CHAPTER 2

How to Work Well on Teams

第二章 如何融入團隊

Written by Brian Fitzpatrick

Edited by Riona MacNamara

Because this chapter is about the cultural and social aspects of software engineering at Google, it makes sense to begin by focusing on the one variable over which you definitely have control: you.

因為本章是從文化和社會方面來介紹谷歌軟體工程，首先從焦聚與一個你完全控制的變數開始：你自己。

People are inherently imperfect—we like to say that humans are mostly a collection of intermittent bugs. But before you can understand the bugs in your coworkers, you need to understand the bugs in yourself. We're going to ask you to think about your own reactions, behaviors, and attitudes—and in return, we hope you gain some real insight into how to become a more efficient and successful software engineer who spends less energy dealing with people-related problems and more time writing great code.

人天生是不完美的——我們常說，人類大多是一個個不同缺點的組成集合。但是，在你瞭解同事身上的缺點之前，你需要了解自己身上的缺點。我們將要求你反思自己的反應、行為和態度——作為回報，我們希望你能夠真正瞭解如何成為一名更高效、更成功的軟體工程師，減少處理與人相關的問題的精力，花更多的時間編寫厲害的程式碼。

The critical idea in this chapter is that software development is a team endeavor. And to succeed on an engineering team—or in any other creative collaboration—you need to reorganize your behaviors around the core principles of humility, respect, and trust.

本章的關鍵思想是，軟體開發是團隊的努力。要在工程團隊或任何其他創造性合作中取得成功，你需要圍繞謙遜、尊重和信任的核心原則重新定義你的行為。

Before we get ahead of ourselves, let's begin by observing how software engineers tend to behave in general.

在我們超越自己之前，讓我們首先觀察軟體工程師的一般行為。

Help Me Hide My Code 幫我隱藏我的程式碼

For the past 20 years, my colleague Ben[1] and I have spoken at many programming conferences. In 2006, we launched Google's (now deprecated) open source Project Hosting service, and at first, we used to get lots of questions and requests about the product. But around mid-2008, we began to notice a trend in the sort of requests we were getting:
"Can you please give Subversion on Google Code the ability to hide specific branches?"
"Can you make it possible to create open source projects that start out hidden to the world and then are revealed when they're ready?"
"Hi, I want to rewrite all my code from scratch, can you please wipe all the history?"
Can you spot a common theme to these requests?

在過去的 20 年裡，我和我的同事 Ben 在很多程式設計會議上演講。 在 2006 年，我們推出了 Google 的開源專案託管服務（現已棄用），在開始時，我們收到很多關於該產品的問題和請求。但到了 2008 年年中左右，我們發現，我們收到的請求中很多是這樣的：

"你能否讓 Google Code 上的 Subversion 能夠隱藏指定分支？"

"你能否讓建立的開源專案開始時對外隱藏，在它們準備好後再公開？"

"嗨，我想從頭開始重構我所有的程式碼，你能把所有的歷史記錄都刪除嗎？"

你能找出這些要求的共同點嗎？

The answer is insecurity. People are afraid of others seeing and judging their work in progress. In one sense, insecurity is just a part of human nature—nobody likes to be criticized, especially for things that aren't finished. Recognizing this theme tipped us off to a more general trend within software development: insecurity is actually a symptom of a larger problem.

答案是缺乏安全感。人們害怕別人看到和評價他們正在進行的工作。從某種意義上說，缺乏安全感是人性的一部分——沒有人喜歡被批評，尤其是那些沒有完成的事情。認識到這個主題讓我們看到了軟體開發中一個更普遍的趨勢：缺乏安全實際上是一個更大問題的徵兆。

1 Ben Collins-Sussman，也是本書的作者之一。

The Genius Myth 天才的神話

Many humans have the instinct to find and worship idols. For software engineers, those might be Linus Torvalds, Guido Van Rossum, Bill Gates—all heroes who changed the world with heroic feats. Linus wrote Linux by himself, right?

許多人有尋找和崇拜偶像的本能。對於軟體工程師來說，他們可能是 Linus Torvalds, Guido Van Rossum, Bill Gates——他們都是改變世界的英雄。是 Linus 自己寫的 Linux？

Actually, what Linus did was write just the beginnings of a proof-of-concept Unix- like kernel and show it to an email list. That was no small accomplishment, and it was definitely an impressive achievement, but it was just the tip of the iceberg. Linux is hundreds of times bigger than that initial kernel and was developed by thousands of smart people. Linus' real achievement was to lead these people and coordinate their work; Linux is the shining result not of his original idea, but of the collective labor of the community. (And Unix itself was not entirely written by Ken Thompson and Dennis Ritchie, but by a group of smart people at Bell Labs.)

實際上，Linus 所做的只是編寫了概念驗證類 Unix 核心的開頭，並將電子郵件傳送出去。這是不小的成就，絕對是一個令人印象深刻的成就，但這只是冰山一角。Linux 比最初的核心大幾百倍，是由成千上萬的聰明人開發的。Linus 真正的成就是領導並協調他們的完成工作；Linux 不僅僅是他最初創意的成果，更是社群集體努力的成果。 （Unix 本身並非完全由肯·湯普森和丹尼斯·裡奇編寫，而是由貝爾實驗室的一群聰明人編寫的。）

On that same note, did Guido Van Rossum personally write all of Python? Certainly, he wrote the first version. But hundreds of others were responsible for contributing to subsequent versions, including ideas, features, and bug fixes. Steve Jobs led an entire team that built the Macintosh, and although Bill Gates is known for writing a BASIC interpreter for early home computers, his bigger achievement was building a successful company around MS-DOS. Yet they all became leaders and symbols of the collective achievements of their communities. The Genius Myth is the tendency that we as humans need to ascribe the success of a team to a single person/leader.

同樣，Guido Van Rossum 是否親自編寫了所有 Python？當然，他寫了第一個版本。但還有數百人為後續版本做出貢獻，包括想法、功能和 bug 修復。史蒂夫·喬布斯領導了一個建造麥金塔的整個團隊，儘管比爾·蓋茨以為早期家用電腦編寫 BASIC 直譯器而聞名，但他更大的成就是圍繞 MS-DOS 系統建立了一家成功的公司。然而，他們都成為集體成就的領袖和象徵。天才的神話是一種趨勢，即我們需要將團隊的成功歸因於一個人/領導者。

And what about Michael Jordan?

那麼 Michael Jordan 呢？

It's the same story. We idolized him, but the fact is that he didn't win every basketball game by himself. His true genius was in the way he worked with his team. The team's coach, Phil Jackson, was extremely clever, and his coaching techniques are legendary.

這是同一個故事。我們崇拜他，但事實是他並不是一個人贏得每一場籃球比賽的。他真正的厲害在於他與團隊合作的方式。這支球隊的教練 Phil Jackson 非常聰明，他的教練技術堪稱傳奇。

He recognized that one player alone never wins a championship, and so he assembled an entire "dream team" around MJ. This team was a well-oiled machine—at least as impressive as Michael himself.

他認識到只有一個球員永遠無法贏得冠軍，因此他圍繞 Michael Jordan 組建了一支完美的"夢之隊"。這支球隊是一臺運轉良好的機器——至少和邁克爾本人一樣令人印象深刻。

So, why do we repeatedly idolize the individual in these stories? Why do people buy products endorsed by celebrities? Why do we want to buy Michelle Obama's dress or Michael Jordan's shoes?

那麼，為什麼我們在這些故事中一再崇拜個人呢？為什麼人們會購買名人代言的產品？我們為什麼要買米歇爾·奧巴馬的裙子或 Michael Jordan 的鞋子？

Celebrity is a big part of it. Humans have a natural instinct to find leaders and role models, idolize them, and attempt to imitate them. We all need heroes for inspiration, and the programming world has its heroes, too. The phenomenon of “techie- celebrity” has almost spilled over into mythology. We all want to write something world-changing like Linux or design the next brilliant programming language.

