java 8

java 8 (又稱為 jdk 1.8)。Oracle 公司於 2014 年 3 月 18 日發佈 Java 8

1. 它支持函數式程式語言(函數式介面)➔一個介面只有一個抽象方法
2. 新的 JavaScript 引擎
3. 新的日期 API
4. 新的 Stream API 等

java8 的新特性

1. 介面的宣告使用 default 預設方法與 static 靜態方法 這兩個新概念來擴充套件
2. Lambda 表達式
3. 函數式介面➔一個介面只有一個抽象方法
4. 方法與建構函數的引用➔Java 8 允許你使用 :: 關鍵字來傳遞方法或者建構函數引用
5. Lambda 的有效範圍

   在 Lambda 表達式中訪問外層作用域(有效範圍)和老版本的匿名類別中的方式很相似。你可以直接訪問標記了 final 的外層區域變數，或者物件變數以及靜態變數
6. 訪問 local Variable 區域變數

   我們可以直接在 Lambda 表達式中訪問外層的區域變數，這裡的區域變數可以不用聲明為 final，不過這裡的區域變數必須不可被後面的程式碼修改，即隱性的具有 final 的語意
7. 訪問 object field 物件變數與 static Variable 靜態變數➔Lambda 內部對於物件變數以及靜態變數是既可讀又可寫
8. 訪問介面的 default(默認)方法

   JDK 1.8 API 包含了很多內建的函數式介面，在老 Java 中常用到的如 Comparator 或者 Runnable 介面，這些介面都增加了 @FunctionalInterface 註解以便能用在 Lambda 上

   <1>.Predicate 介面

   <2>.Function 介面

   <3>.BiFunction 介面

   <4>.BinaryOperator 介面

   <5>.UnaryOperator 介面

   <6>.Supplier 介面(供應商)

   <7>.Consumer 介面(消費者)

   <8>.Comparator 介面

   <9>.Optional 介面

   <10>.Stream 介面

   (1).Filter 過濾

   (2).Sort 排序

   (3).Map 映射

   (4).Match 匹配

   (5).Count 計數

   (6).Reduce 規約
9. Date API

   <1>.LocalDate 本地日期

   <2>.LocalTime 本地時間

   <3>.LocalDateTime 本地日期時間

   <4>.Timezones 時區
10. Annotation 注解：支持多重注解

1. 介面的宣告用 default(默認方法)與 static(靜態方法)這兩個新概念來擴充套件

Java 8 允許我們給介面添加一個非抽象的方法，只需要使用 default 關鍵字即可，這個特徵又叫做擴展方法

<1>.jdk8 以前，介面中的方法都是抽象方法，通常由介面的實作類別來做具體功能實現。

<2>.如果有一天，產品提需求了，你需要给所有介面的實作類別都添加一個新的功能，那怎麼辦？

<3>.jdk8 以前的做法是在介面中定義這個抽象方法，然後所有實作類別必須實作這個方法，如果實作類別比較多，那改起來會很麻煩，這種情況下是不利於維護的。

<4>.在 jdk8 中就有了好的解決方式，就是在介面中加一個 default 方法，這個 default 方法有具體實作，這樣就不用去修改實作類別，很省事

```java
interface Super01 {

    void a();

    default void b() {
        System.out.println("b");
    }

    //工廠函數
    static int c() {
        return 123;
    }
}

class Sub0101 implements Super01 {

    public void a() {
        System.out.println("x");
    }
}

class Sub0102 implements Super01 {

    public void a() {
        System.out.println("y");
    }
}
////////////////////////////////////////////////

public class Test01 {

    public static void main(String[] args) {
        System.out.println(Super01.c()); //123
        Super01 p = new Sub0101();
        p.a(); //"x"
        p.b(); //"b"
    }
}
```

```java
interface Super02{

    double calculate(int x);

    default double sqrt(int x) {
        return Math.sqrt(x);
    }
}
class Sub02 implements Super02 {

    @Override
    public double calculate(int x) {
        return sqrt(x * 100);
    }
}
////////////////////////////////////////////////////////
public class Test02 {

    public static void main(String[] args) {
        //方法實作
        Super02 a = new Sub02();
        System.out.println(a.calculate(100)); //100.0
        System.out.println(a.sqrt(100));  //10.0

        //匿名類別
        Super02 b = new Super02() {
            @Override
            public double calculate(int x) {
                return sqrt(x * 100);
            }
        };
        System.out.println(b.calculate(100));  //100.0
        System.out.println(b.sqrt(100));  //10.0

        //Lambda表達式中是無法訪問到默認方法的==>sqrt
        Super02 c = x -> sqrt(x * 100);
        System.out.println(c.calculate(100));
    }
}
```

注意事項

1. 介面中 default 方法的注意事項

   <1>.介面不能擁有狀態(java 中無狀態的物件就是指某種沒有任何的屬性，僅僅由方法組成的物件)，這意味着它在實作一些 Object 方法中存在一定的局限性。

   <2>.Java 中如果[Object 父類]和介面擁有相同的方法，子類別會去優先實現[Object 父類]中的方法而不是介面中的方法，這樣的話會使得我們的介面 default 方法沒有意義

2. 如果介面中有 default 修飾的方法不需要重寫。
3. 如果一個類別實作兩個介面，這兩個介面同時有相同的抽象方法，在類別中只需要重寫一次這個方法。
4. 如果兩個介面裡的方法名相同都是 default 方法，裡面的方法體不同，在類別中需要重寫該方法。
5. 如果兩個介面中方法名，參數都相同的方法，一個介面是抽象方法，另一個是 default 修飾有方法體。這是該類也必須重寫該方法。
6. Lambda 表達式中是無法訪問到 default 方法的

```java
//以下不允許 ，介面不可定義跟 Object 一樣的方法
interface Super03 {

//    default int hashCode() {
//
//    }
//
//    default boolean equals(Object object) {
//
//    }
}
class Sub03 implements Super03 {

}
//////////////////////////////////////////////////
public class Test03 {

    public static void main(String[] args) {

    }
}
```

```java
interface Super04 {

    void a();

    default void b() {
        System.out.println("b");
    }
}
//如果介面中有default修飾的方法不需要重寫
class Sub04 implements Super04 {

    public void a() {
        System.out.println("a");
    }
}
public class Test04 {

    public static void main(String[] args) {
        Super04 p=new Sub04();
        p.a();
        p.b();
    }
}
```

```java
interface Super0501 {
    void a();
}
interface Super0502 {
    void a();
}
//////////////////////////////////////////////////
//不同介面只需要實作一次
class Sub05 implements Super0501, Super0502 {

    public void a() {
        System.out.println("a");
    }
}
public class Test05 {

    public static void main(String[] args) {
        Super0501 p = new Sub05();
        p.a();
    }
}
```

```java
interface Super0601 {
    default void a() {
        System.out.println("a");
    }
}
interface Super0602 {
    default void a() {
        System.out.println("b");
    }
}
//////////////////////////////////////////////////
//如果兩個介面裡的方法名相同都是default方法,在類別中需要重寫該方法
class Sub06 implements Super0601, Super0602 {

    public void a() {
        System.out.println("c");
    }
}
public class Test06 {

    public static void main(String[] args) {
        Super0601 p = new Sub06();
        p.a();
    }
}
```

```java
interface Super0701 {
    default void a() {
        System.out.println("a");
    }
}
interface Super0702 {
    void a();
}
//////////////////////////////////////////////////////
//如果兩個介面中方法名，參數都相同的方法，一個介面是抽象方法
//，另一個是default修飾有方法體。這是該類也必須重寫該方法
class Sub07 implements Super0701, Super0702 {

    public void a() {
        System.out.println("c");
    }
}
public class Test07 {

    public static void main(String[] args) {
        Super0701 p = new Sub07();
        p.a();
    }
}
```

```
//定義一個介面，實現加減乘除
interface Calculator1 {

    default int addThree(int first, int second, int Third) {
        return first + second + Third;
    }

    int add(int first, int second);

    int subtract(int first, int second);

    int divide(int first, int second);

    int multiply(int first, int second);
}
class BasicCalculator1 implements Calculator1 {

    @Override
    public int add(int first, int second) {
        return first + second;
    }

    @Override
    public int subtract(int first, int second) {
        return first - second;
    }

    @Override
    public int divide(int first, int second) {
        return first / second;
    }

    @Override
    public int multiply(int first, int second) {
        return first * second;
    }
}
//加減乘除的工廠另外定義
class CalculatorFactory {

    public static Calculator1 getInstance() {
        return new BasicCalculator1();
    }
}
public class Test08 {

    public static void main(String[] args) {
        //呼叫 CalculatorFactory工廠函數方式呼叫
        System.out.println(CalculatorFactory.getInstance().add(1, 2));
        System.out.println(CalculatorFactory.getInstance().addThree(1, 2, 3));
    }
}
```

```
interface Calculator2 {

//介面中的靜態方法可以幫我們實現靜態工廠類，不需要你去額外寫一個工廠類了
    static Calculator2 getInstance() {
        return new BasicCalculator2();
    }

    default int addThree(int first, int second, int Third) {
        return first + second + Third;
    }

    int add(int first, int second);

    int subtract(int first, int second);

    int divide(int first, int second);

    int multiply(int first, int second);
}
class BasicCalculator2 implements Calculator2 {

    @Override
    public int add(int first, int second) {
        return first + second;
    }

    @Override
    public int subtract(int first, int second) {
        return first - second;
    }

    @Override
    public int divide(int first, int second) {
        return first / second;
    }

    @Override
    public int multiply(int first, int second) {
        return first * second;
    }
}
public class Test09 {

    public static void main(String[] args) {
        //呼叫 Calculator2 自身工廠函數方式呼叫
        System.out.println(Calculator2.getInstance().add(1, 2));
        System.out.println(Calculator2.getInstance().addThree(1, 2, 3));
    }
}
```

2. Lambda 表達式

<1>.Lambda 為一個函數，可以根據輸入的值，決定輸出的值。但 Lambda 與一般函數不同的是，Lambda 並不需要替函數命名(如 F(X)=X+2，G(X)=F(X)+3 中的 F、G 便是函數的名稱)，所以我們常常把 Lambda 形容為「匿名的」(Anonymous)

<2>.Lambda 語法只能用來表示一個「只擁有一個方法的介面」所實作出來的匿名類別，但這樣講也是不太正確，現在姑且先將其認知成這樣，「只擁有一個方法的介面」在 Java 中很常使用到，例如執行緒的 Runnable 介面只擁有一個 run 方法，或是 AWT 的 ActionListener 只擁有一個 actionPerformed 方法。這類介面，都可以直接使用 Lambda，將它們擁有的那單個方法快速實作出來。在 Java8 之前，我們將這類的介面稱為 Single Abstract Method type(SAM)，在 Java 8 之後，因為這類的介面變得十分重要，所以將其另稱為 Functional Interface(函數式介面)

<3>.Lambda 包含 3 個部分：

(1).括弧包起來的參數➔參數可以寫類型，也可以不寫，JVM 很智能的，它能自己推算出來

(2).一個箭頭

(3).方法體，可以是單個語句，也可以是語句塊➔單行不用大括號，多行要用大括號，單行有回傳值不用 return，多行有回傳值需要加上 return

<4>.Lambda 的定義：

(1).Lambda 是方法的實現

(2).Lambda 是延遲執行的➔ Lambda 是如何做到的呢？可以反編譯後查看字節碼。

裡面有一個叫做 invokedynamic 的指令。invokedynamic 是從 jdk7 開始引入的，jdk8 開始落地。

可以看出來 Lambda 並不是語法糖，它不是像匿名內部類別那樣生成那種帶有$的匿名類。

簡單的說，這裡只是定義了一個方法調用點，具體調用那個方法要到運行時才能決定，這就是前面所说的：延遲執行。

(3).Lambda(λ)表達式本質上是一個匿名方法

<5>.Lambda 語法結構➔input ->body

input

沒有參數(一定要加括號) ()
1 個參數(有型態一定要加括號) **(int x)**
1 個參數(省略型態) (沒型態不一定要加括號) **(x)**
1 個參數(省略型態) (不用加括號) **x**
2 個參數(不省略型態)(一定要加括號) (int x，int y)
2 個參數(省略型態) (一定要加括號) (x，y)

body

單行不回傳值，加大括號 {System.out.println("NO");};
單行不回傳值，沒加大括號 System.out.println("NO");

多行不回傳值，一定要加大括號
{
System.out.println("xxx");
System.out.println("yyy");
};

單行回傳值，有加大括號，一定要加 return {return x+y;};

