推薦序

毫無疑問，Java 8 的新功能，尤其是 lambda 表達式與 Streams API，使 Java 語言往前跨出一大步。我已經使用 Java 8 好幾年了，也曾在工作坊與透過部落格文章告訴開發者新的功能。我知道雖然 lambda 與串流為 Java 帶來更多泛函（functional）程式設計風格（也可讓我們無縫地發揮平行處理的威力），但開發者必須開始使用這些特殊功能，才能知道它們吸引人的地方—使用這些語法，可讓你更輕鬆地處理某些問題，並帶來更多成效。

身為開發者、貢獻者與作者，我不但熱切地希望其他的開發者知道 Java 語言的演變，也想要讓他們知道這種演變可讓我們開發者更輕鬆—有很多更簡單的問題處理方式，甚至可以處理很困難的問題。我欣賞 Ken 的地方，正是他把焦點放在這裡，他會協助你學習新事物，把重心放在對真實世界的開發者而言有價值的技術上，又不會談論你已經知道或不需要的細節。

我第一次接觸到 Ken 的作品，是他在 JavaOne 提出的“Making Java Groovy”。當時，我的工作團隊正煩惱該如何寫出易讀且實用的測試程式，我們考慮的解決方案之一就是 Groovy。身為長期的 Java 程式員，我不想要為了編寫測試程式而學習全新的語言，特別是我認為自己已經知道如何編寫測試程式了。Ken 為 Java 程式員講解的 Groovy 課程讓我學到許多應瞭解的知識，且不會重複我已經知道的東西。他讓我明白，只要使用正確的教材，其實不需要費心瞭解所有的細節，只要學習我在乎的部分就可以了。於是我立刻購買他的書籍。

Modern Java Recipes 這本新書採取類似的基調—身為資深的開發者，我們不需要像這種語言的新手一樣學習 Java 8 與 9 的所有新功能，也沒有時間做這種事。我們需要一本可讓我們快速使用想用的功能、具備實際範例，並能我們在工作上應用的指南，本書正是這種指南。書中的訣竅展示如何使用 Java 8 與 9 的新功能來處理日常工作的問題，讓我們以更自然的方式熟悉語言的新功能，藉此提升技術。

即使是用過 Java 8 與 9 的人也可以學到一些新知識。說明 Reduction 運算子的章節確實協助我在不需要讓大腦重新學習的情況下瞭解這種泛函風格的程式寫法。本書談到的 Java 9 功能正是可幫助我們開發者的功能，而且它們（尚）未廣為人知。要快速、有效地上手 Java，閱讀這本書是最好的方式。想要提升知識的每位 Java 開發者，也可以在書中找到他們想要的東西。

— Trisha Gee
Jet Brains 的 Java Champion 與
Java Developer
2017 年 7 月

前言

現代 Java

我們很難相信，一種需要考慮整整 20 年的回溯相容性的語言，會有這麼劇烈的變化。當 Java SE 8 在 2014 年 3 月釋出之前[1]，Java 這種成功的伺服器端程式語言已經獲得“21 世紀 COBOL”的美名了。它既穩定、普遍，又持續地專注於性能的改善。它的變化非常緩慢，導致每當有新版本可用時，企業界往往會覺得不需要急著升級。

不過當 Java SE 8 釋出時，一切都改變了。Java SE 8 納入“Lambda 專案”，這是一項重大的創新，在這個佔有世界主導地位的物件導向語言中，納入泛函程式設計的概念。Lambda 表達式、方法參考，以及串流，從根本上改變了這個語言的語法，讓許多開發人員必須嘗試追上腳步。

本書不想要評論這種改變究竟是好是壞，或是否應該採取不同的做法，本書的態度是“這就是我們擁有的東西，也是你用來完成工作的方式。”這就是本書採取訣竅設計的原因，它們闡述的都是你得做的事情，以及 Java 的新功能如何協助你完成它。

