# 前言

> 大腦不是要人裝填的容器，而是要人點燃的木材。
>
> — 希臘史學家・*Plutarch*

每位 Java 程式設計師都應該知道的事情是什麼？這點會因情況而異，取決於你向誰詢問、詢問的理由是什麼，以及你詢問的時機點。因此，有多少觀點，就至少會有多少建議出現。尤其是在一個影響軟體與眾人生活甚鉅的程式語言、平台、生態系統和社群裡，其影響力從一個世紀到下一個世紀、從一個核心延伸出許多核心、從百萬位元組到十億位元組，取決的條件會比你希望一位作者在一本書裡涵蓋的內容還多。

與其如此，本書借鑑眾多觀點之中的一些想法，為讀者匯聚成集體智慧，呈現 Java 詩句中橫跨不同面向的代表性思維。本書當然無法納入所有知識，但這些內容是由 73 位優秀的程式設計師貢獻他們的智慧，歸納而成的 97 件事。引用《程式設計人應該知道的 97 件事》（97 Things Every Programmer Should Know，歐萊禮出版）的前言所述：

> 有太多知識需要知道，有太多事情需要去做，又有太多的方法可以實現，沒有任何一個人或單一來源能自稱擁有「獨一無二的解決方法」。這些文稿並沒有像木工榫件一樣地相互契合，作者們並非以此為目的，——若要說起來，甚至還相互對立。每一篇文稿的價值在於各自的獨特性，而整本書的價值在於每一篇文稿是如何實現、證實，甚至是互相牴觸。這裡並沒有整體的結論：而是讓你去回應、反省、串起所有你閱讀過的訊息，並基於你自己的狀況、知識及經驗做出衡量。

那麼，每位 Java 程式設計師都應該知道的事情是什麼？在本書精選出來的 97 件事裡，答案遍及程式語言、JVM、測試技巧、JDK、社群、歷史、敏捷思維、應用實務知識、職業精神、程式風格、介面函式庫 Substance、程式設計典範、程式設計師身為人的一面、軟體架構、程式碼背後的技術能力、工具化、Java 垃圾回收機制、JVM 上的非 Java 語言等等。

## 使用許可

延續《*97 件事*》系列書籍最初的精神，本書的每一篇文章均依循不限制使用的開放授權模式，採用創用 CC 授權 4.0 版條款（*https://oreil.ly/zPsKK*）。許多文章也是首次在《*97 件事*》網路發行媒體上曝光（*https://medium.com/97-things*），所有這些內容都將成為點燃你思考與程式碼創作的燃料。

## 致謝

許多人直接與間接貢獻他們的時間和見解予《*Java 程式設計師應該知道的 97 件事*》，他們是這個專案成功的最大功臣。在此我們要謝謝所有曾經投入時間和精力貢獻予本書的人士，還要特別感謝 Brian Goetz 撥冗提供他對本書的回饋、意見和建議。感謝歐萊禮團隊提供予本專案的所有支持。謝謝 Zan McQuade、Corbin Collins 指導專案並且培養本書內容的貢獻者，也謝謝 Rachel Roumeliotis、Susan Conant、Mike Loukides 在整個專案進行過程中的貢獻。

身兼本書撰稿者之一與彙編者的 Kevlin 要在此感謝他的太太 Carolyn，謝謝她理解他滿腦子荒謬的理想，也謝謝他的兩個兒子 Stefan 和 Yannick 能理解他們的父母。

同樣也是本書撰稿者之一與彙編者的 Trisha 要補充她對丈夫 Isra 的感謝之意，謝謝他幫忙排解壓力，讓她了解到覺得自己做得不夠好這樣的情緒無助於解決任何事情，還要謝謝她的兩個女兒 Evie 和 Amy，無條件給予她愛和擁抱。

我們希望本書能提供讀者知識性的內容、精闢的獨特見解，同時具有啟發性。請各位盡情享受本書所帶來的閱讀饗宴！

# 你只需要 Java

*Anders Norås*

當年 Visual Studio 還在進行第一次重大改版時，微軟團隊就已經為這個世界導入三位開發者人物誌（persona）：Mort、Elvis 和 Einstein。

Mort 是投機型的開發人員，在工作過程中往往會快速修正問題並且喜歡無中生有；Elvis 是務實型的程式設計師，多年來從事解決方案的開發工作，持續從工作中學習；Einstein 則是偏執型的程式設計師，著迷於設計最有效率的解決方案，著手寫程式碼之前會先搞清楚每一件事。

程式語言裡也有宗教分歧，在 Java 方面，我們嘲笑 Mort 這類的人，希望成為 Einstein 們，跟他們一樣建立框架，確保 Elvise 這些人以「正確的方法」寫出程式碼。

在那個年代，除非你精通最新、最厲害的物件關聯映射（object relational mapper）和控制反轉框架（inversion of control framework），否則你就不是夠格的 Java 程式設計師。隨著框架時代崛起，函式庫逐漸發展為具有規範的架構，這些框架後來還變成了技術生態系統，於是，我們很多人都忘記了 Java 這個小型語言可以做的事。

