第二章

Java 的多執行緒

每一支 Android 應用程式都應該遵循 Java 語言內建的多執行緒編程模型。多執行緒改善應用程式的效能與良好操作體驗所需的回應性，但是，應用程式的複雜度也隨之增加：

- 以 Java 處理並發編程模型（concurrent programming model）
- 在多執行緒環境裡保持資料一致性
- 建立任務執行策略

執行緒基礎

軟體編程全然關乎指示硬體執行某些動作（例如，在螢幕上顯示圖像、將資料儲存在檔案系統…等等），這些指示由 CPU 依序處理的應用程式程式碼來定義，那是**執行緒**的高階定義。從應用程式的觀點來看，執行緒沿著循序處理之 Java 陳述式的程式碼路徑（code path）而執行。在執行緒上循序執行的一條程式碼路徑被稱作一項**任務**（*task*），那是在一個執行緒上執行的工作單位。執行緒能夠循序執行一或多個任務。

執行

Android 應用程式裡的執行緒由 `java.lang.Thread` 表示，它是在 Android 裡執行任務的最基本執行環境（當它啟動時），並且在完成任務或沒有其他任務要執行時終止；執行緒的存活時間由任務的長度來決定。執行緒支援任務的執行，而任務是 `java.lang.Runnable` 介面的實作，該實作在 `run` 方法裡定義任務：

```
private class MyTask implements Runnable {
    public void run() {
```

```
            int i = 0; // 儲存在執行緒本地堆疊。
        }
    }
```

run() 方法呼叫裡的所有本地變數 — 直接或間接的 — 都會被儲存在執行緒的本地記憶體堆疊中。任務的執行從實例化及啟動 Thread 開始：

```
Thread myThread = new Thread(new MyTask());
myThread.start();
```

在作業系統層級上，執行緒具有指令指標與堆疊指標。指令指標（instruction pointer）參照下一個要處理的指令，堆疊指標（stack pointer）參照私有記憶體區域 — 其他執行緒無法存取 — 在那裡，執行緒本地資料會被儲存。執行緒本地資料通常是在應用程式的 Java 方法裡定義的變數資料。

CPU 一次能夠處理一個執行緒的指令，但系統通常有多個執行緒需要同時被處理，例如，同時要執行多個應用程式的系統。為了讓使用者能夠並發執行多個應用程式，CPU 必須將它的處理時間（processing time）分攤給多個應用程式執行緒。CPU 處理時間的分攤是由**排程器**（scheduler）來處理的，它決定 CPU 應該處理什麼執行緒以及執行多久。這個排程策略能夠以各種方式來實作，但主要是奠基於執行緒**優先權**（priority）：高優先權執行緒在低優先權執行緒之前獲得 CPU 配置，這樣會讓高優先權執行緒得到較多的執行時間。Java 裡的執行緒優先權能夠被設定為 1（最低）到 10（最高），不過，除非明確指定，一般的優先權為 5：

```
myThread.setPriority(8);
```

不過，如果排程只是奠基於優先權，那麼，低優先權執行緒可能無法得到足夠的處理時間來完成它所負責的工作 — 稱作**執行緒飢餓**（starvation）。因此，在改變到新執行緒時，排程器也會考慮各個執行緒的處理時間。執行緒改變被稱作**上下文切換**（context switch）。上下文切換從儲存執行中之執行緒的狀態開始，好讓執行工作能夠在稍後被重啟，該執行緒隨後必須進入等待的狀態，接著，排程器必須回復並處理另一個等待中的執行緒。

兩個並發執行的執行緒 — 由單一處理器執行 — 被分割成多個執行間隔（execution interval），如圖 2-1 所示：

```
Thread T1 = new Thread(new MyTask());
T1.start();
Thread T2 = new Thread(new MyTask());
T2.start();
```

