CHAPTER 11

Testing Overview

第十一章 測試概述

Written by Adam Bender

Edited by Tom Manshreck

Testing has always been a part of programming. In fact, the first time you wrote a computer program you almost certainly threw some sample data at it to see whether it performed as you expected. For a long time, the state of the art in software testing resembled a very similar process, largely manual and error prone. However, since the early 2000s, the software industry's approach to testing has evolved dramatically to cope with the size and complexity of modern software systems. Central to that evolution has been the practice of developer-driven, automated testing.

測試一直是程式設計的一部分。事實上，當你第一次編寫計算機程式時，你幾乎肯定向它丟擲一些樣本資料，看看它是否按照你的預期執行。在很長一段時間裡，軟體測試的技術水平類似於一個非常相近的過程，主要是手動且容易出錯的。然而，自 21 世紀初以來，軟體行業的測試方法已經發生了巨大的變化，以應對現代軟體系統的規模和複雜性。這種演進的核心是開發人員驅動的自動化測試實踐。

Automated testing can prevent bugs from escaping into the wild and affecting your users. The later in the development cycle a bug is caught, the more expensive it is; exponentially so in many cases.[1] However, "catching bugs" is only part of the motivation. An equally important reason why you want to test your software is to support the ability to change. Whether you're adding new features, doing a refactoring focused on code health, or undertaking a larger redesign, automated testing can quickly catch mistakes, and this makes it possible to change software with confidence.

自動化測試可以防止 bug 外逃並影響使用者。開發週期越晚發現 bug，成本就越高；在許多情況下都是指數級的高。然而，"捕捉 bug"只是動機的一部分。你希望測試軟體的一個同樣重要的原因是支援更改的能力。無論你是在新增新功能、進行以程式碼健康為重點的重構，還是進行更大規模的重新設計，自動化測試都可以快速發現錯誤，這使得有信心地更改軟體成為可能。

Companies that can iterate faster can adapt more rapidly to changing technologies, market conditions, and customer tastes. If you have a robust testing practice, you needn't fear change—you can embrace it as an essential quality of developing software. The more and faster you want to change your systems, the more you need a fast way to test them.

迭代速度更快的公司可以更快地適應不斷變化的技術、市場條件和客戶需求。如果你有一個健壯的測試實踐，你不必害怕改變，你可以把它作為開發軟體的基本品質。你越想高效改變你的系統，你就越需要一種快速的方法來測試它們。

The act of writing tests also improves the design of your systems. As the first clients of your code, a test can tell you much about your design choices. Is your system too tightly coupled to a database? Does the API support the required use cases? Does your system handle all of the edge cases? Writing automated tests forces you to confront these issues early on in the development cycle. Doing so generally leads to more modular software that enables greater flexibility later on.

編寫測試的行為也改善了你的系統設計。作為你的程式碼的第一個客戶，測試可以告訴你很多關於設計選擇的資訊。你的系統是否與資料庫結合得太緊密了？API 是否支援所需的使用案例？你的系統是否能處理所有的邊界情況？編寫自動化測試迫使你在開發週期的早期就面對這些問題。這樣做通常會導致更多的模組化軟體，使以後有更大的靈活性。

Much ink has been spilled about the subject of testing software, and for good reason: for such an important practice, doing it well still seems to be a mysterious craft to many. At Google, while we have come a long way, we still face difficult problems getting our processes to scale reliably across the company. In this chapter, we'll share what we have learned to help further the conversation.

關於測試軟體的話題，人們已經傾注了大量的筆墨，這是有充分理由的：對於如此重要的實踐，對許多人來說，把它做好似乎仍然是一門神秘的技藝。在谷歌，雖然我們已經取得了長足的進步，但我們仍然面臨著讓流程在整個公司內可靠擴充的難題。在本章中，我們將分享我們所學到的有助於進一步對話的知識。

> 1 請參閱"缺陷預防：降低成本和提高品質"

Why Do We Write Tests? 為什麼我們要編寫測試？

To better understand how to get the most out of testing, let's start from the beginning. When we talk about automated testing, what are we really talking about?

為了更好地理解如何最大限度地利用測試，讓我們從頭開始。當我們談論自動化測試時，我們真正談論的是什麼？

The simplest test is defined by:

- A single behavior you are testing, usually a method or API that you are calling
- A specific input, some value that you pass to the API
- An observable output or behavior
- A controlled environment such as a single isolated process

最簡單的測試定義如下：

- 你要測試的單一行為，通常是你要呼叫的一個方法或 API
- 一個特定的輸入，你傳遞給 API 的一些值
- 可觀察的輸出或行為
- 受控的環境，如一個單獨的隔離過程

When you execute a test like this, passing the input to the system and verifying the output, you will learn whether the system behaves as you expect. Taken in aggregate, hundreds or thousands of simple tests (usually called a *test suite*) can tell you how well your entire product conforms to its intended design and, more important, when it doesn't.

當你執行這樣的測試時，將輸入傳給系統並驗證輸出，你將瞭解到系統是否期望執行。總的來說，成百上千個簡單的測試（通常稱為*測試套件*）可以告訴你整個產品符合預期設計的程度，更重要的是，何時不符合預期設計。

Creating and maintaining a healthy test suite takes real effort. As a codebase grows, so too will the test suite. It will begin to face challenges like instability and slowness. A failure to address these problems will cripple a test suite. Keep in mind that tests derive their value from the trust engineers place in them. If testing becomes a productivity sink, constantly inducing toil and uncertainty, engineers will lose trust and begin to find workarounds. A bad test suite can be worse than no test suite at all.

建立和維護一個健康的測試套件需要付出艱辛的努力。隨著程式碼庫的增長，測試套件也會增長。它將開始面臨諸如不穩定和執行緩慢的挑戰。如果不能解決這些問題，測試套件將被削弱。請記住，測試的價值來自工程師對測試的信任。如果測試成為生產力的短板，不斷地帶來辛勞和不確定性，工程師將失去信任並開始尋找解決辦法。一個糟糕的測試套件可能比根本沒有測試套件更糟糕。

In addition to empowering companies to build great products quickly, testing is becoming critical to ensuring the safety of important products and services in our lives. Software is more involved in our lives than ever before, and defects can cause more than a little annoyance: they can cost massive amounts of money, loss of property, or, worst of all, loss of life.[2]

除了使公司能夠快速生產出優秀的產品外，測試對於確保我們生活中重要產品和服務的安全也變得至關重要。軟體比以往任何時候都更多地參與到我們的生活中，而缺陷可能會造成更多的煩惱：它們可能會造成大量的金錢損失，財產損失，或者最糟糕的是，生命損失。

At Google, we have determined that testing cannot be an afterthought. Focusing on quality and testing is part of how we do our jobs. We have learned, sometimes painfully, that failing to build quality into our products and services inevitably leads to bad outcomes. As a result, we have built testing into the heart of our engineering culture.

在谷歌，我們已經確定測試不能是事後諸葛亮。關注品質和測試是我們工作的一部分。我們已經瞭解到，有時是痛苦地認識到，未能將品質融入我們的產品和服務不可避免地會導致糟糕的結果。因此，我們將測試融入了我們工程文化的核心。

> 2 參見"達蘭的失敗"

The Story of Google Web Server 谷歌網路伺服器的故事

In Google's early days, engineer-driven testing was often assumed to be of little importance. Teams regularly relied on smart people to get the software right. A few systems ran large integration tests, but mostly it was the Wild West. One product in particular seemed to suffer the worst: it was called the Google Web Server, also known as GWS.

在谷歌的早期，工程師驅動的測試往往被認為是不重要的。團隊經常依靠聰明的人把軟體寫正確。有幾個系統進行了大規模的整合測試，但大多數情況下，這是一個狂野的美國西部。有一個產品似乎受到了最嚴重的影響：它被稱為谷歌網路伺服器，也被稱為 GWS。

GWS is the web server responsible for serving Google Search queries and is as important to Google Search as air traffic control is to an airport. Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically. Releases were becoming buggier, and it was taking longer and longer to push them out. Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working in production. (At one point, more than 80% of production pushes contained user-affecting bugs that had to be rolled back.)

GWS 是負責為谷歌搜尋查詢提供服務的網路伺服器，它對谷歌搜尋的重要性就像空中交通管制對機場的重要性一樣。早在 2005 年，隨著專案規模和複雜性的增加，生產效率急劇下降。釋出的版本越來越多的錯誤，推送的時間也越來越長。團隊成員在對服務進行修改時信心不足，往往是在功能停止工作時才發現有問題。(有一次，超過 80%的生產推送包含了影響使用者的 bug，不得不回滾）。

To address these problems, the tech lead (TL) of GWS decided to institute a policy of engineer-driven, automated testing. As part of this policy, all new code changes were required to include tests, and those tests would be run continuously. Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred despite the fact that the project was seeing a record number of new changes every quarter. Even in the face of unprecedented growth and change, testing brought renewed productivity and confidence to one of the most critical projects at Google. Today, GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.

為了解決這些問題，GWS 的技術負責人（TL）決客製定一項由工程師驅動的自動化測試策略。作為這項策略的一部分，所有新的程式碼修改都需要包括測試，而且這些測試將被持續執行。在實行這一策略的一年內，緊急推送的數量*下降了一半*。儘管該專案每季度都有創紀錄的新改動，但還是出現了這種下降。即使面對前所未有的增長和變化，測試也給谷歌最關鍵的專案之一帶來了新的生產力和信心。如今，GWS 幾乎每天都有數萬個測試和釋出，幾乎沒有客戶可見的故障。

The changes in GWS marked a watershed for testing culture at Google as teams in other parts of the company saw the benefits of testing and moved to adopt similar tactics.

GWS 的變化標誌著谷歌測試文化的一個分水嶺，因為公司其他部門的團隊看到了測試的好處，並開始採用類似的策略。

One of the key insights the GWS experience taught us was that you can't rely on programmer ability alone to avoid product defects. Even if each engineer writes only the occasional bug, after you have enough people working on the same project, you will be swamped by the ever-growing list of defects. Imagine a hypothetical 100-person team whose engineers are so good that they each write only a single bug a month. Collectively, this group of amazing engineers still produces five new bugs every workday. Worse yet, in a complex system, fixing one bug can often cause another, as engineers adapt to known bugs and code around them.

GWS 的經驗告訴我們的一個重要啟示是，你不能僅僅依靠程式設計師的能力來避免產品缺陷。即使每個工程師只是偶爾寫一些 bug，當你有足夠多的人在同一個專案上工作時，你也會被不斷增長的缺陷列表所淹沒。想象一下，一個假設的 100 人的團隊，其工程師非常優秀，他們每個人每月只寫一個 bug。而這群了不起的工程師在每個工作日仍然會產生 5 個新的 bug。更糟糕的是，在一個複雜的系統中，修復一個錯誤往往會導致另一個錯誤，因為工程師們會適配已知的 bug 並圍繞它們編寫程式碼。

The best teams find ways to turn the collective wisdom of its members into a benefit for the entire team. That is exactly what automated testing does. After an engineer on the team writes a test, it is added to the pool of common resources available to others. Everyone else on the team can now run the test and will benefit when it detects an issue. Contrast this with an approach based on debugging, wherein each time a bug occurs, an engineer must pay the cost of digging into it with a

debugger. The cost in engineering resources is night and day and was the fundamental reason GWS was able to turn its fortunes around.

最好的團隊會想辦法將其成員的集體智慧轉化為整個團隊的利益。這正是自動化測試的作用。在團隊中的一個工程師寫了一個測試後，它被新增到其他人可用的公共資源池中。團隊中的每個人現在都可以執行這個測試，當它檢測到一個問題時，就會受益。這與基於除錯的方法形成鮮明對比，在這種方法中，每次出現 bug，工程師都必須付出代價，用偵錯程式去深挖 bug。工程資源的成本是夜以繼日的，是 GWS 能夠扭轉其命運的根本原因。

Testing at the Speed of Modern Development 以現代開發的速度測試

Software systems are growing larger and ever more complex. A typical application or service at Google is made up of thousands or millions of lines of code. It uses hundreds of libraries or frameworks and must be delivered via unreliable networks to an increasing number of platforms running with an uncountable number of configurations. To make matters worse, new versions are pushed to users frequently, sometimes multiple times each day. This is a far cry from the world of shrink-wrapped software that saw updates only once or twice a year.

