序

Java 的模組化是什麼？對有些人而言，它是開發的原則：編寫介面，並隱藏實作的細節。這是**封裝派**的說法。對一些其他人來說，它代表重度依賴類別載入器來提供動態執行環境，這是**隔離派**的說法。對其他人來說，它是關於成品、存放區與工具的使用，這是**配置派**的說法。對我來說，這些觀點都是對的，但它們就像管中窺天，談的都是模糊片段的概念。如果開發者知道某些程式碼只供內部使用，何不像隱藏類別或欄位那樣輕鬆地隱藏套件？如果程式碼只能在它的依賴項目存在的情況下被編譯以及執行，何不讓這些依賴項目流暢地通過編譯、包裝、安裝與執行等過程？如果工具只能在具備原始的自我描述成品（pristine self-describing artifacts）的情況下運作，該如何讓每一個人都可以重複使用只是一般的 JAR 檔案的舊程式庫？

Java 9 藉由加入模組，並將模組當成 Java 平台的一級特徵，來提供一個連貫的模組化方案。模組是一組為了重複使用而設計的套件。這個簡單的概念，對程式碼的開發、部署與執行方式產生驚人且重大的影響。為了促進與控制程式的重複使用，Java 有許多存在已久的機制，包括介面、操作控制、JAR 檔案、類別載入器、動態連結等等，它們都可在套件被放入模組之後更良好地運作。

首先，模組採取與其他機制不同的方式來闡明程式的架構。許多開發者都很驚訝他們的程式架構不如原本想像的良好。例如，涵蓋多個 JAR 檔案的基礎程式，可能在多個 JAR 檔案之間，存在類別的循環結構，但是不同模組的類別是不能有循環結構的。人們投入程式碼庫模組化的其中一個動機，就是他們認識到，完成模組化之後，就不會有以前的循環依賴關係造成的泥漿球（ball of mud，指沒有清晰結構的系統）。使用模組來開發，也可產生服務導向程式設計，進一步減少耦合，並增加抽象。

其次，模組可讓你產生對程式碼的責任感，這是其他的機制無法提供的。當開發者匯出模組內的套件時，相當於承諾提供一個穩定的 API，甚至模組的名稱也是 API 的一部分。當開發者將太多功能綁成單一模組時，那個模組會拉入許多與任何單一工作無關的依賴關係；雖然模組的內部已被隱藏起來，但任何重複使用它的人都會發現它的蔓延性質。使用模組來開發，可以鼓勵所有的開發者考慮程式的穩定性與內聚性。

多數人都知道拉桌布戲法，也就是快速拉開桌布，讓盤子與杯子停在原處。對已經用過 Java 9 的人而言，設計一個模組系統，並將它插入從 1990 年以來被開發的上百萬個類別底下的 Java Virtual Machine 之中，就像是反向執行那個戲法。將 JDK 模組化會讓這個戲法失敗，因為有一些著名的程式庫之所以能夠施展它的魔法，就是因為它們忽略模組系統施加在 JDK 模組的封裝。這種 Java 9 的設計造成的張力沒有簡單的學術解答。最後，社群的回饋終於讓模組系統提供各種工具給開發者，如此一來，被模組化的平台程式碼就可得到強力的封裝，被模組化的應用程式碼也可以得到“足夠強力”的封裝。隨著時間的推移，我們認為模組化 JDK 這個大膽的選擇，可讓所有的程式碼更加可靠。

當所有人都使用模組系統時，它的效果最好。今天有愈多開發者創造模組，明天就有愈多開發者創造模組。但是，**尚未**創造模組的開發者該怎麼辦？我們毫不誇張地說，Java 9 是平等看待非模組程式與模組化程式的。唯有基礎程式的作者應該將它模組化，但是在這件事發生之前，模組系統必須設法讓模組內的程式碼與模組外的程式碼接觸。所以本書會詳加說明自動模組的設計。

Sander 與 Paul 是 Java 業界專家，也是值得信賴的 Java 9 生態系統導遊。他們身處 Java 9 開發前線，也是遷移熱門的開放原始碼程式庫的先鋒。對想知道 Java 模組化的核心原則與最佳做法的人而言，包括：想要建立可維護組件的應用程式開發者、尋求遷移與反射的建議的程式庫開發者，與想要探索模組系統進階功能的框架開發者，*Java 9 Modularity* 是獻給他們的手冊。我希望這本書可協助你建立一個經得起時間考驗的 Java 程式架構。

— Alex Buckley
Java Platform Group，Oracle
Santa Clara，2017 年 7 月

前言

Java 9 在這個平台中加入模組系統。這是很重大的一步，開啟了 Java 平台模組化軟體開發的新時代。我們為這種改變感到開心，希望你在閱讀這本書之後，也有相同的感受。你將會充分利用模組系統，甚至在你意識到這件事之前。

