CHAPTER 8

Style Guides and Rules

第八章 風格指導和規則

Written by Shaindel Schwartz

Edited by Tom Manshreck

Most engineering organizations have rules governing their codebases—rules about where source files are stored, rules about the formatting of the code, rules about naming and patterns and exceptions and threads. Most software engineers are working within the bounds of a set of policies that control how they operate. At Google, to manage our codebase, we maintain a set of style guides that define our rules.

大多數軟體工程機構都有管理其程式碼庫的規則——關於原始檔的儲存位置、程式碼格式化規則、命名規則和模式以及例外和執行緒。大多數軟體工程師就在這組控制他們如何運作的策略的範圍內工作。在谷歌，要管理我們的程式碼庫，我們維護了一套風格指南來定義我們的規則。

Rules are laws. They are not just suggestions or recommendations, but strict, mandatory laws. As such, they are universally enforceable—rules may not be disregarded except as approved on a need-to-use basis. In contrast to rules, guidance provides recommendations and best practices. These bits are good to follow, even highly advisable to follow, but unlike rules, they usually have some room for variance.

規則就像法律。 它們不僅僅是建議或提議，而是嚴格的、強制性的法律。 因此，規則具有普遍可執行性——不得無視規則除非在需要使用的基礎上獲得豁免。 與規則相反，指導提供幫助建議和最佳實踐。 指導值得遵循，甚至是高度建議能夠遵守，但與規則不同的是，指導通常允許出現一些變化的空間。

We collect the rules that we define, the do's and don'ts of writing code that must be followed, in our programming style guides, which are treated as canon. "Style" might be a bit of a misnomer here, implying a collection limited to formatting practices. Our style guides are more than that; they are the full set of conventions that govern our code. That's not to say that our style guides are strictly prescriptive; style guide rules may call for judgement, such as the rule to use names that are "as descriptive as possible, within reason." Rather, our style guides serve as the definitive source for the rules to which our engineers are held accountable.

我們把我們定義的規則，即寫程式碼時必須遵守的 "做 "和 "不做"，收集在我們的程式設計風格指南中，這些指南被視為典範。 "風格"可能這裡有點名不副實，暗示著範圍僅限於格式化實踐。我們的風格指南不止於此； 它們是一套完整的約定約束我們的程式碼。 這並不是說我們的風格指南是嚴格規定的； 是指導還是規則可能需要判斷，例如有一條命名規則是"[在合理範圍內使用與描述性相同的名稱。]" 並且，我們的風格指南是最終的來源我們的工程師必須遵守的規則。

We maintain separate style guides for each of the programming languages used at Google [1].At a high level, all of the guides have similar goals, aiming to steer code development with an eye to sustainability. At the same time, there is a lot of variation among them in scope, length, and content. Programming languages have different strengths, different features, different priorities, and different historical paths to adoption within Google's ever-evolving repositories of code. It is far more practical, therefore, to independently tailor each language's guidelines. Some of our style guides are concise, focusing on a few overarching principles like naming and formatting, as demonstrated in our Dart, R, and Shell guides. Other style guides include far more detail, delving into specific language features and stretching into far lengthier documents—notably, our C++, Python, and Java guides. Some style guides put a premium on typical non-Google use of the language—our Go style guide is very short, adding just a few rules to a summary directive to adhere to the practices outlined in the externally recognized conventions. Others include rules that fundamentally differ from external norms; our C++ rules disallow use of exceptions, a language feature widely used outside of Google code.

我們為每一門在谷歌使用的程式語言都單獨維護一套程式碼風格指南。在高層次上，所有指南都有相似的目標，旨在引導程式碼開發並著眼於可持續性。同時，也有很多變化其中包括範圍、長度和內容。程式語言有不同的優勢，不同的特點，不同的重點，以及在谷歌不斷髮展的程式碼庫中採用的不同歷史路徑。因此，獨立客製每種語言的指南要實際得多。我們的一部分風格指南注重簡潔，專注於一些總體原則，如命名和格式，如在我們的 Dart、R 和 Shell 指南中進行示範的樣子。另一部分風格指南注重更多細節方面，深入研究特定的語言特徵並擴充內容，特別是我們的 C++、Python 和 Java 指南。還有一部分風格指南重視該語言在谷歌之外的慣例——我們的 Go 風格指南非常簡短，只是在一個總結指令中添加了一些規則，以遵循外部公認的慣例。也有部分指南的規則和外部規範根本不同；我們的 C++ 風格指南中不允許使用例外，而使用例外是一種在 Google 程式碼之外廣泛使用語言特性。

The wide variance among even our own style guides makes it difficult to pin down the precise description of what a style guide should cover. The decisions guiding the development of Google's style guides stem from the need to keep our codebase sustainable. Other organizations' codebases will inherently have different requirements for sustainability that necessitate a different set of tailored rules. This chapter discusses the principles and processes that steer the development of our rules and guidance, pulling examples primarily from Google's C++, Python, and Java style guides.

即使是我們自己的風格指南也存在很大的差異，這使得我們很難精確地描述一個風格指南應該涵蓋什麼內容。指導谷歌風格指南開發的決定源於保持我們程式碼庫可持續性的需要。其他組織的程式碼庫天生對可持續性有不同的要求，這就需要一套不同的客製規則。本章討論了指導我們規則和指南開發的原則和過程，主要從谷歌的 c++、Python 和 Java 風格指南中抽取範例。

> 1 我們的許多風格指南都有外部版本，你可以在https://google.github.io/styleguide。我們在本章中參考了這些指南中的許多例子。

Why Have Rules? 為什麼需要規則？

So why do we have rules? The goal of having rules in place is to encourage "good" behavior and discourage "bad" behavior. The interpretation of "good" and "bad" varies by organization, depending on what the organization cares about. Such designations are not universal preferences; good versus bad is subjective, and tailored to needs. For some organizations, "good" might promote usage patterns that support a small memory footprint or prioritize potential runtime optimizations. In other organizations, "good" might promote choices that exercise new language features. Sometimes, an organization cares most deeply about consistency, so that anything inconsistent with existing patterns is "bad." We must first recognize what a given

organization values; we use rules and guidance to encourage and discourage behavior accordingly.

那麼我們為什麼需要規則呢?制定規則的目的是鼓勵“好的”行為，阻止“壞的”行為。對“好”和“壞”的解釋因組織而異，這取決於組織關心的是什麼。這樣的設計不是普遍的偏好;好與壞是主觀的，是根據需要而定的。對於一些組織，“好”可能會促進支援小記憶體佔用或優先考慮潛在執行時最佳化的使用模式。在其他組織中，“好”可能促進使用新語言特性的選擇。有時，組織非常關心一致性，因此與現有模式不一致的任何東西都是“不好的”。我們必須首先認識到一個給定的組織的價值;我們使用規則和指導來鼓勵和阻止相應的行為。

As an organization grows, the established rules and guidelines shape the common vocabulary of coding. A common vocabulary allows engineers to concentrate on what their code needs to say rather than how they're saying it. By shaping this vocabulary, engineers will tend to do the "good" things by default, even subconsciously. Rules thus give us broad leverage to nudge common development patterns in desired directions.

隨著組織的發展，已建立的規則和指導方針形成了通用的編碼詞彙表。通用詞彙表可以讓工程師專注於他們的程式碼需要表達什麼，而不是如何表達。透過塑造這種詞彙，工程師會傾向於預設地、甚至是潛意識地去做“好”事情。因此，規則為我們提供了廣泛的槓桿作用，以便將共同的開發模式推向所需的方向。

Creating the Rules 建立規則

When defining a set of rules, the key question is not, "What rules should we have?" The question to ask is, "What goal are we trying to advance?" When we focus on the goal that the rules will be serving, identifying which rules support this goal makes it easier to distill the set of useful rules. At Google, where the style guide serves as law for coding practices, we do not ask, "What goes into the style guide?" but rather, "Why does something go into the style guide?" What does our organization gain by having a set of rules to regulate writing code?

當定義一組規則時，關鍵問題不是“我們應該有什麼規則?”我們要問的問題是:“我們想要實現的目標是什麼?”當我們關注規則將服務的目標時，識別哪些規則支援這個目標，可以更容易地提取有用的規則集。在谷歌，風格指南作為編碼實踐的法規，我們不會問，“風格指南中包含什麼?”而是“為什麼要把一些東西放進風格指南?”我們的組織透過制定一套規範程式碼編寫的規則獲得了什麼?

Guiding Principles 指導原則

Let's put things in context: Google's engineering organization is composed of more than 30,000 engineers. That engineering population exhibits a wild variance in skill and background. About 60,000 submissions are made each day to a codebase of more than two billion lines of code that will likely exist for decades. We're optimizing for a different set of values than most other organizations need, but to some degree, these concerns are ubiquitous——we need to sustain an engineering environment that is resilient to both scale and time.

讓我們把事情放在背景中:谷歌的工程組織由 3 萬多名工程師組成。工程師群體在技能和背景方面表現出巨大的差異。每天大約有 6 萬份檔案提交給超過 20 億行程式碼的程式碼庫，這些程式碼庫可能會存在幾十年。我們正在最佳化一套不同於大多數其他組織所需要的價值，但在某種程度上，這些關注是無處不在的——我們需要維持一個對規模和時間都有擴充的工程環境。

In this context, the goal of our rules is to manage the complexity of our development environment, keeping the codebase manageable while still allowing engineers to work productively. We are making a trade-off here: the large body of rules that helps us toward this goal does mean we are restricting choice. We lose some flexibility and we might even offend some people, but the gains of consistency and reduced conflict furnished by an authoritative standard win out.

在這種情況下，我們規則的目標是管理開發環境的複雜性，保持程式碼庫的可管理性，同時仍然允許工程師高效地工作。我們在這裡做了權衡:幫助我們實現這一目標的大量規則確實意味著我們在限制選擇。我們失去了一些靈活性，甚至可能會冒犯某些人，但權威標準所帶來的一致性和減少衝突的收益是最重要的。

Given this view, we recognize a number of overarching principles that guide the development of our rules, which must:

- Pull their weight
- Optimize for the reader
- Be consistent
- Avoid error-prone and surprising constructs
- Concede to practicalities when necessary

鑑於這一觀點，我們認識到一些指導我們制定規則的首要原則，這些原則必須:

- 發揮其作用
- 為讀者最佳化
- 保持一致
- 避免容易出錯和令人驚訝的結構
- 必要時對實際問題讓步

Rules must pull their weight 規則必須發揮其作用

Not everything should go into a style guide. There is a nonzero cost in asking all of the engineers in an organization to learn and adapt to any new rule that is set. With too many rules [2] ,not only will it become harder for engineers to remember all relevant rules as they write their code, but it also becomes harder for new engineers to learn their way. More rules also make it more challenging and more expensive to maintain the rule set.

並不是所有的東西都應該放在風格指南中。要求組織中的所有工程師學習和適應任何新規則的成本是有一定代價的。有太多的規則，不僅會讓工程師在寫程式碼時更難記住所有相關的規則，而且也會讓新工程師更難學會他們的方法。更多的規則也會使維護規則集更具挑戰性和更昂貴。

To this end, we deliberately chose not to include rules expected to be self-evident. Google's style guide is not intended to be interpreted in a lawyerly fashion; just because something isn't explicitly outlawed does not imply that it is legal. For example, the C++ style guide has no rule against the use of goto. C++ programmers already tend to avoid it, so including an explicit rule forbidding it would introduce unnecessary overhead. If just one or two engineers are getting something wrong, adding to everyone's mental load by creating new rules doesn't scale.