軟體系統正在變得越來越大，越來越複雜。谷歌的一個典型的應用程式或服務是由數千或數百萬行程式碼組成的。它使用數以百計的函式庫或框架，必須透過不可靠的網路傳遞到越來越多的平臺上，並以難以計數的配置執行。更糟糕的是，新版本被頻繁地推送給使用者，有時每天推送多次。這與每年只更新一到兩次的壓縮套件安裝的軟體世界相去甚遠。

The ability for humans to manually validate every behavior in a system has been unable to keep pace with the explosion of features and platforms in most software. Imagine what it would take to manually test all of the functionality of Google Search, like finding flights, movie times, relevant images, and of course web search results (see Figure 11-1). Even if you can determine how to solve that problem, you then need to multiply that workload by every language, country, and device Google Search must support, and don't forget to check for things like accessibility and security. Attempting to assess product quality by asking humans to manually interact with every feature just doesn't scale. When it comes to testing, there is one clear answer: automation.

人工手動驗證系統中每一個行為的能力已經無法跟上大多數軟體中功能和平台的爆炸性增長的步伐。想象一下，要手動測試谷歌搜尋的所有功能，比如尋找航班、電影時間、相關圖片，當然還有網頁搜尋結果（見圖 11-1），需要花費多少時間。即使你能確定如何解決這個問題，你也需要把這個工作量乘以谷歌搜尋必須支援的每一種語言、國家和裝置，而且別忘了檢查諸如可及性和安全性。試圖透過要求人工手動與每個功能互動來評估產品品質是不可行的。當涉及到測試時，有一個明確的答案：自動化。

Google sfo to london

All Flights Maps Images Shopping More Settings Tools

About 15,500,000 results (1.25 seconds)

Flights from San Francisco, CA (SFO) to London, United Kingdom (all airports)

Sponsored

www.google.com/flights

San Francisco, CA (SFO) | London, United Kingdom (all airports)

Sun, September 15 | Mon, September 30

Multiple airlines	10h 10m+	Connecting	from $353
British Airways	13h 35m+	Connecting	from $365
Multiple airlines	10h 15m	Nonstop	from $422
United	10h 30m	Nonstop	from $422
Delta	10h 15m	Nonstop	from $628
Virgin Atlantic	10h 15m	Nonstop	from $628
Air France	10h 15m	Nonstop	from $629
KLM	10h 15m	Nonstop	from $629
Lufthansa	10h 30m	Nonstop	from $692
Austrian	10h 30m	Nonstop	from $692
Brussels Airlines	10h 30m	Nonstop	from $692
Norwegian Air UK	10h 10m	Nonstop	from $760
American	10h 25m	Nonstop	from $939
British Airways	10h 25m	Nonstop	from $939
Iberia	10h 25m	Nonstop	from $939
Other airlines	10h 15m+	Connecting	from $415

Search flights

Cheap Flights from San Francisco to London from $347 - KAYAK

https://www.kayak.com › Flights › Worldwide › Europe › United Kingdom › England

Fly from San Francisco to **London** on Air Canada from $347, Finnair from $348, Lufthansa from $363... Search ... **SFO** - LHR San Francisco - **London** Heathrow ...

How does KAYAK find such low flight prices?

How can Hacker Fares save me money?

Does KAYAK query more flight providers than competitors?

Show more

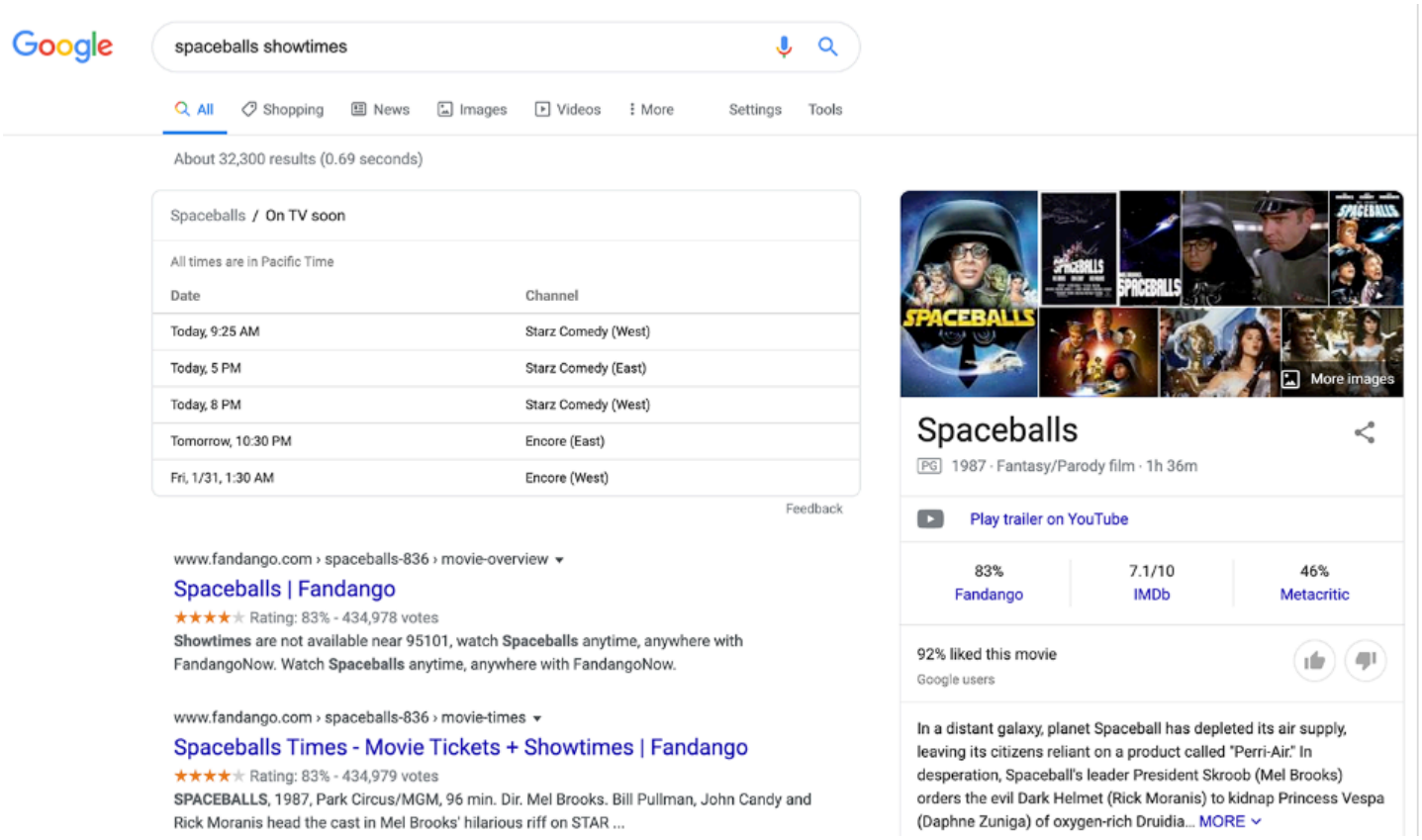

Figure 11-1. Screenshots of two complex Google search results 圖 11-1. 兩個複雜的谷歌搜尋結果的截圖

Write, Run, React 編寫、執行、反應

In its purest form, automating testing consists of three activities: writing tests, running tests, and reacting to test failures. An automated test is a small bit of code, usually a single function or method, that calls into an isolated part of a larger system that you want to test. The test code sets up an expected environment, calls into the system, usually with a known input, and verifies the result. Some of the tests are very small, exercising a single code path; others are much larger and can involve entire systems, like a mobile operating system or web browser.

在最純粹的形式中，自動化測試包括三個活動：編寫測試、執行測試和對測試失敗作出反應。自動測試是一小段程式碼，通常是單個函式或方法，它呼叫要測試的較大系統的一個獨立部分。測試程式碼設定預期的環境，呼叫系統（通常使用已知的輸入），並驗證結果。有些測試非常小，只執行一條程式碼方式；另一些則要大得多，可能涉及整個系統，如移動作業系統或 web 瀏覽器。

Example 11-1)presents a deliberately simple test in Java using no frameworks or testing libraries. This is not how you would write an entire test suite, but at its core every automated test looks similar to this very simple example.

例 11-1 紹了一個特意用 Java 寫的簡單測試，沒有使用框架或測試庫。這不是你編寫整個測試套件的方式，但在其核心部分，每個自動化測試都與這個非常簡單的例子類別似。

Example 11-1. An example test 例 11-1. 一個測試例項

```
// Verifies a Calculator class can handle negative results.
public void main(String[] args) {
  Calculator calculator = new Calculator();
  int expectedResult = -3;
  int actualResult = calculator.subtract(2, 5); // Given 2, Subtracts 5.
  assert(expectedResult == actualResult);
}
```

Unlike the QA processes of yore, in which rooms of dedicated software testers pored over new versions of a system, exercising every possible behavior, the engineers who build systems today play an active and integral role in writing and running automated tests for their own code. Even in companies where QA is a prominent organization, developer-written tests are commonplace. At the speed and scale that today's systems are being developed, the only way to keep up is by sharing the development of tests around the entire engineering staff.

與過去的 QA 過程不同的是，在過去的 QA 過程中，專門的軟體測試人員仔細研究系統的新版本，練習各種可能的行為，而今天建構系統的工程師在為自己的程式碼編寫和執行自動測試方面發揮著積極和不可或缺的作用。即使在 QA 是一個重要組織的公司，開發人員編寫的測試也是很常見的。以當今系統開發的速度和規模，跟上的唯一方法是與整個工程人員共享測試開發。

Of course, writing tests is different from writing *good tests*. It can be quite difficult to train tens of thousands of engineers to write good tests. We will discuss what we have learned about writing good tests in the chapters that follow.

當然，寫測試和寫*好測試*是不同的。要訓練數以萬計的工程師寫出好的測試是相當困難的。我們將在接下來的章節中討論關於編寫好測試的知識。

Writing tests is only the first step in the process of automated testing. After you have written tests, you need to run them. Frequently. At its core, automated testing consists of repeating the same action over and over, only requiring human attention when something breaks. We will discuss this Continuous Integration (CI) and testing in Chapter 23. By expressing tests as code instead of a manual series of steps, we can run them every time the code changes—easily thousands of times per day. Unlike human testers, machines never grow tired or bored.

編寫測試只是自動化測試過程中的第一步。編寫測試後，需要執行它們。頻繁地自動化測試的核心是一遍又一遍地重複相同的操作，只有在出現故障時才需要人的注意。我們將在第 23 章討論這種持續整合（CI）和測試。透過將測試表達為程式碼，而不是手動的一系列步驟，我們可以在每次程式碼改變時執行它們——每天很容易地執行數千次。與人工測試人員不同，機器從不感到疲勞或無聊。

Another benefit of having tests expressed as code is that it is easy to modularize them for execution in various environments. Testing the behavior of Gmail in Firefox requires no more effort than doing so in Chrome, provided you have configurations for both of these systems.^3 Running tests for a user interface (UI) in Japanese or German can be done using the same test code as for English.

將測試轉變為程式碼的另一個好處是，很容易將它們模組化，以便在各種環境中執行。在 Firefox 中測試 Gmail 的行為並不需要比在 Chrome 中測試更多的努力，只要你有這兩個系統的配置。可以使用與英語相同的測試程式碼對日語或德語使用者介面（UI）進行測試。

Products and services under active development will inevitably experience test failures. What really makes a testing process effective is how it addresses test failures. Allowing failing tests to pile up quickly defeats any value they were providing, so it is imperative not to let that happen. Teams that prioritize fixing a broken test within minutes of a failure are able to keep confidence high and failure isolation fast, and therefore derive more value out of their tests.

正在開發的產品和服務將不可避免地經歷測試失敗。真正使測試過程有效的是如何解決測試失敗。允許失敗的測試迅速堆積起來，就會破壞它們提供的任何價值，所以一定不要讓這種情況發生。在發生故障後幾分鐘內優先修復故障測試的團隊能夠保持較高的信心和快速的故障隔離，從而從他們的測試中獲得更多的價值。

In summary, a healthy automated testing culture encourages everyone to share the work of writing tests. Such a culture also ensures that tests are run regularly. Last, and perhaps most important, it places an emphasis on fixing broken tests quickly so as to maintain high confidence in the process.

總之，一個健康的自動化測試文化鼓勵每個人分享編寫測試的工作。這種文化也確保了測試的定期執行。最後，也許也是最重要的，它強調快速修復損壞的測試，以保持對測試過程的高度信心。

> 3 在不同的瀏覽器和語言中獲得正確的行為是一個不同的說法! 但是，理想情況下，終端使用者的體驗對每個人來說都應該是一樣的。

Benefits of Testing Code 測試程式碼的好處

To developers coming from organizations that don't have a strong testing culture, the idea of writing tests as a means of improving productivity and velocity might seem antithetical. After all, the act of writing tests can take just as long (if not longer!) than implementing a feature would take in the first place. On the contrary, at Google, we've found that investing in software tests provides several key benefits to developer productivity:

- *Less debugging*
  As you would expect, tested code has fewer defects when it is submitted. Critically, it also has fewer defects throughout its existence; most of them will be caught before the code is submitted. A piece of code at Google is expected to be modified dozens of times in its lifetime. It will be changed by other teams and even automated code maintenance systems. A test written once continues to pay dividends and prevent costly defects and annoying debugging sessions through the lifetime of the project. Changes to a project, or the dependencies of a project, that break a test can be quickly detected by test infrastructure and rolled back before the problem is ever released to production.
- *Increased confidence in changes*
  All software changes. Teams with good tests can review and accept changes to their project with confidence because all important behaviors of their project are continuously verified. Such projects encourage refactoring. Changes that refactor code while preserving existing behavior should (ideally) require no changes to existing tests.

