CHAPTER 10

Documentation

第十章 文件

Written by Tom Manshreck

Edited by Riona MacNamara

Of the complaints most engineers have about writing, using, and maintaining code, a singular common frustration is the lack of quality documentation. “What are the side effects of this method?” “I got an error after step 3.” “What does this acronym mean?” “Is this document up to date?” Every software engineer has voiced complaints about the quality, quantity, or sheer lack of documentation throughout their career, and the software engineers at Google are no different.

在大多數工程師對編寫、使用和維護程式碼的抱怨中，一個常見的問題是缺乏高品質的文件。"這個方法的副作用是什麼？" "我在第三步之後出錯了。”“這個縮寫詞是什麼意思？”“這份文件是最新的嗎？”每個軟體工程師在他們的職業生涯中都對文件的品質、數量或完全缺失提出過抱怨，谷歌的軟體工程師也不例外。

Technical writers and project managers may help, but software engineers will always need to write most documentation themselves. Engineers, therefore, need the proper tools and incentives to do so effectively. The key to making it easier for them to write quality documentation is to introduce processes and tools that scale with the organization and that tie into their existing workflow.

技術撰稿人和專案經理可以提供幫助，但軟體工程師總是需要自己編寫大部分的文件。因此，工程師需要適當的工具和激勵來有效地做到這一點。讓他們更便捷地寫出高品質的文件的關鍵是引入隨組織擴充並與現有工作流程相結合的流程和工具。

Overall, the state of engineering documentation in the late 2010s is similar to the state of software testing in the late 1980s. Everyone recognizes that more effort needs to be made to improve it, but there is not yet organizational recognition of its critical benefits. That is changing, if slowly. At Google, our most successful efforts have been when documentation is treated like code and incorporated into the traditional engineering workflow, making it easier for engineers to write and maintain simple documents.

總體而言，2010 年代末的工程文件狀況與 1980 年代末的軟體測試狀態相似。每個人都認識到需要做出更多的努力來改善它，但還沒有組織上認識到它的關鍵好處。這種情況正在改變，儘管很緩慢。在谷歌，我們最成功的努力是將文件像程式碼一樣對待，並將其納入傳統的工程工作流程，使工程師更便捷地編寫和維護文件。

What Qualifies as Documentation? 什麼是合格的文件？

When we refer to “documentation,” we’re talking about every supplemental text that an engineer needs to write to do their job: not only standalone documents, but code comments as well. (In fact, most of the documentation an engineer at Google writes comes in the form of code comments.) We’ll discuss the various types of engineering documents further in this chapter.

當我們提到“文件”時，我們談論的是工程師為完成工作需要編寫的每一個補充文字：不僅是獨立文件，還有程式碼註釋。(事實上，谷歌的工程師所寫的大部分文件都是以程式碼註釋的形式出現的）。我們將在本章進一步討論各種型別的工程文件。

Why Is Documentation Needed? 為什麼需要文件？

Quality documentation has tremendous benefits for an engineering organization. Code and APIs become more comprehensible, reducing mistakes. Project teams are more focused when their design goals and team objectives are clearly stated. Manual processes are easier to follow when the steps are clearly outlined. Onboarding new members to a team or code base takes much less effort if the process is clearly documented.

高品質的文件對一個工程組織有巨大的好處。程式碼和 API 變得更容易理解，減少了錯誤。當專案團隊的設計目標和團隊目標明確時，他們會更加專注。當步驟被清楚地列出時，手動流程更容易被遵循。如果流程有明確的文件記錄，那麼將新成員加入團隊或程式碼庫所需的工作量要小得多。

But because documentation’s benefits are all necessarily downstream, they generally don’t reap immediate benefits to the author. Unlike testing, which (as we’ll see) quickly provides benefits to a programmer, documentation generally requires more effort up front and doesn’t provide clear benefits to an author until later. But, like investments in testing, the investment made in documentation will pay for itself over time. After all, you might write a document only once,[1] but it will be read hundreds, perhaps thousands of times afterward; its initial cost is amortized across all the future readers. Not only does documentation scale over time, but it is critical for the rest of the organization to scale as well. It helps answer questions like these:

- Why were these design decisions made?
- Why did we implement this code in this manner?
- Why did I implement this code in this manner, if you’re looking at your own code two years later?

但是，由於文件的好處都必然是延後的，它們通常不會給作者帶來直接的好處。與測試不同，測試（正如我們將看到的）很快就能給程式設計師帶來好處，而文件編寫通常需要更多的前期工作，直到後來才會給作者帶來明確的好處。但是，就像對測試的投入一樣，對文件的投入會隨著時間的推移而得到回報。畢竟，你可能只寫了一次文件，但之後會被閱讀數百次，甚至數千次；其最初的成本會在所有未來的讀者中攤銷。文件不僅可以隨著時間的推移而擴充，而且對於組織的其他部分也是至關重要的。它有助於回答這樣的問題：

- 為什麼做出這些設計決策？
- 為什麼我們要以這種方式實現這段程式碼？
- 如果你兩年後再看自己的程式碼，我為什麼要以這種方式實現這些程式碼？

If documentation conveys all these benefits, why is it generally considered “poor” by engineers? One reason, as we’ve mentioned, is that the benefits aren’t immediate, especially to the writer. But there are several other reasons:

- Engineers often view writing as a separate skill than that of programming. (We’ll try to illustrate that this isn’t quite the case, and even where it is, it isn’t necessarily a separate skill from that of software engineering.)
- Some engineers don’t feel like they are capable writers. But you don’t need a robust command of English [2] to produce workable documentation. You just need to step outside yourself a bit and see things from the audience’s perspective.
- Writing documentation is often more difficult because of limited tools support or integration into the developer workflow.
- Documentation is viewed as an extra burden—something else to maintain— rather than something that will make maintenance of their existing code easier.

如果文件能傳達這麼多的好處，為什麼工程師們普遍認為它 "很糟糕"？正如我們所提到的，其中一個原因是，這些好處並不直接，尤其是對作者而言。但還有其他幾個原因：

- 工程師們通常認為寫作是一種獨立於程式設計的技能。(我們將試圖說明，事實並非如此，即使是這樣，它也不一定是與軟體工程不同的技能。)
- 有些工程師覺得他們不是有寫作能力的人。但是，你不需要精通英語，就能寫出可行的文件。你只需要跳出自己視角，從聽眾的角度看問題。
- 由於有限的工具支援或整合到開發人員的工作流程中，編寫文件往往更加困難。
- 文件被看作是一個額外的負擔——需要維護的其他東西——而不是能使他們現有的程式碼維護更容易的東西。

> 1 好的，你將需要維護它並偶爾修改它。
>
> 2 英語仍然是大多數程式設計師的主要語言，大多數程式設計師的技術文件都依賴於對英語的理解。

Not every engineering team needs a technical writer (and even if that were the case, there aren’t enough of them). This means that engineers will, by and large, write most of the documentation themselves. So, instead of forcing engineers to become technical writers, we should instead think about how to make writing documentation easier for engineers. Deciding how much effort to devote to documentation is a decision your organization will need to make at some point.

不是每個工程團隊都需要技術撰稿人（即使是需要，也沒有足夠的技術撰稿人）。這意味著，工程師基本上會自己寫大部分的文件。因此，我們不應該強迫工程師成為技術撰稿人，而應該考慮如何讓工程師更容易編寫文件。決定在文件上投入多少精力是你的組織在某個時候需要做出的決定。

Documentation benefits several different groups. Even to the writer, documentation provides the following benefits:

- It helps formulate an API. Writing documentation is one of the surest ways to figure out if your API makes sense. Often, the writing of the documentation itself leads engineers to reevaluate design decisions that otherwise wouldn’t be questioned. If you can’t explain it and can’t define it, you probably haven’t designed it well enough.

- It provides a road map for maintenance and a historical record. Tricks in code should be avoided, in any case, but good comments help out a great deal when you're staring at code you wrote two years ago, trying to figure out what's wrong.
- It makes your code look more professional and drive traffic. Developers will naturally assume that a well-documented API is a better-designed API. That's not always the case, but they are often highly correlated. Although this benefit sounds cosmetic, it's not quite so: whether a product has good documentation is usually a pretty good indicator of how well a product will be maintained.
- It will prompt fewer questions from other users. This is probably the biggest benefit over time to someone writing the documentation. If you have to explain something to someone more than once, it usually makes sense to document that process.

文件對幾個不同的群體都有好處。即使對作者來說，文件也有以下好處：

- 它有助於制定 API。編寫文件是確定 API 是否合理的最可靠方法之一。通常情況下，文件的編寫本身會導致工程師重新評估設計決策，否則這些決策將會被質疑。如果你不能解釋它，也不能定義它，那麼你可能設計得不夠好。
- 它提供了維護路線圖和歷史記錄。無論如何，應該避免程式碼中的技巧，但是當你盯著兩年前編寫的程式碼，試圖找出錯誤的地方時，好的註釋會有很大幫助。
- 它使你的程式碼看起來更專業，並帶來流量。開發人員通常認為，一個有良好文件的 API 是一個設計更好的 API。情況並非總是如此，但它們往往是高度相關的。雖然這個好處聽起來很表象，但也不盡然：一個產品是否有良好的文件記錄通常是一個很好的指標，表明一個產品的維護情況如何。
- 它將減少其他使用者提出的問題。隨著時間的推移，這可能是編寫文件的人最大的收穫。如果你必須向別人解釋不止一次，通常記錄這個過程是有意義的。

As great as these benefits are to the writer of documentation, the lion's share of documentation's benefits will naturally accrue to the reader. Google's C++ Style Guide notes the maxim "optimize for the reader." This maxim applies not just to code, but to the comments around code, or the documentation set attached to an API. Much like testing, the effort you put into writing good documents will reap benefits many times over its lifetime. Documentation is critical over time, and reaps tremendous benefits for especially critical code as an organization scales.