名人效應是一個重要原因。人類有尋找領導者和榜樣的本能，崇拜他們，並試圖模仿他們。我們都需要英雄來激發靈感，程式設計世界也有自己的英雄。“科技名人”已經幾乎被神化，我們都想寫一些改變世界的東西，比如 Linux 或者設計下一種優秀的程式語言。

Deep down, many engineers secretly wish to be seen as geniuses. This fantasy goes something like this:

- You are struck by an awesome new concept.
- You vanish into your cave for weeks or months, slaving away at a perfect implementation of your idea.
- You then “unleash” your software on the world, shocking everyone with your genius.
- Your peers are astonished by your cleverness.
- People line up to use your software.
- Fame and fortune follow naturally.

在內心深處，許多工程師暗中希望被視為天才。這種幻想是這樣的：

- 你會被一個了不起的新概念所震撼。
- 你消失數週或數月躲在洞穴中，努力實現你的理想。
- 然後世界上“釋出”你的軟體，用你的天才震撼每個人。
- 你的同齡人對你的聰明感到驚訝。
- 人們排隊使用你的軟體。
- 名利自然隨之而來。

But hold on: time for a reality check. You’re probably not a genius.

是時候迴歸現實了。你很可能不是天才。

No offense, of course—we’re sure that you’re a very intelligent person. But do you realize how rare actual geniuses really are? Sure, you write code, and that’s a tricky skill. But even if you are a genius, it turns out that that’s not enough. Geniuses still make mistakes, and having brilliant ideas and elite programming skills doesn’t guarantee that your software will be a hit. Worse, you might find yourself solving only analytical problems and not human problems. Being a genius is most definitely not an excuse for being a jerk: anyone—genius or not—with poor social skills tends to be a poor teammate. The vast majority of the work at Google (and at most companies!) doesn’t require genius-level intellect, but 100% of the work requires a minimal level of social skills. What will make or break your career, especially at a company like Google, is how well you collaborate with others.

無意冒犯，我們當然相信你是個非常聰明的人。但你要知道真正的天才有多稀有嗎？當然，你需要編寫程式碼，這是一項棘手的技能。即使你是個天才，會程式設計還不夠。天才仍然會犯錯誤，擁有卓越的想法和精英程式設計技能並不能保證你的軟體會大受歡迎。更糟糕的是，你可能會發現自己只解決了分析性問題，而沒有解決人的問題。作為一個天才絕對不是成為一個混蛋的藉口： 天才與否，社交能力差的人，往往是一個豬隊友。谷歌（以及大多數公司！）的絕大多數工作不需要天才水平的智力，但100%的工作需要最低水平的社交技能。決定你職業生涯成敗，尤其是在谷歌這樣的公司，更取決於你與他人合作的程度。

It turns out that this Genius Myth is just another manifestation of our insecurity. Many programmers are afraid to share work they’ve only just started because it means peers will see their mistakes and know the author of the code is not a genius.

事實證明，這種天才神話只是我們缺乏安全感的另一種表現。許多程式設計師害怕分享他們剛剛開始的工作，因為這意味著同行會看到他們的錯誤，知道程式碼的作者不是天才。

To quote a friend:
I know I get SERIOUSLY insecure about people looking before something is done. Like they are going to seriously judge me and think I’m an idiot.

參考一位朋友的話：
我知道，別人在我完成某事之前就來看，會讓我感到非常不安全。好像他們會認真地評判我，認為我是個白痴。

This is an extremely common feeling among programmers, and the natural reaction is to hide in a cave, work, work, work, and then polish, polish, polish, sure that no one will see your goof-ups and that you’ll still have a chance to unveil your masterpiece when you’re done. Hide away until your code is perfect.

這是程式設計師群體中非常普遍的感覺，第一反應就是躲到山洞裡，工作，工作，努力工作，然後打磨，打磨，再打磨，確定沒有人會看到你的錯誤後，當你完成後，你才會揭開你的作品面紗。躲起來吧，直到你的程式碼變得完美。

Another common motivation for hiding your work is the fear that another programmer might take your idea and run with it before you get around to working on it. By keeping it secret, you control the idea.

隱藏工作的另一個常見動機是擔心另一個程式設計師可能會在你開始工作之前就拿著你的想法跑了。透過保密，你可以控制這個想法。

We know what you’re probably thinking now: so what? Shouldn’t people be allowed to work however they want?
Actually, no. In this case, we assert that you’re doing it wrong, and it is a big deal. Here’s why.

我們知道你現在在想什麼：那又怎樣？難道不應該允許我隨心所欲地工作嗎？

事實上，不應該。在這種情況下，我們斷定你錯了，很是一個大錯誤。原因如下。

Hiding Considered Harmful 隱藏不利

If you spend all of your time working alone, you're increasing the risk of unnecessary failure and cheating your potential for growth. Even though software development is deeply intellectual work that can require deep concentration and alone time, you must play that off against the value (and need!) for collaboration and review.

如果你把所有的時間都花在獨自工作上，增加了不必要失敗的風險，耽誤了你的成長潛力。儘管軟體開發是一項需要高度集中精力和獨處時間的深度智力工作，但你必須權衡協作和審查的價值（以及需求！）。

First of all, how do you even know whether you're on the right track?

首先，你怎麼知道自己是否在正確的軌道上？

Imagine you're a bicycledesign enthusiast, and one day you get a brilliant idea for a completely new way to design a gear shifter. You order parts and proceed to spend weeks holed up in your garage trying to build a prototype. When your neighbor—also a bike advocate—asks you what's up, you decide not to talk about it. You don't want anyone to know about your project until it's absolutely perfect. Another few months go by and you're having trouble making your prototype work correctly. But because you're working in secrecy, it's impossible to solicit advice from your mechanically inclined friends.

想象一下，你是一個腳踏車設計愛好者，有一天你有了一個厲害的想法，可以用一種全新的方式來設計變速器。你訂購零件，然後花數週時間躲在車庫裡，嘗試製造一個原型。當你的鄰居——也是腳踏車倡導者——問你怎麼了，你決定閉口不談。你不想讓任何人知道你的專案，直到它絕對完美。又過幾個月，你在讓原型執行時遇到了麻煩。但因為你是在保密的情況下工作，所以不可能徵求你那些有機械專家朋友的意見。

Then, one day your neighbor pulls his bike out of his garage with a radical new gear- shifting mechanism. Turns out he's been building something very similar to your invention, but with the help of some friends down at the bike shop. At this point, you're exasperated. You show him your work. He points out that your design had some simple flaws—ones that might have been fixed in the first week if you had shown him. There are a number of lessons to learn here.

然後，有一天，你的鄰居將他的腳踏車從車庫中拉出，這輛腳踏車使用一種全新的換檔機構。事實表明，他也在製造一些與你的發明非常相似的東西，但是他得到了腳踏車店的一些朋友的幫助。這時候，你很生氣。你給他看你的原型。他指出，你的設計有一些簡單的缺陷，如果你給他看的話，這些缺陷可能在第一週就被修復了。這裡有許多教訓要學習。

Early Detection 及早發現

If you keep your great idea hidden from the world and refuse to show anyone anything until the implementation is polished, you're taking a huge gamble. It's easy to make fundamental design mistakes early on. You risk reinventing wheels.2 And you forfeit the benefits of collaboration, too: notice how much faster your neighbor moved by working with others? This is why people dip their toes in the water before jumping in the deep end: you need to make sure that you're working on the right thing, you're doing it correctly, and it hasn't been done before. The chances of an early misstep are high. The more feedback you solicit early on, the more you lower this risk.3 Remember the tried-and-true mantra of "Fail early, fail fast, fail often."

如果你對世界隱瞞你的厲害想法，並在未完美之前拒絕向任何人展示，那麼你就是在進行一場下注巨大的賭博。早期很容易犯基本的設計錯誤。你冒著重新發明輪子的風險。[2] 而且你也失去了協作的好處：注意到你的鄰居透過與他人合作而效率有多高？這就是人們在跳入深水區之前將腳趾浸入水中的原因：你需要確保你在做正確的事情，你在做正確的事情，而且以前從未做過。早期失誤的可能性很高。你越早徵求反饋，這種風險就越低。[3] 記住“早失敗、快失敗、經常失敗”這句經得起考驗的至理名言。

Early sharing isn't just about preventing personal missteps and getting your ideas vetted. It's also important to strengthen what we call the bus factor of your project.

