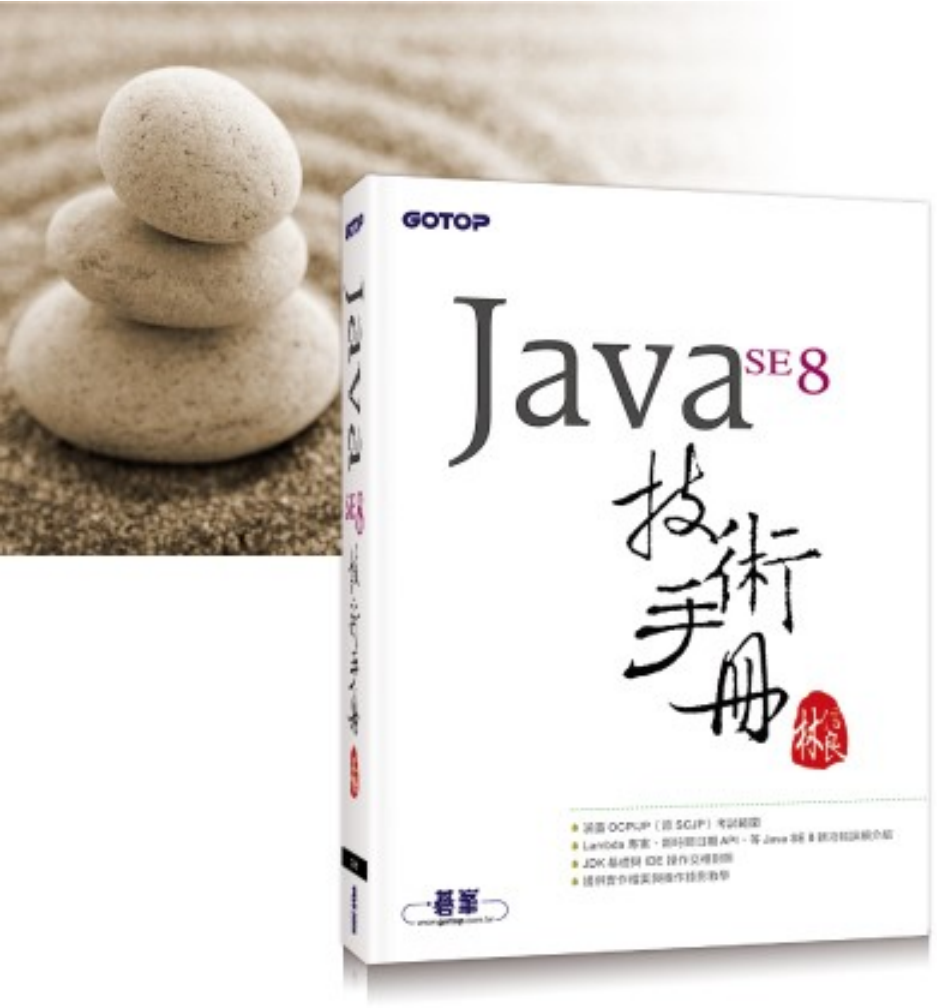


Java SE 8 技術手冊

林信良

- 涵蓋 OCP/JP（原 SCJP）考試範圍
- Lambda 專案、新時間日期 API、等 Java SE 8 新功能詳細介紹
- JDK 基礎與 IDE 操作交相對照
- 提供實作檔案與操作錄影教學

碁峰資訊

CHAPTER

執行緒與並行 API

學習目標

- 認識 `Thread` 與 `Runnable`
- 使用 `synchronized`
- 使用 `wait()`、 `notify()`、 `notifyAll()`
- 運用高階並行 API

簡介執行緒

- 如果要設計一個龜兔賽跑遊戲，賽程長度為 10 步，每經過一秒，烏龜會前進一步，兔子則可能前進兩步或睡覺，那該怎麼設計呢？

❶ 由 JVM 處理例外

```
public class TortoiseHareRace {
    public static void main(String[] args) throws InterruptedException {
        boolean[] flags = {true, false};
        int totalStep = 10;
        int tortoiseStep = 0;
        int hareStep = 0;
        System.out.println("龜兔賽跑開始...");
        while(tortoiseStep < totalStep && hareStep < totalStep) {
            Thread.sleep(1000);
            tortoiseStep++;
            System.out.printf("烏龜跑了 %d 步...%n", tortoiseStep);
            boolean isHareSleep = flags[((int) (Math.random() * 10)) % 2];
            if(isHareSleep) {
                System.out.println("兔子睡著了 zzzz");
            } else {
                hareStep += 2;
                System.out.printf("兔子跑了 %d 步...%n", hareStep);
            }
        }
    }
}
```

❷ 暫停 1000 毫秒，也就是 1 秒

❸ 烏龜走一步

❹ 隨機睡覺

❺ 兔子走兩步

簡介執行緒

- 如果可以撰寫程式再啟動兩個流程，一個是烏龜流程，一個免子流程，程式邏輯會比較清楚 …
- 如果想在 `main()` 以外獨立設計流程，可以撰寫類別實作 **`java.lang.Runnable`** 介面，流程的進入點是實作在 **`run()`** 方法中 …

```
public class Tortoise implements Runnable {
    private int totalStep;
    private int step;

    public Tortoise(int totalStep) {
        this.totalStep = totalStep;
    }

    @Override
    public void run() {
        while (step < totalStep) {
            step++;
            System.out.printf("烏龜跑了 %d 步...%n", step);
        }
    }
}
```

```
public class Hare implements Runnable {
    private boolean[] flags = {true, false};
    private int totalStep;
    private int step;

    public Hare(int totalStep) {
        this.totalStep = totalStep;
    }

    @Override
    public void run() {
        while (step < totalStep) {
            boolean isHareSleep = flags[((int) (Math.random() * 10)) % 2];
            if (isHareSleep) {
                System.out.println("兔子睡著了 zzzz");
            } else {
                step += 2;
                System.out.printf("兔子跑了 %d 步...%n", step);
            }
        }
    }
}
```

簡介執行緒

- 可以建構 **Thread** 實例來執行 **Runnable** 實例定義的 **run()** 方法

```
Tortoise tortoise = new Tortoise(10);
Hare hare = new Hare(10);
Thread tortoiseThread = new Thread(tortoise);
Thread hareThread = new Thread(hare);
tortoiseThread.start();
hareThread.start();
```

Thread 與 Runnable

- JVM 是台虛擬電腦，只安裝一顆稱為主執行緒的 CPU，可執行 `main()` 定義的執行流程
- 如果想要為 JVM 加裝 CPU，就是建構 `Thread` 實例，要啟動額外 CPU 就是呼叫 `Thread` 實例的 `start()` 方法
- 額外 CPU 執行流程的進入點，可以定義在 `Runnable` 介面的 `run()` 方法中

Thread 與 Runnable

- 除了將流程定義在 **Runnable** 的 **run()** 方法中之外，另一個撰寫多執行緒程式的方式，是繼承 **Thread** 類別，重新定義 **run()** 方法

Thread 與 Runnable

```
public class TortoiseThread extends Thread {
    ...與 Tortoise 相同，故略....

    @Override
    public void run() {
        ...與 Tortoise 相同，故略....
    }
}

public class HareThread extends Thread {
    ...與 Hare 相同，故略....

    @Override
    public void run() {
        ...與 Hare 相同，故略....
    }
}
```

```
new TortoiseThread(10).start();
new HareThread(10).start();
```

Thread 與 Runnable

- Thread 類別本身也實作了 Runnable 介面，而 run() 方法的實作如下：

```
...
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
    }
...
```

Thread 與 Runnable

- 實作 Runnable 於 run() 中定義額外流程好？還是繼承 Thread 於 run() 中定義額外流程好？
  - 實作 **Runnable** 介面的好處就是較有彈性，你的類別還有機會繼承其它類別
  - 若繼承了 **Thread**，那該類別就是一種 **Thread**，通常是為了直接利用 **Thread** 中定義的一些方法，才會繼承 **Thread** 來實作

Thread 與 Runnable

- 在 JDK8 中，由於可以使用 Lambda 表示式

```
Thread someThread = new Thread() {
    public void run() {
        // 方法實作內容…
    }
};

Thread someThread = new Thread(() -> {
    // 方法實作內容…
});
```

