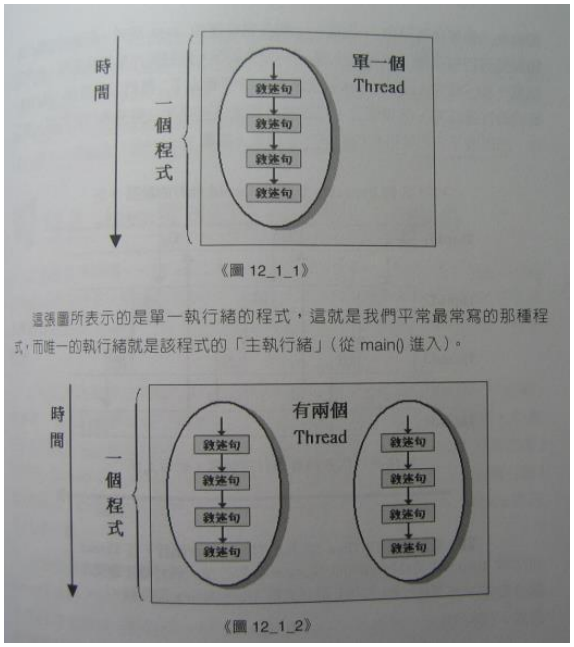
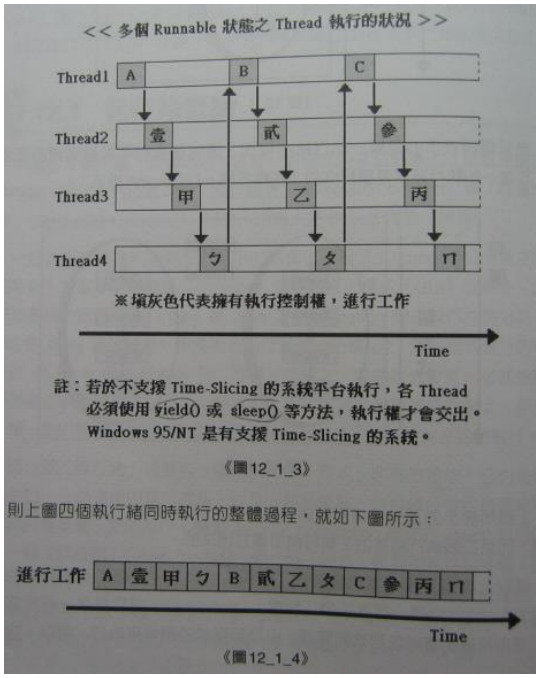
執行緒

1. thread 是一個程式內某個結構化(單程連續)的流程控制，有時我們也稱為 “執行環境”
2. 一個 “執行緒” 其實和一個結構化的 “程式” 非常的相似，它也是有一個開始點，一串連續的執行過程，以及一個結束點
3. 像 Java 之 Application 程式的 main()執行時，就是一個 “執行緒”(main thread) ，但一個 “程式” 裡面卻允許有多個執行緒同時執行





4. 上圖所表示的是一個擁有兩個執行緒的程式，其中有一個是主執行緒，而第二個執行緒是由主執行緒所建立出來的
5. 一個多執行緒的 Java 程式，即使它在執行期間能夠同時由多個 “執行緒” 進行不同的工作，但對於作業平台而言，仍然當是一個程序(Process)在運作，它們是不斷地在作換手(轉換執行控制權)的工作，而形成多個執行緒同時運作的表象
6. 每一個執行緒必須從它所在的程式(指執行時的 Process)中，切出一部份資源來使用，則每個執行緒都必須擁有自己的 “堆疊” 和 “程式計數器”，以備用來記錄交出控制權時的狀態

Java 程式的主執行緒---main()

1. 一個 Java 應用程式之主類別的 public static void main()方法是它的主程式，所以當我們以 “java.exe” 來執行 java 程式時，切入點就是在這個 main()方法
2. 對於由系統所自動建立的 “主執行緒” 我們可利用 Thread 類別的 currentTread()方法來取得它的物件參考

```
public static void 主執行緒() {

    Thread theMain = Thread.currentThread();

    for (int i = 1; i <= 10; i++) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

```
執行緒名稱：main, 執行緒ID：1  ,i=1
執行緒名稱：main, 執行緒ID：1  ,i=2
執行緒名稱：main, 執行緒ID：1  ,i=3
執行緒名稱：main, 執行緒ID：1  ,i=4
執行緒名稱：main, 執行緒ID：1  ,i=5
執行緒名稱：main, 執行緒ID：1  ,i=6
執行緒名稱：main, 執行緒ID：1  ,i=7
執行緒名稱：main, 執行緒ID：1  ,i=8
執行緒名稱：main, 執行緒ID：1  ,i=9
執行緒名稱：main, 執行緒ID：1  ,i=10
```

建立多執行緒

1. 宣告一個用來產生所需之 Thread 的類別，然後再用他來建立您需要的 Thread 物件，有兩種選擇

   <1>.讓它繼承 Thread 類別

   <2>.實作 Runnable 介面➔已繼承其它的類別，不能再繼承一次

2. 覆寫 執行緒 run()的方法
3. 執行緒從 “建立” 到 “終止” 的生命週期，當執行緒建立之後，必須以 start()來啟動它，之後它將處於 Runnable(就緒)狀態
4. 當它在 Runnable 狀態不代表它就一定在進行工作，只代表此時它已就緒，可等候獲得執行控制權，而它擁有控制權時才能進行工作

