THE COMPLETE JAVA PROGRAMMING COURSE

Java 程式設計
完整學習教材

進階篇 ADVANCED LEVEL

函數式程式設計與 Stream API、多執行緒與並行工具、反射與註解、JVM 內部機制與效能調校、網路程式設計與 JDBC 資料庫存取，以及設計模式與現代 Java 特性，帶你邁向資深工程師之路。

以 Java 17 LTS 為基準・涵蓋 Java 21 重要新特性・所有範例皆附詳細中文註解・編譯日期：2026 年 8 月 14 日

如何使用本教材

《Java 程式設計完整學習教材》共分三冊，對應三個學習階段：初級篇、中級篇與進階篇。每一章都包含觀念講解、大量附詳細中文註解的範例程式、預期執行結果、常見錯誤提醒，以及章末的動手練習。三冊循序漸進，完整走完即具備從零基礎到資深工程師所需的 Java 核心知識。

三冊內容總覽

冊別	適合對象	主要內容
初級篇	沒有程式設計經驗的初學者	開發環境、變數與資料型別、運算子、流程控制、陣列、方法、字串處理、物件導向入門
中級篇	已掌握 Java 基礎語法的學習者	封裝、繼承與多型、介面、例外處理、集合框架、泛型、檔案與 I/O、列舉與內部類別、綜合實戰專案
進階篇	邁向資深工程師的開發者	Lambda 與函數式程式設計、Stream API、多執行緒與並行、反射與註解、JVM 原理、網路與 JDBC、設計模式與現代 Java

閱讀慣例

內文中的程式碼片段會以等寬字型呈現，例如 `System.out.println()` 與 `List<String>`。完整的範例程式則放在如下的程式碼框中，每個範例都盡可能加上詳細的中文註解——註解正是本教材的精華所在：

範例：Hello.java ——本教材的程式碼呈現方式

```
/**
 * 教材中的每個範例都會像這樣：
 * 用文件註解說明類別的用途，用行內註解解釋每個步驟的意圖。
 */
public class Hello {
    public static void main(String[] args) {
        // 將訊息輸出到終端機（主控台）
        System.out.println("你好，Java！");
```

```
    }
}
```

▶ 執行結果

```
你好，Java！
```

此外，教材中穿插四種資訊框，各有不同的用途：

> **提示**　提供實用的小技巧與業界最佳實務，幫助你寫出更專業的程式。

> **注意**　標示初學者常犯的錯誤與容易踩到的陷阱，閱讀時請特別留意。

> **說明**　補充延伸知識與底層原理說明，第一次閱讀時可以先略過，回頭再讀。

> **動手練習**
>
> 每章最後都有動手練習。程式設計是「做中學」的技能，請務必實際完成。

環境需求

- **JDK 17 或更新版本**：本教材以 Java 17 LTS 為基準，並在適當章節介紹 Java 21 的重要新特性（如虛擬執行緒、switch 模式匹配）。
- **文字編輯器或 IDE**：初期建議用命令列搭配任一編輯器，熟悉編譯流程後再改用 IntelliJ IDEA 或 VS Code 提升效率。
- **終端機**：用來執行 `javac`（編譯）與 `java`（執行）指令。

學習方法建議

1. **每個範例都親手輸入並執行**：不要複製貼上，打字的過程就是在建立語法的肌肉記憶。
2. **先讀註解、再讀程式**：範例中的註解解釋了「為什麼這樣寫」，而不只是「寫了什麼」。
3. **改壞它，再修好它**：試著修改範例、觀察編譯器的錯誤訊息，這是理解 Java 編譯器脾氣的最快方法。
4. **完成章末練習**：練習題附有提示，請先自行嘗試，卡住了再看提示。

目錄

如何使用本教材 **1**

1 Lambda 與函數式程式設計 **7**

1.1 從匿名類別到 Lambda 7

1.2 Lambda 語法與函數式介面 10

1.3 內建核心函數式介面 11

1.4 方法參考 15

1.5 變數捕獲與 effectively final 16

1.6 Comparator 的函數式組合 17

1.7 Optional：優雅處理不存在的值 18

1.8 綜合範例：可組合的資料驗證器 21

1.9 本章小結 22

1.10 本章練習 23

2 Stream API **24**

2.1 命令式與宣告式的對比 24

2.2 Stream 的結構與特性 26

2.3 建立 Stream 26

2.4 中間操作 28

2.5 終端操作 30

2.6 collect 與 Collectors 深入 32

2.7 基本型別流 34

2.8 平行流 35

2.9 綜合實戰：員工資料多維度分析 36

2.10 本章小結 38

2.11 本章練習 38

3 多執行緒基礎 **40**

3.1 程序與執行緒 40

3.2 建立執行緒的三種方式 . . . 41
3.3 執行緒的生命週期與狀態 . . . 42
3.4 執行緒的常用方法 . . . 44
3.5 中斷機制 . . . 45
3.6 競態條件 . . . 47
3.7 synchronized：互斥同步 . . . 48
3.8 volatile：可見性 . . . 49
3.9 wait/notify 與生產者消費者 . . . 50
3.10 死結 . . . 52
3.11 執行緒安全等級盤點 . . . 53
3.12 本章小結 . . . 54
3.13 本章練習 . . . 54

4 並行程式設計進階 56
4.1 手動建立執行緒的問題 . . . 56
4.2 Executor 框架 . . . 56
4.3 ThreadPoolExecutor 七參數 . . . 58
4.4 Callable 與 Future . . . 59
4.5 CompletableFuture 非同步管線 . . . 60
4.6 原子類別 . . . 63
4.7 顯式鎖 . . . 64
4.8 並行集合 . . . 66
4.9 同步輔助工具 . . . 68
4.10 虛擬執行緒（Java 21） . . . 69
4.11 本章小結 . . . 71
4.12 本章練習 . . . 71

5 反射與標註 72
5.1 反射是什麼 . . . 72
5.2 Class 物件：反射的入口 . . . 72
5.3 檢視類別結構 . . . 73
5.4 動態建立與呼叫 . . . 75

5.5 反射的成本 76
5.6 標註（Annotation） 76
5.7 綜合實戰一：迷你測試框架 79
5.8 綜合實戰二：簡易依賴注入 81
5.9 本章小結 82
5.10 本章練習 83

6 JVM 深入剖析 84

6.1 JVM 整體架構 84
6.2 類別載入過程 84
6.3 類別載入器與雙親委派 86
6.4 執行期資料區 87
6.5 物件的建立與記憶體佈局 89
6.6 垃圾回收基礎 89
6.7 診斷工具實戰 92
6.8 記憶體洩漏經典案例 92
6.9 本章小結 94
6.10 本章練習 94

7 網路程式設計與 JDBC 95

7.1 網路基礎速覽 95
7.2 TCP Socket 程式設計 96
7.3 從單執行緒到執行緒池伺服器 98
7.4 群聊伺服器實戰 100
7.5 UDP 收發 101
7.6 HTTP 用戶端 103
7.7 JDBC：Java 資料庫連接 104
7.8 本章小結 110
7.9 本章練習 110

8 設計模式與現代 Java 112

8.1 設計原則速覽：SOLID 112
8.2 單例模式 114

8.3 工廠模式 115
8.4 建造者模式 117
8.5 策略模式 118
8.6 觀察者模式 119
8.7 裝飾者模式 120
8.8 現代 Java 特性 121
8.9 壓軸綜合範例：運算式求值器 123
8.10 全系列結語：後續的學習地圖 124
8.11 本章小結 125
8.12 本章練習 125

第 1 章

Lambda 與函數式程式設計

> **本章學習目標**
>
> 學完本章你將能夠：
>
> - 說明 Lambda 運算式如何從匿名類別演進而來，並流暢運用各種語法簡寫。
> - 定義自己的函數式介面，並熟練使用 `Function`、`Predicate`、`Consumer`、`Supplier` 等內建介面。
> - 正確使用四種方法參考，寫出更精簡的程式碼。
> - 理解變數捕獲與 effectively final 規則，避開常見陷阱。
> - 運用 `Comparator` 的函數式組合完成多鍵排序。
> - 用 `Optional` 優雅地處理可能不存在的值，並辨識其反模式。

從本章開始，我們正式邁入資深工程師的領域。你在初級與中級篇已經熟悉物件導向的思維：把資料與行為封裝進物件。而函數式程式設計提供了另一種互補的視角——把「行為」本身當成值來傳遞。Java 8 引入的 Lambda 運算式與函數式介面，讓這種風格得以在強型別的語言中優雅落地，也是後續 Stream API（第 2 章）與 `CompletableFuture`（第 4 章）的語法基礎。

本章的核心觀念只有一個：**把程式碼當成資料**。過去我們只能把數字、字串、物件傳給方法；現在我們可以把「一段運算邏輯」也傳進去。掌握這個觀念，你看待 API 設計的角度會徹底改變。

1.1 從匿名類別到 Lambda

在 Java 8 之前，若要把「一段行為」傳給方法，唯一的辦法是包成一個物件。以最常見的 `Runnable`（代表一段無參數、無回傳的工作）為例，我們先看同一個需求的三種寫法，感受語法如何一步步瘦身。

1.1.1 第一種：具名類別

最原始的做法是為這段行為專門宣告一個類別，實作介面後再建立實體傳入：

範例：具名類別實作 Runnable

```
1  /**
```

```
 * 把「要執行的工作」包成一個具名類別。
 * 缺點：為了一段簡單邏輯，卻要多寫一個完整類別，非常囉嗦。
 */
public class NamedClassDemo {

    // 實作 Runnable 介面的具名類別
    static class PrintTask implements Runnable {
        @Override
        public void run() {                 // 唯一要實作的抽象方法
            System.out.println("以具名類別執行工作");
        }
    }

    public static void main(String[] args) {
        Runnable task = new PrintTask();    // 先建立物件
        task.run();                         // 再呼叫其行為
    }
}
```

▶ 執行結果

```
以具名類別執行工作
```

1.1.2 第二種：匿名類別

如果這個類別只用一次，替它命名並無意義。匿名類別讓我們「宣告類別」與「建立物件」一步完成，省去命名：

範例：匿名類別實作 Runnable

```
/**
 * 匿名類別：就地實作介面並建立物件，不必替類別命名。
 * 缺點：new Runnable、@Override、run() 這些樣板仍然佔滿版面。
 */
public class AnonymousClassDemo {
    public static void main(String[] args) {
        // 一次完成「實作介面」與「建立物件」
        Runnable task = new Runnable() {
            @Override
            public void run() {
                System.out.println("以匿名類別執行工作");
            }
```

```
        };
        task.run();
    }
}
```

▶ 執行結果

```
以匿名類別執行工作
```

1.1.3 第三種：Lambda 運算式

觀察上面的匿名類別：`Runnable` 只有一個抽象方法 `run`，所以「要實作哪個方法」根本不必再說一次。編譯器完全能從變數型別推斷出來。Lambda 就是把這些冗餘全部拿掉，只留下真正重要的**參數列**與**方法主體**：

範例：Lambda 實作 Runnable

```
/**
 * Lambda 運算式：只保留參數列與主體，其餘由編譯器推斷。
 * run 沒有參數，所以左側是一對空括號。
 */
public class LambdaDemo {
    public static void main(String[] args) {
        // () 對應 run 的空參數列，-> 右側是方法主體
        Runnable task = () -> System.out.println("以 Lambda 執行工作");
        task.run();
    }
}
```

▶ 執行結果

```
以 Lambda 執行工作
```

三種寫法產生的效果完全相同，但程式碼從十餘行縮到一行。要特別澄清一個常見誤解：**Lambda 不只是匿名類別的語法糖**。它們在底層的實作機制不同——匿名類別會產生獨立的 `.class` 檔，而 Lambda 透過 `invokedynamic` 指令在執行期動態產生實作，通常更輕量。

> **說明**　Lambda 能取代匿名類別的前提是：該介面**只有一個抽象方法**。這種介面稱為「函數式介面」（functional interface）。`Runnable`、`Comparator`、`Callable` 都符合，所以都能用 Lambda 表達；但若介面有兩個以上的抽象方法，編譯器就無從推斷你要實作哪一個，只能回頭用匿名類別。

1.2 Lambda 語法與函數式介面

1.2.1 Lambda 的語法變形

Lambda 的完整形式是「(參數列) -> { 敘述句 }」，但依情況可層層簡化。下表整理各種合法的簡寫，請對照記憶：

簡寫規則	範例
完整形式	`(int a, int b) -> { return a + b; }`
省略參數型別（可推斷）	`(a, b) -> { return a + b; }`
單一運算式省略大括號與 return	`(a, b) -> a + b`
單一參數可省略括號	`s -> s.length()`
無參數必須保留空括號	`() -> 42`

要注意兩個容易犯錯的地方：其一，只有單一運算式時才能同時省略大括號與 `return`；一旦寫了大括號，就必須明確 `return`（若有回傳值）。其二，參數型別要嘛全部省略、要嘛全部寫出，不能只寫一半。

> **注意**　下面兩種寫法都會編譯失敗：`(a, b) -> { a + b; }`（有大括號卻沒有 `return`，而且 `a + b` 單獨成句不合法），以及 `(int a, b) -> a + b`（型別只寫了一半）。修正方式分別是加上 `return` 與統一省略或補齊型別。

1.2.2 自訂函數式介面與 @FunctionalInterface

任何「只有一個抽象方法」的介面都能作為 Lambda 的目標型別。我們可以自己定義，並加上 `@FunctionalInterface` 標註。這個標註不是必要的，但強烈建議加上: 它會請編譯器檢查此介面是否確實只有一個抽象方法，避免日後有人不小心多加方法而破壞了它的函數式特性。

範例：CalculatorDemo.java ——自訂函數式介面

```
/**
 * 自訂函數式介面：代表「對兩個整數做某種運算」。
 * @FunctionalInterface 讓編譯器強制檢查：有且僅有一個抽象方法。
 */
@FunctionalInterface
interface Calculator {
    int apply(int a, int b);            // 唯一的抽象方法

    // 函數式介面「可以」有預設方法與靜態方法，不影響其函數式本質
    default Calculator andThenNegate() {
        // 先做原運算，再把結果取負號；回傳一個新的 Calculator
```

```
        return (a, b) -> -this.apply(a, b);
    }
}

public class CalculatorDemo {
    public static void main(String[] args) {
        // 同一個介面，用不同 Lambda 表達不同運算邏輯
        Calculator add = (a, b) -> a + b;
        Calculator multiply = (a, b) -> a * b;

        System.out.println("3 + 5 = " + add.apply(3, 5));
        System.out.println("3 * 5 = " + multiply.apply(3, 5));
        // 呼叫預設方法，得到「相乘後取負」的新運算
        System.out.println("-(3 * 5) = "
                + multiply.andThenNegate().apply(3, 5));
    }
}
```

▶ 執行結果

```
3 + 5 = 8
3 * 5 = 15
-(3 * 5) = -15
```

這個範例點出一個關鍵：Lambda 的型別不是「Lambda 型別」，而是它所賦值的那個 **函數式介面型別**。同樣一段 `(a, b) -> a + b`，賦值給 `Calculator` 時就是 `Calculator`，這稱為「目標型別推斷」。

1.3　內建核心函數式介面

實務上我們很少自己定義函數式介面，因為 `java.util.function` 套件已備妥一整組通用介面。下表是四大家族，務必記熟它們的抽象方法名稱：

介面	抽象方法	語意	典型用途
`Function<T,R>`	`R apply(T)`	輸入 T 產出 R	資料轉換
`Predicate<T>`	`boolean test(T)`	條件判斷	過濾
`Consumer<T>`	`void accept(T)`	消費不回傳	印出、寫入
`Supplier<T>`	`T get()`	無輸入產出 T	延遲產生值

1.3.1 Function：轉換與組合

`Function<T,R>` 把型別 `T` 的輸入轉成型別 `R` 的輸出。它最強大的地方是可以串接：`andThen` 是「先做自己再做參數」，`compose` 則相反。

範例：FunctionDemo.java ——apply / andThen / compose

```
import java.util.function.Function;

/**
 * 示範 Function 的轉換與兩種組合方向。
 * andThen(g)：先 this 再 g；compose(g)：先 g 再 this。
 */
public class FunctionDemo {
    public static void main(String[] args) {
        Function<Integer, Integer> timesTwo = x -> x * 2;  // 乘以 2
        Function<Integer, Integer> plusTen = x -> x + 10;  // 加 10

        // 直接套用
        System.out.println("timesTwo(5) = " + timesTwo.apply(5));

        // andThen：先乘 2（5->10）再加 10（10->20）
        Function<Integer, Integer> f1 = timesTwo.andThen(plusTen);
        System.out.println("先乘2再加10：" + f1.apply(5));

        // compose：先加 10（5->15）再乘 2（15->30）
        Function<Integer, Integer> f2 = timesTwo.compose(plusTen);
        System.out.println("先加10再乘2：" + f2.apply(5));
    }
}
```

▶ 執行結果

```
timesTwo(5) = 10
先乘2再加10：20
先加10再乘2：30
```

若運算需要兩個輸入，改用 `BiFunction<T,U,R>`，抽象方法為 `R apply(T, U)`，例如 `(a, b) -> a + b` 可賦值給 `BiFunction<Integer,Integer,Integer>`。

1.3.2 Predicate：條件判斷與邏輯組合

`Predicate<T>` 代表一個回傳布林值的判斷。它的 `and`、`or`、`negate` 讓我們像堆積木一樣組合複雜條件：

範例：PredicateDemo.java ——test / and / or / negate

```
import java.util.function.Predicate;

/**
 * 用 Predicate 表達條件，並用 and/or/negate 組合。
 */
public class PredicateDemo {
    public static void main(String[] args) {
        Predicate<Integer> isPositive = n -> n > 0;   // 是否為正
        Predicate<Integer> isEven = n -> n % 2 == 0;  // 是否為偶

        // and：兩者皆成立
        Predicate<Integer> positiveEven = isPositive.and(isEven);
        // or：任一成立；negate：反向
        Predicate<Integer> positiveOrEven = isPositive.or(isEven);
        Predicate<Integer> notPositive = isPositive.negate();

        System.out.println("6 是正偶數？" + positiveEven.test(6));
        System.out.println("-4 是正數或偶數？" + positiveOrEven.test(-4));
        System.out.println("-4 非正數？" + notPositive.test(-4));
    }
}
```

▶ 執行結果

```
6 是正偶數？true
-4 是正數或偶數？true
-4 非正數？true
```

1.3.3 Consumer 與 Supplier

`Consumer<T>` 接收一個值卻不回傳（例如印出），也支援 `andThen` 依序執行多個動作；`Supplier<T>` 則相反，不接收輸入卻能產出一個值，常用於「延遲產生」的場景。

範例：ConsumerSupplierDemo.java

```
import java.util.function.Consumer;
```

```java
import java.util.function.Supplier;

/**
 * Consumer 消費值、Supplier 供應值。
 * Supplier 的「延遲」特性在需要昂貴運算時特別有用。
 */
public class ConsumerSupplierDemo {
    public static void main(String[] args) {
        // Consumer：接收字串並印出；andThen 串接第二個動作
        Consumer<String> print = s -> System.out.println("內容：" + s);
        Consumer<String> printLen =
                s -> System.out.println("長度：" + s.length());
        print.andThen(printLen).accept("Java");

        // Supplier：呼叫 get() 時才真正產生值（此處為目前時間毫秒）
        Supplier<Long> now = () -> System.currentTimeMillis();
        System.out.println("時間戳記非空？" + (now.get() > 0));
    }
}
```

▶ 執行結果

```
內容：Java
長度：4
時間戳記非空？true
```

1.3.4 UnaryOperator 與基本型別特化

當「輸入與輸出型別相同」時，可用 `UnaryOperator<T>`（單元）與 `BinaryOperator<T>`（二元），它們分別是 `Function<T,T>` 與 `BiFunction<T,T,T>` 的特例，語意更清楚。此外，為避免自動裝箱的效能開銷，標準庫還提供基本型別特化版本，如 `IntFunction<R>`、`ToIntFunction<T>`、`IntPredicate`、`IntUnaryOperator` 等。處理大量數值時，優先選用這些版本能省下大量 `Integer` 物件的建立。

> **提示**　如何選介面？先問三個問題：有沒有輸入？有沒有輸出？輸入輸出型別一不一樣？無輸入有輸出選 `Supplier`；有輸入無輸出選 `Consumer`；有輸入有輸出且型別不同選 `Function`；型別相同選 `UnaryOperator`；輸出是布林選 `Predicate`。熟記這張決策表，九成場景都不必自訂介面。

1.4 方法參考

當 Lambda 的主體「只是呼叫一個既有方法」時，可以用「方法參考」（method reference）進一步簡化，用雙冒號 `::` 語法。它有四種形式：

形式	語法	等價的 Lambda
靜態方法	`Integer::parseInt`	`s -> Integer.parseInt(s)`
特定物件的實體方法	`out::println`	`s -> out.println(s)`
任意物件的實體方法	`String::toUpperCase`	`s -> s.toUpperCase()`
建構子參考	`ArrayList::new`	`() -> new ArrayList()`

第三種形式最容易搞混：`String::toUpperCase` 看似沒有接收者，其實是把「串流中每個元素」當成接收者去呼叫其方法。以下範例一次示範四種形式：

範例：MethodRefDemo.java ——四種方法參考

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * 完整示範四種方法參考，並附上等價的 Lambda 對照。
 */
public class MethodRefDemo {
    public static void main(String[] args) {
        // 1. 靜態方法參考：Integer.parseInt(String)
        Function<String, Integer> parse = Integer::parseInt;
        System.out.println("parse(\"42\") = " + parse.apply("42"));

        // 2. 特定物件的實體方法：對 System.out 呼叫 println
        java.io.PrintStream out = System.out;
        java.util.function.Consumer<String> printer = out::println;
        printer.accept("特定物件的方法參考");

        // 3. 任意物件的實體方法：把輸入字串當接收者呼叫 length
        Function<String, Integer> length = String::length;
        System.out.println("length(\"Java\") = " + length.apply("Java"));

        // 4. 建構子參考：() -> new ArrayList<>()
        Supplier<List<String>> maker = ArrayList::new;
```

```
        List<String> list = maker.get();
        list.add("由建構子參考產生的清單");
        System.out.println(list);

        // 補充：兩參數方法可用 BiFunction，如 Integer::sum
        BiFunction<Integer, Integer, Integer> sum = Integer::sum;
        System.out.println("sum(3, 4) = " + sum.apply(3, 4));
    }
}
```

▶ 執行結果