儘管這些好處對文件的作者來說是巨大的，但文件的大部分好處自然會累積到讀者身上。谷歌的《C++風格指南》指出了 "為讀者最佳化 "的格言。這句格言不僅適用於程式碼，也適用於程式碼周圍的註釋，或者附加到 API 的文件集。和測試一樣，你為編寫好的文件所付出的努力將在其生命週期內多次獲得收益。隨著時間的推移，文件是非常重要的，隨著組織規模的擴大，對於特別重要的程式碼，文件將獲得巨大的好處。

Documentation Is Like Code 把文件當做程式碼

Software engineers who write in a single, primary programming language still often reach for different languages to solve specific problems. An engineer might write shell scripts or Python to run command-line tasks, or they might write most of their backend code in C++ but write some middleware code in Java, and so on. Each language is a tool in the toolbox.

用單一的、主要的程式語言編寫的軟體工程師仍然經常使用不同的語言來解決特定的問題。工程師可以編寫 shell 指令碼或 Python 來執行命令列任務，或者他們可以用 C++編寫他們的大部分後端程式碼，但是在 java 中編寫一些中介軟體程式碼，等等。每種語言都是工具箱中的一種工具。

Documentation should be no different: it’s a tool, written in a different language (usually English) to accomplish a particular task. Writing documentation is not much different than writing code. Like a programming language, it has rules, a particular syntax, and style decisions, often to accomplish a similar purpose as that within code: enforce consistency, improve clarity, and avoid (comprehension) errors. Within technical documentation, grammar is important not because one needs rules, but to standardize the voice and avoid confusing or distracting the reader. Google requires a certain comment style for many of its languages for this reason.

文件應該沒有什麼不同：它是一種工具，用不同的語言（通常是英語）編寫，用於完成特定任務。編寫文件與編寫程式碼沒有太大區別。與程式語言一樣，它有規則、特定語法和樣式規範，通常用於實現與程式碼中類似的目的：加強一致性、提高畫質晰度和避免（理解）錯誤。在技術文件中，語法很重要，不是因為需要規則，而是為了使聲音標準化，避免混淆或分散讀者的注意力。出於這個原因，谷歌對其許多語言都要求有一定的註釋風格。

Like code, documents should also have owners. Documents without owners become stale and difficult to maintain. Clear ownership also makes it easier to handle documentation through existing developer workflows: bug tracking systems, code review tooling, and so forth. Of course, documents with different owners can still conflict with one another. In those cases, it is important to designate canonical documentation: determine the primary source and consolidate other associated documents into that primary source (or deprecate the duplicates).

與程式碼一樣，文件也應該有所有者。沒有所有者的文件會變得陳舊且難以維護。明確的所有權還可以透過現有的開發人員工作流程（bug 追蹤系統、程式碼審查工具等）更輕鬆地處理文件。當然，不同所有者的文件仍然可能相互衝突。在這些情況下，指定規範文件非常重要：確定主要來源，並將其他相關文件合併到該主要來源中（或棄用副本）。

The prevalent usage of “go/links” at Google (see Chapter 3) makes this process easier. Documents with straightforward go/links often become the canonical source of truth. One other way to promote canonical documents is to associate them directly with the code they document by placing them directly under source control and alongside the source code itself.

在谷歌，"go/links "的普遍使用（見第三章）使這一過程更加容易。有直接的 "go/links "的檔案往往成為權威的標準來源。促進規範化文件的另一種方法是，透過將它們直接置於原始碼控制之下並與原始碼本身一起，將它們與它們所記錄的程式碼直接關聯。

Documentation is often so tightly coupled to code that it should, as much as possible, be treated as code. That is, your documentation should:

- Have internal policies or rules to be followed
- Be placed under source control
- Have clear ownership responsible for maintaining the docs
- Undergo reviews for changes (and change with the code it documents)
- Have issues tracked, as bugs are tracked in code
- Be periodically evaluated (tested, in some respect)
- If possible, be measured for aspects such as accuracy, freshness, etc. (tools have still not caught up here)

文件通常與程式碼緊密相連，所以應該儘可能地把它當作程式碼來對待。也就是說，你的文件應該：

- 有需要遵循的內部策略或規則
- 被置於原始碼控制之下
- 有明確的所有權，負責維護文件
- 對修改進行審查（並與它所記錄的程式碼一起改變）。
- 追蹤問題，就像追蹤程式碼中的 bug 一樣
- 定期評估（在某種程度上測試）。
- 如有可能，對準確度、新鮮度等方面進行衡量（這裡還沒有工具）

The more engineers treat documentation as "one of" the necessary tasks of software development, the less they will resent the upfront costs of writing, and the more they will reap the long-term benefits. In addition, making the task of documentation easier reduces those upfront costs.

工程師們越是把文件工作當作軟體開發的必要任務之一，他們就越是不反感寫文件的前期成本，也就越能獲得長期的收益。此外，讓文件工作變得更容易，可以減少這些前期成本。

Case Study: The Google Wiki 案例研究：谷歌維基

When Google was much smaller and leaner, it had few technical writers. The easiest way to share information was through our own internal wiki (GooWiki). At first, this seemed like a reasonable approach; all engineers shared a single documentation set and could update it as needed.

當谷歌規模更小、更精簡時，幾乎沒有技術作家。分享資訊的最簡單方法是透過我們自己的內部維基（GooWiki）。起初，這似乎是一個合理的方法；所有工程師共享一個文件集，可以根據需要進行更新。

But as Google scaled, problems with a wiki-style approach became apparent. Because there were no true owners for documents, many became obsolete.[3] Because no process was put in place for adding new documents, duplicate documents and document sets began appearing. GooWiki had a flat namespace, and people were not good at applying any hierarchy to the documentation sets. At one point, there were 7 to 10 documents (depending on how you counted them) on setting up Borg, our production compute environment, only a few of which seemed to be maintained, and most were specific to certain teams with certain permissions and assumptions.

但隨著谷歌規模的擴大，維基風格方法的問題變得明顯。因為沒有真正的文件所有者，許多文件變得過時了。因為沒有建立新增新文件的流程，重複的文件和文件集開始出現了。GooWiki 有一個扁平的名稱空間，人們不擅長將任何層次結構應用於文件集。在某些點上，有 7 到 10 個文件（取決於你如何計算）用於設定我們的生產計算環境 Borg，其中只有少數文件似乎得到了維護，大多數文件都是特定於具有特定許可權和設定的特指定團隊的。

Another problem with GooWiki became apparent over time: the people who could fix the documents were not the people who used them. New users discovering bad documents either couldn't confirm that the documents were wrong or didn't have an easy way to report errors. They knew something was wrong (because the document didn't work), but they couldn't "fix" it. Conversely, the people best able to fix the documents often didn't need to consult them after they were written. The documentation became so poor as Google grew that the quality of documentation became Google's number one developer complaint on our annual developer surveys.

隨著時間的推移，GooWiki 的另一個問題變得顯而易見：能夠修復文件的人不是使用它們的人。發現不良文件的新使用者要麼無法確認文件是否有誤，要麼無法便捷報告錯誤。他們知道出了問題（因為文件不起作用），但他們無法“修復”它。相反，最能修復文件的人通常不需要在編寫文件後查閱它們。隨著谷歌的發展，文件品質變得如此之差，以至於在我們的年度開發者調查中，文件品質成了谷歌對開發者的第一大抱怨。

The way to improve the situation was to move important documentation under the same sort of source control that was being used to track code changes. Documents began to have their own owners, canonical locations within the source tree, and processes for identifying bugs and fixing them; the documentation began to dramatically improve. Additionally, the way documentation was written and maintained began to look the same as how code was written and maintained. Errors in the documents could be reported within our bug tracking software. Changes to the documents could be handled using the existing code review process. Eventually, engineers began to fix the documents themselves or send changes to technical writers (who were often the owners).

改善這種情況的方法是將重要的文件轉移到與追蹤程式碼變化相同的原始碼控制之下。文件開始有自己的所有者，在原始碼樹中的規範位置，以及識別 bug 和修復 bug 的過程；文件品質開始顯著改善。此外，編寫和維護文件的方式開始與編寫和維護程式碼的方式相同。文件中的錯誤可以在我們的錯誤追蹤軟體中報告。對文件的修改可以透過現有的程式碼審查過程來處理。最終，工程師們開始自己修改文件，或將修改內容傳送給技術作家（他們往往是文件的所有者）。

Moving documentation to source control was initially met with a lot of controversy.
Many engineers were convinced that doing away with the GooWiki, that bastion of freedom of information, would lead to poor quality because the bar for documentation (requiring a review, requiring owners for documents, etc.) would be higher. But that wasn't the case. The documents became better.

將文件轉移到原始碼控制中，最初遇到了很多爭議。
許多工程師相信，取消 GooWiki 這個資訊自由的堡壘，會導致品質下降，因為對文件的要求（要求審查，要求文件的所有者等）會更高。但事實並非如此。文件變得更好了。

The introduction of Markdown as a common documentation formatting language also helped because it made it easier for engineers to understand how to edit documents without needing specialized expertise in HTML or CSS. Google eventually introduced its own framework for embedding documentation within code: g3doc.With that framework, documentation improved further, as documents existed side by side with the source code within the engineer's development environment. Now, engineers could update the code and its associated documentation in the same change (a practice for which we're still trying to improve adoption).

引入 Markdown 作為通用的文件格式化語言也有幫助，因為它使工程師更容易理解如何編輯文件，而不需要 HTML 或 CSS 方面的專業知識。谷歌最終引入了自己的框架，用於在程式碼中嵌入文件：g3doc.有了這個框架，文件得到了進一步的改善，因為在工程師的開發環境中，文件與原始碼並列存在。現在，工程師們可以在相同的變更中更新程式碼及其相關的文件（我們仍在努力改進這種做法）。

The key difference was that maintaining documentation became a similar experience to maintaining code: engineers filed bugs, made changes to documents in changelists, sent changes to reviews by experts, and so on. Leveraging of existing developer workflows, rather than creating new ones, was a key benefit.