誰應閱讀本書？

本書的對象是想要改善應用程式的設計與架構的 Java 開發者。Java 模組系統可改善我們設計與組建 Java 應用程式的方式。就算你未以正確的方式來使用模組，瞭解 JDK 本身的模組化也是很重要的開始。我們希望你在本書的第一部分瞭解模組之後，也能夠欣賞隨後的遷移章節。將既有的程式碼遷移到 Java 9 與模組系統會逐漸變成常見的工作。

本書的目的不是做一般性的 Java 介紹。我們假設你已經在團隊中寫過比較大型的 Java 應用程式了，這正是愈來愈需要模組化的領域。身為一位資深的 Java 開發者，你已經知道類別路徑造成的問題，這一點可以協助你欣賞模組系統與它的功能。

Java 9 除了加入模組系統之外也有其他的改變。但是，本書把焦點放在模組系統及相關的功能上。在適當的情況下，我們會在討論模組系統時，說明其他的 Java 9 功能。

為什麼要寫這本書

我們從 Java 早期就開始使用 Java 了，當時 applet 仍然是很熱門的技術。在那幾年，我們也用了許多其他的平台與語言，但 Java 仍然是我們的主要工具。若要建立可維護的軟體，模組化是很重要的原則。多年來花費許多精力建立模組化軟體之後，開發模組化應用程式已經成為我們的熱情所在。在 Java 平台本身不支援的情況下，我們大量

地採用 OSGi 之類的技術來進行模組化。我們也研究過不屬於 Java 世界的工具，例如 JavaScript 的模組系統。當我們確定 Java 9 即將具備期待已久的模組系統之後，我們就決定不但要自行採用這項功能，也要協助業界的其他開發者。

或許你在過去十年間曾經聽過 Project Jigsaw。多年來，Project Jigsaw 已經建立許多可行的 Java 模組系統的實作原型。在這幾年，Java 不斷地準備納入模組系統又排除這項計畫，原本 Java 7 與 8 都準備納入 Project Jigsaw 的成果。

Java 9 終於將這項長時間的實驗納入官方模組系統的實作了。多年來，各種模組系統原型的範圍與功能已經發生許多改變。就算你曾經密切地關注這個過程，也很難發現 Java 9 的模組系統最終繼承了什麼東西。我們希望透過這本書來詳盡說明模組系統，更重要的是，讓你瞭解它可以為你的應用程式的設計與架構帶來什麼幫助。

導覽本書

這本書有三個部分：

1. 介紹 Java 模組系統
2. 遷移
3. 模組化開發工具

第一個部分會教你如何使用模組系統。我們從 JDK 本身的模組化談起，接著說明如何建立你自己的模組。接下來會討論服務，它可實現模組的解耦。第一個部分的結尾會討論模組化模式，以及如何使用模組來將可維護性與擴展性最大化。

本書的第二部分會討論遷移。你應該已經有一些 Java 程式碼了，或許也會使用不是為模組系統設計的 Java 程式庫。在本書的這個部分，你將會學習如何將既有的程式遷移到模組，以及使用還不是模組的既有程式庫。如果你是程式庫的作者或維護者，這個部分會用一章特別講解在程式庫中支援模組。

本書的第三部分，也是最後一個部分，將會討論工具。在這個部分，你會學到目前 IDE 與組建工具的狀態。你也會學習如何測試模組，因為就（單位）測試而言，模組帶來一些新的挑戰與機會。最後，你也會學到**連結**，這是模組系統另一個令人興奮的功能。它可讓你建立高度最佳化的自訂執行期映像，藉由模組來改變發表 Java 應用程式的方式。

本書的設計，是讓讀者從頭開始讀起，直到結束的，但我們也理解，並非每位讀者都會採取這種方式。我們建議你至少詳細閱讀前四章，如此一來，你將會具備基本知識，可

善用本書其他的部分。如果你真的沒有時間，而且有程式碼需要遷移，可以在那之後跳到第二部分，當你完成工作之後，再回到較進階的章節。

使用範例程式

這本書有許多範例程式。你可以在 GitHub 的 *https://github.com/java9-modularity/examples* 取得所有範例程式。這個存放區是以章節來安排範例程式的。在這本書中，我們用以下的方式來指明特定的範例程式：➥ *chapter3/helloworld*，這代表你可以在 *https://github.com/java9-modularity/examples/chapter3/helloworld* 找到這個範例。

我們建議你在閱讀這本書時下載程式碼，因為使用程式編輯器比較容易閱讀較長的程式片段。我們也建議你試著執行一下程式，例如，重現書中談到的錯誤。在實作中學習比只是閱讀文字有效。

