由 Kotlin 推出的 kotlinx-coroutine 函式庫是瞄準非同步任務的解決方案，目的是用來解決在非同步任務上會遇到的問題，所以要學習 Coroutine，我們首先需要從非同步任務的基礎開始講起。

非同步任務 aka 異步任務，能夠讓系統在同時間內執行多個任務，比起循序程式設計它需要考慮的事情更多，譬如：在不同的執行緒上共享資料、任務結束後要如何從不同的執行緒上取回運算結果、要如何中斷執行緒…

本章將從循序程式設計開始介紹，了解循序程式設計與非同步程式設計的差異。非同步任務通常是使用多個執行緒來解決，要了解執行緒，就需要從系統的角度來認識。接著，為了要在不同作業系統上都可以呼叫相同的方法來使用執行緒，因此採用執行緒函式庫來作為應用程式與作業系統內執行緒的橋樑，知道如何使用執行緒後，最後來談談使用執行緒會遇上的問題。

1-1 當循序程式設計遇上耗時任務

程式是用來解決生活中的問題，換句話說，程式是由真實世界的事件抽象而成的。在我們學習程式語言初期階段，最先學習到的是循序程式設計（Sequential Programming），在這種程式設計方式之下，所有的程式按照排列的順序來執行，會由這種方式開始，是因為它最直覺、也最容易實作。

用真實世界的範例 - 大隊接力，講述循序程式設計的概念：起跑槍響之後，選手依照排定的順序一個接著一個的往前跑，當選手跑完自己負責的距離後，將接力棒傳給下一棒，下一棒選手接棒後便開始繼續往下跑，最後一名選手抵達終點後，比賽就結束，最後只剩下跑步成績。將起跑槍響看作是呼叫函式，而每一位選手可以看作是子函式，而選手在跑步則是執行程式，將接力棒傳給下一棒的動作可以當作是把子函式的回傳值傳給下一個子函式當

作輸入，當最後一名選手跑到終點後，這個函式就結束了，最後得到的就是跑步成績，跑步成績對應的就是函式的輸出值。

回到程式端，假如有一個名為 **displayLatestContents** 的函式，其目的為當使用者輸入帳號密碼登入後，系統會給予一個 Token，我們可以利用這個 Token 查詢登入的使用者相關資訊。利用 Token 去向系統詢問最新的內容，系統在收到我們傳入的 Token 後會回傳最新的內容傳出來，最後將內容呈現在畫面上。根據描述，這個函式應包含三個步驟：

1. 登入，並取得 Token。
2. 根據 Token 取得最新的內容。
3. 將最新的內容呈現在畫面上。

按照上面的步驟，我們完成了以下的 Pseudo Code：

```
fun displayLatestContents() {
    val token = login(userName, password)
    val contents = fetchLatestContents(token)
    showContents(contents)
}
```

login 函式會回傳 token，**fetchLatestContents** 函式會使用 login 函式所回傳的 Token 來取得最新的內容，最後則是把 **fetchLatestContentes** 函式回傳的結果傳給 **showContents** 函式顯示內容。

其中三個函式的定義如下：

```
fun login(userName: String, password: String): Token {...}

fun fetchLatestContents(token: Token): List<Content> {...}

fun showContents(contents: List<Content>) {...}
```

Coroutine 是由 cooperation 加上 routine 所組合而成的複合字，cooperation 指共同合作，這裡的 routine 意指 function、method，意思是協同處理多個程序（協程）。

> *Routine(n.) - a sequence of computer instructions for performing a particular task*
>
> *Merriam-webster*

前一章提到為了不讓耗時任務影響使用者體驗，我們會建立新執行緒讓這些耗時任務在不同的執行緒上執行，避免佔用主執行緒，這種任務的執行方式稱為非同步任務。在非同步的任務有些地方需要特別注意的，例如要如何在不同執行緒上取回結果，頻繁的 Context Switch 會造成系統的負擔 ... 等等，而 Coroutine 就是瞄準解決使用執行緒上的問題，讓開發者能夠用更直覺、容易地方式來解決這些問題。在本章中，我們從執行緒與 Coroutine 的差異開始講起，接著介紹 Coroutine 的架構，最後是 Kotlin Coroutine 的三大要素，從本章開始，我們將逐步進入 Coroutine 的領域，揭開神秘的面紗。