為此，我們有意排除了大家公認的不言而喻的規則。谷歌的風格指南不打算以法律的方式解釋;沒有明確規定的東西並不意味著它是合法的。例如，c++風格指南沒有規定禁止使用 goto。c++程式設計師已經傾向於避免使用它，所以包含禁止使用它的顯式規則將引入不必要的開銷。如果只有一兩個工程師犯了錯誤，那麼透過建立新規則來增加每個人的負擔是不利於以後擴充的。

> 2 這裡的工具很重要。衡量 "太多 "的標準不是規則的初始數量，而是一個工程師需要記住多少規則。例如，在 clang-format 之前的糟糕時代，我們需要記住大量的格式化規則。這些規則並沒有消失，但在我們目前的工具中，遵守規則的成本已經大大降低了。我們已經達到了這樣的程度：任何人都可以新增任意數量的格式化規則，而沒有人會在意，因為工具只是為你處理好。

Optimize for the reader 為讀者最佳化

Another principle of our rules is to optimize for the reader of the code rather than the author. Given the passage of time, our code will be read far more frequently than it is written. We'd rather the code be tedious to type than difficult to read. In our Python style guide, when discussing conditional expressions, we recognize that they are shorter than if statements and therefore more convenient for code authors. However, because they tend to be more difficult for readers to understand than the more verbose if statements, we restrict their usage. We value "simple to read" over "simple to write." We're making a trade-off here: it can cost more upfront when engineers must repeatedly type potentially longer, descriptive names for variables and types. We choose to pay this cost for the readability it provides for all future readers.

我們規則的另一個原則是為程式碼的讀者而不是作者最佳化。隨著時間的推移，我們的程式碼被閱讀的頻率將遠遠高於編寫的頻率。我們寧願程式碼是繁瑣的輸入，而不是難以閱讀。在我們的 Python 風格指南中，當討論條件表示式時，我們認識到它們比 if 陳述式短，因此對程式碼作者來說更方便。然而，由於它們往往比更冗長的 if 陳述式更難讓讀者理解，所以我們限制了它們的使用。我們認為"讀起來簡單"比"寫起來簡單"更重要。我們在這裡做了一個權衡:當工程師必須重複地為變數和型別輸入可能更長的描述性名稱時，前期的成本會更高。我們選擇支付這筆費用，是因為它為所有未來的讀者提供了可讀性。

As part of this prioritization, we also require that engineers leave explicit evidence of intended behavior in their code. We want readers to clearly understand what the code is doing as they read it. For example, our Java, JavaScript, and C++ style guides mandate use of the override annotation or keyword whenever a method overrides a superclass method. Without the explicit in-place evidence of design, readers can likely figure out this intent, though it would take a bit more digging on the part of each reader working through the code.

作為優先順序的一部分，我們還要求工程師在他們的程式碼中留下預期行為的明確證明。我們希望讀者在閱讀程式碼時能夠清楚地理解程式碼在做什麼。例如，我們的 Java、JavaScript 和 C++風格指導在方法重寫超類方法時手動使用 override 註釋或關鍵字。如果沒有明確的設計證明，讀者很可能會發現這一意圖，但是這需要每個閱讀程式碼的讀者對程式碼進行更多的挖掘。

Evidence of intended behavior becomes even more important when it might be surprising. In C++, it is sometimes difficult to track the ownership of a pointer just by reading a snippet of code. If a pointer is passed to a function, without being familiar with the behavior of the function, we can't be sure what to expect. Does the caller still own the pointer? Did the function take ownership? Can I continue using the pointer after the function returns or might it have been deleted? To avoid this problem, our C++ style guide prefers the use of std::unique_ptr when ownership transfer is intended. unique_ptr is a construct that manages pointer ownership, ensuring that only one copy of the pointer ever exists. When a function takes a unique_ptr as an argument and intends to take ownership of the pointer, callers must explicitly invoke move semantics:

這可能令人驚訝，有意行為的證明變得更加重要。在 C++中，僅僅透過閱讀一段程式碼，有時很難追蹤指標的所有權。如果一個指標被傳遞給一個函式，在不熟悉該函式的行為的情況下，我們不能確定將會發生什麼。呼叫者仍然擁有指標嗎?這個函式擁有所有權了嗎?我可以在函式返回後繼續使用指標嗎?或者它可能已經被刪除了?為了避免這個問題，我們的 C++風格指南更傾向於使用 std::unique_ptr 來實現所有權轉移。Unique_ptr 是一個管理指標所有權的構造，確保指標只有一個副本存在。當函式接受一個 unique_ptr 作為引數，並打算獲得指標的所有權時，呼叫者必須明確地呼叫 move 語義:

```
// Function that takes a Foo* and may or may not assume ownership of
// the passed pointer.
void TakeFoo(Foo* arg);
// Calls to the function don't tell the reader anything about what to
// expect with regard to ownership after the function returns.
Foo* my_foo(NewFoo());
TakeFoo(my_foo);
```

Compare this to the following:
將此與以下內容進行比較:

```
// Function that takes a std::unique_ptr<Foo>.
void TakeFoo(std::unique_ptr<Foo> arg);
// Any call to the function explicitly shows that ownership is
// yielded and the unique_ptr cannot be used after the function
// returns.
std::unique_ptr<Foo> my_foo(FooFactory()); TakeFoo(std::move(my_foo));
```

Given the style guide rule, we guarantee that all call sites will include clear evidence of ownership transfer whenever it applies. With this signal in place, readers of the code don't need to understand the behavior of every function call. We provide enough information in the API to reason about its interactions. This clear documentation of behavior at the call sites ensures that code snippets remain readable and understandable. We aim for local reasoning, where the goal is clear understanding of what's happening at the call site without needing to find and reference other code, including the function's implementation.

鑑於風格指南規則，我們保證所有呼叫方將包括明確的所有權轉移證明，無論何時適用。有了這個訊號，程式碼的讀者就不需要理解每個函式呼叫的行為了。我們在 API 中提供了足夠的資訊來推斷它的互動。這種清晰的呼叫方行為文件確保了程式碼片段的可讀性和可理解性。我們的目標是區域性推理，目標是清楚地瞭解在呼叫方發生了什麼，而不需要查詢和參考其他程式碼，包括函式的具體實現。

Most style guide rules covering comments are also designed to support this goal of in-place evidence for readers. Documentation comments (the block comments prepended to a given file, class, or function) describe the design or intent of the code that follows. Implementation comments (the comments interspersed throughout the code itself) justify or highlight non-obvious choices, explain tricky bits, and underscore important parts of the code. We have style guide rules covering both types of comments, requiring engineers to provide the explanations another engineer might be looking for when reading

through the code.

大多數涉及註釋的風格指南規則也被設計成支援為讀者提供原地證明的目標。文件註釋(預先掛在給定檔案、類或函式上的塊註釋)描述了後面程式碼的設計或意圖。實現註釋(註釋穿插在程式碼本身中)說明或突出不明顯的選擇，解釋棘手的部分，並強調程式碼的重要部分。這兩種型別註釋的指導風格規則在指南中都有涵蓋，要求工程師提供其他工程師在閱讀程式碼時可能正在尋找的解釋。

Be consistent 保持一致性

Our view on consistency within our codebase is similar to the philosophy we apply to our Google offices. With a large, distributed engineering population, teams are frequently split among offices, and Googlers often find themselves traveling to other sites. Although each office maintains its unique personality, embracing local flavor and style, for anything necessary to get work done, things are deliberately kept the same. A visiting Googler's badge will work with all local badge readers; any Google devices will always get WiFi; the video conferencing setup in any conference room will have the same interface. A Googler doesn't need to spend time learning how to get this all set up; they know that it will be the same no matter where they are. It's easy to move between offices and still get work done.

我們對程式碼庫一致性的看法類似於我們應用於谷歌辦公室的理念。由於工程人員眾多，分佈廣泛，團隊經常被分到不同的辦公室，而且谷歌員工經常發現自己在其他地方出差。儘管每個辦公室都保留了自己獨特的個性，融入了當地的風味和風格，但為了完成工作，有一些東西被刻意保持一致。來訪的谷歌員工的徽章可以工作在所有當地的徽章閱讀器上;任何谷歌裝置都可以使用WiFi;任何一間會議室的視訊會議設定都將具有相同的介面。谷歌員工不需要花時間去學習如何設定這些;他們知道，無論他們在哪裡，這個基本條件都是一樣的。在不同的辦公室之間轉換工作很容易，而且還能完成工作。

That's what we strive for with our source code. Consistency is what enables any engineer to jump into an unfamiliar part of the codebase and get to work fairly quickly. A local project can have its unique personality, but its tools are the same, its techniques are the same, its libraries are the same, and it all Just Works.

這就是我們在原始碼中所追求的。一致性是使任何工程師即使進入程式碼庫中不熟悉的部分，也能夠相當迅速地開始工作的原因。一個本地專案可以有它獨特的個性，但是它的工具是一樣的，它的技術是一樣的，它的函式庫是一樣的，而且都是正常工作的。

Advantages of consistency 一致性的優點

Even though it might feel restrictive for an office to be disallowed from customizing a badge reader or video conferencing interface, the consistency benefits far outweigh the creative freedom we lose. It's the same with code: being consistent may feel constraining at times, but it means more engineers get more work done with less effort: [3]

- When a codebase is internally consistent in its style and norms, engineers writing code and others reading it can focus on what's getting done rather than how it is presented. To a large degree, this consistency allows for expert chunking. [4] When we solve our problems with the same interfaces and format the code in a consistent way, it's easier for experts to glance at some code, zero in on what's important, and understand what it's doing. It also makes it easier to modularize code and spot duplication. For these reasons, we focus a lot of attention on consistent naming conventions, consistent use of common patterns, and consistent formatting and structure. There are also many rules that put forth a decision on a seemingly small issue solely to guarantee that things are done in only one way.For example, take the choice of the

number of spaces to use for indentation or the limit set on line length .[5] It's the consistency of having one answer rather than the answer itself that is the valuable part here.

- Consistency enables scaling. Tooling is key for an organization to scale, and consistent code makes it easier to build tools that can understand, edit, and generate code. The full benefits of the tools that depend on uniformity can't be applied if everyone has little pockets of code that differ—if a tool can keep source files updated by adding missing imports or removing unused includes, if different projects are choosing different sorting strategies for their import lists, the tool might not be able to work everywhere. When everyone is using the same components and when everyone's code follows the same rules for structure and organization, we can invest in tooling that works everywhere, building in automation for many of our maintenance tasks. If each team needed to separately invest in a bespoke version of the same tool, tailored for their unique environment, we would lose that advantage.
- Consistency helps when scaling the human part of an organization, too. As an organization grows, the number of engineers working on the codebase increases. Keeping the code that everyone is working on as consistent as possible enables better mobility across projects, minimizing the ramp-up time for an engineer switching teams and building in the ability for the organization to flex and adapt as headcount needs fluctuate. A growing organization also means that people in other roles interact with the code—SREs, library engineers, and code janitors, for example. At Google, these roles often span multiple projects, which means engineers unfamiliar with a given team's project might jump in to work on that project's code. A consistent experience across the codebase makes this efficient.
- Consistency also ensures resilience to time. As time passes, engineers leave projects, new people join, ownership shifts, and projects merge or split. Striving for a consistent codebase ensures that these transitions are low cost and allows us nearly unconstrained fluidity for both the code and the engineers working on it, simplifying the processes necessary for long-term maintenance.