Java 是很棒的程式語言，針對每種情況都有某個類別函式庫可以派上用場。你需要處理檔案嗎？`java.nio` 類別可以幫你搞定；資料庫呢？`java.sql` 類別是你理想的去處。幾乎每個 Java 版本甚至還能突變為發展成熟的 HTTP 伺服器，即使你必須捨棄 Java-named 類別，轉而使用 `com.sun.net.httpserver` 套件。

隨著應用程式朝向無伺服器架構（serverless architecture）發展，部署單元變成一個個單一函式，我們原本從應用程式框架獲得的優勢正逐漸流失。這是因為我們可能會花更少的時間處理技術和基礎設施方面的議題，轉而將我們在程式設計上的精力投入程式要實現的商業能力。

如同美國教育家 Bruce Joyce 所述：

> 每隔一段時間我們就必須改造輪子，不是因為我們需要大量的輪子，而是因為我們需要大量的發明家。

許多人著手建立通用的商業邏輯框架，希望將再利用性提到最大的程度，但是多數人都失敗了，因為沒有任何商業問題真的存在共通性。以特定方法進行某件特別的事情，正是一項商業活動和其他商業活動之間有所區別之處，這也就是為什麼幾乎每個專案都一定會撰寫商業邏輯的原因。打著共通性和再利用性的名義提出某種做法，企圖導入規則引擎或是其他類似的東西，說穿了，配置規則引擎就是程式設計，而且往往還使用比 Java 更不如的程式語言。既然如此，為何不嘗試只用 Java 來撰寫程式呢？最後的結果將令你驚豔，你會發現程式碼易於閱讀，連非 Java 程式設計師的人也能輕鬆維護程式碼。

雖然你常常會發現 Java 類別函式庫有些限制，可能需要借助某些力量才能順利處理日期、網路等等其他功能，但是沒關係，此時利用函式庫就好，差別在於你現在是因為發生特殊需求才利用函式庫，而不是像過去因為函式庫是你一直在使用的堆疊的一部分才用。

下次當你腦海裡冒出小型程式 Spring 框架的想法時，請將你的 Java 類別函式庫知識從冬眠中喚醒，而不是伸手用 JHipster 搭起鷹架。趕流行的做法已經過時，簡約生活才是現在的主流，我敢打賭 Mort 一定會喜歡簡約生活。

# 認定測試

*Emily Bache*

你是否曾經撰寫類似以下的程式碼，在測試時利用空值或空白作為斷言的期望值呢？

```
assertEquals("", functionCall())
```

在上面的程式碼裡，你無法十分確定 `functionCall` 函式回傳的字串應該會是什麼，但是當你看到字串時，你知道那是正確的嗎？當然，第一次執行測試時，因為 `functionCall` 函式回傳的不是空字串，所以測試會失敗。（你可能會做多次嘗試，直到回傳值看起來正確為止。）然後你貼上這個回傳值，取代 `assertEquals` 函式裡的空字串，現在斷言測試應該是通過了，終於！這就是我說的**認定測試**（approval testing）。

此處關鍵的一步是，當你決定正確的輸出結果後，利用該項結果作為期望值；也就是說，你「核准」了這個結果，認為它足以保留。我希望你在沒有實際思考之前，就已經在做這樣的事，或許你稱之為**快照測試**（snapshot testing）或**特徵測試**（golden master testing）。在我的經驗裡，如果你有專為支持這項測試方法而設計的架構，多數工作都能按部就班進行，也能更順利進行這個測試方法。

使用像 JUnit 這類經典的單元測試架構時，當你需要更新這些作為期望值的字串會有一點痛苦，最終你要在原始程式碼裡到處貼入修改的資料。反之，如果是利用認定測試工具，會先將核准過的字串儲存在一個檔案裡，這立即開啟全新的可能性。你可以利用適合的差異性工具審視要更改的字串，並且將它們全部合併在一個檔案裡，以語法標示出 JSON 字串，還可以在橫跨不同類別的數個測試裡，搜尋和替換更新。

那麼，有哪些情況適合利用認定測試呢？這裡有幾個想法：

不需要更改單元測試的程式碼

如果程式碼已經在營運環境中，原則上，這份程式碼做的任何事都要被認為是正確的而且被核准。建立測試時的難題會變成找出程式碼之中的接縫處（seam），切分出某些你能核准的有趣邏輯區塊。

回傳 JSON 或 XML 的 REST API 和函式

如果你的結果是較長的字串，將字串儲存在原始程式碼之外的人才是最大的贏家。JSON 和 XML 兩者都可以利用一致性的空白字元格式化，如此一來，便能輕鬆與期望值比對。如果 JSON 和 XML 裡的值差異很大 —— 例如，日期和時間，在你以固定字串取代這些值還有核准剩餘的值之前，可能需要分別檢查它們。

建立複雜回傳物件的商業邏輯

一開始要先撰寫 `Printer` 類別，將複雜回傳物件格式化為一個字串。請思考一下**收據**、**處方籤**或是**訂單**，不管是哪一種文件都能適當地以讓人類易懂、多行字串的方式呈現。`Printer` 類別可以選擇只印出摘要 —— 查找整個物件圖，從中拉出相關細節。測試會執行各種商業規則，並且利用 Printer 類別去建立作為核准使用的字串。即使產品負責人或業務分析師沒有程式設計方面的背景，甚至也能讓他們看懂測試結果，並且查證結果的正確性。