單行回傳值，沒加大括號，不用加 return

```
x+y;
```

多行回傳值，一定要加大括號，一定要加 return

```
{
 x++;
 y++;
 return x+y;
};
```

<6>.Lambda 範例➔函數式介面(只有一個方法的介面)才能用 Lambda 語法

```
//定義介面
interface Super01 {

    void a();
}

interface Super02 {

    void a(int x);
}

interface Super03 {

    void a(int x, int y);
}

interface Supe04 {

    int a();
}

interface Super05 {

    int a(int x);
}

interface Super06 {

    int a(int x, int y);
}
```

```
//子類別實作
class Sub01 implements Super01 {

    public void a() {
        System.out.println("我愛你");
    }
}
class Sub02 implements Super02 {

    public void a(int x) {
        System.out.println(x);
    }
}
class Sub03 implements Super03 {

    public void a(int x, int y) {
        System.out.println(x);
        System.out.println(y);
    }
}
class Sub04 implements Supe04 {

    public int a() {
        return 999;
    }
}
class Sub05 implements Super05 {

    public int a(int x) {
        return x;
    }
}
class Sub06 implements Super06 {

    public int a(int x, int y) {
        x = x + 1;
        y = y + 1;
        return x + y;
    }
}
```

```
public class Test01 {

    public static void main(String[] args) {
        //1.1 正常實作
        Super01 a1 = new Sub01();
        a1.a(); //"我愛你"

        //2.1 正常實作
        Super02 a2 = new Sub02();
        a2.a(123); //123

        //3.1 正常實作
        Super03 a3 = new Sub03();
        a3.a(123, 456);

        //4.1 正常實作
        Supe04 a4 = new Sub04();
        System.out.println(a4.a());

        //5.1 正常實作
        Super05 a5 = new Sub05();
        System.out.println(a5.a(123));

        //6.1 正常實作
        Super06 a6 = new Sub06();
        System.out.println(a6.a(123, 456));
```

```
    //1.2 匿名類別
    Super01 b1 = new Super01() {
        public void a() {
            System.out.println("我愛你");
        }
    };
    b1.a(); //"我愛你"

    //2.2 匿名類別
    Super02 b2 = new Super02() {
        @Override
        public void a(int x) {
            System.out.println(x);
        }
    };
    b2.a(123); //123

    //3.2 匿名類別
    Super03 b3 = new Super03() {
        public void a(int x, int y) {
            System.out.println(x);
            System.out.println(y);
        }
    };
    b3.a(123, 456);
```

```
    //4.2 匿名類別
    Supe04 b4 = new Supe04() {
        public int a() {
            return 999;
        }
    };
    System.out.println(b4.a());
    //5.2 匿名類別
    Super05 b5 = new Super05() {
        public int a(int x) {
            return x;
        }
    };
    System.out.println(b5.a(123));
    //6.2 匿名類別
    Super06 b6 = new Super06() {
        public int a(int x, int y) {
            x = x + 1;
            y = y + 1;
            return x + y;
        }
    };
    System.out.println(b6.a(123, 456));
```

```
        //1.3 Lambda 沒有參數 , 單行不回傳
        Super01 c1 = () -> System.out.println("我愛你");
        c1.a(); //"我愛你"

        //2.3 Lambda 一個參數 , 單行不回傳
        Super02 c2 = x -> System.out.println(x);
        c2.a(123); //123

        //3.3 Lambda 二個參數 , 多行不回傳
        Super03 c3 = (x, y) -> {
            System.out.println(x);
            System.out.println(y);
        };
        c3.a(123, 456);
        //4.3 Lambda 沒有參數 , 單行回傳
        Supe04 c4 = () -> 999;
        System.out.println(c4.a());

        //5.3 Lambda 一個參數 , 單行回傳
        Super05 c5 = x -> x;
        System.out.println(c5.a(123));

        //6.3 Lambda 二個參數 , 多行回傳
        Super06 c6 = (x, y) -> {
            x = x + 1;
            y = y + 1;
            return x + y;
        };
        System.out.println(c6.a(123, 456));
    }
}
```

```java
//Runnable介面是 系統 API
public class Test02 {

    public static void main(String[] args) {
        //1. 匿名類別
        Runnable a = new Runnable() {

            public void run() {
                for (int i = 1; i <= 10; i += 2) {
                    System.out.print(i + " ");
                    try {
                        Thread.sleep(1000);
                    } catch (Exception e) {
                    }
                }
            }
        };
        new Thread(a).start();
        //2. Lambda 沒有介面 , 多行沒有回傳
        Runnable b = () -> {
            for (int i = 2; i <= 10; i += 2) {
                System.out.print(i + " ");
                try {
                    Thread.sleep(1000);
                } catch (Exception e) {
                }
            }
        };
        new Thread(b).start();

    }
}
```

```java
//Comparator 是 系統的 API ==>武斷順序 可以由大到小排
public class Test03 {

    public static void main(String[] args) {

        List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
        //1. 匿名類別
        Collections.sort(names, new Comparator<String>() {
            public int compare(String o1, String o2) {
                return o2.compareTo(o1);
            }
        });
        System.out.println(names); //[xenia, peter, mike, anna]

        //2. Lambda
        Collections.sort(names, (o1, o2) -> o2.compareTo(o1));
        System.out.println(names); //[xenia, peter, mike, anna]
    }
}
```

Lambda 實際應用與效能比較

1. Lambda 是用來取代 Functional Interface(函數式介面)產出的匿名類別
   在 Java 中，許多只有一個方法的介面，如果要使用這些介面，往往需要使用到至少 4 行程式碼才有辦法達成
2. 使用 Lambda 來取代以往 Functional Interface 的使用方式，可以大大的縮短程式碼，在編譯的過程中，也可以避免掉產生新的.class 檔案出來，執行的時候，也不會再重新 new 出一個物件實體，而是直接將 Lambda 的 body 程式碼存放在記憶體，直接以類似 call function 的方式去執行，大大的提升程式的執行效能

```
public class Test04 {

    public static void main(String[] args) {
        new Test04().test();
    }
    public void test() {
        //傳統方法-至少要 4 行
        Runnable r1 = new Runnable() {
            public void run() {
                System.out.println("r1: " + this.getClass().getName());
            }
        };
        //Lambda
        Runnable r2 = () -> System.out.println("r2: " + this.getClass().getName());

        new Thread(r1).start();
        new Thread(r2).start();
    }
}
```

```
r1: p02_Lambda表達式.Test04$1
r2: p02_Lambda表達式.Test04
```

雖然兩個執行緒都是呼叫 this.getClass()，但 print 出來的結果卻不一樣。
而且可以知道使用 Lambda 的 r2，this 所指的物件就是此行 Lambda 語法所在的物件，並不會像沒使用 Lambda 的 r1 一樣變成一個匿名類別的物件

3. **函數式介面**

<1>.是一個介面，只有一個抽象的方法

<2>.因為 jdk8 開始，介面可以有 default 方法，所以，函數式介面也是可以有 default 方法的，但是，只能有一個抽象的方法。

<3>.與此對應，新引入了一個註解：@FunctionalInterface。這個注解只是 “註解” 的作用，說明這個介面是函數式介面

<4>.加不加 @Functional Interface 對於介面是不是函數式介面沒有影響，該注解只是提醒編譯器去檢查該介面是否只包含一個抽象方法，編譯器並不會使用這個註解來決定一個介面是不是函數式介面

<5>.不是每個介面都可以縮寫成 Lambda 表達式，只有那些函數式介面（Functional Interface）才能縮寫成 Lambda 表示式

```java
//函數式介面 --> 只能有一個抽象方法 , 但可以有 default 與 static 方法
@FunctionalInterface
interface Super01 {

    void a();
    void b();

    default void c() {
        System.out.println("c");
    }

    static String d() {
        return "d";
    }
}

public class Test01 {

    public static void main(String[] args) {
        Super01 p = () -> System.out.println("a");;
        p.a();
        p.c();
        System.out.println(Super01.d());
    }
}
```

```java
@FunctionalInterface
interface Converter<F, T> {

    T convert(F from);
}

public class Test02 {

    public static void main(String[] args) {
        //匿名類別 從 String 轉成 Integer
        Converter<String, Integer> a = new Converter<String, Integer>() {
            public Integer convert(String from) {
                return Integer.valueOf(from);
            }
        };
        System.out.println(a.convert("123")); // @123@
        ////////////////////////////////////////////////////////////////
        //Lambda 從 String 轉成 Integer
        Converter<String, Integer> b = from -> Integer.valueOf(from);
        System.out.println(b.convert("123")); // @123@
    }
}
```

4. 方法與建構函數引用➔Lambda 的方法引用形式有以下四種➔允許你使用 :: 關鍵字來傳遞方法

<1>.物件名 :: 實例方法名

<2>.類名 :: 靜態方法名

<3>.類名 :: 實例方法名

<4>.類別名 :: new ➔建構函數引用

重點➔方法的參數和回傳值要一致。

```
@FunctionalInterface
interface Super01<T> {

    void accept(T t);
}

//Info類中的show方法實現了相同的功能
class Info<T> {

    public void show(T t) {
        if (t != null) {
            System.out.println(t);
        } else {
            System.out.println("t 為空");
        }
    }
}
////////////////////////////////////////////////////////////////
public class Test01 {

    public static void main(String[] args) {
        //匿名類別
        Super01<String> a = new Super01<String>() {
            public void accept(String t) {
                if (t != null) {
                    System.out.println(t);
                } else {
                    System.out.println("t 為空");
                }
            }
        };
        a.accept("我愛你");
        a.accept(null);
        System.out.println("------------");
        //Lambda
        Super01<String> b = t -> {
            if (t != null) {
                System.out.println(t);
            } else {
                System.out.println("t 為空");
            }
        };
        b.accept("我愛你");
        b.accept(null);
        System.out.println("------------");
        //方法引用-1. 物名 :: 實例方法名
        Info<String> info = new Info<String>();
        Super01<String> c = info::show;
        c.accept("我愛你");
        c.accept(null);
        System.out.println("------------");
    }
}
```

```java
@FunctionalInterface
interface Super02<F, T> {

    T convert(F from);
}

//////////////////////////////////////////////////////////////////////////
public class Test02 {

    public static void main(String[] args) {

        //匿名類別
        Super02<String, Integer> a = new Super02<String, Integer>() {
            public Integer convert(String from) {
                return Integer.valueOf(from);
            }
        };
        System.out.println(a.convert("123")); //@123@
        //Lambda
        Super02<String, Integer> b = from -> Integer.valueOf(from);
        System.out.println(b.convert("123")); //@123@
        //方法引用-2.類名 :: 靜態方法名
        Super02<String, Integer> c = Integer::valueOf;
        Integer converted = c.convert("123");
        System.out.println(converted);
    }
}
```

```java
interface Super03<T1, T2> {

    boolean test(T1 x, T2 y);
}

//////////////////////////////////////////////////////////////////////////
public class Test03 {

    public static void main(String[] args) {

        //匿名類別
        Super03<String, String> a = new Super03<String, String>() {
            @Override
            public boolean test(String x, String y) {
                return x.equals(y);
            }
        };
        System.out.println(a.test("abc", "abc")); //true
        //Lambda
        Super03<String, String> b = (x, y) -> x.equals(y);
        System.out.println(b.test("abc", "abc")); //true
        //方法引用-3. 類名 :: 實例方法名
        Super03<String, String> c = String::equals;
        System.out.println(c.test("abc", "abc")); //true
    }
}
```

建構函數(constructor)引用

建構函式和其他方法一樣是方法。它們有點特殊，它們是物件初始化方法。可以像其他方法引用一樣建立建構函式的方法引用。

```
class Person {

    public String firstName;
    public String lastName;

    public Person(String firstName, String lastName) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString() {
        return firstName + " " + lastName;
    }
}

@FunctionalInterface
interface PersonFactory {
    Person create(String firstName, String lastName);
}
////////////////////////////////////////////////////////////////////////////
public class Test04 {

    public static void main(String[] args) {
        //匿名類別
        PersonFactory b = new PersonFactory() {
            public Person create(String firstName, String lastName) {
                return new Person(firstName, lastName);
            }
        };
        Person p2 = b.create("Lai", "Yu-Sang");
        System.out.println(p2); //Lai Yu-Sang
        //Lambda
        PersonFactory c = (firstName, lastName) -> new Person(firstName, lastName);
        Person p3 = c.create("Lai", "Yu-Sang");
        System.out.println(p3); //Lai Yu-Sang
        //方法引用-4.建構函數
        PersonFactory d = Person::new;
        Person p4 = d.create("Lai", "Yu-Sang");
        System.out.println(p4); //Lai Yu-Sang
    }
}
```

5. Lambda 的有效範圍➔Lambda 是方法呼叫，不會產生$類別，所以會呼叫包含他的外圍類別的方法

```
// 匿名類別語法
class Hello1 {

    public Runnable r = new Runnable() {
        public void run() {
            // 這裡的this指的是匿名類別，而非Hello類別。
            System.out.println("-->1 " + this.toString()); //-->1 p05_Lambda作用範圍.Hello1$1@15db9742
            // 想要引用Hello類別需要Hello.this!!!
            System.out.println("-->2 " + Hello1.this.toString()); //-->2 Hello's custom toString()
        }
    };
    public String toString() {
        return "Hello's custom toString()";
    }
}
//////////////////////////////////////////////////////////////////
// Lambda語法
class Hello2 {

    public Runnable r = () -> { //會引用到 Hello2 的 toString()
        System.out.println(this); //Hello's custom toString()
        System.out.println(this.toString()); //Hello's custom toString()
    };

    public String toString() {
        return "Hello's custom toString()";
    }
}
//////////////////////////////////////////////
public class Test01 {

    public static void main(String[] args) {
        Hello1 h1 = new Hello1();
        h1.r.run();

        Hello2 h2 = new Hello2();
        h2.r.run();
    }
}
```

6. 訪問區域變數

<1>.匿名類別只能引用作用範圍外面的 final 的變數，在 Lambda 中對這個限制做了削弱，只需要是“等價 final”就可以，沒必要用 final 關鍵字來標識。我們可以直接在 Lambda 表達式中訪問外層的區域變數

<2>.Lambda 的運算，包含三個部分，一段程式碼，引數和自由變數的值，其中，自由變數是指那些不是引數且沒有被宣告的變數

➔含有自由變數的程式區塊被稱為”閉包”Lambda 運算式是可以捕捉到閉合作範圍中的變數值，被引用的變數的值不可被修改介面宣告應該是功能性的和無狀態的。如果將這樣的閉包傳遞給接受功能物件的內建函式，則可能導致崩潰或錯誤行為

```
public class Test01 {

    public static void main(String[] args) {
        //不需要是final的 , 但只要是 匿名類別或是 Lambda中有用到 就不可修改
        String message = "Howdy, world!";
        //匿名類別
        Runnable a = new Runnable() {
            @Override
            public void run() {
                System.out.println(message);
            }
        };
        //Lambda
        Runnable b = () -> System.out.println(message);
        b.run();
        //不可修改
        message = "change it";
    }
}
```

local variables referenced from an inner class must be final or effectively final
---
(Alt-Enter shows hints)

<3>.Lambda 運算式的方法體與巢狀程式碼有著相同的作用範圍，因此它也適用同樣的命名衝突和遮蔽規則，在 Lambda 運算式中不允許宣告一個與區域變數同名的引數或者區域變數

```
public class Test02 {

    public static void main(String[] args) {

        Path first = Paths.get("/usr/bin");
        //下面的first與上面的產生衝突。
        Comparator<String> comp = (first, second) -> Integer.compare(first.length(), second.length());

    }
}
```

variable first is already defined in method main(String[])

(Alt-Enter shows hints)

7. 訪問物件變數與靜態變數➔和區域變數(Local variables)不同的是，Lambda 內部對於物件變數以及靜態變數是既可讀又可寫。

```
interface Super01<F, T> {

    T convert(F from);
}

//////////////////////////////////////////////////////////////
public class Test01 {

    public static int num1; //靜態變數
    public int num2; //物件變數

    public void testScopes() {

        Super01<Integer, String> stringConverter = from -> {
            //可以修改 類別變數
            num1 = 72;
            //可以修改 物件變數
            num2 = 23;
            return String.valueOf(from);
        };
        System.out.println(stringConverter.convert(123));
    }

    public static void main(String[] args) {
        new Test01().testScopes();
    }
}
```