儘管不允許辦公室客製徽章閱讀器或視訊會議介面可能會讓人覺得受到限制，但一致性帶來的好處遠遠大於我們失去的創作自由。程式碼也是如此:一致性有時可能會讓人感到約束，但這意味著更多的工程師用更少的努力完成更多的工作:

- 編寫程式碼的工程師和閱讀程式碼的其他人就可以專注於正在完成的工作，而不是它的呈現方式。在很大程度上，這種一致性允許專家分塊閱讀。當我們用相同的介面解決問題，並以一致的方式格式化程式碼時，專家們更容易瀏覽一些程式碼，鎖定重要的內容，並理解它在做什麼。它還使模組化程式碼和定位重複變得更容易。基於這些原因，我們將重點放在一致的命名約定、一致的通用模式使用以及一致的格式和結構上。也有許多規則對看似很小的問題做出決定，只是為了保證事情只能以一種方式完成。例如，選擇用於縮排的空格數或行長限制只有一個答案而不是答案本身的才是這裡有價值的部分。
- 一致性可以使規模擴大。工具是組織擴充的關鍵，而一致的程式碼使建構能夠理解、編輯和產生程式碼的工具變得更容易。如果每個人都有少量不同的程式碼，那麼依賴於一致性的工具的全部好處就無法應用——如果一個工具可以透過新增缺失的匯入或刪除未使用的包含來更新原始檔，如果不同的專案為他們的匯入列表選擇不同的排序策略，這個工具可能不能在任何地方都適用。當每個人都使用相同的元件，當每個人的程式碼都遵循相同的結構和組織規則時，我們就可以投資於在任何地方都能工作的工具，為我們的許多維護任務建構自動化。如果每個團隊需要分別投資同一工具的客製版本，為他們獨特的環境量身客製，我們就會失去這種優勢。
- 在擴充組織的人力部分時，一致性也有幫助。隨著組織的增長，從事程式碼庫工作的工程師數量也會增加。讓每個人都在編寫的程式碼儘可能一致，這樣可以更好地跨專案移動，最大限度地減少工程師轉換團隊的過渡時間，併為組織建構適應員工需求波動的能力。一個成長中的組織還意味著其他角色的人與程式碼 SREs、庫工程師和程式碼管理員進行互動。在谷歌，這些角色通常跨越多個專案，這意味著不熟悉某個團隊專案的工程師可能會參與到專案程式碼的編寫中。跨程式碼庫的一致體驗使得這種方法非常有效。

- 一致性也確保了對時間的適應性。隨著時間的推移，工程師離開專案，新人加入，所有權轉移，專案合併或分裂。努力實現一致的程式碼庫可以確保這些轉換的成本較低，並允許我們對程式碼和工作在程式碼上的工程師幾乎不受約束的流動，從而簡化長期維護所需的過程。

> 3 歸功於 H.Wright 的現實世界的比較，這是在訪問了大約 15 個不同的谷歌辦公室後做出的。
>
> 4 "分塊 "是一種認知過程，它將資訊碎片組合成有意義的 "塊"，而不是單獨記下它們。例如，國際象棋高手考慮的是棋子的配置，而不是個人的位置。
>
> 5 查閱 4.2 Block indentation: +2 spaces, Spaces vs. Tabs, 4.4 Column limit:100 and Line Length

---

At Scale 規模效應

A few years ago, our C++ style guide promised to almost never change style guide rules that would make old code inconsistent: “In some cases, there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency.”

幾年前，我們的 C++風格指南承諾，幾乎不會改變會使舊程式碼不一致的風格指南規則:“在某些情況下，改變某些風格規則可能有很好的理由，但我們仍然保持事物的原樣，以保持一致性。”

When the codebase was smaller and there were fewer old, dusty corners, that made sense.

當代碼庫比較小的時候，老舊的程式碼比較少的時候，這是有意義的。

When the codebase grew bigger and older, that stopped being a thing to prioritize. This was (for the arbiters behind our C++ style guide, at least) a conscious change: when striking this bit, we were explicitly stating that the C++ codebase would never again be completely consistent, nor were we even aiming for that.

當代碼庫變得更大、更老舊時，這就不再是需要優先考慮的事情了。這是(至少對於我們 C++風格指南背後的裁定者來說)一個有意識的改變:當改變這一點時，我們明確地宣告 C++程式碼庫將不再是完全一致的，我們甚至也不打算這樣做。

It would simply be too much of a burden to not only update the rules to current best practices, but to also require that we apply those rules to everything that’s ever been written. Our Large Scale Change tooling and processes allow us to update almost all of our code to follow nearly every new pattern or syntax so that most old code exhibits the most recent approved style (see Chapter 22). Such mechanisms aren’t perfect, however; when the codebase gets as large as it is, we can’t be sure every bit of old code can conform to the new best practices. Requiring perfect consistency has reached the point where there’s too much cost for the value.

不僅要將規則更新到當前的最佳實踐，而且還要將這些規則應用到已經編寫的所有內容，這樣的負擔太大了。我們的大規模變更工具和過程允許我們更新幾乎所有的程式碼，以遵循幾乎每一個新的模式或語法，所以大多數舊的程式碼都呈現出最新的被認可的風格(見第 22 章)。然而，這種機制並不完美;當代碼庫變得足夠大時，我們不能保證每一段舊程式碼都能符合新的最佳實踐。對完美一致性的要求需要付出的價代價太大了。

Setting the standard. When we advocate for consistency, we tend to focus on internal consistency. Sometimes, local conventions spring up before global ones are adopted, and it isn't reasonable to adjust everything to match. In that case, we advocate a hierarchy of consistency: "Be consistent" starts locally, where the norms within a given file precede those of a given team, which precede those of the larger project, which precede those of the overall codebase. In fact, the style guides contain a number of rules that explicitly defer to local conventions [6], valuing this local consistency over a scientific technical choice.

設定標準。 當我們提倡一致性時，我們傾向於關注內部一致性。有時，區域性的慣例規則在整體慣例規則產生之前就已經出現了，因此調整一切來適應整體慣例規則是不合理的。在這種情況下，我們提倡一種層級的一致性:"保持一致性"從區域性開始，一個檔案中的規範優先於一個團隊的規範，優先於更大的專案的規範，也優先於整個程式碼庫的規範。事實上，風格指南包含了許多明確遵守區域性慣例的規則，重視區域性的一致性，而不是科學技術的選擇。

However, it is not always enough for an organization to create and stick to a set of internal conventions. Sometimes, the standards adopted by the external community should be taken into account.

然而，對於一個組織來說，僅僅建立並遵守一套內部慣例是不夠的。有時，應考慮到外部環境的慣例。

> 6 使用 const，例子。

Counting Spaces 空格的計算

The Python style guide at Google initially mandated two-space indents for all of our Python code. The standard Python style guide, used by the external Python community, uses four-space indents. Most of our early Python development was in direct support of our C++ projects, not for actual Python applications. We therefore chose to use two-space indentation to be consistent with our C++ code, which was already formatted in that manner. As time went by, we saw that this rationale didn't really hold up. Engineers who write Python code read and write other Python code much more often than they read and write C++ code. We were costing our engineers extra effort every time they needed to look something up or reference external code snippets. We were also going through a lot of pain each time we tried to export pieces of our code into open source, spending time reconciling the differences between our internal code and the external world we wanted to join.

谷歌的 Python 風格指南最初要求我們所有的 Python 程式碼都採用雙空格縮排。外部 Python 社群使用的標準 Python 風格指南使用四空格縮排。我們早期的大部分 Python 開發都是直接支援我們的 C++專案，而不是實際的 Python 應用程式。因此，我們選擇使用雙空格縮排，以與我們的 C++程式碼保持一致，C++程式碼已經以這種方式格式化了。隨著時間的推移，我們發現這種理論並不成立。編寫 Python 程式碼的工程師讀和寫其他 Python 程式碼的頻率要比讀和寫 c++程式碼的頻率高得多。每次我們的工程師需要查詢或參考外部程式碼片段時，我們都要花費額外的精力。每次我們試圖將程式碼片段輸出到開源時，我們都經歷了很多痛苦，花了很多時間來調和內部程式碼和我們想要加入的外部世界之間的差異。

When the time came for Starlark (a Python-based language designed at Google to serve as the build description language) to have its own style guide, we chose to change to using four-space indents to be consistent with the outside world. [7]

當 Starlark(一種基於 python 的語言，設計於谷歌，作為建構描述語言)有了自己的風格指南時，我們選擇使用四間距縮排來與外界保持一致。

If conventions already exist, it is usually a good idea for an organization to be consistent with the outside world. For small, self-contained, and short-lived efforts, it likely won't make a difference; internal consistency matters more than anything happening outside the project's limited scope. Once the passage of time and potential scaling become factors, the likelihood of your code interacting with outside projects or even ending up in the outside world increase. Looking long-term, adhering to the widely accepted standard will likely pay off.

如果慣例已經存在，那麼一個組織與外界保持一致通常是一個好主意。對於小的，獨立的，生命週期短的專案，它可能不會有什麼不同;內部一致性比發生在專案有限範圍之外的任何事情都重要。一旦時間的推移和潛在的擴充性成為要素，程式碼與外部專案互動甚至最終與外部世界互動的可能性就會增加。從長遠來看，堅持被廣泛接受的標準可能會有回報。

> 7 用 Starlark 實現的 BUILD 檔案的樣式格式由 buildifier 應用。參見https://github.com/ bazelbuild/buildtools。

Avoid error-prone and surprising constructs 避免容易出錯和令人驚訝的結構

Our style guides restrict the use of some of the more surprising, unusual, or tricky constructs in the languages that we use. Complex features often have subtle pitfalls not obvious at first glance. Using these features without thoroughly understanding their complexities makes it easy to misuse them and introduce bugs. Even if a construct is well understood by a project's engineers, future project members and maintainers are not guaranteed to have the same understanding.

我們的風格指南限制了我們使用的語言中一些更令人驚訝、不尋常或棘手的結構的使用。複雜的特徵往往有一些乍一看並不明顯的細微缺陷。在沒有徹底瞭解其複雜性的情況下使用這些特性，很容易誤用它們並引入錯誤。即使現在的專案的工程師可以很好地理解這個結構，也不能保證未來的專案成員和維護者有同樣的理解。

This reasoning is behind our Python style guide ruling to avoid using power features such as reflection. The reflective Python functions hasattr() and getattr() allow a user to access attributes of objects using strings:

這就是我們 Python 風格指南中避免使用反射等功能特性的原因。Python 反射函式 hasattr()和 getattr()允許使用者使用字串存取物件的屬性:

```
if hasattr(my_object, 'foo'):
some_var = getattr(my_object, 'foo')
```

Now, with that example, everything might seem fine. But consider this: some_file.py:

現在，在這個例子中，一切看起來都很好。但是考慮一下這個:

some_file.py:

```
A_CONSTANT = [
    'foo',
    'bar',
    'baz',
]
```

other_file.py:

```
values = []
for field in some_file.A_CONSTANT:
values.append(getattr(my_object, field))
```

When searching through code, how do you know that the fields foo, bar, and baz are being accessed here? There's no clear evidence left for the reader. You don't easily see and therefore can't easily validate which strings are used to access attributes of your object. What if, instead of reading those values from A_CONSTANT, we read them from a Remote Procedure Call (RPC) request message or from a data store? Such obfuscated code could cause a major security flaw, one that would be very difficult to notice, simply by validating the message incorrectly. It's also difficult to test and verify such code.

