CHAPTER 1

What Is Software Engineering?

第一章 軟體工程是什麼？

Written by Titus Winters

Edited by Tom Manshreck

Nothing is built on stone; all is built on sand, but we must build as if the sand were stone. --Jorge Luis Borges

We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

我們看到，程式設計和軟體工程之間有三個關鍵的區別：時間、規模和權衡取捨。在一個軟體工程專案中，工程師需要更多關注時間成本和需求變更。在軟體工程中，我們需要更加關注規模和效率，無論是對我們生產的軟體，還是對生產軟體的組織。最後，作為軟體工程師，我們被要求做出更復雜的決策，其結果風險更大，而且往往是基於對時間和規模增長的不確定性的預估。

Within Google, we sometimes say, “Software engineering is programming integrated over time.” Programming is certainly a significant part of software engineering: after all, programming is how you generate new software in the first place. If you accept this distinction, it also becomes clear that we might need to delineate between programming tasks (development) and software engineering tasks (development, modification, maintenance). The addition of time adds an important new dimension to programming. Cubes aren’t squares, distance isn’t velocity. Software engineering isn’t programming.

在谷歌內部，我們有時會說，"軟體工程是隨著時間推移的程式設計。"程式設計當然是軟體工程的一個重要部分：畢竟，程式設計首先是生成新軟體的方式。如果你接受這一區別，那麼很明顯，我們可能需要在程式設計任務（開發）和軟體工程任務（開發、修改、維護）之間進行劃分。時間的增加為程式設計增加了一個重要的新維度。這是一個立方體三維模型不是正方形的二維模型，距離不是速度。軟體工程不是程式設計。

One way to see the impact of time on a program is to think about the question, “What is the expected life span[^1] of your code?” Reasonable answers to this question vary by roughly a factor of 100,000. It is just as reasonable to think of code that needs to last for a few minutes as it is to imagine code that will live for decades. Generally, code on the short end of that spectrum is unaffected by time. It is unlikely that you need to adapt to a new version of your underlying libraries, operating system (OS), hardware, or language version for a program whose utility spans only an hour. These short-lived systems are effectively “just” a programming problem, in the same way that a cube compressed far enough in one dimension is a square. As we expand that time to allow for longer life spans, change becomes more important. Over a span of a decade or more, most program dependencies, whether implicit or explicit, will likely change. This recognition is at the root of our distinction between software engineering and programming.

瞭解時間對程式的影響的一種方法是思考“程式碼的預期生命週期是多少？”這個問題的合理答案大約相差100,000倍。想到生命週期幾分鐘的程式碼和想象將持續執行幾十年的程式碼是一樣合理。通常，週期短的程式碼不受時間的影響。對於一個只需要存活一個小時的程式，你不太可能考慮其底層庫、作業系統（OS）、硬體或語言版本的新版本。這些短期系統實際上“只是”一個程式設計問題，就像在一個維度中壓縮得足夠扁的立方體是正方形一樣。隨著我們擴大時間維度，允許更長的生命週期，改變顯得更加重要。在十年或更長的時間裡，大多數程式依賴關係，無論是隱式的還是顯式的，都可能發生變化。這一認識是我們區分軟體工程和程式設計的根本原因。

> [^1]: We don’t mean “execution lifetime,” we mean “maintenance lifetime”—how long will the code continue to be built, executed, and maintained? How long will this software provide value?
>
> [1] 我們不是指“開發生命週期”，而是指“維護生命週期”——程式碼將持續構建、執行和維護多長時間？這個軟體能提供多長時間的價值？

This distinction is at the core of what we call sustainability for software. Your project is sustainable if, for the expected life span of your software, you are capable of reacting to whatever valuable change comes along, for either technical or business reasons. Importantly, we are looking only for capability—you might choose not to perform a given upgrade, either for lack of value or other priorities.[^2] When you are fundamentally incapable of reacting to a change in underlying technology or product direction, you’re placing a high-risk bet on the hope that such a change never becomes critical. For short-term projects, that might be a safe bet. Over multiple decades, it probably isn’t.[^3]

這種區別是我們所說的軟體可持續性的核心。如果在軟體的預期生命週期內，你能夠對任何有價值的變化做出反應，無論是技術還是商業原因，那麼你的專案是可持續的。重要的是，我們只關注能力——你可能因為缺乏價值或其他優先事項而選擇不進行特定的升級。當你基本上無法對基礎技術或產品方向的變化做出反應時，你就把高風險賭注押在希望這種變化永遠不會變得至關重要。對於短期專案，這可能是一個安全的賭注。幾十年後，情況可能並非如此。

Another way to look at software engineering is to consider scale. How many people are involved? What part do they play in the development and maintenance over time? A programming task is often an act of individual creation, but a software engineering task is a team effort. An early attempt to define software engineering produced a good definition for this viewpoint: “The multiperson development of multiversion programs.”[^4] This suggests the difference between software engineering and programming is one of both time and people. Team collaboration presents new problems, but also provides more potential to produce valuable systems than any single programmer could.

另一種看待軟體工程的方法是考慮規模。有多少人蔘與？隨著時間的推移，他們在開發和維護中扮演什麼角色？程式設計任務通常是個人的創造行為，但軟體工程任務是團隊的工作。早期定義軟體工程的嘗試為這一觀點提供了一個很好的定義：“多人開發的多版本程式”。這表明軟體工程和程式設計之間的區別是時間和人的區別。團隊協作帶來了新的問題，但也提供了比任何單個程式設計師更多的潛力來產生有價值的系統。

> [^2]: This is perhaps a reasonable hand-wavy definition of technical debt: things that “should” be done, but aren’t yet—the delta between our code and what we wish it was.
>
> 2 這也許是一個合理且簡單的技術債務定義：那些“應該”做卻還未完成的事————我們程式碼的現狀和理想程式碼之間的差距。
>
> [^3]: Also consider the issue of whether we know ahead of time that a project is going to be long lived.

> 3 也要考慮我們是否提前知道專案將長期存在的問題。
>
> [^4]: There is some question as to the original attribution of this quote; consensus seems to be that it was originally phrased by Brian Randell or Margaret Hamilton, but it might have been wholly made up by Dave Parnas. The common citation for it is “Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee,” Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO.
>
> 4 關於這句話的原始出處有一些疑問；人們似乎一致認為它最初是由Brian Randell或Margaret Hamilton提出的，但它可能完全是由Dave Parnas編造的。這句話的常見引文是 "軟體工程技術。由北約科學委員會主辦的會議報告1969年10月27日至31日，義大利羅馬，布魯塞爾，北約科學事務司。

Team organization, project composition, and the policies and practices of a software project all dominate this aspect of software engineering complexity. These problems are inherent to scale: as the organization grows and its projects expand, does it become more efficient at producing software? Does our development workflow become more efficient as we grow, or do our version control policies and testing strategies cost us proportionally more? Scale issues around communication and human scaling have been discussed since the early days of software engineering, going all the way back to the Mythical Man Month. [^5] Such scale issues are often matters of policy and are fundamental to the question of software sustainability: how much will it cost to do the things that we need to do repeatedly?

團隊組織、專案組成以及軟體專案的策略和實踐都支配著軟體工程複雜性。這些問題是規模所固有的：隨著組織的增長和專案的擴充套件，它在生產軟體方面是否變得更加高效？我們的開發工作流程隨著我們的發展，效率會提高，還是版本控制策略和測試策略的成本會相應增加？從軟體工程的早期開始，人們就一直在討論溝通和人員的規模問題，一直追溯到《人月神話》。這種規模問題通常是策略的問題，也是軟體可持續性問題的基礎：重複做我們需要做的事情要花多少錢？

We can also say that software engineering is different from programming in terms of the complexity of decisions that need to be made and their stakes. In software engineering, we are regularly forced to evaluate the trade-offs between several paths forward, sometimes with high stakes and often with imperfect value metrics. The job of a software engineer, or a software engineering leader, is to aim for sustainability and management of the scaling costs for the organization, the product, and the development workflow . With those inputs in mind, evaluate your trade-offs and make rational decisions. We might sometimes defer maintenance changes, or even embrace policies that don’t scale well, with the knowledge that we’ll need to revisit those decisions. Those choices should be explicit and clear about the deferred costs.

我們還可以說，軟體工程與程式設計的不同之處在於需要做出的決策的複雜性及其風險。在軟體工程中，我們經常被迫在幾個路徑之間做評估和權衡，有時風險很高，而且價值指標不完善。軟體工程師或軟體工程負責人的工作目標是實現組織、產品和開發工作流程的可持續性和管理擴充套件成本為目標。考慮到這些投入，評估你的權衡並做出理性的決定。有時，我們可能會推遲維護更改，甚至接受擴充套件性不好的策略，因為我們知道需要重新審視這些決策。這些決策應該是明確的和清晰的遞延成本。

Rarely is there a one-size-fits-all solution in software engineering, and the same applies to this book. Given a factor of 100,000 for reasonable answers on “How long will this software live,” a range of perhaps a factor of 10,000 for “How many engineers are in your organization,” and who-knows-how-much for “How many compute resources are available for your project,” Google’s experience will probably not match yours. In this book, we aim to present what we’ve found that works for us in the construction and maintenance of software that we expect to last for decades, with tens of thousands of engineers, and world-spanning

compute resources. Most of the practices that we find are necessary at that scale will also work well for smaller endeavors: consider this a report on one engineering ecosystem that we think could be good as you scale up. In a few places, super-large scale comes with its own costs, and we'd be happier to not be paying extra overhead. We call those out as a warning. Hopefully if your organization grows large enough to be worried about those costs, you can find a better answer.

在軟體工程中很少有一刀切的解決方案，這本書也是如此。考慮到"這個軟體能使用多久"的合理答案是100,000倍，而"你的組織中有多少工程師"的範圍可能是10,000，誰知道"你的專案有多少計算資源可用"的範圍是多少，谷歌的經驗可能與你的經驗不一致。在本書中，我們的目標是介紹我們在構建和維護軟體方面的發現，這些軟體預計將持續數十年，擁有數萬計的工程師和遍佈世界的計算資源。我們發現在這種規模下所需要的大多數做法也能很好地適用於複雜度較小的系統：考慮一下這是一個我們認為在你們擴大的時候可以做的很好的工程生態系統的報告。在一些地方，超大規模有其自身的成本，我們更傾向於不付出額外的管理成本。我們發出警告。希望如果你的組織發展到足以擔心這些成本，你可以找到更好的答案。

Before we get to specifics about teamwork, culture, policies, and tools, let's first elaborate on these primary themes of time, scale, and trade-offs.

在我們討論團隊合作、文化、策略和工具的細節之前，讓我們首先闡述一下時間、規模和權衡這些主要主題。

> [^5]: Frederick P. Brooks Jr. The Mythical Man-Month: Essays on Software Engineering (Boston: Addison-Wesley, 1995)
>
> Frederick P. Brooks Jr. The Mythical Man-Month: 關於軟體工程的論文（波士頓：Addison-Wesley，1995）。

Time and Change 時間與變化

When a novice is learning to program, the life span of the resulting code is usually measured in hours or days. Programming assignments and exercises tend to be write- once, with little to no refactoring and certainly no long-term maintenance. These programs are often not rebuilt or executed ever again after their initial production. This isn't surprising in a pedagogical setting. Perhaps in secondary or post-secondary education, we may find a team project course or hands-on thesis. If so, such projects are likely the only time student code will live longer than a month or so. Those developers might need to refactor some code, perhaps as a response to changing requirements, but it is unlikely they are being asked to deal with broader changes to their environment.

