保留它们, 以免其他编辑者不小心对旧的特性产生依赖.

在你认为恰当的时候, 可以使用其他来自 `from __future__` 的语句.

4.3.20 代码类型注释

Tip: 你可以根据 PEP-484 来对 python3 代码进行注释, 并使用诸如 pytype 之类的类型检查工具来检查代码.

类型注释既可以写在源码里, 也可以写在 pyi 中. 推荐尽量写在源码里. 对于第三方代码和扩展包, 请使用 pyi 文件.

定义:

用在函数参数和返回值上:

```
def func(a: int) -> List[int]:
```

也可以使用 PEP-526 中的语法来声明变量类型:

```
a: SomeType = some_func()
```

优点:

可以提高代码可读性和可维护性. 类型检查器可以把运行时错误变成编译错误, 并阻止你使用威力过大的功能.

缺点:

必须时常更新类型声明. 正确的代码也可能有误报. 无法使用威力大的功能.

结论:

强烈推荐你在更新代码时启用 python 类型分析. 在添加或修改公开 API 时, 请添加类型注释, 并在构建系统 (build system) 中启用 pytype. 由于 python 静态分析是新功能, 因此一些意外的副作用 (例如类型推导错误) 可能会阻碍你的项目采纳这一功能. 在这种情况下, 建议作者在 BUILD 文件或者代码中添加一个 TODO 注释或者链接, 描述那些阻碍采用类型注释的问题.

(译者注: 代码类型注释在帮助 IDE 或是 vim 等进行补全倒是很有效)

4.4 Python 风格规范

4.4.1 分号

Tip: 不要在行尾加分号, 也不要用分号将两条语句合并到一行.

4.4.2 行宽

Tip: 最大行宽是 80 个字符.

例外:

1. 长的导入 (import) 语句.
2. 注释里的 URL、路径名以及长的标志 (flag).
3. 不便于换行、不包含空格、模块级的长字符串常量, 比如 URL 或路径名.
4. Pylint 禁用注释. (例如: `# pylint: disable=invalid-name`)

不要用反斜杠表示 显式续行 (explicit line continuation).

应该利用 Python 的 圆括号, 中括号和花括号的隐式续行 (implicit line joining) . 如有需要, 你可以在表达式外围添加一对括号.

正确:

```
foo_bar(self, width, height, color='黑', design=None, x='foo',
        emphasis=None, highlight=0)

if (width == 0 and height == 0 and
    color == '红' and emphasis == '加粗'):

(bridge_questions.clarification_on
 .average_airspeed_of.unladen_swallow) = '美国的还是欧洲的?'

with (
    very_long_first_expression_function() as spam,
    very_long_second_expression_function() as beans,
    third_thing() as eggs,
):
    place_order(eggs, beans, spam, beans)
```

错误:

```
if width == 0 and height == 0 and \
    color == '红' and emphasis == '加粗':

bridge_questions.clarification_on \
    .average_airspeed_of.unladen_swallow = '美国的还是欧洲的?'

with very_long_first_expression_function() as spam, \
        very_long_second_expression_function() as beans, \
        third_thing() as eggs:
    place_order(eggs, beans, spam, beans)
```

如果字符串的字面量 (literal) 超过一行, 应该用圆括号实现隐式续行:

```
x = ('这是一个很长很长很长很长很长很长'
     '很长很长很长很长很长很长的字符串')
```

最好在最外层的语法结构上分行. 如果你需要多次换行, 应该在同一层语法结构上换行.

正确:

```
bridgekeeper.answer(
     name="亚瑟", quest=questlib.find(owner="亚瑟", perilous=True))

 answer = (a_long_line().of_chained_methods()
           .that_eventually_provides().an_answer())

 if (
     config is None
     or 'editor.language' not in config
     or config['editor.language'].use_spaces is False
 ):
   use_tabs()
```

错误:

```
bridgekeeper.answer(name="亚瑟", quest=questlib.find(
    owner="亚瑟", perilous=True))

answer = a_long_line().of_chained_methods().that_eventually_provides(
    ).an_answer()

if (config is None or 'editor.language' not in config or config[
    'editor.language'].use_spaces is False):
  use_tabs()
```

必要时, 注释中的长 URL 可以独立成行.

正确:

```
# 详情参见
# http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_
↪extension_full_specification.html
```

错误:

```
# 详情参见
# http://www.example.com/us/developer/documentation/api/content/\
# v2.0/csv_file_name_extension_full_specification.html
```

注意上面各个例子中的缩进; 详情参见缩进 章节的解释.

如果一行超过 80 个字符, 且 Black 或 Pyink 自动格式化工具无法继续缩减行宽, 则允许该行超过 80 个字符. 我们也鼓励作者根据上面的规则手动拆分.

4.4.3 括号

Tip: 使用括号时宁缺毋滥.

可以把元组 (tuple) 括起来, 但不强制. 不要在返回语句或条件语句中使用括号, 除非用于隐式续行或表示元组.

正确:

```
if foo:
    bar()
while x:
    x = bar()
if x and y:
    bar()
if not x:
    bar()
# 对于包含单个元素的元组, 括号比逗号更直观.
onesie = (foo,)
return foo
return spam, beans
return (spam, beans)
for (x, y) in dict.items(): ...
```

错误:

```
if (x):
    bar()
if not(x):
    bar()
return (foo)
```

4.4.4 缩进

Tip: 用 4 个空格作为缩进.

不要使用制表符. 使用隐式续行时, 应该把括起来的元素垂直对齐 (参见行宽 章节的示例), 或者添加 4 个空格的悬挂缩进. 右括号 (圆括号, 方括号或花括号) 可以置于表达式结尾或者另起一行. 另起一行时右括号应该和左括号所在的那一行缩进相同.

正确:

```
# 与左括号对齐.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

(continues on next page)

(continued from previous page)

```
meal = (spam,
        beans)

# 与字典的左括号对齐.
foo = {
    'long_dictionary_key': value1 +
                           value2,
    ...
}

# 4 个空格的悬挂缩进; 首行没有元素
foo = long_function_name(
    var_one, var_two, var_three,
    var_four)
meal = (
    spam,
    beans)

# 4 个空格的悬挂缩进; 首行没有元素
# 右括号另起一行.
foo = long_function_name(
    var_one, var_two, var_three,
    var_four
)
meal = (
    spam,
    beans,
)

# 字典中的 4 空格悬挂缩进.
foo = {
    'long_dictionary_key':
        long_dictionary_value,
    ...
}
```

错误:

```
# 首行不能有元素.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 禁止 2 个空格的悬挂缩进.
foo = long_function_name(
  var_one, var_two, var_three,
  var_four)
```

(continues on next page)

(continued from previous page)

```
# 字典没有悬挂缩进.
foo = {
    'long_dictionary_key':
    long_dictionary_value,
    ...
}
```

4.4.5 序列的尾部要添加逗号吗?

Tip: 仅当 `]`, `)`, `}` 和最后一个元素不在同一行时, 推荐在序列尾部添加逗号. 我们的 Python 自动格式化工具会把尾部的逗号视为一种格式提示.

4.4.6 Shebang 行

Tip: 大部分 `.py` 文件不必以 `#!` 开始. 可以根据 PEP-394 , 在程序的主文件开头添加 `#!/usr/bin/env python3` (以支持 virtualenv) 或者 `#!/usr/bin/python3`.

(译者注: 在计算机科学中, Shebang (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.)

内核会通过这行内容找到 Python 解释器, 但是 Python 解释器在导入模块时会忽略这行内容. 这行内容仅对需要直接运行的文件有效.

4.4.7 注释和文档字符串 (docstring)

Tip: 模块、函数、方法的文档字符串和内部注释一定要采用正确的风格.

文档字符串

Python 的文档字符串用于注释代码. 文档字符串是包、模块、类或函数里作为第一个语句的字符串. 可以用对象的 `__doc__` 成员自动提取这些字符串, 并为 `pydoc` 所用. (可以试试在你的模块上运行 `pydoc` 并观察结果). 文档字符串一定要用三重双引号 `"""` 的格式 (依据 PEP-257). 文档字符串应该是一行概述 (整行不超过 80 个字符), 以句号、问号或感叹号结尾. 如果要写更多注释 (推荐), 那么概述后面必须紧接着一个空行, 然后是剩下的内容, 缩进与文档字符串的第一行第一个引号对齐. 下面是更多有关文档字符串的格式规范.

模块

每个文件应该包含一个许可协议模版. 应根据项目使用的许可协议 (例如, Apache 2.0, BSD, LGPL, GPL) 选择合适的模版.

文件的开头应该是文档字符串, 其中应该描述该模块内容和用法.

```
""" 模块或程序的一行概述, 以句号结尾.

留一个空行. 接下来应该写模块或程序的总体描述. 也可以选择简要描述导出的类和函数,
和/或描述使用示例.

经典的使用示例:

foo = ClassFoo()
bar = foo.FunctionBar()
"""
```

测试模块

测试文件不必包含模块级文档字符串. 只有在文档字符串可以提供额外信息时才需要写入文件.

例如, 你可以描述运行测试时所需的特殊要求, 解释不常见的初始化模式, 描述外部环境的依赖等等.

```
""" 这个 blaze 测试会使用样板文件 (golden files) .

若要更新这些文件, 你可以在 `google3` 文件夹中运行
`blaze run //foo/bar:foo_test -- --update_golden_files`
"""
```

不要使用不能提供额外信息的文档字符串.

```
"""foo.bar 的测试."""
```

函数和方法

本节中的函数是指函数、方法、生成器 (generator) 和特性 (property).

满足下列任意特征的任何函数都必须有文档字符串:

1. 公开 API 的一部分
2. 长度过长
3. 逻辑不能一目了然

文档字符串应该提供充分的信息, 让调用者无需阅读函数的代码就能调用函数. 文档字符串应该描述函数的调用语法和语义信息, 而不应该描述具体的实现细节, 除非这些细节会影响函数的用法. 比如, 如果函数的副作用是会修改某个传入的对象, 那就需要在文档字符串中说明. 对于微妙、重要但是与调用者无关的实现细节, 相较于在文档字符串里说明, 还是在代码中间加注释更好.

文档字符串可以是陈述句 (`"""Fetches rows from a Bigtable."""`) 或者祈使句 (`"""Fetch rows from a Bigtable."""`), 不过一个文件内的风格应当一致. 对于

`@property` 修饰的数据描述符 (data descriptor), 文档字符串应采用和属性 (attribute) 或*函数参数*一样的风格 (`"""Bigtable 路径."""` 而非 `""" 返回 Bigtable 路径."""`).

对于覆写 (override) 基类 (base class) 方法的子类方法, 可以用简单的文档字符串引导读者阅读基类方法的文档字符串, 比如 `""" 参见基类."""`. 这样是为了避免到处复制基类方法中已有的文档字符串. 然而, 如果覆写的子类方法与基类方法截然不同, 或者有更多细节需要记录 (例如有额外的的副作用), 那么子类方法的文档字符串中至少要描述这些区别.

函数的部分特征应该在以下列出特殊小节中记录. 每小节有一行标题, 标题以冒号结尾. 除标题行外, 小节的其他部分应有 2 个或 4 个空格 (同一文件内应保持一致) 的悬挂缩进. 如果函数名和函数签名 (signature) 可以见名知意, 以至于一行文档字符串就能恰当地描述该函数, 那么可以省略这些小节.

Args: (参数:)

列出所有参数名. 参数名后面是一个冒号, 然后是一个空格或者换行符, 最后是描述. 如果描述过长以至于一行超出了 80 字符, 则描述部分应该比参数名所在的行多 2 个或者 4 个空格 (文件内应当一致) 的悬挂缩进. 如果代码没有类型注解, 则描述中应该说明所需的类型. 如果一个函数有形如 `*foo` (可变长参数列表) 或者 `**bar` (任意关键字参数) 的参数, 那么列举参数名时应该写成 `*foo` 和 `**bar` 的这样的格式.

Returns: (“返回:”)

生成器应该用 “Yields:” (“生成:”)

描述返回值的类型和意义. 如果函数仅仅返回 `None`, 这一小节可以省略. 如果文档字符串以 Returns (返回) 或者 Yields (生成) 开头 (例如 `""" 返回 Bigtable 的行, 类型是字符串构成的元组."""`) 且这句话已经足以描述返回值, 也可以省略这一小节. 不要模仿 Numpy 风格的文档 (例子). 他们在文档中记录作为返回值的元组时, 写得就像返回值是多个值且每个值都有名字 (没有提到返回的是元组). 应该这样描述此类情况: “返回: 一个元组 (mat_a, mat_b), 其中 mat_a 是…, 且…” . 文档字符串中使用的辅助名称不需要和函数体的内部变量名一致 (因为这些名称不是 API 的一部分).

Raises: (抛出:)

列出与接口相关的所有异常和异常描述. 用类似 Args (参数) 小节的格式, 写成异常名 + 冒号 + 空格/换行, 并添加悬挂缩进. 不要在文档中记录违反 API 的使用条件时会抛出的异常 (因为这会让违背 API 时出现的效果成为 API 的一部分, 这是矛盾的).

```
def fetch_smalltable_rows(
    table_handle: smalltable.Table,
    keys: Sequence[bytes | str],
    require_all_keys: bool = False,
) -> Mapping[bytes, tuple[str, ...]]:
    """ 从 Smalltable 获取数据行.

    从 table_handle 代表的 Table 实例中检索指定键值对应的行. 如果键值是字符串,
    字符串将用 UTF-8 编码.

    参数:
        table_handle: 处于打开状态的 smalltable.Table 实例.
        keys: 一个字符串序列, 代表要获取的行的键值. 字符串将用 UTF-8 编码.
```

(continues on next page)

(continued from previous page)

```
        require_all_keys: 如果为 True, 只返回那些所有键值都有对应数据的
            行.

    返回:
        一个字典, 把键值映射到行数据上. 行数据是字符串构成的元组. 例如:

        {b'Serak': ('Rigel VII', 'Preparer'),
         b'Zim': ('Irk', 'Invader'),
         b'Lrrr': ('Omicron Persei 8', 'Emperor')}

        返回的键值一定是字节串. 如果字典中没有 keys 参数中的某个键值, 说明
        表格中没有找到这一行 (且 require_all_keys 一定是 false).

    抛出:
        IOError: 访问 smalltable 时出现错误.
    """
```

以下这种在 Args (参数) 小节中换行的写法也是可以的:

```
def fetch_smalltable_rows(
    table_handle: smalltable.Table,
    keys: Sequence[bytes | str],
    require_all_keys: bool = False,
) -> Mapping[bytes, tuple[str, ...]]:
    """ 从 Smalltable 获取数据行.

    从 table_handle 代表的 Table 实例中检索指定键值对应的行. 如果键值是字符串,
    字符串将用 UTF-8 编码.

    参数:
        table_handle:
          处于打开状态的 smalltable.Table 实例.
        keys:
          一个字符串序列, 代表要获取的行的键值. 字符串将用 UTF-8 编码.
        require_all_keys:
          如果为 True, 只返回那些所有键值都有对应数据的行.

    返回:
        一个字典, 把键值映射到行数据上. 行数据是字符串构成的元组. 例如:

        {b'Serak': ('Rigel VII', 'Preparer'),
         b'Zim': ('Irk', 'Invader'),
         b'Lrrr': ('Omicron Persei 8', 'Emperor')}

        返回的键值一定是字节串. 如果字典中没有 keys 参数中的某个键值, 说明
        表格中没有找到这一行 (且 require_all_keys 一定是 false).
```

(continues on next page)

(continued from previous page)

```
    抛出:
        IOError: 访问 smalltable 时出现错误.
    """
```

类 (class)

类的定义下方应该有一个描述该类的文档字符串. 如果你的类包含公有属性 (attributes), 应该在 `Attributes` (属性) 小节中记录这些属性, 格式与函数的 `Args` (参数) 小节类似.

```
class SampleClass(object):
    """ 这里是类的概述.

    这里是更多信息....
    这里是更多信息....

    属性:
        likes_spam: 布尔值, 表示我们是否喜欢午餐肉.
        eggs: 用整数记录的下蛋的数量.
    """

    def __init__(self, likes_spam = False):
        """ 用某某某初始化 SampleClass."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """ 执行某某操作."""
```

类的文档字符串开头应该是一行概述, 描述类的实例所代表的事物. 这意味着 `Exception` 的子类 (subclass) 应该描述这个异常代表什么, 而不是描述抛出异常时的环境. 类的文档字符串不应该有无意义的重复, 例如说这个类是一种类.

正确:

```
class CheeseShopAddress:
""" 奶酪店的地址.

...
"""

class OutOfCheeseError(Exception):
""" 没有可用的奶酪."""
```

错误:

```
class CheeseShopAddress:
""" 一个描述奶酪店地址的类.
```

(continues on next page)

(continued from previous page)

```
...
"""


class OutOfCheeseError(Exception):
""" 在没有可用的奶酪时抛出."""
```

块注释和行注释

最后一种需要写注释的地方是代码中复杂的部分. 如果你可能在以后 代码评审 (code review) 时要解释某段代码, 那么现在就应该给这段代码加上注释. 应该在复杂的操作开始前写上若干行注释. 对于不是一目了然的代码, 应该在行尾添加注释.

```
# 我们用加权的字典搜索, 寻找 i 在数组中的位置. 我们基于数组中的最大值和数组
# 长度, 推断一个位置, 然后用二分搜索获得最终准确的结果.

if i & (i-1) == 0:  # 如果 i 是 0 或者 2 的整数次幂, 则为真.
```

为了提高可读性, 注释的井号和代码之间应有至少 2 个空格, 井号和注释之间应该至少有一个空格.

除此之外, 绝不要仅仅描述代码. 应该假设读代码的人比你更懂 Python, 只是不知道你的代码要做什么.

```
# 不好的注释: 现在遍历数组 b, 确保每次 i 出现时, 下一个元素是 i+1
```

4.4.8 标点符号、拼写和语法

Tip: 注意标点符号、拼写和语法. 文笔好的注释比差的注释更容易理解.

注释应该和记叙文一样可读, 使用恰当的大小写和标点. 一般而言, 完整的句子比残缺句更可读. 较短的注释 (比如行尾注释) 可以更随意, 但是你要保持风格一致.

尽管你可能会因为代码审稿人指出你误把冒号写作逗号而灰心, 但是保持源代码清晰可读也是非常重要的. 正确的标点、拼写和语法有助于实现这一目标.

4.4.9 字符串

Tip: 应该用 f-string、`%` 运算符或 `format` 方法来格式化字符串. 即使所有参数都是字符串, 也如此. 你可以自行评判合适的选项. 可以用 + 实现单次拼接, 但是不要用 + 实现格式化.

正确:

```
x = f'名称: {name}; 分数: {n}'
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}'.format(first, second)
x = '名称: %s; 分数: %d' % (name, n)
x = '名称: %(name)s; 分数: %(score)d' % {'name':name, 'score':n}
x = '名称: {}; 分数: {}'.format(name, n)
x = a + b
```

错误:

```
x = first + ', ' + second
x = '名称: ' + name + '; 分数: ' + str(n)
```

不要在循环中用 + 和 += 操作符来堆积字符串. 这有时会产生平方而不是线性的时间复杂度. 有时 CPython 会优化这种情况, 但这是一种实现细节. 我们无法轻易预测这种优化是否生效, 而且未来情况可能出现变化. 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` 拼接列表. 也可以将每个子串写入一个 `io.StringIO` 缓冲区中. 这些技巧保证始终有线性的平摊 (amortized) 时间复杂度.

正确:

```
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)
```

错误:

```
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>'
```

应该保持同一文件中字符串引号的一致性. 选择 ' 或者 " 以后不要改变主意. 如果需要避免用反斜杠来转义引号, 则可以使用另一种引号.

正确:

```
Python('为什么你要捂眼睛?')
Gollum("I'm scared of lint errors. (我害怕格式错误.)")
Narrator('"很好!" 一个开心的 Python 审稿人心想.')
```

(译者注: 注意“I’m”中间有一个单引号，所以这一行的外层引号可以用不同的引号.)

错误:

```
Python("为什么你要捂眼睛?")
Gollum('格式检查器. 它在闪耀. 它要亮瞎我们.')
Gollum("伟大的格式检查器永在. 它在看. 它在看.")
```

多行字符串推荐使用 """ 而非 '''. 当且仅当项目中用 ' 给常规字符串打引号时, 才能在文档字符串以外的多行字符串上使用 '''. 无论如何, 文档字符串必须使用 """.

多行字符串不会跟进代码其他部分的缩进. 如果需要避免字符串中的额外空格, 可以用多个单行字符串拼接, 或者用 textwrap.dedent() 删除每行开头的空格.

错误:

```
    long_string = """这样很难看.
不要这样做.
"""
```

正确:

```
long_string = """如果你可以接受多余的空格,
    就可以这样."""

long_string = ("如果你不能接受多余的空格,\n" +
               "可以这样.")

long_string = ("如果你不能接受多余的空格,\n"
               "也可以这样.")
```

```
import textwrap

long_string = textwrap.dedent("""\
  这样也行, 因为 textwrap.dedent()
  会删除每一行开头共有的空格.""")
```

注意, 这里的反斜杠没有违反显式续行的禁令. 此时, 反斜杠用于在字符串字面量 (literal) 中 对换行符转义.

日志

对于那些第一个参数是格式字符串 (包含 % 占位符) 的日志函数: 一定要用字符串字面量 (而非 f-string!) 作为第一个参数, 并用占位符的参数作为其他参数. 有些日志的实现会收集未展开的格式字符串, 作为可搜索的项目. 这样也可以免于渲染那些被设置为不用输出的消息.

正确;

```
import tensorflow as tf
logger = tf.get_logger()
logger.info('TensorFlow 的版本是: %s', tf.__version__)
```

```
import os
from absl import logging

logging.info('当前的 $PAGER 是: %s', os.getenv('PAGER', default=''))

homedir = os.getenv('HOME')
```

(continues on next page)

(continued from previous page)

```
if homedir is None or not os.access(homedir, os.W_OK):
    logging.error('无法写入主目录, $HOME=%r', homedir)
```

错误:

```
import os
from absl import logging

logging.info('当前的 $PAGER 是:')
logging.info(os.getenv('PAGER', default=''))

homedir = os.getenv('HOME')
if homedir is None or not os.access(homedir, os.W_OK):
    logging.error(f'无法写入主目录, $HOME={homedir!r}')
```

错误信息

错误信息 (例如: 诸如 `ValueError` 等异常的信息字符串和展示给用户的信息) 应该遵守以下三条规范:

1. 信息需要精确地匹配真正的错误条件.
2. 插入的片段一定要能清晰地分辨出来.
3. 要便于简单的自动化处理 (例如正则搜索, 也就是 grepping).

正确:

```
if not 0 <= p <= 1:
    raise ValueError(f'这不是概率值: {p!r}')

try:
    os.rmdir(workdir)
except OSError as error:
    logging.warning('无法删除这个文件夹 (原因: %r): %r',
                    error, workdir)
```

错误:

```
if p < 0 or p > 1:  # 问题: 遇到 float('nan') 时也为假!
    raise ValueError(f'这不是概率值: {p!r}')

try:
    os.rmdir(workdir)
except OSError:
    # 问题: 信息中存在错误的揣测,
    # 删除操作可能因为其他原因而失败, 此时会误导调试人员.
    logging.warning('文件夹已被删除: %s', workdir)

try:
```

(continues on next page)

(continued from previous page)

```
    os.rmdir(workdir)
except OSError:
    # 问题: 这个信息难以搜索, 而且某些 `workdir` 的值会让人困惑.
    # 假如有人调用这段代码时让 workdir = '已删除'. 这个警告会变成:
    # " 无法删除已删除文件夹."
    logging.warning('无法删除%s文件夹.', workdir)
```

4.4.10 文件、套接字 (socket) 和类似的有状态资源

Tip: 使用完文件和套接字以后, 显式地关闭它们. 自然地, 这条规则也应该扩展到其他在内部使用套接字的可关闭资源 (比如数据库连接) 和其他需要用类似方法关停的资源. 其他例子还有 mmap 映射、h5py 的文件对象 和 matplotlib.pyplot 的图像窗口 .

如果保持不必要的文件、套接字或其他有状态对象开启, 会产生很多缺点:

1. 它们可能消耗有限的系统资源, 例如文件描述符. 如果代码需要使用大量类似的资源而没有及时返还给系统, 就有可能出现原本可以避免的资源枯竭情况.
2. 保持文件的开启状态会阻碍其他操作, 例如移动、删除文件, 卸载 (unmont) 文件系统等等.
3. 如果程序的多个部分共享文件和套接字, 即使逻辑上文件已经关闭了, 仍然有可能出现意外的读写操作. 如果这些资源真正关闭了, 读写操作会抛出异常, 让问题早日浮出水面.

此外, 即使文件和套接字 (以及其他行为类似的资源) 会在析构 (destruct) 时自动关闭, 把对象的生命周期和资源状态绑定的行为依然不妥:

1. 无法保证运行时 (runtime) 调用 `__del__` 方法的真正时机. 不同的 Python 实现采用了不同的内存管理技巧 (比如延迟垃圾处理机制, delayed garbage collection), 可能会随意、无限期地延长对象的生命周期.
2. 意想不到的文件引用 (例如全局对象和异常的堆栈跟踪, exception tracebacks) 可能让文件的存续时间比想象的更长.

依赖于终结器 (finalizer) 实现自动清理的方法有显著的副作用. 这在几十年的时间里、在多种语言中 (参见 这篇 Java 的文章) 多次引发严重问题.

推荐使用 “with” 语句 管理文件和类似的资源:

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print line
```

对于不支持 `with` 语句且类似文件的对象, 应该使用 `contextlib.closing()`:

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front_page:
```

(continues on next page)

(continued from previous page)

```
    for line in front_page:
        print line
```

少数情况下无法使用基于上下文 (context) 的资源管理, 此时文档应该清楚地解释代码会如何管理资源的生命周期.

4.4.11 TODO (待办) 注释

Tip: 在临时、短期和不够完美的代码上添加 TODO (待办) 注释.

待办注释以 `TODO` (待办) 这个全部大写的词开头, 紧跟着是用括号括起来的上下文标识符 (最好是 bug 链接, 有时是你的用户名). 最好是诸如 `TODO(https://crbug.com/<bug 编号>)`: 这样的 bug 链接, 因为 bug 有历史追踪和评论, 而程序员可能发生变动并忘记上下文. TODO 后面应该解释待办的事情.

统一 TODO 的格式是为了方便搜索并查看详情. TODO 不代表注释中提到的人要做出修复问题的保证. 所以, 当你创建带有用户名的 TODO 时, 大部分情况下应该用你自己的用户名.

```
# TODO(crbug.com/192795): 研究 cpufreq 的优化.
# TODO(你的用户名): 提交一个议题 (issue), 用 '*' 代表重复.
```

如果你的 TODO 形式类似于”将来做某事”, 请确保其中包含特别具体的日期 (“2009 年 11 月前解决”) 或者特别具体的事件 (“当所有客户端都能处理 XML 响应时, 删除这些代码”), 以便于未来的代码维护者理解.

4.4.12 导入 (import) 语句的格式

Tip: 导入语句应该各自独占一行. *typing 和 collections.abc 的导入除外*. 例如:

正确:

```
from collections.abc import Mapping, Sequence
import os
import sys
from typing import Any, NewType
```

错误:

```
import os, sys
```

导入语句必须在文件顶部, 位于模块的注释和文档字符串之后、全局变量和全局常量之前. 导入语句应该按照如下顺序分组, 从通用到特殊:

1. 导入 Python 的 `__future__`. 例如:

```
from __future__ import annotations
```

参见前文有关 __future__ 语句的描述.

2. 导入 Python 的标准库. 例如:

```
import sys
```

3. 导入 第三方 模块和包. 例如:

```
import tensorflow as tf
```

4. 导入代码仓库中的子包. 例如:

```
from otherproject.ai import mind
```

5. **已废弃的规则**: 导入应用专属的、与该文件属于同一个子包的模块. 例如:

```
from myproject.backend.hgwells import time_machine
```

你可能会在较老的谷歌风格 Python 代码中遇到这样的模式, 但现在不再执行这条规则. **我们建议新代码忽略这条规则.** 同等对待应用专属的子包和其他子包即可.

在每个分组内部, 应该按照模块完整包路径 (例如 `from path import ...` 中的 `path`) 的字典序排序, 忽略大小写. 可以选择在分组之间插入空行.

```
import collections
import queue
import sys

from absl import app
from absl import flags
import bs4
import cryptography
import tensorflow as tf

from book.genres import scifi
from myproject.backend import huxley
from myproject.backend.hgwells import time_machine
from myproject.backend.state_machine import main_loop
from otherproject.ai import body
from otherproject.ai import mind
from otherproject.ai import soul

# 旧的代码可能会把这些导入语句放在下面这里:
#from myproject.backend.hgwells import time_machine
#from myproject.backend.state_machine import main_loop
```

4.4.13 语句

Tip: 通常每个语句应该独占一行.

不过, 如果判断语句的主体与判断条件可以挤进一行, 你可以将它们放在同一行. 特别注意这不适用于 `try` / `except`, 因为 `try` 和 `except` 不能放在同一行. 只有在 `if` 语句没有对应的 `else` 时才适用.

正确:

```
if foo: bar(foo)
```

错误:

```
if foo: bar(foo)
else:   baz(foo)

try:                bar(foo)
except ValueError: baz(foo)

try:
    bar(foo)
except ValueError: baz(foo)
```

4.4.14 访问器 (getter) 和设置器 (setter)

Tip: 在访问和设置变量值时, 如果访问器和设置器 (又名为访问子 accessor 和变异子 mutator) 可以产生有意义的作用或效果, 则可以使用.

特别来说, 如果在当下或者可以预见的未来, 读写某个变量的过程很复杂或者成本高昂, 则应该使用这种函数.

如果一对访问器和设置器仅仅用于读写一个内部属性 (attribute), 你应该直接用公有属性取代它们. 相较而言, 如果设置操作会让部分状态无效化或引发重建, 则需要使用设置器. 显式的函数调用表示可能出现特殊的操作. 如果只有简单的逻辑, 或者在重构代码后不再需要访问器和设置器, 你可以用属性 (property) 替代.

(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)

访问器和设置器应该遵守命名规范, 例如 `get_foo()` 和 `set_foo()`.

如果之前的代码通过属性获取数据, 则不能把重新编写的访问器/设置器与这一属性绑定. 应该让任何用老办法访问变量的代码出现显眼的错误, 让使用者意识到代码复杂度有变化.

4.4.15 命名

Tip: 模块名: `module_name`; 包名: `package_name`; 类名: `ClassName`; 方法名: `method_name`; 异常名: `ExceptionName`; 函数名: `function_name`, `query_proper_noun_for_thing`, `send_acronym_via_https`; 全局常量名: `GLOBAL_CONSTANT_NAME`; 全局变量名: `global_var_name`; 实例名: `instance_var_name`; 函数参数名: `function_parameter_name`; 局部变量名: `local_var_name`.

函数名、变量名和文件名应该是描述性的, 避免缩写. 特别要避免那些对于项目之外的人有歧义或不熟悉的缩写, 也不要通过省略单词中的字母来进行缩写.

必须用 `.py` 作为文件后缀名. 不要用连字符.

需要避免的名称

1. 只有单个字符的名称, 除了以下特别批准的情况:
    1. 计数器和迭代器 (例如, `i`, `j`, `k`, `v` 等等).
    2. 在 `try/except` 语句中代表异常的 `e`.
    3. 在 `with` 语句中代表文件句柄的 `f`.
    4. 私有的、没有约束 (constrain) 的类型变量 (type variable, 例如 `_T = TypeVar("_T")`, `_P = ParamSpec("_P")`).
2. 包含连字符 (-) 的包名/模块名.
3. 首尾均为双下划线的名称, 例如 `__double_leading_and_trailing_underscore__` (此类名称是 Python 的保留名称).
4. 包含冒犯性词语的名称.
5. 在不必要的情况下包含变量类型的名称 (例如 `id_to_name_dict`).

命名规范

1. “内部 (Internal)”一词表示仅在模块内可用, 或者在类内是受保护/私有的.
2. 在一定程度上, 在名称前加单下划线 (_) 可以保护模块变量和函数 (格式检查器会对受保护的成员访问操作发出警告).
3. 在实例的变量或方法名称前加双下划线 (__, 又名为 dunder) 可以有效地把变量或方法变成类的私有成员 (基于名称修饰 name mangling 机制). 我们不鼓励这种用法, 因为这会严重影响可读性和可测试性, 而且没有 **真正**实现私有. 建议使用单下划线.
4. 应该把相关的类和顶级函数放在同一个模块里. 与 Java 不同, 不必限制一个模块只有一个类.
5. 类名应该使用首字母大写的形式 (如 CapWords), 但是模块名应该用小写加下划线的形式 (如 lower_with_under.py). 尽管有些旧的模块使用类似于 CapWords.py 这样的形式, 现在我们不再鼓励这种命名方式, 因为模块名和类名相同时会让人困惑 (“等等, 我刚刚写的是 `import StringIO` 还是 `from StringIO import StringIO`?”).

6. 新的 **单元测试**文件应该遵守 PEP 8, 用小写加下划线格式的方法名, 例如 `test_< 被测试的方法名>_< 状态>`. 有些老旧的模块有形如 `CapWords` 这样大写的方法名, 为了保持风格一致, 可以在 test 这个词和方法名之后, 用下划线分割名称中不同的逻辑成分. 比如一种可行的格式之一是 `test<被测试的方法>_< 状态>`.

文件名

所有 Python 文件名都应该以 `.py` 为文件后缀且不能包含连字符 (-). 这样便于导入这些文件并编写单元测试. 如果想通过不含后缀的命令运行程序, 可以使用软链接文件 (symbolic link) 或者 `exec "$0.py" "$@"` 这样简单的 bash 脚本.

根据 Python 之父 Guido 的建议所制定的规范

Table1: 描述

类型	公有	内部
包	小写下划线	
模块	小写下划线	下划线 + 小写下划线
类	大驼峰	下划线 + 大驼峰
异常	大驼峰	
函数	小写下划线	下划线 + 小写下划线
全局常量/类常量	大写下划线	下划线 + 大写下划线
全局变量/类变量	小写下划线	下划线 + 小写下划线
实例变量	小写下划线	下划线 + 小写下划线 (受保护)
方法名	小写下划线	下划线 + 小写下划线 (受保护)
函数参数/方法参数	小写下划线	
局部变量	小写下划线	

Table2: 例子

类型	公有	内部
包	`lower_with_under`	
模块	`lower_with_under`	`_lower_with_under`
类	`CapWords`	`_CapWords`
异常	`CapWords`	
函数	`lower_with_under()`	`_lower_with_under()`
全局常量/类常量	`CAPS_WITH_UNDER`	`_CAPS_WITH_UNDER`
全局变量/类变量	`lower_with_under`	`_lower_with_under`
实例变量	`lower_with_under`	`_lower_with_under`
方法名	`lower_with_under()`	`_lower_with_under()`
函数参数/方法参数	`lower_with_under`	
局部变量	`lower_with_under`	

数学符号

对于涉及大量数学内容的代码, 如果相关论文或算法中有对应的符号, 则可以忽略以上命名规范并使用较短的变量名. 若要采用这种方法, 应该在注释或者文档字符串中注明你所使用的命名规范的来源. 如果原

文无法访问, 则应该在文档中清楚地记录命名规范. 建议公开的 API 使用符合 PEP8 的、描述性的名称, 因为使用 API 的代码很可能缺少相关的上下文信息.

4.4.16 主程序

Tip: 使用 Python 时, 提供给 `pydoc` 和单元测试的模块必须是可导入的. 如果一个文件是可执行文件, 该文件的主要功能应该位于 `main()` 函数中. 你的代码必须在执行主程序前检查 `if __name__ == '__main__'`, 这样导入模块时不会执行主程序.

使用 absl 时, 请调用 `app.run`:

```
from absl import app
...

def main(argv):
    # 处理非标志 (non-flag) 参数
    ...

if __name__ == '__main__':
    app.run(main)
```

否则, 使用:

```
def main():
    ...

if __name__ == '__main__':
    main()
```

导入模块时会执行该模块的所有顶级代码. 注意顶级代码中不能有 `pydoc` 不该执行的操作, 比如调用函数, 创建对象等.

4.4.17 函数长度

Tip: 函数应该小巧且专一.

我们承认有时长函数也是合理的, 所以不硬性限制函数长度. 若一个函数超过 40 行, 应该考虑在不破坏程序结构的前提下拆分这个函数.

即使一个长函数现在没有问题, 几个月后可能会有别人添加新的效果. 此时容易出现隐蔽的错误. 保持函数简练, 这样便于别人阅读并修改你的代码.

当你使用某些代码时, 可能发现一些冗长且复杂的函数. 要勇于修改现有的代码: 如果该函数难以使用或者存在难以调试的错误, 亦或是你想在不同场景下使用该函数的片段, 不妨考虑把函数拆分成更小、更容易管理的片段.

4.4.18 类型注解 (type annotation)

通用规则

1. 熟读 PEP-484 .
2. 仅在有额外类型信息时才需要注解方法中 `self` 或 `cls` 的类型. 例如:

```
@classmethod
def create(cls: Type[_T]) -> _T:
    return cls()
```

3. 类似地, 不需要注解 `__init__` 的返回值 (只能返回 `None`).
4. 对于其他不需要限制变量类型或返回类型的情况, 应该使用 `Any`.
5. 无需注解模块中的所有函数.
    1. 至少需要注解你的公开 API.
    2. 你可以自行权衡, 一方面要保证代码的安全性和清晰性, 另一方面要兼顾灵活性.
    3. 应该注解那些容易出现类型错误的代码 (比如曾经出现过错误或疑难杂症).
    4. 应该注解晦涩难懂的代码.
    5. 应该注解那些类型已经确定的代码. 多数情况下，即使注解了成熟的代码中所有的函数，也不会丧失太多灵活性.

换行

尽量遵守前文所述的缩进规则.

添加类型注解后, 很多函数签名 (signature) 会变成每行一个参数的形式. 若要让返回值单独成行, 可以在最后一个参数尾部添加逗号.

```
def my_method(
    self,
    first_var: int,
    second_var: Foo,
    third_var: Bar | None,
) -> int:
    ...
```

尽量在变量之间换行, 避免在变量和类型注解之间换行. 当然, 若所有东西可以挤进一行, 也可以接受.

```
def my_method(self, first_var: int) -> int:
    ...
```

若最后一个参数加上返回值的类型注解太长, 也可以换行并添加 4 格缩进. 添加换行符时, 建议每个参数和返回值都在单独的一行里, 并且右括号和 `def` 对齐.

正确:

```
def my_method(
    self,
    other_arg: MyLongType | None,
) -> tuple[MyLongType1, MyLongType1]:
    ...
```

返回值类型和最后一个参数也可以放在同一行.

可以接受:

```
def my_method(
    self,
    first_var: int,
    second_var: int) -> dict[OtherLongType, MyLongType]:
    ...
```

pylint 也允许你把右括号放在新行上, 与左括号对齐, 但相较而言可读性更差.

错误:

```
def my_method(self,
              other_arg: MyLongType | None,
             ) -> dict[OtherLongType, MyLongType]:
    ...
```

正如上面所有的例子, 尽量不要在类型注解中间换行. 但是有时注解过长以至于一行放不下. 此时尽量保持子类型中间不换行.

```
def my_method(
    self,
    first_var: tuple[list[MyLongType1],
                     list[MyLongType2]],
    second_var: list[dict[
        MyLongType3, MyLongType4]],
) -> None:
    ...
```

若某个名称和对应的类型注解过长, 可以考虑用*别名 (alias)* 代表类型. 下策是在冒号后换行并添加 4 格缩进.

正确:

```
def my_function(
    long_variable_name:
        long_module_name.LongTypeName,
) -> None:
    ...
```

错误:

```
def my_function(
    long_variable_name: long_module_name.
        LongTypeName,
) -> None:
    ...
```

前向声明 (foward declaration)

若需要使用一个尚未定义的类名 (比如想在声明一个类时使用自身的类名), 可以使用 `from __future__ import annotations` 或者字符串来代表类名.

正确:

```
from __future__ import annotations

class MyClass:
    def __init__(self, stack: Sequence[MyClass], item: OtherClass) ->␣
↪None:

class OtherClass:
    ...
```

```
class MyClass:
    def __init__(self, stack: Sequence['MyClass'], item: 'OtherClass') ->␣
↪None:

class OtherClass:
    ...
```

默认值

根据 PEP-008 , **只有**对于同时拥有类型注解和默认值的参数, = 的周围应该加空格.

正确:

```
def func(a: int = 0) -> int:
    ...
```

错误:

```
def func(a:int=0) -> int:
    ...
```

NoneType

在 Python 的类型系统中, `NoneType` 是 “一等” 类型. 在类型注解中, `None` 是 `NoneType` 的别名. 如果一个变量可能为 `None`, 则必须声明这种情况! 你可以使用 `|` 这样的并集 (union) 类型表达式 (推荐在新的 Python 3.10+ 代码中使用) 或者老的 `Optional` 和 `Union` 语法.

应该用显式的 `X | None` 替代隐式声明. 早期的 PEP 484 允许将 `a: str = None` 解释为 `a: str | None = None`, 但这不再是推荐的行为.

正确:

```
# 现代的并集写法.
def modern_or_union(a: str | int | None, b: str | None = None) -> str:
    ...
# 采用 Union / Optional.
def union_optional(a: Union[str, int, None], b: Optional[str] = None) ->␣
↪str:
    ...
```

错误:

```
# 用 Union 代替 Optional.
def nullable_union(a: Union[None, str]) -> str:
    ...
# 隐式 Optional.
def implicit_optional(a: str = None) -> str:
    ...
```

类型别名 (alias)

你可以为复杂的类型声明一个别名. 别名的命名应该采用大驼峰 (例如 `CapWorded`). 若别名仅在当前模块使用, 应在名称前加 _ 代表私有 (例如 `_Private`).

注意下面的： `TypeAlias` 类型注解只能在 3.10 以后的版本使用.

```
from typing import TypeAlias

_LossAndGradient: TypeAlias = tuple[tf.Tensor, tf.Tensor]
ComplexTFMap: TypeAlias = Mapping[str, _LossAndGradient]
```

忽略类型

你可以使用特殊的注释 `# type: ignore` 禁用某一行的类型检查.

`pytype` 有针对特定错误的禁用选项 (类似格式检查器):

```
# pytype: disable=attribute-error
```

标注变量的类型

带类型注解的赋值

如果难以自动推理某个内部变量的类型, 可以用带类型注解的赋值操作来指定类型: 在变量名和值的中间添加冒号和类型, 类似于有默认值的函数参数.

```
a: Foo = SomeUndecoratedFunction()
```

类型注释

你可能在代码仓库中看到这种残留的注释 (在 Python 3.6 之前必须这样写注释), 但是不要再添加 `# type: <类型>` 这样的行尾注释了:

```
a = SomeUndecoratedFunction()  # type: Foo
```

元组还是列表

有类型的列表中只能有一种类型的元素. 有类型的元组可以有相同类型的元素或者若干个不同类型的元素. 后面这种情况多用于注解返回值的类型.

(译者注: 注意这里是指的类型注解中的写法, 实际 python 中,list 和 tuple 都是可以在一个序列中包含不同类型元素的, 当然, 本质其实 list 和 tuple 中放的是元素的引用)

```
a: list[int] = [1, 2, 3]
b: tuple[int, ...] = (1, 2, 3)
c: tuple[int, str, float] = (1, "2", 3.5)
```

类型变量 (type variable)

Python 的类型系统支持 泛型 (generics) . 使用泛型的常见方式是利用类型变量, 例如 `TypeVar` 和 `ParamSpec`.

例如:

```
from collections.abc import Callable
from typing import ParamSpec, TypeVar
_P = ParamSpec("_P")
_T = TypeVar("_T")
...
def next(l: list[_T]) -> _T:
    return l.pop()

def print_when_called(f: Callable[_P, _T]) -> Callable[_P, _T]:
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        print('函数被调用')
        return f(*args, **kwargs)
return inner
```

`TypeVar` 可以有约束条件.

```
AddableType = TypeVar("AddableType", int, float, str)
def add(a: AddableType, b: AddableType) -> AddableType:
    return a + b
```

`AnyStr` 是 `typing` 模块中常用的预定义类型变量. 可以用它注解那些接受 `bytes` 或 `str` 但是必须保持一致的类型.