```
public class Thread1 extends Thread {

    public void run() {
        Thread theMain = Thread.currentThread();

        for (int i = 1; i <= 10; i += 2) {
            System.out.println("執行緒名稱：" + theMain.getName()
                    + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}

class Thread2 extends Thread {

    public void run() {
        Thread theMain = Thread.currentThread();
        for (int i = 2; i <= 10; i += 2) {
            System.out.println("執行緒名稱：" + theMain.getName()
                    + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```
public class Thread3 implements Runnable {

    public void run() {
        Thread theMain = Thread.currentThread();
        for (int i = 1; i <= 10; i += 2) {
            System.out.println("執行緒名稱：" + theMain.getName()
                    + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}

class Thread4 implements Runnable {

    public void run() {
        Thread theMain = Thread.currentThread();
        for (int i = 2; i <= 10; i += 2) {
            System.out.println("執行緒名稱：" + theMain.getName()
                    + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```
執行緒名稱：Thread-0, 執行緒ID：8  ,i=1
執行緒名稱：Thread-1, 執行緒ID：9  ,i=2
執行緒名稱：main, 執行緒ID：1  ,i=1
執行緒名稱：Thread-0, 執行緒ID：8  ,i=3
執行緒名稱：main, 執行緒ID：1  ,i=2
執行緒名稱：Thread-1, 執行緒ID：9  ,i=4
執行緒名稱：Thread-1, 執行緒ID：9  ,i=6
執行緒名稱：main, 執行緒ID：1  ,i=3
執行緒名稱：Thread-0, 執行緒ID：8  ,i=5
執行緒名稱：main, 執行緒ID：1  ,i=4
執行緒名稱：Thread-1, 執行緒ID：9  ,i=8
執行緒名稱：Thread-0, 執行緒ID：8  ,i=7
執行緒名稱：main, 執行緒ID：1  ,i=5
執行緒名稱：Thread-1, 執行緒ID：9  ,i=10
執行緒名稱：Thread-0, 執行緒ID：8  ,i=9
```

```
public static void 產生執行緒_繼承() {
    Thread1 obj1 = new Thread1();
    obj1.start();

    Thread2 obj2 = new Thread2();
    obj2.start();

    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}

public static void 產生執行緒_實作() {
    Thread obj1 = new Thread(new Thread3(), "AAAA");
    obj1.start();
    Thread obj2 = new Thread(new Thread4(), "BBBB");
    obj2.start();

    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

用巢狀類別的方式

```
public static void 產生執行緒_區域巢狀類別_繼承() {
    class Thread1 extends Thread {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }
    new Thread1().start();

    class Thread2 extends Thread {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }
    new Thread2().start();
    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

```java
public static void 產生執行緒_區域巢狀類別_實作() {
    class Thread3 implements Runnable {

        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }
    new Thread(new Thread3(), "AAAA").start();
    class Thread4 implements Runnable {

        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }
    new Thread(new Thread4(), "BBBB").start();
    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

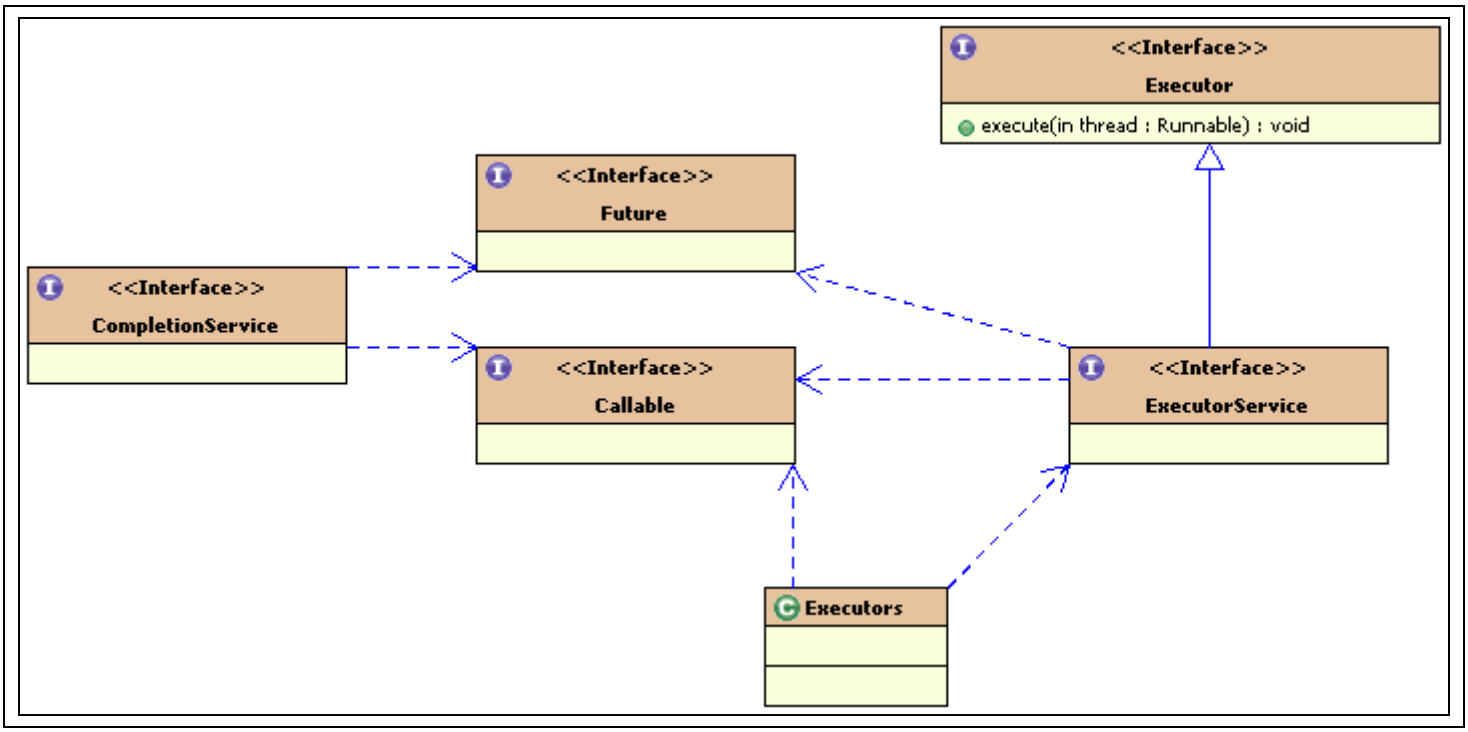
```
public static void 產生執行緒_匿名類別_繼承() {
    new Thread() {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }.start();

    new Thread() {
        public void run() {

            Thread theMain = Thread.currentThread();

            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }.start();

    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