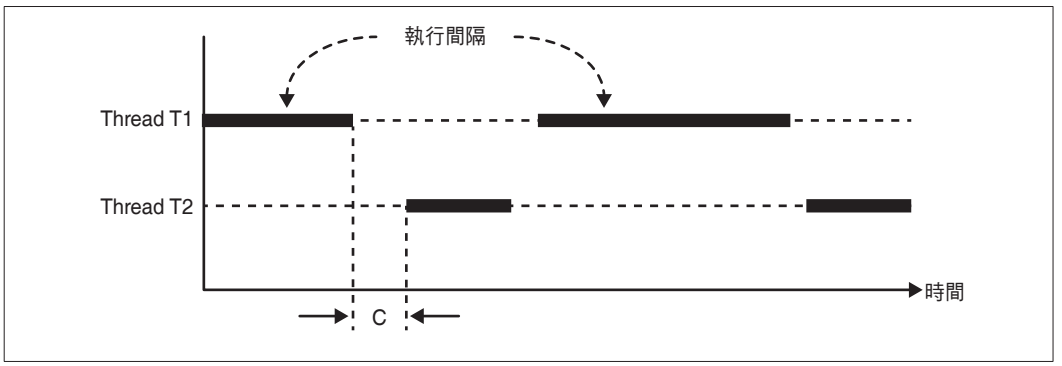

圖 2-1. 二個執行緒在一個 CPU 上執行，上下文切換被標示成 C。

每一個排程點均包含上下文切換，在那裡，作業系統必須利用 CPU 進行切換工作，如上圖中的 C 所示。

單執行緒應用程式

每個應用程式至少都有一個執行緒，定義執行的程式碼路徑。如果沒有其他執行緒被建立，所有的程式碼都將沿著相同的程式碼路徑被處理，而且，某個指令在可以被處理之前，必須等待所有在它之前的指令結束。

單執行緒執行是具有既定執行順序的簡單編程模型，但很多時候，這並不是足以勝任的方法，因為某個指令可能被先前的指令嚴重拖延到，即使這個指令根本不需要倚賴先前的指令。例如，按下裝置按鈕的使用者應該立刻得到該按鈕被壓下的視覺回饋；但是，在單執行緒的環境裡，UI 事件可能被耽誤，直到先前的指令已經執行完成，這降低了應用程式的效能與回應性。為了解決這個問題，應用程式需要將執行工作分割成多個程式碼路徑，亦即，由多個執行緒來執行。

多執行緒應用程式

利用多執行緒，應用程式的程式碼可被分割成幾個程式碼路徑，因此，多個操作能夠並發被執行。如果執行中的執行緒數量超過處理器的數量，真正的並發性就無法被達成，但是，排程器會在要被處理的執行緒之間迅速切換，讓每個程式碼路徑被分割成多個執行間隔，依序被處理。

多執行緒是必要的，但效能提升是有代價的 — 增加的複雜度、多出的記憶體消耗、不確定的執行順序 — 應用程式必須管理這些事情。

增加的資源消耗

執行緒會耗用記憶體與處理器，每個執行緒配置私有的記憶體區域，主要是在方法執行期間用來儲存本地變數與參數。當執行緒被建立時，該私有記憶體區域被配置，並且在執行緒終止時取消配置（亦即，只要執行緒是活躍的，它就會緊緊抓住系統的資源 — 即使它是閒置的或被阻塞的）。

處理器必須承擔建立及銷毀執行緒所需要的工作，並且在上下文切換時儲存及回復執行緒，處理器執行的執行緒越多，上下文切換就越頻繁，效能就越差。

增加的複雜度

分析單執行緒應用程式的執行相對簡單，因為執行順序已知。相反地，在多執行緒應用程式中，要分析程式如何被執行以及程式碼要以什麼順序被處理，會比單執行緒應用程式來得困難許多。不同執行緒之間的執行順序是非既定的，因為無法事先知道排程器將如何為不同執行緒分配執行時間，因此，多執行緒會為執行工作引進不確定性，這樣的不確定性不僅讓程式碼偵錯困難很多，而且協調不同執行緒的必要性將增加引進新錯誤的風險。