也就是說，當你習慣新的程式設計模型之後，會得到許多好處。泛函程式比較簡單，也易於編寫與瞭解。泛函做法有助於不可變性，可讓我們以更簡潔的方式編寫並行（concurrent）程式，而且更有可能成功。在 Java 初創時期，你可以信賴摩爾定律，處

1　是的，Java SE 8 第一版的問世，其實已經是三年前的事。就連我都很難相信。

理器的速度大約每 18 個月就會加倍一次。但近年來，從連手機都有多處理器這件事情，就可以讓我們看到效能改善的速度有多快。

因為 Java 一向對回溯相容性很敏感，許多公司與開發者雖然都已改用 Java SE 8，卻不使用新的語法。即使如此，既然這個平台更強大了，就值得你使用它，更不用說 Oracle 在 2015 年 4 月正式宣布 Java 7 的結束。

經過這幾年，多數的 Java 開發者都在使用 Java 8 JDK 了，現在是深入瞭解它並知道它可為你未來的開發帶來什麼成果的時候了。本書就是為了加速這個過程而設計的。

誰該閱讀本書？

本書的訣竅是假設典型的讀者都已經熟悉 Java SE 8 之前的版本。你不需要是一位專家，也不需要復習舊的觀念，不過，這本書不是 Java 或物件導向的初學指南。如果你曾經在專案中使用 Java，而且已經熟悉標準程式庫，就可以閱讀這本書了。

這本書會討論幾乎所有的 Java SE 8，也有一章把重點放在 Java 9 的新改變上。如果你想要知道這種語言新增的泛函語法如何改變你的程式寫法，本書會透過許多使用案例來完成這項目標。

普遍存在於伺服器端的 Java 擁有豐富的開放原始碼程式庫與工具支援系統。Spring Framework 與 Hibernate 是其中兩種最熱門的開放原始碼框架，它們若不是需要 Java 8，就是很快便會如此。如果你準備在這個生態系統中工作，本書是為你而寫的。

本書架構

這本書是以訣竅來架構的，但是我們很難在不談到其他功能的情況下，在訣竅中只單獨討論 lambda 表達式、方法參考與串流。事實上，前七章都在討論彼此相關的概念，你不需要按照特定的順序來閱讀它們。

本書的章節架構如下：

- **第一章「基礎知識」**討論 *lambda* **表達式**、**方法參考**的基礎知識，並採用介面的新功能：*default* **方法**與**靜態方法**。它也定義了泛函介面這個名詞，並解釋為何它是瞭解 lambda 表達式的關鍵。
- **第二章**「*Java.util.function* **套件**」將說明 Java 8 新增的 `java.util.function` 套件。這個套件內的介面分為四種（consumer、supplier、predicate 與 function），供標準程式庫的其他部分使用。
- **第三章**「**串流**」會加入**串流**的概念，以及它們如何表示可用來轉換與篩選資料（而非迭代處理）的抽象。本章的訣竅會展示串流的“map”、“filter”與“reduce”概念。它們最終引出第九章談到的平行與並行概念。
- **第四章**「**比較器與收集器**」談論串流資料排序，以及將它們轉換成集合。本書也會討論分割與分群，它們可將一般被視為資料庫操作的事物轉換成簡單的程式庫呼叫。
- **第五章**「**串流、*Lambda* 與方法參考的問題**」是混合的一章，因為你現在已經知道如何使用 lambda、方法參考與串流，所以可以瞭解如何結合它們來處理有趣的問題。本章也會討論令人討厭的例外處理主題：惰性、延遲執行與 closure 組合等概念。
- **第六章**「*Optional* **型態**」討論這個語言較具爭議性的新功能— `Optional` 型態。本章的訣竅會說明設計者希望你如何使用這種新型態，以及你該如何用它們來創建實例與從中取值。這一章也會回顧對於 `Optionals` 的 map 與 flat-map 操作的泛函概念，以及它們與對串流執行同一種操作的差異。
- **第七章**「**檔案** *I/O*」會切換到輸入 / 輸出串流的實際主題（與泛函串流對照），以及在處理檔案與目錄時，展現泛函新概念的標準程式庫新功能。
- **第八章**「*java.time* **套件**」將展示新的 Date-Time API 的基本知識，以及它們（終於）如何取代舊的 Date 與 Calendar 類別。新的 API 是以 Joda-Time 程式庫為基礎，它受到許多開發者多年使用經驗的支持，而且是用 java.time 套件來改寫的。坦白說，就算 Java 8 只新增這種功能，也值得你升級。
- **第九章**「**平行與並行**」會解決串流模型的隱含承諾之一：你可以使用一個方法呼叫式來將一個連續的串流改成平行，以充分利用機器上所有可用的處理器。並行是個龐大的主題，本章只討論當本益比划算時，可讓你輕鬆實驗與使用的 Java 程式庫新功能。