```
from typing import AnyStr
def check_length(x: AnyStr) -> AnyStr:
    if len(x) <= 42:
        return x
    raise ValueError()
```

(译者注: 这个例子中, x 和返回值必须同时是 `bytes` 或者同时是 `str`.)

类型变量必须有描述性的名称, 除非满足以下所有标准:

1. 外部不可见
2. 没有约束条件

正确:

```
_T = TypeVar("_T")
_P = ParamSpec("_P")
AddableType = TypeVar("AddableType", int, float, str)
AnyFunction = TypeVar("AnyFunction", bound=Callable)
```

错误:

```
T = TypeVar("T")
P = ParamSpec("P")
_T = TypeVar("_T", int, float, str)
_F = TypeVar("_F", bound=Callable)
```

字符串类型

不要在新代码中使用 `typing.Text`. 这种写法只能用于处理 Python 2/3 的兼容问题.

用 `str` 表示字符串/文本数据. 用 `bytes` 处理二进制数据.

```
# 处理文本数据
def deals_with_text_data(x: str) -> str:
    ...
# 处理二进制数据
def deals_with_binary_data(x: bytes) -> bytes:
    ...
```

若一个函数中的字串类型始终一致, 比如上述代码中返回值类型和参数类型相同, 应该使用 AnyStr.

导入类型

为了静态分析和类型检查而导入 `typing` 和 `collections.abc` 模块中的符号时, 一定要导入符号本身. 这样常用的类型注解更简洁, 也符合全世界的习惯. 特别地, 你可以在一行内从 `typing` 和 `collections.abc` 模块中导入多个特定的类, 例如:

```
from collections.abc import Mapping, Sequence
from typing import Any, Generic
```

采用这种方法时, 导入的类会进入本地命名空间, 因此所有 `typing` 和 `collections.abc` 模块中的名称都应该和关键词 (keyword) 同等对待. 你不能在自己的代码中定义相同的名字, 无论你是否采用类型注解. 若类型名和某模块中已有的名称出现冲突, 可以用 `import x as y` 的导入形式:

```
from typing import Any as AnyType
```

只要可行, 就使用内置类型. 利用 Python 3.9 引入的 PEP-585, 可以在类型注解中使用参数化的容器类型.

```
def generate_foo_scores(foo: set[str]) -> list[float]:
    ...
```

注意: Apache Beam 的用户应该继续导入 `typing` 模块提供的参数化容器类型.

```
from typing import Set, List

# 只有在你使用了 Apache Beam 这样没有为 PEP 585 更新的代码, 或者你的
# 代码需要在 Python 3.9 以下版本中运行时, 才能使用这种旧风格.
def generate_foo_scores(foo: Set[str]) -> List[float]:
    ...
```

有条件的导入

仅在一些特殊情况下, 比如在运行时必须避免导入类型检查所需的模块, 才能有条件地导入. 不推荐这种写法. 替代方案是重构代码, 使类型检查所需的模块可以在顶层导入.

可以把仅用于类型注解的导入放在 `if TYPE_CHECKING:` 语句块内.

1. 在类型注解中, 有条件地导入的类型必须用字符串表示, 这样才能和 Python 3.6 之前的代码兼容. 因为 Python 3.6 之前真的会对类型注解求值.
2. 只有那些仅仅用于类型注解的实例才能有条件地导入, 别名也是如此. 否则会引发运行时错误, 因为运行时不会导入这些模块.
3. 有条件的导入语句应紧随所有常规导入语句之后.
4. 有条件的导入语句之间不能有空行.
5. 和常规导入一样, 请对有条件的导入语句排序.

```
import typing
if typing.TYPE_CHECKING:
    import sketch
def f(x: "sketch.Sketch"): ...
```

循环依赖

若类型注解引发了循环依赖, 说明代码可能存在问题. 这样的代码适合重构. 虽然技术上我们可以支持循环依赖, 但是很多构建系统 (build system) 不支持.

可以用 `Any` 替换引起循环依赖的模块. 起一个有意义的别名, 然后使用模块中的真实类型名 (Any 的任何属性依然是 Any). 定义别名的语句应该和最后一行导入语句之间间隔一行.

```
from typing import Any

some_mod = Any  # 因为 some_mod.py 导入了我们的模块.
...
```

(continues on next page)

(continued from previous page)

```
def my_method(self, var: "some_mod.SomeType") -> None:
    ...
```

泛型 (generics)

在注解类型时, 尽量为泛型类型填入类型参数. 否则, 泛型参数默认为 Any .

正确:

```
def get_names(employee_ids: Sequence[int]) -> Mapping[int, str]:
    ...
```

错误:

```
# 这表示 get_names(employee_ids: Sequence[Any]) -> Mapping[Any, Any]
def get_names(employee_ids: Sequence) -> Mapping:
    ...
```

如果泛型类型的参数的确应该是 Any, 请显式地标注, 不过注意 TypeVar 很可能更合适.

错误:

```
def get_names(employee_ids: Sequence[Any]) -> Mapping[Any, str]:
    """ 返回员工 ID 到员工名的映射."""
```

正确:

```
_T = TypeVar('_T')
def get_names(employee_ids: Sequence[_T]) -> Mapping[_T, str]:
    """ 返回员工 ID 到员工名的映射."""
```

4.5 临别赠言

务必保持一致性.

编辑代码时, 请花几分钟观察一下周边代码的风格. 如果这些代码在所有运算符的周围加上了空格, 那么你也应该这样做. 如果这些代码的注释都用井号形成的框包围起来, 那么你的注释也要用井号形成的框包起来.

制定风格指南是为了像字典一样让代码有章可循. 这样人们可以专注于”写什么”, 而不是纠结”怎么写”. 我们在这里列出的全局规范就像字典, 但是局部的规范同样重要. 如果你添加的代码和周围原有的代码大相径庭, 就会打乱读者的阅读节奏. 不要这样.

CHAPTER 5

Shell 风格指南 - 内容目录

Contents

5.1 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

5.2 背景

5.2.1 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。

5.2.2 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。

5.3 Shell 文件和解释器调用

5.3.1 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。

5.3.2 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` 。

5.4 环境

5.4.1 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
```

5.5 注释

5.5.1 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.
```

5.5.2 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() {
  ...
}
```

5.5.3 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。

5.5.4 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 ####)
```

5.6 格式

5.6.1 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。

5.6.2 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
```

(continues on next page)

(continued from previous page)

```
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
  long string."
```

5.6.3 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
```

5.6.4 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
```

(continues on next page)

(continued from previous page)

```
    mkdir -p "${dir}/${ORACLE_SID}"
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  fi
done
```

5.6.5 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' ;;
```

(continues on next page)

(continued from previous page)

```
    f) files="${OPTARG}" ;;
    v) verbose='true' ;;
    *) error "Unexpected option ${flag}" ;;
  esac
done
```

5.6.6 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.
```

(continues on next page)

(continued from previous page)

```
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"
```

5.6.7 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"
```

(continues on next page)

(continued from previous page)

```
# 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 "$#, $@")
```

5.7 特性及错误

5.7.1 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)")"
```

(continues on next page)

(continued from previous page)

```
# This is not:
var="`command \`command1\``"
```

5.7.2 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
```

5.7.3 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，而不是填充字符，使得代码更易于阅读。

```
# Do this:
if [[ "${my_var}" = "some_string" ]]; then
  do_something
fi
```

(continues on next page)

(continued from previous page)

```
# -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
```

5.7.4 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:
```

(continues on next page)

(continued from previous page)

```
psa@bilby$ rm -v ./*
removed `./-f'
removed `./-r'
rm: cannot remove `./somedir': Is a directory
removed `./somefile'
```

5.7.5 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)"
```

5.7.6 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
```

5.8 命名约定

5.8.1 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
```

(continues on next page)

(continued from previous page)

```
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。

5.8.2 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
```

5.8.3 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
```

5.8.4 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` 。

5.8.5 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，所以在使用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
```

5.8.6 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
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

5.8.7 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。

5.8.8 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。

5.9 调用命令

5.9.1 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
```

(continues on next page)

(continued from previous page)

```
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
```

5.9.2 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/')"
```

5.10 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。

CHAPTER 6

Javascript 风格指南 - 内容目录

6.1 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。

6.2 Javascript 语言规范

6.2.1 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。

6.2.2 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

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 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。

6.2.3 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
```

6.2.4 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。

6.2.5 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 ，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() {}
}
```

6.2.6 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。

6.2.7 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。坏的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。

6.2.8 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）。

6.2.9 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。

6.2.10 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 中的 `goog.inherits()` 或类似的东西。

```
function D() {
    goog.base(this)
}
goog.inherits( D, B );

D.prototype.method =function() {
    ...
};
```

6.2.11 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 。

6.2.12 删除

请使用 `this.foo = null` 。

```
o.prototype.dispose = function() {
    this.property_ = null;
};
```

而不是：

```
Foo.prototype.dispose = function() {
    delete his.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关键字列表删除一个属性，或改变 `if (key in obj)` 结果。

6.2.13 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 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 */ }
}
```

6.2.14 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
```

6.2.15 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` 。

6.2.16 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）、或者其他的对象（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）

6.2.17 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
```

(continues on next page)

(continued from previous page)

```
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]);
    }
}
```

6.2.18 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）。

6.2.19 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 ' +
```

(continues on next page)

(continued from previous page)

```
  '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.';
```

6.2.20 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
};
```

6.2.21 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。

6.2.22 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。

6.3 Javascript 风格规范

6.3.1 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 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 和 Dojo toolkit 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 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上引起混淆，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。

6.3.2 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。

6.3.3 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。

6.3.4 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。

6.3.5 代码格式

我们原则上遵循 C++ 格式规范 ，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,
```

(continues on next page)

(continued from previous page)

```
  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。
```

(continues on next page)

(continued from previous page)

```
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(
      veryLongFunctioNameE(
          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 可用于在使用 the Closure Library 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 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(0xfff)) +
                    '';

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();
```

6.3.6 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。

6.3.7 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';
```

6.3.8 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 。

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;
```

(continues on next page)

(continued from previous page)

```
};

/** @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）。

6.3.9 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 。

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。	
实例类型	{Object} 实例对象或空。 {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。	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。 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}> 。	
函数类型	{function(string, boolean)} 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。	
函数返回类型	{function(): number} 一个函数没有参数并返回一个数字。	指定函数的返回类型。	

JavaScript 中的类型

类型举例	取值举例	描述

number	`1`
	`1.0`
	`-5`
	`1e5`
	`Math.PI`	

Number	`new Number(true)`	Number 对象

string	`'Hello'`
	`"World"`
	`String(42)`
		字符串

String	`new String('Hello')` `new String(42)`	String 对象

boolean	`true`
	`false`
	`Boolean(0)`
		Boolean 值

Boolean	`new Boolean(true)`	Boolean 对象
RegExp	`new RegExp('hello')` `/world/g`	
Date	`new Date` `new Date()`	
null	`null`	
undefined	`undefined`	

void	`function f() {`
	`  return;`
	`}`
		没有返回值

Array	`['foo', 0.3, null]` `[]`	无类型数组

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; });
```

6.3.10 注释

使用 JSDoc。

我们使用 c++ 的注释风格 。所有的文件、类、方法和属性都应该用合适的 JSDoc 的 标签 和 类型 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，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>` ， `<strong>` ， `<ul>` ， `<ol>` ， `<li>` ， `<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 timestamp
```

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 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 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 ，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 occured.
*/
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

标签	模板及实例	描述

@author	@author username@google.com (first last)
	例如：
	`/**`
	`* @fileoverview Utilities for handling textareas.`
	`* @author kuth@google.com (Uthur Pendragon)`
	`*/`
		说明文件的作者是谁，一般只会在 `@fileoverview` 里用到。

@code	{@code …}
	例如：
	`/**`
	• Moves to the next position in the selection.
	• Throws {@code goog.iter.StopIteratio when it
	• passes the end of the range.
	• @return {Node} The node at the next position.
	*/
	goog.dom.RangeIterator.pr = function() {
	// …
	};
		表示这是一段代码，他能在文档中正确的格式化。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@const	@const @const {type}
	例如：
	`/** @const \*/ var MY_BEER = 'stout';`
	`/**`
	`* My namespace's favorite kind of beer.`
	`* @const {string}`
	`*/`
	`mynamespace.MY_BEER = 'stout';`
	`/** @const \*/ MyClass.MY_BEER = 'stout';`
	`/**`
	`* Initializes the request.`
	`* @const`
	`*/`
	`mynamespace.Request.prototype.initialize = function() {`
	`  // This method cannot be overriden in a subclass.`
	`}`
		说明变量或者属性是只读的，适合内联。
		标记为 @const 的变量是不可变的。如果变量或属性试图覆盖他的值，那么 js 编译器会给出警告。
		如果某一个值可以清楚地分辨出是不是常量，可以省略类型声明。变量附加的注释是可选的。
		当一个方法被标记为 @const ，意味着这个方法不仅不可以被覆盖，而且也不能在子类中重写。
		@const 的更多信息，请看 常量 部分

@constructor	@constructor
	例如：
	`/**`
	`* A rectangle.`
	`* @constructor`
	`*/`
	`function GM_Rect() {`
	`  ...`
	`}`
		在一个类的文档中表示构造函数。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@define	@define {Type} description
	例如：
	`/** @define {boolean}␣↪*/`
	`var TR_FLAGS_ENABLE_↪DEBUG = true;`
	
	`/** @define {boolean}␣↪*/`
	`goog.userAgent.ASSUME_↪IE = false;`
		指明一个在编译时可以被覆盖的常量。
		在这个例子中，编译器标志 `--define='goog.userAgent.ASSUME_IE=true'` 表明在构建文件的时侯变量 `goog.userAgent.ASSUME_IE` 可以被赋值为 `true` 。

@deprecated	@deprecated Description
	例如：
	`/**`
	`* Determines whether␣↪a node is a field.`
	`* @return {boolean}␣↪True if the␣↪contents of`
	`*  the element are␣↪editable, but the␣↪element`
	`*  itself is not.`
	`* @deprecated Use␣↪isField().`
	`*/`
	`BN_EditUtil.↪isTopEditableField␣↪= function(node) {`
	`  // ...`
	`};`
		说明函数、方法或者属性已经不可用，常说明替代方法或者属性。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@dict	@dict Description
	例如:
	`/**`
	`* @constructor`
	`* @dict`
	`*/`
	`function Foo(x) {`
	`  this['x'] = x;`
	`}`
	`var obj = new␣`
	`↪Foo(123);`
	`var num = obj.x;  //␣`
	`↪warning`
	`(/** @dict \*/ { x: 1␣`
	`↪}).x = 123;  //␣`
	`↪warning`
		当构造函数 (例子里的 Foo) 被标记为 `@dict` ，你只能使用括号表示法访问 `Foo` 的属性。这个注释也可以直接使用对象表达式。

@enum	@enum {Type}
	例如:
	`/**`
	`* Enum for tri-state␣`
	`↪values.`
	`* @enum {number}`
	`*/`
	`project.TriState = {`
	`  TRUE: 1,`
	`  FALSE: -1,`
	`  MAYBE: 0`
	`};`	

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@export	@export
	例如：
	`/** @export */`
	`foo.MyPublicClass.prototype.myPublicMethod = function() {`
	`  // ...`
	`};`
		对于例子中的代码，当编译到 `--generate_exports` 标记时，将会产生以下代码：
		`goog.exportSymbol('foo.MyPublicClass.prototype.myPublicMethod', foo.MyPublicClass.prototype.myPublicMethod);`
		也就是输出了没有编译的代码。使用 @export 标签时，应该：
		1. 包含 `//javascript/closure/base.js`，或者
		2. 同时定义 `goog.exportSymbol` 和 `goog.exportProperty` 并且要使用相同的调用方法。

@expose	@expose
	例如：
	`/** @expose */`
	`MyClass.prototype.exposedProperty = 3;`
		声明一个公开的属性，表示这个属性不可以被删除、重命名或者由编译器进行优化。相同名称的属性也不能由编译器通过任何方式进行优化。`@expose` 不可以出现在代码库里，因为他会阻止这个属性被删除。

@extends	@extends Type @extends {Type}
	例如：
	`/**`
	`  * Immutable empty node list.`
	`  * @constructor`
	`  * @extends goog.ds.BasicNodeLi`
	`  */`
	`goog.ds.EmptyNodeList = function() {`
	`  ...`
	`};`
		和 `@constructor` 一起使用，表示从哪里继承过来的。类型外的大括号是可选的。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述
@externs	@externs 例如: /** • @fileoverview This is an externs file. • @externs */ var document;	声明一个外部文件。
@fileoverview	@fileoverview Description 例如: /** • @fileoverview Utilities for doing things that require this very long • but not indented comment. • @author kuth@google.com (Uthur Pendragon) */	使注释提供文件级别的信息。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@implements	@implements Type @implements {Type}
	例如：
	/**
	• A shape.
	• @interface
	*/
	function Shape() {};
	Shape.prototype.draw = function() {};
	/**
	• @constructor
	• @implements {Shape}
	*/
	function Square() {};
	Square.prototype.draw = function() {
	…
	};
		使用 `@constructor` 来表示一个类实现了某个接口。类型外的大括号是可选的。

@inheritDoc	@inheritDoc
	例如：
	`/** @inheritDoc */`
	`project.SubClass.prototype.toString() {`
	`  // ...`
	`};`
		已废弃。使用 @override 代替
		表示一个子类中的方法或者属性覆盖父类的方法或者属性，并且拥有相同的文档。注意，`@inheritDoc` 等同 `@override`

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@interface	@interface
	例如：
	`/**`
	`* A shape.`
	`* @interface`
	`*/`
	`function Shape() {};`
	`Shape.prototype.draw = function() {};`
	
	`/**`
	`* A polygon.`
	`* @interface`
	`* @extends {Shape}`
	`*/`
	`function Polygon() {};`
	`Polygon.prototype.getSides = function() {};`
		表示一个函数定义了一个接口。

@lends	@lends objectName @lends {objectName}
	例如：
	`goog.object.extend(`
	`  Button.prototype,`
	`  /** @lends {Button.prototype} */ {`
	`  isButton: function() { return true; }`
	`  });`
		表示对象的键是另外一个对象的属性。这个标记只能出现在对象字面量中。
		注意，括号中的名称和其他标记中的类型名称不一样，它是一个对象名，表明是从哪个对象“借过来”的属性。例如，`@type {Foo}` 意味着 Foo 的一个实例，但是 `@lends {Foo}` 意味着“Foo 构造函数”.
		JSDoc Toolkit docs 中有关于更多此标记的信息。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@license or @preserve	@license Description
	例如：
	`/**`
	`* @preserve Copyright 2009 SomeThirdParty.`
	`* Here is the full license text and copyright`
	`* notice for this file. Note that the notice can span several`
	`* lines and is only terminated by the closing star and slash:`
	`*/`
		由 `@licenseor` 或 `@preserve` 标记的内容，会被编译器保留并放到文件的顶部。
		这个标记会让被标记的重要内容（例如法律许可或版权文本）原样输出，换行也是。

@noalias	@noalias
	例如：
	`/** @noalias */`
	`function Range() {}`
		用在外部文件当中，告诉编译器，这里的变量或者方法不可以重命名。

@nosideeffects	@nosideeffects
	例如：
	`/** @nosideeffects */`
	`function noSideEffectsFn1() {`
	`  // ...`
	`};`
	`/** @nosideeffects */`
	`var noSideEffectsFn2 = function() {`
	`  // ...`
	`};`
	`/** @nosideeffects */`
	`a.prototype.noSideEffectsFn3 = function() {`
	`  // ...`
	`};`
		用于函数和构造函数，说明调用这个函数没有副作用。如果返回值未被使用，此注释允许编译器移除对该函数的调用。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述
@override	@override 例如： `/** * @return {string} Human-readable representation of project.SubClass. * @override */ project.SubClass.prototype.toString() { // ... };`	表示子类的方法或者属性故意隐藏了父类的方法或属性。如果子类没有其他的文档，方法或属性也会从父类那里继承文档。
@param	@param {Type} varname Description 例如： `/** * Queries a Baz for items. * @param {number} groupNum Subgroup id to query. * @param {string|number|null} term An itemName, * or itemId, or null to search everything. */ goog.Baz.prototype.query = function(groupNum, term) { // ... };`	给方法、函数、构造函数的参数添加文档说明。参数类型 一定要写在大括号里。如果类型被省略，编译器将不做类型检测。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述
@private	@private @private {type} 例如: `/** * Handlers that are listening to this logger. * @private {!Array.<Function>} */ this.handlers\_ = [];`	与方法或属性名结尾使用一个下划线来联合表明该成员是 私有的 。随着工具对 `@private` 的认可，结尾的下划线可能最终被废弃。
@protected	@protected @protected {type} 例如: `/** * Sets the component's root element to the given element. Considered * protected and final. * @param {Element} element Root element for the component. * @protected */ goog.ui.Component.prototype.setElementInternal = function(element) { // ... };`	用来表明成员或属性是``受保护的 <http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#Visibility__private_and_protected_fields_>`_ 。成员或属性应使用没有跟随下划线的名称。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@return	@return {Type} Description
	例如：
	`/**`
	`* @return {string}`
	`↪The hex ID of the`
	`↪last item.`
	`*/`
	`goog.Baz.prototype.`
	`↪getLastId =`
	`↪function() {`
	`  // ...`
	`  return id;`
	`};`
		在方法或函数调用时使用，来说明返回类型。给布尔值写注释时，写成类似 “这个组件是否可见” 比 “如果组件可见则为 true，否则为 false” 要好。如果没有返回值，不使用 `@return` 标签。
		类型 名称必须包含在大括号内。如果省略类型，编译器将不会检查返回值的类型。

@see	@see Link
	例如：
	`/**`
	`* Adds a single item,`
	`↪recklessly.`
	`* @see #addSafely`
	`* @see goog.Collect`
	`* @see goog.`
	`↪RecklessAdder#add`
	`...`
		参考查找另一个类或方法。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@struct	@struct Description
	例如:
	`/**`
	`* @constructor`
	`* @struct`
	`*/`
	`function Foo(x) {`
	`  this.x = x;`
	`}`
	`var obj = new Foo(123);`
	`var num = obj['x'];  // warning`
	`obj.y = "asdf";  // warning`
	
	`Foo.prototype = /** @struct */ {`
	`  method1: function() {}`
	`};`
	`Foo.prototype.method2 = function() {};  // warning`
		当一个构造函数（在本例中 `Foo` ）注释为 `@struct` ，你只能用点符号访问 Foo 对象的属性。此外，Foo 对象创建后不能加新的属性。此注释也可以直接使用于对象字面量。

@supported	@supported Description
	例如:
	`/**`
	`* @fileoverview Event Manager`
	`* Provides an abstracted interface to the`
	`* browsers' event systems.`
	`* @supported So far tested in IE6 and FF1.5`
	`*/`
		用于在文件信息中说明该文档被哪些浏览器支持

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述
@suppress	@suppress {warning1|warning2} 例如: `/** * @suppress {deprecated} */ function f() { deprecatedVersionOfF() }`	标明禁止工具发出的警告。警告类别用 | 分隔。
@template	@template 例如: `/** * @param {function(this:T, ...)} fn * @param {T} thisObj * @param {...*} var_args * @template T */ goog.bind = function(fn, thisObj, var_args) { ... };`	这个注释可以用来声明一个 模板类型名 。

continues on next page

Table 1 – continued from previous page

标签	模板及实例	描述

@this	@this Type @this {Type}
	例如:
	`pinto.chat.RosterWidget.extern('getRosterElement',`
	`/**`
	`* Returns the roster widget element.`
	`* @this pinto.chat.RosterWidget`
	`* @return {Element}`
	`*/`
	`function() {`
	`  return this.getWrappedComponent_().getElement();`
	`});`
		标明一个特定方法在其上下文中被调用的对象类型。用于 this 关键字是从一个非原型方法中使用时

@type	@type Type @type {Type}
	例如:
	`/**`
	• The message hex ID.
	• @type {string}
	`*/`
	var hexId = hexId;
		标识变量，属性或表达式的 类型 。大多数类型不需要大括号，但有些项目为了保持一致性而要求所有类型都使用大括号。

@typedef	@typedef
	例如:
	`/** @typedef {(string|number)} */`
	`goog.NumberLike;`
	`/** @param {goog.NumberLike} x A number or a string. */`
	`goog.readNumber = function(x) {`
	`  ...`
	`}`
		使用此注释来声明一个更 复杂的类型 的别名。

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 ，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

6.3.11 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');
```