關鍵的區別在於，維護文件變成了與維護程式碼類似的流程：工程師們提交錯誤，在變更列表中對文件進行修改，將修改傳送給專家審查，等等。利用現有的開發者工作流程，而不是建立新的工作流程，是一個關鍵的好處。

> 3 當我們棄用 GooWiki 時，我們發現大約 90%的文件在前幾個月沒有檢視或更新。

Know Your Audience 瞭解你的受眾

One of the most important mistakes that engineers make when writing documentation is to write only for themselves. It's natural to do so, and writing for yourself is not without value: after all, you might need to look at this code in a few years and try to figure out what you once meant. You also might be of approximately the same skill set as someone reading your document. But if you write only for yourself, you are going to make certain assumptions, and given that your document might be read by a very wide audience (all of engineering, external developers), even a few lost readers is a large cost. As an organization grows, mistakes in documentation become more prominent, and your assumptions often do not apply.

工程師在寫文件時犯的最主要的錯誤之一是隻為自己寫。這樣做是很自然的，而且為自己寫也不是沒有價值：畢竟，你可能需要在幾年後看一下這段程式碼，並試圖弄清楚你曾經的設計。你也可能與閱讀你的文件的人具有大致相同的技能。但是，如果你只為自己寫，你就會做出某些假設，考慮到你的文件可能會被非常廣泛的讀者閱讀（所有的工程人員、外部開發人員），即使失去幾個讀者也是一個很大的代價。隨著組織的發展，文件中的錯誤變得更加突出，你的假設往往不適用。

Instead, before you begin writing, you should (formally or informally) identify the audience(s) your documents need to satisfy. A design document might need to persuade decision makers. A tutorial might need to provide very explicit instructions to someone utterly unfamiliar with your codebase. An API might need to provide complete and accurate reference information for any users of that API, be they experts or novices. Always try to identify a primary audience and write to that audience.

相反，在你開始寫作之前，你應該（正式地或非正式地）確定你的檔案需要滿足的受眾。設計文件可能需要說服決策者。課程可能需要為完全不熟悉你的程式碼庫的人提供非常明確的說明。API 可能需要為該 API 的任何使用者（無論是專家還是新手）提供完整準確的參考資訊。始終嘗試確定主要受眾併為該受眾寫作。

Good documentation need not be polished or "perfect." One mistake engineers make when writing documentation is assuming they need to be much better writers. By that measure, few software engineers would write. Think about writing like you do about testing or any other process you need to do as an engineer. Write to your audience, in the voice and style that they expect. If you can read, you can write. Remember that your audience is standing where you once stood, but without your new domain knowledge. So you don't need to be a great writer; you just need to get someone like you as familiar with the domain as you now are. (And as long as you get a stake in the ground, you can improve this document over time.)

好的文件不需要潤色或 "完美"。工程師在寫文件時犯的一個錯誤是假設他們需要成為更好的作家。按照這個標準，很少有軟體工程師會寫。思考寫作，就像你做測試一樣，或者你作為一名工程師需要做的任何其他過程。以聽眾期望的聲音和風格向他們寫信。如果你能讀，你就能寫。記住，你的受眾就站在你曾經站過的地方，但沒有你的新領域知識。所以你不需要成為一個偉大的作家；你只需要找一個像你一樣熟悉這個領域的人。（而且，只要你能從中獲益，你就可以隨著時間的推移改進這份檔案。）

Types of Audiences 受眾型別

We’ve pointed out that you should write at the skill level and domain knowledge appropriate for your audience. But who precisely is your audience? Chances are, you have multiple audiences based on one or more of the following criteria:

- Experience level (expert programmers, or junior engineers who might not even be familiar—gulp!—with the language).
- Domain knowledge (team members, or other engineers in your organization who are familiar only with API endpoints).
- Purpose (end users who might need your API to do a specific task and need to find that information quickly, or software gurus who are responsible for the guts of a particularly hairy implementation that you hope no one else needs to maintain).

我們已經指出，你應該按照適合你的受眾的技能水平和領域知識來寫作。但究竟誰是你的受眾？根據以下一個或多個標準，你可能擁有多個受眾：

- 經驗水平（專家級程式設計師，或者甚至可能不熟悉語言的初級工程師）。
- 領域知識（團隊成員或組織中只熟悉 API 端點的其他工程師）。
- 目的（可能需要你的 API 來完成特定任務並需要快速找到該資訊的終端使用者，或負責你希望沒有其他人需要維護的特別複雜的實現的核心的軟體專家）。

In some cases, different audiences require different writing styles, but in most cases, the trick is to write in a way that applies as broadly to your different audience groups as possible. Often, you will need to explain a complex topic to both an expert and a novice. Writing for the expert with domain knowledge may allow you to cut corners, but you’ll confuse the novice; conversely, explaining everything in detail to the novice will doubtless annoy the expert.

在某些情況下，不同的受眾需要不同的寫作風格，但在大多數情況下，技巧是以一種儘可能廣泛地適用於不同受眾群體的方式進行寫作。通常，你需要同時向專家和新手解釋一個複雜的主題。為有領域知識的專家寫作方式可能會讓你少走彎路，但你會讓新手感到困惑；反之，向新手詳細解釋一切無疑會讓專家感到厭煩。

Obviously, writing such documents is a balancing act and there’s no silver bullet, but one thing we’ve found is that it helps to keep your documents short. Write descriptively enough to explain complex topics to people unfamiliar with the topic, but don’t lose or annoy experts. Writing a short document often requires you to write a longer one (getting all the information down) and then doing an edit pass, removing duplicate information where you can. This might sound tedious, but keep in mind that this expense is spread across all the readers of the documentation. As Blaise Pascal once said, “If I had more time, I would have written you a shorter letter.” By keeping a document short and clear, you will ensure that it will satisfy both an expert and a novice.

顯然，編寫這樣的文件是一種平衡行為，沒有什麼靈丹妙藥，但我們發現，它有助於保持文件的簡短。寫下足夠的描述，向不熟悉該主題的人解釋複雜的主題，但不要失去或惹惱專家。編寫一個簡短的文件通常需要你編寫一個較長的文件（將所有資訊記錄下來），然後進行編輯，儘可能刪除重複的資訊。這聽起來可能很乏味，但請記住，這項費用會分攤到文件的所有讀者身上。正如布萊斯·帕斯卡（Blaise Pascal）曾經說過的那樣，“如果我有更多的時間，我會給你寫一封更短的信。”透過保持文件的簡短和清晰，你將確保它能讓專家和新手都滿意。

Another important audience distinction is based on how a user encounters a document:

- Seekers are engineers who know what they want and want to know if what they are looking at fits the bill. A key pedagogical device for this audience is consistency. If you are writing reference documentation for this group—within a code file, for example—you will want to have your comments follow a similar format so that readers can quickly scan a reference and see whether they find what they are looking for.
- Stumblers might not know exactly what they want. They might have only a vague idea of how to implement what they are working with. The key for this audience is clarity. Provide overviews or introductions (at the top of a file, for example) that explain the purpose of the code they are looking at. It's also useful to identify when a doc is not appropriate for an audience. A lot of documents at Google begin with a "TL;DR statement" such as "TL;DR: if you are not interested in C++ compilers at Google, you can stop reading now."

另一個重要的受眾區分是基於使用者如何使用文件：

- 尋求者，工程師知道他們想要什麼，並且想知道他們所看到的是否符合要求。對於這些聽眾來說，一個關鍵的教學手段是一致性。如果你為這一群體寫參考文件——在一個程式碼檔案內，例如——你希望註釋遵循類似的格式，以便受眾可以快速掃描參考並檢視是否找到所需內容。
- 瀏覽者，可能不知道他們到底想要什麼。他們可能對如何實施他們正在使用的東西只有一個模糊的概念。這類讀者的關鍵是清晰。提供概述或介紹（例如，在檔案的頂部），解釋他們正在檢視的程式碼的用途。確定文件何時不適合受眾也很有用。谷歌的很多檔案都以 "TL;DR 宣告 "開始，如 "TL;DR：如果你對谷歌的 C++編譯器不感興趣，你現在可以停止閱讀。"

Finally, one important audience distinction is between that of a customer (e.g., a user of an API) and that of a provider (e.g., a member of the project team). As much as possible, documents intended for one should be kept apart from documents intended for the other. Implementation details are important to a team member for maintenance purposes; end users should not need to read such information. Often, engineers denote design decisions within the reference API of a library they publish. Such reasonings belong more appropriately in specific documents (design documents) or, at best, within the implementation details of code hidden behind an interface.

最後，一個重要的受眾區分是客戶（例如，API 的使用者）和供應方（例如，專案組的成員）。為一方準備的檔案應儘可能與為另一方準備的檔案分開儲存。實施細節對於團隊成員的維護非常重要；終端使用者不需要閱讀此類資訊。通常，工程師在他們釋出的函式庫的參考 API 中表示設計決策。這種推理更適合於特定文件（設計文件）中，或者充其量是隱藏在介面後面的程式碼的實現細節中。

Documentation Types 文件型別

Engineers write various different types of documentation as part of their work: design documents, code comments, how-to documents, project pages, and more. These all count as "documentation." But it is important to know the different types, and to not mix types. A document should have, in general, a singular purpose, and stick to it. Just as an API should do one thing and do it well, avoid trying to do several things within one document. Instead, break out those pieces more logically.

工程師編寫各種不同型別的文件作為他們工作的一部分：設計文件、程式碼註釋、操作文件、專案頁面等等。這些都算作 "文件"。但重要的是，要了解不同的型別，不要混合型別。一般來說，一個文件應該有一個單一的用途，並堅持這個職責。就像一個 API 應該做一件事並且做得很好一樣，避免試圖在一個文件中做幾件事。相反，更邏輯合理地分解這些部分。

There are several main types of documents that software engineers often need to write:

- Reference documentation, including code comments
- Design documents
- Tutorials
- Conceptual documentation
- Landing pages

軟體工程師經常需要寫的文件主要有幾種型別：

- 參考文件，包括程式碼註釋
- 設計文件
- 課程
- 概念文件
- 著陸頁

It was common in the early days of Google for teams to have monolithic wiki pages with bunches of links (many broken or obsolete), some conceptual information about how the system worked, an API reference, and so on, all sprinkled together. Such documents fail because they don't serve a single purpose (and they also get so long that no one will read them; some notorious wiki pages scrolled through several dozens of screens). Instead, make sure your document has a singular purpose, and if adding something to that page doesn't make sense, you probably want to find, or even create, another document for that purpose.

