隨著第八版的內容修訂工作，我們思考《Java 技術手冊》的核心意義是什麼，嘗試更新本書想要表達的觀念。現今的 Java 開發人員不只需要知道語法和 API，Java 開發環境已臻成熟，對所有開發人員來說，並行性、物件導向設計、記憶體和 Java 型態系統等這些主題變得越來越重要。

在這次的版本裡，我們採取的做法是只關注最新的 Java 版本，這也是多數 Java 開發人員會有興趣的版本，因此，本書通常是介紹 Java 8 之後推出的新功能。

以 Java 9 釋出時推出的模組系統為例，代表 Java 語言的重大改變，至少對某些開發人員來說還是新功能。然而，模組系統算是進階主題，某種程度來說，可以從 Java 語言獨立出去，變成其他部分，所以本書將這個主題的內容單獨限制在一個章節裡。

本書內容簡介

前六章記述的內容是以 Java 語言和 Java 平台為主，這些算是本書必讀的章節。整體內容雖然偏向在 Oracle/OpenJDK（Open Java Development Kit）的環境下實作 Java，但也不盡然如此。使用其他 Java 環境的開發人員同樣也可以在本書找到大量的實用內容。第一部分的章節有：

第 1 章「Java 開發環境簡介」

本章內容焦點會放在 Java 語言和 Java 平台的入門簡介，解釋 Java 語言的重要特性和優勢，包括 Java 程式的生命週期。部分內容還會提到 Java 安全性，回應一些 Java 多年來受到的批評。

第 2 章「從零開始學習 Java 語法」

本章將詳細解釋 Java 程式語言的用法，包括 Java 8 推出時 Java 語言的主要異動。設計本章內容時，我們假設讀者沒有豐富的程式設計經驗，所以內容十分詳盡，篇幅相較於其他章節來得長。已有 Java 開發經驗的程式設計師可將本書作為程式語言參考書；在其他程式語言擁有豐富經驗的程式設計師，像是 C 和 C++ 語言的開發人員閱讀本章內容後，應該可以快速掌握 Java 語法；經驗不多的新手程式

碁峰 www.gotop.com.tw

設計師若能仔細研讀本章內容，就能學到 Java 程式設計的基礎，建議搭配其他入門書籍一起學習，例如，由 Kathy Sierra、Bert Bates 和 Trisha Gee 所撰寫的《深入淺出 *Java* 程式設計》（Head First Java，O'Reilly 出版）。

第 *3* 章「*Java* 物件導向程式設計入門」

本章將說明如何利用第 2 章記載的 Java 語法來撰寫簡單的物件導向程式，學習使用 Java 類別和物件。本章同樣假設讀者以前沒有物件導向程式設計的經驗，新手程式設計師可將本章內容視為教學課程，已有 Java 開發經驗的程式設計師請將本書作為程式語言參考書。

第 *4* 章「*Java* 型態系統」

本章內容是以 Java 物件導向程式設計的基本說明為基礎，導入 Java 型態系統的其他面向，例如，泛型、列舉和標註型態。掌握更完整的 Java 面貌之後，就能進一步討論 Java 8 最大的變化——Lambda 表達式。

第 *5* 章「*Java* 物件導向程式設計進階技巧」

本章將介紹幾個基本技巧，用於設計健全的物件導向程式，還會簡短提到設計模式以及這個主題在軟體工程方面的應用。

第 *6* 章「*Java* 處理記憶體和並行性的做法」

本章會解釋 Java 虛擬機如何自動為程式設計師管理記憶體，以及 Java 在並行程式設計和執行緒方面的支援機制如何密切結合記憶體和可視性。

前六章著重於學習 Java 語言，帶領讀者快速掌握使用 Java 平台需要的觀念。第二部分的章節則聚焦於如何在 Java 開發環境中實際完成程式設計的工作。本書不僅納入大量的程式範例，我們設計這些章節的目的是作為補充資料，協助讀者了解某些程式書籍中發現的錦囊妙計。這部分的章節有：