6.3.12 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 。

6.3.13 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);
}
```

CHAPTER 7

TypeScript 风格指南

7.1 导言

这份指南基于谷歌内部的 TypeScript 风格指南，但在其基础上做了些许调整，删去了仅适用于谷歌内部的部分内容。这是由于谷歌内部对 TypeScript 的使用方式有着不同于外部其它环境的限制，因此，尽管这些内容对有意向谷歌官方项目提交代码的开发者而言十分有用，却并不一定适合其它的开发环境。

本指南并非定期自动部署，而是由志愿者们根据需求进行维护与更新。

7.1.1 术语说明

指南中对 *必须*、*禁止*、*应当*、*不应*、*可以*等术语的用法遵循 RFC 2119 中的规定。术语 *倾向于*和 *避免*分别对应 *应当*和 *不应*。命令式和声明式的语句与术语 *必须*相对应，表示其内容是规定性的。

7.1.2 指南说明

文中的所有示例 **均非规范性示例**，仅用于对指南中的规范性说明加以演示。也就是说，虽然这些示例遵循谷歌的代码风格，但它们并不一定是唯一符合规范的写法。因此，示例中选用的代码样式不能作为规则强制执行。

7.1.3 翻译信息

上次更新日期

2024 年 02 月 29 日。

译者

- Frank Li

原文链接

Google TypeScript Style Guide

中文版链接

谷歌 TypeScript 风格指南

修订历史

- **2021 年 09 月 02 日：** Frank Li 提交了第一个版本。
- **2024 年 02 月 29 日：** Frank Li 重新修订了所有内容。

7.2 语法规范

7.2.1 标识符

命名规范

在 TypeScript 中，标识符只能使用 ASCII 码表中的字母、数字、下划线与 `(`。因此，合法的标识符可以使用正则表达式 `[\)\w]+` 进行匹配。根据标识符的用途不同，使用的命名法也不同，如下表所示：

命名法	分类
帕斯卡命名法（`UpperCamelCase`）	类、接口、类型、枚举、装饰器、类型参数
驼峰式命名法（`lowerCamelCase`）	变量、参数、函数、方法、属性、模块别名
全大写下划线命名法（`CONSTANT_CASE`）	全局常量、枚举值
私有成员命名法（`#ident`）	不允许使用

缩写

缩写应被视为一个词。例如，应使用 `loadHttpUrl`，而非 `loadHTTPURL`。平台有特殊要求的标识符例外，如 `XMLHttpRequest`。

美元符号 $

一般情况下，标识符不应使用 *$*，除非为了与第三方框架的命名规范保持一致。关于 *$* 的使用，可参见*命名风格* 一节对 `Observable` 类型的说明。

类型参数

形如 `Array<T>` 的类型参数既可以使用单个大写字母（如 `T`），也可以使用帕斯卡命名法（如 `UpperCamelCase`）。

测试用例

无论是在 Closure 库的 `testSuites` 还是 xUnit 风格的测试框架中，都可以使用 _ 作为标识符的分隔符，例如 `testX_whenY_doesZ()`。

_ 前缀与后缀

标识符禁止使用下划线 _ 作为前缀或后缀。这也意味着，禁止使用单个下划线 _ 作为标识符（例如：用来表示未被使用的参数）。

如果需要从数组或元组中取出某个或某几个特定的元素的话，可以在解构语句中插入额外的逗号，忽略掉不需要的元素：

```
const [a, , b] = [1, 5, 10];  // a <- 1, b <- 10
```

导入模块

导入模块的命名空间时使用驼峰命名法(`lowerCamelCase`), 文件名则使用蛇形命名法(`snake_case`)。例如：

```
import * as fooBar from './foo_bar';
```

一些库可能会在导入命名空间时使用某种特定的前缀，这与这里规定的命名规范有所冲突。然而，由于其中的一些库已经被广泛使用，因此遵循它们的特殊规则反而能够获得更好的可读性。这些特例包括：

- jQuery，使用 `$` 前缀。
- three.js，使用 `THREE` 前缀。

常量

常量命名（CONSTANT_CASE）表示某个值不可被修改。它还可以用于虽然技术上可以实现，但是用户不应当试图修改的值，比如并未进行深度冻结（deep frozen）的值。

```
const UNIT_SUFFIXES = {
    'milliseconds': 'ms',
    'seconds': 's',
};
// UNIT_SUFFIXES 使用了常量命名，
// 这意味着用户不应试图修改它，
// 即使它实际上是一个可变的值。
```

这里所说的常量，也包括类中的静态只读属性：

```
class Foo {
    private static readonly MY_SPECIAL_NUMBER = 5;

    bar() {
        return 2 * Foo.MY_SPECIAL_NUMBER;
    }
}
```

其他

如果某个值在程序的整个运行生命周期中会被多次实例化或被用户以任何方式进行修改，则它必须使用驼峰式命名法。

如果某个值是作为某个接口的实现的箭头函数，则它也可以使用驼峰式命名法。

别名

在为一个已有的标识符创建具有局部作用域的别名时，别名的命名方式应当与现有的标识符和现有的命名规范保持一致。声明别名时，应使用 const （如果它是一个变量）或 readonly （如果它是类里的一个字段）。

```
const {Foo} = SomeType;
const CAPACITY = 5;

class Teapot {
    readonly BrewStateEnum = BrewStateEnum;
    readonly CAPACITY = CAPACITY;
}
```

命名风格

TypeScript 中的类型表达了丰富的信息，因此在起名时不应与类型中所携带的信息重复。（关于更多在起名时应避免的内容，可参见谷歌的 Testing Blog。）

这里有几个具体的例子：

- 不要为私有属性或方法名添加下划线 _ 前缀或后缀。
- 不要为可选参数添加 *opt_* 前缀。
  - 关于在存取器中的特例，参见后文*1.6. #include 的路径及顺序* 。
- 除非在项目中已成惯例，否则不要显式地标记接口类型（例如不要使用 `IMyInterface` 或者 `MyFooInterface` ）。在为类添加接口时，接口名称中应包含创建这一接口的原因。（例如，在为类 `TodoItem` 创建一个将其转为 JSON 格式以用于存储或者序列化的接口时，可以将这一接口命名为 `TodoItemStorage` 。）
- 对于 `Observable` 类型的值，通常的惯例是使用 `$` 前缀将其与一般类型的值进行区分，使之不致混淆。各个团队可以在与项目内部的现有做法保持一致的前提下，自行决定是否采用这一做法。

描述性命名

命名应当具有描述性且易于读者理解。不要使用对项目以外的用户而言含糊不清或并不熟悉的缩写，不要通过删减单词中的字母来强行创造缩写。

这一规则的例外是，对不超过十行的作用域中的变量，以及内部 API 的参数，可以使用短变量名（例如 `i` 、`j` 等只有单个字母的变量名）。

7.2.2 文件编码

使用 UTF-8 文件编码。

对于非 ASCII 字符，应使用实际的 Unicode 字符（例如 ∞ ）。对于非输出字符，使用对应的十六进制编码或 Unicode 转义编码（如 `\u221e` ），并添加注释进行说明。

```
// 应当这样做！即使没有注释也十分易懂。
const units = 'μs';

// 应当这样做！对非输出字符进行转义。
const output = '\ufeff' + content;  // 字节顺序标记（Byte Order Mark，BOM）
```

```
// 不要这样做！即使加上注释也不太好读，而且容易出错。
const units = '\u03bcs'; // Greek letter mu, 's'

// 不要省略注释！读者在缺少注释的情况下很难理解这个字符的含义。
const output = '\ufeff' + content;
```

7.2.3 注释与文档

用 JSDoc 还是注释？

TypesScript 中有两种类型的注释：JSDoc `/** ... */` 和普通注释 `// ...` 或者 `/* ... */`。

- 对于文档，也就是用户应当阅读的注释，使用 `/** JSDoc */`。
- 对于实现说明，也就是只和代码本身的实现细节有关的注释，使用 `// 行注释`。

JSDoc 注释能够为工具（例如编辑器或文档生成器）所识别，而普通注释只能供人阅读。

JSDoc 规范

JSDoc 的规范大部分遵循 JavaScript 风格指南中的规定。具体地说，遵循 JavaScript 风格指南中注释 一节的规则。本节的剩余部分只对与这些规则不一致的部分进行说明。

对所有导出的顶层模块进行注释

使用 `/** JSDoc */` 注释为代码的用户提供信息。这些注释应当言之有物，切忌仅仅将属性名或参数名重抄一遍。如果代码的审核人认为某个属性或方法的作用不能从它的名字上一目了然地看出来的话，这些属性和方法同样应当使用 `/** JSDoc */` 注释添加说明文档，无论它们是否被导出，是公开还是私有的。

省略对于 TypeScript 而言多余的注释

例如，不要在 `@param` 或 `@return` 注释中声明类型，不要在使用了 `implements`、`enum`、`private` 等关键字的地方添加 `@implements`、`@enum`、`@private` 等注释。

不要使用 `@override`

不要在 TypeScript 代码中使用 `@override` 注释。`@override` 并不会被编译器视为强制性约束，这会导致注释与实现上的不一致性。如果纯粹为了文档添加这一注释，反而令人困惑。

注释必须言之有物

虽然大多数情况下文档对代码十分有益，但对于那些并不用于导出的符号，有时其函数或参数的名称与类型便足以描述自身了。

注释切忌照抄参数类型和参数名，如下面的反面示例：

```
// 不要这样做！这个注释没有任何有意义的内容。
/** @param fooBarService Foo 应用的 Bar 服务 */
```

因此，只有当需要添加额外信息时才使用 `@param` 和 `@return` 注释，其它情况下直接省略即可。

```
/**
 * 发送 POST 请求，开始煮咖啡
 * @param amountLitres 煮咖啡的量，注意和煮锅的尺寸对应！
 */
brew(amountLitres: number, logger: Logger) {
    // ...
}
```

参数属性注释

通过为构造函数的参数添加访问限定符，参数属性同时创建了构造函数参数和类成员。例如，如下的构造函数

```
class Foo {
    constructor(private readonly bar: Bar) { }
}
```

为 Foo 类创建了 Bar 类型的成员 bar 。

如果要为这些成员添加文档，应使用 JSDoc 的 @param 注释，这样编辑器会在调用构造函数和访问属性时显示对应的文档描述信息。

```
/** 这个类演示了如何为参数属性添加文档 */
class ParamProps {
    /**
     * @param percolator 煮咖啡所用的咖啡壶。
     * @param beans 煮咖啡所用的咖啡豆。
     */
    constructor(
        private readonly percolator: Percolator,
        private readonly beans: CoffeeBean[]) {}
}
```

```
/** 这个类演示了如何为普通成员添加文档 */
class OrdinaryClass {
    /** 下次调用 brew() 时所用的咖啡豆。 */
    nextBean: CoffeeBean;

    constructor(initialBean: CoffeeBean) {
        this.nextBean = initialBean;
    }
}
```

函数调用注释

如果有需要，可以在函数的调用点使用行内的 `/* 块注释 */` 为参数添加文档，或者使用字面量对象为参数添加名称并在函数声明中进行解构。注释的格式和位置没有明确的规定。

```
// 使用行内块注释为难以理解的参数添加说明：
new Percolator().brew(/* amountLitres= */ 5);

// 或者使用字面量对象为参数命名，并在函数 brew 的声明中将参数解构：
new Percolator().brew({amountLitres: 5});
```

```
/** 一个古老的咖啡壶 {@link CoffeeBrewer} */
export class Percolator implements CoffeeBrewer {
    /**
     * 煮咖啡。
     * @param amountLitres 煮咖啡的量，注意必须和煮锅的尺寸对应！
     */
    brew(amountLitres: number) {
        // 这个实现煮出来的咖啡味道差极了，不管了。
        // TODO(b/12345): 优化煮咖啡的过程。
    }
}
```

将文档置于装饰器之前

文档、方法或者属性如果同时具有装饰器（例如 `@Component`）和 JSDoc 注释，应当将 JSDoc 置于装饰器之前。

禁止将 JSDoc 置于装饰器和被装饰的对象之间。

```
// 不要这样做！JSDoc 被放在装饰器 @Component 和类 FooComponent 中间了！
@Component({
    selector: 'foo',
    template: 'bar',
})
/** 打印 "bar" 的组件。 */
export class FooComponent {}
```

应当将 JSDoc 置于装饰器之前。

```
/** 打印 "bar" 的组件。 */
@Component({
    selector: 'foo',
    template: 'bar',
})
export class FooComponent {}
```

7.3 语言特性

7.3.1 可见性

限制属性、方法以及类型的可见性有助于代码解耦合。因此：

- 应当尽可能限制符号的可见性。
- 可以将私有方法在同一文件中改写为独立于所有类以外的内部函数，并将私有属性移至单独的内部类中。
- 在 TypeScript 中，符号默认的可见性即为 `public`，因此，除了在构造函数中声明公开（`public`）且非只读（`readonly`）的参数属性之外，不要使用 `public` 修饰符。

```
class Foo {
    public bar = new Bar();  // 不要这样做！不需要 public 修饰符！

    constructor(public readonly baz: Baz) {}  // 不要这样做！readonly 修饰符已经表明了␣
↪baz 是默认 public 的属性，因此不需要 public 修饰符！
}
```

```
class Foo {
    bar = new Bar();  // 应当这样做！将不需要的 public 修饰符省略！

    constructor(public baz: Baz) {}  // 可以这样做！公开且非只读的参数属性允许使用 public␣
↪修饰符！
}
```

关于可见性，还可参见导出可见性一节。

7.3.2 构造函数

调用构造函数时必须使用括号，即使不传递任何参数。

```
// 不要这样做！
const x = new Foo;

// 应当这样做！
const x = new Foo();
```

没有必要提供一个空的或者仅仅调用父类构造函数的构造函数。在 ES2015 标准中，如果没有为类显式地提供构造函数，编译器会提供一个默认的构造函数。但是，含有参数属性、访问修饰符或参数装饰器的构造函数即使函数体为空也不能省略。

```
// 不要这样做！没有必要声明一个空的构造函数！
class UnnecessaryConstructor {
    constructor() {}
}
```

```
// 不要这样做！没有必要声明一个仅仅调用基类构造函数的构造函数！
class UnnecessaryConstructorOverride extends Base {
    constructor(value: number) {
        super(value);
    }
}
```

```
// 应当这样做！默认构造函数由编译器提供即可！
class DefaultConstructor {
}

// 应当这样做！含有参数属性的构造函数不能省略！
class ParameterProperties {
    constructor(private myService) {}
}

// 应当这样做！含有参数装饰器的构造函数不能省略！
class ParameterDecorators {
    constructor(@SideEffectDecorator myService) {}
}

// 应当这样做！私有的构造函数不能省略！
class NoInstantiation {
    private constructor() {}
}
```

7.3.3 类成员

#private 语法

不要使用 #private 私有字段（又称私有标识符）语法声明私有成员。

```
// 不要这样做！
class Clazz {
    #ident = 1;
}
```

而应当使用 TypeScript 的访问修饰符。

```
// 应当这样做！
class Clazz {
    private ident = 1;
}
```

为什么？因为私有字段语法会导致 TypeScipt 在编译为 JavaScript 时出现体积和性能问题。同时，ES2015 之前的标准都不支持私有字段语法，因此它限制了 TypeScript 最低只能被编译至 ES2015。另外，在进行静态类型和可见性检查时，私有字段语法相比访问修饰符并无明显优势。

使用 readonly

对于不会在构造函数以外进行赋值的属性，应使用 `readonly` 修饰符标记。这些属性并不需要具有深层不可变性。

参数属性

不要在构造函数中显式地对类成员进行初始化。应当使用 TypeScript 的 参数属性 语法。

```
// 不要这样做！重复的代码太多了！
class Foo {
    private readonly barService: BarService;

    constructor(barService: BarService) {
        this.barService = barService;
    }
}
```

```
// 应当这样做！简洁明了！
class Foo {
    constructor(private readonly barService: BarService) {}
}
```

如果需要为参数属性添加文档，应使用 JSDoc 的 `@param` 标签，参见参数属性注释 一节。

字段初始化

如果某个成员并非参数属性，应当在声明时就对其进行初始化，这样有时可以完全省略掉构造函数。

```
// 不要这样做！没有必要单独把初始化语句放在构造函数里！
class Foo {
    private readonly userList: string[];
    constructor() {
        this.userList = [];
    }
}
```

```
// 应当这样做！省略了构造函数！
class Foo {
    private readonly userList: string[] = [];
}
```

用于类的词法范围之外的属性

如果一个属性被用于它们所在类的词法范围之外，例如用于模板（template）的 AngularJS 控制器（controller）属性，则禁止将其设为 `private`，因为显然这些属性是用于外部的。

对于这类属性，应当将其设为 `public`，如果有需要的话也可以使用 `protected`。例如，Angular 和 Polymer 的模板属性应使用 `public`，而 AngularJS 应使用 `protected`。

此外，禁止在 TypeScript 代码中使用 `obj['foo']` 语法绕过可见性限制进行访问。

为什么?

如果一个属性被设为 `private`，就相当于向自动化工具和读者声明对这个属性的访问局限于类的内部。例如，用于查找未被使用的代码的工具可能会将一个私有属性标记为未使用，即使在其它文件中有代码设法绕过了可见性限制对其进行访问。

虽然 `obj['foo']` 可以绕过 TypeScript 编译器对可见性的检查，但是这种访问方法可能会由于调整了构建规则而失效。此外，它也违反了后文中所提到的*优化属性访问的兼容性* 规则。

取值器与设值器（存取器）

可以在类中使用存取器，其中取值器方法必须是纯函数（即结果必须是一致稳定的，且不能有副作用）。存取器还可以用于隐藏内部复杂的实现细节。

```
class Foo {
    constructor(private readonly someService: SomeService) {}

    get someMember(): string {
        return this.someService.someVariable;
    }

    set someMember(newValue: string) {
        this.someService.someVariable = newValue;
    }
}
```

如果存取器被用于隐藏类内部的某个属性，则被隐藏的属性应当以诸如 `internal` 或 `wrapped` 此类的完整单词作为前缀或后缀。在使用这些私有属性时，应当尽可能地通过存取器进行访问。取值器和设值器二者至少要有一个是非平凡的，也就是说，存取器不能只用于传递属性值，更不能依赖这种存取器对属性进行隐藏。这种情况下，应当直接将属性设为 `public`。对于只有取值器没有设值器的属性，则应当考虑直接将其设为 `readonly`。

```
class Foo {
    private wrappedBar = '';
    get bar() {
        return this.wrappedBar || 'bar';
    }

    set bar(wrapped: string) {
```

(continues on next page)

(continued from previous page)

```
        this.wrappedBar = wrapped.trim();
    }
}
```

```
class Bar {
    private barInternal = '';
    // 不要这样做！取值器和设值器都没有任何逻辑，这种情况下应当直接将属性 bar 设为 public。
    get bar() {
        return this.barInternal;
    }

    set bar(value: string) {
        this.barInternal = value;
    }
}
```

7.3.4 原始类型与封装类

在 TypeScript 中，不要实例化原始类型的封装类，例如 `String` 、`Boolean` 、`Number` 等。封装类有许多不合直觉的行为，例如 `new Boolean(false)` 在布尔表达式中会被求值为 `true`。

```
// 不要这样做！
const s = new String('hello');
const b = new Boolean(false);
const n = new Number(5);
```

```
// 应当这样做！
const s = 'hello';
const b = false;
const n = 5;
```

7.3.5 数组构造函数

在 TypeScript 中，禁止使用 `Array()` 构造函数（无论是否使用 `new` 关键字）。它有许多不合直觉又彼此矛盾的行为，例如：

```
// 不要这样做！同样的构造函数，其构造方式却却完全不同！
const a = new Array(2); // 参数 2 被视作数组的长度，因此返回的结果是 [undefined,␣
↪undefined]
const b = new Array(2, 3); // 参数 2, 3 被视为数组中的元素，返回的结果此时变成了 [2, 3]
```

应当使用方括号对数组进行初始化，或者使用 `from` 构造一个具有确定长度的数组：