在谷歌的早期，團隊擁有單頁的維基頁面是很常見的，其中有成堆的連結（許多連結已死鏈或過時），一些關於系統如何工作的概念資訊，一個 API 參考等等，這些都散落在一起。這些文件之所以失敗，是因為它們沒有一個單一的職責（而且它們也會變得如此之長，以至於沒有人會閱讀它們；一些臭名昭著的 wiki 頁面滾動了幾十個螢幕）。相反，要確保你的文件有一個單一的職責，如果向該頁面新增內容沒有意義，可能希望找到或甚至建立另一個用於該用途的文件。

Reference Documentation 參考文件

Reference documentation is the most common type that engineers need to write; indeed, they often need to write some form of reference documents every day. By reference documentation, we mean anything that documents the usage of code within the codebase. Code comments are the most common form of reference documentation that an engineer must maintain. Such comments can be divided into two basic camps: API comments versus implementation comments. Remember the audience differences between these two: API comments don't need to discuss implementation details or design decisions and can't assume a user is as versed in the API as the author. Implementation comments, on the other hand, can assume a lot more domain knowledge of the reader, though be careful in assuming too much: people leave projects, and sometimes it's safer to be methodical about exactly why you wrote this code the way you did.

參考文件是工程師最常需要寫的型別；事實上，他們經常每天都需要寫某種形式的參考文件。所謂參考文件，我們指的是記錄程式碼庫中的程式碼使用情況的任何東西。程式碼註釋是工程師必須維護的最常見的參考文件形式。這種註釋可以分為兩個基本陣營。API 註釋和實現註釋。記住這兩者之間的受眾差異。API 註釋不需要討論實現細節或設計決策，也不能假設使用者像作者一樣精通 API。另一方面，實現註釋可以假定讀者有更多的領域知識，但要小心假設太多：人們離開了專案，有時更安全的做法是

有條不紊地說明你為什麼這樣寫程式碼。

Most reference documentation, even when provided as separate documentation from the code, is generated from comments within the codebase itself. (As it should; reference documentation should be single-sourced as much as possible.) Some languages such as Java or Python have specific commenting frameworks (Javadoc, PyDoc, GoDoc) meant to make generation of this reference documentation easier. Other languages, such as C++, have no standard “reference documentation” implementation, but because C++ separates out its API surface (in header or .h files) from the implementation (.cc files), header files are often a natural place to document a C++ API.

大多數參考文件，即使是作為獨立於程式碼的文件提供，也是由程式碼庫本身的註釋產生的。(這是應該的；參考文件應該儘可能的單一來源。) 一些語言，如 Java 或 Python 有特定的註釋框架（Javadoc、PyDoc、GoDoc）旨在簡化參考文件的產生。其他語言，如 C++，沒有標準的 "參考文件 "實現，但由於 C++將其 API 表面（標頭檔案或.h 檔案）與實現（.cc 檔案）分開，標頭檔案通常是記錄 C++ API 的自然場所。

Google takes this approach: a C++ API deserves to have its reference documentation live within the header file. Other reference documentation is embedded directly in the Java, Python, and Go source code as well. Because Google’s Code Search browser (see Chapter 17) is so robust, we’ve found little benefit to providing separate generated reference documentation. Users in Code Search not only search code easily, they can usually find the original definition of that code as the top result. Having the documentation alongside the code’s definitions also makes the documentation easier to discover and maintain.

谷歌採取了這種方法：一個 C++ API 應該有它的參考檔案存在標頭檔案中。其他參考文件也直接嵌入到 Java、Python 和 Go 原始碼中。因為 Google 的 Code Search 瀏覽器（見第 17 章）非常強大，我們發現提供單獨的通用參考文件沒有什麼好處。使用者在程式碼搜尋中不僅可以很容易地搜尋到程式碼，而且通常可以找到該程式碼的原始定義作為最重要的結果。將文件與程式碼的定義放在一起，也使文件更容易被發現和維護。

We all know that code comments are essential to a well-documented API. But what precisely is a “good” comment? Earlier in this chapter, we identified two major audiences for reference documentation: seekers and stumblers. Seekers know what they want; stumblers don’t. The key win for seekers is a consistently commented codebase so that they can quickly scan an API and find what they are looking for. The key win for stumblers is clearly identifying the purpose of an API, often at the top of a file header. We’ll walk through some code comments in the subsections that follow. The code commenting guidelines that follow apply to C++, but similar rules are in place at Google for other languages.

我們都知道，程式碼註釋對於一個良好的文件化的 API 來說是必不可少的。但是什麼才是 "好的 "註釋呢？在本章的前面，我們確定了參考文件的兩個主要受眾：尋求者和瀏覽者。尋求者知道他們想要什麼，而瀏覽者不知道。尋求者的關鍵點是一個一致的註釋程式碼庫，這樣他們就可以快速掃描 API 並找到他們正在尋找的東西。對於瀏覽者來說，關鍵的勝利是明確識別 API 的用途，通常是在檔案頭的頂部。我們將在下面的小節中介紹一些程式碼註釋。下面的程式碼註釋指南適用於 C++，但在谷歌，其他語言也有類似的規則。

File comments 檔案註釋
Almost all code files at Google must contain a file comment. (Some header files that contain only one utility function, etc., might deviate from this standard.) File comments should begin with a header of the following form:

在谷歌，幾乎所有的程式碼檔案都必須包含一個檔案註釋。(一些只包含一個實用函式的標頭檔案等，可能會偏離這個標準)。檔案註釋應該以下列形式的標頭檔案開始：

```
// -----------------------------------------------------------------------------
// str_cat.h
// -----------------------------------------------------------------------------
//
// This header file contains functions for efficiently concatenating and appending
// strings: StrCat() and StrAppend(). Most of the work within these routines is
// actually handled through use of a special AlphaNum type, which was designed
// to be used as a parameter type that efficiently manages conversion to
// strings and avoids copies in the above operations.
... ...
```

Generally, a file comment should begin with an outline of what's contained in the code you are reading. It should identify the code's main use cases and intended audience (in the preceding case, developers who want to concatenate strings). Any API that cannot be succinctly described in the first paragraph or two is usually the sign of an API that is not well thought out. Consider breaking the API into separate components in those cases.

通常，檔案註釋應該以你正在閱讀的程式碼中所包含的內容的概要開始。它應該確定程式碼的主要使用案例和目標受眾（在前面的例子中，是想要連線字串的開發者）。在第一段或第二段中無法簡潔描述的任何 API 通常都是未經過深思熟慮的 API 的標誌。在這種情況下，可以考慮將 API 分成獨立的元件。

Class comments 類註釋

Most modern programming languages are object oriented. Class comments are therefore important for defining the API "objects" in use in a codebase. All public classes (and structs) at Google must contain a class comment describing the class/struct, important methods of that class, and the purpose of the class. Generally, class comments should be "nouned" with documentation emphasizing their object aspect. That is, say, "The Foo class contains x, y, z, allows you to do Bar, and has the following Baz aspects," and so on.

大多數現代程式語言都是面向物件的。因此，類註釋對於定義程式碼庫中使用的 API "物件 "非常重要。谷歌的所有公共類（和結構）必須包含一個類別註釋，描述該類/結構、該類別的重要方法以及該類別的目的。一般來說，類別的註釋應該是 "名詞化 "的，檔案強調其物件方面。也就是說，"Foo 類包含 x、y、z，允許你做 Bar，並且有以下 Baz 方面的內容"，等等。

Class comments should generally begin with a comment of the following form:

類別的註釋一般應該以下列形式的註釋開始：

```
// -----------------------------------------------------------------------------
// AlphaNum
// -----------------------------------------------------------------------------
//
// The AlphaNum class acts as the main parameter type for StrCat() and
// StrAppend(), providing efficient conversion of numeric, boolean, and
// hexadecimal values (through the Hex type) into strings.
```

Function comments 函式註釋

All free functions, or public methods of a class, at Google must also contain a function comment describing what the function *does*. Function comments should stress the *active* nature of their use, beginning with an indicative verb describing what the function does and what is returned.

在谷歌的所有公開函式或類別的公共方法也必須包含一個函式註釋，說明函式的功能。函式註釋應該強調其使用的主動性，以一個指示性動詞開始，描述函式的作用和返回的內容。

Function comments should generally begin with a comment of the following form:

函式註釋一般應以下列形式的註釋開始：

```
// StrCat()
//
// Merges the given strings or numbers, using no delimiter(s),
// returning the merged result as a string.
... ...
```

Note that starting a function comment with a declarative verb introduces consistency across a header file. A seeker can quickly scan an API and read just the verb to get an idea of whether the function is appropriate: “Merges, Deletes, Creates,” and so on.

請注意，用一個宣告性的動詞來開始一個函式註釋，可以在標頭檔案中引入一致性。尋求者可以快速掃描一個 API，唯讀動詞就可以知道這個函式是否合適。"合併、刪除、建立"等等。

Some documentation styles (and some documentation generators) require various forms of boilerplate on function comments, like “Returns:”, “Throws:”, and so forth, but at Google we haven’t found them to be necessary. It is often clearer to present such information in a single prose comment that’s not broken up into artificial section boundaries:

一些文件樣式（和一些文件產生器）要求在函式註釋中加入各種形式的範本，如 "Returns:"，"Throws:"等等，但在谷歌，我們發現它們並不是必須的。在一個鬆散的註釋中呈現這樣的資訊通常更清晰，而不是將其分解為人為的段落邊界：

```
// Creates a new record for a customer with the given name and address,
// and returns the record ID, or throws `DuplicateEntryError` if a
// record with that name already exists.
int AddCustomer(string name, string address);
```

Notice how the postcondition, parameters, return value, and exceptional cases are naturally documented together (in this case, in a single sentence), because they are not independent of one another. Adding explicit boilerplate sections would make the comment more verbose and repetitive, but no clearer (and arguably less clear).

