01

使用 JUnit 執行單元測試

本章提要

1.1 正式程式碼與測試程式碼的差異

在撰寫**正式程式碼 (production code)** 時，我們需要同時撰寫**測試程式碼 (test code)** 以確保正式程式碼的執行結果可以如規格或需求所預期。

因為兩者目的不同，在撰寫程式碼時標準也不同。對於正式程式碼，著眼於未來程式碼的維護性，我們會依循一些常見的物件導向開發 (Object Oriented Programming, OOP) 原則，如 Robert C. Martin 提出的「SOLID」，SOLID 分別是 SRP、OCP、LSP、ISP、DIP 等原則的首字縮寫詞，分別是：

1. 單一責任原則 (Single Responsibility Principle, SRP)
2. 開放封閉原則 (Open-Closed Principle, OCP)
3. 里氏替換原則 (Liskov Substitution Principle, LSP)
4. 介面分割原則 (Interface Segregation Principle, ISP)
5. 依賴反轉原則 (Dependency Inversion Principle, DIP)

對於測試程式碼，因為希望一個單元測試方法可以滿足一個測試情境，通常具備 Given、When、Then 的三段式結構：

1. Given：在什麼條件下
2. When：做什麼事情
3. Then：預期得到什麼結果

所以程式碼看起來比較瑣碎，不會太計較重複的程式碼，以看得清楚每一個單元測試的邏輯為優先考量。又一個單元測試像是在確認一項正式程式碼的規格需求，所以方法的命名也可以很長，常見使用底線區隔個別單字，不計較駝峰規則 (camel case)，以說明清楚為原則。

1.2 單元測試常見名詞說明

撰寫單元測試和小時後做自然科學實驗很像，老師常說：「做一個成功的實驗需要掌控會影響實驗結果的變因」。這些變因通常包含：

1. 操作變因：實驗中唯一能改變的因素，即實驗組與對照組不同的因素。
2. 控制變因：實驗中維持不變的眾多因素，即實驗組與對照組相同的因素。
3. 應變變因：實驗結果或關注項目。

所以當我們設定好一個環境，把所有**控制變因**都保持固定後，就開始改變**操作變因**，給不同的輸入得到不同的結果，這結果就是**應變變因**。比如說我們要了解「新冠病毒對身體的影響」，則是否有新冠病毒，或該病毒強弱，就是操作變因，身體就是應變變因，實驗過程中自然不能有其他病毒介入，就是控制變因。

撰寫單元測試的原理相似，先介紹一些常見英文名詞：

SUT (System Under Test)

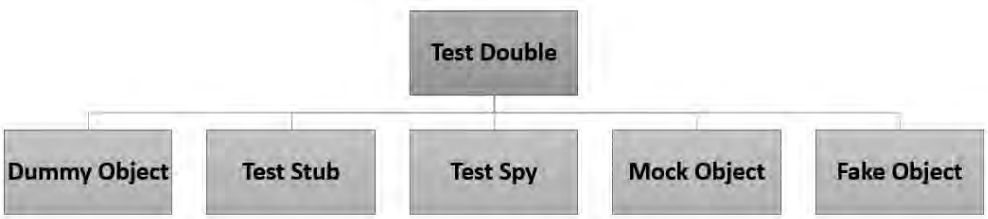
一個單元測試裡會有一個要測試的目標類別或方法，稱為 **SUT**，即為「**待測試元件**」，可以對比前述的應變變因。

DOC (Depended-On Component)

在物件導向的世界中，待測試類別 SUT 會有協同作用的其他類別，稱為 **DOC**，即為「**依賴元件**」，改變 DOC 的行為會影響 SUT 測試結果，如同實驗中我們挑選多個控制變因的其一作為操作變因，改變操作變因就會改變應變變因的結果。

Test Double

要把 DOC 控制不變，如同控制變因，需要控制協同作用的 DOC 類別的方法呼叫結果能符合我們預期；做法通常是以繼承 DOC 類別來建立測試用途的物件，並替換原本的 DOC。這些配合測試而建立的物件又可以再細分不同種類，如圖 1-1，統稱為 **Test Double**，常翻譯為「**測試替身**」，會在後續章節以程式範例說明。這些測試替身不需要和真實 DOC 一樣具備完整功能，只要能讓 SUT 運作正常即可：



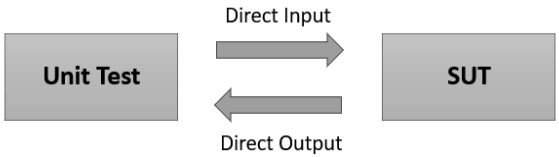
▲ 圖 1-1 測試替身分類

電影業需要拍攝對主角有潛在風險或危險的場景時，他們會聘請「特技替身」來代替演員在場景中的表演。特技替身是受過良好訓練的個人，具有「特定技能」能夠滿足現場的特定要求。他們可能無法全方位精通，但他們知道如何表現從高處墜落、撞車或現場需要的技術。我們也會為不同場景找不同種類的測試替身，所以有上圖的分類。