```java
public static void 產生執行緒_匿名類別_實作1() {

    new Thread(new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();

            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }, "AAAA").start();
    new Thread(new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();

            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }, "BBBB").start();

    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

```
public static void 產生執行緒_匿名類別_實作2() {
    Runnable p = new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                    //令「主執行緒」休眠 1000 毫秒(1秒)
                } catch (InterruptedException e) {
                }
            }
        }
    };
    new Thread(p,"AAAA").start();
    Runnable q = new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();

            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                    //令「主執行緒」休眠 1000 毫秒(1秒)
                } catch (InterruptedException e) {
                }
            }
        }
    };
    new Thread(q,"BBBB").start();
    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

使用 Executors(執行緒池，線程池) 建立和管理執行緒

1. Sun 在 Java5 中，對 Java 執行緒的類別庫做了大量的擴展，其中執行緒池就是 Java5 的新特徵之一
2. 有關 Java5 執行緒新特徵的內容全部在 java.util.concurrent 下面，裡面包含數目眾多的介面和類別，熟悉這部分 API 特徵是一項艱難的學習過程
3. 執行緒池的基本思想還是一種物件池(Object Pool Pattern)的思想，開闢一塊內存空間，裡面存放了眾多(未死亡)的執行緒，池中執行緒執行調度由 池管理器 來處理。當有執行緒任務時，從池中取一個，執行完成後執行緒物件歸池，這樣可以避免反覆創建執行緒物件所帶來的性能開銷，節省了系統的資源。
4. 在 Java5 之前，要實現一個執行緒池是相當有難度的，現在 Java5 為我們做好了一切，我们只需要按照提供的 API 來使用，即可享受執行緒池帶來的極大便利
5. Executor 框架是指 java 5 中引入的一系列開發庫中與 executor 相關的一些功能類別，其中包括執行緒池，Executor，Executors，ExecutorService，CompletionService，Future，Callable 等



Executors 類別，提供了一系列工廠方法用於創建執行緒池，返回的執行緒池都實作了 ExecutorService 介面

6. Java5 的執行緒池分好多種：具體的可以分為兩類，固定尺寸的執行緒池、可變尺寸連接池

public static ExecutorService newCachedThreadPool ( )

建立新執行緒的執行緒池，這些執行緒池通常可提高程序性能。

public static ExecutorService 的 newFixedThreadPool

建立一個固定執行緒數的執行緒池，以共享的無界陣列方式來運行這些執行緒。

public static ExecutorService newSingleThreadExecutor ( )

建立一個使用單工執行緒的執行者，以無界陣列方式來運行該執行緒。

```java
public class MyThread extends Thread {

    @Override

    public void run() {
        System.out.println(Thread.currentThread().getName() + "正在執行。。。");
    }
}
```

```java
public class ExecutorCashedTest {

    public static void main(String[] args) {
        //這種方式的特點是：可根據需要創建新線程的線程池，但是在以前構造的線程可用時將重用它們。
        ExecutorService pool = Executors.newCachedThreadPool();

        // 創建線程
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();

        // 將線程放入池中進行執行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        // 關閉線程池
        pool.shutdown();
    }
}
```

```
pool-1-thread-1正在執行。。。
pool-1-thread-5正在執行。。。
pool-1-thread-4正在執行。。。
pool-1-thread-2正在執行。。。
pool-1-thread-3正在執行。。。
```

```java
public class ExecutorFixTest {

    public static void main(String[] args) {

        // 創建一個可重用固定執行緒數的執行緒池
        //ExecutorService pool = Executors.newFixedThreadPool(5);
        ExecutorService pool = Executors.newFixedThreadPool(2);
        //從以上結果可以看出，newFixedThreadPool的參數指定了可以運行的執行緒的最大數目
        //，超過這個數目的執行緒加進去以後，不會運行。
        //其次，加入執行緒池的執行緒數於托管狀態，執行緒的運行不受加入順序的影響
        // 創建執行緒
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();

        // 將線程放入池中進行執行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        // 關閉執行緒池
        pool.shutdown();
    }
}
```

```
pool-1-thread-2正在執行。。。
pool-1-thread-5正在執行。。。
pool-1-thread-4正在執行。。。
pool-1-thread-1正在執行。。。
pool-1-thread-3正在執行。。。
```

```
pool-1-thread-1正在執行。。。
pool-1-thread-2正在執行。。。
pool-1-thread-2正在執行。。。
pool-1-thread-2正在執行。。。
pool-1-thread-1正在執行。。。
```

```java
public class ExecutorSingleTest {

    public static void main(String[] args) {
        //可以看出，每次調用execute方法，其實最後都是調用了thread-1的run方法。
        ExecutorService pool = Executors.newSingleThreadExecutor();

        // 創建執行緒
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();

        // 將執行緒放入池中進行執行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        // 關閉執行緒池
        pool.shutdown();
    }
}
```

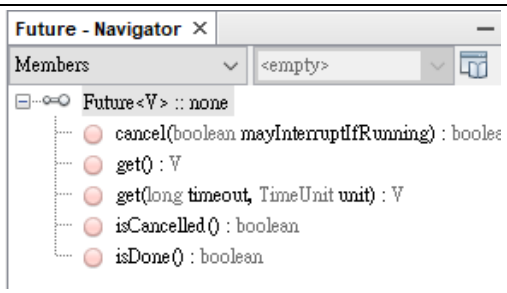
```
pool-1-thread-1正在執行。。。
pool-1-thread-1正在執行。。。
pool-1-thread-1正在執行。。。
pool-1-thread-1正在執行。。。
pool-1-thread-1正在執行。。。
```