當新手學習程式設計時，編碼的生命週期通常以小時或天為單位。程式設計作業和練習往往是一次編寫的，幾乎沒有重構，當然也沒有長期維護。這些程式通常在初始生產後不再重建或再次執行。這在教學環境中並不奇怪。也許在中學或中學後教育，我們可以找到團隊專案課程或實踐論文。如果是這樣的，專案很可能是學生們的程式碼生命週期超過一個月左右的時間。這些開發人員可能需要重構一些程式碼，也許是為了應對不斷變化的需求，但他們不太可能被要求處理環境的更大變化。

We also find developers of short-lived code in common industry settings. Mobile apps often have a fairly short life span,[^6] and for better or worse, full rewrites are relatively common. Engineers at an early-stage startup might rightly choose to focus on immediate goals over long-term investments: the company might not live long enough to reap the benefits of an infrastructure investment that pays off slowly. A serial startup developer could very reasonably have 10 years of development experience and little or no experience maintaining any piece of software expected to exist for longer than a year or two.

我們還在常見的行業環境中找到短期程式碼的開發人員。移動應用程式的生命週期通常很短，而且無論好壞，完全重寫都是相對常見的。初創初期的工程師可能會正確地選擇關注眼前目標而不是長期投資：公司可能活得不夠長，無法從回報緩慢的基礎設施投資中獲益。一個連續工作多年的開發人員可能有10年的開發經驗，並且鮮少或根本沒有維護任何預期存在超過一年或兩年的軟體的經驗。

On the other end of the spectrum, some successful projects have an effectively unbounded life span: we can't reasonably predict an endpoint for Google Search, the Linux kernel, or the Apache HTTP Server project. For most Google projects, we must assume that they will live indefinitely—we cannot predict when we won't need to upgrade our dependencies, language versions, and so on. As their lifetimes grow, these long-lived projects *eventually* have a different feel to them than programming assignments or startup development.

另一方面，一些成功的專案實際上有無限的生命週期：我們無法準確地預測Google搜尋、Linux核心或Apache HTTP伺服器專案的終點。對於大多數谷歌專案，我們必須假設它們將無限期地存在，我們無法預測何時不需要升級依賴項、語言版本等。隨著他們生命週期的延長，這些長期專案最終會有一種不同於程式設計任務或初創企業發展不同的感受。

Consider Figure 1-1, which demonstrates two software projects on opposite ends of this "expected life span" spectrum. For a programmer working on a task with an expected life span of hours, what types of maintenance are reasonable to expect? That is, if a new version of your OS comes out while you're working on a Python script that will be executed one time, should you drop what you're doing and upgrade? Of course not: the upgrade is not critical. But on the opposite end of the spectrum, Google Search being stuck on a version of our OS from the 1990s would be a clear problem.

考慮圖1-1，它演示了兩個軟體專案的"預期生命週期"的範圍。對於從事預期生命週期為小時的任務的程式來說，什麼型別的維護是合理的？也就是說，如果你正在編寫一個只需執行一次的 Python 指令碼，這時作業系統推出了新版本，你應該放下手頭的工作去升級系統嗎？當然不是：升級並不重要。但與之相反，如果谷歌搜尋停留在20世紀90年代的作業系統版本上顯然是一個問題。

> [^6]: Appcelerator, "Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan," Axway Developer blog, December 6, 2012.
>
> 除了死亡、稅收和短暫的移動應用生命，沒有什麼是確定的

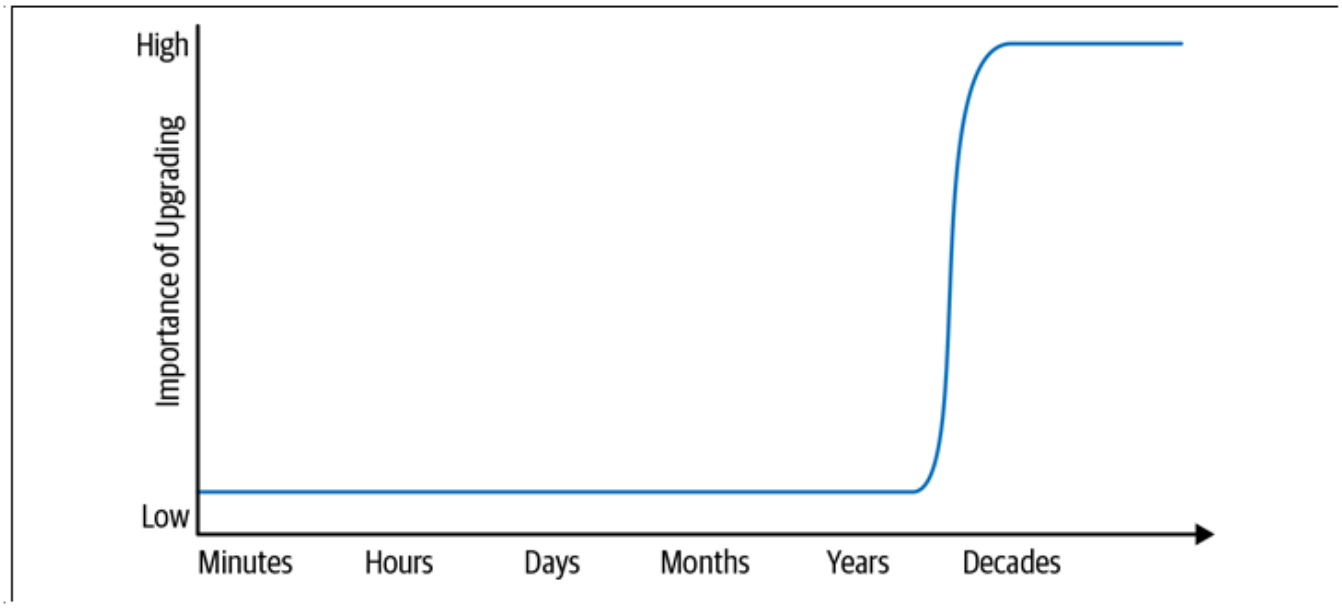

Figure 1-1. Life span and the importance of upgrades

The low and high points on the expected life span spectrum suggest that there's a transition somewhere. Somewhere along the line between a one-off program and a project that lasts for decades, a transition happens: a project must begin to react to changing externalities.[^7] For any project that didn't plan for upgrades from the start, that transition is likely very painful for three reasons, each of which compounds the others:

- You're performing a task that hasn't yet been done for this project; more hidden assumptions have been baked-in.
- The engineers trying to do the upgrade are less likely to have experience in this sort of task.
- The size of the upgrade is often larger than usual, doing several years' worth of upgrades at once instead of a more incremental upgrade.

預期生命週期範圍的低點和高點表明某處有一個過渡。介於一次性計劃和持續十年的專案，發生了轉變：一個專案必須開始對不斷變化的外部因素做出反應。對於任何一個從一開始就沒有升級計劃的專案，這種轉變可能會非常痛苦，原因有三個，每一個都會使其他原因變得複雜：

- 你正在執行本專案尚未完成的任務；更多隱藏的假設已經成立。
- 嘗試進行升級的工程師不太可能具有此類任務的經驗。
- 升級的規模通常比平時大，一次完成幾年的升級，而不是增量升級。

And thus, after actually going through such an upgrade once (or giving up part way through), it's pretty reasonable to overestimate the cost of doing a subsequent upgrade and decide "Never again." Companies that come to this conclusion end up committing to just throwing things out and rewriting their code, or deciding to never upgrade again. Rather than take the natural approach by avoiding a painful task, sometimes the more responsible answer is to invest in making it less painful. It all depends on the cost of your upgrade, the value it provides, and the expected life span of the project in question.

因此，在經歷過一次升級（或中途放棄）之後，高估後續升級的成本並決定"永不再升級"是非常合理的。得出這個結論的公司最終承諾放棄並重寫程式碼，或決定不再升級。有時，更負責任的答案不是採取常規的方法避免痛苦的任務，而是投入資源用於減輕痛苦。這一切都取決於升級的成本、提供的價值以及相關專案的預期生命週期。

> [^7]: Your own priorities and tastes will inform where exactly that transition happens. We've found that most projects seem to be willing to upgrade within five years. Somewhere between 5 and 10 years seems like a conservative estimate for this transition in general.
>
> 7 你自己的優先次序和品味會告訴你這種轉變到底發生在哪裡。我們發現，大多數專案似乎願意在五年內升級。一般來說，5到10年似乎是這一轉變的保守估計。

Getting through not only that first big upgrade, but getting to the point at which you can reliably stay current going forward, is the essence of long-term sustainability for your project. Sustainability requires planning and managing the impact of required change. For many projects at Google, we believe we have achieved this sort of sustainability, largely through trial and error.

不僅完成了第一次大升級，而且達到可靠地保持當前狀態的程度，這是專案長期可持續性的本質。可持續性要求規劃和管理所需變化的影響。對於谷歌的許多專案，我們相信我們已經實現了這種持續能力，主要是通過試驗和錯誤。

So, concretely, how does short-term programming differ from producing code with a much longer expected life span? Over time, we need to be much more aware of the difference between "happens to work" and "is maintainable." There is no perfect solution for identifying these issues. That is unfortunate, because keeping software maintainable for the long-term is a constant battle.

那麼，具體來說，短期程式設計與生成預期生命週期更長的程式碼有何不同？隨著時間的推移，我們需要更多地意識到"正常工作"和"可維護"之間的區別。識別這些問題沒有完美的解決方案。這是不幸的，因為保持軟體的長期可維護性是一場持久戰。

Hyrum's Law 海勒姆定律

If you are maintaining a project that is used by other engineers, the most important lesson about "it works" versus "it is maintainable" is what we've come to call *Hyrum's Law*:

> *With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.*

如果你正在維護一個由其他工程師使用的專案，那麼關於"有效"與"可維護"最重要的一課就是我們所說的海勒姆定律：

> *當一個 API 有足夠多的使用者的時候，在約定中你承諾的什麼都無所謂，所有在你係統裡面被觀察到的行為都會被一些使用者直接依賴。*

In our experience, this axiom is a dominant factor in any discussion of changing software over time. It is conceptually akin to entropy: discussions of change and maintenance over time must be aware of Hyrum's Law[^8] just as discussions of efficiency or thermodynamics must be mindful of entropy. Just because entropy never decreases doesn't mean we shouldn't try to be efficient. Just because Hyrum's Law will apply when maintaining software doesn't mean we can't plan for it or try to better understand it. We can mitigate it, but we know that it can never be eradicated.

根據我們的經驗，這個定律在任何關於軟體隨時間變化的討論中都是一個主導因素。它在概念上類似於熵：對隨時間變化和維護的討論必須瞭解海勒姆定律，正如對效率或熱力學的討論必須注意熵一樣。僅僅因為熵從不減少並不意味著我們不應該努力提高效率。在維護軟體時，"海勒姆定律 "會適用，但這並不意味著我們不能對它進行規劃或試圖更好地瞭解它。我們可以減輕它，但我們知道，它永遠不可能被根除。

Hyrum's Law represents the practical knowledge that—even with the best of intentions, the best engineers, and solid practices for code review—. As an API owner, you will gain some flexibility and freedom by being clear about interface promises, but in practice, the complexity and difficulty of a given change also depends on how useful a user finds some observable behavior of your API. If users cannot depend on such things, your API will be easy to change. Given enough time and enough users, even the most innocuous change will break something;[^9] your analysis of the value of that change must incorporate the difficulty in investigating, identifying, and resolving those breakages.