第 *7* 章「程式設計和文件註解的使用慣例」

本章記錄 Java 程式設計廣泛採用的重要使用慣例。解釋如何加入特殊格式的文件註解，讓 Java 程式碼可以自動產生技術文件。

碁峰 www.gotop.com.tw

第 *8* 章「*Java* 集合的用法」

本章會介紹 Java 集合使用的標準函式庫，這些函式庫內建的資料結構，幾乎可說是每個 Java 程式運作的關鍵，例如，`List`、`Map` 和 `Set`。本章還會詳細解釋 Java 8 新導入的 `Stream` 抽象層以及 Lambda 表達式與 Java 集合之間的關係。

第 *9* 章「處理常見的資料格式」

本章會討論如何使用 Java 語言有效處理常見的資料格式，例如，文字、數字和時間資訊（日期和時間）。

第 *10* 章「檔案處理與 *I/O*」

本章會介紹幾個不同的檔案存取方法，從 Java 早期版本使用的經典做法到近期的現代方法，甚至還會提到非同步設計。最後結尾會簡短介紹 Java 平台的核心 API，說明如何利用這些 API 撰寫網路應用程式。

第 *11* 章「類別載入、反射機制和方法控制碼」

本章主要內容是介紹超程式設計（metaprogramming）蘊含的微妙藝術。首先是跟 Java 型態有關的詮釋資料（metadata）概念，接著會談到類別載入的議題，以及 Java 安全模型如何動態載入 Java 型態。本章結尾會介紹幾個類別載入的應用範例，和方法控制碼（method handles）近期發展的新特性。

第 *12* 章「*Java* 平台模組」

本章會介紹 Java 9 導入的重大特性——Java 平台模組系統（JPMS），介紹這套系統帶來的廣大影響。

第 *13* 章「*Java* 開發平台內含的工具組」

Oracle 推出的 JDK（以及 OpenJDK）內含許多好用的 Java 開發工具，其中大家最熟知的就是 Java 直譯器和 Java 編譯器。本章不僅記錄了這些工具，還會介紹 jshell 互動式環境，以及用於處理 Java 模組化的新工具。

附錄 *A*「*Java 17* 的後續版本」

附錄會介紹 Java 17 以後的版本內容，包括 Java 18 和 Java 19 持續進行的研究和開發專案，致力於強化 Java 語言和 JVM。

碁峰 www.gotop.com.tw

Java 只有八種不同的基本資料型態，程式設計師無法自行定義新的基本資料型態。

基本資料型態和參考型態之間的關鍵差異在於，基本資料型態的值不可改變；以數字 2 這個值為例，永遠都會是相同的值。相較之下，參考型態所指向的物件內容通常可以改變，這種程式設計特性一般稱為物件內容的變異（mutation）。

另外要注意的程式設計特性是，變數只能儲存適當型態的值。尤其是參考型態的變數值，其所儲存的引用位址一定會是指向該物件的記憶體位置，而不會直接包含該物件的內容。這也說明了為什麼 Java 沒有支援類似解引用運算子（dereference operator）或這類結構（`struct`）的概念。

Java 的設計目的是想簡化一個 C++ 程式設計師常常會混淆的概念：「物件內容」和「物件引用」之間的差異。不幸的是，Java 無法完全隱藏兩者之間的差異，所以程式設計師仍然必須了解 Java 平台中參考型態值的運作機制。

Java 是「傳址」語言嗎？

Java 是「透過引用位址」處理物件，但不能將這個做法跟「傳址」（pass by reference）搞混，後面這個術語是用來描述各種程式語言的方法呼叫慣例。採用「傳址」的程式語言不能直接將值傳送給方法，甚至連基本資料型態的值也不行。反過來說，方法永遠都是接收值的引用位址。因此，如果在方法內部修改自身參數，則方法回傳時也會反映出這些參數的修改，即使是基本資料型態也是一樣。

Java 語言並非採用「傳址」，而是一種「傳值」（pass by value）語言。然而，涉及參考型態時，傳送的值實際上是引用位址的副本（作為值），不同於「傳址」。若假設 Java 是「傳址」語言，當參考型態傳送給某個方法時，就會作為引用位址的引用位址來傳送。