如果你已經寫好一些測試，裡面有斷言要驗證長度超過一行的字串，那麼我會建議你找出更多認定測試方面的資訊，開始利用支持這項測試的工具。

# 利用 AsciiDoc 強化 Javadoc

*James Elliott*

Java 開發人員都知道 Java 文件產生器 —— Javadoc，那些在程式界打滾已久的人應該都還記得 Javadoc 當年是如何讓大家徹底改觀。當 Java 成為第一個主流程式語言時，它將文件產生器直接整合到編譯器裡，成為標準工具鏈。結果造成 API 文件的數量爆炸性增加（即使不是所有文件都很棒或是精美），讓我們所有人大幅受益，這股趨勢還傳到許多其他語言裡。如同 Java 之父 James Gosling 所述（*https://oreil.ly/Y_7rk*），Javadoc 剛推出的時候引起很多爭議，因為「一名好的技術文件工程師可以做更多的工作而且品質更好」，然而現實是 API 的數量遠超過能撰寫文件的技術文件工程師的人數，而且 Javadoc 在協助某件事情普及上，已經確實建立起它的價值。

不過，有時候你需要建立的文件不只是 API 說明文件，你需要的內容不只有 Javadoc 在套件和專案概要裡提供的內容，像是面向終端使用者提供的指南和操作指令、詳細說明架構和理論的背景、解釋多個元件之間如何相互配合……等等，這些內容全都無法納入 Javadoc。

那麼，我們要採用什麼方法來滿足這些其他方面的需求？這個問題的答案隨時空背景不同而改變。在 80 年代，當時的強者是 FrameMaker，不僅相當創新而且是可以跨平台的 GUI 技術文件，Javadoc 甚至還曾經納入 MIF Doclet，利用 FrameMaker 產生精美印刷的 API 文件，但僅有功能不全的 Windows 版本。後來，DocBook XML 提供了類似的結構和連結功能，還具有開放規格和跨平台的工具，但要直接處理原始的 XML 格式卻是不切實際的想法，再加上其編輯工具逐漸變得昂貴而且無趣，就連優秀的編輯器都讓人感到笨重而且妨礙文件的撰寫流程。

非常高興我現在已經找到更好的答案：AsciiDoc（*https://oreil.ly/NYrJI*），其以容易撰寫（而且易懂）的文字格式，提供 DocBook 所具備的一切功能，輕而易舉就能處理簡單的工作，也可能解決一些複雜的任務。大部分的 AsciiDoc 結構就跟 Markdown 這類輕量級標記式語言的格式一樣，立刻就能看懂而且易於理解，漸漸透過線上論壇廣為人知。此外，當你需要花俏的內容時，可以利用 MathML 或 LaTeX 格式加入複雜的方程式、在原始程式碼清單的格式裡加入編號並且連結文字段落、顯示不同種類的警告區塊等等。

AsciiDoc 最早是在 2002 年導入 Python 實作之中，目前的正式實作版本（和管理語言）是 2013 年釋出的 Asciidoctor（*https://oreil.ly/aRRvG*），其以 Ruby 語言撰寫的程式碼也能透過 AsciidoctorJ（利用外掛程式 Maven 和 Gradle）在 JVM 上運行（*https://oreil.ly/UT8EP*），或者是將程式碼轉譯成 JavaScript（*https://oreil.ly/E_6qn*），兩者都能在持續整合的環境中運作良好。當你需要建立整個網站的相關文件（甚至是來自於多個資源庫），像 Antora（*https://antora.org*）這樣的工具就能讓你的工作變得極為輕鬆。AsciiDoc 的社群（*https://oreil.ly/PtWwa*）相當友善而且提供多方面的支持，看著它過去這幾年的成長和進步，令人相當振奮。而且，如果你對 AsciiDoc 有興趣的話，它正在進行正式規格的標準化處理流程（*https://oreil.ly/BaXa8*）。

我喜歡為自己分享的專案建立豐富而且吸引人的說明文件（*https://oreil.ly/H_rSW*），AsciiDoc 正好可以幫助我簡化這方面的工作，它提供如此快速的處理週期，把建立精美而且完善文件的工作變得如此有趣（*https://oreil.ly/7sbtj*）。我希望你也能從中發現相同的樂趣，而且，繞了一圈回來之後，你會發現許多方法彼此之間都有關係。如果你決定全部文件都採用 AsciiDoc，Java 甚至還支援一個 Asciidoclet 的 Doclet 元件（*https://oreil.ly/9KgQq*），讓你能利用 AsciiDoc 撰寫 Javadoc ！

# 請特別注意容器周遭的環境

*David Delabassee*

將傳統 Java 應用程式及其使用的舊版 Java 虛擬機器（JVM）封裝在容器裡會有危險，因為在 Docker 容器（container）裡執行時，這些舊版 JVM 在處理垃圾回收機制的優化過程（ergonomics）中會發生混淆的情況。

