第六章

Java 處理記憶體與並行處理的方法

本章會介紹平行處理（多執行緒）與 Java 平台的記憶體。這些主題基本上是糾纏在一起，所以把它們放在一起談是合理的。我們會涵蓋：

- Java 記憶體管理的介紹
- 基本標記與掃除垃圾回收演算法
- HotSpot JVM 會如何最佳化 GC，根據物件的生命期
- Java 的平行處理指令
- 資料可見度與可變性

Java 在記憶體管理的基本概念

在 Java，若物件不再被需要時，該物件所佔用的記憶體會自動地被回收。這是透過**記憶體回收**（garbage collection）（或稱自動記憶體管理）程序完成。記憶體回收技術，在某些程式語言（如 Lisp）中已使用多年。對習慣使用 C 語言與 C++（這些語言必須呼叫 `free()` 函式或使用 `delete` 運算子來回收記憶體）的程式設計師來說，會慢慢習慣。

事實是，在 Java 不需要記得去毀滅物件，這個特色也使 Java 成為受歡迎的語言。它也是特色之一，使得 Java 程式比起那些不支援自動記憶體回收的程式語言，較不會出錯。

不同的 VM 實作，用不同方式處理垃圾回收集合，且規格對 GC 的實作方式沒有加諸非常嚴格的限制。本章稍後，會討論 HotSpot JVM（這是 Oracle 與 OpenJDK 等 Java 版本的實作）。然而，這個不只是你可能碰到的 JVM，它是最普遍的伺服器端部署，且提供當代 JVM 產品的好範例。

Java 的記憶體洩漏

事實是，Java 支援記憶體回收，可大幅降低**記憶體洩漏**（memory leak）的發生率。記憶體洩漏會在記憶體持續配置，但都沒有回收時發生。乍看之下，記憶體回收之所以看似能預防記憶體洩漏，是因為它會回收所有不再使用的物件。

然而，如果有效（但不再使用）的參考指向不再使用的物件，在 Java 還是可能會發生記憶體洩漏。例如，若某個方法執行了很長時間（或永遠），在該方法中的區域變數的物件參考，其保留時間可能會比實際所需的時間還長。以下的程式碼會說明：

```
public static void main(String args[]) {
  int bigArray[] = new int[100000];

  // 使用 big_array 做一些計算並取得結果。
  int result = compute(bigArray);

  // 我們不再需要 big_array 了。它會在沒有任何參考指向它時，就會被垃圾回收。
  // 因為 big_array 是個區域變數，它會參考至該陣列，直到方法結束為止。
  // 但這個方法不會回傳。
  // 因此，我們必須明確地擺脫指向自己的參考，所以讓垃圾回收器知道可以回收該陣列。
  bigArray = null;

  // 無窮迴圈，處理使用者的輸入
  for(;;) handle_input(result);
}
```

記憶體洩漏也會發生，其時機是在使用 `HashMap` 或其他類似的資料結構，來把某個物件關連到另外一個物件的時候。甚至在這兩個物件都不再需要時，它們之間的關聯仍會保留在雜湊表，直到該雜湊表回收時，該物件才會跟著被回收。如果雜湊表的生命期比該物件長很多，就會發生記憶體洩漏。

標記與掃除簡介

JVM 很清楚知道它分配了那些物件與陣列。它們會存放在某幾種內部資料結構，這我們稱其為**分配表**（allocation table）。JVM 也能夠理解哪個區域

變數在每個堆疊框架參考至堆積中的物件與陣列。最後，藉由以下由堆積中的物件與陣列存放的參考，不管其關係的間接程度如何，JVM 可以追蹤與尋找所有仍然能參考到的物件與陣列。

因此，執行時期能夠知道分配的物件何時不再被其他有效的物件或變數所參考。若直譯器找到這樣的物件時，它知道可以安全地收回該物件的記憶體，也確實有這麼做。注意，垃圾回收器也可以偵測並回收出現參考循環，但並未被其他有效物件所參考的物件。

我們對**可抵達物件**（reachable object）的定義，指的是該物件能夠從某些應用程式執行緒的堆疊追蹤裡的某個方法中的一些區域變數開始，再沿著參考直到抵達該物件為止。這種類型的物件也可稱為是**活的**（live）物件。[註 1]

參考鏈除了從區域變數開始，還有其他可能性。通往可抵達物件的參考鏈，其根一般為 *GC 的根*（GC root）。

搭配這些簡單的定義，現在來看個簡單的方法，以這些原則為基礎，執行垃圾回收。

基本的標記與掃除演算法

一般（且簡單）的演算法處理集合程序，稱為**標記與掃除**（mark and sweep）。這發生在三個階段：

1. 走訪整個分配表，把每個物件都標記為**死去**（dead）。
2. 從指向堆積的區域變數開始，後面跟著所有從我們抵達的所有物件的所有參考。每次我們抵達某個尚未看過的物件或陣列，把它標記為**活的**（live）。繼續進行下去，直到我們完全探索所有可以從區域變數抵達的參考。
3. 掃除會再次橫跨分配表。對於沒有被標記為活的每個物件，回收在堆積的記憶體，並置回自由的記憶體列表。從分配表移除物件。

[註 1] 從 GC 的根開始的透徹探索程序，會產生稱為活物件的遞移閉包（transitive closure）— 這個術語是從圖形理論的抽象數學借用來的。

剛剛概略提到的標記與掃除，是演算法最簡單的理論形式。因為在以下章節會看到，真實的垃圾回收器會做的比這個更多。這邊敘述的背後基礎是基本理論，且設計目的是要容易了解。

因為所有物件都從分配表分配，GC 會在堆積滿溢前觸發。在這個標記與掃除說明中，GC 需要明確地存取整體堆積。這是因為應用程式碼會不斷地運作、建立，以及修改了物件，這會竄改結果。