Direct Input/Output

單元測試的程式碼如同一般用戶端，將建立 SUT 物件，然後提供輸入參數呼叫 SUT 的方法，進而改變 SUT 的狀態，這些輸入稱為「Direct Input (直接輸

入)」。經由 SUT 的方法運算後，若回傳特定結果，稱為「Direct Output (直接輸出)」，如下圖：



▲ 圖 1-2 直接輸入 & 直接輸出

Indirect Input

事實上系統中只有極少數類別可以獨立運行並實現預期功能；絕大多數類別都會有與其協作的相依類別進行交互作用，以提供完整的功能，這就是先前討論的 SUT 與 DOC 的關係。當 SUT 狀態被 DOC 運作的結果所影響，就稱這些 DOC 結果是 SUT 的「Indirect Input (間接輸入)」，如下圖：

▲ 圖 1-3 間接輸入

Test Stub

對於提供 Indirect Input (間接輸入) 給 SUT 的 DOC，可以使用 Test Double (測試替身) 中的「Test Stub」來替換。英文單字 stub 常見翻譯為「存根」或「殘端 / 枝」，筆者認為「殘端 / 枝」可能是比較好的解釋，因為 Test Stub 相較於完整 DOC，通常只實作被 SUT 呼叫的方法，如同殘枝的殘缺不全。Test Stub 有實作的少數方法通常固定做某些簡單的事，或回傳某些固定值給 SUT。以一般開發網站應用程式為例，Controller 會呼叫 Service；若要對 Controller 進行單元測試，就要對 Service 進行控制：

1. 以 Test Stub 實作 Service 的 interface 然後取代原本的 Service。
2. 若 Controller 需要呼叫 Service 的 A 方法，就要實作 Test Stub 的 A 方法以得到預先控制的目的，但不需要 Test Stub 的全部方法都予以控制。

Indirect Output

當 SUT 的「某些方法」的執行結果無法藉由本身的其他方法取得狀態變化，只能依靠由 SUT 輸出至 DOC 而造成 DOC 的變化來理解，就稱這些方法為「Indirect Output (間接輸出)」，如圖 1-4。

常見能讓我們知道 SUT 變化的間接輸出如：

1. SUT 傳送訊息至 MQ 或 JMS。
2. SUT 新增資料至資料庫。
3. SUT 寫資料到檔案等。

後續就可以藉由觀察 MQ、JMS、資料庫、檔案等 DOC 的改變來判斷 SUT 的狀態變化：

▲ 圖 1-4 間接輸出

Mock Object

要觀察 SUT 的 Indirect Output (間接輸出)，可以使用測試替身中的 Mock Object 攔截輸出，再與預期值進行比較。以前例 Controller 呼叫 Service 的情境來說，對 Controller 進行單元測試時，除了要有 Test Stub 讓 Controller 會呼叫 Service 的 A 方法時不會拋出 NullPointerException，有些時候我們也要釐清 Service 的 A 方法實際上有無被呼叫？被呼叫的次數是否如預期？這時候就需要 Mock Object 攔截間接輸出，如同過濾器或攔截器的概念。實務上 Mock Object 通常也具備 Test Stub 的功能，在之後要介紹的 Mockito 框架中，Mock Object 可以取代 Test Stub。

Test Spy

Test Spy 是強化版的 Test Stub，除了需要提供 Indirect Input 給 SUT，執行時也可以捕獲 SUT 的 Indirect Output 並保存它們以供測試驗證。

Fake Object

Fake Object 具備和 DOC 相似的功能，通常是簡化版，當真實 DOC 不可用於測試、或用於測試時速度太慢將使用 Fake Object。以 DAO 為例，真實 DAO 將存取實體資料庫，對於測試而言效率太差，因此 DAO 的 Fake Object 就可以 HashMap 取代資料庫，或是使用內嵌於記憶體的資料庫以提高測試效率！

Dummy Object

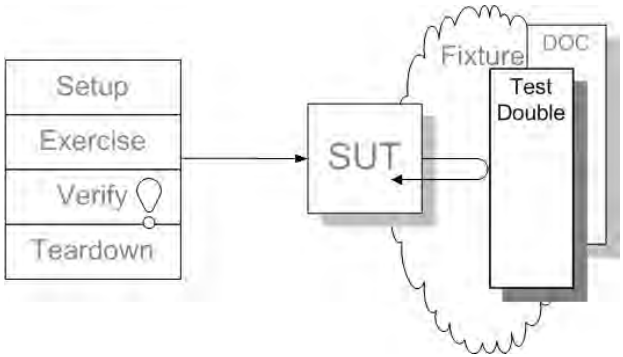
SUT 的某些方法可能需要某些物件參考作為參數，但若因為測試情境的關係，該參數實際上不會影響測試結果，我們就可以傳入一個「Dummy Object」，可能是一個 null 的物件參考、或是 Object 類別的物件實例、或是符合型別而沒有欄位狀態的簡單物件等等，目的是讓測試正常進行。