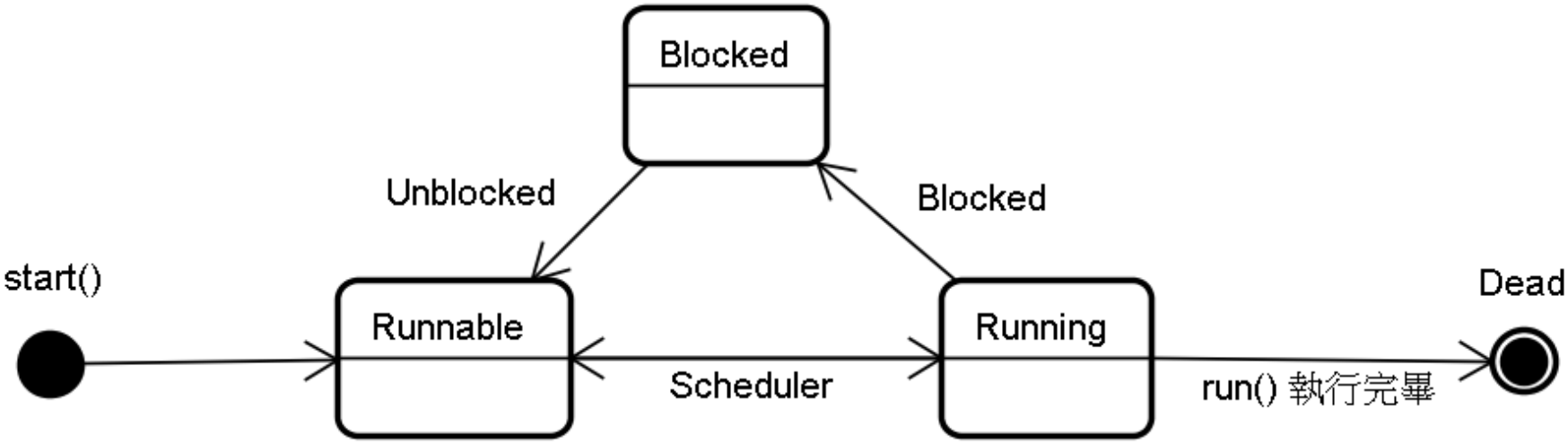
執行緒生命周期

- 如果主執行緒中啟動了額外執行緒，預設會等待被啟動的所有執行緒都執行完 `run()` 方法才中止 JVM
- 如果一個 `Thread` 被標示為 Daemon 執行緒，在所有的非 Daemon 執行緒都結束時，JVM 自動就會終止

執行緒生命周期

```
Thread thread = new Thread(() -> {
    while (true) {
        System.out.println("Orz");
    }
});
// thread.setDaemon(true);
thread.start();
```

執行緒生命周期


Blocked
Unblocked
Blocked
start()
Runnable
Running
Dead
Scheduler
run() 執行完畢

執行緒生命周期

- 執行緒有其優先權，可使用 `Thread` 的 **`setPriority()`** 方法設定優先權
  - 可設定值為 1 （**`Thread.MIN_PRIORITY`**）到 10 （**`Thread.MAX_PRIORITY`**），預設是 5 （**`Thread.NORM_PRIORITY`**）
  - 超出 1 到 10 外的設定值會拋出 `IllegalArgumentException`
- 數字越大優先權越高，排班器越優先排入 CPU，如果優先權相同，則輪流執行（Round-robin）

執行緒生命周期

- 有幾種狀況會讓執行緒進入 Blocked 狀態
  - 呼叫 `Thread.sleep()` 方法
  - 進入 `synchronized` 前競爭物件鎖定的阻斷
  - 呼叫 `wait()` 的阻斷
  - 等待輸入輸出完成
- 當某執行緒進入 Blocked 時，讓另一執行緒排入 CPU 執行（成為 Running 狀態），避免 CPU 空閒下來，經常是改進效能的方式之一

```
public static void main(String[] args) throws Exception {
    URL[] urls = {
        new URL("http://openhome.cc/Gossip/Encoding/"),
        new URL("http://openhome.cc/Gossip/Scala/"),
        new URL("http://openhome.cc/Gossip/JavaScript/"),
        new URL("http://openhome.cc/Gossip/Python/")
    };

    String[] fileNames = {
        "Encoding.html",
        "Scala.html",
        "JavaScript.html",
        "Python.html"
    };

    for(int i = 0; i < urls.length; i++) {
        dump(urls[i].openStream(), new FileOutputStream(fileNames[i]));
    }
}

static void dump(InputStream src, OutputStream dest)
                            throws IOException {
    try (InputStream input = src; OutputStream output = dest) {
        byte[] data = new byte[1024];
        int length;
        while ((length = input.read(data)) != -1) {
            output.write(data, 0, length);
        }
    }
}
```

```
URL[] urls = {
    new URL("http://openhome.cc/Gossip/Encoding/"),
    new URL("http://openhome.cc/Gossip/Scala/"),
    new URL("http://openhome.cc/Gossip/JavaScript/"),
    new URL("http://openhome.cc/Gossip/Python/")
};

String[] fileNames = {
    "Encoding.html",
    "Scala.html",
    "JavaScript.html",
    "Python.html"
};

for (int i = 0; i < urls.length; i++) {
    int index = i;
    new Thread(() -> {
        try {
            dump(urls[index].openStream(),
                new FileOutputStream(fileNames[index]));
        } catch(IOException ex) {
            throw new RuntimeException(ex);
        }
    }).start();
}
```

執行緒生命周期

- 執行緒因輸入輸出進入 Blocked 狀態後，在完成輸入輸出後，會回到 Runnable 狀態，等待排班器排入執行（Running 狀態）
- 一個進入 Blocked 狀態的執行緒，可以由另一個執行緒呼叫該執行緒的 `interrupt()` 方法，讓它離開 Blocked 狀態

執行緒生命周期

```
Thread thread = new Thread() {
    @Override
    public void run() {
        try {
            Thread.sleep(99999);
        } catch (InterruptedException ex) {
            System.out.println("我醒了 XD");
        }
    }
};
thread.start();
thread.interrupt(); // 主執行緒呼叫 thread 的 interrupt()
```

執行緒生命周期

- 如果 A 執行緒正在運行，流程中允許 B 執行緒加入，等到 B 執行緒執行完畢後再繼續 A 執行緒流程，則可以使用 **join()** 方法完成需求
- 當執行緒使用 join() 加入至另一執行緒時，另一執行緒會等待被加入的執行緒工作完畢，然後再繼續它的動作

```
out.println("Main  thread 開始...");

Thread threadB = new Thread(() -> {
    out.println("Thread  B 開始...");
    for (int i = 0; i < 5; i++) {
        out.println("Thread  B 執行...");
    }
    out.println("Thread  B 將結束...");
});

threadB.start();
threadB.join();  // Thread B 加入 Main thread 流程

out.println("Main  thread 將結束...");
```

執行緒生命周期

- 有時候加入的執行緒可能處理太久，你不想無止境等待這個執行緒工作完畢，則可以在 `join()` 時指定時間

執行緒生命周期

- 執行緒完成 `run()` 方法後，就會進入 Dead
- 進入 Dead （或已經呼叫過 `start()` 方法）的執行緒不可以再度呼叫 `start()` 方法，否則會拋出 `IllegalThreadStateException`

執行緒生命周期

- `Thread` 類別上定義有 `stop()` 方法，不過被標示為 Deprecated

stop

```
@Deprecated
public final void stop()
```

Deprecated. *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*

執行緒生命周期

- 使用了被標示為 Deprecated 的 API，編譯器會提出警告，而在 IDE 中，通常會出現刪除線表示不建議使用

```
threadB.stop();
```

- 直接呼叫 `Thread` 的 `stop()` 方法，將不理會你所設定的釋放、取得鎖定流程，執行緒會直接釋放所有已鎖定物件，有可能使物件陷入無法預期狀態
- 除了 `stop()` 方法外，`Thread` 的 `resume()`、`suspend()`、`destroy()` 等方法也不建議再使用

執行緒生命周期

- 要停止執行緒，最好自行實作，讓執行緒跑完應有的流程，而非呼叫 `Thread` 的 `stop()` 方法

```
public class Some implements Runnable {
    private boolean isContinue = true;
    ...
    public void stop() {
        isContinue = false;
    }

    public void run() {
        while(isContinue) {
            ...
        }
    }
}
```

執行緒生命周期

- 不僅有停止執行緒必須自行根據條件實作，執行緒的暫停、重啟，也必須視需求實作，而不是直接呼叫 `suspend()`、`resume()` 等方法

關於 ThreadGroup

- 每個執行緒都屬於某個執行緒群組
- 可以使用以下程式片段取得目前執行緒所屬執行緒群組名稱：

```
Thread.currentThread().getThreadGroup().getName();
```

關於 ThreadGroup

- 每個執行緒產生時，都會歸入某個執行緒群組，這視執行緒是在哪個群組中產生
- 如果沒有指定，則歸入產生該子執行緒的執行緒群組
- 可以自行指定執行緒群組，執行緒一但歸入某個群組，就無法更換群組

關於 ThreadGroup

- **java.lang.ThreadGroup** 類別可以管理群組中的執行緒，可以使用以下方式產生群組，並在產生執行緒時指定所屬群組：

```
ThreadGroup threadGroup1 = new ThreadGroup("group1");
ThreadGroup threadGroup2 = new ThreadGroup("group2");
Thread thread1 = new Thread(threadGroup1, "group1's member");
Thread thread2 = new Thread(threadGroup2, "group2's member");
```