8. 訪問介面的 default(默認)方法

<1>.Google Guava

工具類 就是封裝平常用的方法，節省開發人員時間，提高工作效率。谷歌作為大公司，當然會從日常的工作中提取中很多高效率的方法出來。所以就誕生了 guava。Guava 工程包含了若干被 Google 的 Java 項目廣泛依賴 的核心庫，例如：

(1).集合 [collections]

(2).缓存 [caching] 快取

(3).原生類型支持 [primitives support]

(4).並發庫[concurrency libraries]

(5).通用注解 [common annotations]

(6).字符串處理 [string processing]

(7).I/O 等等。

<2>.JDK 1.8 API 包含了很多內建的函數式介面，在老 Java 中常用到的比如 Comparator 或者 Runnable 介面，這些介面都增加了 @FunctionalInterface 註解以便能用在 Lambda 上。Java 8 API 同樣還提供了很多全新的函數式介面來讓工作更加方便，有一些介面是來自 Google Guava 庫裡的

(1).Predicate 介面 : 只有一個參數，回傳 boolean 類型。該介面包含多種默認方法來將 Predicate 組合成其他複雜的羅輯(比如：與，或，非)

(2).Function 介面：接收一個任一類型的參數並且回傳一個任一類型的結果，並附帶了一些可以和其他函數組合的 default(默認)方法 (compose， andThen)

(3).BiFunction 介面：接收二個任一類型的參數並且回傳一個任一類型的結果

(4).BinaryOperator 介面：接收二個同類型的參數並且回傳一個同類型的結果

(5).UnaryOperator 介面：接收一個任一類型的參數並且回傳一個同類型的結果

(6).Supplier 介面：不接受任何參數，回傳一個任意類型的值

(7).Consumer 介面：接收一個任一類型的參數，沒有任何的回傳。

(8).Comparator 介面：是老 Java 中的經典介面， Java 8 在此之上添加了多種 default(默認)方法

(9).Optional 介面：不是函数是介面，這是個用來防止 NullPointerException 異常的輔助類型

(10).Stream 介面：表示能應用在一組元素上，一次執行的操作序列。Stream 操作分為中間操作或者最終操作兩種，最終操作回傳一特定類型的計算結果，而中間操作回傳 Stream 本身，這樣你就可以將多個操作依次串起來

Predicate 介面(斷言)

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);     ...
}
```

Predicate 介面只有一個參數，回傳 boolean 類型。該介面包含多種 default 方法將 Predicate 組合成其他複雜的邏輯
(比如：與，或，非)

```
public class Test0101_Predicate {

    public static void main(String[] args) {
//        Predicate1();
//        Predicate2();
//        Predicate3();
    }
```

```java
//Predicate==>判斷 , 回傳布林值
public static void Predicate1() {
    //匿名類別
    Predicate<Integer> a = new Predicate<Integer>() {
        @Override
        public boolean test(Integer t) {
            return t > 123;
        }
    };
    System.out.println(a.test(456)); //true
    //Lambda
    Predicate<Integer> b = t -> t > 123;
    System.out.println(b.test(456));  //true
}
```

```java
public static void Predicate2() {
    Predicate<Integer> c = t -> t > 10;
    Predicate<Integer> d = t -> t < 20;
    //not--->15 有 > 10 嗎
    System.out.println(c.negate().test(15));  //false
    // 15 有 > 10 and 15 有 < 20 嗎
    System.out.println(c.and(d).test(15)); //true
    //not ----> 15 有 > 10 and 15 有 < 20 嗎
    System.out.println(c.and(d).negate().test(15)); //false
}
```

```java
    public static void Predicate3() {
        Predicate<String> a = t -> t.length() > 10;
        Predicate<String> b = t -> t.contains("A");
        // "And" 的 長度 有 >10 or "And" 有包含 "A" 嗎
        System.out.println(a.or(b).test("And")); //true
    }
}
```

Function 介面

```
@FunctionalInterface
public interface Function<T， R> {
  R apply(T t);
}
```

Function 介面的 apply 方法收一個參數，回傳一個值，並附帶了一些可以和其他函數組合的 default 方法（compose，andThen）
就是一個方法，有種 c++裡面函數指標的感覺，一個指標變數可以指向一個方法，
並且可以把兩個方法組合起來使用(使用 compose 和 andThen) ，

```java
public class Test0201_Function {

    public static void main(String[] args) {
//        Function1();
//        Function2();
    }
```

```
    //Function==>傳一個值進去 , 消費 , 回傳一個值回來(可以不同型態)
    public static void Function1() {
        //匿名類別
        Function<String, Integer> a = new Function<String, Integer>() {
            @Override
            public Integer apply(String t) {
                return Integer.valueOf(t);
            }
        };
        System.out.println(a.apply("123")); //@123@
        //Lambda
        Function<String, Integer> b = t -> Integer.valueOf(t);
        System.out.println(b.apply("123")); //@123@
    }
```

```
    public static void Function2() {
        Function<Integer, Integer> a = t -> t * 2;
        Function<Integer, Integer> b = t -> t * t;
        //調用者 是 a , 參數 是 b
        System.out.println(a.apply(4)); //8
        System.out.println(b.apply(4)); //16
        //先4x4然後16x2, 先執行參數,再執行調用者
        System.out.println(a.compose(b).apply(4));  //32
        //先4x2,然後8x8, 先執行調用者,再執行參數。
        System.out.println(a.andThen(b).apply(4));  //64
    }
}
```

BiFunction 介面

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
  R apply(T t, U u);
}
```

BiFunction 介面,接收兩個不同型態參數,回傳一個不同型態的結果

```
public class Test0301_BiFunction {

    public static void main(String[] args) {
//        BiFunction1();
//        BiFunction2();
    }
```

```
    //BiFunction==>傳兩個值進去(可以不同型態) , 消費 , 回傳一個值回來(可以不同型態)
    public static void BiFunction1() {
        //匿名類別
        BiFunction<String, String, String> a = new BiFunction<String, String, String>() {
            public String apply(String t1, String t2) {
                return t1 + "==== " + t2;
            }
        };
        System.out.println(a.apply("賴玉珊", "早安")); //賴玉珊==== 早安
        //Lambda
        BiFunction<String, String, String> b = (t1, t2) -> t1 + "==== " + t2;
        System.out.println(b.apply("賴玉珊", "早安")); //賴玉珊==== 早安
    }
```

```
    public static void BiFunction2() {
        BiFunction<String, String, String> a = (t1, t2) -> t1 + "===" + t2;
        //調用者是 a , 參數是 b
        //先執行調用者 a 再執行參數 b
        //andThen表示將在apply方法後面執行
        Function<String, String> b = t -> t + " c";
        System.out.println(a.andThen(b).apply("a ", " b")); //a===b c
    }
}
```

BinaryOperator 介面

```
@FunctionalInterface
public interface BinaryOperator<T> {
  T apply(T t，T u);
}
```

BinaryOperator 介面繼承了 BiFunction，接受兩個相同型態的參數，回傳與參數相同型態的結果

```
public class Test0401_BinaryOperator {

    public static void main(String[] args) {
        // BinaryOperator1();
    }
    //BinaryOperator==>傳兩個值進去(都要同型態) ,消費 , 回傳一個值回來 (同型態)
    public static void BinaryOperator1() {
        //匿名類別
        BinaryOperator<String> a = new BinaryOperator<String>() {
            public String apply(String t1, String t2) {
                return t1 + "===" + t2;
            }
        };
        System.out.println(a.apply("賴玉珊", "想回家")); //賴玉珊===想回家
        //Lambda
        BinaryOperator<String> b = (t1, t2) -> t1 + "===" + t2;
        System.out.println(b.apply("賴玉珊", "想回家")); //賴玉珊===想回家
    }
```

UnaryOperator 介面

```
@FunctionalInterface
public interface UnaryOperator<T> {
  T apply(T t);
}
```

UnaryOperator 介面。繼承了 Function，接受一個參數並回傳相同型態的值

```
public class Test0501_UnaryOperator {

    public static void main(String[] args) {
//        UnaryOperator1();
//        UnaryOperator2();
    }
```

```
    //UnaryOperator==>傳一個值進去(同型態) , 消費 , 回傳一個值回來 (同型態)
    public static void UnaryOperator1() {
        //匿名類別
        UnaryOperator<String> a = new UnaryOperator<String>() {
            public String apply(String t1) {
                return t1 + "===" + "想回家";
            }
        };
        System.out.println(a.apply("賴玉珊")); //賴玉珊===想回家
        //Lambda
        UnaryOperator<String> b = (t1) -> t1 + "===" + "想回家";
        System.out.println(b.apply("賴玉珊")); //賴玉珊===想回家
    }
```

```
    public static void UnaryOperator2() {
        UnaryOperator<Integer> a = t -> t + 1;
        System.out.println(a.apply(10));// 11
    }
}
```

Supplier 介面

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

如果需要的行為是不接受任何引數，然後傳回值，那可以使用 Supplier 介面(供應商)

```
public class Test0601_Supplier {

    public static void main(String[] args) {
        // Supplier1();
    }
```

```
//Supper(供應商)==>提供值輸出(回傳)
public static void Supplier1() {
    //匿名類別
    Supplier<String> a = new Supplier<String>() {
        @Override
        public String get() {
            return " 想回家";
        }
    };
    System.out.println(a.get());
    //Lambda
    Supplier<String> b = () -> " 想回家";
    System.out.println(b.get());
}
```

```
    public static void Supplier2() {
        //直接回傳 20
        Supplier<Integer> c = () -> 20;
        System.out.println(c.get()); //20
    }
}
```

Consumer 介面

```
@FunctionalInterface
public interface Consumer <T> {
   void accept(T t );
}
```

如果需要的行為是接受一個引數，然後處理後不傳回值，就可以使用 Consumer 介面(消費者➔專門做輸出用的)
既然接受了引數但沒有回傳值，這行為就像 純粹消耗了引數，也就是命名為 Consumer 的原因，如果產出，就是以副作用(Side effect)形式呈現，像是改變某物件狀態，或者是進行了輸入輸出，例如，使用 System.out 的 println( )進行輸出：
Arrays.asList("Justin"， "Monica"， "Irene").forEach(System.out::println);

```
public class Test0701_Consumer {

    public static void main(String[] args) {
        //Consumer1();
        Consumer2();
    }
```

```
//Consumer(消費者)==>提供列印
    public static void Consumer1() {
        //匿名類別
        Consumer<String> a = new Consumer<String>() {
            @Override
            public void accept(String t) {
                System.out.println(t);
            }
        };
        a.accept("Hello");
        //Lambda
        Consumer<String> b = t -> System.out.println(t);
        b.accept("Hello"); //Hello
    }
```

```
    public static void Consumer2() {
        Consumer<String> a = t -> System.out.print(t);
        Consumer<String> b = t -> System.out.println(" after bbb");
        a.andThen(b).accept("aaa"); //"aaa after bbb"
    }
}
```

Comparator 介面

```
public class Test0801_Comparator {

    public static void main(String[] args) {
        Comparator1();
    }
    public static void Comparator1() {

        List<String> users = Arrays.asList("John","Jane","Alex");

        System.out.println("Before sort:");
        System.out.println(users); //[John, Jane, Alex]
        //匿名類別
//        Collections.sort(users, new Comparator<String>() {
//            @Override
//            public int compare(String t1, String t2) {
//                return t2.compareTo(t1);
//            }
//        });
//        System.out.println("\nAfter sort:");
//        System.out.println(users); //[John, Jane, Alex]
        //Lambda
        Collections.sort(users, (t1, t2) -> t2.compareTo(t1));
        System.out.println("\nAfter sort:");
        System.out.println(users); //[John, Jane, Alex]
    }
}
```

Optional 介面

Optional 是值的容器，只有兩種狀態，不是有值就是沒值。目的是做為 null 的替代方案。Optional 提供工廠方法，將你輸入的值產生為 Optional 物件，這時 Optional 物件即為該值的容器，若要取回該值，必須使用 get() 方法。

1. 將值轉為 Optional 的方法
   <1>.of()：接受非 null 的值並回傳 Optional 物件。
   <2>.ofNullable()：可以接受 null 的值，回傳 Optional 物件。
2. 取得放在 Optional 物件內的值的方法
   <1>.get()：如果值存在就回傳這個值，否則就丟出 NoSuchElementException。
   <2>.orElse(T other)：如果值存在就回傳這個值，否則回傳 other 。
   <3>.orElseGet(Supplier<? extends T> other)：如果值存在就回傳這個值，否則就呼叫 other 並回傳它的結果。
   <4>.orElseThrow(Supplier<? extends X> exceptionSupplier)：如果值存在就回傳這個值，否則就丟出由 exceptionSupplier 建立的例外。
3. 檢查值是否存在
   <1>.boolean isPresent()：如果值存在，回傳 true；不存在則回傳 false。
   <2>.void ifPresent(Consumer<? super T> consumer)：如果值存在，呼叫指定的 consumer 物件並將值傳給它處理；不存在則什麼都不做。

```java
public class Test0901_Optional {

    public static void main(String[] args) {
        // Optional1();
        //Optional2();
        // Optional3();
        // Optional4();
        // Optional5();
        // Optional6();
        // Optional7();
        // Optional8();
       // Optional9();
      //  Optional10();
          Optional11();
    }
```

```java
class MyEntity {

    int id;
    String name;

    String getName() {
        return name;
    }
}
```

```java
//傳統判斷 null 方式
public static void Optional1() {

    MyEntity myEntity = null;
    //System.out.println(myEntity.getName()); //當掉 , 大魔王
    if (myEntity != null) {
        System.out.println(myEntity.getName());
    } else {
        System.out.println("ERROR");
    }
}
```

```java
    //值不是null的情況
    public static void Optional2() {
        String name = "Tony";
        Optional<String> a = Optional.of(name);
        System.out.println(a.get());//Tony
    }
```

```java
//值變成 null 了
    public static void Optional3() {
        String name = null;
        Optional<String> a = Optional.of(name);
        System.out.println(a.get());//NullPointerException大魔王
    }
```

```java
//改用ofNullable()
public static void Optional4() {
    String name = null;
    Optional<String> a = Optional.ofNullable(name);
    System.out.println(a.get());//NoSuchElementException小魔王
}
```

```java
//使用前先檢查
public static void Optional5() {
    String name = null;
    Optional<String> a = Optional.ofNullable(name);
    if (a.isPresent()) { //一定要用 Optional.ofNullable 不然會當
        System.out.println(a.get());
    } else {
        System.out.println("Name is null"); //"Name is null"
    }
}
```

```
//使用 orElse 就可以丟掉 if 了
public static void Optional6() {
    String name = null;
    Optional<String> a = Optional.ofNullable(name);
    System.out.println(a.orElse("Name is null."));//"Name is null"
}
```

```
//包裝值的另一個方式
public static void Optional7() {
    String name = null;
    Optional<String> a;
    if (name == null) {
        a = Optional.empty();
    } else {
        a = Optional.of(name);
    }
    System.out.println(a.get());//NoSuchElementException
}
```

```
//使用 orElseGet
public static void Optional8() {
    String name = null;
    Optional<String> a = Optional.ofNullable(name);
    System.out.println(a.orElseGet(() -> "WHAT! null!"));//"WHAT! null!"
}
```

```
//使用 OrElseThrow 丟出例外
public static void Optional9() {
    class MyException extends Exception {

        public MyException(String message) {
            super(message);
        }
    }
    String name = null;
    Optional<String> a = Optional.ofNullable(name);
    try {
        System.out.println(a.orElseThrow(() -> new MyException("WHAT! NULL!")));
    } catch (MyException e) {
        System.out.println("My Exception! " + e.getMessage());//"WHAT! NULL!"
    }
}
```