本書編排方式

本書使用下列的編排規則：

斜體字（*Italic*）

代表新的術語、URL、電子郵件地址、檔案名稱及副檔名。中文以楷體表示。

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`**）

代表指令，或其他應由使用者逐字輸入的文字。

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。

這個圖示代表一般注意事項。

第一章

模組化很重要

你是不是曾經困惑地搔著頭，自問“為什麼有這段程式？它與這個巨大的基礎程式的別處有什麼關係？我該從哪裡開始看起？”或者，你是否在看了許多與應用程式碼綁在一起的 Java Archives（JARs）之後眼神呆滯？我們當然有這些經驗。

建構大型程式碼的藝術是被低估的一種。這既不是新的問題，也不是 Java 專屬的問題。但是，Java 是主流語言之一，許多大型的應用程式都是用它來建構的，而且通常會使用 Java 生態系統的許多程式庫。在這種情況下，系統的成長幅度，可能會超越我們能夠理解和有效開發的範圍。根據經驗，長期而言，缺乏健全的結構會讓你付出昂貴的代價。

模組化是用來管理與降低這種複雜性的技術之一。Java 9 加入新的模組系統，可讓我們更輕鬆地進行模組化。模組化的開發，是建構在 Java 本來就具備的抽象之上。就某種意義而言，它是將既有的大型 Java 程式最佳開發方法提升為 Java 語言的一部分。

Java 模組系統會對 Java 的開發造成深遠的影響。它代表將模組化變成整個 Java 平台的一級公民，這是一種根本的轉變。模組化是從底層開始進行的，包括改變語言、Java Virtual Machine（JVM）與標準程式庫。雖然這代表一個巨大的工程，但它不像（舉例）在 Java 8 中加入串流與 lambda 那麼華麗。lambda 這類的功能與 Java 模組系統之間還有其他基本的區別。模組系統與整個應用程式的大型結構有關。將內部類別轉換成 lambda 只是在一個類別的範圍內，相當小型且區域性的改變。將一個應用程式模組化，會影響設計、編譯、封裝、部署，等等。顯然，它並非只是另一個語言功能。

在每個 Java 新版本問世時，我們通常會立刻一頭栽入使用新功能。為了充分利用模組系統，我們應該先後退一步，把焦點放在瞭解模組是什麼。更重要的是，為何我們應該關心它。

什麼是模組化？

到目前為止，我們已經知道模組化的目標了（管理與減少複雜度），但還不知道模組化需要什麼因素。在核心，**模組化**是將一個系統分解成獨立但互相連結的模組。**模組**是可識別的成品（artifacts），裡面有程式碼，以及描述模組以及它與其他模組之間的關係的詮釋資料（metadata）。理想情況下，這些成品從編譯期一路到執行期都是可識別的。所以，應用程式是由許多一起工作的模組組成的。

因此，模組會聚集彼此相關的程式碼，但它做的事情不止於此。模組必須遵守三個核心原則：

強力的封裝

模組必須能夠隱藏部分的程式碼，不讓其他模組看到。如此一來，我們就可以明確地區分“可公開使用”與“應視為內部實作細節”的程式。這可防止模組之間意外或沒必要的耦合：你根本無法使用被封裝的東西。因此，你可以自由地更改被封裝的程式，不至於影響模組的使用者。

具有良好的定義的介面

封裝很好，但如果模組是要一起工作的，就不是所有東西都可以封裝。根據定義，未封裝的程式是模組的公開 API 的一部分。因為其他的模組可以使用公開的程式碼，所以你必須非常謹慎地管理它。當你對未封裝的程式做破壞性的改變時，也會損壞依賴它的其他模組。因此，模組應該公開定義良好且穩定的介面給其他模組使用。

明確的依賴關係

模組通常需要其他的模組來履行它們的義務。模組的定義必須納入這種依賴關係，來讓模組可以自給自足。明確的依賴關係會產生**模組圖**（*module graph*）：裡面的節點代表模組，直線代表模組之間的依賴關係。要瞭解應用程式，以及使用所有必要的模組來執行它，取得模組圖非常重要，它提供可靠的基礎，來讓我們配置模組。

靈活性、易懂性，與重複使用性都與模組有關。模組可以靈活地組合成不同的配置，利用明確的依賴關係來確保每件事都可以一起合作。封裝可確保你永遠不需要瞭解實作的細節，而且永遠都不會無意間依賴它們。要使用模組，瞭解它的公開 API 就夠了。此外，如果你的模組公開定義良好的介面，並封裝實作細節，你就可以輕鬆地將它換成具有相同 API 的替代品。