```
const a = [2];
const b = [2, 3];

// 等价于 Array(2):
const c = [];
c.length = 2;

// 生成 [0, 0, 0, 0, 0]
Array.from<number>({length: 5}).fill(0);
```

7.3.6 强制类型转换

在 TypeScript 中，可以使用 `String()` 和 `Boolean()` 函数（注意不能和 `new` 一起使用!）、模板字符串和 `!!` 运算符进行强制类型转换。

```
const bool = Boolean(false);
const str = String(aNumber);
const bool2 = !!str;
const str2 = `result: ${bool2}`;
```

不建议通过字符串连接操作将类型强制转换为 `string`，这会导致加法运算符两侧的运算对象具有不同的类型。

在将其它类型转换为数字时，必须使用 `Number()` 函数，并且，在类型转换有可能失败的场合，必须显式地检查其返回值是否为 `NaN`。

Tip: `Number('')`、`Number(' ')` 和 `Number('\t')` 返回 `0` 而不是 `NaN`。`Number('Infinity')` 和 `Number('-Infinity')` 分别返回 `Infinity` 和 `-Infinity`。这些情况可能需要特殊处理。

```
const aNumber = Number('123');
if (isNaN(aNumber)) throw new Error(...);  // 如果输入字符串有可能无法被解析为数字，就需要处理返回 NaN 的情况。
assertFinite(aNumber, ...);                // 如果输入字符串已经保证合法，可以在这里添加断言。
```

禁止使用一元加法运算符 `+` 将字符串强制转换为数字。用这种方法进行解析有失败的可能，还有可能出现奇怪的边界情况。而且，这样的写法往往成为代码中的坏味道，`+` 在代码审核中非常容易被忽略掉。

```
// 不要这样做!
const x = +y;
```

同样地，代码中也禁止使用 `parseInt` 或 `parseFloat` 进行转换，除非用于解析表示非十进制数字的字符串。因为这两个函数都会忽略字符串中的后缀，这有可能在无意间掩盖了一部分原本会发生错误的情形（例如将 `12 dwarves` 解析成 `12`）。

```
const n = parseInt(someString, 10);  // 无论传不传基数，
const f = parseFloat(someString);     // 都很容易造成错误。
```

对于需要解析非十进制数字的情况，在调用 parseInt 进行解析之前必须检查输入是否合法。

```
if (!/^[a-fA-F0-9]+$/.test(someString)) throw new Error(...);
// 需要解析 16 进制数。
// tslint:disable-next-line:ban
const n = parseInt(someString, 16);  // 只允许在非十进制的情况下使用 parseInt。
```

应当使用 Number() 和 Math.floor 或者 Math.trunc （如果支持的话）解析整数。

```
let f = Number(someString);
if (isNaN(f)) handleError();
f = Math.floor(f);
```

不要在 if 、for 或者 while 的条件语句中显式地将类型转换为 boolean ，因为这里原本就会执行隐式的类型转换。

```
// 不要这样做!
const foo: MyInterface|null = ...;
if (!!foo) {...}
while (!!foo) {...}
```

```
// 应当这样做!
const foo: MyInterface|null = ...;
if (foo) {...}
while (foo) {...}
```

最后，在代码中使用显式和隐式的比较均可。

```
// 显式地和 0 进行比较，没问题!
if (arr.length > 0) {...}

// 依赖隐式类型转换，也没问题!
if (arr.length) {...}
```

7.3.7 变量

必须使用 const 或 let 声明变量。尽可能地使用 const ，除非这个变量需要被重新赋值。禁止使用 var 。

```
const foo = otherValue;  // 如果 foo 不可变，就使用 const。
let bar = someValue;     // 如果 bar 在之后会被重新赋值，就使用 let。
```

与大多数其它编程语言类似，使用 const 和 let 声明的变量都具有块级作用域。与之相反的是，使用 var 声明的变量在 JavaScript 中具有函数作用域，这会造成许多难以理解的 bug，因此禁止在 TypeScript

中使用 var 。

```
// 不要这么做!
var foo = someValue;
```

最后，变量必须在使用前进行声明。

7.3.8 异常

在实例化异常对象时，必须使用 new Error() 语法而非调用 Error() 函数。虽然这两种方法都能够创建一个异常实例，但是使用 new 能够与代码中其它的对象实例化在形式上保持更好的一致性。

```
// 应当这样做!
throw new Error('Foo is not a valid bar.');

// 不要这样做!
throw Error('Foo is not a valid bar.');
```

7.3.9 对象迭代

对对象使用 for (... in ...) 语法进行迭代很容易出错，因为它同时包括了对象从原型链中继承得来的属性。因此，禁止使用裸的 for (... in ...) 语句。

```
// 不要这样做!
for (const x in someObj) {
    // x 可能包括 someObj 从原型中继承得到的属性。
}
```

在对对象进行迭代时，必须使用 if 语句对对象的属性进行过滤，或者使用 for (... of Object.keys(...)) 。

```
// 应当这样做!
for (const x in someObj) {
    if (!someObj.hasOwnProperty(x)) continue;
    // 此时 x 必然是定义在 someObj 上的属性。
}
```

```
// 应当这样做!
for (const x of Object.keys(someObj)) { // 注意: 这里使用的是 for _of_ 语法!
    // 此时 x 必然是定义在 someObj 上的属性。
}
```

```
// 应当这样做!
for (const [key, value] of Object.entries(someObj)) { // 注意: 这里使用的是 for _of_ 语法!
```

(continues on next page)

(continued from previous page)

```
    // 此时 key 必然是定义在 someObj 上的属性。
}
```

7.3.10 容器迭代

不要在数组上使用 `for (... in ...)` 进行迭代。这是一个违反直觉的操作，因为它是对数组的下标而非元素进行迭代（还会将其强制转换为 `string` 类型）！

```
// 不要这样做！
for (const x in someArray) {
    // 这里的 x 是数组的下标！（还是 string 类型的！）
}
```

如果要在数组上进行迭代，应当使用 `for (... of someArr)` 语句或者传统的 `for` 循环语句。

```
// 应当这样做！
for (const x of someArr) {
    // 这里的 x 是数组的元素。
}
```

```
// 应当这样做！
for (let i = 0; i < someArr.length; i++) {
    // 如果需要使用下标，就对下标进行迭代，否则就使用 for/of 循环。
    const x = someArr[i];
    // ...
}
```

```
// 应当这样做！
for (const [i, x] of someArr.entries()) {
    // 上面例子的另一种形式。
}
```

不要使用 `Array.prototype.forEach`、`Set.prototype.forEach` 和 `Map.prototype.forEach`。这些方法会使代码难以调试，还会令编译器的某些检查（例如可见性检查）失效。

```
// 不要这样做！
someArr.forEach((item, index) => {
    someFn(item, index);
});
```

为什么？考虑下面这段代码：

```
let x: string|null = 'abc';
myArray.forEach(() => { x.charAt(0); });
```

从读者的角度看，这段代码并没有什么问题：`x` 没有被初始化为 `null`，并且在被访问之前也没有发生过任何变化。但是对编译器而言，它并不知道传给 `.forEach()` 的闭包 `() => { x.charAt(0); }` 会

被立即执行。因此，编译器有理由认为闭包有可能在之后的某处代码中被调用，而到那时 x 已经被设为 null 。于是，这里出现了一个编译错误。与之等价的 for-of 形式的迭代就不会有任何问题。

读者可以在 这里 对比这两个版本的代码。

在工程实践中，代码路径越复杂、越违背直觉，越容易在进行控制流分析时出现这类问题。

7.3.11 展开运算符

在复制数组或对象时，展开运算符 [...foo]、{...bar} 是一个非常方便的语法。使用展开运算符时，对于同一个键，后出现的值会取代先出现的值。

```
const foo = {
    num: 1,
};

const foo2 = {
    ...foo,
    num: 5,
};

const foo3 = {
    num: 5,
    ...foo,
}

// 对于 foo2 而言，1 先出现，5 后出现。
foo2.num === 5;

// 对于 foo3 而言，5 先出现，1 后出现。
foo3.num === 1;
```

在使用展开运算符时，被展开的值必须与被创建的值相匹配。也就是说，在创建对象时只能展开对象，在创建数组时只能展开可迭代类型。

禁止展开原始类型，包括 null 和 undefined 。

```
// 不要这样做!
const foo = {num: 7};
const bar = {num: 5, ...(shouldUseFoo && foo)}; // 展开运算符有可能作用于 undefined。
```

```
// 不要这样做! 这会创建出一个没有 length 属性的对象 {0: 'a', 1: 'b', 2: 'c'}。
const fooStrings = ['a', 'b', 'c'];
const ids = {...fooStrings};
```

```
// 应当这样做! 在创建对象时展开对象。
const foo = shouldUseFoo ? {num: 7} : {};
const bar = {num: 5, ...foo};
```

(continues on next page)

(continued from previous page)

```
// 应当这样做！在创建数组时展开数组。
const fooStrings = ['a', 'b', 'c'];
const ids = [...fooStrings, 'd', 'e'];
```

7.3.12 控制流语句 / 语句块

多行控制流语句必须使用大括号。

```
// 应当这样做！
for (let i = 0; i < x; i++) {
    doSomethingWith(i);
    andSomeMore();
}
if (x) {
    doSomethingWithALongMethodName(x);
}
```

```
// 不要这样做！
if (x)
    x.doFoo();
for (let i = 0; i < x; i++)
    doSomethingWithALongMethodName(i);
```

这条规则的例外时，能够写在同一行的 `if` 语句可以省略大括号。

```
// 可以这样做！
if (x) x.doFoo();
```

7.3.13 `switch` 语句

所有的 `switch` 语句都必须包含一个 `default` 分支，即使这个分支里没有任何代码。

```
// 应当这样做！
switch (x) {
    case Y:
        doSomethingElse();
        break;
    default:
        // 什么也不做。
}
```

非空语句组（`case ...`）不允许越过分支向下执行（编译器会进行检查）：

```
// 不能这样做!
switch (x) {
    case X:
        doSomething();
        // 不允许向下执行!
    case Y:
        // ...
}
```

空语句组可以这样做：

```
// 可以这样做!
switch (x) {
    case X:
    case Y:
        doSomething();
        break;
    default: // 什么也不做。
}
```

7.3.14 相等性判断

必须使用三等号（===）和对应的不等号（!==）。两等号会在比较的过程中进行类型转换，这非常容易导致难以理解的错误。并且在 JavaScript 虚拟机上，两等号的运行速度比三等号慢。参见 JavaScript 相等表 。

```
// 不要这样做!
if (foo == 'bar' || baz != bam) {
    // 由于发生了类型转换，会导致难以理解的行为。
}
```

```
// 应当这样做!
if (foo === 'bar' || baz !== bam) {
    // 一切都很好!
}
```

例外：和 `null` 字面量的比较可以使用 == 和 != 运算符，这样能够同时覆盖 `null` 和 `undefined` 两种情况。

```
// 可以这样做!
if (foo == null) {
    // 不管 foo 是 null 还是 undefined 都会执行到这里。
}
```

7.3.15 函数声明

使用 `function foo() { ... }` 的形式声明具名函数，包括嵌套在其它作用域中，例如其它函数内部的函数。

不要使用将函数表达式赋值给局部变量的写法（例如 `const x = function() {...};` ）。TypeScript 本身已不允许重新绑定函数，所以在函数声明中使用 `const` 来阻止重写函数是没有必要的。

例外：如果函数需要访问外层作用域的 `this` ，则应当使用将箭头函数赋值给变量的形式代替函数声明的形式。

```
// 应当这样做!
function foo() { ... }
```

```
// 不要这样做!
// 在有上一段代码中的函数声明的情况下，下面这段代码无法通过编译：
foo = () => 3;  // 错误：赋值表达式的左侧不合法。

// 因此像这样进行函数声明是没有必要的。
const foo = function() { ... }
```

请注意这里所说的函数声明（`function foo() {}` ）和下面要讨论的函数表达式（`doSomethingWith(function() {});` ）之间的区别。

顶层箭头函数可以用于显式地声明这一函数实现了一个接口。

```
interface SearchFunction {
    (source: string, subString: string): boolean;
}

const fooSearch: SearchFunction = (source, subString) => { ... };
```

7.3.16 函数表达式

在表达式中使用箭头函数

不要使用 ES6 之前使用 `function` 关键字定义函数表达式的版本。应当使用箭头函数。

```
// 应当这样做!
bar(() => { this.doSomething(); })
```

```
// 不要这样做!
bar(function() { ... })
```

只有当函数需要动态地重新绑定 `this` 时，才能使用 `function` 关键字声明函数表达式，但是通常情况下代码中不应当重新绑定 `this` 。常规函数（相对于箭头函数和方法而言）不应当访问 `this` 。

表达式函数体和代码块函数体

使用箭头函数时，应当根据具体情况选择表达式或者代码块作为函数体。

```
// 使用函数声明的顶层函数。
function someFunction() {
    // 使用代码块函数体的箭头函数，也就是使用 => { } 的函数，没问题：
    const receipts = books.map((b: Book) => {
        const receipt = payMoney(b.price);
        recordTransaction(receipt);
        return receipt;
    });

    // 如果用到了函数的返回值的话，使用表达式函数体也没问题：
    const longThings = myValues.filter(v => v.length > 1000).map(v => String(v));

    function payMoney(amount: number) {
        // 函数声明也没问题，但是不要在函数中访问 this。
    }
}
```

只有在确实需要用到函数返回值的情况下才能使用表达式函数体。

```
// 不要这样做！如果不需要函数返回值的话，应当使用代码块函数体（{ ... }）。
myPromise.then(v => console.log(v));
```

```
// 应当这样做！使用代码块函数体。
myPromise.then(v => {
    console.log(v);
});

// 应当这样做！即使需要函数返回值，也可以为了可读性使用代码块函数体。
const transformed = [1, 2, 3].map(v => {
    const intermediate = someComplicatedExpr(v);
    const more = acrossManyLines(intermediate);
    return worthWrapping(more);
});
```

重新绑定 this

不要在函数表达式中使用 this，除非它们明确地被用于重新绑定 this 指针。大多数情况下，使用箭头函数或者显式指定函数参数都能够避免重新绑定 this 的需求。

```
// 不要这样做！
function clickHandler() {
    // 这里的 this 到底指向什么？
    this.textContent = 'Hello';
```

(continues on next page)

(continued from previous page)

```
}

// 不要这样做！this 指针被隐式地设为 document.body。
document.body.onclick = clickHandler;
```

```
// 应当这样做！在箭头函数中显式地对对象进行引用。
document.body.onclick = () => { document.body.textContent = 'hello'; };

// 可以这样做！函数显式地接收一个参数。
const setTextFn = (e: HTMLElement) => { e.textContent = 'hello'; };
document.body.onclick = setTextFn.bind(null, document.body);
```

使用箭头函数作为属性

通常情况下，类不应该将任何属性初始化为箭头函数。箭头函数属性需要调用函数意识到被调用函数的 this 已经被绑定了，这让 this 的指向变得令人费解，也让对应的调用和引用在形式上看着似乎是不正确的，也就是说，需要额外的信息才能确认这样的使用方式是正确的。在调用实例方法时，必须使用箭头函数的形式（例如 const handler = (x) => { this.listener(x); }; ）。此外，不允许持有或传递实例方法的引用（例如不要使用 const handler = this.listener; handler(x); 的写法）。

Tip: 在一些特殊的情况下，例如需要将函数绑定到模板时，使用箭头函数作为属性是很有用的做法，同时还能令代码的可读性提高。因此，在这些情况下对于这条规则可视具体情况加以变通。此外，事件句柄 一节中有相关讨论。

```
// 不要这样做！
class DelayHandler {
    constructor() {
        // 这里有个问题，回调函数里的 this 指针不会被保存。
        // 因此回调函数里的 this 不再是 DelayHandler 的实例了。
        setTimeout(this.patienceTracker, 5000);
    }
    private patienceTracker() {
        this.waitedPatiently = true;
    }
}
```

```
// 不要这样做！一般而言不应当使用箭头函数作为属性。
class DelayHandler {
    constructor() {
        // 不要这样做！这里看起来就是像是忘记了绑定 this 指针。
        setTimeout(this.patienceTracker, 5000);
    }
    private patienceTracker = () => {
```

(continues on next page)

(continued from previous page)

```
        this.waitedPatiently = true;
    }
}
```

```
// 应当这样做！在调用时显式地处理 this 指针的指向问题。
class DelayHandler {
    constructor() {
        // 在这种情况下，应尽可能使用匿名函数。
        setTimeout(() => {
            this.patienceTracker();
        }, 5000);
    }
    private patienceTracker() {
        this.waitedPatiently = true;
    }
}
```

事件句柄

对于事件句柄，如果它不需要被卸载的话，可以使用箭头函数的形式，例如事件是由类自身发送的情况。如果句柄必须被卸载，则应当使用箭头函数属性，因为箭头函数属性能够自动正确地捕获 `this` 指针，并且能够提供一个用于卸载的稳定引用。

```
// 应当这样做！事件句柄可以使用匿名函数或者箭头函数属性的形式。
class Component {
    onAttached() {
        // 事件是由类本身发送的，因此这个句柄不需要卸载。
        this.addEventListener('click', () => {
            this.listener();
        });
        // 这里的 this.listener 是一个稳定引用，因此可以在之后被卸载。
        window.addEventListener('onbeforeunload', this.listener);
    }
    onDetached() {
        // 这个事件是由 window 发送的。如果不卸载这个句柄，this.listener
        // 会因为绑定了 this 而保存对 this 的引用，从而导致内存泄漏。
        window.removeEventListener('onbeforeunload', this.listener);
    }
    // 使用箭头函数作为属性能够自动地正确绑定 this 指针。
    private listener = () => {
        confirm('Do you want to exit the page?');
    }
}
```

不要在注册事件句柄的表达式中使用 `bind` ，这会创建一个无法卸载的临时引用。

```
// 不要这样做！对句柄使用 bind 会创建一个无法卸载的临时引用。
class Component {
    onAttached() {
        // 这里创建了一个无法卸载的临时引用。
        window.addEventListener('onbeforeunload', this.listener.bind(this));
    }
    onDetached() {
        // 这里的 bind 创建了另一个引用，所以这一行代码实际上没有实现任何功能。
        window.removeEventListener('onbeforeunload', this.listener.bind(this));
    }
    private listener() {
        confirm('Do you want to exit the page?');
    }
}
```

7.3.17 自动分号插入

不要依赖自动分号插入（ASI），必须显式地使用分号结束每一个语句。这能够避免由于不正确的分号插入所导致的 Bug，也能够更好地兼容对 ASI 支持有限的工具（例如 clang-format）。

7.3.18 `@ts-ignore`

不要使用 `@ts-ignore` 。表面上看，这是一个“解决”编译错误的简单方法，但实际上，编译错误往往是由其它更大的问题导致的，因此正确的做法是直接解决这些问题本身。

举例来说，如果使用 `@ts-ignore` 关闭了一个类型错误，那么便很难推断其它相关代码最终会接收到何种类型。对于许多与类型相关的错误，*any 类型* 一节有一些关于如何正确使用 `any` 的有用的建议。

7.3.19 类型断言与非空断言

类型断言（`x as SomeType`）和非空断言（`y!`）是不安全的。这两种语法只能够绕过编译器，而并不添加任何运行时断言检查，因此有可能导致程序在运行时崩溃。

因此，除非有明显或确切的理由，否则 *不应* 使用类型断言和非空断言。

```
// 不要这样做！
(x as Foo).foo();

y!.bar();
```

如果希望对类型和非空条件进行断言，最好的做法是显式地编写运行时检查。

```
// 应当这样做！

// 这里假定 Foo 是一个类。
if (x instanceof Foo) {
```

(continues on next page)

(continued from previous page)

```
    x.foo();
}

if (y) {
    y.bar();
}
```

有时根据代码中的上下文可以确定某个断言必然是安全的。在这种情况下，应当添加注释详细地解释为什么这一不安全的行为可以被接受：

```
// 可以这样做!