資料不一致

當資源存取的順序不確定時，多執行緒程式會出現一組新問題。如果兩個或多個執行緒共用資源，不知道它們會以何種順序存取及處理該資源，例如，如果執行緒 `t1` 與 `t2` 試圖修改成員變數 `sharedResource`，存取順序是不確定的 — 可能先被遞增，也可能先被遞減：

```
public class RaceCondition {

    int sharedResource = 0;

    public void startTwoThreads() {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                sharedResource++;
            }
        });
        t1.start();
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                sharedResource--;
            }
        });
        t2.start();
    }
}
```

sharedResource 暴露於**競態條件**（race condition）下，因為程式碼執行的順序每次都不一樣；無法保證執行緒 t1 總是在執行緒 t2 之前出現。在此案例中，有問題的不只是順序，而且遞增與遞減操作也是多個位元組碼指令（byte-code instruction）的結合 — 讀取、修改、與寫入。上下文切換會發生在不同的位元組碼指令之間，讓 sharedResource 的最後結果取決於執行順序：可能是 0、–1 或 1。第一種結果發生在第一個執行緒在第二個執行緒讀取它之前寫入該值，而後面二種結果發生在兩個執行緒都先讀到初始值 0，而最後進行寫入的執行緒決定最終的結果。

因為上下文切換能夠發生在一個執行緒正在執行不應該被打斷的一部分程式碼時，我們必須建立**不可切割的程式碼區域**（atomic region），那總是會依序被執行，而不會被其他執行緒介入。假如執行緒執行在不可切割的區域，其他執行緒就會被**阻塞**（blocked），直到沒有別的執行緒執行在同一個不可切割的區域。因此，Java 裡的不可切割區域被視為**互斥的**（mutually exclusive），因為它一次只容許一個執行緒進行存取。不可切割區域能夠透過不同方式被建立（參見第 20 頁的〈固有鎖與 Java 監視器〉），然而，最基本的同步化機制是 *synchronized* 關鍵字：

```
synchronized (this) {
    sharedResource++;
}
```

如果共用資源的每一個存取均被同步化，儘管有多個執行緒在存取，資料也不會發生不一致。在這本書所討論的執行緒機制中，有很多設計都是為了降低發生這類錯誤的風險。

執行緒安全

讓多個執行緒存取相同物件是讓多個執行緒快速溝通的好辦法 — 一個執行緒在寫入，另一個執行緒在讀取 — 但這樣會威脅到正確性。多個執行緒能夠同時執行相同的物件實例，因而同時存取共用記憶體裡的狀態，那會增加執行緒的風險，不是看到被更新之前的舊狀態值，就是自顧自地篡改該值。

當物件總是在被多個執行緒存取時保持正確的狀態，執行緒安全（thread safety）能夠被確保。這可以透過同步化該物件的狀態來達成，讓狀態的存取獲得控制。同步化應該被

運用於讀取或寫入任何變數的程式碼，否則，它可能會被一個執行緒存取，同時又被另一個執行緒改變。這樣的程式碼區域被稱作**臨界區段**（critical section），並且必須以不可切割的方式被執行 — 也就是說，一次只能被一個執行緒執行。同步化透過**鎖定機制**（locking mechanism）而達成，它會檢查目前是否有執行緒在臨界區段中執行，如果有的話，所有試圖進入臨界區段的其他執行緒都會被阻塞，直到該執行緒完成臨界區段的執行。

如果共用資源可以從多個執行緒存取，而且狀態是可變的 — 亦即，在資源的生命週期中，值是可以改變的 — 針對該資源的每一個存取必須由相同的鎖定（lock）加以保護。

簡言之，鎖定機制確保上鎖的區域是以不可切割的方式被執行。Android 的鎖定機制包括：

- 物件固有鎖（object intrinsic lock）
  - `synchronized` 關鍵字
- 顯式鎖（explicit lock）
  - `java.util.concurrent.locks.ReentrantLock`
  - `java.util.concurrent.locks.ReentrantReadWriteLock`