海勒姆定律代表了一種實踐知識，即使有最好的規劃、最好的工程師和可靠的程式碼評審實踐，我們也不能假設完全遵守已釋出的契約或最佳實踐。作為API所有者，通過明確地介面約定，你將獲得一定的靈活性和自由度，但在實踐中，給定更改的複雜性和難度還取決於使用者對你的API的一些可觀察行為的有用程度。如果使用者不能依賴這些東西，那麼你的API將很容易更改。如果有足夠的時間和足夠的使用者，即使是最無害的變更也會破壞某些東西；你對變更價值的分析必須包含調查、識別和解決這些缺陷的難度。

> [^8]: To his credit, Hyrum tried really hard to humbly call this "The Law of Implicit Dependencies," but "Hyrum's Law" is the shorthand that most people at Google have settled on.
>
> 值得稱道的是，海勒姆非常努力地將其稱為 "隱性依賴定律"，但 "海勒姆定律 "是谷歌公司大多數人都認可的簡稱。
>
> [^9]: See "Workflow," an xkcd comic.
>
> 見 "工作流程"，一幅xkcd漫畫。

Example: Hash Ordering 雜湊排序

Consider the example of hash iteration ordering. If we insert five elements into a hash-based set, in what order do we get them out?

考慮雜湊迭代排序的例子。如果我們在一個基於雜湊的集合中插入五個元素，我們將以什麼順序將它們取出？

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

Most programmers know that hash tables are non-obviously ordered. Few know the specifics of whether the particular hash table they are using is intending to provide that particular ordering forever. This might seem unremarkable, but over the past decade or two, the computing industry's experience using such types has evolved:

- Hash flooding[^10] attacks provide an increased incentive for nondeterministic hash iteration.
- Potential efficiency gains from research into improved hash algorithms or hash containers require changes to hash iteration order.
- Per Hyrum's Law, programmers will write programs that depend on the order in which a hash table is traversed, if they have the ability to do so.

大多數程式設計師都知道雜湊表是無序的。很少有人知道他們使用的特定雜湊表是否打算永遠提供特定的排序。這似乎不起眼，但在過去的一二十年中，計算行業使用這類型別的經驗不斷髮展：

- 雜湊洪水攻擊增加了非確定性雜湊迭代的動力。
- 研究改進的雜湊演算法或雜湊容器的潛在效率收益需要更改雜湊迭代順序。
- 根據海勒姆定律，如有能力程式設計師可根據雜湊表的遍歷順序編寫程式。

As a result, if you ask any expert "Can I assume a particular output sequence for my hash container?" that expert will presumably say "No." By and large that is correct, but perhaps simplistic. A more nuanced answer is, "If your code is short-lived, with no changes to your hardware, language runtime, or choice of data structure, such an assumption is fine. If you don't know how long your code will live, or you cannot promise that nothing you depend upon will ever change, such an assumption is incorrect." Moreover, even if your own implementation does not depend on hash container order, it might be used by other code that implicitly creates such a dependency. For example, if your library serializes values into a Remote Procedure Call (RPC) response, the RPC caller might wind up depending on the order of those values.

因此，如果你問任何一位專家"我能為我的雜湊容器設定一個的輸出序列嗎？"這位專家大概會說"不"。總的來說，這是正確的，但過於簡單。一個更微妙的回答是，"如果你的程式碼是短期的，沒有對硬體、語言執行時或資料結構的選擇進行任何更改，那麼這樣的假設是正確的。如果你不知道程式碼的生命週期，或者你不能保證你所依賴的任何東西都不會改變，那麼這樣的假設是不正確的。"，即使你自己的實現不依賴於雜湊容器順序，也可能被隱式建立這種依賴關係的其他程式碼使用。例如，如果庫將值序列化為遠端過程呼叫（RPC）響應，則RPC呼叫程式可能會根據這些值的順序結束。

This is a very basic example of the difference between "it works" and "it is correct." For a short-lived program, depending on the iteration order of your containers will not cause any technical problems. For a software engineering project, on the other hand, such reliance on a defined order is a risk—given enough time, something will make it valuable to change that iteration order. That value can manifest in a number of ways, be it efficiency, security, or merely future-proofing the data structure to allow for future changes. When that value becomes clear, you will need to weigh the trade- offs between that value and the pain of breaking your developers or customers.

這是"可用"和"正確"之間區別的一個非常基本的例子。對於一個短期的程式，依賴容器的迭代順序不會導致任何技術問題。另一方面，對於一個軟體工程專案來說，如果有足夠的時間，這種對已定義順序的依賴是一種風險使更改迭代順序變得有價值。這種價值可以通過多種方式體現出來，無論是效率、安全性，還是僅僅是資料結構的未來驗證，以允許將來的更改。當這一價值變得清晰時，你需要權衡這一價值與破壞開發人員或客戶的痛苦之間的平衡。

> [^10]: A type of Denial-of-Service (DoS) attack in which an untrusted user knows the structure of a hash table and the hash function and provides data in such a way as to degrade the algorithmic performance of operations on the table.
>
> 一種拒絕服務（DoS）攻擊，其中不受信任的使用者知道雜湊表和雜湊函式的結構，並以降低表上操作的演算法效能的方式提供資料。

Some languages specifically randomize hash ordering between library versions or even between execution of the same program in an attempt to prevent dependencies. But even this still allows for some Hyrum's Law surprises: there is code that uses hash iteration ordering as an inefficient random-number generator. Removing such randomness now would break those users. Just as entropy increases in every thermodynamic system, Hyrum's Law applies to every observable behavior.

一些語言專門在庫版本之間，甚至在執行相同程式的隨機雜湊排序，以防止依賴關係。但即使這樣，也會出現一些令人驚訝的海勒姆定律：有些程式碼使用雜湊迭代排序作為一個低效的隨機數生成器。現在消除這種隨機性將破壞這些使用者原使用方式。正如熵在每個熱力學系統中增加一樣，海勒姆定律適用於所有可觀察到的行為。

Thinking over the differences between code written with a "works now" and a "works indefinitely" mentality, we can extract some clear relationships. Looking at code as an artifact with a (highly) variable lifetime requirement, we can begin to categorize programming styles: code that depends on brittle and unpublished features of its dependencies is likely to be described as "hacky" or "clever," whereas code that follows best practices and has planned for the future is more likely to be described as "clean" and "maintainable." Both have their purposes, but which one you select depends crucially on the expected life span of the code in question. We've taken to saying, "It's programming if 'clever' is a compliment, but it's software engineering if 'clever' is an accusation."

思考一下用"當前可用"和"一直可用"心態編寫的程式碼之間的差異，我們可以提取出一些明確的關係。將程式碼視為具有（高度）可變生命週期需求的構件，我們可以開始對程式設計風格進行分類：依賴其依賴性的脆弱和未釋出特性的程式碼可能被描述為"黑客"或"聰明"而遵循最佳實踐併為未來規劃的程式碼更可能被描述為"乾淨"和"可維護"。兩者都有其目的，但你選擇哪一個關鍵取決於所討論程式碼的預期生命週期。我們常說，"如果'聰明'是一種恭維，那就是程式，如果'聰明'是一種指責，那就是軟體工程。"

Why Not Just Aim for "Nothing Changes"? 為什麼不以"無變化"為目標？

Implicit in all of this discussion of time and the need to react to change is the assumption that change might be necessary. Is it?

在所有關於時間和對變化作出反應的討論中，隱含著一個假設，即變化可能是必要的？

As with effectively everything else in this book, it depends. We'll readily commit to "For most projects, over a long enough time period, everything underneath them might need to be changed." If you have a project written in pure C with no external dependencies (or only external dependencies that promise great long-term stability, like POSIX), you might well be able to avoid any form of refactoring or difficult upgrade. C does a great job of providing stability—in many respects, that is its primary purpose.

與本書中的其他內容一樣，這取決於實際情況。我們很樂意承諾"對於大多數專案，在足夠長的時間內，它們下面的一切都可能需要更改。"如果你有一個用純C編寫的專案，沒有外部依賴項（或者只有保證長期穩定性的外部依賴項，如POSIX），你完全可以避免任何形式的重構或困難的升級。C在提供多方面穩定性方面做了大量工作，這是其首要任務。

Most projects have far more exposure to shifting underlying technology. Most programming languages and runtimes change much more than C does. Even libraries implemented in pure C might change to support new features, which can affect downstream users. Security problems are disclosed in all manner of technology, from processors to networking libraries to application code. Every piece of technology upon which your project depends has some (hopefully small) risk of containing critical bugs and security

vulnerabilities that might come to light only after you've started relying on it. If you are incapable of deploying a patch for Heartbleed or mitigating speculative execution problems like Meltdown and Spectre because you've assumed (or promised) that nothing will ever change, that is a significant gamble.

大多數專案更多地接觸到不斷變化的基礎技術。大多數程式語言和執行時的變化要比C大得多。甚至用純C實現的庫也可能改變以支援新特性，這可能會影響下游使用者。從處理器到網路庫，再到應用程式程式碼，各種技術都會暴露安全問題。你的專案所依賴的每一項技術都有一些（希望很小）包含關鍵bug和安全漏洞的風險，這些漏洞只有在你開始依賴它之後才會暴露出來。如果你無法部署心臟出血或緩解推測性執行漏洞（如熔燬和幽靈）的修補程式，因為你假設（或保證）什麼都不會改變，這是一場巨大的賭博。

Efficiency improvements further complicate the picture. We want to outfit our datacenters with cost-effective computing equipment, especially enhancing CPU efficiency. However, algorithms and data structures from early-day Google are simply less efficient on modern equipment: a linked-list or a binary search tree will still work fine, but the ever-widening gap between CPU cycles versus memory latency impacts what "efficient" code looks like. Over time, the value in upgrading to newer hardware can be diminished without accompanying design changes to the software. Backward compatibility ensures that older systems still function, but that is no guarantee that old optimizations are still helpful. Being unwilling or unable to take advantage of such opportunities risks incurring large costs. Efficiency concerns like this are particularly subtle: the original design might have been perfectly logical and following reasonable best practices. It's only after an evolution of backward-compatible changes that a new, more efficient option becomes important. No mistakes were made, but the passage of time still made change valuable.

效率的提高使情況更加複雜。我們希望為資料中心配備經濟高效的計算裝置，特別是提高CPU效率。然而，早期谷歌的演算法和資料結構在現代裝置上效率較低：連結串列或二叉搜尋樹仍能正常工作，但CPU週期與記憶體延遲之間的差距不斷擴大，影響了看起來還像"高效"程式碼。隨著時間的推移，升級到較新硬體的價值會降低，而無需對軟體進行相應的設計更改。向後相容性確保了舊系統仍能正常工作，但這並不能保證舊的優化仍然有用。不願意或無法利用這些機會可能會帶來巨大的成本。像這樣的效率問題尤其微妙：最初的設計可能完全符合邏輯，並遵循合理的最佳實踐。只有在向後相容的變化演變之後，新的、更有效的選擇才變得重要。雖然沒有犯錯誤，但隨著時間的推移，變化仍然是有價值的。

Concerns like those just mentioned are why there are large risks for long-term projects that haven't invested in sustainability. We must be capable of responding to these sorts of issues and taking advantage of these opportunities, regardless of whether they directly affect us or manifest in only the transitive closure of technology we build upon. Change is not inherently good. We shouldn't change just for the sake of change. But we do need to be capable of change. If we allow for that eventual necessity, we should also consider whether to invest in making that capability cheap. As every system administrator knows, it's one thing to know in theory that you can recover from tape, and another to know in practice exactly how to do it and how much it will cost when it becomes necessary. Practice and expertise are great drivers of efficiency and reliability.