2-1 在專案中使用 Coroutine

在 Kotlin 官方 **`kotlinx.coroutine`** 函式庫中，包含了多個部分：core、ui、test…本章及之後章節主要是進行核心部分的介紹，只需要加上 core 即可。

開始進入本章的範例之前，先把 kotlinx.coroutines-core 加入至專案中，方法如下：

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.6.4")
}
```

編寫本書時，Coroutine 的版本為 1.6.4。

Coroutine 常被拿來與執行緒做比較，那麼它們兩個到底有什麼不同呢？

2-2 搶佔式多工 VS 協同式多工

2-2-1 搶佔式多工

首先，我們知道每一個應用程式有一個行程以及至少一個執行緒。行程有一塊獨立的記憶體用來把自己應用程式所需的資源與其他應用程式的資源隔離開來，並且依照需求建立執行緒來執行任務，行程可以看作是容器，執行緒可以看作是執行單位。

雖然每一個行程可以建立一個或多個執行緒，但是它們與其他行程所共同分享的是 CPU 的使用時間，系統會根據執行緒的優先權來分配 CPU 時間給執行緒來使用，也就是說，只要有一個優先權較高的任務準備好，就可以中斷當下優先權較低的任務。這種由系統根據執行緒的優先權來分配 CPU 時間的多工方式稱作**搶佔式多工（Preemptive Multitasking）**。

執行緒優先權的設定有三點需要特別注意：

1. 我們使用的是執行緒函式庫提供的優先權，雖然在 Thread.java 中有三個優先權可以設定（MIN_PRIORITY、NORM_PRIORITY 以及 MAX_PRIORITY），不過卻不一定能一對一的對應到各個平台的執行緒優先權。
2. 設定優先權只是告知系統這邊有一個優先權比較高的任務，系統會不會分配 CPU 時間給這個任務，最終的決定權還是交由系統作決定。
3. 如果每件任務都把優先權設成 MAX_PRIORITY，那麼其實跟沒設定一樣。

3-2 async 建構器

3-1 小節已介紹無回傳值的 launch 建構器，用來建立一個 Coroutine 區塊執行無回傳值的非同步任務；而另一種建構器則是用來建構一個具有回傳值的 Coroutine 區塊，也就是本節所要介紹的 **async**。

3-2-1 async 簡介

開始之前，先想想看在實際案例中，有什麼情況需要執行非同步任務並包含回傳值，例如：登入任務 - 登入通常是透過 API 的方式將資料送往伺服器端，而伺服器將傳入的資料處理之後，將登入結果返還給呼叫端（用戶端）。

async 與 **launch** 相同，都是屬於 **CoroutineScope** 的擴充函式，而函式的參數與 launch 一樣都包含了三個項目：**CoroutineContext**、**CoroutineStart** 以及 **suspend CoroutineScope.() -> T**。與 launch 不同的是 **async** 的回傳值是繼承 Job 介面的 **Deferred** 型別。在實際應用上，因無法確定一個非同步任務從呼叫到取得結果的時間，必須要等到該任務結束後，才有辦法得到正確的結果，所以只要在需要結果的地方呼叫 **await** 函式，呼叫端就會等到 **async** 有結果時才可以繼續執行。

```
public fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T> {
    ...
}
```

async 函式

```
public interface Deferred<out T> : Job {
    ...
    public suspend fun await(): T
}
```

Deferred 介面

在 Example 3-4 中，同樣使用 runBlocking 建立 Coroutine 區塊，為了要從 Coroutine 回傳結果，改成使用 async 建立一個 Coroutine 區塊。因 Kotlin 高階函式最後一行就是回傳值，所以 async 區塊會回傳 true。

```
fun main() = runBlocking{
    val result = async {
        delay(100)
        true
    }
    println("Start async task")
    println("Result is ${result.await()}")
}
```

Example 3-4，含有回傳值的非同步任務