請注意後置條件、引數、返回值和例外情況是如何自然地記錄在一起的（在本例中，在一句話中），因為它們不是相互獨立的。新增明確的樣板部分會使註釋更加冗長和重複，但不會更清晰（也可能不那麼清晰）。

Design Docs 設計文件

Most teams at Google require an approved design document before starting work on any major project. A software engineer typically writes the proposed design document using a specific design doc template approved by the team. Such documents are designed to be collaborative, so they are often shared in Google Docs, which has good collaboration tools. Some teams require such design documents to be discussed and debated at specific team meetings, where the finer points of the design can be discussed or critiqued by experts. In some respects, these design discussions act as a form of code review before any code is written.

谷歌的大多數團隊在開始任何重大專案之前都需要獲得批准的設計文件。軟體工程師通常使用團隊批准的特定設計文件範本編寫擬定設計檔案。這些文件是為了協作而設計的，所以它們通常在谷歌文件中共享，谷歌文件有很好的協作工具。一些團隊要求在特定的團隊會議上討論和辯論此類設計檔案，專家可以討論或評論設計的細節。在某些方面，這些設計討論就像是在編寫任何程式碼之前的一種程式碼審查形式。

Because the development of a design document is one of the first processes an engineer undertakes before deploying a new system, it is also a convenient place to ensure that various concerns are covered. The canonical design document templates at Google require engineers to consider aspects of their design such as security implications, internationalization, storage requirements and privacy concerns, and so on. In most cases, such parts of those design documents are reviewed by experts in those domains.

由於設計文件的開發是工程師在部署新系統之前首先進行的過程之一，因此也是確保涵蓋了各種關切。谷歌的典型設計文件範本要求工程師考慮其設計的各個方面，如安全影響、國際化、儲存要求和隱私問題等等。在大多數情況下，這些設計文件的這類部分都是由這些領域的專家來審查的。

A good design document should cover the goals of the design, its implementation strategy, and propose key design decisions with an emphasis on their individual trade-offs. The best design documents suggest design goals and cover alternative designs, denoting their strong and weak points.

一個好的設計文件應該包括設計目標、實施策略，並提出關鍵的設計決策，重點放在它們各自的權衡上。最好的設計文件建議設計目標，涵蓋替代設計，指出其優缺點。

A good design document, once approved, also acts not only as a historical record, but as a measure of whether the project successfully achieved its goals. Most teams archive their design documents in an appropriate location within their team documents so that they can review them at a later time. It’s often useful to review a design document before a product is launched to ensure that the stated goals when the design document was written remain the stated goals at launch (and if they do not, either the document or the product can be adjusted accordingly).

一份好的設計文件一旦獲得批准，不僅可以作為歷史記錄，還可以作為衡量專案是否成功實現其目標的指標。大多數團隊將其設計文件歸檔在團隊文件中的適當位置，以便日後進行審查。在產品釋出之前審查設計文件通常很有用，以確保在編寫設計文件時所述的目標保持在釋出時所述的目標（如果沒有，則可以相應地調整文件或產品）。

Tutorials 課程

Every software engineer, when they join a new team, will want to get up to speed as quickly as possible. Having a tutorial that walks someone through the setup of a new project is invaluable; “Hello World” has established itself is one of the best ways to ensure that all team members start off on the right foot. This goes for documents as well as code. Most projects deserve a “Hello World” document that assumes nothing and gets the engineer to make something “real” happen.

每個軟體工程師，當他們加入一個新的團隊時，都希望能儘快進入狀態。有一個指導別人完成新專案設定的課程是非常有價值的；"Hello World "是確保所有團隊成員從正確的角度出發的最佳方式之一。這適用於檔案和程式碼。大多數專案都應該有一個 "Hello World “文件，該文件不做任何假設，並讓工程師去做一些 "真實 "的事情。

Often, the best time to write a tutorial, if one does not yet exist, is when you first join a team. (It’s also the best time to find bugs in any existing tutorial you are following.) Get a notepad or other way to take notes, and write down everything you need to do along the way, assuming no domain knowledge or special setup constraints; after you’re done, you’ll likely know what mistakes you made during the process—and why —and can then edit down your steps to get a more streamlined tutorial. Importantly, write everything you need to do along the way; try not to assume any particular setup, permissions, or domain knowledge. If you do need to assume some other setup, state that clearly in the beginning of the tutorial as a set of prerequisites.

通常，如果還沒有課程，編寫課程的最佳時間是你第一次加入團隊時。（這也是在你正在學習的任何現有課程中查詢 bug 的最佳時機。）使用記事本或其他方式記筆記，並在沒有領域知識或特殊設定限制的情況下，寫下你需要做的所有事情；完成後，你可能會知道在這個過程中犯了哪些錯誤——原因——然後可以編輯你的步驟，以獲得更精簡的課程。重要的是，寫下你需要做的一切；儘量不要假設任何特定的設定、許可權或領域知識。如果你確實需要採用其他設定，請在本課程的開頭明確說明這是一組先決條件。

Most tutorials require you to perform a number of steps, in order. In those cases, number those steps explicitly. If the focus of the tutorial is on the user (say, for external developer documentation), then number each action that a user needs to undertake. Don’t number actions that the system may take in response to such user actions. It is critical and important to number explicitly every step when doing this. Nothing is more annoying than an error on step 4 because you forget to tell someone to properly authorize their username, for example.

大多數課程要求你按順序執行許多步驟。在這些情況下，請明確為這些步驟編號。如果本課程的重點是使用者（例如，對於外部開發人員文件），則對使用者需要執行的每個操作進行編號。不要對系統響應此類使用者操作可能採取的操作進行編號。在執行此操作時，對每個步驟進行明確編號是至關重要的。沒有什麼比步驟 4 中的錯誤更令人惱火的了，例如，你忘記告訴某人對其使用者名稱進行授權。

Example: A bad tutorial

1. Download the package from our server at http://example.com
2. Copy the shell script to your home directory
3. Execute the shell script
4. The foobar system will communicate with the authentication system
5. Once authenticated, foobar will bootstrap a new database named “baz”
6. Test “baz” by executing a SQL command on the command line
7. Type: CREATE DATABASE my_foobar_db;

範例：糟糕的課程

1. 從我們的伺服器下載軟體套件，網址為http://example.com
2. 將 shell 指令碼複製到主目錄
3. 執行 shell 指令碼
4. foobar 系統將與認證系統通訊
5. 經過身份驗證後，foobar 將引導一個名為“baz”的新資料庫
6. 透過在命令列上執行 SQL 命令來測試“baz”
7. 型別：建立資料庫 my_foobar_db；

In the preceding procedure, steps 4 and 5 happen on the server end. It’s unclear whether the user needs to do anything, but they don’t, so those side effects can be mentioned as part of step 3. As well, it’s unclear whether step 6 and step 7 are different. (They aren’t.) Combine all atomic user operations into single steps so that the user knows they need to do something at each step in the process. Also, if your tutorial has user-visible input or output, denote that on separate lines (often using the convention of a monospaced bold font).

在前面的程式中，步驟 4 和 5 發生在伺服器端。不清楚使用者是否需要做什麼，但他們不需要，所以這些副作用可以作為步驟 3 的一部分提及。同樣，也不清楚步驟 6 和步驟 7 是否不同。(它們不是。)將所有原子使用者操作組合到單個步驟中，以便使用者知道他們需要在流程的每個步驟中做一些事情。另外，如果你的課程有使用者可見的輸入或輸出，請用單獨的行來表示（通常使用單間距粗體字型）。

Example: A bad tutorial made better

1. Download the package from our server at http://example.com:

```
curl -I http://example.com
```

2. Copy the shell script to your home directory:

```
cp foobar.sh ~
```

3. Execute the shell script in your home directory:

```
cd ~; foobar.sh
```

The foobar system will first communicate with the authentication system. Once authenticated, foobar will bootstrap a new database named “baz” and open an input shell. 4. Test “baz” by executing a SQL command on the command line:

```
baz:$ CREATE DATABASE my_foobar_db;
```

例子：一個不好的課程會變得更好

1. 從我們的伺服器下載軟體套件，網址為http://example.com:

```
$curl -I http://example.com
```

2. 將 shell 指令碼複製到主目錄：

```
$cp foobar.sh ~
```

3. 在主目錄中執行 shell 指令碼：

```
$cd ~; foobar.sh
```

foobar 系統將首先與身份驗證系統通訊。經過身份驗證後，foobar 將引導一個名為“baz”的新資料庫並開啟一個輸入 shell。

4. 透過在命令列上執行 SQL 命令來測試“baz”：

```
baz:$CREATE DATABASE my_foobar_db;
```

Note how each step requires specific user intervention. If, instead, the tutorial had a focus on some other aspect (e.g., a document about the “life of a server”), number those steps from the perspective of that focus (what the server does).

注意每個步驟都需要指定的使用者操作。相反，如果本課程側重於其他方面（例如，關於“伺服器生命週期”的文件），請從該重點的角度對這些步驟進行編號（伺服器的功能）。

Conceptual Documentation 概念文件

Some code requires deeper explanations or insights than can be obtained simply by reading the reference documentation. In those cases, we need conceptual documentation to provide overviews of the APIs or systems. Some examples of conceptual documentation might be a library overview for a popular API, a document describing the life cycle of data within a server, and so on. In almost all cases, a conceptual document is meant to augment, not replace, a reference documentation set. Often this leads to duplication of some information, but with a purpose: to promote clarity. In those cases, it is not necessary for a conceptual document to cover all edge cases (though a reference should cover those cases religiously). In this case, sacrificing some accuracy is acceptable for clarity. The main point of a conceptual document is to impart understanding.

有些程式碼需要比閱讀參考文件更深入的解釋或見解。在這些情況下，我們需要概念文件來提供 API 或系統的概述。概念文件的一些範例可能是流行 API 的函式庫概述、描述伺服器內資料生命週期的文件等。在幾乎所有情況下，概念文件都是為了補充而不是取代參考文件集。這通常會導致某些資訊的重複，但目的是：提高畫質晰度。在這些情況下，概念文件不必涵蓋所有邊緣情況（儘管參考文件應嚴格涵蓋這些情況）。在這種情況下，為了清晰起見，犧牲一些準確性是可以接受的。概念檔案的要點是傳達瞭解。

“Concept” documents are the most difficult forms of documentation to write. As a result, they are often the most neglected type of document within a software engineer’s toolbox. One problem engineers face when writing conceptual documentation is that it often cannot be embedded directly within the source code because there isn’t a canonical location to place it. Some APIs have a relatively broad API surface area, in which case, a file comment might be an appropriate place for a “conceptual” explanation of the API. But often, an API works in conjunction with other APIs and/or modules. The only logical place to document such complex behavior is through a separate conceptual document. If comments are the unit tests of documentation, conceptual documents are the integration tests.