// x 是一个 Foo 类型的示例，因为……
(x as Foo).foo();

// y 不可能是 null，因为……
y!.bar();
```

如果使用断言的理由很明显，注释就不是必需的。例如，生成的协议代码总是可空的，但有时根据上下文可以确认其中某些特定的由后端提供的字段必然不为空。在这些情况下应当根据具体场景加以判断和变通。

类型断言语法

类型断言必须使用 `as` 语法，不要使用尖括号语法，这样能强制保证在断言外必须使用括号。

```
// 不要这样做!
const x = (<Foo>z).length;
const y = <Foo>z.length;
```

```
// 应当这样做!
const x = (z as Foo).length;
```

类型断言和对象字面量

使用类型标记（`: Foo`）而非类型断言（`as Foo`）标明对象字面量的类型。在日后对接口的字段类型进行修改时，前者能够帮助程序员发现 Bug。

```
interface Foo {
    bar: number;
    baz?: string;  // 这个字段曾经的名称是“bam”，后来改名为“baz”。
}

const foo = {
    bar: 123,
```

(continues on next page)

(continued from previous page)

```
    bam: 'abc',  // 如果使用类型断言，改名之后这里并不会报错!
} as Foo;

function func() {
    return {
        bar: 123,
        bam: 'abc',  // 如果使用类型断言，改名之后这里也不会报错!
    } as Foo;
}
```

7.3.20 成员属性声明

接口和类的声明必须使用 ; 分隔每个成员声明。

```
// 应当这样做!
interface Foo {
    memberA: string;
    memberB: number;
}
```

为了与类的写法保持一致，不要在接口中使用 , 分隔字段。

```
// 不要这样做!
interface Foo {
    memberA: string,
    memberB: number,
}
```

然而，内联对象类型声明必须使用 , 作为分隔符。

```
// 应当这样做!
type SomeTypeAlias = {
    memberA: string,
    memberB: number,
};

let someProperty: {memberC: string, memberD: number};
```

优化属性访问的兼容性

不要混用方括号属性访问和句点属性访问两种形式。

```
// 不要这样做!
// 必须从两种形式中选择其中一种，以保证整个程序的一致性。
console.log(x['someField']);
console.log(x.someField);
```

代码应当尽可能为日后的属性重命名需求进行优化，并且为所有程序外部的对象属性声明对应的字段。

```
// 应当这样做! 声明一个对应的接口。
declare interface ServerInfoJson {
    appVersion: string;
    user: UserJson;
}
const data = JSON.parse(serverResponse) as ServerInfoJson;
console.log(data.appVersion); // 这里是类型安全的，如果需要重命名也是安全的!
```

优化模块对象导入的兼容性

导入模块对象时应当直接访问对象上的属性，而不要传递对象本身的引用，以保证模块能够被分析和优化。也可以将导入的模块视作命名空间，参见选择模块导入还是解构导入？一节。

```
// 应当这样做!
import {method1, method2} from 'utils';
class A {
    readonly utils = {method1, method2};
}
```

```
// 不要这样做!
import * as utils from 'utils';
class A {
    readonly utils = utils;
}
```

例外情况

这里所提到的优化规则适用于所有的 Web 应用，但不需要强制应用于只运行在服务端的程序。不过，出于代码整洁性的考虑，这里仍然强烈建议声明所有的类型，并且避免混用两种属性访问的形式。

7.3.21 枚举

对于枚举类型，必须使用 `enum` 关键字，但不要使用 `const enum` 。TypeScript 的枚举类型本身就是不可变的，`const enum` 的写法是另一种独立的语言特性，其目的是让枚举对 JavaScript 程序员透明。

7.3.22 debugger 语句

不允许在生产环境代码中添加 `debugger` 语句。

```
// 不要这样做!
function debugMe() {
    debugger;
}
```

7.3.23 装饰器

装饰器以 `@` 为前缀，例如 `@MyDecorator` 。

不要定义新的装饰器，只使用框架中已定义的装饰器，例如：

- Angular（例如 `@Component` 、`@NgModule` 等等）
- Polymer（例如 `@property` 等等）

为什么?

通常情况下，应当避免使用装饰器。这是由于装饰器是一个实验性功能，仍然处于 TC39 委员会的提案阶段，且目前存在已知的无法被修复的 Bug。

使用装饰器时，装饰器必须紧接被装饰的符号，中间不允许有空行。

```
/** JSDoc 注释应当位于装饰器之前 */
@Component({...})  // 装饰器之后不能有空行。
class MyComp {
    @Input() myField: string;  // 字段的装饰器和和字段位于同一行……

    @Input()
    myOtherField: string;  // ……或位于字段之前。
}
```

7.4 代码管理

7.4.1 模块

导入路径

TypeScript 代码必须使用路径进行导入。这里的路径既可以是相对路径，以 `.` 或 `..` 开头，也可以是从项目根目录开始的绝对路径，如 `root/path/to/file` 。

在引用逻辑上属于同一项目的文件时，应使用相对路径 `./foo`，不要使用绝对路径 `path/to/foo`。

应尽可能地限制父层级的数量（避免出现诸如 `../../../` 的路径），过多的层级会导致模块和路径结构难以理解。

```
import {Symbol1} from 'google3/path/from/root';
import {Symbol2} from '../parent/file';
import {Symbol3} from './sibling';
```

用命名空间还是模块？

在 TypeScript 有两种组织代码的方式：命名空间（namespace）和模块（module）。

不允许使用命名空间，在 TypeScript 中必须使用模块（即 ES6 模块）。也就是说，在引用其它文件中的代码时必须以 `import {foo} from 'bar'` 的形式进行导入和导出。

不允许使用 `namespace Foo { ... }` 的形式组织代码。命名空间只能在所用的外部第三方库有要求时才能使用。如果需要在语义上对代码划分命名空间，应当通过分成不同文件的方式实现。

不允许在导入时使用 `require` 关键字（形如 `import x = require('...');`）。应当使用 ES6 的模块语法。

```
// 不要这样做！不要使用命名空间！
namespace Rocket {
    function launch() { ... }
}

// 不要这样做！不要使用 <reference> ！
/// <reference path="..."/>

// 不要这样做！不要使用 require() ！
import x = require('mydep');
```

Tip: TypeScript 的命名空间早期也被称为内部模块并使用 `module` 关键字，形如 `module Foo { ... }`。不要使用这种用法。任何时候都应当使用 ES6 的导入语法。

7.4.2 导出

代码中必须使用具名的导出声明。

```
// Use named exports:
export class Foo { ... }
```

不要使用默认导出，这样能保证所有的导入语句都遵循统一的范式：

```
// 不要这样做！不要使用默认导出！
export default class Foo { ... }
```

为什么？因为默认导出并不为被导出的符号提供一个标准的名称，这增加了维护的难度和降低可读性的风险，同时并未带来明显的益处。如下面的例子所示：

```
// 默认导出会造成如下的弊端
import Foo from './bar';  // 这个语句是合法的。
import Bar from './bar';  // 这个语句也是合法的。
```

具名导出的一个优势是，当代码中试图导入一个并未被导出的符号时，这段代码会报错。例如，假设在 `foo.ts` 中有如下的导出声明：

```
// 不要这样做!
const foo = 'blah';
export default foo;
```

如果在 `bar.ts` 中有如下的导入语句：

```
// 编译错误!
import {fizz} from './foo';
```

会导致编译错误：`error TS2614: Module '"./foo"' has no exported member 'fizz'`。反之，如果在 `bar.ts` 中的导入语句为：

```
// 不要这样做! 这定义了一个多余的变量 fizz!
import fizz from './foo';
```

结果是 `fizz === foo`，这往往不符合预期，且难以调试。

此外，默认导出会鼓励程序员将所有内容全部置于一个巨大的对象当中，这个对象实际上充当了命名空间的角色：

```
// 不要这样做!
export default class Foo {
    static SOME_CONSTANT = ...
    static someHelpfulFunction() { ... }
    ...
}
```

显然，这个文件中具有文件作用域，它可以被用做命名空间。但是，这里创建了第二个作用域——类 `Foo`，这个类在其它文件中具有歧义：它既可以被视为类型，又可以被视为值。

因此，应当使用文件作用域作为实质上的命名空间，同时使用具名的导出声明：

```
// 应当这样做!
export const SOME_CONSTANT = ...
export function someHelpfulFunction()
export class Foo {
    // 只有类 Foo 中的内容
}
```

导出可见性

TypeScript 不支持限制导出符号的可见性。因此，不要导出不用于模块以外的符号。一般来说，应当尽量减小模块的外部 API 的规模。

可变导出

虽然技术上可以实现，但是可变导出会造成难以理解和调试的代码，尤其是对于在多个模块中经过了多次重新导出的符号。这条规则的一个例子是，不允许使用 `export let` 。

```
// 不要这样做!
export let foo = 3;
// 在纯 ES6 环境中，变量 foo 是一个可变值，导入了 foo 的代码会观察到它的值在一秒钟之后发生了改变。
// 在 TypeScript 中，如果 foo 被另一个文件重新导出了，导入该文件的代码则不会观察到变化。
window.setTimeout(() => {
    foo = 4;
}, 1000 /* ms */);
```

如果确实需要允许外部代码对可变值进行访问，应当提供一个显式的取值器。

```
// 应当这样做!
let foo = 3;
window.setTimeout(() => {
    foo = 4;
}, 1000 /* ms */);
// 使用显式的取值器对可变导出进行访问。
export function getFoo() { return foo; };
```

有一种常见的编程情景是，要根据某种特定的条件从两个值中选取其中一个进行导出：先检查条件，然后导出。这种情况下，应当保证模块中的代码执行完毕后，导出的结果就是确定的。

```
function pickApi() {
    if (useOtherApi()) return OtherApi;
    return RegularApi;
}
export const SomeApi = pickApi();
```

容器类

不要为了实现命名空间创建含有静态方法或属性的容器类。

```
// 不要这样做!
export class Container {
    static FOO = 1;
    static bar() { return 1; }
}
```

应当将这些方法和属性设为单独导出的常数和函数。

```
// 应当这样做!
export const FOO = 1;
export function bar() { return 1; }
```

7.4.3 导入

在 ES6 和 TypeScript 中，导入语句共有四种变体：

导入类型	示例	用途
模块	`import * as foo from '...';`	TypeScript 导入方式
解构	`import {SomeThing} from '...';`	TypeScript 导入方式
默认	`import SomeThing from '...';`	只用于外部代码的特殊需求
副作用	`import '...';`	只用于加载某些库的副作用（例如自定义元素）

```
// 应当这样做! 从这两种变体中选择较合适的一种（见下文）。
import * as ng from '@angular/core';
import {Foo} from './foo';

// 只在有需要时使用默认导入。
import Button from 'Button';

// 有时导入某些库是为了其代码执行时的副作用。
import 'jasmine';
import '@polymer/paper-button';
```

选择模块导入还是解构导入？

根据使用场景的不同，模块导入和解构导入分别有其各自的优势。

虽然模块导入语句中出现了通配符 `*`，但模块导入并不能因此被视为其它语言中的通配符导入。相反地，模块导入语句为整个模块提供了一个名称，模块中的所有符号都通过这个名称进行访问，这为代码提供了更好的可读性，同时令模块中的所有符号可以进行自动补全。模块导入减少了导入语句的数量（模块中的所有符号都可以使用），降低了命名冲突的出现几率，同时还允许为被导入的模块提供一个简洁的名称。在从一个大型 API 中导入多个不同的符号时，模块导入语句尤其有用。

解构导入语句则为每一个被导入的符号提供一个局部的名称，这样在使用被导入的符号时，代码可以更简洁。对那些十分常用的符号，例如 Jasmine 的 `describe` 和 `it` 来说，这一点尤其有用。

```
// 不要这样做! 无意义地使用命名空间中的名称使得导入语句过于冗长。
import {TableViewItem, TableViewHeader, TableViewRow, TableViewModel,
TableViewRenderer} from './tableview';
let item: TableViewItem = ...;
```

```
// 应当这样做! 使用模块作为命名空间。
import * as tableview from './tableview';
let item: tableview.Item = ...;
```

```
import * as testing from './testing';

// 所有的测试都只会重复地使用相同的三个函数。
// 如果只需要导入少数几个符号，而这些符号的使用频率又非常高的话，
// 也可以考虑使用解构导入语句直接导入这几个符号（见下文）。
testing.describe('foo', () => {
testing.it('bar', () => {
    testing.expect(...);
    testing.expect(...);
});
});
```

```
// 这样做更好! 为这几个常用的函数提供局部变量名。
import {describe, it, expect} from './testing';

describe('foo', () => {
it('bar', () => {
    expect(...);
    expect(...);
});
});
...
```

重命名导入

在代码中，应当通过使用模块导入或重命名导出解决命名冲突。此外，在需要时，也可以使用重命名导入（例如 `import {SomeThing as SomeOtherThing}`）。

在以下几种情况下，重命名导入可能较为有用：

1. 避免与其它导入的符号产生命名冲突。
2. 被导入符号的名称是自动生成的。
3. 被导入符号的名称不能清晰地描述其自身，需要通过重命名提高代码的可读性，如将 RxJS 的 `from` 函数重命名为 `observableFrom`。

import type 和 export type

不要使用 `import type ... from` 或者 `export type ... from`。

Tip: 这一规则不适用于导出类型定义，如 `export type Foo = ...;`。

```
// 不要这样做!
import type {Foo} from './foo';
export type {Bar} from './bar';
```

应当使用常规的导入语句。

```
// 应当这样做!
import {Foo} from './foo';
export {Bar} from './bar';
```

TypeScript 的工具链会自动区分用作类型的符号和用作值的符号。对于类型引用，工具链不会生成运行时加载的代码。这样做的原因是为了提供更好的开发体验，否则在 `import type` 和 `import` 之间反复切换会非常繁琐。同时，`import type` 并不提供任何保证，因为代码仍然可以通过其它的途径导入同一个依赖。

如果需要在运行时加载代码以执行其副作用，应使用 `import '...'`，参见导入一节。

使用 `export type` 似乎可以避免将某个用作值的符号导出为 API。然而，和 `import type` 类似，`export type` 也不提供任何保证，因为外部代码仍然可以通过其它途径导入。如果需要拆分对 API 作为值的使用和作为类型的使用，并保证二者不被混用的话，应当显式地将其拆分成不同的符号，例如 `UserService` 和 `AjaxUserService`，这样不容易造成错误，同时能更好地表达设计思路。

7.4.4 根据特征组织代码

应当根据特征而非类型组织代码。例如，一个在线商城的代码应当按照 `products`，`checkout`，`backend` 等分类，而不是 `views`，`models`，`controllers`。

7.5 类型系统

7.5.1 类型推导

对于所有类型的表达式（包括变量、字段、返回值，等等），都可以依赖 TypeScript 编译器所实现的类型推导。google3 编译器会拒绝所有缺少类型记号又无法推导出其类型的代码，以保证所有的代码都具有类型（即使其中可能包括显式的 `any` 类型）。

```
const x = 15;  // x 的类型可以推导得出.
```

当变量或参数被初始化为 `string`，`number`，`boolean`，`RegExp` 正则表达式字面量或 `new` 表达式时，由于明显能够推导出类型，因此应当省略类型记号。

```
// 不要这样做！添加 boolean 记号对提高可读性没有任何帮助！
const x: boolean = true;
```

```
// 不要这样做！Set 类型显然可以从初始化语句中推导得出。
const x: Set<string> = new Set();
```

```
// 应当这样做！依赖 TypeScript 的类型推导。
const x = new Set<string>();
```

对于更为复杂的表达式，类型记号有助于提高代码的可读性。此时是否使用类型记号应当由代码审查员决定。

返回类型

代码的作者可以自由决定是否在函数和方法中使用类型记号标明返回类型。代码审查员 可以要求对难以理解的复杂返回类型使用类型记号进行阐明。项目内部 可以自行规定必须标明返回值，本文作为一个通用的 TypeScript 风格指南，不做硬性要求。

显式地标明函数和方法的返回值有两个优点：

- 能够生成更精确的文档，有助于读者理解代码。
- 如果未来改变了函数的返回类型的话，可以让因此导致的潜在的错误更快地暴露出来。

7.5.2 Null 还是 Undefined ？

TypeScript 支持 `null` 和 `undefined` 类型。可空类型可以通过联合类型实现，例如 `string | null` 。对于 `undefined` 也是类似的。对于 `null` 和 `undefined` 的联合类型，并无特殊的语法。

TypeScript 代码中可以使用 `undefined` 或者 `null` 标记缺少的值，这里并无通用的规则约定应当使用其中的某一种。许多 JavaScript API 使用 `undefined` （例如 `Map.get` ），然而 DOM 和 Google API 中则更多地使用 `null` （例如 `Element.getAttribute` ），因此，对于 `null` 和 `undefined` 的选择取决于当前的上下文。

可空/未定义类型别名

不允许为包括 `|null` 或 `|undefined` 的联合类型创建类型别名。这种可空的别名通常意味着空值在应用中会被层层传递，并且它掩盖了导致空值出现的源头。另外，这种别名也让类或接口中的某个值何时有可能为空变得不确定。

因此，代码 必须在使用别名时才允许添加 `|null` 或者 `|undefined` 。同时，代码 应当在空值出现位置的附近对其进行处理。

```
// 不要这样做！不要在创建别名的时候包含 undefined ！
type CoffeeResponse = Latte|Americano|undefined;

class CoffeeService {
```

(continues on next page)

(continued from previous page)

```
    getLatte(): CoffeeResponse { ... };
}
```

```
// 应当这样做! 在使用别名的时候联合 undefined !
type CoffeeResponse = Latte|Americano;

class CoffeeService {
    getLatte(): CoffeeResponse|undefined { ... };
}
```

```
// 这样做更好! 使用断言对可能的空值进行处理!
type CoffeeResponse = Latte|Americano;

class CoffeeService {
    getLatte(): CoffeeResponse {
        return assert(fetchResponse(), 'Coffee maker is broken, file a ticket');
    };
}
```

可选参数还是 undefined 类型？

TypeScript 支持使用 ? 创建可选参数和可选字段，例如：

```
interface CoffeeOrder {
    sugarCubes: number;
    milk?: Whole|LowFat|HalfHalf;
}

function pourCoffee(volume?: Milliliter) { ... }
```

可选参数实际上隐式地向类型中联合了 `|undefined` 。不同之处在于，在构造类实例或调用方法时，可选参数可以被直接省略。例如，`{sugarCubes: 1}` 是一个合法的 `CoffeeOrder` ，因为 `milk` 字段是可选的。

应当使用可选字段（对于类或者接口）和可选参数而非联合 `|undefined` 类型。

对于类，应当尽可能避免使用可选字段，尽可能初始化每一个字段。

```
class MyClass {
    field = '';
}
```

7.5.3 结构类型与指名类型

TypeScript 的类型系统使用的是结构类型而非指名类型。具体地说，一个值，如果它拥有某个类型的所有属性，且所有属性的类型能够递归地一一匹配，则这个值与这个类型也是匹配的。

在代码中，可以在适当的场景使用结构类型。具体地说，在测试代码之外，应当使用接口而非类对结构类型进行定义。在测试代码中，由于经常要创建 Mock 对象用于测试，此时不引入额外的接口往往较为方便。

在提供基于结构类型的实现时，应当在符号的声明位置显式地包含其类型，使类型检查和错误检测能够更准确地工作。

```
// 应当这样做!
const foo: Foo = {
    a: 123,
    b: 'abc',
}
```

```
// 不要这样做!
const badFoo = {
    a: 123,
    b: 'abc',
}
```

为什么要这样做?

这是因为在上文中，badFoo 对象的类型依赖于类型推导。badFoo 对象中可能添加额外的字段，此时类型推导的结果就有可能发生变化。

如果将 badFoo 传给接收 Foo 类型参数的函数，错误提示会出现在函数调用的位置，而非对象声明的位置。在大规模的代码仓库中修改接口时，这一点区别会很重要。

```
interface Animal {
    sound: string;
    name: string;
}

function makeSound(animal: Animal) {}

/**
 * 'cat' 的类型会被推导为 '{sound: string}'
 */
const cat = {
    sound: 'meow',
};

/**
 * 'cat' 的类型并不满足函数参数的要求,
 * 因此 TypeScript 编译器会在这里报错,
```

(continues on next page)

(continued from previous page)

```
 * 而这里有可能离 'cat' 的定义相当远。
 */
makeSound(cat);

/**
 * Horse 具有结构类型，因此这里会提示类型错误，而函数调用点不会报错。
 * 这是因为 'horse' 不满足接口 'Animal' 的类型约定。
 */
const horse: Animal = {
    sound: 'niegh',
};

const dog: Animal = {
    sound: 'bark',
    name: 'MrPickles',
};

makeSound(dog);
makeSound(horse);
```

7.5.4 接口还是类型别名？

TypeScript 支持使用 类型别名 为类型命名。这一功能可以用于基本类型、联合类型、元组以及其它类型。

然而，当需要声明用于对象的类型时，应当使用接口，而非对象字面量表达式的类型别名。

```
// 应当这样做!
interface User {
    firstName: string;
    lastName: string;
}
```

```
// 不要这样做!
type User = {
    firstName: string,
    lastName: string,
}
```

为什么?

这两种形式是几乎等价的，因此，基于从两个形式中只选择其中一种以避免项目中出现变种的原则，这里选择了更常见的接口形式。另外，这里选择接口还有一个 有趣的技术原因 。这篇博文引用了 TypeScript 团队负责人的话：“老实说，我个人的意见是对于任何可以建模的对象都应当使用接口。相比之下，使用类型别名没有任何优势，尤其是类型别名有许多的显示和性能问题”。

7.5.5 Array<T> 类型

对于简单类型（名称中只包含字母、数字和点 . 的类型），应当使用数组的语法糖 T[] ，而非更长的 Array<T> 形式。

对于其它复杂的类型，则应当使用较长的 Array<T> 。

这条规则也适用于 readonly T[] 和 ReadonlyArray<T> 。

```
// 应当这样做!
const a: string[];
const b: readonly string[];
const c: ns.MyObj[];
const d: Array<string|number>;
const e: ReadonlyArray<string|number>;
```

```
// 不要这样做!
const f: Array<string>;              // 语法糖写法更短。
const g: ReadonlyArray<string>;
const h: {n: number, s: string}[]; // 大括号和中括号让这行代码难以阅读。
const i: (string|number)[];
const j: readonly (string|number)[];
```

7.5.6 索引类型 {[key: string]: number}

在 JavaScript 中，使用对象作为关联数组（又称“映射表”、“哈希表”或者“字典”）是一种常见的做法：

```
const fileSizes: {[fileName: string]: number} = {};
fileSizes['readme.txt'] = 541;
```

在 TypeScript 中，应当为键提供一个有意义的标签名。（当然，这个标签只有在文档中有实际意义，在其它场合是无用的。）

```
// 不要这样做!
const users: {[key: string]: number} = ...;
```

```
// 应当这样做!
const users: {[userName: string]: number} = ...;
```

然而，相比使用上面的这种形式，在 TypeScript 中应当考虑使用 ES6 新增的 Map 与 Set 类型。因为 JavaScript 对象有一些 令人困惑又不符合预期的行为 ，而 ES6 的新增类型能够更明确地表达程序员的设计思路。此外，Map 类型的键和 Set 类型的元素都允许使用 string 以外的其他类型。

TypeScript 内建的 Record<Keys, ValueType> 允许使用已定义的一组键创建类型。它与关联数组的不同之处在于键是静态确定的。关于它的使用建议，参见映射类型与条件类型 一节。

7.5.7 映射类型与条件类型

TypeScript 中的 映射类型 与 条件类型 让程序员能够在已有类型的基础上构建出新的类型。在 TypeScript 的标准库中有许多类型运算符都是基于这一机制（例如 `Record` 、`Partial` 、`Readonly` 等等）。

TypeScript 类型系统的这一特性让创建新类型变得简洁，还程序员在设计代码抽象时，既能实现强大的功能，同时海能保证类型安全。然而，它们也有一些缺点：

- 相较于显式地指定属性与类型间关系（例如使用接口和继承，参见下文中的例子），类型运算符需要读者在头脑中自行对后方的类型表达式进行求值。本质上说，这增加了程序的理解难度，尤其是在类型推导和类型表达式有可能横跨数个文件的情况下。
- 映射类型与条件类型的求值模型并没有明确的规范，且经常随着 TypeScript 编译器的版本更新而发生变化，因此并不总是易于理解，尤其是与类型推导一同使用时。因此，代码有可能只是碰巧能够通过编译或者给出正确的结果。在这种情况下，使用类型运算符增加了代码未来的维护成本。
- 映射类型与条件类型最为强大之处在于，它们能够从复杂且/或推导的类型中派生出新的类型。然而从另一方面看，这样做也很容易导致程序难于理解与维护。
- 有些语法工具并不能够很好地支持类型系统的这一特性。例如，一些 IDE 的“查找引用”功能（以及依赖于它的“重命名重构”）无法发现位于 `Pick<T, Keys>` 类型中的属性，因而在查找结果中不会将其设为高亮。

因此，推荐的代码规范如下：

- 任何使用都应当使用最简单的类型构造方式进行表达。
- 一定程度的重复或冗余，往往好过复杂的类型表达式带来的长远维护成本。
- 映射类型和条件类型必须在符合上述理念的情况下使用。

例如，TypeScript 内建的 `Pick<T, Keys>` 类型允许以类型 `T` 的子集创建新的类型。然而，使用接口和继承的方式实现往往更易于理解。