儘早分享不僅僅是為了防止個人失誤和檢驗你的想法，對於加強我們稱之為專案的巴士因子也是十分重要的。

> 2 實際上，如果你是一個腳踏車設計師。
>
> 3 我應該注意到，如果你仍然不確定自己的總體方向或目標，那麼在過程中過早地獲得太多反饋是很危險的。

The Bus Factor 巴士因子

Bus factor (noun): the number of people that need to get hit by a bus before your project is completely doomed.

巴士因子：團隊裡因巴士撞倒的多少人，會導致專案失敗。

How dispersed is the knowledge and know-how in your project? If you're the only person who understands how the prototype code works, you might enjoy good job security—but if you get hit by a bus, the project is toast. If you're working with a colleague, however, you've doubled the bus factor. And if you have a small team designing and prototyping together, things are even better—the project won't be marooned when a team member disappears. Remember: team members might not literally be hit by buses, but other unpredictable life events still happen. Someone might get married, move away, leave the company, or take leave to care for a sick relative. Ensuring that there is at least good documentation in addition to a primary and a secondary owner for each area of responsibility helps future-proof your project's success and increases your project's bus factor. Hopefully most engineers recognize that it is better to be one part of a successful project than the critical part of a failed project.

你的專案中的知識和技能分散程度如何？如果你是唯一瞭解原型程式碼工作原理的人，你需要會受到良好的工作保障，但如果你被公車撞倒，專案就完蛋了。但是，如果你與同事合作，你的巴士因子就翻了一番。如果你有一個小團隊一起進行設計和製作原型，情況會更好——當團隊某個成員消失時，專案不會被孤立。記住：團隊成員可能不會被公車撞到，但其他不可預知的事件仍然會發生。有人可能會結婚、搬走、離開公司或請假照顧生病的親屬。確保每個責任領域除了一個主要和一個次要所有者之外，至少還有可用的文件，這有助於確保專案的成功，提高專案的成功率。希望大多數工程師認識到，成為成功專案的一部分比成為失敗專案的關鍵部分要好。

Beyond the bus factor, there's the issue of overall pace of progress. It's easy to forget that working alone is often a tough slog, much slower than people want to admit. How much do you learn when working alone? How fast do you move? Google and Stack Overflow are great sources of opinions and information, but they're no substitute for actual human experience. Working with other people directly increases the collective wisdom behind the effort. When you become stuck on something absurd, how much time do you waste pulling yourself out of the hole? Think about how different the experience would be if you had a couple of peers to look over your shoulder and tell you—instantly—how you goofed and how to get past the problem. This is exactly why teams sit together (or do pair programming) in software engineering companies. Programming is hard. Software engineering is even harder. You need that second pair of eyes.

除了巴士因子，還有整體進度的問題。人們很容易忘記，獨自工作往往是一項艱苦的工作，比人們自認為的慢得多。你獨自工作能學到多少？你推進地有多快？Google 和 Stack Overflow 是觀點和資訊的重要來源，但它們不能替代人的真實體驗。與他人一起工作會直接增加工作背後的集體智慧。當你陷入誤區時，你需要浪費多少時間才能從困境中解脫？想想如果你有幾個同齡人看著你並立即告知你是如何犯錯以及如何解決問題，體驗會有多麼不同。這正是軟體工程公司中團隊坐在一起（或進行配對程式設計）的原因。程式設計很難。軟體工程更難。你需要另一雙眼睛。

Pace of Progress 進展速度

Here's another analogy. Think about how you work with your compiler. When you sit down to write a large piece of software, do you spend days writing 10,000 lines of code, and then, after writing that final, perfect line, press the "compile" button for the very first time? Of course you don't. Can you imagine what sort of disaster would result? Programmers work best in tight feedback loops: write a new function, compile. Add a test, compile. Refactor some code, compile. This way, we discover and fix typos and bugs as soon as possible after generating code. We want the compiler at our side for every little step; some environments can even compile our code as we type. This is how we keep code quality high and make sure our software is evolving correctly, bit by bit. The current DevOps philosophy toward tech productivity is explicit about these sorts of goals: get feedback as early as possible, test as early as possible, and think about security and production environments as early as possible. This is all bundled into the idea of "shifting left" in the developer workflow; the earlier we find a problem, the cheaper it is to fix it.

這是另一個類別比。考慮一下如何使用編譯器。當你坐下來編寫一個大型軟體時，你是否會花上幾天的時間編寫 10,000 行程式碼，然後在編寫完最後一行完美的程式碼後，第一次按下"編譯"按鈕？你當然不知道。你能想象會發生什麼樣的災難嗎？程式設計師在密集的迴圈反饋中工作做得最好：編寫一個新函式 compile，新增一個測試，編譯。重構一些程式碼，編譯。這樣，我們可以在產生程式碼後儘快發現並修復拼寫錯誤和 bug。我們希望完成每一小步都要使用編譯器；有些環境甚至可以在我們寫入程式碼時自動編譯。這就是我們如何保持程式碼高品質並確保我們的軟體一點一點地正確迭代的方法。當前 DevOps 對技術生產力的理念明確了這些目標：儘早獲得反饋，儘早進行測試，儘早考慮安全和生產環境。這一切都與開發人員工作流程中的"左移"思想捆綁在一起；我們越早發現問題，修復它的成本就越低。

The same sort of rapid feedback loop is needed not just at the code level, but at the whole-project level, too. Ambitious projects evolve quickly and must adapt to changing environments as they go. Projects run into unpredictable design obstacles or political hazards, or we simply discover that things aren't working as planned. Requirements morph unexpectedly. How do you get that feedback loop so that you know the instant your plans or designs need to change? Answer: by working in a team. Most engineers know the quote, "Many eyes make all bugs shallow," but a better version might be, "Many eyes make sure your project stays relevant and on track." People working in caves awaken to discover that while their original vision might be complete, the world has changed and their project has become irrelevant.

同樣的快速反饋迴圈不僅在程式碼級別需要，在整個專案級別也需要。雄心勃勃的專案快速發展，必須適應不斷變化的環境。專案遇到不可預測的設計障礙或政治風險，或者我們只是發現事情沒有按計劃進行。需求又發生變化了。你如何獲得反饋迴圈，以便你知道你的計劃或設計需要更改的時候？答：透過團隊合作。大多數工程師都知道這樣一句話："人多走得更快"，但更好的說法可能是，"人多走得更遠。"在洞穴中工作的人們醒來後發現，雖然他們最初的願景可能已經實現，但世界已經改變，他們的專案變得無關緊要。

Case Study: Engineers and Offices

Twenty-five years ago, conventional wisdom stated that for an engineer to be productive, they needed to have their own office with a door that closed. This was supposedly the only way they could have big, uninterrupted slabs of time to deeply concentrate on writing reams of code.

25 年前，傳統觀念認為，工程師要想提高工作效率，就需要有一間自己的辦公室，還要有一扇關著的門。據說，只有這樣，他們才能有充足的時間、不受干擾的編寫程式碼。

I think that it’s not only unnecessary for most engineers [4] to be in a private office, it’s downright dangerous. Software today is written by teams, not individuals, and a high- bandwidth, readily available connection to the rest of your team is even more valuable than your internet connection. You can have all the uninterrupted time in the world, but if you’re using it to work on the wrong thing, you’re wasting your time.

我認為，對大多數工程師來說，在私人辦公室裡不僅沒有必要，而且是完全錯誤的。今天的軟體是由團隊而不是個人編寫的，與團隊其他成員的高頻寬、隨時可用的連線甚至比你使用網際網路更有價值。你可以擁有不受打擾的時間，但如果你用它來做錯誤的事情，你這是在浪費時間。

Unfortunately, it seems that modern-day tech companies (including Google, in some cases) have swung the pendulum to the exact opposite extreme. Walk into their offices and you’ll often find engineers clustered together in massive rooms—a hundred or more people together—with no walls whatsoever. This “open floor plan” is now a topic of huge debate and, as a result, hostility toward open offices is on the rise. The tiniest conversation becomes public, and people end up not talking for risk of annoying dozens of neighbors. This is just as bad as private offices!