- *Improved documentation*
  Software documentation is notoriously unreliable. From outdated requirements to missing edge cases, it is common for documentation to have a tenuous relationship to the code. Clear, focused tests that exercise one behavior at a time function as executable documentation. If you want to know what the code does in a particular case, look at the test for that case. Even better, when requirements change and new code breaks an existing test, we get a clear signal that the "documentation" is now out of date. Note that tests work best as documentation only if care is taken to keep them clear and concise.
- *Simpler reviews*
  All code at Google is reviewed by at least one other engineer before it can be submitted (see Chapter 9 for more details). A code reviewer spends less effort verifying code works as expected if the code review includes thorough tests that demonstrate code correctness, edge cases, and error conditions. Instead of the tedious effort needed to mentally walk each case through the code, the reviewer can verify that each case has a passing test.
- *Thoughtful design*
  Writing tests for new code is a practical means of exercising the API design of the code itself. If new code is difficult to test, it is often because the code being tested has too many responsibilities or difficult-to-manage dependencies. Well-designed code should be modular, avoiding tight coupling and focusing on specific responsibilities. Fixing design issues early often means less rework later.
- *Fast, high-quality releases*
  With a healthy automated test suite, teams can release new versions of their application with confidence. Many projects at Google release a new version to production every day—even large projects with hundreds of engineers and thousands of code changes submitted every day. This would not be possible without automated testing.

對於來自沒有強大測試文化的組織的開發者來說，把編寫測試作為提高生產力和速度的手段的想法可能看起來是對立的。畢竟，編寫測試所需的時間（如果不是更長的話！）可能與實現功能所需的時間一樣長。相反，在谷歌，我們發現投入於軟體測試對開發人員的生產力有幾個關鍵的好處：

- *更少的除錯*
  正如你所期望的那樣，經過測試的程式碼在提交時有更少的缺陷。重要的是，它在整個存在過程中也有較少的缺陷；大多數缺陷在程式碼提交之前就會被發現。在谷歌，一段程式碼在其生命週期內預計會被修改幾十次。它將被其他團隊甚至是自動程式碼維護系統所改變。一次寫好的測試會繼續帶來紅利，並在專案的生命週期中防止昂貴的缺陷和惱人的除錯過程。對專案或專案的依賴關係的改變，如果破壞了測試，可以被測試基礎設施迅速發現，並在問題被髮布到生產中之前回滾。
- *在變更中增加了信心*
  所有的軟體變更。具有良好測試的團隊可以滿懷信心地審查和接受專案的變更，因為他們的專案的所有重要行為都得到了持續驗證。這樣的專案鼓勵重構。在保留現有行為的情況下，重構程式碼的變化應該（最好）不需要改變現有的測試。
- *改進文件*
  軟體文件是出了名的不可靠。從過時的需求到缺失的邊緣案例，文件與程式碼的關係很脆弱，這很常見。清晰的、有針對性的測試，一次行使一個行為的功能是可執行的文件。如果你想知道程式碼在某一特定情況下做了什麼，看看該情況的測試。更好的是，當需求發生變化，新的程式碼破壞了現有的測試時，我們會得到一個明確的訊號，"文件 "現在已經過時了。請注意，只有在注意保持測試的清晰和簡潔的情況下，測試才能作為文件發揮最佳效果。

- *簡單審查*
  在谷歌，所有的程式碼在提交之前都要經過至少一名其他工程師的審查（詳見第九章）。如果程式碼審查包括徹底的測試，證明程式碼的正確性、邊緣情況和錯誤情況，那麼程式碼審查員花在驗證程式碼是否按預期執行的精力就會減少。審查員可以驗證每個案例都有一個合格的測試，而不必費心費力地在程式碼中對每個案例進行解讀。
- *深思熟慮設計*
  為新程式碼編寫測試是鍛鍊程式碼本身的 API 設計的一種實用手段。如果新程式碼難以測試，往往是因為被測試的程式碼有太多的職責或難以管理的依賴關係。設計良好的程式碼應該是模組化的，避免緊密耦合，並專注於特定的責任。儘早修復設計問題往往意味著以後的返工更少。
- *快速、高品質的釋出*
  有了健康的自動化測試套件，團隊可以放心地釋出新版本的應用程式。谷歌的許多專案每天都會向生產部門釋出一個新的版本--即使是有數百名工程師的大型專案，每天都會提交成千上萬的程式碼修改。如果沒有自動化測試，這是不可能的。

Designing a Test Suite 設計測試套件

Today, Google operates at a massive scale, but we haven't always been so large, and the foundations of our approach were laid long ago. Over the years, as our codebase has grown, we have learned a lot about how to approach the design and execution of a test suite, often by making mistakes and cleaning up afterward.

今天，谷歌的營運規模很大，但我們並不一直以來都這麼大，我們的方法的基礎在很久以前就已奠定。多年來，隨著我們程式碼庫的增長，我們學到了很多關於如何設計和執行測試套件的方法，往往是透過犯錯和事後覆盤。

One of the lessons we learned fairly early on is that engineers favored writing larger, system-scale tests, but that these tests were slower, less reliable, and more difficult to debug than smaller tests. Engineers, fed up with debugging the system-scale tests, asked themselves, "Why can't we just test one server at a time?" or, "Why do we need to test a whole server at once? We could test smaller modules individually." Eventually, the desire to reduce pain led teams to develop smaller and smaller tests, which turned out to be faster, more stable, and generally less painful.

我們很早就學到的一個經驗是，工程師們喜歡寫更大的、系統規模的測試，但這些測試比小型測試更慢，更不可靠，更難除錯。工程師們厭倦了除錯系統規模的測試，問自己："為什麼我們不能一次只測試一臺伺服器？"或者，"為什麼我們需要一次測試整個伺服器？我們可以單獨測試較小的模組"。最終，減輕痛苦的願望促使團隊開發越來越小的測試，結果證明，這些測試更快、更穩定，而且通常也更少痛苦。

This led to a lot of discussion around the company about the exact meaning of "small." Does small mean unit test? What about integration tests, what size are those? We have come to the conclusion that there are two distinct dimensions for every test case: size and scope. Size refers to the resources that are required to run a test case: things like memory, processes, and time. Scope refers to the specific code paths we are verifying. Note that executing a line of code is different from verifying that it worked as expected. Size and scope are interrelated but distinct concepts.

這導致公司上下對 "小 "的確切含義進行了大量的討論。小是指單元測試嗎？那整合測試呢，是什麼規模？我們得出的結論是，每個測試使用案例都有兩個不同的維度：規模和範圍。規模是指執行一個測試使用案例所需的資源：如記憶體、處理序和時間。範圍指的是我們要驗證的具體程式碼路徑。請注意，執行一行程式碼與驗證它是否按預期工作是不同的。規模和範圍是相互關聯但不同的概念。

Test Size 測試規模

At Google, we classify every one of our tests into a size and encourage engineers to always write the smallest possible test for a given piece of functionality. A test's size is determined not by its number of lines of code, but by how it runs, what it is allowed to do, and how many resources it consumes. In fact, in some cases, our definitions of small, medium, and large are actually encoded as constraints the testing infrastructure can enforce on a test. We go into the details in a moment, but in brief, *small tests* run in a single process, *medium tests* run on a single machine, and *large tests* run wherever they want, as demonstrated in Figure 11-2.^4

在谷歌，我們把每一個測試都歸為一個規模，並鼓勵工程師總是為一個給定的功能編寫儘可能小的測試。一個測試的規模大小不是由它的程式碼行數決定的，而是由它的執行方式、它被允許做什麼以及它消耗多少資源決定的。事實上，在某些情況下，我們對小、中、大的定義實際上被編碼為測試基礎設施可以在測試上執行的約束。我們稍後會討論細節，但簡單地說，*小型測試*在一個處理序中執行，*中型測試*在一臺機器上執行，而*大型測試*在任何地方執行，如圖 11-2 所展示。

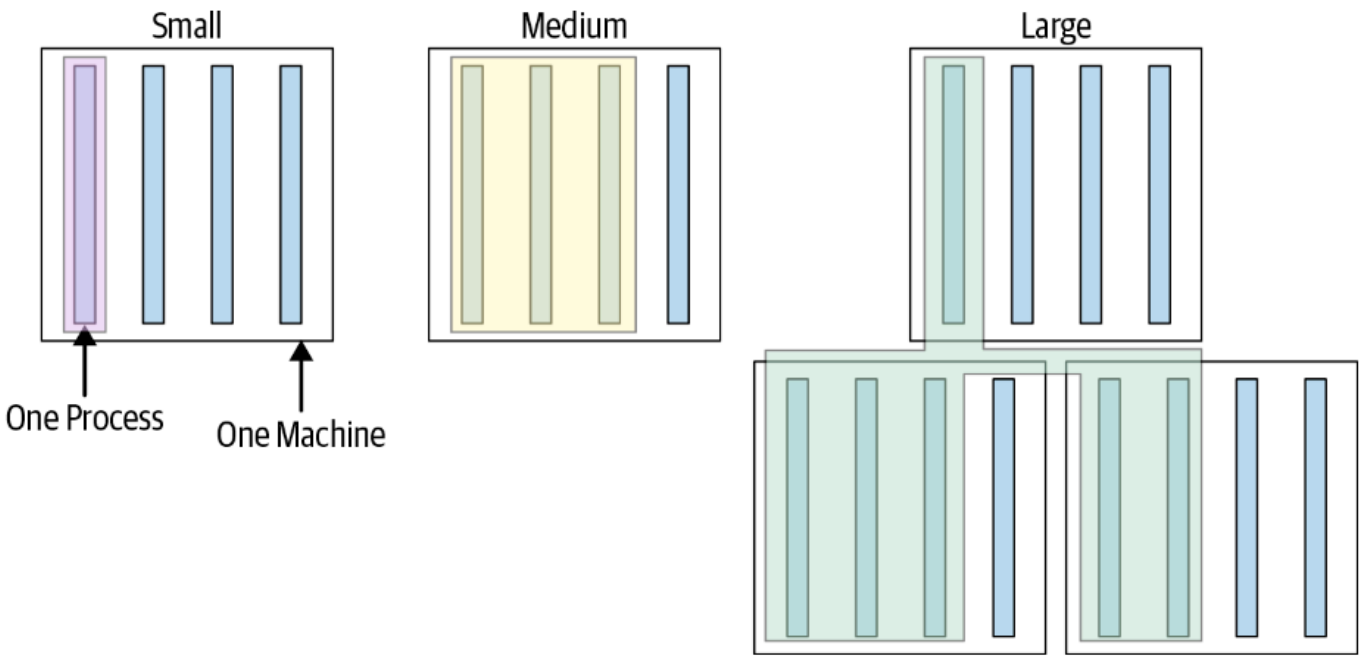

Figure 11-2. Test sizes Figure 11-2. 測試規模

We make this distinction, as opposed to the more traditional "unit" or "integration," because the most important qualities we want from our test suite are speed and determinism, regardless of the scope of the test. Small tests, regardless of the scope, are almost always faster and more deterministic than tests that involve more infrastructure or consume more resources. Placing restrictions on small tests makes speed and determinism much easier to achieve. As test sizes grow, many of the restrictions are relaxed. Medium tests have more flexibility but also more risk of nondeterminism. Larger tests are saved for only the most complex and difficult testing scenarios. Let's take a closer look at the exact constraints imposed on each type of test.

我們做出這樣的區分，而不是更傳統的 "單元 "或 "整合"，因為我們希望從我們的測試套件中得到的最重要的品質是速度和確定性，無論測試的範圍如何。小型測試，無論範圍如何，幾乎總是比涉及更多基礎設施或消耗更多資源的測試更快、更有確定性。對小型測試施加限制，使速度和確定性更容易實現。隨著測試規模的增長，許多限制都被放鬆了。中型測試有更多的靈活性，但也有更多的非確定性的風險。大型測試只儲存在最複雜和困難的測試場景中。讓我們仔細看看每一種測試的確切約束條件。

> 4 從技術上講，我們在谷歌有四種規模的測試：小型、中型、大型和超大型。大型和超大型之間的內部差異實際上是微妙的和歷史性的；因此，在本書中，大多數關於大型的描述實際上適用於我們的超大型概念。

Small tests 小型測試

Small tests are the most constrained of the three test sizes. The primary constraint is that small tests must run in a single process. In many languages, we restrict this even further to say that they must run on a single thread. This means that the code performing the test must run in the same process as the code being tested. You can't run a server and have a separate test process connect to it. It also means that you can't run a third-party program such as a database as part of your test.