像剛才提到的那些擔憂，沒有對可持續性的長期專案進行投入是存在巨大風險。我們必須能夠應對這些問題，並利用好機會，無論它們是否直接影響我們，或者僅僅表現為我們所建立的技術的過渡性封閉中。**變化本質上不是好事**。我們不應該僅僅為了改變而改變。但我們確實需要有能力改變。如果我們考慮到最終的必要性，我們也應該考慮是否加大投入使這種能力變得簡單易用（成本更低）。正如每個系統管理員都知道的那樣，從理論上知道你可以從磁帶恢復是一回事，在實踐中確切地知道如何進行恢復以及在必要時需要花費多少錢是另一回事。實踐和專業知識是效率和可靠性的重要驅動力。

Scale and Efficiency 規模和效率

As noted in the Site Reliability Engineering (SRE) book,[^11] Google's production system as a whole is among the most complex machines created by humankind. The complexity involved in building such a machine and keeping it running smoothly has required countless hours of thought, discussion, and redesign from experts across our organization and around the globe. So, we have already written a book about the complexity of keeping that machine running at that scale.

正如（SRE）這本書所指出的，谷歌的生產系統作為一個整體是人類創造的最複雜的系統之一。構建這樣複雜系統並保持其平穩執行所涉及的複雜性需要我們組織和全球各地的專家進行無數小時的思考、討論和重構。因此，我們已經寫了一本書，講述了保持機器以這種規模執行的複雜性。

Much of this book focuses on the complexity of scale of the organization that produces such a machine, and the processes that we use to keep that machine running over time. Consider again the concept of codebase sustainability: "Your organization's codebase is sustainable when you are able to change all of the things that you ought to change, safely, and can do so for the life of your codebase." Hidden in the discussion of capability is also one of costs: if changing something comes at inordinate cost, it will likely be deferred. If costs grow superlinearly over time, the operation clearly is not scalable.[^12] Eventually, time will take hold and something unexpected will arise that you absolutely must change. When your project doubles in scope and you need to perform that task again, will it be twice as labor intensive? Will you even have the human resources required to address the issue next time?

本書的大部分內容都集中在產生這種系統的組織規模的複雜性，以及我們用來保持系統長期執行的過程。再考慮程式碼庫可持續性的概念："當你能夠安全地改變你應該改變的所有事情，你的組織的程式碼庫是可持續的，並且可以為你的程式碼庫的生命做這樣的事情。"隱藏在能力的討論中也是成本的一個方面：如果改變某事的代價太大，它可能會被推遲。如果成本隨著時間的推移呈超線性增長，運營顯然是不可擴充套件的。最終，時間會佔據主導地位，出現一些意想不到的情況，你必須改變。當你的專案範圍擴大了一倍，並且你需要再次執行該任務時，它會是勞動密集型的兩倍嗎？下次你是否有足夠的人力資源來解決這個問題？

Human costs are not the only finite resource that needs to scale. Just as software itself needs to scale well with traditional resources such as compute, memory, storage, and bandwidth, the development of that software also needs to scale, both in terms of human time involvement and the compute resources that power your development workflow. If the compute cost for your test cluster grows superlinearly, consuming more compute resources per person each quarter, you're on an unsustainable path and need to make changes soon.

人力成本不是唯一需要擴大規模的有限資源。就像軟體本身需要與傳統資源（如計算、記憶體、儲存和頻寬）進行良好的可擴充套件一樣，軟體的開發也需要進行擴充套件，包括人力時間的參與和支援開發工作流程的計算資源。如果測試叢集的計算成本呈超線性增長，每個季度人均消耗更多的計算資源，那麼你的專案就走上了一條不可持續的道路，需要儘快做出改變。

Finally, the most precious asset of a software organization—the codebase itself—also needs to scale. If your build system or version control system scales superlinearly over time, perhaps as a result of growth and increasing changelog history, a point might come at which you simply cannot proceed. Many questions, such as "How long does it take to do a full build?", "How long does it take to pull a fresh copy of the repository?", or "How much will it cost to upgrade to a new language version?" aren't actively monitored and change at a slow pace. They can easily become like the metaphorical boiled frog; it is far too easy for problems to worsen slowly and never manifest as a singular moment of crisis. Only with an organization-wide awareness and commitment to scaling are you likely to keep on top of these issues.

最後，軟體系統最寶貴的資產*程式碼庫*本身也需要擴充套件。如果你的構建系統或版本控制系統隨著時間的推移呈超線性擴充套件，也許是由於內容增長和不斷增加的變更日誌歷史，那麼可能會出現無法持續的情況。許多問題，如"完成完整構建需要多長時間？"、"拉一個新的版本庫需要多長時間？"或"升級到新語言版本需要多少成本？"都沒有受到有效的監管，並且效率變得緩慢。這些問題很容易地變得像溫水煮青蛙；問題很容易慢慢惡化，而不會表現為單一的危機時刻。只有在整個組織範圍內提高意識並致力於擴大規模，才可能保持對這些問題的關注。

Everything your organization relies upon to produce and maintain code should be scalable in terms of overall cost and resource consumption. In particular, everything your organization must do repeatedly should be scalable in terms of human effort. Many common policies don't seem to be scalable in this sense.

你的組織生產和維護程式碼所依賴的一切都應該在總體成本和資源消耗方面具有可擴充套件性。特別是，你的組織必須重複做的每件事都應該在人力方面具有可擴充套件性。從這個意義上講，許多通用策略似乎不具有可擴充套件性。

> [^11]: Beyer, B. et al. Site Reliability Engineering: How Google Runs Production Systems. (Boston: O'Reilly Media,2016).
>
> Beyer, B. et al. Site Reliability Engineering: 谷歌如何執行生產系統。(Boston: O'Reilly Media, 2016).
>
> [^12]: Whenever we use "scalable" in an informal context in this chapter, we mean "sublinear scaling with regard to human interactions."
>
> 在本章中，當我們在非正式語境中使用"可擴充套件性"時，我們的意思是"在人類互動的次線性伸縮性"

Policies That Don't Scale 不可擴充套件的策略

With a little practice, it becomes easier to spot policies with bad scaling properties. Most commonly, these can be identified by considering the work imposed on a single engineer and imagining the organization scaling up by 10 or 100 times. When we are 10 times larger, will we add 10 times more work with which our sample engineer needs to keep up? Does the amount of work our engineer must perform grow as a function of the size of the organization? Does the work scale up with the size of the codebase? If either of these are true, do we have any mechanisms in place to automate or optimize that work? If not, we have scaling problems.

只要稍加練習，就可以更容易地發現具有不可擴充套件的策略。最常見的情況是，可以通過考慮施加在單個設計並想象組織規模擴大10倍或100倍。當我們的規模增大10倍時，我們會增加10倍的工作量，而我們的工程師能跟得上嗎？我們的工程師的工作量是否隨著組織的規模而增長？工作是否隨著程式碼庫的大小而變多？如果這兩種情況都是真實的，我們是否有機制來自動化或優化這項工作？如果沒有，我們就有擴充套件問題。

Consider a traditional approach to deprecation. We discuss deprecation much more in Chapter 15, but the common approach to deprecation serves as a great example of scaling problems. A new Widget has been developed. The decision is made that everyone should use the new one and stop using the old one. To motivate this, project leads say "We'll delete the old Widget on August 15th; make sure you've converted to the new Widget."

考慮傳統的棄用方式。我們在第15章中詳細討論了棄用，但常用的棄用方法是擴充套件問題的一個很好的例子。開發了一個新的小元件。決定是每個人都應該使用新的，停止使用舊的。為了激發這一點，專案負責人

說：“我們將在8月15日刪除舊的小元件；確保你已轉換為新的小元件。”

This type of approach might work in a small software setting but quickly fails as both the depth and breadth of the dependency graph increases. Teams depend on an ever- increasing number of Widgets, and a single build break can affect a growing percentage of the company. Solving these problems in a scalable way means changing the way we do deprecation: instead of pushing migration work to customers, teams can internalize it themselves, with all the economies of scale that provides.

這種方法可能適用於小型軟體專案，但隨著依賴關係圖的深度和廣度的增加，很快就會失敗。團隊依賴越來越多的小部件，單個構建中斷可能會影響公司不斷增長的百分比。以一種可擴充套件的方式解決這些問題，意味著需要改變我們廢棄的方式： 不是將遷移工作推給客戶，團隊可以將其內部消化，並提供所需資源投入。

In 2012, we tried to put a stop to this with rules mitigating churn: infrastructure teams must do the work to move their internal users to new versions themselves or do the update in place, in backward-compatible fashion. This policy, which we've called the “Churn Rule,” scales better: dependent projects are no longer spending progressively greater effort just to keep up. We've also learned that having a dedicated group of experts execute the change scales better than asking for more maintenance effort from every user: experts spend some time learning the whole problem in depth and then apply that expertise to every subproblem. Forcing users to respond to churn means that every affected team does a worse job ramping up, solves their immediate problem, and then throws away that now-useless knowledge. Expertise scales better.

2012年，我們試圖通過降低流失規則來阻止這種情況：**基礎架構團隊必須將內部使用者遷移到新版本，或者以向後相容的方式進行更新**。我們稱之為“流失規則”的這一策略具有更好的擴充套件性：依賴專案不再為了跟上進度而花費更多的精力。我們還瞭解到，有一個專門的專家組來執行變更規模比要求每個使用者付出更多的維護工作要好：專家們花一些時間深入學習整個問題，然後將專業知識應用到每個子問題上。迫使使用者對流失作出反應意味著每個受影響的團隊做了更糟糕的工作，解決了他們眼前的問題，然後扔掉了那些對現在無效的知識。專業知識的擴充套件性更好。

The traditional use of development branches is another example of policy that has built-in scaling problems. An organization might identify that merging large features into trunk has destabilized the product and conclude, “We need tighter controls on when things merge. We should merge less frequently.” This leads quickly to every team or every feature having separate dev branches. Whenever any branch is decided to be “complete,” it is tested and merged into trunk, triggering some potentially expensive work for other engineers still working on their dev branch, in the form of resyncing and testing. Such branch management can be made to work for a small organization juggling 5 to 10 such branches. As the size of an organization (and the number of branches) increases, it quickly becomes apparent that we're paying an ever-increasing amount of overhead to do the same task. We'll need a different approach as we scale up, and we discuss that in Chapter 16.

傳統的開發分支的使用是另一個有內在擴充套件問題的例子。一個組織可能會發現，將大的功能分支合併到主幹中會破壞產品的穩定性，並得出結論：“我們需要對分支的合併時間進行控制，還要降低合併的頻率”。這很快會導致每個團隊或每個功能都有單獨的開發分支。每當任何分支被確定為“完整”時，都會對其進行測試併合併到主幹中，從而引發其他仍在開發分支上工作的工程師以重新同步和測試，造成巨大的工作量。這樣的分支機構管理模式可以應用在小型組織裡，管理5到10個這樣的分支機構。隨著一個組織的規模（以及分支機構的數量）的增加，我們很快就會發現，為了完成同樣的任務，我們付出越來越多的管理成本。隨著規模的擴大，我們需要一種不同的方法，我們將在第16章中對此進行討論。

Policies That Scale Well 規模化策略

What sorts of policies result in better costs as the organization grows? Or, better still, what sorts of policies can we put in place that provide superlinear value as the organization grows?