不幸的是，現代科技公司（在某些情況下包括谷歌）似乎已經走向了另一個極端。走進他們的辦公室，你經常會發現工程師們聚集在一個巨大的房間裡———百多人聚集在一起，沒有任何牆壁。這中“開放式平面圖”現在是一個大辯論的話題，因此，對開放式辦公室的敵意正在上升。最小範圍談話都會公開，工程師們不再說話，以免惹惱周圍幾十個鄰居。這和私人辦公室一樣糟糕！

We think the middle ground is really the best solution. Group teams of four to eight people together in small rooms (or large offices) to make it easy (and non- embarrassing) for spontaneous conversation to happen.

我們覺得折中的方案是最好的解決方法。在小房間（或大辦公室）將四至八人組成小組，方便大家輕鬆（且不令人尷尬）地自由對話。

Of course, in any situation, individual engineers still need a way to filter out noise and interruptions, which is why most teams I’ve seen have developed a way to communicate that they’re currently busy and that you should limit interruptions. Some of us used to work on a team with a vocal interrupt protocol: if you wanted to talk, you would say “Breakpoint Mary,” where Mary was the name of the person you wanted to talk to. If Mary was at a point where she could stop, she would swing her chair around and listen. If Mary was too busy, she’d just say “ack,” and you’d go on with other things until she finished with her current head state.

當然，在很多情況下，個別工程師還會需要一種方法來過濾噪音和干擾，這就是為什麼我所見過的大多數團隊都開發了一種方法來表示他們目前很忙，你不應該來打擾。 我們中的一些人曾經在一個團隊中工作，有一個發聲中斷協議：如果你想說話，你會說 “Breakpoint Mary”，其中 Mary 是你想對話人的名字。如果 Mary 能停下來，她會把椅子轉過來聽。如果 Mary 太忙，她只會說 “確認”，你會繼續做其他事情，直到她完成她當前的工作。

Other teams have tokens or stuffed animals that team members put on their monitor to signify that they should be interrupted only in case of emergency. Still other teams give out noise-canceling headphones to engineers to make it easier to deal with background noise—in fact, in many companies, the very act of wearing headphones is a common signal that means “don’t disturb me unless it’s really important.” Many engineers tend to go into headphones-only mode when coding, which may be useful for short spurts but, if used all the time, can be just as bad for collaboration as walling yourself off in an office.

其他團隊有標記或布娃娃，團隊成員將它們放在顯示器上，以表示只有在緊急情況下才應打擾。還有一些團隊向工程師分發降噪耳機，以便於處理背景噪音。事實上，在許多公司，佩戴耳機的行為是一種常見的訊號，表示“除非非常重要，否則不要打擾我。”許多工程師在編碼時傾向於只使用耳機模式，這可能對短時間的使用是有效，但如果一直使用，對協作的影響和把自己獨自關在辦公室裡一樣糟糕。

Don’t misunderstand us—we still think engineers need uninterrupted time to focus on writing code, but we think they need a high-bandwidth, low-friction connection to their team just as much. If less-knowledgeable people on your team feel that there’s a barrier to asking you a question, it’s a problem: finding the right balance is an art.

不要誤解我們，我們仍然認為工程師需要不受打擾的時間來專注於編寫程式碼，但我們認為他們同樣需要一個高頻寬、低衝突的團隊連線。如果你的團隊中新人覺得向你提問存在障礙，那就是一個問題：找到正確的平衡是一門藝術。

> 4 然而，我承認，嚴肅內向的人可能比大多數人需要更多的平靜、安靜和獨處的時間，如果不是他們自己的辦公室，他們可能會從一個更安靜的環境中受益。

In Short, Don’t Hide 總之，不要隱藏

So, what “hiding” boils down to is this: working alone is inherently riskier than working with others. Even though you might be afraid of someone stealing your idea or thinking you’re not intelligent, you should be much more concerned about wasting huge swaths of your time toiling away on the wrong thing.

Don’t become another statistic.

因此，“隱藏”歸結起來就是：獨自工作比與他人一起工作具有更高的內在風險。即使你可能害怕有人竊取你的想法或認為你不聰明，你更應該擔心浪費大量時間在錯誤的事情上。

不要成為另一個統計數字。

It’s All About the Team 一切都是為了團隊

So, let’s back up now and put all of these ideas together.

那麼，讓我們現在回顧一下，把所有這些想法放在一起。

The point we’ve been hammering away at is that, in the realm of programming, lone craftspeople are extremely rare—and even when they do exist, they don’t perform superhuman achievements in a vacuum; their world-changing accomplishment is almost always the result of a spark of inspiration followed by a heroic team effort.

我們反覆強調的一點是，在程式設計領域，孤獨的工匠極其罕見，即使他們確實存在，他們也不會在真空中完成超人的成就；他們改變世界的成就幾乎總是靈感迸發、團隊英勇努力的結果。

A great team makes brilliant use of its superstars, but the whole is always greater than the sum of its parts. But creating a superstar team is fiendishly difficult.

一個偉大的團隊能夠出色地利用它的超級明星，但整體總是大於各部分的總和。但打造一支集合多個超級明星球隊是極其困難的。

Let’s put this idea into simpler words: software engineering is a team endeavor.

讓我們把這個想法用更簡單的話來說：軟體工程是一個團隊的努力。

This concept directly contradicts the inner Genius Programmer fantasy so many of us hold, but it’s not enough to be brilliant when you’re alone in your hacker’s lair. You’re not going to change the world or delight millions of computer users by hiding and preparing your secret invention. You need to work with other people. Share your vision. Divide the labor. Learn from others. Create a brilliant team.

這個概念直接與我們許多人幻想的天才程式設計師幻想相矛盾，但當你獨自一人在駭客的巢穴中時，這也是不夠聰明。你不能透過隱藏和準備你的秘密發明來改變世界或取悅數百萬計的使用者。你需要和其他人一起工作。分享你的願景，分工，向別人學習，建立一個出色的團隊。

Consider this: how many pieces of widely used, successful software can you name that were truly written by a single person? (Some people might say “LaTeX,” but it’s hardly “widely used,” unless you consider the number of people writing scientific papers to be a statistically significant portion of all computer users!)

想一想：有多少種被廣泛使用併成功的軟體，你能說出真正由一個人寫的嗎？（有些人可能會說“LaTeX”，但它幾乎不被廣泛使用，除非你認為撰寫科學論文的人數在所有計算機使用者中佔一大部分！）

High-functioning teams are gold and the true key to success. You should be aiming for this experience however you can.

高效的團隊是黃金，是成功的真正關鍵。你應該儘可能地追求這種體驗。

The Three Pillars of Social Interaction 社交的三大支柱

So, if teamwork is the best route to producing great software, how does one build (or find) a great team?

那麼，如果團隊合作是生產優秀軟體的最佳路徑，那麼如何建立（或找到）一個優秀的團隊呢？

To reach collaborative nirvana, you first need to learn and embrace what I call the “three pillars” of social skills. These three principles aren’t just about greasing the wheels of relationships; they’re the foundation on which all healthy interaction and collaboration are based:

Pillar 1: Humility
You are not the center of the universe (nor is your code!). You’re neither omniscient nor infallible. You’re open to self-improvement.
Pillar 2: Respect
You genuinely care about others you work with. You treat them kindly and appreciate their abilities and accomplishments.
Pillar 3: Trust
You believe others are competent and will do the right thing, and you’re OK with letting them drive when appropriate.[5]

要達到協作的最佳效果，你首先需要學習並接受我所說的社交的“三大支柱”。這三個原則不僅僅是人際關係的潤滑劑，更是一切健康互動和協作的基礎：

支柱 1：謙遜
你不是宇宙的中心（你的程式碼也不是！）。你既不是全方位的，也不是絕對正確的。你願意不斷提升自我。
支柱 2：尊重
你真誠地關心與你一起工作的人。你善待他們，欣賞他們的能力和成就。
支柱 3：信任
你相信其他人有能力並且會做正確的事情，你可以讓他們在適當的時候牽頭。

If you perform a root-cause analysis on almost any social conflict, you can ultimately trace it back to a lack of humility, respect, and/or trust. That might sound implausible at first, but give it a try. Think about some nasty or uncomfortable social situation currently in your life. At the basest level, is everyone being appropriately humble? Are people really respecting one another? Is there mutual trust?