關於 ThreadGroup

- `ThreadGroup` 的某些方法，可以對群組中所有執行緒產生作用
  - 例如 **`interrupt()`** 方法可以中斷群組中所有執行緒，**`setMaxPriority()`** 方法可以設定群組中所有執行緒最大優先權（本來就擁有更高優先權的執行緒不受影響）

關於 ThreadGroup

- 想要一次取得群組中所有執行緒，可以使用 **enumerate()** 方法

```
Thread[] threads = new Thread[threadGroup1.activeCount()];
threadGroup1.enumerate(threads);
```

關於 ThreadGroup

- ThreadGroup 中有個 **uncaughtException()** 方法，群組中某個執行緒發生例外而未捕捉時， JVM 會呼叫此方法進行處理
  - 如果 ThreadGroup 有父 ThreadGroup，就會呼叫父 ThreadGroup 的 **uncaughtException()** 方法
  - 否則看看例外是否為 ThreadDeath 實例，若是什麼都不作，若否則呼叫例外的 printStrackTrace()
- 如果必要定義 ThreadGroup 中執行緒的例外處理行為，可以重新定義此方法

關於 ThreadGroup

```
ThreadGroup group = new ThreadGroup("group") {
    @Override
    public void uncaughtException(Thread thread, Throwable throwable) {
        System.out.printf("%s: %s%n",
           thread.getName(), throwable.getMessage());
    }
};

Thread thread = new Thread(group, () -> {
    throw new RuntimeException("測試例外");
});

thread.start();
```

關於 ThreadGroup

- 在 JDK5 之後，如果 ThreadGroup 中的執行緒發生例外時 …
  - 如果 ThreadGroup 有父 ThreadGroup，就會呼叫父 ThreadGroup 的 **uncaughtException()** 方法
  - 否則，看看 Thread 是否使用 **setUncaughtExceptionHandler()** 方法設定 **Thread.UncaughtExceptionHandler** 實例，有的話就會呼叫其 uncaughtException() 方法
  - 否則，看看例外是否為 ThreadDeath 實例，若「是」什麼都不作，若「否」則呼叫例外的 printStrackTrace()

關於 ThreadGroup

- 對於執行緒本身未捕捉的例外，可以自行指定處理方式了 …

```
ThreadGroup group = new ThreadGroup("group");

Thread thread1 = new Thread(group, () -> {
    throw new RuntimeException("thread1 測試例外");
});
thread1.setUncaughtExceptionHandler((thread, throwable) -> {
    System.out.printf("%s: %s%n",
            thread.getName(), throwable.getMessage());
});

Thread thread2 = new Thread(group, () -> {
    throw new RuntimeException("thread2 測試例外");
});

thread1.start();
thread2.start();
```

synchronized 與 volatile

- 還記得 6.2.5 曾經開發過一個 ArrayList 類別嗎？

```
ArrayList list = new ArrayList();
Thread thread1 = new Thread(() -> {
    while (true) {
        list.add(1);
    }
});

Thread thread2 = new Thread(() -> {
    while (true) {
        list.add(2);
    }
});

thread1.start();
thread2.start();
```

synchronized 與 volatile

- 如果你嘗試執行程式，「有可能」會發生 `ArrayIndexOutOfBoundsException` 例外…
- 執行緒存取同一物件相同資源時所引發的競速情況（Race condition）…

```
public void add(Object o) {
    if(next == elems.length) {
        elems = Arrays.copyOf(elems, elems.length * 2);
    }
    elems[next++] = o;
}
```

synchronized 與 volatile

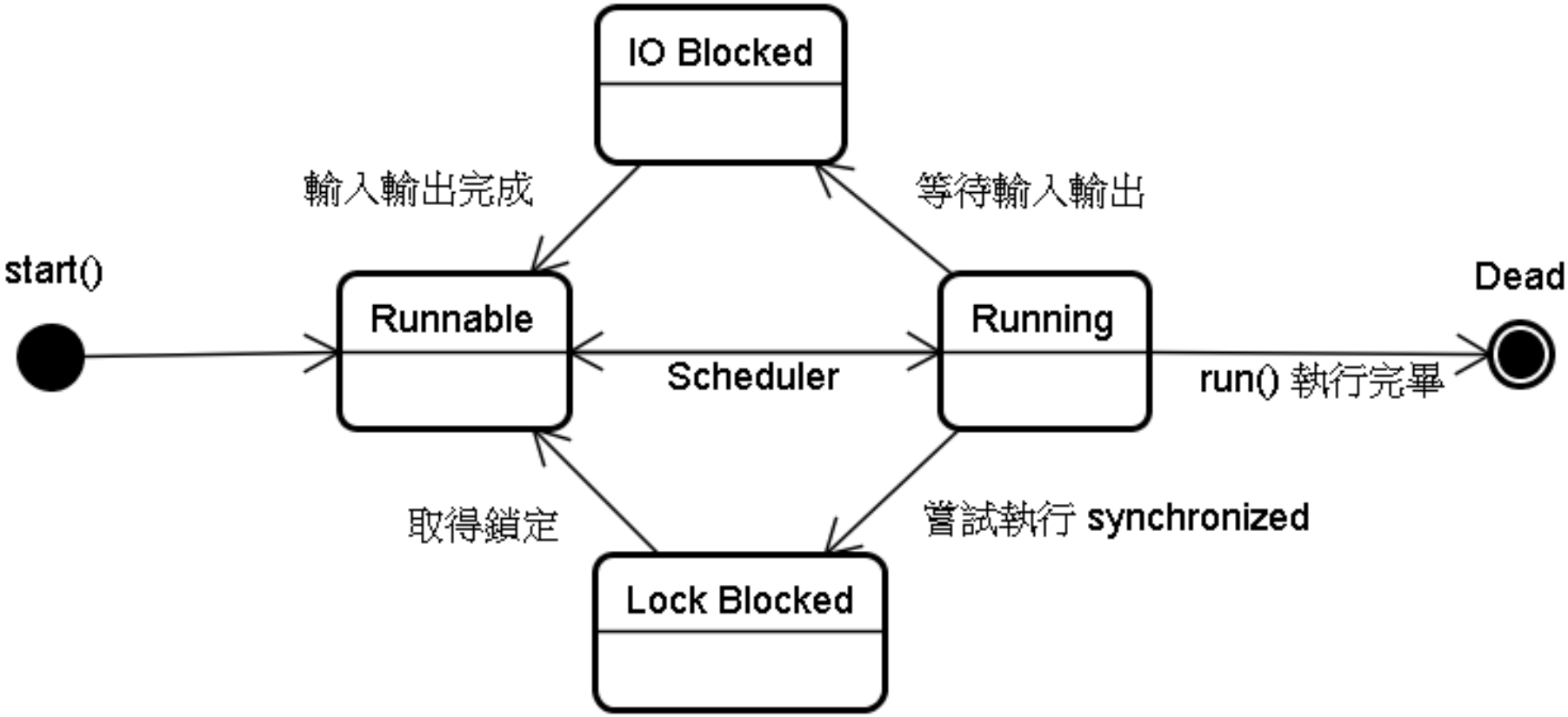
- 可以在 add() 方法上加上 **synchronized** 關鍵字

```
public synchronized void add(Object o) {
    if(next == elems.length) {
        elems = Arrays.copyOf(elems, elems.length * 2);
    }
    elems[next++] = o;
}
```

synchronized 與 volatile

- 每個物件都會有個內部鎖定（IntrinsicLock），或稱為監控鎖定（Monitor lock）
- 被標示為 synchronized 的區塊將會被監控，任何執行緒要執行 synchronized 區塊都必須先取得指定的物件鎖定

synchronized 與 volatile



synchronized 與 volatile

- **synchronized** 不只可宣告在方法上，也可以陳述句方式使用

```
...
    public void add(Object o) {
        synchronized(this) {
            if(next == list.length) {
                list = Arrays.copyOf(list, list.length * 2);
            }
            list[next++] = o;
        }
    }
...
```

synchronized 與 volatile

- 對於本身設計時沒有考慮競速問題的 API 來說，也可以如下撰寫：

```
ArrayList list = new ArrayList();
Thread thread1 = new Thread(() -> {
    while(true) {
        synchronized(list) {
            list.add(1);
        }
    }
});

Thread thread2 = new Thread(() -> {
    while(true) {
        synchronized(list) {
            list.add(2);
        }
    }
});
```

synchronized 與 volatile

- 第 9 章介紹過的 Collection 與 Map，都未考慮執行緒安全
- 可以使用 Collections 的…
  - synchronizedCollection()
  - synchronizedList()
  - synchronizedSet()
  - synchronizedMap()
- 這些方法會將傳入的 Collection、List、Set、Map 實作物件包裹，傳回具執行緒安全的物件

synchronized 與 volatile

- 如果你經常如下進行 List 操作：

```
List<String> list = new ArrayList<>();
synchronized(list) {
    ...
    list.add("...");
}
...
synchronized(list) {
    ...
    list.remove("...");
}
```

```
List<String> list = Collection.synchronizedList(new ArrayList<String>());
...
list.add("...");
...
list.remove("...");
```