Test Fixture

測試時需要一些裝置或環境，如測試用的資料，稱為「Test Fixture (測試裝置)」，或稱為「Test Context (測試情境)」。

單元測試架構

下圖是節錄自 xunitpatterns.com 網站的單元測試架構示意：



▲ 圖 1-5 單元測試架構示意 (http://xunitpatterns.com/Test Double.html)

可以把目前理解的名詞與觀念做一串聯：

1. 左側代表一個單元測試程式碼的結構，包含：
   - Setup：初始設定。
   - Exercise：測試活動。
   - Verify：驗證結果。
   - Teardown：測試結束，回收測試資源。
2. 單元測試程式碼將直接操作 SUT。
3. SUT 坐落在 Fixture 上，同時還有 DOC。
4. 以 Test Double 取代 DOC，因此 SUT 與 Test Double 互動。

1.3 使用 JUnit 5

1.3.1 簡介 JUnit 5

JUnit 是相當廣泛使用的 Java 測試框架，通常會再搭配其他測試框架如 Mockito 使用，目前已經相當成熟。JUnit 5 是本書出版時最新版測試框架 (2017 年釋出)，和前版本 JUnit 4 (2006 年釋出) 就開發上最大的差異應該是支援 Java 8 的 lambda 表達式，但架構上的設計也允許向前相容，因此依然可以執行用 JUnit 4 編寫的測試。

與 JUnit 4 相比，JUnit 5 由 3 個子項目組成，分別是：

1. JUnit Platform
2. JUnit Jupiter
3. JUnit Vintage

在 JUnit 5 的架構下要執行單元測試，需要在 Java 工具上具備基礎的「測試平台 (platform) 」，然後以「測試引擎 (engine) 」執行開發的「測試程式碼 (code) 」。

3 個子項目簡單說明如下：

JUnit Platform

為了能夠在 JVM 上進行單元測試，包含使用 IDE 開發工具如 Eclipse、構建工具 (build tools) 如 Gradle 與 Maven、或擴充插件 (plugins) 等，必須先具備測試平台，就是指 JUnit Platform。

JUnit Jupiter

JUnit Jupiter 是 JUnit 5 裡最直接影響開發者編寫測試程式碼的部分，主要包含 2 個函式庫：

1. junit-jupiter-api
   我們使用 API 裡的函式庫來撰寫測試和進行套件擴充，包含新增的標註 (annotation) 類別和 lambda 表達式。

2. junit-jupiter-engine
   要執行符合 Jupiter 編寫的測試程式碼，執行時期就需要 junit-jupiter-engine 提供的測試引擎。

JUnit Vintage

JUnit 5 藉由 JUnit Vintage 達成向前相容的功能性，只要把測試引擎由 junit-**jupiter**-engine 改為 junit-**vintage**-engine，就可以執行以 JUnit 4 編寫的舊版測試程式碼。

後續我們以專案「ch01-junit5-helloWorld」與「ch01-junit5-run-junit4」來驗證前述說明。

1.3.2 建立第一個 JUnit 5 測試專案

專案「ch01-junit5-helloWorld」的 pom.xml 設定 **junit-jupiter-engine** 的依賴項目如下行 4：

範例：/ch01-junit5-helloWorld/pom.xml

```
1 <dependencies>
2     <dependency>
3         <groupId>org.junit.jupiter</groupId>
4         <artifactId>junit-jupiter-engine</artifactId>
5         <version>5.7.2</version>
6         <scope>test</scope>
7     </dependency>
8 </dependencies>
```

雖然只有設定 junit-jupiter-engine，實際上 Maven 卻包含了 JUnit Platform 與 JUnit Jupiter 等 JUnit 5 的 2 個子項目的相關函式庫，如下圖：

▲ 圖 1-6 Maven 引入的函式庫

撰寫簡單的計算器類別：

範例：/ch01-junit5-helloWorld/src/main/java/lab/Calculator.java

```
1  public class Calculator {
2      public int add(int a, int b) {
3          return a + b;
4      }
5      public int subtract(int a, int b) {
6          return a - b;
7      }
8  }
```

接著撰寫單元測試。我們故意分拆兩個測試類別 CalculatorAddTest 與 CalculatorSubtractTest，單元測試以類別裡的方法為單位，在執行測試的方法上標註 **@Test**，如以下範例行 6：