隨著公司的發展，什麼樣的策略會帶來更低的成本？或者，最好是，隨著組織化的發展，我們可以制定什麼樣的策略來提供超高的價值？

One of our favorite internal policies is a great enabler of infrastructure teams, protecting their ability to make infrastructure changes safely. “If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn’t surfaced by tests in our Continuous Integration (CI) system, it is not the fault of the infrastructure change.” More colloquially, this is phrased as “If you liked it, you should have put a CI test on it,” which we call “The Beyoncé Rule.”[^13] From a scaling perspective, the Beyoncé Rule implies that complicated, one-off bespoke tests that aren’t triggered by our common CI system do not count. Without this, an engineer on an infrastructure team could conceivably need to track down every team with any affected code and ask them how to run their tests. We could do that when there were a hundred engineers. We definitely cannot afford to do that anymore.

我們最喜歡的內部策略之一是為基礎架構團隊提供強大的支援，維護他們安全地進行基礎措施更改的能力。“如果一個產品由於基礎架構更改而出現停機或其他問題，但我們的持續整合（CI）系統中的測試沒有發現問題，這不是基礎架構變更的錯。”更通俗地說，這是“如果你喜歡它，你應該對它進行CI測試”，我們稱之為“碧昂斯規則。”從可伸縮性的角度來看，碧昂斯規則意味著複雜的、一次性的定製測試（不是由我們的通用CI系統觸發的）不算數。如果沒有這一點，基礎架構團隊的工程師需要跟蹤每個有任何受影響程式碼的團隊，問他們如何進行測試。當有一百個工程師的時候，我們可以這樣做。我們絕對不能這樣做。

We’ve found that expertise and shared communication forums offer great value as an organization scales. As engineers discuss and answer questions in shared forums, knowledge tends to spread. New experts grow. If you have a hundred engineers writing Java, a single friendly and helpful Java expert willing to answer questions will soon produce a hundred engineers writing better Java code. Knowledge is viral, experts are carriers, and there’s a lot to be said for the value of clearing away the common stumbling blocks for your engineers. We cover this in greater detail in Chapter 3.

我們發現，隨著組織規模的擴大，專業知識和共享交流論壇提供了巨大的價值。隨著工程師在共享論壇中討論和回答問題，知識往往會傳播。新的專家人數不斷增加。如果你有100名工程師編寫Java，那麼一位願意回答問題的友好且樂於助人的Java專家很快就會產生一個數百名工程師編寫更好的Java程式碼。知識是病毒，專家是載體，掃除工程師常見的絆腳石是非常有價值的。我們將在第3章更詳細地介紹這一點。

> [^13]: This is a reference to the popular song “Single Ladies,” which includes the refrain “If you liked it then you shoulda put a ring on it.”
>
> 這是指流行歌曲《單身女士》，其中包括 "如果你喜歡它，你就應該給它戴上戒指。

Example: Compiler Upgrade 示例：編譯器升級

Consider the daunting task of upgrading your compiler. Theoretically, a compiler upgrade should be cheap given how much effort languages take to be backward compatible, but how cheap of an operation is it in practice? If you’ve never done such an upgrade before, how would you evaluate whether your codebase is compatible with that change?

考慮升級編譯器的艱鉅任務。從理論上講，編譯器的升級應該是簡單的，因為語言需要多少工作才能向後相容，但是在實際操作中又有多簡單呢？如果你以前從未做過這樣的升級，你將如何評價你的程式碼庫是否與相容升級？

In our experience, language and compiler upgrades are subtle and difficult tasks even when they are broadly expected to be backward compatible. A compiler upgrade will almost always result in minor changes to behavior: fixing miscompilations, tweaking optimizations, or potentially changing the results of anything that was previously undefined. How would you evaluate the correctness of your entire codebase against all of these potential outcomes?

根據我們的經驗，語言和編譯器升級是微妙而困難的任務，即使人們普遍認為它們是向後相容的。編譯器升級幾乎總是會導致編譯的微小變化：修復錯誤編譯、調整優化，或者潛在地改變任何以前未定義的結果。你將如何針對所有這些潛在的結果來評估你整個程式碼庫的正確性？

The most storied compiler upgrade in Google's history took place all the way back in 2006. At that point, we had been operating for a few years and had several thousand engineers on staff. We hadn't updated compilers in about five years. Most of our engineers had no experience with a compiler change. Most of our code had been exposed to only a single compiler version. It was a difficult and painful task for a team of (mostly) volunteers, which eventually became a matter of finding shortcuts and simplifications in order to work around upstream compiler and language changes that we didn't know how to adopt.[^14] In the end, the 2006 compiler upgrade was extremely painful. Many Hyrum's Law problems, big and small, had crept into the codebase and served to deepen our dependency on a particular compiler version. Breaking those implicit dependencies was painful. The engineers in question were taking a risk: we didn't have the Beyoncé Rule yet, nor did we have a pervasive CI system, so it was difficult to know the impact of the change ahead of time or be sure they wouldn't be blamed for regressions.

谷歌歷史上最具傳奇色彩的編譯器升級發生在2006年。當時，我們已經執行了幾年，擁有數千名工程師。我們大約有五年沒有升級過編譯器。我們的大多數工程師都沒有升級編譯器的經驗。我們的大部分程式碼只針對在單一編譯器版本。對於一個由（大部分）志願者組成的團隊來說，這是一項艱難而痛苦的任務，最終變成了尋找捷徑和簡化的問題，以便繞過我們不知道如何採用的上游編譯器和語言變化。最後，2006年的編譯器升級過程非常痛苦。許多海勒姆定律問題，無論大小，都潛入了程式碼庫，加深了我們對特定編譯器版本的依賴。打破這些隱式依賴性是痛苦的。相關工程師正在冒風險：我們還沒有碧昂斯規則，也沒有通用的CI系統，因此很難提前知道更改的影響，或者確保他們不會因回退而受到指責。

This story isn't at all unusual. Engineers at many companies can tell a similar story about a painful upgrade. What is unusual is that we recognized after the fact that the task had been painful and began focusing on technology and organizational changes to overcome the scaling problems and turn scale to our advantage: automation (so that a single human can do more), consolidation/consistency (so that low-level changes have a limited problem scope), and expertise (so that a few humans can do more).

這個故事一點也不稀奇。許多公司的工程師都可以講述一個關於痛苦升級的故事。不同的是，我們在經歷了痛苦的任務之後認識到了這一點，並開始關注技術和組織變革，以克服規模問題，並將規模轉變為我們的優勢：自動化（這樣一個人就可以做到更多）、整合/一致性（這樣低階別的更改影響有限的問題範圍）和專業知識（以便少數人就可以做得更多）。

The more frequently you change your infrastructure, the easier it becomes to do so. We have found that most of the time, when code is updated as part of something like a compiler upgrade, it becomes less brittle and easier to upgrade in the future. In an ecosystem in which most code has gone through several upgrades, it stops depending on the nuances of the underlying implementation; instead, it depends on the actual abstraction guaranteed by the language or OS. Regardless of what exactly you are upgrading, expect the first upgrade for a codebase to be significantly more expensive than later upgrades, even controlling for other factors.

你更改基礎設施的頻率越高，更改就越容易。我們發現，在大多數情況下，當程式碼作為編譯器升級的一部分進行更新時，它會變得沒那麼脆弱，將來更容易升級。大多數程式碼都經歷了幾次升級的一個系統中，它的停止取決於底層實現的細微差別。相反，它取決於語言或作業系統所保證的抽象。無論你升級的是什麼，程式碼庫的第一次升級都比以後的升級要複雜得多，甚至可以控制其他因素。

> [^14]: Specifically, interfaces from the C++ standard library needed to be referred to in namespace std, and an optimization change for std::string turned out to be a significant pessimization for our usage, thus requiring some additional workarounds.
>
> 具體來說，來自C++標準庫的介面需要在名稱空間std中被引用，而針對std::string的優化改變對我們的使用來說是一個重大的減值，因此需要一些額外的解決方法。

Through this and other experiences, we've discovered many factors that affect the flexibility of a codebase:

- *Expertise*
  We know how to do this; for some languages, we've now done hundreds of compiler upgrades across many platforms.
- *Stability*
  There is less change between releases because we adopt releases more regularly; for some languages, we're now deploying compiler upgrades every week or two.
- *Conformity*
  There is less code that hasn't been through an upgrade already, again because we are upgrading regularly.
- *Familiarity*
  Because we do this regularly enough, we can spot redundancies in the process of performing an upgrade and attempt to automate. This overlaps significantly with SRE views on toil.[^15]
- *Policy*
  We have processes and policies like the Beyoncé Rule. The net effect of these processes is that upgrades remain feasible because infrastructure teams do not need to worry about every unknown usage, only the ones that are visible in our CI systems.

通過這些和其他經驗，我們發現了許多影響程式碼庫靈活性的因素：

- *專業知識* 我們知道如何做到這一點；對於某些語言，我們現在已經在許多平臺上進行了數百次編譯器升級。
- *穩定性*
  版本之間的更改更少，因為我們更有規律的採用版本；對於某些語言，我們現在每一到兩週進行一次編譯器升級部署。
- *一致性*
  沒有經過升級的程式碼更少了，這也是因為我們正在定期升級。
- *熟悉*
  因為我們經常這樣做，所以我們可以在執行升級的過程中發現冗餘並嘗試自動化。這是與SRE觀點一致的地方。
- *策略*
  我們有類似碧昂斯規則的流程和策略。這些程式的淨效果是，升級仍然是可行的，因為基礎設施團隊不需要擔心每一個未知的使用，只需要擔心我們的CI系統中常規的使用。

The underlying lesson is not about the frequency or difficulty of compiler upgrades, but that as soon as we became aware that compiler upgrade tasks were necessary, we found ways to make sure to perform those

tasks with a constant number of engineers, even as the codebase grew.[^16] If we had instead decided that the task was too expensive and should be avoided in the future, we might still be using a decade-old compiler version. We would be paying perhaps 25% extra for computational resources as a result of missed optimization opportunities. Our central infrastructure could be vulnerable to significant security risks given that a 2006-era compiler is certainly not helping to mitigate speculative execution vulnerabilities. Stagnation is an option, but often not a wise one.

潛在的教訓不是關於編譯器升級的頻率或難度，而是一旦我們意識到編譯器升級任務是必要的，我們就找到了方法，確保在程式碼庫增長的情況下，由固定數量的工程師執行這些任務。如果我們認為任務成本太高，應該學會避免，我們可以仍然使用十年前的編譯器版本。由於錯過了優化機會，我們需要額外支付25%的計算資源。考慮到2006年的編譯器對緩解推測性執行漏洞沒有效果，我們的中央基礎設施可能會面臨重大的安全風險，這不是一個明智的選擇。

> [^15]: Beyer et al. Site Reliability Engineering: How Google Runs Production Systems, Chapter 5, “Eliminating Toil.”
>
> Beyer等人，《SRE：Google運維解密》,第五章 減少瑣事。
>
> [^16]: In our experience, an average software engineer (SWE) produces a pretty constant number of lines of code per unit time. For a fixed SWE population, a codebase grows linearly—proportional to the count of SWE- months over time. If your tasks require effort that scales with lines of code, that’s concerning.
>
> 根據我們的經驗，平均軟體工程師（SWE）每單位時間產生相當恆定的程式碼行數。對於固定的SWE總體，隨著時間的推移，程式碼庫的增長與SWE月數成線性比例。如果你的任務需要與程式碼行數成比例的工作，這是值得關注的。

Shifting Left 左移

One of the broad truths we’ve seen to be true is the idea that finding problems earlier in the developer workflow usually reduces costs. Consider a timeline of the developer workflow for a feature that progresses from left to right, starting from conception and design, progressing through implementation, review, testing, commit, canary, and eventual production deployment. Shifting problem detection to the “left” earlier on this timeline makes it cheaper to fix than waiting longer, as shown in Figure 1-2.