事實上，容器現在已經變成程式執行期間的封裝機制，其提供的優點有：具有一定程度的隔離效果、提高資源利用效率、可以橫跨不同環境部署應用程式等等。當應用程式封裝到可攜式容器裡，容器還有助於降低應用程式和底層平台之間的耦合性，因此，這項技術有時會用在協助傳統應用程式現代化的工作上。在 Java 的環境裡，容器嵌入傳統 Java 應用程式時會一併納入該應用程式使用的關聯函式庫，包含其使用的舊版 JVM。

將傳統 Java 應用程式及其環境嵌入容器裡，實務上的確能藉由將它們完全從舊有、無法支援的基礎設施環境中解耦，幫助舊版應用程式繼續在現代環境中運行，獲得現有基礎設施的支援。這種做法在實務上可能會帶來一些好處，但是，在 JVM 處理垃圾回收機制的優化過程中也可能伴隨一連串的風險。

JVM 處理垃圾回收機制的優化過程（ergonomics，*https://oreil.ly/h3hTh*）是透過兩個關鍵環境指標來校調 JVM 本身的效能：CPU 數量和可用記憶體。JVM 利用這些指標來決定一些重要參數，例如，該使用哪一個垃圾回收器、如何設定垃圾回收器、堆積記憶體的大小、ForkJoinPool 物件的大小等等。

JDK 8 釋出的更新 191（*https://oreil.ly/C_1AW*）讓 Linux Docker 容器可以支援 JVM 依賴 Linux cgroups 機制（*https://oreil.ly/nDIwb*），取得 JVM 運行容器所分配到的資源指標。JDK 8 版本以前的所有舊版 JVM 都不知道自己正在容器環境下運行，所以會從主機作業系統取得指標資料，而非從容器本身，可是，在大部分的情況下，主機只會配置一部份的資源給容器，所以從主機取得指標的 JVM 就會仰賴不正確的指標來校調自身的效能，導致不穩定的情況快速發生。這是因為容器試圖消耗更多資源，在超出本身所擁有的可用資源情況下，主機可能會強制刪除容器。

以下命令說明 JVM 處理垃圾回收機制的優化過程中，JVM 設定了哪些參數：

```
java -XX:+PrintFlagsFinal -version | grep ergonomic
```

JVM 容器支援在預設情況下會處於啟用狀態，但是，可以利用 JVM 旗標 `-XX:-UseContainerSupport` 停用這項支援。在（CPU 和記憶體）資源有限的容器裡，利用這項旗標，你可以觀察和探討有無容器支援對 JVM 在處理垃圾回收機制優化過程的影響。

在 Docker 容器裡運行舊版的 JVM 明顯不是一個好的建議方案，但是，如果這是你唯一的選擇，至少應該設定舊版 JVM 使用的資源不能超過其運行的容器環境本身所分配到的資源。顯然，理想的解決方案是使用有支援的新版 JVM（例如，JDK 11 以上的版本），在預設狀態下不僅能獲得容器支援，還能提供最新而且安全的程式執行環境。

# 行為引起的問題很「簡單」，困難的是由狀態引起的問題

*Edson Yanaga*

當年我第一次接觸物件導向程式設計時，剛開始教給我的三個基礎觀念是：多型（polymorphism）、繼承（inheritance）和封裝（encapsulation）。說真的，我們花了相當多的時間，嘗試理解這些觀念並且利用它們撰寫程式碼，但是，我覺得過於強調前兩者（至少對我來說是如此），卻很少著墨於第三項而且也是最重要的一項：封裝。

封裝讓我們有能力駕馭軟體開發領域裡不斷成長的狀態和複雜性，這已經變成一種常態。設計複雜資訊系統和撰寫其程式碼時，核心想法是我們可以將狀態初始化、對其他元件隱藏狀態，以及針對任何狀態發生變異的情況，只提供精心設計過的 API surface 元件。

然而，在建立完善封裝系統這方面，我們無法將一些最佳實務做法擴散出去（至少在 Java 界是如此）。JavaBean 的屬性經常可以看到貧血類別（anemic class），透過 *getter* 和 *setter*，簡單地暴露出內部狀態。從 Java Enterprise 的架構裡，我們似乎看見這樣的觀念已經十分普及，即使不是所有也可以說是大部分的商業邏輯應該在 Service 類別裡實作，在這些類別裡利用 getter 獲取資訊、處理 getter 以獲得結果，然後利用 setter 將結果放回物件裡。

等到程式發生臭蟲時，我們從紀錄檔案裡挖掘線索、使用偵測器，嘗試搞清楚營運環境裡的程式碼究竟發生了什麼問題。要指出由行為問題引起的臭蟲相當「容易」，就是：某些區塊的程式碼正在做某種它們不應該做的行為；另一方面，當我們的程式碼似乎表現正常卻仍舊出現臭蟲，想要搞清楚問題在哪裡就會變得更加複雜。從我過往的經驗來看，最難解決的臭蟲是由**狀態不一致**所引起的問題。你的系統達到不應該

發生的狀態，但事實上，它就是發生了 —— 某個屬性永遠不應該出現空值、原本應該是正的資料值卻變成負的等等，諸如此類的情況造成了 `NullPointerException`。