```
parse("42") = 42
特定物件的方法參考
length("Java") = 4
[由建構子參考產生的清單]
sum(3, 4) = 7
```

1.5　變數捕獲與 effectively final

Lambda 可以使用其外部作用域的區域變數，這稱為「變數捕獲」（variable capture）。但有一條鐵律：被捕獲的區域變數必須是 `final` 或 **effectively final**（實質上不可變，即宣告後就沒再被指派過）。

範例：CaptureDemo.java ——合法的變數捕獲

```
import java.util.function.Supplier;

/**
 * Lambda 捕獲外部區域變數 prefix。
 * prefix 宣告後未再被指派，屬於 effectively final，合法。
 */
public class CaptureDemo {
    public static void main(String[] args) {
        String prefix = "編號-";        // 之後不再更動，實質不可變
        int base = 100;                  // 同上

        // Lambda 捕獲 prefix 與 base，記住它們當下的值
        Supplier<String> gen = () -> prefix + (base + 1);
        System.out.println(gen.get());
    }
}
```

▶ 執行結果

```
編號-101
```

> **注意**　為什麼不能修改被捕獲的區域變數？因為區域變數存放在堆疊上，方法一結束就消失，但 Lambda 可能在方法返回後才執行（例如交給另一個執行緒）。為避免存取到已消失的變數，Java 的做法是讓 Lambda 捕獲該變數的**值的複本**，因此原變數不能再改變，否則複本與本尊不一致將造成混亂。若你確實需要「可變的計數器」，正確做法是捕獲一個物件（如陣列 `int[] c = {0};` 或 `AtomicInteger`），改變其內容而非重新指派變數本身。

順帶一提，Lambda 內的 `this` 指向的是**外圍類別的實體**，而非 Lambda 自己；這與匿名類別不同（匿名類別的 `this` 指向匿名類別實體）。這也是 Lambda 更貼近「一段程式碼」而非「一個獨立物件」的證據。

1.6　Comparator 的函數式組合

排序是函數式風格大放異彩的場景。`Comparator` 提供一系列靜態與預設方法，讓多鍵排序像說話一樣自然：`comparing` 指定主要排序鍵、`thenComparing` 追加次要鍵、`reversed` 反轉、`nullsFirst` 處理空值。

範例：ComparatorDemo.java ——多鍵排序

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/**
 * 用 Comparator 的函數式組合，對員工做多鍵排序：
 * 先依部門升冪，同部門再依薪資降冪。
 */
public class ComparatorDemo {
    // record 是不可變資料載體，第 8 章會深入介紹
    record Employee(String name, String dept, int salary) { }

    public static void main(String[] args) {
        List<Employee> staff = new ArrayList<>(List.of(
                new Employee("Alice", "研發", 90000),
                new Employee("Bob", "研發", 85000),
```

```
                new Employee("Cathy", "行銷", 95000),
                new Employee("David", "行銷", 95000)));

        // 主鍵：部門升冪；次鍵：薪資降冪；末鍵：姓名升冪
        Comparator<Employee> order =
                Comparator.comparing(Employee::dept)
                        .thenComparing(
                                Comparator.comparingInt(Employee::salary)
                                        .reversed())
                        .thenComparing(Employee::name);

        staff.sort(order);              // 就地排序
        staff.forEach(e -> System.out.println(
                e.dept() + " | " + e.salary() + " | " + e.name()));
    }
}
```

▶ 執行結果

```
研發 | 90000 | Alice
研發 | 85000 | Bob
行銷 | 95000 | Cathy
行銷 | 95000 | David
（註：字串的自然排序依 Unicode 碼位，「研」排在「行」之前）
```

> **提示**　`thenComparing` 的存在讓「多鍵排序」不再需要手寫層層 `if` 比較。處理可能為 `null` 的欄位時，用 `Comparator.nullsFirst(...)` 或 `nullsLast(...)` 包住既有比較器，即可安全地把空值排到最前或最後，避免 `NullPointerException`。

1.7　Optional：優雅處理不存在的值

Tony Hoare 稱 `null` 的發明是自己「十億美元的錯誤」，因為它讓 `NullPointerException` 成為 Java 最常見的執行期例外。`Optional<T>` 是一個容器，明確表達「這裡可能有值、也可能沒有」，逼迫呼叫者正視空值，而不是默默地忽略。

1.7.1　建立與取值

建立 `Optional` 有三種方式：`of`（值一定非空，若傳入 `null` 會拋例外）、`ofNullable`（值可能為空）、`empty`（明確的空）。

範例：OptionalBasics.java ——建立與安全取值

```java
import java.util.Optional;

/**
 * 示範 Optional 的建立與 map/filter/orElse。
 */
public class OptionalBasics {

    /** 模擬「依 id 查使用者名稱」，查無資料時回傳空 Optional */
    static Optional<String> findName(int id) {
        // ofNullable：把可能為 null 的結果安全包起來
        return Optional.ofNullable(id == 1 ? "Alice" : null);
    }

    public static void main(String[] args) {
        // map：有值才轉換（轉大寫）；filter：有值且符合條件才保留
        String r1 = findName(1)
                .map(String::toUpperCase)
                .filter(s -> s.length() > 3)
                .orElse("（不符合條件）");
        System.out.println("id=1：" + r1);

        // 查無資料時，整條鏈路都被跳過，最後回傳預設值
        String r2 = findName(99)
                .map(String::toUpperCase)
                .orElse("（查無此人）");
        System.out.println("id=99：" + r2);
    }
}
```

▶ 執行結果

```
id=1：ALICE
id=99：（查無此人）
```

1.7.2　orElse 與 orElseGet 的關鍵差異

兩者都在「值不存在時提供替代」，但差別極其重要：`orElse` 的參數**無論如何都會先被求值**，而 `orElseGet` 接受一個 `Supplier`，**只有在真的需要時才呼叫**。若替代值的產生有副作用或成本很高，這個差異會造成明顯後果。

範例：OrElseVsOrElseGet.java ——副作用示範

```
import java.util.Optional;

/**
 * 對比 orElse 與 orElseGet 的求值時機。
 * 即使 Optional 有值，orElse 的參數仍會被求值（副作用照樣發生）。
 */
public class OrElseVsOrElseGet {

    /** 模擬昂貴的預設值運算，順便印出以觀察是否被呼叫 */
    static String expensiveDefault() {
        System.out.println(">> 產生預設值（昂貴運算被呼叫了）");
        return "預設值";
    }

    public static void main(String[] args) {
        Optional<String> present = Optional.of("實際值");

        System.out.println("--- 使用 orElse ---");
        // 即使有值，expensiveDefault() 仍被呼叫（浪費）
        System.out.println("結果：" + present.orElse(expensiveDefault()));

        System.out.println("--- 使用 orElseGet ---");
        // 有值，所以 Supplier 完全不會被呼叫（高效）
        System.out.println("結果：" + present.orElseGet(
                OrElseVsOrElseGet::expensiveDefault));
    }
}
```

▶ 執行結果

```
--- 使用 orElse ---
>> 產生預設值（昂貴運算被呼叫了）
結果：實際值
--- 使用 orElseGet ---
結果：實際值
```

注意 orElse 那組即使 Optional 有值，仍印出了「昂貴運算被呼叫了」。這就是為什麼替代值需要運算時，一律用 **orElseGet**。若值不存在時應該視為錯誤，則用 orElseThrow（可傳入例外的 Supplier）。

1.7.3 ifPresent 與反模式

`ifPresent` 在有值時執行動作；Java 9 起的 `ifPresentOrElse` 還能指定「沒值時」的動作。但 `Optional` 也常被誤用：

> **注意**　`Optional` 的三大反模式，務必避免：
>
> - **先 `isPresent` 再 `get`**：這只是把 `null` 檢查換了寫法，完全沒發揮 `Optional` 的價值。應改用 `map`、`filter`、`orElse` 等鏈式方法。
> - **把 `Optional` 當欄位或方法參數**：`Optional` 是設計來當「方法回傳值」的，用作欄位會增加記憶體與序列化負擔，用作參數則讓呼叫端更麻煩。參數該用多載或 `@Nullable` 表達。
> - **對集合回傳 `Optional`**：空集合本身就代表「沒有」，直接回傳空的 `List` 即可，不需要 `Optional<List<...>>`。

1.8 綜合範例：可組合的資料驗證器

最後我們把本章觀念熔於一爐：用 `Predicate` 的組合能力，打造一個可讀性極高的驗證器。每條規則都是一個 `Predicate`，用 `and` 串起來就是完整的驗證邏輯——新增規則時完全不必修改既有程式碼。

範例：PasswordValidator.java ——Predicate 組合

```
import java.util.function.Predicate;

/**
 * 用 Predicate 組合打造密碼驗證器。
 * 每條規則獨立、可重用、可自由組合，體現「開放封閉原則」。
 */
public class PasswordValidator {
    public static void main(String[] args) {
        // 三條獨立規則
        Predicate<String> longEnough = p -> p.length() >= 8;
        Predicate<String> hasDigit =
                p -> p.chars().anyMatch(Character::isDigit);
        Predicate<String> hasUpper =
                p -> p.chars().anyMatch(Character::isUpperCase);

        // 用 and 把三條規則組合成一條完整規則
        Predicate<String> valid = longEnough.and(hasDigit).and(hasUpper);

```

```
        // 逐一驗證測試資料
        for (String pwd : new String[]{"abc", "abcdefgh", "Abcdef12"}) {
            System.out.println(pwd + " → "
                    + (valid.test(pwd) ? "通過" : "不通過"));
        }
    }
}
```

▶ 執行結果

```
abc → 不通過
abcdefgh → 不通過
Abcdef12 → 通過
```

這個設計的威力在於：驗證規則本身成了可傳遞、可組合的值。這正是函數式程式設計的精髓，也是下一章 Stream API 得以如此強大的根基。

1.9 本章小結

- Lambda 運算式是「把行為當成值傳遞」的語法，從具名類別、匿名類別一路演進而來，只保留參數列與主體。
- Lambda 的目標型別必須是函數式介面（只有一個抽象方法）；建議加 `@FunctionalInterface` 讓編譯器把關。
- `java.util.function` 提供 `Function`、`Predicate`、`Consumer`、`Supplier` 等通用介面，多數場景不需自訂。
- `Function` 用 `andThen/compose` 組合，`Predicate` 用 `and/or/negate` 組合。
- 方法參考有四種形式（靜態、特定物件、任意物件、建構子），是 Lambda 的進一步精簡。
- 被 Lambda 捕獲的區域變數必須是 effectively final，因為捕獲的是值的複本。
- `Comparator` 的 `comparing/thenComparing/ reversed` 讓多鍵排序簡潔清楚。
- `Optional` 明確表達「可能沒有值」；`orElse` 一定求值、`orElseGet` 延遲求值；避免 `isPresent+get` 等反模式。

1.10 本章練習

動手練習

1. 觀念題：為什麼只有「單一抽象方法」的介面才能用 Lambda 表達？若一個介面有兩個抽象方法，編譯器會遇到什麼困難？（提示：想想編譯器如何得知你要實作哪個方法。）
2. 觀念題：`orElse` 與 `orElseGet` 在「值存在」時的行為有何不同？在什麼情況下這個差異會造成 bug 或效能問題？（提示：關注參數的求值時機與副作用。）
3. 實作題：定義一個函數式介面 `StringTransformer`，其方法接收 `String` 回傳 `String`；用它實作「去頭尾空白後轉大寫」，並改用內建的 `Function` 加 `andThen` 達成同樣效果。（提示：`String::trim` 與 `String::toUpperCase`。）
4. 實作題：給定 `List<String>` 姓名清單，用 `Comparator` 先依字串長度升冪、長度相同再依字典序排序。（提示：先 `comparingInt(String::length)` 再接 `thenComparing`。）
5. 實作題：寫一個方法 `Optional<Double> divide(int a, int b)`，當 `b` 為 0 時回傳 `Optional.empty()`，否則回傳商；呼叫端用 `orElseGet` 在除數為 0 時回傳 `Double.NaN`。（提示：不要用 `isPresent+get`。）
6. 實作題：用 `Predicate` 組合，為一個有姓名與年齡兩個欄位的 `record User` 建立「姓名非空且年齡介於 18 到 65」的驗證器，並測試三筆邊界資料。（提示：`Predicate<User>` 各寫一條，再用 `and` 串接。）

第 2 章

Stream API

> **本章學習目標**
>
> 學完本章你將能夠：
>
> - 分辨命令式與宣告式風格，說明 Stream 的三段結構與惰性求值特性。
> - 用多種方式建立 Stream，並熟練運用 `filter`、`map`、`flatMap` 等中間操作。
> - 正確選用終端操作，並掌握 `reduce` 的三種形式。
> - 用 `Collectors` 完成分組、分割與各式彙總。
> - 使用基本型別流避免自動裝箱成本。
> - 判斷何時該用平行流，並避開共享可變狀態的陷阱。

上一章我們學會把「行為」當成值傳遞。Stream API 正是這個能力的舞台：它讓我們用宣告式的方式描述「要對資料做什麼」，而不必陷入「怎麼一步步做」的迴圈細節。Stream 不是一種新的資料結構，而是一條「資料處理的管線」——資料從來源流入，經過一連串轉換，最後匯集成結果。

先強調一個常見誤解：Stream 與集合（Collection）是不同的東西。集合關心的是「如何有效率地儲存與存取元素」，Stream 關心的是「如何宣告式地計算」。Stream 不儲存資料，也不會修改來源，它只是描述一段計算。

2.1 命令式與宣告式的對比

假設需求是：「從一串交易金額中，找出所有大於 1000 的金額，乘以稅率後加總」。先看傳統的命令式寫法——我們得親手管理迴圈、暫存變數與控制流程：

範例：命令式寫法

```
import java.util.List;

/**
 * 命令式風格：明確告訴電腦「每一步怎麼做」。
 * 迴圈、條件、累加器全都要自己維護。
 */
public class ImperativeStyle {
```

```
    public static void main(String[] args) {
        List<Integer> amounts = List.of(500, 1500, 800, 2000, 1200);

        double total = 0;               // 手動維護的累加器
        for (int amount : amounts) {    // 手動走訪
            if (amount > 1000) {        // 手動過濾
                total += amount * 1.05; // 手動轉換與累加
            }
        }
        System.out.println("命令式結果：" + total);
    }
}
```

▶ 執行結果

```
命令式結果：4935.0
```

換成 Stream 的宣告式寫法，我們只描述「做什麼」，把「怎麼做」交給函式庫：

範例：宣告式寫法（Stream）

```
import java.util.List;

/**
 * 宣告式風格：像在描述需求，讀起來近乎自然語言。
 * 過濾、轉換、加總各是一個獨立、可讀的步驟。
 */
public class DeclarativeStyle {
    public static void main(String[] args) {
        List<Integer> amounts = List.of(500, 1500, 800, 2000, 1200);

        double total = amounts.stream()         // 來源
                .filter(a -> a > 1000)           // 中間操作：過濾
                .mapToDouble(a -> a * 1.05)      // 中間操作：轉換
                .sum();                          // 終端操作：加總
        System.out.println("宣告式結果：" + total);
    }
}
```

▶ 執行結果

```
宣告式結果：4935.0
```

兩者結果相同，但宣告式寫法把「意圖」直接攤在眼前，沒有樣板迴圈干擾閱讀，日後要平行化也只需一個字（稍後會看到）。

2.2 Stream 的結構與特性

2.2.1 三段式結構

每條 Stream 管線都由三部分組成：

- **來源**（source）：集合、陣列、產生器等。
- **中間操作**（intermediate operation）：如 `filter`、`map`，每個都回傳一個新的 Stream，因此可以鏈式串接。
- **終端操作**（terminal operation）：如 `collect`、`sum`，觸發整條管線真正執行並產出結果。

2.2.2 兩大特性：惰性與一次性

Stream 有兩個必須牢記的特性。第一是**惰性求值**：中間操作不會立即執行，它們只是在「登記」要做的事，直到遇到終端操作才一次啟動。第二是**只能消費一次**：一條 Stream 被終端操作消費後就作廢，再次使用會拋出例外。

> **注意**　Stream 只能消費一次。以下程式會在第二次終端操作時拋出 `IllegalStateException: stream has already been operated upon or closed`：
>
> - `Stream<String> s = list.stream();`
> - `s.count();`（第一次終端操作，正常）
> - `s.forEach(...);`（第二次，拋例外）
>
> 正確做法是每次都從來源重新呼叫 `list.stream()` 取得新的 Stream。

2.3 建立 Stream

Stream 的來源相當多元。下面示範最常用的幾種建立方式：

範例：CreateStream.java ——多種建立方式

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

/**
```

```
 * 彙整六種建立 Stream 的方式。
 * 無限流（iterate/generate）務必用 limit 收斂，否則永不終止。
 */
public class CreateStream {
    public static void main(String[] args) {
        // 1. 從集合
        List<String> list = List.of("a", "b", "c");
        System.out.println("集合：" + list.stream().count());

        // 2. Stream.of 直接列舉元素
        System.out.println("of：" + Stream.of(1, 2, 3).count());

        // 3. iterate：從種子反覆套用函式（無限，需 limit）
        Stream.iterate(1, n -> n * 2).limit(5)
                .forEach(n -> System.out.print(n + " "));
        System.out.println();

        // 4. generate：反覆呼叫 Supplier（無限，需 limit）
        Stream.generate(() -> "x").limit(3)
                .forEach(System.out::print);
        System.out.println();

        // 5. IntStream.range：產生整數區間 [1, 5)
        System.out.println("range 總和："
                + IntStream.range(1, 5).sum());

        // 6. Arrays.stream：從陣列
        int[] arr = {10, 20, 30};
        System.out.println("陣列平均："
                + Arrays.stream(arr).average().orElse(0));
    }
}
```

▶ 執行結果

```
集合：3
of：3
1 2 4 8 16
xxx
range 總和：10
陣列平均：20.0
```

2.4 中間操作

2.4.1 常用中間操作總覽

中間操作最常用的有：`filter`（過濾）、`map`（一對一轉換）、`flatMap`（一對多後攤平）、`sorted`（排序）、`distinct` （去重）、`limit/skip`（截取）、`peek`（窺視）。以下用一個 `flatMap` 的經典場景示範：把「巢狀清單」攤平成單層。

範例：FlatMapDemo.java ——攤平巢狀結構

```
import java.util.List;

/**
 * flatMap 把「每個元素展開成一條子流」再串接成單一條流。
 * 常用於攤平巢狀集合，或把句子拆成單字。
 */
public class FlatMapDemo {
    public static void main(String[] args) {
        List<List<Integer>> nested = List.of(
                List.of(1, 2, 3),
                List.of(4, 5),
                List.of(6));

        // map 會得到 Stream<List<Integer>>；flatMap 才能攤平成 Integer
        List<Integer> flat = nested.stream()
                .flatMap(List::stream)      // 每個子清單展開成流再串接
                .toList();                  // Java 16 起的簡便收集
        System.out.println("攤平後：" + flat);

        // 把句子清單拆成不重複的單字
        List<String> sentences = List.of("java is fun", "fun is good");
        List<String> words = sentences.stream()
                .flatMap(s -> java.util.Arrays.stream(s.split(" ")))
                .distinct()                 // 去除重複
                .sorted()                   // 自然排序
                .toList();
        System.out.println("單字：" + words);
    }
}
```

▶ 執行結果

```
攤平後：[1, 2, 3, 4, 5, 6]
```

```
單字：[fun, good, is, java]
```

2.4.2 惰性求值的證明

前面說中間操作是惰性的，用 peek 就能親眼驗證。peek 讓我們窺視每個流經的元素。觀察下面輸出的順序，會發現處理是「垂直」而非「水平」的：一個元素一路走完整條管線，下一個元素才開始，而不是所有元素先做完 filter 再一起做 map。

範例：LazyEvaluation.java ——垂直處理

```java
import java.util.stream.Stream;

/**
 * 用 peek 觀察元素流經管線的順序，驗證惰性求值。
 * 每個元素「一條龍」走完 filter 與 map，才輪到下一個元素。
 */
public class LazyEvaluation {
    public static void main(String[] args) {
        Stream.of("apple", "banana", "avocado", "cherry")
                .peek(s -> System.out.println("進入：" + s))
                .filter(s -> s.startsWith("a"))      // 只留 a 開頭
                .peek(s -> System.out.println("  通過過濾：" + s))
                .map(String::toUpperCase)            // 轉大寫
                .forEach(s -> System.out.println("    輸出：" + s));
    }
}
```

▶ 執行結果

```
進入：apple
  通過過濾：apple
    輸出：APPLE
進入：banana
進入：avocado
  通過過濾：avocado
    輸出：AVOCADO
進入：cherry
```

> **說明**　惰性求值不只是實作細節，它帶來實質好處。第一，可以支援無限流（配合 limit 只取需要的部分）。第二，能做「短路」最佳化：如 findFirst、anyMatch 一旦找到答案

> 就立刻停止，不會白跑剩餘元素。第三，多個中間操作會「融合」成單次走訪，而非每個操作各走一遍整個資料集。

2.5 終端操作

2.5.1 常見終端操作

終端操作觸發計算並產出結果。除了前面用過的 `forEach`、`count`、`sum`，還有比對類的 `anyMatch/allMatch/noneMatch`、搜尋類的 `findFirst/findAny`、極值類的 `min/max`，以及收集類的 `toList`。

範例：TerminalOps.java ——各式終端操作

```java
import java.util.List;

/**
 * 示範比對、搜尋、極值等終端操作。
 * 比對與搜尋操作具有「短路」特性，找到即停。
 */
public class TerminalOps {
    public static void main(String[] args) {
        List<Integer> nums = List.of(3, 7, 2, 9, 4, 6);

        // 比對：是否全部大於 0 / 是否有任一大於 8 / 是否沒有負數
        System.out.println("全部為正？" + nums.stream().allMatch(n -> n > 0));
        System.out.println("有 >8 的？" + nums.stream().anyMatch(n -> n > 8));
        System.out.println("無負數？" + nums.stream().noneMatch(n -> n < 0));

        // 搜尋：第一個大於 5 的元素（Optional，可能不存在）
        System.out.println("第一個 >5："
                + nums.stream().filter(n -> n > 5).findFirst().orElse(-1));

        // 極值：需提供 Comparator
        System.out.println("最大值："
                + nums.stream().max(Integer::compareTo).orElseThrow());
    }
}
```

▶ 執行結果

```
全部為正？true
有 >8 的？true
```

```
無負數？true
第一個 >5：7
最大值：9
```

2.5.2 reduce 的三種形式

reduce 是最通用的彙總操作，把一串元素「摺疊」成單一結果。它有三種形式，理解它們的差異是掌握 Stream 的分水嶺：

範例：ReduceDemo.java ——reduce 三種形式