小型測試是三種測試規模中最受限制的。主要的限制是，小測試必須在一個處理序中執行。在許多語言中，我們甚至進一步限制，說它們必須在一個單執行緒上執行。這意味著執行測試的程式碼必須與被測試的程式碼在同一處理序中執行。你不能執行一個伺服器，而讓一個單獨的測試處理序連線到它。這也意味著你不能執行第三方程式，如資料庫作為你測試的一部分。

The other important constraints on small tests are that they aren't allowed to sleep, perform I/O operations,[3] or make any other blocking calls. This means that small tests aren't allowed to access the network or disk. Testing code that relies on these sorts of operations requires the use of test doubles (see Chapter 13) to replace the heavyweight dependency with a lightweight, in-process dependency.

對小測試的其他重要限制是，它們不允許休眠，執行 I/O 操作，或進行任何其他阻塞呼叫。這意味著，小測試不允許存取網路或磁碟。測試依賴於這類操作的程式碼需要使用測試替代（見第 13 章）），用輕量級的處理序內依賴取代重量級依賴。

The purpose of these restrictions is to ensure that small tests don't have access to the main sources of test slowness or nondeterminism. A test that runs on a single process and never makes blocking calls can effectively run as fast as the CPU can handle. It's difficult (but certainly not impossible) to accidentally make such a test slow or nondeterministic. The constraints on small tests provide a sandbox that prevents engineers from shooting themselves in the foot.

這些限制的目的是確保小的測試沒有機會使用到測試緩慢或非確定性的主要來源。在單個處理序上執行且從不進行阻塞呼叫的測試可以有效地以 CPU 能夠處理的速度執行。除非一些極端場景， 小型測試絕大部分情況下執行高效、結果穩定準確。對小型測試的限制確保了這一點， 防止工程師自食其果。

These restrictions might seem excessive at first, but consider a modest suite of a couple hundred small test cases running throughout the day. If even a few of them fail nondeterministically (often called flaky tests), tracking down the cause becomes a serious drain on productivity. At Google's scale, such a problem could grind our testing infrastructure to a halt.

起初，這些限制可能看起來太過誇張，但考慮一套完整的一組數百個小測試使用案例，在一天內執行完成。如果哪怕是其中的一小部分測試不確定地失敗（通常稱為鬆散測試），那麼追蹤原因將嚴重影響生產力。按照谷歌的規模，這樣的問題可能會使我們的測試基礎設施陷入停頓。

At Google, we encourage engineers to try to write small tests whenever possible, regardless of the scope of the test, because it keeps the entire test suite running fast and reliably. For more discussion on small versus unit tests, see Chapter 12.

在谷歌，我們鼓勵工程師儘可能地編寫小型測試，而不管測試的範圍如何，因為這樣可以使整個測試套件快速可靠地執行。有關小測試與單元測試的更多討論，請參閱第 12 章。

> 5 這個策略有一點回旋餘地。如果測試使用的是密封的、記憶體中的實現，則允許存取檔案系統。

Medium tests 中型測試

The constraints placed on small tests can be too restrictive for many interesting kinds of tests. The next rung up the ladder of test sizes is the medium test. Medium tests can span multiple processes, use threads, and can make blocking calls, including network calls, to localhost. The only remaining restriction is that medium tests aren't allowed to make network calls to any system other than localhost. In other words, the test must be contained within a single machine.

對於許多有趣的測試型別來說，對小型測試的限制可能過於嚴格。測試規模的下一個階梯是中型測試。中型測試可以跨越多個處理序，使用執行緒，並可以對本地主機進行阻塞呼叫，包括網路呼叫。剩下的唯一限制是，中型測試不允許對 localhost 以外的任何系統進行網路呼叫。換句話說，測試必須包含在一臺機器內。

The ability to run multiple processes opens up a lot of possibilities. For example, you could run a database instance to validate that the code you're testing integrates correctly in a more realistic setting. Or you could test a combination of web UI and server code. Tests of web applications often involve tools like WebDriver that start a real browser and control it remotely via the test process.

執行多個處理序的能力提供了很多可能性。例如，你可以執行一個數據庫例項來驗證你正在測試的程式碼是否正確地整合在更現實的設定中。或者你可以測試網路使用者介面和伺服器程式碼的組合。網路應用程式的測試經常涉及到像 WebDriver 這樣的工具，它可以啟動一個真正的瀏覽器，並透過測試過程遠端控制它。

Unfortunately, with increased flexibility comes increased potential for tests to become slow and nondeterministic. Tests that span processes or are allowed to make blocking calls are dependent on the operating system and third-party processes to be fast and deterministic, which isn't something we can guarantee in general. Medium tests still provide a bit of protection by preventing access to remote machines via the network, which is far and away the biggest source of slowness and nondeterminism in most systems. Still, when writing medium tests, the "safety" is off, and engineers need to be much more careful.

不幸的是，隨著靈活性的增加，測試變得緩慢和不確定性也在增加。如果測試可以跨處理序執行被允許進行阻塞性呼叫，那麼它是否執行高效並保證結果穩定將依賴於作業系統和第三方處理序， 這在一般情況下我們是無法保證的。中型測試仍然透過防止透過網路存取遠端機器來提供一些保護，而網路是大多數系統中速度慢和非確定性的最大來源。儘管如此，在編寫中型測試時，"安全 "是關閉的，工程師需要更加小心。

Large tests 大型測試

Finally, we have large tests. Large tests remove the localhost restriction imposed on medium tests, allowing the test and the system being tested to span across multiple machines. For example, the test might run against a system in a remote cluster.

最後，我們有大型測試。大型測試取消了對中型測試的本地主機限制，允許測試和被測系統跨越多臺機器。例如，測試可能針對遠端叢集中的系統執行。

As before, increased flexibility comes with increased risk. Having to deal with a system that spans multiple machines and the network connecting them increases the chance of slowness and nondeterminism significantly compared to running on a single machine. We mostly reserve large tests for full-system end-to-end tests that are more about validating configuration than pieces of code, and for tests of legacy components for which it is impossible to use test doubles. We'll talk more about use cases for large tests in Chapter 14. Teams at Google will frequently isolate their large tests from their small or medium tests, running them only during the build and release process so as not to impact developer workflow.

和以前一樣，靈活性的提高伴隨著風險的增加。與在單一機器上執行相比，必須處理跨多臺機器的系統以及連線這些機器的網路會顯著增加速度慢和不確定性的機率。我們主要為全系統端到端測試保留大型測試，這些測試更多的是驗證配置，而不是程式碼片段，並且為不可能使用測試替代的遺留元件的測試保留大型測試。我們將在第 14 章中更多地討論大型測試的使用案例。谷歌的團隊經常將大型測試與小型或中型測試隔離開來，只在建構和釋出過程中執行它們，以免影響開發人員的工作流程。

Case Study: Flaky Tests Are Expensive 案例研究：鬆散（不穩定）測試很昂貴

If you have a few thousand tests, each with a very tiny bit of nondeterminism, running all day, occasionally one will probably fail (flake). As the number of tests grows, statistically so will the number of flakes. If each test has even a 0.1% of failing when it should not, and you run 10,000 tests per day, you will be investigating 10 flakes per day. Each investigation takes time away from something more productive that your team could be doing.

如果你有幾千個測試，每個測試都有非常小的不確定性，整天執行，偶爾會有一個可能會失敗（鬆散）。隨著測試數量的增加，從統計學上看，鬆散的數量也會增加。如果每個測試都有 0.1%的失敗率，而你每天執行 10,000 個測試，你每天將調查 10 個鬆散。每次調查都會從團隊可能會做的更有效率的事情中抽出時間。

In some cases, you can limit the impact of flaky tests by automatically rerunning them when they fail. This is effectively trading CPU cycles for engineering time. At low levels of flakiness, this trade-off makes sense. Just keep in mind that rerunning a test is only delaying the need to address the root cause of flakiness.

在某些情況下，你可以透過在測試失敗時自動重新執行它們來限制鬆散測試的影響。這實際上是用 CPU 週期換取工程時間。在低水平的鬆散情況下，這種權衡是有意義的。記住，重新執行測試只會推遲解決片狀問題根本原因的需要。

If test flakiness continues to grow, you will experience something much worse than lost productivity: a loss of confidence in the tests. It doesn't take needing to investigate many flakes before a team loses trust in the test suite. After that happens, engineers will stop reacting to test failures, eliminating any value the test suite provided. Our experience suggests that as you approach 1% flakiness, the tests begin to lose value. At Google, our flaky rate hovers around 0.15%, which implies thousands of flakes every day. We fight hard to keep flakes in check, including actively investing engineering hours to fix them.

如果測試失誤繼續增長，你將經歷比生產效率損失更糟糕的事情：對測試的信心喪失。在團隊失去對測試套件的信任之前，不需要投入太多的精力。發生這種情況後，工程師將停止對測試失敗的反應，消除測試套件提供的任何價值。我們的經驗表明，當你接近 1%的鬆散率時，測試開始失去價值。在谷歌，我們的鬆散率徘徊在 0.15%左右，這意味著每天有成千上萬的不穩定。我們努力地控制故障率，包括積極地投入工程時間來修復它們。

In most cases, flakes appear because of nondeterministic behavior in the tests themselves. Software provides many sources of nondeterminism: clock time, thread scheduling, network latency, and more. Learning how to isolate and stabilize the effects of randomness is not easy. Sometimes, effects are tied to low-level concerns like hardware interrupts or browser rendering engines. A good automated test infrastructure should help engineers identify and mitigate any nondeterministic behavior.

在大多數情況下，鬆散出現是因為測試本身的不確定性行為。軟體提供了許多不確定性的來源：時鐘時間、執行緒排程、網路延遲，等等。學習如何隔離和穩定隨機性的影響並不容易。有時，影響與硬體中斷或瀏覽器渲染引擎等低層的問題聯絡在一起。一個好的自動化測試基礎設施應該幫助工程師識別和緩解任何不確定性行為。

Properties common to all test sizes 所有測試規模的共同屬性

All tests should strive to be hermetic: a test should contain all of the information necessary to set up, execute, and tear down its environment. Tests should assume as little as possible about the outside environment, such as the order in which the tests are run. For example, they should not rely on a shared database. This constraint becomes more challenging with larger tests, but effort should still be made to ensure isolation.

所有測試應力求封閉性：測試應包含設定、執行和拆除其環境所需的所有資訊。測試應該儘可能少地假設外部環境，例如測試的執行順序。例如，他們不應該依賴共享資料庫。這種約束在大型測試中變得更具挑戰性，但仍應努力確保隔離

A test should contain *only* the information required to exercise the behavior in question. Keeping tests clear and simple aids reviewers in verifying that the code does what it says it does. Clear code also aids in diagnosing failure when they fail. We like to say that “a test should be obvious upon inspection.” Because there are no tests for the tests themselves, they require manual review as an important check on correctness. As a corollary to this, we also strongly discourage the use of control flow statements like conditionals and loops in a test. More complex test flows risk containing bugs themselves and make it more difficult to determine the cause of a test failure.

測試應該*僅*包含測試行為所需的資訊。保持測試的清晰和簡單，有助於審查人員驗證程式碼是否做了它所說的事情。清晰的程式碼也有助於在測試失敗時診斷失敗。我們喜歡說，"測試應該在檢查時是明顯的"。因為測試本身沒有測試，所以需要手動檢查，作為對正確性的重要檢查。作為一個推論，我們也強烈反對在測試中使用控制流陳述式，如條件和迴圈。更加複雜的測試流程有可能本身包含 bug，並使確定測試失敗的原因更加困難。

Remember that tests are often revisited only when something breaks. When you are called to fix a broken test that you have never seen before, you will be thankful someone took the time to make it easy to understand. Code is read far more than it is written, so make sure you write the test you’d like to read!

請記住，測試只有在出現故障時，才會重新檢查測試。當你被要求修復一個你從未見過的失敗測試時，你會感激有人花時間讓它變得容易理解。程式碼的讀取量遠遠大於寫入量，所以要確保你寫的測試是你看得懂的！

Test sizes in practice. Having precise definitions of test sizes has allowed us to create tools to enforce them. Enforcement enables us to scale our test suites and still make certain guarantees about speed, resource utilization, and stability. The extent to which these definitions are enforced at Google varies by language. For example, we run all Java tests using a custom security manager that will cause all tests tagged as small to fail if they attempt to do something prohibited, such as establish a network connection.