固有鎖與 Java 監視器

`synchronized` 關鍵字作用於每個 Java 物件裡頭都有隱含的固有鎖（intrinsic lock）。固有鎖是互斥的，表示不同執行緒在臨界區段中是互相排斥的，試圖存取正被佔用之臨界區段的其他執行緒都會被阻塞，並且無法繼續執行，直到該鎖定被釋放。固有鎖的作用就像監視器（參閱圖 2-2），Java 監視器可以被塑模成三種狀態：

被阻塞（*Blocked*）

在等待監視器被另一個執行緒釋放時，執行緒被暫停。

執行中（*Executing*）

擁有監視器並且正在執行臨界區段裡的程式碼的唯一執行緒。

等待中（*Waiting*）

在到達臨界區段的末端之前，自願放棄監視器所有權的執行緒。這類執行緒在它們能夠再次取得所有權之前會等待信號通知（signalled）。

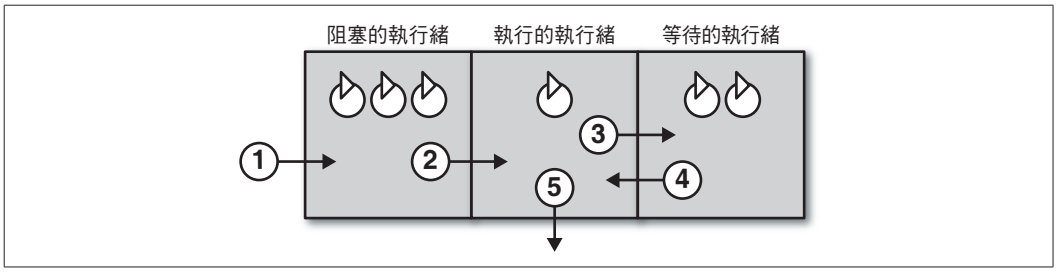

圖 2-2. Java 監視器

當執行緒到達及執行受到固有鎖保護的程式碼區塊時，它會在不同的監視器狀態之間作轉換：

1. **進入監視器**。執行緒試圖存取受固有鎖保護的區段，它進入監視器，然而，假如另一個執行緒已經取得鎖定，它就會被暫停。
2. **取得鎖定**。如果沒有其他執行緒擁有監視器，被阻塞的執行緒就能夠取得鎖定，並且執行臨界區段裡的程式碼。如果被阻塞的執行緒超過一個，排程器會選擇要執行哪個執行緒。在被阻塞的執行緒中，並沒有先進先出（FIFO）的規則；換句話說，第一個進入監視器的執行緒未必是第一個被選擇來執行的執行緒。
3. **釋放鎖定並且等待**。執行緒透過 `Object.wait()` 自行暫停，因為在繼續執行之前，它想要等待某種條件被實現。
4. **收到信號後取得鎖定**。透過 `Object.notify()` 或 `Object.notifyAll()`，等待中的執行緒收到來自其他執行緒的信號，並且在被排程器選到的情況下，再次取得鎖定。不過，等待中的執行緒並未比被阻塞的執行緒（也想要擁有鎖定）來得優先。
5. **釋放鎖定並且退出監視器**。在臨界區段的末端，執行緒退出監視器，並且騰出機會讓其他執行緒取得鎖定。

據此，這些轉變對映到同步化程式碼區塊：

```
synchronized (this) { // (1)
    // 執行程式碼 (2)
    wait(); // (3)
    // 執行程式碼 (4)
} // (5)
```

同步化共用資源的存取

能夠被多個執行緒存取及修改的共用可變狀態（shared mutable state）需要利用同步化策略在並發執行期間保持資料一致，這個策略包含針對這個狀況選擇正確的鎖定類型，以及設定臨界區段的範圍。

使用固有鎖

固有鎖能夠根據關鍵字 synchronized 的用法，以不同的方式保護共用可變狀態：

- 在方法層級上使用 synchronized 關鍵字，並且使用外層物件實例作為固有鎖：

```
synchronized void changeState() {
    sharedResource++;
}
```