synchronized 與 volatile

- 使用 synchronized 陳述句，可以作到更細部控制，例如提供不同物件作為鎖定來源…

```
public class Material {
    private int data1 = 0;
    private int data2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void doSome() {
        ...
        synchronized(lock1) {
            ...
            data1++;
            ...
        }
        ...
    }

    public void doOther() {
        ...
        synchronized(lock2) {
            ...
            data2--;
            ...
        }
        ...
    }
}
```

synchronized 與 volatile

- Java 的 synchronized 提供的是可重入同步（Reentrant Synchronization）
  - 執行緒取得某物件鎖定後，若執行過程中又要執行 `synchronized` 時，嘗試取得鎖定的物件來源又是同一個，則可以直接執行
- 由於執行緒無法取得鎖定時會造成阻斷，不正確地使用 `synchronized` 有可能造成…
  - 效能低落
  - 死結（Dead lock）

synchronized 與 volatile

```
class Resource {
    private String name;
    private int resource;

    Resource(String name, int resource) {
        this.name = name;
        this.resource = resource;
    }

    String getName() {
        return name;
    }

    synchronized int doSome() {
        return ++resource;
    }

    synchronized void cooperate(Resource resource) {
        resource.doSome();
        System.out.printf("%s 整合 %s 的資源%n",
                          this.name, resource.getName());
    }
}
```

synchronized 與 volatile

```
Resource resource1 = new Resource("resource1", 10);
Resource resource2 = new Resource("resource2", 20);

Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        resource1.cooperate(resource2);
    }
});
Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        resource2.cooperate(resource1);
    }
});

thread1.start();
thread2.start();
```

synchronized 與 volatile

- synchronized 要求達到的所標示區塊的互斥性（Mutual exclusion）與可見性（Visibility）
  - 互斥性是指 synchronized 區塊同時間只能有一個執行緒
  - 可見性是指執行緒離開 synchronized 區塊後，另一執行緒接觸到的就是上一執行緒改變後的物件狀態

synchronized 與 volatile

- 在 Java 中對於可見性的要求，可以使用 `volatile` 達到變數範圍

```
class Variable1 {
    static int i = 0, j = 0;

    static void one() {
        i++;
        j++;
    }

    static void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable1Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            while (true) {
                Variable1.one();
            }
        });
        Thread thread2 = new Thread(() -> {
            while (true) {
                Variable1.two();
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

```
略...
i = 256531535, j = 256531550
i = 256539063, j = 256539076
i = 256546975, j = 256546989
i = 256554509, j = 256554524
i = 256561967, j = 256561981
i = 256561967, j = 256561981
略...
```

```
class Variable2 {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable2Test {
    public static void main(String[] args) {
        Thread t1 = new Thread() {
            public void run() {
                while(true) {
                    Variable2.one();
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                while(true) {
                    Variable2.two();
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

```
略...
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
略...
```

synchronized 與 volatile

- 可以在變數上宣告 **volatile**，表示變數是不穩定、易變的
  - 也就是可能在多執行緒下存取，這保證變數的可見性，也就是若有執行緒變動了變數值，另一執行緒一定可看到變更
- 被標示為 volatile 的變數，不允許執行緒快取，變數值的存取一定是在共享記憶體中進行

```
class Variable3 {
    volatile static int i = 0, j = 0;

    static void one() {
        i++;
        j++;
    }

    static void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable3Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            while (true) {
                Variable3.one();
            }
        });
        Thread thread2 = new Thread(() -> {
            while (true) {
                Variable3.two();
            }
        });
        thread1.start();
        thread2.start();
    }
}
```

```
略...
i = 13235787, j = 13235787
i = 13236353, j = 13236353
i = 13236905, j = 13236906
i = 13237459, j = 13237459
i = 13238014, j = 13238013
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
略...
```

synchronized 與 volatile

- volatile 保證的是單一變數的可見性,
  - 執行緒對變數的存取一定是在共享記憶體中，不會在自己的記憶體空間中快取變數，執行緒對共享記憶體中變數的存取，另一執行緒一定看得到

synchronized 與 volatile

- 以下是個正確使用 volatile 的例子：

```
public class Some implements Runnable {
    private volatile boolean isContinue = true;
    ...
    public void stop() {
        isContinue = false;
    }

    public void run() {
        while(isContinue) {
            ...
        }
    }
}
```

等待與通知

- **wait()**、**notify()** 與 **notifyAll()** 是 Object 定義的方法
  - 可以透過這三個方法控制執行緒釋放物件的鎖定，或者通知執行緒參與鎖定競爭
- 執行 synchronized 範圍的程式碼期間，若呼叫鎖定物件的 wait() 方法，執行緒會釋放物件鎖定，並進入物件等待集合（Wait set）而處於阻斷狀態
- 其它執行緒可以競爭物件鎖定，取得鎖定的執行緒可以執行 synchronized 範圍的程式碼

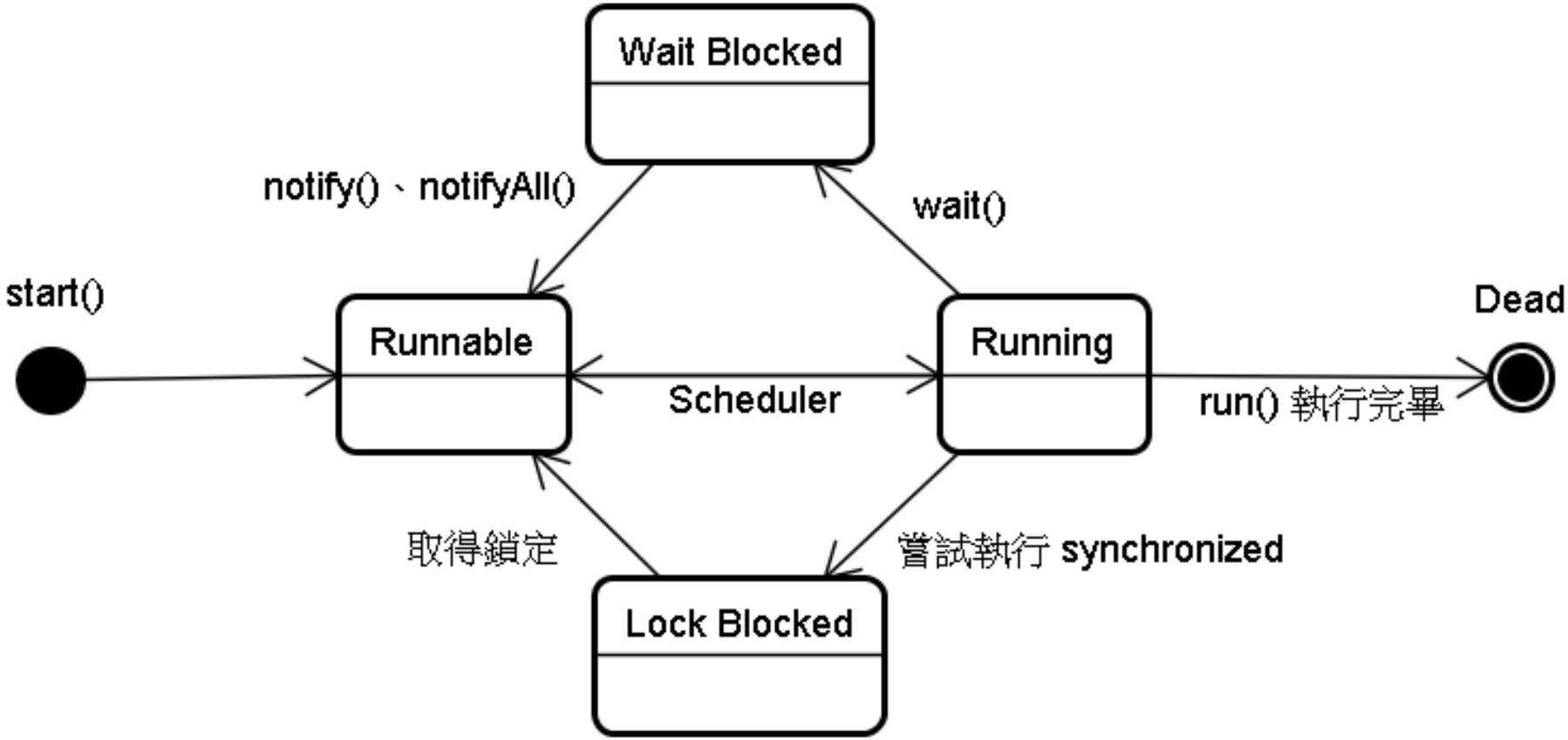
等待與通知

- 放在等待集合的執行緒不會參與 CPU 排班
- `wait()` 可以指定等待時間，時間到之後執行緒會再度加入排班
  - 如果指定時間 0 或不指定，則執行緒會持續等待，直到被中斷（呼叫 `interrupt()`）或是告知（`notify()`）可以參與排班