在圖 6-1，我們顯露的效果是嘗試去垃圾收集物件，然而應用程式執行緒正在執行。

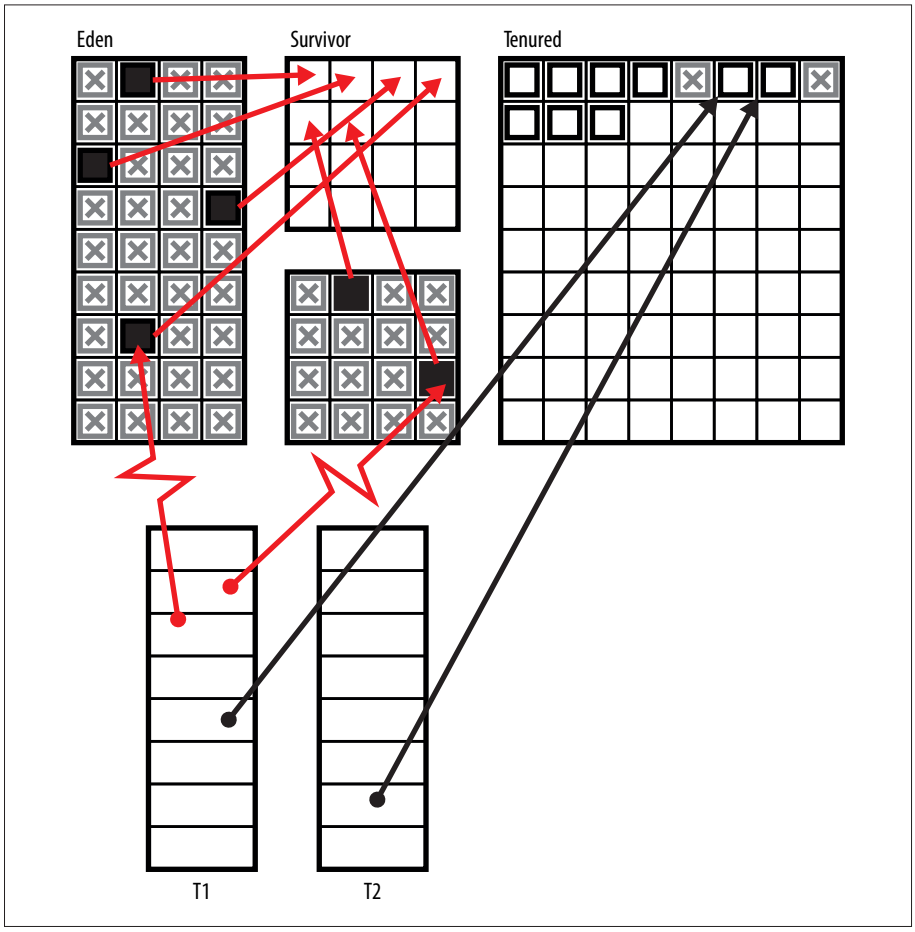

圖 6-1　堆積變化

要避免這點，一個簡單的 GC，就像某個剛示範的，會導致在執行時出現靜止世界（stop-the-world, STW）暫停 — 因為所有應用程式執行緒會停止，之後發生 GC，且最後應用程式執行緒會再次啟動。執行時期關切這點的方式，是在它們抵達**安全點**（safepoint）時中斷應用程式執行緒 — 比方說迴圈的開端或何時會呼叫方法。在這些執行點上，執行時期知道它會停止應用程式執行緒，也沒有問題。

這些中斷，有時候會令開發人員擔憂，但為了大部分主流用法，Java 是在作業系統的頂層運作，該系統會常態地把程序從處理器核心置換，所以這個稍微額外的中止通常沒什麼關係。在 HotSpot 的狀況，有很大部分的運作是在最佳化 GC，以及減少 STW 的次數，是因為它對應用程式的工作量很重要。我們會在下一節討論某些最佳化。

JVM 如何最佳化垃圾回收機制？

弱世代假說（weak generational hypothesis, WGH）是關於執行時期的實際情況之一，也是很適合的例子，這跟我們在第 1 章介紹的軟體有關。簡單來說，就是物件的平均壽命傾向（稱為**世代**（generation））。

通常物件的存活時間很短（有時稱為暫留物件（transient object）），且之後會符合垃圾收集的資格。然而，少部分物件的存活時間較長，且指定成為程式的較長期狀態的部分（有時候稱為程式的**工作集合**（working set））。這可以在圖 6-2 中看到，該處可以看到記憶體用量對照預期壽命繪製。

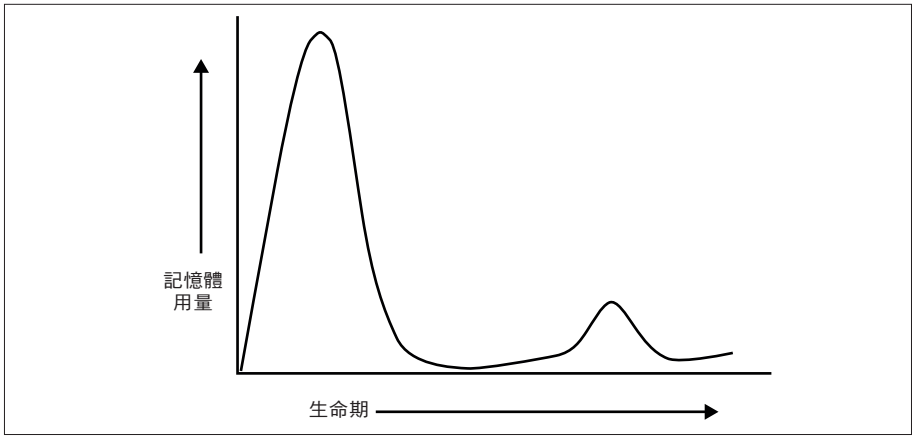

圖 6-2　弱世代假說

這個事實無法從靜態分析推導而來，且衡量軟體的執行時期行為時也還看不到，我們會發現在大量工作下，圖 6-2 大體上是對的。

HotSpot JVM 有個垃圾回收的子系統，它是特別設計來利用弱世代假說的優點，且在這一節，我們會討論這些技術如何應用到短壽命的物件（這是大部分狀況）。這個討論是直接適用 HotSpot，但其他伺服器等級的 JVM 通常使用了類似或相關的技術。

在其最簡單的形式中，**世代垃圾收集器**（generational garbage collector）僅有注意到 WGH。它們的工作，是某些額外的簿記，以監視記憶體，藉由支援 WGH，能夠得到的會比付出的多。世代收集器的最簡單形式，通常只有兩個世代 — 通常稱為年輕世代與老世代。

撤離

在我們原本的標記與掃除規劃中，在清理階段期間，會回收獨立的物件，且把空間歸還到自由清單。然而，若 WGH 成立，且在任何給定的 GC 循環上大部分物件都死了，之後它可能會合理使用替代的方式給回收中的空間。

這會藉由把堆積分隔到獨立的記憶體空間來運作。之後，在每次 GC 運作上，我們只會確定活物件的地點，並把它們移到不同空間，這個程序稱為**撤離**（evacuation）。做這件事的收集器，都稱為**撤離收集器**（evacuating collector）— 且它們具備的特性，是整體記憶體空間會在集合尾端抹除，一再被重複使用。圖 6-3 所示為運作中的撤離收集器。