碁峰 www.gotop.com.tw

執行以下這個簡單的程式碼，能證明 Java 是「傳值」語言這個事實：

```
public void manipulate(Circle circle) {
    circle = new Circle(3);
}

Circle c = new Circle(2);
System.out.println("Radius: "+ c.getRadius());
manipulate(c);
System.out.println("Radius: "+ c.getRadius());
```

在這段範例程式碼裡，兩次的輸出結果都是「`Radius: 2`」，顯示即使在呼叫 `manipulate()` 方法之後，變數 `c` 擁有的值仍然維持不變，其所儲存的引用位址依舊指向半徑為 2 的 `Circle` 物件。如果 Java 真的是傳址語言，變數儲存的引用位址就應該改為指向半徑為 3 的 `Circle` 物件。

如果我們非常嚴格區分這些概念，將物件引用視為一種 Java 可能使用的值，則一些 Java 原本看似令人驚訝的特性顯然也就變得容易理解了。各位讀者要特別小心！一些比較舊的文章在解釋這個概念時的說明很模糊。後續第 6 章討論記憶體和垃圾回收機制時，會再次討論 Java 值的概念。

重要而且常用的方法

先前我們就已經提過，所有類別都會直接或間接繼承 `java.lang.Object` 類別。這個類別定義了許多實用的方法，其中一些方法就是設計成讓我們撰寫的類別可以覆寫。範例 5-1 展示的類別有覆寫其中一些方法，接下來的章節將以這個範例來說明每個方法的預設實作內容，解釋我們為什麼會想覆寫這些方法。

請注意，下方程式碼的寫法僅針對示範目的；實務上會以 Record 類別來表示 `Circle` 這種類別，許多這一類的方法會由編譯器自動實作。

範例 5-1　示範類別如何覆寫重要的 Object 方法

```
// 這個類別能表示一個圓，位置和半徑都不能改變
public class Circle implements Comparable<Circle> {
    // 這些欄位是用來儲存圓心座標和半徑
    // 針對資料封裝目的而設為私有欄位，final 是設定欄位值具有不可變異性
    private final int x, y, r;
```


碁峰 www.gotop.com.tw

請記住：一個類別如果繼承了某個抽象類別，就不能再繼承任何其他類別，而且，介面依舊不能包含非常數的欄位。表示我們在 Java 程式中使用繼承時，仍需考慮一些限制。

介面和抽象類別之間另一個重要的差異在於向下相容性。如果在公開 API 中定義介面，然後為該介面新增強制實作性方法，這會破壞所有實作舊版介面的類別；換句話說，任何新的介面方法都必須宣告為預設方法，並且提供實作程式碼。

然而，使用抽象類別時，有安全的方式可以為該類別新增非抽象方法，不需要修改任何已經繼承該抽象類別的現有類別。

在介面和抽象類別這兩種情況下，新增的方法都可能會與子類別中同名而且簽名格式相同的方法產生衝突——此時永遠會優先執行子類別的方法。基於這個原因，新增方法時要審慎考慮，尤其是當方法名稱對該型態來說「顯然很容易聯想到」，或方法名稱可能有多重含義時。

一般而言，需要設計 API 規格時，建議各位優先使用介面。介面的強制實作性方法應該是非預設方法，因為這些方法代表 API 的部分功能，所以必須為這些方法提供有效的實作。預設方法只用於兩種情況：該方法確實可以選擇性實作，或是該方法只需要一種實作方式。

最後還要介紹一個 Java 8 以前的舊技術，這個做法是在文件裡宣告介面中哪些方法會視為「選擇性實作」，並且指示實作方式：如果程式設計師不想實作這些方法，會拋出 `java.lang.UnsupportedOperationException` 例外。這種做法存在許多問題，不應該在後來新開發的程式碼中使用。

預設方法會改變 Java 繼承模式嗎？

Java 8 發布以前，Java 語言很明顯只允許單一繼承模式。每一個類別（除了 `Object` 類別以外）都只能直接繼承一個父類別，方法的實作內容只會定義在類別裡，或是從父類別階層繼承。