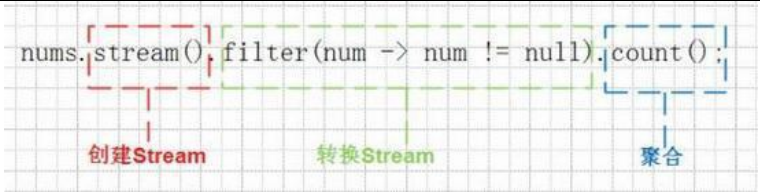
```
    public static void Optional10() {
        Integer value1 = null;
        Integer value2 = 123;
        Optional<Integer> a = Optional.ofNullable(value1);
        Optional<Integer> b = Optional.of(value2);
        System.out.println("第一個參數值存在: " + a.isPresent()); //false
        System.out.println("第二個參數值存在: " + b.isPresent()); //true
    }
}
```

```
public static void Optional11() {
    List<String> str = Arrays.asList("my", "pen", "is", "your", "pen");
    Predicate<String> test = s -> {
        int i = 0;
        boolean result = s.contains("pen");
        System.out.print(i++ + " : ");
        return result;
    };
    str.stream()
            .filter(test)
            .findFirst()
            .ifPresent(System.out::print);
}
```

Stream 介面

1. Java 8 中 java.util.Stream (流) ，一個函數式語言+多核時代綜合影響的產物，是對集合（Collection）功能的增强，它專注於對集合進行各種非常便利、高效的聚合操作（aggregate operation），或者大批量數據操作 (bulk data operation)。借助於同樣新出現的 Lambda 表達式，極大的提高程式效率和程序可讀性
2. Stream(流) 不是集合元素，它不是數據結構並不保存數據，它是有關算法和計算的，它更像一個高級版本的 Iterator(迭代器)，單向，不可往復，數據只能遍歷一次，遍歷過一次後即用盡了，就好比流水從面前流過，一去不復返，原始版本的 Iterator(迭代器)，用户只能顯式地一個一個遍歷元素並對其執行某些操作；高級版本的 Stream，用户只要給出需要對其包含的元素執行什麼操作，比如 “過濾掉長度大於 10 的字串” 、 “獲取每個字串的首字母” 等，Stream 會隱式地在內部進行遍歷，做出相應的數據轉換。
3. Stream 可使用串行和併行兩種模式進行匯聚操作，開發模式能够充分利用多核處理器的優勢，使用 fork/join 並行方式來拆分任務和加速處理過程，通常編寫併行代碼很難而且容易出錯，但使用 Stream API 無需編寫一行多線程的代碼，就可以很方便地寫出高性能的併行程序。Iterator(迭代器)只能命令式地、串行方式操作。當使用串行方式去遍歷時，每個 item 讀完後再讀下一個 item。而使用併行方式去遍歷，數據會被分成多個段，其中每一個都在不同的線程中處理，然後將結果一起輸出。Java 的併行 API 演變歷程基本如下
   <1>.1.0-1.4 中的 java.lang.Thread
   <2>.5.0 中的 java.util.concurrent
   <3>.6.0 中的 Phasers 等
   <4>.7.0 中的 Fork/Join 框架
   <5>.8.0 中的 Lambda
4. 流的構成
   <1>.創建 Stream；
   <2>.轉換 Stream，每次轉換原有 Stream 對象不改變，回傳一個新的 Stream 對象（**可以有多次轉換**）➔中間操作
   <3>.對 Stream 進行聚合操作，獲取想要的結果➔尾端操作

```
int sum = widgets.stream() //創建
.filter(w -> w.getColor() == RED) //轉換
.mapToInt(w -> w.getWeight())//轉換
.sum(); //結果
```



紅色框中的語句是一個 Stream 的生命開始的地方，負責創建一個 Stream 實例；綠色框中的語句是賦予 Stream 靈魂的地方，把一個 Stream 轉換成另外一個 Stream，紅框的語句生成的是一個包含所有 nums 變量的 Stream，進過綠框的 filter 方法以後，重新生成了一個過濾掉原 nums 列表所有 null 以後的 Stream；藍色框中的語句是豐收的地方，把 Stream 的裡面包含的內容按照某種算法來匯聚成一個值，例子中是獲取 Stream 中包含的元素個數

5. 最常用的創建 Stream 有兩種途徑：

<1>.通過 Stream 介面的靜態工廠方法

(1).java.util.stream.IntStream.range( )

(2).java.nio.file.Files.walk( )

<2>.通過 Collection 介面的 default 方法

(1).集合.stream()

(2).集合.parallelStream()

(3).Arrays.stream(T array) or Stream.of(T array)

<3>.BufferedReader

(1).java.io.BufferedReader.lines()

6. 流的操作類型分為兩種：

<1>.Intermediate 中間操作：一個流可以後面跟隨零個或多個 intermediate 操作。其目的主要是打開流，做出某種程度的數據映射/過濾，然後回傳一個新的流，交給下一個操作使用。這類操作都是惰性化的（lazy），就是說，僅僅調用到這類方法，並沒有真正開始流的遍歷。

map (mapToInt， flatMap 等)、 filter、 distinct、 sorted、 peek、 limit、 skip、 parallel、 sequential、 unordered

<2>.Terminal 終點操作：一個流只能有一個 terminal 操作，當這個操作執行後，流就被使用 “光” 了，無法再被操作。所以這必定是流的最後一個操作。Terminal 操作的執行，才會真正開始流的遍歷，並且會生成一個結果，或者一個 side effect (副作用→改變某物件狀態)。

forEach、 forEachOrdered、 toArray、 reduce、 collect、 min、 max、 count、 anyMatch、 allMatch、 noneMatch、 findFirst、 findAny、 iterator

<3>.short-circuiting 短路操作：如果由無限的流，轉到有限的流，不管是中間操作還是終點操作都叫做短路

(1).對於一個 intermediate 操作，如果它接受的是一個無限大的 Stream，但回傳一個有限的新 Stream。

(2).對於一個 terminal 操作，如果它接受的是一個無限大的 Stream，但能在有限的時間計算出結果。

由無限流轉成有限流的中間操作有 limit

由無限流轉成有限流的終點操作有 anyMatch、 allMatch、 noneMatch、 findFirst、 findAny

10. Stream 注意事項

<1>.在對於一個 Stream 進行多次轉換操作 (Intermediate 操作)，這樣時間複雜度就是 N（轉換次數）個 for 循環裡把所有操作都做掉的總和嗎？其實不是這樣的，轉換操作都是 lazy 的，多個轉換操作只會在 Terminal 操作的時候融合起來，一次循環完成。我們可以這樣簡單的理解，Stream 裡有個操作函數的集合，每次轉換操作就是把轉換函數放入這個集合中，在 Terminal 操作的時候循環 Stream 對應的集合，然後對每個元素執行所有的函數

<2>.流的使用➔Stream 的使用就是實現一個 filter-map-reduce 過程，產生一個最終結果，或者導致一個副作用（side effect）。

(1).每個 stream 方法回傳的值有兩種，並且行為也不同。

a. 當 stream 的方法回傳的值是 Stream 型別時，並不會實際去執行運算，中間串接的永遠是回傳 Stream 型別的方法，只有最後一個才是串接傳回某個非 Stream 型別的值的方法。

b. 當 stream 的方法回傳除了 Stream 型別以外的值時(包含 void)，才真正執行運算。在 API 文件中，屬於終止操作的方法，會有「 This is a terminal operation. 」這樣的標示，表示這個方法只能串接在最後一個。

(2).Stream 和 Collection 的區別是，Collection 是一種靜態的資料結構，使用的是記憶體空間，而 Stream 則是運算，使用的是 CPU

流的構造與轉換➔幾種常用

```
Stream stream
String [] strArray = {"a"， "b"， "c"};
List<String> list = Arrays.asList(strArray);
// 1. Individual values
stream = Stream.of("a"， "b"， "c");
// 2. Array
stream = Stream.of(strArray);
stream = Arrays.stream(strArray);
// 3. Collection
stream = list.stream();
```

需要注意的是，對於基本數值型，目前有三種對應的包裝類型 Stream：

IntStream、LongStream、DoubleStream。當然我們也可以用 Stream<Integer>、Stream<Long> >、Stream<Double>，但是 boxing 和 unboxing 會很耗時，所以特別為這三種基本數值型提供了對應的 Stream。

數值流的構造

```
IntStream.of(new int[]{1， 2， 3}).forEach(System.out::println);
IntStream.range(1， 3).forEach(System.out::println);
```

流轉換為其它數據结構

```
// 1. Array
String[] strArray1 = stream.toArray(String[]::new);
// 2. Collection
List<String> list1 = stream.collect(Collectors.toList());
Set set1 = stream.collect(Collectors.toSet());
// 3. String
String str = stream.collect(Collectors.joining()).toString();
```

forEach

forEach 方法接收一個 Lambda 表達式，然後在 Stream 的每一個元素上執行該表達式

```
public class Test1001_forEach {

    public static void main(String[] args) {
        //forEach1();
        //forEach2();
        // forEach3();
        //forEach4();
        // forEach5();
        forEach6();
       // forEach7();
      //  forEach8();
    }
```

```
public static void forEach1() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    //快捷迴圈
    for (String s : list) {
        System.out.println(s);
    }
}
```

```
public static void forEach2() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    // 使用Java 8 forEach() 搭配 匿名類別
    list.forEach(new Consumer<String>() {
        @Override
        public void accept(String s) {
            System.out.println(s);
        }
    });
}
```

```
public static void forEach3() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    // 使用Java 8 forEach() 搭配 Lambda語法
    list.forEach(s -> System.out.println(s));
}
```

```java
public static void forEach4() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    //方法引用
    list.forEach(System.out::println); //"one", "two", "three", "four"
}
```

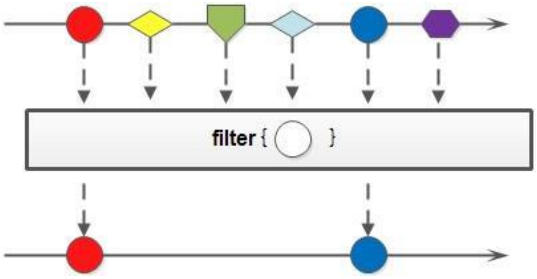
```java
public static void forEach5() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    list.stream().forEach(s -> System.out.println(s));
}
```

```java
public static void forEach6() {
    List<String> list = Arrays.asList("one", "two", "three", "four");
    list.stream().forEach(System.out::println);
}
```

```java
public static void forEach7() {
    Stream.of("one", "two", "three", "four").forEach(System.out::println);
}
```

filter

filter 方法主要用來做資料的檢查。當你要用迴圈來檢查集合中的每個元素是否符合你的期望時，主要用來過濾 Stream 中的元素




```java
public class Test1002_filter {

    public static void main(String[] args) {
//        filter1();
//        filter2();
//        filter3();
//        filter4();
    }
```

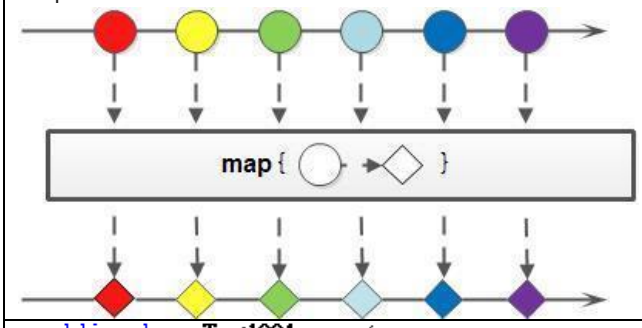
```java
public static void filter1() {
    List<String> list = Arrays.asList("nanjing", "beijing", "nantong", "haimen", "shangrao");
    List<String> names = new ArrayList<>();
    //傳統方法
    for (String name : list) {
        if (name.length() > 7) {
            names.add(name);
        }
    }
    System.out.println(names);//shangrao
}
```

```java
//Stream
public static void filter2() {
    List<String> list = Arrays.asList("nanjing", "beijing", "nantong", "haimen", "shangrao");
    list.stream().filter(x -> x.length() > 7)
            .forEach(System.out::println); //shangrao
}
```

```
    public static void filter3() {
    //Stream
        List<String> names = Stream.of("nanjing", "beijing", "nantong", "haimen", "shangrao")
                .filter(x -> x.length() > 7)
                .collect(Collectors.toList());
        System.out.println(names);//[shangrao]
    }
}
```

map

map 方法主要用來將 Stream 中的元素轉換成其他的值




```
public class Test1004_map {

    public static void main(String[] args) {
//        map1();
//        map2();
//        map3();
    }
```

```
public static void map1() {
    List<String> list = Arrays.asList("tony", "tom", "john");
    //傳統方法
    List<String> names = new ArrayList<>();
    for (String name : list) {
        String upperName = name.toUpperCase();
        names.add(upperName);
    }
    System.out.println(names); //[TONY, TOM, JOHN]
}
```

```
public static void map2() {
    List<String> list = Arrays.asList("tony", "tom", "john");
    list.stream()
            //.map(name -> name.toUpperCase())
            .map(String::toUpperCase) //方法引用
            .forEach(System.out::println);  //TONY , TOM , JOHN
}
```

```
public static void map3() {
    //Stream
    List<String> names = Stream.of("tony", "tom", "john")
            .map(String::toUpperCase) //方法引用
            .collect(Collectors.toList());
    System.out.println(names); //[TONY, TOM, JOHN]
}
```

```
    public static void map4() {
        List<Integer> list = Arrays.asList(10, 20, 30);
        list.stream().map(x -> x * 10)
                .forEach(System.out::println);  //方法引用100,200,300
    }
}
```