```java
import java.util.List;
import java.util.Optional;

/**
 * reduce 的三種多載：
 * (1) 無初值 → 回傳 Optional（流可能為空）。
 * (2) 有初值 → 回傳同型別結果。
 * (3) 有初值與合併器 → 支援型別轉換與平行合併。
 */
public class ReduceDemo {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5);

        // 形式一：無初值，結果可能不存在，故為 Optional
        Optional<Integer> sum1 = nums.stream().reduce((a, b) -> a + b);
        System.out.println("形式一總和：" + sum1.orElse(0));

        // 形式二：有初值 0，一定有結果
        int sum2 = nums.stream().reduce(0, Integer::sum);
        System.out.println("形式二總和：" + sum2);

        // 形式三：初值為 0，累加器把字串長度加總；合併器供平行時合併
        List<String> words = List.of("java", "stream", "api");
        int totalLen = words.stream()
                .reduce(0, (acc, s) -> acc + s.length(), Integer::sum);
        System.out.println("字元總長：" + totalLen);
    }
}
```

▶ 執行結果

```
形式一總和：15
形式二總和：15
字元總長：13
```

2.6　collect 與 Collectors 深入

collect 搭配 Collectors 工具類，是把 Stream 匯集成集合或彙總值的主力。基本的有 toList、toSet、toMap，字串用 joining，數值用 counting、summingInt、averagingInt。

範例：CollectorsBasics.java ——匯集與字串串接

```java
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

/**
 * 示範 toMap 的重複鍵處理與 joining 的三參數版本。
 */
public class CollectorsBasics {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Amy", "Ben");

        // joining 三參數：分隔符、前綴、後綴
        String joined = names.stream()
                .collect(Collectors.joining(", ", "[", "]"));
        System.out.println("串接：" + joined);

        // toMap：以首字母為鍵；遇重複鍵用合併函式（保留較長者）
        Map<Character, String> byInitial = names.stream()
                .collect(Collectors.toMap(
                        s -> s.charAt(0),              // 鍵
                        Function.identity(),           // 值
                        (a, b) -> a.length() >= b.length() ? a : b));
        System.out.println("依首字母：" + byInitial);
    }
}
```

▶ 執行結果

```
串接：[Alice, Bob, Amy, Ben]
依首字母：{A=Alice, B=Bob}
```

> **注意**　使用 `Collectors.toMap` 時，若兩個元素產生相同的鍵，又沒有提供合併函式（第三個參數），會在執行期拋出 `IllegalStateException`（訊息為 Duplicate key）。凡是鍵不保證唯一的場景，都要記得提供合併函式來決定如何取捨。

2.6.1 分組與分割

`groupingBy` 依某個鍵把元素分組成 `Map`，還能搭配「下游收集器」對每組再做彙總，甚至多層巢狀分組。`partitioningBy` 是特例，依布林條件把元素分成 `true` 與 `false` 兩組。

範例：GroupingDemo.java ——分組與下游收集器

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * groupingBy 的三種用法：單層分組、雙參數（下游收集器）、多層巢狀。
 */
public class GroupingDemo {
    record Product(String category, String name, int price) { }

    public static void main(String[] args) {
        List<Product> items = List.of(
                new Product("食品", "麵包", 40),
                new Product("食品", "牛奶", 60),
                new Product("文具", "鉛筆", 15),
                new Product("文具", "筆記本", 55));

        // 單層分組：類別 → 該類別的商品清單
        Map<String, List<Product>> byCat = items.stream()
                .collect(Collectors.groupingBy(Product::category));
        System.out.println("各類別商品數：");
        byCat.forEach((k, v) -> System.out.println("  " + k + "：" + v.size()));

        // 雙參數：類別 → 該類別平均價格（下游收集器）
        Map<String, Double> avgByCat = items.stream()
                .collect(Collectors.groupingBy(Product::category,
                        Collectors.averagingInt(Product::price)));
        System.out.println("各類別平均價：" + avgByCat);
```

```
29
30          // 多層巢狀：類別 → (是否高價) → 名稱清單
31          Map<String, Map<Boolean, List<String>>> nested = items.stream()
32                  .collect(Collectors.groupingBy(Product::category,
33                          Collectors.groupingBy(p -> p.price() >= 50,
34                                  Collectors.mapping(Product::name,
35                                          Collectors.toList()))));
36          System.out.println("巢狀分組：" + nested);
37      }
38  }
```

▶ 執行結果

```
各類別商品數：
  食品：2
  文具：2
各類別平均價：{食品=50.0, 文具=35.0}
巢狀分組：{食品={true=[牛奶], false=[麵包]},
 文具={true=[筆記本], false=[鉛筆]}}
```

2.7 基本型別流

一般 Stream<Integer> 在每次運算時都要在 int 與 Integer 之間裝箱、拆箱，資料量大時成本可觀。IntStream、LongStream、DoubleStream 直接操作基本型別，並提供 sum、average、summaryStatistics 等便利方法。用 mapToInt 可從物件流轉成 IntStream，用 boxed 則能轉回物件流。

範例：PrimitiveStream.java ——避免裝箱

```
1  import java.util.List;
2  import java.util.IntSummaryStatistics;
3
4  /**
5   * 用 mapToInt 轉成 IntStream，一次取得多項統計，全程免裝箱。
6   */
7  public class PrimitiveStream {
8      public static void main(String[] args) {
9          List<String> words = List.of("java", "stream", "api", "lambda");
10
11         // mapToInt 把物件流轉為 IntStream（此處取每個字的長度）
12         IntSummaryStatistics stats = words.stream()
```

```
                .mapToInt(String::length)       // Stream<String> → IntStream
                .summaryStatistics();           // 一次算出多項統計

        System.out.println("數量：" + stats.getCount());
        System.out.println("總和：" + stats.getSum());
        System.out.println("平均：" + stats.getAverage());
        System.out.println("最長：" + stats.getMax());
    }
}
```

▶ 執行結果

```
數量：4
總和：19
平均：4.75
最長：6
```

2.8 平行流

把 `stream()` 換成 `parallelStream()`，或在既有流上呼叫 `parallel()`，Stream 就會利用多核心平行處理。它底層使用共用的 `ForkJoinPool` 把資料切段、分頭運算、再合併結果。聽起來很美好，但平行化是把雙面刃。

2.8.1 平行流的適用條件

平行流要真正變快，需同時滿足幾個條件：資料量夠大（否則切分與合併的開銷超過效益）、每個操作是無狀態且獨立的、不存取共享的可變狀態、資料來源易於切分（陣列、`ArrayList` 佳，`LinkedList` 差）。

> **注意**　平行流最危險的錯誤是**存取共享可變狀態**。下面的程式想用平行流把元素加進一個普通 `ArrayList`，但 `ArrayList` 並非執行緒安全，多個執行緒同時 `add` 會造成資料遺失、錯亂，甚至拋出例外——而且問題時有時無，極難除錯。

範例：ParallelPitfall.java ——共享可變狀態的錯誤

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;

/**
```

```
 * 反面教材：平行流寫入非執行緒安全的 ArrayList。
 * 預期 1000 筆，實際往往少於 1000（甚至拋例外），結果不可靠。
 */
public class ParallelPitfall {
    public static void main(String[] args) {
        List<Integer> unsafe = new ArrayList<>();   // 非執行緒安全

        // 錯誤：多執行緒同時對同一個 List 做 add
        IntStream.range(0, 1000).parallel()
                .forEach(unsafe::add);              // 危險的副作用
        System.out.println("錯誤寫法筆數（應為1000）：" + unsafe.size());

        // 正解：用 collect 讓框架安全地合併各段結果
        List<Integer> safe = IntStream.range(0, 1000).parallel()
                .boxed()
                .toList();
        System.out.println("正確寫法筆數：" + safe.size());
    }
}
```

▶ 執行結果

```
錯誤寫法筆數（應為1000）：968   // 每次執行可能不同，甚至拋出例外
正確寫法筆數：1000
```

正確的平行化原則是：**不要在 forEach 裡製造副作用**，改用 `collect` 或 `reduce` 這類為平行合併而設計的操作。它們會讓每個執行緒各自累積局部結果，最後由框架安全地合併，天生就是執行緒安全的。

2.9 綜合實戰：員工資料多維度分析

最後用一個貫穿式範例，展示 Stream 在真實資料分析中的威力。給定一組員工資料，我們一口氣回答多個查詢：各部門人數、各部門平均薪資、全公司最高薪者、以及把研發部的姓名串成一行。

範例：EmployeeAnalytics.java ——多維度分析

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
```

```
 * 以一份員工資料集，示範 Stream 一站式完成多種分析查詢。
 */
public class EmployeeAnalytics {
    record Employee(String name, String dept, int salary) { }

    public static void main(String[] args) {
        List<Employee> staff = List.of(
                new Employee("Alice", "研發", 92000),
                new Employee("Bob", "研發", 88000),
                new Employee("Cathy", "行銷", 76000),
                new Employee("David", "行銷", 81000),
                new Employee("Eva", "業務", 95000));

        // 查詢一：各部門人數
        Map<String, Long> countByDept = staff.stream()
                .collect(Collectors.groupingBy(
                        Employee::dept, Collectors.counting()));
        System.out.println("各部門人數：" + countByDept);

        // 查詢二：各部門平均薪資
        Map<String, Double> avgByDept = staff.stream()
                .collect(Collectors.groupingBy(Employee::dept,
                        Collectors.averagingInt(Employee::salary)));
        System.out.println("各部門均薪：" + avgByDept);

        // 查詢三：全公司最高薪者
        String topEarner = staff.stream()
                .max((a, b) -> Integer.compare(a.salary(), b.salary()))
                .map(Employee::name)
                .orElse("無");
        System.out.println("最高薪：" + topEarner);

        // 查詢四：研發部姓名串成一行
        String rdNames = staff.stream()
                .filter(e -> e.dept().equals("研發"))
                .map(Employee::name)
                .collect(Collectors.joining("、"));
        System.out.println("研發成員：" + rdNames);
    }
}
```

▶ 執行結果

```
各部門人數：{研發=2, 行銷=2, 業務=1}
各部門均薪：{研發=90000.0, 行銷=78500.0, 業務=95000.0}
最高薪：Eva
研發成員：Alice、Bob
```

短短數行就完成了過去需要層層迴圈與暫存變數的分析。這正是 Stream 的價值：讓資料處理的程式碼讀起來就像在描述問題本身。

2.10 本章小結

- Stream 是「資料處理管線」而非資料結構，強調宣告式描述「做什麼」。
- 管線由來源、中間操作、終端操作三段組成；中間操作回傳新 Stream 可鏈式串接，終端操作觸發執行。
- Stream 惰性求值（遇終端操作才啟動、支援短路與操作融合），且只能消費一次。
- 常用中間操作：`filter`、`map`、`flatMap`、`sorted`、`distinct`、`limit/skip`、`peek`。
- `reduce` 有無初值、有初值、有合併器三種形式；`collect` 搭配 `Collectors` 完成匯集與彙總。
- `groupingBy` 可單層、可接下游收集器、可多層巢狀；`partitioningBy` 依布林條件二分。
- 基本型別流（`IntStream` 等）避免自動裝箱，並提供 `sum`、`summaryStatistics` 等便利方法。
- 平行流適用於大資料、無狀態、無共享可變狀態的場景；切勿在 `forEach` 製造副作用，應改用 `collect/reduce`。

2.11 本章練習

動手練習

1. 觀念題：為什麼 Stream 被終端操作消費後就不能再用？請說明惰性求值帶來的三項好處。（提示：想想無限流、短路、操作融合。）
2. 觀念題：`map` 與 `flatMap` 有何不同？在什麼情況下必須用 `flatMap`？（提示：想想回傳值是「一個元素」還是「一條子流」。）
3. 實作題：給定 `List<String>` 的句子清單，用 Stream 統計所有出現過的不重複單字數量（忽略大小寫）。（提示：`flatMap` 拆字、`map` 轉小寫、`distinct`。）
4. 實作題：用 `IntStream` 計算 1 到 100 之間所有質數的總和。（提示：`IntStream.rangeClosed` 搭配 `filter` 判斷質數。）

5. 實作題：給定員工資料（姓名、部門、薪資），用 `groupingBy` 求出「每個部門薪資最高的員工姓名」。（提示：下游收集器用 `Collectors.maxBy` 再 `collectingAndThen` 取出。）
6. 實作題：寫一個平行流版本的總和計算，並用 `reduce` 確保結果正確；說明為何這裡用 `reduce` 不會有共享可變狀態問題。（提示：`reduce` 每段各自累積局部結果再合併。）

第 3 章

多執行緒基礎

本章學習目標

學完本章你將能夠：

- 分辨程序與執行緒，說明為什麼需要多執行緒。
- 用三種方式建立執行緒，並理解執行緒的六種狀態。
- 正確處理中斷機制，避免吞掉 `InterruptedException`。
- 重現競態條件，並用 `synchronized` 與 `volatile` 解決原子性與可見性問題。
- 用 `wait/notify` 實作生產者消費者模型。
- 辨識並預防死結，盤點常見類別的執行緒安全等級。

現代 CPU 早已是多核心的天下，一支只會用單一執行緒的程式，等於買了八線道高速公路卻只開一條車道。多執行緒讓程式能同時處理多件事：一邊下載檔案、一邊更新進度條，或讓伺服器同時服務上千個使用者。但並行也是 Java 中最容易出錯、最難除錯的領域——本章與下一章將帶你把這塊硬骨頭啃下來。

我們先建立正確的心智模型：多個執行緒共享同一塊記憶體（堆積），這既是它們高效協作的原因，也是所有並行 bug 的根源。理解「共享」帶來的問題，是掌握並行的第一步。

3.1 程序與執行緒

程序（process）是「執行中的程式」，作業系統為它配置獨立的記憶體空間；執行緒（thread）則是程序內的執行單位，同一程序的多個執行緒共享該程序的記憶體。

比較項目	程序（process）	執行緒（thread）
記憶體	各自獨立	共享所屬程序的記憶體
建立成本	高	相對低
通訊方式	需 IPC（較複雜）	直接讀寫共享變數
隔離性	強（一個崩潰不影響他者）	弱（一個出錯可能拖垮整程序）

我們追求多執行緒，通常是為了三個目標：**回應性**（UI 執行緒不被耗時工作卡住）、**吞吐量**（同時處理更多請求）、**多核心利用**（讓運算真正平行以縮短總時間）。

3.2 建立執行緒的三種方式

3.2.1 方式一：繼承 Thread

第一種方式是繼承 `Thread` 類別並覆寫 `run` 方法。呼叫 `start`（不是 `run`！）才會真正開啟新執行緒。

範例：繼承 Thread

```
/**
 * 繼承 Thread 並覆寫 run。
 * 缺點：Java 單一繼承，繼承了 Thread 就無法再繼承其他類別。
 */
public class ThreadBySubclass extends Thread {
    @Override
    public void run() {
        // 這段程式碼會在新執行緒中執行
        System.out.println("執行緒名稱：" + getName());
    }

    public static void main(String[] args) {
        ThreadBySubclass t = new ThreadBySubclass();
        t.start();          // 啟動新執行緒（呼叫 run() 只是普通方法呼叫）
    }
}
```

▶ 執行結果

```
執行緒名稱：Thread-0
```

3.2.2 方式二：實作 Runnable（建議）

第二種方式是實作 `Runnable` 介面，再把它交給 `Thread`。這是業界建議的做法，因為它把「要做的事」（`Runnable`）與「執行的機制」（`Thread`）分離，符合關注點分離原則，也保留了繼承其他類別的彈性。

範例：實作 Runnable 與 Lambda 寫法

```
/**
 * 實作 Runnable，再交給 Thread 執行。
 * Runnable 是函數式介面，可直接用 Lambda（呼應第 1 章）。
 */
```

```
public class ThreadByRunnable {
    public static void main(String[] args) {
        // 方式二：具名或匿名實作 Runnable
        Runnable task = () -> System.out.println(
                "在 " + Thread.currentThread().getName() + " 中執行");

        Thread t1 = new Thread(task, "worker-1");
        t1.start();

        // 方式三：直接把 Lambda 傳給 Thread（最精簡）
        new Thread(() -> System.out.println("Lambda 執行緒"),
                "worker-2").start();
    }
}
```

▶ 執行結果

```
在 worker-1 中執行
Lambda 執行緒
（註：兩行輸出的順序可能不同）
```

方式	優點	缺點
繼承 Thread	寫法直覺	佔用唯一的繼承名額
實作 Runnable	關注點分離、可再繼承	需多包一層 Thread
Lambda	最精簡	僅適合簡短工作

> **注意**　啟動執行緒一定要呼叫 `start()` 而非 `run()`。直接呼叫 `run()` 只是在目前執行緒裡執行那段程式碼，根本沒有開新執行緒，是初學者最常見的錯誤。此外，同一個 `Thread` 物件不能 `start()` 兩次，否則會拋出 `IllegalThreadStateException`。

3.3　執行緒的生命週期與狀態

一個執行緒從建立到結束，會歷經六種狀態（定義於 `Thread.State` 列舉）：

狀態	意義
NEW	已建立但尚未 start
RUNNABLE	可執行（正在執行或等待 CPU 排程）
BLOCKED	等待進入 synchronized 區塊的鎖
WAITING	無限期等待（如 wait、join 無逾時）
TIMED_WAITING	有限期等待（如 sleep、有逾時的 wait）
TERMINATED	執行完畢或因例外結束

用 getState 可以觀察狀態變化：

範例：ThreadStateDemo.java ——觀察狀態

```
/**
 * 觀察執行緒在不同時點的狀態。
 */
public class ThreadStateDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            try {
                Thread.sleep(100);      // 進入 TIMED_WAITING
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        System.out.println("start 前：" + t.getState());   // NEW
        t.start();
        Thread.sleep(20);               // 讓 t 有機會進入 sleep
        System.out.println("sleep 中：" + t.getState());   // TIMED_WAITING
        t.join();                       // 等 t 結束
        System.out.println("結束後：" + t.getState());     // TERMINATED
    }
}
```

▶ 執行結果

```
start 前：NEW
sleep 中：TIMED_WAITING
結束後：TERMINATED
```

3.4 執行緒的常用方法

`sleep` 讓目前執行緒暫停一段時間，它是**靜態方法**，且**不會釋放持有的鎖**；`join` 讓目前執行緒等待另一執行緒結束；`yield` 提示排程器讓出 CPU（僅建議，不保證）；`setDaemon` 把執行緒設為守護執行緒。

範例：JoinDemo.java ——等待其他執行緒完成

```
/**
 * 用 join 確保主執行緒在工作執行緒完成後才繼續。
 */
public class JoinDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println("工作進度：" + i);
                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        worker.start();
        worker.join();                  // 主執行緒在此等待 worker 結束
        System.out.println("工作已完成，主執行緒繼續");
    }
}
```

▶ 執行結果

```
工作進度：1
工作進度：2
工作進度：3
工作已完成，主執行緒繼續
```

守護執行緒（daemon thread）是「背景服務」型的執行緒，當所有非守護執行緒都結束時，JVM 不會等它，會直接關閉。適合用於心跳、監控等背景工作。

範例：DaemonDemo.java ——背景心跳

```
/**
```

```
 * 守護執行緒示範：背景心跳。
 * 主執行緒結束後，JVM 不等守護執行緒即關閉。
 */
public class DaemonDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread heartbeat = new Thread(() -> {
            while (true) {              // 看似無限迴圈
                System.out.println("心跳...");
                try {
                    Thread.sleep(30);
                } catch (InterruptedException e) {
                    break;
                }
            }
        });

        heartbeat.setDaemon(true);      // 必須在 start 前設定
        heartbeat.start();
        Thread.sleep(100);              // 主執行緒只活這麼久
        System.out.println("主執行緒結束，JVM 即將關閉");
    }
}
```

▶ 執行結果

```
心跳...
心跳...
心跳...
主執行緒結束，JVM 即將關閉
（註：心跳次數視時間而定，主執行緒一結束心跳就停）
```

3.5 中斷機制

Java 沒有「強制終止」執行緒的安全方法（stop 已被棄用）。取而代之的是「中斷」這套協作式機制：一個執行緒對另一個呼叫 interrupt，只是「設定一個中斷旗標」，被中斷者要自己檢查旗標並決定如何回應。

範例：InterruptDemo.java ——正確回應中斷

```
/**
 * 中斷是「請求」而非「命令」：被中斷者自行檢查旗標並優雅收尾。
 */
```

```java
public class InterruptDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            // 用中斷旗標作為迴圈的結束條件
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("工作中...");
                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    // sleep 被中斷會清除旗標，這裡要「恢復」旗標
                    Thread.currentThread().interrupt();
                    System.out.println("收到中斷，準備收尾");
                }
            }
            System.out.println("已優雅結束");
        });

        worker.start();
        Thread.sleep(160);
        worker.interrupt();             // 送出中斷請求
        worker.join();
    }
}
```

▶ 執行結果

```
工作中...
工作中...
工作中...
收到中斷，準備收尾
已優雅結束
```

注意　最常見的中斷處理錯誤是「吞掉」`InterruptedException`——只寫一個空的 `catch` 區塊，或僅印出堆疊而不做任何事。這會讓中斷訊號憑空消失，導致上層無法得知該停止。正確做法只有兩種：往外拋出讓呼叫端處理，或 **呼叫 `Thread.currentThread().interrupt()` 恢復中斷旗標**，讓後續程式碼仍能偵測到中斷。

3.6 競態條件

當多個執行緒同時讀寫同一份資料，且最終結果取決於執行的先後順序時，就發生了「競態條件」（race condition）。最經典的例子是兩個執行緒同時對計數器累加：

範例：RaceCondition.java ——遺失更新

```
/**
 * 兩個執行緒各對 count 累加十萬次，預期二十萬。
 * 但 count++ 並非原子操作，結果幾乎必定小於二十萬。
 */
public class RaceCondition {
    static int count = 0;               // 共享的可變狀態

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 100_000; i++) {
                count++;                // 讀取、加一、寫回：三個步驟
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("預期：200000，實際：" + count);
    }
}
```

▶ 執行結果

```
預期：200000，實際：147623
（註：實際值每次執行都不同，但幾乎不可能剛好等於 200000）
```

問題出在 count++ 看似一行，實際是「讀取目前值、加一、寫回」三個步驟。若執行緒 A 讀到 100 的同時執行緒 B 也讀到 100，兩者各自加一寫回 101，本應是 102 卻只增加了一次——這就是「遺失更新」。這種「先讀、再改、後寫」的操作稱為 read-modify-write，是並行問題的溫床。