如果你對所有社會衝突進行根本原因分析，你最終可以追溯到缺乏謙遜、尊重和信任。一開始聽起來似乎不太可信，但不妨試一試。想想你生活中的一些令人尷尬或不舒服的社交場合。在最基本的層面上，每個人都適當地謙虛嗎？人們真的互相尊重嗎？有相互信任嗎？

> 5 如果你過去曾被委派給不稱職的人，這將是非常困難的。

Why Do These Pillars Matter?為什麼這些支柱很重要？

When you began this chapter, you probably weren't planning to sign up for some sort of weekly support group. We empathize. Dealing with social problems can be difficult: people are messy, unpredictable, and often annoying to interface with. Rather than putting energy into analyzing social situations and making strategic moves, it's tempting to write off the whole effort. It's much easier to hang out with a predictable compiler, isn't it? Why bother with the social stuff at all?

當你開始這一章時，你可能沒有計劃參加某種每週支援小組。我們有同情心。處理社會問題可能很困難：人們雜亂無章，不可預測，而且常常令人討厭。與其把精力放在分析社會狀況和採取戰略行動上，不如把所有的努力都一筆勾銷。使用可預測的編譯器要容易得多，不是嗎？為什麼還要為社交活動操心呢？

Here's a quote from a famous lecture by Richard Hamming:

By taking the trouble to tell jokes to the secretaries and being a little friendly, I got superb secretarial help. For instance, one time for some idiot reason all the reproducing services at Murray Hill were tied up. Don't ask me how, but they were. I wanted something done. My secretary called up somebody at Holmdel, hopped [into] the company car, made the hour-long trip down and got it reproduced, and then came back. It was a payoff for the times I had made an effort to cheer her up, tell her jokes and be friendly; it was that little extra work that later paid off for me. By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires.

以下是理查德·哈明（Richard Hamming）著名演講中的一段話：

透過不厭其煩地給秘書們講笑話，並表現出一點友好，我得到了秘書的出色協作。例如，有一次由於某些愚蠢的原因，Murray Hill 的所有複製服務都被佔用了。別問我為什麼，但他們是。但事實就是如此。我的秘書給 Holmdel 的某個人打了電話，跳上他們公司的車，花了一個小時把它複製下來，然後回來了。我努力讓她高興起來、給她講笑話並保持友好，作為一種回報；正是這一點額外的工作後來給了我回報。透過認識到你必須使用這個系統並研究如何讓這個系統來完成你的工作，你就學會了如何使這個系統實現你的需求。

The moral is this: do not underestimate the power of playing the social game. It's not about tricking or manipulating people; it's about creating relationships to get things done. Relationships always outlast projects. When you've got richer relationships with your coworkers, they'll be more willing to go the extra mile when you need them.

寓意是：不要低估社交遊戲的力量。這不是欺騙或操縱人們；這是關於建立關係來完成事情。關係總是比專案更長久。當你和你的同事關係更融洽時，他們會更願意在你需要他們的時候幫助你。

Humility, Respect, and Trust in Practice 謙遜、尊重和信任的實踐

All of this preaching about humility, respect, and trust sounds like a sermon. Let's come out of the clouds and think about how to apply these ideas in real-life situations. We're going to examine a list of specific behaviors and examples that you can start with. Many of them might sound obvious at first, but after you begin thinking about them, you'll notice how often you (and your peers) are guilty of not following them—we've certainly noticed this about ourselves!

所有這些關於謙遜、尊重和信任的說教聽起來像是佈道。讓我們從雲端走出來，思考如何在現實生活中應用這些想法。我們將研究一系列具體的行為和例子，你可以從這些行為和例子入手。其中許多一開始聽起來很明顯，但在你開始思考它們之後，你會注意到你（和你的同齡人）經常因為沒有遵循它們而感到內疚——我們當然注意到了這一點！

Lose the ego 丟掉自負

OK, this is sort of a simpler way of telling someone without enough humility to lose their ’tude. Nobody wants to work with someone who consistently behaves like they’re the most important person in the room. Even if you know you’re the wisest person in the discussion, don’t wave it in people’s faces. For example, do you always feel like you need to have the first or last word on every subject? Do you feel the need to comment on every detail in a proposal or discussion? Or do you know somebody who does these things?

好吧，這是一種更簡單的方式，告訴那些沒有足夠謙卑的人失去他們的理智。沒有人願意和一個總是表現得像房間裡最重要的人一樣的人一起工作。即使你知道自己是討論中最聰明的人，也不要當眾揮舞。例如，你是否總是覺得你需要對每一個主題都說第一句話或最後一句話？你是否覺得有必要對提案或討論中的每一個細節進行評論？或者你認識做的人嗎？

Although it’s important to be humble, that doesn’t mean you need to be a doormat; there’s nothing wrong with self-confidence. Just don’t come off like a know-it-all. Even better, think about going for a “collective” ego, instead; rather than worrying about whether you’re personally awesome, try to build a sense of team accomplishment and group pride. For example, the Apache Software Foundation has a long history of creating communities around software projects. These communities have incredibly strong identities and reject people who are more concerned with self- promotion.

儘管謙虛很重要，但這並不意味著你需要做一個受氣包；自信沒有錯。不要表現得像無所不知。最好的是，考慮以 "集體 "的自我為目標；與其擔心你個人是否了不起，不如嘗試建立一種團隊成就感和團體自豪感。例如，Apache 軟體基金會一直致力於圍繞軟體專案建立社群。這些社群有著令人難以置信的強烈認同感，拒絕那些更關心自我宣傳的人。

Ego manifests itself in many ways, and a lot of the time, it can get in the way of your productivity and slow you down. Here’s another great story from Hamming’s lecture that illustrates this point perfectly (emphasis ours):
John Tukey almost always dressed very casually. He would go into an important office and it would take a long time before the other fellow realized that this is a first-class man and he had better listen. For a long time, John has had to overcome this kind of hostility. It’s wasted effort! I didn’t say you should conform; I said, “The appearance of conforming gets you a long way.” If you chose to assert your ego in any number of ways, “I am going to do it my way,” you pay a small steady price throughout the whole of your professional career. And this, over a whole lifetime, adds up to an enormous amount of needless trouble. […] By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires. Or you can fight it steadily, as a small, undeclared war, for the whole of your life.

自我表現在很多方面，很多時候，它會妨礙你的生產力，拖累你。下面是 Hamming 演講中的另一個精彩故事，完美地說明了這一點（重點是我們的）：

John Tukey 幾乎總是穿得很隨便。他會走進一個重要會議，過了很長一段時間，另一個人才意識到這是一個厲害的人，他最好聽從。長期以來，約翰不得不克服這種敵意。這是白費力氣！我沒說你應該服從；我說，“順從的外表讓你走得更遠。”如果你選擇以任何一種方式來主張你的自我，"我要用我的方式來做"，你就會在整個職業生涯中付出確定的小代價。而這，在整個一生中，加起來就是一個大量的不必要的麻煩。[…] 透過意識到你必須使用這個系統並研究如何讓這個系統完成你的工作，你學會了如何讓這個系統適應你的願望。或者你可以在你的一生中，作為一場小型的、不宣而戰的戰爭，穩紮穩打。

Learn to give and take criticism 學會給出和接受批評

A few years ago, Joe started a new job as a programmer. After his first week, he really began digging into the codebase. Because he cared about what was going on, he started gently questioning other teammates about their contributions. He sent simple code reviews by email, politely asking about design assumptions or pointing out places where logic could be improved. After a couple of weeks, he was summoned to his director’s office. “What’s the problem?” Joe asked. “Did I do something wrong?” The director looked concerned: “We’ve had a lot of complaints about your behavior, Joe. Apparently, you’ve been really harsh toward your teammates, criticizing them left and right. They’re upset. You need to tone it down.” Joe was utterly baffled. Surely, he thought, his code reviews should have been welcomed and appreciated by his peers. In this case, however, Joe should have been more sensitive to the team’s widespread insecurity and should have used a subtler means to introduce code reviews into the culture—perhaps even something as simple as discussing the idea with the team in advance and asking team members to try it out for a few weeks.