範例：/ch01-junit5-helloWorld/src/test/java/lab/CalculatorAddTest.java

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2  import org.junit.jupiter.api.DisplayName;
3  import org.junit.jupiter.api.Test;
4
5  public class CalculatorAddTest {
6      @Test
7      @DisplayName("1 + 1 = 2")
8      void addsTwoNumbers() {
9          Calculator calculator = new Calculator();
10         assertEquals(2, calculator.add(1, 1));
11     }
12 }
```

因為是 JUnit 5，在行 1-3 均匯入以 **org.junit.jupiter.api** 開頭的 API。

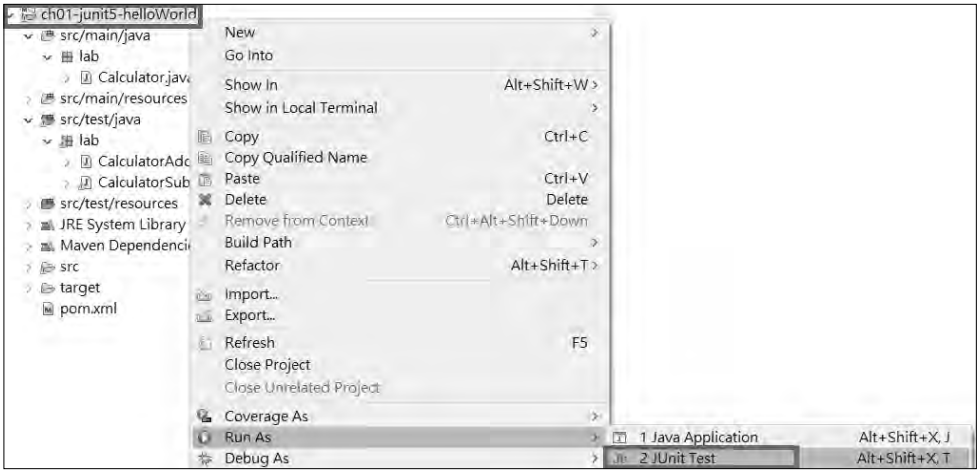
測試類別 CalculatorAddTest 的行 7 使用標註類別 **@DisplayName**，測試結果將以標註的內容「**1 + 1 = 2**」呈現。對照組 CalculatorSubtractTest 則未使用 @DisplayName：

範例：/ch01-junit5-helloWorld/src/test/java/lab/CalculatorSubtractTest.java

```
1  public class CalculatorSubtractTest {
2      @Test
```

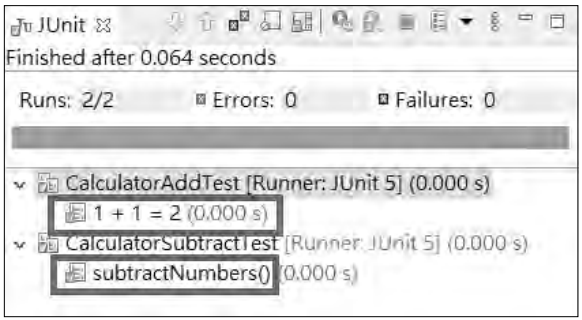
```
    public void subtractNumbers() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.subtract(3, 1));
    }
}
```

執行單元測試：



▲ 圖 1-7 點擊「Junit Test」

單元測試結果：



▲ 圖 1-8 單元測試執行結果

可以發現若單元測試的方法使用 @DisplayName 標註，結果將以標註的內容文字如「1 + 1 = 2」呈現，增加測試報告可讀性；若無則以原單元測試的方法名稱呈現。

1.3.3 在 JUnit 5 的架構下執行 JUnit 4 的測試

專案「ch01-junit5-run-junit4」的 pom.xml 設定 **junit-vintage-engine** 的依賴項目如下行 4：

範例：/ch01-junit5-run-junit4/pom.xml

```
1  <dependencies>
2      <dependency>
3          <groupId>org.junit.vintage</groupId>
4          <artifactId>junit-vintage-engine</artifactId>
5          <version>5.7.2</version>
6          <scope>test</scope>
7      </dependency>
8  </dependencies>
```

因為要驗證 JUnit 5 是否可以向前相容 JUnit 4 程式碼的執行，在以下範例行 1-2 捨棄以 org.junit.jupiter.api 開頭的 API，改匯入 **org.junit** 等開頭的測試 API：

範例：/ch01-junit5-run-junit4/src/test/java/lab/CalculatorAddTest.java

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class CalculatorAddTest {
5      @Test
6      public void addsTwoNumbers() {
7          Calculator calculator = new Calculator();
8          assertEquals(2, calculator.add(1, 1));
9      }
10 }
```

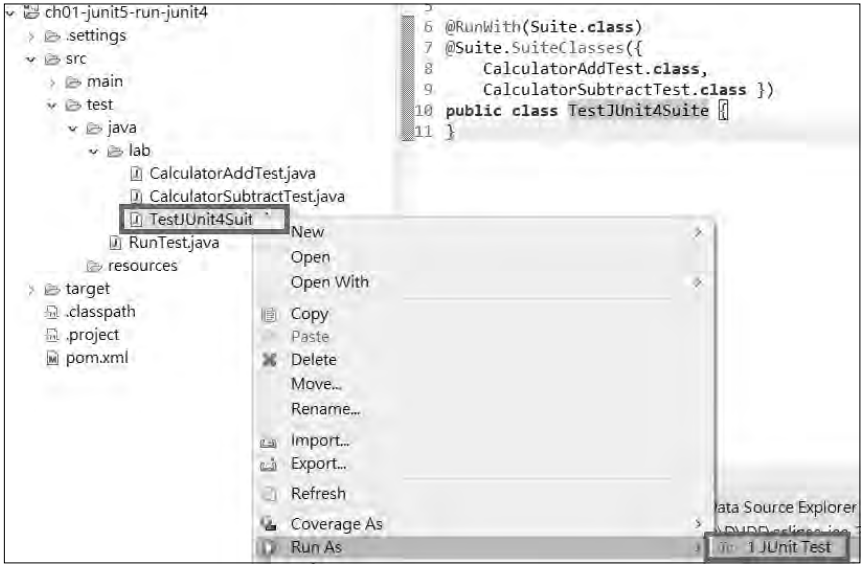
同時，我們再建立 JUnit 4 的成套 (Suite) 測試類別。關鍵做法是在類別名稱上以 **@RunWith(Suite.class)** 和 **@Suite.SuiteClasses** 標註，如行 4 與行 5；並指定該成套測試包含的類別為 CalculatorAddTest.class 與 CalculatorSubtractTest.class，如行 6 與行 7：