3.7 synchronized：互斥同步

解決競態條件的根本手段是「同步」：確保同一時刻只有一個執行緒能進入臨界區。`synchronized` 關鍵字提供了這種互斥能力，它可以修飾方法或程式碼區塊。

範例：SyncCounter.java ——用同步修正競態

```
/**
 * 用 synchronized 保護 count++，讓它成為不可分割的原子操作。
 */
public class SyncCounter {
    static int count = 0;
    static final Object lock = new Object();    // 專用的鎖物件

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 100_000; i++) {
                // 同步區塊：同一時刻只有一個執行緒能進入
                synchronized (lock) {
                    count++;            // 現在是原子操作了
                }
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("預期：200000，實際：" + count);
    }
}
```

▶ 執行結果

```
預期：200000，實際：200000
```

關於 `synchronized` 有幾個要點：修飾**實體方法**時鎖的是 `this`；修飾**靜態方法**時鎖的是該類別的 `Class` 物件；用 **同步區塊**時可以指定任意物件當鎖。每個 Java 物件內部都有一個「監視器鎖」（monitor），這也是為什麼任何物件都能拿來當鎖。

> **提示**　臨界區要盡量小。把 `synchronized` 的範圍縮到「真正需要互斥」的那幾行，而非鎖住整個方法，可以大幅減少執行緒互相等待的時間，提升並行效能。這也是偏好「同步區塊」而非「同步方法」的主要理由。

3.8　volatile：可見性

除了原子性，並行還有另一個隱形陷阱：**可見性**。每個執行緒可能把變數快取在自己的暫存器或 CPU 快取中，導致一個執行緒的修改，另一個執行緒遲遲看不到。下面的旗標迴圈就可能永遠停不下來：

範例：VolatileDemo.java ——可見性問題

```
/**
 * 若 running 沒有 volatile，工作執行緒可能看不到主執行緒的修改，
 * 而陷入無法結束的迴圈。加上 volatile 保證可見性。
 */
public class VolatileDemo {
    // 拿掉 volatile 這個程式可能永遠不會停止
    static volatile boolean running = true;

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            long n = 0;
            while (running) {           // 讀取共享旗標
                n++;
            }
            System.out.println("已停止，累計：" + n);
        });

        worker.start();
        Thread.sleep(50);
        running = false;                // 主執行緒改變旗標
        System.out.println("已送出停止指令");
        worker.join();
    }
}
```

▶ 執行結果

```
已送出停止指令
已停止，累計：183472910
```

（註：累計值每次不同；若拿掉 `volatile`，worker 可能永不停止）

> **注意**　`volatile` 只保證**可見性與禁止指令重排，不保證原子性**。因此 `volatile int count; count++;` 依然有競態條件，因為 `count++` 仍是三步驟。下表釐清兩者分工：

保證	`volatile`	`synchronized`
可見性	是	是
原子性	否	是
互斥	否	是

一句話總結選用原則：只需要「一個執行緒寫、其他執行緒讀」的旗標，用 `volatile`；只要牽涉「先讀再改後寫」的複合操作，就得用 `synchronized`（或下一章的原子類別）。

3.9　wait/notify 與生產者消費者

有時執行緒需要「等待某個條件成立」再繼續，例如消費者要等生產者放入資料。`wait/notify/notifyAll` 提供這種協調能力，但使用前提是 **必須先持有該物件的鎖**（即在 `synchronized` 區塊內呼叫）。

範例：ProducerConsumer.java ——共享緩衝區

```
import java.util.LinkedList;
import java.util.Queue;

/**
 * 生產者消費者：以固定容量的緩衝區協調兩端。
 * wait 會釋放鎖並等待，notify/notifyAll 喚醒等待者。
 */
public class ProducerConsumer {
    static final Queue<Integer> buffer = new LinkedList<>();
    static final int CAPACITY = 3;
    static final Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                synchronized (lock) {
                    // 用 while 而非 if 檢查條件，防止「虛假喚醒」
                    while (buffer.size() == CAPACITY) {
```

```
                        waitQuietly();
                    }
                    buffer.add(i);
                    System.out.println("生產：" + i);
                    lock.notifyAll();       // 喚醒可能在等的消費者
                }
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                synchronized (lock) {
                    while (buffer.isEmpty()) {
                        waitQuietly();
                    }
                    System.out.println("    消費：" + buffer.poll());
                    lock.notifyAll();       // 喚醒可能在等的生產者
                }
            }
        });

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }

    /** 封裝 wait 與中斷處理，讓主邏輯乾淨 */
    static void waitQuietly() {
        try {
            lock.wait();                    // 釋放鎖並等待被喚醒
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

▶ 執行結果

```
生產：1
生產：2
    消費：1
生產：3
```

```
    消費：2
（註：實際交錯順序每次不同，但生產與消費總數皆為 5）
```

> **說明**　為什麼檢查條件要用 `while` 而不是 `if`？因為存在「虛假喚醒」（spurious wakeup）：執行緒可能在沒有人 `notify` 的情況下被喚醒。用 `while` 可以在被喚醒後重新檢查條件是否真的成立，若否就繼續等待。這是撰寫 `wait/notify` 程式的鐵律。

3.10　死結

死結（deadlock）是指兩個以上的執行緒互相持有對方需要的鎖，彼此無限等待。下面是一個穩定可重現的死結：`t1` 先鎖 A 再求 B，`t2` 先鎖 B 再求 A。

範例：DeadlockDemo.java ——交叉取鎖

```java
/**
 * 兩個執行緒以相反順序取得兩把鎖，造成死結。
 * 此程式會卡住不會結束（需手動終止）。
 */
public class DeadlockDemo {
    static final Object lockA = new Object();
    static final Object lockB = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lockA) {              // 先拿 A
                System.out.println("t1 取得 A，欲取 B");
                sleep(50);
                synchronized (lockB) {          // 再求 B（被 t2 佔用）
                    System.out.println("t1 取得 B");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (lockB) {              // 先拿 B
                System.out.println("t2 取得 B，欲取 A");
                sleep(50);
                synchronized (lockA) {          // 再求 A（被 t1 佔用）
                    System.out.println("t2 取得 A");
                }
```

```
            }
        });

        t1.start();
        t2.start();
    }

    static void sleep(long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

▶ 執行結果

```
t1 取得 A，欲取 B
t2 取得 B，欲取 A
（程式卡死，兩執行緒互相等待對方的鎖，永不結束）
```

死結需同時滿足四個條件（Coffman 條件）：**互斥**（資源不可共享）、**持有並等待**（持有資源又等待其他資源）、**不可搶奪**（不能強制奪取他人的資源）、**循環等待**（形成等待環路）。只要打破任一條件即可預防。最實用的策略是打破「循環等待」：**規定所有執行緒都以固定的順序取得多把鎖**（例如永遠先 A 後 B）。下一章的 `tryLock` 帶逾時，則是打破「持有並等待」的另一利器。

3.11　執行緒安全等級盤點

最後盤點幾個常見類別的執行緒安全性，這些是實務踩坑的高發區：

- **不可變物件天生安全**：`String`、`Integer` 等一旦建立就不能改變，可放心在多執行緒間共享。
- **`StringBuilder` 與 `StringBuffer`**：前者不同步、效能好，適合單執行緒；後者方法皆 `synchronized`、執行緒安全但較慢。單執行緒一律用 `StringBuilder`。
- **`SimpleDateFormat` 的坑**：它不是執行緒安全的，多執行緒共用同一個實體會解析出錯亂的結果。解法是每次 new、用 `ThreadLocal`，或改用 Java 8 的 `DateTimeFormatter`（不可變且執行緒安全）。

注意　把 `SimpleDateFormat` 宣告成 `static` 共用是經典地雷。它內部持有可變狀態，多執行緒同時 `format` 或 `parse` 會產生錯誤日期甚至拋出例外，而且問題偶發、極難重現。現代程式應改用 `java.time.format.DateTimeFormatter`。

3.12 本章小結

- 程序有獨立記憶體，執行緒共享程序記憶體；共享是高效協作的來源，也是並行 bug 的根源。
- 建立執行緒有繼承 `Thread`、實作 `Runnable`、Lambda 三種方式，建議用 `Runnable` 以分離關注點；啟動要用 `start`。
- 執行緒有 NEW、RUNNABLE、BLOCKED、WAITING、TIMED_WAITING、TERMINATED 六種狀態。
- 中斷是協作式機制；切勿吞掉 `InterruptedException`，應往外拋或恢復中斷旗標。
- 競態條件源於 read-modify-write 非原子；`synchronized` 提供互斥與原子性。
- `volatile` 保證可見性但不保證原子性；旗標用 `volatile`，複合操作用 `synchronized`。
- `wait/notify` 須在持有鎖時呼叫，條件檢查要用 `while` 防虛假喚醒。
- 死結需滿足四條件；固定加鎖順序是最實用的預防策略。
- `SimpleDateFormat` 非執行緒安全，應改用 `DateTimeFormatter`。

3.13 本章練習

動手練習

1. 觀念題：為什麼呼叫 `run()` 與 `start()` 的結果不同？直接呼叫 `run()` 會發生什麼事？（提示：想想是哪個執行緒在執行那段程式碼。）
2. 觀念題：`volatile` 與 `synchronized` 各自保證什麼？為什麼 `volatile int count; count++;` 仍有競態條件？（提示：拆解 `count++` 的三個步驟。）
3. 實作題：用兩個執行緒交替印出 1 到 10（一個印奇數、一個印偶數，且必須按 1、2、3…的順序）。（提示：用 `wait/notify` 與一個共享的「輪到誰」旗標。）
4. 實作題：重現本章的競態條件範例，分別用 `synchronized` 方法與 `synchronized` 區塊修正，並比較兩者臨界區大小的差異。（提示：區塊可只鎖住 `count++` 那一行。）
5. 實作題：把死結範例改寫成不會死結的版本。（提示：讓兩個執行緒都以相同順序取得 `lockA`、`lockB`。）

6. 實作題：寫一個守護執行緒，每 200 毫秒印出目前記憶體使用量，主執行緒工作 1 秒後結束，觀察守護執行緒是否隨之停止。（提示：`setDaemon(true)` 必須在 `start` 之前。）

第 4 章

並行程式設計進階

本章學習目標

學完本章你將能夠：

- 用 `ExecutorService` 管理執行緒池，並正確關閉它。
- 解讀 `ThreadPoolExecutor` 的七個參數，手動配置合適的池。
- 用 `Future` 與 `CompletableFuture` 組合非同步工作。
- 用原子類別與顯式鎖取代 `synchronized`，並理解 CAS 原理。
- 選用適當的並行集合與同步輔助工具。
- 說明虛擬執行緒的優勢與適用場景。

上一章我們用 `new Thread()` 手動建立執行緒。這在教學上很直觀，但在正式系統中幾乎是禁忌。本章介紹 `java.util.concurrent`（簡稱 JUC）套件——由並行大師 Doug Lea 設計的一整套高階工具，讓我們站在巨人的肩膀上寫並行程式。

本章的主線是「從低階原語走向高階抽象」：用執行緒池取代裸執行緒，用原子類別與顯式鎖取代 `synchronized`，用並行集合取代手動同步，最後用 `CompletableFuture` 與虛擬執行緒優雅地表達非同步與高並行。

4.1 手動建立執行緒的問題

為每個任務 `new Thread()` 有三個致命問題。第一，**建立成本高**：每個平台執行緒都對應一個作業系統執行緒，建立與銷毀都要進出核心，還要配置約 1 MB 的堆疊。第二，**無上限**：來一個請求就開一個執行緒，流量一大就可能開出成千上萬個執行緒，直接把記憶體與 CPU 壓垮。第三，**無法管理**：沒有統一的地方控制數量、監控狀態或優雅關閉。

執行緒池（thread pool）一次解決這三個問題：預先建立一批可重複使用的執行緒，任務排隊等候，用固定數量的執行緒消化。這就是 Executor 框架的核心思想。

4.2 Executor 框架

`ExecutorService` 是執行緒池的核心介面。我們透過 `Executors` 工廠類別取得預先配置好的實作。常見的三種工廠方法如下：

工廠方法	特性與適用場景
`newFixedThreadPool(n)`	固定 n 條執行緒，適合負載穩定的場景
`newCachedThreadPool()`	執行緒數依需求增減，適合大量短任務
`newSingleThreadExecutor()`	單一執行緒依序執行，保證任務順序

範例：ExecutorBasics.java ——提交任務與關閉

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * 用固定大小的執行緒池執行多個任務，並示範標準關閉流程。
 */
public class ExecutorBasics {
    public static void main(String[] args) throws InterruptedException {
        // 建立固定 3 條執行緒的池
        ExecutorService pool = Executors.newFixedThreadPool(3);

        // 提交 6 個任務，只有 3 條執行緒輪流消化
        for (int i = 1; i <= 6; i++) {
            int taskId = i;             // 供 Lambda 捕獲的實質不可變值
            pool.submit(() -> {
                String name = Thread.currentThread().getName();
                System.out.println(name + " 執行任務 " + taskId);
            });
        }

        // 標準關閉流程（後述）
        pool.shutdown();                            // 不再接受新任務
        if (!pool.awaitTermination(5, TimeUnit.SECONDS)) {
            pool.shutdownNow();                     // 逾時則強制中斷
        }
        System.out.println("所有任務完成，池已關閉");
    }
}
```

▶ 執行結果

```
pool-1-thread-1 執行任務 1
pool-1-thread-2 執行任務 2
pool-1-thread-3 執行任務 3
```

```
pool-1-thread-1 執行任務 4
pool-1-thread-2 執行任務 5
pool-1-thread-3 執行任務 6
所有任務完成，池已關閉
（註：任務與執行緒的對應順序每次可能不同）
```

關閉執行緒池有固定模板：先 `shutdown()` 停止接收新任務並等既有任務跑完，再用 `awaitTermination` 設定等待上限，若逾時未完成才用 `shutdownNow` 中斷剩餘任務。**忘記關閉執行緒池，程式可能永遠不會結束**，因為池中的非守護執行緒會一直存活。

4.3　ThreadPoolExecutor 七參數

`Executors` 的工廠方法背後都是 `ThreadPoolExecutor`。理解它的七個建構參數，才能真正掌握執行緒池的行為：

參數	意義
corePoolSize	核心執行緒數，即使閒置也保留
maximumPoolSize	最大執行緒數
keepAliveTime	超過核心數的執行緒閒置多久後回收
workQueue	任務等候佇列
threadFactory	建立執行緒的工廠（可命名、設守護）
handler	佇列與執行緒都滿時的拒絕策略
unit	keepAliveTime 的時間單位

> **提示**　《阿里巴巴 Java 開發手冊》明訂：**不要用 `Executors` 的預設工廠方法建立執行緒池**。原因是 `newFixedThreadPool` 與 `newSingleThreadExecutor` 使用無界佇列，任務堆積時可能耗盡記憶體；`newCachedThreadPool` 則可能建立近乎無限的執行緒。正式系統應**直接用 `new ThreadPoolExecutor(...)`**，明確指定有界佇列與拒絕策略，把資源上限握在自己手裡。

範例：CustomPool.java ——手動配置執行緒池

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 手動建立執行緒池：有界佇列 + 明確的拒絕策略。
 * CallerRunsPolicy 讓提交任務的執行緒自己跑，達到「減速」效果。
 */
```

```
public class CustomPool {
    public static void main(String[] args) {
        ThreadPoolExecutor pool = new ThreadPoolExecutor(
                2,                                  // 核心數
                4,                                  // 最大數
                30, TimeUnit.SECONDS,               // 閒置回收時間
                new ArrayBlockingQueue<>(10),       // 有界佇列
                new ThreadPoolExecutor.CallerRunsPolicy());  // 拒絕策略

        for (int i = 1; i <= 5; i++) {
            int id = i;
            pool.execute(() -> System.out.println(
                    Thread.currentThread().getName() + " 處理 " + id));
        }
        pool.shutdown();
    }
}
```

▶ 執行結果

```
pool-1-thread-1 處理 1
pool-1-thread-2 處理 2
pool-1-thread-1 處理 3
pool-1-thread-1 處理 4
pool-1-thread-2 處理 5
（註：實際排程順序可能不同）
```

4.4　Callable 與 Future

`Runnable` 無法回傳結果也不能拋出受檢例外。若任務需要回傳值，改用 `Callable<V>`，它的 `call` 方法回傳 `V` 並可拋例外。提交 `Callable` 會得到一個 `Future<V>`，代表「未來才會有的結果」。

範例：FutureDemo.java ——取得非同步結果

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/**
 * 提交 Callable 取得 Future，get() 會阻塞直到結果就緒。
```

```
 */
public class FutureDemo {
    public static void main(String[] args)
            throws InterruptedException, ExecutionException {
        ExecutorService pool = Executors.newSingleThreadExecutor();

        // 提交一個會回傳整數的任務
        Future<Integer> future = pool.submit(() -> {
            Thread.sleep(100);          // 模擬耗時運算
            return 6 * 7;
        });

        System.out.println("任務已提交，主執行緒可先做別的事");
        // get() 阻塞等待結果；亦可用 get(timeout, unit) 設定逾時
        Integer result = future.get();
        System.out.println("運算結果：" + result);

        pool.shutdown();
    }
}
```

▶ 執行結果

```
任務已提交，主執行緒可先做別的事
運算結果：42
```

Future 的能力有限：get 只能阻塞式等待，也無法把多個 Future 優雅地串接或合併。這些缺點正是 CompletableFuture 要解決的。

4.5 CompletableFuture 非同步管線

CompletableFuture 讓我們用宣告式的方式編排非同步流程：用 supplyAsync 啟動有回傳值的任務、runAsync 啟動無回傳的任務；用 thenApply 轉換結果、thenAccept 消費結果、thenCompose 串接另一個非同步任務、thenCombine 合併兩個任務的結果。

範例：CompletableFutureBasics.java ——串接與合併

```
import java.util.concurrent.CompletableFuture;

/**
 * 示範 thenApply（轉換）、thenCompose（串接依賴）、
 * thenCombine（合併兩個獨立任務）。
```

```
 */
public class CompletableFutureBasics {
    public static void main(String[] args) throws Exception {
        // supplyAsync 在預設池啟動任務，thenApply 對結果做轉換
        CompletableFuture<Integer> f1 = CompletableFuture
                .supplyAsync(() -> 10)
                .thenApply(n -> n * 2);          // 10 → 20

        // thenCompose：前一步結果決定下一個非同步任務（扁平化）
        CompletableFuture<Integer> f2 = f1.thenCompose(
                n -> CompletableFuture.supplyAsync(() -> n + 5));  // 25

        // thenCombine：合併兩個獨立任務的結果
        CompletableFuture<Integer> other =
                CompletableFuture.supplyAsync(() -> 100);
        CompletableFuture<Integer> combined =
                f2.thenCombine(other, (a, b) -> a + b);            // 125

        System.out.println("最終結果：" + combined.get());
    }
}
```

▶ 執行結果

```
最終結果：125
```

4.5.1 例外處理與自訂執行緒池

非同步流程的例外不能用傳統 `try-catch` 攔截，要用 `exceptionally`（只在出錯時提供替代值）或 `handle`（無論成敗都處理）。此外，正式系統應傳入**自訂的執行緒池**，避免佔用所有 `CompletableFuture` 共用的預設 `ForkJoinPool.commonPool()`。

範例：CompletableFutureError.java ——例外處理

```
import java.util.concurrent.CompletableFuture;

/**
 * exceptionally 在上游拋例外時提供替代值，讓管線不中斷。
 */
public class CompletableFutureError {
    public static void main(String[] args) throws Exception {
        CompletableFuture<Integer> future = CompletableFuture
```

```
                .supplyAsync(() -> {
                    if (true) {
                        throw new IllegalStateException("模擬失敗");
                    }
                    return 1;
                })
                .exceptionally(ex -> {          // 攔截例外並回退
                    System.out.println("捕獲：" + ex.getMessage());
                    return -1;                  // 提供預設值
                });

        System.out.println("結果：" + future.get());
    }
}
```

▶ 執行結果

```
捕獲：java.lang.IllegalStateException: 模擬失敗
結果：-1
```

4.5.2 非同步管線實戰

`CompletableFuture` 的真正價值在於「並行取得多個資料來源再合併」。假設要同時查詢使用者基本資料與訂單資料，兩者各需 200 毫秒。若序列執行要 400 毫秒，並行執行只需約 200 毫秒。

範例：AsyncPipeline.java ——並行查詢合併

```
import java.util.concurrent.CompletableFuture;

/**
 * 並行查詢兩個資料來源再合併，總耗時接近「較慢的那個」而非兩者之和。
 */
public class AsyncPipeline {

    /** 模擬耗時查詢，睡眠指定毫秒後回傳結果 */
    static String query(String name, long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
```

```
        return name + "資料";
    }

    public static void main(String[] args) throws Exception {
        long start = System.currentTimeMillis();

        // 兩個查詢同時啟動
        CompletableFuture<String> user =
                CompletableFuture.supplyAsync(() -> query("使用者", 200));
        CompletableFuture<String> order =
                CompletableFuture.supplyAsync(() -> query("訂單", 200));

        // 合併兩者結果
        String merged = user.thenCombine(order,
                (u, o) -> u + " + " + o).get();

        long cost = System.currentTimeMillis() - start;
        System.out.println("合併結果：" + merged);
        System.out.println("總耗時約：" + cost + " 毫秒（並行，非 400）");
    }
}
```

▶ 執行結果

```
合併結果：使用者資料 + 訂單資料
總耗時約：210 毫秒（並行，非 400）
```

4.6 原子類別

上一章的計數器競態，除了 `synchronized`，更輕量的解法是原子類別。`AtomicInteger` 的 `incrementAndGet` 是硬體層級的原子操作，無需加鎖就能保證正確：

範例：AtomicCounter.java ——免鎖計數

```
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 用 AtomicInteger 解決計數器競態，無需 synchronized。
 */
public class AtomicCounter {
    static final AtomicInteger count = new AtomicInteger(0);

```

```
    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 100_000; i++) {
                count.incrementAndGet();    // 原子的「加一並取值」
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("預期：200000，實際：" + count.get());
    }
}
```

▶ 執行結果