幾年前，喬開始了一份新工作，成為一個程式設計師。在第一週後，他真的開始鑽研程式碼庫。因為他關心正在發生的事情，他開始溫和地詢問其他隊友的提交。他透過電子郵件傳送簡單的程式碼評審，禮貌地詢問設計假設或指出可以改進邏輯的地方。幾周後，他被傳喚到主管辦公室。“怎麼了？”喬問。“我做錯什麼了嗎？”主管看起來很擔心：“喬，我們對你的行為有很多抱怨。顯然，你對你的隊友非常苛刻，到處批評他們。他們很不高興。你需要淡化它。”喬完全不知所措。當然，他認為，他的程式碼審查應該受到同行的歡迎和讚賞。然而，在這種情況下，喬應該對團隊普遍存在的不安全感更加敏感，並且應該使用更巧妙的方法將程式碼審查引入到文化中——甚至可能是一些簡單的事情，比如事先與團隊討論這個想法，並請團隊成員嘗試幾周。

In a professional software engineering environment, criticism is almost never personal—it’s usually just part of the process of making a better project. The trick is to make sure you (and those around you) understand the difference between a constructive criticism of someone’s creative output and a flat-out assault against someone’s character. The latter is useless—it’s petty and nearly impossible to act on. The former can (and should!) be helpful and give guidance on how to improve. And, most important, it’s imbued with respect: the person giving the constructive criticism genuinely cares about the other person and wants them to improve themselves or their work. Learn to respect your peers and give constructive criticism politely. If you truly respect someone, you’ll be motivated to choose tactful, helpful phrasing—a skill acquired with much practice. We cover this much more in Chapter 9.

在專業的軟體工程環境中，批評幾乎從來不是針對個人的，它通常只是建構更好專案過程的一部分。訣竅是確保你（和你周圍的人）理解對某人的創造性產出進行建設性批評和對某人的人身攻擊之間的區別。後者是無用的——它是瑣碎，幾乎不可能採取行動。前者可以（也應該！）提供幫助並指導如何改進。而且，最重要的是，它充滿了尊重：給予建設性批評的人真正關心對方，希望他們提升自己或工作。學會尊重同齡人，禮貌地提出建設性的批評。如果你真的尊重別人——這是一種透過大量實踐獲得的技能。我們在第 9 章中對此有更多的介紹。

On the other side of the conversation, you need to learn to accept criticism as well. This means not just being humble about your skills, but trusting that the other person has your best interests (and those of your project!) at heart and doesn't actually think you're an idiot. Programming is a skill like anything else: it improves with practice. If a peer pointed out ways in which you could improve your juggling, would you take it as an attack on your character and value as a human being? We hope not. In the same way, your self-worth shouldn't be connected to the code you write—or any creative project you build. To repeat ourselves: you are not your code. Say that over and over. You are not what you make. You need to not only believe it yourself, but get your coworkers to believe it, too.

另一方面，你也需要學會接受批評。這意味著不僅要對你的技能保持謙虛，還要相信對方心裡有你的最佳利益（以及你專案的利益！），而不是真的認為你是個白痴。程式設計是一項與其他技能一樣的技能：它隨著實踐而提高。如果一個同伴指出了你可以提升程式設計的方法，你會認為這是對你作為一個人的性格和價值觀的攻擊嗎？我們希望不會。同樣，你的自我價值不應該與你寫的程式碼——或你建立的任何創造性專案相聯絡。反覆說這句話。你不是你做的東西。你不僅要自己相信這一點，還要讓你的同事也相信這一點。

For example, if you have an insecure collaborator, here's what not to say: "Man, you totally got the control flow wrong on that method there. You should be using the standard xyzzy code pattern like everyone else." This feedback is full of antipatterns: you're telling someone they're "wrong" (as if the world were black and white), demanding they change something, and accusing them of creating something that goes against what everyone else is doing (making them feel stupid). Your coworker will immediately be put on the offense, and their response is bound to be overly emotional.

例如，如果你有一個不安全的合作者，下面是不應該說的話："夥計，你在那個方法上完全把控制流搞錯了。你應該像其他人一樣使用標準的 XYZY 程式碼模式。"這個反饋充滿了反模式：你告訴某人他們"錯了"（好像世界是黑白的），要求他們改變一些東西，並指責他們創造了與其他人的做法背道而馳（讓他們覺得自己很愚蠢）。你的同事會立即受到冒犯，他們的反應必然是過於情緒化的。

A better way to say the same thing might be, "Hey, I'm confused by the control flow in this section here. I wonder if the xyzzy code pattern might make this clearer and easier to maintain?" Notice how you're using humility to make the question about you, not them. They're not wrong; you're just having trouble understanding the code. The suggestion is merely offered up as a way to clarify things for poor little you while possibly helping the project's long-term sustainability goals. You're also not demanding anything—you're giving your collaborator the ability to peacefully reject the suggestion. The discussion stays focused on the code itself, not on anyone's value or coding skills.

更好的說法可能是，"嘿，我對這部分的控制流感到困惑。我想知道 XYZY 程式碼模式是否能讓這更清晰、更容易維護？"注意你是如何用謙遜來回答關於你的問題，而不是他們。他們沒有錯；你只是理解程式碼有點困難。這個建議只是作為一種方式，為你澄清事情，同時可能有助於專案的長期可持續性目標。你也沒有要求什麼——你給了你的合作者和平拒絕建議的權力。討論的重點是程式碼本身，而不是任何人的價值或編碼技能。

Fail fast and iterate 快速失敗並迭代

There's a well-known urban legend in the business world about a manager who makes a mistake and loses an impressive $10 million. He dejectedly goes into the office the next day and starts packing up his desk, and when he gets the inevitable "the wants to see you in his office" call, he trudges into the CEO's office and quietly slides a piece of paper across the desk.
"What's this?" asks the CEO.
"My resignation," says the executive. "I assume you called me in here to fire me."

“Fire you?” responds the CEO, incredulously. “Why would I fire you? I just spent $10 million training you!”[6]

商界有一個著名的城市傳奇，說的是一位經理犯了一個錯誤，損失了令人印象深刻的 1000 萬美元。第二天，他沮喪地走進辦公室，開始收拾桌子。當他接到不可抗拒的“CEO 讓你去他辦公室”電話時，他蹣跚地走進 CEO 辦公室，悄悄地把一張紙遞上桌子。
"這是什麼？"CEO 問道。
"我的辭呈，"這位經理說。"我想你叫我來是要解僱我。"
"解僱你？"執行長難以置信地回答道。"我為什麼要解僱你？我剛剛花了 1000 萬美元培訓你！"

It’s an extreme story, to be sure, but the CEO in this story understands that firing the executive wouldn’t undo the $10 million loss, and it would compound it by losing a valuable executive who he can be very sure won’t make that kind of mistake again.

的確，這是一個極端的故事，但在這個故事中，執行長明白解僱這名高管並不能挽回這 1000 萬美元的損失，而且會因為失去一位有價值的高管，讓事情變得更糟糕，他非常確定，這位高管不會再犯類似錯誤。

At Google, one of our favorite mottos is that “Failure is an option.” It’s widely recognized that if you’re not failing now and then, you’re not being innovative enough or taking enough risks. Failure is viewed as a golden opportunity to learn and improve for the next go-around.[7] In fact, Thomas Edison is often quoted as saying, “If I find 10,000 ways something won’t work, I haven’t failed. I am not discouraged, because every wrong attempt discarded is another step forward.”