- **第十章**「*Java 9 的新增功能*」討論 Java 9 新增的許多改變，它預計在 2017 年 9 月 21 日發布。光是 Jigsaw 的細節就需要用整本書來說明，不過它的基本知識很簡單，本章將會說明。其他的訣竅會討論介面中的私用方法、被加入串流的新方法、收集器與 `Optional`，以及如何建立日期串流[2]。

- **附錄 A**「*泛型與 Java 8*」將討論 Java 的泛型功能。雖然泛型這項技術早在 1.5 版本就加入了，但多數的開發者只學到可讓程式動作所需的最少知識而已。你只要稍微看一下 Java 8 與 9 的 Javadocs，就知道這樣的日子已經過去了。附錄的目的，是告訴你如何閱讀與解讀 API，以瞭解更複雜的方法簽章。

這些章節以及訣竅本身，不需要以任何特定的順序來閱讀。它們是互補的，每一個訣竅的結尾都有其他訣竅的參考，但你可以從任何地方開始看。用章節來劃分是為了將相似的訣竅放在一起，但希望你在遇到任何問題時，可在訣竅間互相參考以解決問題。

本書編排方式

本書使用下列編排規則：

斜體字（*Italic*）或楷體字
: 代表新的術語、URL、電子郵件地址、檔案名稱及副檔名。

定寬字（`Constant width`）
: 代表程式，也在文章中代表程式元素，例如變數或函式名稱、資料庫、資料類型、環境變數、陳述式與關鍵字。

定寬粗體字（`Constant width bold`）
: 代表應由使用者逐字輸入的指令或其他文字。

定寬斜體字（`Constant width italic`）
: 代表應換成使用者提供的值，或依上下文而決定的值。

2 是的，我也希望能在第九章寫 Java 9，不過重新排列章節來滿足這種不重要的對稱性應該不是正確的做法。以這個註腳來表明我的心意就夠了。

第一章

基礎知識

Java 8 最大的改變是在這個語言中加入泛函（functional）程式設計的概念。具體來說，這種語言加入 lambda 表達式、方法參考與串流（stream）。

如果你還沒有用過泛函功能，或許會對目前的程式與 Java 舊版本之間的差異程度感到驚訝。Java 8 的改變是這個語言有史以來最大的變化。從許多方面來看，你會覺得自己學習的是全新的語言。

所以，你的問題會變成：為什麼要這樣做？為什麼要對一個已經 20 歲，而且還要規劃回溯相容性的語言做這麼劇烈的改變？為什麼要對一個從各種資訊來看，都已經相當成功的語言做這麼大幅度的修改？為什麼一個已經是有史以來最成功的物件導向語言，在多年之後要改為泛函模式？

答案是，軟體開發世界已經改變了，所以這個希望未來繼續成功的語言也必須改變。回到 90 年代中期，當 Java 還是嶄新的語言時，摩爾定律[1] 仍然是完全有效的。你要做的事情，就是等待幾年，讓電腦的速度加倍。

現今的硬體已經不再依賴增加晶片密度來提升速度了。就連多數的手機都有多核心，代表你在編寫軟體時，應該要準備讓它可在多處理器環境中運行。強調“純”函式（收到相同的輸入會回傳相同的結果，沒有副作用）與不可變性的泛函做法，可簡化平行環境