出現我們需要找出引起這種狀態不一致的步驟的情況，發生的可能性很低。程式類別的表面太容易改變而且輕鬆就能獲取：系統裡任何一塊程式碼，隨處都可能在不經由任何檢查或調整的情況下，讓我們的狀態產生變異。

儘管我們能透過驗證框架，針對使用者提供的輸入資料採取一些安全措施，但是「無辜的」setter 卻仍舊允許任何一塊程式碼呼叫它。我甚至不想討論這種可能性，但某個人可能會在資料庫裡直接使用 `UPDATE` 陳述式，更改資料庫映射內容裡的某些欄位。

我們該如何解決這個問題？可能的答案之一是**不可變異性**（Immutability）。如果我們能保證物件永遠不會出現變異的情況，並且在建立物件時檢查狀態一致性，系統就永遠不會發生狀態不一致的問題。但是，我們必須考慮到多數 Java 框架都不擅長處理不可變異性，所以我們至少應該努力將可變異性降到最低；適當地利用編碼過的工廠方法和編譯器，也有助於達成狀態可變異性最小化的目的。

因此，請不要自動產生 setter。花點時間思考，你的程式碼裡是否真的需要這個 setter ？如果你最後決定使用 setter（或許因為某些框架的要求），請考慮加上**防損毀層**（anticorruption layer），在 setter 進行這些互動後，保護與檢查內部狀態。

# 基準測試很難，但 JMH 能幫助你完成

*Michael Hunger*

在 JVM 上要進行基準測試（benchmarking）很難，尤其是微基準測試（microbenchmarking）更是難上加難。每奈秒圍繞呼叫或迴圈完成測試還不夠，你還必須考慮測試前的預熱準備工作（warm-up）、熱點編譯（HotSpot compilation）、程式碼最佳化，例如，刪除單一檔案和沒有用到的程式碼、多執行緒、衡量方法的一致性等等。

幸運的是，開發出許多優秀 JVM 工具的作者 —— Aleksey Shipilëv 為 OpenJDK 貢獻了一套微基準測試框架（Java Microbenchmarking Harness，簡稱 JMH，*https://oreil.ly/gR0fd*）。這套框架是由一個小型的函式庫和編譯系統外掛程式所組成，函式庫提供了標註和實用程序，宣告測試基準為已經標註的 Java 類別和方法，其中包含 `BlackHole` 類別，用以消耗產生出來的值，避免程式碼被刪除；在多執行緒的環境下，函式庫還提供正確的狀態處理程序。

編譯系統外掛程式則負責產生 JAR 檔案，包含為了正確執行和衡量測試所需要的相關基礎結構程式碼，其中設定了專用的預熱階段、適合的多執行緒、執行多個 fork 函數和計算函數之間的平均值等等。

這項工具還會輸出重要的建議，說明如何使用收集到的資料和其中的限制。以下這個範例是衡量程式預先配置集合大小的影響：

```
public class MyBenchmark {
    static final int COUNT = 10000;
    @Benchmark
    public List<Boolean> testFillEmptyList() {
        List<Boolean> list = new ArrayList<>();
        for (int i=0;i<COUNT;i++) {
            list.add(Boolean.TRUE);
        }
```

```
        return list;
    }
    @Benchmark
    public List<Boolean> testFillAllocatedList() {
        List<Boolean> list = new ArrayList<>(COUNT);
        for (int i=0;i<COUNT;i++) {
            list.add(Boolean.TRUE);
        }
        return list;
    }
}
```

你可以利用 JMH 的 Maven archetype 命令來產生專案和執行專案：

```
mvn archetype:generate \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DinteractiveMode=false -DgroupId=com.example \
-DartifactId=coll-test -Dversion=1.0

cd coll-test

# 新增：com/example/MyBenchmark.java

mvn clean install

java -jar target/benchmarks.jar -w 1 -r 1

...
# JMH 版本：1.21
...
# 預熱準備：5 次迭代，每次一秒
# 衡量：5 次迭代，每次一秒
# 逾時：每次迭代十分鐘
# 執行緒：一個，和迭代同步。
# 基準測試模式：吞吐量（ops / time）
# 基準測試：com.example.MyBenchmark.testFillEmptyList

...

Result "com.example.MyBenchmark.testFillEmptyList":
30966.686 ±(99.9%) 2636.125 ops/s [Average]
(min, avg, max) = (18885.422, 30966.686, 35612.643), stdev = 3519.152
CI (99.9%): [28330.561, 33602.811] (assumes normal distribution)
```

```
# 完成執行。總時間：00:01:45

請記住：以下的數字只是資料，你必須追蹤這些數字背後的原因，
才能獲得可重複利用的觀點。你可以使用剖析器（請見 -prof、-lprof）、
設計因子實驗、執行基線測試和負向測試，以提供實驗控制條件，
確保基準測試環境在 JVM / OS / HW 層級的安全性，
請這個領域裡的專家審視實驗結果，
但是請不要假設數字傳達的意義就是你希望的那樣。
基準測試                                模式    次數    分數          錯誤          單位
MyBenchmark.testFillAllocatedList       吞吐量  25      56786.708   ± 1609.633    ops/s
MyBenchmark.testFillEmptyList           吞吐量  25      30966.686   ± 2636.125    ops/s
```