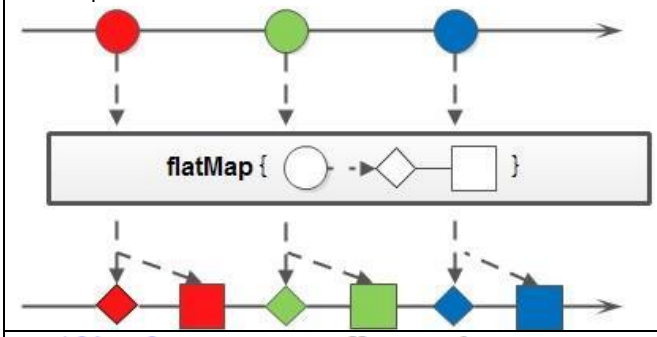
mapToInt

mapToInt 方法是將 Stream 中的元素轉換成 int 類型的

```
public class Test1007_mapToInt {

    public static void main(String[] args) {
        //mapToInt1();
    }
    public static void mapToInt1() {
        List<Integer> list = Arrays.asList(1, 2, 3);
        list.stream()
                .mapToInt(t -> t * 10)
                .forEach(System.out::println); //10,20,30
    }
}
```

flatMap 方法把生成的多個集合“拍扁”成為一個集合




```
public class Test1008_flatMap {

    public static void main(String[] args) {
        //flatMap1();
        //flatMap2();
        //flatMap3();
    }
```

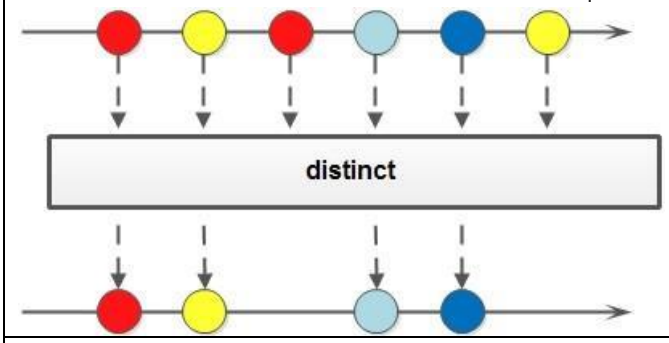
```
public static void flatMap1() {
    Stream.of(Arrays.asList(1, 2), Arrays.asList(3, 4))
            //.flatMap(num -> num.stream()) //傳集合進來變成流
            .flatMap(List::stream) //方法引用
            .forEach(System.out::println); // 1,2,3,4
}
```

```
public static void flatMap2() {
    Stream.of(Arrays.asList("Tony", "Tom", "John"),
            Arrays.asList("Amy", "Emma", "Iris"))
            //.flatMap(names -> names.stream()) //傳集合進來變成流
            .flatMap(List::stream) //方法引用
            .forEach(System.out::println); //[Tony, Tom, John, Amy, Emma, Iris]
}
```

```
    public static void flatMap3() {
        List<String> list = Stream.of(Arrays.asList("Tony", "Tom", "John"),
                Arrays.asList("Amy", "Emma", "Iris"))
                //.flatMap(names -> names.stream()) //傳集合進來變成流
                .flatMap(List::stream) //方法引用
                .collect(Collectors.toList()); //[Tony, Tom, John, Amy, Emma, Iris]
        System.out.println(list);
    }
}
```

distinct

distinct 方法是將 Stream 中的相等元素去除，通過 equals 方法來判斷元素是否相等




```
public class Test1009_distinct {

    public static void main(String[] args) {
        // distinct1();
    }
```

```
public static void distinct() {
    Stream.of(1, 2, 3, 4, 2, 3, 1)
            .distinct()
            .forEach(System.out::println); //1,2,3,4
}
```

sorted

sorted 方法是將 Stream 中的元素進行排序，它內部的元素必現實現 Comparable

```
public class Test1010_sorted {

    public static void main(String[] args) {
//      sorted1();
//      sorted2();
    }
```

```
public static void sorted1() {
    Stream.of("nanjing", "beijing", "nantong", "shangrao")
            .sorted()
            .forEach(System.out::println);//beijing , nanjing , nantong , shangrao
}
```

```
    public static void sorted2() {
        //ASC
        List<Integer> sortedAsc = Stream.of(120, 24, 59, 63, 11, 74)
                .sorted()
                .collect(Collectors.toList());
        System.out.println("sorted asc: "+sortedAsc);// [11, 24, 59, 63, 74, 120]

        //DESC
        List<Integer> sortedDesc = Stream.of(120, 24, 59, 63, 11, 74)
                .sorted((n1, n2) -> n2.compareTo(n1))
                .collect(Collectors.toList());
        System.out.println("sorted desc: " + sortedDesc);//[120, 74, 63, 59, 24, 11]
    }
}
```

reduce

reduce 方法是一個聚合方法，是將 Stream 中的元素進行某種操作，得到一個元素

匯聚操作（也稱為摺疊）接受一個元素序列為輸入，反覆使用某個合併操作，把序列中的元素合併成一個匯總的結果。比如查找一個數字列表的總和或者最大值，或者把這些數字累積成一個 List 對象

```
public class Test1011_reduce {

    public static void main(String[] args) {
        // reduce1();
        // reduce2();
    }
```

```
public static void reduce1() {
    //有初始值
    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, (sum, item) -> sum + item)); //20
    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, Integer::sum)); //20

    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, (min, item) -> Math.min(min, item))); //1
    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, Integer::min)); //1

    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, (max, item) -> Math.max(max, item))); //4
    System.out.println(Stream.of(1, 2, 3, 4)
            .reduce(10, Integer::max)); //4
}
```

```
    public static void reduce2() {
        //沒有初始值 , ifPresent 接收一個 Comsumer
        Stream.of(1, 2, 3, 4)
                .reduce((sum, item) -> sum + item)
                .ifPresent(System.out::println); //10 假如有值就展示
        System.out.println(Stream.of(1, 2, 3, 4)
                .reduce(Integer::sum).orElse(0));//10
        System.out.println(Stream.of(1, 2, 3, 4)
                .reduce(Integer::min).orElse(0)); //1
        System.out.println(Stream.of(1, 2, 3, 4)
                .reduce(Integer::max).orElse(0)); //4
    }
}
```

collect

collect 方法是將 Stream 的通過一個收集器，轉換成一個具體的值，是一個聚合方法 ，必須指定收集的型式，這裡指定為 toList。產生的結果和 Arrays.asList 的結果相同 ，可以以相同的方式取得 Set 集合，Set 集合會去除重覆

```
public class Test1012_collect {

    public static void main(String[] args) {
        //collect1();
        //collect2();
    }
```

```
public static void collect1() {
    List<String> name1 = Stream.of("Tony", "Tony", "Tony", "Tom", "Jonn")
            .map(String::toUpperCase)
            .collect(Collectors.toList());
    System.out.println(name1);  //[TONY, TONY, TONY, TOM, JONN]
}
```

```
    public static void collect2() {
        Set<String> names3 = Stream.of("Tony", "Tony", "Tony", "Tom", "Jonn")
                .map(String::toUpperCase)
                .collect(Collectors.toSet());
        System.out.println(names3);  //[TONY, TOM, JONN]
    }
}
```

min，max

min，max 方法是獲取 Stream 中最小或最大的元素的值，需要傳遞一個 Comparator 比較器參數，是聚合方法回傳值是 Optional<t>

```
public class Test1013_max {

    public static void main(String[] args) {
//        max_min1();
//        max_min2();
    }
```

```
public static void max_min1() {
    int max = Stream.of(120, 24, 59, 63, 11, 74)
            //.max(Comparator.comparing(n -> n))
            .max((n,m)->n.compareTo(m))
            .get();
    System.out.println("max: " + max); //max: 120

    int min = Stream.of(120, 24, 59, 63, 11, 74)
            //.min(Comparator.comparing(n -> n))
            .min((n,m)->n.compareTo(m))
            .get();
    System.out.println("min: " + min);  //min: 11
}
```

```
    public static void max_min2() {
        String max = Stream.of("加利福尼亞州", "喬治亞州", "康涅狄格州")
                .max(Comparator.comparing(String::length))
                .get();
        System.out.println("max: " + max); //"加利福尼亞州"
        String min = Stream.of("加利福尼亞州", "喬治亞州", "康涅狄格州")
                .min(Comparator.comparing(String::length))
                .get();
        System.out.println("min: " + min); //"喬治亞州"
    }
}
```

count

count 方法是獲取 Stream 中元素的數量，是聚合方法

```
public class Test1014_count {

    public static void main(String[] args) {
//        count1();
        count2();
    }
```

```
public static void count1() {
    long count = Stream.of(1, 2, 3).count();
    System.out.println("Count: " + count); //Count: 3
}
```

```
    public static void count2() {
        long count = Stream.of("Tony", "Tom", "John", "Andy")
                .filter(name -> name.startsWith("T"))
                .count();
        System.out.println("Count: " + count); //Count: 2

    }
}
```

anyMatch

方法用來判斷 Stream 中是否存在指定條件的元素

```
public class Test1015_anyMatch {

    public static void main(String[] args) {
        anyMatch1();
    }
    public static void anyMatch1() {
        System.out.println(Stream.of(1, 2, 3).anyMatch(t -> t > 2)); //true
        System.out.println(Stream.of(1, 2, 3).anyMatch(t -> t > 3)); //false
    }
}
```

allMatch

allMatch 方法用來判斷 Stream 中的元素是否都滿足指定的條件

```
public class Test1016_allMatch {

    public static void main(String[] args) {
        allMatch1();
    }
    public static void allMatch1() {
        System.out.println(Stream.of(1, 2, 3).allMatch(t -> t > 2)); //false
        System.out.println(Stream.of(1, 2, 3).allMatch(t -> t > 0)); //true
    }
}
```

noneMatch

noneMatch 方法用來判斷 Stream 中是否所有元素都不滿足指定條件

```
public class Test1017_noneMatch {

    public static void main(String[] args) {
        // noneMatch1();
    }
    public static void noneMatch1() {
        System.out.println(Stream.of(1, 2, 3).noneMatch(t -> t > 2)); //false
        // 全都不符合 > 3
        System.out.println(Stream.of(1, 2, 3).noneMatch(t -> t > 3)); //true
    }
}
```

findFirst

findFirst 方法回傳 Stream 中的第一個元素，回傳的類型為 Optional

```
public class Test1018_findFirst {

    public static void main(String[] args) {
        // findFirst1();
    }
    public static void findFirst1() {
        System.out.println(Stream.of(1, 2, 3).findFirst().get());//1
    }
}
```

peek

peek 方法接受一個實作了 Consumer 介面的類別，peek 不是终结操作，它生成一個包含原 Stream 的所有元素的新 Stream ，同時會提供一個消費函數（Consumer 實例），新 Stream 每個元素被消費的時候都會執行給定的消費函數

```
public class Test1019_peek {

    public static void main(String[] args) {
//        peek1();
//        peek2();
//        peek3();
//        peek4();
//        peek5();
    }
```

```
public static void peek1() {
    Stream.of(0, 2, 4, 6, 8, 10)
            .peek(System.out::println); //不會有任何輸出 , 少了終端操作
}
```

```
public static void peek2() {
    Stream.of(0, 2, 4, 6, 8, 10)
            .peek(System.out::println)
            .count(); //輸出 0 2 4 6 8 10
}
```

```
public static void peek3() {
    List<Integer> list = Stream.of(0, 2, 4, 6, 8, 10)
            .peek(System.out::println)
            .collect(Collectors.toList());
    System.out.println(list);
}
```

```
public static void peek4() {
    Stream.of("one", "two", "three", "four")
            .filter(e -> e.length() > 3)
            .peek(e -> System.out.println("Filtered value: " + e))
            .map(String::toUpperCase)
            .peek(e -> System.out.println("Mapped value: " + e))
            .count();
}
```

```
Filtered value: three
Mapped value: THREE
Filtered value: four
Mapped value: FOUR
```

```
    public static void peek5() {
        Stream.of("one", "two", "three", "four")
                .map(String::toUpperCase)
                .peek(e -> System.out.println("Mapped value: " + e))
                .filter(e -> e.length() > 3)
                .peek(e -> System.out.println("Filtered value: " + e))
                .count();
    }
}
```

```
Mapped value: ONE
Mapped value: TWO
Mapped value: THREE
Filtered value: THREE
Mapped value: FOUR
Filtered value: FOUR
```

parallel

list.stream().parallel()，list.parallelStream()➔都可以

Stream 提供兩種執行運算的方式，一種是循序運算，只會在單一個執行緒上運算，另一種並行運算則是在多個執行緒上同時運算，並行運算的方式可以充份利用 CPU 多核多執行緒的特點，因此執行速度會比較快

```
public class Test1020_parallel {

    public static void main(String[] args) {
        //parallel1();
    }
    public static void parallel1() {
        Stream.of("John", "Mike", "Ryan", "Donald", "Matthew")
                .parallel()
                .forEach(System.out::println); //Ryan , Mike , Matthew , John , Donald
    }
}
```

```
    public static void parallel2() {
        List<String> list = Arrays.asList("John", "Mike", "Ryan", "Donald", "Matthew");
        list.parallelStream()
                .forEach(System.out::println); //Ryan , Mike , Matthew , John , Donald
    }
}
```

stream 順序輸出，而 parallelStream 無序輸出；parallelStream 執行耗時是 stream 的五分之一。

parallelStream 獲得的相對較好的執行性能，那 parallelStream 背後到底是什麼呢？

要深入了解 parallelStream，首先要弄明白 ForkJoin 框架和 ForkJoinPool。ForkJoin 框架是 java7 中提供的並行執行框架，他的策略是分而治之。就是把一個大的任務切分成很多小的子任務，子任務執行完畢後，再把結果合併起來。

```
public class Test1021_parallelStream {

    public static void main(String[] args) {
        // parallelStream1();
    }
    public static void parallelStream1() {
        //結果 658374921
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
        numbers.parallelStream()
                .forEach(System.out::print);
    }
}
```