我們看到的一個普遍真理是，在開發人員的工作流程中發現的問題，通常可以降低成本。考慮開發人員工作流程的時間表，從左到右，從概念和設計開始，通過實施、評審、測試、提交、金絲雀和最終的生產部署來進行。在此時間線之前，將問題發現轉移到“左側”會使修問題解決成本更低，如圖1-2所示。

This term seems to have originated from arguments that security mustn’t be deferred until the end of the development process, with requisite calls to “shift left on security.” The argument in this case is relatively simple: if a security problem is discovered only after your product has gone to production, you have a very expensive problem. If it is caught before deploying to production, it may still take a lot of work to identify and remedy the problem, but it’s cheaper. If you can catch it before the original developer commits the flaw to version control, it’s even cheaper: they already have an understanding of the feature; revising according to new security constraints is cheaper than committing and forcing someone else to triage it and fix it.

這個術語似乎源一種觀點，即安全問題不能推遲到開發過程的最後階段，必須要求“在安全上向左轉移”。這種情況下的論點相對簡單：如果安全問題是在產品投入生產後才發現的，修復的成本就非常高。如果在部署到生產之前就發現了安全問題，那也需要花費大量的工作來檢測和修復問題，但成本更低些。如果你能夠在

最初的開發之前發現安全問題，將缺陷提交到版本控制就被發現，修復的成本更低：他們已經瞭解該功能；根據新的安全約束規範進行開發，要比提交程式碼後再讓其他人分類標識並修復它更簡單。

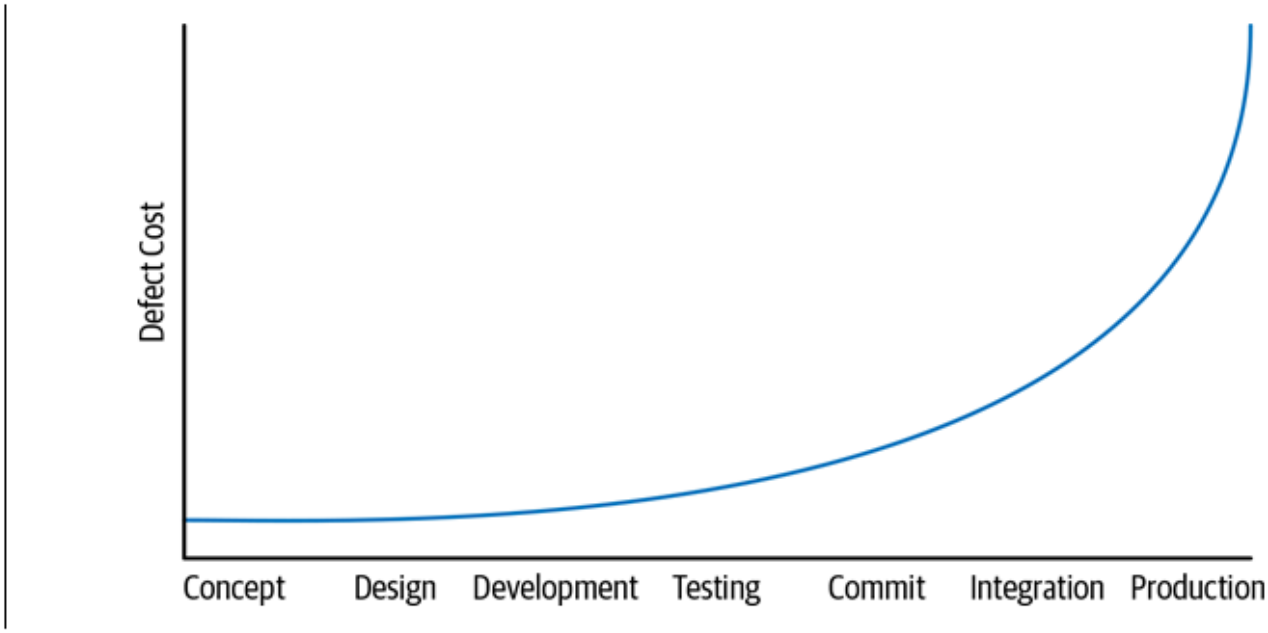

Figure 1-2. Timeline of the developer workflow

The same basic pattern emerges many times in this book. Bugs that are caught by static analysis and code review before they are committed are much cheaper than bugs that make it to production. Providing tools and practices that highlight quality, reliability, and security early in the development process is a common goal for many of our infrastructure teams. No single process or tool needs to be perfect, so we can assume a defense-in-depth approach, hopefully catching as many defects on the left side of the graph as possible.

同樣的基本模式在本書中多次出現。在提交之前通過靜態分析和程式碼審查發現的bug要比投入生產的bug成本更低。在開發過程的早期提供高質量、可靠性和安全性的工具和實踐是我們許多基礎架構團隊的共同目標。沒有一個過程或工具是完美的，所以我們可以採取縱深防禦的方法，希望儘早抓住圖表左側的缺陷。

Trade-offs and Costs 權衡和成本

If we understand how to program, understand the lifetime of the software we're maintaining, and understand how to maintain it as we scale up with more engineers producing and maintaining new features, all that is left is to make good decisions. This seems obvious: in software engineering, as in life, good choices lead to good outcomes. However, the ramifications of this observation are easily overlooked. Within Google, there is a strong distaste for "because I said so." It is important for there to be a decider for any topic and clear escalation paths when decisions seem to be wrong, but the goal is consensus, not unanimity. It's fine and expected to see some instances of "I don't agree with your metrics/valuation, but I see how you can come to that conclusion." Inherent in all of this is the idea that there needs to be a reason for everything; "just because," "because I said so," or "because everyone else does it this way" are places where bad decisions lurk. Whenever it is efficient to do so, we should be able to explain our work when deciding between the general costs for two engineering options.

如果我們瞭解如何程式設計，瞭解我們所維護的軟體的生命週期，並且隨著在我們隨著更多的工程師一起開發和維護新功能，瞭解擴大規模時如何運維它，那麼剩下的就是做出正確的決策。這是顯而易見的：在軟體工程中，如同生活一樣，好的選擇會帶來好的結果。然而，這一觀點很容易被忽視。在谷歌內部，人們對"因為我這麼說了"有反對的意見。重要的是，任何議題都要有一個決策者，當決策是錯誤的時候，要有明確的改

進路徑，但目標是共識，而不是一致。看到一些 "我不同意你的衡量標準/評價，但我知道你是如何得出這個結論的 "的情況是沒有問題的，也是可以預期的。所有這一切的內在想法是，每件事都需要一個理由；“僅僅因為”、“因為我這麼說”或“因為其他人都這樣做”是潛在錯誤的決策。 只要這樣做是有效的，在決定兩個工程方案的一般成本時，我們應該能夠解釋清楚。

What do we mean by cost? We are not only talking about dollars here. “Cost” roughly translates to effort and can involve any or all of these factors:

- Financial costs (e.g., money)
- Resource costs (e.g., CPU time)
- Personnel costs (e.g., engineering effort)
- Transaction costs (e.g., what does it cost to take action?)
- Opportunity costs (e.g., what does it cost to not take action?)
- Societal costs (e.g., what impact will this choice have on society at large?)

我們所說的成本是什麼呢？我們這裡不僅僅是指金錢。“成本”大致可以轉化為努力的方向，可以包括以下任何或所有因素：

- 財務成本（如金錢）
- 資源成本（如CPU時間）
- 人員成本（例如，工作量）
- 交易成本（例如，採取行動的成本是多少？）
- 機會成本（例如，不採取行動的成本是多少？）
- 社會成本（例如，這個選擇將對整個社會產生什麼影響？）

Historically, it’s been particularly easy to ignore the question of societal costs. However, Google and other large tech companies can now credibly deploy products with billions of users. In many cases, these products are a clear net benefit, but when we’re operating at such a scale, even small discrepancies in usability, accessibility, fairness, or potential for abuse are magnified, often to the detriment of groups that are already marginalized. Software pervades so many aspects of society and culture; therefore, it is wise for us to be aware of both the good and the bad that we enable when making product and technical decisions. We discuss this much more in Chapter 4.

從歷史上看，忽視社會成本的問題尤其容易出現。然而，谷歌和其他大型科技公司現在可以可靠地部署擁有數十億使用者的產品。在許多情況下，這些產品是高淨效益的，但當我們以這樣的規模運營時，即使在可用性、可訪問性和公平性方面或潛在的濫用方面的微小差異也會被放大，往往對邊緣化的群體產生不利影響。軟體滲透到社會和文化的各個方面；因此，明智的做法是，在做出產品和技術決策時，我們要意識到我們所能帶來的好處和壞處。我們將在第4章對此進行更多討論。

In addition to the aforementioned costs (or our estimate of them), there are biases: status quo bias, loss aversion, and others. When we evaluate cost, we need to keep all of the previously listed costs in mind: the health of an organization isn’t just whether there is money in the bank, it’s also whether its members are feeling valued and productive. In highly creative and lucrative fields like software engineering, financial cost is usually not the limiting factor—personnel cost usually is. Efficiency gains from keeping engineers happy, focused, and engaged can easily dominate other factors, simply because focus and productivity are so variable, and a 10-to-20% difference is easy to imagine.

除了上述的成本（或我們對其的估計），還有一些偏差：維持現狀偏差（個體在決策時，傾向於不作為、維持當前的或者以前的決策的一種現象。這一定義揭示個體在決策時偏好事件當前的狀態，而且不願意採取行

動來改變這一狀態，當面對一系列決策選項時，傾向於選擇現狀選項），損失厭惡偏差（人們面對同樣的損失和收益時感到損失對情緒影響更大）等。當我們評估成本時，我們需要牢記之前列出的所有成本：一個組織的健康不僅僅是銀行裡是否有錢，還包括其成員是否感到有價值和有成就感。在軟體等高度創新和利潤豐厚的領域在工程設計中，財務成本通常不是限制因素，而人力資源是。保持工程師的快樂、專注和參與所帶來的效率提升會成為主導因素，僅僅是因為專注力和生產力變化大，會有10-20%的差異很容易想象。

Example: Markers 示例：標記

In many organizations, whiteboard markers are treated as precious goods. They are tightly controlled and always in short supply. Invariably, half of the markers at any given whiteboard are dry and unusable. How often have you been in a meeting that was disrupted by lack of a working marker? How often have you had your train of thought derailed by a marker running out? How often have all the markers just gone missing, presumably because some other team ran out of markers and had to abscond with yours? All for a product that costs less than a dollar.

在許多組織中，白板記號筆被視為貴重物品。它們受到嚴格的控制，而且總是供不應求。在任何的白板上，都有一半的記號筆是乾的，無法使用。你有多少次因為沒有一個好用的記號筆而中斷會議程序？多少次因為記號筆水用完而打斷思考？多少次所有的記號筆都不見了，大概是因為其他團隊的記號筆用完了，不得不拿走你的記號筆？所有這些都是因為一個價格不到一美元的產品。

Google tends to have unlocked closets full of office supplies, including whiteboard markers, in most work areas. With a moment’s notice it is easy to grab dozens of markers in a variety of colors. Somewhere along the line we made an explicit trade- off: it is far more important to optimize for obstacle-free brainstorming than to protect against someone wandering off with a bunch of markers.