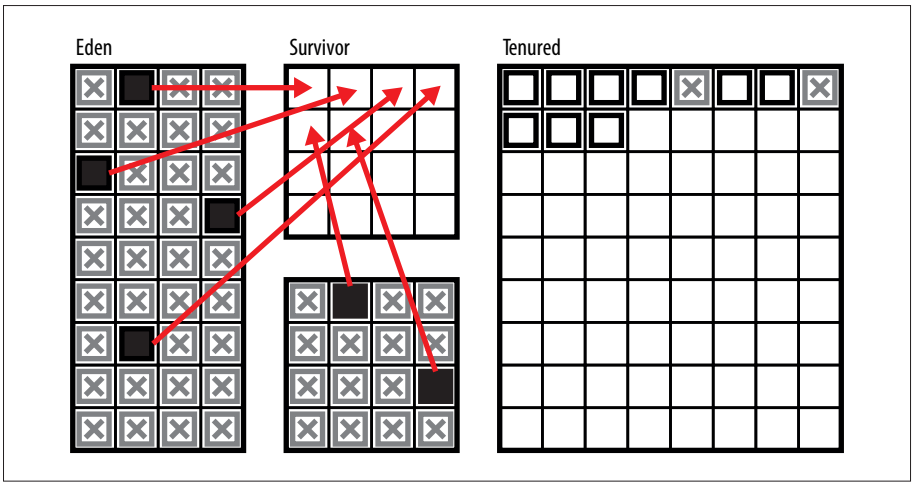

圖 6-3　撤離收集器

這可能比單純的收集方法還有效率的多，因為死的物件不會再接觸。GC 循環的長度，會跟活物件的數量成正比，而不是跟遭分配的物件數量。唯一的不利是稍微要簿記一下 — 我們必須付出複製活物件的代價，但跟藉由撤離策略實現的較大回報比起來，這幾乎都是非常少的代價。

HotSpot 自己管理 JVM 堆積，完全在使用者空間中，且不需要執行系統呼叫以分配或釋放記憶體。物件被初始建立的區域，通常稱為 Eden 或 Nursery，且大多數 JVM 產品（至少在 SE ／ EE 空間）都會在收集 Eden 時使用撤離策略。

撤離收集器的使用也允許每個執行緒分配的使用。這表示每個應用程式執行緒都可以被給定一大塊連續的記憶體（稱為**本地執行緒分配緩衝區**（thread-local allocation buffer）），在分配新物件時讓其排外使用。分配新物件時，這只牽涉到在分配緩衝區中移動指標，是一個成本極低的運算。

如果物件的建立，是在收集啟動之前，之後它不會有時間完成其目的，並在 GC 循環開始前死亡。在只有兩個世代的收集器內，這個短命的物件會被移到長生命期的地區，幾乎立刻死亡，且之後會待在那裏，直到下次完整的收集。因為這些很不頻繁（且計算成本通常貴很多），這似乎相當浪費。

要緩和這點，HotSpot 有個**生存者空間**（survivor space）的概念 — 這個區域儲藏的物件是從先前的年輕物件集合中存活的。存活的物件，會在不同存活者空間之間，被撤離收集器複製，直到抵達 *tenuring threshold* 為止，那時物件會**晉升**（promote）到老年世代。

生存者空間的完整討論，以及調校 GC 的方法，屬於本書討論範疇以外 — 對於應用程式的產品，應該查閱專門的資料。

HotSpot 堆積

HotSpot JVM 是相對複雜的程式片段，是由直譯器、即時編譯器，以及使用者空間的記憶體管理子系統組成。它包括了 C 語言、C++，以及相當大量的平台專屬的組合語言程式碼。

在這個基礎上，來總結一下對 HotSpot 堆積的說明，並翻新其基本特色。Java 堆積是記憶體的連續區塊，這會在 JVM 啟動時保留，但只有某些堆積是初次分配到各種記憶體池。當應用程式運作時，記憶體池會視需求調整大小。這些重新調整，會由 GC 子系統執行。

堆積中的物件

由應用程式執行緒建立在 Eden 的物件，會被未定的垃圾收集循環移除。GC 循環會在需要時（即是在記憶體越來越少時）執行。堆積會分成兩個世代，年輕世代與年老世代。年輕世代是由三個空間組成，Eden 與兩個生存者空間，而年老時代就只有一個記憶體空間。

在存活幾個 GC 循環後，物件會晉升到年老世代。只收集年輕世代的集合，計算成本通常都非常低（就需要的計算量來說）。HotSpot 使用的標記與掃除形式，比目前為止看過的都更進階，且準備好做額外的簿記去提昇 GC 的效能。在下一節，讓我們來討論年老世代，以及 HotSpot 如何處理長期存活的物件。

收集年老世代

討論垃圾收集器時，有一個重要部分，是開發人員應該知道的重要術語：

平行收集器（*parallel collector*）
: 垃圾收集器會使用多個執行緒去執行收集。

並行收集器（*concurrent collector*）
: 應用程式執行緒持續執行時，垃圾收集器仍可以同時執行。

到目前為止，討論的所有收集器都是平行的，但不是並行的收集器。預設上，代表年老時代的收集器，也是平行（但不是並行）的標記與掃除收集器，但 HotSpot 允許去插上不同的收集器。比方說，在本節稍後，我們會看到 CMS，這是平行且大多是並行的收集器，搭載在 HotSpot 之上。

回傳到預設的收集器，乍看之下是類似用在年輕世代的收集器。然而，它的差異體現在一個非常重要方面 — 它**不是**個撤離收集器。反而，年老世代會在收集發生時遭**壓縮**（compacted）。這很重要，以致於記憶體空間在一段時間後不會不完整。

其他收集器

這一節完全是專屬 HotSpot，且詳細的處理不屬於本書範疇，但值得知道關於替代收集器的存在。對於非 HotSpot 使用者，你應該查閱 JVM 的文件，去看還能有什麼選擇。

並行的標記與掃除

在 HotSpot，大多數廣泛使用的替代收集器，是並行的標記與掃除（concurrent mark and sweep, CMS）。這個收集器只用在收集年老世代 — 與它協力合作的平行處理器，其責任是清理年輕世代。