所以，我們看到預先配置集合的速度幾乎比預設情況快上兩倍，這是因為元素新增期間不必重新調整大小的緣故。

撰寫正確的微基準測試時，JMH 會是你工具箱裡強大的工具。如果你在相同的環境裡執行測試，甚至能相互比較，這才應該是闡述測試結果的主要方法；此外，還能用於剖析目的，因為它們能提供穩定、可重複的結果。如果你對這方面有興趣，Aleksey 分享了更多關於這個主題的想法（*https://oreil.ly/5zWU1*）。

# 程式碼結構品質程式化與驗證的優點

*Daniel Bryant*


持續交付編譯工作流程應該是主要負責應用程式結構品質的地方，將眾人一致同意的品質程式化並且加以嚴格執行。然而，這些負責自動驗證品質的工作流程不應該取代團隊在標準和品質層面的持續討論，而且絕對不能用來避開團隊內部或團隊之間的溝通；也就是說，在編譯工作過程裡檢查和發布品質指標，可以防止結構品質逐漸低落，否則團隊可能很難注意到這個情況。

如果你對為什麼應該測試程式碼結構背後的原因很好奇，這個（*https://oreil.ly/q1OCY*）說明 ArchUnit 動機的網頁能讓你搞懂一切。ArchUnit 的問世起始於一個大家都很熟悉的故事：從前從前，有一位架構師畫了一系列漂亮的架構圖，利用這些圖來表示系統裡的各個元件，以及元件之間應該如何互動。後來，專案的規模變得越來越大，使用者案例也變得越來越複雜，有新的開發人員加入專案，也有舊的開發人員離開，最終變成開發人員覺得哪個方法適合，就以那個方法來加入新功能。不久之後，所有元件都相互依賴，團隊無法預測任何改變會對專案裡所有其他元件帶來什麼樣的影響。我敢肯定許多讀者都能體會我說的這個情境。

ArchUnit（*https://www.archunit.org*）是一個開放原始碼、可擴充的函式庫，利用 Java 單元測試（像是 JUnit 或 TestNG）來檢查 Java 程式碼的結構。ArchUnit 藉由分析位元組碼和匯入所有類別來進行一切的檢查工作，可以檢查迴圈依賴關係，以及套件與類別、呼叫層次與切面之間的依賴關係等等。

在 JUnit 4 的環境下加入以下來自 Maven 中央資源庫的依賴資源，就可以使用 ArchUnit 工具：

```
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit-junit</artifactId>
    <version>0.5.0</version>
    <scope>test</scope>
</dependency>
```

ArchUnit 核心提供的基礎結構是將 Java 位元組碼匯入 Java 程式碼結構裡。做法是利用 `ClassFileImporter` 物件，使用類 DSL 流暢 API，依序評估匯入的類別，制定像「只有控制器才能獲得服務」這樣的結構規則：

```
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition;
// ...
@Test
public void Services_should_only_be_accessed_by_Controllers() {
    JavaClasses classes =
        new ClassFileImporter().importPackages("com.mycompany.myapp");
    ArchRule myRule = ArchRuleDefinition.classes()
        .that().resideInAPackage("..service..")
        .should().onlyBeAccessed()
        .byAnyPackage("..controller..", "..service..");
    myRule.check(classes);
}
```

延伸前面這個範例，你還可以利用這個測試加強更多以層次為基礎的存取規則：

```
@ArchTest
public static final ArchRule layer_dependencies_are_respected =
layeredArchitecture()
.layer("Controllers").definedBy("com.tngtech.archunit.
eg.controller..")
.layer("Services").definedBy("com.tngtech.archunit.eg.service..")
.layer("Persistence").definedBy("com.tngtech.archunit.
eg.persistence..")
.whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
.whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
.whereLayer("Persistence").mayOnlyBeAccessedByLayers("Services");
```

此外，還能確保程式碼有遵照命名慣例（例如，名稱的前置詞），或者是指定以某種方式命名的類別必須放在適當的套件裡。GitHub 平台上有一系列使用 ArchUnit 的範例（*https://oreil.ly/Xv8CI*），不僅能幫助你起步，還有許多想法供你參考。

你可以嘗試請有經驗的開發人員或架構師每週幫你看一次程式碼，找出有違反規則的程式碼並且修正它們，藉此偵測和修改所有本文提過的程式碼結構問題。然而，人類前後行為不一致的特性可是出了名的，當專案逃不過壓在身上的時間壓力，通常第一個被犧牲掉的項目就是手動驗證。

更實際的方法是利用自動化測試、ArchUnit 或其他工具，將眾人一致認同的結構標準和規則程式化，並且將其納入、成為持續整合編譯流程裡的一部份，日後發生這個問題的工程師能快速偵測並且修正。

# 將問題和任務拆解成小的工作區塊

*Jeanne Boyarsky*

目前正在學習程式設計的你收到了一份小作業，要寫一千行以下的程式碼。於是，你輸入程式碼並且測試，然後加了一些列印陳述式、使用了偵錯器，這中間可能還去拿了杯咖啡，接著陷入苦思。