在谷歌，我們最喜歡的格言之一是“失敗也是一種選擇”。我們普遍認為，如果你沒有遭遇過失敗，你就沒有足夠的創新或承擔足夠的風險的能力。失敗被視為一個黃金機會，可以在下一次嘗試中學習和改進。事實上，人們經常參考托馬斯·愛迪生的話說：“如果我發現有一萬種方法不能成功，我就沒有失敗。我並不氣餒，因為每一個被拋棄的錯誤嘗試都是向前邁出的另一步”

Over in Google X—the division that works on “moonshots” like self-driving cars and internet access delivered by balloons—failure is deliberately built into its incentive system. People come up with outlandish ideas and coworkers are actively encouraged to shoot them down as fast as possible. Individuals are rewarded (and even compete) to see how many ideas they can disprove or invalidate in a fixed period of time. Only when a concept truly cannot be debunked at a whiteboard by all peers does it proceed to early prototype.

在谷歌 X 部門——該部門負責研究自動駕駛汽車和透過熱氣球提供網際網路接入等 "登月計劃"——故意將失敗次數納入其激勵系統。人們會想出一些稀奇古怪的想法，同事們也會受到積極的鼓勵儘快實現它們。每個人都會得到獎勵（甚至是競爭），看看他們能在一段固定的時間內反駁或否定多少觀點。只有當一個概念真的不能在白板上被所有同行揭穿時，它才能進入早期原型。

6 你可以在網上找到這一傳說的十幾種變體，它們都是由不同的著名經理人創造的。

7 同樣的道理，如果你一次又一次地做同樣的事情，卻不斷失敗，那不是失敗，而是無能。

Blameless Post-Mortem Culture 無責的事後文化

The key to learning from your mistakes is to document your failures by performing a root-cause analysis and writing up a “postmortem,” as it’s called at Google (and many other companies). Take extra care to make sure the postmortem document isn’t just a useless list of apologies or excuses or finger-pointing—that’s not its purpose. A proper postmortem should always contain an explanation of what was learned and what is going to change as a result of the learning experience. Then, make sure that the postmortem is readily accessible and that the team really follows through on the proposed changes. Properly documenting failures also makes it easier for other people (present and future) to know what happened and avoid repeating history. Don’t erase your tracks—light them up like a runway for those who follow you!

從錯誤中學習的關鍵是透過進行根因分析和撰寫“事後總結”來記錄你的失敗，在谷歌（和許多其他公司）成為事後總結（國內成為覆盤）。要格外小心，確保 "事後總結 "檔案不只是一份無用的道歉、藉口或指責的清單，這不是它的目的。正確事後總結應該總是包含對所學到的內容的解釋，以及作為學習經驗作為後續的改進落地。然後，確保事後總結可以隨時查閱，並確保團隊真正貫徹執行所建議的改變。好的故障覆盤要讓其他人（現在和將來）知道發生了什麼，避免重蹈覆轍。不要抹去你的足跡——讓它們在道路上照亮給那些追隨你的人!

A good postmortem should include the following:

- A brief summary of the event
- A timeline of the event, from discovery through investigation to resolution
- The primary cause of the event
- Impact and damage assessment
- A set of action items (with owners) to fix the problem immediately
- A set of action items to prevent the event from happening again
- Lessons learned

一個好的事後總結應該包括以下內容：

- 事件的簡要概述
- 事件的時間線，從發現、調查到解決的過程
- 事件的主要原因
- 影響和損害評估
- 一套立即解決該問題的行動專案（包括執行人）。
- 一套防止事件再次發生的行動專案
- 經驗教訓

Learn patience 學會耐心

Years ago, I was writing a tool to convert CVS repositories to Subversion (and later, Git). Due to the vagaries of CVS, I kept unearthing bizarre bugs. Because my longtime friend and coworker Karl knew CVS quite intimately, we decided we should work together to fix these bugs.

幾年前，我正在編寫一個將 CVS 儲存庫轉換為 Subversion（後來是 Git）的工具。由於 CVS 的變化無常，我不斷發現各種奇怪的 bug。因為我的老朋友兼同事 Karl 非常熟悉 CVS，我們決定一起修復這些 bug。

A problem arose when we began pair programming: I'm a bottom-up engineer who is content to dive into the muck and dig my way out by trying a lot of things quickly and skimming over the details. Karl, however, is a top-down engineer who wants to get the full lay of the land and dive into the implementation of almost every method on the call stack before proceeding to tackle the bug. This resulted in some epic interpersonal conflicts, disagreements, and the occasional heated argument. It got to the point at which the two of us simply couldn't pair-program together: it was too frustrating for us both.

當我們開始結對程式設計時，一個問題出現了：我是一個自下而上的工程師，透過快速嘗試許多事情和瀏覽細節，我滿足於潛入泥潭並挖掘出路。然而，Karl 是一名自上而下的工程師，他希望在著手解決 bug 之前，全面瞭解呼叫堆疊上所有方法的實現。這導致了一些史詩般的人際衝突、分歧和偶爾的激烈爭論。到了這個地步，我們兩個人根本無法一起配對程式設計：這對我們倆來說都太令人沮喪了。

That said, we had a longstanding history of trust and respect for each other. Combined with patience, this helped us work out a new method of collaborating. We would sit together at the computer, identify the bug, and then split up and attack the problem from two directions at once (top-down and bottom-up) before coming back together with our findings. Our patience and willingness to improvise new working styles not only saved the project, but also our friendship.

儘管如此，我們對彼此的信任和尊重由來已久。再加上耐心，這幫助我們制定了一個新的合作方法。我們會一起坐在電腦前，找出 bug，然後從兩個方向（自上而下和自下而上）拆分並同時解決問題，然後再返回繼續查詢 bug。我們的耐心和即興創作新工作方式的意願不僅挽救了這個專案，而且也挽救了我們的友誼。

Be open to influence 接受影響

The more open you are to influence, the more you are able to influence; the more vulnerable you are, the stronger you appear. These statements sound like bizarre contradictions. But everyone can think of someone they've worked with who is just maddeningly stubborn—no matter how much people try to persuade them, they dig their heels in even more. What eventually happens to such team members? In our experience, people stop listening to their opinions or objections; instead, they end up "routing around" them like an obstacle everyone takes for granted. You certainly don't want to be that person, so keep this idea in your head: it's OK for someone else to change your mind. In the opening chapter of this book, we said that engineering is inherently about trade-offs. It's impossible for you to be right about everything all the time unless you have an unchanging environment and perfect knowledge, so of course you should change your mind when presented with new evidence. Choose your battles carefully: to be heard properly, you first need to listen to others. It's better to do this listening before putting a stake in the ground or firmly announcing a decision—if you're constantly changing your mind, people will think you're wishy-washy.

你對影響的態度越開放，你就越能影響。越是脆弱，越是強勢。這些說法聽起來相互矛盾。但每個人都能想到他們曾經共事過的人，他們的固執讓人抓狂——無論人們如何勸說他們，他們都會更加鑽牛角尖。這樣的團隊成員最終會發生什麼？根據我們的經驗，人們不再聽取他們的任何意見，不論是贊同還是反對的意見；如果，這些固執的人解決了自己固執的問題，那麼他們解決了大家都認為是障礙問題。你當然不希望成為這樣的人，所以要把這個想法記在腦子裡： 別人可以改變你的想法。在本書的開始，我們說過，工程本質上是關於權衡的。 除非你有一個不變的環境和完美的知識，否則你不可能一直對所有事情都是正確的，所以當有新的證據時，你當然應該改變你的最初想法。謹慎選擇你的戰鬥：要想讓別人正確地聽取你的意見，你首先需要傾聽別人的意見。最好在下定決心或堅定地宣佈決定之前進行傾聽——如果你不斷地改變主意，人們會認為你不堅定。

The idea of vulnerability can seem strange, too. If someone admits ignorance of the topic at hand or the solution to a problem, what sort of credibility will they have in a group? Vulnerability is a show of weakness, and that destroys trust, right?

脆弱性的想法似乎也很奇怪。如果有人承認不知道手頭的話題或問題的解決方案，那麼他們在團隊中會有什麼樣的可信度？脆弱是軟弱的表現，這會破壞信任，對嗎？

Not true. Admitting that you’ve made a mistake or you’re simply out of your league can increase your status over the long run. In fact, the willingness to express vulnerability is an outward show of humility, it demonstrates accountability and the willingness to take responsibility, and it’s a signal that you trust others’ opinions. In return, people end up respecting your honesty and strength. Sometimes, the best thing you can do is just say, “I don’t know.”