模組化的應用程式有許多優點。有經驗的開發者都知道當基礎程式沒有被模組化時會產生什麼情況。諸如**義大利麵式結構**、**混亂的龐然大物**、**大泥球**等可愛的術語甚至還不足以形容它帶來的痛苦。不過，模組化不是萬靈丹，它是一種架構原則，當你正確地採納它們時，可相當程度地**預防**這些問題。

雖然如此，這一節的**模組化**定義是蓄意抽象化的。它可能會讓你想到元件開發（在上世紀風靡一時）、服務導向架構，或目前被大肆炒作的微服務。事實上，這些案例都是在各種抽象層面上試著處理類似的問題。

怎樣才能瞭解 Java 的模組？花一點時間來思考一下 Java 已經具備的模組核心原則（以及它缺乏的）是具啟發性的。

準備好了嗎？接下來，你可以進入下一節了。

在 Java 9 之前

Java 已被用來開發各式各樣與各種大小的軟體。有上百萬行程式的應用程式並不罕見。顯然，談到建立大型的系統，Java 已經做了一些正確的事情，甚至在 Java 9 問世之前。我們來針對 Java 9 模組系統出現之前的 Java 一一檢視模組化的三條核心原則。

我們可以使用**套件**與**操作修飾詞**的組合（例如 `private`、`protected` 或 `public`）來完成型態的封裝。例如，藉由讓一個類別成為 `protected`，你可以防止其他的類別操作它，除非它們屬於同一個套件。這會產生一個有趣的問題：如果你要從另一個套件使用那個類別，但仍然希望其他的套件不可使用它時，該怎麼辦？你會束手無策。當然，你可以讓類別成為 public。但是，public 代表向系統的所有其他的型態公開，也就是沒有封裝。你可以將這個類別放入 `.impl` 或 `.internal` 套件，來提醒使用者，使用這種類別是不明智的。但是，坦白說，誰會看那種東西？大家還是會任意使用它，只因為他們可以這樣做。我們無法隱藏這種實作套件。

就良好定義的介面而言，Java 從它問世以來一直都做得很好。你猜對了，我們要來討論 Java 自己的 `interface` 關鍵字。公開一個公用介面並且將實作類別隱藏在工廠後面，或使用依賴注入，是一種已被證實良好的做法。正如你將在這本書中看到的，介面在模組化系統中扮演核心的角色。

明確的依賴關係是事情開始瓦解的地方。是的，Java 確實有明確的 `import` 陳述式，不幸的是，這些 imports 是嚴格的編譯期結構。當你將程式碼包裝成 JAR 時，並無法知道有哪些其他的 JAR 具備你的 JAR 需要的型態。事實上，這個問題嚴重的程度，讓許多隨著 Java 語言一起演化的外部工具也試著解決這個問題。詳見以下的專欄。

用來管理依賴關係的外部工具：Maven 與 OSGi

Maven

Maven 組建工具解決的其中一個問題是編譯期依賴關係的管理。它將 JAR 之間的依賴關係定義在外部的 Project Object Model（POM）檔案裡面。Maven 的成功因素不是組建工具本身，而是它催生了一種稱為 Maven Central 的規範庫（canonical repository）。幾乎所有的 Java 程式庫在發表時，都會提供它們的 POM 給 Maven Central。各種其他的組建工具，例如 Gradle 或 Ant（連同 Ivy）都會使用同樣的規範庫與詮釋資料。它們都可在編譯期為你自動解析（傳遞性）依賴關係。

OSGi

OSGi 會在執行期做 Maven 在編譯期做的事情。OSGi 要求你在 JAR 中，用詮釋資料來列出被匯入的套件，這些套件稱為**包裹**（*bundles*）。你也必須明確地定義哪些套件是被匯出的，也就是說，可被其他的包裹看到。在應用程式開始執行時，所有的包裹都會被檢查：看看是不是每一個匯入的包裹都可以連接到一個匯出包裹。藉由自訂類別載入器這種巧妙的做法，我們可以確保執行期不會載入未被詮釋資料允許的包裹。如同 Maven，這需要全世界的人在他們的 JAR 內提供正確的 OSGi 詮釋資料。但是，雖然 Maven 因為 Maven Central 與 POMs 而獲得很大的成功，但可在 OSGi 使用的 JAR 沒有那麼普及。

Maven 與 OSGi 都建構在 JVM 與 Java 語言之上，是 JVM 與 Java 無法控制的。Java 9 在 JVM 核心與語言中解決了一些相同的問題。模組系統的目的，不是為了完全取代這些工具。Maven 與 OSGi（以及較小型的工具）仍然有它們的地位，差別只在於現在它們可以建構在一個完全模組化的 Java 平台之上。