測試規模的實踐。有了測試規模的精確定義，我們就可以建立工具來執行它們。強制執行使我們能夠擴充我們的測試套件，並仍然對速度、資源利用和穩定性做出一定的保證。在谷歌，這些定義的執行程度因語言而異。例如，我們使用一個自訂的安全管理器來執行所有的 Java 測試，如果它們試圖做一些被禁止的事情，如建立網路連線，就會導致所有被標記為小型測試失敗。

Test Scope 測試範圍

Though we at Google put a lot of emphasis on test size, another important property to consider is test scope. Test scope refers to how much code is being validated by a given test. Narrow-scoped tests (commonly called “unit tests”) are designed to validate the logic in a small, focused part of the codebase, like an individual class or method. Medium-scoped tests (commonly called *integration tests*) are designed to verify interactions between a small number of components; for example, between a server and its database. Large-scoped tests (commonly referred to by names like *functional tests*, *end-to-end* tests, or *system tests*) are designed to validate the interaction of several distinct parts of the system, or emergent behaviors that aren’t expressed in a single class or method.

儘管我們在谷歌非常強調測試規模，但另一個需要考慮的重要屬性是測試範圍。測試範圍是指給定的測試要驗證多少程式碼。狹小範圍的測試（通常稱為 "單元測試"）被設計用來驗證程式碼庫中一小部分的邏輯，比如單獨的類或方法。中等範圍的測試（通常稱為*整合測試*）被設計用來驗證少量元件之間的相互作用；例如，在伺服器和它的資料庫之間。大範圍測試（通常被稱為*功能測試*，*端到端*測試，或*系統測試*）被設計用來驗證系統的幾個不同部分的相互作用，或不在單個類或方法中表達的出現的行為。

It’s important to note that when we talk about unit tests as being narrowly scoped, we’re referring to the code that is being *validated*, not the code that is being *executed*. It’s quite common for a class to have many dependencies or other classes it refers to, and these dependencies will naturally be invoked while testing the target class. Though some other testing strategies make heavy use of test doubles (fakes or mocks) to avoid executing code outside of the system under test, at Google, we prefer to keep the real dependencies in place when it is feasible to do so. Chapter 13 discusses this issue in more detail.

值得注意的是，當我們談論單元測試是狹義的範圍時，我們指的是正在*驗證*的程式碼，而不是正在*執行*的程式碼。一個類別有許多依賴關係或它參考的其他類是很常見的，這些依賴關係在測試目標類時自然會被呼叫。儘管有其他測試策略大量使用測試替代（假的或模擬的）來避免執行被測系統之外的程式碼，但在 Google，我們更願意在可行的情況下保持真正的依賴關係。第 13 章更詳細地討論了這個問題。

Narrow-scoped tests tend to be small, and broad-scoped tests tend to be medium or large, but this isn’t always the case. For example, it’s possible to write a broad-scoped test of a server endpoint that covers all of its normal parsing, request validation, and business logic, which is nevertheless small because it uses doubles to stand in for all out-of-process dependencies like a database or filesystem. Similarly, it’s possible to write a narrow-scoped test of a single method that must be medium sized. For example, modern web frameworks often bundle HTML and JavaScript together, and testing a UI component like a date picker often requires running an entire browser, even to validate a single code path.

狹小範圍的測試往往較小，而廣泛範圍的測試往往是中等或較大的，但這並不總是如此。例如，有可能對一個伺服器端點寫一個大範圍的測試，包括所有正常的解析、請求驗證和業務邏輯，但這是小範圍的，因為它用替代來代替所有處理序外的依賴，如資料庫或檔案系統。同樣，也可以對一個單一的方法寫一個範圍較窄的測試，但必須是中等大小。例如，現代網路框架經常將 HTML 和 JavaScript 捆綁在一起，測試像日期選擇器這樣的 UI 元件經常需要執行整個瀏覽器，即使是驗證單個的程式碼路徑。

Just as we encourage tests of smaller size, at Google, we also encourage engineers to write tests of narrower scope. As a very rough guideline, we tend to aim to have a mix of around 80% of our tests being narrow-scoped unit tests that validate the majority of our business logic; 15% medium-scoped integration tests that validate the interactions between two or more components; and 5% end-to-end tests that validate the entire system. Figure 11-3 depicts how we can visualize this as a pyramid.

正如我們鼓勵在谷歌進行更小規模的測試一樣，我們也鼓勵工程師編寫範圍更狹小的測試。作為一個非常粗略的指導方針，我們傾向於將大約 80%的測試混合在一起，這些測試是驗證大多數業務邏輯的狹小範圍單元測試；15%的中型整合測試，用於驗證兩個或多個元件之間的相互作用；以及驗證整個系統的 5%端到端測試。圖 11-3 描述了我們如何將其視為金字塔。

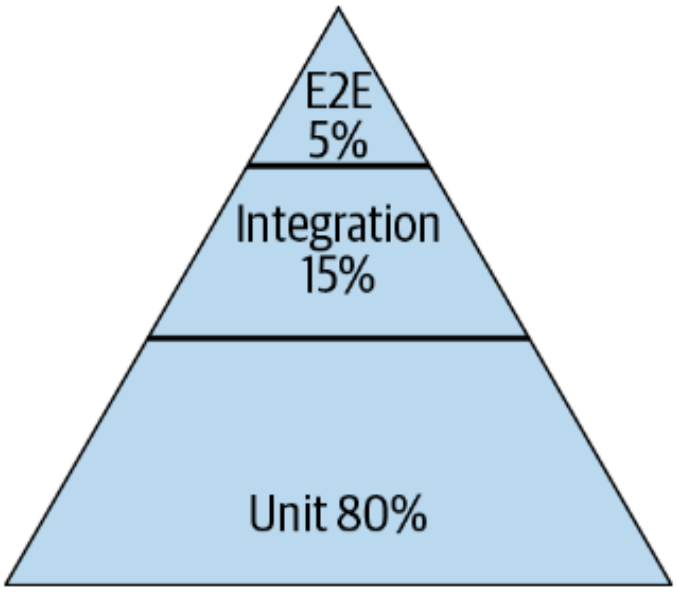

Figure 11-3. Google's version of Mike Cohn's test pyramid;[4] *percentages are by test case count, and every team's mix will be a little different 圖 11-3. 谷歌對 Mike Cohn 的測試金字塔的版本百分比是按測試案例來計算的，每個團隊的組合都會有一些不同*

Unit tests form an excellent base because they are fast, stable, and dramatically narrow the scope and reduce the cognitive load required to identify all the possible behaviors a class or function has. Additionally, they make failure diagnosis quick and painless. Two antipatterns to be aware of are the "ice cream cone" and the "hourglass," as illustrated in Figure 11-4.

單元測試是一個很好的基礎，因為它們快速、穩定，並且極大地縮小了範圍，減少了識別一個類別或函式的所有可能行為所需的認知負荷。此外，它們使故障診斷變得快速而無感。需要注意的兩個反模式是 "冰淇淋筒 "和 "沙漏"，如圖 11-4 所示。

With the ice cream cone, engineers write many end-to-end tests but few integration or unit tests. Such suites tend to be slow, unreliable, and difficult to work with. This pattern often appears in projects that start as prototypes and are quickly rushed to production, never stopping to address testing debt.

在冰淇淋筒中，工程師們寫了許多端到端的測試，但很少有整合或單元測試。這樣的套件往往是速度慢、不可靠，而且難以使用。這種模式經常出現在以原型開始的專案中，並很快投入生產，從來沒有停止過解決測試債務。

The hourglass involves many end-to-end tests and many unit tests but few integration tests. It isn't quite as bad as the ice cream cone, but it still results in many end-to- end test failures that could have been caught quicker and more easily with a suite of medium-scope tests. The hourglass pattern occurs when tight coupling makes it difficult to instantiate individual dependencies in isolation.

沙漏涉及許多端到端的測試和許多單元測試，但很少有整合測試。它不像冰淇淋筒那樣糟糕，但它仍然導致許多端到端的測試失敗，而這些失敗本可以透過一套中等範圍的測試更快、更容易地發現。當緊密耦合使單獨例項化單個依賴項變得困難時，就會出現沙漏模式。

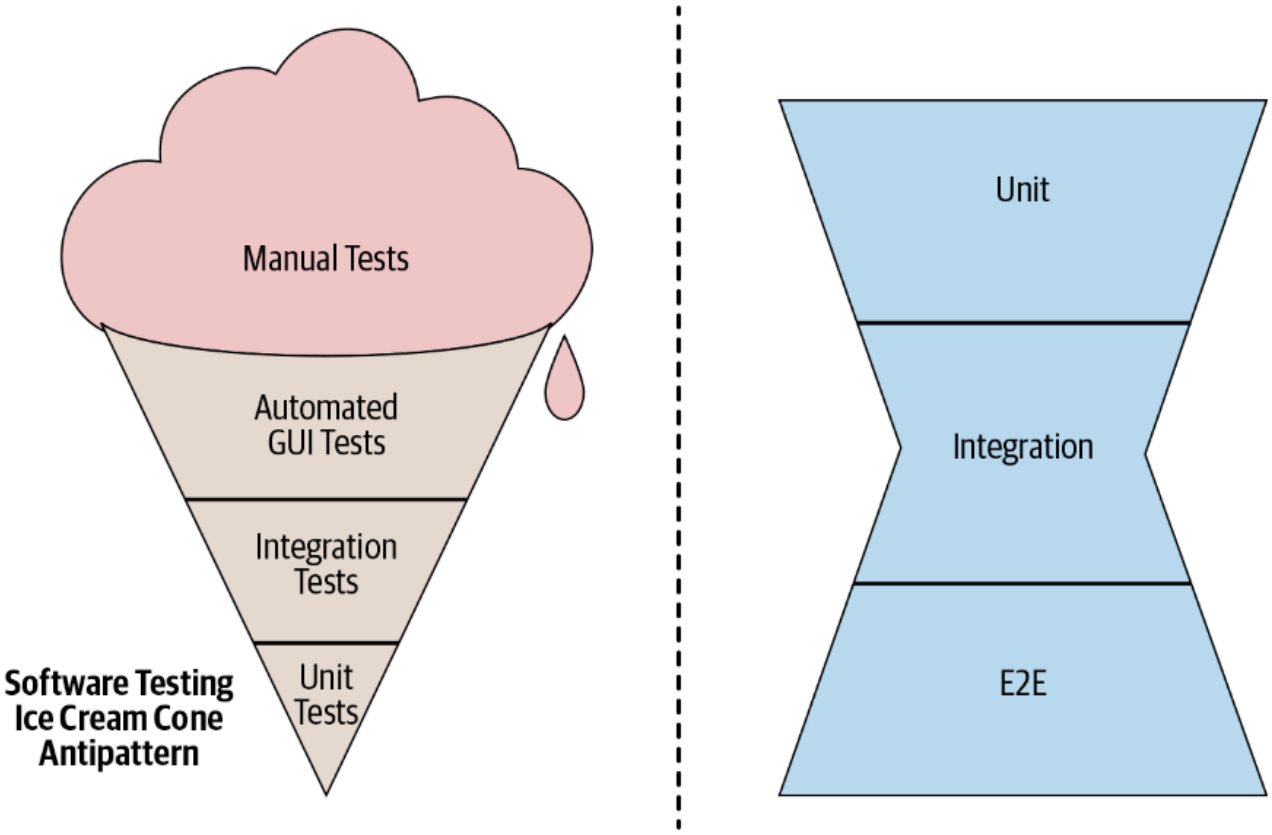

Figure 11-4. Test suite antipatterns 圖 11-4. 測試套件的反模式

Our recommended mix of tests is determined by our two primary goals: engineering productivity and product confidence. Favoring unit tests gives us high confidence quickly, and early in the development process. Larger tests act as sanity checks as the product develops; they should not be viewed as a primary method for catching bugs.

我們推薦的測試組合是由我們的兩個主要目標決定的：工程生產效率和產品信心。在開發過程的早期，傾向於單元測試可以讓我們迅速獲得高的信心。在產品的開發過程中，大型測試可以作為健康檢查；它們不應被視為捕獲 bug 的主要方法

When considering your own mix, you might want a different balance. If you emphasize integration testing, you might discover that your test suites take longer to run but catch more issues between components. When you emphasize unit tests, your test suites can complete very quickly, and you will catch many common logic bugs. But, unit tests cannot verify the interactions between components, like a contract between two systems developed by different teams. A good test suite contains a blend of different test sizes and scopes that are appropriate to the local architectural and organizational realities.