當搜尋程式碼時，怎麼知道這裡存取的是欄位 foo、bar 和 baz ?沒有給讀者留下明確的證明。你不容易看到，因此也不容易驗證哪些字串用於存取物件的屬性。如果不是從 A_CONSTANT 讀取這些值，而是從遠端過程呼叫(Remote Procedure Call, RPC)請求訊息或資料儲存讀取這些值，那會怎麼樣呢?這種模糊化的程式碼可能會導致一個嚴重的安全漏洞，一個很難被發現的漏洞，只需錯誤地驗證訊息就可能發生。測試和驗證這樣的程式碼也很困難。

Python's dynamic nature allows such behavior, and in very limited circumstances, using hasattr() and getattr() is valid. In most cases, however, they just cause obfuscation and introduce bugs.

Python 的動態特性允許這樣的行為，並且在非常有限的情況下，使用 hasattr()和 getattr()是有效的。然而，在大多數情況下，它們只會造成混淆並引入錯誤。

Although these advanced language features might perfectly solve a problem for an expert who knows how to leverage them, power features are often more difficult to understand and are not very widely used. We need all of our engineers able to operate in the codebase, not just the experts. It's not just support for the novice software engineer, but it's also a better environment for SREs—if an SRE is debugging a production outage, they will jump into any bit of suspect code, even code written in a language in which they are not fluent. We place higher value on simplified, straightforward code that is easier to understand and maintain.

儘管這些高階語言特性可能完美地解決了知道如何利用它們的專家的問題，但強大的特性通常更難理解，而且沒有得到廣泛的應用。我們需要我們所有的工程師都能夠在程式碼庫中操作，而不僅僅是專家。它不僅支援新手軟體工程師，而且對 SRE 來說也要建立一個更好的環境——如果 SRE 在除錯生產中斷，他們會跳入任何可疑的程式碼，甚至包括一些用他們不熟練的語言編寫的程式碼。我們更加重視易於理解和維護的簡化、直接的程式碼。

Concede to practicalities 為實用性讓步

In the words of Ralph Waldo Emerson: “A foolish consistency is the hobgoblin of little minds.” In our quest for a consistent, simplified codebase, we do not want to blindly ignore all else. We know that some of the rules in our style guides will encounter cases that warrant exceptions, and that’s OK. When necessary, we permit concessions to optimizations and practicalities that might otherwise conflict with our rules.

用拉爾夫·沃爾多·愛默生的話說:“愚蠢的一致性是心胸狹隘的妖怪（為渺小的政治家、哲學家和神學家所崇拜。一個偉大的靈魂與一致性毫無關係）”。在我們追求一致的、簡化的程式碼庫時，我們不想盲目地忽略所有其他內容。我們知道風格指南中的一些規則會遇到需要例外的情況，這都是可以的。必要時，我們允許對可能與我們的規則相沖突的最佳化和和實際問題做出讓步。

Performance matters. Sometimes, even if it means sacrificing consistency or readability, it just makes sense to accommodate performance optimizations. For example, although our C++ style guide prohibits use of exceptions, it includes a rule that allows the use of noexcept, an exception-related language specifier that can trigger compiler optimizations.

效能很重要。有時，即使這意味著犧牲一致性或可讀性，適應性能最佳化也是有意義的。例如，儘管我們的 C++風格指南禁止使用例外，但它包含了一條允許使用 noexcept 的規則，noexcept 是一個與例外相關的語言說明符，可以觸發編譯器最佳化。

Interoperability also matters. Code that is designed to work with specific non-Google pieces might do better if tailored for its target. For example, our C++ style guide includes an exception to the general CamelCase naming guideline that permits use of the standard library’s snake_case style for entities that mimic standard library features.[8] The C++ style guide also allows exemptions for Windows programming, where compatibility with platform features requires multiple inheritance, something explicitly forbidden for all other C++ code. Both our Java and JavaScript style guides explicitly state that generated code, which frequently interfaces with or depends on components outside of a project’s ownership, is out of scope for the guide’s rules.[^9] Consistency is vital; adaptation is key.

互操作性也很重要。為特定的非 google 部分而設計的程式碼，如果為其目標量身定做，可能會做得更好。例如，我們的 C++風格指南有一個通用 CamelCase 命名準則的例外，它允許對模仿標準庫功能的實體使用標準庫的 snake_case 風格。C++風格指南還允許對 Windows 程式設計的豁免，在 Windows 程式設計中，與平台特性的相容性需要使用多重繼承，這對其他情況的 C++程式碼來說都是明確禁止的。我們的 Java 和 JavaScript 風格指南都明確指出，產生的程式碼中，經常與專案之外的元件互動或依賴於這些元件的程式碼不在本指南的範圍內。一致性是非常重要的；適應更是關鍵所在。

> 8 見命名規則的例外情況。作為一個例子，我們的開源 Abseil 庫對打算替代標準型別的型別使用了 snake_case 命名。參見https://github.com/abseil/abseilcpp/blob/master/absl/utility/utility.h 中定義的型別。這些是 C++11 對 C++14 標準型別的實現，因此使用了標準所青睞的 snake_case 風格，而不是谷歌所青睞的 CamelCase 形式。
>
> [^9}; See Generated code: mostly exempt.
>
> 9 查閱 產生的程式碼：主要是豁免./

The Style Guide 風格指南

So, what does go into a language style guide? There are roughly three categories into which all style guide rules fall:

- Rules to avoid dangers
- Rules to enforce best practices
- Rules to ensure consistency

那麼，語言風格指南應該包含哪些內容呢?所有的風格指南規則大致分為三類:

- 規避危險的規則
- 執行最佳實踐的規則
- 確保一致性的規則

Avoiding danger 規避危險

First and foremost, our style guides include rules about language features that either must or must not be done for technical reasons. We have rules about how to use static members and variables; rules about using lambda expressions; rules about handling exceptions; rules about building for threading, access control, and class inheritance. We cover which language features to use and which constructs to avoid. We call out standard vocabulary types that may be used and for what purposes. We specifically include rulings on the hard-to-use and the hard-to-use-correctly—some language features have nuanced usage patterns that might not be intuitive or easy to apply properly, causing subtle bugs to creep in. For each ruling in the guide, we aim to include the pros and cons that were weighed with an explanation of the decision that was reached. Most of these decisions are based on the need for resilience to time, supporting and encouraging maintainable language usage.

首先，我們的風格指南包括關於語言特性的規則，這些規則出於技術原因必須或不必須做。我們有關於如何使用靜態成員和變數的規則；關於使用 lambda 表示式的規則;例外處理規則；關於建構執行緒、存取控制和類繼承的規則。我們涵蓋了要使用的語言特性和要避免的結構。我們列出了可以使用的標準詞彙型別以及用途。我們特別包括了關於難以使用和難以正確使用的規則——一些語言特性具有微妙的使用模式，可能不直觀或不容易正確應用，會導致一些 Bug。對於指南中的每個規則，我們的目標是在描述所達成的決定時，解釋權衡過的利弊。這些決定（規則）大多是基於對時適應性的需要，或者支援和鼓勵可維護的語言使用。

Enforcing best practices 執行最佳實踐

Our style guides also include rules enforcing some best practices of writing source code. These rules help keep the codebase healthy and maintainable. For example, we specify where and how code authors must include comments.[9] Our rules for comments cover general conventions for commenting and extend to include specific cases that must include in-code documentation—cases in which intent is not always obvious, such as fall-through in switch statements, empty exception catch blocks, and template metaprogramming. We also have rules detailing the structuring of source files, outlining the organization of expected content. We have rules about naming: naming of packages, of classes, of functions, of variables. All of these rules are intended to guide engineers to practices that support healthier, more sustainable code.

我們的風格指南還包括一些編寫原始碼的最佳實踐的規則。這些規則有助於保持程式碼庫的健康和可維護性。例如，我們指定程式碼作者必須在哪裡以及如何包含註釋。我們的註釋規則涵蓋了註釋的一般慣例，並擴充到包括必須包含程式碼內文件的特定情況——在這些情況下，意圖並不總是明顯的，例如 switch 陳述式中的失敗，空的例外捕獲塊，以及範本超程式設計。我們還有詳細說明原始檔結構的規則，概述了預期內容的組織。我們有關於命名的規則:包、類、函式、變數的命名。所有這些規則都是為了指導工程師採用更健康、更可持續的程式碼的實踐。

Some of the best practices enforced by our style guides are designed to make source code more readable. Many formatting rules fall under this category. Our style guides specify when and how to use vertical and horizontal whitespace in order to improve readability. They also cover line length limits and brace alignment. For some languages, we cover formatting requirements by deferring to autoformatting tools— gofmt for Go, dartfmt for Dart. Itemizing a detailed list of formatting requirements or naming a tool that must be applied, the goal is the same: we have a consistent set of formatting rules designed to improve readability that we apply to all of our code.

我們的風格指南中實施的一些最佳實踐旨在使原始碼更具可讀性。許多格式規則都屬於這一類別。我們的樣式指南指定了何時以及如何使用換行和水平空格，以提高可讀性。它們還包括行長度限制和大括號對齊。對於某些語言，我們透過使用自動格式化工具來滿足格式化要求——Go 的 gofmt 和 Dart 的 dartfmt。逐項列出格式化需求的詳細列表，或者命名必須應用的工具，目標是相同的:我們有一套一致的格式化規則，旨在提高我們所有程式碼的可讀性。

Our style guides also include limitations on new and not-yet-well-understood language features. The goal is to preemptively install safety fences around a feature's potential pitfalls while we all go through the learning process. At the same time, before everyone takes off running, limiting use gives us a chance to watch the usage patterns that develop and extract best practices from the examples we observe. For these new features, at the outset, we are sometimes not sure of the proper guidance to give. As adoption spreads, engineers wanting to use the new features in different ways discuss their examples with the style guide owners, asking for allowances to permit additional use cases beyond those covered by the initial restrictions. Watching the waiver requests that come in, we get a sense of how the feature is getting used and eventually collect enough examples to generalize good practice from bad. After we have that information, we can circle back to the restrictive ruling and amend it to allow wider use.

我們的風格指南還包括對新的和尚未被很好理解的語言特性的限制。目的是在學習過程中，在一個功能的潛在缺陷周圍預先安裝安全圍欄。同時，在每個人都應用起來之前，限制使用讓我們有機會觀察，從我們觀察的例子中開發和提取最佳實踐的使用模式。對於這些新特性，在開始的時候，我們有時並不確定該如何給予適當的指導。隨著採用範圍的擴大，希望以不同方式使用新特性的工程師會與風格指南的所有者討論他們的例子，要求允許超出最初限制範圍的額外使用案例。透過觀察收到的豁免請求，我們瞭解了該特性是如何被使用的，並最終收集了足夠多的範例來總結好的實踐。在我們得到這些資訊之後，我們可以回到限制性規則，並修改它以允許更廣泛的使用。

> 10 查閱 `https://google.github.io/styleguide/cppguide.html#Comments`, `http://google.github.io/styleguide/pyguide#38-comments-and-docstrings`, 以及 `https://google.github.io/styleguide/javaguide.html#s7-javadoc`, where multiple languages define general comment rules.其中多種語言定義了一般的評論規則。

Case Study: Introducing std::unique_ptr 案例分析：介紹 std::unique_ptr

When C++11 introduced std::unique_ptr, a smart pointer type that expresses exclusive ownership of a dynamically allocated object and deletes the object when the unique_ptr goes out of scope, our style guide initially disallowed usage. The behavior of the unique_ptr was unfamiliar to most engineers, and the related move semantics that the language introduced were very new and, to most engineers, very confusing. Preventing the introduction of std::unique_ptr in the codebase seemed the safer choice. We updated our tooling to catch references to the disallowed type and kept our existing guidance recommending other types of existing smart pointers.