1 由 Fairchild Semiconductor 與 Intel 的共同創辦人之一：Gordon Moore 提出，他發現可塞入積體電路的電晶體數量大概每 18 個月就會增加大約兩倍，因而提出這個定律。詳情請見 Wikipedia 的 Moore's law 項目（*https://en.wikipedia.org/wiki/Moore%27s_law*）。

的程式設計。當你的程式沒有任何共享、可變的狀態，而且可以分解為簡單的函式集合，就可更輕鬆地瞭解與預測它的行為。

但是，這本書談的不是 Haskell、Erlang、Frege 或任何其他泛函程式設計語言，它談的是 Java，以及對於一種基本上依然是物件導向語言的語言中加入泛函概念的改變。

Java 現在已經支援 lambda 表達式，它本質上是一種被視為一級物件的方法。這種語言也有方法參考，可讓你在準備接收 lambda 表達式的地方使用既有的方法。為了利用 lambda 表達式與方法參考，這個語言也加入一種串流模型，它可產生元素並透過轉換管道與篩選器來傳遞它們，而不需要修改原始的來源。

本章的訣竅將說明 lambda 表達式、方法參考與泛函介面的基本語法，以及這個語言對於介面的靜態與 default 方法的新支援。第三章將會詳細說明串流。

1.1 Lambda 表達式

問題

你想要在程式中使用 lambda 表達式。

解決方案

使用其中一種 lambda 表達式語法，並將結果指派給泛函介面型態的參考。

說明

泛函介面是擁有單一抽象方法（single abstract method，SAM）的介面。類別是藉由實作介面裡面的所有方法來實作任何介面的。這可以用頂級（top-level）類別、內部類別，甚至匿名內部類別都可以做這件事。

例如，考慮從 Java 1.0 就有的 `Runnable` 介面。它有單一抽象方法，稱為 `run`，這個方法不接收引數，且回傳 `void`。`Thread` 類別建構式會以引數接收 `Runnable`，範例 1-1 將示範一個匿名內部類別實作。

範例 1-1. Runnable 匿名內部類別實作

```
public class RunnableDemo {
    public static void main(String[] args) {
        new Thread(new Runnable() {  ❶
            @Override
            public void run() {
                System.out.println(
                    "inside runnable using an anonymous inner class");
            }
        }).start();
    }
}
```

❶ 匿名內部類別

匿名內部類別語法是由單字 `new` 以及後面的 `Runnable` 介面名稱與括號組成的，意味著你定義的是一個實作了該介面，但沒有明確名稱的類別。接著大括號（`{}`）內的程式會覆寫 `run` 方法，它的工作只是將一個字串印到主控台上。

範例 1-2 的程式是使用 lambda 表達式的同一個範例。

範例 1-2. 在 Thread 建構式內使用 lambda 表達式

```
new Thread(() -> System.out.println(
    "inside Thread constructor using lambda")).start();
```

這個語法使用箭頭將引數（因為這裡沒有引數，所以只有一對空括號）與內文隔開。這個範例的內文只有一行，所以不需要大括號。這就是 lambda 表達式。表達式算出來的值都會被自動回傳。在此範例中，因為 `println` 會回傳 `void`，所以表達式也會回傳 `void`，這符合 `run` 方法的回傳型態。

lambda 表達式的引數型態與回傳型態必須符合介面唯一的抽象方法的簽章。這稱為與方法簽章**相容**。因而，lambda 表達式就是介面方法的實作，你也可以將它指派給該介面型態的參考。

範例 1-3 是將 lambda 指派給一個變數的情形。

範例 1-3. 將 lambda 表達式指派給變數

```
Runnable r = () -> System.out.println(
    "lambda expression implementing the run method");
new Thread(r).start();
```

Java 程式庫沒有名為 Lambda 的類別。你只能將 lambda 表達式指派給泛函介面參考。

將 lambda 指派給泛函介面代表：lambda 是介面唯一的抽象方法的實作。你可以將 lambda 當成介面的匿名內部類別之實作內容。這就是 lambda 必須與抽象方法相容的原因；它的引數型態與回傳型態必須匹配方法的簽章。但是，應該注意的是實作方法的名稱並不重要，它不會在 lambda 表達式語法中的任何地方出現。