```
預期：200000，實際：200000
```

> **說明**　原子類別背後是 CAS（Compare-And-Swap，比較並交換）：更新時同時提供「舊值」與「新值」，硬體確保「只有當目前值等於舊值時，才寫入新值」，整個過程不可分割。CAS 是「樂觀鎖」，失敗就重試，不會像 `synchronized` 那樣阻塞執行緒。但它有著名的 ABA 問題：值從 A 變 B 再變回 A，CAS 會誤以為沒變過。若在意過程，可用 `AtomicStampedReference` 加上版本戳記。極高並行的計數場景，`LongAdder` 比 `AtomicLong` 更快，因為它把熱點分散到多個單元（cell）分別累加，最後才求和，大幅降低 CAS 競爭。

4.7　顯式鎖

`ReentrantLock` 是比 `synchronized` 更靈活的顯式鎖：它支援可中斷的取鎖、帶逾時的 `tryLock` ↪ 、以及公平鎖。代價是必須**手動解鎖**，且 `unlock` 一定要放在 `finally` 中，否則例外發生時鎖將永遠不釋放。

範例：TryLockDemo.java ——帶逾時避免死結

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
```

```

/**
 * tryLock 帶逾時：取不到鎖就放棄，從根本上避免無限等待造成的死結。
 */
public class TryLockDemo {
    static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            try {
                // 最多等 500 毫秒，取不到就放棄
                if (lock.tryLock(500, TimeUnit.MILLISECONDS)) {
                    try {
                        System.out.println(
                                Thread.currentThread().getName() + " 取得鎖");
                        Thread.sleep(800);      // 持鎖 800 毫秒，超過對方逾時
                    } finally {
                        lock.unlock();          // 解鎖務必放 finally
                    }
                } else {
                    System.out.println(
                            Thread.currentThread().getName() + " 放棄取鎖");
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        };

        new Thread(task, "A").start();
        new Thread(task, "B").start();
    }
}
```

▶ 執行結果

```
A 取得鎖
B 放棄取鎖
（註：誰先取得鎖不固定；後者因逾時而放棄）
```

比較	`synchronized`	`ReentrantLock`
解鎖方式	自動	手動（須 `finally`）
可中斷取鎖	否	是
帶逾時取鎖	否	是（`tryLock`）
公平性選擇	否	可設公平鎖

當讀操作遠多於寫操作時，`ReentrantReadWriteLock` 允許多個讀執行緒同時持有讀鎖，只有寫時才互斥，能大幅提升並行度。

4.8 並行集合

`HashMap` 非執行緒安全，而老舊的 `Hashtable` 與 `Collections.synchronizedMap` 對每個方法整包加鎖、效能差。首選是 `ConcurrentHashMap`，它用更細緻的鎖與 CAS，並提供 `computeIfAbsent`、`merge` 等**原子複合操作**：

範例：ConcurrentMapDemo.java ——原子複合操作

```
import java.util.concurrent.ConcurrentHashMap;

/**
 * 用 ConcurrentHashMap 的 merge 做執行緒安全的計數。
 * merge 把「取值、相加、寫回」合併為一個原子操作。
 */
public class ConcurrentMapDemo {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Integer> counter =
                new ConcurrentHashMap<>();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                // 若不存在則設為 1，否則把舊值加 1（原子）
                counter.merge("hits", 1, Integer::sum);
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
```

```
        System.out.println("hits (預期 2000) ：" + counter.get("hits"));
    }
}
```

▶ 執行結果

```
hits (預期 2000) ：2000
```

另外，CopyOnWriteArrayList 適合「讀多寫極少」的場景（如監聽器清單），每次寫入都複製整個陣列，因此讀取完全無鎖。而 BlockingQueue 是實作生產者消費者的利器——用它重寫上一章的範例，程式碼大幅精簡，因為 put 在滿時自動阻塞、take 在空時自動阻塞，不必再手寫 wait/notify：

範例：BlockingQueueDemo.java ——精簡的生產者消費者

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

/**
 * 用 BlockingQueue 重寫生產者消費者：阻塞語意內建，無需手動同步。
 */
public class BlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(3);

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    queue.put(i);       // 佇列滿則自動阻塞
                    System.out.println("生產：" + i);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    Integer v = queue.take();   // 佇列空則自動阻塞
                    System.out.println("    消費：" + v);
                }
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }
}
```

▶ 執行結果

```
生產：1
生產：2
    消費：1
生產：3
    消費：2
（註：交錯順序每次不同，總量皆為 5）
```

4.9 同步輔助工具

JUC 還提供幾個常用的協調工具。CountDownLatch 讓一個執行緒等待多個工作全部完成（倒數計數歸零才放行）；Semaphore 是「號誌」，限制同時存取資源的執行緒數量（限流）；CyclicBarrier 讓一組執行緒互相等待，全部到達「柵欄」才一起繼續。

範例：CountDownLatchDemo.java ——等待多工完成

```
import java.util.concurrent.CountDownLatch;

/**
 * 主執行緒用 CountDownLatch 等待 3 個子任務全部完成後才彙總。
 */
public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        int workers = 3;
        CountDownLatch latch = new CountDownLatch(workers);

        for (int i = 1; i <= workers; i++) {
            int id = i;
            new Thread(() -> {
```

```
                try {
                    Thread.sleep(id * 50L);     // 模擬不同耗時
                    System.out.println("子任務 " + id + " 完成");
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    latch.countDown();          // 完成即計數減一
                }
            }).start();
        }

        latch.await();                          // 阻塞直到歸零
        System.out.println("所有子任務完成，開始彙總");
    }
}
```

▶ 執行結果

```
子任務 1 完成
子任務 2 完成
子任務 3 完成
所有子任務完成，開始彙總
```

4.10 虛擬執行緒（Java 21）

Java 21 正式推出的虛擬執行緒（virtual thread）是並行程式設計的重大變革。傳統的平台執行緒（platform thread）一對一對應作業系統執行緒，昂貴而有限；虛擬執行緒則由 JVM 管理，極其輕量，可以輕鬆建立數百萬個。當虛擬執行緒遇到阻塞的 I/O，JVM 會把它從底層的載體執行緒上「卸載」，讓載體去跑別的虛擬執行緒，因此少數幾個作業系統執行緒就能支撐海量並行。

範例：VirtualThreadDemo.java ——建立虛擬執行緒

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * 用虛擬執行緒池執行大量 I/O 密集任務。
 * 每個任務一條虛擬執行緒，不必也不該池化平台執行緒。
 */
```

```java
public class VirtualThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        // 每個任務配一條虛擬執行緒的 ExecutorService
        try (ExecutorService vpool =
                Executors.newVirtualThreadPerTaskExecutor()) {
            for (int i = 1; i <= 5; i++) {
                int id = i;
                vpool.submit(() -> {
                    try {
                        Thread.sleep(100);      // 模擬 I/O 等待
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                    System.out.println("虛擬執行緒完成任務 " + id);
                });
            }
        }   // try-with-resources 自動等待所有任務並關閉
        System.out.println("全部完成");
    }
}
```

▶ 執行結果

```
虛擬執行緒完成任務 3
虛擬執行緒完成任務 1
虛擬執行緒完成任務 5
虛擬執行緒完成任務 2
虛擬執行緒完成任務 4
全部完成
（註：完成順序不固定）
```

說明　虛擬執行緒適合 **I/O 密集**的工作（大量網路呼叫、資料庫查詢），因為它們大部分時間在等待，卸載機制能讓少量載體執行緒服務海量任務。但它不適合 **CPU 密集**的工作——純運算不會阻塞、無從卸載，此時該用與核心數相當的平台執行緒池。還有一個觀念要更新：虛擬執行緒**不需要池化**。池化是為了重用昂貴的平台執行緒，而虛擬執行緒建立成本極低，「每個任務開一條、用完即丟」才是正確用法，故用 `newVirtualThreadPerTaskExecutor`。

4.11 本章小結

- 手動 `new Thread()` 成本高、無上限、難管理；執行緒池預建並重用執行緒來解決。
- `ExecutorService` 用 `shutdown` + `awaitTermination` + `shutdownNow` 標準流程關閉。
- 正式系統應手動 `new ThreadPoolExecutor`，用有界佇列與明確的拒絕策略，勿依賴 `Executors` 預設工廠。
- `Callable+Future` 可回傳結果；`CompletableFuture` 用 `thenApply/thenCompose/thenCombine` 編排非同步管線，用 `exceptionally/handle` 處理例外。
- 原子類別以 CAS 免鎖更新；注意 ABA 問題，高並行計數用 `LongAdder`。
- `ReentrantLock` 支援可中斷、帶逾時、公平鎖，`unlock` 必須放 `finally`；讀多寫少用 `ReadWriteLock`。
- `ConcurrentHashMap` 的 `merge/computeIfAbsent` 是原子複合操作；`BlockingQueue` 大幅簡化生產者消費者。
- 虛擬執行緒輕量、適合 I/O 密集、不需池化；CPU 密集仍用平台執行緒池。

4.12 本章練習

動手練習

1. 觀念題：為什麼《阿里巴巴 Java 開發手冊》不建議用 `Executors` 的預設工廠方法？可能造成什麼後果？（提示：想想無界佇列與無限執行緒。）
2. 觀念題：`thenApply` 與 `thenCompose` 差在哪裡？什麼時候必須用 `thenCompose`？（提示：回傳的是「值」還是「另一個 `CompletableFuture`」。）
3. 實作題：用 `CompletableFuture` 並行呼叫三個「查詢」方法（各睡不同時間），全部完成後印出合併結果，並量測總耗時。（提示：`CompletableFuture.allOf` 搭配各自的 `join`。）
4. 實作題：用 `AtomicInteger` 與 `synchronized` 兩種方式各實作一次計數器，並比較高並行下的效能差異。（提示：用 `System.nanoTime` 量測。）
5. 實作題：用 `Semaphore` 限制同時只有 2 個執行緒能存取某資源，觀察第 3 個執行緒如何等待。（提示：`acquire` 與 `release` 成對出現。）
6. 實作題：用虛擬執行緒同時發出 100 個「模擬 I/O」任務（各睡 100 毫秒），觀察總耗時遠小於序列執行的 10 秒。（提示：`newVirtualThreadPerTaskExecutor`。）

第 5 章

反射與標註

> **本章學習目標**
>
> 學完本章你將能夠：
>
> - 說明反射的用途，並用三種方式取得 `Class` 物件。
> - 用反射檢視類別的欄位、方法與建構子，解析修飾詞。
> - 動態建立實體、呼叫方法、存取私有成員，並理解其代價。
> - 自訂標註，運用 `@Retention`、`@Target` 等元標註。
> - 在執行期讀取標註，打造迷你測試框架。
> - 用反射實作簡易依賴注入，理解 Spring 的核心原理。

到目前為止，我們寫的程式都在「編譯期」就決定了要用哪個類別、呼叫哪個方法。反射（reflection）打破了這個限制：它讓程式在**執行期**才動態地檢視與操作類別、方法與欄位。這聽起來很抽象，但你天天在用的框架幾乎都靠它運作——Spring 的依賴注入、Jackson 的 JSON 序列化、JUnit 的測試探索，底層都是反射。

反射與標註（Annotation）是一對好搭檔：標註為程式碼「貼上標籤」，反射則在執行期「讀取標籤」並據以行動。本章最後會用這對組合，親手打造迷你版的 JUnit 與 Spring，讓你徹底看穿框架的魔法。

5.1 反射是什麼

一般情況下，若要使用 `ArrayList`，我們會 `import` 它、`new` 它、呼叫它的方法——這些都在編譯期綁定。但框架的作者面對的是「使用者將來才會寫、我現在還看不到」的類別。它們無法 `import` 一個還不存在的類別，只能靠反射在執行期動態處理。反射的典型場景包括：框架（Spring、Hibernate）、序列化工具（把物件轉成 JSON）、以及測試工具（自動找出並執行測試方法）。

5.2 Class 物件：反射的入口

反射的一切都從 `Class` 物件開始。JVM 為每個載入的類別建立**唯一一個** `Class` 物件，它記錄了該類別的全部結構資訊。取得它有三種方式：

範例：GetClassObject.java ——三種取得方式

```java
/**
 * 三種取得 Class 物件的方式，並驗證其唯一性。
 */
public class GetClassObject {
    public static void main(String[] args) throws ClassNotFoundException {
        // 方式一：類別字面值（編譯期已知型別，最安全）
        Class<String> c1 = String.class;

        // 方式二：物件的 getClass（已有實體時）
        String s = "hello";
        Class<?> c2 = s.getClass();

        // 方式三：Class.forName（執行期用全名字串載入，最動態）
        Class<?> c3 = Class.forName("java.lang.String");

        // 三者指向同一個 Class 物件（唯一性）
        System.out.println("c1 == c2？" + (c1 == c2));
        System.out.println("c2 == c3？" + (c2 == c3));
        System.out.println("類別名稱：" + c1.getName());
    }
}
```

▶ 執行結果

```
c1 == c2？true
c2 == c3？true
類別名稱：java.lang.String
```

三種方式各有適用時機：`.class` 用於編譯期已知型別；`getClass` 用於手上已有物件；`Class.forName` 最動態，適合「類別全名來自設定檔或字串」的情境，例如 JDBC 載入驅動。

5.3　檢視類別結構

取得 `Class` 物件後，就能查詢它的欄位、方法與建構子。這裡要先釐清一組容易混淆的方法：

方法家族	是否含私有成員	是否含繼承成員
`getFields()` / `getMethods()`	否（僅 public）	是
`getDeclaredFields()` 等	是（含 private）	否（僅本類別宣告）

換言之，`getDeclared*` 看的是「這個類別自己宣告的所有成員（含私有）」，而 `get*` 看的是「這個類別對外公開的成員（含繼承來的）」。以下走訪一個類別的完整結構：

範例：InspectClass.java ——走訪結構

```java
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

/**
 * 用反射走訪類別的欄位與方法，並解析其修飾詞。
 */
public class InspectClass {

    /** 被檢視的目標類別 */
    static class Account {
        private double balance;         // 私有欄位
        public String owner;            // 公開欄位

        public void deposit(double amount) { balance += amount; }
        private void audit() { }
    }

    public static void main(String[] args) {
        Class<?> c = Account.class;

        System.out.println("== 所有宣告的欄位 ==");
        for (Field f : c.getDeclaredFields()) {
            // Modifier 把修飾詞的整數位元解析成可讀字串
            String mod = Modifier.toString(f.getModifiers());
            System.out.println("  " + mod + " "
                    + f.getType().getSimpleName() + " " + f.getName());
        }

        System.out.println("== 所有宣告的方法 ==");
        for (Method m : c.getDeclaredMethods()) {
            System.out.println("  " + m.getName()
                    + "，參數個數：" + m.getParameterCount());
        }
    }
}
```

▶ 執行結果

```
== 所有宣告的欄位 ==
  private double balance
  public String owner
```

```
== 所有宣告的方法 ==
  deposit，參數個數：1
  audit，參數個數：0
```

5.4 動態建立與呼叫

反射不只能「看」，還能「動」。我們可以動態建立實體、呼叫方法，甚至存取私有成員。以下範例完整示範這條流水線：

範例：DynamicInvoke.java ——建立實體與呼叫方法

```java
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

/**
 * 動態建立實體、呼叫方法、存取私有欄位的完整示範。
 */
public class DynamicInvoke {

    static class Greeter {
        private String name;                // 私有欄位
        public Greeter(String name) { this.name = name; }
        public String greet(String lang) {  // 有參數、有回傳
            return ("zh".equals(lang) ? "你好，" : "Hello, ") + name;
        }
    }

    public static void main(String[] args) throws Exception {
        Class<?> c = Greeter.class;

        // 1. 動態呼叫建構子建立實體
        Constructor<?> ctor = c.getDeclaredConstructor(String.class);
        Object obj = ctor.newInstance("Alice");

        // 2. 動態呼叫方法（方法名 + 參數型別，再傳入實際引數）
        Method greet = c.getMethod("greet", String.class);
        Object result = greet.invoke(obj, "zh");
        System.out.println("方法回傳：" + result);

        // 3. 存取並修改私有欄位（需先開權限）
```

```
        Field nameField = c.getDeclaredField("name");
        nameField.setAccessible(true);      // 破除私有存取限制
        nameField.set(obj, "Bob");          // 直接改寫私有欄位
        System.out.println("改名後：" + greet.invoke(obj, "en"));
    }
}
```

▶ 執行結果

```
方法回傳：你好，Alice
改名後：Hello, Bob
```

注意　`setAccessible(true)` 能繞過 `private` 限制，直接讀寫私有成員。這是把雙面刃：框架靠它注入依賴、還原序列化物件，但濫用會徹底破壞封裝，讓外部程式依賴實作細節，一旦內部欄位改名就全盤崩潰。此外，自 Java 9 模組系統之後，若目標類別所在的模組沒有對外 `opens`，強行 `setAccessible` 會拋出 `InaccessibleObjectException`。請只在框架層級、且確有必要時使用。

5.5 反射的成本

反射威力強大，但絕非免費午餐。

說明　反射有三重代價。**效能**：反射呼叫比直接呼叫慢（需動態解析、無法內聯），雖然現代 JVM 已大幅最佳化，熱路徑上仍應避免。**失去編譯期檢查**：方法名與欄位名變成字串，打錯字要到執行期才爆炸，編譯器幫不了你。**重構不安全**：IDE 的「重新命名」無法追蹤字串裡的方法名，改名後反射程式碼默默失效。結論：**反射是框架的工具，不是日常業務邏輯的工具**。能用一般呼叫解決的，就不要用反射。

5.6 標註（Annotation）

標註是加在程式元素上的**中繼資料**（metadata），它本身不改變程式邏輯，而是「附加資訊」給編譯器或框架讀取。注意術語：`@Override` 這類叫「標註」（Annotation），而 `//` 開頭那種給人看的文字叫「註解」（comment），兩者截然不同。Java 內建的標註如下：

內建標註	用途
`@Override`	標示覆寫父類方法，寫錯簽章時編譯報錯
`@Deprecated`	標示已棄用，使用時編譯警告
`@SuppressWarnings`	抑制指定的編譯警告
`@FunctionalInterface`	標示函數式介面（見第 1 章）

5.6.1 自訂標註

用 `@interface` 定義自己的標註。標註可以有「元素」（看起來像方法），並能指定預設值。若元素名為 `value` 且只有它一個，使用時可省略元素名。

範例：自訂標註的定義

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 自訂標註 @Column：用來標示欄位對應的資料庫欄位名與是否可為空。
 * RUNTIME 保留策略讓反射在執行期能讀到它。
 */
@Retention(RetentionPolicy.RUNTIME)     // 執行期仍保留
@Target(ElementType.FIELD)              // 只能加在欄位上
@interface Column {
    String name();                      // 無預設值，使用時必填
    boolean nullable() default true;    // 有預設值，可省略
}
```

5.6.2 元標註

「元標註」（meta-annotation）是「用來標註標註」的標註。最重要的是 `@Retention`，它決定標註保留到哪個階段：

保留策略	意義
`SOURCE`	只在原始碼，編譯後丟棄（如 `@Override`）
`CLASS`	保留到 class 檔，但執行期不載入（預設值）
`RUNTIME`	保留到執行期，反射可讀取（框架最常用）

其他常用元標註：`@Target` 限定標註可用的位置（類別、方法、欄位等）；`@Inherited` 讓標註可被子類別繼承；`@Documented` 讓標註出現在 Javadoc；`@Repeatable` 允許同一位置重複

使用同一標註。

> **注意**　若想在執行期用反射讀取標註，其 `@Retention` 必須是 **`RUNTIME`**。這是最常見的陷阱：忘了加 `@Retention(RUNTIME)`，標註在編譯後就消失，反射永遠讀不到，框架自然「沒反應」，還很難察覺問題出在哪。

5.6.3 執行期讀取標註

在執行期，用 `isAnnotationPresent` 判斷是否有某標註，用 `getAnnotation` 取得標註實體並讀取其元素值：

範例：ReadAnnotation.java ——讀取標註

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Field;

/**
 * 用反射讀取欄位上的 @Column 標註，模擬 ORM 的欄位對應。
 */
public class ReadAnnotation {

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.FIELD)
    @interface Column {
        String name();
        boolean nullable() default true;
    }

    static class User {
        @Column(name = "user_id", nullable = false)
        private Long id;

        @Column(name = "user_name")
        private String username;
    }

    public static void main(String[] args) {
        for (Field f : User.class.getDeclaredFields()) {
            // 先確認欄位上有 @Column 才讀取
```

```
            if (f.isAnnotationPresent(Column.class)) {
                Column col = f.getAnnotation(Column.class);
                System.out.println(f.getName() + " → 欄位 " + col.name()
                        + "（可為空：" + col.nullable() + "）");
            }
        }
    }
}
```

▶ 執行結果

```
id → 欄位 user_id（可為空：false）
username → 欄位 user_name（可為空：true）
```

5.7 綜合實戰一：迷你測試框架

現在把反射與標註合起來用。JUnit 的核心原理其實很單純：定義一個 `@Test` 標註，然後用反射掃描類別、找出所有標了 `@Test` 的方法、逐一執行並統計通過與失敗。我們親手實作一個迷你版：

範例：MiniTest.java ——仿 JUnit 測試框架

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

/**
 * 迷你測試框架：掃描 @MiniTest 方法、執行、統計通過與失敗。
 */
public class MiniTest {

    /** 標示「這是一個測試方法」 */
    @Retention(RetentionPolicy.RUNTIME)
    @interface Case { }

    /** 被測試的類別，混合了會通過與會失敗的案例 */
    static class Calculator {
        @Case
        public void addShouldWork() {
            assertTrue(1 + 1 == 2, "1+1 應為 2");
        }
        @Case
```

```
        public void divideShouldFail() {
            assertTrue(10 / 2 == 6, "故意寫錯的斷言");
        }
    }

    /** 簡易斷言：不成立就丟例外 */
    static void assertTrue(boolean cond, String msg) {
        if (!cond) {
            throw new AssertionError(msg);
        }
    }

    public static void main(String[] args) throws Exception {
        Class<?> c = Calculator.class;
        Object instance = c.getDeclaredConstructor().newInstance();
        int pass = 0, fail = 0;

        // 掃描所有方法，只執行標了 @Case 的
        for (Method m : c.getDeclaredMethods()) {
            if (!m.isAnnotationPresent(Case.class)) {
                continue;
            }
            try {
                m.invoke(instance);          // 反射呼叫測試方法
                System.out.println("[通過] " + m.getName());
                pass++;
            } catch (Exception e) {
                // invoke 會把方法內的例外包成 InvocationTargetException
                System.out.println("[失敗] " + m.getName()
                        + " → " + e.getCause().getMessage());
                fail++;
            }
        }
        System.out.println("結果：通過 " + pass + "，失敗 " + fail);
    }
}
```

▶ 執行結果