範例：/ch01-junit5-run-junit4/src/test/java/lab/TestJUnit4Suite.java

```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3
```

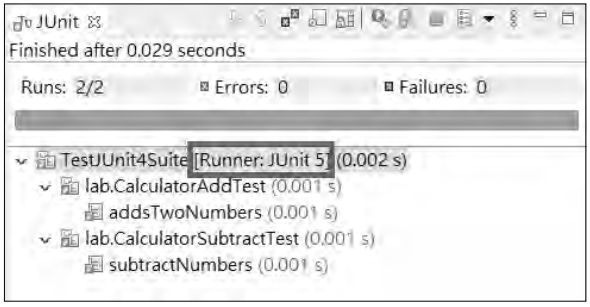
```
4  @RunWith(Suite.class)
5  @Suite.SuiteClasses({
6      CalculatorAddTest.class,
7      CalculatorSubtractTest.class })
8  public class TestJUnit4Suite {
9  }
```

對成套測試類別 TestJUnit4Suite 啟動單元測試，就可以一次啟動成套內所有測試類別的單元測試：



▲ 圖 1-9 點擊「Junit Test」

測試 TestJUnit4Suite 類別時如下圖顯示使用的 Runner 為 JUnit 5：



▲ 圖 1-10 單元測試執行結果

在本小節中，範例都是匯入 JUnit 4 的 API，但執行時期都是使用 JUnit 5 的子項目 JUnit Vintage，驗證了 JUnit 5 的向前相容包含 JUnit 4 的個別與成套單元測試。

1.3.4 以程式驅動專案內大量測試

要以 JUnit 5 執行專案內的大量單元測試，除了使用 IDE 如 Eclipse 驅動測試外，也可以使用程式驅動測試，更可以達到大量且分類執行的目的，將以專案「ch01-junit5-suite」示範。

以下使用程式驅動測試類別 CalculatorAddTest.java、CalculatorSubtractTest.java 的執行：

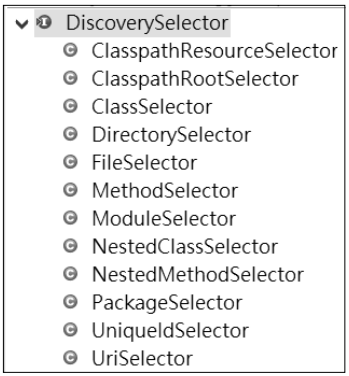
範例：/ch01-junit5-suite/src/test/java/RunTest.java

```
1  public class RunTest {
2
3      SummaryGeneratingListener listener = new SummaryGeneratingListener();
4
5      private void testSelectClass() {
6        LauncherDiscoveryRequest request =
7            LauncherDiscoveryRequestBuilder
8            .request()
9            .selectors(DiscoverySelectors.selectClass(CalculatorAddTest.class))
10           .build();
11       Launcher launcher = LauncherFactory.create();
12       launcher.registerTestExecutionListeners(listener);
13       launcher.execute(request);
14     }
15
16     private void testSelectPackage() {
17       LauncherDiscoveryRequest request =
18           LauncherDiscoveryRequestBuilder
19             .request()
20             .selectors(DiscoverySelectors.selectPackage("lab"))
21             .filters(includeClassNamePatterns(".*AddTest"))
22             .build();
23       Launcher launcher = LauncherFactory.create();
24       launcher.registerTestExecutionListeners(listener);
25       launcher.execute(request);
26     }
```

```
27
28      public static void main(String[] args) {
29          RunTest runner = new RunTest();
30          TestExecutionSummary summary = null;
31
32          System.out.println("testSelectClass(): ");
33          runner.testSelectClass();
34          summary = runner.listener.getSummary();
35          summary.printTo(new PrintWriter(System.out));
36
37          System.out.println("testSelectPackage(): ");
38          runner.testSelectPackage();
39          summary = runner.listener.getSummary();
40          summary.printTo(new PrintWriter(System.out));
41      }
42  }
```