```
Start async task
Result is true
```

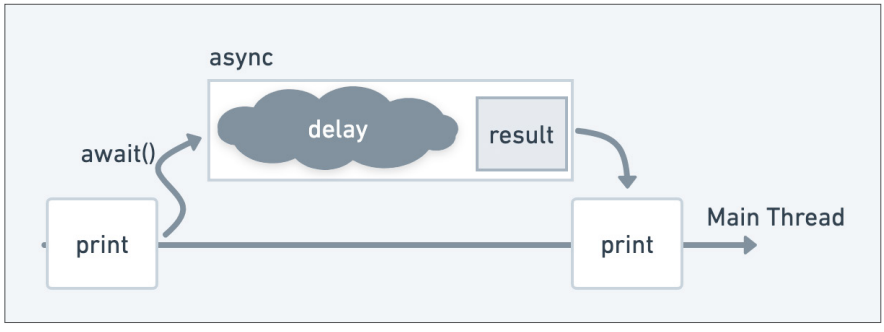


圖 3-3　使用 async 建立含有回傳值的 Coroutine

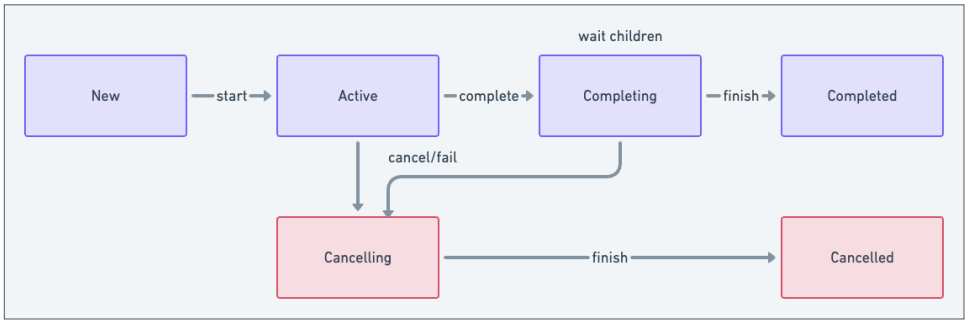


圖 4-1　Job 生命週期

因為每一個 Job 皆包含了其生命週期以及子 Job，當任務結束之後，會等待所有子 Job 完成，如此就能夠避免因父 Job 任務結束後子 Job 還沒結束時所造成的問題，如資源無法收回 ...。另外，取消父 Job 也會同時取消子 Job，這讓我們能夠更輕鬆的執行多個併發任務，而不用擔心考慮其例外狀態。

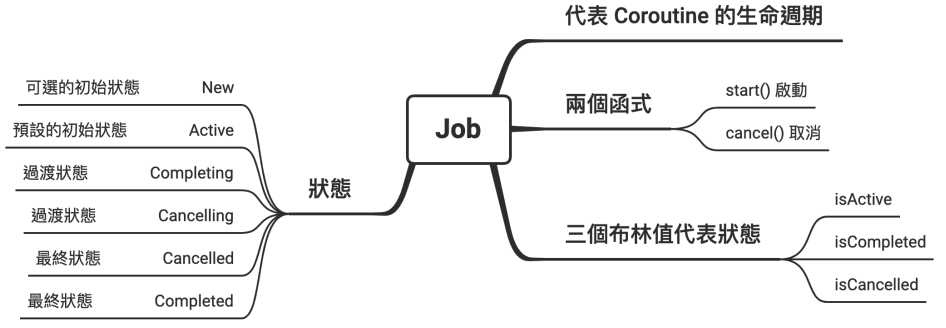
從 Job 的程式碼可以發現，並沒有一個專屬的屬性是用來儲存其狀態的，取而代之的是使用三個布林值：**`isActive`**、**`isCancelled`** 以及 **`isCompleted`**。下表展示了這三個布林值與狀態值的關係。

狀態	isActive	isCompleted	isCancelled
New（可選的初始狀態）	false	false	false
Active（預設的初始狀態）	true	false	false
Completing（過渡狀態）	true	false	false
Cancelling（過渡狀態）	false	false	true
Cancelled（最終狀態）	false	true	true
Completed（最終狀態）	false	true	false

表 4-1　Job 的狀態值

在第三章介紹 launch 時有提到，使用 launch 建立 Coroutine 是會立刻啟動的，因為 Job 的初始值是 **Active**，所以我們不需要另外呼叫 **`start`** 函式啟動。如果不希望自動啟動，只要在建立 Coroutine 時，將 CoroutineStart 使用 CoroutineStart.LAZY 帶入，如 **`launch(start = CoroutineStart.LAZY)`**，如此就可以依照需求在特定的位置呼叫 **`start`** 函式啟動任務，而用 CoroutineStart.LAZY 建立的 Job 的狀態為 **New**。

心智圖



4-3 取消任務

4-3-1 使用 delay 函式取消任務

Coroutine 比起執行緒厲害的地方，其中一點是 Coroutine 能夠輕鬆的取消任務。Example 4-1 使用 runBlocking 建立一個 Coroutine 作用域，並使用兩個 launch 各自建立 Coroutine 來執行非同步任務，將每一個 launch 的 Job 分別存在變數 - job1, job2 內。其中，任務一：暫停 100 毫秒，列印 Job1 done，任務二：暫停 1000 毫秒，列印 Job2 done。在這兩個 launch 區塊底下，列印 start launch，暫停 300 毫秒後呼叫第二個任務的 **`cancel`** 函式取消該任

在前面的章節，我們知道如何使用 Coroutine 建構器（launch、async）建立一個 Coroutine 區塊，在呼叫 **launch** 或 **async** 時，可以帶入不同的調度器（Dispatcher）選擇不同的執行緒、執行緒池。除了調度器之外，我們也可以將 Job 傳入建構器中，如此我們就能夠在不同的任務中使用相同的 Job 來控制其生命週期。那麼為什麼調度器、Job 都能夠傳入建構器中呢？

6-1 CoroutineScope

在前幾章的範例中，我們都是在 **main()** 內使用 **runBlocking** 來啟動 Coroutine。而 **launch** 以及 **async** 若沒有在 **runBlocking** 中呼叫，就會出現錯誤訊息 **Unresolved reference: launch**。