聽起來是不是很耳熟？這還只是一個經過簡化的玩具問題，現實生活中的工作任務和系統遠比這還要龐大。解決大型問題需要時間，更糟的是，太多事物占據在你的大腦記憶體裡。

有一個好方法能解決這個處境，就是將大問題分解成一塊塊的小問題，而且是越小越好。如果你能處理每一塊小問題，就不會再陷入苦思之中，可以繼續處理下一個問題。當你能順利處理問題時，就會想為每個小問題撰寫自動化測試。不僅如此，你還應該增加提交程式碼的頻率，當日後工作進行狀況不如預期時，你還能有回溯點。

我記得以前曾經在某個團隊裡，幫助過某位陷入工作困境的成員。因為最簡單的修復方法是回溯到前面版本的程式碼，然後重改程式，於是我問他最後一次提交程式碼是什麼時候，他回答我「一個禮拜前」，這下就蹦出兩個問題了：一個是原來卡住的問題，另一個問題是我不想幫他除錯一整個禮拜份量的程式碼。

有了這次的經驗後，我為團隊辦了一場訓練課程，內容是如何將工作任務分解成更小的工作區塊。那時有幾位資深開發人員告訴我，他們的工作任務「很特別」而且「不太可能拆解」。當你聽到**很特別**這個詞和工作任務連結在一起，應該立刻抱持懷疑的態度。

當下我決定再安排第二次會議。我請每位出席會議的人都提出一個例子——一個他們認為「很特別」的工作任務，然後由我來協助他們拆解

工作區塊。第一個例子是預計要花兩週時間開發的畫面，我將開發工作拆解成：

- 在正確的網址建立一個 *hello world* 的畫面——沒有資料，就只是在畫面上印出 *hello world*。
- 新增功能，用以顯示資料庫清單。
- 新增文字區域。
- 新增下拉式選單。
- 〈一長串更多細小的工作任務。〉

你猜猜看這麼做會發生什麼事？在這些細小的工作任務裡，每當有一個項目完成後可能就會提交一次程式碼，這意味著一天之中可能會提交很多次程式碼。

然後，有一位開發人員告訴我，畫面的開發工作可以用這種方式完成，但是檔案處理的工作「很特別」。現在，**很特別**這個詞又出現了，我會怎麼說？當然也是將這個工作拆解：

- 從檔案裡讀取的一行資料。
- 驗證第一個欄位，包含資料庫呼叫。
- 驗證第二個欄位，並且利用商業邏輯轉換這項資料。
- 〈一長串資料欄位。〉
- 將第一個商業邏輯規則應用在所有欄位。
- 〈一堆規則。〉
- 新增訊息到佇列裡。

同樣地，這個工作任務一點也不特別。如果你認為某項工作任務很特殊，請停下來思考一下為什麼，往往會發現仍然可以套用這個技巧。

最後，有一位開發人員告訴我說，他無法在一週的時間內提交出程式碼。這個工作任務後來重新分配到我手上，開發過程中我特別多提交了幾次程式碼。最後我花了兩天完成這項工作任務，計算了一下，兩天之中提交了 22 次程式碼。如果這位開發人員願意增加提交的頻率，我想他會更快完成工作！

# 建立多元化的團隊

*Ixchel Ruiz*

多年前，一位優秀的醫生是無所不知、無所不能：整骨、動手術、抽血，樣樣精通；優秀的醫生要獨立而且自己搞定所需要的一切，因此，自主性獲得高度重視。

時間快轉到今日，這是一個知識爆炸的年代，凌駕個人之上，帶來了專業化。為了從頭到尾提供一個完整、適當的解決方案，需要許多專家一起參與，許多團隊必須一起互動。

這點在軟體開發中也是如此。

現在，能與他人合作變成「優秀」專業人士身上最有價值的特質之一。過去，獨立而且能自己搞定一切的人就足以稱為「優秀」，但是現今我們所有人都必須表現得像後勤維修人員，也就是：團隊成員。

因此，我們的挑戰就是要組一支成功又多元化的團隊。

和創新有正相關的多元性有四種：產業背景、原來的國籍、職涯道路與性別。在同質性團隊裡，不論團隊成員的學術背景如何，都可能出現重複的觀點，例如，女性會帶來破壞性創新。

性別對團隊的影響有多大？我們已經觀察到，在性別多元化高的管理團隊中，來自創新的營收增加了 8%。

小組成員之間的差異性也會是各種見解的來源——具有不同背景、經驗和想法的成員會增加資訊、技能和人際關係的共用資源。從更多角度來看，要達成團隊共識需要進行建設性的辯論。如果整體環境鼓勵團隊成員積極交換想法，自然就會冒出有創意的解決方案。

然而，要在群體裏增加多元性並非一件容易的事。當異質團體無法有效溝通或是分散成好幾個小圈圈時，就會產生衝突。一般人都偏好和自己同質性高的人合作，因此，關係緊密的團體會發展出自己的語言和文