RunTest.java 裡的 testSelectClass() 和 testSelectPackage() 方法內容相近，差別在：

1. 行 9 與行 20 的 **LauncherDiscoveryRequestBuilder** 的 **selectors()** 方法傳入的參數不同。該方法接受 **DiscoverySelector** 介面的實作，列舉如下圖，顧名思義是找出要執行的單元測試類別。



▲ 圖 1-11 DiscoverySelector 介面的實作

我們不直接建立 DiscoverySelector 介面的實作，而是由 org.junit.platform.engine.discovery.**DiscoverySelectors** 的靜態方法如 **selectClass**() 與 **selectPackage**() 提供，第一個是選擇指定的類別，第二個是選擇指定的 package 內的所有類別。使用 static import 之後可以簡化程式碼。

2. 行 21 使用的 **LauncherDiscoveryRequestBuilder** 的 **filters()** 方法可以將篩選出來的測試類別再過濾一次，傳入的參數為 org.junit.platform.engine.Filter 介面的實作，本例由 org.junit.platform.engine.discovery.ClassNameFilter 介面的靜態方法 **includeClassNamePatterns**() 提供，傳入的字串必須符合正規表示式，有過濾效果。

無論是 **selectors()** 或 **filters()** 方法都回傳 **LauncherDiscoveryRequestBuilder**，這是 builder 設計模式的應用，因此可以重複呼叫多次達到多層選項、多重過濾的效果，而且保持程式碼精簡！

兩個方法執行結果如下，篩選後都只有測試類別 CalculatorAddTest 進行測試：

結果

```
testSelectClass():

Test run finished after 100 ms
[         2 containers found      ]
[         0 containers skipped    ]
[         2 containers started     ]
[         0 containers aborted    ]
[         2 containers successful  ]
[         0 containers failed      ]
[         1 tests found          ]
[         0 tests skipped         ]
[         1 tests started          ]
[         0 tests aborted         ]
[         1 tests successful       ]
[         0 tests failed           ]

testSelectPackage():

Test run finished after 3 ms
[         2 containers found      ]
[         0 containers skipped    ]
[         2 containers started     ]
[         0 containers aborted    ]
[         2 containers successful  ]
[         0 containers failed      ]
[         1 tests found          ]
[         0 tests skipped         ]
[         1 tests started          ]
```

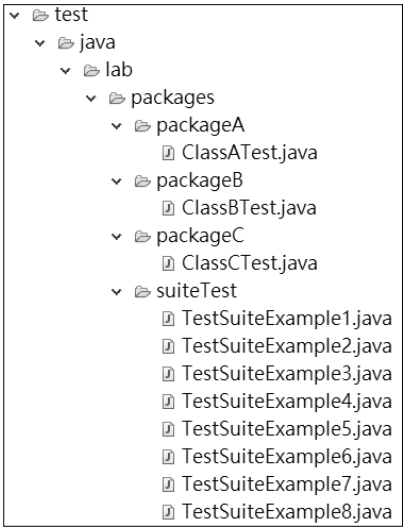
```
[         0 tests aborted         ]
[         1 tests successful       ]
[         0 tests failed           ]
```

1.3.5 以成套 (Suite) 驅動專案內大量測試

要以 JUnit 5 執行專案內的大量單元測試，除了使用 IDE、與程式驅動測試外，也可以使用成套 Suite 驅動測試。

JUnit 5 在成套測試的支援相較於 JUnit 4 擴充了許多標註類別，讓分類與過濾的功能更加完整，以下將逐一介紹常用的標註類別與其效果。

本範例套件結構設計如下圖：



▲ 圖 1-12 專案 ch01-junit5-suite 的 Suite 驅動測試結構

需要被篩選出來執行的測試類別為 packageA.ClassATest、packageB.ClassBTest、packageC.ClassCTest；類別 TestSuiteExample1~8 則是以不同的方式挑選出前述 3 個測試類別的組合。另我們在 TestSuiteExample1~8 都會加上 **@RunWith (JUnitPlatform.class)**，主要目的是讓不支援 JUnit 5 的包版工具 (build tools) 和 IDE 可以使用 JUnit 4 正常執行成套測試類別，一般測試類別如 ClassATest 等則不需要。

測試類別 packageA.ClassATest 內容如下，行 2 的標籤類別 **@Tag** 顧名思義是一個標籤 (tag)，可以作為 TestSuiteExample1~8 的挑選條件：

範例：/ch01-junit5-suite/src/test/java/lab/packages/packageA/ClassATest.java

```
1 public class ClassATest {
2     @Tag("production")
3     @Test
4     @DisplayName("packageA > ClassATest")
5     public void testCaseA() {
6     }
7 }
```

測試類別 packageB.ClassBTest 也具備 **@Tag**：

範例：/ch01-junit5-suite/src/test/java/lab/packages/packageB/ClassBTest.java

```
1 public class ClassBTest {
2     @Tag("development")
3     @Test
4     @DisplayName("packageB > ClassBTest")
5     public void testCaseB() {
6     }
7 }
```