Java 8 導入預設方法改變了這個情況，因為預設方法允許從多個來源繼承方法實作——從父類別階層或介面提供的預設實作。當不同預設方法來自不同介面，這些方法之間可能會存在任何衝突，導致編譯時期發生錯誤。

碁峰 www.gotop.com.tw

也就是說繼承多個實作來源時，不可能發生衝突，因為只要發生衝突，Java 會要求程式設計師手動消除造成歧義的方法。

當然，狀態也不會有多重繼承，因為：介面依舊沒有非常數的欄位。

這表示 Java 的多重繼承和其他程式語言（例如 C++）認知的一般多重繼承不同。對熟悉 C++ 語言的讀者來說，Java 的預設方法其實就是 C++ 的 *Mixin* 模式。有些開發人員也會將預設成員視為某些物件導向語言（例如 Scala）中，以某種形式提供的**特徵**語言特性。

雖然 Java 語言設計者代表的官方立場是，預設方法還不足以成為完整的特徵（trait）。但某種程度來說，JDK 內部的程式碼讓這種觀點站不住腳，因為就連 `java.util.function` 套件內的介面都表現得像是簡單的特徵，例如 `Function` 介面本身。

請思考以下這個範例：

```
public interface IntFunc {
    int apply(int x);

    default IntFunc compose(IntFunc before) {
        return (int y) -> apply(before.apply(y));
    }

    default IntFunc andThen(IntFunc after) {
        return (int z) -> after.apply(apply(z));
    }

    static IntFunc id() {
        return x -> x;
    }
}
```

這個範例是簡化版的 `Function` 介面（來自 `java.util.function` 套件），移除泛型而且 `int` 只能作為資料型態處理。

這個例子還展示了一個重要的觀點：現有的函式組合方法（`compose()` 和 `andThen()`）只會以標準方式組合這些函式，而且覆寫 `compose()` 方法預設的實作內容，幾乎不可能有合理的結果。

碁峰 www.gotop.com.tw

Java 的特性之一是我們不用記得要手動銷毀自己建立的物件，這讓程式設計變得更輕鬆愉快。相較於那些沒有支援自動垃圾回收機制的程式語言，這項特性也讓 Java 語言撰寫的程式比較不容易出錯。

不同的虛擬機器實作對垃圾回收機制有不同的處理方式，而且 Java 規範並沒有嚴格限制垃圾回收機制的實作方式。本章稍後會討論 HotSpot JVM，這是 Oracle 和 OpenJDK 這兩種 Java 實作的基礎。各位讀者之後可能會遇到其他的 JVM，但是在伺服器端部署中，HotSpot 是最常見的 JVM，而且為現代生產環境 JVM 提供一個參考範例。

Java 記憶體流失

Java 支援垃圾回收機制這項特性，大幅降低了**記憶體流失**（memory leak）發生的機率。當系統分配出去的記憶體空間一直無法回收再利用時，此時就會發生記憶體流失。因為垃圾回收機制會自動回收所有未使用的物件，表面上看起來，似乎能防止所有記憶體流失的問題。

不過，只要系統中仍然殘留有效（但未使用）的引用位址，Java 還是有可能會發生記憶體流失的情況。舉例來說，當某個方法長時間運行或永不結束時，該方法中的區域變數可能會一直保留某些物件的引用位址，即使這些物件其實早就不需要使用。以下方程式碼來解說這個情況：

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

  // 使用 bigArray 陣列進行一些計算並且獲得計算結果
  int result = compute(bigArray);

  // 當我們不再需要 bigArray 陣列
  // 沒有任何引用位址指向這個陣列時，垃圾回收機制就會自動回收這個陣列
  // 由於 bigArray 是區域變數，所以會一直指向陣列，直到方法回傳結果
  // 但是這個方法永遠不會回傳
  // 如果直接將引用位址指定為 null
  // 垃圾回收器就會知道可以回收這個陣列
  bigArray = null;