因為 run 方法不接收引數，且回傳 void，所以這個範例特別簡單。我們來探討泛函介面 java.io.FilenameFilter，它從第 1.0 版開始也屬於 Java 標準程式庫。File.list 方法會將 FilenameFilter 的實例當成引數，來將回傳的檔案限制成只含有滿足這個方法的檔案。

Javadoc 指出，FilenameFilter 類別有單一抽象方法 accept，它的簽章是：

```
boolean accept(File dir, String name)
```

File 引數是準備從中尋找檔案的目錄，String name 是檔案的名稱。

範例 1-4 使用一個匿名內部類別實作 FilenameFilter，它只會回傳 Java 原始檔案。

範例 1-4. 使用匿名內部類別實作 FilenameFilter

```
File directory = new File("./src/main/java");

String[] names = directory.list(new FilenameFilter() {  ❶
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(".java");
    }
});
System.out.println(Arrays.asList(names));
```

❶ 匿名內部類別

在這個範例中，當檔名的結尾是 *.java* 時，accept 方法會回傳 true，否則 false。

範例 1-5 是 lambda 表達式版本。

範例 1-5. 以 lambda 表達式實作 FilenameFilter

```
File directory = new File("./src/main/java");

String[] names = directory.list((dir, name) -> name.endsWith(".java"));  ❶
    System.out.println(Arrays.asList(names));
}
```

❶ Lambda 表達式

這段程式簡單多了。這次在括號裡面有引數，但沒有宣告型態。在編譯階段，編譯器知道 list 方法會接收 FilenameFilter 型態的引數，因而知道它唯一的抽象方法（accept）的簽章，因此也知道 accept 的引數是一個 File 與一個 String，所以相容的 lambda 表達式引數必須匹配這些型態。accept 的回傳型態是布林，所以箭頭右邊的表達式也必須回傳布林。

你也可以在程式中指定資料型態，如範例 1-6 所示。

範例 1-6. 明確使用資料型態的 lambda 表達式

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) ->  ❶
    name.endsWith(".java"));
```

❶ 明確指定資料型態

最後，如果 lambda 的實作需要多行程式，你必須使用大括號以及一個 return 陳述式，如範例 1-7 所示。

範例 1-7. lambda 區塊

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) -> {  ❶
    return name.endsWith(".java");
});
System.out.println(Arrays.asList(names));
```

❶ 區塊語法

這稱為 lambda 區塊。在這個範例中，內文仍然只有一行，但大括號可讓你加入多行陳述式。現在它需要使用 return 關鍵字。

lambda 表達式永遠不會單獨存在。表達式永遠都有一個 *context*（指上下文、前後關係，爾後直接使用 context），指出這個表達式會被指派給哪個泛函介面。你可以將 lambda 當成方法的引數、方法的回傳型態，或將它指派給一個參考，在這些情況下，賦值的型態必須是泛函介面。

1.2 方法參考

問題

你想要使用方法參考來存取既有的方法，並將它視為 lambda 表達式。

解決方案

使用雙冒號語法將實例參考或類別名稱與方法隔開。

說明

如果 lambda 表達式實質上將方法視為物件，則方法參考是將既有的方法視為 lambda。

例如，Iterable 的 forEach 方法會接收 Consumer 引數。範例 1-8 為展示 Consumer 可用 lambda 表達式或方法參考實作。

範例 1-8. 使用方法參考存取 println

```
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(x -> System.out.println(x));      ❶

Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(System.out::println);             ❷

Consumer<Integer> printer = System.out::println;   ❸
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(printer);
```

❶ 使用 lambda 表達式

❷ 使用方法參考

❸ 將方法參考指派給泛函介面

雙冒號語法可提供 System.out 實例的 println 方法之參考，也就是 PrintStream 型態的參考。方法參考的結尾不用放上括號。在這個範例中，串流的每一個元素都會被印到標準輸出上[2]。