目前 Java 提供一些堅實的結構來建立大型的模組化應用程式。不過，顯然，它絕對有改善的空間。

將 JAR 當成模組？

在 Java 9 之前，JAR 檔案應該是最接近模組的東西。它們有名稱、與群組有關的程式碼，也可以提供良好定義的公用介面。我們來看一個典型的，在 JVM 上執行的 Java 應用程式範例，來探討“將 JAR 當成模組”這種想法，見圖 1-1。

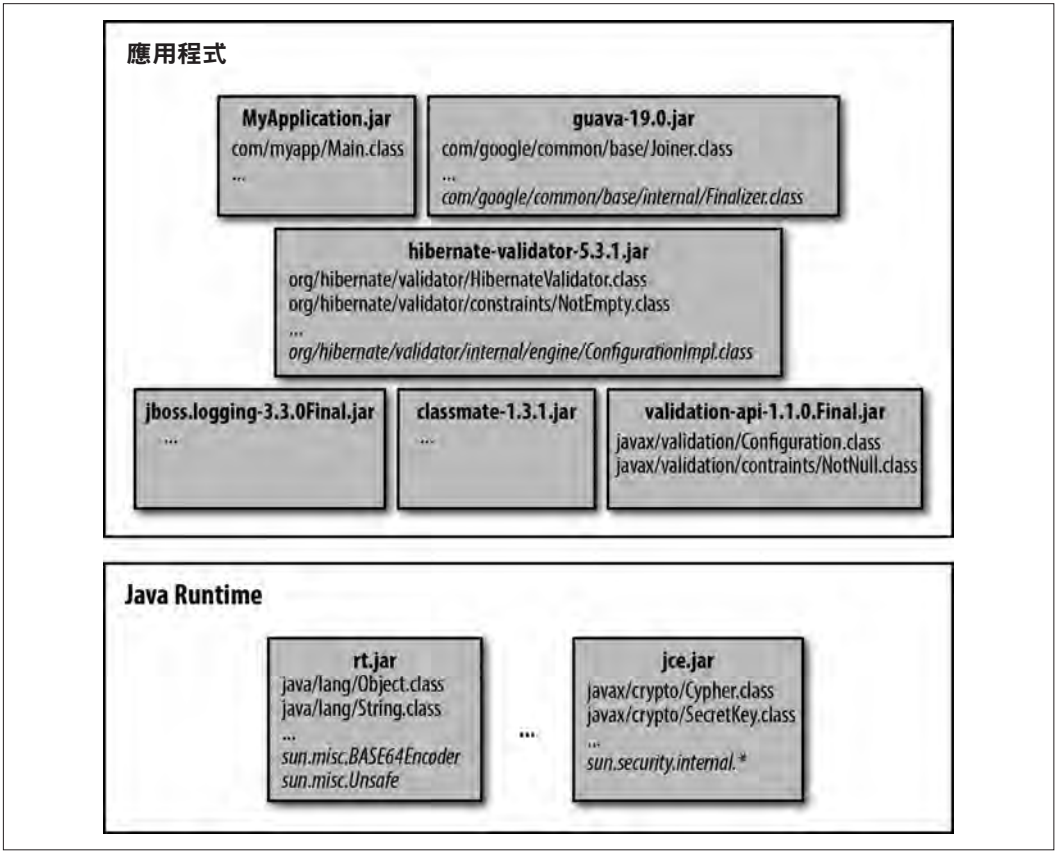


圖 1-1　MyApplication 是個典型的 Java 應用程式，它被包成 JAR，並使用其他的程式庫

這張圖有個稱為 *MyApplication.jar* 的應用程式 JAR，裡面有自訂的應用程式碼。這個應用程式使用兩個程式庫：Google Guava 與 Hibernate Validator。除此之外還有三個 JAR。它們是 Hibernate Validator 的傳遞性依賴項目，Maven 之類的組建工具或許會幫我們解析。MyApplication 是在 Java 9 之前的 runtime 上執行的，它本身會透過一些被包裹的 JAR 來公開 Java 平台類別。這個 Java 9 之前的 runtime 可能是 Java Runtime

Environment（JRE）或 Java Development Kit（JDK），無論哪一種，都會 include *rt.jar*（*runtime* 程式庫），它裡面含有 Java 標準程式庫的類別。

仔細看一下圖 1-1，你可以看到有些 JAR 用**斜體字**來列出類別。這些類別是預設的程式庫內部類別。例如，`com.google.common.base.internal.Finalizer` 是 Guava 本身使用的，但它不是官方 API 的一部分。它是個公用類別，因為有其他的 Guava 套件使用 `Finalizer`。不幸的是，這也代表 `com.myapp.Main` 在使用 `Finalizer` 這類的類別時不會有任何阻礙。換句話說，這裡沒有強力封裝。