  // 這個方法會無限循環處理使用者輸入的內容
  for(;;) handle_input(result);
}
```


碁峰 www.gotop.com.tw

使用 HashMap 等類似資料結構來建立物件之間的關聯時，也可能造成記憶體無法正確釋放的問題。即使兩個物件都不再需要使用，物件之間的關聯性仍然存在於 Map 集合裡，要等到 Map 集合本身回收之後，才能回收這些物件。如果 Map 集合的生命週期遠超出集合內的物件，就會造成記憶體流失。

「標記／清除」概念簡介

Java 提供的 GC 機制通常是依賴一套普遍稱為「**標記／清除**」（mark-and-sweep）的演算法。為了理解這些演算法，請各位回想一下先前提過的內容，所有的 Java 物件都是在堆積記憶體中建立，產生物件時，這些物件的引用位址（基本上就是指標）會儲存在 Java 的區域變數（或欄位）。區域變數存活於方法的堆疊框架，如果從方法回傳物件，該物件的引用位址會在方法退出時傳送回去給呼叫者的堆疊框架。

當所有物件都配置在堆積記憶體裡，而堆積記憶體爆滿時（或是快滿的時候，取決於具體情況）就會觸發 GC 機制。標記／清除的基本概念是**追蹤**堆積記憶體，辨識出哪些物件仍然在使用中。做法是透過檢查每個 Java 執行緒的堆疊框架（以及少數幾個其他引用來源），並且追蹤堆積記憶體中的任何引用。每個被定位到的物件都會被**標記**為仍然存活，然後檢查物件是否有任何欄位是參考型態。如果有，就會追蹤這些引用位址並且標記。

當遞迴追蹤活動完成後，而且確定所有剩餘未標記的物件都不再需要，這些物件占用的堆積記憶體空間會被當作垃圾**清除**，亦即這些物件使用的記憶體會被回收，分配給後續的物件使用。如果可以精準執行這項分析，毫無意外地，這種類型的回收器應該會被稱為**精準式垃圾回收器**（exact garbage collector）。就所有實際目的而言，Java 的垃圾回收器都可以視為精準式，但在其他軟體環境中可能並非如此。