如果你寫的 lambda 表達式是由一行呼叫某個方法的程式所構成，可考慮改用等效的方法參考。

方法參考有一些（小）優點是 lambda 語法沒有的。首先，它往往比較短，其次，它通常有該方法的類別名稱，可讓程式更容易讓人理解。

方法參考也可以用於靜態方法，如範例 1-9 所示。

範例 1-9. 使用靜態方法的方法參考

```
Stream.generate(Math::random)          ❶
        .limit(10)
        .forEach(System.out::println); ❷
```

❶ 靜態方法

❷ 實例方法

Stream 的 generate 方法會接收 Supplier 引數，它是個泛函介面，介面唯一的抽象方法不接收引數，並且會產生一個結果。Math 類別的 random 方法與該簽章相容，因為它也不接收引數，並且可產生一個介於 0 與 1 間均勻分布的偽亂數 double。方法參考 Math::random 代表該方法是 Supplier 介面的實作。

因為 Stream.generate 會產生一個無限的串流，我們用 limit 方法來確保它只產生 10 個值，接著使用 System.out::println 方法參考當成 Consumer 的實作，將值印至標準輸出。

2 我們很難在不討論串流的情況下討論 lambda 或方法參考，稍後我會專門用一章來討論串流。你只要先知道，串流會依序產生一系列的元素，且不會將它們儲存在任何地方，也不會修改原始來源。

語法

方法參考的語法有三種形式，其中一種可能會造成誤解：

`object::instanceMethod`

使用你提供的物件之參考，來參考一個實例方法，例如 `System.out::println`

`Class::staticMethod`

參考靜態方法，例如 `Math::max`

`Class::instanceMethod`

呼叫 context 提供的物件參考裡面的實例方法，例如 `String::length`

最後一個案例是會令人困惑的一種，因為身為 Java 開發者，我們已經習慣看到藉由類別名稱來呼叫靜態方法。之前提過 lambda 表達式與方法參考永遠不會單獨存在，它一定有個 context。在物件參考的案例中，context 會提供引數給方法。在列印的範例中，等效的 lambda 表達式是（如範例 1-8 的 context 所示）：

```
// 相當於 System.out::println
x -> System.out.println(x)
```

context 提供 `x` 的值，它會被當成方法引數使用。

這個情況類似靜態的 `max` 方法：

```
// 相當於 Math::max
(x,y) -> Math.max(x,y)
```

現在 context 需要提供兩個引數，lambda 會回傳較大的那一個。

“透過類別名稱來指定實例方法”這種語法要用不同的方式來解釋。它的等效 lambda 是：

```
// 相當於 String::length
x -> x.length()
```

這一次， context 提供的 x 會被當成方法的目標來使用，而非當成引數。

如果你要參考的是透過類別名稱來接收多個引數的方法，則 context 提供的第一個元素會變成目標，其餘的元素會變成方法的引數。

見範例 1-10。

範例 1-10. 用類別參考呼叫有多個引數的實例方法

```
List<String> strings =
    Arrays.asList("this", "is", "a", "list", "of", "strings");
List<String> sorted = strings.stream()
        .sorted((s1, s2) -> s1.compareTo(s2))  ❶
        .collect(Collectors.toList());

List<String> sorted = strings.stream()
        .sorted(String::compareTo)             ❶
        .collect(Collectors.toList());
```

❶ 方法參考與等效的 lambda

`Stream` 的 `sorted` 方法會接收 `Comparator<T>` 引數，它唯一的抽象方法是 `int compare(String other)`。`sorted` 方法會提供各對字串給比較器（comparator），並根據回傳的整數符號來排序它們。在這裡，context 是每一對字串。使用類別名稱 `String` 的方法參考會對第一個元素（lambda 表達式的 `s1`）呼叫 `compareTo` 方法，並將第二個元素 `s2` 當成方法的引數。