```
public static void 產生執行緒_匿名類別_實作3() {
    ExecutorService pool = Executors.newCachedThreadPool();
    Runnable p = new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 1; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    };
    Runnable q = new Runnable() {
        public void run() {
            Thread theMain = Thread.currentThread();
            for (int i = 2; i <= 10; i += 2) {
                System.out.println("執行緒名稱：" + theMain.getName()
                        + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    };
    //execute(Runnable 對象)方法
    //其實就是對Runnabl  start()方法
    pool.execute(p);
    pool.execute(q);
    //主執行緒的工作
    Thread theMain = Thread.currentThread();
    int i = 1;
    while (i <= 5) {
        System.out.println("執行緒名稱：" + theMain.getName()
                + ", 執行緒ID：" + theMain.getId() + "  ,i=" + i);
        i++;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
    pool.shutdown();
}
```

Callable 介面

與 Runnable 類似，有個必須實作的方法，可以啟動為另一個執行緒來執行，不過 Callable 工作完成後，可以傳回結果物件，Callable 介面的定義如下：

```
public interface Callable<V> {
    V call() throws Exception;
}
```



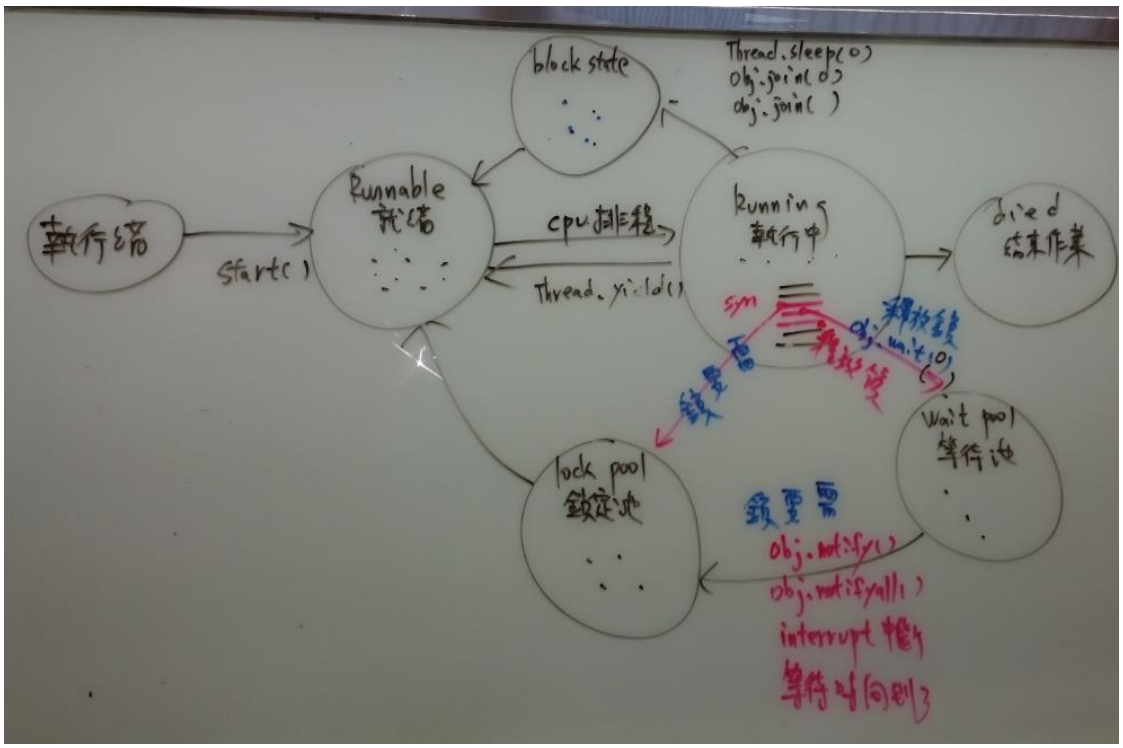
Future 是代表一個非同步呼叫的回傳結果，而這個結果會在未來某一個時間點可以取得。這樣講有點抽象，那舉個例子好了，送洗衣服就是一個非同步的呼叫，因為衣服是交給別人洗而非自己洗，而洗好的衣服是一個**未來**會發生的結果，這個結果被 Future 這個 class 包裝起來。洗衣店提供了一個非同步的服務，所以回傳一個 Future 代表的是非同步的結果(Asynchronous Result)

```
public class CallableApp {
//submit有返回值，而execute沒有
    public static void main(String[] args) throws InterruptedException, ExecutionException {

        ExecutorService pool = Executors.newSingleThreadExecutor();
        System.out.println("計算 10 階乘");
        Future result10 = pool.submit(new FactorialCalculator(10));
        Object factorialof10 = result10.get();
        System.out.println("計算 15 階乘");
        Future result15 = pool.submit(new FactorialCalculator(15));
        Object factorialof15 = result15.get();
        System.out.println("計算 20 階乘");
        Future result20 = pool.submit(new FactorialCalculator(20));
        Object factorialof20 = result20.get();

        System.out.println("5 階乘結果 : " + factorialof10);
        System.out.println("15 階乘結果 : " + factorialof15);
        System.out.println("20 階乘結果 : " + factorialof20);
        pool.shutdown();
    }
}
class FactorialCalculator implements Callable<Long> {
    private int number;

    public FactorialCalculator(int number) {
        this.number = number;
    }
    @Override
    public Long call() throws Exception {
        return factorial(number);
    }
    private long factorial(int n) {
        long total = 1L;
        for (int i = n; i >= 1; i--) {
            total = total * i;
        }
        return total;
    }
}
```

```
計算 10 階乘
計算 15 階乘
計算 20 階乘
5 階乘結果 : 3628800
15 階乘結果 : 1307674368000
20 階乘結果 : 2432902008176640000
```

執行緒的中斷

1. sleep➔讓執行緒休息一下➔從 running 到 blocked states 睡了 n 秒後再進入 runnabble state
2. join➔當一個系統中有好幾個執行緒在執行，有時候某個 thread 甲需要等待另一個 thread 乙完成某件事後，甲才能繼續執行下去➔會從 running state 變為 blocked state ，一直到目標執行緒死亡或指定時間到了，再變到 runnable state
3. yield➔用來避免一個 thread 使用 CPU 太久，當執行到 yield()時，thread 會從 running state 換到 runnable state 讓 thread scheduler 重新挑一個 thread 到 running state，但 Thread.yield()有可能會 do nothing ，因為有可能 scheduler 又挑到它去 CPU 執行



join