等待與通知

- 被競爭鎖定的物件之 `notify()` 呼叫時，會從物件等待集隨機通知一個執行緒加入排班
  - 再度執行 `synchronized` 前，被通知的執行緒會與其它執行緒共同競爭物件鎖定
- 如果呼叫 `notifyAll()`，所有等待集中的執行緒都會被通知參與排班，這些執行緒會與其它執行緒共同競爭物件鎖定

等待與通知


Wait Blocked
notify()、notifyAll()
wait()
start()
Runnable
Running
Dead
Scheduler
run() 執行完畢
取得鎖定
嘗試執行 synchronized
Lock Blocked

等待與通知

- 生產者與消費者 …

```
public class Producer implements Runnable {
    private Clerk clerk;

    public Producer(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("生產者開始生產整數.....");
        for(int product = 1; product <= 10; product++) {
            try {
                clerk.setProduct(product);
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

❶ 產生 1 到 10 的整數

❷ 將產品交給店員

等待與通知

```
public class Consumer implements Runnable {
    private Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("消費者開始消耗整數.....");
        for(int i = 1; i <= 10; i++) {
            try {
                clerk.getProduct();
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

❶ 消費 10 次整數

❷ 從店員取走產品

等待與通知

```
public class Clerk {
    private int product = -1;  ←— ❶ 只持有一個產品，-1 表示沒有產品

    public synchronized void setProduct(int product)
                                    throws InterruptedException {
        waitIfFull();          ←— ❷ 看看店員有沒有空間收產品，沒有的話就稍候
        this.product = product; ←— ❸ 店員收貨
        System.out.printf("生產者設定 (%d)%n", this.product);
        notify();  ←— ❹ 通知等待集合中的執行緒（例如消費者）
    }

    private synchronized void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            wait();
        }
    }
```

等待與通知

```
    public synchronized int getProduct() throws InterruptedException {
        waitIfEmpty();      ←——  ❺ 看看目前店員有沒有貨，沒有的話就稍候
        int p = this.product; ←—  ❻ 準備交貨
        this.product = -1;   ←—  ❼ 表示貨品被取走
        System.out.printf("消費者取走 (%d)%n", p);
        notify();  ←—  ❽ 通知等待集合中的執行緒（例如生產者）
        return p;  ←—  ❾ 交貨了
    }

    private synchronized void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            wait();
        }
    }
}
```

等待與通知

- 使用以下的程式來示範 Producer、Consumer與Clerk：

```
public class ProducerConsumerDemo {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        new Thread(new Producer(clerk)).start();
        new Thread(new Consumer(clerk)).start();
    }
}
```

Lock、ReadWriteLock 與 Condition

- java.util.concurrent.locks 套件中提供 Lock、ReadWriteLock、Condition 介面以及相關實作類別
  - 可以提供類似 synchronized、wait()、notify()、notifyAll() 的作用，以及更多高階功能

Lock、ReadWriteLock與Condition

```
public class ArrayList<E> {
    private Lock lock = new ReentrantLock();  ← ❶ 使用 ReentrantLock
    private Object[] elems;
    private int next;

    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList() {
        this(16);
    }

    public void add(E elem) {
        lock.lock();  ← ❷ 進行鎖定
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.unlock();  ← ❸ 解除鎖定
        }
    }
```

Lock、ReadWriteLock與Condition

```
    public E get(int index) {
        lock.lock();
        try {
            return (E) elems[index];
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        lock.lock();
        try {
            return next;
        } finally {
            lock.unlock();
        }
    }
}
```

Lock、ReadWriteLock 與 Condition

- Lock 介面還定義了 **tryLock()** 方法，如果執行緒呼叫 tryLock() 可以取得鎖定會傳回 true，若無法取得鎖定傳回 false

```
class Resource {
    private ReentrantLock lock = new ReentrantLock();
    private String name;

    Resource(String name) {
        this.name = name;
    }

    void cooperate(Resource res) {
        while (true) {
            try {
                if (lockMeAnd(res)) {
                    System.out.printf("%s 整合 %s 的資源%n", this.name, res.name);
                    break;
                }
            } finally {
                unLockMeAnd(res);
            }
        }
    }
```

❶ 取得目前與傳入的 Resource 之 Lock 鎖定

❷ 如果兩個 Resource 的 Lock 都取得鎖定，才執行資源整合

❸ 資源整合成功，離開迴圈

❹ 解除目前與傳入的 Resource 之 Lock 鎖定

```
    private boolean lockMeAnd(Resource res) {
        return this.lock.tryLock() && res.lock.tryLock();
    }

    private void unLockMeAnd(Resource res) {
        if (this.lock.isHeldByCurrentThread()) {
            this.lock.unlock();
        }
        if (res.lock.isHeldByCurrentThread()) {
            res.lock.unlock();
        }
    }
}
```

Lock、ReadWriteLock與Condition

```
Resource res1 = new Resource("resource1");
Resource res2 = new Resource("resource2");

Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        res1.cooperate(res2);
    }
});
Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        res2.cooperate(res1);
    }
});

thread1.start();
thread2.start();
```

Lock、ReadWriteLock 與 Condition

- 前面設計了執行緒安全的 `ArrayList`，無法兩個執行緒同時呼叫 `get()` 與 `size()`
- 如果只是讀取操作，可允許執行緒同時並行的話，那對讀取效率將會有所改善

Lock、ReadWriteLock 與 Condition

- **ReadWriteLock** 介面定義了讀取鎖定與寫入鎖定行為，可以使用 **readLock()**、**writeLock()** 方法傳回 Lock 實作物件
- **ReentrantReadWriteLock** 是 ReadWriteLock 介面的主要實作類別
  - readLock() 方法會傳回 **ReentrantReadWriteLock.ReadLock** 實例
  - writeLock() 方法會傳回 **ReentrantReadWriteLock.WriteLock** 實例

Lock、ReadWriteLock 與 Condition

- ReentrantReadWriteLock.ReadLock 實作了 Lock 介面，呼叫其 lock() 方法時…
  - 若沒有任何 ReentrantReadWriteLock.WriteLock 呼叫過 lock() 方法，也就是沒有任何寫入鎖定時，就可以取得讀取鎖定
- ReentrantReadWriteLock.WriteLock 實作了 Lock 介面，呼叫其 lock() 方法時…
  - 若沒有任何 ReentrantReadWriteLock.ReadLock 或 ReentrantReadWriteLock.WriteLock 呼叫過 lock() 方法，也就是沒有任何讀取或寫入鎖定時，才可以取得寫入鎖定。

```
public class ArrayList2<E> {
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    private Object[] elems;
    private int next;

    public ArrayList2(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList2() {
        this(16);
    }

    public void add(E elem) {
        lock.writeLock().lock();
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.writeLock().unlock();
        }
    }
```

❶ 使用 ReadWriteLock

❷ 取得寫入鎖定

❸ 解除寫入鎖定

```
    public E get(int index) {
        lock.readLock().lock();
        try {
            return (E) elems[index];
        } finally {
            lock.readLock().unlock();
        }
    }

    public int size() {
        lock.readLock().lock();  ←—— ❹ 取得讀取鎖定
        try {
            return next;
        } finally {
            lock.readLock().unlock();  ←—— ❺ 解除讀取鎖定
        }
    }
}
```

使用 StampedLock

- ReadWriteLock 在沒有任何讀取或寫入鎖定時，才可以取得寫入鎖定，這可用於實現悲觀讀取（Pessimistic Reading）
- 如果執行緒進行讀取時，經常可能有另一執行緒有寫入需求，為了維持資料一致，ReadWriteLock 的讀取鎖定就可派上用場

使用 StampedLock

- 如果讀取執行緒很多，寫入執行緒甚少的情況下，使用 ReadWriteLock 可能會使得寫入執行緒遭受**飢餓（Starvation）問題**
- 寫入執行緒可能遲遲無法競爭到鎖定，而一直處於等待狀態

使用 StampedLock

- JDK8 新增了 StampedLock 類別，可支援樂觀讀取（**Optimistic Reading**）實作
- 若讀取執行緒很多，寫入執行緒甚少的情況下，你可以樂觀地認為，寫入與讀取同時發生的機會甚少，因此不悲觀地使用完全的讀取鎖定

```
public class ArrayList3<E> {
    private StampedLock lock = new StampedLock();  ← ❶ 使用 StampedLock
    private Object[] elems;
    private int next;

    public ArrayList3(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList3() {
        this(16);
    }

    public void add(E elem) {
        long stamp = lock.writeLock();  ← ❷ 取得寫入鎖定
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.unlockWrite(stamp);  ← ❸ 解除寫入鎖定
        }
    }
```

```
    public E get(int index) {
        long stamp = lock.tryOptimisticRead();   ← ❹ 試著樂觀讀取鎖定
        Object elem = elems[index];
        if (!lock.validate(stamp)) {   ← ❺ 查詢是否有排他的鎖定
            stamp = lock.readLock();   ← ❻ 真正的讀取鎖定
            try {
                elem = elems[index];
            } finally {
                lock.unlockRead(stamp);   ← ❼解除讀取鎖定
            }
        }
        return (E) elem;
    }

    public int size() {
        long stamp = lock.tryOptimisticRead();
        int size = next;
        if (!lock.validate(stamp)) {
            stamp = lock.readLock();
            try {
                size = next;
            } finally {
                lock.unlockRead(stamp);
            }
        }
        return size;
    }
}
```