CMS 的設計只用在暫停較少的應用程式，那些程式不能處理超過數毫秒的 STW 暫停。這是驚人的小類別 — 除了財務交易以外的應用程式，很少有這種需要。

CMS 是個複雜的收集器，且常常難以有效地調校。在開發人員的寶庫裡，它可以是非常有用的工具，但應該不是草率或盲目的部署。它有的這些基本特性，是你應該知道的，但一個 CMS 的完整討論，超出本書範疇。有興趣的讀者應查閱專家的部落格與郵寄清單（如「Friends of jClarity」郵寄清單相當常處理關於 GC 的效能相關問題）：

- CMS 只有收集年老世代。
- CMS 沿著應用程式執行緒執行大多數的 GC 循環，減少暫停。
- 應用程式執行緒不必停止一段長時間。
- 有六個階段，全都設計去最小化 STW 暫停時間。
- 用兩個（通常非常短）STW 暫停，代替主要的 STW 暫停。
- 使用相當多簿記，以及多很多的 CPU 時間。
- GC 循環整體需要更久。
- 預設上，在並行執行時，有一半 CPU 會用在 GC。
- 除非是暫停較少的應用程式，否則不應該被使用。
- 若應用程式需要長時間的大量計算，則確定不應該用在應用程式。
- 不能壓縮，且假如有大量記憶體碎片，便退回使用預設的（平行的）收集器。

G1

垃圾優先（Garbage First）收集器（稱為 G1）是一個新的垃圾收集器，這是在 Java7 出現期間開發的（Java 6 就完成了一些前置工作）。它是設計來作為暫停較少的收集器，與 CMS 有所區別，且允許使用者就暫停的時間長短與頻率方面，去指定 GC 時的**暫停目標**（pause goal）。跟 CMS 不像的是，它的目的是用在需要長時間大量計算的工作。

G1 處理記憶體的方式較為粗糙，把記憶體分成數個地區（region），且 G1 會聚焦在垃圾較多的地區，記憶體釋放的效果最好。它是個撤離收集器，且在撤離個別地區時作了漸增的壓縮。

要開發適合一般用途，且到達產品等級的新收集器，無法在短時間內做到。因此，雖然 G1 已經在開發階段數年，但直到 2014 前期，G1 在大多數平台上仍是比 CMS 沒效率。不過，這個鴻溝已經被穩定的關閉，而 G1 現在某些工作是領先。可以相信，G1 在未來數月至數年，會變成最普遍的暫停較少的收集器。

最後，HotSpot 還有個 Serial（與 SerialOld）收集器，另外還有個收集器稱為「Incremental CMS」。這些收集器全都不建議使用，也不應該被使用。

終結

在資源處理方面有個舊技術，稱為**終結**（finalization），是開發人員應該要知道的。然而，這個技術極度不建議使用，且 Java 開發人員不應該直接在任何環境底下使用它。

符合終結規範的使用案例非常少，且只有少數的 Java 開發人員會碰到。若有任何疑點，就不要使用終結機制 — 嘗試關閉資源（try-with-resource）通常是正確的替代作法。。

終結機制的意圖，是在一旦不需要時，自動釋出資源。垃圾收集會自動地釋放物件所使用的記憶體空間，但物件也會存放其他種類的資源，如開啟檔案與網路連線。垃圾收集器不能幫你釋放這些額外資源，所以終結機制的意圖是允許開發人員在關閉檔案時執行清除任務，如終結網路連線、刪除暫存檔案等。

終結機制的運作如下：若物件有個 `finalize()` 方法（通常稱為 *finalizer*），物件不再被使用後（或無法取得時），但在垃圾收集器要回收分配給物件的空間前，會在某個時候呼叫。finalizer 會用在對物件執行資源清除。

在 Oracle ／ OpenJDK，技術使用如下：

1. 無法抵達要終結之物件時，指向它的參考會放在內部的**終結佇列**（finalization queue），且物件會被標記，因為 GC 執行的目的，會認為該物件活著。

2. 在終結佇列上的物件，會一個接一個遭移除，且會呼叫其 `finalize()` 方法。
3. 在呼叫 finalizer 後，物件並不是立刻就被釋放。這是因為 finalizer 方法可以使某個物件再復活，方式是把這個參考存放在某處（比方說，在某些類別的 public static 屬性），所以該物件又有參考指向它。
4. 因此，呼叫 `finalize()` 之後，在實際地把物件回收之前，垃圾收集子系統必須再次判定該物件未被參考。
5. 然而，即使物件復活，finalizer 方法也不會再呼叫一次。
6. 總結這些，意思是呼叫了 `finalize()` 的物件，通常會存活（至少）一個額外的 GC 循環（且若它們壽命夠長，那意思是一個額外完整的 GC）。

終結機制的核心問題，是 Java 不保證何時發生垃圾收集，也不保證收集的順序。因此，平台也不保證何時（是否會）呼叫 finalizer，而且不保證 finalizer 的呼叫順序。

這意思是，因為自動清除機制，是要保護稀少的資源（如檔案處理），這個機制會被設計破壞。我們不能保證終結機制會很快發生，快到足以防止資源用光。

finalizer 唯一真實的使用案例，是類別搭配本地方法的例子，存放著要開啟的一些非 Java 的資源。甚至這裡，也傾向使用嘗試關閉資源（try-with-resource）的區塊結構的方式，但它也可以合理的宣告一個 `public native finalize()`（這會被 `close()` 方法呼叫）－這會釋出本地資源，包括堆積以外的記憶體，這不在 Java 垃圾收集器的控制下。

終結機制的細節

對於適用終結機制的少數幾個例子，我們把使用該機制時的某些額外細節與警告也加進來：

- JVM 可以在未回收某些物件的情況下結束執行，因此不會呼叫某些 finalizer。這種情況下，資源（如網路連線）會由作業系統關閉及回收。但要注意，若 finalizer 要刪除某個檔案，但卻沒有去執行，則作業系統將不會刪除該檔案。
- 為了確保在 VM 結束前一定會執行特定動作，Java 提供了 `Runtime::addShutdownHook()` — 它可以安全地執行任意的程式碼，之後 JVM 才結束。

- `finalize()` 方法是個實體方法，而 finalizer 是在實體上運作。對於類別的終結，並沒有提供相同的機制。
- finalizer 是一個實體方法，沒有任何的引數或回傳值。每個類別只能有一個 finalizer，且它的名稱必須是 `finalize()`。
- finalizer 可以拋出任何種類的例外或錯誤，但當 finalizer 會自動地被垃圾收集子系統呼叫時，它拋出的例外與錯誤都會被忽略，而且只會導致 finalizer 方法回傳。