在實際運行的 JVM 中，非常有可能存在不同的堆積記憶體區域，實際運行的程式在正常操作情況下會使用所有這些區域。圖 6-1 展示一種可能存在的堆積記憶體配置，其中兩個執行緒（T1 和 T2）擁有的引用位址指向堆積記憶體。

碁峰 www.gotop.com.tw

這些不同的記憶體區域稱為 *Eden* 區、*Survivor* 區和 *Tenured* 區，本章稍後會介紹這些區域，了解它們彼此之間的關係。為了簡化說明，圖中顯示一種較舊的 Java 堆積記憶體形式，其中每個記憶體區域都是一塊單獨的記憶體。現代的垃圾回收器實際上不會以這種方式配置物件，但先以這種方式思考會更容易理解！

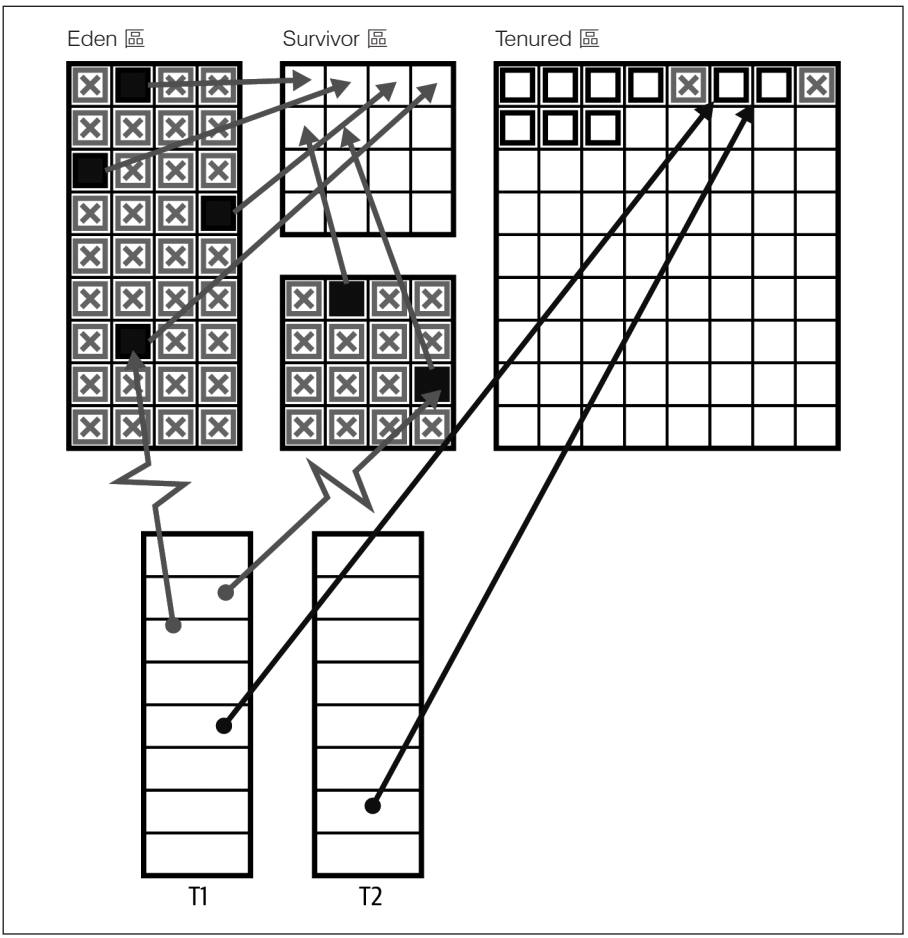

圖 6-1　堆積記憶體的結構

圖 6-1 中還顯示出，在程式運行過程中，移動應用程式執行緒引用的物件是很危險的操作。

碁峰 www.gotop.com.tw

為了避免發生這種情況，方才提過的簡單追蹤式 GC 機制會在運行時引發**暫停世界機制**（stop-the-world，簡稱 STW）。這套機制之所以有效，原因在於所有應用程式執行緒都會被迫停止，然後進行垃圾回收，最後再重新啟動應用程式執行緒。執行環境會針對這個情況進行特別的處理，當應用程式執行緒到達**安全點**（safepoint）就停止執行緒，例如在迴圈開始或方法呼叫即將回傳之前。執行環境知道可以在這些執行點停止應用程式執行緒，而不會造成問題。

這些暫停機制有時會令開發人員感到擔憂，但是對多數主流用途來說，Java 是在作業系統之上執行（可能還有多個虛擬層），而作業系統本身經常會在處理器核心上切換程序，所以這種短暫的額外停頓通常不是問題。以 HotSpot 的情況來看，已經進行了大量的工作來為垃圾回收機制最佳化，並且減少 STW 時間，尤其是那些應用程式工作負載很重要的情況，下一節會討論其中幾個最佳化策略。