當 C++11 引入 std::unique_ptr 是一種智慧指標型別，它表達了對動態分配物件的獨佔所有權，並在 unique_ptr 超出範圍時刪除該物件，我們的風格指南最初不允許使用。unique_ptr 的使用方式對於大多數工程師來說是不熟悉的，並且該語言引入的相關的 move 語義是非常新的，對大多數工程師來說，非常令人困惑。防止在程式碼庫中引入 std::unique_ptr 似乎是更安全的選擇。我們更新了工具來捕獲對不允許型別的參考，並保留了現有的指南規則，建議使用其他型別的智慧指標。

Time passed. Engineers had a chance to adjust to the implications of move semantics and we became increasingly convinced that using std::unique_ptr was directly in line with the goals of our style guidance. The information regarding object ownership that a std::unique_ptr facilitates at a function call site makes it far easier for a reader to understand that code. The added complexity of introducing this new type, and the novel move semantics that come with it, was still a strong concern, but the significant improvement in the long-term overall state of the codebase made the adoption of std::unique_ptr a worthwhile trade-off.

時間飛逝。工程師有機會適應 move 語義的語義，我們也越來越相信使用 std::unique_ptr 直接符合我們的風格指南的目標。在函式呼叫處上，std::unique_ptr 所提供的關於物件所有權的資訊使讀者更容易理解該程式碼。引入這種新型別所增加的複雜性，以及隨之而來的新的 move 語義，仍然是一個值得關注的問題，但是程式碼庫長期整體狀態的顯著改善使得采用 std::unique_ptr 是一個值得的權衡。

Building in consistency 建構一致性

Our style guides also contain rules that cover a lot of the smaller stuff. For these rules, we make and document a decision primarily to make and document a decision. Many rules in this category don't have significant technical impact. Things like naming conventions, indentation spacing, import ordering: there is usually no clear, measurable, technical benefit for one form over another, which might be why the technical community tends to keep debating them.[^11] By choosing one, we've dropped out of the endless debate cycle and can just move on. Our engineers no longer spend time discussing two spaces versus four. The important bit for this category of rules is not what we've chosen for a given rule so much as the fact that we have chosen.

我們的風格指南還包含了一些規則，涵蓋了許多較小的內容。對於這些規則，我們主要是為了做出和記錄一個決定。這類規則中的許多規則都沒有重大的技術影響。諸如命名約定、縮排間距、匯入順序等:通常一種形式相對於另一種形式沒有明確的、可衡量的技術優勢，這可能是技術社群傾向於對它們爭論不休的原因。選擇一個，我們就退出了無休止的辯論迴圈，可以繼續前進了。我們的工程師不再花時間去討論兩個空格與四個空格。這類規則的重要部分不是我們為給定的規則選擇的內容，而是我們選擇的這個動作本身。

> 11 Such discussions are really just bikeshedding, an illustration of Parkinson's law of triviality.

> 11 這樣的討論實際上只是在騎腳踏車，是對帕金森法則的零散的描述。

And for everything else... 至於其他的一切……

With all that, there's a lot that's not in our style guides. We try to focus on the things that have the greatest impact on the health of our codebase. There are absolutely best practices left unspecified by these documents, including many fundamental pieces of good engineering advice: don't be clever, don't fork the codebase, don't reinvent the wheel, and so on. Documents like our style guides can't serve to take a complete novice all the way to a master-level understanding of software engineering—there are some things we assume, and this is intentional.

除此之外，還有很多內容不在我們的風格指南中。我們試著專注於那些對我們程式碼庫的健康狀況有重大影響的事情。這些文件中絕對有一些沒有詳細說明的最佳實踐，包括許多很好的工程建議:不要自做聰明，不要分支程式碼庫，不要重新發明輪子，等等。像我們的風格指南這樣的文件不能讓一個完全的新手擁有軟體工程大師的理解——有些事情是我們假設的，這是故意的。

Changing the Rules 改變規則

Our style guides aren't static. As with most things, given the passage of time, the landscape within which a style guide decision was made and the factors that guided a given ruling are likely to change. Sometimes, conditions change enough to warrant reevaluation. If a new language version is released, we might want to update our rules to allow or exclude new features and idioms. If a rule is causing engineers to invest effort to circumvent it, we might need to reexamine the benefits the rule was supposed to provide. If the tools that we use to enforce a rule become overly complex and burdensome to maintain, the rule itself might have decayed and need to be revisited. Noticing when a rule is ready for another look is an important part of the process that keeps our rule set relevant and up to date.

我們的風格指南不是一成不變的。與大多數事情一樣，隨著時間的推移，做出風格指導決策的環境和指導給定裁決的因素可能會發生變化。有時，情況的變化足以使人們需要重新評估。如果釋出了新的語言版本，我們可能需要更新規則以允許或排除新的特性和習慣用法。如果某個規則是工程師需要付出努力來規避它的，我們可能需要重新審視該規則本來應該提供的好處。如果我們用來執行規則的工具變得過於複雜和難於維護，那麼規則本身可能已經腐化，需要重新修訂。注意到一條規則何時準備好，以便再次審視，這是過程中的一個重要部分，它使我們的規則集保持相關性和時效性。

The decisions behind rules captured in our style guides are backed by evidence. When adding a rule, we spend time discussing and analyzing the relevant pros and cons as well as the potential consequences, trying to verify that a given change is appropriate for the scale at which Google operates. Most entries in Google's style guides include these considerations, laying out the pros and cons that were weighed during the process and giving the reasoning for the final ruling. Ideally, we prioritize this detailed reasoning and include it with every rule.

在我們的風格指南中，規則背後的決定是有證據支援的。在新增規則時，我們將花時間討論和分析相關的利弊以及潛在的後果，並試圖驗證給定的更改是否適合谷歌營運規模。谷歌風格指南中的大多數條目都包含了這些考慮因素，列出了在過程中權衡的利弊，並給出了最終裁決的理由。理想情況下，我們優先考慮這種詳細的推理，並將其包含在每條規則中。

Documenting the reasoning behind a given decision gives us the advantage of being able to recognize when things need to change. Given the passage of time and changing conditions, a good decision made previously might not be the best current one. With influencing factors clearly noted, we are able to identify when changes related to one or more of these factors warrant reevaluating the rule.

記錄給定決策背後的原因，使我們能夠識別何時需要更改。考慮到時間的流逝和環境的變化，以前做出的好決定可能不是現在的最佳決定。明確地指出影響因素後，我們就能夠確定何時與一個或多個因素相關的更改需要重新評估規則。

Case Study:CamelCase Naming 案例分析：駝峰命名法

At Google, when we defined our initial style guidance for Python code, we chose to use CamelCase naming style instead of snake_case naming style for method names. Although the public Python style guide (PEP 8) and most of the Python community used snake_case naming, most of Google's Python usage at the time was for C++ developers using Python as a scripting layer on top of a C++ codebase. Many of the defined Python types were wrappers for corresponding C++ types, and because Google's C++ naming conventions follow CamelCase style, the cross-language consistency was seen as key.

在谷歌，當我們為 Python 程式碼定義初始風格指導時，我們選擇使用駝峰命名風格，而不是使用 snake_case 命名風格來命名方法名。儘管公共 Python 風格指南(PEP 8)和大多數 Python 社群使用了 snake_case 命名，但當時谷歌的大多數 Python 用法是為 c++開發人員使用 Python 作為 c++程式碼庫之上的指令碼層。許多定義的 Python 型別是相應 c++型別的包裝器，因為 Google 的 C++命名慣例遵循駝峰命名風格，在這裡跨語言的一致性被視為關鍵。

Later, we reached a point at which we were building and supporting independent Python applications. The engineers most frequently using Python were Python engineers developing Python projects, not C++ engineers pulling together a quick script. We were causing a degree of awkwardness and readability problems for our Python engineers, requiring them to maintain one standard for our internal code but constantly adjust for another standard every time they referenced external code. We were also making it more difficult for new hires who came in with Python experience to adapt to our codebase norms.

後來，我們到了建構和支援獨立 Python 應用程式的地步。最經常使用 Python 的工程師是那些開發 Python 專案的 Python 工程師，而不是編寫快速指令碼的 C++工程師。我們給我們的 Python 工程師造成了一定程度的尷尬和可讀性問題，要求他們為我們的內部程式碼維護一個標準，但每次他們參考外部程式碼時都要不斷地調整另一個標準。我們還讓有 Python 經驗的新員工更難以適應我們的程式碼庫規範。

As our Python projects grew, our code more frequently interacted with external Python projects. We were incorporating third-party Python libraries for some of our projects, leading to a mix within our codebase of our own CamelCase format with the externally preferred snake_case style. As we started to open source some of our Python projects, maintaining them in an external world where our conventions were nonconformist added both complexity on our part and wariness from a community that found our style surprising and somewhat weird.

隨著 Python 專案的增長，我們的程式碼與外部 Python 專案的互動越來越頻繁。我們在一些專案中合併了第三方 Python 庫，導致我們的程式碼庫中混合了我們自己的駝峰命名格式和外部偏愛的 snake_case 樣式。當我們開始開源我們的一些 Python 專案時，在一個我們的沒有約定規範限制的外部世界中維護它們，既增加了我們的複雜性，也增加了社群對我們風格的警惕，他們覺得我們的風格令人驚訝，有些怪異。

Presented with these arguments, after discussing both the costs (losing consistency with other Google code, reeducation for Googlers used to our Python style) and benefits (gaining consistency with most other Python code, allowing what was already leaking in with third-party libraries), the style arbiters for the Python style guide decided to change the rule. With the restriction that it be applied as a file-wide choice, an exemption for existing code, and the latitude for projects to decide what is best for them, the Google Python style guide was updated to permit snake_case naming.

提出這些論點後，討論了成本(失去與其他谷歌程式碼的一致性，對習慣於我們的 Python 風格的 Google 員工進行再教育)和好處(獲得與大多數其他 Python 程式碼的一致性，允許已經洩露到第三方庫的內容)，Python 風格指南的風格仲裁者決定改變規則。有了它被應用為檔案範圍的選擇的限制，現有程式碼的例外，以及專案決定什麼最適合他們的自由度，谷歌 Python 風格指南已更新為允許 snake_case 命名。

The Process 過程

Recognizing that things will need to change, given the long lifetime and ability to scale that we are aiming for, we created a process for updating our rules. The process for changing our style guide is solution based. Proposals for style guide updates are framed with this view, identifying an existing problem and presenting the proposed change as a way to fix it. “Problems,” in this process, are not hypothetical examples of things that could go wrong; problems are proven with patterns found in existing Google code. Given a demonstrated problem, because we have the detailed reasoning behind the existing style guide decision, we can reevaluate, checking whether a different conclusion now makes more sense.

考慮到我們所追求的長生命週期和擴充能力，我們認識到事情需要改變，因此我們建立了一個更新規則的過程。改變我們的風格指南的過程是基於解決方案的。風格指南更新的建議是以此觀點為框架的，識別現有的問題，並將建議的更改作為修復問題的一種方法。在這個過程中，“問題”並不是可能出錯的假設例子;問題是透過在現有谷歌程式碼中發現的模式來證明的。給定一個被證明的問題，因為我們有現有風格指南決策背後的詳細理由，我們可以重新評估，檢查和之前不同的結論現在是否更有意義。

The community of engineers writing code governed by the style guide are often best positioned to notice when a rule might need to be changed. Indeed, here at Google, most changes to our style guides begin with community discussion. Any engineer can ask questions or propose a change, usually by starting with the language-specific mailing lists dedicated to style guide discussions.