Java 對並行處理的支援度

執行緒（thread）的概念，是一個輕量級的執行單元－比程序還小，但仍能執行任意的 Java 程式碼。對作業系統來說，一般的實作方式是每個執行緒會是個成熟的執行單元，但要屬於某一個程序，而程序的位址空間會由所有組成該程序的執行緒共享。這個意思是，每個執行緒都可以獨立地排程，且有其自己的堆疊與程式計數器，但跟在相同程序內的其他執行緒共享記憶體與物件。

Java 平台從非常早期就開始支援多執行緒的程式設計，開發人員可以建立執行新的執行緒。建立的方式通常跟下面的例子一樣簡單：

```
Thread t = new Thread(() -> {System.out.println("Hello Thread");});
t.start();
```

這一小段程式會建立並開啟了新的執行緒，這執行了 lambda 運算式的主體。對於使用舊版 Java 的程式設計師，lambda 會有效率地被轉換為 `Runnable` 的實體，之後才傳遞給 `Thread` 建構子。

執行緒機制允許新執行緒搭配原本的應用程式執行緒並行地執行，且 JVM 本身的執行緒會因為各種目的啟動。

對於 Java 平台的大部分實作，應用程式執行緒讓它們自己去存取由作業系統**排程器**（scheduler）控制的 CPU－作業系統的內建部分是負責管理處理器執行的時間段落（且不允許應用程式執行緒超出其分配時間）。

Java 近期版本，逐漸成長的趨勢是往**執行時期管理並行**（runtime-managed concurrency）的方向。這個觀念，對於許多希望明確由開發人員管理執行緒的人來說，並非他們想要的。執行時期反而應該提供「自主管理（fire and

forget）」的功能，藉以讓程式指出什麼需要完成，但關於該如何完成的低階細節，會留到執行時期才處理。

這個觀點可以在 java.util.concurrent 包含的並行工具集裡看到，關於這完整的討論超出本書的範疇。有興趣的讀者應該參考由 Brian Goetz 等人合著、Addison-Wesley 出版的《Java Concurrency in Practice》。

本章剩餘的部分，會介紹 Java 平台提供的低階並行機制，以及每個 Java 開發人員對這部分應該了解的部分。

執行緒生命循環

現在開始來看應用程式執行緒的生命週期。每個作業系統都有對執行緒的觀點，會在細節上有所不同（但高階觀點大多相似）。Java 會努力嘗試把這些細節抽象化，且有個列舉稱為 Thread.State — 這會包裝整個作業系統對執行緒狀態的觀點。Thread.State 的值會提供執行緒生命循環的概觀：

NEW
: 執行緒已經建立，但其 start() 方法尚未被呼叫。所有執行緒都會從這個狀態開始。

RUNNABLE
: 執行緒正在執行，或在作業系統對它排程時就可執行。

BLOCKED
: 執行緒因為在等待取得鎖定，以便進入 synchronized 的方法或區塊，所以並未執行。本節稍後會看到更多關於 synchronized 方法和區塊的資訊。

WAITING
: 執行緒因為呼叫了 Object.wait() 或 Thread.join() 而未執行。

TIMED_WAITING
: 執行緒因為呼叫了 Thread.sleep()，或搭配逾時值來呼叫 Object.wait() 或 Thread.join() 而未執行。

TERMINATED
: 執行緒已執行完畢。它的 run() 方法已正常結束，或藉由拋出例外而結束。

這些狀態表達了普遍的（至少在主流作業系統）執行緒觀點，導致如圖 6-4 中的現象。

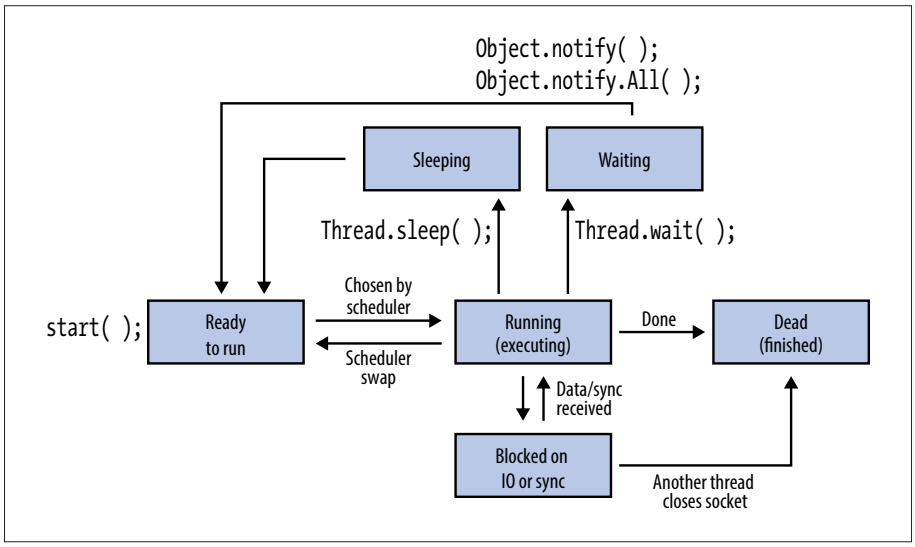

圖 6-4　執行緒生命循環

只要用 Thread.sleep() 方法，執行緒也可以睡眠。該方法需要一個以毫秒為單位的引數，這指出執行緒會睡多久，如下所示：

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

關於睡眠時間的引數，是對作業系統的請求，而不是絕對要求。比方說，可能會睡的比請求的還久，這會視負載或其他執行時期環境的特定因素而定。

本章稍後會討論 Thread 的其他方法，但首先需要加入一些重要的理論，用來處理執行緒如何存取記憶體，且那是了解多執行緒程式設計困難的原因之基礎，且了解它帶給開發人員很多問題的原因。

可見度與可變性

在 Java，這本質上等同在某個程序中全部的 Java 應用程式執行緒，都有它們自己的堆疊（與區域變數），但它們共享單一堆積。這會使得在不同執行緒之間共享物件變得非常簡單，因為需要做的事只有把參考從一個執行緒傳遞到另一個執行緒。這會在圖 6-5 說明。