測試類別 packageC.ClassCTest 不具備 **@Tag**：

範例：/ch01-junit5-suite/src/test/java/lab/packages/packageC/ClassCTest.java

```
1 public class ClassCTest {
2     @Test
3     @DisplayName("packageC > ClassCTest")
4     public void testCaseC() {
5     }
6 }
```

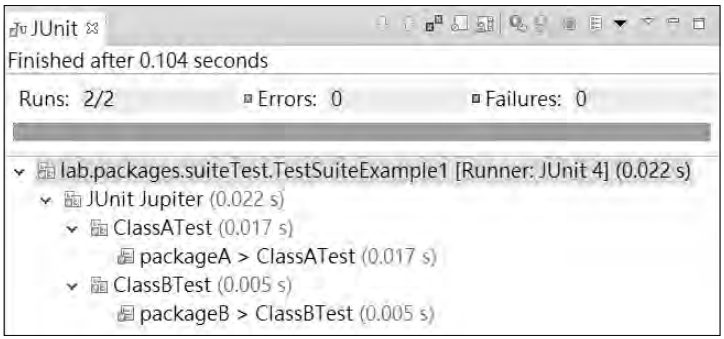
1. 使用 @SelectPackages

類別 TestSuiteExample1 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages.packageA 與 lab.packages.packageB：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample1.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages({ "lab.packages.packageA", "lab.packages.packageB" })
3 public class TestSuiteExample1 {
4 }
```

結果如預期，ClassATest 與 ClassBTest 皆被選出並執行：



▲ 圖 1-13 單元測試執行結果

讀者可以注意一下，圖片中有標示 [Runner: JUnit 4]，就是回應類別標註 **@RunWith(JUnitPlatform.class)** 的結果。

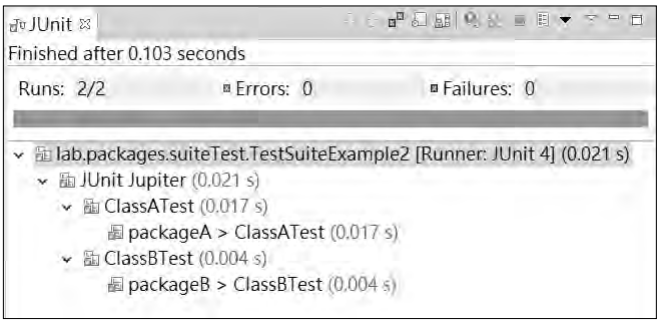
2. 使用 @SelectClasses

類別 TestSuiteExample2 範例如下。行 2 以 **@SelectClasses** 指定了類別 ClassATest 與 ClassBTest：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample2.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectClasses({ ClassATest.class, ClassBTest.class })
3 public class TestSuiteExample2 {
4 }
```

結果如預期，ClassATest 與 ClassBTest 皆被選出並執行：



▲ 圖 1-14 單元測試執行結果

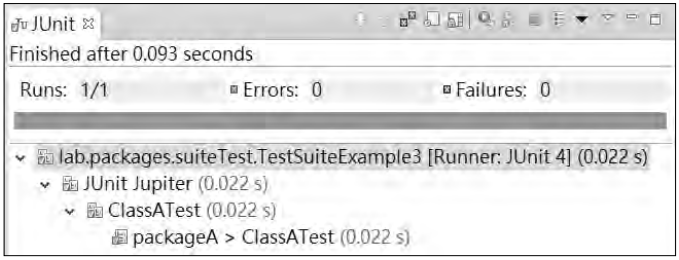
3. 使用 @SelectPackages 搭配 @IncludePackages

類別 TestSuiteExample3 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages，則所有子套件如 lab.packages.packageA、lab.packages.packageB、lab.packages.packageC 都被包含；行 3 則以 **@IncludePackages** 約束套件只有 lab.packages.packageA，通常和 @SelectPackages 搭配使用：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample3.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("lab.packages")
3 @IncludePackages("lab.packages.packageA")
4 public class TestSuiteExample3 {
5 }
```

結果如預期，只有 ClassATest 被選出並執行：



▲ 圖 1-15 單元測試執行結果

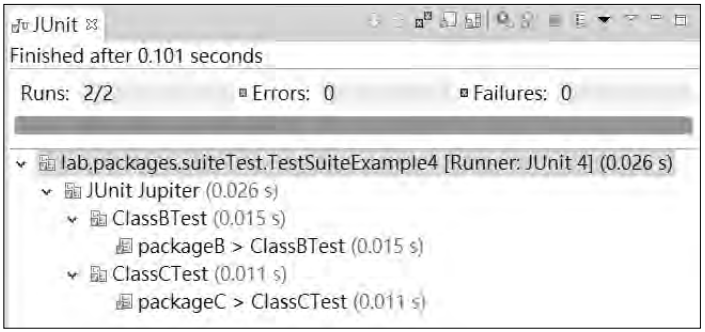
4. 使用 @SelectPackages 搭配 @ExcludePackages

類別 TestSuiteExample4 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages，則所有子套件如 lab.packages.packageA、lab.packages.packageB、lab.packages.packageC 都被包含；行 3 則以 **@ExcludePackages** 排除套件 lab.packages.packageA，通常和 @SelectPackages 搭配使用：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample4.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("lab.packages")
3 @ExcludePackages("lab.packages.packageA")
4 public class TestSuiteExample4 {
5 }
```