編寫風格指南所規範的程式碼的工程師群體，往往最能注意到何時需要改變規則。事實上，在谷歌，我們的風格指南的大多數改變都是從社群討論開始的。任何工程師都可以提出問題或提出更改建議，通常從專門用於討論風格指南的特定語言郵件列表開始。

Proposals for style guide changes might come fully-formed, with specific, updated wording suggested, or might start as vague questions about the applicability of a given rule. Incoming ideas are discussed by the community, receiving feedback from other language users. Some proposals are rejected by community consensus, gauged to be unnecessary, too ambiguous, or not beneficial. Others receive positive feedback, gauged to have merit either as-is or with some suggested refinement. These proposals, the ones that make it through community review, are subject to final decision- making approval.

關於風格指南更改的建議可能是完整的，包括建議的具體的、更新的建議，或者可能以關於給定規則的適用性的模糊問題開始。社群會討論新想法，並接收其他語言使用者的反饋。一些提案被社群共識拒絕，被認為是不必要的、過於模糊的或無益的。另一些則收到了積極的反饋，被認為是有價值的，或者有一些建議的改進。這些提案，經過社群審查後，需要得到最終決策的批准。

The Style Arbiters 風格仲裁者

At Google, for each language's style guide, final decisions and approvals are made by the style guide's owners—our style arbiters. For each programming language, a group of long-time language experts are the owners of the style guide and the designated decision makers. The style arbiters for a given language are often senior members of the language's library team and other long-time Googlers with relevant language experience.

在谷歌，對於每種語言的風格指南，最終的決定和批准都是由風格指南的所有者——我們的風格仲裁者——做出的。對於每一種程式語言，都有一群資深的語言專家是風格指南的所有者和指定的決策者。特定語言的風格仲裁人通常是該語言庫團隊的高階成員，以及其他具有相關語言經驗的長期谷歌員工。

The actual decision making for any style guide change is a discussion of the engineering trade-offs for the proposed modification. The arbiters make decisions within the context of the agreed-upon goals for which the style guide optimizes. Changes are not made according to personal preference; they're trade-off judgments. In fact, the C++ style arbiter group currently consists of four members. This might seem strange: having an odd number of committee members would prevent tied votes in case of a split decision. However, because of the nature of the decision making approach, where nothing is "because I think it should be this way" and everything is an evaluation of trade-off, decisions are made by consensus rather than by voting. The four-member group is happily functional as-is.

對於任何風格指南的更改，實際的決策都是對提議修改的工程權衡的反覆討論。仲裁者在風格指南最佳化的一致目標上下文中做出決定。更改並非根據個人喜好。它們是權衡判斷。事實上，C++風格仲裁組目前由四個成員組成。這可能看起來很奇怪:如果委員會成員人數為奇數，就可以防止出現出現意見分歧的情況，出現票數平手的情況。然而，由於決策制定方法的本質，沒有什麼是"因為我認為它應該是這樣的"，一切都是一種權衡，決策是透過共識而不是投票做出的。這個由四名成員組成的小組就這樣愉快地運作著。

Exceptions 例外

Yes, our rules are law, but yes, some rules warrant exceptions. Our rules are typically designed for the greater, general case. Sometimes, specific situations would benefit from an exemption to a particular rule. When such a scenario arises, the style arbiters are consulted to determine whether there is a valid case for granting a waiver to a particular rule.

沒錯，我們的規則就是法律，但也有例外。我們的規則通常是為更大的一般情況而設計的。有時，特定的情況會受益於對特定規則的豁免。當出現這種情況時，會諮詢風格仲裁者，以確定是否存在授予某個特定規則豁免的有效案例。

Waivers are not granted lightly. In C++ code, if a macro API is introduced, the style guide mandates that it be named using a project-specific prefix. Because of the way C++ handles macros, treating them as members of the global namespace, all macros that are exported from header files must have globally unique names to prevent collisions. The style guide rule regarding macro naming does allow for arbiter-granted exemptions for some utility macros that are genuinely global. However, when the reason behind a waiver request asking to exclude a project-specific prefix comes down to preferences due to macro name length or project consistency, the waiver is rejected. The integrity of the codebase outweighs the consistency of the project here.

豁免不是輕易就能獲得的。在 C++程式碼中，如果引入了宏 API，風格指南要求使用特定於專案的字首來命名它。由於 C++處理宏的方式，將它們視為全域名稱空間的成員，所有從標頭檔案匯出的宏必須具有全域唯一的名稱，以防止衝突。關於宏命名的風格指南規則確實允許對一些真正全域的實用宏進行仲裁授予的豁免。但是，當請求排除專案特定字首的豁免請求背後的原因歸結為由於宏名稱長度或專案一致性而導致的偏好時，豁免被拒絕。這裡程式碼庫的完整性勝過專案的一致性。

Exceptions are allowed for cases in which it is gauged to be more beneficial to permit the rule-breaking than to avoid it. The C++ style guide disallows implicit type conversions, including single-argument constructors. However, for types that are designed to transparently wrap other types, where the underlying data is still accurately and precisely represented, it's perfectly reasonable to allow implicit conversion. In such cases, waivers to the no-implicit-conversion rule are granted. Having such a clear case for valid exemptions might indicate that the rule in question needs to be clarified or amended. However, for this specific rule, enough waiver requests are received that appear to fit the valid case for exemption but in fact do not—either because the specific type in question is not actually a transparent wrapper type or because the type is a wrapper but is not actually needed—that keeping the rule in place as-is is still worthwhile.

在被認為允許違反規則比避免違反規則更有利的情況下，允許例外。C++ 風格指南不允許隱含型別轉換，包括單引數建構函式。然而，對於那些被設計成透明地包裝其他型別的型別，其中的底層資料仍然被精確地表示出來，允許隱含轉換是完全合理的。在這種情況下，授予對無隱含轉換規則的豁免。擁有如此明確的有效豁免案例可能表明需要澄清或修改相關規則。然而,對於此特定規則，收到了足夠多的豁免請求，這些請求似乎適合豁免的有效案例，但實際上不符合。因為所討論的特定型別實際上不是透明包裝器型別，或者因為該型別是包裝器但實際上並不需要——保持原樣的規則仍然是值得的。

Guidance 指導

In addition to rules, we curate programming guidance in various forms, ranging from long, in-depth discussion of complex topics to short, pointed advice on best practices that we endorse.

除了規則之外，我們還以各種形式提供程式設計指導，從對複雜主題的長而深入的討論到對我們認可的最佳實踐的簡短而有針對性的建議。

Guidance represents the collected wisdom of our engineering experience, documenting the best practices that we've extracted from the lessons learned along the way. Guidance tends to focus on things that we've observed people frequently getting wrong or new things that are unfamiliar and therefore subject to confusion. If the rules are the "musts," our guidance is the "shoulds."

指導代表了我們收集的工程經驗的智慧，記錄了我們從一路走來的經驗教訓中提取的最佳實踐。指導往往側重於我們觀察到人們經常出錯的事情或不熟悉並因此容易混淆的新事物。如果規則是"必須"，那麼我們的指導就是"應該"。

One example of a pool of guidance that we cultivate is a set of primers for some of the predominant languages that we use. While our style guides are prescriptive, ruling on which language features are allowed and which are disallowed, the primers are descriptive, explaining the features that the guides endorse. They are quite broad in their coverage, touching on nearly every topic that an engineer new to the language's use at Google would need to reference. They do not delve into every detail of a given topic, but they provide explanations and recommended use. When an engineer needs to figure out how to apply a feature that they want to use, the primers aim to serve as the go-to guiding reference.

我們培養的指導庫的一個例子是我們使用的一些主要語言的編寫入門指南。雖然我們的風格指南是規範性的，規定了哪些語言特

性是允許的，哪些是不允許的，而入門指南是描述性的，解釋了指南認可的特性。他們的內容相當廣泛，幾乎涵蓋了谷歌新使用該語言的工程師需要參考的所有主題。它們不會深入研究特定主題的每一個細節，但它們提供解釋和推薦使用。當工程師需要弄清楚如何應用他們想要使用的功能時，入門指南的目的是作為指導參考。

A few years ago, we began publishing a series of C++ tips that offered a mix of general language advice and Google-specific tips. We cover hard things—object lifetime, copy and move semantics, argument-dependent lookup; new things—C++ 11 features as they were adopted in the codebase, preadopted C++17 types like string_view, optional, and variant; and things that needed a gentle nudge of correction—reminders not to use using directives, warnings to remember to look out for implicit bool conversions. The tips grow out of actual problems encountered, addressing real programming issues that are not covered by the style guides. Their advice, unlike the rules in the style guide, are not true canon; they are still in the category of advice rather than rule. However, given the way they grow from observed patterns rather than abstract ideals, their broad and direct applicability set them apart from most other advice as a sort of “canon of the common.” Tips are narrowly focused and relatively short, each one no more than a few minutes’ read. This “Tip of the Week” series has been extremely successful internally, with frequent citations during code reviews and technical discussions.[^12]

幾年前，我們開始釋出一系列 C++提示，其中包括通用語言建議和谷歌特有的提示。我們涵蓋了困難的事情——物件生命期、複製和移動語義、依賴於引數的查詢;新事物- C++11 特性，它們在程式碼庫中被採用，預採用的 C++17 型別，如 string_view, optional, 和 variant;還有一些東西需要溫和的調整——提醒不要使用 using 指令，警告要記住尋找隱含布林轉換。這些技巧來源於所遇到的實際問題，解決了樣式指南中沒有涉及的實際程式設計問題。與風格指南中的規則不同，它們的建議並不是真正的經典;它們仍然屬於建議而非規則的範疇。然而，考慮到它們是從觀察到的模式而不是抽象的理想中發展出來的，它們廣泛而直接的適用性使它們有別於大多數其他建議，成為一種“共同的經典”。這些小貼士的內容都比較狹隘，篇幅也相對較短，每一條都不超過幾分鐘的閱讀時間。這個“每週技巧”系列在內部已經非常成功，在程式碼評審和技術討論中經常被參考。

Software engineers come in to a new project or codebase with knowledge of the programming language they are going to be using, but lacking the knowledge of how the programming language is used within Google. To bridge this gap, we maintain a series of “@Google 101” courses for each of the primary programming languages in use. These full-day courses focus on what makes development with that language different in our codebase. They cover the most frequently used libraries and idioms, in-house preferences, and custom tool usage. For a C++ engineer who has just become a Google C++ engineer, the course fills in the missing pieces that make them not just a good engineer, but a good Google codebase engineer.

軟體工程師進入一個新的專案或程式碼庫時，已經掌握了他們將要使用的程式語言的知識，但缺乏關於如何在 Google 中使用該程式語言的知識。為了彌補這一差距，我們為使用中的每一種主要程式語言設定了一系列的“<語言>@Google 101”課程。這些全日制課程側重於在我們的程式碼庫中使用該語言進行開發的不同之處。它們涵蓋了最常用的函式庫和習慣用法、內部首選項和自訂工具的使用。對於一個剛剛成為谷歌 C++工程師的 C++工程師，該課程填補了缺失的部分，使他們不僅是一名優秀的工程師，而且是一名優秀的 Google 程式碼庫工程師。

In addition to teaching courses that aim to get someone completely unfamiliar with our setup up and running quickly, we also cultivate ready references for engineers deep in the codebase to find the information that could help them on the go. These references vary in form and span the languages that we use. Some of the useful references that we maintain internally include the following:

- Language-specific advice for the areas that are generally more difficult to get correct (such as concurrency and hashing).

- Detailed breakdowns of new features that are introduced with a language update and advice on how to use them within the codebase.
- Listings of key abstractions and data structures provided by our libraries. This keeps us from reinventing structures that already exist and provides a response to, “I need a thing, but I don’t know what it’s called in our libraries.”