9. Date API

<1>.java8 以前在 java.util 包中提供了

(1).java.util.Date➔遠從 JDK 1.0 就已存在的古老 API，如果你只是要取得日期時間作些顯示，也許還沒有問題，如果你打算以它作為基礎進行日期時間，那就很可能落人陷阱，這個類別的大部份函式都已經廢棄了，Date 的函式已經被廢棄了 ， 但仍可以使用 getTime 和 setTime 函式，可以用這個類別連接 Calendar 和 DateFormat 這兩個類別，一個 Date 的實體是表示不可變的日期和時間，精準度直達毫秒

雖然叫作 Date，然而 **Date 實例真正代表的並不是日期**，如果你認真去解讀它，**Date 實例真正代表的概念，最接近的應該是特定的瞬時 (a specific instant in time)，時間精度是毫秒**，例如 ... 1375430498832 代表從 "the epoch"(時代)，也就是 UTC (Temps Universel Coordonné，簡稱 **UTC**) 最主要的世界時間標準，其以原子時秒長為基礎，在時刻上儘量接近於格林威治標準時間，時間 1970 年 1 月 1 日零時零分零毫秒至今經過 1375430498832 毫秒數的特定瞬時

(2).java.util.Calendar➔這個類別提供大量的函式，用來協助你轉換和操作日期及時間，假設你想要對指定的日期加一個月，或找出 3000 年的 1 月 1 號是星期幾，使用 Calendar 這個類別就可以大大減輕你的負擔

(3).java.text.DateFormat➔這個類別是用來格式化日期的表示字串，不僅僅只是 "01/01/70" 或 "January 1，1970" 這幾種格式 ，而是包含許多其他地域的各種日期格式

(4).java.util.Locale➔這個類別是和 DateFormat 及 NumberFormat 一起使用來為特定的地域格式化 日期，數字 及 貨幣

使用 Locale 的實體來客製化成特定地域的輸出

如果想要在程式內表示基本的義大利語 ， 所需的只是語言碼 ， 但想要表示瑞士所使用的義大利語 ， 將需要指定 country 引數為瑞士 但 language 引數仍然為義大利

Locale(String language)

Locale(String language ， String country)

Locale locPT=new Locale("it")

Locale locBR=new Locale("it" ，" CH")

```
public class Test00_Date1 {

    public static void main(String[] args) {

//        Date方法1();
//        Date方法2();
//        Date方法3();
//        Date_加1小時();
//        Calendar_加1小時();
//        Calendar計算();
//        DateFormat方法();
//        Locale方法();
    }
```

```
public static void Date方法1() {
    Date now = new Date(1000000000000L);  //1970/1/1
    System.out.println("now date " + now.toString()); //now date Sun Sep 09 09:46:40 CST 2001
    // 1000=1秒
}

public static void Date方法2() {
    //January 1, 1970 00:00:00 GMT.
    Date date = new Date(1000);
    System.out.println(date); //Thu Jan 01 08:00:01 CST 1970 時間多 8 小時 因為位於 東 8 時區
}

public static void Date方法3() {
    Date now = new Date();  //1970/1/1
    System.out.println("now date " + now.toString()); //now date Sat Oct 26 21:40:34 CST 2019
    // 1000=1秒
}
```

```java
public static void Date_加1小時() {
    Date now = new Date();
    System.out.println("今天是 : " + now.toString());  //今天是 : Thu Feb 09 11:35:19 CST 2017

    now.setTime(now.getTime() + 3600000); // 3600000 millis / hour
    System.out.println("加 1個小時 是 " + now.toString());  //加 1個小時 是 Thu Feb 09 12:35:19 CST 2017
}

public static void Calendar_加1小時() {

    Date now1 = new Date();
    System.out.println("今天是 :" + now1.toString()); //今天是 :Thu Feb 09 11:33:47 CST 2017
    Calendar c = Calendar.getInstance();
    c.setTime(now1);
    c.add(Calendar.HOUR, 1);
    Date now2 = c.getTime();
    System.out.println("加 1個小時 是 : " + now2.toString()); //加 1個小時 是 : Thu Feb 09 12:33:47 CST 2017
}

public static void Calendar計算() {
    Date now1 = new Date();
    Date now2;
    System.out.println("今天是 :" + now1.toString()); //今天是 :Thu Feb 09 11:32:24 CST 2017
    Calendar c = Calendar.getInstance();

    c.setTime(now1);
    c.add(Calendar.YEAR, 2);
    now2 = c.getTime();
    System.out.println("加 2 年 新日期是 : " + now2.toString()); //加 2 年 新日期是 : Sat Feb 09 11:32:24 CST 2019

    c.setTime(now1);
    c.add(Calendar.MONTH, 1);
    now2 = c.getTime();
    System.out.println("加1個月後新日期是 : " + now2.toString()); //加1個月後新日期是 : Thu Mar 09 11:32:24 CST 2017

    c.setTime(now1);
    c.add(Calendar.HOUR, -4);
    now2 = c.getTime();
    System.out.println("減 4 小時 新日期是 : " + now2.toString()); //減 4 小時 新日期是 : Thu Feb 09 07:32:24 CST 2017
}
```

```java
public static void DateFormat方法() {
    Date now = new Date();
    DateFormat[] dfa = new DateFormat[7];

    dfa[0] = DateFormat.getInstance();
    dfa[1] = DateFormat.getDateInstance();
    dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
    dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
    dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
    dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);
    dfa[6] = DateFormat.getDateInstance(DateFormat.DEFAULT);
    for (DateFormat df : dfa) {
        System.out.println(df.format(now));
    }
}
```

```
    public static void Locale方法() {

        Calendar c = Calendar.getInstance();
        c.set(2011, 9, 15); // Sat Oct 15 09:30:30 CST 2011 // (month is 0-based)
        Date d2 = c.getTime();
        System.out.println(d2);
        Locale locIT = new Locale("it", "IT"); // Italy
        Locale locPT = new Locale("pt"); // Portugal
        Locale locBR = new Locale("pt", "BR"); // Brazil
        Locale locIN = new Locale("hi", "IN"); // India
        Locale locJA = new Locale("ja"); // Japan
        Locale locZH = new Locale("zh", "TW"); // Taiwan

        DateFormat dfUS = DateFormat.getInstance();
        System.out.println("US " + dfUS.format(d2)); //US 2010/12/14 上午 10:59
        DateFormat dfUSfull = DateFormat.getDateInstance(DateFormat.FULL);
        System.out.println("US full " + dfUSfull.format(d2));  //US full 2010年12月14日 星期二
        DateFormat dfIT = DateFormat.getDateInstance(DateFormat.FULL, locIT);
        System.out.println("Italy " + dfIT.format(d2)); //Italy marted? 14 dicembre 2010
        DateFormat dfPT = DateFormat.getDateInstance(DateFormat.FULL, locPT);
        System.out.println("Portugal " + dfPT.format(d2)); //Portugal Ter?a-feira, 14 de Dezembro de 2010
        DateFormat dfBR = DateFormat.getDateInstance(DateFormat.FULL, locBR);
        System.out.println("Brazil " + dfBR.format(d2)); //Brazil Ter?a-feira, 14 de Dezembro de 2010
        DateFormat dfIN = DateFormat.getDateInstance(DateFormat.FULL, locIN);
        System.out.println("India " + dfIN.format(d2)); //India शनिवार, १५ अक् तूबर, २०१९
        DateFormat dfJA = DateFormat.getDateInstance(DateFormat.FULL, locJA);
        System.out.println("Japan " + dfJA.format(d2)); //Japan 2010年12月14日
        DateFormat dfZH = DateFormat.getDateInstance(DateFormat.FULL, locZH);
        System.out.println("Taiwan " + dfZH.format(d2)); //Taiwan 2010年12月14日 星期二
        DateFormat dfKA = DateFormat.getDateInstance(DateFormat.FULL, Locale.KOREA);
        System.out.println("Korea " + dfKA.format(d2)); //Korea 2011□ 10□ 15□ □□□
        DateFormat dfDA = DateFormat.getDateInstance(DateFormat.FULL, Locale.getDefault());
        System.out.println("Default " + dfDA.format(d2)); //Default 2011年10月15日 星期六
    }
}
```

<2>.Java 中的 Date 和時區轉換

Date 中保存的是什麼

在 java 中，只要我們執行 Date date = new Date()；就可以得到當前時間。

如：Date date = new Date();

System.out.println(date) ;

輸出結果是：Thu Aug 24 10:15:29 CST 2017

也就是執行上述代碼的時刻：2017 年 8 月 24 日 10 點 15 分 29 秒。

是不是 Date 對象裏存了年月日時分秒呢？

不是的，Date 對象裏存的只是一個 long 型的變量，其值爲自 1970 年 1 月 1 日 0 點至 Date 對象所記錄時刻經過的毫秒數，調用 Date 對象 getTime()方法就可以返回這個毫秒數，如下代碼：

Date date = new Date()；

System.out.println(date + "， " + date.getTime());

輸出如下：Thu Aug 24 10:48:05 CST 2017， 1503542885955

即上述程序執行的時刻是 2017 年 8 月 24 日 10 點 48 分 05 秒，

該時刻距離 1970 年 1 月 1 日 0 點經過了 1503542885955 毫秒。

反過來說，輸出的年月日時分秒其實是根據這個毫秒數來反算出來的。

```
public static void 不同時區的Date() {

    Date date = new Date(1503544630000L);  // 對應的北京時間是2017-08-24 11:17:10

    SimpleDateFormat bjSdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");     // 北京
    bjSdf.setTimeZone(TimeZone.getTimeZone("Asia/Shanghai"));  // 設置北京時區

    SimpleDateFormat tokyoSdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  // 東京
    tokyoSdf.setTimeZone(TimeZone.getTimeZone("Asia/Tokyo"));  // 設置東京時區

    SimpleDateFormat londonSdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); // 倫敦
    londonSdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));  // 設置倫敦時區

    System.out.println("毫秒數:" + date.getTime() + ", 北京時間:" + bjSdf.format(date));
    System.out.println("毫秒數:" + date.getTime() + ", 東京時間:" + tokyoSdf.format(date));
    System.out.println("毫秒數:" + date.getTime() + ", 倫敦時間:" + londonSdf.format(date));
}
```

```
毫秒數:1503544630000, 北京時間:2017-08-24 11:17:10
毫秒數:1503544630000, 東京時間:2017-08-24 12:17:10
毫秒數:1503544630000, 倫敦時間:2017-08-24 04:17:10
```

<3>.時區

(1).全球分爲 24 個時區，相鄰時區時間相差 **1 個小時**。比如北京處於**東八時區**，東京處於**東九時區**，北京時間比東京時間晚 1 個小時，而英國倫敦時間比北京晚 7 個小時（英國採用夏令時，8 月英國處於夏令時）。比如此刻北京時間是 2017 年 8 月 24 日 11:17:10，則東京時間是 2017 年 8 月 24 日 12:17:10，倫敦時間是 2017 年 8 月 24 日 4:17:10。

既然 Date 裏存放的是當前時刻距 1970 年 1 月 1 日 0 點時刻的毫秒數，如果此刻在倫敦、北京、東京有三個程序員同時執 Date date = new Date();那這三個 date 對象裏存的毫秒數是相同的嗎？還是北京的比東京的小 3600000（北京時間比東京時間晚 1 小時，1 小時爲 3600 秒即 3600000 毫秒）？答案是，這 3 個 Date 裏的毫秒數是完全一樣的。

(2).確切的說，Date 對象裏存的是自格林威治時間（ GMT ）1970 年 1 月 1 日 0 點至 Date 對象所表示時刻所經過的毫秒數。所以，如果某一時刻遍佈於世界各地的程序員同時執行 new Date 語句，這些 Date 對象所存的毫秒數是完全一樣的。也就是說，Date 裏存放的毫秒數是與時區無關的。

(3).繼續上述例子，如果上述 3 個程序員調用那一刻的時間是北京時間 2017 年 8 月 24 日 11:17:10，他們繼續調用 System.out.println(date) ;那麼北京的程序員將會打印出 2017 年 8 月 24 日 11:17:10，而東京的程序員會打印出 2017 年 8 月 24 日 12:17:10，倫敦的程序員會打印出 2017 年 8 月 24 日 4:17:10。

(4).既然 Date 對象只存了一個毫秒數，爲什麼這 3 個毫秒數完全相同的 Date 對象，可以打印出不同的時間呢？這是因爲 **System.out.println 函數在打印時間時，會取操作系統當前所設置的時區，然後根據這個時區將同毫秒數解釋成該時區的時間**。

當然我們也可以手動設置時區，以將同一個 Date 對象按不同的時區輸出

可以看出，同一個 Date 對象，按不同的時區來格式化，將得到不同時區的時間。由此可見，Date 對象裏保存的毫秒數和具體輸出的時間（即年月日時分秒）是**模型**和**視圖**的關係，而時區（**即 Timezone**)則決定了將同一個模型展示成什麼樣的視圖

<4>.Java 8 之後，新增了一組日期和時間相關的 API，都包含在「java.time」這個 Package 內 ，Java 8 ，借鑒了第三方開源庫 (open source 用於描述那些程式碼可以被公眾使用） **Joda-Tim** ，相關 API 介面全部位於 java.time。