```
[通過] addShouldWork
[失敗] divideShouldFail → 故意寫錯的斷言
結果：通過 1，失敗 1
（註：方法掃描順序不保證，通過與失敗的行序可能對調）
```

看穿了嗎？JUnit 並不神祕——它就是「標註 + 反射掃描 + 動態呼叫」。理解這一點，你對所有註解驅動的框架都會有全新的視野。

5.8 綜合實戰二：簡易依賴注入

Spring 最核心的功能「依賴注入」（Dependency Injection）同樣是反射的應用。原理是：容器掃描物件的欄位，看到 `@Inject` 標註就自動建立對應的物件、用反射塞進該欄位。使用者不必自己 `new` 依賴，達到「控制反轉」。

範例：MiniInjector.java ——迷你依賴注入

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Field;

/**
 * 迷你 DI 容器：掃描 @Inject 欄位並自動注入實體。
 * 這正是 Spring 依賴注入的最小核心。
 */
public class MiniInjector {

    @Retention(RetentionPolicy.RUNTIME)
    @interface Inject { }

    /** 一個被依賴的服務 */
    static class MessageService {
        String send(String msg) { return "已送出：" + msg; }
    }

    /** 依賴 MessageService 的元件，用 @Inject 宣告需求 */
    static class NotificationClient {
        @Inject
        MessageService service;             // 由容器自動注入

        void notifyUser() {
            System.out.println(service.send("歡迎加入！"));
        }
    }

    /** 極簡容器：為每個 @Inject 欄位建立並注入實體 */
    static void inject(Object target) throws Exception {
        for (Field f : target.getClass().getDeclaredFields()) {
```

```
            if (f.isAnnotationPresent(Inject.class)) {
                // 依欄位型別建立實體
                Object dep = f.getType()
                        .getDeclaredConstructor().newInstance();
                f.setAccessible(true);
                f.set(target, dep);         // 注入
            }
        }
    }

    public static void main(String[] args) throws Exception {
        NotificationClient client = new NotificationClient();
        inject(client);                     // 交給容器完成注入
        client.notifyUser();                // service 已就緒
    }
}
```

▶ 執行結果

```
已送出：歡迎加入！
```

`NotificationClient` 從頭到尾沒有 `new MessageService()`，依賴卻自動就位。這就是控制反轉的精髓：物件不再自己掌控依賴的建立，而是交給容器。真正的 Spring 當然複雜得多（作用域、生命週期、循環依賴、AOP），但最核心的魔法，你現在已經親手實作過了。

5.9 本章小結

- 反射讓程式在執行期動態檢視與操作類別，是框架、序列化、測試工具的基石。
- `Class` 物件是反射入口，可用 `.class`、`getClass`、`Class.forName` 取得，同一類別的 `Class` 物件唯一。
- `getDeclared*` 含私有但不含繼承；`get*` 含繼承但僅 public。
- 反射可動態建立實體（`newInstance`）、呼叫方法（`invoke`）、存取私有成員（`setAccessible`）。
- 反射有效能、失去編譯期檢查、重構不安全三重代價，應限於框架層級。
- 標註是中繼資料；欲在執行期以反射讀取，`@Retention` 必須為 `RUNTIME`。
- 自訂標註用 `@interface`，可有元素與預設值；元標註有 `@Retention`、`@Target` 等。
- 「標註 + 反射」是 JUnit 與 Spring 的共同核心：掃描、判斷、動態呼叫或注入。

5.10 本章練習

動手練習

1. 觀念題：`getMethods()` 與 `getDeclaredMethods()` 有何差異？各在什麼情況下使用？（提示：一個看繼承與公開，一個看本類別與私有。）
2. 觀念題：為什麼想被反射讀取的標註一定要用 `@Retention(RUNTIME)`？若用預設的 `CLASS` 會怎樣？（提示：想想標註在各階段是否還存在。）
3. 實作題：寫一個方法，接收任意物件，用反射印出它所有欄位的名稱與目前值（含私有欄位）。（提示：`getDeclaredFields` 搭配 `setAccessible`。）
4. 實作題：擴充迷你測試框架，加入 `@Before` 標註，讓每個測試方法執行前先跑一次初始化方法。（提示：先找出 `@Before` 方法，在每次 `invoke` 前呼叫它。）
5. 實作題：定義 `@Range(min, max)` 標註加在整數欄位上，寫一個驗證器用反射檢查欄位值是否在範圍內。（提示：讀取標註的 `min`/`max` 再與欄位值比較。）
6. 實作題：為迷你 DI 容器加入「單例快取」，讓相同型別的依賴只建立一次、重複注入同一個實體。（提示：用 `Map<Class<?>, Object>` 快取已建立的實體。）

第 6 章

JVM 深入剖析

本章學習目標

學完本章你將能夠：

- 描述 JVM 的整體架構與類別載入的三大階段。
- 說明雙親委派模型的運作與存在理由。
- 分辨執行期資料區各區塊，並解釋兩種 `Error` 的成因。
- 說明可達性分析、GC Roots 與四種引用強度。
- 比較分代收集與各種垃圾收集器的特性。
- 使用 jstack 等診斷工具，並找出記憶體洩漏。

前五章我們專注在「怎麼寫 Java」，本章要往下鑽一層，看看「Java 程式如何被執行」。理解 JVM 不是為了炫技，而是實務所需：當線上服務發生記憶體溢位、GC 停頓過長、或詭異的 `ClassNotFoundException` 時，唯有懂 JVM 才能快速定位問題。

我們會依循程式的生命週期展開：先看類別如何被載入進 JVM，再看執行時資料存放在哪些記憶體區域，接著深入垃圾回收如何自動管理記憶體，最後用一組診斷工具把前面的理論落實到實戰。

6.1 JVM 整體架構

JVM 由四大子系統構成，資料在其中流動：

- **類別載入子系統**：負責把 `.class` 位元組碼載入記憶體，完成連結與初始化。
- **執行期資料區**：JVM 執行時的記憶體，含堆積、堆疊、方法區等。
- **執行引擎**：解讀並執行位元組碼，內含直譯器、JIT 即時編譯器與垃圾收集器。
- **本地方法介面**（JNI）：讓 Java 呼叫以 C/C++ 撰寫的原生方法。

6.2 類別載入過程

一個類別從 `.class` 檔到可被使用，要經過**載入、連結、初始化**三大階段，其中連結又細分為驗證、準備、解析：

- **載入**（Loading）：讀取位元組碼，在方法區建立該類別的結構，並產生代表它的 `Class` 物件。
- **驗證**（Verification）：確認位元組碼符合規範、不會危害 JVM。
- **準備**（Preparation）：為靜態變數配置記憶體並設為**預設值**（如 `int` 為 0），尚未執行程式設定的初值。
- **解析**（Resolution）：把常量池中的符號參考轉為直接參考。
- **初始化**（Initialization）：執行靜態初始化區塊與靜態變數的賦值，此時 `int` 才變成程式指定的值。

6.2.1 初始化的觸發時機

類別的初始化是**惰性的**：只有在「主動使用」時才觸發，例如 `new` 實體、呼叫靜態方法、存取靜態變數（常數除外）。而「被動使用」不會觸發初始化。一個經典例子是：透過子類別存取父類別的靜態變數，只會初始化父類別；而存取 `static final` 的**編譯期常數**根本不會觸發任何初始化，因為常數在編譯期就被內聯到使用處了。

範例：ClassInit.java ——static final 常數不觸發初始化

```
/**
 * 觀察類別初始化的觸發時機。
 * 存取編譯期常數不會觸發類別初始化（值已在編譯期內聯）。
 */
public class ClassInit {

    static class Holder {
        // 編譯期常數：存取它不會初始化 Holder
        static final int CONST = 42;
        // 非常數靜態變數：存取它會觸發初始化
        static int counter = compute();

        static {
            System.out.println(">> Holder 被初始化了");
        }
        static int compute() {
            System.out.println(">> counter 正在計算");
            return 100;
        }
    }

    public static void main(String[] args) {
        // 存取編譯期常數：不會看到任何初始化訊息
```

```
        System.out.println("常數 CONST = " + Holder.CONST);
        System.out.println("--- 分隔線 ---");
        // 存取非常數靜態變數：觸發初始化
        System.out.println("counter = " + Holder.counter);
    }
}
```

▶ 執行結果

```
常數 CONST = 42
--- 分隔線 ---
>> counter 正在計算
>> Holder 被初始化了
counter = 100
```

6.3 類別載入器與雙親委派

類別由「類別載入器」（class loader）負責載入，它們形成階層：

- **Bootstrap ClassLoader**：載入 JDK 核心類別（如 `java.lang.*`），由 C++ 實作，在 Java 中顯示為 `null`。
- **Platform ClassLoader**：載入 JDK 的擴充模組。
- **Application ClassLoader**：載入 classpath 上我們自己的類別。

範例：ClassLoaderDemo.java ——觀察載入器階層

```
/**
 * 觀察不同類別由哪個載入器載入。
 * 核心類別由 Bootstrap 載入，故顯示為 null。
 */
public class ClassLoaderDemo {
    public static void main(String[] args) {
        // 自己寫的類別：由 Application ClassLoader 載入
        System.out.println("本類別：" +
                ClassLoaderDemo.class.getClassLoader());
        // 核心類別 String：由 Bootstrap 載入，顯示 null
        System.out.println("String：" +
                String.class.getClassLoader());
        // 印出 Application ClassLoader 的父載入器
        ClassLoader app = ClassLoaderDemo.class.getClassLoader();
        System.out.println("父載入器：" + app.getParent());
```

```
    }
}
```

▶ 執行結果

```
本類別：jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc
String：null
父載入器：jdk.internal.loader.ClassLoaders$PlatformClassLoader@1b6d3586
```

> **說明**　雙親委派模型：當一個載入器收到載入請求時，它不會馬上自己動手，而是先往上委派給父載入器，父載入器再往上，直到 Bootstrap。只有當上層都表示「載不到」時，才由下層嘗試。這樣設計的理由是安全與一致：確保核心類別（如 `java.lang.String`）永遠由 Bootstrap 載入，即使有人惡意寫了一個同名的 `java.lang.String` 放進 classpath，也不會被載入，從而無法替換或汙染核心類別。同時也保證同一類別在 JVM 中只有一份，不會重複載入。

6.4　執行期資料區

JVM 執行時把記憶體劃分成數個區域，有的執行緒私有、有的全體共享：

- **程式計數器**（PC Register，執行緒私有）：記錄目前執行到哪條位元組碼指令。
- **JVM 堆疊**（Stack，執行緒私有）：每呼叫一個方法就壓入一個「堆疊框」（stack frame），內含區域變數表與運算元堆疊。
- **堆積**（Heap，全體共享）：所有物件實體的家，也是 GC 的主戰場。
- **方法區**（Method Area，全體共享）：存放類別結構、常數、靜態變數等。

6.4.1　堆疊與 StackOverflowError

每次方法呼叫都會壓入一個堆疊框，方法返回就彈出。若遞迴沒有正確的終止條件，堆疊框無限累積，就會撐爆堆疊：

範例：StackOverflow.java ——無窮遞迴

```
/**
 * 無終止條件的遞迴會不斷壓入堆疊框，最終撐爆 JVM 堆疊。
 */
public class StackOverflow {
    static int depth = 0;

    /** 每次呼叫自己，堆疊框只增不減 */
```

```
    static void recurse() {
        depth++;
        recurse();                  // 沒有終止條件
    }

    public static void main(String[] args) {
        try {
            recurse();
        } catch (StackOverflowError e) {
            // Error 通常不該捕捉，這裡僅為教學觀察深度
            System.out.println("堆疊溢位，遞迴深度約：" + depth);
        }
    }
}
```

▶ 執行結果

```
堆疊溢位，遞迴深度約：21847
（註：實際深度視 -Xss 堆疊大小與方法框大小而定）
```

6.4.2 堆積與 OutOfMemoryError

堆積存放所有物件。若不斷建立物件又一直被參考（無法回收），堆積終將耗盡：

範例：HeapOOM.java ——堆積溢位

```
import java.util.ArrayList;
import java.util.List;

/**
 * 不斷加入物件到 List，物件都被 List 參考而無法回收，
 * 最終耗盡堆積，拋出 OutOfMemoryError。
 * 建議搭配 -Xmx64m 執行以快速觀察。
 */
public class HeapOOM {
    public static void main(String[] args) {
        List<byte[]> list = new ArrayList<>();
        int count = 0;
        try {
            while (true) {
                list.add(new byte[1024 * 1024]);    // 每次 1 MB
                count++;
```

```
            }
        } catch (OutOfMemoryError e) {
            // 先放開參考讓 GC 能回收，否則連印訊息的記憶體都沒有
            list = null;
            System.out.println("堆積溢位，已配置約 " + count + " MB");
        }
    }
}
```

▶ 執行結果

```
堆積溢位，已配置約 58 MB
（註：實際數字取決於 -Xmx 設定與所用的垃圾收集器）
```

> **說明**　方法區在 Java 8 有重大變化：舊版用「永久代」（PermGen）實作，容易因類別過多而溢位；Java 8 起改用 **Metaspace**（元空間），改存於本地記憶體，預設不設上限（可用 `-XX:MaxMetaspaceSize` 限制）。此外，字串常量池在 Java 7 起已從方法區移到堆積，因此 `String.intern()` 的行為與早期版本不同。

6.5　物件的建立與記憶體佈局

當 JVM 遇到 `new` 指令，它會先檢查該類別是否已載入，若否則先載入；接著在堆積為物件配置記憶體、把實體欄位設為預設值、執行建構子完成初始化。一個物件在堆積中的佈局分為三部分：**物件頭**（object header，含 mark word 與指向類別的指標）、**實體資料**（各欄位的值）、**對齊填充**（padding，補齊到 8 位元組的倍數以利存取）。

6.6　垃圾回收基礎

6.6.1　如何判斷垃圾

GC 的第一個問題是：哪些物件是垃圾？一種天真的想法是「引用計數」——記錄每個物件被幾個地方參考，歸零就回收。但它有致命缺陷：**循環引用**。若 A 參考 B、B 也參考 A，即使外界都不再用它們，兩者的計數都不為零，永遠無法回收。

因此主流 JVM 採用**可達性分析**：從一組稱為「GC Roots」的根物件出發（如堆疊中的區域變數、靜態變數、常數），沿參考鏈遍歷，凡是「走得到」的物件就存活，「走不到」的就是垃圾。這樣循環引用但整體不可達的物件群，會被正確地一併回收。

6.6.2 四種引用強度

Java 提供四種強度的引用，讓我們對「物件何時該被回收」有更細緻的控制：

引用強度	回收時機
強引用（Strong）	一般的參考，只要可達就絕不回收
軟引用（Soft）	記憶體不足時才回收，適合做快取
弱引用（Weak）	只要 GC 發生就回收
虛引用（Phantom）	隨時可能被回收，用於追蹤回收事件

範例：ReferenceDemo.java ——弱引用與快取

```
import java.lang.ref.WeakReference;

/**
 * 弱引用：GC 一旦執行，被弱引用指向且無強引用的物件即被回收。
 */
public class ReferenceDemo {
    public static void main(String[] args) {
        // 建立物件，只用弱引用指向它（無強引用）
        WeakReference<byte[]> weak =
                new WeakReference<>(new byte[10 * 1024 * 1024]);

        System.out.println("GC 前，物件存在？" + (weak.get() != null));
        System.gc();                          // 建議 JVM 執行 GC
        try {
            Thread.sleep(100);                // 給 GC 一點時間
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("GC 後，物件存在？" + (weak.get() != null));
    }
}
```

▶ 執行結果

```
GC 前，物件存在？true
GC 後，物件存在？false
```

6.6.3 分代收集

主流 GC 基於一個經驗觀察——「大部分物件朝生暮死」（弱分代假說），把堆積分成 **新生代**與 **老年代**。新生代又分為一個 Eden 區與兩個 Survivor 區。新物件先進 Eden；一次 Minor GC 後，存活者被複製到一個 Survivor 區，如此在兩個 Survivor 間來回複製，每熬過一次 GC 年齡加一；年齡達到門檻（預設 15）就晉升到老年代。老年代滿了才觸發成本較高的 Major/Full GC。

6.6.4 GC 演算法與收集器

不同區域適用不同的基礎演算法，各有取捨：

演算法	做法	適用
標記清除	標記垃圾後清除	會產生碎片
複製	存活者複製到另一半	新生代（存活少）
標記整理	標記後把存活者往一端壓縮	老年代（無碎片）

垃圾收集器則是這些演算法的具體實作，多年來不斷演進：

收集器	特性
Serial	單執行緒，簡單，適合小型應用
Parallel	多執行緒，追求高吞吐量
CMS	低停頓，但已於 Java 14 移除
G1	區域化設計，可預測停頓，Java 9 起的預設
ZGC	超低延遲，停頓不隨堆大小成長

6.6.5 常用 JVM 參數與 GC 日誌

調校 JVM 時，這些參數最常用：

參數	作用
`-Xms` / `-Xmx`	堆積初始 / 最大大小
`-Xss`	每個執行緒的堆疊大小
`-Xlog:gc`	輸出 GC 日誌（Java 9 起的統一格式）
`-XX:+HeapDumpOnOutOfMemoryError`	OOM 時自動產生堆積傾印

> **提示**　正式環境通常把 `-Xms` 與 `-Xmx` 設為**相同值**。因為堆積動態擴張與收縮本身要付出成本，且可能引發額外的 Full GC；設成相同大小可避免執行期反覆調整堆積，讓效能更穩定、可預測。

以下是一段 `-Xlog:gc` 輸出的逐行解讀：

▶ 執行結果

```
[0.312s][info][gc] GC(0) Pause Young (G1 Evacuation Pause) 24M->6M(64M) 3.821ms
```

這行的意思是：程式啟動後 0.312 秒發生第 0 次 GC，類型是 G1 的年輕代回收；回收前堆積用了 24 MB，回收後降到 6 MB（總堆 64 MB）；本次停頓 3.821 毫秒。從「24M 降到 6M」可知大量短命物件被回收，這是健康的年輕代 GC 表現。

6.7 診斷工具實戰

JDK 內建一組命令列工具，是排查線上問題的第一線武器：`jps` 列出 Java 程序、`jinfo` 看設定、`jstat` 看 GC 統計、`jmap` 產生堆積傾印、`jstack` 印出執行緒堆疊。其中 `jstack` 最能立竿見影——它能自動偵測死結。用第 3 章的死結程式示範：

▶ 執行結果

```
$ jps
18244 DeadlockDemo
18255 Jps

$ jstack 18244
...
Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00007f... (object 0x000000071..., a Object),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00007f... (object 0x000000072..., a Object),
  which is held by "Thread-0"

Found 1 deadlock.
```

`jstack` 直接點名「Found one Java-level deadlock」並列出互相等待的兩個執行緒與鎖，讓死結無所遁形。圖形化的 VisualVM 與內建的 JFR（Java Flight Recorder）則能持續記錄 CPU、記憶體、GC 與鎖競爭，適合做長時間的效能剖析。

6.8 記憶體洩漏經典案例

Java 有 GC，為什麼還會記憶體洩漏？因為 GC 只回收「不可達」的物件，若你不小心讓一個**該丟棄的物件仍被某處參考**，GC 就無能為力。最經典的案例是用 `static` 集合當快取卻只進不出：

範例：MemoryLeak.java ——static 集合洩漏

```java
import java.util.ArrayList;
import java.util.List;

/**
 * 反面教材：static 集合持續加入卻從不移除，物件永遠可達，
 * 形同記憶體洩漏。修正方式見註解。
 */
public class MemoryLeak {
    // static 欄位的生命週期與類別相同，幾乎等於「永遠不死」
    static final List<byte[]> CACHE = new ArrayList<>();

    static void handleRequest() {
        // 每次請求都塞入資料，卻從未清理 → 洩漏
        CACHE.add(new byte[1024 * 1024]);
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            handleRequest();
            System.out.println("處理第 " + i + " 個請求，快取筆數："
                    + CACHE.size());
        }
        // 正解：改用有淘汰機制的快取（如 LinkedHashMap 的 LRU、
        // 或 WeakHashMap），並在適當時機移除不再需要的項目。
    }
}
```

▶ 執行結果

```
處理第 1 個請求，快取筆數：1
處理第 2 個請求，快取筆數：2
處理第 3 個請求，快取筆數：3
處理第 4 個請求，快取筆數：4
處理第 5 個請求，快取筆數：5
```

在真實系統中，這個 CACHE 會隨著請求量無限膨脹，最終導致 OOM。修正之道是採用有上限與淘汰策略的快取（如 LRU），或用 WeakHashMap 讓不再被別處強引用的項目能被 GC 自動清除。診斷這類問題時，就靠前一節的 jmap 產生堆積傾印，再用 VisualVM 分析「誰佔用了最多記憶體、又是誰在參考它」。

6.9 本章小結

- JVM 由類別載入子系統、執行期資料區、執行引擎、本地方法介面組成。
- 類別載入歷經載入、連結（驗證/準備/解析）、初始化；初始化是惰性的，存取編譯期常數不會觸發。
- 雙親委派模型讓核心類別只由 Bootstrap 載入，確保安全與一致。
- 堆疊為執行緒私有、堆積為全體共享；無窮遞迴致 `StackOverflowError`，物件累積不放致 `OutOfMemoryError`。
- Java 8 起方法區改用 Metaspace，字串常量池 Java 7 起移入堆積。
- GC 用可達性分析（GC Roots）判斷垃圾，解決引用計數的循環引用缺陷。
- 四種引用強度提供回收控制；分代收集依「朝生暮死」把堆分新生代與老年代。
- G1 是現代預設收集器；`-Xms` 與 `-Xmx` 建議設為相同值。
- `jstack` 可自動偵測死結；`static` 集合只增不減是經典的記憶體洩漏。

6.10 本章練習

動手練習

1. 觀念題：為什麼引用計數無法處理循環引用？可達性分析如何解決這個問題？（提示：想想從 GC Roots 是否「走得到」。）
2. 觀念題：雙親委派模型如何防止核心類別被惡意替換？若沒有它會有什麼風險？（提示：假設有人自寫一個 `java.lang.String`。）
3. 實作題：寫一個會觸發 `StackOverflowError` 的程式與一個會觸發 `OutOfMemoryError` 的程式，分別用 `-Xss256k` 與 `-Xmx32m` 執行並觀察。（提示：一個靠無窮遞迴，一個靠不斷 `new` 並保留參考。）
4. 實作題：用 `SoftReference` 實作一個簡易圖片快取，在記憶體吃緊時自動釋放，並驗證其行為。（提示：快取的值用 `SoftReference` 包住。）
5. 實作題：故意寫一個會死結的程式，用 `jps` 找到程序號，再用 `jstack` 印出堆疊，找出死結報告。（提示：可直接用第 3 章的死結範例。）
6. 觀念題：解讀這行 GC 日誌代表什麼——`Pause Young 40M->8M(128M) 5.2ms`，並說明它是否健康。（提示：關注回收前後的用量差與停頓時間。）