```
fun main() {
    launch { // compile error
        println("run launch block")
    }
}
```

Example 6-1，在 runBlocking 以外無法使用 launch

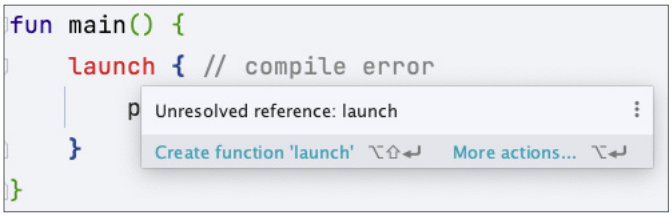


圖 6-1　launch 未在 runBlocking 內使用

為什麼 launch 一定要在 runBlocking 裡面才能使用呢？

首先，我們先來看一下 **runBlocking** 的函式簽名：

```
public actual fun <T> runBlocking(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T {
    ...
}
```

在 **runBlocking** 函式內有兩個參數，**CoroutineContext** 以及 **suspend CoroutineScope.() -> T**。

- **CoroutineContext**：將 Coroutine 的內容存在 Context 中，並傳遞給子 Coroutine。
- **suspend CoroutineScope.() -> T**：**CoroutineScope** 的擴充函式，且是一個 **suspend** 函式。

在 Kotlin 內，假如函式的最後一個參數是函數型別（Function Type），就能夠使用 lambda 表達式來呈現。所以 **runBlocking** 的大括弧就代表了 **suspend CoroutineScope.() -> T**。

所以為什麼 **launch** 以及 **async** 都能在 **runBlocking** 裡執行呢？因為它們都是 **CoroutineScope** 的擴充函式。

launch

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job{
    ...
}
```

前一章介紹如何使用 Channel 來處理多個非同步的任務，Flow 與 Channel 相同，都是用來處理多個非同步任務的解決方案。那麼，這兩種方式有什麼差異呢？

Flow 是用來處理非同步資料流的一種方式，它會按照發射（emit）的順序來執行。

> *An asynchronous data stream that sequentially emits values and completes normally or with an exception.*

將非同步任務透過 Flow 的方式發送，在結果被接收之前，這個非同步任務都不會被執行，甚至可以在執行之前透過一些函式來將這些資料轉換成我們所希望的樣子。

跟 Channel 不太一樣的地方是，Channel 一次取出一個值，而 Flow 取出的是一個資料流（Stream）；換句話說，使用 Channel 時，我們必須要呼叫多次的 **`receive`** 函式來接收 **`send`** 函式傳送出來的值，而 Flow 只需要使用 **`collect`** 函式就可以處理這個資料流內的所有內容。