谷歌往往在大多數工作區域都有未上鎖的櫃子，裡面裝滿了辦公用品，包括記號筆。只要稍加註意，就可以很容易地拿到各種顏色的幾十支記號筆。在某種程度上，我們做了一個明確的權衡：優化無障礙的頭腦風暴遠比防止有人拿著一堆記號筆亂跑要重要得多。

We aim to have the same level of eyes-open and explicit weighing of the cost/benefit trade-offs involved for everything we do, from office supplies and employee perks through day-to-day experience for developers to how to provision and run global- scale services. We often say, “Google is a data-driven culture.” In fact, that’s a simplification: even when there isn’t *data*, there might still be *evidence*, *precedent*, and *argument*. Making good engineering decisions is all about weighing all of the available inputs and making informed decisions about the trade-offs. Sometimes, those decisions are based on instinct or accepted best practice, but only after we have exhausted approaches that try to measure or estimate the true underlying costs.

我們的目標是對我們所做的每件事都有同樣程度的關注和明確的成本/收益權衡，從辦公用品和員工津貼到開發者的日常體驗，再到如何提供和執行全球規模的服務。我們經常說，“谷歌是一家資料驅動的公司。”事實上，這很簡單：即使沒有資料，也會有證據、先例和論據。做出好的工程決策就是權衡所有可用的輸入，並就權衡做出明智的決策。有時，這些決策是基於本能或公認的最佳實踐，但僅是一種假設之後，我們用盡了各種方法來衡量或估計真正的潛在成本。

In the end, decisions in an engineering group should come down to very few things:

- We are doing this because we must (legal requirements, customer requirements).
- We are doing this because it is the best option (as determined by some appropriate decider) we can see at the time, based on current evidence.

最後，工程團隊的決策應該歸結為幾件事：

- 我們這樣做是因為我們必須這麼做（法律要求、客戶要求）。
- 我們之所以這樣做，是因為根據當前證據，這是我們當時能看到的最佳選擇（由一些適當的決策者決策）。

Decisions should not be “We are doing this because I said so.”[^17]

決策不應該是“我們這樣做是因為我這麼說。”[^17]

> [^17]: This is not to say that decisions need to be made unanimously, or even with broad consensus; in the end, someone must be the decider. This is primarily a statement of how the decision-making process should flow for whoever is actually responsible for the decision.
>
> 這並不是說決策需要一致做出，甚至需要有廣泛的共識；最終，必須有人成為決策者。這主要是說明決策過程應該如何為實際負責決策的人進行。

Inputs to Decision Making 對決策的輸入

When we are weighing data, we find two common scenarios:

- All of the quantities involved are measurable or can at least be estimated. This usually means that we’re evaluating trade-offs between CPU and network, or dollars and RAM, or considering whether to spend two weeks of engineer-time in order to save N CPUs across our datacenters.
- Some of the quantities are subtle, or we don’t know how to measure them. Sometimes this manifests as “We don’t know how much engineer-time this will take.” Sometimes it is even more nebulous: how do you measure the engineering cost of a poorly designed API? Or the societal impact of a product choice?

當我們權衡資料時，我們發現兩種常見情況：

- 所有涉及的數量都是可測量的或至少可以預估的。這通常意味著我們正在評估CPU和網路、美金和RAM之間的權衡，或者考慮是否花費兩週的工作量，以便在我們的資料中心節省N個CPU。
- 有些數量是微妙的，或者我們不知道如何衡量。有時這表現為“我們不知道這需要多少工作量”。有時甚至更模糊：如何衡量設計拙劣的API的工程成本？或產品導致的社會影響？

There is little reason to be deficient on the first type of decision. Any software engineering organization can and should track the current cost for compute resources, engineer-hours, and other quantities you interact with regularly. Even if you don’t want to publicize to your organization the exact dollar amounts, you can still produce a conversion table: this many CPUs cost the same as this much RAM or this much network bandwidth.

在第一類決策上沒有什麼理由存在缺陷。任何軟體工程的組織都可以並且應該跟進當前的計算資源成本、工程師工作量以及你經常接觸的其他成本。即使你不想向你的組織公佈確切的金額，你仍然可以制定一份版本表：這麼多CPU的成本與這麼多RAM或這麼多網路頻寬。

With an agreed-upon conversion table in hand, every engineer can do their own analysis. “If I spend two weeks changing this linked-list into a higher-performance structure, I’m going to use five gibibytes more production RAM but save two thousand CPUs. Should I do it?” Not only does this question depend upon the relative cost of RAM and CPUs, but also on personnel costs (two weeks of support for a software engineer) and opportunity costs (what else could that engineer produce in two weeks?).

有了一個協定的轉換表，每個工程師都可以自己進行分析。“如果我花兩週的時間將這個連結串列轉換成一個更高效能的資料結構，我將多使用5Gb的RAM，但節省兩千個CPU。我應該這樣做嗎？”這個問題不僅取決於RAM和CPU的相對成本，還取決於人員成本（對軟體工程師的兩週支援）和機會成本（該工程師在兩週內還能生產什麼？）。

For the second type of decision, there is no easy answer. We rely on experience, leadership, and precedent to negotiate these issues. We're investing in research to help us quantify the hard-to-quantify (see Chapter 7). However, the best broad suggestion that we have is to be aware that not everything is measurable or predictable and to attempt to treat such decisions with the same priority and greater care. They are often just as important, but more difficult to manage.

對於第二類決策，沒有簡單的答案。我們依靠經驗、領導和先例來協商這些問題。我們正在投入研究，以幫助我們量化難以量化的問題（見第7章）然而，我們所擁有的最好的廣泛建議是，意識到並非所有的事情都是可衡量或可預測的，並嘗試以同樣的優先權和更大的謹慎對待此類決策。它們往往同樣重要，但更難管理。

Example: Distributed Builds 示例：分散式構建

Consider your build. According to completely unscientific Twitter polling, something like 60 to 70% of developers build locally, even with today's large, complicated builds. This leads directly to nonjokes as illustrated by this "Compiling" comic—how much productive time in your organization is lost waiting for a build? Compare that to the cost to run something like distcc for a small group. Or, how much does it cost to run a small build farm for a large group? How many weeks/months does it take for those costs to be a net win?

考慮到你的構建。根據不可靠的推特投票結果顯示，大約有60到70%的開發者在本地構建，即使是今天的大型、複雜的構建。才有了這樣的笑話，如“編譯”漫畫所示。你的組織中有多少時間被浪費在等待構建上？將其與為一個小團隊執行類似distcc的成本進行比較。或者，為一個大團隊執行一個小構建場需要多少成本？這些成本需要多少周/月才能成為一個淨收益？

Back in the mid-2000s, Google relied purely on a local build system: you checked out code and you compiled it locally. We had massive local machines in some cases (you could build Maps on your desktop!), but compilation times became longer and longer as the codebase grew. Unsurprisingly, we incurred increasing overhead in personnel costs due to lost time, as well as increased resource costs for larger and more powerful local machines, and so on. These resource costs were particularly troublesome: of course we want people to have as fast a build as possible, but most of the time, a high- performance desktop development machine will sit idle. This doesn't feel like the proper way to invest those resources.

早在2000年代中期，谷歌就完全依賴於本地構建系統：你切出程式碼，然後在本地編譯。在某些情況下，我們有大量的本地機器（你可以在桌面電腦上構建地圖！），但隨著程式碼庫的增長，編譯時間變得越來越長。不出所料，由於時間的浪費，我們的人員成本增加，以及更大、更強大的本地機器的資源成本增加等等。這些資源成本特別麻煩：當然，我們希望人們有一個儘可能快的構建，但大多數時候，一個高效能的桌面開發機器將被閒置。這感覺不像是投資這些資源的正確方式。

Eventually, Google developed its own distributed build system. Development of this system incurred a cost, of course: it took engineers time to develop, it took more engineer time to change everyone's habits and workflow and learn the new system, and of course it cost additional computational resources. But the overall savings were clearly worth it: builds became faster, engineer-time was recouped, and hardware investment could focus on managed shared infrastructure (in actuality, a subset of our production fleet)

rather than ever-more-powerful desktop machines. Chapter 18 goes into more of the details on our approach to distributed builds and the relevant trade-offs.

最終，谷歌開發了自己的分散式構建系統。開發這個系統當然要付出代價：工程師花費了時間，工程師花更多的時間來改變每個人的習慣和工作流程，學習新系統，當然還需要額外的計算資源。但總體節約顯然值得我去做： 構建速度變快了，工程師的時間被節約了，硬體投資可以集中在管理的共享基礎設施上（實際上是我們生產機群的一個子集），而不是日益強大的桌面機。第18章詳細介紹了我們的分散式構建方法和相關權衡。

So, we built a new system, deployed it to production, and sped up everyone's build. Is that the happy ending to the story? Not quite: providing a distributed build system made massive improvements to engineer productivity, but as time went on, the distributed builds themselves became bloated. What was constrained in the previous case by individual engineers (because they had a vested interest in keeping their local builds as fast as possible) was unconstrained within a distributed build system. Bloated or unnecessary dependencies in the build graph became all too common. When everyone directly felt the pain of a nonoptimal build and was incentivized to be vigilant, incentives were better aligned. By removing those incentives and hiding bloated dependencies in a parallel distributed build, we created a situation in which consumption could run rampant, and almost nobody was incentivized to keep an eye on build bloat. This is reminiscent of Jevons Paradox: consumption of a resource may increase as a response to greater efficiency in its use.

因此，我們構建了一個新系統，將其部署到生產環境中，並加快了每個人的構建速度。這就是故事的圓滿結局嗎？不完全是這樣：提供分散式構建系統極大地提高了工程師的工作效率，但隨著時間的推移，分散式構建本身變得臃腫起來。在以前的情況下，單個工程師受到的限制（因為他們盡最大可能保持本地構建的速度）在分散式構建系統中是不受限制的。構建圖中的臃腫或不必要的依賴關係變得非常普遍。當每個人都直接感受到非最佳構建的痛苦，並被要求去保持警惕時，激勵措施會更好地協調一致。通過取消這些激勵措施，並將臃腫的依賴關係隱藏在並行的分散式構建中，我們創造了一種情況，在這種情況下，消耗可能猖獗，而且幾乎沒有人被要求去關注構建的臃腫。這讓人想起傑文斯悖論（Jevons Paradox）：一種資源的消耗可能會隨著使用效率的提高而增加。

Overall, the saved costs associated with adding a distributed build system far, far outweighed the negative costs associated with its construction and maintenance. But, as we saw with increased consumption, we did not foresee all of these costs. Having blazed ahead, we found ourselves in a situation in which we needed to reconceptualize the goals and constraints of the system and our usage, identify best practices (small dependencies, machine-management of dependencies), and fund the tooling and maintenance for the new ecosystem. Even a relatively simple trade-off of the form "We'll spend $$$s for compute resources to recoup engineer time" had unforeseen downstream effects.

總的來說，與新增分散式構建系統相關的節省成本遠遠超過了與其構建和維護相關的負成本。但是，正如我們看到的消耗增加，我們並沒有以前預見到這些成本。勇往直前之後，我們發現自己處於這樣一種境地：我們需要重新認識系統的目標和約束以及我們的使用方式，確定最佳實踐（小型依賴項、依賴項的機器管理），併為新生態系統的工具和維護提供資金。即使是相對簡單的 "我們花美元購買計算資源以收回工程師時間 "的權衡，也會產生不可預見的下游影響。

Example: Deciding Between Time and Scale 示例：在時間和規模之間做決定

Much of the time, our major themes of time and scale overlap and work in conjunction. A policy like the Beyoncé Rule scales well and helps us maintain things over time. A change to an OS interface might require

many small refactorings to adapt to, but most of those changes will scale well because they are of a similar form: the OS change doesn't manifest differently for every caller and every project.