<5>.java.time 包提供

(1).Instant 類別

(2).Duration 類別

(3).Period 類別

(4).LocalDate 類別不包含具體時間的日期，比如 2014-01-14。它可以用來儲存生日，周年紀念日，入職日期等。

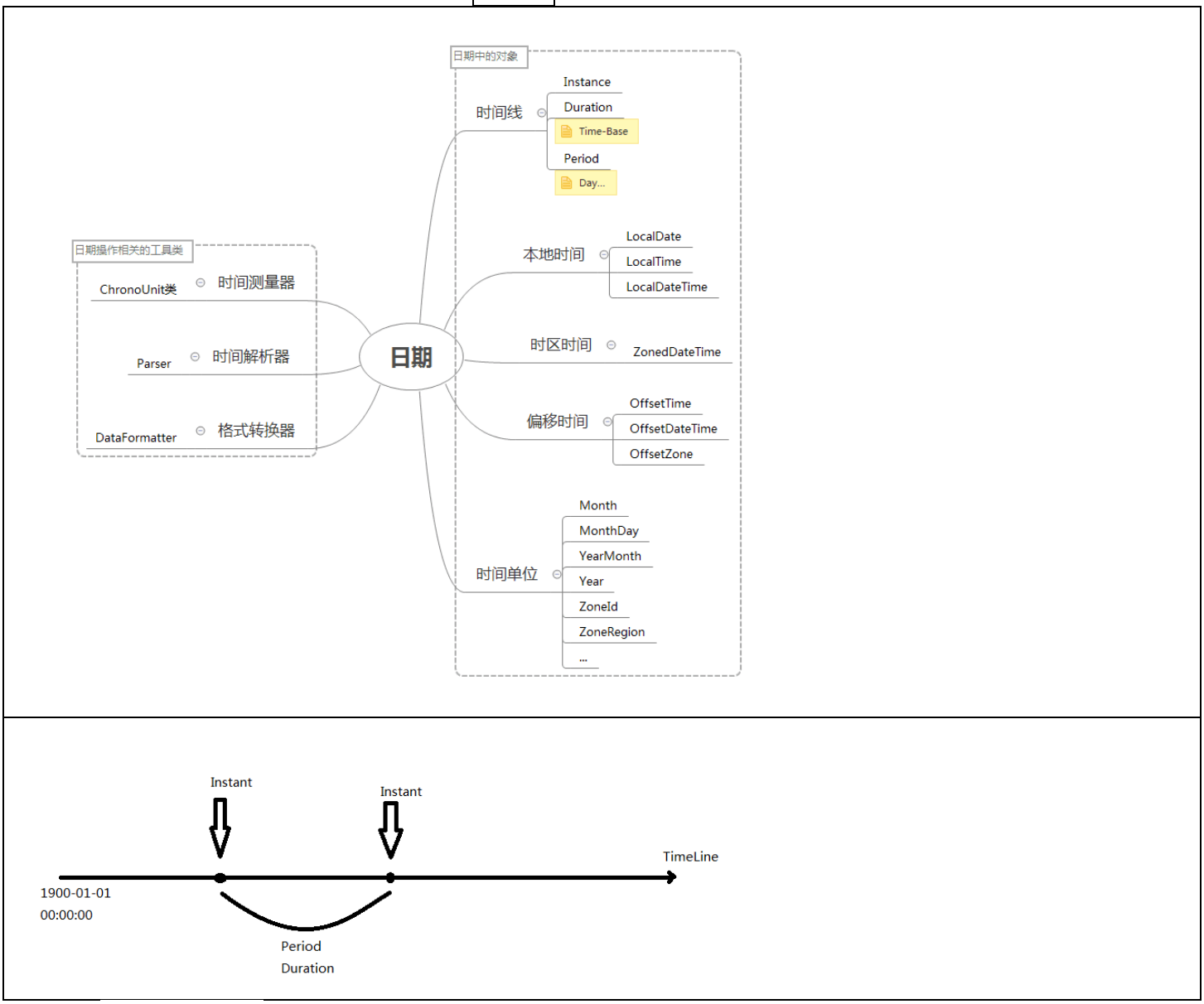
(5).LocalTime 類別代表的是不含日期的時間

(6).LocalDateTime 類別包含了日期及時間，不過没有偏移信息或者說時區。

(7).ZonedDateTime 類別是一個包含時區的完整的日期時間，偏移量是以 UTC/格林威治時間為基準的。

世界協調時間（英語：Coordinated Universal Time，簡稱 UTC）是最主要的世界標準時間，其以原子時秒長為基礎，在時刻上盡量接近於格林威治標準時間

<6>.由於 LocalDate、LocalTime 和 LocalDateTime 物件的建構子都被宣告為 private，因此無法直接使用 new 來實體化出它們。如果要建立出它們的物件，必須透過它們的類別方法來達成



<7>.常用的幾種方法

(1).now 是類別方法➔可以使用現在系統的時間來建立出 LocalDate、LocalTime 和 LocalDateTime 物件。取得目前系統的日期與時間

```
LocalDateTime  localDate Time= LocalDateTime.now(); //直接使用 LocalDateTime 類別來取得日期與時間
```

(2).of 是類別方法➔可以直接傳入年、月、日、小時、分鐘、秒等時間數值來產生 LocalDate、LocalTime 和 LocalDateTime 物件。

注意➔LocalDate 和 LocalTime 這兩個類別都已經定義好它們運用的領域(Domain-driven design)，因此，LocalDate 是無法使用 of 方法指定時間的，LocalTime 也無法使用 of 方法指定日期，如果要同時使用 of 方法指定日期與時間，需得用 LocalDateTime

```
LocalDateTime  localDate Time = LocalDateTime.of(2017， 12， 17， 9， 31， 31， 31);
```

(3).parse 是類別方法➔可以直接傳入字串來產生 LocalDate、LocalTime 和 LocalDateTime 物件。傳入的字串格式可由 **DateTimeFormatter** 來決定，預設是使用 DateTimeFormatter.ISO_LOCAL_DATE_TIME

```
LocalDateTime  localDate Time = LocalDateTime.parse("2015-04-05T12:30:30");
```

<8>.本地日期 LocalDate➔與時區無關 默認回傳的日期格式採用 ISO-8601 標準日期即 YYYY-MM-DD

<9>.本地時間 LocalTime➔與時區無關，它的默認回傳的格式也是採用 ISO-8601 即 HH:mm:ss

```
LocalTime  localTime = LocalTime.of(int hour， int minute， int second);
```

```
LocalTime localTime = LocalTime.of(int hour, int minute, int second, int nanoOfSecond)
LocalTime localTime = LocalTime.of(int hour, int minute);
```

<10>.本地日期時間的組合 LocalDateTime➔與時區無關，所以，如果要轉成 Epoch，就必需要加入時區資訊，沒有加入時區，就無法直接轉換。LocalDateTime，不管在世界上任何一個地方 '2014-1-1 12:00:00' 都是指 2014 年第一天的中午 12 點，不用去管時區，但，如果把這個時間轉成 格林威治時間 Greenwich Mean Time (GMT) 或 世界協調時間 Coordinated Universal Time (UTC) ，就必需加上時區的資料，才會正確，美國紐約的 '2014-1-1 12:00:00' 跟台灣的' 2014-1-1 12:00:00' 一定是不一樣的➔ Epoch，時期； 紀元；世；新时代；指的是一個特定的時間：1970-01-01 00:00:00 UTC

<11>.Instant➔ Instant 和 Date 一樣，表示一時間戳，用於描述一個時刻，只不過它較 Date 而言，可以描述更加精確的時刻，譬如說：「現在 UTC 的時間是西元 2015 年 4 月 3 日凌晨 1 點整。」這就是一個瞬間時間點的描述，Date 最多可以表示**毫秒**級別的时刻，而 Instant 可以表示**納秒**級別的時刻（Nano-Second），System.currentTimeMillis()方法只精確到毫秒（Milli-Second）

<12>.**Duration**➔用來定義一個時間區段，譬如說：「從甲地到乙地，開車開了 20 分鐘。」，以秒和納秒為基準的時長例如， "23.6 秒" 。 以 分十秒為單位，在時間的操作中，可以作為參數，來作為時間的加減運算，Duration 的內部實現與 Instant 類似，也是包含兩部分：seconds 表示秒，nanos 表示納秒。兩者的區別是 Instant 用於表示一個時間點，而 Duration 表示一個時間段，所以 Duration 類中不包含 now() 靜態方法。可以通過 Duration.between( )方法創建 Duration 物件

<13>.**Period**➔用來定義一個時間週期以年、月、日為單位比如 2 年 3 個月 6 天，有點類似 Duration，比較不同的是，Duration 最小的單位可到奈秒(Nanosecond)，而 **Period** 表示以年、月、日衡量的時長

<14>.ZonedDateTime➔ LocalDate、LocalTime 與 LocalDateTime 都沒有時區的概念，如果有要使用到時區的話，應該要使用 ZonedDateTime 這個類別。ZonedDateTime 物件的建立方式和 LocalDateTime 挺像的，只不過 ZonedDateTime 需要在建立時多代入 ZoneId 物件來指定時間的時區，除了使用 ZonedDateTime 類別來建立 ZonedDateTime 物件之外，也可以直接用 LocalDateTime 物件的 atZone 方法，參數傳入 ZoneId 物件，就能轉成 ZonedDateTime 物件

<15>.時間計算器 ChrononUnit➔ChronoUnit 類別可用於在單個時間單位內測量一段時間，例如天數或秒

<16>.格式化日期➔新的日期 API 中提供了一個 DateTimeFormatter 類別用於處理日期格式化操作，它被包含在 java.time.format 包中，Java 8 的日期類別有一個 format( )方法用於將日期格式化為字串，該方法接收一個 DateTimeFormatter 類型參數

預定格式 FormatStyle➔FormatSyle 是一個列舉類，它包括了 Full，LONG，MEDIUM，SHORT 幾種常規的格式

```
FormatStyle.FULL  // 'Tuesday, April 12, 1952 AD' or '3:30:42pm PST'.
FormatStyle.LONG // 'January 12, 1952'
FormatStyle.MEDIUM // 'Jan 12, 1952'.
FormatStyle.SHORT //'12.13.52' or '3:30pm'.
```

<17>.Java 日期時區

(1).世界協調時間（UTC）

a. 如果時間是以世界協調時（UTC）表示，則在時間後面直接加上一個「Z」(不加空格)。「Z」是世界協調時中 0 時區的標誌。因此，「09:30 UTC」就寫作「09:30Z」或是「0930Z」。「14:45:15 UTC」則為「14:45:15Z」或「144515Z」。

b. UTC 時間也被叫做祖魯時間，因為在北約音標字母中用「Zulu」表示「Z」

(2).UTC 偏移量

a. UTC 偏移量是世界協調時（UTC）和特定地點的日期與時間差異，其單位為小時和分鐘。它通常以 ±[hh]:[mm]、±[hh][mm]、或 ±[hh]的格式顯示。所以，如果被描述的時間比 UTC 早一小時（例如柏林的冬季時間），UTC 的偏移量將是"+01:00"、"+0100"、或簡單顯示為"+01"。"UTC+8"表示當協調世界時（UTC）時間為凌晨 2 點的時候，當地的時間為 2+8 點，即早上 10 點

b. 時區和時間偏移➔時區是地理上的一個區域，在那裏的每個人都要遵守同一的標準時間。時間偏移是從 UTC 增加或減少時間，以獲得當前的民用時 - 無論是標準時間或夏令時（日光節約時）。在任何特定的時區，居民在一年四季中都要遵守標準時間（在俄羅斯或南非），或者在冬天使用標準時間，在夏天使用夏令時