8-1 第一個 Flow

Kotlin 提供多種方式建立 Flow，在 Example 8-1（使用 **`flow`** 函式來建立）中，在 **`flow{}`** 內使用 **`repeat(10)`** 重複執行十次任務，而每次執行時都會先呼叫 **`delay(100)`** 暫停 Coroutine，並將當下的 it 使用 **`emit(it)`** 發射出去。**`flow{}`** 建立出來的函式並不是一個 suspend 函式，原因是 Flow 會等到接收的時候才會去執行，所以接收的函式才會是一個 suspend 函式。

```
fun firstFlow(): Flow<Int> = flow {
    repeat(10) {
        delay(100)
        emit(it)
    }
}
```

Example 8-1，第一個 Flow

另外，在 **flow{ }** 裡包含著一個 suspend 函式 - **delay**，所以也就是表示這個 Lambda 表達式是一個 suspend 函式（因為 suspend 函式只能在 suspend 函式裡被執行）。

flow{ } 函式的簽名：

```
public fun <T> flow(@BuilderInference block: suspend
FlowCollector<T>.() -> Unit): Flow<T> = SafeFlow(block)
```

flow{ } 內的確包含了一個 suspend 函式：**suspend FlowCollector<T>.() -> Unit**。

在 Example 8-1 的 **flow{ }** 裡，最後使用了 **emit** 函式將整數傳進資料流中。

而 **emit** 函式其實就是 FlowCollector 介面的函式。

```
public interface FlowCollector<in T> {

    /**
     * Collects the value emitted by the upstream.
     * This method is not thread-safe and should not be invoked
concurrently.
     */
    public suspend fun emit(value: T)
}
```

在前面的文章中，我們的程式都是直接跑在 **main()** 函式，並且使用 **runBlocking** 來建立一個 Coroutine 作用域，不過 **runBlocking** 是一種會阻塞執行緒的作用域，當我們在 runBlocking 區塊內執行我們的 Coroutine 程式時，將會阻塞目前的執行緒。

不過這跟測試有什麼關係呢？

如果我們在測試的程式碼中，直接使用 **runBlocking** 來呼叫我們的 suspend 函式，且在這些 suspend 函式內如果有像是 **delay** 函式會暫停目前 Coroutine 的函式，那就會真的需要耗費那麼長的時間，這麼一來，就會影響我們執行測試的時間。

另外，我們會使用 **Dispatcher.Main** 讓 Coroutine 在主執行緒上執行，要使用 **Dispatcher.Main** 則需要依據不同的平台加入不同的函式庫，不過只能在正式的環境下使用，在單元測試的環境下則無法，因為單元測試的環境與正式的環境不太相同。例如：Android 的主執行緒只存在於 Android 環境下，所以在單元測試的環境底下，會無法使用 Android 的主執行緒，因為是使用本機的 JVM 環境而不是 Android 環境。

9-1 | kotlinx-coroutines-test

Kotlin Coroutine 提供了一個用於測試的函式庫，要使用 Coroutine 測試相關的函式，需要在 **build.gradle.kts** 將相依套件加入至專案內：

```
dependencies {
  testImplementation('org.jetbrains.kotlinx:kotlinx-coroutines-
test:1.6.4')
}
```

9-2 runTest

在底下的 Example 9-1 裡，UserRepo 類別裡有一個名為 **fetchUserName** 的 suspend 函式，我們使用 **withTimeout** 函式限制呼叫的時間，當執行的時間沒有超過 1000 毫秒，我們便可以將取得的值回傳；反之，**withTimeout** 就會拋出 **TimeoutCancellationException**。而為了方便測試，我們將取值的 Service 注入，而不是在類別中建立。

```
class UserRepo {
    suspend fun fetchUserName(service: Service, id: Int = 0): String {
        return try {
            val name = withTimeout(1000) {
                service.getName(id)
            }
            name
        } catch (e: TimeoutCancellationException) {
            "fail"
        }
    }
}
```

Example 9-1，UserRepo 類

其中，Service 是一個簡單的介面，內含一個 suspend 函式：**getName**。

```
interface Service {
    suspend fun getName(id: Int): String
}
```

fetchUserName 函式有兩個部分需要被測試：一個是沒有超時；另一個則是超時的情況。我們可以讓注入的 Service 依照我們的需求給定預期的結果。