```
class MotherThread implements Runnable {

    public void run() {
        System.out.println("媽媽準備煮飯");
        System.out.println("媽媽發現米酒用完了");
        System.out.println("媽媽叫兒子去買米酒");

        Thread son = new Thread(new SonThread());
        son.start();

        System.out.println("媽媽等待兒子把米酒買回來");

        try {
            son.join();
        } catch (InterruptedException ie) {
            System.err.println("發生例外！");
            System.err.println("媽媽中斷煮飯");
            System.exit(1);
        }

        System.out.println("媽媽開始煮飯");
        System.out.println("媽媽煮好飯了");
    }
}
```

```
class SonThread implements Runnable {

    public void run() {
        System.out.println("兒子出門去買米酒");
        System.out.println("兒子買東西來回需5分鐘");

        try {
            for (int i = 1; i <= 5; i++) {
                Thread.sleep(1000);
                System.out.print(i + "分鐘 ");
            }
        } catch (InterruptedException ie) {
            System.err.println("兒子發生意外");
        }

        System.out.println("\n兒子買米酒回來了");
    }
}
```

```
public static void join練習() {
    Thread mother = new Thread(new MotherThread());
    mother.start();
}
```

```
媽媽準備煮飯
媽媽發現米酒用完了
媽媽叫兒子去買米酒
媽媽等待兒子把米酒買回來
兒子出門去買米酒
兒子買東西來回需5分鐘
1分鐘
2分鐘
3分鐘
4分鐘
5分鐘
兒子買米酒回來了
媽媽開始煮飯
媽媽煮好飯了
```

yield

```
class Hello1 extends Thread {

    String name;

    public Hello1(String n) {
        name = n;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(name + " Hello " + i);
            Thread.yield();
        }
    }
}
```

```
class Hello2 extends Thread {

    String name;

    public Hello2(String n) {
        name = n;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(name + " Hello " + i);

        }
    }
}
```

```
public static void yield練習() {
    Hello1 t1 = new Hello1("Thread1");
    Hello2 t2 = new Hello2("Thread2");

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

```
Thread1 Hello 1
Thread2 Hello 1
Thread2 Hello 2
Thread2 Hello 3
Thread2 Hello 4
Thread2 Hello 5
Thread2 Hello 6
Thread2 Hello 7
Thread2 Hello 8
Thread2 Hello 9
Thread2 Hello 10
Thread1 Hello 2
Thread1 Hello 3
Thread1 Hello 4
Thread1 Hello 5
Thread1 Hello 6
Thread1 Hello 7
Thread1 Hello 8
Thread1 Hello 9
Thread1 Hello 10
```

同步➔Synchronization

1. 當多個 thread 同時在存取同一個 data structure 時，會破壞 data structure 資料的完整性
   例如，若有一個 thread 在 sort 一個 list，並同時有一個 thread 在對這個 list 做 insert，而同時又有一個 thread 在對這個 list 做 delete 時，這個 list 的資料就沒有意義
2. java 使用 synchronized block 及 synchronized method
   <1>. 當任何的執行緒要進入這個 synchronized block 時，必須先取得 obj 的 lock (每一個 java object 都會有 one and only one lock ，若 obj 的 lock 已被別的 thread 先取走，則正想要執行這個 synchronized block 的 thread 會自動從 running state 變成 blocked state，即進入 obj 的 lock pool 去等待
   <2>. 一直等到這個 blocked thread 獲得了 obj 的 lock，這個 thread 才帶著 obj 的 lock 從 blocked state 進入 runnable state
   <3>. 當 thread scheduler 黑箱作業挑到它到 CPU 執行時，它才進入這個 synchronized block 執行，直到離開了 synchronized block 它才釋放 obj 的 lock
3. thread synchronization 須注意兩點
   <1>. 所有會存取重要 data 的程式碼，必須在 synchronized block 之內
   <2>. 這些被 synchronized block 所保護的重要 data 必須宣告為 private
4. synchronized block 的語法需注意兩點
   <1>. obj 必須是一個 reference variable
   <2>. 若 obj 是一個 local variable ，則無保護作用，因為對於 local variable，每一個 thread 都有自己的一份 copy

沒有同步化

```
class ShareData implements Runnable {
    private int i;

    public void run() {
        while (i < 10) {
            i++;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}
```

```
public static void 共用程式不共用資料() {

    Thread t1 = new Thread(new ShareData());
    Thread t2 = new Thread(new ShareData());

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

```
Thread-0:1
Thread-1:1
Thread-0:2
Thread-1:2
Thread-1:3
Thread-0:3
Thread-1:4
Thread-0:4
Thread-1:5
Thread-0:5
Thread-1:6
Thread-0:6
Thread-1:7
Thread-1:8
Thread-0:7
Thread-1:9
Thread-0:8
Thread-1:10
Thread-0:9
Thread-0:10
```

```
public static void 共用程式共用資料() {

    ShareData s = new ShareData();
    Thread t1 = new Thread(s);
    Thread t2 = new Thread(s);

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

		Thread-0				Thread-1	
(1)	i+1	i=1		(2)	i+1	i=2	印 2
(3)			印 2	(4)	i+1	i=3	
(5)	i+1	i=4	印 4				
(6)	i+1	i=5		(7)			印 5
				(8)	i+1	i=6	印 6
				(9)	i+1	i=7	
(10)	i+1	i=8		(11)			印 8
				(12)	i+1	i=9	印 9
				(13)	i+1	i=10	
(14)			印 10	(15)			印 10

```
Thread-1:2
Thread-0:2
Thread-0:4
Thread-1:5
Thread-1:6
Thread-0:7
Thread-1:8
Thread-1:9
Thread-0:10
Thread-1:10
```