“概念”文件是最難編寫的文件形式。因此，它們通常是軟體工程師工具箱中最被忽視的文件型別。工程師在編寫概念文件時面臨的一個問題是，它通常無法直接嵌入到原始碼中，因為沒有一個規範的位置來放置它。一些 API 具有相對廣泛的 API 表面積，在這種情況下，檔案註釋可能是對 API 進行“概念性”解釋的合適位置。但是，API 通常與其他 API 和/或模組一起工作。記錄這種複雜行為的唯一合理之處是透過一個單獨的概念文件。如果註釋是文件的單元測試，那麼概念文件就是整合測試。

Even when an API is appropriately scoped, it often makes sense to provide a separate conceptual document. For example, Abseil’s StrFormat library covers a variety of concepts that accomplished users of the API should understand. In those cases, both internally and externally, we provide a format concepts document.

即使 API 的範圍適當，提供一個單獨的概念文件通常也是有意義的。例如，Abseil 的 StrFormat 庫涵蓋了 API 的熟練使用者應該理解的各種概念。在這些情況下，無論是內部還是外部，我們都提供了一個格式概念文件。

A concept document needs to be useful to a broad audience: both experts and novices alike. Moreover, it needs to emphasize clarity, so it often needs to sacrifice completeness (something best reserved for a reference) and (sometimes) strict accuracy. That’s not to say a conceptual document should intentionally be inaccurate; it just means that it should focus on common usage and leave rare usages or side effects for reference documentation.

概念文件需要對廣大受眾有用：包括專家和新手。此外，它還需要強調清晰性，因此通常需要犧牲完整性（最好留作參考）和（有時）嚴格的準確性。這並不是說概念性文件應該故意不準確；這只是意味著它應該關注常見用法，並將罕見用法或副作用留給參考文件。

Landing Pages 著陸頁

Most engineers are members of a team, and most teams have a “team page” somewhere on their company’s intranet. Often, these sites are a bit of a mess: a typical landing page might contain some interesting links, sometimes several documents titled “read this first!”, and some information both for the team and for its customers. Such documents start out useful but rapidly turn into disasters; because they become so cumbersome to maintain, they will eventually get so obsolete that they will be fixed by only the brave or the desperate.

大多數工程師都是一個團隊的成員，而大多數團隊在其公司內部網的某個地方都有一個 "團隊頁面"。通常情況下，這些網站有點混亂：一個典型的著陸頁面可能包含一些有趣的連結，有時是幾個標題為 "先閱讀此文！"的檔案，以及一些既為團隊又為客戶的資訊。這些的文件一開始很有用，但很快就變成了災難；因為它們的維護變得非常麻煩，最終會變得非常陳舊，只有勇敢的人或絕望的人才會去修復它們。

Luckily, such documents look intimidating, but are actually straightforward to fix: ensure that a landing page clearly identifies its purpose, and then include only links to other pages for more information. If something on a landing page is doing more than being a traffic cop, it is not doing its job. If you have a separate setup document, link to that from the landing page as a separate document. If you have too many links on the landing page (your page should not scroll multiple screens), consider breaking up the pages by taxonomy, under different sections.

幸運的是，這些文件看起來很嚇人，但實際上很容易修復：確保著陸頁清楚地標識其用途，然後只包含指向其他頁面的連結以獲取更多資訊。如果著陸頁面上的某件事不僅僅是做一名交通警察，那它就沒有做好自己的工作。如果你有單獨的設定文件，請從著陸頁作為單獨的文件連結到該文件。如果你在著陸頁面上有太多連結（你的頁面不應該滾動多個螢幕），考慮按分類法將頁面分成不同部分。

Most poorly configured landing pages serve two different purposes: they are the “goto” page for someone who is a user of your product or API, or they are the home page for a team. Don’t have the page serve both masters—it will become confusing. Create a separate “team page” as an internal page apart from the main landing page. What the team needs to know is often quite different than what a customer of your API needs to know.

大多數配置不好的著陸頁有兩個不同的用途：它們是產品或 API 使用者的“入門”頁面，或者是團隊的主頁。不要讓頁面同時為兩個主體服務——這將變得混亂。建立一個獨立的“團隊頁面”，作為主著陸頁面之外的內部頁面。團隊內部需要知道的東西往往與你的 API 的客戶需要知道的東西完全不同。

Documentation Reviews 文件評審

At Google, all code needs to be reviewed, and our code review process is well understood and accepted. In general, documentation also needs review (though this is less universally accepted). If you want to “test” whether your documentation works, you should generally have someone else review it.

在谷歌，所有的程式碼都需要評審，我們的程式碼評審是被充分理解和接受的。一般來說，文件也需要評審（儘管這不太被普遍接受）。如果你想 "測試 "你的文件是否有效，你一般應該讓別人來評審。

A technical document benefits from three different types of reviews, each emphasizing different aspects:

- A technical review, for accuracy. This review is usually done by a subject matter expert, often another member of your team. Often, this is part of a code review itself.
- An audience review, for clarity. This is usually someone unfamiliar with the domain. This might be someone new to your team or a customer of your API.
- A writing review, for consistency. This is often a technical writer or volunteer.

一份技術文件得益於三種不同型別的評審，每一種都關注不同的方面：

- 技術評審，以保證準確性。這種審查通常是由主題專家完成的，通常是你的團隊的另一個成員。通常，這也是程式碼審查本身的一部分。
- 受眾評審，以確保清晰度。這通常是對該領域不熟悉的人。這可能是新加入你的團隊的人或你的 API 的客戶。
- 寫作評審，以保證一致性。這通常是一個技術撰稿人或志願者。

Of course, some of these lines are sometimes blurred, but if your document is high profile or might end up being externally published, you probably want to ensure that it receives more types of reviews. (We’ve used a similar review process for this book.) Any document tends to benefit from the aforementioned reviews, even if some of those reviews are ad hoc. That said, even getting one reviewer to review your text is preferable to having no one review it.

當然，其中一些界限有時是模糊的，但如果你的文件引人矚目或最終可能會在外部發布，你可能希望確保它收到更多型別的評審。(我們對這本書採用了類似的評審程式。)任何文件都傾向於從上述評審中受益，即使其中一些評審是臨時性的。也就是說，即使讓一個審查員評審你的文字也比沒有人評審要好。

Importantly, if documentation is tied into the engineering workflow, it will often improve over time. Most documents at Google now implicitly go through an audience review because at some point, their audience will be using them, and hopefully letting you know when they aren’t working (via bugs or other forms of feedback).

重要的是，如果文件與工程工作流程聯絡在一起，它往往會隨著時間的推移而改進。現在，谷歌的大多數文件都隱含地經過受眾審查，因為在某個時候，他們的讀者會使用這些文件，並希望在它們不起作用時（透過 bug 或其他形式的反饋）讓你知道。

Case Study: The Developer Guide Library 案例研究：開發者指南庫
As mentioned earlier, there were problems associated with having most (almost all) engineering documentation contained within a shared wiki: little ownership of important documentation, competing documentation, obsolete information, and difficulty in filing bugs or issues with documentation. But this problem was not seen in some documents: the Google C++ style guide was owned by a select group of senior engineers (style arbiters) who managed it. The document was kept in good shape because certain people cared about it. They implicitly owned that document. The document was also canonical: there was only one C++ style guide.

如前所述，大多數（幾乎所有）工程檔案都包含在一個共享的維基中，這其中存在一些問題：重要的文件沒有所有者、重複的文件、過時資訊，以及難以歸檔的錯誤或檔案問題。但是，這個問題在一些文件中並沒有出現：谷歌 C++風格指南是由一組精選的高階工程師（風格仲裁者）管理的。該文件被保持良好的狀態，因為有人關心它。他們隱含地擁有該文件。該文件也是規範的：只有一個 C++風格指南。

As previously mentioned, documentation that sits directly within source code is one way to promote the establishment of canonical documents; if the documentation sits alongside the source code, it should usually be the most applicable (hopefully). At Google, each API usually has a separate g3doc directory where such documents live (written as Markdown files and readable within our Code Search browser). Having the documentation exist alongside the source code not only establishes de facto ownership, it makes the documentation seem more wholly "part" of the code.

如前所述，直接位於原始碼中的文件是促進規範文件建立的一種方法；如果文件與原始碼放在一起，它通常應該是最適用的（希望如此）。在谷歌，每個 API 通常都有一個單獨的 g3doc 目錄，這些文件就在這裡（寫為標記檔案，在我們的程式碼搜尋瀏覽器中可讀）。將文件與原始碼放在一起不僅建立了事實上的所有權，而且使文件看起來更完全是程式碼的"一部分"。

Some documentation sets, however, cannot exist very logically within source code. A "C++ developer guide" for Googlers, for example, has no obvious place to sit within the source code. There is no master "C++" directory where people will look for such information. In this case (and others that crossed API boundaries), it became useful to create standalone documentation sets in their own depot. Many of these culled together associated existing documents into a common set, with common navigation and look-and-feel. Such documents were noted as "Developer Guides" and, like the code in the codebase, were under source control in a specific documentation depot, with this depot organized by topic rather than API. Often, technical writers managed these developer guides, because they were better at explaining topics across API boundaries.