在處理串流時，當你要處理一系列的輸入時，經常會在方法參考內使用類別名稱來存取實例方法。範例 1-11 展示如何對串流的每一個 `String` 呼叫 `length` 方法。

範例 1-11. 使用方法參考來對 String 呼叫 length 方法

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(String::length)              ❶
        .forEach(System.out::println);    ❷
```

❶ 使用類別名稱的實例方法

❷ 使用物件參考的實例方法

這個範例會呼叫 length 方法來將每一個字串轉換成一個整數，並印出每一個結果。

方法參考基本上是 lambda 的縮寫語法。lambda 表達式比較常見，因為每一個方法參考都有一個等效的 lambda 表達式，但反過來並非如此。範例 1-12 使用範例 1-11 中的方法參考的等效 lambda。

範例 1-12. 方法參考的等效 lambda 表達式

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(s -> s.length())
        .forEach(x -> System.out.println(x));
```

如同所有 lambda 表達式，context 至關重要。如果你不確定做法，也可以在方法參考的左邊使用 this 或 super。

參見

你也可以使用方法參考語法呼叫建構式。建構式參考是訣竅 1.3 的主題。第二章將會說明泛函介面套件，包括這個訣竅談到的 Supplier 介面。

1.3 建構式參考

問題

你想要在串流管道中，使用方法參考來實例化一個物件。

解決方案

在方法參考中使用 new 關鍵字。

說明

當人們談到 Java 8 新增的語法時，指的都是 lambda 表達式、方法參考與串流。例如，假設你有一份名單，相要將它們轉換成人名串列。範例 1-13 的程式段落是其中一種做法。

範例 1-13. 將一份名單轉換成名稱串列

```
List<String> names = people.stream()
    .map(person -> person.getName())  ❶
    .collect(Collectors.toList());

// 或者

List<String> names = people.stream()
    .map(Person::getName)             ❷
    .collect(Collectors.toList());
```

❶ Lambda 表達式

❷ 方法參考

如果你想要採取其他的方式呢？如果你有一個字串串列，想要用它來建立一個 Person 參考的串列？此時，你可以使用方法參考，但是這次要使用關鍵字 new。這種語法稱為**建構式參考**（*constructor reference*）。

為了展示它的用法，我們從 Person 類別看起，它是你想像得到最簡單的 Plain Old Java Object（POJO）。範例 1-14 中，它的工作只是包裝一個簡單的字串屬性 name。

範例 1-14. Person 類別

```
public class Person {
    private String name;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    // getters 與 setters ...

    // equals、hashCode、與 toString 方法 ...
}
```

當你取得一個字串集合之後，可以使用 lambda 表達式或建構式參考將每一個字串對映到 Person，如範例 1-15 所示。

範例 1-15. 將字串轉換成 Person 實例

```
List<String> names =
    Arrays.asList("Grace Hopper", "Barbara Liskov", "Ada Lovelace",
        "Karen Sparck Jones");

List<Person> people = names.stream()
    .map(name -> new Person(name)) ❶
    .collect(Collectors.toList());

// 或者

List<Person> people = names.stream()
    .map(Person::new)              ❷
    .collect(Collectors.toList());
```

❶ 使用 lambda 表達式呼叫建構式

❷ 使用建構式參考實例化 Person

語法 Person::new 代表的是 Person 類別的建構式。如同所有的 lambda 表達式，context 會決定該執行哪個建構式。因為 context 提供字串，所以它執行的是有一個引數的 String 建構式。

複製建構式

複製建構式會接收 Person 引數，並回傳一個擁有相同屬性的新 Person，如範例 1-16 所示。

範例 1-16. Person 的複製建構式

```
public Person(Person p) {
    this.name = p.name;
}
```

當你想要將串流程式與原始實例分開，這種方法相當方便。例如，如果你已經有一個人名串列，將這個串列轉換成串流，再轉換回串列時，參考會是相同的（見範例 1-17）。

範例 1-17. 將串列轉換成串流，並轉換回去

```
Person before = new Person("Grace Hopper");

List<Person> people = Stream.of(before)
    .collect(Collectors.toList());
Person after = people.get(0);

assertTrue(before == after);                          ❶

before.setName("Grace Murray Hopper");                ❷
assertEquals("Grace Murray Hopper", after.getName()); ❸
```