加上同步化

```
class SyncShareData1 implements Runnable {
    private int i;

    public void run() {
        while (i < 10) {
            synchronized (this) {
                i++;
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
                System.out.println(Thread.currentThread().getName() + ":" + i);
            }
        }
    }
}
```

```
public static void 共用程式共用資料_同步化區塊() {

    SyncShareData1 s = new SyncShareData1();
    Thread t1 = new Thread(s);
    Thread t2 = new Thread(s);

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

```
Thread-0:1
Thread-0:2
Thread-1:3
Thread-1:4
Thread-1:5
Thread-1:6
Thread-1:7
Thread-0:8
Thread-0:9
Thread-0:10
Thread-1:11
```

```
class SyncShareData2 implements Runnable {
    private int i;

    public synchronized void run() {
        while (i < 10) {
            i++;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}
```

```
public static void 共用程式共用資料_同步化方法() {

    SyncShareData2 s = new SyncShareData2();
    Thread t1 = new Thread(s);
    Thread t2 = new Thread(s);

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

```
Thread-0:1
Thread-0:2
Thread-0:3
Thread-0:4
Thread-0:5
Thread-0:6
Thread-0:7
Thread-0:8
Thread-0:9
Thread-0:10
```

```
public static void 共用程式不共用資料_同步化方法() {

    Thread t1 = new Thread(new SyncShareData2());
    Thread t2 = new Thread(new SyncShareData2());

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

```
Thread-0:1
Thread-1:1
Thread-1:2
Thread-0:2
Thread-1:3
Thread-0:3
Thread-1:4
Thread-0:4
Thread-1:5
Thread-0:5
Thread-1:6
Thread-0:6
Thread-1:7
Thread-0:7
Thread-1:8
Thread-0:8
Thread-1:9
Thread-0:9
Thread-1:10
Thread-0:10
```

執行緒的合作

1. 有些情況我們需要 thread t1 暫停一下，等待 thread t2 處理完某件事，而當 thread t2 處理完某件事後，再通知 thread t1 繼續執行，這是所謂的 thread interation
2. thread interation 是靠➔wait(timeout)，notify()，notifyall()➔在 Object 類別內
3. 每一個 java object 都各有一 lock ，也各有一個 lock pool 還各有一個 wait pool，當一個 thread 執行到 obj.wait() 時，它會從 running state 變換到 blocked state，這個 thread 會被關到 obj 的 wait pool，被關之前，這個 thread 會自動釋放 obj 的 lock
4. 若一個 thread t1 被關在 obj 的 wait pool，當下列情況發生時，thread t1 會從 obj 的 wait pool 轉換到 obj 的 lock pool
   <1>. 當一個 thread 執行了 obj.notify() 時，JVM 會從 obj 的 wait pool 中，任意挑一個 thread ，把它放入 obj 的 lock pool
   <2>. 當一個 thread 執行了 obj.notifyall() 時，在 wait pool 中的所有 thread，都會被轉放到 obj 的 lock pool
   <3>. 當另一個 thread 執行了 t1.interrupt() 時，t1 會從 obj 的 wait pool 轉到 obj 的 lock pool，如果執行緒因為執行 sleep() 或是 wait()而進入 Not Runnable 狀態，而您想要停止它，您可以使用 interrupt() ，程式就會丟出 InterruptedException 例外，因而使得執行緒 離開 run()方法
   <4>. 指定時間到了 ➔ wait(2000)

```
public static void 生產者與消費者() {

    Storage s = new Storage(5);
    Producer p1 = new Producer("Producer1", s);
    Producer p2 = new Producer("Producer2", s);

    Consumer c1 = new Consumer("Consumer1", s);
    Consumer c2 = new Consumer("Consumer2", s);

    p1.start();
    p2.start();
    c1.start();
    c2.start();
}
```

```
生產者1 make  data  count= 1
生產者2 make  data  count= 2
消費者1 use  data  count: 1
消費者1 use  data  count: 0
生產者2 make  data  count= 1
生產者1 make  data  count= 2
消費者1 use  data  count: 1
生產者2 make  data  count= 2
生產者1 make  data  count= 3
消費者1 use  data  count: 2
生產者2 make  data  count= 3
消費者1 use  data  count: 2
生產者1 make  data  count= 3
消費者1 use  data  count: 2
生產者1 make  data  count= 3
消費者1 use  data  count: 2
生產者1 make  data  count= 3
消費者1 use  data  count: 2
生產者1 make  data  count= 3
消費者1 use  data  count: 2
生產者2 make  data  count= 3
生產者1 make  data  count= 4
消費者1 use  data  count: 3
生產者2 make  data  count= 4
生產者1 make  data  count= 5
消費者1 use  data  count: 4
生產者2 make  data  count= 5
```

```
通知消費者
Producer1 make  data  count= 1
通知消費者
Producer2 make  data  count= 2
通知生產者
Consumer1 use  data  count: 1
通知消費者
Producer1 make  data  count= 2
通知消費者
Producer2 make  data  count= 3
通知生產者
Consumer1 use  data  count: 2
通知消費者
Producer1 make  data  count= 3
通知消費者
Producer2 make  data  count= 4
通知消費者
Producer2 make  data  count= 5
生產者等待
通知生產者
Consumer1 use  data  count: 4
通知消費者
Producer1 make  data  count= 5
通知生產者
Consumer1 use  data  count: 4
通知消費者
```

```
class Storage {

    private int count; //現在庫存量
    private int size; //庫存量上限

    public Storage(int s) {
        size = s;
    }

    public synchronized void addData(String n) {
        while (count == size) {

            try {
                System.out.println("生產者等待");
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        count++;
        System.out.println(n + " make  data  count= " + count);
        System.out.println("通知消費者");
        this.notify();
    }

    public synchronized void delData(String n) {
        while (count == 0) {
            try {
                System.out.println("消費者等待");
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        count--;
        System.out.println(n + " use  data  count: " + count);
        System.out.println("通知生產者");
        this.notify();
    }
}
```