- 在區塊層級上使用 synchronized 關鍵字，並且使用外層物件實例作為固有鎖：

```
void changeState() {
    synchronized(this) {
        sharedResource++;
    }
}
```

- 在區塊層級上使用 synchronized 關鍵字，並且使用其他物件作為固有鎖：

```
private final Object mLock = new Object();

void changeState() {
    synchronized(mLock) {
            sharedResource++;
    }
}
```

- 在方法層級上使用 synchronized 關鍵字，並且使用外層類別實例作為固有鎖：

```
synchronized static void changeState() {
    staticSharedResource++;
}
```

- 在區塊層級上使用 synchronized 關鍵字，並且使用外層類別實例作為固有鎖：

```
static void changeState() {
    synchronized(MyClass.class) {
        staticSharedResource++;
    }
}
```

在區塊層級同步化裡指向 this 物件的參考使用與方法層級同步化相同的固有鎖，但藉由這個語法，你可以控制臨界區段所涵蓋的確切程式碼區塊，因而只包含與你想要保護的狀態實際有關的程式碼。建立比實際所需還要大的不可切割區域是不好的實務做法，因為那可能導致其他執行緒產生不必要的阻塞，造成應用程式的執行變緩慢。

以其他物件作為固有鎖的同步化能夠在類別裡使用多個鎖定，應用程式能夠努力使用它自己的鎖定保護每個獨立狀態。因此，如果類別擁有多個獨立狀態，效能會因為使用數個鎖定而被提升。

synchronized 關鍵字能夠作用在不同的固有鎖上。記住，靜態方法的同步化作用在類別實例的固有鎖，而不是物件實例。

使用顯式鎖定機制

如果需要更進階的鎖定策略，可以使用 ReentrantLock 或 ReentrantReadWriteLock 類別代替 synchronized 關鍵字。臨界區段透過在程式碼裡明確地鎖定與解鎖特定區域而受到保護：

```
int sharedResource;
private ReentrantLock mLock = new ReentrantLock();

public void changeState() {
    mLock.lock();
    try {
        sharedResource++;
    }
    finally {
        mLock.unlock();
    }
}
```

synchronized 關鍵字與 ReentrantLock 具有相同的語義：在有執行緒進入臨界區段時，它們都會阻塞試圖執行臨界區段的所有其他執行緒。這是一種假設所有並發存取都有問題的防禦策略，然而，讓多個執行緒同時讀取共用變數並無害，因此，在這種情況下，synchronized 與 ReentrantLock 會是過度保護的。

ReentrantReadWriteLock 讓不同的讀取執行緒可以並發執行，但還是阻塞讀取執行緒 vs. 寫入執行緒，以及寫入執行緒 vs. 寫入執行緒的狀況：

```
int sharedResource;
private ReentrantReadWriteLock mLock = new ReentrantReadWriteLock();

public void changeState() {
    mLock.writeLock().lock();
    try {
                sharedResource++;
        }
        finally {
                mLock.writeLock().unlock();
        }
}

public int readState() {
        mLock.readLock().lock();
        try {
                return sharedResource;
        }
        finally {
                mLock.readLock().unlock();
        }
}
```

ReentrantReadWriteLock 相對複雜，導致必須在效能上付出一些代價，原因是，相較於 synchronized 與 ReentrantLock，它需要花更多功夫去判斷執行緒應該允許被執行或者被阻塞，因此，在讓多個執行緒同時讀取共用資源所得到的效能提升與增加判斷複雜度所造成的效能損失之間會有個平衡點。ReentrantReadWriteLock 的典型使用案例是發生在具有大量讀取執行緒以及少量寫入執行緒的狀況。

範例：消費者與生產者

執行緒協同合作的常見使用案例是*消費者與生產者*（consumer-producer）設計模式 — 亦即，一個執行緒產生資料，一個執行緒消耗資料。這些執行緒透過在它們之間共用的清單（list）協同合作。當清單未滿時，生產者執行緒將項目（item）添加到清單，而消