Lock、ReadWriteLock 與 Condition

- **Condition** 介面用來搭配 Lock，最基本用法就是達到 Object 的 wait()、notify()、notifyAll() 方法之作用…

```
public class Clerk {
    private int product = -1;
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void setProduct(int product) throws InterruptedException {
        lock.lock();
        try {
            waitIfFull();
            this.product = product;
            System.out.printf("生產者設定 (%d)%n", this.product);
            condition.signal();
        } finally {
            lock.unlock();
        }
    }

    private void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            condition.await();
        }
    }
```

❶ 建立 Condition 物件

❷ 用 Condition 的 signal() 取代 Object 的 notify()

❸ 用 Condition 的 await() 取代 Object 的 wait()

```
    public int getProduct() throws InterruptedException {
        lock.lock();
        try {
            waitIfEmpty();
            int p = this.product;
            this.product = -1;
            System.out.printf("消費者取走 (%d)%n", p);
            condition.signal();
            return p;
        } finally {
            lock.unlock();
        }
    }

    private void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            condition.await();
        }
    }
}
```

Lock、ReadWriteLock 與 Condition

- 一個 Condition 物件可代表有一個等待集合，可以重複呼叫 Lock 的 newCondition()，取得多個 Condition 實例，這代表了可以有多個等待集合…
- 如果可以有兩個等待集合，一個是給生產者執行緒用，一個給消費者執行緒用，生產者只通知消費者等待集合，消費者只通知生產者等待集合，那會比較有效率 …

```
public class Clerk2 {
    private int product = -1;
    private Lock lock = new ReentrantLock();
    private Condition producerCond = lock.newCondition();
    private Condition consumerCond = lock.newCondition();

    public void setProduct(int product) throws InterruptedException {
        lock.lock();
        try {
            waitIfFull();
            this.product = product;
            System.out.printf("生產者設定 (%d)%n", this.product);
            consumerCond.signal();
        } finally {
            lock.unlock();
        }
    }

    private void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            producerCond.await();
        }
    }
```

❶ 擁有生產者等待集合

❷ 擁有消費者等待集合

❸ 通知消費者等待集合中的消費者執行緒

❹至生產者等待集合等待

```
    public int getProduct() throws InterruptedException {
        lock.lock();
        try {
            waitIfEmpty();
            int p = this.product;
            this.product = -1;
            System.out.printf("消費者取走 (%d)%n", p);
            producerCond.signal();
            return p;
        } finally {
            lock.unlock();
        }
    }

    private void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            consumerCond.await();
        }
    }
}
```

❺ 通知生產者等待集合中的生產執行緒

❻至消費者等待集合等待

使用 Executor

- 從 JDK5 開始，定義了 **java.util.concurrent.Executor** 介面，目的在將 Runnable 的指定與實際如何執行分離

```
package java.util.concurrent;
public interface Executor {
    void execute(Runnable command);
}
```

```
public class Pages {
    private URL[] urls;
    private String[] fileNames;
    private Executor executor;

    public Pages(URL[] urls, String[] fileNames, Executor executor) {
        this.urls = urls;
        this.fileNames = fileNames;
        this.executor = executor;
    }

    public void download() {
        for (int i = 0; i < urls.length; i++) {
            URL url = urls[i];
            String fileName = fileNames[i];
            executor.execute(() -> {
                try {
                    dump(url.openStream(), new FileOutputStream(fileName));
                } catch (IOException ex) {
                    throw new RuntimeException(ex);
                }
            });
        }
    }
```

```
    private void dump(InputStream src, OutputStream dest) throws IOException {
        try (InputStream input = src; OutputStream output = dest) {
            byte[] data = new byte[1024];
            int length;
            while ((length = input.read(data)) != -1) {
                output.write(data, 0, length);
            }
        }
    }
}
```

使用 Executor

- 定義一個 DirectExecutor，單純呼叫傳入 execute() 方法的 Runnable 物件之 run() 方法：

```
public class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

```
new Pages(urls, fileNames, new DirectExecutor()).download();
```

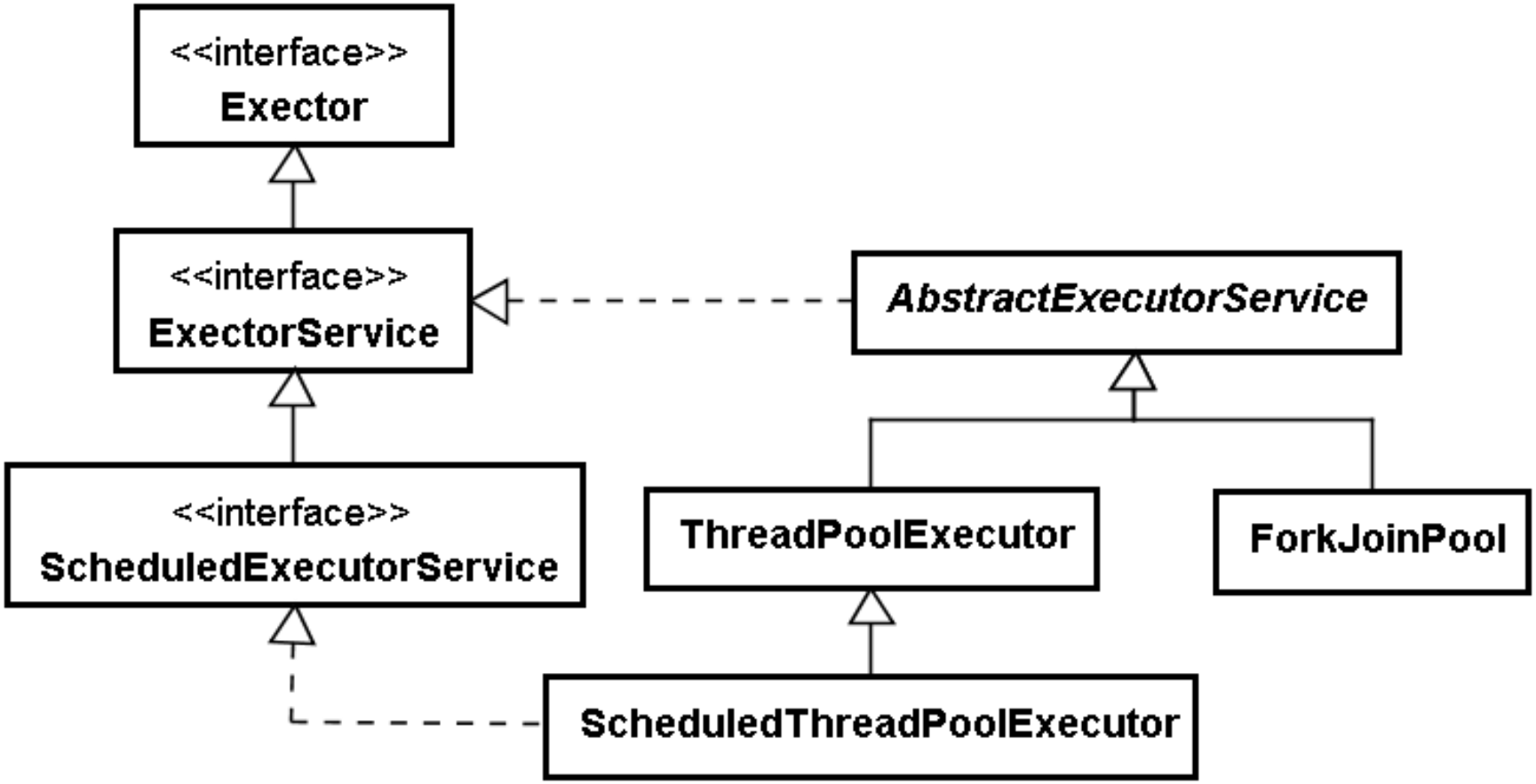
使用 Executor

- 如果定義 ThreadPerTaskExecutor：

```
public class ThreadPerTaskExecutor implements Executor {
     public void execute(Runnable r) {
         new Thread(r).start();
     }
}
```

