在了解 Java 8 新增的時間系列之前，我們需要先了解時間相關的知識

**1. 首先是要有時區的概念**

在台灣的時區是 GMT+8，而在英國的時區為 GMT+0，所以在同一瞬間，在英國看見的時間是 2020/06/29 14:00:00，但在台灣看見的時間卻會是 2020/06/29 22:00:00

**2. 要有 Timestamp 時間戳的概念**

在電腦的世界裡，有一個東西叫做 Timestamp（時間戳），這個 Timestamp 是一個整數，代表著從 UTC 1970 年 1 月 1 日 0 時 0 分 0 秒 起至現在的總秒數，UTC 代表的是英國格林威治時間，也就是 GMT+0（可以簡單的認為 UTC = GMT+0）

所以說假設現在你人在英國格林威治，然後你現在看手錶上的時間是 2020/06/29 14:00:00，則這個當下的 Timestamp的值就為 1593439200（代表從 1970/1/1 00:00:00 到現在經過了 1593439200 秒），然後假設同一個瞬間你的朋友在台灣，因為台灣的時區為 GMT+8，所以你朋友在台灣看手錶的時間會是 2020/06/29 22:00:00，但是你朋友當下的 Timestamp 也會是 1593439200

也就是說，雖然你和你朋友手錶上呈現的時間不一樣，但是你們兩個的 Timestamp 會是一模一樣的，因為 Timestamp 是唯一表示著 UTC 1970/1/1 00:00:00 起至現在的總秒數

而在Java中，可以使用 System.currentTimeMillis()，取得當前的 Timestamp

**為什麼 Java 中舊版的 Date 不好 ?**

因為 Date 硬是把 **時區** 和 **Timestamp** 這兩個概念混合在一起了，這就是為什麼 Date 不好的關係

通常我們 new 一個 Date object時，都是使用 Date date = new Date()，而這個 Constructor 其實是會去取得當前的 Timestamp，然後把 Timestamp 存放在 Date 內部的 fastTime 變量裡，所以 Date 內部存放的，實際上是 Timestamp 的值

**public** **class** **Date** **{**

**private** **transient** **long** fastTime**;** *//存放timestamp的值*

**public** **Date()** **{**

**this(**System**.**currentTimeMillis**());**

**}**

**public** **Date(long** date**)** **{**

fastTime **=** date**;**

**}**

**}**

雖然 Date 內部只存放了 Timestamp 的值，但是 Date 卻在 toString() 時，會去取得當前程式運行的時區，然後再把 Timestamp 換算成當地時區的時間印出來

**public** **class** **MyTest** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

*// Date 內部明明是只存放 Timestamp 的值 1593439200，也就是 UTC 2020/06/29 14:00:00*

*// (因為 Date 的構造方法要輸入 ms 毫秒，所以要把 Timestamp 乘以 1000)*

Date date **=** **new** Date**(**1593439200L **\*** 1000**);**

*// 但是卻會在 toString() 方法裡面再去取得當前程式運行的時區(我的時區是GMT+8)*

*// 然後把 Timestamp 的值轉換成該時區的時間*

*// 所以這裡會輸出 Mon Jun 29 22:00:00 CST 2020*

System**.**out**.**println**(**date**);**

**}**

**}**

所以 Java 為了改善 Date 混合 **時區** 和 **Timestamp** 的問題，在 Java 8 時提出了新的時間系列類，將 Timestamp 和時間分成兩組

* Timestamp組 : Instant
* 時間組 : LocalDate、LocalTime、LocalDateTime、ZonedDateTime

**Instant 用法**

在 Java 8 新增的這一堆時間系列裡面，每個類都有工廠方法可以使用，可以直接用這些工廠方法來產生出一個新的 object

所以我們就可以直接使用 Instant.of() 去創建一個 instant object

**public** **class** **MyTest** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

*// 用當下的 Timestamp 來創建in1*

Instant in1 **=** Instant**.**now**();**

*// 自定義 Timestamp*

Instant in2 **=** Instant**.**ofEpochSecond**(**1593439200L**);**

**}**

**}**

**LocalDate、LocalTime、LocalDateTime 用法**

1. LocalDate : 只存日期 (2020/06/29)
   * 和Date不同，LocalDate內部用了三個Integer變量 year、month、day，分別存放日期的值，徹底將時間和Timestamp的概念區分開來
2. LocalTime : 只存時間 (14:00:00)
   * 也是用了三個Integer變量 hour、minutes、second來分別存放時間的值
3. LocalDateTime : 存了日期和時間 (2020/06/29 14:00:00)
   * LocalDateTime內部其實用一個LocalDate變量存放日期，用另一個LocalTime變量存放時間，reuse的極致！