費者執行緒在清單非空時移除項目。假如清單滿了，生產者執行緒會被阻塞，假如清單空了，消費者執行緒會被阻堵塞。

ConsumerProducer 類別包含共用資源 LinkedList 與兩個方法：produce() 增加項目，consume() 移除項目：

```
public class ConsumerProducer {

    private LinkedList<Integer> list = new LinkedList<Integer>();
    private final int LIMIT = 10;
    private Object lock = new Object();

    public void produce() throws InterruptedException {

        int value = 0;

        while (true) {
            synchronized (lock) {
                while(list.size() == LIMIT) {
                        lock.wait();
                }
                list.add(value++);
                lock.notify();
            }
        }
    }

    public void consume() throws InterruptedException {

        while (true) {
            synchronized (lock) {
                while(list.size() == 0) {
                    lock.wait();
                }
                int value = list.removeFirst();
                lock.notify();
            }
        }
    }
}
```

produce 與 consume 使用相同的固有鎖保護共用清單，試圖存取清單的執行緒會被阻塞，直到另一個執行緒釋放固有鎖，然而，假如清單滿了，生產者執行緒會放棄執行（亦即，wait()）；假如清單空了，消費者執行緒會放棄執行。

當項目從清單中被移除或增加時，監視器（monitor）接獲信號通知 — 亦即，`notify()` 被呼叫 — 因此，等待的執行緒能夠再次執行。消費者執行緒通知生產者執行緒，反之亦然。

下列程式碼顯示執行生產操作與消費操作的兩個執行緒：

```
final ConsumerProducer cp = new ConsumerProducer();

new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            cp.produce();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}).start();

new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            cp.consume();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}).start();
```

任務執行策略

為了確保多執行緒正確地被用來建立具回應性的應用程式，應用程式應該被設計成具有建立執行緒與執行任務的能力，兩種未達最佳標準的設計與極端狀況如下：

所有任務使用一個執行緒

所有的任務在相同的執行緒上被執行，結果往往是不具回應性的應用程式，而且無法利用有效的處理器。

每個任務使用一個執行緒

針對每一項任務總是有一個新的執行緒被啟動及終止，假如任務經常被建立並且具有短暫的生命週期，執行緒建立與銷毀的成本會對效能產生相當程度的影響。

雖然這些極端狀況應該被避免，但是它們表現出循序執行與同步執行的最極端狀況：

循序執行

多個任務循序被執行，一個任務必須在處理下一個任務之前先被完成。因此，任務的執行時間不重疊。這種設計的優點是：

- 它天生是執行緒安全的。
- 能夠在一個執行緒上執行，消耗的記憶體比多執行緒來得少。

缺點包括：

- 生產力不佳。
- 每個任務的開始皆取決於前一個任務的結束，任務的開始可能會被延宕，或甚至根本不會被執行。

並發執行

任務並行且相互穿插地被執行，優點是有較好的 CPU 利用率，缺點是這種設計並非天生是執行緒安全的，因此，可能需要做一些同步化處理。

有效的多執行緒設計利用兼具循序與並發執行的執行環境；如何選擇取決於任務的特性。隔絕及獨立的任務能夠並發執行，提高生產力，然而，需要特定順序或者共用沒有同步化之共通資源的任務應該循序被執行。

並行執行設計

並發執行可以透過許多方式來實作，因此，設計必須考慮如何管理執行緒的數量以及它們的關係，基本原則包括：

- 取代總是建立新的執行緒，改為重利用執行緒，減少建立及銷毀資源的頻率。
- 不使用超過實際需要的執行緒，使用的執行緒越多，耗用的記憶體與處理器時間就越多。

總結

Android 應用程式應該是多執行緒的，不管是在單處理器或多處理器的平台上，皆能夠提升效能。多執行緒可以共享單處理器上的執行資源，或者在多處理器上有效地利用真實的並發性。效能的提升伴隨著複雜度的增加，以及保護共用資源和維護資料一致性的責任。