Java 平台本身的內部類別也一樣。`sun.misc` 這類的套件一定可被應用程式碼使用，雖然技術文件嚴厲地警告它們不支援的 API，你不應該使用它們。儘管有這種警告，但是有許多應用程式碼一直都在使用 `sun.misc.BASE64Encoder` 這些工具類別。技術上來說，那些程式**可能**會因為 Java runtime 的任何更新而損壞，因為它們是內部實作類別。缺乏封裝，實際上會強迫 Java 將這些類別視為半公用 API，因為 Java 非常重視回溯相容性。這是一種不幸的情況，起因是缺乏真正的封裝。

明確的依賴關係呢？你已經知道，如果你嚴謹地看待 JAR，它沒有任何依賴關係資訊。你會這樣執行 MyApplication：

```
java -classpath lib/guava-19.0.jar:\
                lib/hibernate-validator-5.3.1.jar:\
                lib/jboss-logging-3.3.0Final.jar:\
                lib/classmate-1.3.1.jar:\
                lib/validation-api-1.1.0.Final.jar \
     -jar MyApplication.jar
```

設定正確的類別路徑是使用者的工作。而且，因為沒有明確的依賴資訊，這項工作不適合讓膽小的人執行。

類別路徑地獄

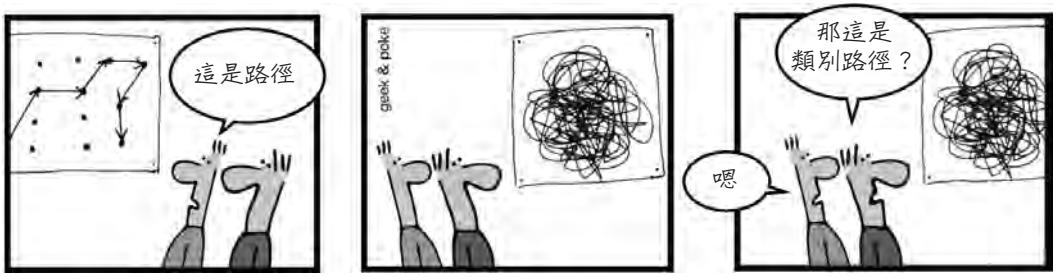
類別路徑是 Java runtime 用來找到類別的機制。在範例中，當我們執行 `Main` 之後，這個類別直接或間接參考的所有類別都會在某個時刻載入。你可以將類別路徑視為一個包含**可能**會在執行階段載入的所有類別的清單。雖然還有很多細節，但這種說法已經足以讓你瞭解類別路徑的問題了。

以下是 MyApplication 的類別路徑的概略內容：

```
java.lang.Object
java.lang.String
...
sun.misc.BASE64Encoder
sun.misc.Unsafe
...
javax.crypto.Cypher
javax.crypto.SecretKey
...
com.myapp.Main
...
com.google.common.base.Joiner
...
com.google.common.base.internal.Joiner
org.hibernate.validator.HibernateValidator
org.hibernate.validator.constraints.NotEmpty
...
org.hibernate.validator.internal.engine.ConfigurationImpl
...
javax.validation.Configuration
javax.validation.constraints.NotNull
```

這裡已經沒有 JAR 或邏輯群組的概念了。所有的類別都被列成一個平面（flat）清單，按照 -classpath 引數定義的順序。當 JVM 載入一個類別時，它會依序讀取類別路徑，來找出正確的那一個。當它找到類別之後，就會停止搜尋，將該類別載入。

如果它無法在類別路徑中找到類別時會發生什麼事情？你會得到一個執行期例外。因為類別是惰性（lazily）載入的，當某位倒楣的使用者第一次在你的應用程式按下按鈕時，可能會觸發這個例外。JVM 無法在啟動時有效地確認類別路徑的完整性。我們無法事先知道類別路徑是否完全，或者你是否應該加入另一個 JAR。顯然，這不太妙。



如果類別路徑有重複的類別，就會有更多隱患。假設你想要避免手動設定路徑，所以讓 Maven 用 POMs 的明確依賴資訊來建構正確的 JAR 集合來放入類別路徑。因為 Maven 會解析依賴關係的傳遞性，所以在這個集合中，經常會有某個程式庫的兩個版本（例如，Guava 19 與 Guava 18），雖然這不是你的錯。現在兩個程式庫 JAR 都被壓平到類別路徑內，按照不確定的順序。先出現的程式庫類別的版本會先被載入。但是，其他的類別可能希望與（可能不相容的）其他版本的類別合作，這也會導致執行期例外。一般來說，當類別路徑有兩個名稱（完全）相同的類別時，就算它們是完全無關的，也只有一個類別會“勝出”。