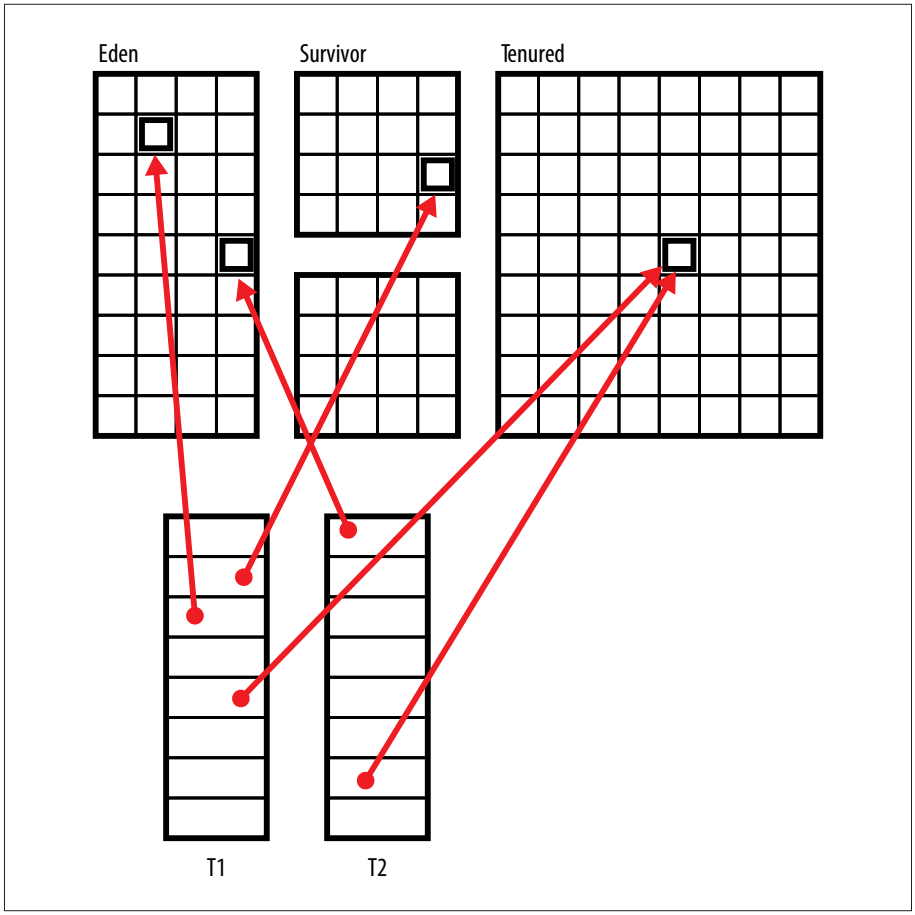

圖 6-5　在不同執行緒之間共享記憶體

這會形成 Java 一般的設計原則 — 該物件**預設是可見的**（visible by default）。若有一個指向物件的參考，我可以複製它，並把它傳遞給另一個執行緒，沒有任何限制。Java 的參考本質上是種指標，指向某個記憶體位址 — 且執行緒會共享相同的記憶體空間，所以預設可見是個自然的模型。

除了預設可見之外，Java 有另一個特質，對完整了解並行性很重要，就是該物件是**可變的**（mutable）－物件實體的屬性內容通常可以改變。藉由使用 `final` 關鍵字，我們可以製作獨立的變數或參考常數，但這不適用物件的內容。

本章剩餘部分，會看到這兩個特性的結合－橫跨執行緒的可見度與物件可變性－在嘗試推論並行 Java 程式時，會提昇很多複雜度。

並行安全性

若要撰寫正確的多執行緒程式，那麼會希望程式滿足特定的重要特性。我們想要的是：

不管呼叫什麼方法，也不管應用程式執行緒如何被作業系統排程，**安全的多執行緒程式**（safe multithreaded program）不可能有任何物件不符規則，或跟另一物件有不一致的狀態。

第 5 章定義的安全物件導向程式，是只要呼叫其存取方法，那麼物件可以在符合規則的狀態之間轉移。這個定義在單一執行緒的程式中會運作的很好。然而，若嘗試把這個定義延伸到並行的程式，還有個特別的難處。

對於大多數狀況，作業系統會排程執行緒，在各個時間分配到特定處理器核心上執行，這是依負載與其他在系統上運作的部分而定。若負載很重，也可能是需要執行其他的程序。

作業系統會在需要時，強制從 CPU 核心移除 Java 執行緒，此時不管執行緒在做什麼（即便方法執行到一半），都會立即中止。然而，如第 5 章討論的，當方法在操作某個物件時，可以暫時讓物件處於不符規則的狀態，假設在方法結束前會修正這一點。

這個意思是，若某個執行緒交換之後，才完成長時運作的方法，**即使程式遵守安全規則**，也可能會使物件處於不一致狀態。這個觀念的另一個說法是，即使資料型別已經正確地針對單執行緒的狀況建立模型，仍然會需要保護，以防並行的效果。添加在額外保護層上的程式碼，便具備**並行性安全**（concurrently safe）的特性。

在下一節，會討論滿足安全的主要方式，且在本章末尾，會碰到在某些環境下有用的其他機制。

排外與保護狀態

任何要修改或**讀取**（read）狀態的程式碼，若狀態可能不一致，那麼該程式碼必須受到保護。要做到這點，Java 平台只提供一個機制：**排外**（exclusion）。

思考一個方法，其中包含一連串運算，若在中途中斷，可以使物件處於不一致或不符規則的狀態。若這個不符規則的狀態，可由另一個物件看見，則會發生不正確的程式行為。

比方說，思考一台 ATM 或其他的提款機：

```
public class Account {
    private double balance = 0.0;  // 必須大於等於 0
    // 假設其他屬性（如 name）與方法（如 deposit()、checkBalance()，
    以及 dispenseNotes()）的存在

    public Account(double openingBal) {
        balance = openingBal;
    }

    public boolean withdraw(double amount) {
        if (balance >= amount) {
            try {
                Thread.sleep(2000);  // 模擬風險檢查
            } catch (InterruptedException e) {
                return false;
            }
            balance = balance -amount;
            dispenseNotes(amount);
            return true;
        }
        return false;
    }
}
```

一連串在 `withdraw()` 裡的運算，會使物件處於不一致的狀態。特別是，在我們檢查了帳戶後，第二個執行緒會在第一個執行緒因為模擬風險檢查而睡著時來到，且帳戶可能透支，違反了 `balance >= 0` 的限制。

這是系統的範例，該處對物件的運算都符合單執行緒安全（因為若從單一執行緒呼叫物件，則該物件不會變成不符規則的狀態（`balance < 0`），但不滿足並行安全。

要讓開發人員能夠使程式碼具備並行安全特性，Java 提供了 synchronized 關鍵字。這個關鍵字可以用在區塊或方法上，且使用時，平台便會限制對區塊與方法內部的存取。