除了教授旨在讓完全不熟悉我們的人快速執行的課程外，我們還為深入程式碼庫的工程師培養現成的參考資料，以便在學習途中找到可以幫助他們的資訊。這些參考資料的形式各不相同，並且跨越了我們使用的語言。我們內部維護的一些有用的參考資料包括：

- 針對通常很難正確處理的領域(如併發性和雜湊)提供特定語言的建議。
- 語言更新中引入的新特性的詳細分解，以及如何在程式碼庫中使用它們的建議。
- 我們庫提供的關鍵抽象和資料結構列表。這阻止了我們重新建立已經存在的結構，並提供了對“我需要一個東西，但我不知道它在我們的函式庫中叫什麼”的回應。

> 12 https://abseil.io/tips has a selection of some of our most popular tips.
>
> 12 https://abseil.io/tips 選擇了一些我們最受歡迎的提示。

Applying the Rules 應用規則

Rules, by their nature, lend greater value when they are enforceable. Rules can be enforced socially, through teaching and training, or technically, with tooling. We have various formal training courses at Google that cover many of the best practices that our rules require. We also invest resources in keeping our documentation up to date to ensure that reference material remains accurate and current. A key part of our overall training approach when it comes to awareness and understanding of our rules is the role that code reviews play. The readability process that we run here at Google —where engineers new to Google’s development environment for a given language are mentored through code reviews—is, to a great extent, about cultivating the habits and patterns required by our style guides (see details on the readability process in Chapter 3). The process is an important piece of how we ensure that these practices are learned and applied across project boundaries.

就其本質而言，規則在可執行的情況下會帶來更大的價值。規則可以透過教學和培訓在社會上執行，也可以在技術上透過工具來執行。我們在谷歌提供了各種正式的培訓課程，涵蓋了我們的規則所要求的許多最佳實踐。我們還投入資源使我們的文件保持更新，以確保參考材料保持準確和最新。當涉及到對規則的瞭解和理解時，我們整體培訓方法的一個關鍵部分是程式碼評審所扮演的角色。我們在谷歌執行的可讀性過程——在這裡，針對特定語言的谷歌開發環境的新工程師會透過程式碼審查來指導——在很大程度上是為了培養我們的風格指南所要求的習慣和模式(詳見第 3 章可讀性過程) 。這個過程是我們如何確保這些實踐是跨專案邊界學習和應用的重要部分。

Although some level of training is always necessary—engineers must, after all, learn the rules so that they can write code that follows them—when it comes to checking for compliance, rather than exclusively depending on engineer-based verification, we strongly prefer to automate enforcement with tooling.

儘管一定程度的培訓總是必要的——畢竟，工程師必須學習規則，這樣他們才能編寫遵循規則的程式碼——但在檢查合規性時，我們強烈傾向於不完全依賴基於工程師的驗證使用工具自動執行。

Automated rule enforcement ensures that rules are not dropped or forgotten as time passes or as an organization scales up. New people join; they might not yet know all the rules. Rules change over time; even with good communication, not everyone will remember the current state of everything. Projects grow and add new features; rules that had previously not been relevant are suddenly applicable. An engineer checking for rule compliance depends on either memory or documentation, both of which can fail. As long as our tooling stays up to date, in sync with our rule changes, we know that our rules are being applied by all our engineers for all our projects.

自動執行規則可確保隨著時間的推移或組織規模的擴大，規則不會被丟棄或遺忘。新的人加入；他們可能還不知道所有的規則。規則會隨著時間而改變;即使有良好的溝通，也不是每個人都能記住所有事情的當前狀態。專案不斷髮展並新增新功能;以前不相關的規則突然適用了。檢查規則合規性的工程師取決於記憶或文件，這兩者都可能失敗。只要我們的工具保持更新，與我們的規則更改同步，我們就知道我們的規則被我們所有的工程師應用於我們所有的專案。

Another advantage to automated enforcement is minimization of the variance in how a rule is interpreted and applied. When we write a script or use a tool to check for compliance, we validate all inputs against a single, unchanging definition of the rule. We aren't leaving interpretation up to each individual engineer. Human engineers view everything with a perspective colored by their biases. Unconscious or not, potentially subtle, and even possibly harmless, biases still change the way people view things. Leaving enforcement up to engineers is likely to see inconsistent interpretation and application of the rules, potentially with inconsistent expectations of accountability. The more that we delegate to the tools, the fewer entry points we leave for human biases to enter.

自動執行的另一個優點是最大限度地減少了規則解釋和應用的差異性。當我們編寫指令碼或使用工具檢查合規性時，我們會根據一個單一的、不變的規則定義來驗證所有輸入。我們不會將解釋留給每個單獨的工程師。人類工程師以帶有偏見的視角看待一切。無論是否是無意識的，潛在的微妙的，甚至可能是無害的，偏見仍然會改變人們看待事物的方式。如果讓工程師來執行這些規則，可能會導致對規則的解釋和應用不一致，對責任的期望也可能不一致。我們授權給工具的越多，留給人類偏見進入的入口就越少。

Tooling also makes enforcement scalable. As an organization grows, a single team of experts can write tools that the rest of the company can use. If the company doubles in size, the effort to enforce all rules across the entire organization doesn't double, it costs about the same as it did before.

工具還使執行具有可擴充性。隨著組織的成長，一個專家團隊就可以編寫公司其他部門都可以使用的工具。如果公司的規模擴大一倍，在整個組織內執行所有規則的成本不會增加一倍，它與以前差不多。

Even with the advantages we get by incorporating tooling, it might not be possible to automate enforcement for all rules. Some technical rules explicitly call for human judgment. In the C++ style guide, for example: "Avoid complicated template metaprogramming." "Use auto to avoid type names that are noisy, obvious, or unimportant—cases where the type doesn't aid in clarity for the reader." "Composition is often more appropriate than inheritance." In the Java style guide: "There's no single correct recipe for how to [order the members and initializers of your class]; different classes may order their contents in different ways." "It is very rarely correct to do nothing in response to a caught exception." "It is extremely rare to override Object.finalize." For all of these rules, judgment is required and tooling can't (yet!) take that place.

即使具有我們透過合併工具獲得的優勢，也可能無法自動執行所有規則。一些技術規則明確要求人的判斷。例如，在 C++風格指南中:“避免複雜的範本超程式設計。”“使用 auto 來避免冗長、明顯或不重要的型別名，這些型別名不能幫助讀者清晰地讀懂。“組合通常比繼承更合適。”在 Java 風格指南中:“對於如何[對類別的成員和初始化程式進行排序]並沒有一個正確的方法;不同的類可能以不同的方式排列內容。”“對所捕獲的例外不做任何響應很少是正確的。"覆蓋 Object.finalize 的情況極為罕見。"對於所有這些規則，都需要人工判斷，工具目前還不能代替判斷。

Other rules are social rather than technical, and it is often unwise to solve social problems with a technical solution. For many of the rules that fall under this category, the details tend to be a bit less well defined and tooling would become complex and expensive. It's often better to leave enforcement of those rules to humans. For example, when it comes to the size of a given code change (i.e., the number of files affected and lines modified) we recommend that engineers favor smaller changes. Small changes are easier for engineers to review, so reviews tend to be faster and more thorough. They're also less likely to introduce bugs because it's easier to reason about the potential impact and effects of a smaller change. The definition of small, however, is somewhat nebulous. A change that propagates the identical one-line update across hundreds of files might actually be easy to review. By contrast, a smaller, 20-line change might introduce complex logic with side effects that are difficult to evaluate. We recognize that there are many different measurements of size, some of which may be subjective—particularly when taking the complexity of a change into account. This is why we do not have any tooling to autoreject a proposed change that exceeds an arbitrary line limit. Reviewers can (and do) push back if they judge a change to be too large. For this and similar rules, enforcement is up to the discretion of the engineers authoring and reviewing the code. When it comes to technical rules, however, whenever it is feasible, we favor technical enforcement.

其他規則是社會性的而不是技術性的，用技術性的解決方案來解決社會性問題通常是不明智的。對於這個類別下的許多規則，細節往往不太明確，工具將變得複雜且昂貴。將這些規則的執行留給人類通常會更好。例如，當涉及到給定程式碼更改的大小（即受影響的檔案數和修改的行數）時，我們建議工程師傾向於較小的更改。對於工程師來說，小的變更更容易稽核，所以稽核往往更快、更可靠。它們也不太可能引入 bug，因為更容易推斷出較小更改的潛在影響和效果。然而，“小”的定義有些模糊。一個在數百個檔案中傳播相同的單行更新的變化實際上可能很容易審查。相比之下，一個較小的 20 行修改可能會引入複雜的邏輯，併產生難以評估的副作用。我們認識到有許多不同的衡量尺度，其中一些可能是主觀的——特別是當考慮到變化的複雜性時。這就是為什麼我們沒有任何工具來自動拒絕超過任意行限制的建議更改。如果審閱者認為更改過大，他們可以(而且確實會)推回。對於這種規則和類似的規則，執行由編寫和審查程式碼的工程師自行決定。然而，當涉及到技術規則時，只要是可行的，我們傾向於技術執行。

Error Checkers 錯誤檢查器

Many rules covering language usage can be enforced with static analysis tools. In fact, an informal survey of the C++ style guide by some of our C++ librarians in mid-2018 estimated that roughly 90% of its rules could be automatically verified. Error-checking tools take a set of rules or patterns and verify that a given code sample fully complies. Automated verification removes the burden of remembering all applicable rules from the code author. If an engineer only needs to look for violation warnings— many of which come with suggested fixes—surfaced during code review by an analyzer that has been tightly integrated into the development workflow, we minimize the effort that it takes to comply with the rules. When we began using tools to flag deprecated functions based on source tagging, surfacing both the warning and the suggested fix in-place, the problem of having new usages of deprecated APIs disappeared almost overnight. Keeping the cost of compliance down makes it more likely for engineers to happily follow through.

許多涉及語言使用的規則可以透過靜態分析工具強制執行。事實上，我們的一些 C++類別庫管理員在 2018 年年中對 C++風格指南進行的一項非正式調查估計，其中大約 90%的規則可以自動驗證。錯誤檢查工具採用一組規則或模式，並驗證給定的程式碼範例是否完全符合。自動驗證消除了程式碼作者記住所有適用規則的負擔。如果工程師只需要查詢違規警告——其中許多都帶有建議的修復——在程式碼審查期間由已緊密整合到開發工作流中的分析器發現的，我們將盡可能減少遵守規則所需要的工作量。當我們開始使用工具基於源標籤來標記已棄用的函式時，警告和建議的就都會同時給出，新使用已棄用 API 的問題幾乎在一夜之間消失了。降低合規成本使工程師更有可能愉快地貫徹執行。

We use tools like clang-tidy (for C++) and Error Prone (for Java) to automate the process of enforcing rules. See Chapter 20 for an in-depth discussion of our approach.

我們使用像 clang-tidy(用於 C++)和 Error Prone(用於 Java)這樣的工具來自動化執行規則的過程。有關我們方法的深入討論，請參見第 20 章。

The tools we use are designed and tailored to support the rules that we define. Most tools in support of rules are absolutes; everybody must comply with the rules, so everybody uses the tools that check them. Sometimes, when tools support best practi-ces where there's a bit more flexibility in conforming to the conventions, there are opt-out mechanisms to allow projects to adjust for their needs.