你就可以知道為何**類別路徑地獄**（也稱為 *JAR* **地獄**）在 Java 世界中會如此惡名昭彰了。有些人會用試誤法來調整路徑，但仔細想想，需要做這種事這相當可悲。脆弱的類別路徑仍然是造成問題與挫折的主因。我們希望可以得到更多關於執行期的 JAR 之間的資訊。這就像隱藏在類別路徑中的依賴圖，只是等待被發掘與利用。接著，我們要討論 Java 9 模組了！

Java 9 **模組**

現在，你已經充分瞭解 Java 目前的模組化優勢與限制了。因為 Java 9，我們在尋求良好架構的應用程式的旅途中，加入一位新的盟友：*Java* **模組系統**。在設計 Java 平台模組系統來克服目前的限制時，Java 定義了兩個主要的目標：

- 將 JDK 本身模組化。
- 提供一個模組化系統來讓應用程式使用。

這兩個目標是密切相關的。將 JDK 模組化的方式，是採用身為應用程式開發者的我們在 Java 9 中使用的同一套模組系統來完成的。

這個模組系統在 Java 語言與 runtime 加入模組的原生概念。模組可以匯出套件，或強力封裝套件。此外，它們也明確地表達與其他模組的依賴關係。你可以看到，模組化的三個原則都被 Java 模組系統履行了。