結果如預期，ClassBTest 與 ClassCTest 皆被選出並執行：



▲ 圖 1-16 單元測試執行結果

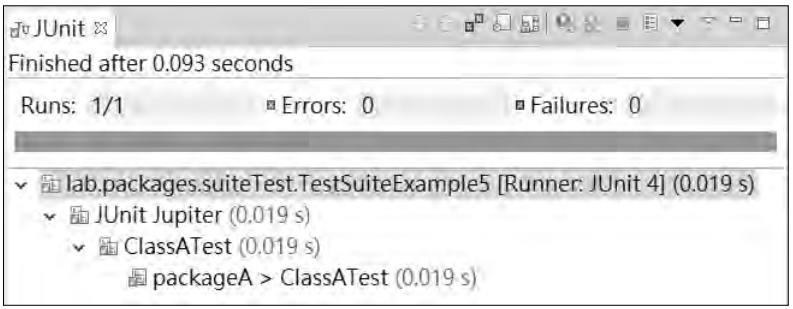
5. 使用 @SelectPackages 搭配 @IncludeClassNamePatterns

類別 TestSuiteExample5 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages，則所有子套件如 lab.packages.packageA、lab.packages.packageB、lab.packages.packageC 都被包含；行 3 則以 **@IncludeClassNamePatterns** 約束類別名稱必須有含 ATest 結尾的字串，通常和 @SelectPackages 搭配使用：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample5.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("lab.packages")
3 @IncludeClassNamePatterns({".*ATest"})
4 public class TestSuiteExample5 {
5 }
```

結果如預期，只有 ClassATest 被選出並執行：



▲ 圖 1-17 單元測試執行結果

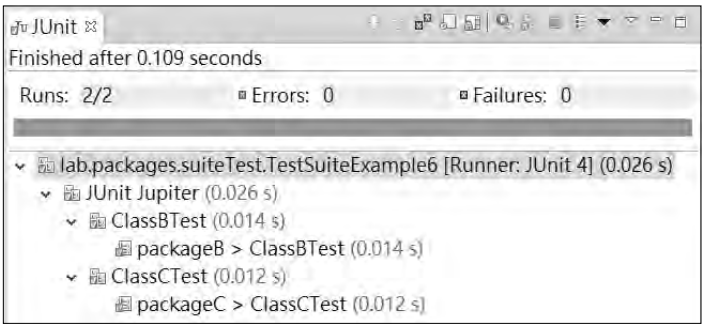
6. 使用 @SelectPackages 搭配 @ExcludeClassNamePatterns

類別 TestSuiteExample6 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages，則所有子套件如 lab.packages.packageA、lab.packages.packageB、lab.packages.packageC 都被包含；行 3 則以 **@ExcludeClassNamePatterns** 排除名稱含 ATest 結尾字串的類別，通常和 @SelectPackages 搭配使用：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample6.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("lab.packages")
3 @ExcludeClassNamePatterns({".*ATest"})
4 public class TestSuiteExample6 {
5 }
```

結果如預期，ClassBTest 與 ClassCTest 皆被選出並執行：



▲ 圖 1-18 單元測試執行結果

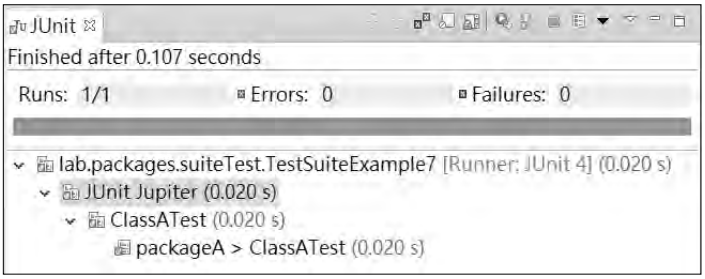
7. 使用 @SelectPackages 搭配 @IncludeTags

類別 TestSuiteExample7 範例如下。行 2 以 **@SelectPackages** 選擇了套件 lab.packages，則所有子套件如 lab.packages.packageA、lab.packages.packageB、lab.packages.packageC 都被包含；行 3 則以 **@IncludeTags("production")** 約束測試類別的方法必須以 **@Tag("production")** 標註，通常和 @SelectPackages 搭配使用：

範例：/ch01-junit5-suite/src/test/java/lab/packages/suiteTest/TestSuiteExample7.java

```
1 @RunWith(JUnitPlatform.class)
2 @SelectPackages("lab.packages")
3 @IncludeTags("production")
4 public class TestSuiteExample7 {
5 }
```

結果如預期，只有 ClassATest 被選出並執行：



▲ 圖 1-19 單元測試執行結果