化，而且不信任團體之外的人。在數位溝通的環境下，人與人之間的距離伴隨著可能發生不幸事件的陷阱，這使得軟體團隊特別容易出現這些問題：「我們 V.S. 他們」以及獲得不完整的資訊。

所以，我們該如何獲得多元性帶來的好處，避免可能產生的缺點呢？

協作的關鍵在於，培養團隊內的心理安全感和信任感。

當身邊周遭都是我們信賴的成員時，即使他們和我們有所差異，我們也能有更高的信心去承擔風險和做一些嘗試。這是因為當我們彼此信任時，會期望對方提供有助於我們解決挑戰性問題的資訊或觀點，從而創造出團隊合作的機會。當他人要求我們提出回饋時，我們也能克服心理上的弱勢情況。

在團隊成員具有心理安全感的前提下，一般人很容易相信說出個人想法的優點會大於成本。人們的參與感會降低他們對改革的抵抗，當他們的參與頻率越高，提出嶄新想法的可能性也越高。

在軟體開發裡，團隊成員的個性也有關係，為不同個性的成員建立環境信任感一樣重要。每個團隊裡都會有一位願意率先測試每個新的函式庫、框架或工具的同事，會有某個人思考如何使用或探索這個閃亮亮的紅色新玩具，有時就會激發出令人驚豔的結果。某些人的個性傾向於開發新流程、撰寫程式碼的格式風格，或者是建立提交訊息的範本，在我們沒有遵循適當的程序時，從旁提醒我們。你可能會有團隊成員對工作過度承諾、表現超乎預期，或者是思考每一個可能出錯的環節：更新程式、依賴關係、安裝更新檔案、安全風險等等。不管是哪種情況，請考量每個人的差異，記得不要操之過急。

我們可以從兩個面向來提升團隊的多元性：背景和個性。如果團隊能具有良好的動力，而且持續建立彼此的信任感，我們就能成為更成功的程式設計師。

# 編譯過程不需要漫長等待和不可靠性

*Jenn Strater*

不久之前我還在一家剛成立不久的新創公司工作，每天程式庫和開發團隊都不斷地成長，隨著我們加入越來越多的測試，編譯程式所需要的時間也越來越長。後來大概是在第八分鐘左右我開始注意到一件事，這也就是為什麼我會記得這個特定的數字，從八分鐘之後，編譯時間幾乎增加了一倍。起初我覺得這樣很好，程式開始編譯時我還去拿杯咖啡，跟其他團隊的同事聊聊天，但是，幾個月過後，我對此感到厭煩。咖啡我已經喝得夠多了，連每個人在做什麼工作我也都知道了，所以我改在等待編譯完成的期間，確認一下 Twitter 上有什麼更新或是協助團隊裡其他的開發者。然後等我回到工作上，又不得不切換環境。

此外，就跟所有軟體專案一樣，當時程式編譯結果也不可靠，我們有一堆不穩定的測試。聽起來雖然幼稚，但我們第一個解決方案就是關掉失敗的測試（也就是 `@Ignore`）。後來，我們終於抓到重點，相較於在本機執行測試，靠持續整合（continuous integration，簡稱 CI）伺服器推動更改會容易得多。問題是，採取這項策略只是把問題往後推延，如果在持續整合步驟發生測試失敗的情況，我們得花更長的時間除錯。萬一不穩定的測試在一開始就通過了，而且只有在合併後才出現，結果就是卡住整個團隊，直到我們做出決定，判斷這個不穩定的測試是否真的算是一個問題才能繼續進行。

這整個過程令人感到沮喪，於是，我嘗試修復某些有問題的測試。在我的腦海裡，有一項測試特別令人印象深刻，而且只有在執行一整組測試時才會出現，每次我做了一項修改，就必須等超過 15 分鐘的時間才能得到回饋結果。如此漫長的回饋週期實在令人不可置信，而且普遍缺乏相關資料，這表示我追蹤這個臭蟲根本是浪費時間。

然而，不只這家公司有這樣的問題。對一個經常跳槽的人來說，優點之一就是我看過許多不同的團隊工作方式，我本來以為這些問題很正常，直到我在某家公司開始真正地處理這些問題，才發現不是如此。

依循開發者生產力工程的團隊，會透過資料去落實改善開發者體驗的理念，這樣的團隊能改善運行時間漫長而且不可靠的編譯工作。如此一來，不僅團隊更開心，連帶也提高吞吐量，讓業務更高興。

不論團隊使用哪種編譯工具，負責開發人員生產力的人都可以有效衡量編譯效能，並且針對本地編譯和持續整合編譯兩者追蹤個案和迴歸測試。他們花時間分析測試結果，從編譯過程中找出瓶頸。當某個地方出錯，他們會和團隊成員分享測試報告，比較沒有通過測試和通過測試的編譯，從中查出確實的問題是什麼，即使他們無法在自己的機器上重現這個問題。

利用這些資料，他們能落實某些流程最佳化的改善工作，進而減少開發人員所面對的挫折感。這項工作永遠沒有完成的一天，所以只能不斷地進行迭代，維持開發人員的生產力。當然，這不是一件容易的事，但是致力於此的團隊能在最初就避免我所描述的問題發生。