Local 這一系列的時間類型，都不帶有時區的資訊，所以說當你new一個LocalDateTime出來之後，不管你是在台灣運行程式還是在英國運行程式，輸出的值都一樣

**public** **class** **MyTest** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

*// LocalDate*

LocalDate date **=** LocalDate**.**of**(**2020**,** 6**,** 29**);** *// 創建一個 2020/06/29的LocalDate*

LocalDate date2 **=** LocalDate**.**parse**(**"2020-06-29"**);** *// parse 字串成 LocalDate*

*// LocalTime*

LocalTime time **=** LocalTime**.**of**(**14**,** 0**,** 0**);** *// 創建一個 14:00:00 的 LocalTime*

LocalTime time2 **=** LocalTime**.**parse**(**"14:00:00"**);** *// parse 字串成 LocalTime*

*// LocalDateTime*

*// 創建一個 2020/06/29 14:00:00 的 LocalDateTime*

LocalDateTime dateTime **=** LocalDateTime**.**of**(**2020**,** 6**,** 29**,** 14**,** 0**,** 0**);**

*// 或是也可以用 LocalDate 和 LocalTime 組合創建*

LocalDateTime dateTime2 **=** LocalDateTime**.**of**(**date**,** time**);**

*// Local 系列可以和同樣也是 Java8 新增的 DateTimeFormatter 一起使用，讓時間 input/output 的更格式化*

*// Java8 新增的 DateTimeFormatter 和 joda-time 中的 DateTimeFormatter 名字一樣，不要 import 錯了*

DateTimeFormatter DTF **=** DateTimeFormatter**.**ofPattern**(**"yyyy-MM-dd HH:mm:ss"**);**

*// 用DTF的格式來parse字串成LocalDateTime*

LocalDateTime dateTime3 **=** LocalDateTime**.**parse**(**"2020-06-29 14:00:00"**,** DTF**);**

*// 將dateTime2用DTF的形式輸出*

String format **=** DTF**.**format**(**dateTime2**);**

**}**

**}**

**ZonedDateTime 用法**

ZonedDateTime 是 LocalDateTime 的進化版，增加了時區的概念，會自動幫我們計算時區之間的時差

**public** **class** **MyTest** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

DateTimeFormatter DTF **=** DateTimeFormatter**.**ofPattern**(**"yyyy-MM-dd HH:mm:ss"**);**

LocalDateTime leaving **=** LocalDateTime**.**of**(**2020**,** 6**,** 29**,** 19**,** 30**);**

*// 將原本的 LocalDateTime 轉成 ZonedDateTime*

*// 因為加上了時區的概念，所以現在就變成了 上海 的 2020/06/29 19:30:00*

ZoneId leavingZone **=** ZoneId**.**of**(**"Asia/Shanghai"**);**

ZonedDateTime departure **=** ZonedDateTime**.**of**(**leaving**,** leavingZone**);**

System**.**out**.**println**(**"離開上海的時間：" **+** departure**.**format**(**DTF**));**

*// 把時區轉成 東京 時區，並且加上飛機的飛行時間 30 分鐘*

ZoneId arrivingZone **=** ZoneId**.**of**(**"Asia/Tokyo"**);**

ZonedDateTime arrival **=** departure**.**withZoneSameInstant**(**arrivingZone**).**plusMinutes**(**30**);**

System**.**out**.**println**(**"抵達東京的時間：" **+** arrival**.**format**(**DTF**));**

**}**

**}**

離開上海的時間 2020**-**06**-**29 19**:**30**:**00 *// 上海 的當地時間*

抵達東京的時間 2020**-**06**-**29 21**:**00**:**00 *// 東京 的當地時間*

*// 縱使飛機只飛 30 分鐘，但是因為上海和東京的時區不同(東京快一小時)，所以抵達東京的時間實際上會變成一個半小時後*

*// 而這中間的時差計算，就是ZonedDateTime幫我們做的*