第 7 章

網路程式設計與 JDBC

本章學習目標

學完本章你將能夠：

- 說明 IP、埠號與 TCP/UDP 的差異，用 `InetAddress` 查詢主機。
- 用 Socket 撰寫 TCP 的伺服器與用戶端，並演進到執行緒池版本。
- 用 Java 11 的 `HttpClient` 發出同步與非同步請求。
- 用 JDBC 完成 CRUD、辨識並防範 SQL 注入。
- 用交易確保資料一致，並用批次處理提升效能。
- 以 DAO 模式分層，理解連線池的價值。

現代軟體幾乎沒有孤島：前端要打 API、後端要連資料庫、服務之間要互相呼叫。本章聚焦兩個最基礎也最關鍵的能力——**網路通訊**與**資料庫存取**。這兩者都建立在前幾章的基礎上：伺服器要用第 4 章的執行緒池服務多個連線，非同步 HTTP 呼叫要用 `CompletableFuture`，連線池的概念也與執行緒池一脈相承。

我們先從網路的底層 Socket 講起，理解「連線」到底是什麼，再往上到便利的 `HttpClient`；資料庫部分則從 JDBC 的原始 API 出發，一路帶到防注入、交易與分層設計等業界實務。

7.1 網路基礎速覽

網路通訊靠兩個座標定位對象：**IP 位址**標定「哪一台主機」，**埠號**（port）標定「主機上的哪個程式」。兩者合起來（如 `140.112.1.1:8080`）就能唯一指向網路上某個服務。傳輸層有兩大協定：

比較	TCP	UDP
連線	需三向交握建立連線	無連線
可靠性	保證送達與順序	不保證
速度	較慢（有確認機制）	較快
適用	網頁、檔案傳輸	直播、遊戲、DNS

TCP 建立連線要經過「三向交握」：用戶端送 SYN、伺服器回 SYN-ACK、用戶端再回 ACK，雙方確認彼此都能收送後才開始傳資料。這個機制是 TCP「可靠」的基礎。

範例：InetAddressDemo.java ——查詢主機資訊

```java
import java.net.InetAddress;

/**
 * 用 InetAddress 查詢本機與指定主機的 IP 位址。
 */
public class InetAddressDemo {
    public static void main(String[] args) throws Exception {
        // 取得本機資訊
        InetAddress local = InetAddress.getLocalHost();
        System.out.println("本機名稱：" + local.getHostName());
        System.out.println("本機位址：" + local.getHostAddress());

        // 以網域名稱查詢 IP（DNS 解析）
        InetAddress site = InetAddress.getByName("localhost");
        System.out.println("localhost → " + site.getHostAddress());
    }
}
```

▶ 執行結果

```
本機名稱：mycomputer.local
本機位址：192.168.1.20
localhost → 127.0.0.1
```

7.2　TCP Socket 程式設計

TCP 通訊的兩端分別用 `ServerSocket`（伺服器）與 `Socket`（用戶端）。伺服器呼叫 `accept` 會阻塞等待，直到有用戶端連上才回傳一個 `Socket` 代表這條連線，之後雙方就透過它的輸入輸出串流讀寫資料。先看一個 Echo（回聲）伺服器，它把收到的每一行原樣送回：

範例：EchoServer.java ——回聲伺服器

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 單執行緒回聲伺服器：接受一個連線，把每行輸入原樣回傳。
```

```
 * try-with-resources 確保 Socket 與串流自動關閉。
 */
public class EchoServer {
    public static void main(String[] args) throws Exception {
        // 綁定 5000 埠，開始監聽
        try (ServerSocket server = new ServerSocket(5000)) {
            System.out.println("伺服器啟動，等待連線...");
            try (Socket socket = server.accept();        // 阻塞等待
                 BufferedReader in = new BufferedReader(
                         new InputStreamReader(socket.getInputStream()));
                 PrintWriter out = new PrintWriter(
                         socket.getOutputStream(), true)) {
                System.out.println("用戶端已連線");
                String line;
                // 逐行讀取，直到用戶端關閉連線（readLine 回傳 null）
                while ((line = in.readLine()) != null) {
                    System.out.println("收到：" + line);
                    out.println("回聲：" + line);        // 原樣送回
                }
            }
        }
    }
}
```

搭配的用戶端連上伺服器、送出訊息並印出回聲：

範例：EchoClient.java ——回聲用戶端

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * 回聲用戶端：連上伺服器，送出兩行訊息並印出回聲。
 */
public class EchoClient {
    public static void main(String[] args) throws Exception {
        // 連到 localhost 的 5000 埠
        try (Socket socket = new Socket("localhost", 5000);
             PrintWriter out = new PrintWriter(
                     socket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(
```

```
                    new InputStreamReader(socket.getInputStream()))) {

            for (String msg : new String[]{"你好", "Java 網路"}) {
                out.println(msg);                          // 送出
                System.out.println(in.readLine());         // 讀回聲
            }
        }
    }
}
```

執行時開兩個終端機：先啟動伺服器，再啟動用戶端。兩邊的輸出如下：

▶ 執行結果

```
# 終端機 1（伺服器）
$ java EchoServer
伺服器啟動，等待連線...
用戶端已連線
收到：你好
收到：Java 網路

# 終端機 2（用戶端）
$ java EchoClient
回聲：你好
回聲：Java 網路
```

> **說明**　macOS 的「AirPlay 接收器」預設會佔用 TCP 5000 埠。若啟動伺服器時出現 `BindException: Address already in use`，請到「系統設定 → 一般 → 隔空播放與接力」關閉 AirPlay 接收器，或把範例中的埠號改成其他值（如 5050，伺服器與用戶端要一起改）。

7.3　從單執行緒到執行緒池伺服器

上面的伺服器有個致命問題：它一次只能服務一個用戶端。因為主執行緒被 `accept` 之後的讀寫迴圈佔住，第二個用戶端只能乾等。解法呼應第 4 章：每接受一條連線，就把它交給執行緒池處理，主執行緒立刻回頭 `accept` 下一個。

範例：PooledEchoServer.java ——執行緒池伺服器

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```java
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 執行緒池版回聲伺服器：可同時服務多個用戶端。
 * 主執行緒只負責 accept，實際讀寫交給執行緒池。
 */
public class PooledEchoServer {
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(10);
        try (ServerSocket server = new ServerSocket(5000)) {
            System.out.println("執行緒池伺服器啟動");
            while (true) {
                Socket socket = server.accept();     // 接受新連線
                // 把這條連線的服務工作丟給執行緒池
                pool.submit(() -> handle(socket));
            }
        }
    }

    /** 服務單一連線：回聲直到對方關閉 */
    static void handle(Socket socket) {
        try (socket;
             BufferedReader in = new BufferedReader(
                     new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(
                     socket.getOutputStream(), true)) {
            String line;
            while ((line = in.readLine()) != null) {
                out.println("回聲：" + line);
            }
        } catch (Exception e) {
            System.out.println("連線異常：" + e.getMessage());
        }
    }
}
```

▶ 執行結果

```
執行緒池伺服器啟動
```

（此時可同時開多個用戶端，各自都能得到回聲，互不阻塞）

> **提示**　「每連線一執行緒」的模型直觀，但連線數極大時（如上萬）會受限於平台執行緒的成本。此時有兩條現代路線：用第 4 章的虛擬執行緒（每連線一條虛擬執行緒，輕量且寫法不變），或改用 NIO 的非阻塞、事件驅動模型（如 Netty）。中小型服務用執行緒池已綽綽有餘。

7.4　群聊伺服器實戰

把多執行緒與共享狀態結合，就能做出群聊伺服器：維護一份「所有連線」的清單，任何人發言就廣播給全部人。關鍵在於這份清單會被多執行緒同時存取，必須用執行緒安全的集合（呼應第 4 章）。

範例：ChatServer.java ——廣播群聊

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 群聊伺服器：維護所有用戶端的輸出串流，收到訊息即廣播給全部人。
 * CopyOnWriteArrayList 保證多執行緒讀寫清單的安全（讀多寫少）。
 */
public class ChatServer {
    // 所有已連線用戶端的輸出串流
    static final List<PrintWriter> clients = new CopyOnWriteArrayList<>();

    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newCachedThreadPool();
        try (ServerSocket server = new ServerSocket(6000)) {
            System.out.println("群聊伺服器啟動於 6000 埠");
            while (true) {
                Socket socket = server.accept();
                pool.submit(() -> serve(socket));
```

```
            }
        }
    }

    /** 服務單一用戶端：註冊、收訊廣播、離線時移除 */
    static void serve(Socket socket) {
        try (socket;
             BufferedReader in = new BufferedReader(
                     new InputStreamReader(socket.getInputStream()))) {
            PrintWriter out = new PrintWriter(
                    socket.getOutputStream(), true);
            clients.add(out);                       // 註冊
            String line;
            while ((line = in.readLine()) != null) {
                broadcast(line);                    // 廣播
            }
        } catch (Exception e) {
            System.out.println("用戶端離線");
        }
    }

    /** 把訊息送給每一個已連線的用戶端 */
    static void broadcast(String msg) {
        for (PrintWriter client : clients) {
            client.println(msg);
        }
    }
}
```

▶ 執行結果

```
群聊伺服器啟動於 6000 埠
（多個用戶端連入後，任一人發言，所有人都會即時收到）
```

7.5　UDP 收發

UDP 沒有連線的概念，直接用 `DatagramSocket` 收送 `DatagramPacket` 封包。它不保證送達，但省去建立連線的開銷，適合可容忍少量遺失、要求低延遲的場景（如即時串流、遊戲同步）。

範例：UdpDemo.java ——UDP 收發概念

```
import java.net.DatagramPacket;
```

```java
import java.net.DatagramSocket;
import java.net.InetAddress;

/**
 * UDP 收發示範：接收端與傳送端各用一個 DatagramSocket。
 * 此處在同一程式內先啟動接收再送出，示意收送流程。
 */
public class UdpDemo {
    public static void main(String[] args) throws Exception {
        // 接收端：綁定 7000 埠，在另一執行緒等待封包
        Thread receiver = new Thread(() -> {
            try (DatagramSocket socket = new DatagramSocket(7000)) {
                byte[] buf = new byte[1024];
                DatagramPacket packet =
                        new DatagramPacket(buf, buf.length);
                socket.receive(packet);             // 阻塞等待
                String text = new String(packet.getData(),
                        0, packet.getLength());
                System.out.println("接收到：" + text);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        receiver.start();
        Thread.sleep(100);                          // 確保接收端就緒

        // 傳送端：把封包送往 localhost 的 7000 埠
        try (DatagramSocket socket = new DatagramSocket()) {
            byte[] data = "UDP 訊息".getBytes();
            DatagramPacket packet = new DatagramPacket(data,
                    data.length, InetAddress.getByName("localhost"), 7000);
            socket.send(packet);
        }
        receiver.join();
    }
}
```

▶ 執行結果

```
接收到：UDP 訊息
```

7.6 HTTP 用戶端

早期用 `HttpURLConnection` 發 HTTP 請求非常繁瑣：要手動設定方法、開串流、讀回應、關資源。Java 11 引入的 `HttpClient` 徹底改善，API 現代且支援非同步。以下示範同步 GET、非同步 GET 與 POST JSON：

範例：HttpClientDemo.java ——同步與非同步請求

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.concurrent.CompletableFuture;

/**
 * Java 11 HttpClient：同步 GET、非同步 GET、POST JSON。
 * HttpClient 可重複使用，建議整個應用共用一個實體。
 */
public class HttpClientDemo {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();

        // 同步 GET：send 會阻塞直到回應返回
        HttpRequest get = HttpRequest.newBuilder()
                .uri(URI.create("https://example.com"))
                .GET()
                .build();
        HttpResponse<String> resp = client.send(get,
                HttpResponse.BodyHandlers.ofString());
        System.out.println("同步狀態碼：" + resp.statusCode());

        // 非同步 GET：回傳 CompletableFuture（呼應第 4 章）
        CompletableFuture<Integer> future = client
                .sendAsync(get, HttpResponse.BodyHandlers.ofString())
                .thenApply(HttpResponse::statusCode);
        System.out.println("非同步狀態碼：" + future.get());

        // POST JSON：設定標頭與請求主體
        HttpRequest post = HttpRequest.newBuilder()
                .uri(URI.create("https://example.com/api"))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(
```

```
                        "{\"name\":\"Java\"}"))
                .build();
        System.out.println("已建立 POST 請求：" + post.method());
    }
}
```

▶ 執行結果

```
同步狀態碼：200
非同步狀態碼：200
已建立 POST 請求：POST
```

7.7 JDBC：Java 資料庫連接

JDBC 是 Java 存取關聯式資料庫的標準 API。它把「如何連接與操作資料庫」抽象成一組介面，各家資料庫廠商提供各自的驅動實作。核心角色有四個：`DriverManager`（依連線 URL 取得連線）、`Connection`（一條資料庫連線）、`Statement/PreparedStatement`（執行 SQL）、`ResultSet`（查詢結果集）。

> **說明**　本章範例以 SQLite 為例（輕量、免安裝伺服器，資料就是一個檔案），需在 classpath 加入 `org.xerial:sqlite-jdbc` 驅動 JAR。連線 URL 格式為 `jdbc:sqlite:test.db`；MySQL 則類似 `jdbc:mysql://localhost:3306/db`。自 JDBC 4.0 起，只要驅動 JAR 在 classpath，會透過 SPI 自動註冊，不必再手動呼叫 `Class.forName`。以下範例的執行結果為合理的示意輸出。

7.7.1 建表與 CRUD

範例：JdbcCrud.java ——建表與新增查詢

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

/**
 * JDBC 基本流程：連線、建表、以 Statement 執行 DDL/DML、查詢走訪。
 * try-with-resources 確保連線與 Statement 自動關閉。
 */
public class JdbcCrud {
    public static void main(String[] args) throws Exception {
```

```
        String url = "jdbc:sqlite:school.db";
        try (Connection conn = DriverManager.getConnection(url);
             Statement stmt = conn.createStatement()) {

            // 建表（若不存在）
            stmt.execute("CREATE TABLE IF NOT EXISTS student ("
                    + "id INTEGER PRIMARY KEY, name TEXT, score INTEGER)");
            // 清空舊資料以利重複執行
            stmt.execute("DELETE FROM student");
            // 新增資料
            stmt.executeUpdate(
                    "INSERT INTO student VALUES (1, 'Alice', 90)");
            stmt.executeUpdate(
                    "INSERT INTO student VALUES (2, 'Bob', 75)");

            // 查詢並走訪結果集
            ResultSet rs = stmt.executeQuery(
                    "SELECT id, name, score FROM student ORDER BY id");
            while (rs.next()) {                    // 逐列前進
                System.out.printf("%d | %s | %d%n",
                        rs.getInt("id"),
                        rs.getString("name"),
                        rs.getInt("score"));
            }
        }
    }
}
```

▶ 執行結果

```
1 | Alice | 90
2 | Bob | 75
```

7.7.2 SQL 注入與 PreparedStatement

用字串拼接組 SQL 是重大的資安漏洞。假設用「使用者輸入」拼查詢，惡意輸入就能竄改語意：

範例：SqlInjection.java ——注入攻擊示範

```
/**
 * 反面教材：用字串拼接組 SQL。
```

```
 * 當輸入是 ' OR '1'='1，WHERE 條件恆真，等於繞過驗證取得全部資料。
 */
public class SqlInjection {
    public static void main(String[] args) {
        String userInput = "' OR '1'='1";          // 惡意輸入
        // 危險的字串拼接
        String sql = "SELECT * FROM users WHERE name = '"
                + userInput + "'";
        System.out.println("實際送出的 SQL：");
        System.out.println(sql);
        // 條件變成 name = '' OR '1'='1'，永遠成立 → 全表外洩
    }
}
```

▶ 執行結果

```
實際送出的 SQL：
SELECT * FROM users WHERE name = '' OR '1'='1'
```

> **注意**　永遠不要用字串拼接把使用者輸入組進 SQL。如上所示，輸入 `' OR '1'='1` 就能讓 WHERE 條件恆真，繞過驗證竊取整張表；更惡意的輸入甚至能刪表。正解是用 PreparedStatement 的**參數化查詢**：先把 SQL 骨架送給資料庫預編譯，再用 setXxx 綁定參數值。此時參數只會被當成「資料」而非「SQL 語法」，注入無從發生。

範例：PreparedStatementDemo.java ——參數化查詢

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * 用 PreparedStatement 參數化查詢：安全且可預編譯重用，效能更好。
 * 問號 ? 是參數佔位符，用 setXxx 依序綁定實際值。
 */
public class PreparedStatementDemo {
    public static void main(String[] args) throws Exception {
        String url = "jdbc:sqlite:school.db";
        try (Connection conn = DriverManager.getConnection(url)) {
            String sql = "SELECT name, score FROM student "
                    + "WHERE score >= ?";
```

```
            try (PreparedStatement ps = conn.prepareStatement(sql)) {
                ps.setInt(1, 80);               // 綁定第 1 個參數
                ResultSet rs = ps.executeQuery();
                while (rs.next()) {
                    System.out.println(rs.getString("name")
                            + "：" + rs.getInt("score"));
                }
            }
        }
    }
}
```

▶ 執行結果

```
Alice：90
```

7.7.3 交易管理

一組操作要嘛「全部成功」、要嘛「全部不做」，這就是**交易**。轉帳是經典例子：從 A 扣款與往 B 加款必須同進退，中途失敗就要回滾。JDBC 預設每條 SQL 自動提交，要手動控制交易需先關閉自動提交。

範例：TransferDemo.java ——交易與回滾

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

/**
 * 轉帳交易：扣款與加款包在同一交易中，任一步失敗即整筆回滾。
 */
public class TransferDemo {
    public static void main(String[] args) throws Exception {
        String url = "jdbc:sqlite:bank.db";
        try (Connection conn = DriverManager.getConnection(url)) {
            conn.setAutoCommit(false);          // 開始手動交易
            try {
                update(conn, "UPDATE account SET balance = balance - 100 "
                        + "WHERE id = 1");      // A 扣 100
                // 模擬中途失敗：若餘額不足應中止
                if (isOverdrawn(conn)) {
                    throw new IllegalStateException("餘額不足");
```

```
                }
                update(conn, "UPDATE account SET balance = balance + 100 "
                        + "WHERE id = 2");      // B 加 100
                conn.commit();                  // 全部成功才提交
                System.out.println("轉帳成功");
            } catch (Exception e) {
                conn.rollback();                // 失敗則回滾
                System.out.println("轉帳失敗已回滾：" + e.getMessage());
            }
        }
    }

    static void update(Connection conn, String sql) throws Exception {
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.executeUpdate();
        }
    }

    /** 示意：檢查是否透支（此處固定回傳 false 讓交易成功） */
    static boolean isOverdrawn(Connection conn) {
        return false;
    }
}
```

▶ 執行結果

```
轉帳成功
（若中途拋例外，會印出「轉帳失敗已回滾」，且兩帳戶餘額不變）
```

7.7.4 批次處理與 DAO 模式

大量寫入時，逐條執行會來回資料庫很多次。用 `addBatch` 累積、`executeBatch` 一次送出，能大幅減少往返、提升效能。而在架構上，我們不會把 SQL 灑滿業務程式，而是用 **DAO 模式**（Data Access Object）把資料存取邏輯集中到專門的類別，對上層只暴露乾淨的介面——這呼應中級篇的 Repository 概念。

範例：StudentDao.java ——DAO 介面與批次

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.util.List;

```

```java
/**
 * DAO 模式：把資料存取封裝在專門類別，對上層暴露乾淨方法。
 * saveAll 用批次一次寫入多筆，減少資料庫往返。
 */
public class StudentDao {

    record Student(int id, String name, int score) { }

    /** DAO 介面：上層只依賴這個抽象 */
    interface Dao {
        void saveAll(List<Student> students) throws Exception;
    }

    /** JDBC 實作 */
    static class JdbcStudentDao implements Dao {
        private final Connection conn;
        JdbcStudentDao(Connection conn) { this.conn = conn; }

        @Override
        public void saveAll(List<Student> students) throws Exception {
            String sql = "INSERT INTO student VALUES (?, ?, ?)";
            try (PreparedStatement ps = conn.prepareStatement(sql)) {
                for (Student s : students) {
                    ps.setInt(1, s.id());
                    ps.setString(2, s.name());
                    ps.setInt(3, s.score());
                    ps.addBatch();              // 累積一筆
                }
                int[] result = ps.executeBatch();   // 一次送出全部
                System.out.println("批次寫入 " + result.length + " 筆");
            }
        }
    }
}
```

▶ 執行結果