```
new Pages(urls, fileNames, new ThreadPerTaskExecutor()).download();
```

使用 Executor


<<interface>>
Exector
<<interface>>
ExectorService
AbstractExecutorService
<<interface>>
ScheduledExecutorService
ThreadPoolExecutor
ForkJoinPool
ScheduledThreadPoolExecutor

使用 Executor

- 執行緒池這類服務定義在 Executor 子介面 **java.util.concurrent.ExecutorService**
- 通用的 ExecutorService 由抽象類別 AbstractExecutorService 實作
  - 如果需要執行緒池功能，可以使用子類別 **java.util.concurrent.ThreadPoolExecutor**

使用 Executor

- ThreadPoolExecutor 擁有不同建構式可供使用
- **java.util.concurrent.Executors** 的 **newCachedThreadPool()**、newFixedThreadPool() 靜態方法可建構 **ThreadPoolExecutor** 實例

使用 Executor

- Executors.newCachedThreadPool() 傳回的 ThreadPoolExecutor 實例，會在必要時建立執行緒
  - 你的 Runnable 可能執行在新建的執行緒，或重複利用的執行緒中
- newFixedThreadPool() 可指定在池中建立固定數量的執行緒
- 這兩個方法也都有接受 **java.util.concurrent.ThreadFactory** 的版本，你可以在 ThreadFactory 的 **newThread()** 方法中，實作如何建立 Thread 實例

使用 Executor

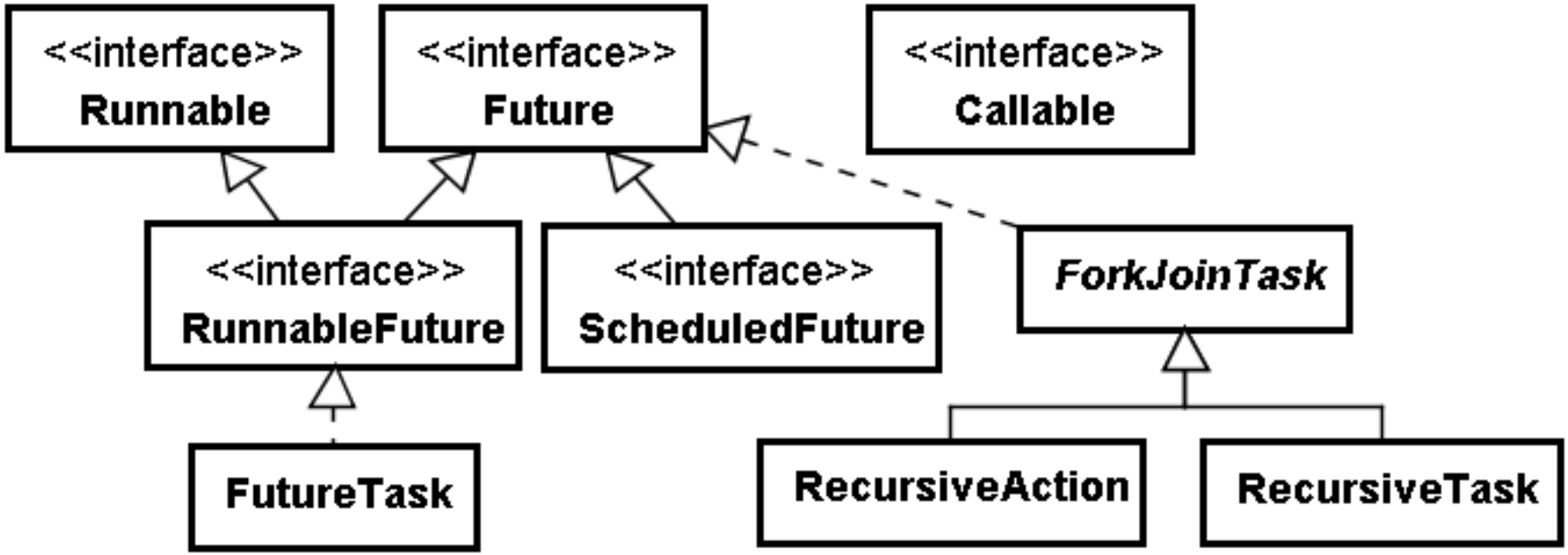
- 可使用 ThreadPoolExecutor 搭配先前的 Pages 使用：

```
ExecutorService executorService = Executors.newCachedThreadPool();
new Pages(urls, fileNames, executorService).download();
executorService.shutdown();
```

使用 Executor

- ExecutorService 還定義了 **submit()**、**invokeAll()**、 **invokeAny()** 等方法
  - 方法中出現了 **java.util.concurrent.Future**、 **java.util.concurrent.Callable** 介面

使用 Executor


<<interface>>
Runnable
<<interface>>
Future
<<interface>>
Callable
<<interface>>
RunnableFuture
<<interface>>
ScheduledFuture
ForkJoinTask
FutureTask
RecursiveAction
RecursiveTask

使用 Executor

- **Future** 定義的行為讓你在將來取得結果
  - 可以將想執行的工作交給 `Future`， `Future` 使用另一執行緒來進行工作，你就可以先忙別的事去
  - 過些時候，再呼叫 `Future` 的 **`get()`** 取得結果，如果結果已經產生， `get()` 會直接返回，否則會進入阻斷直到結果傳回
  - `get()` 的另一版本則可以指定等待結果的時間，若指定的時間到結果還沒產生，就會丟出 **`java.util.concurrent.TimeoutException`**
  - 以使用 `Future` 的 **`isDone()`** 方法，看看結果是否產生

使用 Executor

- Future 經常與 Callable 搭配使用
- Callable 的作用與 Runnable 類似，可讓你定義想要執行的流程
- Runnable 的 run() 方法無法傳回值，也無法拋出受檢例外，然而 Callable 的 call() 方法可以傳回值，也可以拋出受檢例外

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

使用 Executor

- **java.util.concurrent.FutureTask** 是 Future 的實作類別，建構時可傳入 Callable 實作物件指定的執行的內容

```
FutureTask<Long> the30thFibFuture =
        new FutureTask<>(() -> fibonacci(30));

out.println("老闆，我要第 30 個費式數，待會來拿...");

new Thread(the30thFibFuture).start();
while(!the30thFibFuture.isDone()) {
    out.println("忙別的事去...");
}

out.printf("第 30 個費式數：%d%n", the30thFibFuture.get());
```

使用 Executor

- ExecutorService 的 submit() 方法，它可以接受 Callable 物件，呼叫後傳回的 Future 物件，就是讓你在稍後可以取得運算結果

```
ExecutorService service = Executors.newCachedThreadPool();

out.println("老闆，我要第  30 個費式數，待會來拿...");

Future<Long> the30thFibFuture = service.submit(() -> fibonacci(30));
while(!the30thFibFuture.isDone()) {
    out.println("忙別的事去...");
}