然而，有些文件集在原始碼中不能非常合理地存在。例如，Google 的"C++開發者指南"， 在原始碼中沒有明確的位置。沒有主"C++"目錄，人們會在那裡尋找這些資訊。在這種情況下（以及其他跨 API 邊界的情況），在他們自己的儲存庫中建立獨立文件集變得非常有用。其中許多文件將關聯的現有文件挑選到一個公共集合中，具有公共導航和外觀。這些文件被稱為"開發人員指南"，與程式碼庫中的程式碼一樣，在一個特定的文件庫中受原始碼控制，該函式庫是按主題而不是 API 組織的。通常情況下，技術撰稿人管理這些開發者指南，因為他們更善於解釋跨 API 邊界的主題。

Over time, these developer guides became canonical. Users who wrote competing or supplementary documents became amenable to adding their documents to the canonical document set after it was established, and then deprecating their competing documents. Eventually, the C++ style guide became part of a larger "C++ Developer Guide." As the documentation set became more comprehensive and more authoritative, its quality also improved. Engineers began logging bugs because they knew someone was maintaining these documents. Because the documents were locked down under source control, with proper owners, engineers also began sending changelists directly to the technical writers.

隨著時間的推移，這些開發者指南成為經典。編寫重疊或補充文件的使用者在規範文件集建立後，開始願意將他們的文件新增到規範文件集中，然後廢除他們的重複文件。最終，C++風格指南成為一個更大的 "C++開發者指南 "的一部分。隨著文件集變得更全面、更權威，其品質也得到了提高。工程師們開始記錄錯誤，因為他們知道有人在維護這些文件。由於這些文件被鎖定在原始碼控制之下，並有適當的所有者，工程師們也開始直接向技術作者傳送變更列表。

The introduction of go/links (see Chapter 3) allowed most documents to, in effect,more easily establish themselves as canonical on any given topic. Our C++ Developer Guide became established at “go/cpp,” for example. With better internal search, go/links, and the integration of multiple documents into a common documentation set,such canonical documentation sets became more authoritative and robust over time.

引入 go/links（見第 3 章）後，大多數檔案實際上可以更容易地建立自己在任何特定主題上的規範性。例如，我們的《C++開發指南》就建立在 "go/cpp "上。有了更好的內部搜尋、go/links，以及將多個文件整合到一個共同的文件集，隨著時間的推移，這樣的規範文件集變得更加權威和強大。

Documentation Philosophy 寫文件秘訣

Caveat: the following section is more of a treatise on technical writing best practices (and personal opinion) than of “how Google does it.” Consider it optional for software engineers to fully grasp, though understanding these concepts will likely allow you to more easily write technical information.

注意：以下部分更像是一篇關於技術寫作最佳實踐的論文（和個人觀點），而不是 "谷歌是如何做到的"。對於軟體工程師來說，可以考慮讓他們完全掌握，儘管理解這些概念可能會讓你更容易寫出技術資訊。

WHO, WHAT, WHEN, WHERE, and WHY 誰，什麼，何時，何地，為什麼

Most technical documentation answers a “HOW” question. How does this work? How do I program to this API? How do I set up this server? As a result, there’s a tendency for software engineers to jump straight into the “HOW” on any given document and ignore the other questions associated with it: the WHO, WHAT, WHEN, WHERE, and WHY. It’s true that none of those are generally as important as the HOW—a design document is an exception because an equivalent aspect is often the WHY—but without a proper framing of technical documentation, documents end up confusing. Try to address the other questions in the first two paragraphs of any document:

- WHO was discussed previously: that’s the audience. But sometimes you also need to explicitly call out and address the audience in a document. Example: “This document is for new engineers on the Secret Wizard project.”
- WHAT identifies the purpose of this document: “This document is a tutorial designed to start a Frobber server in a test environment.” Sometimes, merely writing the WHAT helps you frame the document appropriately. If you start adding information that isn’t applicable to the WHAT, you might want to move that information into a separate document.
- WHEN identifies when this document was created, reviewed, or updated. Documents in source code have this date noted implicitly, and some other publishing schemes automate this as well. But, if not, make sure to note the date on which the document was written (or last revised) on the document itself.

- WHERE is often implicit as well, but decide where the document should live. Usually, the preference should be under some sort of version control, ideally with the source code it documents. But other formats work for different purposes as well. At Google, we often use Google Docs for easy collaboration, particularly on design issues. At some point, however, any shared document becomes less of a discussion and more of a stable historical record. At that point, move it to someplace more permanent, with clear ownership, version control, and responsibility.
- WHY sets up the purpose for the document. Summarize what you expect someone to take away from the document after reading it. A good rule of thumb is to establish the WHY in the introduction to a document. When you write the summary, verify whether you've met your original expectations (and revise accordingly).

大多數技術文件回答的是 "如何 "的問題。它是如何工作的？我如何對這個 API 進行程式設計？我如何設定這個伺服器？因此，軟體工程師有一種傾向，就是在任何給定的檔案中直接跳到 "如何"，而忽略了與之相關的其他問題：誰、什麼、什麼時候、什麼地方和為什麼。誠然，這些問題通常都不如 "如何 "重要——設計檔案是個例外，因為與之相當的方面往往是 "為什麼"——但如果沒有適當的技術文件框架，文件最終會變得混亂。試著在任何文件的前兩段解決其他問題：

- 之前討論的是 WHO：這就是受眾。但有時你也需要在檔案中明確地叫出並解決受眾的問題。例如。"本文件適用於秘密嚮導專案的新工程師。"
- WHAT 是確定本文件用途的內容：“本文件是一個旨在在測試環境中啟動 Frobber 伺服器的課程。”有時，只需編寫幫助你正確建構文件的內容即可。如果開始新增不適用於 WHAT 的資訊，則可能需要將該資訊移動到單獨的文件中。
- WHEN 是何時確定本檔案的建立、審查或更新時間。原始碼中的文件隱含記錄了該日期，其他一些釋出方案也會自動記錄該日期。但是，如果沒有，請確保在文件本身上註明文件的編寫日期（或最後一次修訂日期）。
- WHERE 通常也是隱含的，但要決定該文件應該放在哪裡。通常情況下，偏好應該在某種版本控制之下，最好是與它所記錄的原始碼一起。但其他格式也適用於不同的目的。在 Google，我們經常使用 Google Docs 以方便協作，特別是在設計問題上。然而，在某些時候，任何共享的檔案都不再是一種討論，而更像是一種穩定的歷史記錄。在這一點上，把它移到一個更永久的地方，有明確的所有權、版本控制和責任。
- WHY 設定檔案的目的。總結一下你希望別人在閱讀後能從檔案中得到什麼。一個好的經驗法則是在檔案的引言中確立 "為什麼"。當你寫總結的時候，驗證你是否達到了你最初的期望（並進行相應的修改）。

The Beginning, Middle, and End 開頭、中間和結尾

All documents—indeed, all parts of documents—have a beginning, middle, and end. Although it sounds amazingly silly, most documents should often have, at a minimum, those three sections. A document with only one section has only one thing to say, and very few documents have only one thing to say. Don't be afraid to add sections to your document; they break up the flow into logical pieces and provide readers with a roadmap of what the document covers.

所有的文件——事實上，文件的所有部分——都有一個開始、中間和結束。雖然這聽起來很愚蠢，但大多數文件通常至少應該有這三個部分。只有一個部分的文件只有一句話要說，很少文件只有一句話要說。不要害怕在文件中新增條款；它們將流程分解為邏輯部分，併為讀者提供文件內容的路線圖。

Even the simplest document usually has more than one thing to say. Our popular “C++ Tips of the Week” have traditionally been very short, focusing on one small piece of advice. However, even here, having sections helps. Traditionally, the first section denotes the problem, the middle section goes through the recommended solutions, and the conclusion summarizes the takeaways. Had the document consisted of only one section, some readers would doubtless have difficulty teasing out the important points.

即使是最簡單的文件通常也有不止一句話要說。我們受歡迎的 "每週 C++提示 "傳統上是非常簡短的，集中在一個小建議上。然而，即使在這裡，有一些章節也是有幫助的。傳統上，第一部分表示問題，中間部分是推薦的解決方案，結論總結了要點。如果該文件只有一個部分，一些讀者無疑會難以找出重要的要點。

Most engineers loathe redundancy, and with good reason. But in documentation, redundancy is often useful. An important point buried within a wall of text can be difficult to remember or tease out. On the other hand, placing that point at a more prominent location early can lose context provided later on. Usually, the solution is to introduce and summarize the point within an introductory paragraph, and then use the rest of the section to make your case in a more detailed fashion. In this case, redundancy helps the reader understand the importance of what is being stated.

大多數工程師厭惡冗餘，這是有道理的。但在文件中，冗餘通常是有用的。隱藏在文字牆內的一個要點可能很難記住或梳理。另一方面，前面將該點放置在更突出的位置可能會丟失後面提供的背景。通常，解決方法是在介紹性段落中介紹和總結要點，然後使用本節的其餘部分以更詳細的方式闡述你的案例。在這種情況下，冗餘有助於讀者理解所述內容的重要性。

The Parameters of Good Documentation 良好文件的衡量標準

There are usually three aspects of good documentation: completeness, accuracy, and clarity. You rarely get all three within the same document; as you try to make a document more “complete,” for example, clarity can begin to suffer. If you try to document every possible use case of an API, you might end up with an incomprehensible mess. For programming languages, being completely accurate in all cases (and documenting all possible side effects) can also affect clarity. For other documents, trying to be clear about a complicated topic can subtly affect the accuracy of the document; you might decide to ignore some rare side effects in a conceptual document, for example,because the point of the document is to familiarize someone with the usage of an API, not provide a dogmatic overview of all intended behavior.

好的文件通常有三個方面：完整性、準確性和清晰性。你很少在同一文件中得到這三點；例如，當你試圖使文件更加“完整”時，清晰度可能開始受到影響。如果你試圖記錄一個 API 的每一個可能的使用案例，你最終可能會得到一個難以理解的混亂。對於程式語言來說，在所有情況下完全準確（以及記錄所有可能的副作用）也會影響清晰度。對於其他文件，試圖弄清楚一個複雜的主題可能會微妙地影響文件的準確性；你可能會決定忽略概念文件中一些罕見的副作用，例如，因為本文件的目的是讓某人熟悉 API 的使用，而不是提供所有預期行為的教條式概述。

In each case, a “good document” is defined as the document that is doing its intended job. As a result, you rarely want a document doing more than one job. For each document (and for each document type), decide on its focus and adjust the writing appropriately. Writing a conceptual document? You probably don’t need to cover every part of the API. Writing a reference? You probably want this complete, but perhaps must sacrifice some clarity. Writing a landing page? Focus on organization and keep discussion to a minimum. All of this adds up to quality, which, admittedly, is stubbornly difficult to accurately measure.