```
interface User {
    shoeSize: number;
    favoriteIcecream: string;
    favoriteChocolate: string;
}

// FoodPreferences 类型拥有 favoriteIcecream 和 favoriteChocolate，但不包括 shoeSize。
type FoodPreferences = Pick<User, 'favoriteIcecream'|'favoriteChocolate'>;
```

这种写法等价于显式地写出 `FoodPreferences` 的属性：

```
interface FoodPreferences {
    favoriteIcecream: string;
    favoriteChocolate: string;
}
```

为了减少重复，可以让 `User` 继承 `FoodPreferences` ，或者在 `User` 中嵌套一个类型为 `Food-Prefences` 的字段（这样做可能更好）：

```
interface FoodPreferences { /* 同上 */ }

interface User extends FoodPreferences {
    shoeSize: number;
    // 这样 User 也包括了 FoodPreferences 的字段。
}
```

使用接口让属性的分类变得清晰，IDE 的支持更完善，方便进一步优化，同时使得代码更易于理解。

7.5.8 any 类型

TypeScript 的 any 类型是所有其它类型的超类，又是所有其它类型的子类，同时还允许解引用一切属性。因此，使用 any 十分危险——它会掩盖严重的程序错误，并且它从根本上破坏了对应的值“具有静态属性”的原则。

尽可能 不要使用 any 。如果出现了需要使用 any 的场景，可以考虑下列的解决方案：

- 提供一个更具体的类型
- 使用 *unknown* 而非 *any*
- 关闭 *Lint* 工具对 *any* 的警告

提供一个更具体的类型

使用接口、内联对象类型、或者类型别名：

```
// 声明接口类型以表示服务端发送的 JSON。
declare interface MyUserJson {
    name: string;
    email: string;
}

// 对重复出现的类型使用类型别名。
type MyType = number|string;

// 或者对复杂的返回类型使用内联对象类型。
function getTwoThings(): {something: number, other: string} {
    // ...
    return {something, other};
}

// 使用泛型，有些库在这种情况下可能会使用 any 表示
// 这里并不考虑函数所作用于的参数类型。
// 注意，对于这种写法，“只有泛型的返回类型”一节有更详细的规范。
function nicestElement<T>(items: T[]): T {
    // 在 items 中查找最棒的元素。
    // 这里还可以进一步为泛型参数 T 添加限制，例如 <T extends HTMLElement>。
}
```

使用 unknown 而非 any

any 类型的值可以赋给其它任何类型，还可以对其解引用任意属性。一般来说，这个行为不是必需的，也不符合期望，此时代码试图表达的内容其实是“该类型是未知的”。在这种情况下，应当使用内建的 unknown 类型。它能够表达相同的语义，并且，因为 unknown 不能解引用任意属性，它较 any 而言更为安全。

```
// 应当这样做!
// 可以将任何值（包括 null 和 undefined）赋给 val，
// 但在缩窄类型或者类型转换之前并不能使用它。
const val: unknown = value;
```

```
// 不要这样做!
const danger: any = value /* 这是任意一个表达式的结果 */;
danger.whoops();  // 完全未经检查的访问!
```

关闭 Lint 工具对 any 的警告

有时使用 any 是合理的，例如用于在测试中构造 Mock 对象。在这种情况下，应当添加注释关闭 Lint 工具对此的警告，并添加文档对使用 any 的合理性进行说明。

```
// 这个测试只需要部分地实现 BookService，否则测试会失败。
// 所以，这里有意地使用了一个不安全的部分实现 Mock 对象。
// tslint:disable-next-line:no-any
const mockBookService = ({get() { return mockBook; }} as any) as BookService;
// 购物车在这个测试里并未使用。
// tslint:disable-next-line:no-any
const component = new MyComponent(mockBookService, /* unused ShoppingCart */ null␣
↪as any);
```

7.5.9 元组类型

应当使用元组类型代替常见的 Pair 类型的写法：

```
// 不要这样做!
interface Pair {
    first: string;
    second: string;
}

function splitInHalf(input: string): Pair {
    // ...
    return {first: x, second: y};
}
```

```
// 应当这样做!
function splitInHalf(input: string): [string, string] {
    // ...
    return [x, y];
}

// 这样使用:
const [leftHalf, rightHalf] = splitInHalf('my string');
```

然而通常情况下，为属性提供一个有意义的名称往往能让代码更加清晰。

如果为此声明一个接口过于繁重的话，可以使用内联对象字面量类型：

```
function splitHostPort(address: string): {host: string, port: number} {
    // ...
}

// 这样使用:
const address = splitHostPort(userAddress);
use(address.port);

// 也可以使用解构进行形如元组的操作:
const {host, port} = splitHostPort(userAddress);
```

7.5.10 包装类型

不要使用如下几种类型，它们是 JavaScript 中基本类型的包装类型：

- `String` 、`Boolean` 和 `Number` 。它们的含义和对应的基本类型 `string` 、`boolean` 和 `number` 略有不同。任何时候，都应当使用后者。
- `Object` 。它和 `{}` 与 `object` 类似，但包含的范围略微更大。应当使用 `{}` 表示“包括除 `null` 和 `undefined` 之外所有类型”的类型，使用 `object` 表示“所有基本类型以外”的类型（这里的“所有基本类型”包括上文中提到的基本类型，`symbol` 和 `bigint` ）。

此外，不要将包装类型用作构造函数。

7.5.11 只有泛型的返回类型

不要创建返回类型只有泛型的 API。如果现有的 API 中存在这种情况，使用时应当显式地标明泛型参数类型。

7.6 一致性

对于本文中并未明确解释的任何与代码风格有关的问题，都应当与同一文件中其它代码的现有写法 **保持一致**。如果问题仍未得到解决，则应当参考同一文件夹下其它文件的写法。

7.6.1 目标

通常情况下，程序员自己是最了解他们的代码需求的人。所以，对于那些答案不唯一、而且最优解取决于实际场景的问题，一般应当由当事人根据情况自行决定解决方案。因此，对于这类问题，默认回答往往都是“不管了”。

以下几点则是其中的特例，它们解释了为什么要在这篇风格指南中编写全局性的规范。对于程序员自行规定的代码风格，应当根据以下几个原则对其进行评估：

1. **应当避免使用已知的会导致问题的代码范式，尤其是对于这门语言的新手而言**

例如：

- `any` 是一个容易被误用的类型（某个变量 *真的*可以既是一个数字，同时还可以作为函数被调用吗?），因此关于它的用法，指南中提出了一些建议。
- TypeScript 的命名空间会为闭包优化带来问题。
- 在文件名中使用句点 `.` 会让导入语句的样式变得不美观且令人困惑。
- 类中的静态函数对优化十分不友好，同样的功能完全可以由文件级函数实现。
- 不熟悉 `private` 关键字的用户会试图使用下划线将函数名变得混乱难懂。

2. **跨项目的代码应当保持一致的用法**

如果有两种语义上等价只是形式上不同的写法，应当只选择其中的一种，以避免代码中发生无意义的发散演化，同时也避免在代码审查的过程中进行无意义的争辩。

除此之外，还应当尽可能与 JavaScript 的代码风格保持一致，因为大部分程序员都会同时使用两种语言。

例如：

- 变量名的首字母大小写风格。
- `x as T` 语法和等价的 `<T>x` 语法（后者不允许使用）。
- `Array<[number, number]>` 和 `[number, number][]` 。

3. **代码应当具有长期可维护性**

代码的生命周期往往比其原始作者为其工作的时间要长，而 TypeScript 团队必须保证谷歌的所有工作在未来依然能顺利进行。

例如：

- 使用自动化工具修改代码，所有的代码均经过自动格式化以符合空格样式的规范。
- 规定了一组 Clousure 编译器标识，使 TS 代码库在编写过程中无需考虑编译选项的问题，也让用户能够安全地使用共享库。
- 代码在使用其它库时必须进行导入（严格依赖），以便依赖项中的重构不会改变其用户的依赖项。

- 用户必须编写测试。如果没有测试，就无法保证对语言或 google3 库中的改动不会破坏用户现有的代码。

4. **代码审查员应当着力于提高代码质量，而非强制推行各种规则**

如果能够将规范实现为自动化检查工具，这通常都是一个好的做法。这对上文中的第三条原则也有所帮助。

对于确实无关紧要的问题，例如语言中十分罕见的边界情况，或者避免了一个不太可能发生的 Bug ，等等，不妨直接无视之。

CHAPTER 8

HTML/CSS 风格指南 - 内容目录

8.1 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。

8.2 整体样式规则

8.2.1 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);
}
```

(continues on next page)

(continued from previous page)

```
/* 推荐 */
.example {
  background: url(//www.google.com/images/example);
}
```

8.3 总体排版规则

8.3.1 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;
}
```

8.3.2 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 -->
<img src="google.png" alt="Google">
```

```
/* 不推荐 */
color: #E5E5E5;

/* 推荐 */
color: #e5e5e5;
```

8.3.3 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.
```

8.4 整体的元数据规则

8.4.1 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）

8.4.2 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。）

8.4.3 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>
```

(continues on next page)

(continued from previous page)

```
 <li>Apples</li>
 <li>Oranges</li>
</ul>
```

8.5 HTML 样式规则

8.5.1 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，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 `<br>` 而不是 `<br />` 。

8.5.2 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。

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>
```

8.5.3 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>
```

8.5.4 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 -->
<img src="spreadsheet.png">

<!-- 推荐 -->
<img src="spreadsheet.png" alt="Spreadsheet screenshot.">
```

8.5.5 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>
```

(continues on next page)

(continued from previous page)

```
<!-- 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!
```

8.5.6 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 `&mdash;`、`&rdquo;` 或 `&#x263a;` 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 "€".
```

8.5.7 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 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 -->
```

(continues on next page)

(continued from previous page)

```
<!DOCTYPE html>
<title>Saving money, saving bytes</title>
<p>Qed.
```

8.5.8 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 和 text/javascript 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。

```
<!-- 不推荐 -->
<link rel="stylesheet" href="//www.google.com/css/maia.css" type="text/css">

<!-- 推荐 -->
<link rel="stylesheet" href="//www.google.com/css/maia.css">

<!-- 不推荐 -->
<script src="//www.google.com/js/gweb/analytics/autotrack.js" type="text/javascript
↪"></script>

<!-- 推荐 -->
<script src="//www.google.com/js/gweb/analytics/autotrack.js"></script>
```

8.6 HTML 格式规则

8.6.1 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
```

(continues on next page)

(continued from previous page)

```
  <li>Curly
</ul>

<table>
  <thead>
    <tr>
      <th scope="col">Income
      <th scope="col">Taxes
  <tbody>
    <tr>
      <td>$ 5.00
      <td>$ 4.50
</table>
```

8.6.2 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>
```

8.7 css 样式规则

8.7.1 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 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。

8.7.2 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 {}
```

8.7.3 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 {}
```

8.7.4 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 的考虑。

```
/* 不推荐 */
ul#example {}
div.error {}

/* 推荐 */
#example {}
.error {}
```

8.7.5 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 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;
```

8.7.6 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;
```

8.7.7 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;
```

8.7.8 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;
```

8.7.9 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 */
```

8.7.10 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 {}
```

8.7.11 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。

8.8 CSS 格式化规则

8.8.1 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;
```

8.8.2 块内容的缩进

缩进块内容。

将包括嵌套及声明的 块内容 进行缩进，以体现层次并提高可读性。

```
@media screen, projection {

  html {
    background: #fff;
    color: #444;
  }

}
```

8.8.3 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;
}
```

8.8.4 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;
}
```

8.8.5 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 声名块 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;
}
```

(continues on next page)

(continued from previous page)

```
/* 不推荐: 不必要的换行符 */
#video
{
  margin-top: 1em;
}

/* 推荐 */
#video {
  margin-top: 1em;
}
```

8.8.6 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;
}
```

8.8.7 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%;
}
```

8.8.8 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，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;
}
```

8.9 CSS 元规则

8.9.1 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 {}
```

8.10 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。

CHAPTER 9

Java 风格指南 - 内容目录

9.1 0. 扉页

原作者

Liam Miller-Cushon

Ted Osborne

详细作者列表请参见英文原版的 Git 仓库.

翻译

John.L

项目主页

- Google Style Guide
- Google 开源项目风格指南 - 中文版

9.2 1. 介绍

本文档 **完整地**定义了 Google 的 Java™ 语言的代码风格。只有当 Java 源文件遵循这里的规则时，才可以说它算遵循了 Google 的编程风格。

与其他编程风格指南一样，该文档涵盖的问题不仅仅是格式的审美问题，还包括其他类型的约定或代码标准。然而，本文档主要关注的是 **我们普遍遵循的硬性规则**，并避免提供无法明确执行的建议（无论是对于人还是工具来说）。

9.2.1 1.1. 术语说明

在本文档中，除非另有说明：

- 1. 术语“类”（class）被广泛地用来指代“普通”的类、枚举类、接口或注释类型（`@interface`）。
- 2. 术语“(类的) 成员”（member）被广泛地用来指代嵌套类、字段、方法或构造函数，即除了初始化块和注释之外类的所有顶级内容。
- 3. 术语“注释”（comment）始终指的是实现注释。我们不使用“文档注释”这种说法，而是使用另一常见的说法“Javadoc”代替。

其他的“术语说明”也将在整个文档中偶尔出现。

9.2.2 1.2. 指南说明

此文档中的中的示例代码 **并不是规范的**。也就是说，尽管示例是按照 Google 风格编写的，但它们可能并不是实现代码的唯一优雅方式。示例中做出的选择性的格式不应被强制作为规则。

9.3 2. 源文件基础

9.3.1 2.1. 文件名

源文件的文件名由其包含的唯一的 顶级类的名称（大小写敏感）加上 `.java` 扩展名构成。

9.3.2 2.2. 文件编码：UTF-8

源文件应使用 **UTF-8** 编码。

9.3.3 2.3. 特殊字符

2.3.1. 空白字符

除了换行符之外，源文件中唯一能够出现的空白字符是 **ASCII 的水平空格字符（0x20）**。这意味着：

- 1. 字符串和字符组成的文本中的所有其他空白字符都应该被转义
- 2. Tab 字符 **不能**用于缩进

2.3.2 特殊转义序列

对于任何 有特殊转义序列 的字符（\b 、\t 、\n 、\f、\r、\"、\'``和 ``\\），应使用以上序列，而不是相应的八进制（例如 \012）或 Unicode（例如 \u000a）转义。

2.3.3. 非 ASCII 字符

对于其余的非 ASCII 字符，可以使用实际的 Unicode 字符（例如 ∞）或等效的 Unicode 转义（例如 \u221e），这只取决于哪种方式使代码 **更容易阅读和理解**，尽管我们强烈不建议在字符串文本和注释之外使用 Unicode 转义。

Tip: 在使用 Unicode 转义的情况下，即使在偶尔使用实际的 Unicode 字符的时候，添加一个解释性的注释可能会非常有帮助。

示例：

例子	点评
`String unitAbbrev = "μs";`	最佳：即使没有注释也完全清晰明了
`String unitAbbrev = "\u03bcs"; // "μs"`	可以，但没有理由这样做
`String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"`	可以，但这样不够优雅且容易出错
`String unitAbbrev = "\u03bcs";`	差：读者完全不知道这是什么
`return '\ufeff' + content; // byte order mark`	好：对不可打印的字符使用转义，并在必要时加上注释

Tip: 永远不要仅仅因为担心某些程序可能无法正确处理非 ASCII 字符而去降低代码的可读性。如果真的发生这种情况，那么说明这些程序是有问题的，你应该修复它们。

9.4 3. 源文件结构

源文件应由以下部分 **按顺序**组成：

- 1. 许可或版权信息（如果有的话）
- 2. 包的声明
- 3. 导入语句（Import statements）
- 4. 有且仅有一个顶级类

每个部分之间用 一**个**空白行隔开。

9.4.1 3.1. 许可或版权信息（如果有的话）

如果文件中应包含许可证或版权信息，那么它应被放在此处。

9.4.2 3.2. 包声明

包声明 **不换行**。列限制（4.4 节，*Column limit: 100*）不适用于包声明。

9.4.3 3.3. 导入语句

3.3.1. 不使用通配符导入

不使用任何形式的 **通配符导入**，无论是静态的还是其他形式。

3.3.2. 不换行

导入语句 **不换行**。列限制（4.4 节，*Column limit: 100* ）不适用于导入语句。

3.3.3. 排序和间距

按以下顺序导入：

- 所有的静态导入都在一个单独的块中。
- 所有非静态导入都在一个单独的块中。

如果既有静态导入又有非静态导入，则两个块之间有一个空白行。导入语句之间没有其他空白行。

在每个块中，导入的名称按 ASCII 排序顺序出现。（**注意**：这不等同于按整个导入语句的 ASCII 排序，因为’.’ 排在’;’ 之前。）（译者注：这种情况只会出现在包名结尾为’.’ 时，所以实际上可以忽略不计。例如：`import packageA.ClassA;` `import packageA.ClassA.;` ，如果按整个语句的 ASCII 排序，后句应排在前句之前，因为’.’ 在 ASCII 中排在’;’ 前。除此之外其他情况按包名排序和按整个语句排序应得到相同的结果）

3.3.4. 不对类使用静态导入

不使用静态导入来导入静态嵌套类，而是使用常规导入。

9.4.4 3.4. 类的声明

3.4.1. 有且仅有一个顶级类声明

每个顶级类都位于其自己的源文件中。

3.4.2. 类中内容的顺序

类中成员和初始化器的顺序对于代码的可读性有很大影响。然而，并没有一个唯一正确的顺序，不同的类可能会以不同的方式排列其内容。

重要的是，每个类都使用 **某种逻辑顺序**，并且在被问及时，其维护者可以做出相应的解释。例如，新方法不仅仅是习惯性地添加到类的末尾，因为那会导致内容是“按添加时间排序”的，这并不是一个有逻辑的排序方式。

3.4.2.1. 重载：永不分割

类中拥有同一名称的方法应出现在一个连续的组中，中间不应该有任何其他成员。对于多个构造函数（它们的名称总是相同的）也是如此。即使在方法之间的修饰符（如 `static` 或 `private` ）不同时，此规则仍然适用。

9.5 4. 格式

术语说明：块状结构指的是类、方法或构造器的主体。注意，根据 4.8.3.1 节关于数组初始化器 的内容，任何数组初始化器都可以选择性地被视为块状结构。

9.5.1 4.1. 花括号

4.1.1. 选择性的花括号的使用

在 `if`、`else` 、`for` 、`do` 和 `while` 语句中，即使主体为空或只包含一个语句，也应该写出花括号。

其他选择性的花括号，比如 lambda 表达式中的花括号，依然不是必须写出的。

4.1.2. 非空块：K & R 风格

非空块和块状结构中的花括号遵循 Kernighan & Ritchie 风格（“埃及括号 ”）：

- 在打开花括号之前不换行，除非下面有详细说明。
- 在打开花括号之后换行。
- 在关闭花括号之前换行。
- 仅在该花括号终止一个语句或终止方法、构造器或命名类的主体时，在关闭花括号之后才需要换行。例如，如果花括号后面跟的是 `else` 或逗号，则不换行。

例外：在某些场景下，尽管规则允许你简单地使用一个以分号（`;` ）结束的语句，但你仍可以在此选择使用一个语句块。在这种情况下，这个语句块开头的花括号前会有一个换行。这种特殊的语句块通常用于限定局部变量的作用范围，例如在 switch 语句中。

例子：

```
return () -> {
    while (condition()) {
        method();
    }
};

return new MyClass() {
    @Override public void method() {
        if (condition()) {
            try {
                something();
            } catch (ProblemException e) {
                recover();
            }
        } else if (otherCondition()) {
            somethingElse();
        } else {
            lastThing();
        }
        {
            int x = foo();
            frob(x);
        }
    }
};
```

关于枚举类的一些例外情况，请参见第 4.8.1 节, 枚举类 。

4.1.3. 空代码块：应简洁表示

一个空的代码块或类似块的结构可以按照 K & R 风格（如第 4.1.2 节非空块：*K & R* 风格 所描述）。或者，你也可以在打开后立即关闭它，中间没有字符也不需要换行（即 `{}`），**除非**它是多块语句的一部分（直接包含多个块的语句，例如 `if` / `else` 或 `try` / `catch` / `finally` ）。

例如：

```
// 这是可以的
void doNothing() {}

// 这同样也是可以的
void doNothingElse() {
}
```

```
// 不可以这样：多块语句中不能出现打开即关闭的简略花括号
try {
    doSomething();
} catch (Exception e) {}
```

9.5.2 4.2. 代码块缩进：增加 2 个空格

每次打开一个新的代码块或块状结构时，缩进增加两个空格。当块结束时，缩进返回到之前的缩进级别。这种缩进级别同时适用于块中的代码以及注释。（参见第 4.1.2 节中的示例，非空块：*K & R* 风格 。）

9.5.3 4.3. 一行一个语句

每个语句后都应换行。

9.5.4 4.4. 列限制：100

Java 代码的列限制为 100 个字符。这里的“字符”指的是任何 Unicode 码位。除非以下特别说明，任何超出此限制的行都必须换行，如第 4.5 节换行 中所解释的。