因為 synchronized 包圍了程式碼，許多開發人員的結論是，Java 的並行性是跟程式碼有關。有些文字甚至把同步區塊或同步方法中的程式碼稱為**臨界區段**（critical section），且認為它是並行性的重要層面。狀況並非如此；而是必須防止的資料不一致，之後就會看到。

對於每個建立的物件，Java 平台都會使用**監控器**（monitor）去追蹤特別的符號。這些監控器（也稱為**鎖**（lock））會被 synchronized 用來指出以下的程式碼會暫時地讓物件處於不一致。對同步區塊或同步方法，是一連串事件的序列：

1. 執行緒需要修改物件，且可能在中間步驟有短暫的不一致。
2. 執行緒獲得監控器，指出它請求暫時排外的存取物件。
3. 執行緒修改了物件，使它在完成時處於一致的、符合規則的狀態。
4. 執行緒釋放了監控器。

若另一個執行緒企圖在物件修改時獲得鎖，之後企圖去獲得鎖定的區塊，直到存放的執行緒釋放鎖為止。

注意你未必要使用 synchronized 指令，除非程式建立多個執行緒來共享資料。如果都只有一個執行緒在存取資料結構的話，就沒有必要使用 synchronized 來保護它。

這至關重要的一點 — 獲得監控器**不會**防止存取物件。它只會預防其他執行緒要求鎖定。正確的並行安全程式碼，會請求開發人員確保所有的存取（可能修改或**讀取**不一致的狀態），都要取得物件監控，之後才能運算或讀取該狀態。

換句話說，若 synchronized 方法操作某個物件，且把它置於一個不符規則的狀態中，且另一個方法（但未同步化）會讀取物件，便還是能看到不一致狀態。

同步化是個合作機制，用來保護狀態，且它會因此非常脆弱。在整體系統安全上，單一的臭蟲（如遺忘需要 synchronized 關鍵字的方法）會有很慘的結果。

使用 synchronized 這個字的理由是，要作為表示「請求暫時排外的存取」的關鍵字，是除了要取得監控之外，JVM 也會在進入區塊時，從記憶體再讀取物件目前的狀態。類似的是，離開 synchronized 區塊或方法時，JVM 會把物件的修改狀態回存至主記憶體。

沒有同步化，系統裡不同的 CPU 核心，可能不會看到記憶體的相同面向，且記憶體的不一致會危害運作程式的狀態，如在 ATM 的例子看到的。

volatile

Java 提供了另一個關鍵字，處理並行的資料存取。這是 volatile 關鍵字，且它會在使用應用程式碼之前指出，屬性或變數的值必須從記憶體再讀取。相同的，在 volatile 值被修改後，之後一寫到變數，就必須寫回記憶體。

volatile 關鍵字有個常見的用法，是在「執行直到關機」（run-until-shutdown）模式。這也會用在多執行緒程式設計，該處有外部的使用者或系統需要發送訊號到處理的執行緒，指出它應該結束目前的工作，並優雅的關閉。這有時稱為「優雅完成」（graceful completion）模式。現在來看個典型範例，假設這個程式碼在談的處理執行緒是實作 Runnable 的類別：

```
private volatile boolean shutdown = false;

public void shutdown() {
    shutdown = true;
}

public void run() {
    while (!shutdown) {
        //... 處理另一個任務
    }
}
```

一直以來，shutdown() 方法都沒有被另一個執行緒呼叫，處理的執行緒持續循序地處理任務（這通常非常有效地結合 BlockingQueue 傳送工作）。一旦 shutdown() 被另一個執行緒呼叫，之後處理執行緒會立即看到 shutsown 旗標改為 true。這不影響運作中的工作，但一旦任務結束，處理執行緒不會接受另一種任務，且反而是優雅的關閉。

Thread 的有用方法

在建立新的應用程式執行緒時，Thread 類別有許多方法可以用，讓程式設計師輕鬆一些。這不是個詳盡的清單 — Thread 有許多其他方法，但這是某些更常見的方法說明。

getId()

這個方法回傳了執行緒的 ID 數字，其型別是 long。這個 ID 會在執行緒的生命期中保持相同。

getPriority() 與 setPriority()

這些方法都被用來控制執行緒的優先度。排程器會判定該如何處理執行緒優先序的問題 — 比方說，某個策略是在有高優先序的執行緒等待時，能夠不去執行低優先序的執行緒。大部分狀況，無法影響排程器如何解讀優先序。執行緒的優先序會在整數 1 到 10 之間表達。

setName() 與 getName()

允許開發人員去設定或擷取個別執行緒的名稱。幫執行緒命名在實務上是好的作法，因為有助於除錯，特別是在使用如 jvisualvm 的工具時，我們會在第 378 頁的「VisualVM」章節中討論。

getState()

回傳 Thread.State 物件，會指出該執行緒處於哪個狀態，如第 219 頁的「執行緒生命循環」章節中定義的每個值。

isAlive()

用來測試執行緒是否仍存活。

start()

這個方法是用來建立新的應用程式執行緒，且對它排程，搭配 run() 方法作為執行的進入點。執行緒通常會在到達 run() 方法的結尾，或執行到那個方法的 return 指令敘述時結束。

interrupt()

若執行緒在呼叫 `sleep()`、`wait()`，或者 `join()` 時中止，之後呼叫在 `Thread` 物件上的 `interrupt()`，該物件表示的執行緒會導致執行緒被 `InterruptedException` 送出（且喚醒）。若執行緒涉及中斷的 I/O，之後 I/O 會終結且執行緒會收到 `ClosedByInterruptException`。即使執行緒未致力於任何可以被中斷的活動，執行緒的中斷狀態，還是會設定為 true。

join()

目前的執行緒會等待，直到執行緒對應的 `Thread` 物件已經死去。可以把它想成是個不繼續執行的指令，直到其他的執行緒完成為止。

setDaemon()

使用者執行緒（user thread）是一種執行緒，會防止程序仍存活時就結束 — 這是執行緒的預設設定。有時候，程式設計師想要執行緒不會阻止程式結束 — 這些都稱為 *daemon* **執行緒**（daemon thread）。daemon 執行緒或使用者執行緒的狀態，可被 `setDaemon()` 方法所控制。

setUncaughtExceptionHandler()