當考慮你自己的組合時，你可能想要一個不同的平衡。如果你強調整合測試，你可能會發現你的測試套件需要更長的時間來執行，但在元件之間捕獲更多的問題。當你強調單元測試時，你的測試套件可以很快完成，而且你會捕捉到許多常見的邏輯錯誤。但是，單元測試無法驗證元件之間的互動，就像由不同團隊開發的兩個系統之間的契約一樣。一個好的測試套件包含不同的測試規模和範圍的混合，適合本地架構和組織的實際情況。

> 6 Mike Cohn，《敏捷的成功：使用 Scrum 的軟體開發》（紐約：Addison-Wesley Professio-nal，2009）。

The Beyoncé Rule 碧昂斯規則

We are often asked, when coaching new hires, which behaviors or properties actually need to be tested? The straightforward answer is: test everything that you don't want to break. In other words, if you want to be confident that a system exhibits a particular behavior, the only way to be sure it will is to write an automated test for it. This includes all of the usual suspects like testing performance, behavioral correctness, accessibility, and security. It also includes less obvious properties like testing how a system handles failure.

在指導新員工時，我們經常被問到，究竟哪些行為或屬性需要被測試？直截了當的回答是：測試所有你不想破壞的東西。換句話說，如果你想確信一個系統表現出一個特定的行為，唯一能確保它會表現出這種行為的方法就是為它編寫一個自動測試。這包括所有常見的可疑因素，如測試效能、行為正確性、可及性和安全性。它還包括不太明顯的屬性，如測試系統如何處理故障。

We have a name for this general philosophy: we call it the Beyoncé Rule. Succinctly, it can be stated as follows: "If you liked it, then you shoulda put a test on it." The Beyoncé Rule is often invoked by infrastructure teams that are responsible for making changes across the entire codebase. If unrelated infrastructure changes pass all of your tests but still break your team's product, you are on the hook for fixing it and adding the additional tests.

我們對這一總體理念有一個名稱：我們稱之為碧昂斯規則。簡而言之，它可以被陳述如下。"如果你喜歡它，那麼你就應該對它進行測試"。碧昂斯規則通常由負責在整個程式碼庫中進行更改的基礎架構團隊參考。如果不相關的基礎設施變化通過了你所有的測試，但仍然破壞了你的團隊的產品，你就得負責修復它並增加額外的測試。

Testing for Failure 失敗測試

One of the most important situations a system must account for is failure. Failure is inevitable, but waiting for an actual catastrophe to find out how well a system responds to a catastrophe is a recipe for pain. Instead of waiting for a failure, write automated tests that simulate common kinds of failures. This includes simulating exceptions or errors in unit tests and injecting Remote Procedure Call (RPC) errors or latency in integration and end-to-end tests. It can also include much larger disruptions that affect the real production network using techniques like Chaos Engineering. A predictable and controlled response to adverse conditions is a hallmark of a reliable system.

系統必須考慮的最重要的情況之一是失敗。失敗是不可避免的，但是要被動等待實際的故障發生後才做對應的補救，是令人痛苦的。與其等待失敗，不如寫自動測試來模擬常見的失敗型別。這包括在單元測試中模擬例外或錯誤，在整合和端到端測試中注入遠端過程呼叫（RPC）錯誤或延遲。它還可以包括使用混沌工程等技術影響真實生產網路的更大的破壞。對不利條件的可預測和可控制的反應是一個可靠系統的標誌。

A Note on Code Coverage 關於程式碼覆蓋率的注意事項

Code coverage is a measure of which lines of feature code are exercised by which tests. If you have 100 lines of code and your tests execute 90 of them, you have 90% code coverage.[5] Code coverage is often held up as the gold standard metric for understanding test quality, and that is somewhat unfortunate. It is possible to exercise a lot of lines of code with a few tests, never checking that each line is doing anything useful. That's because code coverage only measures that a line was invoked, not what happened as a result. (We recommend only measuring coverage from small tests to avoid coverage inflation that occurs when executing larger tests.)

程式碼覆蓋率是衡量哪些特徵程式碼行被哪些測試所執行的標準。如果你有 100 行程式碼，你的測試執行了其中的 90 行，你就有 90%的程式碼覆蓋率。 程式碼覆蓋率經常被認為是理解測試品質的黃金標準，這是很不幸的。有可能用幾個測試來驗證大量的程式碼行，但從未檢查過每一行是否在做任何有用的事情。這是因為程式碼覆蓋率只衡量一行被呼叫的情況，而不是結果。(我們建議只測量小型測試的覆蓋率，以避免執行大型測試時出現覆蓋率膨脹）。

An even more insidious problem with code coverage is that, like other metrics, it quickly becomes a goal unto itself. It is common for teams to establish a bar for expected code coverage—for instance, 80%. At first, that sounds eminently reasonable; surely you want to have at least that much coverage. In practice, what happens is that instead of treating 80% like a floor, engineers treat it like a ceiling. Soon, changes begin landing with no more than 80% coverage. After all, why do more work than the metric requires?

程式碼覆蓋率的一個更隱蔽的問題是，像其他指標一樣，它很快就變成了一個單獨的目標。對於團隊來說，建立一個預期程式碼覆蓋率的標準是很常見的，比如說 80%。起初，這聽起來非常合理；你肯定希望至少有這麼多的覆蓋率。在實踐中，發生的情況是，工程師們不是把 80%當作一個底線，而是把它當作一個上限。很快，變化就開始了，覆蓋率不超過 80%。畢竟，為什麼要做比指標要求更多的工作？

A better way to approach the quality of your test suite is to think about the behaviors that are tested. Do you have confidence that everything your customers expect to work will work? Do you feel confident you can catch breaking changes in your dependencies? Are your tests stable and reliable? Questions like these are a more holistic way to think about a test suite. Every product and team is going to be different; some will have difficult-to-test interactions with hardware, some involve massive datasets. Trying to answer the question "do we have enough tests?" with a single number ignores a lot of context and is unlikely to be useful. Code coverage can provide some insight into untested code, but it is not a substitute for thinking critically about how well your system is tested.

評估測試套件品質的更好方法是考慮測試的行為。你有信心你的客戶所期望的一切都能正常工作嗎？你是否有信心能抓住你的依賴關係中的突發變化？你的測試是否穩定和可靠？像這樣的問題是思考測試套件的一種更全面的方式。每個產品和團隊都是不同的；有些會有難以測試的與硬體的互動，有些涉及到大量的資料集。試圖用一個獨立的數字來回答 "我們有足夠的測試嗎？"忽略了很多背景，不太可能是有用的。程式碼覆蓋率可以提供一些對未測試程式碼的洞察力，但它不能替代對系統測試情況的批判性思考。

> 7 請記住，有不同種類的覆蓋率（行、路徑、分支等），每一種都說明了不同的程式碼被測試的情況。在這個簡單的例子中，我們使用的是行覆蓋。

Testing at Google Scale 以谷歌的規模進行測試

Much of the guidance to this point can be applied to codebases of almost any size. However, we should spend some time on what we have learned testing at our very large scale. To understand how testing works at Google, you need an understanding of our development environment, the most important fact about which is that most of Google's code is kept in a single, monolithic repository (monorepo). Almost every line of code for every product and service we operate is all stored in one place. We have more than two billion lines of code in the repository today.

到此為止的大部分指導可以應用於幾乎任何規模的程式碼庫。然而，我們應該花一些時間來討論我們在非常大的規模下測試所學到的東西。要了解測試在谷歌是如何工作的，你需要了解我們的開發環境，其中最重要的事實是，谷歌的大部分程式碼都儲存在一個單個單版本版本庫（monorepo）。我們營運的每種產品和服務的幾乎每一行程式碼都儲存在一個地方。今天，儲存庫中有 20 多億行程式碼。

Google's codebase experiences close to 25 million lines of change every week. Roughly half of them are made by the tens of thousands of engineers working in our monorepo, and the other half by our automated systems, in the form of configuration updates or large-scale changes (Chapter 22). Many of those changes are initiated from outside the immediate project. We don't place many limitations on the ability of engineers to reuse code.

谷歌的程式碼庫每週都要經歷接近 2500 萬行的變更。其中大約一半是由成千上萬的工程師在我們的 monorepo 中工作，另一半是由我們的自動化系統以配置更新或大規模變更的形式進行的（第 22 章））。其中許多變更是由直接專案以外的人發起的。我們對工程師重用程式碼的能力沒有施加很多限制。

The openness of our codebase encourages a level of co-ownership that lets everyone take responsibility for the codebase. One benefit of such openness is the ability to directly fix bugs in a product or service you use (subject to approval, of course) instead of complaining about it. This also implies that many people will make changes in a part of the codebase owned by someone else.

我們程式碼庫的開放性鼓勵了一定程度的共同所有權，讓每個人都對程式碼庫負責。這種開放性的一個好處是能夠直接修復你使用的產品或服務中的錯誤（當然，需要獲得批准），而不是抱怨它。這也意味著許多人將對其他人擁有的程式碼庫的一部分進行更改。

Another thing that makes Google a little different is that almost no teams use repository branching. All changes are committed to the repository head and are immediately visible for everyone to see. Furthermore, all software builds are performed using the last committed change that our testing infrastructure has validated. When a product or service is built, almost every dependency required to run it is also built from source, also from the head of the repository. Google manages testing at this scale by use of a CI system. One of the key components of our CI system is our Test Automated Platform (TAP).

另一件讓谷歌有點不同的事情是，幾乎沒有團隊使用版本庫分支。所有的更改都提交到了版本庫的 head，並且每個人都可以立即看到。此外，所有的軟體建構都是使用我們的測試基礎設施驗證過的最後一次提交的變更。當一個產品或服務被建構時，幾乎所有執行該產品或服務所需的依賴也都是從原始碼建構的，也是從資源庫的 head。谷歌透過使用 CI 系統來管理這種規模的測試。我們 CI 系統的關鍵組成部分之一是我們的測試自動化平台（TAP）。

Whether you are considering our size, our monorepo, or the number of products we offer, Google's engineering environment is complex. Every week it experiences millions of changing lines, billions of test cases being run, tens of thousands of binaries being built, and hundreds of products being updated—talk about complicated!

無論你考慮的是我們的規模、我們的 monorepo，還是我們提供的產品數量，谷歌的工程環境都很複雜。每週，它都要經歷數百萬條變化的線路，數十億個測試案例的執行，數萬個二進位制檔案的建構，以及數百個產品的更新--說起來很複雜!

The Pitfalls of a Large Test Suite 大型測試套件的缺陷

As a codebase grows, you will inevitably need to make changes to existing code. When poorly written, automated tests can make it more difficult to make those changes. Brittle tests—those that over-specify expected outcomes or rely on extensive and complicated boilerplate—can actually resist change. These poorly written tests can fail even when unrelated changes are made.

隨著程式碼庫的增長，不可避免地需要對現有程式碼進行更改。如果編寫得不好，自動化測試會使進行這些更改變得更加困難。脆性測試——那些過度指定預期結果或依賴廣泛而複雜的範本的測試，實際上可以抵制變化。這些寫得不好的測試可能會失敗，即使是在進行不相關的更改時。

If you have ever made a five-line change to a feature only to find dozens of unrelated, broken tests, you have felt the friction of brittle tests. Over time, this friction can make a team reticent to perform necessary refactoring to keep a codebase healthy. The subsequent chapters will cover strategies that you can use to improve the robustness and quality of your tests.

如果你曾經對一個功能做了五行的修改，卻發現有幾十個不相關的、中斷的測試，你就會感覺到脆性測試的阻力。隨著時間的推移，這種阻力會使一個團隊不願意進行必要的重構來保持程式碼庫的健康。後續章節將介紹可用於提高測試健壯性和品質的策略。

Some of the worst offenders of brittle tests come from the misuse of mock objects. Google's codebase has suffered so badly from an abuse of mocking frameworks that it has led some engineers to declare "no more mocks!" Although that is a strong statement, understanding the limitations of mock objects can help you avoid misusing them.

脆性測試的一些最嚴重的犯錯來自於對模擬物件的濫用。谷歌的程式碼庫因濫用模擬框架而受到嚴重影響，導致一些工程師宣佈"不再使用模擬物件"。雖然這是一個強烈的宣告，但瞭解模擬物件的侷限性可以幫助你避免濫用它們。

In addition to the friction caused by brittle tests, a larger suite of tests will be slower to run. The slower a test suite, the less frequently it will be run, and the less benefit it provides. We use a number of techniques to speed up our test suite, including parallelizing execution and using faster hardware. However, these kinds of tricks are eventually swamped by a large number of individually slow test cases.