❶ 同一個物件

❷ 使用 before 參考改變名稱

❸ 在 after 參考中，名稱已經改變了

你可以使用複製建構式打斷這個關係，如範例 1-18 所示。

範例 1-18. 使用複製建構式

```
people = Stream.of(before)
    .map(Person::new)             ❶
    .collect(Collectors.toList());
after = people.get(0);
assertFalse(before == after);     ❷
assertEquals(before, after);      ❸

before.setName("Rear Admiral Dr.Grace Murray Hopper");
assertFalse(before.equals(after));
```

❶ 使用複製建構式

❷ 不同的物件

❸ 但是是等效的

這一次，當我們呼叫 map 方法時，context 是 Person 實例的串流。因此 Person::new 語法會呼叫接收 Person 的建構式，並回傳新的、等效的實例，並且打斷 *before* 參考與 *after* 參考之間的連結[3]。

Varargs 建構式

接下來探討有一個 varargs 建構式被加入 Person POJO，如範例 1-19 所示。

範例 1-19. 接收可變長度引數 String 串列的 Person 建構式

```
public Person(String... names) {
    this.name = Arrays.stream(names)
                      .collect(Collectors.joining(" "));
}
```

這個建構式可接收零或多個字串引數，並用一個空格作為分隔符號，來將它們串在一起。

該怎麼呼叫這個建構式？任何傳遞以逗號分隔零或多個字串引數的使用方都可呼叫它。其中一種做法是使用 String 的 split 方法來接收一個分隔符號，並回傳一個 String 陣列：

```
String[] split(String delimiter)
```

因此，範例 1-20 的程式會將串列內的每一個字串拆為個別的單字，並呼叫 varargs 建構式。

範例 1-20. 使用 varargs 建構式

```
names.stream()                      ❶
    .map(name -> name.split(" "))   ❷
    .map(Person::new)               ❸
    .collect(Collectors.toList());  ❹
```

❶ 建立字串的串流

❷ 對應至字串陣列的串流

❸ 對應至 Person 的串流

❹ 收集至 Person 的串列

3 我將 Admiral Hopper 當成物件沒有不尊重的意思。毫無疑問，她依然可以輕鬆地打敗我。她在 1992 年去世。

這一次，含有 Person::new 建構式參考的 map 方法的 context 是字串陣列的串流，所以 varargs 建構式會被呼叫。如果你在建構式加入一個簡單的列印陳述式：

```
System.out.println("Varargs ctor, names=" + Arrays.toList(names));
```

結果會是：

```
Varargs ctor, names=[Grace, Hopper]
Varargs ctor, names=[Barbara, Liskov]
Varargs ctor, names=[Ada, Lovelace]
Varargs ctor, names=[Karen, Sparck, Jones]
```

陣列

建構式參考也可以與陣列一起使用。如果你想要一個 Person 實例的陣列，Person[]，而非串列，可使用 Stream 的 toArray 方法，它的簽章是：

```
<A> A[] toArray(IntFunction<A[]> generator)
```

這個方法使用 A 來代表儲存串流元素的回傳陣列之泛型型態，它是用你提供的產生器函式（generator function）建立的。最酷的是，你也可以使用建構式參考，如範例 1-21 所示。

範例 1-21. 建立 Person 參考的陣列

```
Person[] people = names.stream()
    .map(Person::new)          ❶
    .toArray(Person[]::new);   ❷
```

❶ Person 的建構式參考

❷ Person 陣列的建構式參考

toArray 方法引數會建立一個適當大小的 Person 參考之陣列，並對它填入已被實例化的 Person 實例。

建構式參考只是透過另一種名稱來使用的方法參考，以單字 new 來呼叫建構式。一如往常，它的建構式是由 context 決定的。當你要處理串流時，這項技術可提供許多彈性。