Tip: 每个 Unicode 码位都算作一个字符，即使其显示宽度大于或小于一个字符。例如，如果使用 全角字符 ，你可能会选择在此规则严格要求的位置之前就换行。

例外：

- 在某些情况下，遵循列限制是不可能的（例如，Javadoc 中的长 URL，或者一个长的 JSNI 方法引用）。
- 包声明和导入语句（参见第 3.2 节包声明 和第 3.3 节导入语句 ）。
- 注释中可能会被复制并粘贴到 shell 的中命令行。
- 在极少数情况下需要使用的非常长的标识符是允许超过列限制的。在这种情况下，周围代码的换行方式应遵循 google-java-format 。

9.5.5 4.5. 换行

术语说明：将可以在一行内书写的代码分成多行的操作被称为换行。

没有全面且确定的公式明确指出在所有情况下如何进换行。很多时候，同一段代码有多种正确的换行方式。

Tip: 注意：尽管进换行的典型原因是为了避免超过列限制，但实际上在列限制内的代码也可以根据作者的判断进换行。

Tip: 提示：有时重构一个方法或定义一个局部变量可以解决超过列限制的问题，而不需要换行。

4.5.1. 换行的位置

换行的首要准则是：倾向于在 **更高级的句法层次**进换行。此外：

- 1. 当一行在非赋值运算符处断开时，换行的位置位于符号之前。（注意，这与 Google 为其他语言所采用的风格不同，如 C++ 和 JavaScript。）
  - 这同样适用于以下”类运算符”的符号：
    - 点分隔符（`.`）
    - 方法引用的两个冒号（`::`）
    - 类型约束中的与符号（`<T extends Foo & Bar>`）
    - catch 块中的管道符号（`catch (FooException | BarException e)`）
- 2. 当一行在赋值运算符处断开时，换行位置通常位于符号之后，但两种方式都是可以接受的。
  - 这也适用于增强的 `for` 循环（”foreach”）语句中的”类赋值运算符”————冒号。
- 3. 方法或构造函数的名称应紧挨着其后的左括号（`(`）。
- 4. 逗号（`,`）应紧挨着它前面的标记。
- 5. lambda 箭头旁绝不换行，但是如果 lambda 的主体仅由单个未括起来的表达式组成，那么可以紧跟在箭头后面换行。示例：

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

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

Tip: 注意：换行的主要目的是使代码更清晰，其不一定是行数最少的。

4.5.2. 行缩进至少 4 个空格

换行时，第一行之后的每一行（每一个连续的行）至少从原行缩进 4 个空格。

当有多个连续行时，根据需要，缩进可以在 4 个空格之外变化。通常，只有当两个连续行从语法上开始于平行的元素时，它们才使用相同的缩进级别。

第 4.6.3 节关于水平对齐 讨论了一种不鼓励使用的做法————用数量变化的空格来使某些标记与前面的行对齐。

9.5.6 4.6. 空白字符

4.6.1. 垂直空白

单个空白行始终应出现在：

- 1. 一个类的连续成员或初始化器之间：字段、构造函数、方法、嵌套类、静态初始化器和实例初始化器。
  - **例外：**两个连续字段之间（它们之间没有其他代码）的空白行是可选的。这样的空行一般根据需要用于创建字段的逻辑分组。
  - **例外：**枚举常量之间的空白行在第 *4.8.1 节* 中有描述。
- 2. 此文档的其他部分所要求的（例如第 3 节，源文件结构，和第 3.3 节导入语句）。

单个空白行也可以出现在任何使用它可以提高代码可读性的位置，例如在语句之间以将代码组织成逻辑子部分。类的第一个成员或初始化器之前，或者最后一个成员或初始化器之后的空白行既不被鼓励也不被反对。

多个连续的空白行是允许的，但从不被要求（或鼓励）。

4.6.2. 水平空白

除了语言或其他风格规则所要求的地方，以及字符串文本、注释和 Javadoc 之外，单个 ASCII 空格字符也**仅**出现在以下位置。

- 1. 将任何保留字，如 `if` 、`for` 或 `catch` ，与其后面的左括号（`(`）隔开
- 2. 将任何保留字，如 `else` 或 `catch` ，与其前面的右花括号（`}`）隔开
- 3. 在任何左花括号（`{`）之前，但有两个例外：
  - `@SomeAnnotation({a, b})`（无空格）
  - `String[][] x = {{"foo"}};`（根据下面的第 9 项，`{{` 之间不需要空格）
- 4. 在任何二元或三元运算符的两侧。这也适用于以下的”类运算符”的符号：
  - 并行类型约束中的与符号：`<T extends Foo & Bar>`
  - 处理多个异常的 catch 块中的管道符号：`catch (FooException | BarException e)`
  - 增强的 `for` （”foreach”）语句中的冒号（`:`）
  - lambda 表达式中的箭头：`(String str) -> str.length()`

  但不包括：
  - 方法引用中的两个冒号（`::`），正确的写法应类似 `Object::toString`
  - 点分隔符（`.`），正确的写法应类似 `object.toString()`
- 5. 在 `,` 、`:` 、`;` 或类型转换的右括号 `)` 之后
- 6. 在代码中任何内容和开始注释的双斜杠 `//` 之间。允许多个空格。

- 7. 在开始注释的双斜杠 `//` 和注释内容之间。允许多个空格。
- 8. 在声明的类型和变量名之间：`List<String> list`
- 9. 在数组初始化器的两个花括号内部（可选）
  - `new int[] {5, 6}` and `new int[] { 5, 6 }` 都是可行的
- 10. 在类型注解和 `[]` 或 `...` 之间

此规则不应被解读为在行的开始或结束时要求或禁止额外的空格；它只涉及内部空格。

4.6.3. 水平对齐：永远不是必要的

术语说明：水平对齐是指在代码中添加变化数量的额外空格，目的是使某些标记直接出现在前面几行的某些其他标记的下方

这种做法是允许的，但 Google 风格永远不要求它。即使在已经使用了水平对齐的地方，也不要求保持水平对齐。

以下是一个不使用对齐的例子，然后是一个使用对齐的例子：

```
private int x;       // 这样很好
private Color color; // 这也是

private int    x;       // 这是允许的，但未来的编辑
private Color color;  // 可能会使它不再对齐
```

Tip: 提示：对齐确实可以帮助提高可读性，但它为未来的维护带来了问题。考虑一个将来需要触碰某一行的更改，该更改可能会使之前令人满意的格式变得混乱（当然这种混乱是允许的）。更常见的是，它会促使编码者（也许是你）也调整附近行的空白，可能会触发一系列新的格式化。如此一来，那一行的更改现在就有了一个“爆炸半径”。在最坏的情况下，这可能导致毫无意义的繁忙工作，但即使在最好的情况下，它仍然会破坏版本历史信息，减慢审查者的速度，并加剧合并冲突（merge conflicts）。

9.5.7 4.7. 分组括号：推荐使用

对于非必须的分组括号，只有当作者和审查者都认为代码在没有它们的情况下不可能被误解，且它们不会使代码更易于阅读时，才能省略它们。假设每个读者都记住了整个 Java 运算符优先级表是不合理的。

9.5.8 4.8. 具体结构

4.8.1. 枚举类

在枚举常量后面的每个逗号后，换行是可选的。也允许额外的空白行（通常只有一行）。以下是一种可能的写法：

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

    NO,
    MAYBE
}
```

一个没有方法且其常量上没有文档注释的枚举类可以选择按照数组初始化器的格式进行编写（参见 4.8.3.1 节有关*数组初始化器* 的内容）。

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

由于枚举类是类，因此编写类的所有其他格式规则都适用于它。

4.8.2. 变量声明

4.8.2.1. 一次只声明一个变量

每个变量声明（字段或局部变量）只声明一个变量：不能使用 `int a, b;` 这样的声明。

例外： `for` 循环的头部中，多个变量的声明是可以接受的。

4.8.2.2. 按需声明

局部变量一般不在其所在的块或块状结构的开始处声明。相反，局部变量在首次使用的地方附近（合理范围内）声明，以最小化它们的作用域。局部变量声明时通常会设定其初始值，或在声明后立即进行初始化。

4.8.3. 数组

4.8.3.1. 数组初始化器：可以是“块状”的

任何数组初始化器都可以选择按照“块状结构”的格式进行编写。例如，以下写法都是可以接受的（这里**并未**列出所有可行的写法）：

```
new int[] {
    0, 1, 2, 3
}

new int[] {
    0, 1,
    2, 3
```

(continues on next page)

(continued from previous page)

```
}

new int[] {
    0,
    1,
    2,
    3
}

new int[]
    {0, 1, 2, 3}
```

4.8.3.2. 不要使用 C 语言风格的声明

方括号是类型的一部分，而非变量的一部分：正确的写法应该为 `String[] args` ，而不是 `String args[]` 。

4.8.4. switch 语句

术语说明：在 switch 语句块的花括号内有一个或多个语句组。每个语句组由一个或多个 switch 标签（要么是 `case FOO:` 要么是 `default:` ）组成，后面跟着一个或多个语句（对于最后一个语句组，可能是零个或多个语句）。

4.8.4.1. 缩进

与任何其他块一样，switch 块中内容的缩进为 2 格。

在 switch 标签后应有一个换行，且缩进级别增加 2 格，就好像一个块正在被打开一样。接下来的 switch 标签回到了之前的缩进级别，就好像一个块已经被关闭了一样。

4.8.4.2. 贯穿：需要注释

在 switch 块内，每个语句组要么突然终止（使用 `break` 、`continue`、`return` 或抛出异常），要么用注释标记，以指示执行会或可能继续进入下一个语句组。任何传达贯穿意思的注释都是足够的（通常是 `// fall through` ）。在 switch 块的最后一个语句组中，不需要这个特殊注释。示例如下

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

(continues on next page)

(continued from previous page)

```
    default:
        handleLargeNumber(input);
}
```

注意单个的 switch 标签 `case 1` 后面贯穿不需要做任何注释，只有在语句组的最后贯穿才需要注释。

4.8.4.3. 必须要有 `default` 标签

每个 switch 语句都包括一个 default 语句组，即使其中不包含任何代码。

例外： 对于 `enum` 类的 switch 语句，如果它明确地包括覆盖该类型的所有可能值的情况，则可以省略 `default` 语句组。如果遗漏了任何可能值，这样做可以让 IDE 或其他静态分析工具能够发出警告。

4.8.5. 注解

4.8.5.1. 类型注解

应用于类型的注解直接出现在被注解的类型之前。如果注解是用 `@Target(ElementType.TYPE_USE)` 进行元注解的，那么它就是一个类型注解。示例如下：

```
final @Nullable String name;

public @Nullable Person getPersonByName(String name)
```

4.8.5.2. 类注解

应用于类的注解紧跟在文档块之后，并且每个注解都列在自己的行上（也就是每行一个注解）。这些换行并不构成换行（见 4.5 节，换行），所以缩进级别不增加。示例如下：

```
@Deprecated
@CheckReturnValue
public final class Frozzler { ... }
```

4.8.5.3. 方法/构造函数注解

应用于方法或构造函数的注解的规则和上一节 一样。实例如下：

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

4.8.5.4. 字段注解

应用于字段的注解也紧跟在文档块之后，但在这种情况下，多个注解（可能带参数）可以列在同一行上；例如：

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

4.8.5.5. 参数/局部变量注解

应用于参数或局部变量的注解没有一个具体的编写规则（当然，当注解是类型注解时除外）。

4.8.6. 注释

本节讨论注释的实现。关于 Javadoc 的内容在第 7 节*Javadoc* 中单独讨论。

尽管一行可能只包含注释和空白字符，但由于有注释的存在，它并不被视为一个完全的空白行。

4.8.6.1. 块状注释

块状注释与周围的代码具有相同的缩进级别。它们可以是 `/* ... */` 或者 `// ...` 形式的。对于多行的 `/* ... */` 注释，后续的行必须以 `*` 开始，并且这个 `*` 要与上一行的 `*` 对齐。

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

注释不应被星号或其他字符所构成的框框起来。

Tip: 提示：当编写多行注释时，如果你希望在必要时，自动的代码格式化工具能重新调整行的格式（段落样式），你应该使用 `/* ... */` 形式。大多数格式化工具不会重新调整 `// ...` 形式的注释块中的行。

4.8.7. 修饰符

当存在类和成员的修饰符时，它们出现的顺序应遵循 Java 语言规范推荐的顺序：

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

4.8.8. 数值字面量

长整型（long 类型）字面量使用大写的 L 作为后缀，而绝不能使用小写的（以避免与数字 1 混淆）。例如，应写为 3000000000L 而非 3000000000l。

9.6 5. 命名

9.6.1 5.1. 所有标识符通用的规则

标识符只使用 ASCII 字母、数字以及在下面提到的极少数情况中才会使用的下划线。因此，每个有效的标识符名称都应与正则表达式 \w+ 匹配。

在 Google 风格中，不使用特殊的前缀或后缀。例如，以下都不是 Google 风格的命名：name_ ，mName ，s_name 和 kName 。

9.6.2 5.2. 不同类型标识符的规则

5.2.1. 包名

包名只使用小写字母和数字（不使用下划线）。连续的单词直接连接在一起。例如 com.example.deepspace ，而不是 com.example.deepSpace 或 com.example.deep_space 。

5.2.2. 类名

类名使用大驼峰命名法（*UpperCamelCase*）。

类名通常是名词或名词短语。例如 Character 、ImmutableList 。接口名称也可能是名词或名词短语（例如 List ），但有时可能是形容词或形容词短语（例如 Readable ）。

对于注解的命名，至今还没有具体的规则，甚至也没有任何不成文的规定。

测试类的名称以 Test 结尾，例如，HashIntegrationTest 。如果它覆盖一个单一的类，其名称是该类的名称后加上 Test，例如 HashImplTest 。

5.2.3. 方法名

方法名使用小驼峰命名法（*lowerCamelCase*）。

方法名通常是动词或动词短语。例如 sendMessage 、stop 。

在 JUnit 测试方法名中，可以使用下划线来分隔名称的逻辑组件，每个组件都使用小驼峰命名法 编写，例如 transferMoney_deductsFromSource 。命名测试方法没有唯一正确的方式。

5.2.4. 常量字段名

常量名使用 UPPER_SNAKE_CASE（大蛇式）：全部为大写字母，每个单词之间用单个下划线分隔。但是，什么才算是一个常量呢？

常量是指那些由 static final 修饰的字段，其内容是深度不可变（译者注：指该对象以及其内部所有可能的引用或对象都是不可变的，与“浅层不可变”相对）的，且其方法不会产生可检测到的副作用。例子包括基本数据类型、字符串、不可变的值类和设置为 null 的任何东西。如果实例的任何可观察状态可以被改变，那它就不是一个常量。仅有不改变对象的意图是不够的。例如：

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final Map<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.
↪of(mutable);
static final ImmutableMap<String, SomeMutableType> mutableValues =
    ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

常量名通常是名词或名词短语。

5.2.5. 非常量字段名

非常量字段名，无论其是否为静态的，都使用小驼峰命名法 格式编写。

非常量字段名通常是名词或名词短语。例如，computedValues 或 index 。

5.2.6. 参数名

参数名使用小驼峰命名法 格式编写。

公共（public）方法中应避免使用单字符参数名。

5.2.7. 局部变量名

局部变量名使用小驼峰命名法格式编写。

即使是 final 且不可变的，局部变量也不被视为常量，因此不应按常量的风格命名。

5.2.8. 类型变量名

每个类型变量的命名方式有两种：

- 单个大写字母，后面可选择性地跟一个数字（如 `E`，`T`，`X`，`T2`）
- 按照类的命名方式命名（参见第 5.2.2 节，类名），然后跟一个大写字母 `T`（例如：`RequestT`，`FooBarT`）。

9.6.3 5.3. 驼峰命名法：明确规定

有时将英文短语转换为驼峰命名有多种合理的方式，例如当存在缩略词或如”IPv6”或”iOS”这样不寻常的结构时。为了提高可预测性，Google 风格指定了以下的（几乎）明确的规定。

从名称的口头表达开始：

- 1. 将短语转换为纯 ASCII 并去除所有的撇号。例如，“Müller’s algorithm”可能变为“Muellers algorithm”。
- 2. 将这个结果分割为单词，以空格和任何剩余的标点符号（通常是连字符）为分隔。
  - 推荐：如果任何单词在常见用法中已经有一个传统的驼峰命名形式，那么将其分割成其组成部分（例如，”AdWords”变为”ad words”）。注意，像”iOS”这样的单词并不真的是驼峰命名，实际上它违反了任何惯例，所以这个建议不再适用。
- 3. 现在将所有内容（包括缩略词）全部转为小写，然后只将：
  - …每个单词的第一个字母大写，得到大驼峰命名，或
  - …除第一个单词外的每个单词的第一个字母大写，得到小驼峰命名
- 4. 最后，将所有的单词连接成一个标识符。

注意，原始单词的大小写几乎完全被忽略。示例：

口头表达	正确形式	错误形式
“XML HTTP request”	`XmlHttpRequest`	`XMLHTTPRequest`
“new customer ID”	`newCustomerId`	`newCustomerID`
“inner stopwatch”	`innerStopwatch`	`innerStopWatch`
“supports IPv6 on iOS?”	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
“YouTube importer”	`YouTubeImporter` 、`YoutubeImporter` *	

* 可以接受，但不推荐。

Tip: 注意：在英文中，有些单词的连字符使用是模糊的：例如，“nonempty”和“non-empty”都是正确的，因此方法名 `checkNonempty` 和 `checkNonEmpty` 同样都是正确的。

9.7 6. 编程习惯

9.7.1 6.1 `@Override`：始终使用

只要合法，方法就应使用 `@Override` 注解进行标记。这包括一个类方法覆盖一个超类方法，一个类方法实现一个接口方法，以及一个接口方法重新指定一个超接口方法的情况。

例外：当父方法被 `@Deprecated` 标记时，可以省略 `@Override` 。

9.7.2 6.2. 捕获的异常：不应忽略

除了以下特别提到的情况，响应捕获的异常时什么也不做是非常少见的做法。(典型的响应是记录它，或者如果认为它是”不可能的”则将其重新抛出为 `AssertionError` 。)

当在 `catch` 块中确实合适地不采取任何行动时，应在注释中解释这样做的原因。

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

例外：在测试中，如果捕获的异常的名称是或以 `expected` 开头的，则可以在不进行注释的情况下忽略它。以下是一个确保测试中的代码抛出预期类型的异常的非常常见的习语，所以这里不需要注释。

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

9.7.3 6.3. 静态成员：使用类名进行限定

当必须对静态类成员进行限定时，应使用该类的名称进行限定，而不是使用该类的类型引用或表达式进行限定。

```
Foo aFoo = ...;
Foo.aStaticMethod(); // 好
aFoo.aStaticMethod(); // 差
somethingThatYieldsAFoo().aStaticMethod(); // 非常差
```

9.7.4 6.4. 析构方法：不使用

重写 `Object.finalize` 方法是 **极其罕见**的。

Tip: 提示：不要这样做。如果你确实必须这样做，首先仔细阅读并理解 《Effective Java》第 8 条：”避免使用析构方法和清理器”，然后还是不要这样做。

9.8 7. Javadoc

9.8.1 7.1. 格式

7.1.1. 一般形式

Javadoc 块的基本格式如下例所示：

```
/**
 * 这里写了多行 Javadoc 文本，
 * 像通常一样包裹起来...
 */
public int method(String p1) { ... }
```

或者如以下的单行示例所示：

```
/** 一段特别短的 Javadoc 文本。 */
```

基本格式始终是可以接受的。当整个 Javadoc 块（包括注释标记）可以放在一行时，可以使用单行格式。注意，这只适用于没有块标签（如 `@return` ）的情况。

7.1.2. 段落

在段落之间，以及在存在的块标签组之前，有一个空白行————也就是说，一个只包含对齐的前导星号（*）的行。除了第一个段落，每个段落在第一个单词之前都有 <p>，其后没有空格。其他块级元素的 HTML 标签，如 <ul> 或 <table>，前面不加 <p>。

7.1.3. 块标签

使用的任何标准“块标签”都按照 @param、@return、@throws、@deprecated 的顺序出现，这四种类型的标签的描述永远不会为空。当一个块标签不能放在一行上时，续行缩进 4 个（或更多）空格从 @ 的位置开始。

9.8.2 7.2. 摘要片段

每个 Javadoc 块都以一个简短的摘要片段开始。这个片段非常重要：它是在某些上下文中出现的唯一部分，如类和方法索引。

这应该是一个片段————一个名词短语或动词短语，而不是一个完整的句子。它不是以 A {@code Foo} is a... 或 This method returns... 开头的，也不形成像 Save the record. 这样的完整祈使句。然而，这个片段就像是一个完整的句子一样需要首字母大写和相应的标点符号。

Tip: 提示：一个常见的错误是以这种形式编写简单的 Javadoc：/** @return the customer ID */。这是不正确的，应该改为 /** Returns the customer ID. */。

9.8.3 7.3. Javadoc 的使用位置

至少每一个 public 类以及这样的类中的每一个 public 或 protected 的成员都有 Javadoc，除了下文中提到的一些例外。

如第 7.3.4 节*非必需的 Javadoc* 所述，还可能存在其他的 Javadoc 内容。

7.3.1. 例外：不言自明的成员

对于简单且一目了然的成员，如 getFoo()，在确实没有其他值得一提的内容的情况下，Javadoc 是可选的，只需说明”Returns the foo”即可。

Tip: 重要：如果某些典型的读者需要知道的相关信息被省略了，那么不适合用这个例外来为省略文档的错误做法辩护。例如，对于一个名为 getCanonicalName 的方法，如果读者很可能不知道”canonical name”是什么意思，那么不要省略其文档（以这里只需要说明 /** Returns the canonical name. */ 为借口，但实际这里的成员名并非一目了然）！

7.3.2. 例外：重写

当一个方法重写超类型的方法时，不必总是在该方法上附带 Javadoc。

7.3.4. 非必需的 Javadoc（译者注：原文中并没有 7.3.3 节）

其他类和成员可根据需要或作者意愿添加 Javadoc。

每当需要使用实现注释来定义类或成员的总体目的或行为时，该注释应以 Javadoc 的形式编写（使用 `/**`）。

非必需的 Javadoc 并不严格要求遵循第 7.1.1、7.1.2、7.1.3 和 7.2 节的格式规则，尽管这样做当然是推荐的。