```
class Producer extends Thread {

    private String name;
    private Storage s;

    public Producer(String n, Storage s) {
        name = n;
        this.s = s;
    }
    public void run() {
        while (true) {
            s.addData(name);
            try {
                Thread.sleep((int) Math.random() * 3000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```
class Consumer extends Thread {

    private String name;
    private Storage s;

    public Consumer(String n, Storage s) {
        name = n;
        this.s = s;
    }
    public void run() {
        while (true) {
            s.delData(name);

            try {
                Thread.sleep((int) Math.random() * 3000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

執行緒相關方法

Method name	Features	Method Type	Class
yield()	讓出CPU使用權	static	Thread
run()	instance	Thread	
start()		instance	Thread
interrupt()		instance	Thread
sleep(ms)	會丟出InterruptException ms 是long type，單位是millisecond ms 不可省略	static	Thread
join(ms)	會丟出InterruptException ms 是long type，單位是millisecond 若 ms 為 0，則表等待的時間為 ∞ 若省略ms，則表等待的時間為 ∞	instance	Thread
wait(ms)	會丟出InterruptException ms 是long type，單位是millisecond 若 ms 為 0，則表等待的時間為 ∞ 若省略ms，則表等待的時間為 ∞ 須事先拿到 this object的lock	instance	Object
notify()	須事先拿到 this object的lock	instance	Object
notifyall()	須事先拿到 this object的lock	instance	Object

濫用/誤用 synchronized➔死結 deadlock

1. Race condition 競速➔二條執行緒看誰被當下排程給的 cpu 時間多，或被選到的次數多，誰就容易先達到目標
2. Greedy 貪婪➔某一個執行緒長時間佔有共享資源，讓其他執行緒無法使用或經常阻塞
3. Starvation 飢餓➔飢餓是指某一個或多個執行緒因為種種原因無法獲得所需要的資源，導致一直無法執行，比如它的執行緒優先順序可能太低，而高優先順序執行緒不斷搶佔它所需的資源，導致低優先順序執行緒無法工作。還有一種情況就是某一執行緒一直占著關鍵資源不放，導致其他需要這個資源的執行緒無法正常執行，這種情況也是飢餓的一種。
4. Deadlock 死結

   <1>.在 multi-threading 的環境下， 使用 synchronization 的機制，可能會產生 deadlock ，例如 thread t1 已擁有 object a 的 lock thread t2 已擁有 object b 的 lock 但 t1 又想去取得 object b 的 lock (b 的 lock 已被 thread t2 取走)，所以 JVM 把 thread t1 放在 b 的 lock pool，同時 t2 也想去取得 object a 的 lock (a 的 lock 已被 thread t1 取走)，所以 JVM 把 t2 放 在 a 的 lock pool，結果是 thread t1 拿著 a 的 lock 在等 b 的 lock 而 thread t2 拿著 b 的 lock 在等 a 的 lock deadlock (死結) 就發生了，這兩個 thread 就在那裡呆呆地一直等下去，直到永遠

   <2>.Java 技術 沒有提供任何機制，來偵測或防止 deadlock 發生，因為 防止 deadlock 發生是 programmer 的責任，在設計 synchronizing threads 時，programmer 若想防止 deadlock 發生，須遵守下列規則 ： 每一個執行緒都按一定的規則取得鎖，而且按反順序釋放鎖
5. 活鎖➔活鎖指事物 1 可以使用資源，但它讓其他事物先使用資源；事物 2 可以使用資源，但它也讓其他事物先使用資源，於是兩者一直謙讓，都無法使用資源

```
public class Deadlock {
    static StringBuffer a = new StringBuffer("");
    static StringBuffer b = new StringBuffer("");

    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                synchronized (a) {
                    a.append("a");
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException ie) {
                    }
                    synchronized (b) {
                        b.append("b");
                    }
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                synchronized (b) {
                    b.append("B");
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException ie) {
                    }
                    synchronized (a) {
                        a.append("A");
                    }
                }
            }
        });
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException ie) {
        }
        System.out.println(a);
        System.out.println(b);
    }
}
```

遞迴

```
public static void 階乘() {
    int num = 5;
    System.out.println("階乘_一般方法 " + num + " degree = " + 階乘_一般方法(num));
    System.out.println("階乘_遞迴 " + num + " degree = " + 階乘_遞迴(num));
}

// 5 * 階乘_遞迴(4)
// 5 * 4 * 階乘_遞迴(3)
// 5 * 4 * 3 * 階乘_遞迴(2)
// 5 * 4 * 3 * 2 * 階乘_遞迴(1)
// 5 * 4 * 3 * 2 * 1 * 階乘_遞迴(0)
public static int 階乘_一般方法(int num) {//沒有使用遞迴

    int total = 1;

    for (int i = 1; i <= num; i++) {
        total = total * i;
    }
    return total;
}

public static int 階乘_遞迴(int num) { //使用遞迴

    if (num == 0) {
        return 1;
    } else {
        return num * 階乘_遞迴(num - 1);
    }
}
```