out.printf("第  30 個費式數：%d%n",  the30thFibFuture.get());
```

使用 Executor

- ScheduledExecutorService 為 ExecutorService 的子介面，可以讓你進行工作排程
  - **schedule()** 方法用來排定 Runnable 或 Callable 實例延遲多久後執行一次，並傳回 Future 子介面 **ScheduledFuture** 的實例
  - 對於重複性的執行，可使用 **scheduleWithFixedDelay()** 與 **scheduleAtFixedRate()** 方法

使用 Executor

- 在一個執行緒只排定一個 Runnable 實例的情況下
  - scheduleWithFixedDelay() 方法可排定延遲多久首次執行 Runnable，執行完 Runnable 會排定延遲多久後再度執行
  - scheduleAtFixedRate() 可指定延遲多久首次執行 Runnable，同時依指定週期排定每次執行 Runnable 的時間
  - 不管是 scheduleWithFixedDelay() 與 scheduleAtFixedRate()，上次排定的工作拋出例外，不會影響下次排程的進行

使用 Executor

- ScheduledExecutorService 的實作類別 **ScheduledTreadPoolExecutor** 為 ThreadPoolExecutor 的子類別，具有執行緒池與排程功能。
  - 可以使用 Executors 的 **newScheduledThreadPool()** 方法指定傳回內建多少個執行緒的 ScheduledTreadPoolExecutor
  - 使用 **newSingleThreadScheduledExecutor()** 則可使用單一執行緒執行排定的工作

使用 Executor

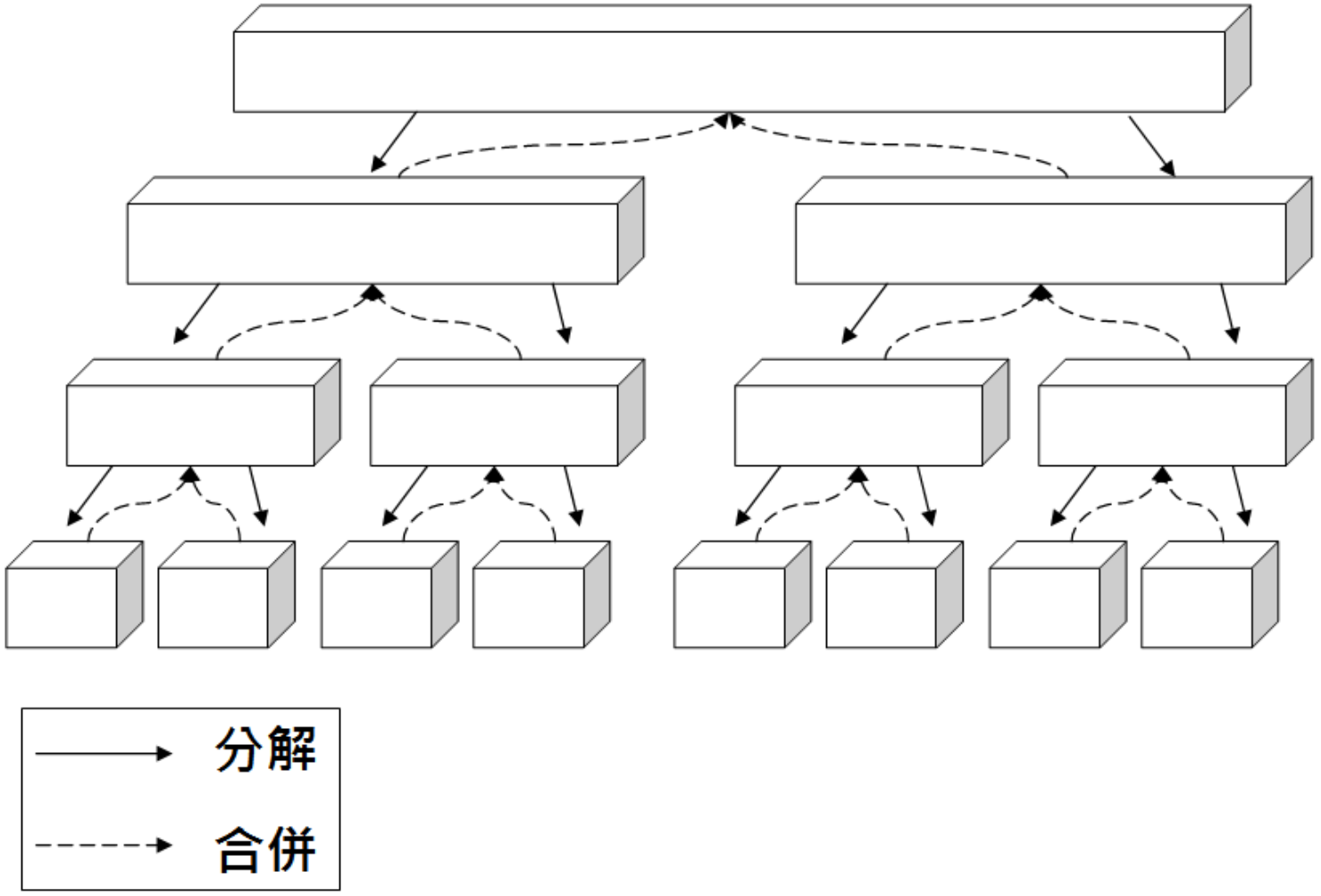
```
ScheduledExecutorService service
        = Executors.newSingleThreadScheduledExecutor();

service.scheduleWithFixedDelay(
        () -> {
            System.out.println(new java.util.Date());
            try {
                Thread.sleep(2000);  // 假設這個工作會進行兩秒
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }, 2000, 1000, TimeUnit.MILLISECONDS);
```

使用 Executor

- Future 的另一實作類別 **java.util.concurrent.ForkJoinTask** 及其子類別，與 ExecutorService 的實作類別 **java.util.concurrent.ForkJoinPool** 有關
  - 都是 JDK7 中新增的 API，主要目的是在解決分而治之（Divide and conquer）的問題

使用 ForkJoinPool



使用 ForkJoinPool

- ForkJoinTask 子類別 RecursiveTask，用於執行後會傳回結果的時候
- 使用時必須繼承它，並實作 compute() 方法

```
class Fibonacci extends RecursiveTask<Long> {
    final long n;
    Fibonacci(long n) {
        this.n = n;
    }

    @Override
    public Long compute() {
        if (n <= 20) {
            return solveFibonacciSequentially(n);
        }
        ForkJoinTask<Long> subTask = new Fibonacci(n - 1).fork();
        return new Fibonacci(n - 2).compute() + subTask.join();
    }

    static long solveFibonacciSequentially(long n) {
        if (n <= 1) {
            return n;
        }
        return solveFibonacciSequentially(n - 1)
                + solveFibonacciSequentially(n - 2);
    }
}

public class FibonacciForkJoin {
    public static void main(String[] args) {
        Fibonacci fibonacci = new Fibonacci(45);
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(fibonacci));
    }
}
```

❶ 繼承 RecursiveTask

❷ 實作 compute() 方法

❸ 20 以下就不分解了，直接循序運算

❹ 分解出 n-1 子任務、請 ForkJoinPool 分配執行緒來執行這個子任務

❺ 分解出 n-2 子任務並直接運算

❻ 取得此子任務執行結果

❼ 開始分而治之

簡介並行 Collection

- 使用第 9 章介紹過的 `List` 實作，由於它們都非執行緒安全類別，為了要在使用迭代器時，不受另一執行緒寫入操作的影響，你必須作類似以下的動作：

```
List list = new ArrayList();
...
synchornized(list) {
    Iterator iterator = list.iterator();
    while(iterator.hasNext()) {
        ...
    }
}
```

簡介並行 Collection

- 使用 Collections.synchronizedList() 並非就不用加此操作：

```
List list = Collections.synchronizedList(new ArrayList());
...
synchornized(list) {
    Iterator iterator = list.iterator();
    while(iterator.hasNext()) {
        ...
    }
}
```

簡介並行 Collection

- **CopyOnWriteArrayList** 實作了 List 介面
  - 在寫入操作時（例如 add()、 set() 等），內部會建立新陣列，並複製原有陣列索引的參考，然後在新陣列上進行寫入操作，寫入完成後，再將內部原參考舊陣列的變數參考至新陣列
  - 對寫入而言，這是個很耗資源的設計
  - 使用迭代器時，寫入不影響迭代器已參考的物件
  - 對於一個鮮少進行寫入操作，而使用迭代器頻繁的情境下，可使用 CopyOnWriteArrayList 提高迭代器效率

簡介並行 Collection

- **CopyOnWriteArraySet** 實作了 Set 介面
  - 內部使用 CopyOnWriteArrayList 來完成 Set 的各種操作，因此一些特性與 CopyOnWriteArrayList 是相同的

簡介並行 Collection

- **BlockingQueue** 是 Queue 的子介面
  - 執行緒若呼叫 put() 方法，在佇列已滿的情況下會被阻斷
  - 執行緒若呼叫 take() 方法，在佇列為空的情況下會被阻斷

```
public class Producer3 implements Runnable {
    private BlockingQueue<Integer> productQueue;

    public Producer3(BlockingQueue<Integer> productQueue) {
        this.productQueue = productQueue;
    }

    public void run() {
        System.out.println("生產者開始生產整數.....");
        for(int product = 1; product <= 10; product++) {
            try {
                productQueue.put(product);
                System.out.printf("生產者提供整數  (%d)%n", product);
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

```
public class Consumer3 implements Runnable {
    private BlockingQueue<Integer> productQueue;

    public Consumer3(BlockingQueue<Integer> productQueue) {
        this.productQueue = productQueue;
    }

    public void run() {
        System.out.println("消費者開始消耗整數.....");
        for(int i = 1; i <= 10; i++) {
            try {
                int product = productQueue.take();
                System.out.printf("消費者消費整數 (%d)%n", product);
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

簡介並行 Collection

```
BlockingQueue queue = new ArrayBlockingQueue(1); // 容量為 1
new Thread(new Producer3(queue)).start();
new Thread(new Consumer3(queue)).start();
```

簡介並行 Collection

- **ConcurrentMap** 是 Map 的子介面，其定義了 **putIfAbsent()**、 **remove()** 與 **replace()** 等方法，這些方法都是原子（Atomic）操作

簡介並行 Collection

- putIfAbsent() 相當於自行於 synchronized 中進行以下動作：

```
if (!map.containsKey(key)) {
    return map.put(key, value);
} else {
    return map.get(key);
}
```

簡介並行 Collection

- remove() 相當於自行於 synchronized 中進行以下動作：

```
if (map.containsKey(key) && map.get(key).equals(value)) {
    map.remove(key);
    return true;
} else return false;
```

簡介並行 Collection

- replace() 有兩個版本，其一相當於自行於 synchronized 中進行以下動作：

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
   map.put(key, newValue);
   return true;
} else return false;
```

簡介並行 Collection

- replace() 另外一個版本相當於你自行於 synchronized 中進行以下動作：

```
if (map.containsKey(key)) {
    return map.put(key, value);
} else return null;
```

簡介並行 Collection

- **ConcurrentHashMap** 是 ConcurrentMap 的實作類別
- ConcurrentNavigableMap 是 ConcurrentMap 子介面，其實作類別為 **ConcurrentSkipListMap**，可視為支援並行操作的 TreeMap 版本