並非如此。從長遠來看，承認自己犯了錯誤，或者根本不在你的能力範圍，都會提高你的地位。事實上，表達脆弱性的意願是一種謙遜的外在表現，它表明了責任感和承擔責任的意願，也是你信任他人意見的訊號。作為回報，人們最終會尊重你的誠實和力量。有時，你能做的最好的事情就是說，"我不知道"。

Professional politicians, for example, are notorious for never admitting error or ignorance, even when it’s patently obvious that they’re wrong or unknowledgeable about a subject. This behavior exists primarily because politicians are constantly under attack by their opponents, and it’s why most people don’t believe a word that politicians say. When you’re writing software, however, you don’t need to be continually on the defensive—your teammates are collaborators, not competitors. You all have the same goal.

例如，職業政客因從不承認錯誤或無知而臭名昭著，即使他們在某個問題上顯然是錯的或無知的。存在這種行為主要是因為政客們經常受到對手的攻擊，這就是為什麼大多數人不相信政客們說的一個字。然而，當你在編寫軟體時，你不需要不斷地保持防備狀態——你的團隊成員是合作者，而不是競爭對手。你們都有相同的目標。

Being Googley 成為谷歌範

At Google, we have our own internal version of the principles of “humility, respect, and trust” when it comes to behavior and human interactions.

在谷歌，當涉及到行為和人際交往時，我們有自己的內部版本的“謙遜、尊重和信任”原則。

From the earliest days of our culture, we often referred to actions as being “Googley” or “not Googley.” The word was never explicitly defined; rather, everyone just sort of took it to mean “don’t be evil” or “do the right thing” or “be good to each other.” Over time, people also started using the term “Googley” as an informal test for culture-fit whenever we would interview a candidate for an engineering job, or when writing internal performance reviews of one another. People would often express opinions about others using the term; for example, “the person coded well, but didn’t seem to have a very Googley attitude.”

從我們早期文化開始，我們經常將行為稱為“谷歌的”或“不是谷歌的”；相反，每個人都把它理解為“不作惡”、“做正確的事”或“善待彼此”。隨著時間的推移，人們也開始使用“Googley”一詞作為一種非正式的文化契合度測試，無論是在我們面試一名工程師崗位的應聘者時，還是在撰寫彼此的內部績效評估時。人們經常會用這個詞表達對他人的看法；例如，“這個人編碼很好，但似乎沒有很好的 Googley 態度。”

Of course, we eventually realized that the term “Googley” was being overloaded with meaning; worse yet, it could become a source of unconscious bias in hiring or evaluations. If “Googley” means something different to every employee, we run the risk of the term starting to mean “is just like me.” Obviously, that’s not a good test for hiring
—we don’t want to hire people “just like me,” but people from a diverse set of backgrounds and with different opinions and experiences. An interviewer’s personal desire to have a beer with a candidate (or coworker) should never be considered a valid signal about somebody else’s performance or ability to thrive at Google.

當然，我們最終意識到“Googley”這個詞的含義太多了；更糟糕的是，它可能成為招聘或評估中無意識偏見的來源。如果“Googley”對每個員工都有不同的含義，那麼我們就有可能把“Googley”這個詞的意思改成“和我一樣”。顯然，這不是一個好的招聘模式——我們不希望招聘 "和我一樣 "的人，而是來自不同背景、有不同意見和經驗的人。面試官想和候選人（或同事）一起喝啤酒的願望，絕不應該被認為是關於其他人的表現或在谷歌發展的能力的有效訊號。

Google eventually fixed the problem by explicitly defining a rubric for what we mean by “Googleyness”—a set of attributes and behaviors that we look for that represent strong leadership and exemplify “humility, respect, and trust”:

- *Thrives in ambiguity*
  Can deal with conflicting messages or directions, build consensus, and make progress against a problem, even when the environment is constantly shifting.
- *Values feedback*
  Has humility to both receive and give feedback gracefully and understands how valuable feedback is for personal (and team) development.
- *Challenges status quo*
  Is able to set ambitious goals and pursue them even when there might be resistance or inertia from others.
- *Puts the user first*
  Has empathy and respect for users of Google’s products and pursues actions that are in their best interests.
- *Cares about the team*
  Has empathy and respect for coworkers and actively works to help them without being asked, improving team cohesion.
- *Does the right thing*
  Has a strong sense of ethics about everything they do; willing to make difficult or inconvenient decisions to protect the integrity of the team and product.

谷歌最終解決了這個問題，明確定義了我們所說的“谷歌特質”（Googleyness）——我們所尋找的一套屬性和行為，代表了強大的領導力，體現了 "謙遜、尊重和信任"：

- *在模稜兩可中茁壯成長*
  即使在環境不斷變化的情況下，也能處理相互衝突的資訊或方向，建立共識，並對問題做出改進。
- *重視反饋*
  謙虛優雅地接受和給出反饋，理解反饋對個人（和團隊）發展的價值。
- *走出舒適區*
  能夠設定宏偉的目標並去追求，即使有來自他人的抵制或惰性。
- *客戶第一*
  對谷歌產品的使用者抱有同情和尊重，並追求符合其最佳利益的行動。
- *關心團隊*
  對同事抱有同情心和尊重，並積極主動地幫助他們，提高團隊凝聚力。
- *做正確的事*
  對自己所做的一切有強烈的主人感；願意做出困難或不易的決定以保護團隊和產品的完整。

Now that we have these best-practice behaviors better defined, we've begun to shy away from using the term "Googley." It's always better to be specific about expectations!

現在我們有了這些最佳實踐行為的更好定義，我們已經開始避免使用 "Googley "這個詞了。更好的是對期望值有一個具體的說明。

Conclusion 結論

The foundation for almost any software endeavor—of almost any size—is a well- functioning team. Although the Genius Myth of the solo software developer still persists, the truth is that no one really goes it alone. For a software organization to stand the test of time, it must have a healthy culture, rooted in humility, trust, and respect that revolves around the team, rather than the individual. Further, the creative nature of software development requires that people take risks and occasionally fail; for people to accept that failure, a healthy team environment must exist.

幾乎任何規模的軟體工作的基礎都是一個運作良好的團隊。儘管軟體開發者單打獨鬥的 "天才神話 "仍然存在，但事實是，沒有人能夠真正地單幹。一個軟體組織要想經受住時間的考驗，就必須有一種健康的文化，植根於謙遜、信任和尊重，圍繞著團隊而不是個人。此外，軟體開發的創造性要求人們承擔風險並偶爾失敗；為了讓人們接受這種失敗，必須有一個健康的團隊環境。

TL;DRs 內容提要

- Be aware of the trade-offs of working in isolation.
- Acknowledge the amount of time that you and your team spend communicating and in interpersonal conflict. A small investment in understanding personalities and working styles of yourself and others can go a long way toward improving productivity.
- If you want to work effectively with a team or a large organization, be aware of your preferred working style and that of others.
- 意識到孤立工作的得失。

- 認識到你和你的團隊花在溝通和人際衝突上的時間。在瞭解自己和他人的個性和工作風格方面進行少量投入，對提高生產力有很大幫助。
- 如果你想在一個團隊或一個大型組織中有效地工作，要意識到你和其他人喜歡的工作風格。

---

1. Ben Collins-Sussman, also an author within this book. ↩

2. Literally, if you are, in fact, a bike designer. ↩

3. I should note that sometimes it's dangerous to get too much feedback too early in the process if you're still unsure of your general direction or goal. ↩

4. I do, however, acknowledge that serious introverts likely need more peace, quiet, and alone time than most people and might benefit from a quieter environment, if not their own office. ↩

5. This is incredibly difficult if you've been burned in the past by delegating to incompetent people./ ↩

6. You can find a dozen variants of this legend on the web, attributed to different famous managers. ↩

7. By the same token, if you do the same thing over and over and keep failing, it's not failure, it's incompetence. ↩