除了脆性測試引起的阻力外，大型測試套件的執行速度也會更慢。測試套件越慢，它的執行頻率就越低，提供的好處也就越少。我們使用一些技術來加快我們的測試套件，包括並行執行和使用更快的硬體。然而，這些技巧甚至被大量單獨的緩慢測試使用案例所淹沒。

Tests can become slow for many reasons, like booting significant portions of a system, firing up an emulator before execution, processing large datasets, or waiting for disparate systems to synchronize. Tests often start fast enough but slow down as the system grows. For example, maybe you have an integration test exercising a single dependency that takes five seconds to respond, but over the years you grow to depend on a dozen services, and now the same tests take five minutes.

測試會因為很多原因而變得緩慢，比如啟動系統的重要部分，在執行前啟動模擬器，處理大型資料集，或者等待不同的系統同步。測試開始時往往足夠快，但隨著系統的發展，速度會變慢。例如，也許你有一個整合測試，它請求某個依賴，需要 5 秒鐘的響應，但隨著時間的推移，你逐步依賴十幾個服務，現在同樣的測試需要 5 分鐘。

Tests can also become slow due to unnecessary speed limits introduced by functions like sleep() and setTimeout(). Calls to these functions are often used as naive heuristics before checking the result of nondeterministic behavior. Sleeping for half a second here or there doesn't seem too dangerous at first; however, if a "wait-and-check" is embedded in a widely used utility, pretty soon you have added minutes of idle time to every run of your test suite. A better solution is to actively poll for a state transition with a frequency closer to microseconds. You can combine this with a timeout value in case a test fails to reach a stable state.

由於`sleep()`和`setTimeout()`等函式引入的不必要的速度限制，測試也會變得緩慢。在檢查不確定性行為的結果之前，對這些函式的呼叫通常被用作簡單的啟發式。在這裡或那裡休眠半秒鐘，起初看起來並不太危險；然而，如果 "等待和檢查 "被嵌入到一個廣泛使用的工具中，很快你就會在你的測試套件的每次執行中增加幾分鐘的等待時間。更好的解決方案是以接近微秒的頻率主動輪詢狀態轉換。你可以把它和一個超時值結合起來，以防測試無法達到穩定狀態。

Failing to keep a test suite deterministic and fast ensures it will become roadblock to productivity. At Google, engineers who encounter these tests have found ways to work around slowdowns, with some going as far as to skip the tests entirely when submitting changes. Obviously, this is a risky practice and should be discouraged, but if a test suite is causing more harm than good, eventually engineers will find a way to get their job done, tests or no tests.

如果不能保持測試套件的確定性和速度，那麼它將成為生產力的障礙。在谷歌，遇到這些測試的工程師們已經找到了解決速度慢的方法，有些人甚至在提交更改時完全跳過測試。顯然，這是一種危險的做法，應該被阻止，但如果測試套件利大於弊，最終工程師會找到一種方法來完成他們的工作，不管有沒有測試。

The secret to living with a large test suite is to treat it with respect. Incentivize engineers to care about their tests; reward them as much for having rock-solid tests as you would for having a great feature launch. Set appropriate performance goals and refactor slow or marginal tests. Basically, treat your tests like production code. When simple changes begin taking nontrivial time, spend effort making your tests less brittle.

使用大型測試套件的秘訣是尊重它。激勵工程師關心他們的測試；獎勵他們擁有堅如磐石的測試，就像獎勵他們推出一個偉大的功能一樣。設定適當的效能目標，重構漸進或邊緣測試。基本上，把你的測試當作生產程式碼。當簡單的修改開始花費大量的時間時，要花精力讓你的測試不那麼脆弱。

In addition to developing the proper culture, invest in your testing infrastructure by developing linters, documentation, or other assistance that makes it more difficult to write bad tests. Reduce the number of frameworks and tools you need to support to increase the efficiency of the time you invest to improve things.[6] If you don't invest in making it easy to manage your tests, eventually engineers will decide it isn't worth having them at all.

除了發展適當的文化，透過開發工具、文件或其他援助，投資於你的測試基礎設施，這些幫助會使其更難寫出糟糕的測試。減少你需要支援的框架和工具的數量，以提高改進工作的時間效率。如果不投資於簡化測試管理，工程師最終會認為根本不值得擁有它們。

> 8 谷歌支援的每種語言都有一個標準的測試框架和一個標準的模擬/打樁庫。一套基礎設施在整個程式碼庫中執行所有語言的大多數測試。

History of Testing at Google 谷歌的測試歷史

Now that we've discussed how Google approaches testing, it might be enlightening to learn how we got here. As mentioned previously, Google's engineers didn't always embrace the value of automated testing. In fact, until 2005, testing was closer to a curiosity than a disciplined practice. Most of the testing was done manually, if it was done at all. However, from 2005 to 2006, a testing revolution occurred and changed the way we approach software engineering. Its effects continue to reverberate within the company to this day.

既然我們已經討論了谷歌是如何進行測試的，那麼瞭解一下我們是如何做到這一點可能會有所啟發。如前所述，谷歌的工程師並不總是接受自動化測試的價值。事實上，直到 2005 年，測試更像是一種好奇心，而不是一種嚴格的實踐。大部分的測試都是手動完成的，如果有的話。然而，從 2005 年到 2006 年，發生了一場測試革命，改變了我們對待軟體工程的方式。其影響至今仍在公司內部迴響。

The experience of the GWS project, which we discussed at the opening of this chapter, acted as a catalyst. It made it clear how powerful automated testing could be. Following the improvements to GWS in 2005, the practices began spreading across the entire company. The tooling was primitive. However, the volunteers, who came to be known as the Testing Grouplet, didn't let that slow them down.

我們在本章開頭討論的 GWS 專案的經驗，起到了催化劑的作用。它清晰地展示了自動化測試的強大功能。在 2005 年對 GWS 的改進之後，這種做法開始在整個公司推廣。工具是原始的。然而，被稱為 "測試小組 "的志願者們並沒有因此而懈怠。

Three key initiatives helped usher automated testing into the company's consciousness: Orientation Classes, the Test Certified program, and Testing on the Toilet. Each one had influence in a completely different way, and together they reshaped Google's engineering culture.

三個關鍵的舉措有助於將自動化測試引入公司的意識。定向班、測試認證計劃和廁所測試。每一項都以完全不同的方式產生影響，它們共同重塑了谷歌的工程文化。

Orientation Classes 定向班

Even though much of the early engineering staff at Google eschewed testing, the pioneers of automated testing at Google knew that at the rate the company was growing, new engineers would quickly outnumber existing team members. If they could reach all the new hires in the company, it could be an extremely effective avenue for introducing cultural change. Fortunately, there was, and still is, a single choke point that all new engineering hires pass through: orientation.

儘管谷歌早期的工程人員大多回避測試，但 Google 自動化測試的工程師們知道，按照公司的發展速度，新加入的工程師會很快超過現有的團隊成員。如果他們能接觸到公司所有的新員工，這可能是一個引入文化變革的極其有效的途徑。幸運的是，所有新的工程人員都要經歷一個節點：入職培訓。

Most of Google's early orientation program concerned things like medical benefits and how Google Search worked, but starting in 2005 it also began including an hour- long discussion of the value of automated testing.[7] The class covered the various benefits of testing, such as increased productivity, better documentation, and support for refactoring. It also covered how to write a good test. For many Nooglers (new Googlers) at the time, such a class was their first exposure to this material. Most important, all of these ideas were presented as though they were standard practice at the company. The new hires had no idea that they were being used as trojan horses to sneak this idea into their unsuspecting teams.

谷歌早期的新人培訓大多涉及諸如醫療福利和谷歌搜尋如何工作，但從 2005 年開始，它也開始包括一個長達一小時的關於自動化測試價值的討論。該課程涵蓋了測試的各種好處，如提高生產力，更好的文件，以及對重構的支援。它還包括如何寫一個好的測試。對於當時的許多 Nooglers（新的 Googlers）來說，這樣的課程是他們第一次接觸到這種材料。最重要的是，所有這些想法都是作為公司的標準做法來介紹的。新員工們不知道他們被當作特洛伊木馬，把這種想法偷偷帶入他們毫無戒心的團隊。

As Nooglers joined their teams following orientation, they began writing tests and questioning those on the team who didn't. Within only a year or two, the population of engineers who had been taught testing outnumbered the pretesting culture engineers. As a result, many new projects started off on the right foot.

當 Noogler 加入他們的團隊後，他們開始寫測試，並質疑團隊中那些沒有寫的人。在短短的一兩年內，接受過測試教學的工程師人數超過了預先測試的文化工程師。因此，許多新專案一開始就很順利。

Testing has now become more widely practiced in the industry, so most new hires arrive with the expectations of automated testing firmly in place. Nonetheless, orientation classes continue to set expectations about testing and connect what Nooglers know about testing outside of Google to the challenges of doing so in our very large and very complex codebase.

現在，測試已經在行業中得到了更廣泛的應用，所以大多數新員工來到這裡時，對自動化測試的期望已經很高了。儘管如此，迎新課程仍然要設定對測試的期望，並將 Nooglers 在谷歌以外的測試知識與在我們非常大和非常複雜的程式碼庫中進行測試的挑戰聯絡起來。

> 9 這門課非常成功，以至於今天仍在教授更新的版本。事實上，它是公司歷史上執行時間最長的定向課程之一。

Test Certified 測試認證

Initially, the larger and more complex parts of our codebase appeared resistant to good testing practices. Some projects had such poor code quality that they were almost impossible to test. To give projects a clear path forward, the Testing Grouplet devised a certification program that they called Test Certified. Test Certified aimed to give teams a way to understand the maturity of their testing processes and, more critically, cookbook instructions on how to improve it.

最初，我們的程式碼庫中較大和較複雜的部分似乎對良好的測試實踐有抵抗力。有些專案的程式碼品質很差，幾乎無法測試。為了給專案提供一個明確的前進道路，測試小組設計了一個認證計劃，他們稱之為測試認證。測試認證的目的是讓團隊瞭解他們的測試過程的成熟度，更關鍵的是，提供關於如何改進測試的說明書。

The program was organized into five levels, and each level required some concrete actions to improve the test hygiene on the team. The levels were designed in such a way that each step up could be accomplished within a quarter, which made it a convenient fit for Google's internal planning cadence.

該計劃分為五個級別，每個級別都需要一些具體的行動來改善團隊的測試狀況。這些級別的設計方式是，每個級別都可以在一個季度內完成，這使得它很適合谷歌的內部規劃節奏。

Test Certified Level 1 covered the basics: set up a continuous build; start tracking code coverage; classify all your tests as small, medium, or large; identify (but don't necessarily fix) flaky tests; and create a set of fast (not necessarily comprehensive) tests that can be run quickly. Each subsequent level added more challenges like "no releases with broken tests" or "remove all nondeterministic tests." By Level 5, all tests were automated, fast tests were running before every commit, all nondeterminism had been removed, and every behavior was covered. An internal dashboard applied social pressure by showing the level of every team. It wasn't long before teams were competing with one another to climb the ladder.

測試認證的第一級涵蓋了基礎知識：建立持續建構；開始追蹤程式碼覆蓋率；將你的所有測試分類為小型、中型或大型；識別（但不一定要修復）鬆散（不穩定）測試；建立一套可以快速執行的快速（不一定全面）測試。隨後的每一級都增加了更多的挑戰，如 "不釋出有問題的測試 "或 "刪除所有不確定性的測試"。到了第五級，所有的測試都是自動化的，快速測試在每次提交前都在執行，所有的不確定性都被移除，每一個行為都被覆蓋。一個內部儀表板透過顯示每個團隊的水平來施加競爭壓力。沒過多久，各團隊就開始互相競爭，爭先恐後。

By the time the Test Certified program was replaced by an automated approach in 2015 (more on pH later), it had helped more than 1,500 projects improve their testing culture.

到 2015 年測試認證專案被自動化方法取代時（後面會有更多關於 pH 值的介紹），它已經幫助超過 1500 個專案改善了他們的測試文化。

Testing on the Toilet 廁所測試

Of all the methods the Testing Grouplet used to try to improve testing at Google, perhaps none was more off-beat than Testing on the Toilet (TotT). The goal of TotT was fairly simple: actively raise awareness about testing across the entire company. The question is, what's the best way to do that in a company with employees scattered around the world?

在測試小組用來改善谷歌測試的所有方法中，也許沒有一種方法比 "廁所測試"（TotT）更離譜。TotT 的目標相當簡單：積極提高整個公司的測試意識。問題是，在一個員工分散在世界各地的辦公地，怎樣做才是最好的？

The Testing Grouplet considered the idea of a regular email newsletter, but given the heavy volume of email everyone deals with at Google, it was likely to become lost in the noise. After a little bit of brainstorming, someone proposed the idea of posting flyers in the restroom stalls as a joke. We quickly recognized the genius in it: the bathroom is one place that everyone must visit at least once each day, no matter what. Joke or not, the idea was cheap enough to implement that it had to be tried.