當執行緒以拋出例外的方式結束時，預設的行為就是列出執行緒的名稱、例外類型、例外訊息，以及堆疊追蹤。如果這不足夠，你可以安裝自製的處理器，給執行緒中未捕捉的例外使用。參考下例：

```
// 這個執行緒剛拋出例外
Thread handledThread =
   new Thread(() -> { throw new UnsupportedOperationException(); });

// 給執行緒一個名稱，以輔助除錯
handledThread.setName("My Broken Thread");

// 這是針對錯誤的處理程序。
handledThread.setUncaughtExceptionHandler((t, e) -> {
    System.err.printf("Exception in thread %d '%s':" +
        "%s at line %d of %s%n",
        t.getId(), // 執行緒的 id
        t.getName(), // 執行緒的名稱
        e.toString(), // 例外名稱與訊息
        e.getStackTrace()[0].getLineNumber(),
        e.getStackTrace()[0].getFileName()); });
handledThread.start();
```

這在某些情形中有用，比方說，若某個執行緒正監督一群工作執行緒，之後這個模式會用來重啟任何已死去的執行緒。

Thread 中不建議再使用的方法

Thread 中除了有用的方法，也有許多開發人員不該用的不安全方法。這些方法構成部分的原始 Java 執行緒 API，但很快地發現它們不適合開發人員用。不幸的是，由於 Java 需要與過去相容，所以到目前為止都不可能從 API 中移除。開發人員僅需要知道它們，且在**所有**狀況下都要避免使用。

stop()

Thread.stop() 幾乎不可能在正確使用之下，能不侵犯並行安全性，因為 stop() 立即中止了執行緒，而不給它任何機會去把物件恢復到符合規則的狀態。這是對原則（如並行安全性）的直接反抗，且因此絕不應該被使用。

suspend()、resume()，以及 countStackFrames()

在 suspend() 機制暫停時，不會釋放任何它存放的監控器，所以其他企圖存取那些監控器的執行緒，會陷入僵局。實務上，這個機制會產生這些僵局間的競爭狀況與 resume()，讓這群方法處於不可使用的狀況。

destroy()

這個方法絕對不會被實作－如果有實作該方法的話，它會跟 suspend() 方法一樣，受相同的競爭狀況所害。

這些不建議使用的方法，全都應該要避免。取代作法是，一組安全的替代模式，會跟前面開發的方法達成相同的預期目標。這些模式中有一個好的例子，是之前看過的「執行直到關機」模式。

操作執行緒

為了要有效地用多執行緒程式工作，重點是在命令中關於監控器與鎖定的基本事實。這個檢查清單包含該知道的主要事實：

- 同步化是跟保護物件狀態與記憶體有關，而非與程式碼有關。
- 同步化是執行緒之間的合作機制。有個臭蟲會破壞合作的模型，且有深遠的結果。

- 取得監控器只會防止其他執行緒取得監控器 — 它不會保護物件。
- 非同步的方法會看到（且修改）不一致的狀態，甚至是在物件的監控器已經鎖上的狀況。
- 鎖定 `Object[]` 不會鎖定個別的物件。
- 基本型別不可變，所以它們不能（且不需要去）被鎖定。
- `synchronized` 不會出現在介面中的方法宣告上。
- 內部類別只是語法上的便利，所以對內部類別的鎖定，對外圍類別沒有效果（反之亦然）。
- Java 的鎖定是**重新進入**（reentrant）。這個意思是，若存放著監控器的執行緒，碰到了有相同監控器的同步化區塊，便會進入區塊。[註 2]

我們也會看到，可以要求執行緒在某段時間內睡眠。也很有用的是，在未指定的一段時間內睡眠，且等到條件符合為止。在 Java，這是由 `wait()` 與 `notify()` 方法來處理，那是在 `Object` 上呈現。

就和每個 Java 物件都有與其相關的鎖定一樣，每個物件都可以維護等待的執行緒清單。當執行緒呼叫物件的 `wait()` 方法時，執行緒所持有的每一個鎖都會被暫時釋放，而執行緒會被加到那個物件的等待執行緒清單，並停止執行。當另一個執行緒呼叫同一個物件的 `notifyAll()` 方法時，物件就會喚醒等待中的執行緒，並允許它們繼續執行。

比方說，讓我們看一個佇列的簡化版，對多執行緒使用是安全的：

```
/*
 * 有個執行緒會呼叫 push()，把一個物件放在佇列上。
 * 另一個執行緒會呼叫 pop()，把物件從佇列取出。
 * 若沒有資料，pop() 會等到有一些資料，使用 wait()/notify()。
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>();   // 儲藏庫
    public synchronized void push(E o) {
        q.add(o);                            // 將物件附加到串列的結尾
        this.notifyAll();                    // 告訴等待中的執行緒，資料已備妥
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
```

[註 2] Java 的外部，並非所有的鎖定實作都具備這個特性。

```
            return q.remove();
        }
    }
```

若佇列為空（這會讓 pop() 出錯），這個類別使用了 WaitingQueue 實體的 wait()。等待中的執行緒，會暫時釋放其監控器允許另一個執行緒回收它 — 是一個可能 push() 某些新東西到佇列上的執行緒。當原本的執行緒再次被喚醒時，它會重新啟動的位置，是在原本開始去等待的地方 — 且它會再取得其監控器。

wait() 與 notify() 必定被用在 synchronized 方法或區塊，因為暫時鬆開鎖定，會要求它們去適當地工作。

一般來說，大部分開發人員不應該啟動它們自己的類別，就像這個例子 — 反而要利用 Java 平台提供的函式庫與元件。

總結

本章討論 Java 在記憶體與並行性的觀點，且看到這些主題本質上互相聯繫。因為處理器開發更多的核心，我們會需要用並行程式設計的技術，去有效利用那些核心。未來的應用程式要能執行良好，關鍵在於並行性。

Java 的執行緒模型，是基於三個基本概念：

共享的、預設可見的可變狀態（*Shared, visible-by-default mutable state*）

這意思是，物件會在程序中的不同執行緒之間能夠簡單地分享，且可以由執行緒修改的物件，會存放指向自身的參考。

先佔式執行緒排程（*preemptive thread scheduling*）

每過一段時間，作業系統的執行緒排程器，會把執行緒交換進出核心。

物件狀態只能用鎖定保護（*Object state can only be protected by locks*）

鎖定會難以正確地使用，且狀態是相當脆弱 — 甚至是在未預期的地方（如讀取運算）出錯。

把這些兜在一起，Java 並行性的這三個層面，解釋了多執行緒程式設計會讓開發人員頭痛的原因。