JVM 如何最佳化垃圾回收機制

弱世代假設（weak generational hypothesis，簡稱 WGH）是一個絕佳的例子，呼應我們在第 1 章介紹過跟軟體執行環境有關的幾項事實。簡單來說，這項假設指出物件通常只有少數幾種可能的生命週期長度（也稱為**世代**）。

物件通常只會存活非常短的時間（有時稱為臨時物件），之後就會成為可進行垃圾回收的物件。然而，有一小部分物件會存活較長的時間，而且最後會成為程式長期狀態的一部分（有時也稱為**工作集**）。從圖 6-2 可以看到，該圖展示了記憶體用量（或創建的物件數量）與預期生命週期之間的關係。

碁峰 www.gotop.com.tw

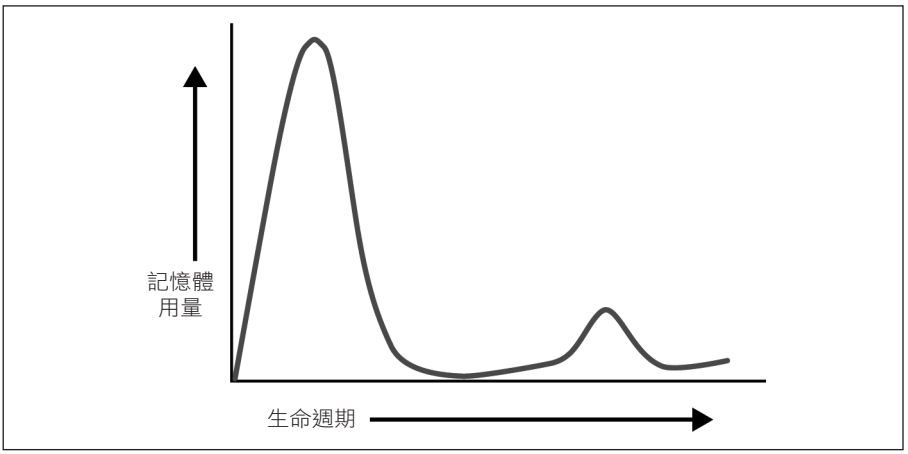

圖 6-2　弱世代假設

程式的靜態分析無法推論出這個現象，但是當我們衡量軟體運行時的行為，會在各種工作負載中發現這是普遍存在的情況。

HotSpot JVM 擁有特別設計的垃圾回收子系統，其設計目的是充分利用弱世代假設的優勢，本節將討論這些技巧如何應用在短期存活物件（這是最常見的情況）。此處討論的內容可以直接應用在 HotSpot，但其他 JVM 通常也會採用類似或相關的技術。

世代型垃圾回收器（generational garbage collector）最基本的概念是特別考量弱世代假設。這些設計者的立場是，為了監控記憶體而進行的一些額外記錄工作，其所付出的成本卻遠遠超出採用弱世代假設所得到的收益。最簡單的世代型回收器通常只有兩個世代——一般稱為新生代與古生代。

撤離

標記／清除演算法原本的形式是在清除階段時，GC 機制會回收個別物件以供重新使用。截至目前為止，這種方式運作正常，但是會導致一些問題，例如記憶體碎片化，以及 GC 機制需要維護一份記憶體區塊的「閒置列表」。然而，如果弱世代假設成立，也就是說在任何已知的 GC 週期中，多數物件都會死亡，則使用另一種方法來回收記憶體空間可能更有意義。