c. 日光節約時間➔在北美的有些地區、歐洲和澳洲會使用夏令時（DST，daylight saving time）。在使用夏令時的期間，UTC 偏移量通常會比標準時間增加一小時。歐洲中部時間 UTC+01:00 被歐洲夏令時 UTC+02:00 取代，太平洋標準時 UTC-08:00 被太平洋夏令時 UTC-07:00 取代，**夏令時間**，另譯**夏時制**（英語：Summer time），又稱**日光節約時制**、**日光節約時間**（英語：Daylight saving time），是一種為節約能源而人為規定地方時間的制度，在這一制度實行期間所採用的統一時間稱為「夏令時間」。一般在天亮較早的夏季人為將時間調快一小時，可以使人早起早睡，減少照明量，以充分利用光照資源，從而節約照明用電。各個採納夏令時間的國家規定不同

```java
public class Test01_LocalDate {

    public static void main(String[] args) {
        LocalDate1();
        //LocalDate2();
    }
public static void LocalDate1() {
    //獲取當前的日期 LocalDate
    System.out.println(LocalDate.now());//2019-02-14
    //根據年月日的值獲取 LocalDate
    System.out.println(LocalDate.of(2016, 11, 30));//2016-11-30
    //根據某年的第n天獲取 LocalDate
    System.out.println(LocalDate.ofYearDay(2016, 300));//2016-10-26

    System.out.println("加法運算----------------------------");
    System.out.println("當前：" + LocalDate.now()); //2019-02-14
    System.out.println("加1天：" + LocalDate.now().plusDays(1)); //2019-02-15
    System.out.println("加1周：" + LocalDate.now().plusWeeks(1));//2019-02-21
    System.out.println("加1月：" + LocalDate.now().plusMonths(1)); //2019-03-14
    System.out.println("加1年：" + LocalDate.now().plusYears(1));//2020-02-14
    System.out.println("減法運算----------------------------");
    System.out.println("當前：" + LocalDate.now());//2019-02-14
    System.out.println("減1天：" + LocalDate.now().minusDays(1));//2019-02-13
    System.out.println("減1周：" + LocalDate.now().minusWeeks(1));//2019-02-07
    System.out.println("減1月：" + LocalDate.now().minusMonths(1));//2019-01-14
    System.out.println("減1年：" + LocalDate.now().minusYears(1));//2018-02-14
    System.out.println("替換運算----------------------------");
    System.out.println("當前：" + LocalDate.now());//2019-10-26
    System.out.println("替換[日期]為10：" + LocalDate.now().withDayOfMonth(10));//2019-10-10
    System.out.println("替換[天數]為200：" + LocalDate.now().withDayOfYear(200));//2019-07-19
    System.out.println("替換[月份]為1：" + LocalDate.now().withMonth(1));//2019-01-26
    System.out.println("替換[年份]為2020：" + LocalDate.now().withYear(2020));//2020-10-26
    LocalDate 當天 = LocalDate.of(2019, 9, 1);
    System.out.println("2019/9/1：" + 當天);//2019-9-1
    System.out.println("今天是否在當天之前：" + 當天.isBefore(LocalDate.now()));//true
    System.out.println("是否在當天之後：" + 當天.isAfter(LocalDate.now()));//false
    System.out.println("是否在當天：" + 當天.isEqual(LocalDate.now()));//false
    System.out.println("今年是否是閏年：" + LocalDate.now().isLeapYear());//false
}
```

```java
public static void LocalDate2() {
    LocalDate localDate = LocalDate.of(2019, 9, 9);
    int year = localDate.getYear();
    System.out.println("year=" + year);  // year=2019
    Month month = localDate.getMonth();
    System.out.println("month=" + month); // month=SEPTEMBER （英文[enum]表示）
    int monthvalue = localDate.getMonthValue();
    System.out.println("monthvalue=" + monthvalue);  // monthvalue=9
    int dayOfMonth = localDate.getDayOfMonth();
    System.out.println("dayOfMonth=" + dayOfMonth); // dayOfMonth=9 一年中的第幾個月
    int getDayOfYear = localDate.getDayOfYear();
    System.out.println("getDayOfYear=" + getDayOfYear);// getDayOfYear=252 當天所在這一年的第幾天（從1開始）
    DayOfWeek dayOfWeek = localDate.getDayOfWeek();
    System.out.println("dayOfWeek=" + dayOfWeek); // 一周的星期幾：MONDAY
    int lengthOfYear = localDate.lengthOfYear();
    System.out.println("lengthOfYear=" + lengthOfYear);// 當年的天數 365
    int lengthOfMonth = localDate.lengthOfMonth();
    System.out.println("lengthOfYear=" + lengthOfMonth); // 當月的天數 ：31
    boolean leapYear = localDate.isLeapYear();
    System.out.println("leapYear=" + leapYear); // 是否為閏年：false
}
```

```java
public class Test02_LocalTime {

    public static void main(String[] args) {
        LocalTime1();
    }

    public static void LocalTime1() {
        LocalTime localTime = LocalTime.of(17, 23, 52);// 初始化一個時間：17:23:52
        System.out.println("hour=" + localTime.getHour()); // 時：17
        System.out.println("minute =" + localTime.getMinute());// 分：23
        System.out.println("second =" + localTime.getSecond()); // 秒：52
    }
}
```

```java
public class Test03_LocalDateTime {

    public static void main(String[] args) {
        LocalDateTime1();
        //LocalDateTime2();
        //LocalDateTime3();
    }
```

```
public static void LocalDateTime1() {

    LocalDateTime localDateTime1 = LocalDateTime.now();
    System.out.println(localDateTime1); //2019-09-09T13:26:17.836
    System.out.println(localDateTime1.getYear()); //2019
    System.out.println(localDateTime1.getMonthValue()); //9
    System.out.println(localDateTime1.getDayOfMonth()); //9
    System.out.println(localDateTime1.getHour()); //13
    System.out.println(localDateTime1.getMinute()); //26
    System.out.println(localDateTime1.getSecond()); //17
    System.out.println(localDateTime1.getNano()); //836000000

    LocalDateTime localDateTime2 = LocalDateTime.of(2017, 12, 17, 9, 31, 31, 31);
    System.out.println(localDateTime2); //2017-12-17T09:31:31.000000031

    LocalDateTime localDateTime3 = localDateTime2.plusDays(12); //加 12 天
    System.out.println(localDateTime3); //2017-12-29T09:31:31.000000031

    LocalDateTime localDateTime4 = localDateTime3.minusYears(2); //减 2 年
    System.out.println(localDateTime4); //2015-12-29T09:31:31.000000031
}
```

```
public static void LocalDateTime2() {
    LocalDateTime localDateTime1 = LocalDateTime.of(2017, Month.JANUARY, 4, 17, 23, 52);
    System.out.println("localDateTime1=" + localDateTime1); //localDateTime1=2017-01-04T17:23:52
    LocalDate localDate = LocalDate.of(2017, Month.JANUARY, 4);
    LocalTime localTime = LocalTime.of(17, 23, 52);
    LocalDateTime localDateTime2 = localDate.atTime(localTime);
    System.out.println("localDateTime2=" + localDateTime2); //localDateTime2=2017-01-04T17:23:52
}
```

```
    public static void LocalDateTime3() {
        //LocalDateTime也提供用於向LocalDate和LocalTime的轉化
        LocalDateTime date1 = LocalDateTime.of(2017, Month.JANUARY, 4, 17, 23, 52);
        LocalDate localDate = date1.toLocalDate();
        LocalTime localTime = date1.toLocalTime();
        System.out.println("localDate=" + localDate); //localdate=2017-01-04
        System.out.println("localTime=" + localTime); //localtime=17:23:52
    }
}
```

```
public class Test04_Instant {

    public static void main(String[] args) {
        //Instant1();
        //Instant2();
        //Instant3();
        Instant4();
    }
```

```java
public static void Instant1() {
    Instant instant1 = Instant.now();
    System.out.println(instant1); //2019-02-10T13:30:17.690Z
    //方法的第一個參數為秒，第二個參數為納秒，上面的代碼表示從1970-01-01 00:00:00開始後兩分鐘的10萬納秒的時刻
    Instant instant2 = Instant.ofEpochSecond(120, 100000);
    System.out.println(instant2); //1970-01-01T00:02:00.000100Z
}
```

```java
public class Test05_Duration {

    public static void main(String[] args) {
        //  Duration1();
        Duration2();
    }
```

```java
public static void Duration1() {
    // 2017-01-05 10:07:00
    LocalDateTime from = LocalDateTime.of(2017, Month.JANUARY, 5, 10, 7, 0);
   // 2017-02-05 10:07:00
    LocalDateTime to = LocalDateTime.of(2017, Month.FEBRUARY, 5, 10, 7, 0);
   // 表示從 2017-01-05 10:07:00 到 2017-02-05 10:07:00 這段時間
    Duration duration = Duration.between(from, to);
    System.out.println(duration.toDays()); // 31 這段時間的總天數
    System.out.println(duration.toHours());  //744  這段時間的小時數
    System.out.println(duration.toMinutes());  //44640 這段時間的分鐘數
    System.out.println(duration.getSeconds());  //2678400 這段時間的秒數
    System.out.println(duration.toMillis());  //2678400000 這段時間的毫秒數
    System.out.println(duration.toNanos());  //2678400000000000 這段時間的納秒數
}
```

```java
    public static void Duration2() {
        //Duration對象還可以通過of()方法創建，該方法接受一個時間段長度，和一個時間單位作為參數
        Duration duration1 = Duration.of(5, ChronoUnit.DAYS);       // 5天
        Duration duration2 = Duration.of(1000, ChronoUnit.MILLIS);  // 1000毫秒
    }
}
```

```java
public class Test06_Period {

    public static void main(String[] args) {
        // Period1();
         Period2();
    }
```

```java
  public static void Period1() {
        //Period在概念上和Duration類似，區別在於Period是以年月日來衡量一個時間段，比如2年3個月6天
        Period period = Period.of(2, 3, 6);
        System.out.println(period); //P2Y3M6D
    }

    public static void Period2() {
        // 2017-01-05 到 2017-02-05 這段時間
        Period period = Period.between( LocalDate.of(2017, 1, 5), LocalDate.of(2017, 2, 5));
        System.out.println(period); //P1M
    }
}
```

```java
public class Test07_ZonedDateTime {

    public static void main(String[] args) {
        //ZonedDateTime1();
        //  ZonedDateTime2();
        // ZonedDateTime3();
        //   ZonedDateTime4();
        ZonedDateTime5();
    }
    public static void ZonedDateTime1() {
        final ZonedDateTime currentPoint = ZonedDateTime.now();
        System.out.println(currentPoint);//2019-10-26T22:53:18.875+08:00[Asia/Taipei]
    }

    public static void ZonedDateTime2() {
        final LocalDateTime currentDateTime = LocalDateTime.now();
        final ZonedDateTime zonedCurrentDateTime = currentDateTime.atZone(ZoneId.of("+8"));
        System.out.println(zonedCurrentDateTime);//2019-10-26T22:57:14.506+08:00
    }

    public static void ZonedDateTime3() {
        final ZoneId zoneidDefault = ZoneId.systemDefault(); //系統預設時區
        System.out.println(zoneidDefault);//Asia/Taipei
        //使用of()工廠方法創建ZoneId
        final ZoneId zoneidPlus8 = ZoneId.of("UTC+8"); //UTC時間+8
        System.out.println(zoneidPlus8); //UTC+08:00
    }

    public static void ZonedDateTime4() {
        ZonedDateTime zonedDateTime = ZonedDateTime.now();
        //2019-10-26T23:01:34.628+08:00[Asia/Taipei]
        System.out.println(zonedDateTime);

        Instant instant = Instant.now();
        ZoneId zoneId1 = ZoneId.of("GMT"); //格林威治時間
        ZonedDateTime zonedDateTime2 = ZonedDateTime.ofInstant(instant, zoneId1);
        //2019-10-26T15:01:34.629Z[GMT]
        System.out.println(zonedDateTime2);
    }

    public static void ZonedDateTime5() {
        //ZoneId 中的 getAvailableZoneIds()返回所有已知時區ID。
        Set<String> zoneIds = ZoneId.getAvailableZoneIds();
        for (String zoneId : zoneIds) {
            System.out.println(zoneId);
        }
    }
}
```

```java
public class Test08_ChrononUnit {

    public static void main(String[] args) {
       ChrononUnit1();
    }
  public static void ChrononUnit1() {
        LocalDate startDate = LocalDate.of(1993, Month.OCTOBER, 19);
        System.out.println("開始時間  : " + startDate); //1993-10-19

        LocalDate endDate = LocalDate.of(2017, Month.JUNE, 16);
        System.out.println("結束時間 : " + endDate); //2017-06-16

        long daysDiff = ChronoUnit.DAYS.between(startDate, endDate);
        System.out.println("兩天之間的差在天數   : " + daysDiff); //8641
    }
}
```

```java
public class Test09_DateTimeFormatter {

    public static void main(String[] args) {
        // DateTimeFormatter1();
         DateTimeFormatter2();
    }
```

```java
public static void DateTimeFormatter1() {
    //預定格式 FormatStyle
    LocalDateTime dateTime = LocalDateTime.now();
    String strDate5;
    DateTimeFormatter shortDateTime1 = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
    DateTimeFormatter shortDateTime2 = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
    DateTimeFormatter shortDateTime3 = DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
    strDate5 = dateTime.format(shortDateTime1);
    System.out.println(strDate5); //2019/2/14
    strDate5 = dateTime.format(shortDateTime2);
    System.out.println(strDate5); //2019/2/14 下午 11:46:24
    strDate5 = dateTime.format(shortDateTime3);
    System.out.println(strDate5); //下午 11:46:24
}
```

```java
    public static void DateTimeFormatter2() {
        //自定義格式 Pattern
        LocalDateTime dateTime = LocalDateTime.now();
        DateTimeFormatter f1 = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
        System.out.println(dateTime.format(f1)); // January 20, 2020, 01:12
        DateTimeFormatter f2 = DateTimeFormatter.ofPattern("MM dd, yyyy, hh:mm");
        System.out.println(dateTime.format(f2)); // 01 20, 2020, 01:12

        DateTimeFormatter f3 = DateTimeFormatter.ofPattern("MMMM DD, YY, HH:mm");
        System.out.println(dateTime.format(f3)); // January 20, 2020, 13:12
        DateTimeFormatter f4 = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
        System.out.println(dateTime.format(f4)); // January 20, 2020, 01:12

        DateTimeFormatter f5 = DateTimeFormatter.ofPattern("MMMM dd, yy, hh:mm:ss");
        System.out.println(dateTime.format(f5)); // January 20, 2020, 01:12:34
        DateTimeFormatter f6 = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm:ss:SS");
        System.out.println(dateTime.format(f6)); // January 20, 2020, 01:12:00
    }
}
```

夏令时

夏令时 (DST) 是年份的一部分，在该部分中，地区的本地时间超前于该地区的标准官方时间。

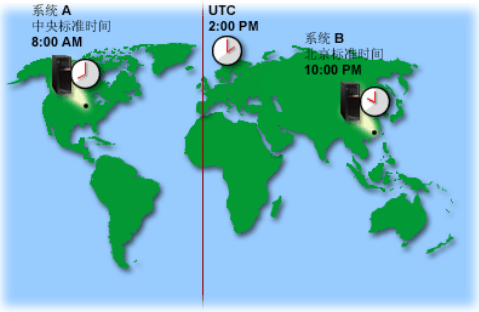
夏令时变化

夏令时变化表示当 DST 开始时本地时间向前移动的分钟数，或者表示当 DST 结束时向后移动的分钟数。

时区

时区设置指定与 UTC 的偏移量以及是否要遵守 DST。系统上的每个逻辑分区都可以指定要使用的时区。

要将这些时间概念组合成与系统相关联的各种时间值，请研究以下两个使用不同时区的系统。



此图显示了两个不同时区中的两个系统。系统 A 位于中部标准时区，系统 B 位于北京标准时区。

每个系统的时间值如下：

	系统 A	系统 B
本地系统时间	上午 8 点	晚上 10 点
本地作业时间	上午 8 点	晚上 10 点
时区	中部标准时间	北京标准时间
与 UTC 的偏移量	-6:00	+8:00
UTC	下午 2 点	下午 2 点

各地的標準時間為格林威治時間（G.M.T）加上（+）或減去（-）時區中所標的小時和分鐘數時差。

UTC (世界標準時間) = GMT (格林威治時間)

(UTC+8) = 當格林尼治時間（G.M.T）為凌晨 00:00 時，中國標準時間剛好為上午 08:00。

=（**HKT**）**香港標準時間**

=（CST）中國大陸的北京時間

=（MOT）澳門標準時間

=（NST）台北時間

=（SGT）新加坡標準時間

CET ．Central European Time 歐洲中部時間=（地理位置為 UTC-1）

實際使用時間(各國不同)：冬季時間為 UTC+1，夏季歐洲夏令時為 UTC+2。

PS:

歐洲有些地方 因為很多因素 ex:二次大戰.政治因素.習慣..等

使得地理上的時間 跟 實際使用得時間不一樣...

PS: 夏令時 (DST)

「夏日節約時間」Daylight Saving Time（簡稱 D.S.T.），是指在夏天太陽升起的比較早時，將時鐘撥快一小時. 所以 = (UTC 再+1).大約從當年的四月開始， 到當年的十月底，時鐘才恢復到標準時間。