在每種情況下，“良好的文件”都被定義為有效的文件。因此，你很少希望文件執行多個任務。對於每個文件（以及每種文件型別），確定其重點並適當調整寫作。寫概念文件？你可能不需要涵蓋 API 的每個部分。寫參考文件？你可能希望這是完整的，但可能必須犧牲一些清晰度。寫著陸頁？專注於組織，並儘量減少討論。所有這些都是為了提高品質，誠然，這是很難準確衡量的。

How can you quickly improve the quality of a document? Focus on the needs of the audience. Often, less is more. For example, one mistake engineers often make is adding design decisions or implementation details to an API document. Much like you should ideally separate the interface from an implementation within a welldesigned API, you should avoid discussing design decisions in an API document. Users don't need to know this information. Instead, put those decisions in a specialized document for that purpose (usually a design doc).

如何快速提高文件的品質？關注受眾的需求。通常，少就是多。例如，工程師經常犯的一個錯誤是將設計決策或實現細節新增到 API 文件中。就像你應該在一個設計良好的 API 中理想地將介面與實現分離一樣，你應該避免在 API 文件中討論設計決策。使用者不需要知道這些資訊。相反，將這些決策放在專門的文件中（通常是設計文件）。

Deprecating Documents 廢棄文件

Just like old code can cause problems, so can old documents. Over time, documents become stale, obsolete, or (often) abandoned. Try as much as possible to avoid abandoned documents, but when a document no longer serves any purpose, either remove it or identify it as obsolete (and, if available, indicate where to go for new information). Even for unowned documents, someone adding a note that "This no longer works!" is more helpful than saying nothing and leaving something that seems authoritative but no longer works.

就像舊程式碼可能導致問題一樣，舊文件也可能導致問題。隨著時間的推移，文件會變得陳舊、過時或（通常）被廢棄。儘可能避免使用過時的文件，但當文件不再具有任何用途時，請將其刪除或將其標識為已過時（如果可用，請指明獲取新資訊的位置）。即使對於無主文件，有人加上"這不再有效！"的註釋也比什麼都不說，留下一些看似權威但不再有效的東西更有幫助。

At Google, we often attach "freshness dates" to documentation. Such documents note the last time a document was reviewed, and metadata in the documentation set will send email reminders when the document hasn't been touched in, for example, three months. Such freshness dates, as shown in the following example—and tracking your documents as bugs—can help make a documentation set easier to maintain over time, which is the main concern for a document:

在谷歌，我們經常在文件中附加"保鮮日期"。此類文件會記錄文件最後一次審閱的時間，文件集中的元資料會在文件未被觸及時（例如，三個月）傳送電子郵件提醒。以下範例中所示的這些更新日期以及作為 bug 追蹤文件有助於使文件集隨著時間的推移更易於維護，這是文件的主要問題：

```
<!--*
# Document freshness: For more information, see go/fresh-source. freshness: { owner: `username`
reviewed: '2019-02-27' }

# 文件的新鮮度：更多資訊，請看 go/fresh-source。 freshness: { owner: `username` reviewed: '2019-02-27' }
*-->
```

Users who own such a document have an incentive to keep that freshness date current (and if the document is under source control, that requires a code review). As a result, it's a low-cost means to ensure that a document is looked over from time to time. At Google, we found that including the owner of a document in this freshness date within the document itself with a byline of "Last reviewed by..." led to increased adoption as well.

擁有此類文件的使用者有保持該新鮮度的動力（如果文件受原始碼控制，則需要程式碼審查）。因此，它是一種低成本的方法，可以確保文件不時被檢視。在谷歌，我們發現在這種新鮮感中包括文件的所有者文件中署名為“Last Review by…”的日期也增加了採用。

When Do You Need Technical Writers? 何時需要技術撰稿人？

When Google was young and growing, there weren’t enough technical writers in software engineering. (That’s still the case.) Those projects deemed important tended to receive a technical writer, regardless of whether that team really needed one. The idea was that the writer could relieve the team of some of the burden of writing and maintaining documents and (theoretically) allow the important project to achieve greater velocity. This turned out to be a bad assumption.

當谷歌年輕和成長時，軟體工程中沒有足夠的技術撰稿人。(現在仍然如此。）那些被認為是重要的專案往往會得到一個技術撰稿人，不管這個團隊是否真的需要。我們的想法是，技術撰稿人可以減輕團隊編寫和維護文件的一些負擔，（理論上）讓重要的專案取得更快的發展。這被證明是一個錯誤的假設。

We learned that most engineering teams can write documentation for themselves (their team) perfectly fine; it’s only when they are writing documents for another audience that they tend to need help because it’s difficult to write to another audience. The feedback loop within your team regarding documents is more immediate, the domain knowledge and assumptions are clearer, and the perceived needs are more obvious. Of course, a technical writer can often do a better job with grammar and organization, but supporting a single team isn’t the best use of a limited and specialized resource; it doesn’t scale. It introduced a perverse incentive: become an important project and your software engineers won’t need to write documents. Discouraging engineers from writing documents turns out to be the opposite of what you want to do.

我們瞭解到，大多數工程團隊可以為他們自己（他們的團隊）完美地編寫文件；只有當他們為另一個受眾編寫文件時，他們才傾向於需要幫助，因為為另一個受眾編寫文件很困難。團隊內部關於文件的反饋迴路更直接，領域知識和假設更清晰，感知的需求也更明顯。當然，技術撰稿人通常可以在語法和組織方面做得更好，但支援一個團隊並不是對有限的專業資源的最佳利用；它沒有規模化。它引入了一個不正當的激勵措施：成為一個重要的專案，你的軟體工程師就不需要寫文件了。不鼓勵工程師寫文件，結果是與你想做的事相反。結果證明，阻止工程師編寫文件與想要做的恰恰相反。

Because they are a limited resource, technical writers should generally focus on tasks that software engineers don’t need to do as part of their normal duties. Usually, this involves writing documents that cross API boundaries. Project Foo might clearly know what documentation Project Foo needs, but it probably has a less clear idea what Project Bar needs. A technical writer is better able to stand in as a person unfamiliar with the domain. In fact, it’s one of their critical roles: to challenge the assumptions your team makes about the utility of your project. It’s one of the reasons why many, if not most, software engineering technical writers tend to focus on this specific type of API documentation.

因為他們是有限的資源，技術撰稿人通常應該關注軟體工程師作為其正常職責的一部分需要完成的任務。通常，這涉及到編寫跨 API 邊界的文件。專案 Foo 可能清楚地知道專案 Foo 需要什麼文件，但它可能不太清楚專案 Bar 需要什麼。技術撰稿人更能以不熟悉該領域的人的身份出現。事實上，這是他們的關鍵角色之一：挑戰團隊對專案效用的假設。這就是為什麼許多（如果不是大多數的話）軟體工程技術撰稿人傾向於關注這種特定型別的 API 文件的原因之一。

Conclusion 總結

Google has made good strides in addressing documentation quality over the past decade, but to be frank, documentation at Google is not yet a first-class citizen. For comparison, engineers have gradually accepted that testing is necessary for any code change, no matter how small. As well, testing tooling is robust, varied and plugged into an engineering workflow at various points. Documentation is not ingrained at nearly the same level.

在過去的十年中，谷歌在解決文件品質方面取得了長足的進步，但坦率地說，谷歌的文件還不是一等公民。相比之下，工程師們已經逐漸接受了測試對於任何程式碼修改都是必要的，無論更改多麼小。同樣，測試工具是健壯的、多樣的，並在不同的點上插入工程工作流程中。文件還達不到在相同的層次上紮根的。

To be fair, there's not necessarily the same need to address documentation as with testing. Tests can be made atomic (unit tests) and can follow prescribed form and function. Documents, for the most part, cannot. Tests can be automated, and schemes to automate documentation are often lacking. Documents are necessarily subjective; the quality of the document is measured not by the writer, but by the reader, and often quite asynchronously. That said, there is a recognition that documentation is important, and processes around document development are improving. In this author's opinion, the quality of documentation at Google is better than in most software engineering shops.

公平地說，解決文件問題的必要性不一定和測試一樣。測試可以是原子化的（單元測試），可以遵循規定的形式和功能。在大多數情況下，文件都做不到。測試可以自動化，而文件自動化的方案通常是缺乏的。文件必然是主觀的；文件的品質不是由作者來衡量的，而是由讀者來衡量的，而且通常是非同步的。儘管如此，人們認識到文件的重要性，圍繞文件開發的過程也在不斷改進。在筆者看來，谷歌公司的文件品質比大多數軟體工程公司的要好。

To change the quality of engineering documentation, engineers—and the entire engineering organization—need to accept that they are both the problem and the solution. Rather than throw up their hands at the state of documentation, they need to realize that producing quality documentation is part of their job and saves them time and effort in the long run. For any piece of code that you expect to live more than a few months, the extra cycles you put in documenting that code will not only help others, it will help you maintain that code as well.

為了改變工程文件的品質，工程師和整個工程組織需要接受他們既是問題又是解決方案。他們需要意識到，製作高品質的文件是他們工作的一部分，從長遠來看，這可以節省他們的時間和精力，而不是在當前文件狀態下束手無策。對於任何一段生命週期超過幾個月的程式碼，記錄該程式碼的額外週期不僅有助於其他人，也有助於維護該程式碼。

TL;DRs 內容提要

- Documentation is hugely important over time and scale.
- Documentation changes should leverage the existing developer workflow.
- Keep documents focused on one purpose.
- Write for your audience, not yourself.
- 隨著時間和規模的增長，文件是非常重要的。
- 文件的變化應該利用現有的開發人員的工作流程。
- 讓文件集中在一個職責（用途）上。

- 為你的受眾而不是你自己而寫。

---

1. OK, you will need to maintain it and revise it occasionally. ↩
2. English is still the primary language for most programmers, and most technical documentation for programmers relies on an understanding of English. ↩
3. When we deprecated GooWiki, we found that around 90% of the documents had no views or updates in the previous few months. ↩