```
批次寫入 3 筆
```

提示　每次操作都 `DriverManager.getConnection` 建立新連線非常昂貴（TCP 交握、認證、資源配置）。正式系統一律用**連線池**：預先建立一批連線重複使用，用完歸還而非關閉。業界最主流的是 `HikariCP`（Spring Boot 的預設），以輕量與高效著稱。連線池與第 4 章的執行緒池是同一種「資源池化」思想的體現。

7.8 本章小結

- IP 標定主機、埠號標定程式；TCP 可靠有序需交握，UDP 無連線但低延遲。
- `ServerSocket.accept` 阻塞等待連線；用 Socket 的輸入輸出串流讀寫資料。
- 單執行緒伺服器一次只能服務一人；用執行緒池（或虛擬執行緒）可同時服務多個連線。
- 群聊伺服器需以執行緒安全的集合維護連線清單並廣播。
- Java 11 `HttpClient` 提供現代的同步與非同步 HTTP API，非同步版回傳 `CompletableFuture`。
- JDBC 四大角色：`DriverManager`、`Connection`、`PreparedStatement`、`ResultSet`。
- 字串拼接 SQL 會招致注入攻擊；一律用 `PreparedStatement` 參數化查詢，兼顧安全與效能。
- 交易用 `setAutoCommit(false)` + `commit/rollback` 確保一致；批次用 `addBatch` ↪ `/executeBatch` 提效。
- DAO 模式集中資料存取邏輯；連線池（如 HikariCP）避免重複建立連線的開銷。

7.9 本章練習

動手練習

1. 觀念題：為什麼單執行緒的 Socket 伺服器一次只能服務一個用戶端？執行緒池如何解決？（提示：想想 `accept` 之後主執行緒在做什麼。）
2. 觀念題：`PreparedStatement` 為何能防止 SQL 注入？它相較字串拼接還有什麼效能優勢？（提示：關鍵在「預編譯」與「參數只當資料」。）
3. 實作題：擴充回聲伺服器，當收到 `bye` 時主動關閉該連線並印出「用戶端離線」。（提示：在讀迴圈內判斷內容後 `break`。）
4. 實作題：用 `HttpClient` 對某公開 API 發出非同步 GET，取得回應後印出前 100 個字元。（提示：`sendAsync` 後接 `thenApply`。）
5. 實作題：用 JDBC 與 SQLite 建立 `product` 表，實作完整 CRUD，全部查詢改用 `PreparedStatement`。（提示：更新與刪除用 `executeUpdate`。）

6. 實作題：寫一個轉帳交易，故意在扣款後拋出例外，驗證回滾後兩個帳戶的餘額都沒有變動。（提示：`catch` 內呼叫 `rollback`，事後查詢餘額確認。）

第 8 章

設計模式與現代 Java

本章學習目標

學完本章你將能夠：

- 用一句話說明 SOLID 五大原則並辨識違反範例。
- 實作單例、工廠、建造者、策略、觀察者、裝飾者六個常用模式。
- 用 Lambda 現代化策略模式，並辨認 JDK 中的模式應用。
- 熟練使用 `record`、`sealed` 類別與 text block。
- 用 switch 模式匹配搭配 `sealed` 達成編譯期窮盡性檢查。
- 綜合以上打造一個簡潔的運算式求值器。

歡迎來到進階篇的最後一章，也是全系列的終點。設計模式是前輩們把「反覆出現的設計問題」歸納出的通用解法，是工程師之間的共通詞彙。但學設計模式有個常見誤區: 把模式當成必須硬套的教條。本章希望傳達的是——**模式是詞彙，不是規則**；理解每個模式「解決什麼問題」，遠比背下它的類別圖重要。

更棒的是，現代 Java 讓許多模式變得更輕巧：Lambda 讓策略模式幾乎不必寫類別，`record` 讓不可變物件一行搞定，`sealed` 加上 switch 模式匹配則帶來函數式語言般的表達力。本章前半回顧經典模式，後半擁抱現代語法，最後用一個壓軸範例把兩者融合，為你的 Java 學習之旅畫下句點。

8.1 設計原則速覽：SOLID

好的設計模式都是 SOLID 原則的具體展現。先用一句話掌握每一條：

- **S 單一職責**：一個類別只該有一個改變的理由。
- **O 開放封閉**：對擴充開放、對修改封閉。
- **L 里氏替換**：子型別必須能無痛替換父型別。
- **I 介面隔離**：不該強迫實作用不到的方法。
- **D 依賴反轉**：依賴抽象而非具體實作。

其中**開放封閉原則**最為核心，值得展開。反例是：每新增一種形狀就要去改「計算面積」的那個大 `if-else`（修改既有程式，容易改壞）。正解是讓每種形狀實作共同介面，新增形狀

時只需新增一個類別，完全不動既有程式碼：

範例：OpenClosed.java ——開放封閉原則

```java
/**
 * 開放封閉：面積計算依賴 Shape 抽象。
 * 新增形狀只要新增類別，不必修改既有的計算邏輯。
 */
public class OpenClosed {

    interface Shape {
        double area();                  // 抽象：每種形狀自己算面積
    }

    record Circle(double r) implements Shape {
        public double area() { return Math.PI * r * r; }
    }
    record Square(double side) implements Shape {
        public double area() { return side * side; }
    }

    public static void main(String[] args) {
        Shape[] shapes = { new Circle(2), new Square(3) };
        for (Shape s : shapes) {
            // 這段程式碼永遠不必因新增形狀而修改
            System.out.printf("%s 面積：%.2f%n",
                    s.getClass().getSimpleName(), s.area());
        }
    }
}
```

▶ 執行結果

```
Circle 面積：12.57
Square 面積：9.00
```

提示　設計模式傳統上分三大類：**建立型**（如單例、工廠、建造者，關注「如何產生物件」）、**結構型**（如裝飾者，關注「如何組合物件」）、**行為型**（如策略、觀察者，關注「物件間如何協作」）。學習心法是：先認出「我遇到的是哪一類問題」，再挑對應的模式，切勿為用模式而用模式。

8.2 單例模式

單例（Singleton）確保一個類別全程只有一個實體。它看似簡單，卻是最能考驗並行功力的模式。我們逐步演進各種寫法：

範例：Singletons.java ——單例的多種寫法

```
/**
 * 彙整單例的常見寫法，各有取捨。
 */
public class Singletons {

    /** 餓漢式：類別載入即建立，天生執行緒安全，但無法延遲 */
    static class Eager {
        private static final Eager INSTANCE = new Eager();
        private Eager() { }
        public static Eager get() { return INSTANCE; }
    }

    /** 雙重檢查鎖：延遲建立又高效，volatile 不可省 */
    static class DoubleChecked {
        // volatile 防止指令重排導致「看到未初始化完成的物件」
        private static volatile DoubleChecked instance;
        private DoubleChecked() { }
        public static DoubleChecked get() {
            if (instance == null) {                 // 第一次檢查（免鎖）
                synchronized (DoubleChecked.class) {
                    if (instance == null) {         // 第二次檢查（持鎖）
                        instance = new DoubleChecked();
                    }
                }
            }
            return instance;
        }
    }

    /** 靜態內部類別：兼顧延遲與執行緒安全，寫法最優雅 */
    static class Holder {
        private Holder() { }
        private static class Lazy {
            static final Holder INSTANCE = new Holder();
        }
```

```
        public static Holder get() { return Lazy.INSTANCE; }
    }

    public static void main(String[] args) {
        System.out.println("餓漢式同一實體？"
                + (Eager.get() == Eager.get()));
        System.out.println("雙重檢查同一實體？"
                + (DoubleChecked.get() == DoubleChecked.get()));
        System.out.println("靜態內部類別同一實體？"
                + (Holder.get() == Holder.get()));
    }
}
```

▶ 執行結果

```
餓漢式同一實體？true
雙重檢查同一實體？true
靜態內部類別同一實體？true
```

寫法	延遲建立	備註
餓漢式	否	簡單、天生安全
懶漢式加 `synchronized`	是	每次取用都鎖，較慢
雙重檢查鎖	是	需 `volatile`
靜態內部類別	是	優雅、推薦
列舉（`enum`）	否	最安全，防反射與序列化破壞

> **說明**　雙重檢查鎖為什麼一定要 `volatile`？因為 `instance = new ...` 並非原子操作，它含「配置記憶體、呼叫建構子、把參考指向物件」三步，而 JVM 可能重排這些步驟。若沒有 `volatile`，某執行緒可能看到「參考已非 null 但物件尚未初始化完成」的半成品。`volatile` 禁止這種重排，呼應第 3 章的可見性。《Effective Java》則推薦用 `enum` 實作單例，因為它連反射與序列化攻擊都能天然抵禦。

8.3　工廠模式

工廠模式把「建立物件」的邏輯集中管理。最簡單的「簡單工廠」用一個方法依參數回傳不同實作；但它每加一種產品就得改工廠內的 `switch`，違反開放封閉。**工廠方法**模式更進一步：把「決定建立哪種產品」下放給子類別，新增產品時只要新增一個工廠子類別。

範例：FactoryMethod.java ——工廠方法

```java
/**
 * 工廠方法：抽象工廠定義「生產」的規格，子工廠決定生產什麼。
 * 新增運輸方式只要新增產品類別與工廠子類別，不動既有程式。
 */
public class FactoryMethod {

    interface Transport {
        String deliver();
    }
    static class Truck implements Transport {
        public String deliver() { return "以卡車走陸運"; }
    }
    static class Ship implements Transport {
        public String deliver() { return "以貨船走海運"; }
    }

    /** 抽象工廠：把建立動作延遲到子類別 */
    abstract static class Logistics {
        abstract Transport createTransport();       // 工廠方法
        String plan() {
            return "配送計畫：" + createTransport().deliver();
        }
    }
    static class RoadLogistics extends Logistics {
        Transport createTransport() { return new Truck(); }
    }
    static class SeaLogistics extends Logistics {
        Transport createTransport() { return new Ship(); }
    }

    public static void main(String[] args) {
        System.out.println(new RoadLogistics().plan());
        System.out.println(new SeaLogistics().plan());
    }
}
```

▶ 執行結果

```
配送計畫：以卡車走陸運
配送計畫：以貨船走海運
```

8.4 建造者模式

當一個物件有很多參數（尤其含許多選填項），建構子會膨脹成難以閱讀的「伸縮建構子」（telescoping constructor）：一長串同型別參數，呼叫時根本分不清誰是誰。建造者模式用流暢的鏈式 API 逐項設定，最後 `build` 出不可變物件。

範例：BuilderDemo.java ——流暢的建造者

```
/**
 * 建造者模式：以鏈式 API 清楚地組裝物件，產出不可變的 HttpRequest。
 * 對照標準庫的 HttpRequest.newBuilder()、StringBuilder。
 */
public class BuilderDemo {

    /** 不可變的目標物件，只能透過 Builder 建立 */
    static final class HttpRequest {
        private final String url;               // 必填
        private final String method;            // 選填，有預設
        private final int timeout;              // 選填，有預設

        private HttpRequest(Builder b) {
            this.url = b.url;
            this.method = b.method;
            this.timeout = b.timeout;
        }
        public String toString() {
            return method + " " + url + " (逾時 " + timeout + "s)";
        }

        static class Builder {
            private final String url;           // 必填放建構子
            private String method = "GET";      // 預設值
            private int timeout = 30;

            Builder(String url) { this.url = url; }
            Builder method(String m) { this.method = m; return this; }
            Builder timeout(int t) { this.timeout = t; return this; }
            HttpRequest build() { return new HttpRequest(this); }
        }
    }

    public static void main(String[] args) {
```

```
        // 只設定需要的項目，可讀性極佳
        HttpRequest req = new HttpRequest.Builder("https://api.example.com")
                .method("POST")
                .timeout(60)
                .build();
        System.out.println(req);
    }
}
```

▶ 執行結果

```
POST https://api.example.com (逾時 60s)
```

8.5 策略模式

策略模式把「可替換的演算法」封裝成獨立物件。傳統寫法會陷入 `if-else` 地獄：每加一種計費方式就多一個分支。策略模式把每種演算法變成一個策略物件，注入使用即可。而在現代 Java，由於策略介面通常是函數式介面，可以直接用 Lambda （呼應第 1 章），連類別都省了。

範例：StrategyDemo.java ——策略模式與 Lambda

```
import java.util.function.DoubleUnaryOperator;

/**
 * 策略模式：折扣演算法可替換。
 * 傳統要為每種策略寫一個類別；用 Lambda 則一行搞定。
 */
public class StrategyDemo {

    /** 結帳時注入折扣策略 */
    static double checkout(double price, DoubleUnaryOperator discount) {
        return discount.applyAsDouble(price);       // 套用策略
    }

    public static void main(String[] args) {
        double price = 1000;
        // 每個策略就是一個 Lambda，取代一整個策略類別
        System.out.println("原價：" + checkout(price, p -> p));
        System.out.println("九折：" + checkout(price, p -> p * 0.9));
        System.out.println("折百：" + checkout(price, p -> p - 100));
    }
}
```

▶ 執行結果

```
原價：1000.0
九折：900.0
折百：900.0
```

8.6 觀察者模式

觀察者模式建立「一對多」的通知關係：一個主題（Subject）狀態改變時，自動通知所有註冊的觀察者（Observer）。它是事件驅動系統、GUI 與訊息系統的骨架。

範例：ObserverDemo.java ——事件通知

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

/**
 * 觀察者模式：新聞社發布消息，所有訂閱者自動收到。
 * 觀察者用 Consumer 表達，註冊即訂閱。
 */
public class ObserverDemo {

    /** 主題：維護訂閱者清單並負責廣播 */
    static class NewsAgency {
        private final List<Consumer<String>> subscribers =
                new ArrayList<>();
        void subscribe(Consumer<String> s) { subscribers.add(s); }
        void publish(String news) {
            // 狀態改變時通知每一個觀察者
            subscribers.forEach(s -> s.accept(news));
        }
    }

    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();
        // 兩個觀察者訂閱
        agency.subscribe(n -> System.out.println("讀者A收到：" + n));
        agency.subscribe(n -> System.out.println("讀者B收到：" + n));

```

```
        agency.publish("Java 21 發布虛擬執行緒");     // 一發布，全員收到
    }
}
```

▶ 執行結果

```
讀者A收到：Java 21 發布虛擬執行緒
讀者B收到：Java 21 發布虛擬執行緒
```

8.7 裝飾者模式

裝飾者模式讓我們動態地為物件疊加功能，而不必製造大量子類別。經典範例是飲料加料：在基礎飲料外「包」一層層配料，每層都增添描述與價格。

範例：DecoratorDemo.java ——飲料加料

```
/**
 * 裝飾者模式：以「包裹」的方式層層疊加功能。
 * 每個裝飾者持有被裝飾者，並在其行為上加料。
 */
public class DecoratorDemo {

    interface Beverage {
        String desc();
        int cost();
    }
    /** 基礎飲料 */
    static class Coffee implements Beverage {
        public String desc() { return "咖啡"; }
        public int cost() { return 50; }
    }
    /** 裝飾者基底：持有一個被裝飾的飲料 */
    abstract static class AddOn implements Beverage {
        protected final Beverage base;
        AddOn(Beverage base) { this.base = base; }
    }
    static class Milk extends AddOn {
        Milk(Beverage b) { super(b); }
        public String desc() { return base.desc() + "+牛奶"; }
        public int cost() { return base.cost() + 10; }
    }
    static class Sugar extends AddOn {
```

```
        Sugar(Beverage b) { super(b); }
        public String desc() { return base.desc() + "+糖"; }
        public int cost() { return base.cost() + 5; }
    }

    public static void main(String[] args) {
        // 一層層包裝：咖啡 → 加牛奶 → 加糖
        Beverage drink = new Sugar(new Milk(new Coffee()));
        System.out.println(drink.desc() + " = " + drink.cost() + " 元");
    }
}
```

▶ 執行結果

```
咖啡+牛奶+糖 = 65 元
```

> **說明**　你其實早在中級篇就用過裝飾者了！寫法 `new BufferedReader(new InputStreamReader(in))` 正是層層裝飾：`InputStreamReader` 為位元組串流加上「解碼成字元」的功能，`BufferedReader` 再加上「緩衝與讀整行」的功能。Java I/O 整個體系就是裝飾者模式的大型實踐。認出這一點，你會對標準庫的設計豁然開朗。

`Iterator` 是迭代器模式、`Runnable` 是命令模式、`Comparator` 是策略模式、`Observable` 與事件監聽是觀察者模式——模式無所不在，下表整理 JDK 中的常見對照：

JDK 元素	對應模式
`Iterator`	迭代器（Iterator）
`Runnable`	命令（Command）
`Comparator`	策略（Strategy）
`InputStream` 家族	裝飾者（Decorator）
`Collections.unmodifiableList`	代理 / 裝飾者

8.8　現代 Java 特性

8.8.1　record 與 text block

`record` 是 Java 16 正式加入的不可變資料載體：一行宣告就自動產生建構子、`equals`、`hashCode`、`toString` 與各欄位的存取方法。它還支援緊湊建構子來做驗證。text block（三個雙引號）則讓多行字串不再需要滿是 `\n` 與跳脫。

範例：RecordDemo.java ——record 與緊湊建構子

```java
/**
 * record 自動產生樣板方法；緊湊建構子集中做驗證。
 */
public class RecordDemo {

    /** 一行定義不可變的座標，緊湊建構子確保非負 */
    record Point(int x, int y) {
        Point {                                  // 緊湊建構子（無參數列）
            if (x < 0 || y < 0) {
                throw new IllegalArgumentException("座標不可為負");
            }
        }
    }

    public static void main(String[] args) {
        Point p = new Point(3, 4);
        System.out.println("toString 自動產生：" + p);
        System.out.println("存取方法：" + p.x() + ", " + p.y());

        // 值相等即 equals 相等（自動實作）
        System.out.println("值相等？" + p.equals(new Point(3, 4)));

        // text block：多行字串免跳脫
        String json = """
                {
                    "x": 3,
                    "y": 4
                }""";
        System.out.println(json);
    }
}
```

▶ 執行結果

```
toString 自動產生：Point[x=3, y=4]
存取方法：3, 4
值相等？true
{
    "x": 3,
    "y": 4
}
```

> **注意**　`record` 不是萬用的。它適合表達「純資料」的不可變值物件（DTO、座標、金額）。若你需要可變狀態、需要繼承其他類別、或物件的身分（identity）比它的值更重要（如 JPA 實體），就不該用 `record`。此外，`record` 的欄位是淺層不可變——若欄位是可變集合，仍可能被外部修改，必要時要在緊湊建構子內做防禦性複製。

8.8.2 sealed 類別與 switch 模式匹配

`sealed`（密封）類別或介面用 `permits` 明確限定「誰可以繼承或實作我」，讓繼承體系封閉而可控。它與 switch 模式匹配是天作之合：因為子型別是有限且已知的，編譯器能檢查 **switch 是否窮盡**所有情況，少寫一個就編譯報錯，從此不必再寫防禦性的 `default`。

8.9 壓軸綜合範例：運算式求值器

讓我們用 `sealed` + `record` + switch 模式匹配，實作一個算術運算式求值器。運算式只有三種：數字、加法、乘法。傳統做法要嘛在父類別塞一個 `eval` 抽象方法（邏輯分散在各子類別），要嘛寫冗長的訪問者模式（Visitor）。現代 Java 讓我們把邏輯集中在一個一目瞭然的 switch 裡：

範例：ExprEval.java ——現代 Java 的表達力

```
/**
 * 用 sealed 介面 + record + switch 模式匹配求值算術運算式。
 * sealed 保證子型別有限，switch 可省略 default 且窮盡性由編譯器把關。
 */
public class ExprEval {

    /** 密封介面：運算式只可能是這三種 */
    sealed interface Expr permits Num, Add, Mul { }
    record Num(double value) implements Expr { }
    record Add(Expr left, Expr right) implements Expr { }
    record Mul(Expr left, Expr right) implements Expr { }

    /** 求值：對每種運算式解構後遞迴計算 */
    static double eval(Expr e) {
        return switch (e) {
            // record 解構：直接把欄位綁定成區域變數
            case Num(double v) -> v;
            case Add(Expr l, Expr r) -> eval(l) + eval(r);
            case Mul(Expr l, Expr r) -> eval(l) * eval(r);
            // 不需要 default：sealed 已保證窮盡所有情況
        };
```

```
    }

    public static void main(String[] args) {
        // 建構運算式：(2 + 3) * 4
        Expr expr = new Mul(new Add(new Num(2), new Num(3)), new Num(4));
        System.out.println("(2 + 3) * 4 = " + eval(expr));
    }
}
```

▶ 執行結果

```
(2 + 3) * 4 = 20.0
```

短短一個 switch，就取代了傳統訪問者模式的一整套介面與方法。更關鍵的是安全性: 若日後為 `Expr` 新增一種 `Sub`（減法），只要忘了在 switch 補上對應分支，**編譯器就會立刻報錯**。這種「編譯期窮盡性檢查」是過去 Java 難以企及的表達力，也是現代 Java 邁向函數式與資料導向設計的里程碑。

8.10　全系列結語：後續的學習地圖

恭喜你走到這裡！從初級篇的第一個 HelloWorld，到中級篇的物件導向與集合，再到進階篇的並行、JVM 與設計模式，你已經具備資深 Java 工程師的核心知識。但學習永無止境，以下是接下來值得投入的方向：

- **框架**：`Spring` 與 Spring Boot 是 Java 後端的事實標準，你在第 5 章親手實作的依賴注入正是它的地基。
- **測試**：`JUnit` 5 與 `Mockito` 是專業開發的必備技能，第 5 章的迷你測試框架已讓你看穿其原理。
- **建置工具**：`Maven` 與 `Gradle` 負責依賴管理與自動化建置，是團隊協作的基礎設施。
- **資料庫進階**：ORM（如 JPA/Hibernate）、SQL 調校、索引設計，延續第 7 章的 JDBC 基礎。
- **分散式系統**：微服務、訊息佇列（Kafka）、容器化（Docker/ Kubernetes），是大型系統的必經之路。

> **提示**　最後一個建議：**持續閱讀優秀的原始碼、持續動手寫專案**。設計模式、並行技巧、JVM 原理，都要在真實專案的千錘百鍊中才會內化成直覺。把這三冊當成隨時可查的地圖，但真正的成長發生在你敲下的每一行程式碼裡。祝你在 Java 的世界走得又遠又穩！

8.11 本章小結

- SOLID 是設計的指導原則，其中開放封閉原則（對擴充開放、對修改封閉）最為核心。
- 單例有餓漢、懶漢、雙重檢查鎖、靜態內部類別、enum 等寫法；雙重檢查鎖必須用 `volatile`，enum 最安全。
- 工廠方法把「建立哪種產品」下放子類別，實現開放封閉；建造者用鏈式 API 組裝不可變物件。
- 策略模式封裝可替換演算法，現代 Java 可直接用 Lambda 取代策略類別。
- 觀察者建立一對多通知；裝飾者動態疊加功能，Java I/O 即其大型實踐。
- `record` 是不可變資料載體，自動產生樣板方法並支援緊湊建構子驗證。
- `sealed` 限定繼承體系，搭配 switch 模式匹配可達成編譯期窮盡性檢查，免寫 `default`。
- `sealed` + `record` + switch 模式匹配能簡潔地取代訪問者模式，展現現代 Java 的表達力。

8.12 本章練習

動手練習

1. 觀念題：雙重檢查鎖為什麼一定要用 `volatile`？拿掉它可能發生什麼問題？（提示：`new` 的三個步驟與指令重排。）
2. 觀念題：什麼情況下不該用 `record`？請至少舉出兩種。（提示：可變狀態、需要繼承、身分重於值。）
3. 實作題：用 `enum` 實作單例，並說明它為何能抵禦反射與序列化的破壞。（提示：`enum` 的實體由 JVM 保證唯一。）
4. 實作題：用建造者模式為一個 `Pizza`（大小必填，配料多個選填）設計流暢 API，並產出不可變物件。（提示：配料可用可變參數或多次呼叫累加。）
5. 實作題：用策略模式（Lambda 版）實作一個排序器，能依「金額」或「時間」排序訂單，策略由呼叫端注入。（提示：策略型別用 `Comparator<Order>`。）
6. 實作題：擴充運算式求值器，新增 `Sub`（減法）與 `Div`（除法），觀察若忘記在 switch 補上分支，編譯器如何報錯。（提示：先在 `permits` 加入新型別，再看 switch 的窮盡性檢查。）