我們來回顧一下 MyApplication 範例，現在採用 Java 9 模組系統，見圖 1-2。

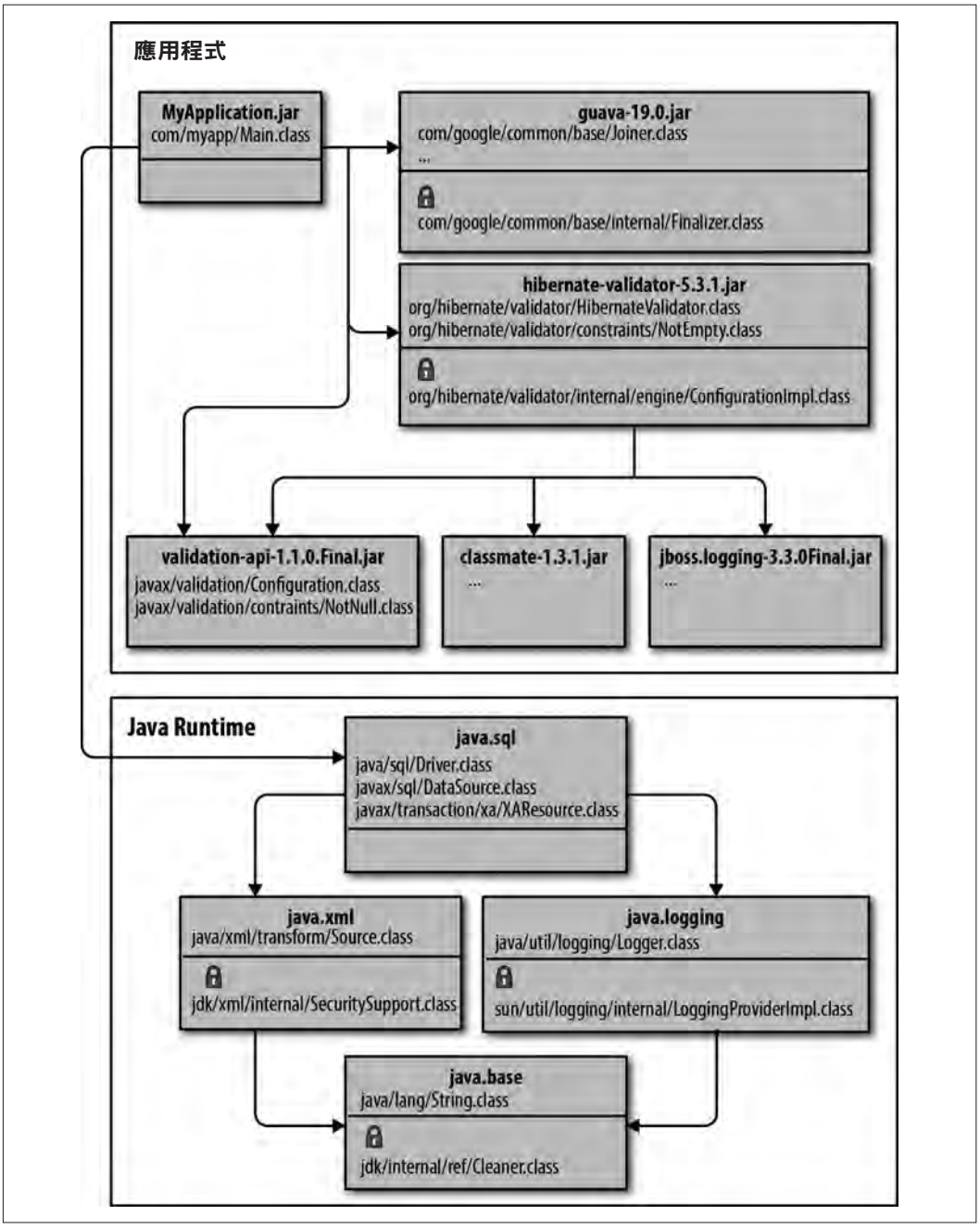


圖 1-2　建構在模組化的 Java 9 之上的 MyApplication 模組化應用程式

圖中的每一個 JAR 都被改為一個模組，模組裡面有其他模組的明確參考。hibernate-validator 使用 jboss-logging、classmate 與 validation-api 這些事實，已經成為它的**模組描述項**（*module descriptor*）的一部分了。模組有一個可公開使用的部分（在上面）與一個封裝的部分（在下面，用鎖頭表示）。因此，MyApplication 再也不能使用 Guava 的 Finalizer 類別了。透過這張圖，我們發現 MyApplication 也使用 validation-api 來註釋它的一些類別。更重要的是，MyApplication 與 JDK 的 java.sql 模組有明確的依賴關係。

圖 1-2 表達的應用程式資訊比圖 1-1 的類別路徑還要多。圖 1-1 只能指出 MyApplication 會與所有的 Java 應用程式一樣，都使用 *rt.jar* 裡面的類別，以及它會與（可能是不正確的）類別路徑上的許多 JAR 一起運行。

這只是應用層而已，一路下來，都有其他的模組。JDK 層也有一些模組（圖 1-2 展示其中一小部分）。如同應用層裡面的模組，它們都有明確的依賴關係，也會公開一些套件與隱藏其他的套件。在模組化的 JDK 中，最基本的**平台模組**是 java.base。它公開了 java.lang 與 java.util 等套件，這些都是其他模組必備的元素。因為你必須使用這些套件裡面的型態，所以每一個模組都潛在需要 java.base。如果應用模組需要除了 java.base 之外的平台模組的功能，那些依賴關係也必須是明確的，如同 MyApplication 依賴 java.sql 的情況。

最後，還有一種方法可以在 Java 語言中用較高的粒度（granularity）等級來表示程式碼的各個部分之間的依賴關係。想像一下，當你在編譯期與執行期擁有全部的資訊時，會有什麼好處。我們可以避免程式與其他未被參考的模組之間出現意外的依賴關係。工具鏈可藉由檢查模組的（傳遞性）依賴關係來得知為了執行它必須要有哪些其他的模組，並且可以使用這種知識來進行最佳化。

現在強力封裝、定義良好的介面，以及明確的依賴關係都已成為 Java 平台的一部分了。簡單來說，以下是 Java Platform Module System 最重要的優點：

可靠的配置

模組系統會在編譯或執行程式之前，先檢查收到的模組組合是否可以滿足所有依賴關係，可減少執行期錯誤。

強力的封裝

模組會明確地選擇哪些東西可公開給其他的模組，以避免內部實作細節的意外依賴關係。

可擴展的開發

明確的界限可讓團隊在平行工作的同時，建立可維護的基礎程式。唯有共用明確公開的公用型態，才可為模組系統建立一個自動強制執行的邊界。

安全

JVM 的最深層會強制進行強力封裝。這可限制 Java runtime 被攻擊的表面積，讓他人再也無法用反射來操作敏感的內部類別。

最佳化

因為模組系統知道哪些模組是在一起的，包含平台模組，所以在啟動 JVM 時不需要考慮其他的程式碼。它也開放一種可能性，可讓我們建立模組的最小配置版本來發布。此外，我們可以對這種模組集合執行整個程式的最佳化。在模組出現之前，這是非常困難的事情，因為沒有明確的依賴關係資訊，類別可能會參考類別路徑上的任何其他類別。

在下一章，我們要討論如何定義模組，以及它們的互動是以哪些概念來管理的。我們會藉由查看 JDK 本身的模組來探討。平台模組的數量遠超過圖 1-2 所示的。

在第 2 章探討模組化 JDK 是一種很棒的方式，可讓我們掌握模組系統概念的同時，熟悉 JDK 內的模組。畢竟，那些都是你會在你的模組化 Java 9 應用程式中最早使用的模組。完成第 2 章之後，我們就可以在第 3 章開始編寫自己的模組了。