很多時候，我們關於時間和規模的主題相互重合，相互影響。符合碧昂斯規則策略具備可擴充套件性，並幫助我們長期維護事物。對作業系統介面的更改需要許多小的重構來適應，但這些更改中的大多數都可以很好地擴充套件，因為它們具有相似的形式： 作業系統的變化對每個呼叫者和每個專案都沒有不同的表現。

Occasionally time and scale come into conflict, and nowhere so clearly as in the basic question: should we add a dependency or fork/reimplement it to better suit our local needs?

有時時間和規模會發生衝突，而且沒有什麼地方能像在基本問題中那麼明顯：我們應該新增一個依賴性，還是分支/重新實現它，來更好地滿足我們的需求？

This question can arise at many levels of the software stack because it is regularly the case that a bespoke solution customized for your narrow problem space may outperform the general utility solution that needs to handle all possibilities. By forking or reimplementing utility code and customizing it for your narrow domain, you can add new features with greater ease, or optimize with greater certainty, regardless of whether we are talking about a microservice, an in-memory cache, a compression routine, or anything else in our software ecosystem. Perhaps more important, the control you gain from such a fork isolates you from changes in your underlying dependencies: those changes aren't dictated by another team or third-party provider. You are in control of how and when to react to the passage of time and necessity to change.

這個問題可能出現在軟體棧（解決方案棧）各個層面，通常情況下，為特定問題定製解決方案優於需要處理所有問題的通用解決方案。通過分支或重新實現程式程式碼，併為特定問題定製它，你可以更便捷地新增新功能，或更確定地進行優化，無論我們談論的是微服務、記憶體快取、壓縮程式還是軟體生態系統中的其他內容。更重要的是，你從這樣一個分支中獲得的控制將你與基礎依賴項中的變更隔離開：這些變化並不是由另一個團隊或第三方供應商所決定的。你隨時決定如何對時間的推移和變化的作出必要地反應。

> [一條微博引發的思考—再談“Software Stack”之“軟體棧”譯法！](https://www.ituring.com.cn/article/1144)
> 軟體棧（Software Stack），是指為了實現某種完整功能解決方案（例如某款產品或服務）所需的一套軟體子系統或元件。

On the other hand, if every developer forks everything used in their software project instead of reusing what exists, scalability suffers alongside sustainability. Reacting to a security issue in an underlying library is no longer a matter of updating a single dependency and its users: it is now a matter of identifying every vulnerable fork of that dependency and the users of those forks.

另一方面，如果每個開發人員都將他們的軟體專案中使用的元件是多樣化，而不是複用現有的元件，那麼可擴充套件性和可持續性都會受到影響。對底層庫中的安全問題作出反應不再是更新單個依賴項及其使用者的問題：現在要做的是識別該依賴關係的每一個易受攻擊的分支以及使用這個分支的使用者。

As with most software engineering decisions, there isn't a one-size-fits-all answer to this situation. If your project life span is short, forks are less risky. If the fork in question is provably limited in scope, that helps, as well—avoid forks for interfaces that could operate across time or project-time boundaries (data structures, serialization formats, networking protocols). Consistency has great value, but generality comes with its own costs, and you can often win by doing your own thing—if you do it carefully.

與大多數軟體工程決策一樣，對於這種情況並沒有一個一刀切的答案。如果你的專案生命週期很短，那麼fork的風險較小。 如果有問題的分支被證明是範圍有限的，那是有幫助的，同時也要避免分支那些可能跨越時間段或專案時間界限的介面（資料結構、序列化格式、網路協議）。一致性有很大的價值，但通用性也有其自身的成本，你往往可以通過做自己的事情來贏得勝利——如果你仔細做的話。

Revisiting Decisions, Making Mistakes 重審決策，標記錯誤

One of the unsung benefits of committing to a data-driven culture is the combined ability and necessity of admitting to mistakes. A decision will be made at some point, based on the available data—hopefully based on good data and only a few assumptions, but implicitly based on currently available data. As new data comes in, contexts change, or assumptions are dispelled, it might become clear that a decision was in error or that it made sense at the time but no longer does. This is particularly critical for a long-lived organization: time doesn't only trigger changes in technical dependencies and software systems, but in data used to drive decisions.

致力於資料驅動文化的一個不明顯的好處是承認錯誤的能力和必要性相結合。在某個時候，將根據現有的資料做出決定——希望是基於準確的資料和僅有的幾個假設，但隱含的是基於目前可用的資料。隨著新資料的出現，環境的變化，或假設的不成了，可能會發現某個決策是錯誤的，或在當時是有意義的，但現在已經沒有意義了。這對於一個長期存在的組織來說尤其重要：時間不僅會觸發技術依賴和軟體系統的變化，還會觸發用於驅動決策的資料的變化。

We believe strongly in data informing decisions, but we recognize that the data will change over time, and new data may present itself. This means, inherently, that decisions will need to be revisited from time to time over the life span of the system in question. For long-lived projects, it's often critical to have the ability to change directions after an initial decision is made. And, importantly, it means that the deciders need to have the right to admit mistakes. Contrary to some people's instincts, leaders who admit mistakes are more respected, not less.

我們堅信資料能為決策提供資訊，但我們也認識到資料會隨著時間的推移而變化，新資料可能會出現。這意味著，本質上，在相關係統的生命週期內，需要不時地重新審視決策。對於長期專案而言，在做出初始決策後，有能力改變方向通常是至關重要的。更重要的是，這意味著決策者需要勇氣承認錯誤。與人的本能相反，勇於承認錯誤的領導人受更多的尊重。

Be evidence driven, but also realize that things that can't be measured may still have value. If you're a leader, that's what you've been asked to do: exercise judgement, assert that things are important. We'll speak more on leadership in Chapters 5 and 6.

以證據為導向，但也要意識到無法衡量的東西可能仍然有價值。如果你是一個領導者，那就是你被要求做的：審時度勢，主張事在人為。我們將在第5章和第6章中詳細介紹領導力

Software Engineering Versus Programming 軟體工程與程式設計

When presented with our distinction between software engineering and programming, you might ask whether there is an inherent value judgement in play. Is programming somehow worse than software engineering? Is a project that is expected to last a decade with a team of hundreds inherently more valuable than one that is useful for only a month and built by two people?

在介紹軟體工程和程式設計之間的區別時，你可能會問，是否存在內在的價值判斷。程式設計是否比軟體工程更糟糕？一個由數百人組成的團隊預計將持續十年的專案是否比一個只有一個月的專案和兩個人構建的專

案更有價值？

Of course not. Our point is not that software engineering is superior, merely that these represent two different problem domains with distinct constraints, values, and best practices. Rather, the value in pointing out this difference comes from recognizing that some tools are great in one domain but not in the other. You probably don't need to rely on integration tests (see Chapter 14) and Continuous Deployment (CD) practices (see Chapter 24) for a project that will last only a few days. Similarly, all of our long-term concerns about semantic versioning (SemVer) and dependency management in software engineering projects (see Chapter 21) don't really apply for short-term programming projects: use whatever is available to solve the task at hand.

當然不是。我們的觀點並不是說軟體工程是優越的，只是它們代表了兩個不同的問題領域，具有不同的約束、價值和最佳實踐。相反，指出這種差異的價值來自於認識到一些工具在一個領域是偉大的，但在另一個領域不是。對於只持續幾天的專案，你不需要依賴整合測試（參見第14章）和持續部署（CD）實踐（參見第24章）。同樣地，我們對軟體工程專案中的版本控制（SemVer）和依賴性管理（參見第21章）的所有長期關注，並不適用於短期程式設計專案：利用一切可以利用的手段來解決手頭的任務。

We believe it is important to differentiate between the related-but-distinct terms "programming" and "software engineering." Much of that difference stems from the management of code over time, the impact of time on scale, and decision making in the face of those ideas. Programming is the immediate act of producing code. Software engineering is the set of policies, practices, and tools that are necessary to make that code useful for as long as it needs to be used and allowing collaboration across a team.

我們認為，區分相關但不同的術語"程式設計"和"軟體工程"是很重要的。這種差異很大程度上源於隨著時間的推移對程式碼的管理、時間對規模的影響以及面對這些想法的決策。程式設計是產生程式碼的直接行為。軟體工程是一組策略、實踐和工具，這些策略、實踐和工具是使程式碼在需要使用的時間內發揮作用，並允許整個團隊的協作。

Conclusion 總結

This book discusses all of these topics: policies for an organization and for a single programmer, how to evaluate and refine your best practices, and the tools and technologies that go into maintainable software. Google has worked hard to have a sustainable codebase and culture. We don't necessarily think that our approach is the one true way to do things, but it does provide proof by example that it can be done. We hope it will provide a useful framework for thinking about the general problem: how do you maintain your code for as long as it needs to keep working?

本書討論了所有這些主題：一個組織和一個程式設計師的策略，如何評估和改進你的最佳實踐，以及用於可維護軟體的工具和技術。谷歌一直在努力打造可持續的程式碼庫和文化。我們不認為我們的方法是做事情的唯一正確方法，但它確實通過例子證明了它是可以做到的。我們希望它將提供一個有用的框架來思考一般問題：你如何維護你的程式碼，讓它正常執行。

TL;DRs 內容提要

- Software engineering" differs from "programming" in dimensionality: programming is about producing code. Software engineering extends that to include the maintenance of that code for its useful life span.

- There is a factor of at least 100,000 times between the life spans of short-lived code and long-lived code. It is silly to assume that the same best practices apply universally on both ends of that spectrum.
- Software is sustainable when, for the expected life span of the code, we are capable of responding to changes in dependencies, technology, or product requirements. We may choose to not change things, but we need to be capable.
- Hyrum's Law: with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.
- Every task your organization has to do repeatedly should be scalable (linear or better) in terms of human input. Policies are a wonderful tool for making process scalable.
- Process inefficiencies and other software-development tasks tend to scale up slowly. Be careful about boiled-frog problems.
- Expertise pays off particularly well when combined with economies of scale.
- "Because I said so"" is a terrible reason to do things.
- Being data driven is a good start, but in reality, most decisions are based on a mix of data, assumption, precedent, and argument. It's best when objective data makes up the majority of those inputs, but it can rarely be all of them.
- Being data driven over time implies the need to change directions when the data changes (or when assumptions are dispelled). Mistakes or revised plans are inevitable.
- "軟體工程"與"程式設計"在維度上不同：程式設計是關於編寫程式碼的。軟體工程擴充套件了這一點，包括在程式碼的生命週期內對其進行維護。
- 短期程式碼和長期程式碼的生命週期之間至少有100,000倍的係數。假設相同的最佳實踐普遍適用於這一範圍的兩端是愚蠢的。
- 在預期的程式碼生命週期內，當我們能夠響應依賴關係、技術或產品需求的變化時，軟體是可持續的。我們可能選擇不改變事情，但我們需要有能力。
- 海勒姆定律：當一個 API 有足夠的使用者的時候，在約定中你承諾的什麼都無所謂，所有在你係統裡面被觀察到的行為都會被一些使用者直接依賴。
- 你的組織重複執行的每項任務都應在人力投入方面具有可擴充套件性（線性或更好）。策略是使流程可伸縮的好工具。
- 流程效率低下和其他軟體開發任務往往會慢慢擴大規模。小心煮青蛙的問題。
- 當專業知識與規模經濟相結合時，回報尤其豐厚。
- "因為我說過"是做事的可怕理由。
- 資料驅動是一個良好的開端，但實際上，大多數決策都是基於資料、假設、先例和論據的混合。最好是客觀資料佔這些輸入的大部分，但很少可能是全部。

- 隨著時間的推移，資料驅動意味著當資料發生變化時（或假設消除時），需要改變方向。錯誤或修訂的計劃不在表中。