碁峰 www.gotop.com.tw

這種方法的運作原理是將堆積記憶體分割成獨立的記憶體空間，讓新的物件建立於稱為 *Eden* 的空間。然後在每次執行垃圾回收時，只會定位存活物件，將這些物件移動到不同的空間，這個過程就稱為「**撤離**」（evacuation）。執行這套流程的垃圾回收器就稱為**撤離式回收器**，其特性是回收結束時會清除整個記憶體空間，以便不斷重新使用。

圖 6-3 展示正在運作的撤離式回收器，其中實心區塊代表存活物件，打叉陰影區域代表已經配置但現在死亡（而且無法抵達）的物件。

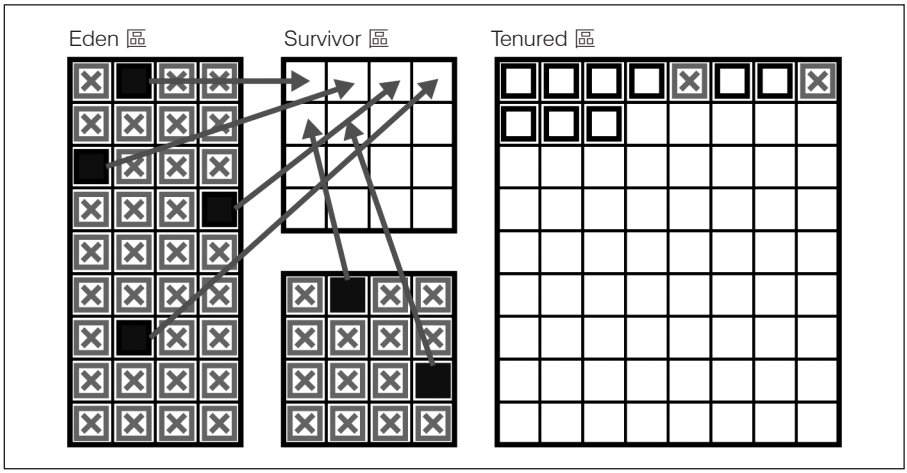

圖 6-3　撤離式回收器

相較於簡單的回收方法，這種做法可能更有效率，因為完全不會接觸到死亡物件。也就是說垃圾回收時間會跟存活物件的數量成正比，而非已配置的物件的數量。唯一的缺點是會需要多一點的記錄工作——必須付出複製存活物件的成本，但是相較於撤離策略帶來的龐大效益，幾乎可以說是非常小的代價。

使用撤離式回收器也允許每個執行緒單獨配置記憶體，表示每個應用程式使用的執行緒都可以獲得一塊連續記憶體，這塊記憶體區域稱為**本機執行緒配置的緩衝區**（thread-local allocation buffer，簡稱 TLAB），專門用來配置新物件。配置新物件時，只需要在配置緩衝區中推進指標，這是一個成本極低的操作。

碁峰 www.gotop.com.tw

圖 11-1 顯示 `Object` 類別遞移閉包的一部分。

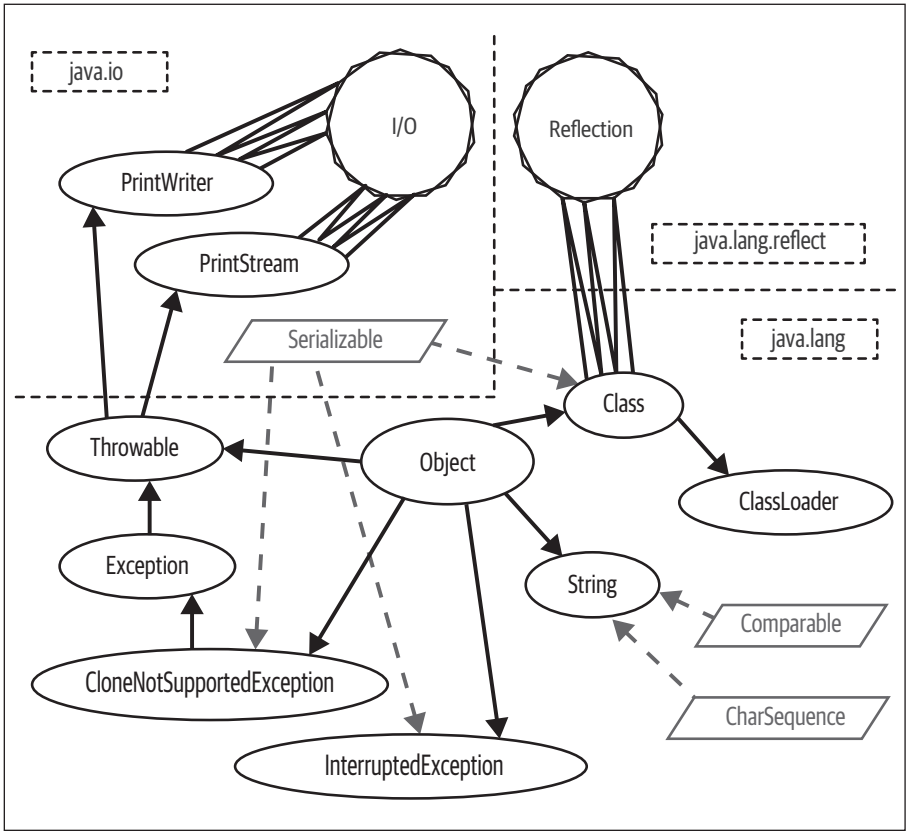

圖 11-1　Java 型態的遞移閉包

初始化

解析完成後，JVM 終於可以開始進行類別初始化。這個階段會對靜態變數進行初始化，執行靜態初始化程式區塊。

這是 JVM 首次執行剛載入的類別位元組碼，靜態程式區塊執行完成後，該類別就算是完全載入，準備就緒。

碁峰 www.gotop.com.tw