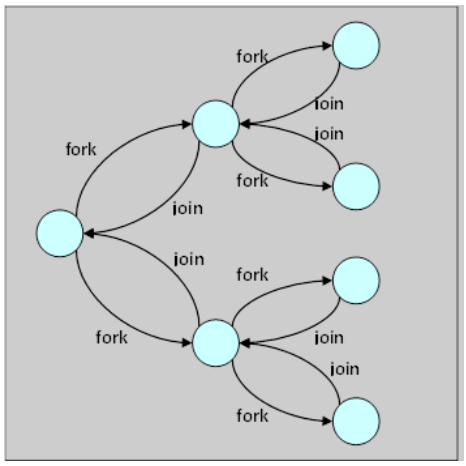
```
階乘_一般方法 5 degree = 120
階乘_遞迴 5 degree = 120
```

```
// 費式數列_遞迴(5)
// 費式數列_遞迴(3) + 費式數列_遞迴(4)
// 費式數列_遞迴(1) + 費式數列_遞迴(2) + 費式數列_遞迴(2) + 費式數列_遞迴(3)
// 費式數列_遞迴(1) + 費式數列_遞迴(0) + 費式數列_遞迴(1)) + 費式數列_遞迴(0) + 費式數列_遞迴(1) +  費式數列_遞迴(1) + 費式數列_遞迴(2)
// 費式數列_遞迴(1) + 費式數列_遞迴(0)  + 費式數列_遞迴(1) +  費式數列_遞迴(0) + 費式數列_遞迴(1)) + 費式數列_遞迴(1) + 費式數列_遞迴(0) + 費式數列_遞迴(1)
// 1+0+1+0+1+1+0+1 ==>5

public static void 費式數列() {
    int num = 45;
    System.out.println("費式數列_遞迴 " + num + "=" + 費式數列_遞迴(num));
}

public static int 費式數列_遞迴(int num) {

    if (num == 0 || num == 1) {
        return num;
    } else {
        return 費式數列_遞迴(num - 1) + 費式數列_遞迴(num - 2);
    }
}
```

```
費式數列_遞迴 45=1134903170
```

Fork/Join 框架➔分而治之

1. 在現行多核系統的硬體架構下，為求執行效率，Java7 實現了 Fork/Join 框架，該任務設計模式是將任務不斷向下分解（fork）至最小單位，程式將計算此最小單位的的邏輯結果，得到結果後再往上合併（join）計算至源頭，到了源頭答案就出來了，就是所謂分而治之
2. 之所以會用到 fork，是因為雖然你的 OS 是多工的 (可以同時執行多個程式)，但你的程式是單工的 (一次只能作一件事)...
   所以當你的程式在跑一個需要長時間運算的動作的時候 (例：求一個大數的因數)，你的程式基本上無法作其他的動作 (例：輸出，接受使用者的輸入等的)...
3. Fork 就提供了這個的解決方法... 你可以 fork 出你本身程式的另一份 copy ，在這個新的 copy 中作需要長時間運算的動作... 而在原本的程式中，就做其他的動作 (輸出跟接受使用者的輸入等等，這樣一來，使用者就不會因為你的程式完全沒動作，而覺得你的程式當了)... 而因為你的 OS 是多工的，你的這兩個程式就可以像是在同時間運算的樣子... (同時間在作運算，也同時間在作輸出) 等到你的長時間運算完成了，你可以再把你 fork 出去的程式 join 回來...



4. 費式數列從 0 和 1 開始，之後 就由之前的兩數相加

0 ，1，1，2 ，3 ，5，8，13，21，34，55，89，144，233，377，610，987，1597，2854，4181，6765，10946

5. ForkJoinTask 是實作了 Future 介面 才能取得子任務所回報的計算結果，ForkJoinTask 下的兩個子類別
   <1>.RecursiveTask 表示需有返回結果
   <2>.RecursiveAction 表示不需要有返回結果

```
//class SumArray extends RecursiveTask<T>{
//    @Override
//    protected <T> compute() {
//
//        if(...<=THRESHOLD) {//門檻值
//            進行商業邏輯作業
//        }
//        else{
//            任務過大要切割子任務...
//        }
//    }
//}
```

費氏數列 1-45

```
public class FibonacciForkJoin extends RecursiveTask<Integer> {

    private int num = 0;
    private int result = 0;

    public FibonacciForkJoin(int num) {
        this.num = num;
    }
    public int getResult() {
        return result;
    }
// Fork/Join
    @Override
    protected Integer compute() {

        if (num < 45) { //門檻值
            result = fibonacci(num);
        } else {
            FibonacciForkJoin task1 = new FibonacciForkJoin(num - 1);
            FibonacciForkJoin task2 = new FibonacciForkJoin(num - 2);
            task1.fork();
            task2.fork();
            result = task1.join() + task2.join();
        }
        return result;
    }
// 遞迴
    public int fibonacci(int num) {

        if (num == 0 || num == 1) {
            return num;
        } else {
            return (fibonacci(num - 1) + fibonacci(num - 2));
        }
    }
```

```
    public static void main(String[] args) {
        int num = 45;

        //ForkJoin
        long t3 = new Date().getTime();
        //取得本機 CPU 核心數==>會影響執行效率
        int processors = Runtime.getRuntime().availableProcessors();
        //建立 FibonacciForkJoin 物件
        FibonacciForkJoin task = new FibonacciForkJoin(num);
        //執行緒池實例 , 並設定執行的 CPU 核心數量
        ForkJoinPool pool = new ForkJoinPool(processors);
        //開始透過 Fork/Join 分派任務
        pool.invoke(task);
        //取出最後的結果
        System.out.println(task.getResult() + ".");
        long t4 = new Date().getTime();
        System.out.println("花費時間 :" + (t4 - t3));
        System.out.println("Processors :" + processors);
    }
}
```

```
102334155.
花費時間 :1475
Processors :2
```

1+2+3+.....+100

```
public class CountTask extends RecursiveTask<Integer> {
    public static final int THRESHOLD = 2;
    private int start;
    private int end;
    int sum = 0;

    public CountTask(int start, int end) {
        this.start = start;
        this.end = end;
    }
    public int getResult() {
        return sum;
    }
    @Override
    protected Integer compute() {

        //如果任務足夠小 , 就計算任務
        boolean canComputer = (end - start) < THRESHOLD;

        if (canComputer) { //門檻值

            for (int i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            //如果任務大於門檻值 , 就分列成兩個子任務計算
            int middle = (start + end) / 2;
            CountTask leftTask = new CountTask(start, middle);
            CountTask rightTask = new CountTask(middle + 1, end);
            leftTask.fork(); //分配一條執行緒來執行此任務
            rightTask.fork(); //分配一條執行緒來執行此任務
            sum = leftTask.join() + rightTask.join();
        }
        return sum;
    }
    public static void main(String[] args) {

        CountTask task = new CountTask(1, 100);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(task);
        System.out.println(task.getResult());
    }
}
```

```
5050
```