我們使用的工具都是為支援我們定義的規則而設計和客製的。大多數支援規則的工具都是絕對的;每個人都必須遵守規則，所以每個人都使用檢查規則的工具。有時，當工具支援最佳實踐時，在遵守約定方面有更多的靈活性，就會有選擇退出機制，允許專案根據自己的需要進行調整。

Code Formatters 程式碼格式器

At Google, we generally use automated style checkers and formatters to enforce consistent formatting within our code. The question of line lengths has stopped being interesting.[^13]Engineers just run the style checkers and keep moving forward. When formatting is done the same way every time, it becomes a non-issue during code review, eliminating the review cycles that are otherwise spent finding, flagging, and fixing minor style nits.

在谷歌，我們通常使用自動樣式檢查器和格式化器來在我們的程式碼中執行一致的格式。行長度的問題已經不再有趣了。工程師只需執行樣式檢查器並繼續前進。如果每次都以相同的方式進行格式化，那麼在程式碼審查期間就不會出現問題，從而消除了用來查詢、標記和修復要樣式細節的審查週期。

In managing the largest codebase ever, we've had the opportunity to observe the results of formatting done by humans versus formatting done by automated tooling. The robots are better on average than the humans by a significant amount. There are some places where domain expertise matters—formatting a matrix, for example, is something a human can usually do better than a general-purpose formatter. Failing that, formatting code with an automated style checker rarely goes wrong.

在管理有史以來最大的程式碼庫時，我們有機會觀察人工格式化和自動化工具格式化的結果。平均而言，機器人比人類好很多。在某些地方，領域專業知識很重要——例如，格式化矩陣，人工通常可以比通用格式化程式做得更好。如果做不到這一點，用自動樣式檢查器格式化程式碼很少出錯。

We enforce use of these formatters with presubmit checks: before code can be submitted, a service checks whether running the formatter on the code produces any diffs. If it does, the submit is rejected with instructions on how to run the formatter to fix the code. Most code at Google is subject to such a presubmit check. For our code, we use clang-format for C++; an in-house wrapper around yapf for Python; gofmt for Go; dartfmt for Dart; and buildifier for our BUILD files.

我們透過預提交檢查強制使用這些格式化程式:在提交程式碼之前，服務會檢查在程式碼上執行格式化器是否會產生任何差異。如果是，提交將被拒絕，並提供有關如何執行格式化程式以修復程式碼的說明。谷歌上的大多數程式碼都要接受這種預提交檢查。在我們的程式碼中，C++使用了 clang-format;Python 使用 yapf 內部包裝器;Go 使用 gofmt; Dart 使用 dartfmt;以及我們的 BUILD 檔案使用 buildifier。

> 13 When you consider that it takes at least two engineers to have the discussion and multiply that by the number of times this conversation is likely to happen within a collection of more than 30,000 engineers, it turns out that “how many characters” can become a very expensive question.
>
> 13 當你考慮到至少需要兩名工程師進行討論，並將其乘以這種對話可能在 30,000 多名工程師的集合中發生的次數時，事實證明，"多少個字元 "可能成為一個成本非常高的問題。

Case Study: gofmt 案例分析：go 格式化工具

Sameer Ajmani

薩米爾·阿吉馬尼

Google released the Go programming language as open source on November 10, 2009. Since then, Go has grown as a language for developing services, tools, cloud infrastructure, and open source software.[10]

谷歌於 2009 年 11 月 10 日以開源方式釋出了 Go 程式語言。從那時起，Go 已經發展成為一種開發服務、工具、雲基礎設施和開源軟體的語言。

We knew that we needed a standard format for Go code from day one. We also knew that it would be nearly impossible to retrofit a standard format after the open source release. So the initial Go release included gofmt, the standard formatting tool for Go.

我們從一開始就知道我們需要一個 Go 程式碼的標準格式。我們也知道，在開源版本釋出後，幾乎不可能改造標準格式。因此，最初的 Go 版本包括 gofmt，這是 Go 的標準格式工具。

Motivations 動機

Code reviews are a software engineering best practice, yet too much time was spent in review arguing over formatting. Although a standard format wouldn’t be everyone’s favorite, it would be good enough to eliminate this wasted time.[11]

程式碼審查是一種軟體工程的最佳實踐，但是太多的時間被花在評審中爭論格式。雖然標準格式不是每個人都喜歡的，但它足以消除這些浪費的時間。

By standardizing the format, we laid the foundation for tools that could automatically update Go code without creating spurious diffs: machine-edited code would be indistinguishable from human-edited code.[12]

透過標準化格式，我們為可以自動更新 Go 程式碼而不會建立虛假差異的工具奠定了基礎：機器編輯的程式碼與人工編輯的程式碼無法區分。

For example, in the months leading up to Go 1.0 in 2012, the Go team used a tool called gofix to automatically update pre-1.0 Go code to the stable version of the language and libraries. Thanks to gofmt, the diffs gofix produced included only the important bits: changes to uses of the language and APIs. This allowed programmers to more easily review the changes and learn from the changes the tool made.

例如，在 2012 年 Go 1.0 的前幾個月，Go 團隊使用了一個名為 gofix 的工具來自動將 1.0 之前的 Go 程式碼更新到語言和庫的穩定版本。多虧了 gofmt, gofix 產生的差異只包括重要的部分：語言和 API 使用的更改。 這使程式設計師可以更輕鬆地檢視更改並從工具所做的更改中學習。

Impact 影響

Go programmers expect that all Go code is formatted with gofmt. gofmt has no configuration knobs, and its behavior rarely changes. All major editors and IDEs use gofmt or emulate its behavior, so nearly all Go code in existence is formatted identically. At first, Go users complained about the enforced standard; now, users often cite gofmt as one of the many reasons they like Go. Even when reading unfamiliar Go code, the format is familiar.

Go 程式設計師希望所有的 Go 程式碼都使用 gofmt 格式。gofmt 沒有配置旋鈕，它的行為很少改變。所有主要的編輯器和 IDE 都使用 gofmt 或者模仿它的行為，所以幾乎所有的 Go 程式碼都採用了相同的格式。起初，Go 使用者抱怨強制執行的標準；現在，使用者經常將 gofmt 作為他們喜歡 Go 的眾多原因之一。即使閱讀不熟悉的 Go 程式碼，格式也是熟悉的。

Thousands of open source packages read and write Go code.[13] Because all editors and IDEs agree on the Go format, Go tools are portable and easily integrated into new developer environments and workflows via the command line.

成千上萬的開源包讀取和編寫 Go 程式碼。因為所有的編輯器和 IDE 都同意 Go 格式，所以 Go 工具是可移植的，並且很容易透過命令列整合到新的開發環境和工作流中。

Retrofitting 改裝

In 2012, we decided to automatically format all BUILD files at Google using a new standard formatter: buildifier. BUILD files contain the rules for building Google's software with Blaze, Google's build system. A standard BUILD format would enable us to create tools that automatically edit BUILD files without disrupting their format, just as Go tools do with Go files.

在 2012 年，我們決定使用一個新的標準格式器來自動格式化谷歌中的所有 BUILD 檔案:buildifier。BUILD 檔案包含了使用 Blaze(谷歌的建構系統)建構谷歌軟體的規則。標準的 BUILD 格式將使我們能夠建立自動編輯 BUILD 檔案而不破壞其格式的工具，就像 Go 工具對 Go 檔案所做的那樣。

It took six weeks for one engineer to get the reformatting of Google's 200,000 BUILD files accepted by the various code owners, during which more than a thousand new BUILD files were added each week. Google's nascent infrastructure for making large-scale changes greatly accelerated this effort. (See Chapter 22.)

一位工程師花了六週時間重新格式化了谷歌的 200,000 個 BUILD 檔案，這些檔案被各個程式碼所有者接受，在此期間每週都會新增一千多個新的 BUILD 檔案。谷歌為進行大規模變革而建立的基礎設施大大加快了這一努力。（見第 22 章）。

> 14 2018 年 12 月，按拉動請求衡量，Go 是 GitHub 上排名第四的語言。
>
> 15 Robert Griesemer 在 2015 年的演講《gofmt 的文化演變》中詳細介紹了 gofmt 的動機、設計。
> 以及 gofmt 對 Go 和其他語言的影響。
>
> 16 Russ Cox 在 2009 年解釋說，gofmt 是關於自動化修改的。"因此，我們有一個程式操作工具的所有硬性部分，只是坐在那裡等著被使用。同意接受'gofmt 風格'是使其在有限的程式碼量中可以做到的部分。"
>
> 17 Go AST 和格式包各自有成千上萬的匯入器。

Conclusion 結論

For any organization, but especially for an organization as large as Google's engineering force, rules help us to manage complexity and build a maintainable codebase. A shared set of rules frames the engineering processes so that they can scale up and keep growing, keeping both the codebase and the organization sustainable for the long term.

對於任何組織，尤其是像 Google 的工程師團隊這樣大的組織，規則幫助我們管理複雜性並建立一個可維護的程式碼庫。一組共享的規則框定了工程流程，以便它們可以擴大規模並保持增長，從而保持程式碼庫和組織的長期可持續性。

TL;DRs(Too long;Don't read) 內容提要

- Rules and guidance should aim to support resilience to time and scaling.
- Know the data so that rules can be adjusted.
- Not everything should be a rule.
- Consistency is key.
- Automate enforcement when possible.
- 規則和指導應旨在支援對時間和規模的擴充性。
- 瞭解資料，以便調整規則。

- 並非所有事情都應該成為規則。
- 一致性是關鍵。
- 在可能的情況下自動化執行。

---

1. Many of our style guides have external versions, which you can find at https://google.github.io/styleguide. We cite numerous examples from these guides within this chapter. ↩

2. Tooling matters here. The measure for “too many” is not the raw number of rules in play, but how many an engineer needs to remember. For example, in the bad-old-days pre-clang-format, we needed to remember a ton of formatting rules. Those rules haven’t gone away, but with our current tooling, the cost of adherence has fallen dramatically. We’ve reached a point at which somebody could add an arbitrary number of formatting rules and nobody would care, because the tool just does it for you. ↩

3. Credit to H. Wright for the real-world comparison, made at the point of having visited around 15 different Google offices. ↩

4. “Chunking” is a cognitive process that groups pieces of information together into meaningful “chunks” rather than keeping note of them individually. Expert chess players, for example, think about configurations of pieces rather than the positions of the individuals. ↩

5. See 4.2 Block indentation: +2 spaces, Spaces vs. Tabs, 4.4 Column limit:100 and Line Length ↩

6. Use of const, for example. ↩

7. Style formatting for BUILD files implemented with Starlark is applied by buildifier. See https://github.com/ bazelbuild/buildtools. ↩

8. See Exceptions to Naming Rules. As an example, our open sourced Abseil libraries use snake_case naming for types intended to be replacements for standard types. See the types defined in https://github.com/abseil/abseilcpp/blob/master/absl/utility/utility.h. These are C++11 implementation of C++14 standard types and therefore use the standard’s favored snake_case style instead of Google’s preferred CamelCase form. ↩

9. See `https://google.github.io/styleguide/cppguide.html#Comments`, `http://google.github.io/styleguide/pyguide#38-comments-and-docstrings`, and `https://google.github.io/styleguide/javaguide.html#s7-javadoc`, where multiple languages define general comment rules. ↩

10. In December 2018, Go was the #4 language on GitHub as measured by pull requests. ↩

11. Robert Griesemer’s 2015 talk, “The Cultural Evolution of gofmt,” provides details on the motivation, design,and impact of gofmt on Go and other languages. ↩

12. Russ Cox explained in 2009 that gofmt was about automated changes: “So we have all the hard parts of a program manipulation tool just sitting waiting to be used. Agreeing to accept ‘gofmt style’ is the piece that makes it doable in a finite amount of code.” ↩

13. The Go AST and format packages each have thousands of importers. ↩