測試小組考慮了定期傳送電子郵件通訊的想法，但鑑於谷歌公司每個人都要處理大量的電子郵件，它很可能會在噪音中消失。經過一番頭腦風暴後，有人提出了在洗手間的隔間裡張貼海報的想法，作為一個玩笑。我們很快就認識到了其中的天才之處：無論如何，衛生間是每個人每天至少要去一次的地方。不管是不是玩笑，這個想法實施起來很簡單，所以必須嘗試一下。

In April 2006, a short writeup covering how to improve testing in Python appeared in restroom stalls across Google. This first episode was posted by a small band of volunteers. To say the reaction was polarized is an understatement; some saw it as an invasion of personal space, and they objected strongly. Mailing lists lit up with complaints, but the TotT creators were content: the people complaining were still talking about testing.

2006 年 4 月，一篇涵蓋如何改進 Python 測試的短文出現在整個谷歌的洗手間裡。這第一集是由一小群志願者釋出的。說反應兩極化是輕描淡寫的；一些人認為這是對個人空間的侵犯，他們強烈反對。郵件列表中的抱怨聲此起彼伏，但 TotT 的創造者們卻很滿意：抱怨的人仍在談論測試。

Ultimately, the uproar subsided and TotT quickly became a staple of Google culture. To date, engineers from across the company have produced several hundred episodes, covering almost every aspect of testing imaginable (in addition to a variety of other technical topics). New episodes are eagerly anticipated and some engineers even volunteer to post the episodes around their own buildings. We intentionally limit each episode to exactly one page, challenging authors to focus on the most important and actionable advice. A good episode contains something an engineer can take back to the desk immediately and try.

最終，喧囂平息下來，TotT 迅速成為谷歌文化的一個重要組成部分。到目前為止，來自整個公司的工程師已經制作了數百集，涵蓋了幾乎所有可以想象的測試方面（除了各種其他技術主題）。人們熱切期待著新的劇集，一些工程師甚至在自己的工位周圍張貼劇集。我們有意將每一集的篇幅限制在一頁以內，要求作者專注於最重要、最可行的建議。一集好的文章包含了工程師可以立即帶回到辦公桌上並進行嘗試的內容。

Ironically for a publication that appears in one of the more private locations, TotT has had an outsized public impact. Most external visitors see an episode at some point in their visit, and such encounters often lead to funny conversations about how Googlers always seem to be thinking about code. Additionally, TotT episodes make great blog posts, something the original TotT authors recognized early on. They began publishing lightly edited versions publicly, helping to share our experience with the industry at large.

具有諷刺意味的是，對於一個出現在比較隱秘的地方的出版物來說，TotT 已經產生了巨大的公共影響。大多數外部存取者在他們的存取中都會看到一集，而這樣的接觸往往會導致有趣的對話，即 Googlers 似乎總是在思考程式碼問題。此外，TotT 劇集是很好的部落格文章，這一點 TotT 的原作者很早就認識到了。他們開始公開發表輕量級的版本，幫助與整個行業分享我們的經驗。

Despite starting as a joke, TotT has had the longest run and the most profound impact of any of the testing initiatives started by the Testing Grouplet.

儘管開始時只是一個玩笑，但 TotT 在測試小組發起的所有測試活動中，執行時間最長，影響最深遠。

Testing Culture Today 當今的測試文化

Testing culture at Google today has come a long way from 2005. Nooglers still attend orientation classes on testing, and TotT continues to be distributed almost weekly. However, the expectations of testing have more deeply embedded themselves in the daily developer workflow.

與 2005 年相比，當前谷歌的測試文化已經有了長足的進步。Nooglers 仍然參加關於測試的指導課程，TotT 幾乎每週都會繼續分發。然而，對測試的期望已經更深入地嵌入到開發人員日常工作流程中。

Every code change at Google is required to go through code review. And every change is expected to include both the feature code and tests. Reviewers are expected to review the quality and correctness of both. In fact, it is perfectly reasonable to block a change if it is missing tests.

谷歌的每一次程式碼更改都需要經過程式碼審查。每一個變更都將包括特性程式碼和測試。評審員應評審這兩個檔案的品質和正確性。事實上，如果某個更改缺少測試，那麼阻止它是完全合理的。

As a replacement for Test Certified, one of our engineering productivity teams recently launched a tool called Project Health (pH). The pH tool continuously gathers dozens of metrics on the health of a project, including test coverage and test latency, and makes them available internally. pH is measured on a scale of one (worst) to five (best). A pH-1 project is seen as a problem for the team to address. Almost every team that runs a continuous build automatically gets a pH score.

作為測試認證的替代品，我們的一個工程生產力團隊最近推出了一個名為專案健康（pH）的工具。pH 工具不斷收集專案執行狀況的幾十個指標，包括測試覆蓋率和測試延遲，並使它們在內部可用。pH 值以 1（最差）到 5（最佳）的比例進行測量。pH-1 專案被視為團隊需要解決的問題。幾乎每個執行連續建構的團隊都會自動獲得 pH 分數。

Over time, testing has become an integral part of Google's engineering culture. We have myriad ways to reinforce its value to engineers across the company. Through a combination of training, gentle nudges, mentorship, and, yes, even a little friendly competition, we have created the clear expectation that testing is everyone's job.

隨著時間的推移，測試已經成為谷歌工程文化不可或缺的一部分。我們有很多方法來增強它對整個公司工程師的價值。透過培訓、輕推、指導，甚至一點友好的競爭，我們已經建立了一個明確的期望，即測試是每個人的工作。

Why didn't we start by mandating the writing of tests?

為什麼我們不開始強制編寫測試？

The Testing Grouplet had considered asking for a testing mandate from senior leadership but quickly decided against it. Any mandate on how to develop code would be seriously counter to Google culture and likely slow the progress, independent of the idea being mandated. The belief was that successful ideas would spread, so the focus became demonstrating success.

測試小組曾考慮要求高階領導提供測試授權，但很快決定拒絕。任何關於如何開發程式碼的要求都將嚴重違背谷歌文化，並且可能會減緩進度，這與被授權的想法無關。人們相信成功的想法會傳播開來，因此重點是人如何展示成功。

If engineers were deciding to write tests on their own, it meant that they had fully accepted the idea and were likely to keep doing the right thing—even if no one was compelling them to.

如果工程師們決定自己寫測試，這意味著他們已經完全接受了這個想法，並有可能繼續做正確的事情——即使沒有人強求他們這樣做。

The Limits of Automated Testing 自動化測試的侷限

Automated testing is not suitable for all testing tasks. For example, testing the quality of search results often involves human judgment. We conduct targeted, internal studies using Search Quality Raters who execute real queries and record their impressions. Similarly, it is difficult to capture the nuances of audio and video quality in an automated test, so we often use human judgment to evaluate the performance of telephony or video-calling systems.

自動測試並不適合所有的測試任務。例如，測試搜尋結果的品質通常需要人工判斷。我們使用搜索品質評測員進行有針對性的內部研究，他們執行真實的查詢並記錄他們的印象。同樣，在自動測試中很難捕捉到音訊和影片品質的細微差別，所以我們經常使用人工判斷來評估電話或視訊通話系統的效能。

In addition to qualitative judgements, there are certain creative assessments at which humans excel. For example, searching for complex security vulnerabilities is something that humans do better than automated systems. After a human has discovered and understood a flaw, it can be added to an automated security testing system like Google's Cloud Security Scanner where it can be run continuously and at scale.

除了定性判斷外，還有一些人擅長的創造性評估。例如，搜尋複雜的安全漏洞是人工比自動化系統做得更好的事情。在人類發現並理解了一個漏洞之後，它可以被新增到一個自動化的安全測試系統中，比如谷歌的雲安全掃描，在那裡它可以被連續和大規模地執行。

A more generalized term for this technique is Exploratory Testing. Exploratory Testing is a fundamentally creative endeavor in which someone treats the application under test as a puzzle to be broken, maybe by executing an unexpected set of steps or by inserting unexpected data. When conducting an exploratory test, the specific problems to be found are unknown at the start. They are gradually uncovered by probing commonly overlooked code paths or unusual responses from the application. As with the detection of security vulnerabilities, as soon as an exploratory test discovers an issue, an automated test should be added to prevent future regressions.

這種技術的一個更概括的術語是探索性測試。探索性測試從根本上說是一種創造性的工作，有人將被測試的應用程式視為一個有待破解的難題，也許是透過執行一組意想不到的步驟或插入預料之外的資料。在進行探索性測試時，要發現的具體問題在開始時是未知的。它們是透過探測通常被忽視的程式碼路徑或來自應用程式的不尋常的反應而逐漸發現的。與安全漏洞的檢測一樣，一旦探索性測試發現了問題，應新增自動測試以防止將來出現倒退。

Using automated testing to cover well-understood behaviors enables the expensive and qualitative efforts of human testers to focus on the parts of your products for which they can provide the most value—and avoid boring them to tears in the process.

透過使用自動化測試來覆蓋被充分理解的行為，測試人員可以將昂貴的定性工作重點放在產品中他們可以提供最大價值的部分，並避免在這個過程中使他們感到無聊。

Conclusion 總結

The adoption of developer-driven automated testing has been one of the most transformational software engineering practices at Google. It has enabled us to build larger systems with larger teams, faster than we ever thought possible. It has helped us keep up with the increasing pace of technological change. Over the past 15 years, we have successfully transformed our engineering culture to elevate testing into a cultural norm. Despite the company growing by a factor of almost 100 times since the journey began, our commitment to quality and testing is stronger today than it has ever been.

採用開發者驅動的自動化測試是谷歌公司最具變革性的軟體工程實踐之一。它使我們能夠以更大的團隊建立更大的系統，比我們想象的要快。它幫助我們跟上了技術變革的步伐。在過去的 15 年裡，我們已經成功地改造了我們的工程文化，將測試提升為一種文化規範。儘管自旅程開始以來，公司增長了近 100 倍，但我們對品質和測試的承諾比以往任何時候都更加堅定。

This chapter has been written to help orient you to how Google thinks about testing. In the next few chapters, we are going to dive even deeper into some key topics that have helped shape our understanding of what it means to write good, stable, and reliable tests. We will discuss the what, why, and how of unit tests, the most common kind of test at Google. We will wade into the debate on how to effectively use test doubles in tests through techniques such as faking, stubbing, and interaction testing. Finally, we will discuss the challenges with testing larger and more complex systems, like many of those we have at Google.

本章旨在幫助你瞭解谷歌如何看待測試。在接下來的幾章中，我們將深入探討一些關鍵主題，這些主題有助於我們理解編寫好的、穩定的、可靠的測試意味著什麼。我們將討論單元測試的內容、原因和方式，這是谷歌最常見的測試型別。我們將深入討論如何透過模擬、打樁和互動測試等技術在測試中有效地使用測試替代。最後，我們將討論測試更大、更復雜的系統所面臨的挑戰，就像我們在谷歌遇到的許多系統一樣。

At the conclusion of these three chapters, you should have a much deeper and clearer picture of the testing strategies we use and, more important, why we use them.

在這三章的結尾，你應該對我們使用的測試策略有一個更深入更清晰的瞭解，更重要的是，我們為什麼使用它們。

TL;DRs 內容提要

- Automated testing is foundational to enabling software to change.
- For tests to scale, they must be automated.
- A balanced test suite is necessary for maintaining healthy test coverage.
- “If you liked it, you should have put a test on it.”
- Changing the testing culture in organizations takes time.
- 自動化測試是實現軟體變革的基礎。
- 為了使測試規模化，它們必須是自動化的。
- 平衡的測試套件對於保持健康的測試覆蓋率是必要的。

- "如果你喜歡它，你應該對它進行測試"。
- 改變組織中的測試文化需要時間。

---

1. See “Defect Prevention: Reducing Costs and Enhancing Quality.” ↩

2. See “Failure at Dhahran.” ↩

3. There is a little wiggle room in this policy. Tests are allowed to access a filesystem if they use a hermetic, in- memory implementation. ↩

4. Mike Cohn, Succeeding with Agile: Software Development Using Scrum (New York: Addison-Wesley Professional, 2009). ↩

5. Keep in mind that there are different kinds of coverage (line, path, branch, etc.), and each says something different about which code has been tested. In this simple example, line coverage is being used. ↩

6. Each supported language at Google has one standard test framework and one standard mocking/stubbing library. One set of infrastructure runs most tests in all languages across the entire codebase. ↩

7. This class was so successful that an updated version is still taught today. In fact, it is one of the longest- running orientation classes in the company’s history. ↩